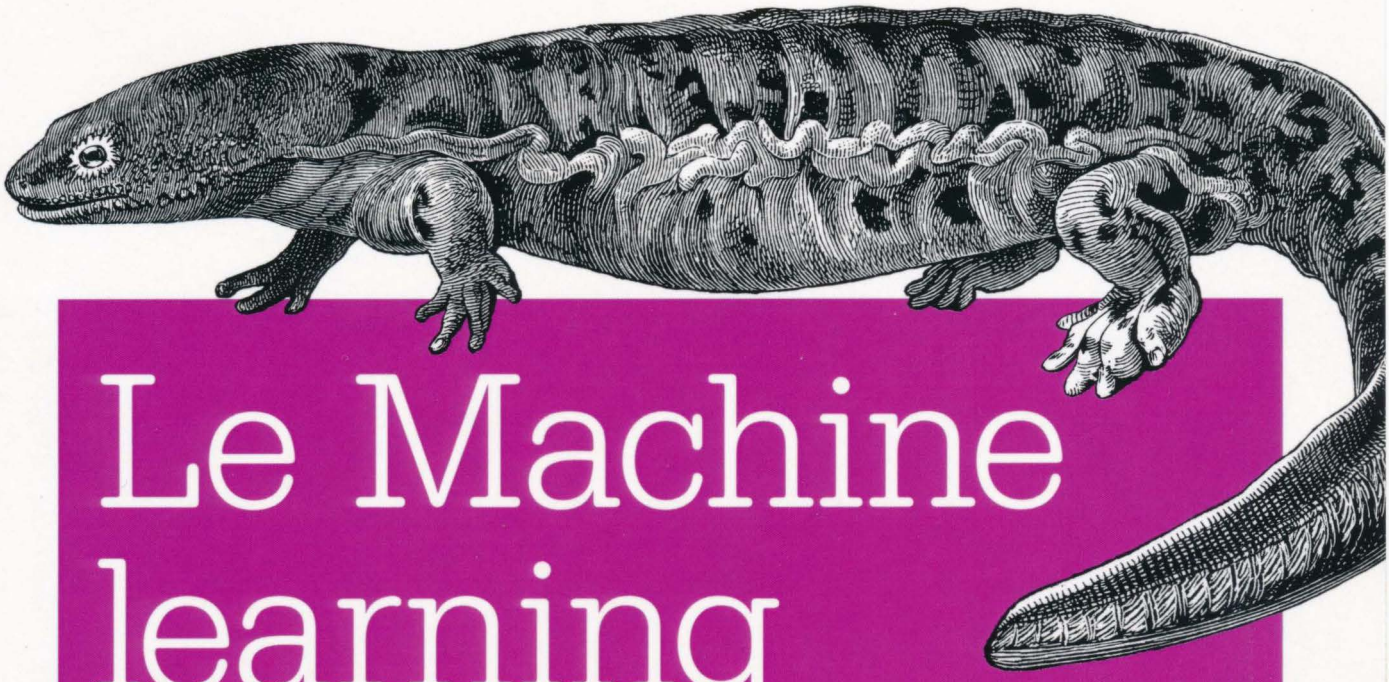
O'REILLY®



# Le Machine learning avec Python

LA BIBLE DES DATA SCIENTISTS

jupyter

Andreas C. Müller et Sarah Guido

# Le Machine learning avec Python

*La bible des data scientists*

*Andreas C. Müller et Sarah Guido*






**Le Machine learning avec Python**

Traduction française publiée et vendue avec l'autorisation de O'Reilly Media, Inc. de *Introduction to Machine Learning with Python* ISBN 9781449369415 © 2017 Sarah Guido, Andreas Müller

© 2018 Éditions First, un département d'Édi8.
12, avenue d'Italie
75013 Paris – France
Tél. : 01 44 16 09 00
Fax : 01 44 16 09 01

Courriel : firstinfo@editionsfirst.fr
Site Internet : lisez.com

ISBN : 978-2-412-03446-0
Dépôt légal : février 2018
Imprimé en France

Traduction de l'anglais : Daniel Rougé
Mise en page : Pierre Brandeis

**Tous droits réservés.** Toute reproduction, même partielle, du contenu, de la couverture ou des icônes, par quelque procédé que ce soit (électronique, photocopie, bande magnétique ou autre) est interdite sans autorisation par écrit d'Édi8.

**Limites de responsabilité et de garantie.** L'auteur et l'éditeur de cet ouvrage ont consacré tous leurs efforts à préparer ce livre. Édi8 et les auteurs déclinent toute responsabilité concernant la fiabilité ou l'exhaustivité du contenu de cet ouvrage. Ils n'assument pas de responsabilité pour ses qualités d'adaptation à quelque objectif que ce soit, et ne pourront être en aucun cas tenus responsables pour quelque perte, profit ou autre dommage commercial que ce soit, notamment mais pas exclusivement particulier, accessoire, conséquent, ou autre.

**Marques déposées.** Toutes les informations connues ont été communiquées sur les marques déposées pour les produits, services et sociétés mentionnés dans cet ouvrage. Édi8 décline toute responsabilité quant à l'exhaustivité et à l'interprétation des informations. Tous les autres noms de marques et de produits utilisés dans cet ouvrage sont des marques déposées ou des appellations commerciales de leur propriétaire respectif. Édi8 n'est lié à aucun produit ou vendeur mentionné dans ce livre. Le logo O'Reilly est une marque déposée de O'Reilly Media, Inc.

# Table des matières

**Préface** ..... **ix**
Qui devrait lire ce livre ? ix
Pourquoi nous avons écrit ce livre x
Naviguer dans le livre x
Conventions utilisées dans ce livre xi
Utiliser les exemples de code xii

**1. Introduction** ..... **1**
Pourquoi l'apprentissage automatique ? 2
Les problèmes que l'apprentissage automatique peut résoudre 2
Comprendre votre but et comprendre vos données 5
Pourquoi Python ? 6
scikit-learn 6
Installer scikit-learn 7
Librairies et outils essentiels 8
Jupyter Notebook 8
NumPy 9
SciPy 9
matplotlib 10
pandas 11
mglearn 12
Python 2 versus Python 3 13
Versions utilisées dans ce livre 13
Une première application : classifier des espèces d'iris 15
À la rencontre des données 16
Mesurer la réussite : entraîner et tester les données 19
Au commencement de tout : vérifier vos données 20
Construire votre premier modèle : les K plus proches voisins 22
Faire des prédictions 24
Évaluer le modèle 25
Résumé et perspectives 26

**2. Apprentissage supervisé ..... 29**
Classification et régression 29
Généralisation, surapprentissage et sous-apprentissage 30
Relation entre complexité du modèle et taille du jeu de données 33
Algorithmes pour l'apprentissage automatique supervisé 33
Exemples de jeux de données 34
Les *k* plus proches voisins 38
Modèles linéaires 48
Classifieurs bayésiens naïfs 71
Arbres de décision 73
Ensembles d'arbres de décision 86
SVM à noyau 95
Réseaux de neurones (*deep learning*) 107
Estimer l'incertitude pour les classifieurs 121
La fonction de décision 122
Prédire les probabilités 124
Classification multiclasse et incertitude 127
Résumé et perspectives 129

**3. Apprentissage non supervisé et prétraitement ..... 133**
Types d'apprentissages non supervisés 133
Les défis de l'apprentissage non supervisé 134
Prétraitement et recalibrage 135
Différents types de prétraitement 135
Appliquer des transformations aux données 136
Recalibrer de la même manière le jeu d'apprentissage et le jeu de test 139
Effet du prétraitement sur l'apprentissage supervisé 141
Réduction de la dimension, extraction de caractéristiques et apprentissage de variétés 142
Analyse en composantes principales (PCA) 143
Factorisation en matrices non négatives (NMF) 159
Apprentissage de variétés avec t-SNE 167
Clustering 171
Partitionnement en *k*-moyennes 171
Clustering agglomératif 185
DBSCAN 189
Comparer et évaluer les algorithmes de clustering 194
Résumons les méthodes de clustering 209
Résumé et perspectives 209

**4. Représentation des données et ingénierie des caractéristiques ..... 213**
Variables catégorielles 214
Encodage one-hot (variables indicatrices) 215
Les nombres peuvent encoder des catégories 220
Binning, discrétisation, modèles linéaires et arbres 222
Interactions et polynômes 226
Transformations non linéaires univariées 234
Sélection automatique de caractéristiques 237
Statistiques univariées 238
Sélection de caractéristiques basée sur le modèle 240
Sélection itérative de caractéristiques 242
Savoir utiliser l'expertise 243
Résumé et perspectives 252

**5. Évaluation et amélioration du modèle ..... 253**
Validation croisée 254
Validation croisée dans `scikit-learn` 255
Bénéfices de la validation croisée 256
Validation croisée à *k*-plis stratifiée et autres stratégies 257
Recherche sur grille 263
Exemple simple de recherche sur grille 263
Le danger du surapprentissage des paramètres et du jeu de validation 264
Recherche sur grille avec validation croisée 266
Métriques d'évaluation et scoring 278
Ne jamais oublier le but final 278
Métriques d'évaluation pour la classification binaire 279
Métriques pour la classification multiclasse 300
Métriques de régression 303
Utiliser des métriques d'évaluation dans la sélection de modèles 303
Résumé et perspectives 305

**6. Chaînage d'algorithmes et pipelines ..... 307**
Sélection de paramètres avec prétraitement 308
Construire des pipelines 310
Utiliser des pipelines dans des recherches sur grille 311
Interface générale de la classe Pipeline 314
Créer facilement un pipeline avec make_pipeline 315
Accéder aux attributs des étapes 317
Accéder aux attributs d'un pipeline depuis GridSearchCV 317
Étapes de prétraitement et paramètres du modèle 319
Effectuer une recherche sur grille pour trouver quel modèle utiliser 321
Résumé et perspectives 322

**7. Travailler avec des données textuelles.....325**
Types de données représentés sous forme de chaînes de caractères 325
Exemple d'application : analyse de sentiment dans des commentaires de films 327
Représenter les données textuelles sous forme de sacs de mots 330
Appliquer une représentation par sac de mots à un jeu de données jouet 331
La représentation par sac de mots appliquée aux critiques de films 333
Mots vides (*stop words*) 337
Pondérer les données avec tf-idf 338
Investiguer les coefficients du modèle 341
Des sacs avec plusieurs mots (*n*-grammes) 342
Tokenisation avancée, racinisation et lemmatisation 347
Modèles de sujets et partitionnement de document 350
L'allocation de Dirichlet latente (LDA) 350
Résumé et perspectives 357

**8. Pour conclure.....359**
Aborder un problème d'apprentissage automatique 359
Est-ce qu'il y des humains dans la boucle ? 360
Du prototype à la production 361
Tester des systèmes de production 362
Construire votre propre estimateur 362
Pour aller plus loin 363
Théorie 363
Autres packages et ressources pour l'apprentissage automatique 364
Ranking, systèmes de recommandation, et autres types d'apprentissage 365
Inférence, programmation et modélisation probabiliste 365
Réseaux de neurones 366
Recalibrer les jeux de données 367
Améliorer vos compétences 368
Conclusion 368

**Index.....369**

L'apprentissage automatique, le *machine learning*, est de nos jours partie intégrante de nombreuses applications commerciales et de projets de recherche, qu'il s'agisse de questions aussi importantes que le diagnostic et le traitement de maladies graves, ou plus prosaïquement de trouver des « amis » sur des réseaux sociaux. Beaucoup de gens croient que l'apprentissage automatique ne peut être mis en œuvre que par de grandes entreprises avec des équipes de recherche très développées. Dans ce livre, nous voulons vous montrer qu'il est plus facile que vous ne le pensez de construire vos propres solutions d'apprentissage automatique, et comment vous pouvez vous y prendre au mieux. Avec les connaissances développées dans ces pages, vous pourrez bâtir votre propre système pour déterminer par exemple le sentiment exprimé sur tel ou tel sujet par les utilisateurs de Twitter, ou bien encore pour faire des prédictions sur le réchauffement climatique. Les applications d'apprentissage automatique n'ont pas de limites. Avec la quantité de données disponibles de nos jours, votre seule frontière est celle de votre imagination.

## Qui devrait lire ce livre ?

Ce livre est destiné aux praticiens présents et futurs de l'apprentissage automatique qui cherchent des solutions d'implémentation pour des problèmes issus du monde réel. Il s'agit d'un livre d'introduction ne nécessitant pas de connaissances préalables sur l'apprentissage automatique ou sur l'intelligence artificielle (IA). Nous allons nous appuyer sur l'utilisation du langage de programmation Python et de la libraire `scikit-learn`, et développer toutes les étapes permettant de créer une application d'apprentissage automatique réussie. Les méthodes que nous présentons dans le livre seront utiles pour les scientifiques comme pour les chercheurs, mais également pour les *data scientists* devant travailler sur des applications commerciales. Vous tirerez le meilleur parti du livre si vous êtes déjà quelque peu familiarisé avec Python ainsi qu'avec les librairies NumPy et `matplotlib`.

Nous nous sommes consciencieusement efforcés de ne pas trop nous focaliser sur les aspects mathématiques de l'apprentissage automatique, mais plutôt sur les utilisations pratiques de ces

algorithmes. Du fait que les mathématiques, et en particulier la théorie des probabilités, constituent la fondation sur laquelle repose l'apprentissage automatique, nous ne pourrons donc pas, partant de ce postulat, analyser en détail tous ces algorithmes. Si cet aspect proprement mathématique vous intéresse (et si l'anglais scientifique ne vous rebute pas), nous vous recommandons la lecture du remarquable (et copieux) ouvrage *The Elements of Statistical Learning* (Springer) de Trevor Hastie, Robert Tibshirani et Jerome Friedman. Vous pouvez le télécharger gratuitement à l'adresse suivante : `https://web.stanford.edu/~hastie/Papers/ESLII.pdf`. De même, nous ne nous attacherons pas à l'écriture d'algorithmes d'apprentissage automatique, mais nous nous focaliserons essentiellement sur l'utilisation de la vaste panoplie de modèles déjà implémentée dans `scikit-learn` et d'autres librairies.

## Pourquoi nous avons écrit ce livre

Il existe de nombreux ouvrages consacrés à l'intelligence artificielle et à l'apprentissage automatique. Cependant, ils sont presque toujours destinés à des étudiants de fin de cycle ou en doctorat, et sont remplis de mathématiques avancées. Cela constitue un contraste considérable avec la manière dont l'apprentissage automatique est utilisé en pratique, en tant qu'outil de base dans des applications commerciales et dans certains secteurs de la recherche. De nos jours, appliquer un algorithme d'apprentissage automatique ne nécessite plus d'être au moins doctorant… Cependant, on trouve finalement peu de ressources s'attachant spécialement à tous les aspects importants de l'implémentation de l'apprentissage automatique d'un point de vue pratique, et sans vous obliger à prendre des cours de mathématiques avancées. Nous espérons que ce livre sera utile aux personnes qui veulent se lancer dans l'apprentissage automatique, sans avoir pour autant un bagage de haute volée en calcul, en algèbre linéaire et en théorie des probabilités.

## Naviguer dans le livre

Ce livre est composé de huit chapitres :

- Le Chapitre 1 présente les concepts fondamentaux de l'apprentissage automatique et de ses applications. Il décrit également la configuration que nous utiliserons tout au long du livre.
- Les Chapitres 2 et 3 décrivent les algorithmes d'apprentissage automatique qui sont les plus largement utilisés en pratique. Ils évaluent également leurs avantages et leurs inconvénients.
- Le Chapitre 4 discute de l'importance de la manière de représenter les données traitées par l'apprentissage automatique, et des aspects des données auxquels il convient de prêter particulière attention.

- Le Chapitre 5 traite de méthodes avancées pour l'évaluation des modèles et l'ajustement des paramètres, avec un accent particulier mis sur la validation croisée et la recherche sur grille.
- Le Chapitre 6 explique le concept de pipeline pour le chaînage des modèles et l'encapsulation du flux de travail.
- Le Chapitre 7 montre comment appliquer les méthodes décrites dans les chapitres précédents aux données textuelles. Il présente aussi certaines techniques de traitement spécifique à ce type de données.
- Le Chapitre 8, enfin, offre un point de vue plus général sur les sujets traités dans les autres chapitres.

Même si les Chapitres 2 et 3 décortiquent les principaux algorithmes de l'apprentissage automatique, comprendre ceux-ci n'est pas forcément nécessaire pour un débutant. Si vous avez besoin de construire rapidement un système d'apprentissage automatique, nous vous suggérons de commencer par le Chapitre 1, puis d'étudier les premières sections du Chapitre 2 où vous trouverez tous les concepts clés. Vous pouvez ensuite passer directement à la fin du Chapitre 2, à la section « Résumé et perspectives » qui liste tous les modèles supervisés couverts ici. Choisissez le modèle qui vous semble convenir le mieux à vos besoins, et revenez en arrière pour découvrir la section qui lui est consacrée. Vous pouvez ensuite utiliser les techniques décrites dans le Chapitre 5 pour évaluer et améliorer votre modèle.

## Conventions utilisées dans ce livre

Dans ce livre, nous utiliserons les conventions typographiques suivantes :

*Italique*

- Indique un terme nouveau, ou nom de fichier ou encore une extension.

`Largeur constante`

- Cette typographie est utilisée dans les listings de programmes, ainsi que dans tout ce qui peut dans un paragraphe faire référence à un nom de librairie, de commande ou de module, de variable ou de fonction, d'instruction, de type de donnée ou encore de mot-clé.



Cet élément signale une astuce ou une suggestion.



Cet élément signale une note d'ordre général.



Cette icône attire l'attention sur un avertissement ou un problème potentiel.

## Utiliser les exemples de code

Tout d'abord, ce livre s'appuie intensivement sur la librairie `scikit-learn`. Pour retrouver la documentation « officielle » des classes et des fonctions de `scikit-learn`, voyez les pages du site (en anglais) :

```
http://scikit-learn.org/stable/
```

D'autre part, tout le code de ce livre est disponible gratuitement à l'adresse :

```
https://github.com/amueller/introduction_to_ml_with_python
```

Ce code est regroupé en chapitres qu'il est possible de télécharger sous la forme de *notebooks* Jupyter que vous pouvez exécuter dans votre navigateur Web par défaut.



*N. d. T.* : Notez bien que, puisque tout le code est accessible directement à partir du site GitHub des auteurs, il n'est pas traduit dans le texte de ce livre (ce qui ne ferait que risquer d'y introduire des erreurs). Cette remarque s'applique aussi aux bases de données publiques utilisées par les auteurs, et donc également aux illustrations de l'ouvrage qui en résultent. En revanche, l'ensemble de ce code a été testé et vérifié, en ajoutant lorsque c'est nécessaire des indications sur les procédures d'installation complémentaires permettant d'obtenir les résultats escomptés.

Si vous devez, pour une raison ou pour une autre, vous resservir de parties de ce code pour votre propre usage, faites-le librement. Par contre, nous vous demandons d'en citer la source dans une éventuelle publication. C'est juste une question de respect du droit d'auteur en particulier, et du droit des auteurs en général.

Enfin, nous tenons à remercier très vivement l'ensemble de la communauté qui participe au développement open source de Python, et en particulier toutes celles et tous ceux qui contribuent à l'amélioration de la librairie `scikit-learn`. D'ailleurs, sans l'appui et l'aide de cette communauté, et notamment de Gaël Varoquaux, Alex Gramfort et Olivier Grisel (trois noms qui montrent combien les scientifiques français sont à la pointe du progrès !), Andreas Mueller ne serait jamais devenu un contributeur majeur du projet `scikit-learn`. Il convient en même temps de remercier aussi toutes les personnes qui donnent de leur temps pour gérer et améliorer cette remarquable librairie.

# CHAPITRE 1
# Introduction

L'apprentissage automatique, le *machine learning* donc, concerne l'extraction de connaissances à partir de données. Il s'agit d'un champ de recherches qui se situe à l'intersection des statistiques, de l'intelligence artificielle et de l'informatique. On parle également à son sujet d'analyse prédictive ou encore d'apprentissage statistique. L'application des méthodes d'apprentissage automatique s'est démultipliée dans la vie quotidienne au cours de ces dernières années. Il peut s'agir de recommandations sur le film que vous *devriez* regarder, sur la nourriture (ou tout autre produit) que vous *devriez* acheter, sur les radios en ligne que vous *devriez* écouter. Mais il peut aussi s'agir par exemple de reconnaître vos amis sur des photos, un risque de tumeur cancéreuse sur des analyses, et bien d'autres choses encore. De nombreux sites Web modernes ainsi que certains appareils intègrent des algorithmes d'apprentissage automatique. Si vous visitez un site Web complexe, comme Facebook, Amazon ou encore Netflix, il est plus que probable que chaque partie qui le constitue contient de multiples modèles d'apprentissage automatique.

En dehors des applications commerciales, l'apprentissage automatique a une influence énorme sur la manière dont les recherches basées sur des données sont effectuées aujourd'hui. Les outils présentés dans ce livre ont été appliqués à divers problèmes scientifiques, comme comprendre les étoiles, trouver des planètes distantes, découvrir de nouvelles particules, analyser des séquences d'A.D.N., ou encore fournir des traitements personnalisés pour des malades atteints de cancer.

Vos propres applications n'ont cependant pas besoin de couvrir des champs d'application aussi étendus, pas plus que votre but n'est sans doute de changer le monde à vous tout seul. Pour autant, dans ce chapitre, nous allons expliquer pourquoi l'apprentissage automatique est devenu si populaire et pourquoi il permet de résoudre tant de problèmes. Nous vous montrerons également comment produire votre premier modèle d'apprentissage automatique, et donc nous vous présenterons en même temps un certain nombre de concepts importants dans ce domaine.

## Pourquoi l'apprentissage automatique ?

Dans les premiers temps des applications dites « intelligentes », de nombreux systèmes utilisaient des règles figées dans le code, basées sur des décisions de type « si... sinon... », pour traiter les données ou encore ajuster les entrées de l'utilisateur. Songez par exemple à un filtre antispam. Son travail consiste à déplacer des messages électroniques entrants dans un dossier spécifique. Vous pourriez créer une liste noire de termes qui ferait en sorte qu'un message soit marqué comme étant un spam. Il s'agirait d'un exemple typique de système de règles « expert » servant à créer une application « intelligente ». De telles règles décisionnelles, ciselées à la main, sont tout à fait applicables dans le cas de certaines applications, en particulier celles pour lesquelles les êtres humains ont une bonne compréhension des processus à modéliser. Cependant, cette façon de procéder présente deux désavantages majeurs :

- La logique nécessaire à la prise de décision est spécifique à un certain domaine et une certaine tâche. Modifier celle-ci, même légèrement, peut impliquer la réécriture de l'ensemble du système.
- Concevoir des règles nécessite une compréhension approfondie de la manière dont une décision devrait être prise par un expert humain.

Un exemple intéressant des limites de ce type d'approche est représenté par la reconnaissance des visages. Aujourd'hui, n'importe quel smartphone est capable de détecter un visage dans une image. Pourtant, ce problème était resté insoluble jusqu'en 2001. La principale difficulté était que la façon dont les pixels qui constituent une image numérique sont « perçus » par l'ordinateur est extrêmement différente de la manière dont les êtres humains fonctionnent. De ce fait, il était fondamentalement impossible pour un humain de définir un bon système de règles permettant de décrire ce qui constitue un visage dans une photo numérique.

Utiliser un système d'apprentissage automatique, à l'inverse, revient « simplement » à présenter à un programme une grande collection d'images de personnes afin qu'un algorithme puisse déterminer les caractéristiques nécessaires à la détection et l'identification des visages.

## Les problèmes que l'apprentissage automatique peut résoudre

Les types les plus efficients d'algorithmes d'apprentissage automatique sont ceux qui automatisent des processus de prise de décision en généralisant des données fournies à partir d'exemples connus. Dans ce schéma, également appelé *apprentissage supervisé*, l'utilisateur alimente en quelque sorte l'algorithme en lui fournissant des paires entrée/sortie désirées, et cet algorithme trouve ensuite une manière de produire la sortie désirée à partir d'une certaine entrée. En particulier, l'algorithme est capable de créer une sortie depuis une entrée qu'il n'a jamais rencontrée auparavant, et ce sans aucune aide humaine. Revenons à notre exemple de classification des

spams. Avec un système d'apprentissage automatique, l'utilisateur va fournir à l'algorithme un grand nombre de messages électroniques (qui constituent donc l'entrée) ainsi que des informations indiquant quels messages sont des spams, et lesquels n'en sont pas (ce qui forme la sortie désirée). Par la suite, cet algorithme sera capable de produire une prédiction sur le fait qu'un nouvel email est ou non un spam.

Les algorithmes qui apprennent à partir de paires entrée/sortie sont dits *supervisés*, car les choses se déroulent comme si un « professeur » supervisait l'apprentissage automatique en expliquant le lien entre entrées et sorties désirées. Alors que la création de jeux de données pour les entrées et les sorties est souvent un processus manuel très laborieux, les algorithmes d'apprentissage automatique supervisés sont bien compris et leurs performances sont faciles à mesurer. Si votre application peut être formulée sous la forme d'un problème d'apprentissage supervisé, et si vous êtes capable de créer un jeu de données incluant la sortie désirée, alors l'apprentissage automatique sera probablement à même de résoudre votre problème.

Voici quelques exemples d'applications de l'apprentissage supervisé :

- *Identifier un code postal parmi des chiffres écrits à la main sur une enveloppe.* Ici, l'entrée est une numérisation de l'écriture manuscrite, et la sortie désirée est une suite de chiffres formant un code postal. Pour créer un jeu de données adapté à un tel algorithme, il est nécessaire de numériser un grand nombre d'enveloppes. Vous pouvez alors lire vous-même les codes, et enregistrer les chiffres correspondants en tant que sorties désirées.
- *Déterminer si une tumeur est bénigne à partir d'une image médicale.* Ici, l'entrée est l'image, et la sortie doit indiquer si la tumeur est bénigne. Pour créer un jeu de données afin de bâtir ce modèle, vous devez disposer d'une base de données formée d'images médicales. Vous avez également besoin de l'avis d'un expert, et donc un docteur doit consulter toutes ces images et décider quelles tumeurs sont bénignes, et lesquelles ne le sont pas. Il peut même être nécessaire de pousser le diagnostic plus loin que l'imagerie elle-même afin de déterminer si la tumeur visible sur une radio est ou non cancéreuse.
- *Détecter une activité frauduleuse dans des transactions faites avec une carte de crédit.* Ici, l'entrée est un enregistrement d'une transaction effectuée avec une carte de crédit, et la sortie doit indiquer si cette transaction est susceptible d'être frauduleuse ou non. En supposant que vous représentez l'entité qui fournit ces cartes de crédit, la constitution du jeu de données va consister à enregistrer toutes les transactions, et à marquer celles pour lesquelles un client a signalé une fraude.

Une chose intéressante à noter avec ces exemples, c'est que, même si les entrées et les sorties semblent assez évidentes dans leur principe, le processus de collecte des données est à chaque fois extrêmement différent. Ainsi, la lecture des enveloppes est une tâche fastidieuse, mais en même temps plutôt facile et peu onéreuse. Obtenir une imagerie médicale et des diagnostics, d'un autre côté, nécessite non seulement l'utilisation d'un matériel coûteux, mais aussi une expertise rare et chère, sans même parler des problèmes éthiques et de la protection de la vie

privée. Dans le cas des cartes bancaires, la collecte des données est bien plus simple. Ce sont les clients eux-mêmes qui vous fournissent les sorties dont vous avez besoin en signalant les transactions qui sont frauduleuses. Tout ce que vous avez à faire pour récupérer les informations dont vous avez besoin afin de constituer votre jeu de données, c'est d'attendre.

Les *algorithmes non supervisés* sont l'autre type d'algorithme que nous allons étudier dans ce livre. Dans ce mode d'apprentissage non supervisé, seules les données en entrées sont déterminées, et aucune sortie connue n'est fournie à l'algorithme. Bien que ces méthodes permettent de créer avec succès de nombreuses applications, elles sont généralement plus difficiles à comprendre et à évaluer.

Voici quelques exemples d'application de l'apprentissage non supervisé :

- *Identifier des sujets dans des posts sur un blog ou un forum.* Si vous disposez d'une vaste collection de données textuelles, vous pouvez souhaiter en faire une compilation afin d'en extraire les thèmes les plus répandus. Mais vous ne savez pas nécessairement à l'avance quels peuvent être ces sujets, pas plus que le nombre de ceux-ci. Par conséquent, vous ne connaissez pas les sorties susceptibles d'être produites.
- *Segmenter une clientèle en groupes ayant des préférences similaires.* Partant d'enregistrements concernant vos clients, vous pourriez vouloir identifier lesquels présentent des similitudes, et s'il est possible de former des groupes avec des préférences de même nature. Dans le cas d'un site de vente en ligne, ces références pourraient par exemple se définir comme « parents », « rats de bibliothèque » ou encore « joueurs ». Comme vous ne savez pas à l'avance ce que pourraient être ces groupes, pas plus que leur nombre, vous ne disposez pas de sorties connues.
- *Détection de séries d'accès anormaux à un site Web.* Pour identifier des abus (ou des bogues), il est souvent utile de trouver des schémas d'accès qui diffèrent des procédures normales. Chaque schéma anormal peut être très différent des autres, et il se peut donc que vous n'ayez pas une trace enregistrée de ce type de comportement. Comme, dans cet exemple, vous ne pouvez rien faire d'autre que d'observer un trafic, et que vous ne savez pas par avance ce qui constitue un comportement normal ou anormal, il s'agit bien d'un problème non supervisé.

Dans les tâches d'apprentissage supervisées comme non supervisées, il est important d'avoir une représentation de vos données d'entrée qui soit compréhensible par un ordinateur. Souvent, il est utile de vous figurer vos données comme formant un tableau. Chaque point de donnée sur lequel vous voulez raisonner (chaque email, chaque client ou encore chaque transaction) est une ligne, et chaque propriété qui décrit ce point de donnée (comme par exemple l'âge d'un client, ou bien le lieu ou encore le montant d'une transaction) est une colonne. Vous pourriez décrire les utilisateurs selon leur âge, leur sexe, la date de création de leur compte, le nombre d'achats réalisés sur votre boutique en ligne, et ainsi de suite. De même, vous pourriez décrire l'image

d'une tumeur en fonction du niveau de gris de chaque pixel, ou bien par exemple en utilisant la taille, la forme et la couleur de la tumeur.

Chaque entité, ou ligne, formant un point de donnée destiné à être représentatif de l'ensemble s'appelle ici un échantillon. Les colonnes, de leur côté, donc les propriétés qui décrivent ces entités, sont appelées des *caractéristiques* (*features*), ou encore *variables*.

Plus loin dans ce livre, nous étudierons plus en détail la problématique de la réalisation d'une bonne représentation de vos données, c'est-à-dire de ce que l'on appelle généralement l'*extraction des caractéristiques*. Mais vous ne devriez cependant jamais perdre de vue le fait qu'aucun algorithme d'apprentissage automatique ne peut être capable de faire une prédiction à partir de données sur lesquelles il n'a pas d'informations. Par exemple, si le seul renseignement que vous possédez sur un patient ou une patiente est son nom de famille, il est évident que jamais un algorithme ne sera en mesure de prédire son sexe. Cette information est tout simplement totalement absente de vos données. Si vous ajoutez une autre caractéristique contenant le prénom de la personne, vous aurez plus de chance d'arriver à une prédiction correcte.

## Comprendre votre but et comprendre vos données

La partie sans doute la plus importante du processus d'apprentissage automatique consiste à comprendre les données sur lesquelles vous travaillez, et comment elles se relient au problème que vous voulez résoudre. Choisir un algorithme de manière aléatoire et lui transmettre vos données est à l'évidence inefficace de ce point de vue. Il est nécessaire de comprendre ce qui se passe dans votre jeu de données avant de commencer à bâtir un modèle. Chaque algorithme est différent en termes de nature des données à traiter, et du type de problème pour lequel il est essentiellement conçu. Lorsque vous construisez une solution d'apprentissage automatique, vous devriez vous poser, ou du moins garder présentes à l'esprit, les questions suivantes :

- À quoi est-ce que j'essaie de répondre ? Plus précisément, est-ce que je pense que les données collectées peuvent répondre à cette question ?
- Quelle est la meilleure manière de rédiger ma question (ou mes questions) en tant que problème d'apprentissage automatique ?
- Est-ce que j'ai collecté suffisamment de données pour représenter le problème que je veux résoudre ?
- Quelles caractéristiques des données dois-je extraire, et vont-elles permettre d'obtenir les bonnes prédictions ?
- Comment puis-je mesurer dans mon application la réussite de ma démarche ?
- Comment la solution d'apprentissage automatique va-t-elle interagir avec d'autres parties de mes recherches ou de mes activités ?

Dans un contexte plus large, les algorithmes et les méthodes de l'apprentissage automatique représentent simplement une partie d'un processus plus large de résolution d'un problème particulier, et il est bon de garder en permanence cette notion présente à l'esprit. Trop de gens passent beaucoup de temps à construire des solutions d'apprentissage automatique complexes, pour finir par s'apercevoir qu'elles ne résolvent pas le bon problème !

Lorsque l'on se plonge dans les aspects techniques de l'apprentissage automatique (ce qui est tout de même l'objet de ce livre), il est facile de perdre de vue les buts ultimes. Même si nous n'allons pas dans le cadre de cet ouvrage discuter en détail des questions soulevées ici, nous vous encourageons fermement à bien enregistrer toutes les suppositions, explicites ou implicites, que vous allez être amené à faire lorsque vous allez commencer à bâtir vos modèles d'apprentissage automatique.

## Pourquoi Python ?

Python est devenu la *lingua franca* de nombreuses applications dans le domaine de la science des données (ou *data science*). Il combine la puissance d'un langage de programmation généraliste avec la facilité d'utilisation de langages de script plus spécifiques à certains domaines, comme MATLAB ou R. Python possède des librairies pour le chargement et la visualisation de données, pour les calculs statistiques, pour les traitements en langage naturel, pour le travail sur les images, et bien plus encore. Cette vaste boîte à outils offre aux spécialistes (les fameux *data scientists*) une très large panoplie de fonctionnalités aussi bien générales que très spécifiques. L'un des principaux avantages offerts par l'utilisation de Python est la capacité à interagir directement avec le code en passant par un terminal ou bien par d'autres outils, comme Jupyter Notebook sur lequel nous allons revenir très bientôt. L'apprentissage automatique et l'analyse des données sont des processus fondamentalement itératifs, dans lesquels l'analyse est pilotée par les données. Il est essentiel pour ces processus de disposer d'outils qui permettent des itérations rapides et une interaction simple.

En tant que langage de programmation généraliste, Python est aussi un atout pour la création d'interfaces graphique utilisateur (ou GUI) et de services web complexes, tout en s'intégrant pleinement avec des systèmes existants.

## scikit-learn

`scikit-learn` est un projet open source, ce qui signifie qu'il est libre tant d'utilisation que de diffusion, et que tout un chacun peut facilement en obtenir le code source pour voir ce qui se passe dans les coulisses. Ce projet est développé et amélioré en permanence. Il s'appuie sur une communauté d'utilisateurs très active. Il contient de nombreux algorithmes d'apprentissage

automatique à la pointe de la recherche, ainsi qu'une solide documentation sur ces algorithmes (voir l'adresse `http://scikit-learn.org/stable/documentation`, en anglais bien sûr).

`scikit-learn` est un outil très populaire, et la librairie Python la plus importante pour tout ce qui concerne l'apprentissage automatique. Il est largement utilisé dans l'industrie comme dans l'université, et vous pouvez trouver en ligne une horde de tutoriels et de fragments de code. De surcroît, `scikit-learn` fonctionne parfaitement conjointement avec d'autres outils scientifiques dédiés à Python, sur lesquels nous allons revenir un peu plus loin dans ce chapitre.

Nous vous recommandons également de parcourir le guide utilisateur de scikit-learn (à l'adresse `http://scikit-learn.org/stable/user_guide.html`) ainsi que la documentation de l'API pour plus de détails et découvrir bien d'autres options pour chaque algorithme. La documentation en ligne est très approfondie, et ce livre devrait vous fournir tous les prérequis dans le domaine de l'apprentissage automatique dont vous avez besoin pour la comprendre en détail.

## Installer scikit-learn

`scikit-learn` dépend de deux autres packages Python, *NumPy* et *SciPy*. Pour des travaux de tracé et de développement interactif, vous devriez aussi installer `matplotlib`, `IPython` et `Jupyter Notebook`. Nous vous recommandons d'utiliser l'une des distributions déjà très complètes de Python suivantes :

### Anaconda

Cette distribution Python est conçue pour des traitements intensifs de données, l'analyse prédictive et les calculs scientifiques (voir l'adresse `https://www.anaconda.com/download/`). Anaconda est fourni avec `NumPy`, `SciPy`, `matplotlib`, `pandas`, `IPython`, `Jupyter Notebook` et `scikit-learn`.

Disponible sous macOS, Windows et Linux, Anaconda fournit une solution très commode, et nous le recommandons pour les personnes qui n'ont pas encore installé les packages scientifiques de Python. Anaconda contient aussi gratuitement la version commerciale de la librairie Intel MKL. Utiliser celle-ci (ce qui est automatiquement fait lorsqu'Anaconda est installé) peut améliorer de façon significative la rapidité d'exécution de nombreux algorithmes de `scikit-learn`.

### Enthought Canopy

Il s'agit d'une autre distribution dédiée à l'informatique scientifique. Elle est fournie avec `NumPy`, `SciPy`, `matplotlib`, `pandas` et `IPython`. En revanche, la version gratuite ne contient pas `scikit-learn`. Si vous êtes étudiant ou professeur, vous pouvez demander une licence spécifique. Enthought Canopy est disponible pour les versions 2.7 et 3.5 de Python, et ce aussi bien sous Windows et Linux que sous macOS.

### Python(x,y)

Il s'agit d'une distribution libre pour les traitements informatiques scientifiques, spécifiquement sous Windows. Python(x,y) est livré avec `NumPy`, `SciPy`, `matplotlib`, `pandas`, `IPython` et `scikit-learn`.

Si vous avez une installation de Python déjà configurée, vous pouvez vous servir de `pip` pour y incorporer tous les packages voulus :

```
$ pip install numpy scipy matplotlib ipython scikit-learn pandas pillow
```



Avec Anaconda, il est préférable d'utiliser `conda` pour cette installation. Voyez par exemple à ce sujet la page `https://anaconda.org/anaconda/repo`.

Pour les visualisations d'arbres du Chapitre 2, vous aurez aussi besoin d'installer les packages graphviz. Nous reviendrons le moment venu sur son installation.

## Librairies et outils essentiels

Comprendre ce qu'est `scikit-learn` et comment l'utiliser est important, mais il existe plusieurs autres librairies qui vous permettront d'améliorer votre expérience. `scikit-learn` est construit au-dessus des librairies scientifiques `NumPy` et `SciPy` de Python. En plus de `NumPy` et de `SciPy`, nous utiliserons également `pandas` et `matplotlib`. Nous allons aussi vous présenter Jupyter Notebook, qui est un environnement de programmation interactif s'exécutant dans la fenêtre de votre navigateur par défaut. Voici quelques notions complémentaires à ce sujet (si vous n'êtes pas familiarisé avec `NumPy` ou `matplotlib`, nous vous conseillons de lire le premier chapitre des notes de lecture SciPy, à l'adresse `http://www.scipy-lectures.org/`).

### Jupyter Notebook

Jupyter Notebook est un excellent environnement interactif pour exécuter du code dans une fenêtre de navigation. C'est un outil très commode pour explorer des analyses de données, et il est largement employé par les *data scientists*. Jupyter Notebook supporte un grand nombre de langages de programmation, mais seul Python nous intéresse dans ce qui suit. Il permet en particulier d'incorporer facilement du code, du texte et des images. En fait, tous les développements proposés dans ce livre ont été écrits à l'aide de Jupyter Notebook. Le code source peut être retrouvé et téléchargé depuis la page dédiée sur GitHub, à l'adresse `https://github.com/amueller/introduction_to_ml_with_python` (c'est d'ailleurs pourquoi les listings du livre reprennent ce code tel quel).

## NumPy

NumPy est l'un des packages fondamentaux pour tout ce qui concerne les traitements scientifiques dans Python. Il contient des fonctionnalités dédiées aux tableaux multidimensionnels, des fonctions mathématiques de haut niveau, comme les opérations d'algèbre linéaire ou la transformation de Fourier, ainsi que des générateurs de nombres pseudo-aléatoires.

Dans `scikit-learn`, les tableaux NumPy constituent la structure de données fondamentale. En effet, `scikit-learn` prend ses données sous la forme de tableaux NumPy. Toutes les données que vous utilisez doivent donc être converties en un tableau NumPy. La fonctionnalité centrale de NumPy est la classe `ndarray`, qui est un tableau multidimensionnel (à *n* dimensions). Tous les éléments du tableau doivent être de même type. Un tableau NumPy peut ressembler à ceci :

**In[1]:**

```
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6]])
print("x:\n{}".format(x))
```

**Out[1]:**

```
x:
[[1 2 3]
 [4 5 6]]
```

Dans ce livre, nous allons utiliser NumPy de manière plutôt intensive. De ce fait, nous ferons régulièrement référence aux objets de sa classe `ndarray` sous une forme plutôt générique, comme « tableaux NumPy » ou même simplement « tableaux ».

## SciPy

SciPy est une collection de fonctions dédiées aux calculs scientifiques sous Python. Elle fournit, parmi d'autres fonctionnalités, des routines avancées pour l'algèbre linéaire, pour l'optimisation de fonctions mathématiques, pour le traitement des signaux, ainsi que des fonctions mathématiques spéciales et des distributions statistiques. `scikit-learn` s'appuie sur la collection de fonctions de SciPy pour implémenter ses algorithmes. La partie la plus importante de SciPy pour ce qui nous concerne ici est `scipy.sparse`. Elle fournit ce que l'on appelle des *matrices creuses* (*sparce matrices*), qui constituent une autre représentation employée pour les données dans `scikit-learn`. Les matrices creuses sont utilisées chaque fois que nous voulons enregistrer un tableau 2D contenant essentiellement des zéros :

**In[2]:**

```
from scipy import sparse

eye = np.eye(4)
print("NumPy array:\n{}".format(eye))
```

**Out[2]:**

```
NumPy array:
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

**In[3]:**

```
sparse_matrix = sparse.csr_matrix(eye)
print("\nSciPy sparse CSR matrix:\n{}".format(sparse_matrix))
```

**Out[3]:**

```
SciPy sparse CSR matrix:
  (0, 0)    1.0
  (1, 1)    1.0
  (2, 2)    1.0
  (3, 3)    1.0
```

En général, il n'est pas possible de créer des représentations denses de données creuses (car elles ne tiendraient pas en mémoire). Nous devons donc créer directement des représentations creuses. Les instructions qui suivent illustrent un moyen de créer la même matrice creuse que ci-dessus en utilisant le format COO (pour *Coordinate Format*) :

**In[4]**

```
data = np.ones(4)
row_indices = np.arange(4)
col_indices = np.arange(4)
eye_coo = sparse.coo_matrix((data, (row_indices, col_indices)))
print("COO representation:\n{}".format(eye_coo))
```

**Out[4]:**

```
COO representation:
  (0, 0)    1.0
  (1, 1)    1.0
  (2, 2)    1.0
  (3, 3)    1.0
```

Vous trouverez plus de détails sur les matrices creuses en consultant les pages du site consacré aux notes de lecture SciPy (à l'adresse `http://www.scipy-lectures.org/`)

## matplotlib

`matplotlib` est la principale libraire de tracé scientifique pour Python. Elle fournit des fonctions permettant de réaliser des visualisations de grande qualité, telles que des graphiques linéaires, des histogrammes, des diagrammes de dispersion et ainsi de suite. Visualiser vos données ainsi que différents aspects de votre analyse peut vous fournir de précieuses indications. Nous utiliserons dans ce livre `matplotlib` pour toutes nos visualisations. Lorsque vous travaillez avec Jupyter Notebook, vous pouvez afficher des graphiques directement dans la fenêtre du naviga-

teur en vous servant des commandes `%matplotlib notebook` et `%matplotlib inline`. Nous vous recommandons d'utiliser `%matplotlib notebook` qui fournit un environnement interactif (même si, pour les besoins de ce livre, vous faisons appel à `%matplotlib inline`). Par exemple, le code qui suit produit le tracé illustré sur la Figure 1.1 :

**In[5]:**

```
%matplotlib inline
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 100)
y = np.sin(x)
plt.plot(x, y, marker="x")
```

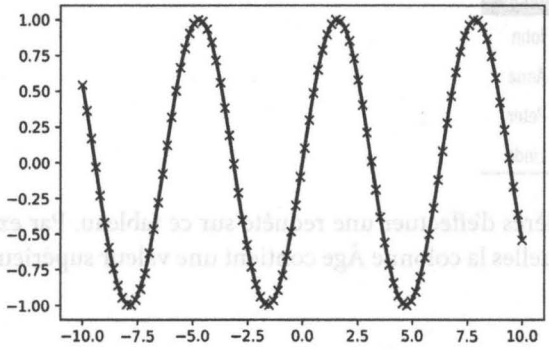



*Figure 1.1 : Tracé de la fonction sinus à l'aide de matplotlib.*

## pandas

`pandas` est une librairie Python servant à la manipulation et à l'analyse des données. Elle est bâtie autour d'une structure de données appelée `DataFrame`. Pour faire simple, disons qu'un `DataFrame` pandas est un tableau comparable à une feuille de calcul Excel. `Pandas` fournit un grand nombre de méthodes pour modifier et traiter ce tableau. En particulier, il permet d'effectuer des requêtes de style SQL et d'opérer des jointures. À la différence de `NumPy`, qui oblige toutes les entrées à être de même type, `pandas` autorise des types différents dans les colonnes (par exemple des entiers, des dates, des nombres en virgule flottante et des chaînes de caractères). Un autre intérêt important de `pandas`, c'est qu'il est capable de travailler avec un grand nombre de formats de fichiers et de bases de données, comme SQL, des fichiers Excel ou encore CSV. L'étude de pandas dépasse évidemment largement le cadre de ce livre, et nous ne pouvons que vous conseiller de vous tourner vers la littérature spécialisée. Voici cependant un petit exemple de création d'un `DataFrame` utilisant un dictionnaire :

In[6]:

```
import pandas as pd
from IPython.display import display

data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location' : ["New York", "Paris", "Berlin", "London"],
        'Age' : [24, 13, 53, 33]
       }

data_pandas = pd.DataFrame(data)
display(data_pandas)
```

Ce code produit la sortie suivante :

| | Âge | Location | Name |
|---|---|---|---|
| 0 | 24 | New York | John |
| 1 | 13 | Paris | Anna |
| 2 | 53 | Berlin | Peter |
| 3 | 33 | London | Linda |

Il existe plusieurs manières d'effectuer une requête sur ce tableau. Par exemple, pour sélectionner les lignes dans lesquelles la colonne Âge contient une valeur supérieure à 30 pourrait s'écrire ainsi : :

In[7]:

```
display(data_pandas[data_pandas.Age > 30])
```

Le résultat produit serait alors celui-ci :

| | Âge | Location | Name |
|---|---|---|---|
| 2 | 53 | Berlin | Peter |
| 3 | 33 | London | Linda |

## mglearn

Tout le code proposé dans ce livre est accessible sur GitHub, à l'adresse `https://github.com/amueller/introduction_to_ml_with_python`. En plus des exemples du livre, vous y trouverez également la librairie `mglearn`. Il s'agit d'un jeu de fonctions utilitaires écrit spécialement pour cet ouvrage, et ce afin d'alléger les listings pour ne pas s'encombrer avec les problèmes de tracé ou de chargement de données. Si vous êtes intéressé, vous pouvez étudier les fichiers stockés dans le sous-dossier `mglearn` sur GitHub. Cependant, les détails de ce module ne sont pas réellement importants pour suivre les explications de ce livre. Quand vous voyez un appel à `mglearn` dans le code, il s'agit généralement d'une façon d'obtenir rapidement un joli tracé, ou encore de mettre

la main sur certaines données intéressantes. Si vous exécutez le code directement depuis le dépôt sur GitHub, `mglearn` est déjà en place et vous n'avez à vous soucier de rien. Si vous voulez pouvoir appeler des fonctions `mglearn` dans un autre contexte, la meilleure manière de s'y prendre consiste à l'installer en appelant la commande `pip install mglearn`.



Ce livre fait un ample usage des librairies `NumPy`, `matplotlib` et `pandas`. Tout le code développé ici suppose donc que soient réalisées les importations suivantes :

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import mglearn
from IPython.display import display
```

Nous supposons également que vous allez exécuter le code dans une fenêtre Jupyter Notebook, avec la présence des lignes `%matplotlib notebook` ou `%matplotlib inline` pour l'affichage des tracés. Si vous n'utilisez pas Jupyter Notebook ou ces commandes magiques, vous devrez appeler `plt.show` pour visualiser les figures.

## Python 2 versus Python 3

À l'heure actuelle, il existe deux versions majeures de Python qui sont très largement utilisées : Python 2 (plus précisément, 2.7) et Python 3 (plus exactement, 3.5 voire même 3.6). Cela peut engendrer une certaine confusion. Python 2 n'est plus développé de manière active. Mais, du fait que Python 3 introduit des changements majeurs, il n'est pas forcément compatible avec du code écrit pour Python 2. Si vous découvrez Python, ou si vous débutez un nouveau projet, nous vous recommandons très vivement de vous en tenir à Python 3 sans vous poser de questions. Si vous disposez d'une vaste bibliothèque de code qui a été écrite pour Python 2, vous êtes pardonné d'avance si vous conservez votre installation actuelle. Cependant, vous devriez essayer de migrer vers Python 3 dès que possible. Dans le cas d'un nouveau code, il est pour l'essentiel assez facile d'assurer sa compatibilité avec Python 2 et Python 3 (vous trouverez à ce sujet des outils utiles à l'adresse `https://pypi.python.org/pypi/six`). Si vous n'avez pas besoin de vous préoccuper de ces questions de compatibilité descendante, vous devriez définitivement vous en tenir à Python 3. Tout le code de ce livre est écrit de manière à pouvoir être exécuté avec Python 2 comme avec Python 3. Cependant, les sorties peuvent différer légèrement sous Python 2.

## Versions utilisées dans ce livre

Dans ce livre, nous allons utiliser les versions suivantes des librairies mentionnées plus haut :

**In[8]:**

```
import sys
print("Python version: {}".format(sys.version))

import pandas as pd
print("pandas version: {}".format(pd.__version__))

import matplotlib
print("matplotlib version: {}".format(matplotlib.__version__))

import numpy as np
print("NumPy version: {}".format(np.__version__))

import scipy as sp
print("SciPy version: {}".format(sp.__version__))

import IPython
print("IPython version: {}".format(IPython.__version__))

import sklearn
print("scikit-learn version: {}".format(sklearn.__version__))
```

**Out[8]:**

```
Python version: 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017,
  13:25:24) [MSC v.1900 64 bit (AMD64)]
pandas version: 0.20.1
matplotlib version: 2.0.2
NumPy version: 1.13.3
SciPy version: 0.19.0
IPython version: 5.3.0
scikit-learn version: 0.19.0
```

Même s'il n'est pas réellement important que vous disposiez exactement des mêmes versions, vous devriez tout de même avoir une version de `scikit-learn` qui soit au moins aussi récente que celle que nous utilisons.



Au moment où ce livre est écrit, Anaconda est fourni avec la version 0.18.1 de `scikit-learn`. Pour les besoins de l'ouvrage, il est conseillé de mettre à niveau cette librairie à sa version 0.19.0 ou ultérieures. Pour cela, rendez-vous sur la page `https://anaconda.org/anaconda/scikit-learn`. Choisissez la version voulue sous l'onglet Files (par exemple win-64/scikit-learn-0.19.0-np113py36_0.tar.bz2 pour la version 64 bits sous Windows de Python 3.6). Copiez le fichier ainsi téléchargé dans le sous-dossier `pkgs` du dossier d'installation d'Anaconda. Ouvrez ensuite la fenêtre de commande d'Anaconda, et saisissez la commande `conda update scikit-learn`. Il ne vous reste plus qu'à valider les messages pour effectuer la mise à niveau de `scikit-learn`.

Maintenant que nous avons sous la main tous les outils dont nous avons besoin, nous allons pouvoir nous lancer dans le développement d'une première application d'apprentissage automatique.



Ce livre suppose que votre version de `scikit-learn` n'est pas antérieure à la référence 0.18. Le module `model_selection` a en effet été ajouté dans la version 0.18. Si vous utilisez une version plus ancienne, vous devrez ajuster vous-même les importations de ce module. Mais il est préférable, comme expliqué ci-dessus, d'effectuer la mise à jour de `scikit-learn` vers la version la plus récente.

## Une première application : classifier des espèces d'iris

Dans cette section, vous allez découvrir une application d'apprentissage automatique simple et créer votre premier modèle. Au cours de ce processus, vous vous familiariserez avec certains concepts et termes clés.

Supposons qu'un ou une botaniste en herbe souhaite pouvoir distinguer diverses espèces d'iris. Notre scientifique a collecté un certain nombre de mesures associées à chaque iris : la longueur et la largeur des pétales, ainsi que la longueur et la largeur des sépales, toutes ces données étant mesurées en centimètres (voir la Figure 1.2).

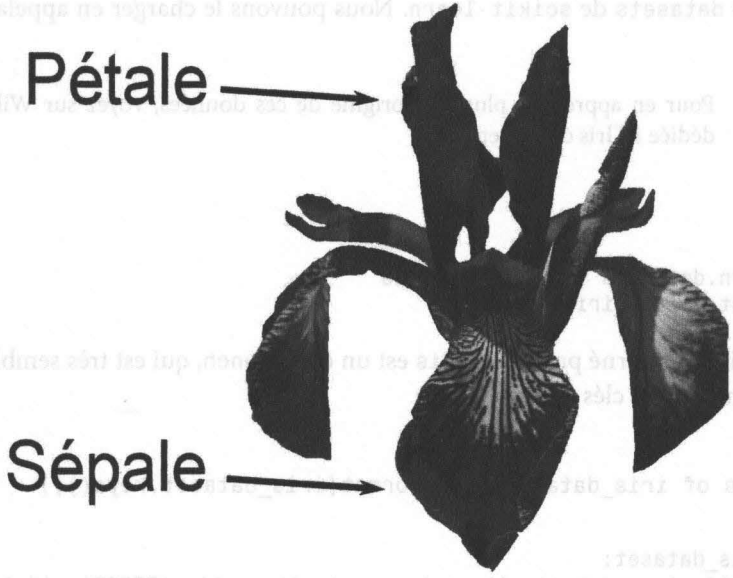



*Figure 1.2 : Les différentes parties d'un iris.*

Notre botaniste dispose également des mesures d'autres iris qui ont été précédemment identifiés par un expert et qui appartiennent aux espèces *setosa*, *versicolor* ou *virginica* (toutes répandues dans l'Amérique du Nord). Pour ces mesures, il y a donc une certitude quant aux espèces auxquelles elles correspondent. Supposons pour notre propos qu'il s'agit des seules espèces que notre botaniste est susceptible de rencontrer au cours de ses sorties.

Notre but est de construire un modèle d'apprentissage automatique capable d'être *entraîné* à partir des mesures des iris dont les espèces sont connues, de manière à pouvoir prédire l'espèce à laquelle appartiendrait un nouvel exemplaire de fleur.

Du fait que nous disposons de mesures pour lesquelles nous connaissons les espèces correctes d'iris, il s'agit bien d'un problème d'apprentissage supervisé. Ici, nous voulons prédire une option parmi plusieurs (en l'occurrence, une espèce d'iris). Il s'agit donc d'un problème de *classification*. Les sorties possibles (les espèces d'iris) sont appelées des *classes*. Chaque iris du jeu de données appartient à l'une des trois classes. En résumé, il s'agit d'un problème de classification à trois classes (classification dite *multiclasse*).

La sortie désirée pour un certain point de donnée (un iris) est l'espèce à laquelle appartient cette fleur. Cette espèce, en l'occurrence, est appelée l'étiquette du point de donnée.

## À la rencontre des données

Les données que nous allons utiliser pour cet exemple se trouvent dans le jeu de données Iris, qui est un grand classique dans l'apprentissage automatique et les calculs statistiques. Il est inclus dans le module `datasets` de `scikit-learn`. Nous pouvons le charger en appelant la fonction `load_iris` :



Pour en apprendre plus sur l'origine de ces données, voyez sur Wikipédia la page dédiée à l'Iris de Fisher.

**In[9] :**

```
from sklearn.datasets import load_iris
iris_dataset = load_iris()
```

L'objet `iris` qui est retourné par `load_iris` est un objet `Bunch`, qui est très semblable à un dictionnaire. Il contient des clés et des valeurs :

**In[10] :**

```
print(«Keys of iris_dataset: {}».format(iris_dataset.keys()))
```

**Out[10] :**

```
Keys of iris_dataset:
dict_keys(['feature_names', 'target_names', 'target', 'DESCR', 'data'])
```

La valeur de la clé DESCR est une courte description du jeu de données. Voici un aperçu du début de cette description (évidemment, vous pouvez consulter la suite vous-même si vous le souhaitez) :

**In[11] :**

```
print(iris_dataset['DESCR'][:193] + "\n...")
```

**Out[11]:**

```
Iris Plants Database
====================

Notes
-----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive att
...
```

La valeur de la clé `target_names` est un tableau de chaînes de caractères contenant les espèces de fleurs que nous voulons prédire :

**In[12]:**

```
print("Target names: {}".format(iris_dataset['target_names']))
```

**Out[12]:**

```
Target names: ['setosa' 'versicolor' 'virginica']
```

La valeur de `feature_names` est une liste de chaînes qui fournit une description de chaque caractéristique :

**In[13]:**

```
print("Feature names: {}".format(iris_dataset['feature_names']))
```

**Out[13]:**

```
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
  'petal width (cm)']
```

Les données elles-mêmes sont contenues dans les champs `target` et `data`. Ce dernier mémorise les mesures de la longueur et de la largeur du pétale comme du sépale dans un tableau NumPy :

**In[14]:**

```
print("Type of data: {}".format(type(iris_dataset['data'])))
```

**Out[14]:**

```
Type of data: <class 'numpy.ndarray'>
```

Les lignes du tableau `data` correspondent aux fleurs, tandis que les colonnes représentent les quatre mesures réalisées pour chacune des fleurs :

**In[15]:**

```
print("Shape of data: {}".format(iris_dataset['data'].shape))
```

**Out[15]:**

```
Shape of data: (150, 4)
```

Nous pouvons constater que le tableau contient les mesures de 150 fleurs différentes. Souvenez-vous qu'en l'occurrence les éléments individuels sont appelés *échantillons*, et que leurs pro-

priétés sont appelées *caractéristiques* (*features*). La *forme* (*shape*) du tableau `data` est le nombre d'échantillons multiplié par le nombre de caractéristiques. Il s'agit d'une convention de `scikit-learn`, et vos données seront toujours supposées être dans cette forme. Voici les valeurs des caractéristiques pour les cinq premiers exemples :

**In[16]:**

```
print("First five rows of data:\n{}".format(iris_dataset['data'][:5]))
```

**Out[16]:**

```
First five columns of data:
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]]
```

Ce résultat permet de s'apercevoir que les cinq premières fleurs ont un pétale de 0,2 cm de large, et que la première fleur a le plus long sépale, à 5,1 cm.

Le tableau `target` contient les espèces de toutes les fleurs qui ont été mesurées, également sous la forme d'un tableau `NumPy` :

**In[17]:**

```
print("Type of target: {}".format(type(iris_dataset['target'])))
```

**Out[17]:**

```
Type of target: <class 'numpy.ndarray'>
```

Le tableau `target` ne possède qu'une seule dimension, avec une entrée par fleur :

**In[18]:**

```
print("Shape of target: {}".format(iris_dataset['target'].shape))
```

**Out[18]:**

```
Shape of target: (150,)
```

Les espèces sont encodées sous la forme d'un entier compris entre 0 et 2 :

**In[19]:**

```
print("Target:\n{}".format(iris_dataset['target']))
```

**Out[19]:**

```
Target:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

La correspondance est donnée par le tableau `iris['target_names']` : 0 signifie *setosa*, 1 signifie *versicolor* et 2 signifie *virginica*.

## Mesurer la réussite : entraîner et tester les données

Nous voulons partir de ces données pour construire un modèle d'apprentissage automatique capable de prédire l'espèce d'un iris à partir d'un nouveau jeu de mesures. Mais avant de pouvoir appliquer notre modèle à de futures mesures, nous devons savoir s'il fonctionne correctement, autrement dit si nous devrions faire confiance à ses prédictions.

Malheureusement, nous ne pouvons pas nous baser sur les données qui ont été utilisées pour bâtir le modèle afin de l'évaluer ! En effet, notre modèle peut se rappeler tout simplement de la totalité du jeu de données qui nous a servi jusqu'ici. Il est donc toujours capable de fournir la bonne réponse pour n'importe quel point de ce jeu de *données d'apprentissage*, qui est la dénomination que nous utiliserons dans ce livre, quoique les formulations *données d'entraînement* ou encore *données de formation* soient aussi parfaitement correctes. Cela ne permet donc pas de nous informer sur la capacité du modèle à effectuer une bonne *généralisation*, autrement dit à prédire le bon résultat pour de nouvelles données.

Pour évaluer les performances du modèle, nous devons lui proposer de nouvelles données (autrement dit, des données qu'il n'a pas encore rencontrées, mais pour lesquelles nous avons collecté les étiquettes voulues). En général, ce travail est réalisé en partageant les données déjà présentes (ici, les mesures faites sur nos 150 fleurs) en deux parties. L'une de ces parties sert à construire le modèle d'apprentissage automatique. Elle fournit ce que l'on appelle les *données d'apprentissage* (*training data* ou bien *training set*). L'autre partie servira à tester l'efficacité du modèle. On parle alors de *données de test*, de *données d'évaluation* ou encore de *données d'essai* (*test data*, *test set* ou encore *hold-out set*).

`scikit-learn` contient une fonction qui mélange le jeu de données et en effectue la division à votre place : il s'agit de la fonction `train_test_split`. Elle extrait 75 % des lignes disponibles afin de former le jeu de données d'apprentissage, avec les étiquettes correspondantes. Les 25 % qui restent constituent le jeu de données de test, également avec les étiquettes correspondantes. Décider d'une proportion bien adaptée de données à choisir aussi bien pour l'apprentissage que pour l'évaluation est évidemment assez arbitraire, mais un rapport 75/25 constitue une bonne règle de base.

Dans `scikit-learn`, les données sont généralement signalées par une lettre X majuscule, tandis que les étiquettes se distinguent par une lettre y minuscule. Cette façon de procéder est inspirée par la formulation standard $f(x)=y$ en mathématiques, où $x$ désigne un argument passé en entrée, et $y$ la sortie obtenue. Plus précisément, nous utilisons un X majuscule car la donnée est un tableau à deux dimensions (une matrice), tandis qu'y est un tableau à une seule dimension (un vecteur).

Voyons comment utiliser la fonction `train_test_split` sur nos données et comment affecter les résultats en suivant cette nomenclature :

**In[20]:**

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)
```

Avant de diviser en deux le jeu de données, la fonction `train_test_split` le mélange en se servant d'un générateur de nombres pseudo-aléatoires. Si on se contentait de prendre simplement les derniers 25 % du jeu, tous les points de données auraient une étiquette égale à 2 (reportez-vous plus haut à la sortie produite par `iris_dataset['target']`). Cela ne nous renseignerait en rien sur le fonctionnement de notre modèle. En mélangeant les données, nous nous assurons de disposer d'un jeu de test qui contiendra des informations provenant de toutes les classes.

Pour nous assurer que nous obtiendrons le même résultat si nous exécutons plusieurs fois la même fonction, nous fournissons au générateur de nombre pseudo-aléatoire une « graine » fixée à l'aide du paramètre `random_state`. Cela rend la sortie produite totalement déterministe, autrement dit parfaitement reproductible à chaque fois. Dans ce livre, nous nous servirons toujours de cette façon de procéder lorsque nous en aurons besoin.

La sortie produite par la fonction `train_test_split` est un ensemble de tableaux `NumPy` : `X_Train`, `X_test`, `y_train` et `y_test`. `X_train` contient 75 % des lignes du jeu de données, et X_test les 25 % qui restent :

**In[21]:**

```
print("X_train shape: {}".format(X_train.shape))
print("y_train shape: {}".format(y_train.shape))
```

**Out[21]:**

```
X_train shape: (112, 4)
y_train shape: (112,)
```

**In[22]:**

```
print("X_test shape: {}".format(X_test.shape))
print("y_test shape: {}".format(y_test.shape))
```

**Out[22]:**

```
X_test shape: (38, 4)
y_test shape: (38,)
```

## Au commencement de tout : vérifier vos données

Avant de commencer à construire un modèle d'apprentissage automatique, il est souvent utile, et même conseillé, d'inspecter les données afin de voir si la tâche pourrait être facilement résolue

sans avoir besoin d'une telle modélisation, ou même si l'information désirée ne pourrait pas tout simplement être déjà contenue dans ces données.

De plus, inspecter vos données est une bonne façon de s'y prendre pour détecter des éléments anormaux ou certaines particularités. Il se pourrait par exemple que certains de vos iris aient été mesurés en pouces, et non en centimètres. Dans le monde réel, un manque de cohérence dans les données, ou encore des mesures mal faites, sont des choses très courantes.

L'une des meilleures méthodes consiste à les visualiser. Une façon d'obtenir cela est l'utilisation du graphique de type *nuage de points*, ou *diagramme de dispersion* (*scatter plot*). Dans ce genre de tracé, l'axe x représente une certaine caractéristique, et l'axe y une autre. Chaque correspondance est indiquée par un point. Malheureusement, les écrans d'ordinateur n'ont que deux dimensions, ce qui ne nous permet de représenter que deux caractéristiques (trois au plus). Au-delà, la complexité devient rapidement insurmontable pour un être humain normal. Une manière de contourner le problème consiste à utiliser ce que l'on appelle un *pair plot*, qui examine toutes les paires possibles de caractéristiques et produit une matrice dite de dispersion ou de variance/covariance. Si vous avez un nombre réduit de caractéristiques, comme les quatre de notre exemple, cette technique est assez raisonnable. N'oubliez pas cependant que ce type de tracé ne vous montre pas les interactions entre toutes les fonctionnalités, et donc que certains aspects intéressants des données peuvent ne pas être révélés dans ce mode de visualisation.

La Figure 1.3 illustre l'application de cette méthode sur le jeu de données d'apprentissage. Pour créer ce tracé, nous commençons par convertir le tableau `NumPy` en un `DataFrame` `pandas`. Pandas possède une fonction appelée `scatter_matrix` qui permet de produire ce type de tracé. La diagonale de cette matrice est remplie avec les histogrammes de chaque caractéristique :

**In[23]:**

```
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
pd.plotting.scatter_matrix(iris_dataframe, c=y_train, figsize=(15, 15),
  marker='o',hist_kwds={'bins': 20}, s=60,
  alpha=.8, cmap=mglearn.cm3)
```

Sur ce graphique, nous pouvons constater que les trois classes semblent être relativement bien distinguables à partir des mesures effectuées sur les pétales et les sépales. Cela nous indique qu'un modèle d'apprentissage automatique devrait être probablement capable d'arriver à distinguer les différentes espèces de fleurs.

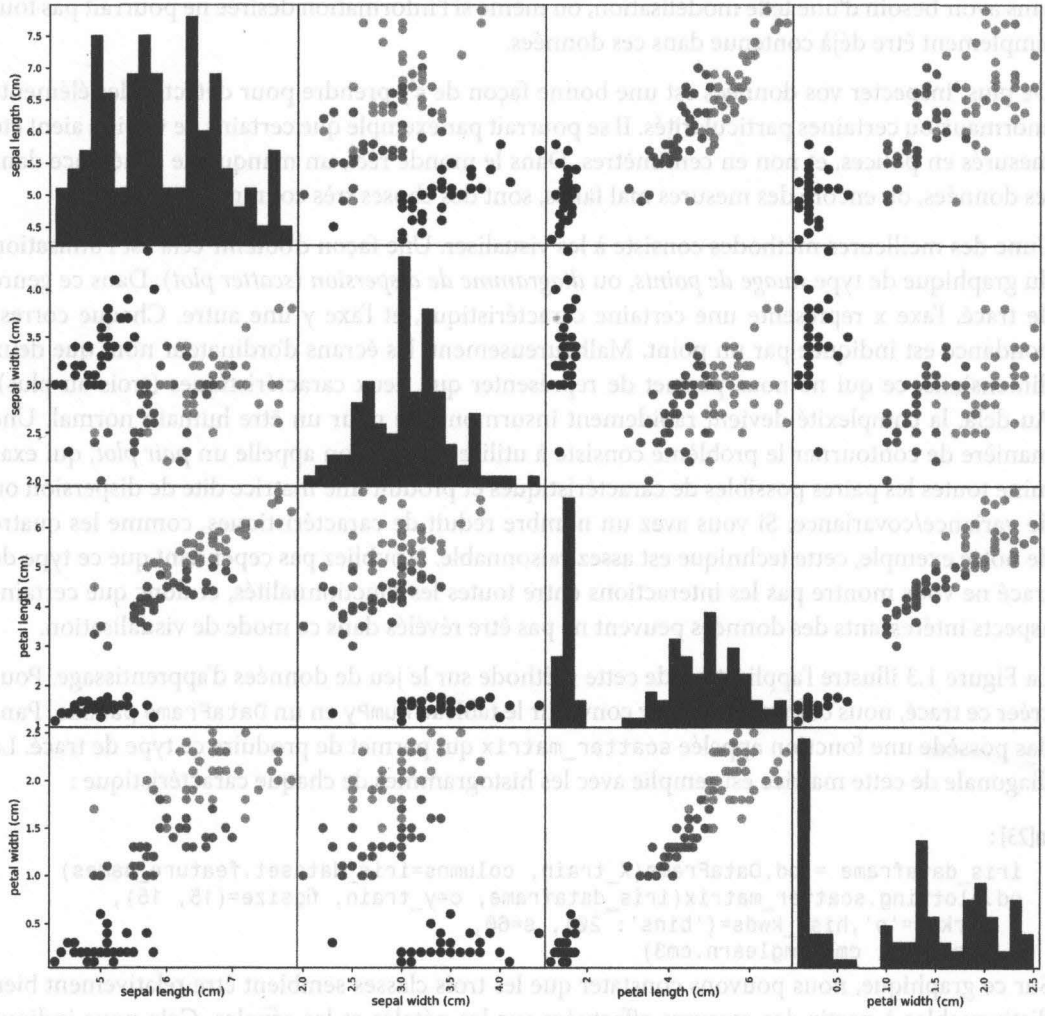



*Figure 1.3 : Exemple d'utilisation de la fonction scatter_matrix à partir du jeu de données Iris.*

## Construire votre premier modèle : les K plus proches voisins

Nous pouvons maintenant commencer à construire un premier modèle d'apprentissage automatique. `scikit-learn` contient de nombreux algorithmes de classification que nous pourrions utiliser. Ici, nous allons nous servir de la méthode dite des *k* plus proches voisins, qui est facile à comprendre. En fait, tout ce que nous avons à faire, c'est d'enregistrer les données du jeu d'apprentissage. Pour faire ensuite une prédiction à partir d'un nouveau point de donnée,

l'algorithme trouve dans le jeu d'apprentissage le point qui est le plus proche du nouveau venu. Il assigne ensuite l'étiquette de cette donnée d'apprentissage au nouveau point de donnée.

Le *k*, dans la formule « *k* plus proches voisins » signifie que, au lieu de nous contenter du voisin le plus proche du nouveau point de donnée, nous pouvons prendre en compte un nombre fixé (*k*) de voisins dans le jeu d'apprentissage (par exemple les trois ou les cinq voisins les plus proches). Partant de là, nous pouvons ensuite faire une prédiction en nous basant sur la classe majoritaire dans ce voisinage. Nous approfondirons ce sujet dans le Chapitre 2. Pour l'instant, nous allons nous limiter à un seul voisin.

Tous les modèles d'apprentissage automatique de `scikit-learn` sont implémentés dans leurs propres classes, qui sont appelées les classes `Estimator` (*estimateur*). L'algorithme de classification des *k* plus proches voisins se trouve dans la classe `KNeighborsClassifier` du module `neighbors`. Avant de pouvoir utiliser ce modèle, nous devons instancier la classe en un objet. C'est là que nous allons définir tous les paramètres du modèle. Le paramètre le plus important de `KNeighborsClassifier` est le nombre de voisins, que nous allons donc ici limiter à un :

**In[24]:**

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```



Ici, `knn` renvoie au nom américain de l'algorithme : *k-nearest neighbors*.

L'objet `knn` encapsule l'algorithme qui va être utilisé pour bâtir le modèle à partir des données d'apprentissage, de même que l'algorithme qui servira à faire des prédictions pour les nouveaux points de données. Il va également contenir les informations que l'algorithme a extraites des données d'apprentissage. Dans le cas de `KNeighborsClassifier`, il s'agira simplement de ces données elles-mêmes.

Pour construire le modèle à partir du jeu de formation, nous appelons la méthode `fit` de l'objet `knn`, qui prend en arguments les tableaux NumPy `X_train` (qui, vous vous en souvenez, contient les données d'apprentissage) et `y_train` (qui contient les étiquettes correspondantes) :

**In[25]:**

```
knn.fit(X_train, y_train)
```

**Out[25]:**

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')
```

La méthode `fit` retourne l'objet `knn` lui-même (tout en le modifiant sur place). Nous obtenons donc une représentation sous forme d'une chaîne de notre classifieur. Cette représentation

nous montre quels paramètres ont été utilisés pour créer le modèle. Pratiquement tous sont les valeurs par défaut, mais vous pouvez aussi retrouver `n_neighbors=1`, qui est le paramètre que nous avons passé. La plupart des modèles de `scikit-learn` possèdent de multiples paramètres, mais la majorité d'entre eux concernent des questions d'optimisation de la vitesse ou des situations très particulières. Il ne sert donc à rien de vous soucier des autres paramètres de cette représentation. Afficher ou imprimer un modèle `scikit-learn` peut générer de très longues chaînes de caractères, mais vous n'avez pas à vous sentir intimidé par ce genre de situation. Les paramètres les plus importants seront étudiés dans le Chapitre 2. Pour tout ce qui suit dans ce livre, nous ne vous présenterons pas la sortie produite par `fit`, car elle ne contient aucune information supplémentaire utile.

## Faire des prédictions

À partir de notre modèle, nous pouvons maintenant faire des prédictions portant sur de nouvelles données pour lesquelles nous ne connaissons pas encore les bonnes étiquettes. Supposons par exemple que nous trouvions dans la nature un iris pour lequel nous mesurons un sépale de 5 cm de long et de 2,9 cm de large, ainsi qu'une longueur de pétale de 1 cm avec une largeur de 0,2 cm. À quelle espèce cette fleur appartient-elle ? Nous pouvons bien entendu placer cette donnée dans un tableau `NumPy`, en calculant à nouveau sa forme, autrement dit le nombre d'échantillons (1) multiplié par le nombre de caractéristiques (4) :

**In[26]:**

```
X_new = np.array([[5, 2.9, 1, 0.2]])
print("X_new.shape: {}".format(X_new.shape))
```

**Out[26]:**

```
X_new.shape: (1, 4)
```

Remarquez que nous avons placé les mesures de cette nouvelle fleur dans une ligne d'un tableau `NumPy` à deux dimensions, ce qui est ce qu'attend `scikit-learn` pour les données.

Pour effectuer une prédiction, nous allons maintenant appeler la méthode `predict` de l'objet `knn` :

**In[27]:**

```
prediction = knn.predict(X_new)
print("Prediction: {}".format(prediction))
print("Predicted target name: {}".format(
       iris_dataset['target_names'][prediction]))
```

**Out[27]:**

```
Prediction: [0]
Predicted target name: ['setosa']
```

Notre modèle prédit que ce nouvel iris appartient à la classe 0, autrement dit à l'espèce *setosa*. Mais comment savoir si nous devons croire ou non les prédictions faites par notre modèle ? Pour cet exemple, nous ne savons rien de l'espèce effective de la fleur, ce qui est tout de même essentiel pour notre propos !

## Évaluer le modèle

C'est là que le jeu de test que nous avons créé plus haut entre en scène. Ces données n'ont pas été utilisées pour créer le modèle, mais nous savons qu'elles contiennent des valeurs correctes pour chaque espèce d'iris.

Nous pouvons donc prédire l'espèce de chaque iris du jeu de test et la comparer avec l'étiquette normalement correspondante (l'espèce connue). Nous pouvons ainsi mesurer la qualité du modèle en calculant son *exactitude*, autrement dit la proportion de fleurs pour laquelle la prédiction de l'espèce s'est révélée juste :

**In[28]:**

```
y_pred = knn.predict(X_test)
print("Test set predictions:\n {}".format(y_pred))
```

**Out[28]:**

```
Test set predictions:
 [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
 2]
```

**In[29]:**

```
print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))
```

**Out[29]:**

```
Test set score: 0.97
```

Il est aussi possible de faire appel à la méthode `score` de l'objet `knn` afin de calculer directement ce niveau d'exactitude de la prédiction sur le jeu de test :

**In[30]:**

```
print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

**Out[30]:**

```
Test set score: 0.97
```

Pour ce premier modèle, l'exactitude est d'environ 0,97, ce qui signifie que la prédiction a été bonne pour 97 % des iris du jeu de test. Sans entrer dans les détails mathématiques, cela indique que nous pouvons nous attendre à ce que notre modèle fournisse 97 % de bonnes réponses pour de nouveaux iris. Si nous en revenons à notre botaniste, ce haut niveau d'exactitude implique que le modèle est crédible et intéressant à utiliser. Dans d'autres chapitres, nous aborderons la question de l'amélioration des performances, ainsi que des considérations soulevées par l'ajustement d'un modèle.

## Résumé et perspectives

Résumons ce que nous avons appris dans ce chapitre. Nous avons commencé par une brève introduction à l'apprentissage automatique (si vous avez l'habitude de parler de *machine learning*, ce n'est pas un souci) et à ses applications. Nous avons ensuite discuté de la distinction entre apprentissage supervisé et non supervisé, et proposé un aperçu des outils que nous allons utiliser dans ce livre. Puis nous nous sommes intéressés à cette passionnante question qui consiste à prédire à quelle espèce un certain iris peut appartenir, connaissant certaines mesures physiques de la fleur. Nous avons pour cela utilisé un jeu de données formé de mesures qui ont été validées par un expert, ce qui nous a indiqué qu'il s'agissait d'une tâche d'apprentissage supervisé. En l'occurrence, nous étions face à un problème de classification comportant trois classes, soit autant d'espèces d'iris rencontrées (*setosa*, *versicolor* et *virginica*). Les espèces possibles sont appelées des *classes*, et l'espèce à laquelle appartient un certain iris est l'*étiquette* de celui-ci.

Le jeu de données Iris est constitué de deux tableaux NumPy : l'un contient les données, et est référencé sous la forme `X` dans `scikit-learn`, l'autre, contenant les sorties correctes ou désirées, est appelé `y` également dans `scikit-learn`. `X` est un tableau de caractéristiques à deux dimensions, soit une ligne par point de donnée, et une colonne par caractéristique. Le tableau `y` possède une seule dimension, en l'occurrence une étiquette de classe qui est un entier entre 0 et 2 associé à chaque échantillon.

Nous avons partagé notre série d'informations en deux parties : un jeu de *données d'apprentissage*, pour construire notre modèle, et un jeu de *données de test* pour juger de ses capacités de *généralisation*, autrement dit des performances du modèle face à de toutes nouvelles données.

Nous avons choisi l'algorithme des *k* plus proches voisins, qui réalise des prédictions pour un nouveau point de donnée en considérant son ou ses voisins les plus proches dans le jeu d'apprentissage. Cette méthode est implémentée dans la classe `KNeighborsClassifier`, qui contient l'algorithme servant à construire le modèle, de même que celui qui s'en sert pour effectuer une prédiction. Nous avons instancié la classe et défini les paramètres dont nous avions besoin. Nous avons ensuite produit le modèle en appelant la méthode `fit`, et en lui passant en paramètres les données d'apprentissage (`X_train`) ainsi que les sorties correspondantes (`y_train`). Après quoi, nous avons évalué l'exactitude du modèle en utilisant la méthode `score`. Nous avons appliqué cette méthode au jeu de données de test et aux étiquettes correspondantes, pour obtenir comme résultat une exactitude plutôt remarquable de l'ordre de 97 %.

Cela nous procure un très bon niveau de confiance pour appliquer le modèle à de nouvelles données (ici, de nouvelles mesures effectuées sur des iris), puisque nous pouvons estimer que le modèle fournira la bonne réponse environ 97 fois sur 100.

Voici un résumé du code nécessaire à la mise en œuvre des procédures d'entraînement et d'évaluation :

**In[31]:**

```
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)

print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

**Out[31]:**

```
Test set score: 0.97
```

Ce fragment contient la base du code permettant d'appliquer un algorithme d'apprentissage automatique avec `scikit-learn`. Les méthodes `fit`, `predict` et `score` constituent l'interface commune aux modèles supervisés dans `scikit-learn`. À partir des concepts présentés dans ce chapitre, vous pouvez appliquer ces modèles à de multiples tâches d'apprentissage automatique. Dans le prochain chapitre, nous allons approfondir ces notions afin d'étudier de plus près les différents modèles supervisés de `scikit-learn` et apprendre comment les appliquer avec succès.

CHAPITRE 2

# Apprentissage supervisé

L'apprentissage supervisé est l'un des types les plus courants et les plus efficaces d'apprentissage automatique. Après l'avoir découvert dans le Chapitre 1, nous allons ici l'approfondir et expliquer plusieurs algorithmes populaires d'apprentissage supervisé. Nous avons déjà vu une application d'apprentissage automatique supervisé dans le Chapitre 1 : la classification d'iris en plusieurs espèces en utilisant un certain nombre de mesures effectuées sur les fleurs.

Souvenez-vous que l'apprentissage supervisé est utilisé lorsque nous voulons prédire une certaine sortie à partir d'une entrée connue et que nous connaissons déjà des paires entrée/sortie. Nous construisons un modèle d'apprentissage automatique à partir de ces paires entrée/sortie, qui constituent notre *jeu d'apprentissage*. Notre but est alors de faire des prévisions précises pour des données nouvelles, qui n'ont pas encore été rencontrées. L'apprentissage supervisé nécessite souvent un effort humain pour bâtir le jeu d'apprentissage, mais, par la suite, il automatise et souvent accélère une tâche qui serait sinon très laborieuse, sinon infaisable.

## Classification et régression

Il existe deux types majeurs de problèmes dans l'apprentissage automatique supervisé appelés *classification* et *régression*.

Dans le cas de la classification, le but est de prédire une *étiquette de classe*, qui est un choix possible parmi une liste prédéfinie de possibilités. Dans le Chapitre 1, nous avons utilisé l'exemple de classification d'iris dans l'une de trois espèces possibles. La classification est parfois scindée entre *classification binaire*, dans laquelle le problème comporte exactement deux classes, et *classification multiclasse*, qui est une classification concernant plus de deux classes. Vous pouvez vous figurer la classification binaire comme une tentative de répondre à une question par oui ou par non. Par exemple, classer une série de courriers électroniques comme étant ou non des

spams est un problème de classification binaire. En l'occurrence, la question à laquelle on doit répondre par oui ou par non serait : « Cet email est-il un spam ? ».



Dans une classification binaire, on dit souvent qu'une des classes est la *classe positive*, et que l'autre est la *classe négative*. Ici, le terme *positif* ne signifie pas avoir une certaine valeur ou représenter un certain bénéfice, mais sert en fait à désigner l'objet de l'étude. Par exemple, lors d'une recherche de spams, le terme *positif* pourrait s'appliquer à la classe spam, et donc *négatif* aux courriers « normaux ». Choisir laquelle des deux classes est dite positive est donc le plus souvent quelque chose de subjectif, et spécifique au domaine concerné.

Nos iris, de leur côté, représentent un problème de classification *multiclasse*. Un autre exemple pourrait consister à prédire le langage utilisé sur un site Web à partir du texte qu'il contient. Dans ce cas, les classes seraient une liste prédéfinie de langues possibles.

Dans le cas des tâches dites de régression, le but est de prédire une valeur continue, c'est-à-dire un *nombre en virgule flottante* en termes de programmation, ou un *nombre réel* du point de vue mathématique. Prédire un salaire annuel en fonction de l'éducation, de l'âge, du sexe et du lieu de vie est un exemple de tâche de régression. Dans ce cas, la valeur prédite est un montant, une quantité, qui peut être un nombre quelconque à l'intérieur d'un intervalle donné. Un autre exemple de régression pourrait consister à prédire la production d'une exploitation céréalière à partir d'attributs comme la production des années précédentes, la météo, le nombre de personnes travaillant à la ferme, *etc.* Là encore, cette production peut être un nombre quelconque.

Une manière de faire la distinction entre tâches de classification et tâches de régression consiste à se demander s'il y a une certaine forme de continuité dans la sortie. Si c'est le cas, alors il s'agit d'un problème de régression. Reprenons l'exemple de la prédiction d'un salaire annuel. Il est clair qu'il existe une continuité dans la sortie. Qu'une personne gagne 30 000 ou 30 001 € ne constitue évidemment pas une différence tangible, même s'il s'agit de montants différents. Et si notre algorithme prédit 29 999 € ou 30 001 € alors qu'il aurait dû trouver 30 000 €, cela ne va pas non plus nous déranger.

Par contraste, dans le cas de la reconnaissance de la langue d'un site Web (qui est un problème de classification), il n'y a pas de continuité. Un site Web est écrit dans une certaine langue, *ou* dans une autre. Il n'y a pas de langage intermédiaire *entre* l'anglais et le français (les linguistes nous excuseront pour cette présentation simplifiée des langues).

## Généralisation, surapprentissage et sous-apprentissage

Dans l'apprentissage supervisé, nous voulons construire un modèle à partir d'un jeu de données d'apprentissage, puis être capables de faire des prédictions précises sur de nouvelles données, qui n'ont jamais été vues auparavant, mais qui possèdent les mêmes caractéristiques que le jeu de départ. Dans ce cas, nous disons que le modèle est capable de *généralisation* des données d'ap-

prentissage vers les données de test. Autrement dit, nous voulons donc construire un modèle qui soit capable d'effectuer une généralisation avec l'exactitude la plus élevée possible.

En général, nous construisons un modèle de manière à ce qu'il puisse faire des prédictions précises à partir du jeu d'apprentissage. Si le jeu d'apprentissage et le jeu de test ont suffisamment de caractéristiques en commun, nous attendons du modèle qu'il soit aussi exact avec les données de test. Cependant, dans certains cas, les choses peuvent ne pas se présenter aussi bien. Supposons par exemple que nous construisions des modèles très complexes. Il est toujours possible d'être aussi exact que nous le souhaitons sur le jeu d'*apprentissage*.

Regardons un exemple fictif pour illustrer ce point. Disons qu'un *data scientist* novice voudrait prédire si un client va acheter un bateau. Il dispose pour cela de données concernant de précédents acheteurs ainsi que des clients dont on sait qu'ils ne sont pas intéressés par ce type d'acquisition. Bien entendu, dans le monde réel, les choses seraient plus compliquées, car une certaine personne pourrait avoir finalement acheté son bateau chez un autre vendeur, ou bien encore être en train d'économiser pour réaliser cet achat dans le futur. Mais contentons-nous ici d'une vision plus simple. Disons que l'objectif consiste à envoyer des emails promotionnels à toutes les personnes qui seraient susceptibles de réaliser un achat, mais pas aux autres.

Nous allons nous baser sur les données présentées dans le Tableau 2.1.

*Tableau 2.1 : Exemples de données concernant des clients.*

| Âge | Nombre de voitures | Propriétaire | Nombre d'enfants | Statut marital | Possède un chien | A déjà acheté un bateau |
|---|---|---|---|---|---|---|
| 66 | 1 | oui | 2 | veuf/veuve | non | oui |
| 52 | 2 | oui | 3 | marié(e) | non | oui |
| 22 | 0 | non | 0 | marié(e) | oui | non |
| 25 | 1 | non | 1 | célibataire | non | non |
| 44 | 0 | non | 2 | divorcé(e) | oui | non |
| 39 | 1 | oui | 2 | marié(e) | oui | non |
| 26 | 1 | non | 2 | célibataire | non | non |
| 40 | 3 | oui | 1 | marié(e) | oui | non |
| 53 | 2 | oui | 2 | divorcé(e) | non | oui |
| 64 | 2 | oui | 3 | divorcé(e) | non | non |
| 58 | 2 | oui | 2 | marié(e) | oui | oui |
| 33 | 1 | non | 1 | célibataire | non | non |

Après avoir étudié pendant un certain temps ces données, notre apprenti *data scientist* en déduit la règle suivante : « Si le/la client(e) a plus de 45 ans, a moins de 3 enfants ou n'est pas divorcé(e), alors il/elle veut acheter un bateau ». Quand on lui demande si cette règle fournit les bons résultats, notre novice répond avec assurance : « Elle est exacte à 100 % ! ». Et c'est vrai. En partant

des données qui se trouvent dans le tableau, cette règle est parfaitement exacte. En fait, il y a bien d'autres règles possibles que nous pourrions extraire de ce tableau et qui expliqueraient tout aussi parfaitement la raison pour laquelle une personne qui y est enregistrée voudrait acheter un bateau. Par exemple, aucun âge n'apparaît deux fois dans les données. Nous pourrions alors en déduire que les personnes qui ont 66, 52, 53 ou 58 ans veulent acheter un bateau, et pas les autres. Et ainsi de suite… Mais l'important, c'est surtout de se rappeler que nous ne voulons pas faire des prédictions pour *ce* jeu de données. Nous connaissons déjà les réponses pour ces clients-là. Ce que nous voulons savoir, c'est si de *nouveaux clients* sont susceptibles d'acheter un bateau. Nous avons donc besoin de trouver une règle qui fonctionne bien pour les nouveaux clients, et atteindre une exactitude de 100 % sur le jeu d'apprentissage ne nous aidera ici en rien. La règle trouvée par notre apprenti *data scientist* sera plus que vraisemblablement fausse pour une série de données totalement nouvelles, car elle semble trop complexe et ne s'appuie que sur très peu d'éléments. Par exemple, la partie de la règle qui dit « ou n'est pas divorcé(e) » s'appuie sur un(e) seul(e) client(e).

La seule manière de savoir si un algorithme va bien s'appliquer sur de nouvelles données consiste à l'évaluer sur le jeu de test. Cependant, nous nous attendons intuitivement à ce que des modèles simples se généralisent mieux avec de nouvelles données. Si la règle était « Les personnes de plus de 50 ans veulent acheter un bateau » et que cela explique le comportement de tous les clients, nous la croirions bien plus que celle qui met en avant les enfants et le statut familial en plus de l'âge. En d'autres termes, nous voulons toujours rechercher le modèle le plus simple possible. Construire un modèle qui est trop complexe pour la quantité d'informations dont nous disposons, comme l'a fait notre novice, s'appelle du *surapprentissage* (*overfitting*, ou encore *surajustement*). Le surapprentissage se produit lorsque votre modèle suit trop précisément les particularités du jeu d'apprentissage. Ce modèle fonctionne alors bien avec celui-ci, mais il n'est pas capable de se généraliser correctement à de nouvelles données. De l'autre côté, si notre modèle est trop simple, comme dans « Toute personne qui possède une maison achète un bateau », alors vous risquez fort de ne pas capturer tous les aspects de vos données ainsi que leur variabilité, et votre modèle sera probablement mauvais, y compris sur le jeu d'apprentissage. Choisir un modèle trop simple est appelé *sous-apprentissage* (*underfitting*, ou encore *sous-ajustement*).

Plus nous permettons à notre modèle d'être complexe, et mieux nous serons capables d'effectuer des prédictions sur le jeu d'apprentissage. En revanche, s'il devient trop complexe, nous allons commencer à trop nous concentrer sur chaque point de donnée individuel, et le modèle ne se généralisera pas bien pour de nouvelles données.

Il existe un « point idéal » entre les deux qui permet de produire les meilleures performances de généralisation. C'est le modèle que nous voulons trouver. Le compromis entre surapprentissage et sous-apprentissage est illustré sur la Figure 2.1.

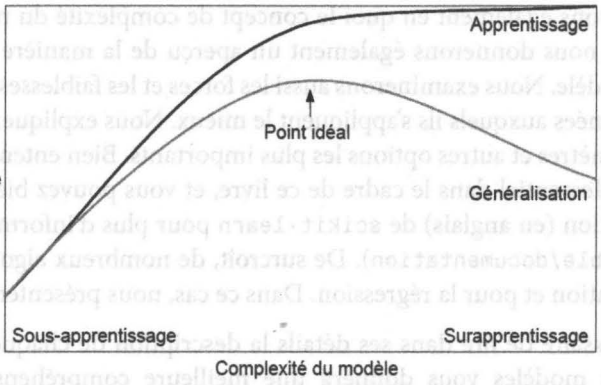



*Figure 2.1 : Compromis à trouver quant à la complexité du modèle entre données d'apprentissage et exactitude pour les données de test.*

### Relation entre complexité du modèle et taille du jeu de données

Il est important de noter que la complexité d'un modèle est intimement liée à la variation des éléments contenus dans votre jeu de données d'apprentissage. Plus ces éléments sont variés, et plus complexe sera le modèle que vous pourrez utiliser sans entraîner de surapprentissage. En règle générale, collecter davantage d'échantillons produit aussi davantage de variété, et donc les jeux de données les plus grands permettent de construire des modèles plus complexes. Par contre, dupliquer simplement les points de données ou collecter des échantillons très semblables n'aide en rien.

Revenons à notre exemple de bateaux. Si nous avions vu 10 000 autres lignes de données utilisateur, toutes étant en accord avec la règle « Si le/la client(e) a plus de 45 ans, a moins de 3 enfants ou n'est pas divorcé(e), alors il/elle veut acheter un bateau », nous serions bien plus tentés de croire qu'il s'agit d'une bonne règle qu'avec simplement les 12 lignes du Tableau 2.1.

Le fait d'avoir plus de données et de pouvoir ainsi construire des modèles plus complexes peut souvent faire des merveilles dans le cas de tâches d'apprentissage supervisées. Dans ce livre, nous allons essentiellement travailler avec des jeux de données ayant une taille fixée. Dans le mode réel, vous avez souvent la possibilité de décider de la quantité de données à collecter, ce qui peut être plus bénéfique que de triturer votre modèle dans tous les sens pour essayer de l'améliorer. Ne sous-estimez jamais la puissance apportée par l'augmentation du volume des données.

## Algorithmes pour l'apprentissage automatique supervisé

Nous allons maintenant passer en revue les algorithmes d'apprentissage supervisé les plus populaires, et expliquer comment ils apprennent des données ainsi que comment ils font des prédic-

tions. Nous verrons également en quoi le concept de complexité du modèle joue pour chacun des modèles, et nous donnerons également un aperçu de la manière dont chaque algorithme construit un modèle. Nous examinerons aussi les forces et les faiblesses de chaque algorithme, et les types de données auxquels ils s'appliquent le mieux. Nous expliquerons également la signification des paramètres et autres options les plus importants. Bien entendu, nous ne pouvons que nous en tenir à l'essentiel dans le cadre de ce livre, et vous pouvez bien entendu vous reporter à la documentation (en anglais) de `scikit-learn` pour plus d'informations (`http://scikit-learn.org/stable/documentation`). De surcroît, de nombreux algorithmes ont une variante pour la classification et pour la régression. Dans ce cas, nous présenterons les deux.

Il n'est pas nécessaire de lire dans ses détails la description de chaque algorithme. Par contre, comprendre les modèles vous donnera une meilleure compréhension quant aux diverses manières dont peuvent fonctionner les algorithmes d'apprentissage automatique. Ce chapitre peut aussi servir de guide de référence, et vous pourrez y revenir si vous avez des doutes sur le fonctionnement de tel ou tel algorithme.

## Exemples de jeux de données

Nous utiliserons plusieurs jeux de données pour illustrer les différents algorithmes. Certains d'entre eux seront de taille réduite et synthétiques (du sur mesure en quelque sorte) afin de mettre en évidence certains aspects des algorithmes. D'autres seront repris du monde réel et d'une taille bien plus importante.

Un exemple de jeu de données synthétique pour une classification à deux classes est fourni par `forge`, qui possède deux caractéristiques. Le code qui suit crée un nuage de points permettant de visualiser tous les points de ce jeu. Ce graphique montre la première caractéristique sur l'axe x, et la seconde sur l'axe y. Comme c'est évidemment toujours le cas pour ce type de graphique, chaque donnée est représentée par un point (au sens large du terme…). La forme de ce point indique à quelle classe il appartient :

**In[1]:**

```
X, y = mglearn.datasets.make_forge()
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Class 0", "Class 1"], loc=4)
plt.xlabel("First feature")
plt.ylabel("Second feature")
print("X.shape: {}".format(X.shape))
```

**Out[1]:**

```
X.shape: (26, 2)
```

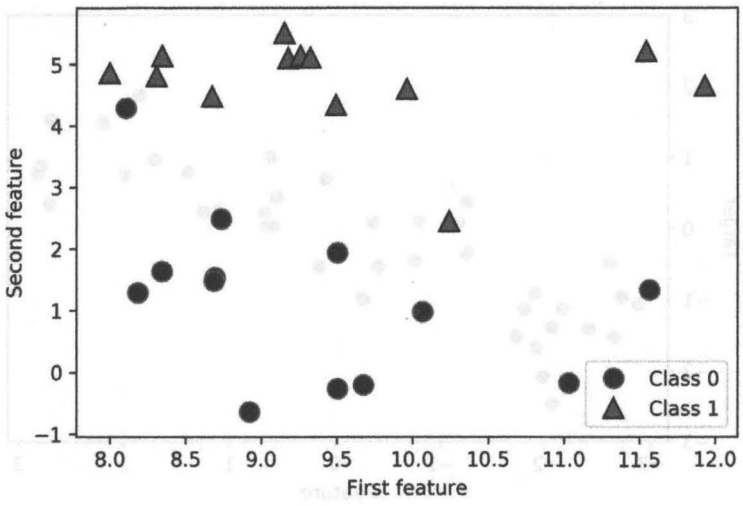



*Figure 2.2 : Nuage de points pour le jeu de données* `forge`.

Comme vous pouvez le vérifier sur la figure, ce jeu de données contient 26 points répartis en deux catégories.

Pour illustrer les algorithmes de régression, nous utiliserons un jeu de données appelé wave. Ce jeu de données a une seule caractéristique en entrée, et une variable cible continue (ou *réponse*, ou encore *réaction*) que nous voulons modéliser. Le tracé illustré sur la Figure 2.3 montre la caractéristique du jeu de données sur l'axe x et la sortie (la variable cible de la régression) sur l'axe y :

**In[2]:**

```
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Feature")
plt.ylabel("Target")
```

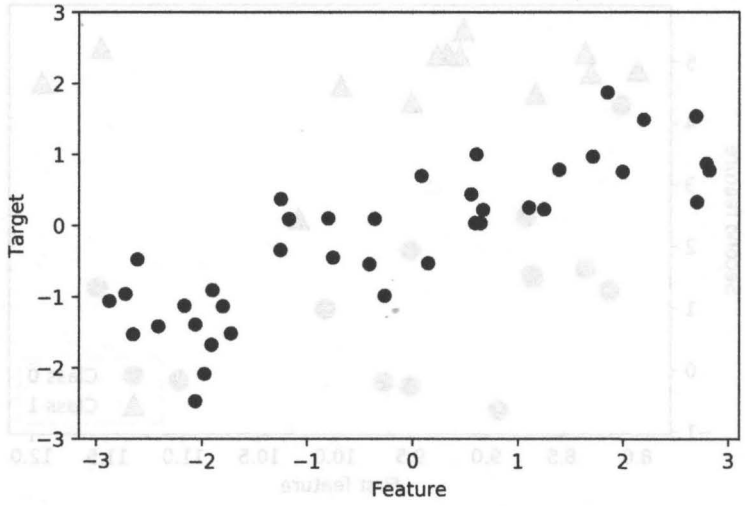



*Figure 2.3 : Tracé du jeu de données* wave, *l'axe x montrant la caractéristique et l'axe y la cible de la régression.*

Nous utilisons ces exemples très simples, de taille réduite, car il est ainsi plus facile de les visualiser. Une page imprimée n'a que deux dimensions, et donc des données possédant plus de deux caractéristiques sont difficiles à montrer... Des intuitions développées à partir de jeux de données de faible dimension (avec peu de caractéristiques) pourraient ne pas valoir pour des jeux de données de grande dimension (avec beaucoup de caractéristiques). Tant que vous gardez cela présent à l'esprit, inspecter des algorithmes sur des jeux de données de faible dimension peut être très instructif.

Nous compléterons ces petits exemples de données synthétiques par deux jeux de données provenant du monde réel et qui sont inclus dans `scikit-learn`. L'un (`cancer`) provient d'une base de données établie dans l'état américain du Wisconsin concernant des enregistrements cliniques de mesures de tumeurs dans le cas de cancers du sein. Chaque tumeur est étiquetée comme bénigne (*benign*, pour des tumeurs inoffensives) ou maligne (*malignant*, pour des tumeurs cancéreuses). La tâche consiste à apprendre à prédire si une tumeur est maligne en se basant sur des mesures des tissus.

Ces données peuvent être chargées en appelant la fonction `load_breast_cancer` de `scikit-learn` :

**In[3] :**

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("cancer.keys(): {}".format(cancer.keys()))
```

**Out[3] :**

```
cancer.keys():
```

```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names'])
```



Les jeux de données qui sont inclus dans `scikit-learn` sont généralement enregistrés sous la forme d'objets `Bunch`, qui contiennent certaines informations aussi bien sur le jeu de données que sur les données réelles. Tout ce que vous avez besoin de savoir sur les objets `Bunch`, c'est qu'ils se comportent comme des dictionnaires, avec l'avantage supplémentaire que vous pouvez accéder aux valeurs en utilisant un point de séparation, comme dans `bunch.key` au lieu de `bunch['key']`.

Le jeu de données comprend 569 échantillons, chacun étant formé de 30 caractéristiques :

**In[4]:**
```
print("Shape of cancer data: {}".format(cancer.data.shape))
```
**Out[4]:**
```
Shape of cancer data: (569, 30)
```

Sur ces 569 points de données, 212 sont étiquetés comme étant malins, 357 comme étant bénins :

**In[5]:**
```
print("Sample counts per class:\n{}".format({
    n: v for n, v in zip(cancer.target_names, np.bincount(cancer.target))}))
```
**Out[5]:**
```
Sample counts per class:
{'malignant': 212, 'benign': 357}
```

Pour obtenir une description de la signification sémantique de chaque caractéristique, nous pouvons jeter un coup d'œil sur l'attribut `feature_names` :

**In[6]:**
```
print("Feature names:\n{}".format(cancer.feature_names))
```
**Out[6]:**
```
Feature names:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Vous pouvez en apprendre plus sur les données en lisant le contenu du fichier `cancer.DESCR` si cela vous intéresse.

Nous utiliserons aussi un jeu de données de régression reprises du monde réel, appelé Boston Housing. La tâche associée à ce jeu de données consiste à prédire la valeur moyenne de maisons

dans plusieurs quartiers de la ville de Boston dans les années 1970 en utilisant des informations comme le taux de criminalité, la proximité avec la rivière Charles, l'accessibilité par autoroute et ainsi de suite. Ce jeu de données est formé de 506 points décrits par 13 caractéristiques :

**In[7]:**

```
from sklearn.datasets import load_boston
boston = load_boston()
print("Data shape: {}".format(boston.data.shape))
```

**Out[7]:**

```
Data shape: (506, 13)
```

Là encore, vous pouvez obtenir davantage d'informations sur le jeu de données en lisant l'attribut DESCR de `boston`. Pour notre propos, nous élargirons notre champ de vision sur ce jeu de données non seulement en prenant ces 13 caractéristiques comme valeurs d'entrée, mais aussi en regardant toutes les *interactions* entre ces caractéristiques. En d'autres termes, nous n'allons pas seulement considérer des caractéristiques comme le taux de criminalité et l'accès par autoroute, mais aussi le croisement, ou l'interaction, entre criminalité et autoroute. C'est ce que l'on appelle *feature engineering* (*ingénierie des caractéristiques*). Nous y reviendrons dans le Chapitre 4. Ce jeu de données dérivé peut être chargé en utilisant la fonction `load_extended_boston` :

**In[8]:**

```
X, y = mglearn.datasets.load_extended_boston()
print("X.shape: {}".format(X.shape))
```

**Out[8]:**

```
X.shape: (506, 104)
```

Le résultat indique 104 caractéristiques, qui sont la combinaison entre les 13 caractéristiques d'origine et les 91 combinaisons de deux caractéristiques prises parmi ces 13 possibilités (il y a 13 interactions pour la première caractéristique, plus 12 pour la seconde après élimination de la première, plus 11 pour la suivante et ainsi de suite jusqu'à 1).

Nous utiliserons ces jeux de données pour expliquer et illustrer les propriétés des différents algorithmes d'apprentissage automatique. Pour l'instant, occupons-nous des algorithmes eux-mêmes. Nous commencerons par revisiter l'algorithme des *k* plus proches voisins déjà rencontré dans le premier chapitre.

## Les *k* plus proches voisins

L'algorithme des *k* plus proches voisins (ou *k*-NN pour *k Nearest Neighbors*) est sans doute le plus simple en apprentissage automatique. Construire le modèle consiste simplement à enregistrer le jeu de données d'apprentissage. Pour effectuer ensuite une prédiction sur un nouveau point de donnée, l'algorithme trouve les éléments qui s'en rapprochent le plus dans le jeu d'apprentissage, donc ses « plus proches voisins ».

### Classification *k*-NN

Dans sa version la plus simple, l'algorithme *k*-NN ne prend en considération qu'un seul plus proche voisin, qui est donc le point de donnée le plus proche dans le jeu d'apprentissage de la nouvelle donnée pour laquelle nous voulons faire une prédiction. Cette prédiction est alors simplement la sortie connue pour ce point du jeu de donnée. La Figure 2.4 illustre cela en prenant dans le cas d'une classification sur le jeu de données `forge` :

**In[9]:**

```
mglearn.plots.plot_knn_classification(n_neighbors=1)
```

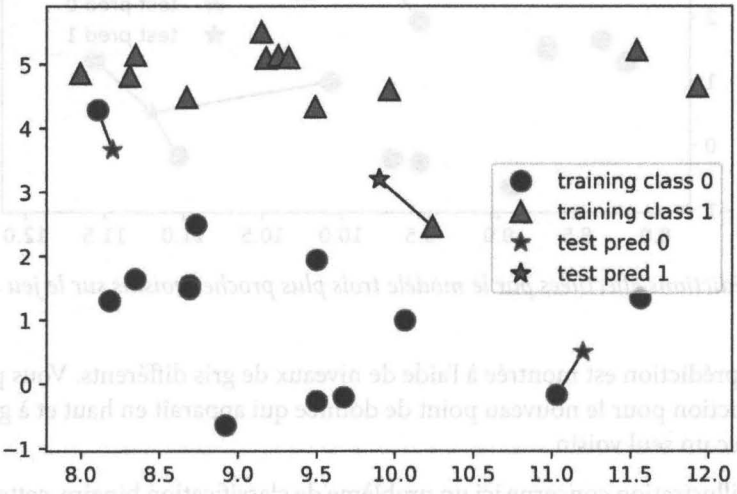



*Figure 2.4 : Prédictions effectuées par le modèle un plus proche voisin sur le jeu de données* `forge`.

Ici, nous avons ajouté trois points de données, qui sont montrés sous forme d'étoiles. Pour chacun d'entre eux, nous avons marqué le point le plus proche dans le jeu d'apprentissage. La prédiction pour cet algorithme est l'étiquette de ce point (représentée par une teinte de gris plus ou moins prononcée).

Au lieu de nous intéresser uniquement au voisin le plus proche, nous pouvons également considérer un nombre arbitraire de voisins, disons *k* (d'où, en passant, le nom donné à cet algorithme). Lorsque nous prenons en compte plus d'un voisin, nous utilisons un *vote* pour affecter une étiquette. Cela signifie que, pour chaque point du jeu de test, nous comptons le nombre de voisins qui appartiennent à la classe 0, et le nombre de ceux qui appartiennent à la classe 1. Nous affectons alors la classe qui est la plus fréquente, soit en d'autres termes la classe majoritaire parmi les *k* plus proches voisins. Dans l'exemple de la Figure 2.5, nous utilisons les trois plus proches voisins :

**In[10] :**

```
mglearn.plots.plot_knn_classification(n_neighbors=3)
```

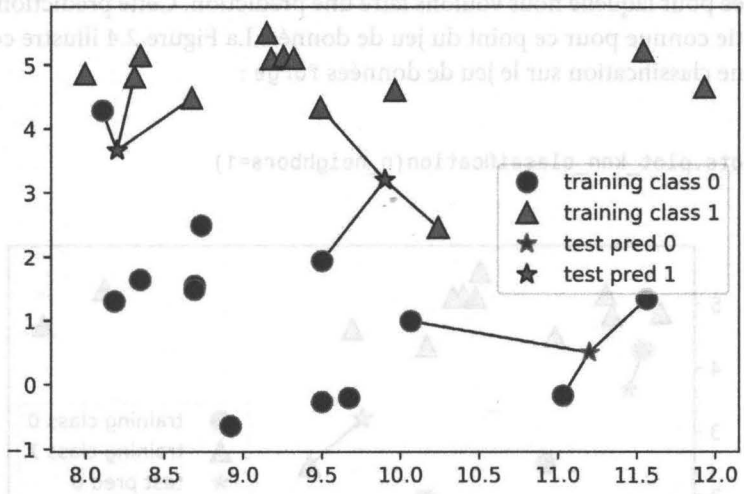



*Figure 2.5 : Prédictions effectuées par le modèle trois plus proches voisins sur le jeu de données* `forge`.

À nouveau, la prédiction est montrée à l'aide de niveaux de gris différents. Vous pouvez constater que la prédiction pour le nouveau point de donnée qui apparaît en haut et à gauche n'est pas la même qu'avec un seul voisin.

Bien que cette illustration concerne ici un problème de classification binaire, cette méthode peut être appliquée à un nombre quelconque de classes. Lorsque ce nombre augmente, nous comptons simplement le nombre de voisins qui appartiennent à chaque classe et nous prédisons la classe alors la plus représentée.

Voyons maintenant comment appliquer l'algorithme des $k$ plus proches voisins en utilisant `scikit-learn`. Tout d'abord, nous allons partager nos données en un jeu d'apprentissage et un jeu de test de manière à pouvoir évaluer les performances de généralisation, comme nous l'avons vu dans le Chapitre 1 :

**In[11] :**

```
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Ensuite, nous importons et instancions la classe. Nous pouvons alors définir des paramètres, comme le nombre de voisins (3 dans cet exemple) :

In[12]:

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

Nous adaptons maintenant le classifieur en utilisant le jeu d'apprentissage. Dans le cas de `KNeighborsClassifier`, cela signifie enregistrer le jeu de données de manière à pouvoir calculer les voisins au cours de la prédiction :

In[13]:

```
clf.fit(X_train, y_train)
```

Pour faire des prédictions sur les données de test, nous appelons la méthode `predict`. Pour chaque point de donnée du jeu de test, elle calcule ses plus proches voisins dans le jeu d'apprentissage et trouve parmi ceux-ci la classe la mieux représentée :

In[14]:

```
print("Test set predictions: {}".format(clf.predict(X_test)))
```

Out[14]:

```
Test set predictions: [1 0 1 0 1 0 0]
```

Pour vérifier si notre modèle se généralise bien, nous pouvons appeler la méthode `score` avec les données de test ainsi que les étiquettes de test :

In[15]:

```
print("Test set accuracy: {:.2f}".format(clf.score(X_test, y_test)))
```

Out[15]:

```
Test set accuracy: 0.86
```

Nous voyons que l'exactitude de notre modèle est évaluée à 86 %, ce qui signifie que le modèle a correctement prédit la classe pour 86 % des échantillons du jeu de données de test.

### Analyser KNeighborsClassifier

Pour les jeux de données en deux dimensions, nous pouvons également illustrer les prédictions pour tous les points de test possibles dans un plan x-y. Nous donnons une certaine teinte au plan en fonction de la classe qui serait affectée à un point dans cette région. Cela nous permet de visualiser la *frontière de décision*, qui est la séparation entre la zone où l'algorithme assigne la valeur de classe 0, et celle où il assigne la valeur de classe 1.

Le code qui suit permet de visualiser les frontières de décision pour un, trois et neuf voisins, comme l'illustre la Figure 2.6 :

In[16]:

```
ig, axes = plt.subplots(1, 3, figsize=(10, 3))
```

```
for n_neighbors, ax in zip([1, 3, 9], axes):
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{} neighbor(s)".format(n_neighbors))
    ax.set_xlabel("feature 0")
    ax.set_ylabel("feature 1")
axes[0].legend(loc=3)
```

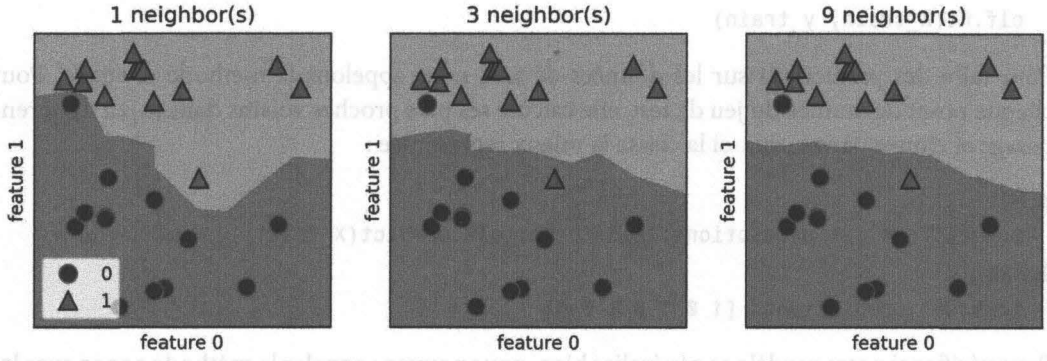



*Figure 2.6 : Frontières de décision créées par le modèle des plus proches voisins pour trois valeurs différentes de* `n_neighbors`.

Comme vous pouvez le voir sur la gauche de la figure, utiliser un seul voisin produit une frontière de décision qui suit de près les données d'apprentissage. Plus il y a de voisins, et plus cette frontière tend à se lisser. Et plus la frontière est lissée, plus le modèle est simple. En d'autres termes, utiliser moins de voisins correspond à une complexité élevée du modèle (comme sur la gauche de la Figure 2.1), et utiliser plus de voisins correspond à une moindre complexité du modèle (comme sur la droite de la Figure 2.1). Si vous considérez le cas extrême où le nombre de voisins serait égal au nombre de tous les points de données dans le jeu d'apprentissage, chaque point de test aurait exactement les mêmes voisins (tous les points d'apprentissage), et toutes les prédictions seraient identiques : la classe avec la plus grande fréquence dans le jeu d'apprentissage.

Voyons alors si nous pouvons confirmer la connexion entre complexité du modèle et généralisation évoquée plus haut. Nous allons pour cela nous servir du jeu de données du monde réel concernant les mesures cliniques des tumeurs du sein, `cancer`. Nous commencerons par partager ces données en un jeu d'apprentissage et un jeu de test. Nous évaluerons ensuite les performances du modèle en nous servant de différentes valeurs pour le nombre de voisins. Les résultats sont illustrés sur la Figure 2.7 :

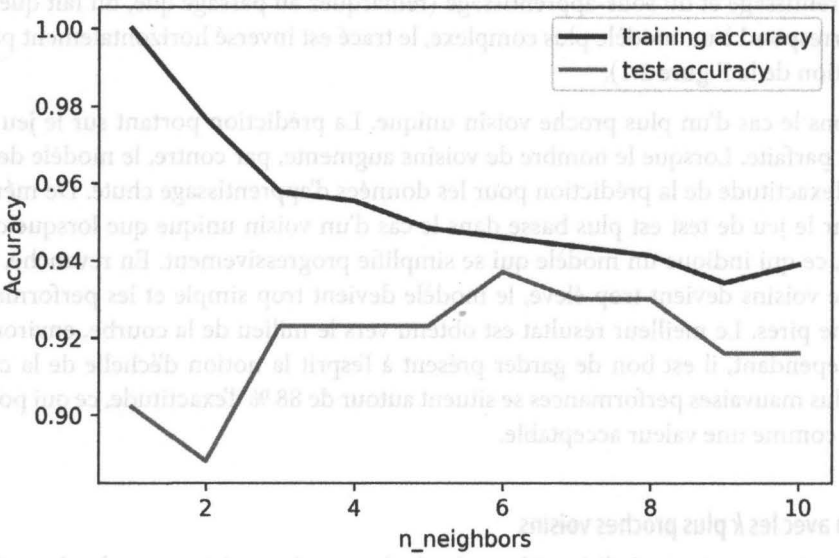



*Figure 2.7 : Comparaison entre exactitude de l'apprentissage et du test en fonction de* `n_neighbors`.

**In[17]:**

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)
training_accuracy = []
test_accuracy = []
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    training_accuracy.append(clf.score(X_train, y_train))
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend()
```

Le tracé montre sur l'axe y l'exactitude des jeux d'apprentissage et de test en fonction du nombre de voisins représenté sur l'axe x. Bien que les tracés provenant de données du monde réel soient rarement très lisses, nous pouvons tout de même y reconnaître certaines des caractéristiques

du surapprentissage et du sous-apprentissage (remarquez au passage que, du fait que moins de voisins correspond à un modèle plus complexe, le tracé est inversé horizontalement par rapport à l'illustration de la Figure 2.1).

Considérons le cas d'un plus proche voisin unique. La prédiction portant sur le jeu d'apprentissage est parfaite. Lorsque le nombre de voisins augmente, par contre, le modèle devient plus simple et l'exactitude de la prédiction pour les données d'apprentissage chute. De même, l'exactitude pour le jeu de test est plus basse dans le cas d'un voisin unique que lorsque ce nombre augmente, ce qui indique un modèle qui se simplifie progressivement. En revanche, lorsque le nombre de voisins devient trop élevé, le modèle devient trop simple et les performances sont alors même pires. Le meilleur résultat est obtenu vers le milieu de la courbe, environ pour six voisins. Cependant, il est bon de garder présent à l'esprit la notion d'échelle de la courbe. En effet, les plus mauvaises performances se situent autour de 88 % d'exactitude, ce qui pourrait être considéré comme une valeur acceptable.

### Régression avec les *k* plus proches voisins

Il existe aussi une variante de l'algorithme des *k* plus proches voisins pour la régression. Commençons à nouveau par un unique voisin en utilisant cette fois le jeu de données `wave`. Nous avons ajouté trois points de données de test : les étoiles qui se trouvent en bas, sur l'axe x. La prédiction que nous effectuons est simplement la valeur cible du plus proche voisin, autrement dit ici les étoiles plus sombres, vers le milieu de la Figure 2.8 :

**In[18] :**

```
mglearn.plots.plot_knn_regression(n_neighbors=1)
```

À nouveau, nous pouvons utiliser plus qu'un simple voisin pour la régression. Dans ce cas, la prédiction est la moyenne des voisins concernés (voir la Figure 2.9) :

**In[19] :**

```
mglearn.plots.plot_knn_regression(n_neighbors=3)
```

L'algorithme des *k* plus proches voisins pour la régression est implémenté dans la classe `KNeighborsRegressor` de `scikit-learn`. Elle s'utilise de la même manière que la classe `KNeighborsClassifier` :

**In[20] :**

```
from sklearn.neighbors import KNeighborsRegressor

X, y = mglearn.datasets.make_wave(n_samples=40)

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

reg = KNeighborsRegressor(n_neighbors=3)
reg.fit(X_train, y_train)
```

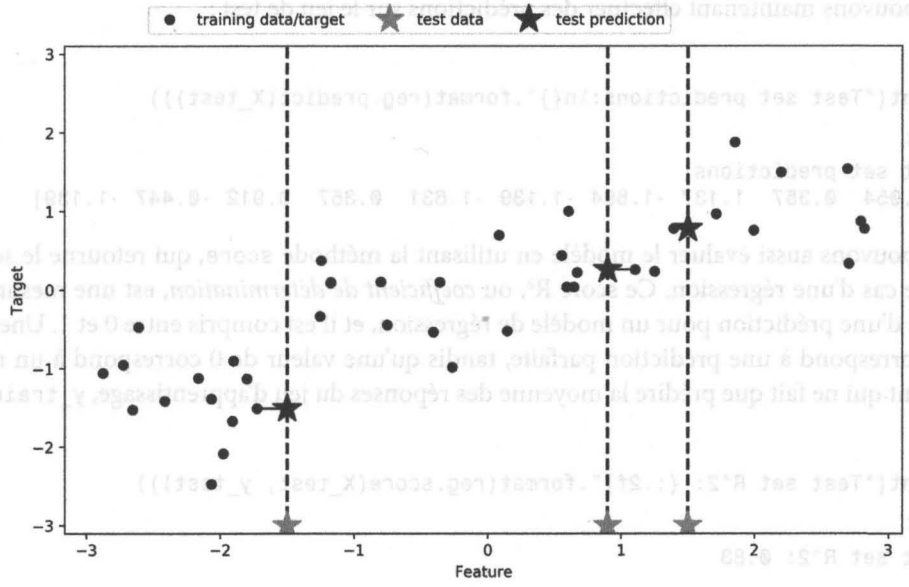



*Figure 2.8 : Prédictions faites par une régression avec un seul plus proche voisin sur le jeu de données* wave.

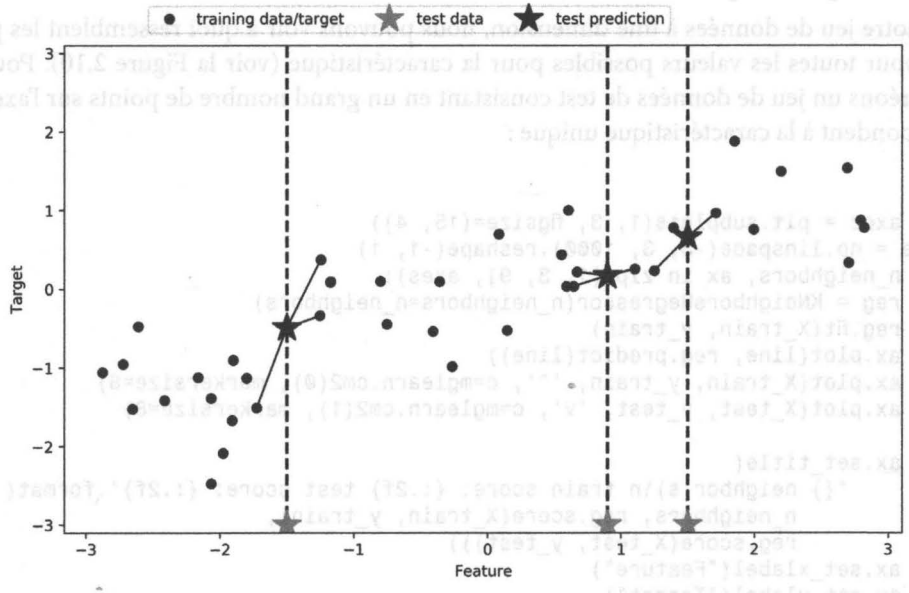



*Figure 2.9 : Prédictions faites par une régression avec trois plus proches voisins sur le jeu de données* wave.

Nous pouvons maintenant effectuer des prédictions sur le jeu de test :

**In[21]:**

```
print("Test set predictions:\n{}".format(reg.predict(X_test)))
```

**Out[21]:**

```
Test set predictions:
[-0.054  0.357  1.137 -1.894 -1.139 -1.631  0.357  0.912 -0.447 -1.139]
```

Nous pouvons aussi évaluer le modèle en utilisant la méthode `score`, qui retourne le score $R^2$ dans le cas d'une régression. Ce score $R^2$, ou *coefficient de détermination*, est une mesure de la qualité d'une prédiction pour un modèle de régression, et il est compris entre 0 et 1. Une valeur de 1 correspond à une prédiction parfaite, tandis qu'une valeur de 0 correspond à un modèle constant qui ne fait que prédire la moyenne des réponses du jeu d'apprentissage, `y_train` :

**In[22]:**

```
print("Test set R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

**Out[22]:**

```
Test set R^2: 0.83
```

Ici, le score est de 0,83, ce qui indique un modèle relativement bon.

### Analyser KNeighborsRegressor

Pour notre jeu de données à une dimension, nous pouvons voir à quoi ressemblent les prédictions pour toutes les valeurs possibles pour la caractéristique (voir la Figure 2.10). Pour cela, nous créons un jeu de données de test consistant en un grand nombre de points sur l'axe x, qui correspondent à la caractéristique unique :

**In[23]:**

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
    reg = KNeighborsRegressor(n_neighbors=n_neighbors)
    reg.fit(X_train, y_train)
    ax.plot(line, reg.predict(line))
    ax.plot(X_train, y_train, '^', c=mglearn.cm2(0), markersize=8)
    ax.plot(X_test, y_test, 'v', c=mglearn.cm2(1), markersize=8)

    ax.set_title(
        "{} neighbor(s)\n train score: {:.2f} test score: {:.2f}".format(
            n_neighbors, reg.score(X_train, y_train),
            reg.score(X_test, y_test)))
    ax.set_xlabel("Feature")
    ax.set_ylabel("Target")
axes[0].legend(["Model predictions", "Training data/target",
                "Test data/target"], loc="best")
```

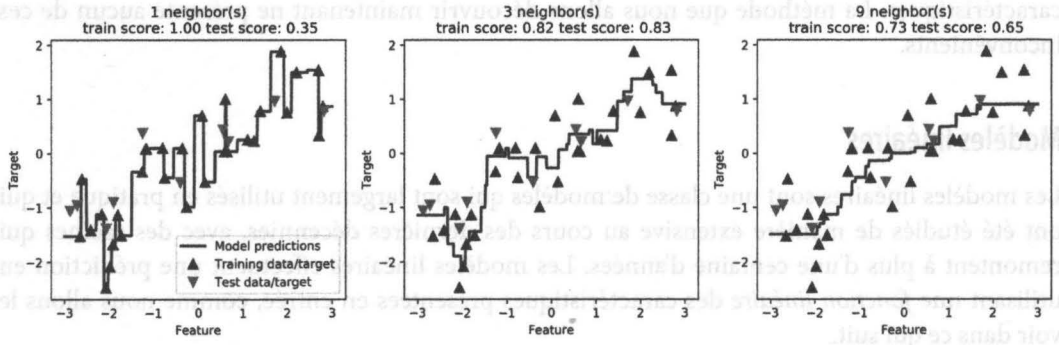



*Figure 2.10 : Comparaison entre les prédictions faites par une régression avec plusieurs plus proches voisins pour différentes valeurs de* `n_neighbors`.

Comme nous pouvons le voir sur le tracé, dans le cas d'un voisin unique, chaque point du jeu d'apprentissage a une évidente incidence sur les prédictions, et les valeurs ainsi prédites passent par tous les points de données. Cela conduit donc à une prédiction très déséquilibrée. Augmenter le nombre de voisins permet de lisser les prédictions, mais dans ce cas elles ne s'ajustent pas aussi bien avec les données d'apprentissage.

### Forces, faiblesses et paramètres

En principe, le classifieur `KNeighbors` possède deux paramètres importants : le nombre de voisins d'une part, et la manière donc vous mesurez la distance entre les points de données d'autre part. En pratique, se limiter à un petit nombre de voisins, comme trois ou cinq, fonctionne souvent bien, mais vous devriez certainement avoir à ajuster ce paramètre. Choisir la bonne mesure de distance déborde quelque peu du cadre de ce livre. Par défaut, c'est la distance euclidienne qui est employée, ce qui donne dans la plupart des cas de bons résultats.

L'une des forces de l'algorithme des *k* plus proches voisins est que ce modèle est très facile à comprendre, et qu'il donne souvent des performances raisonnables sans avoir à effectuer des tas d'ajustements. Utiliser cet algorithme est donc une bonne méthode de base à essayer avant d'envisager des techniques plus avancées. Construire le modèle des plus proches voisins est souvent très rapide, mais le calcul de la prédiction peut être lent si le jeu de données est très étendu (que ce soit en nombre de caractéristiques ou en nombre d'échantillons). Lorsque vous utilisez cet algorithme, il est important d'effectuer un prétraitement sur vos données (voir à ce sujet le Chapitre 3). Cette approche ne fonctionne souvent pas bien avec des jeux de données qui comportent de nombreuses caractéristiques (des centaines, voire plus), et elle est même particulièrement mauvaise dans le cas de jeux de données où la plupart des caractéristiques sont nulles la plupart du temps (ce que l'on appelle des *sparse datasets*, ou jeux de données clairsemés).

Au final, et bien que l'algorithme des *k* plus proches voisins soit facile à comprendre, il n'est pas souvent utilisé en pratique, du fait de sa lenteur et de son incapacité à gérer de nombreuses

caractéristiques. La méthode que nous allons découvrir maintenant ne présente aucun de ces inconvénients.

## Modèles linéaires

Les modèles linéaires sont une classe de modèles qui sont largement utilisés en pratique et qui ont été étudiés de manière extensive au cours des dernières décennies, avec des racines qui remontent à plus d'une centaine d'années. Les modèles linéaires effectuent une prédiction en utilisant une *fonction linéaire* des caractéristiques présentées en entrée, comme nous allons le voir dans ce qui suit.

### Modèles linéaires pour la régression

Dans le cas de la régression, la formule de prédiction générale pour un modèle linéaire se présente de la manière suivante :

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b$$

Ici, $x[0]$ à $x[p]$ représentent les caractéristiques d'un certain point de donnée (dans la formule ci-dessus, il y a donc p+1 caractéristiques, ou variables). De plus, $w$ et $b$ sont les paramètres du modèle qui est *entraîné*, tandis que $\hat{y}$ est la prédiction réalisée par le modèle. Pour un jeu de données avec une seule caractéristique, la formule devient :

$$\hat{y} = w[0] * x[0] + b$$

Cette formule rappelle la fonction affine qui a certainement fait les beaux jours de vos études mathématiques ($y = ax + b$). Ici, $w[0]$ représente le coefficient directeur (la pente de la droite), et $b$ l'ordonnée à l'origine (donc le décalage par rapport à l'axe y). S'il y a plus de caractéristiques, $w$ contient les coefficients directeurs correspondant à chaque axe de caractéristique. De manière alternative, vous pouvez aussi considérer la réponse prédictive comme étant la somme pondérée des caractéristiques d'entrée, avec des « poids » (qui peuvent être négatifs) fournis par les valeurs de $w$.

Essayer d'entraîner les paramètres $w[0]$ et $b$ sur notre jeu de données unidimensionnel wave pourrait conduire à la droite suivante (voir la Figure 2.11) :

**In[24]:**

```
mglearn.plots.plot_linear_regression_wave()
```

**Out[24]:**

```
w[0]: 0.393906  b: -0.031804
```

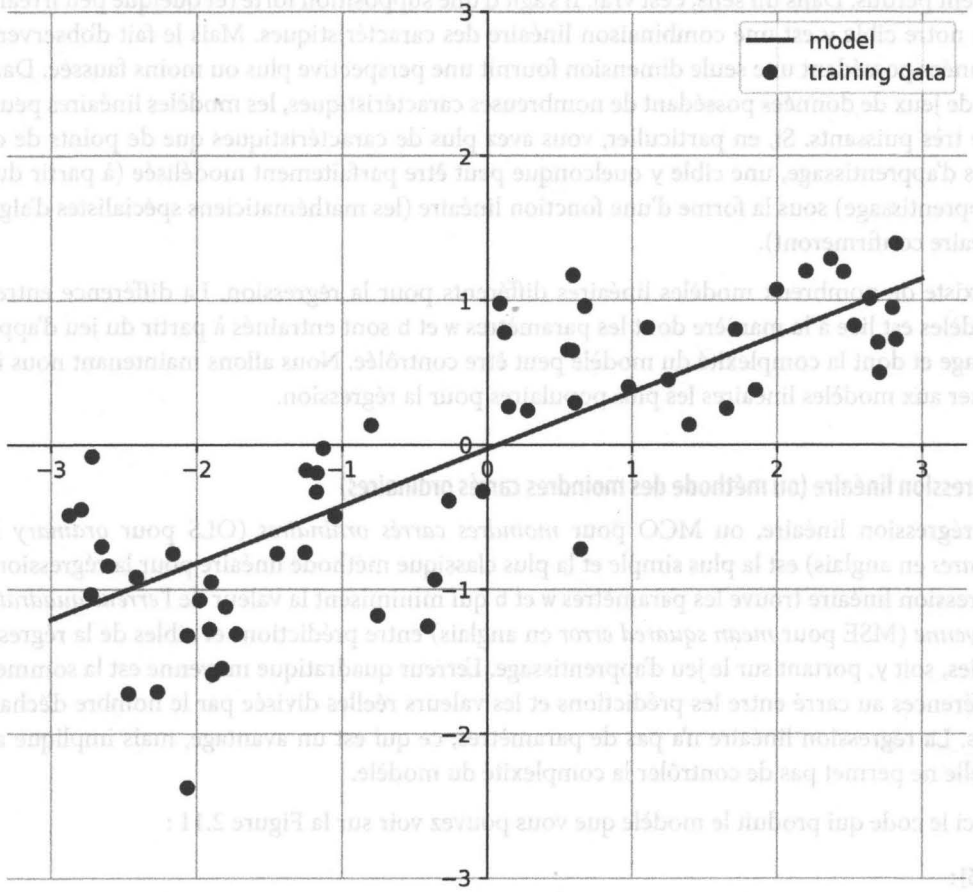



*Figure 2.11 : Prédictions d'un modèle linéaire appliqué au jeu de données* wave.

Nous avons ajouté les valeurs de coordonnées aux axes pour améliorer la visibilité du tracé. En regardant la valeur de `w[0]`, nous constatons que la pente devrait avoir un coefficient d'environ 0.4, ce que confirme visuellement le tracé. L'interception se situe là où la ligne de prédiction devrait couper l'axe y, soit un peu en dessous de zéro, ce que confirme également l'image.

Les modèles linéaires pour la régression peuvent être caractérisés comme des modèles de régression pour lesquels la prédiction est une ligne droite dans le cas d'une caractéristique unique, un plan pour deux caractéristiques, ou un hyperplan pour des dimensions plus élevées (autrement dit avec davantage de caractéristiques).

Si vous comparez les prédictions obtenues avec la ligne droite avec celles réalisées par `KNeighborsRegressor` (reportez-vous à la Figure 2.10), vous pouvez penser que cette ligne droite semble très restrictive. C'est apparemment comme si tous les détails fins des données

étaient perdus. Dans un sens, c'est vrai. Il s'agit d'une supposition forte (et quelque peu irréaliste) que notre cible y est une combinaison linéaire des caractéristiques. Mais le fait d'observer des données possédant une seule dimension fournit une perspective plus ou moins faussée. Dans le cas de jeux de données possédant de nombreuses caractéristiques, les modèles linéaires peuvent être très puissants. Si, en particulier, vous avez plus de caractéristiques que de points de données d'apprentissage, une cible y quelconque peut être parfaitement modélisée (à partir du jeu d'apprentissage) sous la forme d'une fonction linéaire (les mathématiciens spécialistes d'algèbre linéaire confirmeront).

Il existe de nombreux modèles linéaires différents pour la régression. La différence entre ces modèles est liée à la manière dont les paramètres `w` et `b` sont entraînés à partir du jeu d'apprentissage et dont la complexité du modèle peut être contrôlée. Nous allons maintenant nous intéresser aux modèles linéaires les plus populaires pour la régression.

### Régression linéaire (ou méthode des moindres carrés ordinaires)

La régression linéaire, ou MCO pour *moindres carrés ordinaires* (OLS pour *ordinary least squares* en anglais) est la plus simple et la plus classique méthode linéaire pour la régression. La régression linéaire trouve les paramètres `w` et `b` qui minimisent la valeur de l'*erreur quadratique moyenne* (MSE pour *mean squared error* en anglais) entre prédictions et cibles de la régression vraies, soit `y`, portant sur le jeu d'apprentissage. L'erreur quadratique moyenne est la somme des différences au carré entre les prédictions et les valeurs réelles divisée par le nombre d'échantillons. La régression linéaire n'a pas de paramètres, ce qui est un avantage, mais implique aussi qu'elle ne permet pas de contrôler la complexité du modèle.

Voici le code qui produit le modèle que vous pouvez voir sur la Figure 2.11 :

**In[25]:**

```
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

lr = LinearRegression().fit(X_train, y_train)
```

Les paramètres de « pente » (`w`), soit les valeurs de pondération, ou *poids* ou encore *coefficients*, sont enregistrés dans l'attribut `coef_`, tandis que l'ordonnée à l'origine (`b`) se trouve dans l'attribut `intercept_` :

**In[26]:**

```
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
```

**Out[26]:**

```
lr.coef_: [ 0.394]
lr.intercept_: -0.031804343026759746
```



Vous avez sans doute remarqué le trait de soulignement un peu intrigant qui se trouve à la fin de `coef_` et d'`intercept_`. En fait, `scikit-learn` enregistre toujours tout ce qui est dérivé des données d'apprentissage dans des attributs qui se terminent par ce trait de soulignement. Cela permet de les différencier des paramètres qui sont définis par l'utilisateur.

L'attribut `intercept_` est toujours un nombre en virgule flottante, tandis que l'attribut `coef_` est un tableau NumPy contenant une entrée par caractéristique source. Du fait que nous n'avons qu'une unique caractéristique dans le jeu de données `wave`, `coef_` ne contient qu'une seule entrée.

Voyons ce que donnent les performances pour notre jeu d'apprentissage et notre jeu de test :

**In[27]:**

```
print("Training set score: {:.2f}".format(lr.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

**Out[27]:**

```
Training set score: 0.67
Test set score: 0.66
```

Un score $R^2$ aux alentours de 0.66 n'est pas très bon, mais nous pouvons constater en même temps que les scores pour le jeu d'apprentissage et le jeu de test sont très proches l'un de l'autre. Pour ce jeu de données ne possédant qu'une seule dimension, le risque de surapprentissage est faible du fait que le modèle est très simple (ou restreint). Cependant, avec des jeux de données possédant de nombreuses dimensions (autrement dit, avec un grand nombre de caractéristiques), les modèles linéaires gagnent en puissance, donc aussi en complexité, ce qui augmente le risque de surapprentissage. Jetons un coup d'œil sur la manière dont `LinearRegression` se comporte sur un jeu de données plus complexe, en l'occurrence `Boston Housing`. Souvenez-vous que ce jeu de données contient 506 échantillons (lignes) et 104 caractéristiques dérivées. Nous commençons par charger le jeu de données. Il est ensuite divisé en un jeu d'apprentissage et un jeu de test. Puis nous construisons le modèle de régression linéaire comme nous l'avons fait plus haut :

**In[28]:**

```
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

En comparant les scores sur le jeu d'apprentissage et sur le jeu de test, nous constatons que la prédiction est très précise pour le premier, mais encore pire pour le second :

**In[29]:**

```
print("Training set score: {:.2f}".format(lr.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

**Out[29]:**

```
Training set score: 0.95
Test set score: 0.61
```

Cette discordance entre les performances obtenues pour le jeu d'apprentissage et le jeu de test est clairement un signe de surapprentissage. Nous devrions donc chercher un modèle qui nous permette de contrôler sa complexité. Une des alternatives les plus courantes est la *régression ridge* que nous allons étudier maintenant.

### Régression ridge

Il s'agit également d'un modèle linéaire pour la régression, et donc la formule qu'il utilise pour faire des prédictions est la même que dans le cas des moindres carrés ordinaires. Dans la régression ridge, cependant, les coefficients (`w`) sont choisis non seulement de manière à ce qu'ils fournissent de bonnes prédictions sur le jeu d'apprentissage, mais aussi qu'ils remplissent une contrainte supplémentaire. Nous voulons également que les coefficients de pondération soient aussi petits que possible. En d'autres termes, toutes les entrées de `w` devraient être proches de zéro. D'un point de vue intuitif, cela signifie que chaque caractéristique devrait avoir aussi peu d'impact que possible sur la sortie (donc présenter une pente faible) tout en fournissant de bonnes prédictions. Cette contrainte est un exemple de ce qui est appelé *régularisation*. Cette régularisation signifie que l'on restreint explicitement un modèle pour éviter le surapprentissage. Le type particulier utilisé par la régression ridge est connu sous le nom de régularisation L2 (d'un point de vue mathématique, la méthode ridge pénalise le carré de la norme L2, ou encore la longueur euclidienne de `w`).

La régression ridge est implémentée dans `linear_model.Ridge`. Voyons comment elle se comporte avec le jeu de données étendu Boston Housing :

**In[30]:**

```
from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge.score(X_test, y_test)))
```

**Out[30]:**

```
Training set score: 0.89
Test set score: 0.75
```

Comme vous pouvez le constater, le score du jeu d'apprentissage est ici *plus bas* qu'avec le modèle `LinearRegression`, tandis que le score du jeu de test est *plus élevé*.

Cela est cohérent avec notre attente. Avec la régression linéaire, nous provoquions un surapprentissage sur nos données. Le modèle `ridge` est plus restrictif, et donc le surapprentissage est moins probable. Un modèle moins complexe donne de plus mauvaises performances sur le jeu

d'apprentissage, mais, par contre, il se traduit par une meilleure généralisation. Comme nous sommes uniquement intéressés par les performances de généralisation, nous devrions choisir le modèle `ridge` plutôt que le modèle `LinearRegression`.

Le modèle `ridge` établit un compromis entre la simplicité du modèle (coefficients proches de zéro) et les performances sur le jeu d'apprentissage. L'importance relative entre ces deux pôles, simplicité d'une part et performances de l'autre (donc la quantité de régularisation souhaitée), peut être spécifiée par l'utilisateur à l'aide du paramètre `alpha`. Dans l'exemple précédent, nous avons utilisé la valeur par défaut, `alpha = 1.0`. Il n'y a cependant aucune raison pour que cela nous donne le meilleur compromis. La valeur optimale pour `alpha` dépend du jeu de données particulier que nous utilisons. Augmenter `alpha` force les coefficients à se rapprocher davantage de zéro, ce qui fait décroître les performances sur le jeu d'apprentissage mais peut aider à obtenir une meilleure généralisation. Par exemple :

**In[31]:**

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge10.score(X_test, y_test)))
```

**Out[31]:**

```
Training set score: 0.79
Test set score: 0.64
```

Abaisser la valeur d'`alpha` permet aux coefficients d'être moins restreints, ce qui correspond à un déplacement vers la droite sur la Figure 2.1. Pour de très petites valeurs de ce paramètre, le modèle ressemble de plus en plus à une régression linéaire (les deux finissent par se confondre si `alpha` devient nul) :

**In[32]:**

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge01.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge01.score(X_test, y_test)))
```

**Out[32]:**

```
Training set score: 0.93
Test set score: 0.77
```

Ici, `alpha=0.1` semble bien fonctionner. Nous pourrions essayer de le diminuer encore plus pour améliorer la généralisation. Pour l'instant, notez le rapport entre le paramètre `alpha` et la complexité du modèle telle qu'elle est illustrée sur la Figure 2.1. Nous discuterons dans le Chapitre 5 de méthodes servant à aider à la sélection des paramètres.

Nous pouvons également obtenir un aperçu plus qualitatif de la manière dont `alpha` change le modèle en inspectant l'attribut `coef_` pour différentes valeurs d'`alpha`. Une valeur plus élevée de ce paramètre signifie un modèle plus restreint, et nous nous attendons donc dans ce cas à des grandeurs plus petites pour les entrées de `coef_`. C'est ce que confirme le tracé de la Figure 2.12 :

**In[33]:**

```
plt.plot(ridge.coef_, 's', label="Ridge alpha=1")
plt.plot(ridge10.coef_, '^', label="Ridge alpha=10")
plt.plot(ridge01.coef_, 'v', label="Ridge alpha=0.1")

plt.plot(lr.coef_, 'o', label="LinearRegression")
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
xlims = plt.xlim()
plt.hlines(0, xlims[0], xlims[1])
plt.xlim(xlims)
plt.ylim(-25, 25)
plt.legend()
```

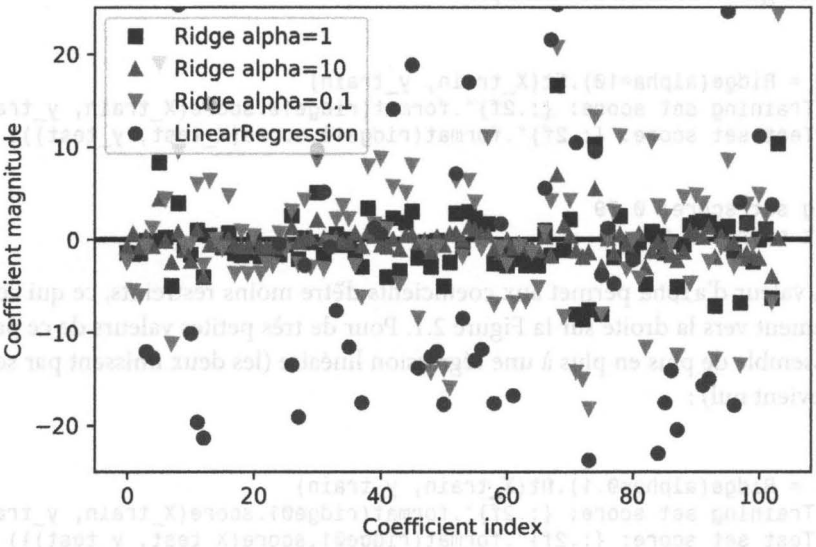



*Figure 2.12 : Comparaison entre les grandeurs des coefficients pour une régression ridge avec différentes valeurs du paramètre* `alpha` *et une régression linéaire.*

Ici, l'axe x énumère les entrées de `coef_`. Autrement dit, `x=0` montre le coefficient associé à la première caractéristique, `x=1` celui associé à la seconde caractéristique, et ainsi de suite jusqu'à 100. L'axe y montre les valeurs numériques correspondantes. On peut observer que, pour `alpha=10`, les valeurs des coefficients se situent pour l'essentiel entre -3 et 3. Les coefficients pour le modèle `ridge` avec `alpha=1` sont un peu plus grands. Les points correspondant à `alpha=0.1` ont une amplitude encore plus importante. Quant à de nombreux points de la régression linéaire sans régularisation (l'équivalent de `alpha=0`), leur valeur absolue est si grande qu'ils ne rentrent même pas dans le tracé.

Une autre manière de comprendre l'influence de la régularisation consiste à fixer une valeur à `alpha`, puis à faire varier la quantité de données disponibles dans le jeu d'apprentissage. Sur la Figure 2.13, nous avons pris des sous-ensembles du jeu de données Boston Housing, et nous avons évalué `LinearRegression` ainsi que `Ridge(alpha=1)` pour différentes tailles de notre jeu d'apprentissage ainsi constitué (les tracés qui montrent les performances d'un modèle en fonction de la taille du jeu de données sont appelés des *courbes d'apprentissage*, ou *learning curves*) :

**In[34]:**

```
mglearn.plots.plot_ridge_n_samples()
```

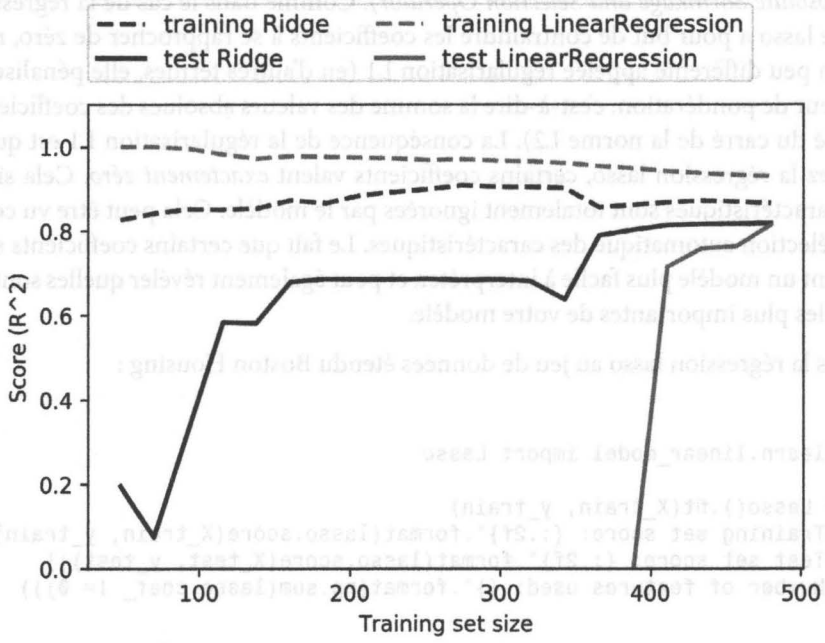



*Figure 2.13 : Courbes d'apprentissage pour la régression ridge et la régression linéaire appliquées au jeu de données Boston Housing.*

Comme on pouvait s'y attendre, le score pour les données d'apprentissage est meilleur que celui des données de test, et ce pour toutes les tailles choisies et aussi bien pour le modèle de régression ridge que pour la régression linéaire. Du fait que la régression ridge est régularisée, son score sur les données d'apprentissage est inférieur dans tous les cas à celui qui est obtenu avec la régression linéaire. Cependant, le score pour la régression ridge est meilleur avec le jeu de test, en particulier pour les jeux de données de taille réduite. En dessous de 400 points de données, la régression linéaire n'est pas capable d'apprendre quoi que ce soit. Au fur et à mesure que la quantité de données augmente, les deux modèles s'améliorent, et, à la fin, la régression linéaire finit par rattraper la régression ridge. La leçon à en tirer est que, avec suffisamment de données d'ap-

prentissage, la régularisation devient moins importante, jusqu'au moment où régression ridge et régression linéaire obtiennent les mêmes performances (le fait que cela se produise lorsque nous utilisons le jeu de données complet est juste un hasard). Un autre aspect intéressant de la Figure 2.13 est la décroissance des performances sur l'apprentissage pour la régression linéaire. Si on ajoute davantage de données, il devient plus difficile pour un modèle de se surajuster, ou de mémoriser les données.

### Lasso

Une alternative à `ridge` pour les régressions linéaires régularisées s'appelle Lasso (acronyme de *Least Absolute Shrinkage and Selection Operator*). Comme dans le cas de la régression ridge, la méthode lasso a pour but de contraindre les coefficients à se rapprocher de zéro, mais d'une manière un peu différente appelée régularisation L1 (en d'autres termes, elle pénalise la norme L1 du vecteur de pondération, c'est-à-dire la somme des valeurs absolues des coefficients au lieu de la moitié du carré de la norme L2). La conséquence de la régularisation L1 est que, lorsque vous utilisez la régression lasso, certains coefficients valent *exactement zéro*. Cela signifie que certaines caractéristiques sont totalement ignorées par le modèle. Cela peut être vu comme une forme de sélection automatique des caractéristiques. Le fait que certains coefficients soient nuls rend souvent un modèle plus facile à interpréter, et peut également révéler quelles sont les caractéristiques les plus importantes de votre modèle.

Appliquons la régression lasso au jeu de données étendu Boston Housing :

**In[35]:**

```
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso.coef_ != 0)))
```

**Out[35]:**

```
Training set score: 0.29
Test set score: 0.21
Number of features used: 4
```

Comme vous pouvez le constater, `lasso` donne des résultats faibles, aussi bien sur le jeu d'apprentissage que sur le jeu de test. Cela indique un sous-apprentissage, et nous remarquons que seules 4 des 104 caractéristiques sont utilisées. Comme `ridge`, la régression lasso a également un paramètre de régularisation, `alpha`, qui contrôle l'impact de la méthode sur la valeur des coefficients. Dans l'exemple précédent, nous avons utilisé la valeur par défaut, `alpha=1.0`. Pour réduire le sous-apprentissage, essayons de diminuer `alpha`. Dans ce cas, nous devons également augmenter la valeur par défaut du paramètre `max_iter` (le nombre maximum d'itérations à effectuer) :

**In[36]:**

```
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso001.coef_ != 0)))
```

**Out[36]:**

```
Training set score: 0.90
Test set score: 0.77
Number of features used: 33
```

Une valeur plus basse d'`alpha` nous a permis d'ajuster un modèle plus complexe, fonctionnant mieux avec les données d'apprentissage et de test. La performance est légèrement meilleure qu'avec `ridge`, et nous utilisons seulement 33 des 104 caractéristiques. Cela rend potentiellement notre modèle plus facile à comprendre.

Cependant, si nous donnons à `alpha` une valeur trop basse, nous perdons à nouveau l'effet de régularisation et nous finissons par un surapprentissage, avec un résultat semblable à celui de LinearRegression :

**In[37]:**

```
lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso00001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso00001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso00001.coef_ != 0)))
```

**Out[37]:**

```
Training set score: 0.95
Test set score: 0.64
Number of features used: 94
```

À nouveau, nous pouvons tracer les coefficients des différents modèles, comme sur la Figure 2.12. Le résultat est montré sur la Figure 2.14.

**In[38]:**

```
plt.plot(lasso.coef_, 's', label="Lasso alpha=1")
plt.plot(lasso001.coef_, '^', label="Lasso alpha=0.01")
plt.plot(lasso00001.coef_, 'v', label="Lasso alpha=0.0001")

plt.plot(ridge01.coef_, 'o', label="Ridge alpha=0.1")
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
```

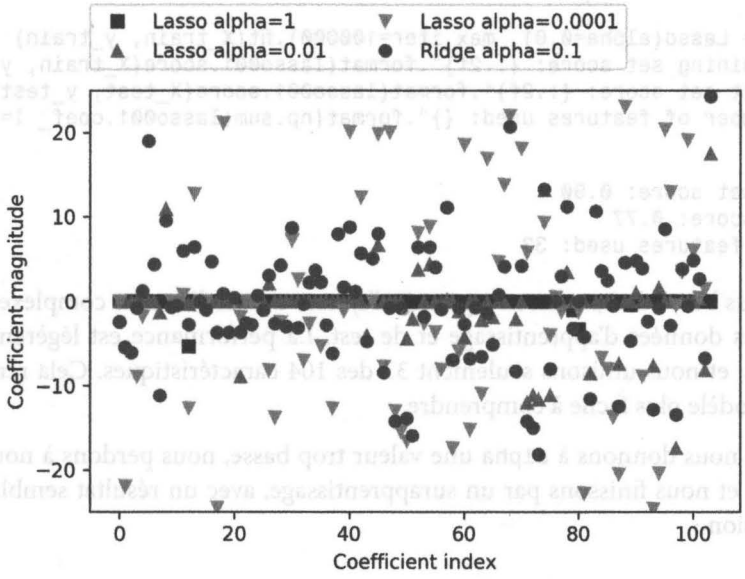



*Figure 2.14 : Comparaison de l'amplitude des coefficients pour une régression lasso avec différentes valeurs pour* `alpha` *et la régression ridge.*

Pour `alpha=1`, nous voyons que non seulement la plupart des coefficients sont nuls (ce que nous savions déjà), mais que les coefficients qui restent présentent une faible amplitude. Si nous diminuons la valeur d'`alpha` jusqu'à `0.01`, nous obtenons la solution représentée par un triangle pointant vers le haut sur la figure. Dans ce cas, il y a plus de caractéristiques qui valent exactement zéro. Avec `alpha=0.0001`, nous obtenons un modèle qui n'est pratiquement pas régularisé, avec la plupart des coefficients qui ne sont pas nuls et qui présentent même une importante amplitude. Par comparaison, la meilleure solution `ridge` est représentée sous forme de cercles. Le modèle `ridge` avec `alpha=0.1` a une performance prédictive semblable au modèle `lasso` avec `alpha=0.01`, mais, dans ce cas, tous les coefficients sont non nuls.

En pratique, la régression ridge est généralement le premier choix à faire entre ces deux modèles. Cependant, si vous avez une grande quantité de caractéristiques et que vous pensiez que quelques-unes seulement sont importantes, le modèle `lasso` pourrait être un meilleur choix. De même, si vous voulez avoir un modèle qui soit facile à interpréter, `lasso` fournit effectivement une solution bien mieux compréhensible du fait qu'elle ne sélectionne qu'un sous-ensemble des caractéristiques. `scikit-learn` propose également la classe `ElasticNet`, qui combine les termes de régularisation des deux régressions `lasso` et `ridge` en introduisant un *ratio de mélange* entre les deux (ou *mix ratio*). En pratique, cette combinaison fonctionne mieux, mais au prix d'un ajustement à effectuer entre deux paramètres : un pour la régularisation L1, et l'autre pour la régularisation L2.

### Modèles linéaires pour la classification

Les modèles linéaires sont également employés de manière intensive pour la classification. Commençons par observer une classification binaire. Dans ce cas, une prédiction est effectuée en utilisant la formule suivante :

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$$

Cette formule est très semblable à celle de la régression linéaire, mais au lieu de retourner simplement la somme pondérée des caractéristiques, nous effectuons un seuillage de la valeur prédite par rapport à zéro. Si la fonction est négative, nous prédisons la classe en tant que -1. Si elle est positive, nous prédisons la classe en tant que +1. Cette règle de prédiction est commune à tous les modèles linéaires utilisés pour la classification. De nouveau, il existe de multiples manières différentes de trouver les coefficients (*w*) et le terme constant ou *intercept* (b).

Dans le cas des modèles linéaires pour une régression, la sortie, $\hat{y}$, est une fonction linéaire des caractéristiques : une ligne, un plan ou un hyperplan (pour les dimensions supérieures). Dans le cas des modèles linéaires pour la classification, la *frontière de décision* est une fonction linéaire de l'entrée. En d'autres termes, un classifieur linéaire (binaire) est un classifieur qui sépare deux classes en utilisant une ligne, un plan ou un hyperplan. Nous allons en voir des exemples dans ce qui suit.

Il existe de nombreux algorithmes pour les modèles linéaires d'apprentissage automatique. Ces algorithmes diffèrent tous sur les deux aspects suivants :

- La manière dont ils mesurent en quoi une combinaison particulière des coefficients et de l'intercept (le terme constant) permet de bien ajuster les données d'apprentissage.
- Si un type de régularisation est utilisé, et lequel.

Différents algorithmes choisissent différentes manières de mesurer ce que « bien ajuster les données d'apprentissage » signifie. Pour des raisons purement mathématiques, il n'est pas possible d'ajuster `w` et `b` de façon à minimiser le nombre de mauvaises classifications produites par l'algorithme. Pour notre propos, et pour de nombreuses applications, les différents choix pour le premier point de la liste ci-dessus (ce que l'on appelle des *fonctions de perte*, ou *loss functions*) sont peu significatifs.

Les deux algorithmes de classification linéaire les plus courants sont la *régression logistique*, implémentée dans `linear_model.LogisticRegression`, et la *classification SVM linéaire* implémentée dans *svm*. En dépit de son nom, la régression logistique est un algorithme de classification, et non de régression, et `LogisticRegression` ne doit pas être confondu avec `LinearRegression`.



SVM signifie *Support Vector Machine*, soit machine à vecteurs de support, mais que l'on traduit généralement par *séparateur à vaste marge*, ce qui permet de conserver les initiales SVM.

Nous pouvons appliquer les modèles `LogisticRegression` et `LinearSVC` au jeu de données `forge`, et visualiser la frontière de décision déterminée par chaque modèle linéaire (voir la Figure 2.15).

**In[39]:**

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5,
                                    ax=ax, alpha=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{}".format(clf.__class__.__name__))
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
axes[0].legend()
```

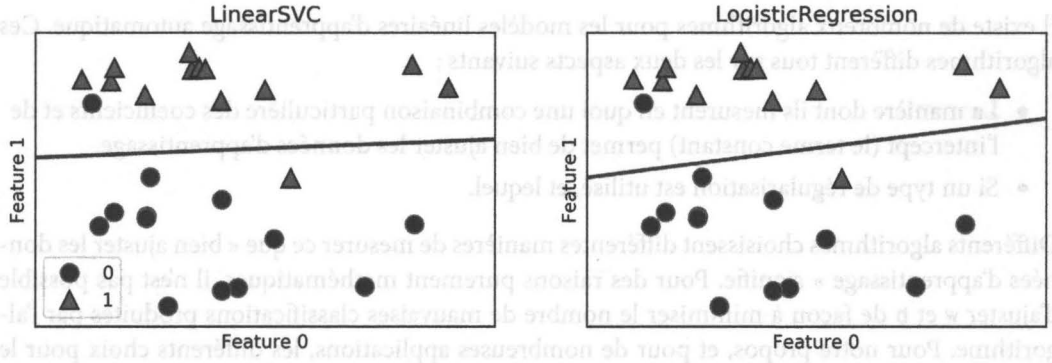



*Figure 2.15 : Frontières de décision pour la classification SVM linéaire et la régression logistique pour le jeu de données `forge` avec les paramètres par défaut.*

Sur cette figure, nous retrouvons à nouveau la première caractéristique du jeu de données `forge` sur l'axe x, et la seconde caractéristique sur l'axe y. Nous affichons les frontières de décision déterminées respectivement par `LinearSVC` et `LogisticRegression` sous la forme de lignes droites, séparant ainsi l'aire classifiée comme formant la classe 1 en haut, et celle formant la classe 0 en bas. En d'autres termes, tout nouveau point de donnée venant se placer au-dessus de la ligne droite sera affecté par le classifieur correspondant à la classe 1. De même, tout nouveau point de donnée venant se placer en dessous de la ligne droite sera affecté par le classifieur correspondant à la classe 0.

Les deux modèles présentent des frontières de décision très semblables. Remarquez que tous les deux se trompent dans le classement de deux des points. Par défaut, ces modèles appliquent une régularisation L2, exactement comme le fait la régression ridge.

Pour `LogisticRegression` comme pour `LinearSVC`, il existe un paramètre d'équilibre, ou de compromis, qui détermine la force de la régularisation et qui est appelé `C`. Plus `C` est élevé, et *moins* il y a de régularisation. En d'autres termes, lorsque vous utilisez une valeur élevée pour le paramètre C, `LogisticRegression` et `LinearSVC` essaient d'ajuster le jeu d'apprentissage le mieux possible, tandis qu'avec de petites valeurs de `C`, les modèles mettent l'accent sur le fait de trouver des coefficients (`w`) proches de zéro.

Il y a un autre aspect intéressant quant à la manière dont le paramètre `C` fonctionne. L'utilisation de petites valeurs pour `C` fait que l'algorithme va essayer d'ajuster la « majorité » des points de données, tandis qu'avec des valeurs plus élevées, il va mettre l'accent sur l'importance pour chaque point de donnée individuel d'être classifié correctement. La Figure 2.16 illustre cela en utilisant `LinearSVC` :

**In[40]:**

```
mglearn.plots.plot_linear_svc_regularization()
```

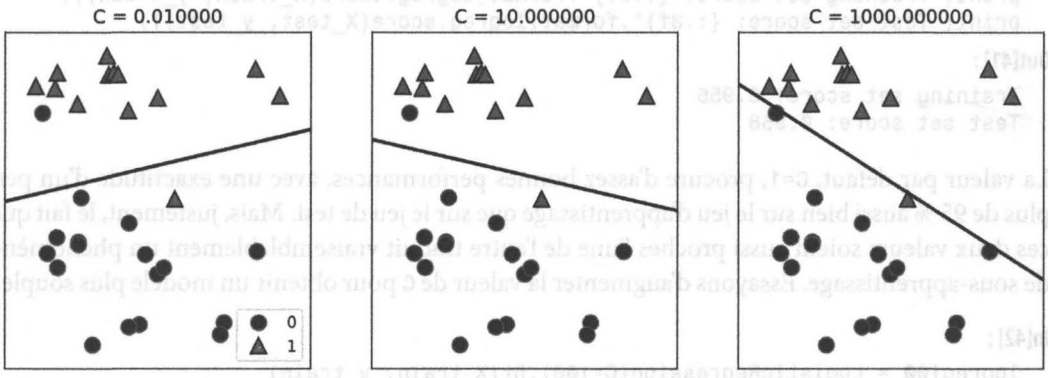



*Figure 2.16 : Frontières de décision avec une classification linéaire SVM pour le jeu de données* `forge` *avec différentes valeurs de* `C`.

Sur la gauche, nous avons une très petite valeur de `C` correspondant à une forte régularisation. La plupart des points de la classe 0 se trouvent en bas, et la plupart des points de la classe 1 sont en haut. Ce modèle fortement régularisé choisit une ligne relativement horizontale, et deux points sont mal classifiés. Au centre, `C` est relativement haut, et le modèle se concentre davantage sur les deux échantillons mal classifiés, ce qui fait basculer l'inclinaison de la frontière de décision. Finalement, sur le tracé de droite, la valeur de `C` est très grande, et la frontière de décision est encore plus inclinée, ce qui permet maintenant de bien classifier tous les points de la classe 0. Mais un des points de la classe 1 reste mal classifié, et il est d'ailleurs impossible de classifier cor-

rectement tous les points de ce jeu de données en utilisant une ligne droite. Même si le modèle illustré sur la droite s'efforce de classifier correctement tous les points, il ne peut pas arriver à capturer correctement la disposition générale des classes. En d'autres termes, il y a probablement ici un problème de surapprentissage.

Comme dans le cas de la régression, les modèles linéaires de classification peuvent sembler très restrictifs dans le cas d'espaces possédant peu de dimensions, en autorisant seulement des frontières de décision qui sont des droites ou des plans. Mais, là encore, les modèles linéaires de classification deviennent très puissants lorsque le nombre de dimensions devient beaucoup plus grand, et se protéger du surapprentissage est un enjeu de plus en plus important au fur et à mesure que le nombre de caractéristiques augmente.

Analysons plus en détail le comportement de `LogisticRegression` avec le jeu de données Breast Cancer :

**In[41]:**

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg.score(X_test, y_test)))
```

**Out[41]:**

```
Training set score: 0.955
Test set score: 0.958
```

La valeur par défaut, `C=1`, procure d'assez bonnes performances, avec une exactitude d'un peu plus de 95 % aussi bien sur le jeu d'apprentissage que sur le jeu de test. Mais, justement, le fait que ces deux valeurs soient aussi proches l'une de l'autre traduit vraisemblablement un phénomène de sous-apprentissage. Essayons d'augmenter la valeur de `C` pour obtenir un modèle plus souple :

**In[42]:**

```
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg100.score(X_test, y_test)))
```

**Out[42]:**

```
Training set score: 0.972
Test set score: 0.965
```

Avec `C=100`, nous améliorons l'exactitude du jeu d'apprentissage, et légèrement aussi celle du jeu de test. Cela confirme donc notre intuition qu'un modèle plus complexe devrait être plus efficace.

Nous pouvons aussi examiner ce qui se passe si nous utilisons un modèle encore plus régularisé en remplaçant la valeur par défaut (`C=1`) par `C=0.01` :

**In[43]:**

```
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg001.score(X_test, y_test)))
```

**Out[43]:**

```
Training set score: 0.934
Test set score: 0.930
```

Comme on pouvait s'y attendre, lorsqu'on se déplace vers la gauche le long de l'échelle illustrée sur la Figure 2.1 avec un modèle déjà en sous-apprentissage, l'exactitude décroît par rapport aux paramètres par défaut, et ce aussi bien pour le jeu d'apprentissage que pour le jeu de test.

Finalement, regardons les coefficients entraînés par les modèles avec les trois valeurs différentes données au paramètre de régularisation `C` (voir la Figure 2.17) :

**In[44]:**

```
plt.plot(logreg.coef_.T, 'o', label="C=1")
plt.plot(logreg100.coef_.T, '^', label="C=100")
plt.plot(logreg001.coef_.T, 'v', label="C=0.001")
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
xlims = plt.xlim()
plt.hlines(0, xlims[0], xlims[1])
plt.xlim(xlims)
plt.ylim(-5, 5)
plt.xlabel("Feature")
plt.ylabel("Coefficient magnitude")
plt.legend()
```



Du fait que `LogisticRegression` applique par défaut une régularisation L2, le résultat présente des ressemblances avec celui produit par la régression ridge sur la Figure 2.12. Une régularisation croissante pousse de plus en plus les coefficients vers zéro, bien qu'ils ne deviennent jamais nuls. En inspectant le tracé de plus près, nous pouvons aussi voir un effet intéressant pour le troisième coefficient, « mean perimeter » (périmètre moyen). Pour `C=100` et `C=1`, le coefficient est négatif, mais il devient positif pour `C=0.01`, avec une amplitude même encore plus grande que pour `C=1`. En interprétant un modèle tel que celui-ci, on pourrait penser que le coefficient nous indique à quelle classe une caractéristique est associée. Par exemple, on pourrait supposer que la caractéristique « texture error » est associée à un échantillon qui soit celui d'une tumeur maligne. Cependant, le changement de signe dans le coefficient « mean perimeter » signifie que, selon le modèle observé, une valeur élevée pour « mean perimeter » pourrait être prise comme indicateur d'un échantillon « benign » ou « malignant ». Cela illustre le fait que les interprétations des coefficients dans les modèles linéaires devraient toujours être réalisées avec une certaine prudence.

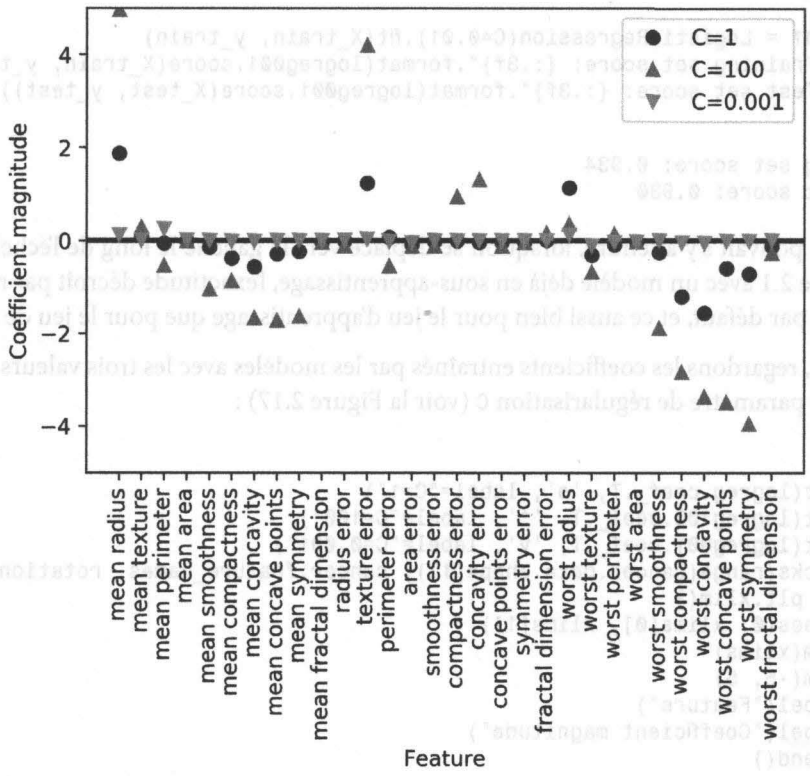



*Figure 2.17 : Coefficients entraînés par la régression logistique pour le jeu de données Breast Cancer avec différentes valeurs du paramètre* `C`.

Si nous désirons obtenir un modèle plus interprétable, utiliser la régularisation L1 pourrait aider, car elle limite le modèle afin qu'il n'utilise que quelques caractéristiques. Voici ce que donnent les calculs d'exactitude de la classification et le tracé des coefficients pour une régularisation L1 (voir la Figure 2.18) :

**In[45] :**

```
for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):
    lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
    print("Training accuracy of l1 logreg with C={:.3f}: {:.2f}".format(
          C, lr_l1.score(X_train, y_train)))
    print("Test accuracy of l1 logreg with C={:.3f}: {:.2f}".format(
          C, lr_l1.score(X_test, y_test)))
    plt.plot(lr_l1.coef_.T, marker, label="C={:.3f}".format(C))

plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
xlims = plt.xlim()
plt.hlines(0, xlims[0], xlims[1])
plt.xlim(xlims)
```

```
plt.xlabel("Feature")
plt.ylabel("Coefficient magnitude")

plt.ylim(-5, 5)
plt.legend(loc=3)
```

**Out[45]:**

```
Training accuracy of l1 logreg with C=0.001: 0.91
Test accuracy of l1 logreg with C=0.001: 0.92
Training accuracy of l1 logreg with C=1.000: 0.96
Test accuracy of l1 logreg with C=1.000: 0.96
Training accuracy of l1 logreg with C=100.000: 0.99
Test accuracy of l1 logreg with C=100.000: 0.98
```

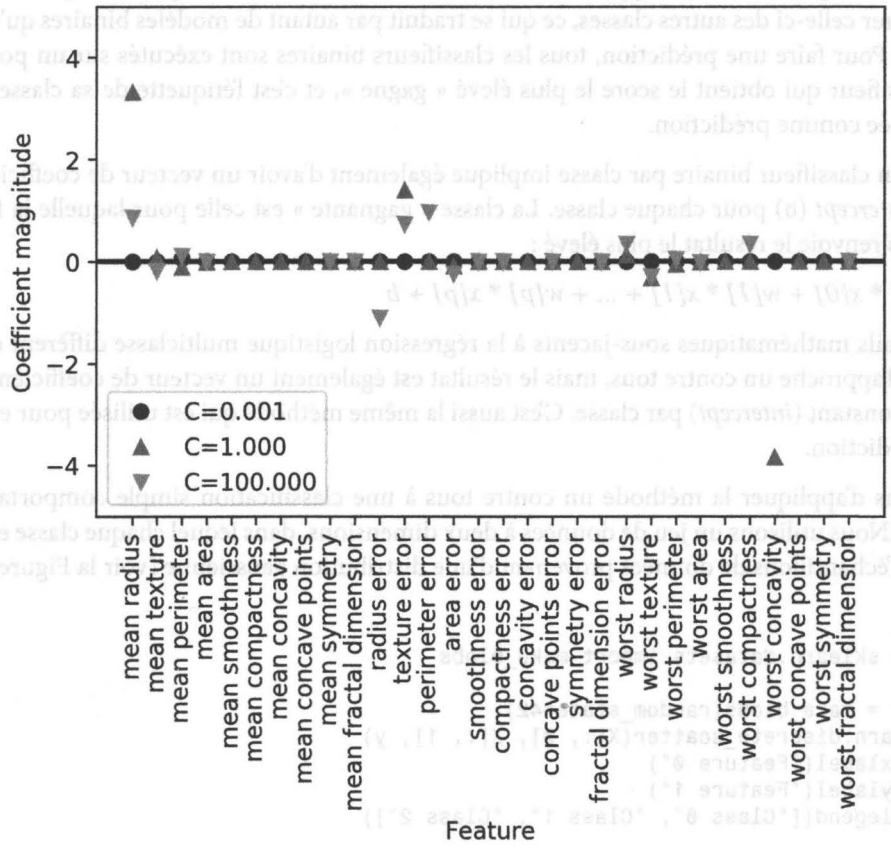



*Figure 2.18 : Coefficients entraînés par la régression logistique avec pénalisation L1 pour le jeu de données Breast Cancer avec différentes valeurs du paramètre* `C`.

Comme vous pouvez le voir, il existe de nombreux parallèles entre les modèles linéaires pour la classification binaire et pour la régression. Comme dans le cas de la régression, la principale différence entre les modèles est le paramètre `penalty`, qui influence la régularisation et l'utilisation par le modèle de l'ensemble des caractéristiques disponibles ou un sous-ensemble de celles-ci.

### Modèles linéaires pour la classification multiclasse

De nombreux modèles de classification linéaire ne s'appliquent qu'à des classifications binaires, et ne s'étendent donc pas naturellement au cas de la classification *multiclasse* (à l'exception de la régression logistique). Une technique courante servant à élargir l'application d'un algorithme de classification binaire à une classification multiclasse est l'approche *un contre tous* (*One vs All* ou OvA en abrégé). Dans cette approche, un modèle binaire est entraîné pour chaque classe afin de séparer celle-ci des autres classes, ce qui se traduit par autant de modèles binaires qu'il y a de classes. Pour faire une prédiction, tous les classifieurs binaires sont exécutés sur un point test. Le classifieur qui obtient le score le plus élevé « gagne », et c'est l'étiquette de sa classe qui est retournée comme prédiction.

Avoir un classifieur binaire par classe implique également d'avoir un vecteur de coefficients (`w`) et un *intercept* (`b`) pour chaque classe. La classe « gagnante » est celle pour laquelle la formule ci-après renvoie le résultat le plus élevé :

$$w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b$$

Les détails mathématiques sous-jacents à la régression logistique multiclasse diffèrent quelque peu de l'approche un contre tous, mais le résultat est également un vecteur de coefficients et un terme constant (*intercept*) par classe. C'est aussi la même méthode qui est utilisée pour effectuer une prédiction.

Essayons d'appliquer la méthode un contre tous à une classification simple comportant trois classes. Nous utilisons un jeu de données à deux dimensions, dans lequel chaque classe est composée d'échantillons de données provenant d'une distribution gaussienne (voir la Figure 2.19) :

**In[46]:**

```
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(["Class 0", "Class 1", "Class 2"])
```

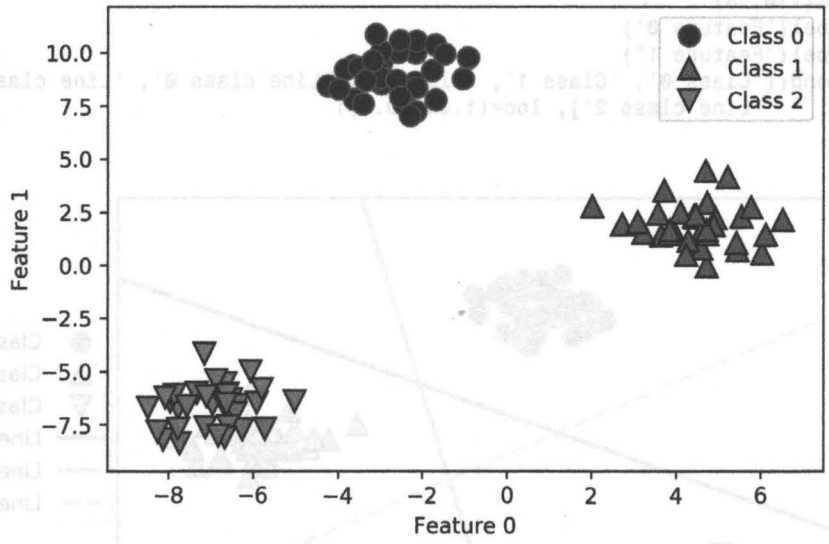



*Figure 2.19 : Jeu de données jouet à deux dimensions contenant trois classes.*

Nous allons maintenant entraîner le classifieur `LinearSVC` sur le jeu de données :

**In[47]:**

```
linear_svm = LinearSVC().fit(X, y)
print("Coefficient shape: ", linear_svm.coef_.shape)
print("Intercept shape: ", linear_svm.intercept_.shape)
```

**Out[47]:**

```
Coefficient shape:  (3, 2)
Intercept shape:  (3,)
```

Nous voyons que la forme de `coef_` est de `(3, 2)`, ce qui signifie que chaque ligne de `coef_` contient le vecteur de coefficients pour une des trois classes, et que chaque colonne contient la valeur de coefficient pour une caractéristique particulière (il y en a deux dans ce jeu de données spécifique). De l'autre côté, `intercept_` est maintenant un tableau à une dimension qui enregistre la valeur du terme constant, *intercept*, pour chaque classe.

Essayons maintenant de visualiser les lignes produites par les trois classifieurs binaires (voir la Figure 2.20) :

**In[48]:**

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                  mglearn.cm3.colors):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
```

```
plt.xlim(-10, 8)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',
            'Line class 2'], loc=(1.01, 0.3))
```

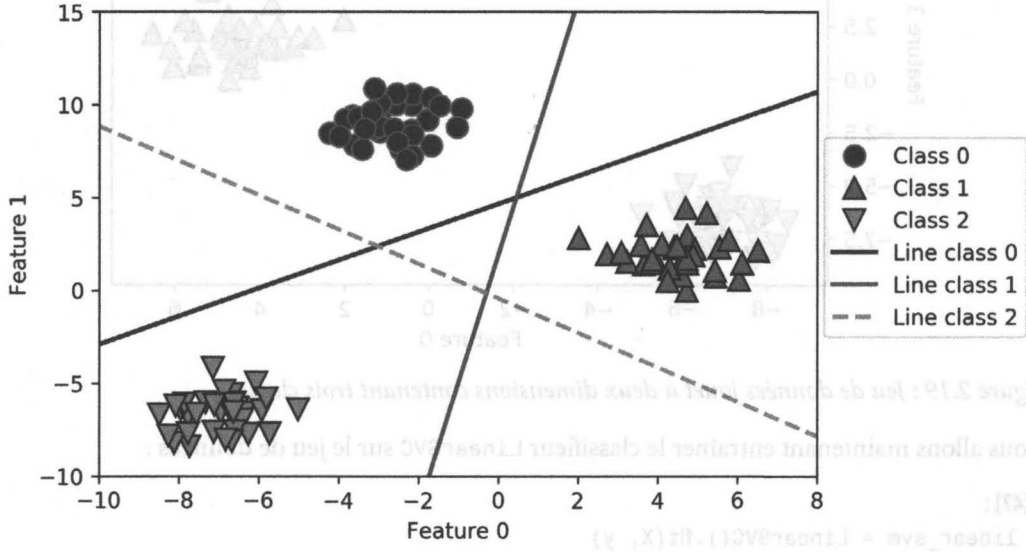



*Figure 2.20 : Frontières de décision entraînées par les trois classifieurs un contre tous.*

Vous pouvez voir que tous les points appartenant à la classe 0 dans les données d'apprentissage se trouvent au-dessus de la ligne correspondant à cette classe 0, ce qui signifie qu'ils sont du côté « classe 0 » de ce classifieur binaire. Les points de la classe 0 se trouvent au-dessus de la ligne correspondant à classe 2, ce qui signifie qu'ils sont classifiés comme « tous » par le classifieur binaire pour la classe 2. Les points qui appartiennent à la classe 0 se trouvent à gauche de la ligne qui correspond à la classe 1, ce qui signifie que le classifieur pour la classe 1 les classifie aussi en tant que « tous ». Par conséquent, tout point de cette zone sera classifié comme appartenant à la classe 0 par le classifieur final (le résultat de la formule de confiance pour la classification 0, exposée plus haut, est plus grande que zéro, tandis qu'elle est inférieure à zéro pour les deux autres classes).

Mais qu'y a-t-il à dire sur le triangle qui se trouve au milieu du tracé ? Les trois classifieurs binaires définissent les points qui se trouvent ici comme faisant partie du « reste ». Dans ce cas, à quelle classe un point serait-il assigné ? La réponse est à celle qui a la valeur la plus élevée pour la formule de classification : la classe de la ligne la plus proche.

L'exemple qui suit montre les prédictions pour toutes les régions de l'espace 2D (voir la Figure 2.21) :

In[49]:

```
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                  mglearn.cm3.colors):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',
            'Line class 2'], loc=(1.01, 0.3))
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

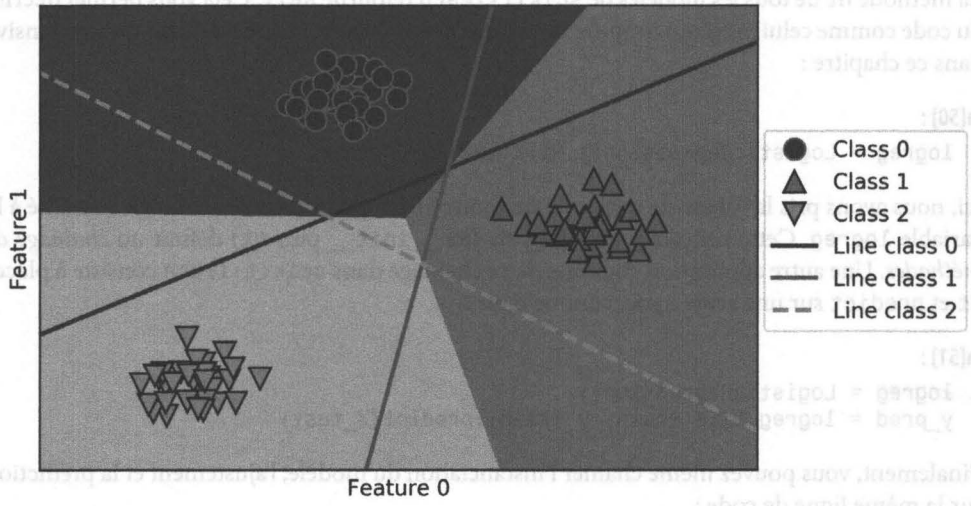



*Figure 2.21 : Frontières de décision multiclasse dérivées des classifieurs un contre tous.*

### Forces, faiblesses et paramètres

Le principal paramètre pour les modèles linéaires est le paramètre de régularisation, appelé `alpha` dans les modèles de régression et `C` dans les modèles `LinearSVC` et `LogisticRegression`. De grandes valeurs pour `alpha` ou de petites valeurs pour `C` signifient des modèles simples. En particulier, pour les modèles de régression, ajuster ces paramètres est assez important. En général, `C` et `alpha` sont recherchés sur une échelle logarithmique. L'autre décision que vous avez à prendre concerne l'utilisation d'une régularisation L1 ou d'une régularisation L2. Si vous supposez que seule une petite partie des caractéristiques est réellement importante, vous devriez choisir L1. Sinon, conservez la régularisation par défaut, L2. L1 peut également être utile s'il faut que le modèle soit facilement interprétable. Du fait que L1 ne se sert que de quelques caractéristiques, il est plus facile d'expliquer lesquelles sont importantes pour le modèle, et quel effet elles produisent.

Les modèles linéaires sont très rapides en termes d'entraînement comme de prédiction. Ils sont capables de s'adapter à des très vastes jeux de données, et travaillent bien avec des données éparses. Si vos données se comptent en centaines de milliers ou en millions d'échantillons, vous pourriez voir ce que donne l'option `solver='sag'` de `LogisticRegression` et de `ridge`. Elle peut accélérer les opérations avec des jeux de données de grande taille. Parmi les autres options, mentionnons les classes `SGDClassifier` et `SGDRegressor` qui implémentent des versions encore plus évolutives des modèles linéaires décrits ici.

### Chaînage de méthodes

La méthode `fit` de tous les modèles de `scikit-learn` retourne `self`. Cela vous permet d'écrire du code comme celui présenté ci-après, forme que nous avons déjà utilisée de manière extensive dans ce chapitre :

**In[50]:**

```
logreg = LogisticRegression().fit(X_train, y_train)
```

Ici, nous avons pris la valeur de retour de `fit` (donc `self`) pour affecter le modèle entraîné à la variable `logreg`. Cette concaténation d'appels (ici `__init__` puis `fit`) définit un *chaînage de méthodes*. Une autre application courante de ce chaînage dans `scikit-learn` consiste à placer `fit` et `predict` sur une seule ligne, comme dans :

**In[51]:**

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

Finalement, vous pouvez même chaîner l'instanciation du modèle, l'ajustement et la prédiction sur la même ligne de code :

**In[52]:**

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

Cependant, cette variante très condensée n'est pas idéale. Il se passe beaucoup de choses sur cette ligne unique, et donc ce code devient plus difficile à lire, à comprendre et éventuellement à corriger. De plus, ce modèle de régression logistique n'est pas enregistré dans une variable, et nous ne pourrons donc pas l'inspecter ou l'utiliser plus tard pour effectuer des prédictions sur d'autres données.

Une autre force des modèles linéaires, c'est qu'ils rendent relativement facile à comprendre le processus de prédiction en partant des formules que nous avons vues plus haut pour la régression et la classification. Par contre, il n'est malheureusement souvent pas aussi simple de se figurer pourquoi les coefficients se comportent comme ils le font. Cela est particulièrement vrai si votre jeu de données comporte des caractéristiques hautement corrélées. Dans ce cas, les coefficients peuvent être réellement difficiles à interpréter.

Les modèles linéaires sont souvent efficaces lorsque le nombre de caractéristiques est grand en comparaison du nombre d'échantillons. Ils sont aussi souvent utilisés avec de grands jeux de données, simplement parce qu'il n'est pas possible d'entraîner d'autres modèles. Dans des espaces de faible dimension, en revanche, d'autres modèles pourraient procurer de meilleures performances de prédiction. Nous y reviendrons plus loin dans ce chapitre lorsque nous aborderons le sujet des *séparateurs à vaste marge à noyau*.

## Classifieurs bayésiens naïfs

Les classifieurs bayésiens naïfs (ou classifieurs naïfs bayésiens) sont une famille de classifieurs qui sont assez semblables aux modèles linéaires discutés dans la section précédente. Cependant, ils tendent à permettre des apprentissages encore plus rapides. Le prix à payer pour cette efficacité est que les modèles bayésiens naïfs fournissent souvent des performances de généralisation légèrement moins bonnes (ou pires selon le cas) qu'avec des classifieurs linéaires tels que `LogisticRegression` et `LinearSVC`.

La raison pour laquelle ces modèles sont si efficaces est qu'ils déterminent les paramètres en analysant individuellement les caractéristiques, et en élaborant leurs règles de classification à partir de ces caractéristiques individuelles. Il y a trois sortes de classifieurs bayésiens naïfs implémentés dans `scikit-learn` : `GaussianNB`, `BernoulliNB` et `MultinomialNB`. `GaussianNB` peut être appliqué à des données continues quelconques. `BernoulliNB` suppose que les données sont binaires, tandis que `MultinomialNB` s'applique à des données qui sont des dénombrements de quelque chose (autrement dit, chaque caractéristique représente un entier qui compte une certaine propriété, comme le nombre d'occurrences d'un mot dans une phrase). Les variantes `BernoulliNB` et `MultinomialNB` sont essentiellement utilisées pour la classification de données textuelles.

La classifieur `BernoulliNB` compte combien de fois chaque caractéristique de chaque classe n'est pas nulle. Il est plus facile de comprendre cette notion en partant d'un exemple :

**In[53]:**

```
X = np.array([[0, 1, 0, 1],
              [1, 0, 1, 1],
              [0, 0, 0, 1],
              [1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

Ici, nous avons quatre points de données, chacun comportant quatre caractéristiques binaires. Il y a deux classes, 0 et 1. Pour la classe 0 (le premier et le troisième point de donnée), la première caractéristique est zéro deux fois, et non nulle zéro fois. La seconde caractéristique est zéro une fois et non nulle une fois, et ainsi de suite. Les mêmes comptes sont ensuite effectués pour les points de données de la seconde classe. Dénombrer les entrées non nulles pour chaque classe se présente ainsi :

In[54]:

```
counts = {}
for label in np.unique(y):
    counts[label] = X[y == label].sum(axis=0)
print("Feature counts:\n{}".format(counts))
```

Out[54]:

```
Feature counts:
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

Les deux autres modèles bayésiens naïfs, `MultinomialNB` et `GaussianNB`, sont légèrement différents en ce qu'ils traitent d'autres types de statistiques. `MultinomialNB` prend en compte la valeur moyenne de chaque caractéristique pour chaque classe, tandis que `GaussianNB` enregistre la valeur moyenne ainsi que la déviation standard de chaque caractéristique pour chaque classe.

Pour effectuer une prédiction, un point de donnée est comparé aux statistiques obtenues pour chacune des classes. La prédiction retenue correspond alors à la meilleure correspondance de classe trouvée. De manière intéressante, cela conduit aussi bien pour `MultinomialNB` et `BernoulliNB` à une formule de prédiction qui est semblable à celle des modèles linéaires (voir plus haut dans ce chapitre). Malheureusement, l'attribut `coef_` pour les modèles bayésiens naïfs a une signification quelque peu différente de ce qui se passe dans les modèles linéaires, dans le sens où ici `coef_` n'est pas le même que `w`.

### Forces, faiblesses et paramètres

`MultinomialNB` et `BernoulliNB` ont un paramètre, `alpha`, qui contrôle la complexité du modèle. Ce paramètre se traduit par un « lissage » des statistiques. Une valeur élevée de ce paramètre signifie davantage de lissage, et donc des modèles moins complexes. La valeur d'`alpha` n'est pas critique en termes de performances de l'algorithme. Cependant, déterminer une valeur bien ajustée permet d'améliorer quelque peu l'exactitude.

`GaussianNB` est essentiellement utilisé avec des données de très grande dimension, tandis que les deux autres variantes bayésiennes naïves sont largement employées pour des données textuelles. L'algorithme `MultinomialNB` donne généralement de meilleurs résultats que `BernoulliNB`, en particulier avec des jeux de données qui possèdent un nombre relativement grand de caractéristiques non nulles (par exemple, de gros documents).

Les modèles bayésiens naïfs partagent nombre de forces et de faiblesses avec les modèles linéaires. Ils sont très rapides pour l'entraînement et la prédiction, et la procédure d'entraînement est facile à comprendre. Ils travaillent également très bien avec des données éparses possédant un grand nombre de dimensions. Ce sont donc d'excellents modèles de référence, et ils sont souvent utilisés avec de très grands jeux de données, là où entraîner un modèle linéaire pourrait prendre trop de temps.

## Arbres de décision

Les arbres de décision sont des modèles largement employés pour des tâches de classification et de régression. Pour l'essentiel, ils se construisent sous la forme d'une hiérarchie de questions de type si/sinon qui amène à une décision.

Ces questions sont semblables à celles auxquelles vous pourriez avoir à répondre dans un jeu. Imaginez que vous vouliez trouver des animaux parmi quatre possibilités : ours, faucons, pingouins et dauphins. Votre objectif consiste à trouver la bonne réponse en posant le plus petit nombre possible de questions si/sinon. Vous pourriez commencer par demander si l'animal a des plumes. Cela vous permet déjà de restreindre à deux le champ des possibilités. Si la réponse est *oui*, vous pouvez poser une autre question qui vous permette de faire la différence entre faucons et pingouins. Il suffit par exemple de demander si l'animal peut voler. Si la réponse était *non*, votre choix est maintenant limité aux dauphins et aux ours. Il vous suffit alors de demander si l'animal a des nageoires pour arriver au résultat voulu.

Cette série de questions pourrait être exprimée sous la forme d'un arbre de décision, comme sur la Figure 2.22 :

**In[55]:**

```
mglearn.plots.plot_animal_tree()
```

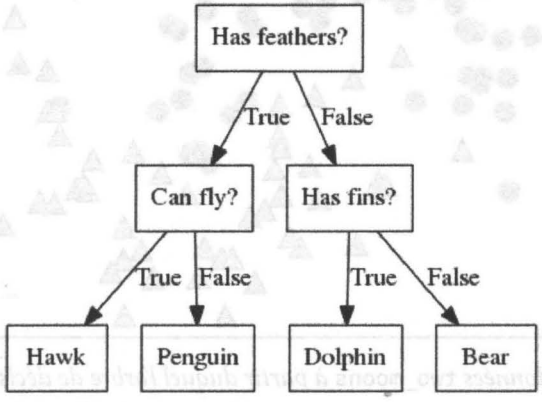



*Figure 2.22 : Un arbre pour se décider entre plusieurs animaux.*



Si vous utilisez Anaconda, par exemple, vous risquez d'obtenir une erreur du fait d'une non-reconnaissance de Graphviz. Voyez alors la page principale de Graphviz (http://www.graphviz.org/Download..php) puis faites une recherche sur *Anaconda graphviz install* (par exemple), ce qui devrait notamment vous diriger vers la page `https://anaconda.org/conda-forge/python-graphviz`.

Sur cette illustration, chaque nœud de l'arbre représente une question ou un nœud terminal (également appelé une *feuille*) qui contient la réponse. Les flèches servent à connecter une certaine question à une nouvelle question en fonction de la réponse obtenue.

Dans le langage de l'apprentissage automatique, nous avons construit un modèle permettant d'effectuer une distinction entre quatre classes d'animaux (faucons, pingouins, dauphins et ours) en utilisant les trois caractéristiques « a des plumes », « peut voler » et « a des nageoires ». Au lieu de dessiner ces modèles à la main, nous pouvons les entraîner à partir de données en utilisant un apprentissage supervisé.

### Construire des arbres de décision

Voyons comment se déroule le processus de construction d'un arbre de décision à partir du jeu de données illustré sur la Figure 2.23. Ce jeu de données est formé à partir de deux formes en demi-lune, chaque classe comportant 75 points de données. Nous désignerons ce jeu sous le nom `two_moons`.

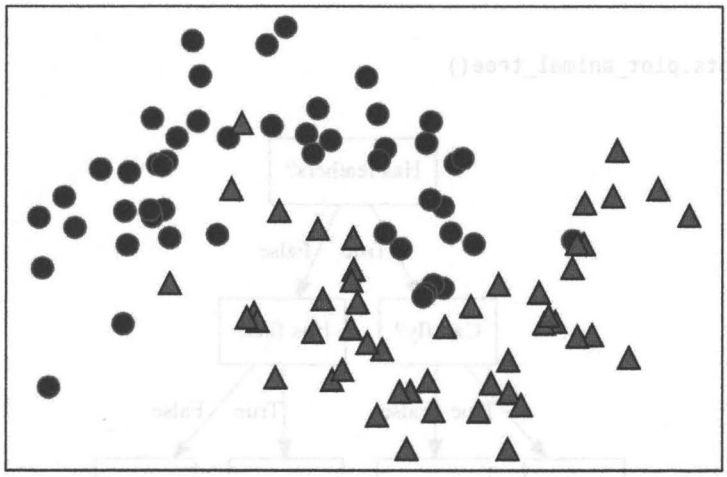


*Figure 2.23 : Le jeu de données* `two_moons` *à partir duquel l'arbre de décision va être construit.*

Entraîner un arbre de décision consiste à former la séquence de questions si/sinon qui nous conduise le plus rapidement possible à la bonne réponse. En termes d'apprentissage automatique, ces questions sont appelées des *tests* (qu'il ne faut bien sûr pas confondre avec les jeux de test, c'est-à-dire les données qui servent à vérifier la manière dont un modèle se généralise). En général, les données ne sont pas des valeurs binaires de type oui/non, comme dans notre exemple d'animaux, mais sont plutôt représentées sous forme de caractéristiques continues, comme c'est le cas avec le jeu de données 2D représenté sur la Figure 2.23. Les tests qui sont

effectués sur des données continues peuvent être formalisés sous la forme de questions du type « est-ce que la caractéristique *i* est plus grande que la valeur *a* ? ».

Pour construire un arbre, l'algorithme recherche tous les tests possibles et trouve celui qui donne le plus d'informations sur la variable cible. La Figure 2.24 illustre le premier test choisi. Partager le jeu de données horizontalement à la position `x[1] = 0.0596` produit le plus d'informations, autrement sépare au mieux les points de la classe 0 de ceux de la classe 1. Le nœud le plus haut (ou *racine*) représente le jeu de données en entier, qui est formé ici de 75 points appartenant à la classe 0 et 75 points appartenant à la classe 1. Le partage est réalisé en testant la réponse à la « question » `x[1] <= 0.0596` et est indiqué par une ligne noire. Si le test est vrai, un point est affecté au nœud de gauche, qui contient deux points provenant de la classe 0 et 32 de la classe 1. Dans le cas contraire, le point est attribué au nœud de droite, qui contient 48 points provenant de la classe 0 et 18 de la classe 1. Ces deux nœuds correspondent aux régions du haut et du bas de la Figure 2.24. Même si ce premier partage a fait du bon travail pour séparer les deux classes, la région inférieure contient encore des points qui appartiennent à la classe 0, celle du haut des points qui appartiennent à la classe 1. Nous pouvons construire un modèle plus exact en répétant le processus de recherche du meilleur test dans les deux régions. La Figure 2.25 montre que le prochain partage le plus efficace dans ce cas est basé sur `x[0]`.

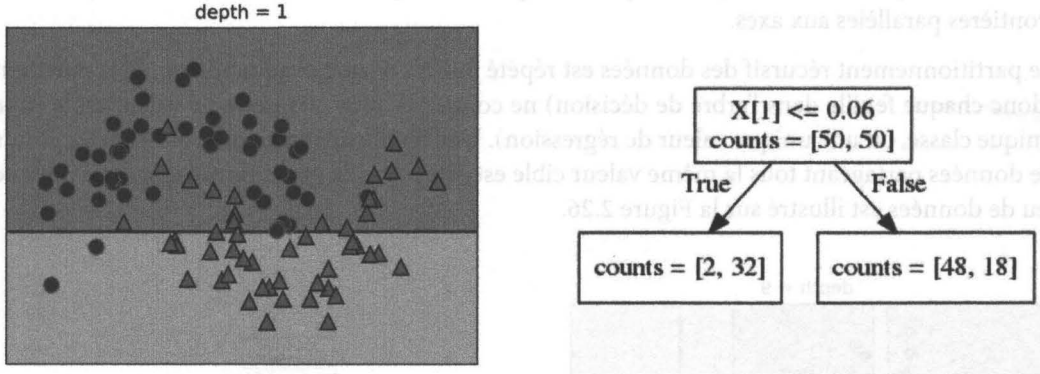



*Figure 2.24 : Frontière de décision avec une profondeur de 1 (à gauche) et l'arbre correspondant (à droite).*

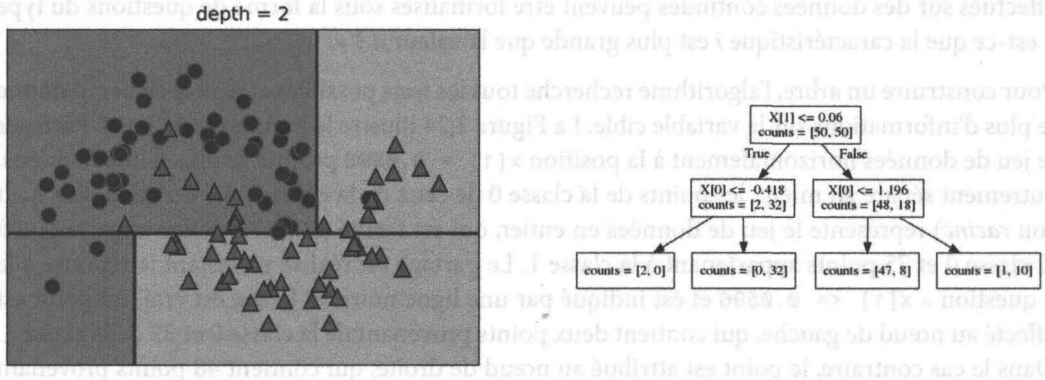



*Figure 2.25 : Frontière de décision avec une profondeur de 2 (à gauche) et l'arbre correspondant (à droite).*

Ce processus récursif produit un arbre de décisions binaire, chaque nœud y contenant un test. D'un autre côté, vous pouvez aussi envisager chaque test comme servant à partager le long d'un axe la partie des données qui sont actuellement considérées. Cela conduit alors à une vue de l'algorithme comme un outil construisant une partition hiérarchique. Du fait que chaque test ne concerne qu'une seule caractéristique, les régions de la partition résultante ont toujours des frontières parallèles aux axes.

Le partitionnement récursif des données est répété jusqu'à ce que chaque région de la partition (donc chaque feuille dans l'arbre de décision) ne contienne plus qu'une seule valeur cible (une unique classe, ou une unique valeur de régression). Une feuille de l'arbre qui contient des points de données partageant tous la même valeur cible est dite *pure*. Le partitionnement final pour ce jeu de données est illustré sur la Figure 2.26.

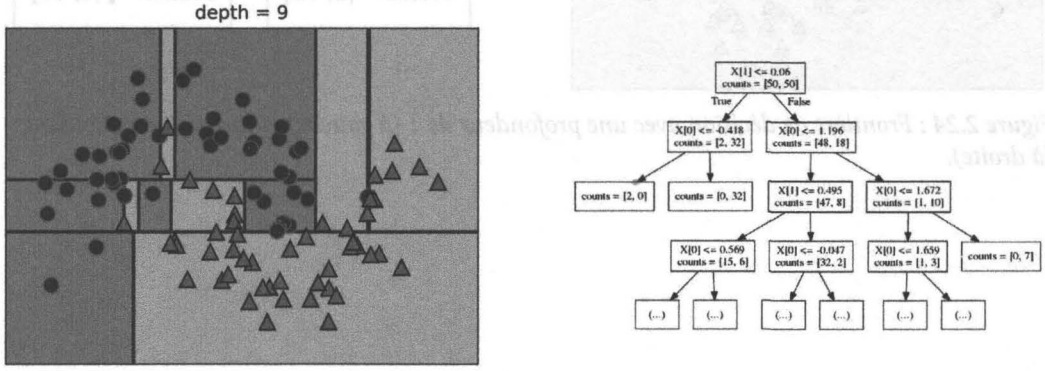



*Figure 2.26 : Frontière de décision avec une profondeur de 9 (à gauche) et une partie de l'arbre correspondant (à droite), l'arbre complet étant assez grand et difficile à visualiser.*

Une prédiction sur un nouveau point de donnée est réalisée en vérifiant dans quelle région de la partition se trouve ce point, et en prédisant alors la cible majoritaire pour cette région (ou la cible unique dans le cas d'une feuille pure). La région peut être trouvée en traversant l'arbre en partant de la racine, puis en se déplaçant vers la gauche ou vers la droite selon que le test est réussi ou non.

Il est aussi possible d'utiliser des arbres pour des tâches de régression en utilisant exactement la même technique. Pour effectuer une prédiction, nous traversons l'arbre en suivant le résultat des tests sur chaque nœud afin de trouver la feuille où va venir se poser le nouveau point de donnée. La sortie obtenue pour ce point est la cible moyenne des points d'entraînement pour cette feuille.

#### Contrôler la complexité des arbres de décision

Typiquement, construire un arbre comme nous venons de le voir et continuer jusqu'à ce que toutes les feuilles soient pures conduit à des modèles extrêmement complexes, ainsi qu'à un fort surapprentissage des données. La présence de feuilles pures signifie qu'un arbre a une exactitude de 100 % pour le jeu d'apprentissage. Chaque point de donnée de ce jeu se trouve sur une feuille qui a la classe majoritaire correcte. Le surapprentissage est visible sur la gauche de la Figure 2.26. Vous pouvez en effet voir les régions déterminées comme appartenant à la classe 1 au milieu de tous les points appartenant à la classe 0. D'un autre côté, il y a une petite bande sur la droite qui est prédite comme relevant de la classe 0 mais qui se trouve au milieu de points appartenant à la classe 1. Ce n'est pas ainsi que l'on pourrait imaginer la frontière de décision à déterminer, et celle-ci se concentre beaucoup sur des points aberrants qui sont très éloignés des autres points de la même classe.

Deux stratégies courantes servent à éviter le surapprentissage : stopper en amont la création de l'arbre (ce que l'on appelle aussi le *pre-pruning*, ou *pré-élagage*), ou construire l'arbre mais en supprimant ou regroupant ensuite les nœuds qui contiennent peu d'information, et conduisent donc au surapprentissage (ce que l'on appelle alors *post-pruning*, ou *post-élagage*). Parmi les critères possibles pour le pré-élagage, on peut mentionner la limitation de la profondeur maximale de l'arbre, celle du nombre maximal de feuilles, ou encore le fait d'imposer un nombre minimum de points dans un nœud pour qu'il soit partageable.

Dans `scikit-learn` les arbres de décision sont implémentés dans les classes `DecisionTreeRegressor` et `DecisionTreeClassifier`. De plus, `scikit-learn` implémente uniquement le pré-élagage, pas le post-élagage.

Regardons plus en détail l'effet du pré-élagage sur le jeu de données Breast Cancer. Comme toujours, nous importons le jeu de données, et nous le partageons entre un jeu d'apprentissage et un jeu de test. Nous construisons ensuite un modèle en utilisant la méthode par défaut consistant à développer totalement l'arbre (c'est-à-dire jusqu'à ce que toutes ses feuilles soient pures). Nous fixons la valeur `random_state` à zéro, ce qui correspond au mode de partage par défaut entre jeu d'apprentissage et jeu de test :

**In[56]:**

```
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

**Out[56]:**

```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```

Comme attendu, l'exactitude sur le jeu d'apprentissage est de 100 %. Puisque les feuilles sont pures, l'arbre a suffisamment grandi pour qu'il soit capable de mémoriser la totalité des étiquettes sur le jeu d'apprentissage. Par contre, l'exactitude sur le jeu de test est un peu plus mauvaise qu'avec les modèles linéaires étudiés plus haut, où cette exactitude était d'environ 95 %.

Si nous ne restreignons pas la profondeur d'un arbre de décision, celui-ci peut devenir arbitrairement profond et complexe. Les arbres qui ne sont pas élagués sont de ce fait sujets au surapprentissage et ne se généralisent pas bien avec de nouvelles données. Appliquons donc maintenant un pré-élagage à l'arbre, qui va par conséquent cesser de croître avant de s'ajuster parfaitement aux données d'apprentissage. Une option consiste à stopper cette croissance lorsqu'une certaine profondeur a été atteinte. Ici, nous allons définir la variable `max_depth` avec la valeur 4, ce qui signifie que seules quatre questions consécutives peuvent être posées (reportez-vous plus haut aux Figures 2.24 et 2.26). Limiter la taille de l'arbre réduit le surapprentissage. De ce fait, l'exactitude diminue pour le jeu d'apprentissage, mais elle s'améliore pour le jeu de test :

**In[57]:**

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

**Out[57]:**

```
Accuracy on training set: 0.988
Accuracy on test set: 0.951
```

### Analyser les arbres de décision

Nous pouvons visualiser l'arbre en nous servant de la fonction `export_graphviz` du module `tree`. Celle-ci écrit les données dans un fichier au format *.dot*, qui est un format textuel servant à enregistrer des graphes. Nous définissons une option pour colorer les nœuds afin de refléter la

classe majoritaire dans chacun d'entre eux, et nous passons les noms des classes et des caractéristiques afin d'étiqueter correctement notre arbre :

**In[58]:**

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["malignant", "benign"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

Nous pouvons maintenant lire ce fichier et le visualiser en utilisant le module `graphviz`, comme l'illustre la Figure 2.27 (mais vous pourriez tout aussi bien faire appel à n'importe quel autre programme sachant lire et interpréter les fichiers *.dot*) :

**In[59]:**

```
import graphviz

with open("tree.dot") as f:
    dot_graph = f.read()
display(graphviz.Source(dot_graph))
```



En cas d'erreur, essayez d'ajouter la ligne `from IPython.display import display` au début de ce code.

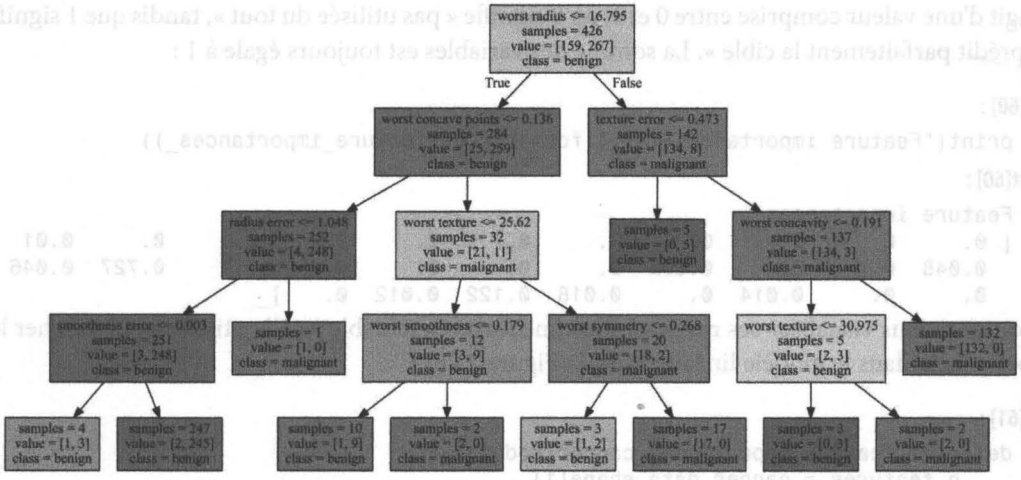



*Figure 2.27 : Visualisation de l'arbre de décision construit sur le jeu de données Breast Cancer.*

La visualisation de l'arbre fournit une vue approfondie sur la manière dont l'algorithme effectue des prédictions, et c'est aussi un bon exemple d'algorithme d'apprentissage automatique qui peut facilement être expliqué à des non-experts. Cependant, même en se limitant à une profondeur sur quatre niveaux de nœuds, comme ci-dessus, l'arbre peut devenir quelque peu « écrasant ». Et les arbres encore plus développés (il n'est pas rare d'atteindre dix niveaux) sont

évidemment encore bien plus difficiles à se représenter concrètement. Une méthode d'inspection des arbres qui peut être utile consiste à trouver quel est le chemin suivi par la majorité des données. Sur la Figure 2.27, la valeur `sample` affichée dans chaque nœud indique le nombre d'échantillons concernés par ce nœud, tandis que `value` en spécifie la quantité par classe. En suivant les branches vers la droite, nous voyons que `worst radius > 16.795` produit un nœud qui contient seulement 8 échantillons bénins, mais 134 échantillons malins. Le reste de ce côté de l'arbre utilise ensuite quelques distinctions plus fines pour traiter ces huit échantillons bénins restants. Sur les 142 échantillons trouvés à droite du premier niveau de nœuds, pratiquement tous (132) se retrouvent dans la feuille la plus à droite.

Si nous allons vers la gauche de l'arbre, nous constatons que, partant de `worst radius <= 16.795`, nous en arrivons à 25 échantillons malins et 259 échantillons bénins. Pratiquement tous les échantillons bénins se retrouvent dans la seconde feuille à gauche, la plupart des autres feuilles ne contenant que quelques échantillons.

### Importance des caractéristiques dans les arbres

Au lieu de regarder la totalité de l'arbre, ce qui peut être très contraignant, il y a quelques propriétés utiles que nous pouvons déduire pour résumer le fonctionnement de l'arbre. La plus courante est l'importance des caractéristiques, ou encore des variables (*feature importance*). Elle mesure l'importance de chaque caractéristique dans la décision à laquelle conduit un arbre. Il s'agit d'une valeur comprise entre 0 et 1, où 0 signifie « pas utilisée du tout », tandis que 1 signifie « prédit parfaitement la cible ». La somme des variables est toujours égale à 1 :

**In[60]:**

```
print("Feature importances:\n{}".format(tree.feature_importances_))
```

**Out[60]:**

```
Feature importances:
[ 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.01
  0.048  0.     0.     0.002  0.     0.     0.     0.     0.727  0.046
  0.     0.     0.014  0.     0.018  0.122  0.012  0.     ]
```

Nous pouvons visualiser ces résultats sous une forme semblable à celle utilisée pour afficher les coefficients dans le modèle linéaire (voir la Figure 2.28) :

**In[61]:**

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")
    plt.ylim(-1, n_features)

plot_feature_importances_cancer(tree)
```

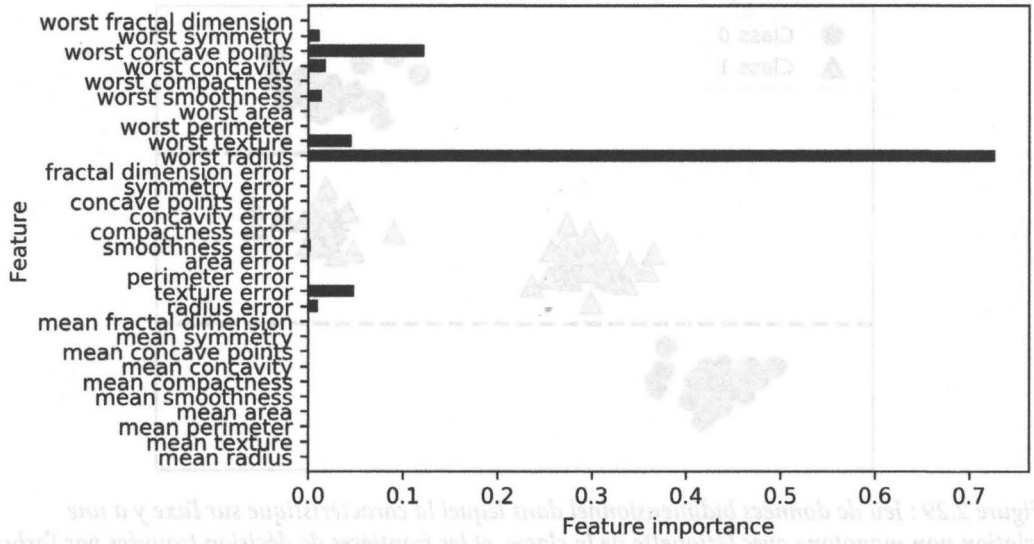



*Figure 2.28 : Importance des variables calculée à partir d'un arbre de décision entraîné sur le jeu de données Breast Cancer.*

Nous voyons ici que la caractéristique utilisée au sommet de l'arbre, *worst radius*, est de très loin la plus importante. Cela confirme notre observation quant à l'analyse de l'arbre sur le fait que le premier niveau sépare déjà plutôt bien les deux classes.

Cependant, si une caractéristique a une valeur faible dans `feature_importance`, cela ne signifie pas pour autant qu'elle ne véhicule pas d'information. Cela veut seulement dire qu'elle n'a pas été choisie par l'arbre, probablement parce qu'une autre caractéristique encode la même information.

Par contraste avec les coefficients que nous avons vus dans les modèles linéaires, les variables retenues ici sont toujours positives et ne signalent pas en soi une classe particulière. Par exemple, l'arbre construit à partir de notre jeu de données Breast Cancer nous dit que *worst radius* est important, mais pas si cette mesure est indicative d'un échantillon correspondant à une tumeur bénigne ou maligne. En fait, il est tout à fait possible qu'il n'existe pas de relation aussi simple entre caractéristiques et classe, comme le montre l'exemple suivant (voir les Figures 2.29 et 2.30) :

**In[62] :**

```
tree = mglearn.plots.plot_tree_not_monotone()
display(tree)
```

**Out[62]**

```
Feature importances: [ 0.  1.]
```

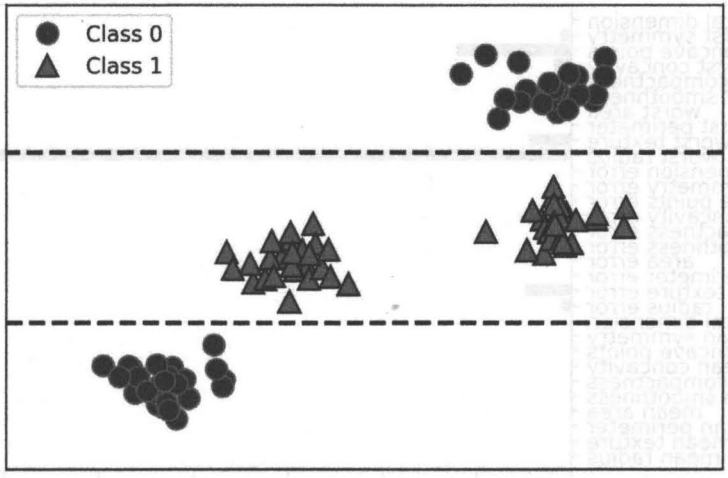



*Figure 2.29 : Jeu de données bidimensionnel dans lequel la caractéristique sur l'axe y a une relation non monotone avec l'étiquette de la classe, et les frontières de décision trouvées par l'arbre.*

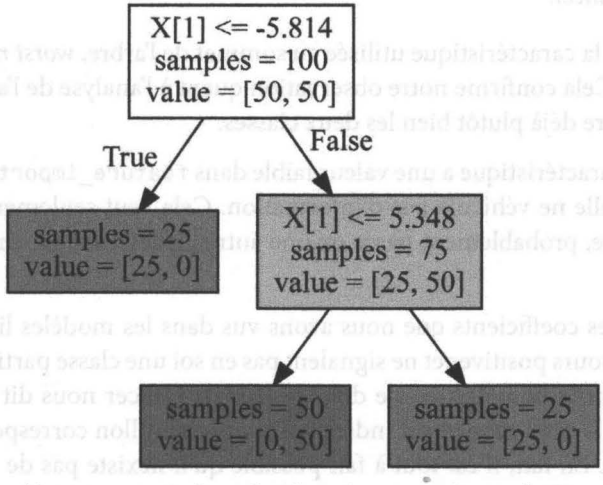



*Figure 2.30 : Arbre de décision entraîné par les données montrées sur la Figure 2.29.*

Le tracé montre un jeu de données avec deux caractéristiques et deux classes. Ici, toutes les informations sont contenues dans `X[1]`, et `X[0]` n'est pas du tout utilisé. Mais la relation entre `X[1]` et la classe de sortie n'est pas monotone, ce qui fait que nous ne pouvons pas dire « une valeur élevée de `X[1]` signifie la classe 0, et une valeur basse la classe 1 », ou l'inverse.

Même si nous focalisons ici notre discussion sur les arbres de décision pour la classification, tout ce que nous avons vu est également vrai dans le cas des arbres de décision pour la régression, qui sont implémentés dans `DecisionTreeRegressor`. L'utilisation et l'analyse des arbres de décision

sont très semblables dans ce cas à ce qui se passe pour les arbres de classification. Cependant, nous voulons attirer l'attention sur une propriété particulière de l'utilisation des modèles basés sur des arbres de décision dans le cas de la régression. En effet, `DecisionTreeRegressor` (et tous les modèles de régression basés sur des arbres) n'est pas capable d'*extrapolation*, autrement dit de faire des prédictions en dehors du champ des données d'apprentissage.

Pour voir cette question plus en détail, nous allons utiliser un jeu de données concernant l'évolution dans le temps du prix des mémoires vives (RAM). La Figure 2.31 montre le jeu de données, avec la date en abscisse et le prix d'un mégaoctet de RAM en ordonnée :

**In[63]:**

```
import os
ram_prices = pd.read_csv(os.path.join(mglearn.datasets.DATA_PATH, "ram_price.
  csv"))

plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Year")
plt.ylabel("Price in $/Mbyte")
```

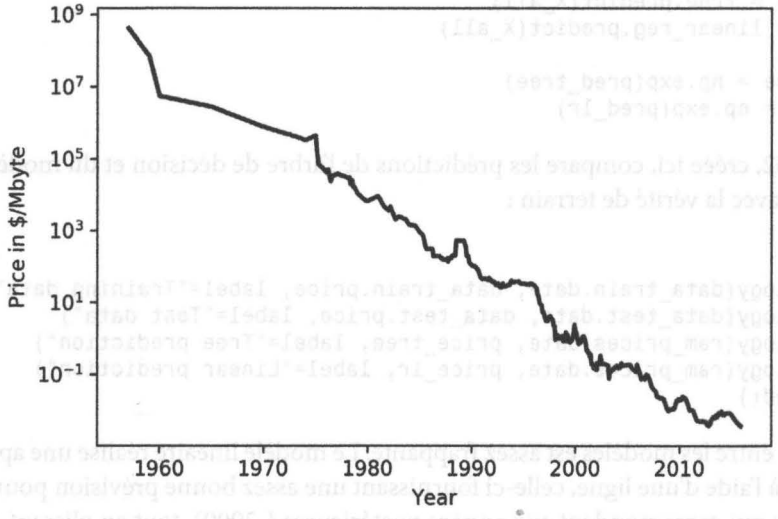



*Figure 2.31 : Évolution du prix de la RAM représenté sur une échelle logarithmique.*

Notez l'échelle logarithmique sur l'axe y. Dans ce cas, la relation semble être assez linéaire et devrait donc être relativement facile à prédire en dehors de quelques petits écarts.

Nous allons effectuer une prédiction pour la période qui se trouve après les années 2000 en utilisant l'historique des données jusqu'à cette date, cette information étant notre seule caractéristique. Nous allons comparer deux modèles simples : `DecisionTreeRegressor` et `LinearRegression`. Nous normalisons les prix en utilisant une échelle logarithmique de manière à obtenir une rela-

tion relativement linéaire. Cela ne fait pas une grande différence pour `DecisionTreeRegressor`, mais cette différence peut être beaucoup plus forte pour `LinearRegression` (nous y reviendrons plus en détail dans le Chapitre 4). Une fois les modèles entraînés et les prédictions effectuées, nous annulons la transformation logarithmique *via* un calcul exponentiel. Nous effectuons ici des prédictions sur la totalité du jeu de données pour des raisons de visualisation, mais nous ne devrions considérer que le jeu de test pour une évaluation quantitative :

**In[64] :**

```
from sklearn.tree import DecisionTreeRegressor
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]

X_train = data_train.date[:, np.newaxis]
y_train = np.log(data_train.price)

tree = DecisionTreeRegressor().fit(X_train, y_train)
linear_reg = LinearRegression().fit(X_train, y_train)

X_all = ram_prices.date[:, np.newaxis]

pred_tree = tree.predict(X_all)
pred_lr = linear_reg.predict(X_all)

price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```

La Figure 2.32, créée ici, compare les prédictions de l'arbre de décision et du modèle de régression linéaire avec la vérité de terrain :

**In[65] :**

```
plt.semilogy(data_train.date, data_train.price, label="Training data")
plt.semilogy(data_test.date, data_test.price, label="Test data")
plt.semilogy(ram_prices.date, price_tree, label="Tree prediction")
plt.semilogy(ram_prices.date, price_lr, label="Linear prediction")
plt.legend()
```

La différence entre les modèles est assez frappante. Le modèle linéaire réalise une approximation des données à l'aide d'une ligne, celle-ci fournissant une assez bonne prévision pour les données de test (celles qui correspondent aux années postérieures à 2000), tout en glissant sur certaines variations plus fines dans les jeux d'apprentissage comme de test. De l'autre côté, l'arbre de décision effectue des prévisions parfaites sur le jeu d'apprentissage. Du fait que nous n'avons pas restreint la complexité de l'arbre, celui-ci a entraîné la totalité du jeu de données. Cependant, une fois que nous quittons la plage pour laquelle le modèle possède des données, celui-ci continue simplement à prédire le dernier point connu. L'arbre n'a pas la capacité de générer des réponses « nouvelles », en dehors de celles qui ont été vues dans les données d'apprentissage. Cette lacune s'applique à tous les modèles basés sur des arbres (il est en effet possible de faire de très bonnes prévisions avec des modèles basés sur des arbres, par exemple en essayant de prédire si un prix

va monter ou descendre – le but de cet exemple n'était pas de montrer que les arbres sont un mauvais modèle pour les séries chronologiques, mais d'illustrer une propriété particulière de la façon dont les arbres font des prédictions).

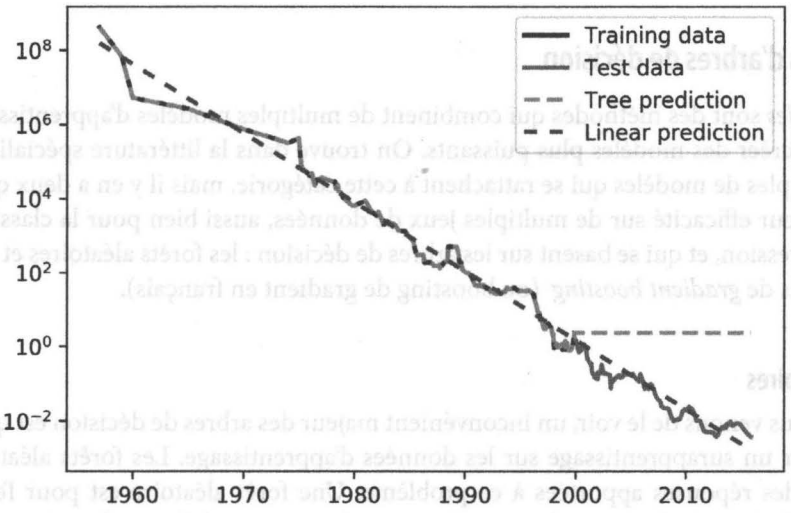



*Figure 2.32 : Comparaisons entre prédictions faites par un modèle linéaire et prédictions faites par un arbre de régression sur le jeu de données des prix de RAM.*

### Forces, faiblesses et paramètres

Comme nous l'avons vu plus haut, les paramètres qui contrôlent la complexité des arbres de décision sont en particulier ceux du pré-élagage, qui stoppe la construction de l'arbre avant qu'il ne soit pleinement développé. En règle générale, opter pour une stratégie de pré-élagage, en choisissant `max_depth`, `max_leaf_nodes` ou `min_samples_leaf` est suffisant pour échapper au surapprentissage.

Les arbres de décision présentent deux avantages sur nombre des algorithmes abordés précédemment. Tout d'abord, le modèle peut facilement être visualisé et compris par des non-experts, du moins pour des arbres de taille raisonnable. De plus, ces algorithmes sont totalement invariants quant au recalibrage des données. Du fait que chaque caractéristique est traitée séparément, et que les possibles partages de données ne font pas intervenir des questions d'échelle, aucun prétraitement, comme la normalisation ou la standardisation des caractéristiques, n'est nécessaire ici. En particulier, les arbres de décision fonctionnent bien lorsque vous avez des caractéristiques qui ont des échelles totalement différentes, ou encore un mélange de caractéristiques binaires et continues.

Le principal inconvénient des arbres de décision est que, même avec un pré-élagage, ils tendent au surapprentissage et fournissent de piètres performances de généralisation. C'est pourquoi, dans la plupart des applications, on utilise, comme nous allons le voir maintenant, des méthodes dites d'ensemble à la place d'arbres de décision uniques.

## Ensembles d'arbres de décision

Les *ensembles* sont des méthodes qui combinent de multiples modèles d'apprentissage automatique pour créer des modèles plus puissants. On trouve dans la littérature spécialisée de nombreux exemples de modèles qui se rattachent à cette catégorie, mais il y en a deux qui ont fait la preuve de leur efficacité sur de multiples jeux de données, aussi bien pour la classification que pour la régression, et qui se basent sur les arbres de décision : les forêts aléatoires et les arbres de décision dits de *gradient boosting* (ou boosting de gradient en français).

### Forêts aléatoires

Comme nous venons de le voir, un inconvénient majeur des arbres de décision est qu'ils tendent à provoquer un surapprentissage sur les données d'apprentissage. Les forêts aléatoires constituent une des réponses apportées à ce problème. Une forêt aléatoire est pour l'essentiel une collection d'arbres de décision, chaque arbre étant légèrement différent des autres. L'idée sous-jacente, c'est que chaque arbre pourrait faire un travail de prédiction relativement bon, mais qu'il va probablement provoquer un surapprentissage sur une partie des données. Si nous construisons de nombreux arbres, chacun travaillant correctement de son côté tout en amenant un surapprentissage différent des autres, nous pouvons réduire le volume de ce surapprentissage en prenant une forme de moyenne des résultats. Cette réduction du surapprentissage, tout en retenant la puissance prédictive des arbres, peut d'ailleurs être démontrée en faisant appel à des mathématiques rigoureuses.

Pour implémenter cette stratégie, nous avons besoin de construire de nombreux arbres de décision. Chaque arbre devrait faire un travail de prédiction acceptable, tout en étant différent des autres arbres. L'expression *forêt aléatoire* vient de cette multiplication des arbres ainsi que de l'injection d'une part aléatoire dans leur construction afin de s'assurer qu'ils sont tous différents. Il existe deux façons de s'y prendre pour introduire ces éléments de hasard dans la construction de la forêt : en sélectionnant les points de données utilisés pour construire un arbre, ou en sélectionnant les caractéristiques retenues pour chaque division. Voyons cela plus en détail.

#### Construire des forêts aléatoires

Pour construire un modèle de forêt aléatoire, vous avez besoin de choisir le nombre d'arbres que comportera cette forêt (c'est le paramètre `n_estimators` des classes `RandomForestRegressor` et `RandomForestClassifier`). Supposons que nous voulions construire dix arbres. Ceux-ci doivent être fabriqués de manière totalement indépendante les uns des autres, et l'algorithme doit donc

faire différents choix aléatoires pour chaque arbre afin que tous diffèrent. Pour construire un arbre, nous commençons par prendre ce que l'on appelle un *échantillon Bootstrap*. Autrement dit, à partir de nos `n_samples` points de données, nous effectuons un tirage aléatoire avec remise – ce qui signifie que le même échantillon peut être choisi de multiples fois – et ce `n_samples` fois (en anglais, un tirage avec remise se dit *bootstrapping*, la méthode décrite ici s'appelant *bootstrap aggregating*, que l'on contracte en *bagging*). Cela va créer un jeu de données de même taille que le jeu d'origine, mais où certains points de données seront répétés et d'autres absents (approximativement un tiers).

Pour illustrer cette méthode, supposons que nous voulions créer un échantillon bootstrap à partir de la liste `['a', 'b', 'c', 'd']`. Un tirage possible pourrait donner `['b', 'd', 'd', 'c']`, et un autre `['d', 'a', 'd', 'a']`.

Un arbre de décision est ensuite construit à partir de ce nouveau jeu de données. Cependant, l'algorithme décrit plus haut est légèrement modifié. Au lieu de rechercher le meilleur test pour chaque nœud, l'algorithme y sélectionne de manière aléatoire un sous-ensemble des caractéristiques, puis il recherche le meilleur test possible impliquant une de ces caractéristiques. Le nombre de caractéristiques qui sont sélectionnées est contrôlé par le paramètre `max_features`. Cette sélection d'un sous-ensemble de caractéristiques est répétée de manière séparée pour chaque nœud, et donc ainsi chaque nœud d'un arbre peut prendre une décision en utilisant un sous-ensemble différent de caractéristiques.

L'échantillon bootstrap fait en sorte que chaque arbre de décision de la forêt aléatoire soit construit à partir d'un jeu de données légèrement différent. Du fait en plus de la sélection de caractéristiques dans chaque nœud, chaque division dans chaque arbre opère sur un sous-ensemble différent de caractéristiques. Ces deux mécanismes conjugués assurent que tous les arbres de la forêt aléatoire sont différents.

Un paramètre critique dans ce processus est `max_features`. Si nous définissons `max_features` avec la valeur `n_features`, chaque partage peut prendre en compte toutes les caractéristiques du jeu de données, et aucun élément aléatoire n'est donc injecté dans la sélection des caractéristiques (hormis évidemment l'aspect aléatoire du tirage avec remise). Si nous définissons `max_features` avec la valeur 1, cela signifie que les partages n'ont pas de choix du tout quant à la caractéristique à tester, et que la seule possibilité est d'effectuer des recherches sur certains seuils de la caractéristique sélectionnée de manière aléatoire. Une valeur élevée de `max_features` signifie donc que les arbres de la forêt aléatoire seront assez semblables, et qu'ils seront capables d'ajuster les données facilement, en utilisant les caractéristiques les plus distinctives. Une valeur basse pour `max_features` signifie que les arbres de la forêt aléatoire seront assez différents, et que chaque arbre pourrait avoir besoin d'être très profond pour bien ajuster les données.

Pour effectuer une prédiction en utilisant une forêt aléatoire, l'algorithme commence par faire une prédiction pour chacun des arbres qui la composent. Dans le cas d'une régression, nous pouvons déterminer une moyenne de ces résultats pour obtenir notre prédiction finale. Dans le cas d'une classification, on utilise une stratégie dite de « vote souple » (*soft voting*). Cela signifie

que chaque algorithme fait une prédiction « souple », fournissant une probabilité pour chaque étiquette de sortie possible. Les probabilités prédites par tous les arbres sont alors moyennées, et la prédiction devient celle de la classe ayant la plus forte probabilité.

#### Analyser les forêts aléatoires

Essayons d'appliquer une forêt aléatoire comportant cinq arbres au jeu de données `two_moons` étudié plus haut :

**In[66] :**

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

Les arbres qui constituent la forêt aléatoire sont enregistrés dans l'attribut `estimator_`. Visualisons maintenant les frontières de décision pour chaque arbre, ainsi que leur prédiction agrégée telle qu'elle est faite par la forêt (voir la Figure 2.33) :

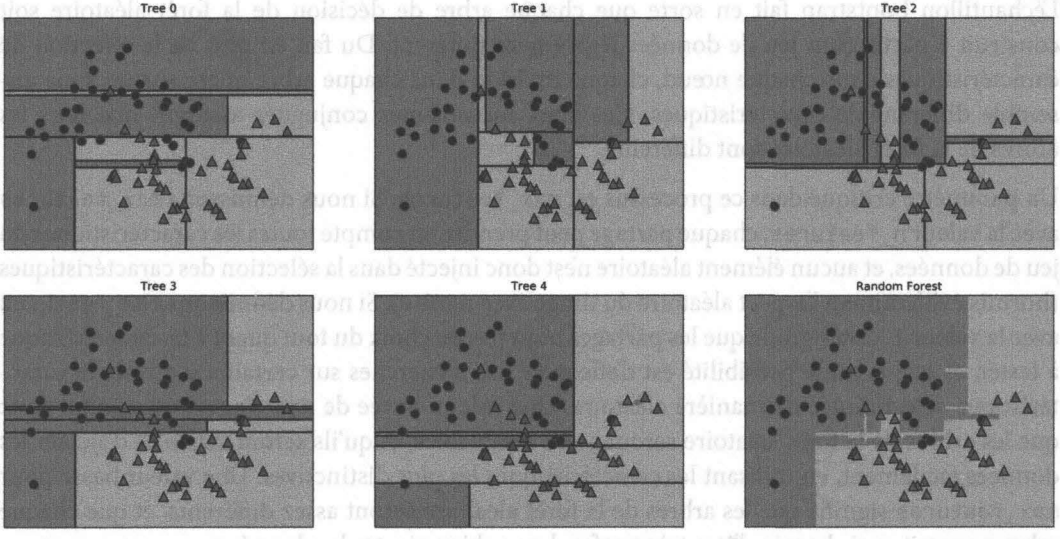



*Figure 2.33 : Les frontières de décision trouvées par cinq arbres produits aléatoirement, et la frontière de décision obtenue en faisant la moyenne de leurs probabilités prédites.*

**In[67]:**

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
                                alpha=.4)
axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_traIn[:, 0], X_traIn[:, 1], y_train)
```

Vous pouvez clairement voir que les frontières de décision sont plutôt différentes pour les cinq arbres. Chacun d'entre eux commet plusieurs erreurs, du fait que certains des points tracés ici ne se trouvent en fait pas dans le jeu d'apprentissage des arbres à cause de la méthode d'échantillonnage par tirage avec remise.

La forêt aléatoire est moins sujette au surapprentissage que chacun des arbres pris individuellement, et elle fournit une frontière de décision bien plus intuitive. Dans une application du monde réel, nous utiliserions bien plus d'arbres (souvent, des centaines ou des milliers), ce qui produirait des frontières encore plus lisses.

Prenons un autre exemple dans lequel nous allons appliquer une forêt aléatoire peuplée de cent arbres sur le jeu de données Breast Cancer :

**In[68]:**

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(forest.score(X_train, y_
    train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))
```

**Out[68]:**

```
Accuracy on training set: 1.000
Accuracy on test set: 0.972
```

La forêt aléatoire nous donne une exactitude de 97 %, ce qui est mieux qu'avec les modèles linéaires ou avec un arbre de décision unique, et sans avoir affiné quelque paramètre que ce soit. Nous pourrions bien sûr ajuster la valeur de `max_features`, ou appliquer un pré-élagage comme nous l'avons fait pour les arbres de décision seuls. Cependant, les paramètres par défaut des forêts aléatoires donnent déjà souvent de bons résultats.

Comme pour l'arbre de décision, la forêt aléatoire permet de mesurer les importances des variables, qui sont calculées en agrégeant les importances des variables de chacun des arbres de la forêt. Le résultat est plus fiable que celui fourni par un arbre unique, comme l'illustre la Figure 2.34 :

In[69]:

```
plot_feature_importances_cancer(forest)
```

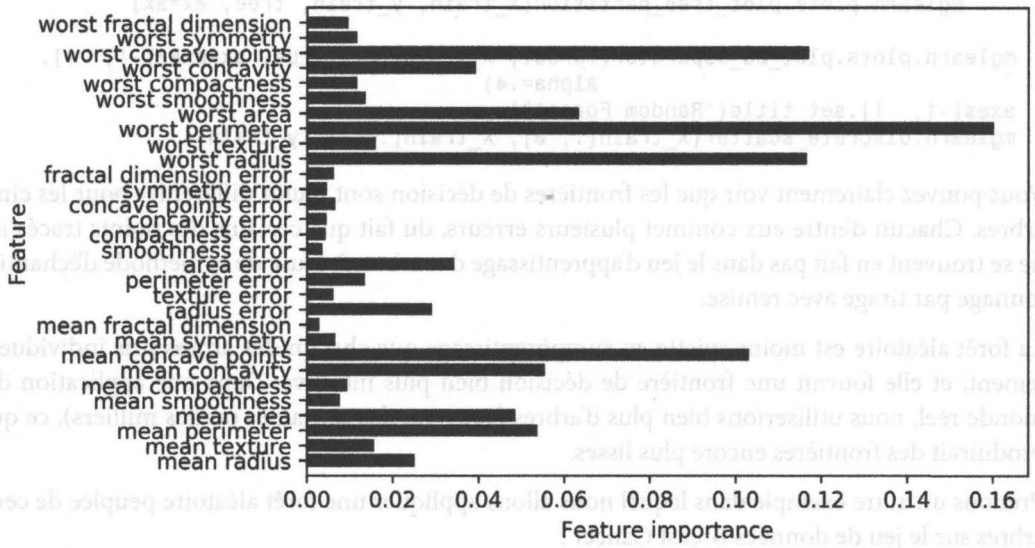



*Figure 2.34 : Importances des variables calculées depuis une forêt aléatoire construite à partir du jeu de données Breast Cancer.*

Comme vous pouvez le voir, la forêt aléatoire donne une importance non nulle à bien plus de caractéristiques que l'arbre seul. Comme notre arbre, la forêt aléatoire considère que la caractéristique « worst radius » est très importante, mais elle choisit « worst perimeter » comme étant globalement la plus porteuse d'informations. Le mode même de construction de la forêt aléatoire force l'algorithme à prendre en considération de nombreuses explications possibles, le résultat étant qu'elle capture une image des données bien plus large qu'un arbre seul.

#### Forces, faiblesses et paramètres

Les forêts aléatoires pour la régression et la classification font à l'heure actuelle partie des méthodes les plus largement utilisées pour l'apprentissage automatique. Elles sont très puissantes, fonctionnent souvent bien sans qu'il soit utile d'ajuster lourdement leurs paramètres, et ne nécessitent pas de recalibrage des données.

Pour l'essentiel, les forêts aléatoires partagent tous les avantages des arbres de décision, tout en compensant certaines de leurs déficiences. Une raison justifiant le fait de continuer à utiliser des arbres de décision est le besoin d'une représentation compacte du processus de prise de décision. Il est fondamentalement impossible d'interpréter en détail des dizaines ou des centaines d'arbres, et les arbres des forêts aléatoires ont tendance à être plus profonds que les arbres de décisions seuls (du fait de l'utilisation de sous-ensembles de caractéristiques). Par voie de consé-

quence, si vous avez besoin de synthétiser de manière visuelle le processus de prédiction pour des non-experts, un arbre de décision seul pourrait être un meilleur choix. Si la construction de forêts aléatoires à partir de jeux de données volumineux a un coût en termes de temps de calcul, cette procédure peut être parallélisée facilement sur de multiples cœurs de CPU. Si vous travaillez avec un ordinateur disposant d'un processeur multicœur (comme c'est le cas de nos machines modernes), vous pouvez faire appel au paramètre `n_jobs` pour ajuster le nombre de cœurs à utiliser. Cela permet d'améliorer les performances de façon linéaire (avec deux cœurs, l'entraînement de la forêt aléatoire sera deux fois plus rapide). En revanche, spécifier une valeur pour `n_jobs` qui soit plus grande que le nombre de cœurs n'aidera en rien. Cependant, il est possible de définir `n_jobs` avec la valeur -1 pour utiliser tous les cœurs disponibles dans l'ordinateur.

N'oubliez surtout pas que, par nature, les forêts aléatoires sont... aléatoires. Par conséquent, définir différents germes pour le générateur de nombres aléatoires (ou ne pas définir du tout `random_state`) peut changer considérablement le modèle qui est construit. Plus la forêt contient d'arbres, et plus elle sera robuste au regard du choix de `random_state`. Si vous voulez avoir des résultats reproductibles, il est donc important de fixer une valeur pour ce paramètre.

Les forêts aléatoires ont tendance à ne pas bien se comporter avec des données éparses et/ou ayant un grand nombre de dimensions, comme des données textuelles. Dans ce cas, les modèles linéaires semblent plus appropriés. Les forêts aléatoires fonctionnent par contre généralement bien avec de très grands jeux de données, et leur entraînement peut facilement être parallélisé sur de nombreux cœurs de CPU avec un ordinateur puissant. Cependant, elles exigent davantage de mémoire et sont plus lentes à entraîner et à prédire que les modèles linéaires. Si le temps et la mémoire sont importants dans une application, faire appel à un modèle linéaire plutôt qu'à une forêt aléatoire est une solution qui pourrait avoir du sens.

Les paramètres importants à ajuster sont `n_estimators`, `max_features` et possiblement les options de pré-élagage comme `max_depth`. Pour `n_estimators`, une valeur élevée est toujours préférable. Avec davantage d'arbres, vous obtenez un ensemble plus robuste en réduisant le surapprentissage. Cependant, cela implique aussi davantage de besoins en termes de mémoire et de temps d'entraînement. Une règle de base courante consiste à construire « autant que vous avez de temps et de mémoire pour le faire ».

Comme nous l'avons vu plus haut, `max_features` détermine l'aspect aléatoire de chaque arbre, et une plus petite valeur de ce paramètre réduit le surapprentissage. En général, il est conseillé de choisir les valeurs par défaut suivantes : `max_features=sqrt(n_features)` pour la classification, et `max_features=n_features` pour la régression. Ajouter `max_features` ou `max_leaf_nodes` peut parfois améliorer les performances. Cela peut également réduire de façon drastique les besoins en termes d'espace et de temps pour l'entraînement et la prédiction.

### Boosting d'arbres de régression

Le boosting d'arbres de régression (ou encore en anglais *gradient boosted regression trees*, en abrégé GBRT) est un autre ensemble de méthodes qui combinent de multiples arbres de décision afin de créer un modèle plus puissant. En dépit du mot « régression », ces modèles peuvent être utilisés aussi bien pour la régression que pour la classification. À la différence de l'approche utilisée pour les forêts aléatoires, le *gradient boosting* fonctionne en construisant des arbres de manière sérielle, chaque arbre essayant de corriger les erreurs faites par le précédent. Par défaut, il n'y a pas ici de part aléatoire, mais l'utilisation d'un fort pré-élagage. Les arbres créés avec le gradient boosting sont souvent très peu profonds, avec de un à cinq niveaux, ce qui donne un modèle plus petit en termes de mémoire et rend les prédictions plus rapides.

L'idée principale derrière cette technique est de combiner de multiples modèles simples (ce que l'on appelle ici des *mauvais élèves*, ou *weak learners*), comme des arbres peu profonds. Chaque arbre ne peut produire des prédictions correctes que sur une partie des données, et de plus en plus d'arbres sont ajoutés pour améliorer les performances de manière itérative.

Le gradient boosting est souvent le vainqueur dans les compétitions d'apprentissage automatique, et il est largement employé dans les applications du monde réel. Ce type d'algorithme est généralement un peu plus sensible à la configuration des paramètres que les forêts aléatoires, mais il peut aussi fournir une meilleure exactitude si ces paramètres sont définis correctement.

En plus du pré-élagage et du nombre d'arbres dans l'ensemble, un autre paramètre important ici est `learning_state`, qui contrôle le degré d'intensité avec lequel chaque arbre essaie de corriger les erreurs des arbres précédents. Une valeur élevée signifie que chaque arbre peut apporter davantage de corrections, ce qui permet donc des modèles plus complexes. Ajouter davantage d'arbres à l'ensemble, ce qui peut être réalisé en augmentant la valeur de `n_estimators`, accroît également la complexité du modèle, car celui-ci a plus de chances de corriger les erreurs sur le jeu d'apprentissage.

Voici un exemple d'utilisation de `GradientBoostingClassifier` avec le jeu de données Breast Cancer. Par défaut, nous avons cent arbres d'une profondeur maximale de trois niveaux et une contribution pour chaque arbre définie par `learning_state` de 0.1 (une valeur faible qui augmente le nombre d'arbres nécessaires pour ajuster le jeu d'apprentissage, mais qui permet généralement aux prédictions de mieux se généraliser) :

**In[70] :**

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

**Out[70]:**

```
Accuracy on training set: 1.000
Accuracy on test set: 0.958
```

Du fait que l'exactitude sur le jeu d'apprentissage est de 100 %, il y a vraisemblablement surapprentissage. Pour réduire ce phénomène, nous pourrions soit renforcer le pré-élagage en limitant la profondeur maximale des arbres, soit diminuer la valeur de `learning_state` :

**In[71]:**

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

**Out[71]:**

```
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

**In[72]:**

```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

**Out[72]:**

```
Accuracy on training set: 0.988
Accuracy on test set: 0.965
```

Les deux méthodes de réduction de la complexité du modèle diminuent comme attendu l'exactitude sur le jeu d'apprentissage. Dans ce cas, abaisser la profondeur maximale des arbres apporte une amélioration significative au modèle, tandis que diminuer la valeur de `learning_rate` n'améliore que légèrement les performances de généralisation.

Comme pour les autres modèles de décision basés sur des arbres, nous pouvons à nouveau visualiser l'importance des variables afin d'avoir un meilleur aperçu (voir la Figure 2.35). Du fait que nous avons utilisé cent arbres, il n'est pas du tout pratique de tous les inspecter, même si leur profondeur est réduite à un niveau :

**In[73]:**

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

plot_feature_importances_cancer(gbrt)
```

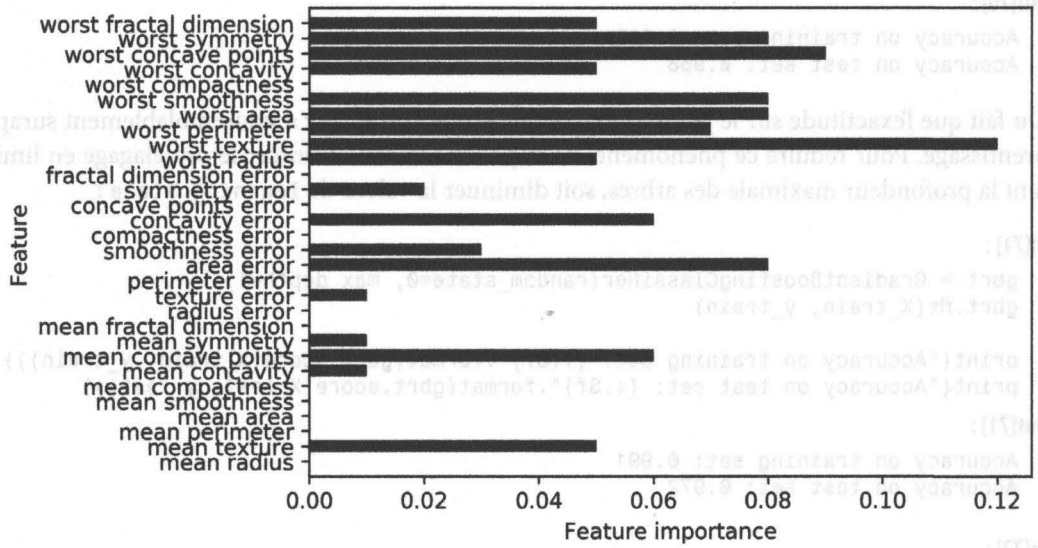



*Figure 2.35 : Importances des variables calculées à partir d'un classifieur de gradient boosting pour le jeu de données Breast Cancer.*

Vous pouvez constater que le résultat est assez semblable à ce que nous avons déjà vu pour les forêts aléatoires, à ceci près que plusieurs caractéristiques sont ici totalement ignorées.

Du fait que les deux types d'algorithmes, forêts aléatoires et gradient boosting, donnent de bons résultats sur des données de même nature, une approche courante consiste à essayer d'abord les forêts aléatoires. Si le temps de traitement est un critère essentiel, ou si le dernier pourcentage d'exactitude du modèle d'apprentissage automatique est important, passer au boosting d'arbres de régression aide souvent à obtenir ce que l'on souhaite.

Si vous voulez appliquer cette technique à un problème de grande ampleur, il pourrait être utile de s'intéresser au package `xgboost` et à son interface Python. En effet, au moment où ce livre est écrit, il est plus rapide (et parfois aussi plus facile à ajuster) que l'implémentation réalisée dans `scikit-learn` pour de nombreux jeux de données.

#### Forces, faiblesses et paramètres

Le boosting d'arbres de régression fait partie des modèles les plus puissants et les plus largement utilisés dans l'apprentissage supervisé. Le principal inconvénient, c'est que cet algorithme nécessite un ajustement précis des paramètres, et que, de plus, l'entraînement peut être long. Comme les autres modèles basés sur des arbres, l'algorithme fonctionne bien sans avoir besoin de recalibrer les données, et il s'applique également à des caractéristiques mélangeant valeurs binaires et continues. De même, dans l'autre sens, il ne s'applique pas bien à des données éparses avec de multiples dimensions.

Les principaux paramètres de ces modèles sont le nombre d'arbres, `n_estimators`, et `learning_rate`, qui contrôle le degré de correction des erreurs faites par les arbres précédents. Ces deux paramètres sont hautement interconnectés, une valeur basse pour `learning_rate` nécessitant en effet un plus grand nombre d'arbres pour construire un modèle ayant le même niveau de complexité. Et, à la différence des forêts aléatoires, où une valeur élevée pour `n_estimators` est toujours préférable, trop augmenter le nombre d'arbres conduit ici à un modèle plus complexe, et donc potentiellement à du surapprentissage. Une pratique courante consiste à ajuster la valeur de `n_estimators` en fonction des budgets alloués au temps et à la mémoire, puis à adapter `learning_rate` en conséquence.

Un autre paramètre important est `max_depth` (ou bien `max_leaf_nodes`) afin de réduire la complexité de chaque arbre. En règle générale, `max_depth` est défini avec une valeur très basse pour cet algorithme, en se limitant le plus souvent à un maximum de cinq divisions.

## SVM à noyau

Le prochain type de modèle supervisé que nous allons aborder est appelé *SVM à noyau*. Rappelons que le sigle SVM signifie *Support Vector Machines*, soit littéralement *machines à vecteurs de support*, mais que l'on traduit généralement par *séparateur à vaste marge* pour conserver les initiales. Nous avons exploré plus haut dans ce chapitre les séparateurs à vaste marge pour la classification linéaire. Les SVM à noyau en constituent une extension qui permet de former des modèles plus complexes qui ne sont pas simplement définis par des hyperplans dans l'espace d'entrée. Même si les SVM à noyau s'appliquent à la classification comme à la régression, nous nous en tiendrons uniquement au premier cas, tel qu'il est implémenté dans `SVC`. Des concepts similaires s'appliquent par ailleurs à la régression, l'implémentation se trouvant alors dans `SVR`. Même si les notions mathématiques qui sous-tendent les SVM à noyau vont bien au-delà de l'objet de ce livre, nous allons tout de même essayer dans ce qui suit de donner à comprendre l'idée générale de cette méthode (les matheux anglicistes pourront avec profit se tourner vers le Chapitre 12 de l'ouvrage de référence *The Elements of Statistical Learning*, de Hastie, Tibshirani et Friedman, disponible gratuitement à l'adresse `https://web.stanford.edu/~hastie/ElemStatLearn/`).

### Modèles linéaires et caractéristiques non linéaires

Comme vous l'avez vu sur la Figure 2.15, les modèles linéaires peuvent être assez limitatifs pour les espaces de faibles dimensions, car les lignes et les hyperplans ont une flexibilité limitée. Une façon de s'y prendre pour rendre un modèle linéaire plus souple consiste à y ajouter des caractéristiques supplémentaires, par exemple des interactions ou des variables polynomiales.

Revenons au jeu de données synthétiques que nous avons déjà utilisé plus haut dans ce chapitre (voir la Figure 2.36) :

In[74]:

```
X, y = make_blobs(centers=4, random_state=8)
y = y % 2

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

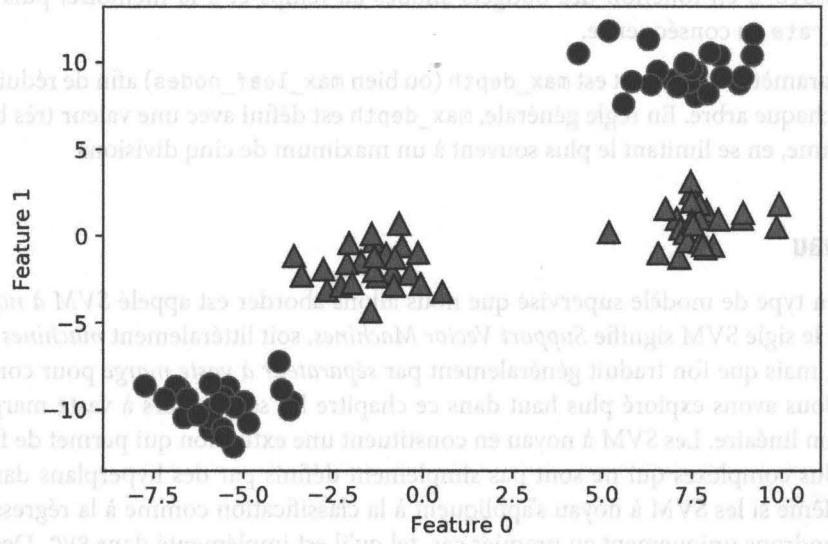



*Figure 2.36 : Jeu de données dans lequel les deux classes ne sont pas linéairement séparables.*

Un modèle linéaire pour la classification ne peut séparer les points qu'avec une ligne droite, et il ne ferait donc pas un très bon travail avec ce jeu de données (voir la Figure 2.37) :

In[75]:

```
from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

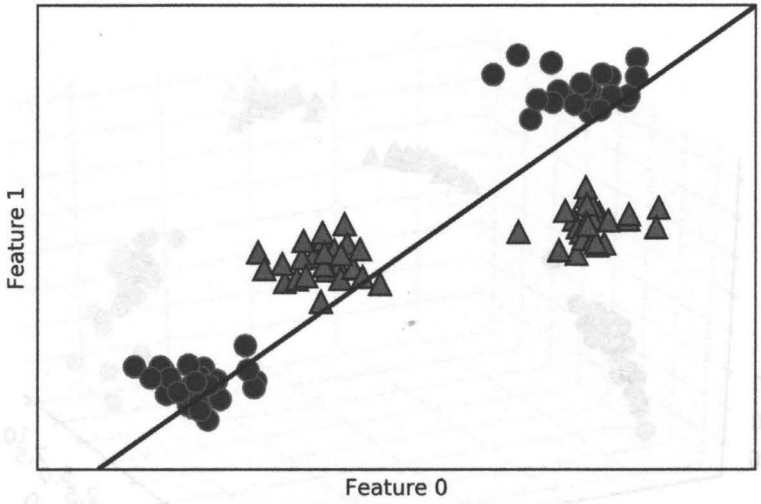



*Figure 2.37 : Frontière de décision trouvée par un SVM linéaire.*

Nous allons maintenant étendre l'ensemble des caractéristiques en ajoutant par exemple le carré de la seconde, soit `feature1 ** 2` (ce choix est pris simplement à titre d'expérimentation, et il n'a pas d'importance particulière). Au lieu de représenter chaque point de donnée en deux dimensions (selon deux axes), nous lui ajoutons une dimension, donc un axe, supplémentaire : `(feature0, feature1, Feature1 ** 2)`. Cette nouvelle représentation est illustrée par un nuage de points sur la Figure 2.38 :

**In[76]:**

```
X_new = np.hstack([X, X[:, 1:] ** 2])


from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
ax = Axes3D(figure, elev=-152, azim=-26)
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60, edgecolor='k')
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r',
           marker='^', cmap=mglearn.cm2, s=60, edgecolor='k')
ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1 ** 2")
```

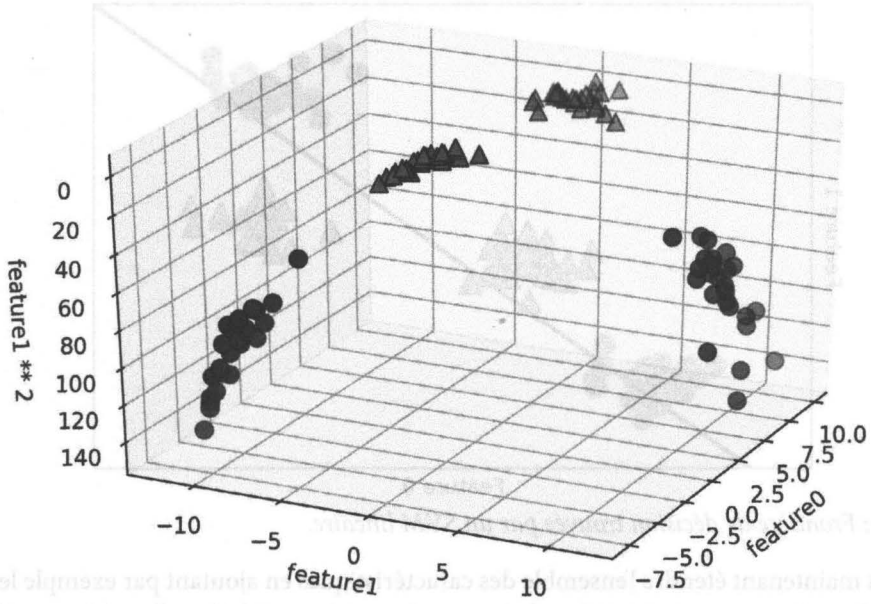



*Figure 2.38 : Extension du jeu de données montré sur la Figure 2.37, créée en ajoutant une troisième caractéristique dérivée de feature1.*

Dans cette nouvelle représentation des données, il devient effectivement possible de séparer les deux classes en utilisant un modèle linéaire, un plan en trois dimensions. Nous pouvons le confirmer en ajustant un modèle linéaire aux données ainsi augmentées (voir la Figure 2.39) :

**In[77]:**

```
linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_

figure = plt.figure()
ax = Axes3D(figure, elev=-152, azim=-26)
xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)

XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60, edgecolor='k')
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r',
           marker='^', cmap=mglearn.cm2, s=60, edgecolor='k')

ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1 ** 2")
```

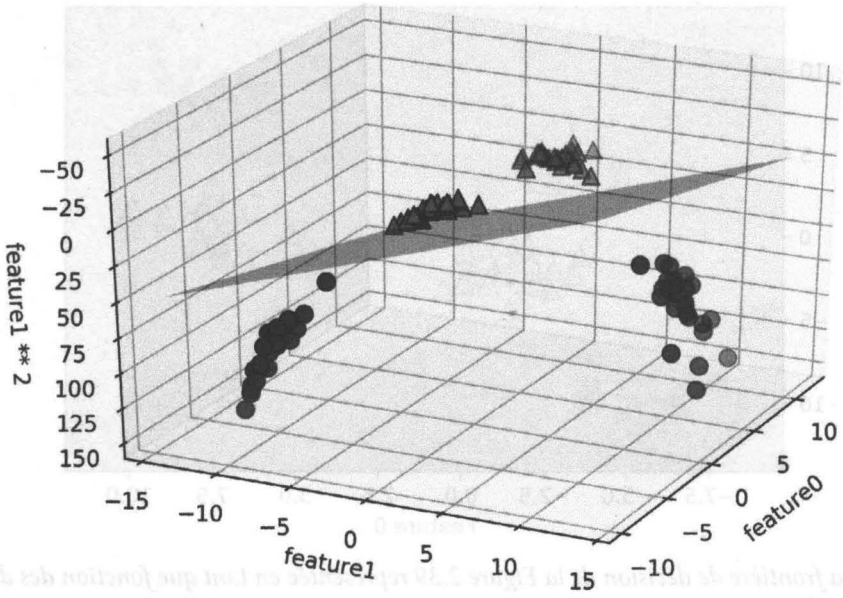



*Figure 2.39 : Frontière de décision trouvée par un SVM linéaire sur le jeu de données en trois dimensions.*

Par rapport aux caractéristiques d'origine, le modèle SVM linéaire n'est en fait plus linéaire. Ce n'est en effet pas une ligne, mais plutôt une ellipse, comme vous pouvez le constater sur le tracé créé ici (voir la Figure 2.40) :

**In[78]:**

```
ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(),
  ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0,
             dec.max()],cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

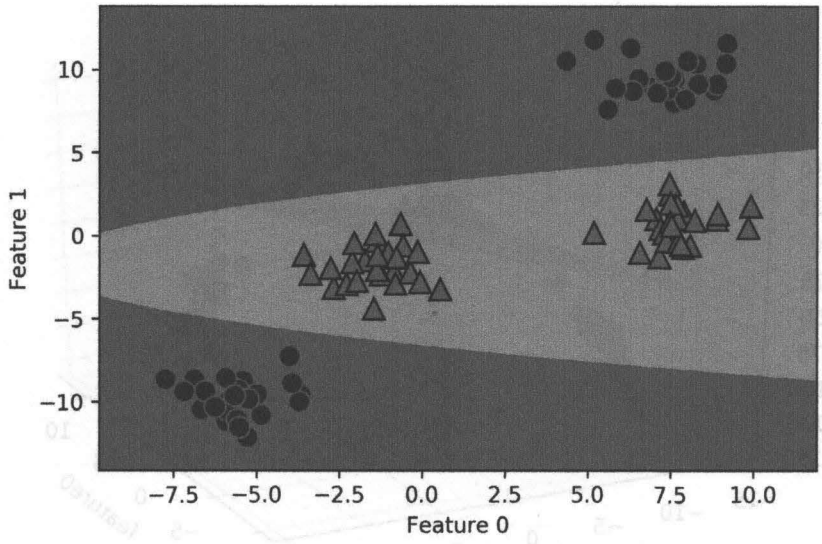



*Figure 2.40 : La frontière de décision de la Figure 2.39 représentée en tant que fonction des deux caractéristiques d'origine.*

### L'astuce du noyau

La leçon à tirer de ce que nous venons de voir est qu'ajouter des caractéristiques non linéaires à la représentation de nos données peut rendre les modèles linéaires bien plus puissants. Cependant, nous ne savons souvent pas quelles caractéristiques ajouter, et en ajouter trop (par exemple, toutes les interactions possibles dans un espace de caractéristiques possédant 100 dimensions) pourrait avoir une incidence considérable en termes de coût de traitement. Mais, heureusement, il existe une habile astuce mathématique qui nous permet d'entraîner un classifieur dans un espace de grandes dimensions sans calculer réellement la nouvelle et possiblement très volumineuse représentation. C'est ce que l'on appelle l'*astuce du noyau*, qui consiste à calculer directement la distance (plus précisément, les produits scalaires) des points de données pour la représentation étendue des caractéristiques, sans jamais calculer réellement cette extension.

Il y a deux manières de s'y prendre qui sont couramment utilisées avec les séparateurs à vaste marge : d'une part le noyau polynomial, qui calcule tous les polynômes possibles jusqu'à un certain degré des caractéristiques d'origine (comme `feature1 ** 2 * feature2 ** 5`), et d'autre part le noyau RBF (pour *Radial Basis Function*) également appelé noyau radial gaussien. Ce dernier est un peu plus difficile à expliquer, mais on peut dire pour faire court qu'il considère tous les polynômes possibles à tous les degrés, mais avec une décroissance de l'importance des caractéristiques pour les degrés les plus élevés.

En pratique, les notions mathématiques sous-jacentes derrière la notion de SVM à noyau ne sont cependant pas si importantes, et la façon dont un SVM avec un noyau RBF prend une décision peut être résumée assez facilement. C'est ce que nous allons voir dans la prochaine section.

### Comprendre les SVM

Pendant la phase d'entraînement, le SVM apprend quelle importance chaque point de donnée du jeu d'apprentissage a pour la représentation de la frontière de décision entre les deux classes. D'une manière typique, seul un sous-ensemble de ces points compte vraiment pour définir la frontière de décision : ce sont les échantillons qui se trouvent près de la frontière entre les classes. On les appelle des *vecteurs support* (soit *support vectors* en anglais, d'où le nom SVM).

Pour faire une prédiction sur un nouveau point, on mesure la distance jusqu'à chacun des vecteurs support. Une décision de classification est effectuée à partir des distances aux vecteurs support, et de l'importance de ces vecteurs support, telle qu'elle a été déterminée au cours de l'entraînement (ces informations sont enregistrées dans l'attribut `dual_coef_` de `SVC`).

La distance entre les points de données est mesurée par le noyau gaussien :

$$k_{rbf}(x_1, x_2) = exp(-\gamma||x_1 - x_2||^2)$$

Ici, $x_1$ et $x_2$ sont les points de données, $||x_1 - x_2||$ indique leur distance euclidienne, et $\gamma$ (gamma) est un paramètre qui contrôle la largeur du noyau gaussien.

La Figure 2.41 montre le résultat de l'entraînement d'un SVM sur un jeu de données à deux dimensions comportant deux classes. La frontière de décision traverse le tracé, et les vecteurs support sont les points de données qui ont un contour plus épais. Le code qui suit produit ce tracé à partir du jeu de données `forge` :

**In[79]:**

```
from sklearn.svm import SVC

X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
sv = svm.support_vectors_
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15,
  markeredgewidth=3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

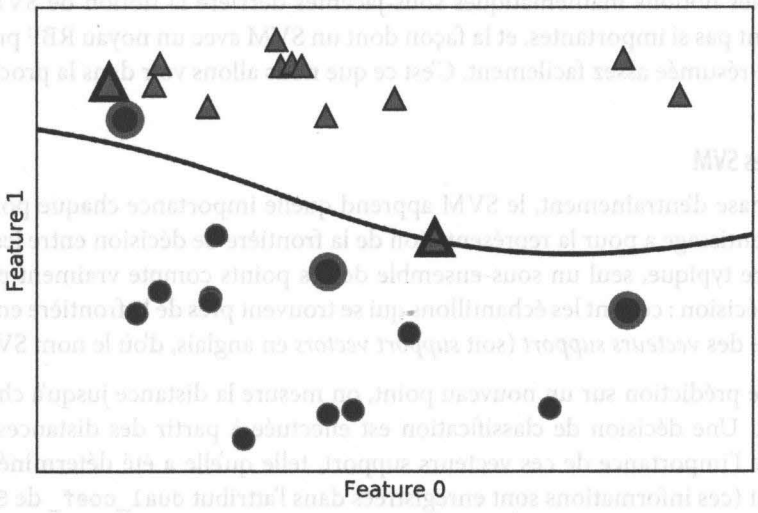



*Figure 2.41 : Frontière de décision et vecteurs support trouvés par un SVM à noyau RBF.*

Dans ce cas, le SVM produit une frontière très lisse et non linéaire (donc pas une ligne droite). Nous avons ajusté ici deux paramètres : `C` et `gamma` sur lesquels nous allons maintenant revenir en détail.

### Ajuster les paramètres du SVM

Le paramètre `gamma`, celui qui apparaît dans la formule donnée dans la section précédente, correspond à l'inverse de la largeur du noyau gaussien. Intuitivement, ce paramètre `gamma` détermine la portée de l'influence d'un certain échantillon de donnée. Une petite valeur correspond à une grande portée, et une grande valeur à une portée limitée. En d'autres termes, plus le rayon du noyau gaussien est large, et plus l'influence de chaque échantillon du jeu d'apprentissage est grande. Quant à `C`, c'est un paramètre de régularisation, semblable à ce que nous avons déjà vu pour les modèles linéaires. Il limite l'importance de chaque point (ou plus précisément leur valeur `dual_coef_`).

Regardons ce qui se passe lorsque nous faisons varier ces paramètres (voir la Figure 2.42) :

**In[80]:**

```
fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)

axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"],
                  ncol=4, loc=(.9, 1.2))
```

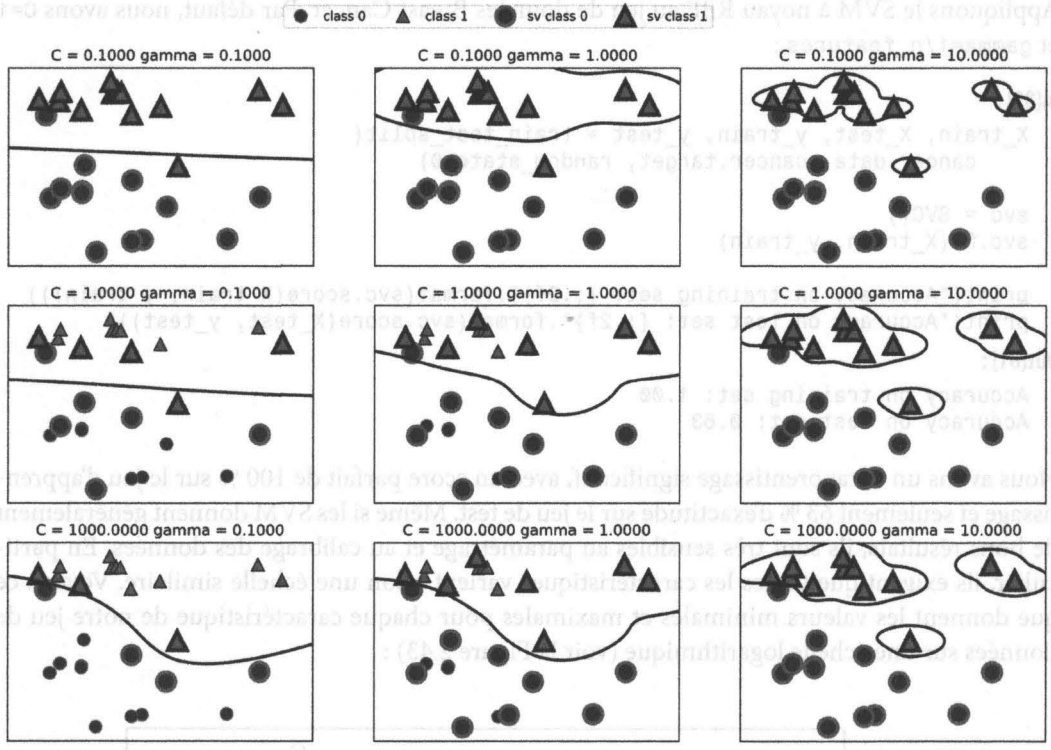



*Figure 2.42 : Frontières de décision et vecteurs support pour différentes valeurs des paramètres* `C` *et* `gamma`.

En allant de gauche à droite, nous augmentons la valeur du paramètre `gamma` de 0.1 à 10. Un petit `gamma` signifie un grand rayon pour le noyau gaussien, et donc que de nombreux points sont considérés comme proches. Cela se traduit par des frontières de décision plus lisses sur la gauche, et des frontières qui se concentrent davantage sur des points individuels à droite. Une valeur basse de `gamma` indique que la frontière de décision va varier lentement, ce qui conduit à un modèle moins complexe, et inversement une valeur élevée va produire un modèle plus complexe.

En allant de haut en bas, nous augmentons la valeur du paramètre `C` de 0.1 à 1000. Comme pour les modèles linéaires, une petite valeur de `C` implique que chaque point de donnée ne peut avoir qu'une influence très limitée sur le modèle. Vous pouvez constater que la frontière de décision semble pratiquement linéaire en haut et à gauche, les points mal classifiés n'ayant qu'une influence négligeable sur la ligne. En augmentant la valeur de `C`, on renforce cette influence, ce qui fait que, comme on peut le constater sur le tracé en bas et à gauche, la frontière de décision prend une forme courbe pour classifier correctement tous les points.

Appliquons le SVM à noyau RBF au jeu de données Breast Cancer. Par défaut, nous avons `C=1` et `gamma=1/n_features` :

**In[81] :**

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

svc = SVC()
svc.fit(X_train, y_train)

print("Accuracy on training set: {:.2f}".format(svc.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(svc.score(X_test, y_test)))
```

**Out[81] :**

```
Accuracy on training set: 1.00
Accuracy on test set: 0.63
```

Nous avons un surapprentissage significatif, avec un score parfait de 100 % sur le jeu d'apprentissage et seulement 63 % d'exactitude sur le jeu de test. Même si les SVM donnent généralement de bons résultats, ils sont très sensibles au paramétrage et au calibrage des données. En particulier, ils exigent que toutes les caractéristiques varient selon une échelle similaire. Voyons ce que donnent les valeurs minimales et maximales pour chaque caractéristique de notre jeu de données sur une échelle logarithmique (voir la Figure 2.43) :

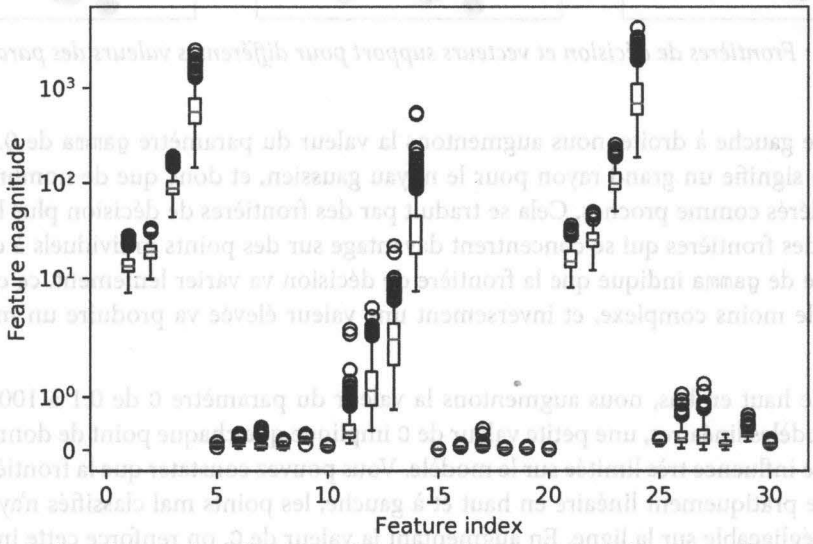



*Figure 2.43 : Ordres de grandeur des caractéristiques du jeu de données Breast Cancer (notez que l'axe y a une échelle logarithmique).*

**In[82]:**

```
plt.boxplot(X_train, manage_xticks=False)
plt.yscale("symlog")
plt.xlabel("Feature index")
plt.ylabel("Feature magnitude")
```

Partant de ce tracé, nous pouvons déterminer que les caractéristiques du jeu de données Breast Cancer ont des ordres de grandeur complètement différents. Cela peut être un problème avec d'autres modèles (comme les modèles linéaires), mais l'effet est dévastateur dans le cas des SVM à noyau. Voyons quelques méthodes pour essayer de corriger ce problème.

### Prétraitement des données pour les SVM

Une manière de résoudre notre problème consiste à recalibrer chacune des caractéristiques afin qu'elles se trouvent approximativement toutes sur une même échelle. Une méthode courante dans le cas des SVM à noyau consiste à recalibrer les données pour qu'elles soient toutes comprises entre 0 et 1. Nous verrons dans le Chapitre 3 comment procéder en utilisant la méthode de prétraitement `MinMaxScaler`. Pour l'instant, contentons-nous de faire ce travail « à la main » :

**In[83]:**

```
min_on_training = X_train.min(axis=0)
range_on_training = (X_train - min_on_training).max(axis=0)

X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minimum for each feature\n{}".format(X_train_scaled.min(axis=0)))
print("Maximum for each feature\n {}".format(X_train_scaled.max(axis=0)))
```

**Out[83]:**

```
Minimum for each feature
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Maximum for each feature
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

**In[84]:**

```
X_test_scaled = (X_test - min_on_training) / range_on_training
```

**In[85]:**

```
svc = SVC()
svc.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
      svc.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

**Out[85]:**

```
Accuracy on training set: 0.948
Accuracy on test set: 0.951
```

Ce recalibrage des données fait une différence considérable ! Nous nous retrouvons en fait ici dans un régime de sous-apprentissage, dans lequel les performances des jeux d'apprentissage et de test sont assez semblables, mais moins près d'une exactitude de 100 %. Partant de là, nous pouvons essayer d'augmenter `C` ou `gamma` pour construire un modèle plus complexe. Par exemple :

**In[86]:**

```
svc = SVC(C=1000)
svc.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    svc.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

**Out[86]:**

```
Accuracy on training set: 0.988
Accuracy on test set: 0.972
```

Ici, le fait d'augmenter la valeur de `C` nous permet d'améliorer le modèle de façon significative, avec une exactitude qui atteint 97,2 %.

#### Forces, faiblesses et paramètres

Les SVM à noyau sont de puissants modèles, et ils sont efficaces avec une large variété de jeux de données. Ils permettent d'obtenir des frontières de décision complexes, même si les données ne présentent que quelques caractéristiques. Ils fonctionnent bien avec des données possédant peu de dimensions, ou au contraire beaucoup de dimensions (donc de caractéristiques). Par contre, ils ne s'adaptent pas forcément facilement aux grandes quantités de données. Exécuter par exemple un SVM sur un jeu de 10 000 données peut parfaitement se dérouler, mais devenir problématique en termes de temps et de consommation de mémoire si on passe à 100 000 données.

Un autre défaut des SVM est qu'ils nécessitent un prétraitement soigneux des données ainsi qu'un ajustement précis des paramètres. C'est pourquoi on préfère souvent aujourd'hui faire appel dans de nombreuses applications à des modèles basés sur des arbres, comme les forêts aléatoires et le *gradient boosting* (qui nécessitent peu ou pas de prétraitement). En outre, les modèles SVM sont difficiles à inspecter. Comprendre pourquoi une certaine prédiction a été faite n'a souvent rien d'évident, et il pourrait être très délicat de tenter d'expliquer le modèle à un non-expert.

Pour autant, il pourrait être bon d'essayer les SVM, particulièrement si toutes vos caractéristiques représentent des mesures faites dans des unités semblables (par exemple, des intensités de pixels) et qu'elles aient des échelles similaires.

Les paramètres importants dans les SVM à noyau sont le paramètre de régularisation `C`, le choix du noyau et les paramètres spécifiques à celui-ci. Nous avons surtout observé de près le noyau RBF, mais il existe d'autres choix disponibles dans `scikit-learn`. Le noyau RBF fait appel à un seul paramètre, `gamma`, qui est l'inverse de la largeur du noyau gaussien. Les paramètres `gamma`

et `C` contrôlent la complexité du modèle, celle-ci augmentant avec leur valeur. C'est pourquoi la corrélation entre ces deux paramètres est forte, et ils devraient donc être ajustés en même temps.

## Réseaux de neurones (*deep learning*)

Une famille d'algorithmes connue sous le nom de réseaux de neurones a récemment retrouvé une nouvelle jeunesse sous l'appellation d'apprentissage profond, ou *deep learning*. Bien que le deep learning soit très prometteur dans de nombreuses applications d'apprentissage automatique, ces algorithmes sont souvent conçus pour répondre de très près à des cas d'utilisation spécifiques. Nous ne discuterons ici que de quelques méthodes relativement simples, à savoir les *perceptrons multicouches* pour la classification et la régression, qui peuvent servir de point de départ pour des techniques de deep learning plus complexes. Les perceptrons multicouches, ou MLP, sont aussi connus sous la dénomination réseaux de neurones *feed-forward* (non bouclés ou acycliques, ou encore à propagation directe), ou parfois juste réseaux de neurones.

### Le modèle de réseau de neurones

Les MLP peuvent être vus comme une généralisation des modèles linéaires effectuant de multiples étapes de traitement pour arriver à prendre une décision.

Rappelez-vous que la formule de prédiction pour une régression linéaire se présente ainsi :

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b$$

En d'autres termes, $\hat{y}$ est une somme pondérée des caractéristiques d'entrée $x[0]$ à $x[p]$, la pondération étant appliquée *via* les coefficients entraînés $w[0]$ à $w[p]$. Nous pourrions visualiser cela graphiquement comme l'illustre la Figure 2.44 :

**In[87]:**

```
display(mglearn.plots.plot_logistic_regression_graph())
```

Ici, chaque nœud de gauche représente une caractéristique d'entrée, les lignes de connexions représentent les coefficients, et le nœud de droite correspond à la sortie, qui est donc une somme pondérée des entrées.

Dans un MLP, ce processus est répété de multiples fois, en commençant par calculer des *unités cachées* qui représentent une étape de traitement intermédiaire, puis qui sont à nouveau combinées en utilisant des sommes pondérées pour produire le résultat final (voir la Figure 2.45) :

**In[88]:**

```
display(mglearn.plots.plot_single_hidden_layer_graph())
```

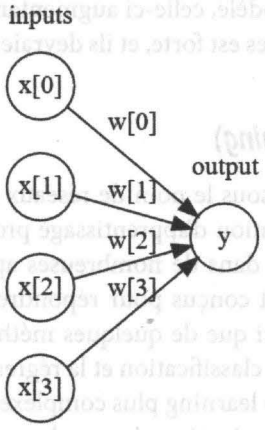



*Figure 2.44 : Visualisation de la régression logistique, les caractéristiques d'entrée et les prédictions étant montrées sous forme de nœuds, les coefficients étant les connexions entre ces nœuds.*

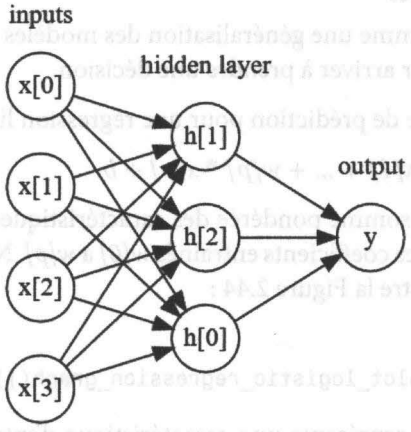



*Figure 2.45 : Illustration d'un perceptron multicouche, avec ici une seule couche cachée.*

Ce modèle a beaucoup plus de coefficients (ou poids) à apprendre. Il y en a en effet un entre chaque entrée et chaque unité cachée (ce qui forme une *couche cachée*), et un autre entre chaque unité de la couche cachée et la sortie.

Calculer une série de sommes pondérées est mathématiquement la même chose que calculer uniquement une somme pondérée. Pour rendre ce modèle véritablement plus puissant qu'un modèle linéaire, nous avons donc besoin d'une astuce supplémentaire. Une fois calculée une somme pondérée pour chaque unité cachée, nous appliquons au résultat une fonction non linéaire, généralement une *unité de rectification linéaire*, ou *relu* (pour *rectified linear unit*) ou bien une *tangente hyperbolique*, ou *tanh*. Le résultat de cette fonction est ensuite utilisé dans la

somme pondérée qui calcule la sortie, $\hat{y}$. Ces deux fonctions, dites d'activation, sont illustrées sur la Figure 2.46. La première, `relu`, coupe les valeurs en dessous de zéro, tandis que `tanh` sature à -1 pour les valeurs d'entrée basses et à +1 pour les valeurs d'entrée élevées. Ces deux fonctions non linéaires permettent au réseau de neurones d'entraîner des fonctions bien plus complexes que ce qu'un modèle linéaire pourrait faire :

**In[89]:**

```
line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best")
plt.xlabel("x")
plt.ylabel("relu(x), tanh(x)")
```

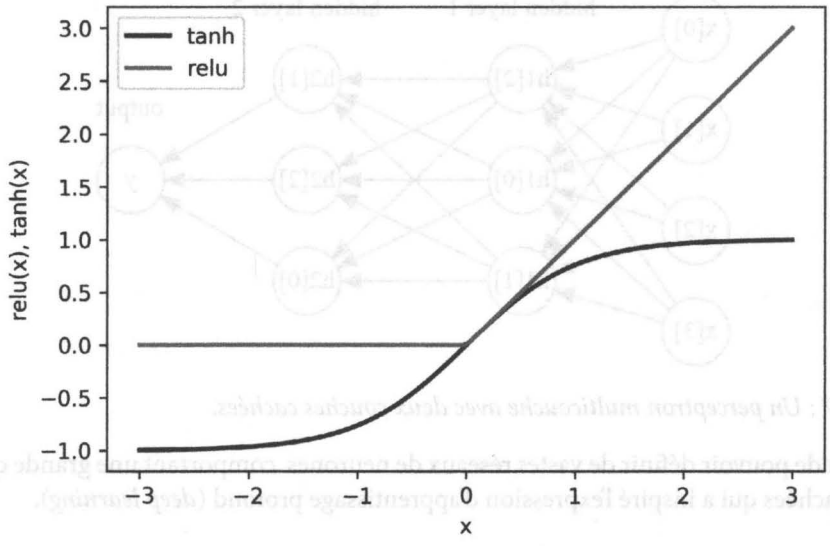



*Figure 2.46 : Fonction d'activation tangente hyperbolique (tanh) et fonction d'activation de rectification linéaire (relu).*

Dans le cas du petit réseau de neurones représenté sur la Figure 2.45, la formule complète de calcul de $\hat{y}$ pour une régression serait (en utilisant une tangente hyperbolique) :

$$h[0] = tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3] + b[0])$$
$$h[1] = tanh(w[0, 1] * x[0] + w[1, 1] * x[1] + w[2, 1] * x[2] + w[3, 1] * x[3] + b[1])$$
$$h[2] = tanh(w[0, 2] * x[0] + w[1, 2] * x[1] + w[2, 2] * x[2] + w[3, 2] * x[3] + b[2])$$
$$\hat{y} = v[0] * h[0] + v[1] * h[1] + v[2] * h[2] + b$$

Ici, les $w$ sont les poids entre l'entrée $x$ et la couche cachée $h$, et les $v$ sont les poids entre la couche cachée $h$ et la sortie $\hat{y}$. Les valeurs de pondération, $v$ et $w$, sont obtenues à partir des données, les $x$ représentent les caractéristiques en entrée, $\hat{y}$ la sortie calculée, et les $h$ les calculs intermédiaires. Un paramètre important qui a besoin d'être défini par l'utilisateur est le nombre de nœuds dans la couche cachée. Il peut s'agir d'une valeur aussi petite que 10 pour des jeux de données réduits ou simples, et une valeur aussi grande que 10 000 pour des jeux de données très complexes. Il est aussi possible d'ajouter davantage de couches cachées, comme l'illustre la Figure 2.47 :

**In[90]:**

```
mglearn.plots.plot_two_hidden_layer_graph()
```

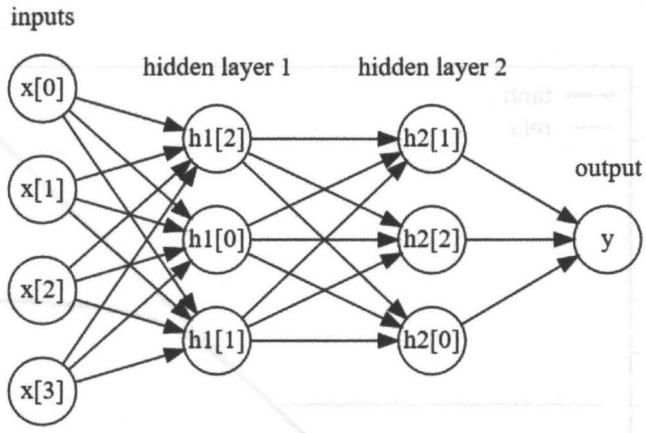



*Figure 2.47 : Un perceptron multicouche avec deux couches cachées.*

C'est le fait de pouvoir définir de vastes réseaux de neurones comportant une grande quantité de couches cachées qui a inspiré l'expression d'apprentissage profond (*deep learning*).

### Ajuster les réseaux de neurones

Voyons plus à fond comment fonctionne un perceptron multicouche en appliquant le classifieur `MLPClassifier` au jeu de données `two_moons` déjà utilisé dans ce chapitre. Les résultats sont montrés sur la Figure 2.48 :

**In[91]:**

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)
```

```
mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Comme vous pouvez le constater, le réseau de neurones a permis d'obtenir une frontière de décision à la fois très non linéaire et en même temps assez lisse. Nous avons utilisé ici `solver='lbfgs'`, sur lequel nous reviendrons plus tard.

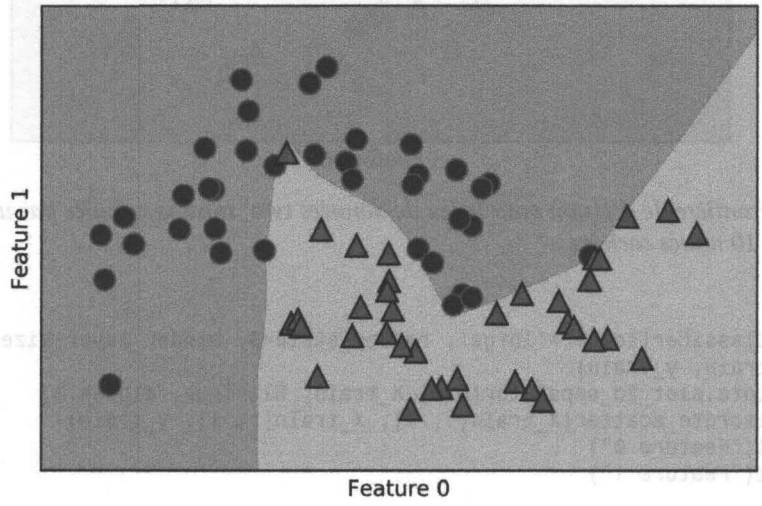



*Figure 2.48 : Frontière de décision pour le jeu de données* `two_moons` *calculée par un réseau de neurones avec 100 unités cachées.*

Par défaut, le MLP utilise 100 nœuds cachés, ce qui est tout de même beaucoup pour ce petit jeu de données. Nous pouvons réduire ce nombre (et donc aussi la complexité du modèle) tout en conservant un bon résultat (voir la Figure 2.49) :

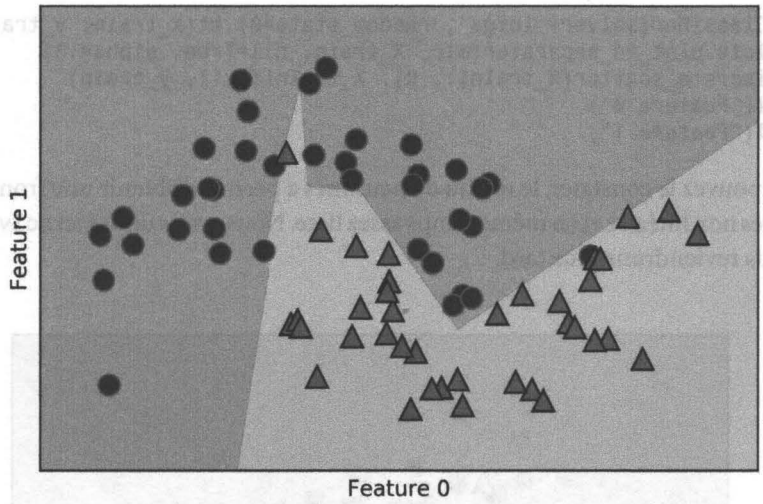



*Figure 2.49 : Frontière de décision pour le jeu de données* **two_moons** *calculée par un réseau de neurones avec 10 unités cachées.*

**In[92] :**

```
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Avec seulement 10 unités cachées, la frontière de décision est un peu plus irrégulière. La fonction non linéaire par défaut est `relu`, montrée sur la Figure 2.46. Avec une unique couche cachée, cela signifie que la fonction de décision comportera 10 segments de ligne droite. Si nous voulons une frontière plus lisse, nous pourrions ajouter davantage d'unités cachées, comme sur la Figure 2.48, ajouter une seconde couche cachée (voir la Figure 2.50) ou utiliser comme fonction d'activation la tangente hyperbolique (voir la Figure 2.51) :

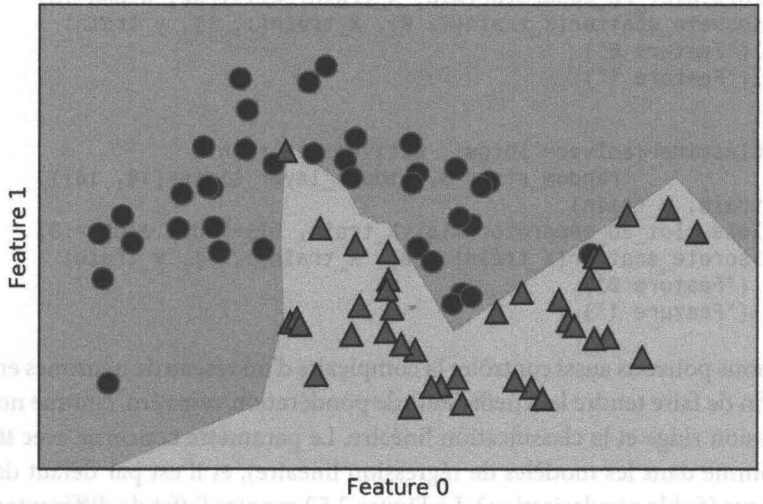



*Figure 2.50 : Frontière de décision calculée en utilisant deux couches cachées avec dix unités cachées chacune, et avec la fonction d'activation* `relu`.

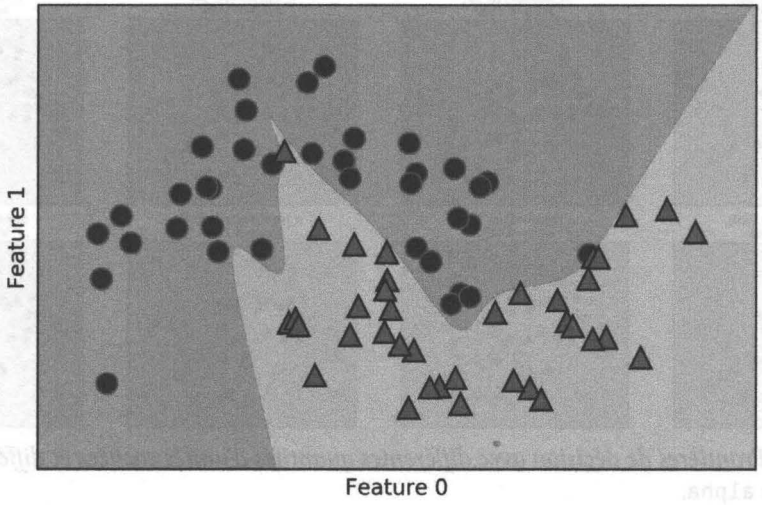



*Figure 2.51 : Frontière de décision calculée en utilisant deux couches cachées avec dix unités cachées chacune, et avec la fonction d'activation* `tanh`.

**In[93]:**

```
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                    hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
```

```
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
    plt.xlabel("Feature 0")
    plt.ylabel("Feature 1")
```

**In[94]:**

```
    mlp = MLPClassifier(solver='lbfgs', activation='tanh',
                        random_state=0, hidden_layer_sizes=[10, 10])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
    plt.xlabel("Feature 0")
    plt.ylabel("Feature 1")
```

Finalement, nous pouvons aussi contrôler la complexité d'un réseau de neurones en utilisant une pénalité L2 afin de faire tendre les coefficients de pondération vers zéro, comme nous l'avons fait dans la régression ridge et la classification linéaire. Le paramètre concerné avec `MLPClassifier` est `alpha` (comme dans les modèles de régression linéaire), et il est par défaut défini avec une très petite valeur (faible régularisation). La Figure 2.52 montre l'effet de différentes valeurs pour `alpha` sur le jeu de données `two_moons`, en utilisant deux couches masquées de 10 ou 100 unités chacune :

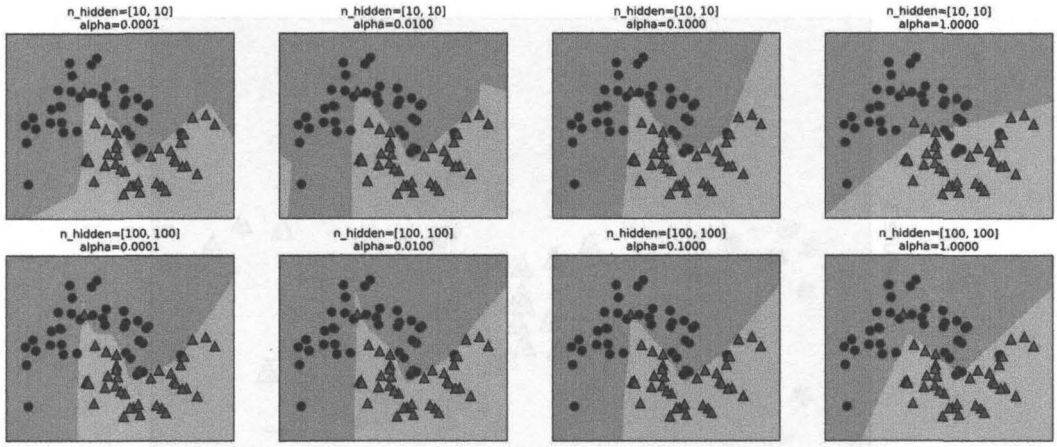



*Figure 2.52 : Frontières de décision avec différentes quantités d'unités cachées et différentes valeurs du paramètre* `alpha`.

**In[95]:**

```
    fig, axes = plt.subplots(2, 4, figsize=(20, 8))
```

```
for axx, n_hidden_nodes in zip(axes, [10, 100]):
    for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(solver='lbfgs', random_state=0,
                            hidden_layer_sizes=[n_hidden_nodes,
                            n_hidden_nodes],alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3,
  ax=ax)
        mglearn.discrete_scatter(X_traIn[:, 0], X_traIn[:, 1], y_train, ax=ax)
        ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(
                     n_hidden_nodes, n_hidden_nodes, alpha))
```

Comme vous l'avez probablement déjà réalisé, il y a de nombreuses manières de contrôler la complexité d'un réseau de neurones : le nombre de couches cachées, le nombre d'unités dans chaque couche cachée, et le paramètre de régularisation (`alpha`). Il y en a en fait encore plus, mais nous n'entrerons pas ici dans les détails.

Une importante propriété des réseaux de neurones est que leurs coefficients de pondération sont définis de manière aléatoire avant que l'entraînement ne débute, et cette initialisation aléatoire affecte évidemment le modèle qui est entraîné. Cela signifie que, même en utilisant exactement les mêmes paramètres, nous pouvons obtenir des modèles très différents selon la manière dont le générateur de nombres aléatoires est initialisé. Si les réseaux sont de grande taille et si leur complexité est correctement choisie, cela ne devrait pas trop affecter l'exactitude, mais il faut garder cette notion présente à l'esprit, en particulier avec des réseaux de taille réduite. La Figure 2.53 montre le tracé de plusieurs modèles, tous entraînés avec la même configuration de paramètres :

**In[96]:**

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs', random_state=i,
                        hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_traIn[:, 0], X_traIn[:, 1], y_train, ax=ax)
```

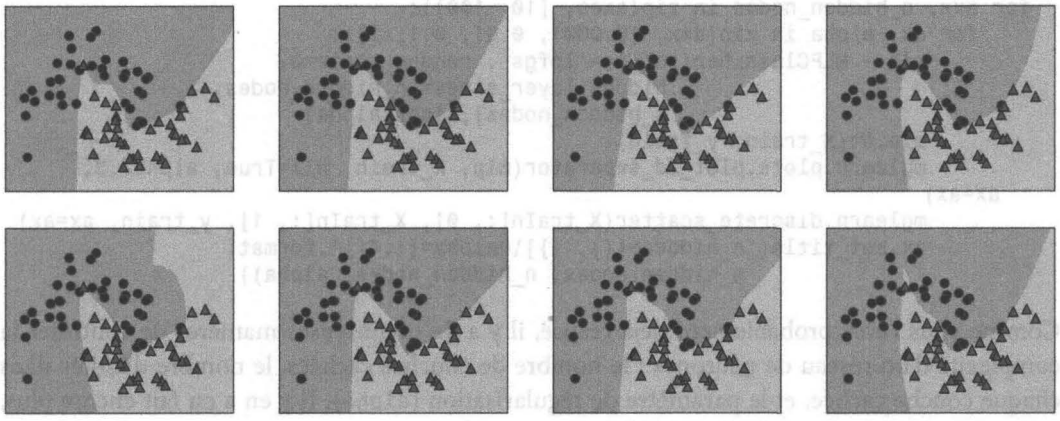


*Figure 2.53 : Fonctions de décision entraînées avec les mêmes paramètres mais avec différentes initialisations aléatoires.*

Pour nous faire une meilleure idée du fonctionnement des réseaux de neurones avec des données du monde réel, essayons d'appliquer le classifieur `MLPClassifier` au jeu de données Breast Cancer. Nous commençons avec les paramètres par défaut :

**In[97]:**

```
print("Cancer data per-feature maxima:\n{}".format(cancer.data.max(axis=0)))
```

**Out[97]:**

```
Cancer data per-feature maxima:
[   28.11     39.28    188.5    2501.       0.163     0.345     0.427
     0.201     0.304     0.097     2.873     4.885    21.98    542.2
     0.031     0.135     0.396     0.053     0.079     0.03     36.04
    49.54    251.2    4254.      0.223     1.058     1.252     0.291
     0.664     0.207]
```

**In[98]:**

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)

print("Accuracy on training set: {:.2f}".format(mlp.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(X_test, y_test)))
```

**Out[98]:**

```
Accuracy on training set: 0.91
Accuracy on test set: 0.88
```

L'exactitude obtenue avec le MLP est assez bonne, mais moins qu'avec d'autres modèles. Comme dans l'exemple de SVC proposé plus haut, cela est probablement dû au calibrage des données. Les réseaux de neurones s'attendent aussi à ce que toutes les caractéristiques d'entrée varient

de manière similaire, et qu'elles aient dans l'idéal une moyenne de 0 et une variance de 1. Nous devons recalibrer nos données pour qu'elles remplissent ces conditions. De nouveau, nous allons faire ce travail à la main, mais nous verrons dans le Chapitre 3 comment automatiser cette tâche avec `StandardScaler` :

**In[99]:**

```
mean_on_train = X_train.mean(axis=0)
std_on_train = X_train.std(axis=0)

X_train_scaled = (X_train - mean_on_train) / std_on_train
X_test_scaled = (X_test - mean_on_train) / std_on_train

mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

**Out[99]:**

```
Accuracy on training set: 0.991
Accuracy on test set: 0.965

ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
  the optimization hasn't converged yet.
```

Les résultats sont maintenant bien meilleurs et déjà assez compétitifs. Cependant, nous avons vu s'afficher un avertissement du modèle qui nous prévient que le nombre maximum d'itérations a été atteint. Il provient de l'algorithme utilisé par défaut pour l'entraînement du modèle, `adam`, et nous indique que nous devrions augmenter le nombre d'itérations. :

**In[100]:**

```
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

**Out[100]:**

```
Accuracy on training set: 0.995
Accuracy on test set: 0.965
```

Augmenter le nombre d'itérations n'a augmenté ici que les performances du jeu d'apprentissage, pas celles du jeu de test (mais cela n'a rien d'une règle générale et dépend des données comme de l'initialisation de l'entraînement). Le modèle fonctionne plutôt bien. Comme il reste un certain décalage entre les performances des jeux d'apprentissage et de test, nous pourrions essayer de diminuer la complexité du modèle afin d'obtenir de meilleures performances de généralisation.

Ici, nous choisissons d'augmenter fortement la valeur du paramètre `alpha` (de 0.0001 à 1) afin de renforcer la régularisation des coefficients :

**In[101]:**

```
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

**Out[101]:**

```
Accuracy on training set: 0.988
Accuracy on test set: 0.972
```

Cela donne un niveau de performance comparable aux meilleurs modèles étudiés jusqu'ici.



Nombre de bons modèles atteignent la même exactitude qui est de 0.972. Cela signifie que tous les modèles font exactement le même nombre d'erreurs, soit quatre. Si vous comparez les prédictions effectuées, vous pouvez même voir qu'ils font exactement les *mêmes* erreurs. Cela peut provenir du fait que le jeu de données a une taille réduite, ou que les points concernés sont réellement différents des autres.

Bien qu'il soit possible d'analyser ce qu'un réseau de neurones a appris, cela est généralement plus compliqué qu'avec un modèle linéaire ou qu'avec un modèle à base d'arbres. Une manière de s'y prendre consiste à examiner les valeurs de pondération du modèle (voyez par exemple à ce sujet la page `http://scikit-learn.org/stable/auto_examples/neural_networks/plot_mnist_filters.html` de `scikit-learn`). Dans le cas du jeu de données Breast Cancer, cette question est assez difficile à comprendre. Le tracé illustré sur la Figure 2.54 montre les poids acquis en connectant l'entrée à la première couche cachée. Les lignes du tracé correspondent aux 30 caractéristiques d'entrée, et les colonnes aux 100 unités cachées. Les nuances les plus claires représentent les grandes valeurs positives, et les teintes sombres représentent les valeurs négatives (vous pouvez vous référer à l'échelle placée à droite) :

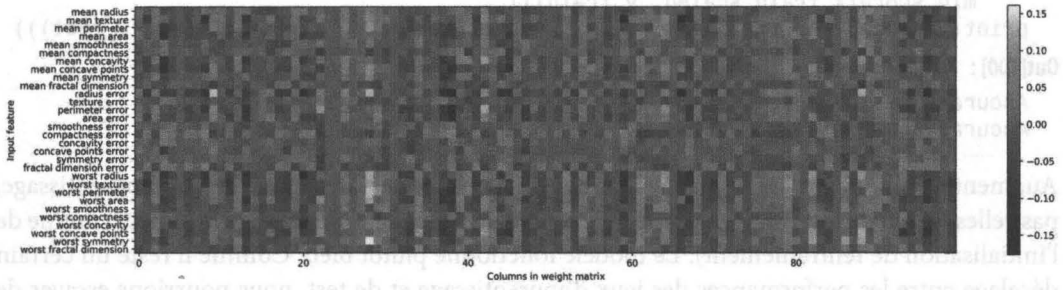



*Figure 2.54 : Carte de chaleur des poids de la première couche cachée dans un réseau de neurones entraîné sur le jeu de données Breast Cancer.*

In[102]:

```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Columns in weight matrix")
plt.ylabel("Input feature")
plt.colorbar()
```

Une inférence possible que nous pouvons faire à partir de cette observation est que les caractéristiques qui ont de très petits poids pour toutes les unités cachées sont « moins importantes » pour le modèle. Nous pouvons voir que les caractéristiques « mean smoothness » et « mean compactness », en plus des caractéristiques trouvées entre « smoothness error » et « fractal dimension error », ont des poids relativement faibles comparativement aux autres caractéristiques. Cela pourrait signifier qu'elles sont moins importantes, ou possiblement que nous ne les avons pas représentées d'une manière utilisable par le réseau de neurones.

Nous pourrions également visualiser les poids connectant la couche cachée à la couche de la sortie, mais ils seraient encore plus difficiles à interpréter.

Bien que les classes `MLPClassifier` et `MLPRegressor` fournissent des interfaces faciles à utiliser pour les architectures de réseaux de neurones les plus courantes, elles ne capturent qu'une petite partie de ce qu'il est possible de faire avec les réseaux de neurones. Si vous souhaitez aller plus loin que ce que `scikit-learn` propose tout en restant dans le monde de Python, tournez-vous vers les packages les plus renommés que sont `keras`, `lasagna` et `tensor-flow`. `Lasagna` est bâti sur la librairie `theano`, tandis que `keras` peut utiliser soit `tensor-flow` soit `theano`. Ces librairies fournissent une interface beaucoup plus souple et puissante pour construire des réseaux de neurones, et elles suivent les progrès rapides des recherches dans le domaine du deep learning. De plus, toutes les librairies de deep learning populaires savent tirer parti des processeurs graphiques à haute performance actuels (GPU), ce que `scikit-learn` n'est pas capable de faire. Travailler en étroite collaboration avec des GPU signifie en particulier multiplier la rapidité des calculs par un facteur pouvant aller de 10 à 100, et qui est essentiel pour appliquer les méthodes du deep learning à de vastes jeux de données.

### Forces, faiblesses et paramètres

Les réseaux de neurones ont refait surface en tant que modèles à la pointe du progrès dans de nombreuses applications d'apprentissage automatique. Un de leurs principaux avantages est qu'ils sont capables de capturer des informations contenues dans de vastes quantités de données et de construire des modèles incroyablement complexes. Avec les « bonnes » données, des réglages soigneusement effectués et un temps de calcul suffisant, les réseaux de neurones battent souvent les autres algorithmes d'apprentissage machine (pour la classification et la régression).

Cela nous conduit aux inconvénients. Les réseaux de neurones, et en particulier ceux qui sont larges et puissants, demandent souvent un temps long pour l'entraînement. Ils nécessitent aussi

un prétraitement soigneux des données, comme nous l'avons vu ici. Comme les SVM, ils travaillent mieux avec des données « homogènes », c'est-à-dire dans lesquelles toutes les caractéristiques ont des significations similaires. Dans le cas de données dont les caractéristiques sont de nature très différente, les modèles basés sur des arbres pourraient mieux fonctionner. Ajuster les paramètres des réseaux de neurones est également une sorte d'art en soi. Dans ce livre, nous n'avons pu qu'effleurer la surface des questions concernant l'ajustement des réseaux de neurones et de l'entraînement de ces modèles.

#### Estimer la complexité des réseaux de neurones

Les paramètres les plus importants sont le nombre de couches et le nombre d'unités cachées par couche. Vous devriez commencer par une ou deux couches cachées, et le cas échéant voir ensuite à les développer. Le nombre de nœuds par couche cachée est souvent similaire au nombre des caractéristiques d'entrée, mais ne dépasse que rarement des valeurs basses ou moyennes.

Une mesure utile pour juger de la complexité d'un réseau de neurones est le nombre de poids ou de coefficients qui ont été entraînés. Si vous avez un jeu de données pour une classification binaire avec 100 caractéristiques, et si vous avez 100 unités cachées, alors cela signifie qu'il y a 100 * 100 = 10 000 poids entre l'entrée et la première couche cachée. Il faut y ajouter également 100 * 1 = 100 poids entre la couche cachée et la couche de sortie. Si vous ajoutez une seconde couche cachée avec toujours 100 unités cachées, cela augmente encore le total de 10 000 poids, soit en tout 20 100. Supposons que nous passions à une couche avec 1 000 unités cachées. Vous devez entraîner 100 * 1 000 = 100 000 poids pour aller de l'entrée à la couche cachée, puis 1 000 * 1 = 1 000 poids entre la couche cachée et la sortie, soit en tout 101 000. Avec une seconde couche, nous devons additionner 1 000 * 1000 = 1 000 000 nouveaux poids, soit un total de 1 101 000, c'est-à-dire 50 fois plus que le modèle avec deux couches cachées de taille 100.

Une manière courante d'ajuster les paramètres dans un réseau de neurones consiste à commencer par créer un réseau suffisamment grand pour engendrer un surapprentissage, en s'assurant que la tâche peut réellement être entraînée par le réseau. Une fois que vous savez que le jeu de données d'apprentissage peut être entraîné, soit vous réduisez le réseau, soit vous augmentez la valeur d'`alpha` pour ajouter de la régularisation, ce qui améliorera les performances de généralisation.

Nous nous sommes ici concentrés essentiellement sur la définition du modèle : le nombre de couches et la quantité de nœuds par couche, la régularisation et la non-linéarité. Tout cela définit le modèle que nous voulons entraîner. Mais il y a aussi la question de savoir *comment* entraîner le modèle, ou l'algorithme qui est utilisé pour entraîner les paramètres, ce qui se définit *via* le paramètre appelé `algorithm`. Il y a deux choix faciles à utiliser pour ce paramètre. Le premier, qui est sa valeur par défaut, est `'adam'`. Il fonctionne bien dans la plupart des situations mais est assez sensible au calibrage des données (c'est pourquoi il est toujours important de recalibrer vos données pour obtenir une moyenne de 0 et une variance de 1). L'autre est `'lbfgs'`, qui est assez robuste mais pourrait être lent avec des modèles très complexes ou des jeux de données volumineux. Il y a aussi l'option plus avancée `'sgd'`, qui est celle dont se servent de nombreux

chercheurs spécialisés en deep learning. Cette option possède des paramètres supplémentaires qui doivent être ajustés pour obtenir de meilleurs résultats. Voyez par exemple les précisions que peut vous donner à ce sujet la page `http://scikit-learn.org/stable/modules/classes.html#module-sklearn.neural_network` du site de `scikit-learn`. Pour débuter avec les MLP, nous vous conseillons de vous en tenir à `adam` et `lbfgs`.



Un point important dans les modèles de `scikit-learn` est qu'un appel à la méthode `fit` réinitialise toujours un modèle précédemment entraîné. Si vous construisez un modèle sur un jeu de données, puis que vous appelez à nouveau `fit` sur un autre jeu de données, ce modèle va « oublier » tout ce qu'il avait appris du premier jeu. Vous pouvez appeler `fit` autant que vous le voulez sur un modèle, mais vous ne devez alors pas oublier que la sortie sera la même qu'en effectuant cet appel sur un « nouveau » modèle.

## Estimer l'incertitude pour les classifieurs

Une autre partie utile de l'interface de `scikit-learn` dont nous n'avons pas encore parlé concerne la possibilité des classifieurs de fournir une estimation du taux de certitude, ou d'incertitude, de leurs prédictions. Souvent, vous n'êtes pas seulement intéressé à savoir quelle classe un classifieur prédit pour un certain point de test, mais aussi par le degré de certitude qu'il a quant à la justesse de cette prédiction. En pratique, différents types d'erreurs peuvent conduire à des résultats très divergents dans les applications du monde réel. Prenons l'exemple d'une application médicale spécialisée dans l'évaluation des risques de cancer. Faire une prédiction qui soit un faux positif pourrait conduire à ce qu'un patient subisse des tests supplémentaires, tandis qu'un faux négatif pourrait se traduire par le fait qu'une tumeur grave ne soit pas traitée. Nous y reviendrons plus en détail dans le Chapitre 6.

Il y a deux fonctions différentes de `scikit-learn` qui peuvent être utilisées pour obtenir une estimation d'incertitude avec des classifieurs : `decision_function` et `predict_proba`. La plupart des classifieurs (mais pas tous) ont au moins l'une des deux, et de nombreux classifieurs possèdent ces deux fonctions. Observons ce qu'elles font sur un jeu de données synthétique à deux dimensions, en appliquant le classifieur `GradientBoostingClassifier` qui possède les deux méthodes `decision_function` et `predict_proba` :

**In[103]:**

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)

y_named = np.array(["blue", "red"])[y]

X_train, X_test, y_train_named, y_test_named, y_train, y_test = \
    train_test_split(X, y_named, y, random_state=0)
```

```
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train_named)
```

## La fonction de décision

Dans le cas de la classification binaire, la valeur de retour de `decision_function` est de la forme `(n_samples,)`, et elle retourne une valeur en virgule flottante pour chaque échantillon :

**In[104]:**

```
print("X_test.shape: {}".format(X_test.shape))
print("Decision function shape: {}".format(
      gbrt.decision_function(X_test).shape))
```

**Out[104]:**

```
X_test.shape: (25, 2)
Decision function shape: (25,)
```

Cette valeur nous dit avec quelle « force » le modèle croit qu'un point de donnée appartient à la classe « positive », en l'occurrence la classe 1. Les valeurs positives marquent une préférence pour la classe positive, et les valeurs négatives une préférence pour l'autre classe (donc « négative ») :

**In[105]:**

```
print("Decision function:\n{}".format(gbrt.decision_function(X_test)[:6]))
```

**Out[105]:**

```
Decision function:
[ 4.136 -1.702 -3.951 -3.626  4.29   3.662]
```

Nous pouvons retrouver la prédiction en regardant uniquement le signe de la fonction de décision :

**In[106]:**

```
print("Thresholded decision function:\n{}".format(
    gbrt.decision_function(X_test) > 0))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

**Out[106]:**

```
Thresholded decision function:
[ True False False False  True  True False  True  True  True False  True
  True False  True False False False  True  True  True  True  True False
 False]
Predictions:
['red' 'blue' 'blue' 'blue' 'red' 'red' 'blue' 'red' 'red' 'red' 'blue'
 'red' 'red' 'blue' 'red' 'blue' 'blue' 'blue' 'red' 'red' 'red' 'red'
 'red' 'blue' 'blue']
```

Pour la classification binaire, la classe « négative » est toujours la première entrée de l'attribut `classes_`, et la classe « positive » la seconde entrée de cet attribut. Par conséquent, si vous

voulez récupérer la totalité de la sortie de la fonction `predict`, vous devez faire appel à l'attribut `classes_` :

**In[107]:**

```
greater_zero = (gbrt.decision_function(X_test) > 0).astype(int)
pred = gbrt.classes_[greater_zero]
print("pred is equal to predictions: {}".format(
    np.all(pred == gbrt.predict(X_test))))
```

**Out[107]:**

```
pred is equal to predictions: True
```

La plage de `decision_function` peut être arbitraire, et dépend des données ainsi que des paramètres du modèle :

**In[108]:**

```
decision_function = gbrt.decision_function(X_test)
print("Decision function minimum: {:.2f} maximum: {:.2f}".format(
    np.min(decision_function), np.max(decision_function)))
```

**Out[108]:**

```
Decision function minimum: -7.69 maximum: 4.29
```

Cette échelle arbitraire rend souvent la sortie de `decision_function` difficile à interpréter.

Dans l'exemple qui suit, nous traçons la fonction `decision_function` pour tous les points dans le plan 2D, avec un codage en niveaux de gris et une échelle affichée sur la droite. Les points du jeu d'apprentissage sont représentés par des cercles, et ceux du jeu de test par des triangles (voir la Figure 2.55) :

**In[109]:**

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4,
                                fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1],
                                            alpha=.4, cm=mglearn.ReBl)

for ax in axes:
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
cbar.set_alpha(1)
cbar.draw_all()
axes[0].legend(["Test class 0", "Test class 1", "Train class 0",
                "Train class 1"], ncol=4, loc=(.1, 1.1))
```

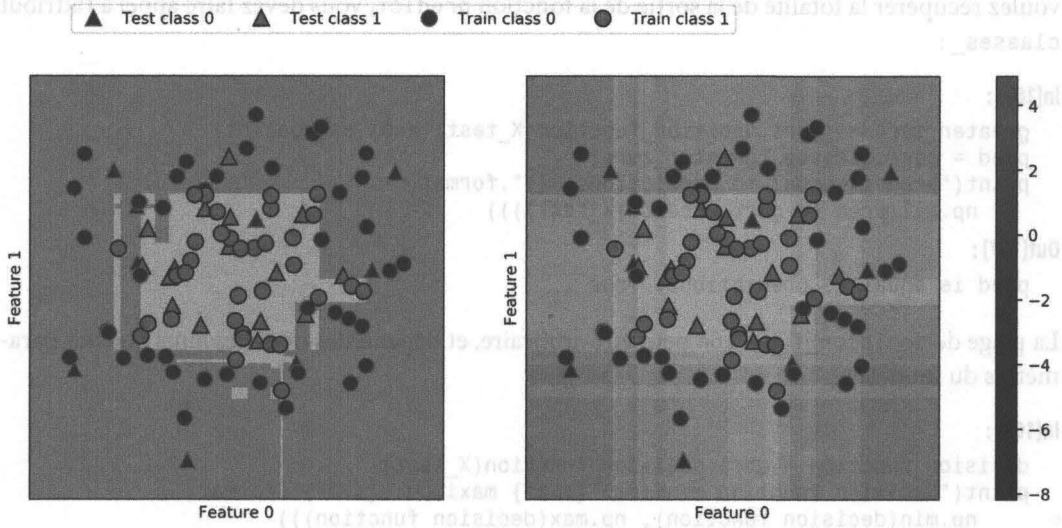



*Figure 2.55 : Frontière de décision (à gauche) et fonction de décision (à droite) pour un modèle de gradient boosting sur un jeu de données à deux dimensions.*

Encoder non seulement la sortie prédite, mais aussi le niveau de certitude de cette prévision par le classifieur fournit des informations supplémentaires. Cependant, dans cette visualisation, il est difficile de distinguer la frontière entre les deux classes.

## Prédire les probabilités

La sortie produite par `predict_proba` est une probabilité pour chaque classe et elle est souvent plus facile à interpréter que celle de `decision_function`. Elle est toujours de la forme `(n_samples, 2)` pour la classification binaire :

**In[110]:**

```
print("Shape of probabilities: {}".format(gbrt.predict_proba(X_test).shape))
```

**Out[110]:**

```
Shape of probabilities: (25, 2)
```

La première entrée de chaque ligne est la probabilité estimée de la première classe, et la seconde entrée celle de la deuxième classe. Comme il s'agit d'une probabilité, la sortie de `predict_proba` se situe forcément entre 0 et 1, et la somme des entrées des deux classes est toujours égale à 1 :

**In[111]:**

```
print("Predicted probabilities:\n{}".format(
    gbrt.predict_proba(X_test[:6])))
```

**Out[111]:**

```
Predicted probabilities:
[[ 0.016  0.984]
 [ 0.846  0.154]
 [ 0.981  0.019]
 [ 0.974  0.026]
 [ 0.014  0.986]
 [ 0.025  0.975]]
```

Du fait que la somme des probabilités pour les deux classes est égale à 1, seule une des classes sera avec certitude au-dessus de 50 % (comme ces probabilités sont des valeurs en virgule flottante, une parité parfaite entre les deux classes est hautement improbable, mais, si c'était le cas, la prédiction serait effectuée de manière aléatoire). C'est donc elle qui est prédite.

Vous pouvez voir sur la sortie précédente que le classifieur est relativement certain de lui pour la plupart des points. La manière dont l'incertitude reflète la réalité de cette même incertitude dans les données dépend du modèle et des paramètres. Par exemple, un surapprentissage plus fort aura tendance à déclarer ses prédictions plus sûres, même s'il peut se tromper. Un modèle moins complexe présente généralement une plus grande incertitude dans ses prédictions. Un modèle est dit *calibré* si une prédiction est en accord avec la réalité, autrement dit si une prédiction faite avec, par exemple, 70 % de certitude se révèle correcte 70 % du temps.

Dans l'exemple qui suit, nous allons de nouveau montrer la frontière de décision sur le jeu de données, mais cette fois en la comparant avec les probabilités pour la classe 1 (voir la Figure 2.56) :

**In[112]:**

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot_2d_separator(
    gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(
    gbrt, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl, function='predict_proba')

for ax in axes:
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
cbar.set_alpha(1)
cbar.draw_all()
axes[0].legend(["Test class 0", "Test class 1", "Train class 0",
                "Train class 1"], ncol=4, loc=(.1, 1.1))
```

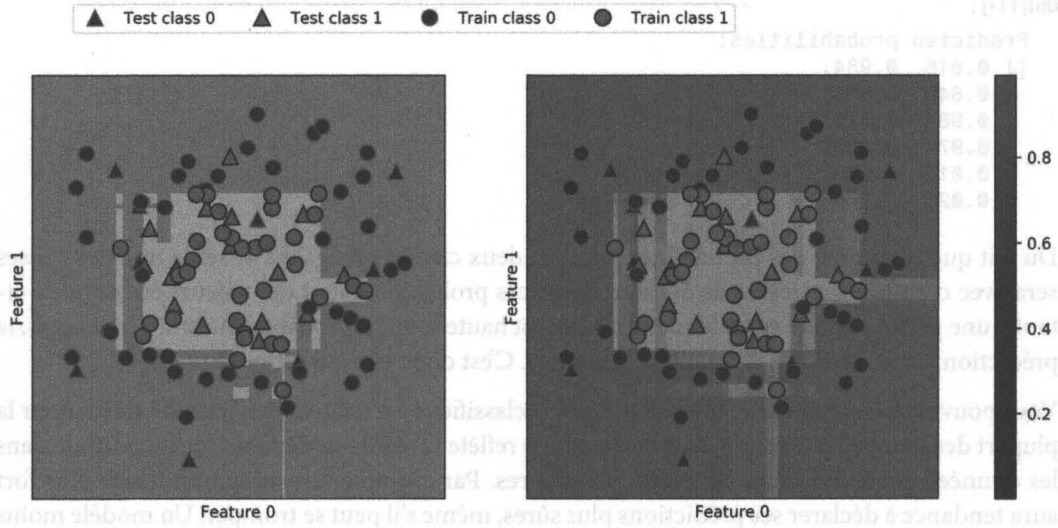



*Figure 2.56 : Frontière de décision (à gauche) et probabilités prédites (à droite) pour le modèle de gradient boosting de la Figure 2.55.*

Dans ce tracé, les frontières sont bien mieux définies de manière correcte, et les petites zones d'incertitude sont clairement visibles.

Le site Web de `scikit-learn` a une excellente page illustrant une comparaison entre plusieurs modèles de classification et ce à quoi ressemble leur estimation de l'incertitude (voir la Figure 2.57). Cette page se trouve à l'adresse :

```
http://scikit-learn.org/stable/auto_examples/classification/
plot_classifier_comparison.html
```

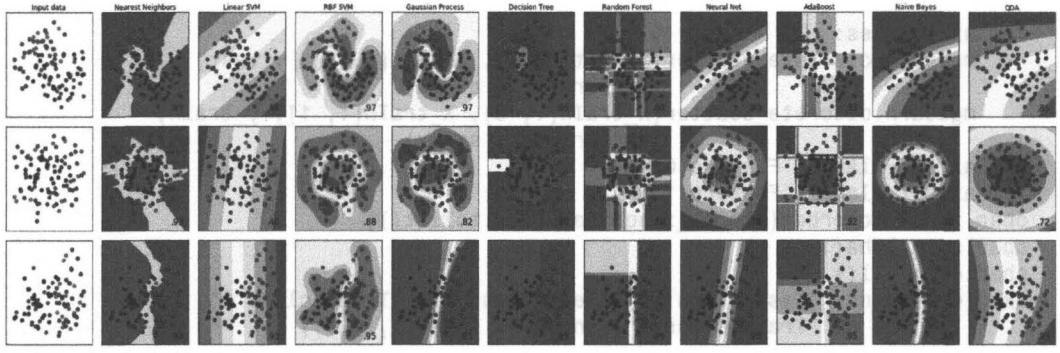



*Figure 2.57 : Comparaison entre plusieurs classifieurs sur des jeux de données synthétiques (avec l'aimable autorisation de scikit-learn.org).*

## Classification multiclasse et incertitude

Jusqu'ici, nous n'avons abordé que les estimations d'incertitudes pour les modèles de classification binaire. Mais les méthodes `decision_function` et `predict_proba` fonctionnent également avec des classifications multiclasses. Essayons donc de les appliquer sur le jeu de données Iris, avec trois classes :

**In[113]:**

```
from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=42)

gbrt = GradientBoostingClassifier(learning_rate=0.01, random_state=0)
gbrt.fit(X_train, y_train)
```

**In[114]:**

```
print("Decision function shape: {}".format(gbrt.decision_function(X_test).
  shape))
print("Decision function:\n{}".format(gbrt.decision_function(X_test)[:6, :]))
```

**Out[114]:**

```
Decision function shape: (38, 3)
Decision function:
[[-0.529  1.466 -0.504]
 [ 1.512 -0.496 -0.503]
 [-0.524 -0.468  1.52 ]
 [-0.529  1.466 -0.504]
 [-0.531  1.282  0.215]
 [ 1.512 -0.496 -0.503]]
```

Dans le cas d'une classification multiclasse, `decision_function` prend la forme `(n_samples, n_classes)` et chaque colonne fournit un « score de certitude » pour chaque classe. Plus ce score est élevé et plus la classe est probable, et inversement. Vous pouvez récupérer les prédictions à partir de ces scores en trouvant l'entrée maximum pour chaque point de donnée :

**In[115]:**

```
print("Argmax of decision function:\n{}".format(
    np.argmax(gbrt.decision_function(X_test), axis=1)))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

**Out[115]:**

```
Argmax of decision function:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Predictions:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

La sortie produite par `predict_proba` a la même forme `(n_samples, n_classes)`. De nouveau, la somme des probabilités des classes possibles pour chaque point de donnée est égale à 1 :

**In[116]:**

```
print("Predicted probabilities:\n{}".format(gbrt.predict_proba(X_test)[:6]))
print("Sums: {}".format(gbrt.predict_proba(X_test)[:6].sum(axis=1)))
```

**Out[116]:**

```
Predicted probabilities:
[[ 0.107  0.784  0.109]
 [ 0.789  0.106  0.105]
 [ 0.102  0.108  0.789]
 [ 0.107  0.784  0.109]
 [ 0.108  0.663  0.228]
 [ 0.789  0.106  0.105]]
Sums: [ 1.  1.  1.  1.  1.  1.]
```

Nous pouvons également récupérer les prédictions en calculant les valeurs argmax pour `predict_proba` :

**In[117]:**

```
print("Argmax of predicted probabilities:\n{}".format(
    np.argmax(gbrt.predict_proba(X_test), axis=1)))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

**Out[117]:**

```
Argmax of predicted probabilities:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Predictions:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

En résumé, `predict_proba` et `decision_function` prennent toujours la forme `(n_samples, n_classes)`, sauf pour `decision_function` dans le cas particulier binaire. En effet, `decision_function` ne contient alors qu'une seule colonne qui correspond à la classe « positive » `classes_[1]`. Cette distinction résulte essentiellement de raisons historiques.

Vous pouvez récupérer les prédictions pour les points des `n_classes` en calculant les valeurs argmax pour chaque colonne. Faites cependant attention si vos classes sont des chaînes de caractères, ou si vous utilisez des entiers qui ne sont pas consécutifs et partant de zéro. Si vous voulez comparer les résultats obtenus avec `predict` avec ceux produits par `decision_function` ou `predict_proba`, assurez-vous d'utiliser l'attribut `classes_` du classifieur pour obtenir les noms réels des classes :

**In[118]:**

```
logreg = LogisticRegression()

named_target = iris.target_names[y_train]
logreg.fit(X_train, named_target)
print("unique classes in training data: {}".format(logreg.classes_))
print("predictions: {}".format(logreg.predict(X_test)[:10]))
argmax_dec_func = np.argmax(logreg.decision_function(X_test), axis=1)
print("argmax of decision function: {}".format(argmax_dec_func[:10]))
```

```
print("argmax combined with classes_: {}".format(
    logreg.classes_[argmax_dec_func][:10]))
```

**Out[118]:**

```
unique classes in training data: ['setosa' 'versicolor' 'virginica']
predictions: ['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor'
  'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
argmax of decision function: [1 0 2 1 1 0 1 2 1 1]
argmax combined with classes_: ['versicolor' 'setosa' 'virginica' 'versicolor'
  'versicolor' 'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
```

# Résumé et perspectives

Nous avons commencé ce chapitre par une discussion sur la complexité des modèles, puis nous avons traité de *généralisation*, autrement dit de l'entraînement d'un modèle qui soit capable de donner de bons résultats avec des données nouvelles, qui n'ont pas encore été rencontrées. Cela nous a conduit aux concepts de sous-apprentissage, qui décrit un modèle qui n'est pas capable de capturer les variations présentes dans le jeu d'apprentissage, et de surapprentissage, qui décrit un modèle se concentrant trop fortement sur les données d'apprentissage et qui n'est pas capable d'une bonne généralisation avec de nouvelles données.

Nous avons ensuite étudié une large palette de modèles d'apprentissage automatique pour la classification et la régression, vu quels sont leurs avantages et leurs inconvénients, et comment contrôler leur complexité. Nous avons noté qu'il est important de définir judicieusement les paramètres pour la plupart des algorithmes afin d'obtenir de bonnes performances. Certains algorithmes sont aussi sensibles à la façon dont nous représentons les données, et en particulier à la manière dont les caractéristiques sont équilibrées. C'est pourquoi appliquer à l'aveugle un algorithme à un jeu de données sans comprendre les suppositions faites par le modèle et la signification de ses paramètres conduit rarement à de bons résultats.

Ce chapitre contient de nombreuses informations sur les algorithmes, et il n'est pas nécessaire de vous rappeler de tous ces détails pour les chapitres suivants. Cependant, une certaine connaissance des modèles décrits ici, et notamment savoir lequel utiliser dans une situation spécifique, est important pour bien appliquer en pratique les techniques de l'apprentissage automatique. Voici un résumé rapide des moments les mieux adaptés à l'application de chaque modèle :

*Plus proches voisins*

Pour les petits jeux de données, bien comme méthode de base, facile à expliquer.

*Modèles linéaires*

Bien comme premier algorithme à essayer, bien aussi pour de très gros jeux de données, ou encore pour des données avec un grand nombre de dimensions.

*Bayésiens naïfs*

Uniquement pour la classification. Plus rapide même que les modèles linéaires, bon pour de très gros jeux de données ainsi que pour des données avec un grand nombre de dimensions. Souvent moins exact que les modèles linéaires.

*Arbres de décision*

Très rapides, ne nécessitent pas de recalibrage des données, peuvent être visualisés et facilement expliqués.

*Forêts aléatoires*

Pratiquement toujours meilleures que les arbres de décision uniques, elles sont très robustes et puissantes. Ne nécessitent pas de recalibrage des données. Pas adaptées pour des données éparses avec de multiples dimensions.

*Boosting d'arbres de décision*

Souvent légèrement plus exact que les forêts aléatoires. Plus lent à entraîner, mais plus rapide pour les prédictions que les forêts aléatoires, et moins gourmand en mémoire. Nécessite un ajustement plus poussé des paramètres que les forêts aléatoires.

*Séparateurs à vaste marge (SVM)*

Puissants pour les jeux de données de taille moyenne avec des caractéristiques dont le sens est similaire. Nécessitent un recalibrage des données, et sont sensibles aux paramètres.

*Réseaux de neurones*

Peuvent produire des modèles très complexes, particulièrement pour de gros jeux de données. Sensibles au calibrage des données et au choix des paramètres. Les grands modèles nécessitent un long temps d'entraînement.

Lorsque l'on travaille avec un nouveau jeu de données, il est souvent préférable de commencer par un modèle simple, par exemple un modèle linéaire ou bayésien naïf, ou encore un classifieur des plus proches voisins, et voir jusqu'où vous pouvez aller. Une fois que vous maîtrisez mieux vos données, vous pouvez envisager de passer à un algorithme capable de construire des modèles plus complexes, comme des forêts aléatoires, des boosting d'arbres de décision, des SVM ou des réseaux de neurones.

Vous devriez maintenant être en mesure d'avoir une idée plus précise de la manière d'appliquer, ajuster et analyser les modèles présentés dans ce qui précède. Dans ce chapitre, nous nous sommes concentrés sur le cas de la classification binaire, car il est généralement plus facile à comprendre. La plupart des algorithmes que nous avons vus ont cependant des variantes pour la classification et la régression, et tous les algorithmes de classification supportent aussi bien une version binaire que multiclasse. Essayez d'appliquer ces algorithmes aux jeux de données prédéfinis de `scikit-learn`, comme `boston_housing` ou `diabetes` pour la régression, ou encore

`digits` pour la classification multiclasse. Jouer avec ces algorithmes en les appliquant à différents jeux de données vous donnera un meilleur aperçu de leur rapidité d'apprentissage, de la facilité avec laquelle les modèles peuvent être analysés, ou encore de leur sensibilité à la manière de représenter les données.

Bien que nous ayons analysé les conséquences des différentes configurations de paramètres pour les algorithmes que nous avons étudiés, construire dans un système de production un modèle qui se généralise bien lorsqu'on lui présente de nouvelles données est un peu plus délicat que cela. Nous verrons dans le Chapitre 6 comment ajuster correctement les paramètres et comment trouver automatiquement les bonnes valeurs.

Mais, pour l'instant, nous allons nous intéresser, dans le prochain chapitre, à l'apprentissage non supervisé et au prétraitement.

CHAPITRE 3

# Apprentissage non supervisé et prétraitement

La seconde famille d'algorithmes d'apprentissage automatique dont nous allons discuter concerne ce que l'on appelle l'apprentissage non supervisé. Celui-ci englobe toutes sortes de situations dans lesquelles il n'y a pas de sortie connue, et pas de professeur pour guider l'algorithme d'apprentissage. Cet algorithme reçoit simplement des données en entrée, et on lui demande d'en extraire des connaissances à la sortie.

## Types d'apprentissages non supervisés

Nous allons nous intéresser dans ce chapitre à deux types d'apprentissage non supervisé : les transformations du jeu de données et le *clustering* (ou encore *partitionnement des données*).

Les *transformations non supervisées* d'un jeu de données sont des algorithmes qui créent une nouvelle représentation des données pouvant être plus facilement compréhensible par des humains ou par d'autres algorithmes d'apprentissage automatique en comparaison de la représentation originale des données. Une application courante des transformations non supervisées est la réduction de dimension, qui part d'une représentation des données avec de nombreuses dimensions, c'est-à-dire un grand nombre de caractéristiques, pour trouver une nouvelle manière de représenter ces données afin d'en déduire ce qui est essentiel, et donc de réduire fortement la quantité de caractéristiques. Cette réduction de dimension est souvent utilisée pour obtenir une représentation en deux dimensions à des fins de visualisation.

Une autre application des transformations non supervisées consiste à trouver les parties ou composants qui structurent les données, par exemple pour extraire des thèmes dans des collections de documents textuels. Ici, la tâche consiste à trouver les sujets inconnus par avance qui sont traités dans chaque document, et plus généralement à apprendre quels sujets apparaissent dans chaque document. Cela peut être utile pour détecter des thèmes de discussion, qu'il s'agisse par

exemple d'élections, de faits de société ou encore de consommation sur des réseaux sociaux, et ainsi de suite.

D'un autre côté, les *algorithmes de clustering* partitionnent les données en groupes distincts d'éléments similaires. Prenons comme exemple le chargement de photos sur le site d'un réseau social. Pour vous permettre d'organiser vos images, ce réseau pourrait essayer de regrouper automatiquement les clichés qui montrent la même personne. Pour autant, le site ne sait pas quelles images montrent qui, et il ne sait pas non plus combien de personnes apparaissent dans votre collection de photographies. Une approche raisonnable pourrait alors consister à extraire tous les visages et à les diviser en un certain nombre de groupes formés de vues similaires. Avec un peu de chance (et surtout beaucoup d'algorithmique), une certaine personne sera classifiée dans le bon groupe, et le site vous proposera donc un classement cohérent de vos images.

## Les défis de l'apprentissage non supervisé

Un défi majeur de l'apprentissage non supervisé est de pouvoir évaluer si l'algorithme a appris quelque chose d'utile. Les algorithmes d'apprentissage non supervisé sont en règle générale appliqués à des données qui ne contiennent aucune information d'étiquette, et donc vous ne pouvez pas savoir ce que devrait être le bon résultat. Il est par conséquent très difficile de dire si un modèle s'est « bien comporté ». Par exemple, notre algorithme de clustering ci-dessus pourrait avoir regroupé d'un côté tous les visages qui sont vus de profil, et d'un autre tous ceux qui se présentent de face. Il s'agit bien entendu de quelque chose qui a un certain sens, mais ce n'est évidemment pas ce qui peut nous intéresser. Cependant, nous n'avons aucun moyen de « dire » à l'algorithme ce que nous recherchons, et souvent la seule manière d'évaluer le résultat d'un algorithme non supervisé, c'est de l'inspecter manuellement.

Par voie de conséquence, les algorithmes non supervisés sont souvent employés dans une démarche exploratoire, lorsqu'un *data scientist* veut mieux comprendre les données qui lui sont présentées, plutôt que dans un système automatisé plus vaste. Une autre application courante de ces algorithmes est de s'en servir comme phase de prétraitement des données pour la construction de modèles supervisés. Former une nouvelle représentation des données peut parfois améliorer l'exactitude des algorithmes supervisés, et/ou permettre de réduire les temps de traitement ainsi que les besoins en mémoire.

Avant de nous lancer avec de « vrais » algorithmes d'apprentissage non supervisé, nous allons brièvement discuter de certaines méthodes pratiques pour assurer un formatage approprié des données. Ce prétraitement ainsi que le recalibrage des données sont certes souvent utilisés en tandem avec les algorithmes d'apprentissage supervisé. Mais les méthodes de recalibrage ne se servent pas des informations supervisées, et sont donc à ce titre non supervisées.

# Prétraitement et recalibrage

Dans le chapitre précédent, nous avons vu que certains algorithmes, comme les réseaux de neurones et les SVM, sont très sensible à tout ce qui concerne l'échelle des valeurs des données. Une pratique courante consiste donc à ajuster les caractéristiques de manière à ce que la représentation des données soit mieux adaptée aux besoins de ces algorithmes. Souvent, il suffit pour cela d'un simple rééchelonnement et d'un décalage éventuel des valeurs. Le code suivant montre un exemple simple de cette problématique (voir la Figure 3.1) :

**In[1] :**

```
mglearn.plots.plot_scaling()
```

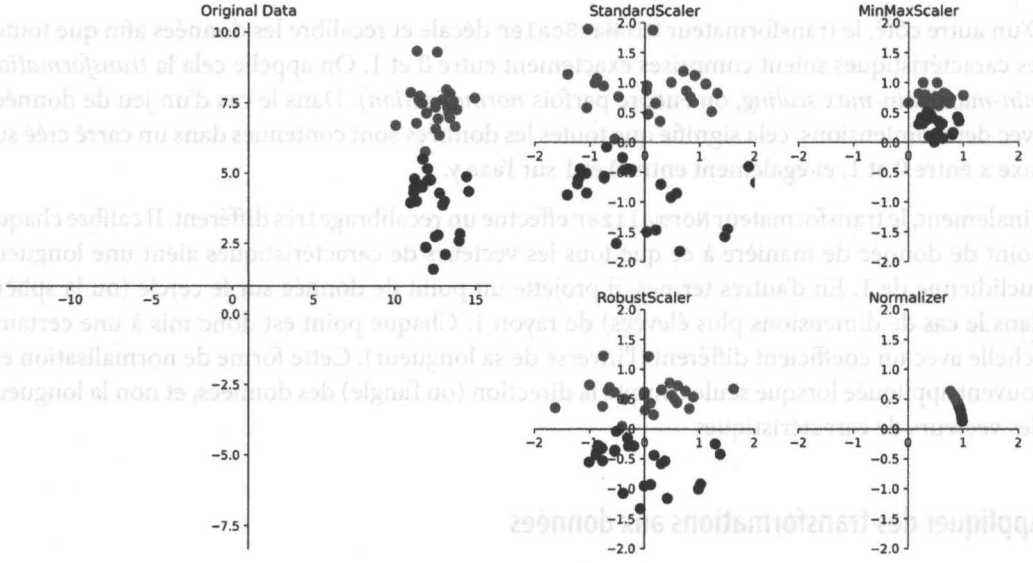



*Figure 3.1 : Différentes manières de prétraiter et de recalibrer un jeu de données.*

## Différents types de prétraitement

Le premier tracé de la Figure 3.1 montre un jeu de données synthétique pour la classification, avec deux classes et deux caractéristiques. La première (l'axe x) a des valeurs comprises entre 10 et 15. La seconde caractéristique (l'axe y) s'échelonne d'environ 1 à 9.

Les quatre tracés qui suivent montrent quatre manières de transformer les données afin d'obtenir des intervalles plus standard. Le *transformateur* de `scikit-learn` appelé `StandardScaler` s'assure que, pour chaque caractéristique, la moyenne soit nulle et la variance égale à 1, et donc que toutes les caractéristiques aient la même amplitude. Cependant, cela ne permet pas de for-

mer des valeurs minimale et maximale particulières pour les caractéristiques. Le transformateur `RobustScaler` fonctionne de la même façon que `StandardScaler` dans le sens où il assure pour chaque caractéristique des propriétés statistiques qui garantissent qu'elles se trouvent sur la même échelle. Cependant, `RobustScaler` utilise la médiane et les quartiles au lieu de la moyenne et de la variance. De ce fait, `RobustScaler` ignore les points de donnée qui sont différents du reste (par exemple ceux qui correspondent à des erreurs de mesure). Ces points mal formés sont aussi dits aberrants (*outliers*) et peuvent occasionner des problèmes pour d'autres techniques de calibrage.



La médiane d'un ensemble de données numériques est la valeur $x$ pour laquelle la moitié des nombres est plus petite que $x$, et l'autre moitié plus grande que $x$. Le premier quartile est la valeur $x$ pour laquelle un quart des nombres est plus petit que $x$, le dernier quartile étant celui pour lequel un quart des nombres est plus grand que $x$.

D'un autre côté, le transformateur `MinMaxScaler` décale et recalibre les données afin que toutes les caractéristiques soient comprises exactement entre 0 et 1. On appelle cela la *transformation min-max* (*min-max scaling*, ou encore parfois *normalization*). Dans le cas d'un jeu de données avec deux dimensions, cela signifie que toutes les données sont contenues dans un carré créé sur l'axe x entre 0 et 1, et également entre 0 et 1 sur l'axe y.

Finalement, le transformateur `Normalizer` effectue un recalibrage très différent. Il calibre chaque point de donnée de manière à ce que tous les vecteurs de caractéristiques aient une longueur euclidienne de 1. En d'autres termes, il projette un point de donnée sur le cercle (ou la sphère dans le cas de dimensions plus élevées) de rayon 1. Chaque point est donc mis à une certaine échelle avec un coefficient différent (l'inverse de sa longueur). Cette forme de normalisation est souvent appliquée lorsque seule compte la direction (ou l'angle) des données, et non la longueur des vecteurs de caractéristiques.

## Appliquer des transformations aux données

Maintenant que nous avons vu ce que font les différents types de transformations, essayons de mettre ces notions en pratique avec `scikit-learn`. Nous allons pour cela utiliser le jeu de données `cancer`, déjà vu à plusieurs reprises dans le Chapitre 2. Les méthodes de prétraitement, comme les transformateurs, sont généralement appliquées avant de faire appel à un algorithme d'apprentissage automatique supervisé. Supposons par exemple que nous voulions appliquer le SVM à noyau (SVC) au jeu de données `cancer`, et utiliser le transformateur `MinMaxScaler` pour le prétraitement de ces données. Nous commencerons par charger notre jeu de données et par le diviser entre un jeu d'apprentissage et un jeu de test (nous avons besoin de procéder ainsi pour pouvoir évaluer le modèle supervisé qui sera construit à partir des données recalibrées) :

**In[2] :**

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
```

```
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=1)
print(X_train.shape)
print(X_test.shape)
```

**Out[2]:**

```
(426, 30)
(143, 30)
```

Souvenez-vous que le jeu `cancer` contient 569 points de données, chacun étant représenté par 30 mesures. Nous divisons ce jeu en 426 échantillons pour le jeu d'apprentissage et 143 échantillons pour le jeu de test (ce qui correspond à la proportion par défaut de 75 % d'un côté et 25 % de l'autre).

Comme pour les modèles supervisés que nous avons construits précédemment, nous allons commencer par importer la classe qui implémente le prétraitement puis l'instancier :

**In[3]:**

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

Nous allons maintenant appliquer à cet objet la méthode `fit` sur le jeu d'apprentissage. Pour le transformateur `MinMaxScaler`, la méthode `fit` calcule la valeur minimale et maximale de chaque caractéristique du jeu d'apprentissage. Contrairement aux cas de classification et de régression étudiés dans le Chapitre 2, nous n'avons besoin ici que de passer en paramètre `X_train`. Par conséquent, `y_train` n'est pas utilisé :

**In[4]:**

```
scaler.fit(X_train)
```

**Out[4]:**

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

Pour appliquer la transformation voulue, autrement dit pour effectuer le recalibrage des données d'apprentissage, nous utilisons la méthode `transform` de l'objet `scaler`. Cette méthode est employée dans `scikit-learn` chaque fois qu'un modèle retourne une nouvelle représentation des données :

**In[5]:**

```
X_train_scaled = scaler.transform(X_train)
print("transformed shape: {}".format(X_train_scaled.shape))
print("per-feature minimum before scaling:\n {}".format(X_train.min(axis=0)))
print("per-feature maximum before scaling:\n {}".format(X_train.max(axis=0)))
print("per-feature minimum after scaling:\n {}".format(
    X_train_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n {}".format(
```

```
    X_train_scaled.max(axis=0)))
```

**Out[5]:**

```
transformed shape: (426, 30)
per-feature minimum before scaling:
 [  6.981    9.71    43.79   143.5      0.053    0.019    0.       0.
    0.106    0.05     0.115    0.36     0.757    6.802    0.002    0.002
    0.       0.       0.01     0.001    7.93    12.02    50.41   185.2
    0.071    0.027    0.       0.       0.157    0.055]
per-feature maximum before scaling:
 [  28.11     39.28    188.5    2501.       0.163     0.287     0.427
    0.201     0.304     0.096    2.873      4.885    21.98     542.2
    0.031     0.135     0.396    0.053      0.061     0.03     36.04
   49.54    251.2    4254.       0.223      0.938     1.17      0.291
    0.577     0.149]
per-feature minimum after scaling:
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
per-feature maximum after scaling:
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

Les données transformées ont le même format que les données d'origine. Les caractéristiques sont simplement décalées et recalibrées. Elles sont maintenant toutes comprises entre 0 et 1, ce que nous voulions effectivement obtenir.

Pour appliquer le SVM à nos données recalibrées, nous devons également transformer le jeu de test. Pour cela, nous appelons à nouveau la méthode `transform`, mais cette fois en lui passant `X_test` :

**In[6]:**

```
X_test_scaled = scaler.transform(X_test)
print("per-feature minimum after scaling:\n{}".format(X_test_scaled.
  min(axis=0)))
print("per-feature maximum after scaling:\n{}".format(X_test_scaled.
  max(axis=0)))
```

**Out[6]:**

```
per-feature minimum after scaling:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.       0.       0.154 -0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.       0.      -0.032  0.007
  0.027  0.058  0.02   0.009  0.109  0.026  0.       0.      -0.     -0.002]
per-feature maximum after scaling:
[ 0.958  0.815  0.956  0.894  0.811  1.22   0.88   0.933  0.932  1.037
  0.427  0.498  0.441  0.284  0.487  0.739  0.767  0.629  1.337  0.391
  0.896  0.793  0.849  0.745  0.915  1.132  1.07   0.924  1.205  1.631]
```

D'une manière peut-être quelque peu surprenante, vous pouvez voir que, pour le jeu de test, les valeurs minimale et maximale après recalibrage ne sont pas égales à 0 et 1. Certaines des caractéristiques se trouvent même en dehors de l'intervalle 0-1 ! L'explication est que `MinMaxScaler` (de même d'ailleurs que tous les autres transformateurs) applique toujours exactement la même

transformation au jeu d'apprentissage et au jeu de test. Cela signifie donc que la méthode `transform` soustrait la valeur minimale du jeu d'apprentissage, puis effectue la division par la plage *maximum-minimum* de ce même jeu. Évidemment, ces valeurs peuvent être tout à fait différentes de ce qui se passe pour le jeu de test.

## Recalibrer de la même manière le jeu d'apprentissage et le jeu de test

Il est important d'appliquer exactement la même transformation au jeu d'apprentissage et au jeu de test pour que le modèle supervisé puisse fonctionner avec ce dernier. L'exemple qui suit illustre ce qui se passerait si nous utilisions à la place le minimum et la plage de valeurs du jeu de test (voir la Figure 3.2) :

**In[7]:**

```
from sklearn.datasets import make_blobs
X, _ = make_blobs(n_samples=50, centers=5, random_state=4, cluster_std=2)
X_train, X_test = train_test_split(X, random_state=5, test_size=.1)

fig, axes = plt.subplots(1, 3, figsize=(13, 4))
axes[0].scatter(X_train[:, 0], X_train[:, 1],
                c=mglearn.cm2(0), label="Training set", s=60)
axes[0].scatter(X_test[:, 0], X_test[:, 1], marker='^',
                c=mglearn.cm2(1), label="Test set", s=60)
axes[0].legend(loc='upper left')
axes[0].set_title("Original Data")

scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

axes[1].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mglearn.cm2(0), label="Training set", s=60)
axes[1].scatter(X_test_scaled[:, 0], X_test_scaled[:, 1], marker='^',
                c=mglearn.cm2(1), label="Test set", s=60)
axes[1].set_title("Scaled Data")

# NE FAITES PAS CECI : c'est juste à titre d'illustration !
test_scaler = MinMaxScaler()
test_scaler.fit(X_test)
X_test_scaled_badly = test_scaler.transform(X_test)

axes[2].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mglearn.cm2(0), label="training set", s=60)
axes[2].scatter(X_test_scaled_badly[:, 0], X_test_scaled_badly[:, 1],
                marker='^', c=mglearn.cm2(1), label="test set", s=60)
axes[2].set_title("Improperly Scaled Data")

for ax in axes:
```

```
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
fig.tight_layout()
```

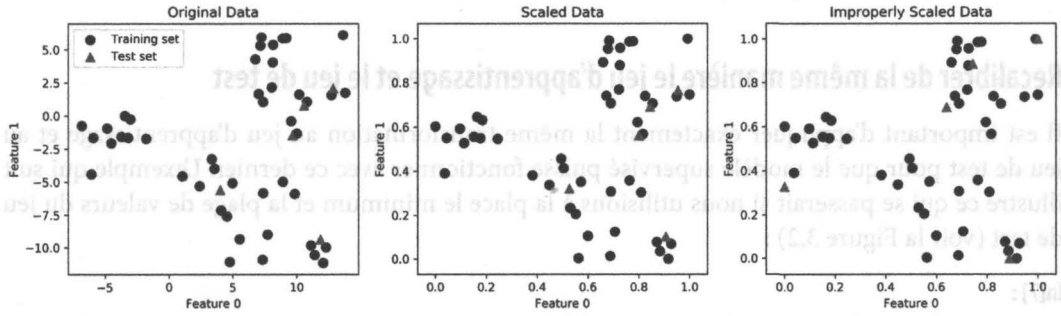



*Figure 3.2 : Effet produit en recalibrant les données d'apprentissage et de test montrées à gauche simultanément (tracé du milieu) et séparément (tracé de droite).*

Le premier panneau présente un jeu de données non recalibrées à deux dimensions, le jeu d'apprentissage étant représenté par des cercles et le jeu de test par des triangles. Le second panneau reprend les mêmes données, mais cette fois après application du transformateur `MinMaxScaler`. Ici, nous avons appelé la méthode `fit` sur le jeu d'apprentissage, puis appelé la méthode `transform` sur les deux jeux. Vous pouvez remarquer la similitude avec le premier tracé. Seules les échelles sur les axes de coordonnées ont été modifiées de manière à ce que toutes les caractéristiques soient comprises entre 0 et 1. Notez également que les valeurs minimale et maximale pour les caractéristiques du jeu de test ne sont pas égales à 0 et à 1.

Le troisième panneau montre ce qui se passerait si nous avions recalibré séparément les jeux d'apprentissage et de test. Dans ce cas, le minima et le maxima sont respectivement de 0 et de 1, et ce pour les deux jeux. Mais, maintenant, nos jeux de données semblent différents. Les points de test se sont déplacés d'une manière incongrue par rapport aux points d'apprentissage, du fait précisément d'un recalibrage différent. Nous avons modifié la disposition des données d'une façon arbitraire. Clairement, ce n'est pas ce que nous voulons faire.

Une autre manière de réfléchir à cette question consiste à se représenter le jeu de test comme un point unique. Il n'y a bien sûr aucun moyen de recalibrer correctement un seul point pour qu'il remplisse les conditions de minima et de maxima de `MinMaxScaler`. Mais la taille de votre jeu de test ne devrait pas modifier le traitement appliqué.

### Raccourcis et alternatives efficaces

Souvent, vous voulez ajuster un modèle sur un certain jeu de données, puis le transformer. C'est une tâche très courante, qui peut dans bon nombre de situations être traitée plus efficacement qu'en appelant successivement les méthodes `fit` puis `transform`. Pour ce type de cas, tous les modèles qui ont une méthode `transform` possèdent également une méthode `fit_transform`. Prenons un exemple en utilisant `StandardScaler` :

**In[8]:**

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit(X_train).transform(X_train)
X_scaled_d = scaler.fit_transform(X_train)
```

Même si `fit_transform` n'est pas nécessairement la méthode la plus efficace pour tous les modèles, l'utiliser pour essayer de transformer le jeu d'apprentissage est une bonne pratique.

## Effet du prétraitement sur l'apprentissage supervisé

Revenons une fois de plus au jeu de données `cancer`, et voyons l'effet de `MinMaxScaler` sur l'entraînement du SVC (il s'agit d'une manière différente d'effectuer le même calibrage que dans le Chapitre 2). Tout d'abord, appliquons à nouveau la méthode `fit` sur les données d'origine à titre de comparaison :

**In[9]:**

```
from sklearn.svm import SVC

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=0)

svm = SVC(C=100)
svm.fit(X_train, y_train)
print("Test set accuracy: {:.2f}".format(svm.score(X_test, y_test)))
```

**Out[9]:**

```
Test set accuracy: 0.63
```

Reprenons maintenant la procédure, mais en utilisant d'abord le transformateur `MinMaxScaler` :

**In[10]:**

```
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

svm.fit(X_train_scaled, y_train)
```

```
print("Scaled test set accuracy: {:.2f}".format(
    svm.score(X_test_scaled, y_test)))
```

**Out[10]:**

```
Scaled test set accuracy: 0.97
```

Une nouvelle fois, l'effet produit par le recalibrage des données est plutôt significatif. Même si ce genre de travail ne nécessite pas des connaissances mathématiques très pointues, utiliser les mécanismes de recalibrage des données fournis par `scikit-learn` au lieu de les réimplémenter vous-même est une bonne pratique, car il est facile de commettre des erreurs, même dans des calculs aussi simples.

Vous pouvez également remplacer un algorithme de prétraitement par un autre en changeant la classe que vous utilisez. En effet, toutes les classes de prétraitement présentent la même interface formée des méthodes `fit` et `transform` :

**In[11]:**

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

svm.fit(X_train_scaled, y_train)

print("SVM test accuracy: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

**Out[11]:**

```
SVM test accuracy: 0.96
```

Maintenant que nous avons vu comment fonctionnent les transformations pour le prétraitement des données, passons à un stade plus élaboré et plus intéressant des transformations en utilisant l'apprentissage non supervisé.

## Réduction de la dimension, extraction de caractéristiques et apprentissage de variétés

Comme nous l'avons déjà vu plus haut, la transformation de données en utilisant l'apprentissage non supervisé peut avoir de multiples motivations. Les plus courantes sont la visualisation, la compression des données et la recherche d'une représentation qui soit plus informative pour des traitements ultérieurs.

L'un des algorithmes les plus simples et les plus largement employés à cet effet est l'analyse en composantes principales (ou PCA, pour l'anglais *principal component analysis*). Nous nous intéresserons également à deux autres algorithmes : la factorisation en matrice non négative (ou

NMF, pour *non-negative matrix factorization* en anglais), qui est couramment, algorithme utilisé pour l'extraction de caractéristiques, et t-SNE (*t-distributed stochastic neighbor embedding*) qui est également couramment employé pour la visualisation *via* des nuages de points en deux dimensions.

## Analyse en composantes principales (PCA)

L'analyse en composantes principales est une méthode qui transforme des caractéristiques dites *corrélées* (disons, liées entre elles) en caractéristiques statistiquement non corrélées. Cette phase est souvent suivie par la sélection d'un sous-ensemble des nouvelles caractéristiques, selon l'importance qu'elles présentent pour la compréhension des données. L'exemple qui suit illustre l'effet de la méthode PCA sur un jeu de données synthétique possédant deux dimensions (voir la Figure 3.3) :

**In[12]:**

```
mglearn.plots.plot_pca_illustration()
```

Le premier tracé, en haut et à gauche, montre les points de données d'origine, l'emploi de nuances de gris permettant de mieux les distinguer. L'algorithme commence par trouver la direction du maximum de la variance (appelée Component 1 sur la figure). Il s'agit de la direction (ou du vecteur) dans les données qui contient la plus grande part d'informations ou, en d'autres termes, de la direction dans laquelle les caractéristiques sont les plus corrélées les unes avec les autres. L'algorithme trouve ensuite la direction qui contient le plus d'informations tout en restant orthogonale (formant un angle droit) à la première. Dans un espace à deux dimensions, il n'existe qu'une seule orientation formant un angle droit. Par contre, dans des espaces possédant de multiples dimensions, il y aurait un nombre (infiniment) plus important de directions orthogonales. Bien que les deux composantes soient représentées par des flèches, leur orientation n'a aucune importance. Par exemple, la première aurait parfaitement pu être dirigée vers le haut et la gauche, au lieu du bas et de la droite. Les directions déterminées par ce processus sont appelées *composantes principales*, car elles déterminent les principales directions de la variance dans les données. En général, il existe autant de composantes principales que de caractéristiques originales.

Le second tracé, en haut et à droite, montre les mêmes données, mais après rotation de manière à ce que la première composante principale soit alignée avec l'axe x, et la seconde avec l'axe y. Avant la rotation, la moyenne a été soustraite des données de manière à ce que celles-ci soient centrées autour de zéro. Dans la représentation ainsi trouvée par l'algorithme PCA, les deux axes sont non corrélés, ce qui signifie que la matrice de corrélation des données est ici nulle, sauf pour la diagonale.

Nous pouvons utiliser PCA pour la réduction de dimension en ne retenant que certaines des composantes principales. Dans cet exemple, nous pourrions ainsi conserver uniquement la première composante principale, comme l'illustre le troisième tracé de la Figure 3.3. Cela réduit le

jeu de données à une seule dimension au lieu de deux. Notez cependant que, au lieu de garder une seule des caractéristiques d'origine, nous avons trouvé la direction la plus intéressante (du haut et à gauche vers le bas et à droite dans le premier tracé) et conservé cette direction, c'est-à-dire la première composante principale.

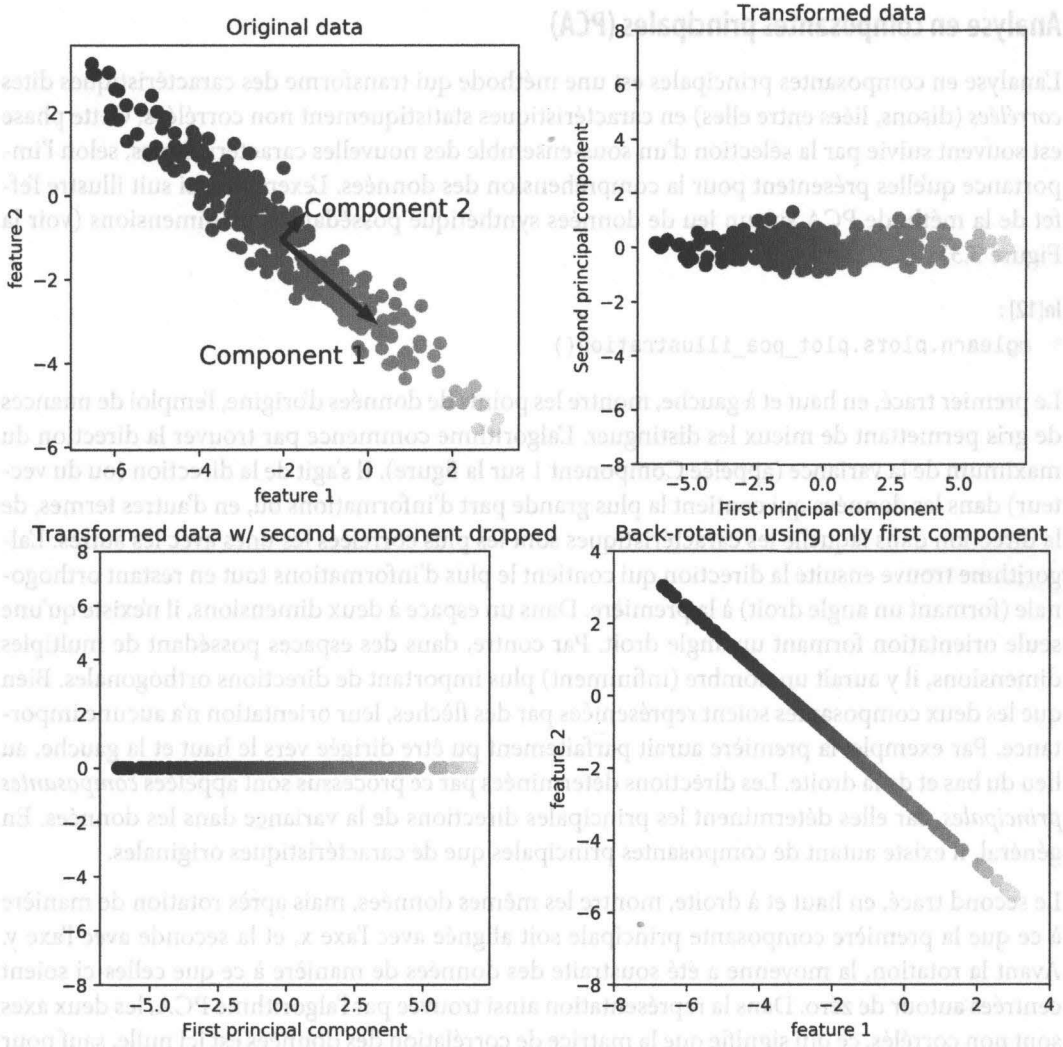



*Figure 3.3 : Transformation de données avec PCA.*

Finalement, nous pouvons annuler la rotation et réintroduire la moyenne dans les données. C'est ce qu'illustre le dernier tracé de la Figure 3.3. Ces points se trouvent dans l'espace d'origine des caractéristiques, mais nous n'avons gardé que les informations contenues dans la première

composante principale. Cette transformation sert parfois à supprimer des effets de bruit dans les données, ou à visualiser quelle partie de l'information est retenue en utilisant les composantes principales.

### Appliquer PCA au jeu de données cancer pour la visualisation

L'une des applications les plus courantes de PCA est la visualisation de jeux de données multi-dimensionnels. Comme nous l'avons vu dans le Chapitre 1, il est difficile de former des nuages de points à partir de données qui possèdent plus de deux caractéristiques. Dans le cas du jeu de données Iris, nous avions pu produire une vue partielle des données en montrant toutes les combinaisons possibles de deux caractéristiques (reportez-vous à la Figure 1.3). Mais cette méthode devient très délicate dans le cas d'un jeu de données tel que Breast Cancer. En effet, ce jeu de donnée contient 30 caractéristiques, ce qui demanderait 30 * 14 = 420 nuages de points ! Nous ne pourrions jamais regarder en détail tous ces tracés, sans même parler d'essayer de les comprendre.

Mais il est cependant possible d'utiliser une visualisation encore plus simple en calculant les histogrammes de chacune des caractéristiques pour les deux classes (tumeur bénigne et tumeur maligne). C'est ce qu'illustre la Figure 3.4 :

**In[13]:**

```
fig, axes = plt.subplots(15, 2, figsize=(10, 20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]

ax = axes.ravel()

for i in range(30):
    _, bins = np.histogram(cancer.data[:, i], bins=50)
    ax[i].hist(malignant[:, i], bins=bins, color=mglearn.cm3(0), alpha=.5)
    ax[i].hist(benign[:, i], bins=bins, color=mglearn.cm3(2), alpha=.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
ax[0].set_xlabel("Feature magnitude")
ax[0].set_ylabel("Frequency")
ax[0].legend(["malignant", "benign"], loc="best")
fig.tight_layout()
```

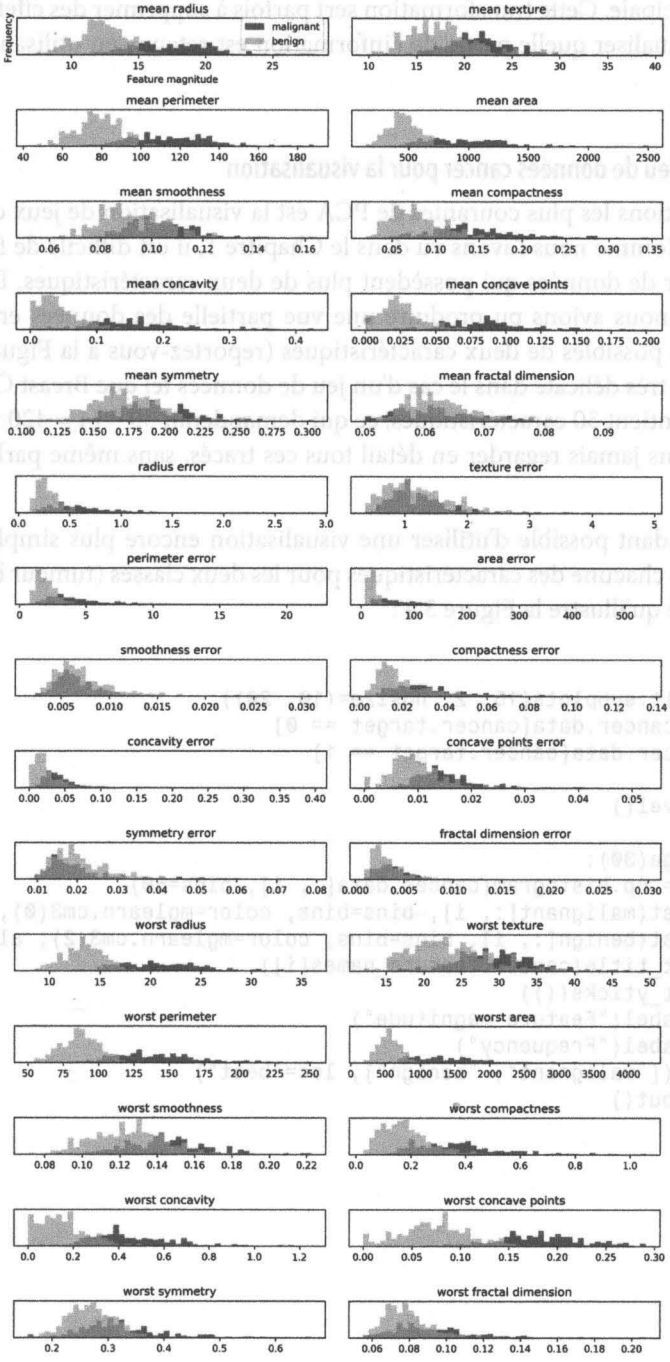



*Figure 3.4 : Histogrammes des caractéristiques par classe pour le jeu de données Breast Cancer.*

Ici, nous avons créé un histogramme pour chacune des caractéristiques, en comptant le nombre d'apparitions des points de données dans une certaine plage de valeurs (ce que l'on appelle aussi un *bin*). Chaque tracé contient deux histogrammes : un pour les points de la classe bénigne (en clair) et un pour les points de la classe maligne (en foncé). Cela nous donne une certaine idée de la manière dont les caractéristiques sont distribuées dans les deux classes, et donc nous aide à nous risquer dans un pronostic quant aux caractéristiques qui permettent le mieux de distinguer dans les échantillons ceux qui correspondent à un cas bénin ou à un cas malin. Par exemple, la caractéristique « smoothness error » ne semble pas fournir d'information utile du fait que les deux histogrammes se superposent presque entièrement. En revanche, ces histogrammes sont bien distincts pour la caractéristique « worst concave points », qui est donc susceptible d'être bien plus informative.

Cependant, ce tracé ne nous dit rien sur les interactions entre les variables et sur leurs relations avec les classes. En utilisant PCA, nous pouvons capturer les principales interactions et former un tableau légèrement plus complet. En effet, PCA nous permet de trouver les deux premières composantes principales et de visualiser les données dans ce nouvel espace en deux dimensions à l'aide d'un seul diagramme de dispersion.

Mais, auparavant, nous allons normaliser nos données à l'aide de `StandardScaler` de manière à ce que chaque caractéristique ait une variance égale à 1 :

**In[14]:**

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()

scaler = StandardScaler()
scaler.fit(cancer.data)
X_scaled = scaler.transform(cancer.data)
```

La transformation PCA se traite comme une transformation de prétraitement quelconque. Nous instancions l'objet PCA, nous trouvons les composantes principales en appelant la méthode `fit`, puis nous appliquons la rotation et la réduction de dimension en appelant `transform`. Par défaut, PCA se charge de la rotation et du décalage des données, mais en conservant toutes les composantes principales. Pour réduire la dimensionnalité des données, nous devons spécifier, lors de la création de l'objet PCA, le nombre de composantes que nous voulons conserver :

**In[15]:**

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X_scaled)

X_pca = pca.transform(X_scaled)
print("Original shape: {}".format(str(X_scaled.shape)))
print("Reduced shape: {}".format(str(X_pca.shape)))
```

**Out[15]:**

```
Original shape: (569, 30)
Reduced shape: (569, 2)
```

Nous pouvons maintenant tracer les deux composantes principales (voir la Figure 3.5) :

**In[16]:**

```
plt.figure(figsize=(8, 8))
mglearn.discrete_scatter(X_pca[:, 0], X_pca[:, 1], cancer.target)
plt.legend(cancer.target_names, loc="best")
plt.gca().set_aspect("equal")
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

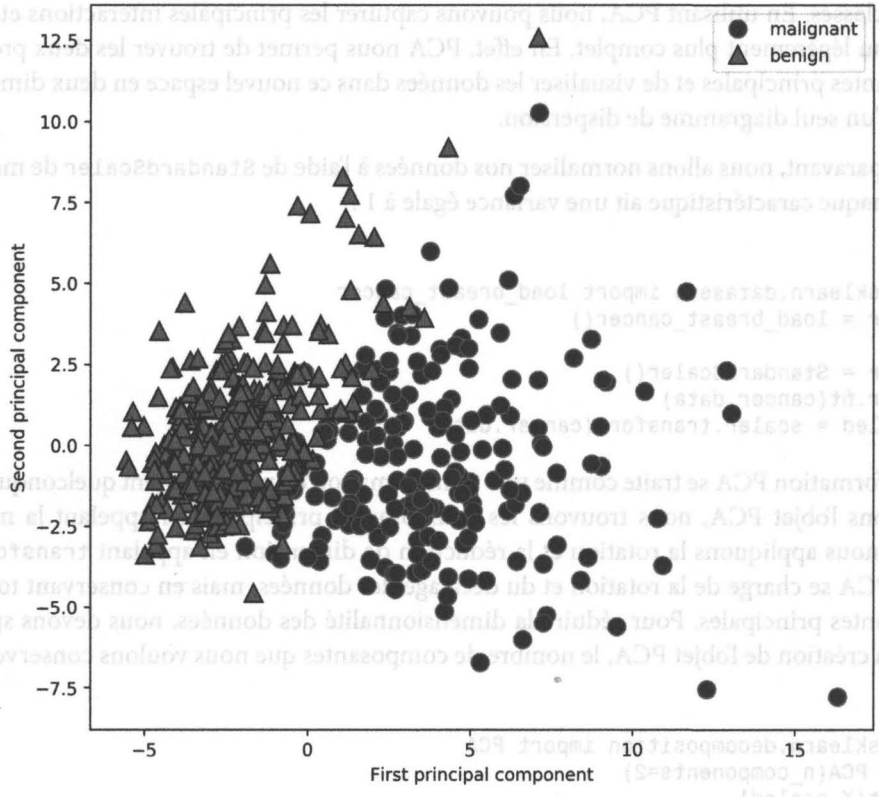



*Figure 3.5 : Nuage de points en deux dimensions du jeu de données Breast Cancer utilisant les deux premières composantes principales.*

Il est important de noter que PCA est une méthode non supervisée, et donc qu'elle n'utilise aucune information provenant des classes pour trouver la rotation. Elle recherche simplement

les corrélations dans les données. Dans le cas du nuage de points montré ici, nous avons tracé les deux composantes principales, puis utilisé les informations de classe pour finaliser l'aspect des points. Vous pouvez constater que les deux classes sont plutôt bien séparées dans cet espace en deux dimensions. Cela nous conduit à croire que même une classifieur linéaire (qui chercherait à partager cet espace à l'aide d'une ligne droite) pourrait faire un travail raisonnablement bon pour opérer une distinction entre les deux classes. Nous pouvons également voir que les points associés aux tumeurs malignes sont plus dispersés que les autres, ce qui est quelque chose que nous pouvions déjà constater sur les histogrammes de la Figure 3.4, mais de manière moins évidente.

Un inconvénient de la méthode PCA est que les deux axes du tracé ne sont souvent pas très faciles à interpréter. Les composantes principales correspondent à des directions dans les données originales, et donc elles sont des combinaisons des caractéristiques d'origine. Cependant, ces combinaisons sont généralement très complexes, comme nous allons bientôt le voir. Les composantes principales elles-mêmes sont mémorisées dans l'attribut `components_` lors de l'exécution de `fit` :

**In[17]:**

```
print("PCA component shape: {}".format(pca.components_.shape))
```

**Out[17]:**

```
PCA component shape: (2, 30)
```

Chaque ligne de l'attribut `components_` correspond à une des composantes principales, le tri étant effectué par ordre d'importance (la première composante principale vient en premier, et ainsi de suite). Les colonnes, de leur côté, correspondent aux caractéristiques d'origine, soit dans cet exemple « mean radius », « mean texture », *etc.* Jetons un coup d'œil sur le contenu de `components_` :

**In[18]:**

```
print("PCA components:\n{}".format(pca.components_))
```

**Out[18]:**

```
PCA components:
[[ 0.219  0.104  0.228  0.221  0.143  0.239  0.258  0.261  0.138  0.064
   0.206  0.017  0.211  0.203  0.015  0.17   0.154  0.183  0.042  0.103
   0.228  0.104  0.237  0.225  0.128  0.21   0.229  0.251  0.123  0.132]
 [-0.234 -0.06  -0.215 -0.231  0.186  0.152  0.06  -0.035  0.19   0.367
  -0.106  0.09  -0.089 -0.152  0.204  0.233  0.197  0.13   0.184  0.28
  -0.22  -0.045 -0.2   -0.219  0.172  0.144  0.098 -0.008  0.142  0.275]]
```

Nous pouvons également visualiser les coefficients sous la forme d'une carte de chaleur qui pourrait être plus facile à comprendre (voir la Figure 3.6) :

**In[19]:**

```
plt.matshow(pca.components_, cmap='viridis')
plt.yticks([0, 1], ["First component", "Second component"])
```

```
plt.colorbar()
plt.xticks(range(len(cancer.feature_names)),
           cancer.feature_names, rotation=60, ha='left')
plt.xlabel("Feature")
plt.ylabel("Principal components")
```

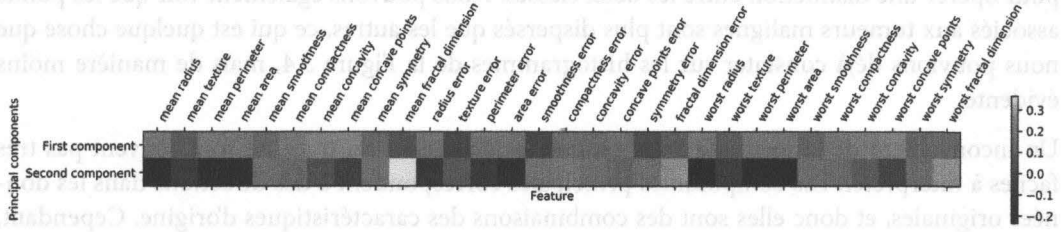



*Figure 3.6 : Carte de chaleur des deux premières composantes principales pour le jeu de données Breast Cancer.*

Vous pouvez voir que, dans la première composante, toutes les caractéristiques ont le même signe (il est positif, mais, comme nous l'avons déjà mentionné, la direction des points n'a pas d'importance particulière). Cela signifie qu'il y a une corrélation générale entre toutes les caractéristiques. Lorsqu'une mesure a une valeur élevée, les autres le seront probablement aussi. La seconde composante a des signes divers, et les deux composantes impliquent la totalité des 30 caractéristiques. Ce mélange entre toutes les caractéristiques est ce qui rend les explications sur les axes de la Figure 3.6 si difficiles à formuler.

### Extraction de caractéristiques par eigenfaces

Une autre application de la méthode PCA mentionnée plus haut est l'extraction de caractéristiques. L'idée sous-jacente est qu'il est possible de trouver une représentation des données qui soit mieux adaptée à l'analyse que le format « brut » qui vous a été fourni. Un exemple remarquable d'application dans laquelle l'extraction de données est utile est l'analyse d'images numériques. Ces images sont constituées de pixels, généralement enregistrés sous la forme de valeurs d'intensité dans le rouge, le vert et le bleu (RVB). Les objets dans les images sont habituellement constitués de milliers de pixels qui ne sont significatifs que reliés les uns aux autres.



On appelle *eigenfaces* des ensembles de vecteurs propres utilisés dans le domaine de la vision artificielle pour le traitement de la reconnaissance du visage humain. Voyez à ce sujet la page Wikipédia `https://en.wikipedia.org/wiki/Eigenface` (en anglais).

Nous allons ici vous proposer une application très simple d'extraction de caractéristiques avec PCA en travaillant avec des photographies de personnalités qui ont été téléchargées sur Internet. Le jeu de données utilisé contient des photographies de visages de célébrités dans différents domaines : politique, chanson, cinéma et sport. Ces images datent du début des années 2000.

Nous nous servirons d'une version en noir et blanc en redimensionnant les fichiers pour que les traitements soient plus rapides. La Figure 3.7 montre quelques-unes de ces images :

**In[20]:**

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape

fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
for target, image, ax in zip(people.target, people.images, axes.ravel()):
    ax.imshow(image)
    ax.set_title(people.target_names[target])
```




*Figure 3.7 : Quelques images du jeu de données intitulé Labeled Faces in the Wild.*



Toutes ces images ne sont pas installées par défaut sur votre disque dur. Lorsque vous exécutez le code ci-dessus, le téléchargement des données est lancé sur le site Labeled Faces in the Wild (`http://vis-www.cs.umass.edu/lfw/`). Le volume des données, une fois celles-ci décompressées, est de plus de 260 Mo. Ce n'est pas négligeable, et ne vous étonnez pas si ce téléchargement prend un peu de temps. Si vous voulez par la suite faire le ménage dans ces données, voyez le contenu du sous-dossier `scikit_learn_data` dans votre dossier utilisateur du disque dur.

Le jeu de données tel qu'il se présente dans notre exemple contient 3 023 images concernant 62 personnes différentes, dans un format de 87 x 65 pixels :

**In[21]:**

```
print("people.images.shape: {}".format(people.images.shape))
print("Number of classes: {}".format(len(people.target_names)))
```

**Out[21]:**

```
people.images.shape: (3023, 87, 65)
Number of classes: 62
```



Le téléchargement donne un nombre de fichiers d'image et de personnes concernées bien plus important, mais seule une fraction est utilisée ici.

En fait, le jeu de données est un peu biaisé car certaines personnalités y sont représentées bien plus que d'autres :

**In[22]:**

```
counts = np.bincount(people.target)
for i, (count, name) in enumerate(zip(counts, people.target_names)):
    print("{0:25} {1:3}".format(name, count), end='   ')
    if (i + 1) % 3 == 0:
        print()
```

**Out[22]:**

```
Alejandro Toledo          39  Alvaro Uribe              35  Amelie Mauresmo           21
Andre Agassi              36  Angelina Jolie            20  Ariel Sharon              77
Arnold Schwarzenegger     42  Atal Bihari Vajpayee      24  Bill Clinton              29
Carlos Menem              21  Colin Powell             236  David Beckham             31
Donald Rumsfeld          121  George Robertson          22  George W Bush            530
Gerhard Schroeder        109  Gloria Macapagal Arroyo   44  Gray Davis                26
Guillermo Coria           30  Hamid Karzai              22  Hans Blix                 39
Hugo Chavez               71  Igor Ivanov               20  Jack Straw                28
Jacques Chirac            52  Jean Chretien             55  Jennifer Aniston          21
Jennifer Capriati         42  Jennifer Lopez            21  Jeremy Greenstock         24
Jiang Zemin               20  John Ashcroft             53  John Negroponte           31
Jose Maria Aznar          23  Juan Carlos Ferrero       28  Junichiro Koizumi         60
Kofi Annan                32  Laura Bush                41  Lindsay Davenport         22
Lleyton Hewitt            41  Luiz Inacio Lula da Silva 48  Mahmoud Abbas             29
Megawati Sukarnoputri     33  Michael Bloomberg         20  Naomi Watts               22
Nestor Kirchner           37  Paul Bremer               20  Pete Sampras              22
Recep Tayyip Erdogan      30  Ricardo Lagos             27  Roh Moo-hyun              32
Rudolph Giuliani          26  Saddam Hussein            23  Serena Williams           52
Silvio Berlusconi         33  Tiger Woods               23  Tom Daschle               25
Tom Ridge                 33  Tony Blair               144  Vicente Fox               32
Vladimir Putin            49  Winona Ryder              24
```

Pour rééquilibrer les données, nous allons nous limiter à 50 images par personne (sinon, l'extraction des caractéristiques serait considérablement déviée par la prééminence de certaines personnalités) :

**In[23]:**

```
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1

X_people = people.data[mask]
y_people = people.target[mask]

X_people = X_people / 255.
```

Une tâche courante dans la problématique de reconnaissance de personnes consiste à se demander si un visage qui n'a pas encore été reconnu correspond à quelqu'un de déjà connu dans une base de données. Ce questionnement a des applications évidentes dans des domaines comme le classement de photographies dans un certain système, sur des réseaux sociaux ou encore dans tout ce qui concerne la sécurité. Une manière de répondre à ce type de problématique consisterait à construire un classifieur pour lequel chaque personne appartiendrait à une classe distincte. Cependant, il y a en général de nombreuses personnes différentes dans ce genre de base de données, et celles-ci contiennent très peu d'images de la même personne (autrement dit, très peu d'échantillons d'apprentissage par classe). Cela rend donc difficile l'entraînement d'un jeu de données par la plupart des classifieurs. De plus, vous voudriez généralement pouvoir être capable d'ajouter facilement de nouvelles personnes sans devoir entraîner à chaque fois un modèle de plus en plus grand.

Une solution simple consiste à utiliser un classifieur *un* plus proche voisin qui va rechercher l'image de visage la plus semblable à celle que vous voulez classifier. Ce classifieur pourrait en principe se contenter d'un seul échantillon d'apprentissage par classe. Voyons comment se comporte `KNeighborsClassifier` dans ce cas :

**In[24]:**

```
from sklearn.neighbors import KNeighborsClassifier
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Test set score of 1-nn: {:.2f}".format(knn.score(X_test, y_test)))
```

**Out[24]:**

```
Test set score of 1-nn: 0.27
```

La classification obtient une exactitude de 26,6 %, ce qui n'est pas mal pour un problème de classification comportant 62 classes, même si le résultat que vous obtenez en réalité peut être inférieur, de l'ordre de 23 %, ce qui dans tous les cas n'est pas extraordinaire (en fait, une prédiction faite totalement au hasard aurait une exactitude théorique de 1/62, soit 1,6 %). Finalement, cela signifie que nous n'identifions une personne qu'environ une fois sur quatre.

C'est ici que la méthode PCA entre en scène. Calculer des distances entre les pixels d'origine n'est pas vraiment une bonne façon de mesurer les similitudes entre visages. Lorsque nous faisons

appel à une représentation sous forme de pixels pour comparer deux images, nous comparons en fait la valeur de niveau de gris pour chaque pixel individuel à la valeur correspondante pour le pixel qui occupe la même position dans l'autre image. Cette représentation est tout à fait différente de la manière dont des êtres humains interpréteraient la vue d'un visage, et il est donc difficile de capturer ce qui constitue un visage particulier en utilisant des données qui se présenteraient sous cette forme. Par exemple, utiliser des distances entre pixels signifierait que décaler un visage d'un seul pixel vers la droite se traduirait par une représentation totalement différente. Ce que nous espérons, c'est qu'utiliser des distances en même temps que les composantes principales puisse améliorer notre exactitude. Ici, nous activons l'option *whitening* (littéralement *blanchiment*) de PCA qui recalibre les composantes principales pour qu'elles aient la même échelle. Cela revient en fait au même qu'utiliser `StandardScaler` après la transformation. En utilisant à nouveau les données de la Figure 3.3, cette méthode correspond non seulement à une rotation des données, mais aussi à un recalibrage de celles-ci de manière à obtenir une distribution circulaire et non plus elliptique (voir la Figure 3.8) :

**In[25] :**

```
mglearn.plots.plot_pca_whitening()
```

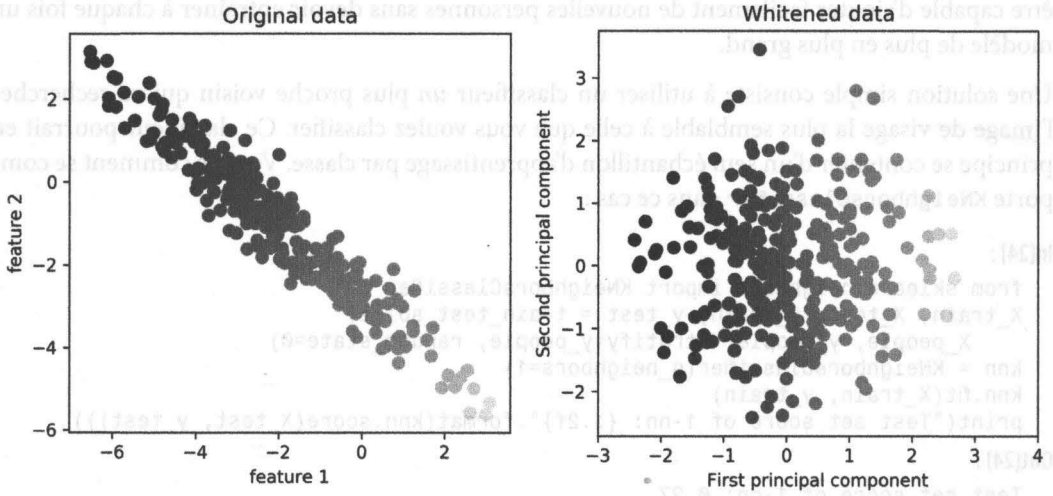



*Figure 3.8 : Transformation des données avec PCA en utilisant l'option whitening.*

Nous ajustons l'objet PCA à partir des données d'apprentissage et nous extrayons les 100 premières composantes principales. Nous transformons ensuite les données d'apprentissage et de test :

**In[26] :**

```
pca = PCA(n_components=100, whiten=True, random_state=0).fit(X_train)
X_train_pca = pca.transform(X_train)
```

```
X_test_pca = pca.transform(X_test)

print("X_train_pca.shape: {}".format(X_train_pca.shape))
```

**Out[26]:**

```
X_train_pca.shape: (1547, 100)
```

Nous obtenons 100 caractéristiques, qui sont les 100 premières composantes principales. Nous pouvons maintenant nous servir de cette nouvelle représentation pour utiliser un classifieur *un plus proche voisin* :

**In[27]:**

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
print("Test set accuracy: {:.2f}".format(knn.score(X_test_pca, y_test)))
```

**Out[27]:**

```
Test set accuracy: 0.36
```

Notre exactitude s'est améliorée de manière assez significative, de 26,6 % à 35,5 %, ce qui confirme notre intuition que les composantes principales pourraient fournir une meilleure représentation des données (il est possible que vous obteniez un résultat moins élevé, mais cela ne change en rien le raisonnement).

Pour les données des images, nous pouvons aussi facilement visualiser les composantes principales qui ont été trouvées. Souvenez-vous que ces composantes correspondent à des directions dans l'espace d'entrée. Celui-ci est formé dans cet exemple d'images en niveaux de gris de 87 x 65 pixels, et donc les directions dans cet espace sont également des images en niveaux de gris de 87 x 65 pixels.

Regardons les premières composantes principales (voir la Figure 3.9) :

**In[28]:**

```
print("pca.components_.shape: {}".format(pca.components_.shape))
```

**Out[28]:**

```
pca.components_.shape: (100, 5655)
```

**In[29]:**

```
fig, axes = plt.subplots(3, 5, figsize=(15, 12),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(pca.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape),
              cmap='viridis')
    ax.set_title("{}. component".format((i + 1)))
```

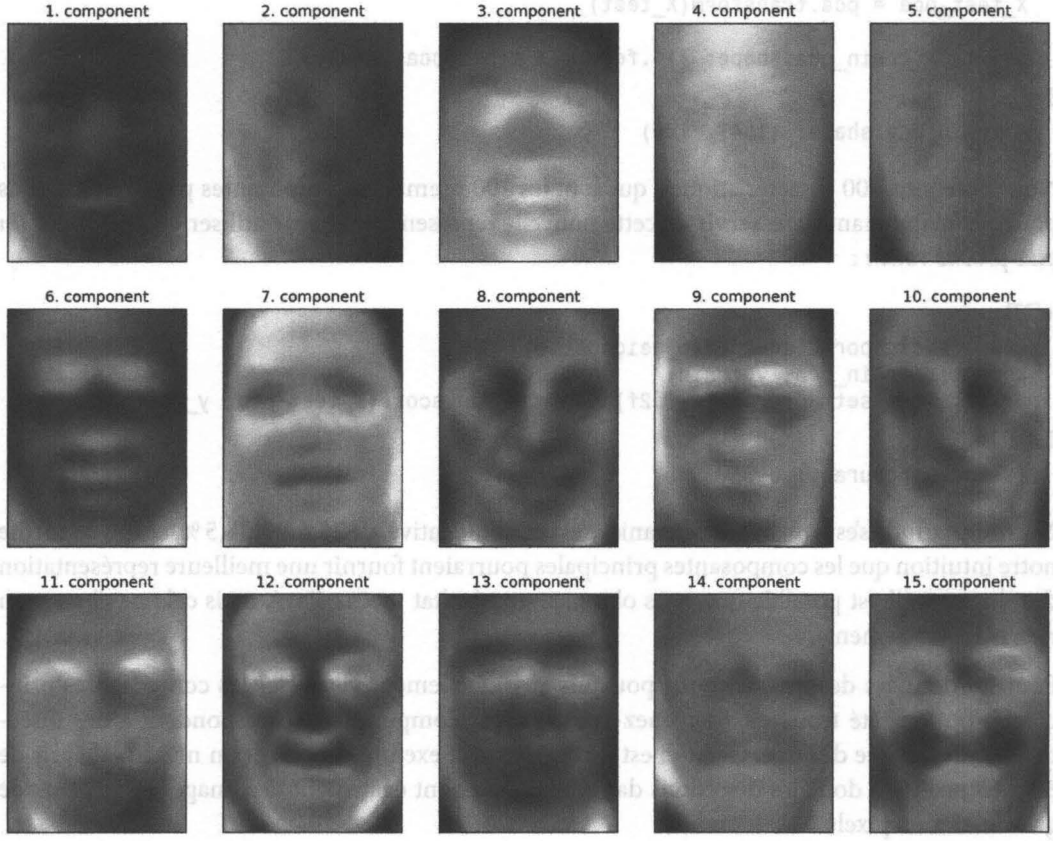



*Figure 3.9 : Visualisation pour les 15 premières composantes principales du jeu de données.*

Bien que nous ne puissions certainement pas comprendre tous les aspects de ces composantes, il est tout de même possible de se faire une idée de ce qu'elles capturent pour certaines d'entre elles. La première composante semble pour l'essentiel encoder le contraste entre le visage et l'arrière-plan. La seconde encode les différences d'éclairage entre la moitié gauche et la moitié droite du visage, et ainsi de suite. Bien que cette représentation soit un peu plus significative d'un point de vue sémantique que les valeurs brutes des pixels, elle reste tout de même assez loin de la manière dont un humain pourrait percevoir un visage. Du fait que le modèle PCA est basé sur des pixels, l'alignement du visage (la position des yeux, du menton et du nez) ainsi que l'éclairage ont tous deux une grande influence sur la similarité de deux images formées de nombreux pixels. Mais cet alignement et cet éclairage ne seront probablement pas ce qu'un humain percevrait en premier. Si vous demandez à des gens d'évaluer la ressemblance entre des visages, ils utiliseront très probablement des attributs comme l'âge, le genre, l'expression faciale ou encore la coiffure, autant d'éléments qu'il est difficile d'inférer à partir d'intensités dans des valeurs de pixels. Il est important de ne pas oublier que les algorithmes interprètent souvent les

données différemment de la manière dont réagissent les êtres humains (et en particulier dans le cas des données visuelles, comme des images, avec lesquelles les humains sont extrêmement familiarisés).

Revenons maintenant au cas particulier de la méthode PCA. Nous l'avons introduite en indiquant qu'il s'agit d'une transformation dans laquelle les données subissent une rotation, les composantes ayant une faible variance étant ensuite abandonnées. Une autre interprétation utile consiste à essayer de trouver certains nombres (les nouvelles valeurs des caractéristiques après la rotation PCA) de manière à pouvoir exprimer les points de test sous la forme d'une somme pondérée des composantes principales (voir la Figure 3.10).

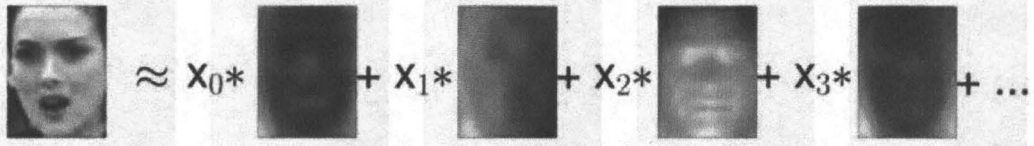



*Figure 3.10 : Vue schématique de l'algorithme PCA sous la forme d'une décomposition d'une image en une somme pondérée de composantes.*

Ici, $x_0$, $x_1$ et ainsi de suite sont les coefficients des composantes principales pour ce point de donnée. En d'autres termes, ils sont la représentation de l'image dans l'espace obtenu par la rotation PCA.

Une autre façon d'essayer de comprendre ce que fait un modèle PCA consiste à regarder la reconstruction des données originales en ne se servant seulement que de certaines composantes. Sur le troisième tracé de la Figure 3.3, après avoir réduit la dimension à une seule composante, nous avions annulé la rotation et rajouté à nouveau la moyenne afin d'obtenir de nouveaux points dans l'espace d'origine, mais sans la seconde composante (c'est le quatrième tracé de la figure). Nous pouvons effectuer une transformation similaire pour les visages en réduisant les données à un certain nombre de composantes principales, puis en inversant la rotation pour revenir à l'espace d'origine. Ce retour à l'espace des caractéristiques originales peut être réalisé en faisant appel à la méthode `inverse_transform`. Sur la Figure 3.11, nous visualisons la reconstruction de quelques visages en utilisant successivement 10, 50, 100 et 500 composantes :

**In[30]:**

```
mglearn.plots.plot_pca_faces(X_train, X_test, image_shape)
```




*Figure 3.11 : Reconstruction de trois visages en choisissant un nombre croissant de composantes principales.*

Vous pouvez voir que, lorsque nous utilisons seulement les dix premières composantes principales, seule l'essence de l'image, comme l'orientation du visage et l'éclairage, est capturée. Plus nous augmentons le nombre de composantes principales, et plus nous préservons de détails dans l'image. Cela revient à étendre la somme illustrée sur la Figure 3.10 en y incluant de plus en plus de termes. En prenant autant de composantes qu'il y a de pixels, nous ne perdrions aucune information à la suite de la rotation, et donc l'image serait parfaitement reconstruite.

Nous pouvons aussi essayer d'utiliser PCA pour visualiser tous les visages du jeu de données sous forme de nuage de points en utilisant les deux premières composantes principales, comme nous l'avions fait pour le jeu de données `cancer` (voir la Figure 3.12) :

**In[31]:**

```
mglearn.discrete_scatter(X_train_pca[:, 0], X_train_pca[:, 1], y_train)
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

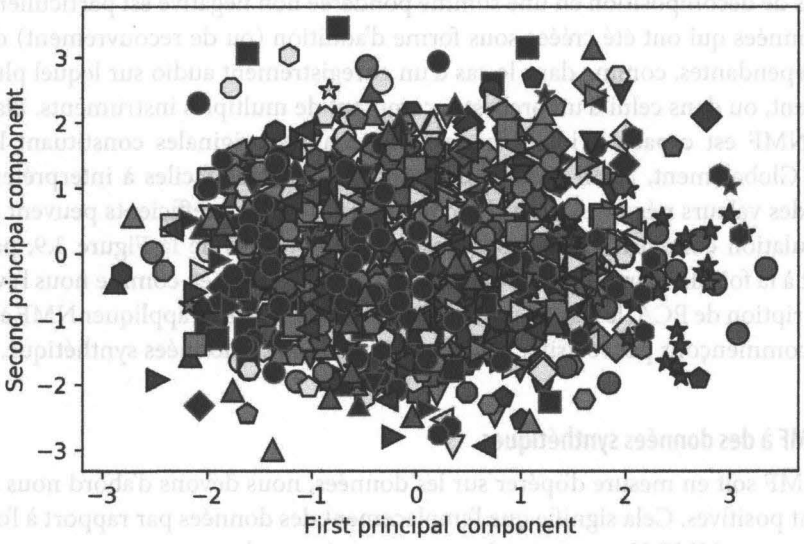



*Figure 3.12 : Nuage de points du jeu de données de visages utilisant les deux premières composantes principales (reportez-vous à la Figure 3.5 pour l'image correspondante du jeu de données cancer).*

Comme vous pouvez le constater, lorsque nous utilisons uniquement les deux premières composantes principales, l'ensemble des données forme une sorte de grosse goutte, sans séparation visible entre les classes. Cela n'est pas très surprenant, étant donné que même avec 10 composantes, comme nous l'avons vu sur la Figure 3.11, PCA ne capture que des caractéristiques très grossières sur les visages.

## Factorisation en matrices non négatives (NMF)

La factorisation en (ou par) matrices non négatives (NMF, pour *non-negative matrix factorization* en anglais) est un autre algorithme d'apprentissage non supervisé visant à extraire des caractéristiques utiles. Il fonctionne d'une manière semblable à PCA, et peut aussi servir à la réduction de dimension. Comme dans le cas de PCA, nous essayons d'écrire chaque point de donnée sous la forme d'une somme pondérée de certaines composantes, comme l'illustre la Figure 3.10. Mais alors que, dans le cas de PCA, nous recherchions des composantes orthogonales et une variance maximale, nous voulons avec NMF que les composantes et les coefficients soient non négatifs, et donc que toutes les valeurs soient supérieures ou égales à zéro. Par conséquent, cette méthode ne peut être appliquée qu'à des données pour lesquelles chaque caractéristique est non négative, puisqu'une somme non négative de composantes également non négatives ne peut pas devenir négative.

Le processus de décomposition en une somme pondérée non négative est particulièrement utile pour des données qui ont été créées sous forme d'addition (ou de recouvrement) de plusieurs sources indépendantes, comme dans le cas d'un enregistrement audio sur lequel plusieurs personnes parlent, ou dans celui d'un orchestre comptant de multiples instruments. Dans de telles situations, NMF est capable d'identifier les composantes originales constituant les données combinées. Globalement, NMF produit des composantes plus faciles à interpréter que PCA, du fait que des valeurs négatives pour des composantes et des coefficients peuvent générer des effets d'annulation complexes à analyser. Les vecteurs propres de la Figure 3.9, par exemple, contiennent à la fois des parties positives et des parties négatives, et, comme nous l'avons signalé dans la description de PCA, le signe est en réalité arbitraire. Avant d'appliquer NMF à nos images de visages, commençons par revisiter brièvement notre jeu de données synthétique.

### Appliquer NMF à des données synthétiques

Pour que NMF soit en mesure d'opérer sur les données, nous devons d'abord nous assurer que celles-ci sont positives. Cela signifie que l'emplacement des données par rapport à l'origine (0,0) est important pour NMF. Vous pouvez donc vous représenter les composantes non négatives qui sont extraites comme des directions allant de (0, 0) vers les données.

L'exemple qui suit montre les résultats produits avec NMF sur le jeu de données jouet à deux dimensions (voir la Figure 3.13) :

**In[32]:**

```
mglearn.plots.plot_nmf_illustration()
```

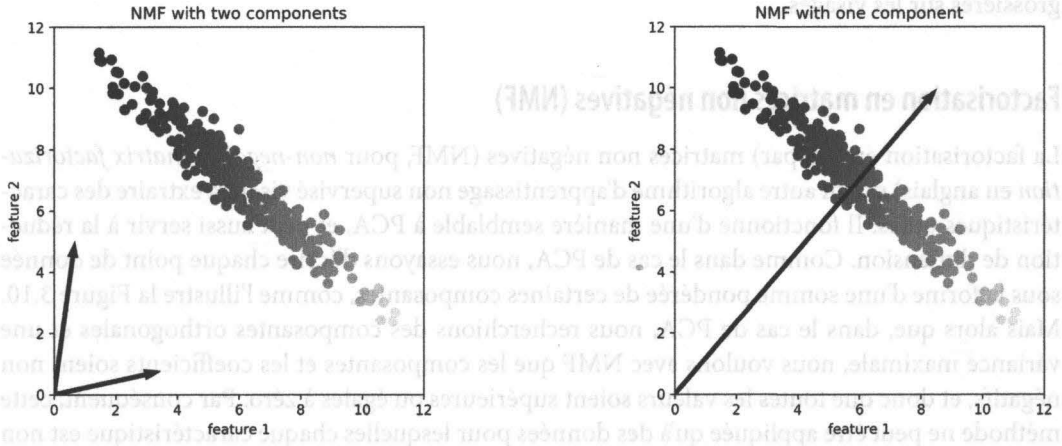



*Figure 3.13 : Composantes trouvées par NMF avec deux composantes (à gauche) et une seule (à droite).*

Dans le cas de NMF avec deux composantes (à gauche sur la figure), il est clair que tous les points dans les données peuvent être écrits sous la forme d'une combinaison positive de ces deux composantes. S'il y a suffisamment de composantes pour reconstruire parfaitement les données (donc autant qu'il y a de caractéristiques), l'algorithme choisira les directions pointant vers les extrémités des données.

Si nous utilisons une seule composante (à droite sur la figure), NMF crée une composante qui pointe vers la moyenne, car c'est la direction qui fournit la meilleure explication sur les données. Vous pouvez voir que, par contraste avec PCA, réduire le nombre de composantes ne fait pas que supprimer certaines directions, mais crée en fait un ensemble totalement différent de composantes ! Dans NMF, les composantes jouent toutes à égalité, car il n'existe pas d'ordre particulier, pas de « première composante non négative ».

NMF utilise une initialisation aléatoire, ce qui pourrait conduire à des résultats différents selon la valeur servant à cette initialisation. Dans des cas relativement simples, comme notre jeu de données synthétique avec deux composantes, où toutes les données peuvent être expliquées parfaitement, cet aspect aléatoire a peu d'effet (quoique cela puisse modifier l'ordre ou l'échelle des composantes). Ces changements pourraient être nettement plus prononcés dans des situations plus complexes.

### Appliquer NMF aux images de visages

Revenons maintenant à notre jeu d'images de visages pour y appliquer la méthode NMF. Le principal paramètre de NMF est le nombre de composantes que nous voulons extraire. Normalement, cette valeur est plus basse que le nombre de caractéristiques en entrée (sinon, les données pourraient être expliquées en considérant que chaque pixel est une composante distincte). Tout d'abord, observons l'impact du nombre de composantes sur la reconstruction des données avec NMF (voir la Figure 3.14) :

**In[33]:**

```
mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
```

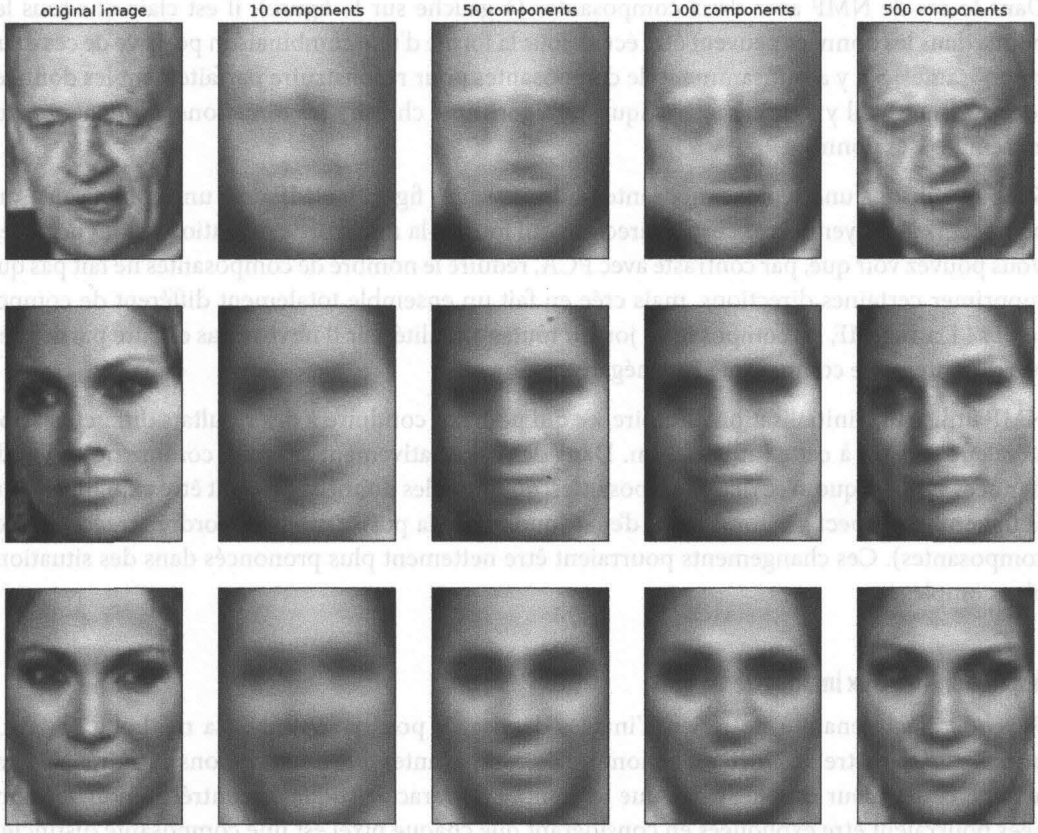



*Figure 3.14 : Reconstruction de trois images de visages en utilisant un nombre croissant de composantes trouvées par NMF.*

La qualité du résultat est comparable à ce que nous avons vu plus haut avec PCA, quoiqu'un peu plus mauvaise. Ce n'est pas surprenant, puisque PCA trouve les directions optimales en termes de reconstruction. NMF n'est généralement pas utilisé pour sa capacité à reconstruire ou encoder des données, mais plutôt pour trouver des motifs intéressants dans ces données.

Pour pousser notre exploration, essayons d'extraire seulement quelques composantes (disons 15). La Figure 3.15 montre le résultat :

**In[34]:**

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=15, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)

fig, axes = plt.subplots(3, 5, figsize=(15, 12),
                         subplot_kw={'xticks': (), 'yticks': ()})
```

```
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("{}. component".format(i))
```

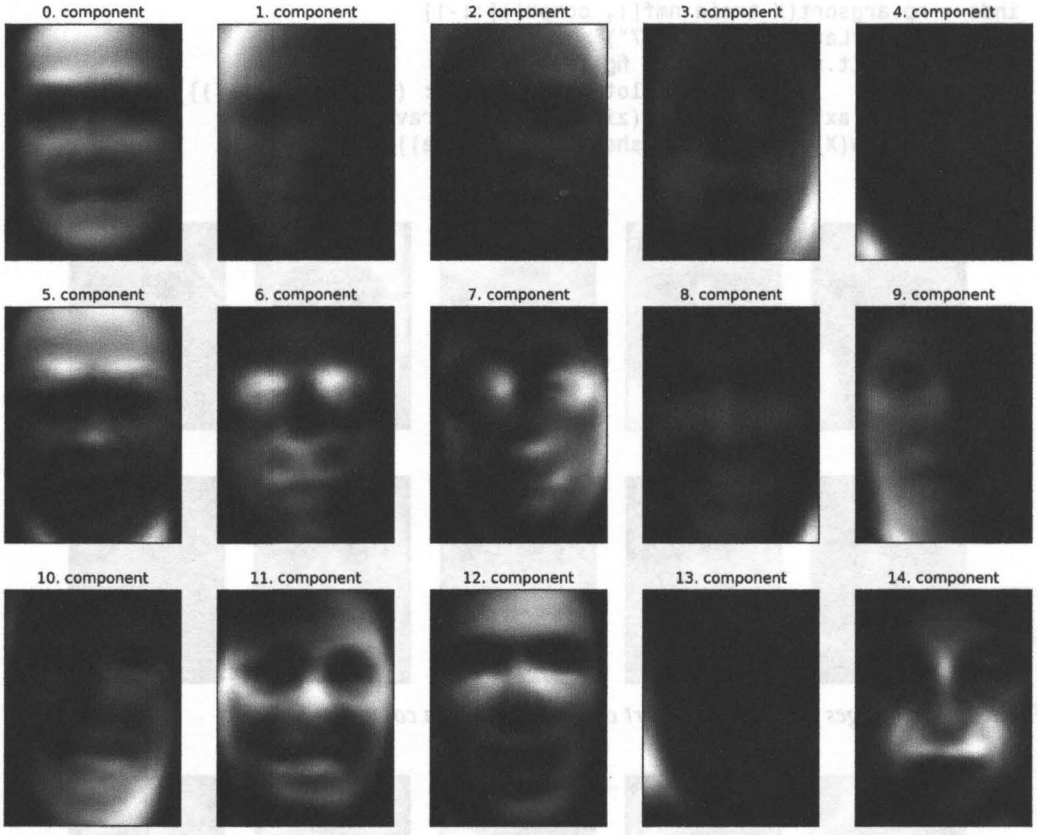



*Figure 3.15 : Les composantes trouvées par NMF sur des visages du jeu de données (avec 15 composantes).*

Ces composantes sont toutes positives, et elles ressemblent bien plus à des prototypes de visages que ce que nous avions obtenu avec PCA (reportez-vous à la Figure 3.9). Par exemple, on peut clairement voir que la composante 3 montre un visage tourné légèrement vers la droite, tandis que la composante 7 montre un visage légèrement tourné vers la gauche. Observons des images pour lesquelles ces caractéristiques sont particulièrement prononcées (voir les Figures 3.16 et 3.17) :

**In[35]:**

```
compn = 3
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
```

```
fig.suptitle("Large component 3")
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))

compn = 7
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig.suptitle("Large component 7")
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))
```



*Figure 3.16 : Visages présentant un fort coefficient pour la composante 3.*



*Figure 3.17 : Visages présentant un fort coefficient pour la composante 7.*

Comme cela a déjà été mentionné, l'extraction de motifs comme ceux-ci fonctionne mieux avec des données qui comportent une structure additive, comme un enregistrement audio, une structure de gène, ou encore des données textuelles. Voyons cela de plus près en partant d'un exemple de jeu de données synthétique.

Supposons que nous soyons intéressés par un signal qui est une combinaison de trois sources différentes (voir la Figure 3.18) :

**In[36]:**

```
S = mglearn.datasets.make_signals()
plt.figure(figsize=(6, 1))
plt.plot(S, '-')
plt.xlabel("Time")
plt.ylabel("Signal")
```

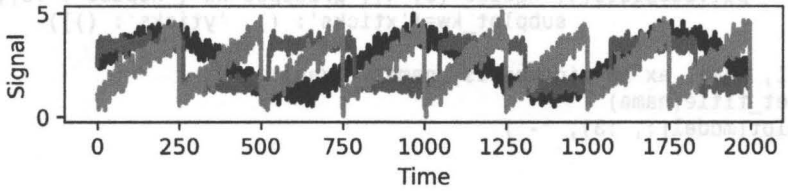



*Figure 3.18 : Les sources originales du signal.*

Malheureusement, nous ne pouvons pas observer les signaux originaux, mais uniquement une addition des trois. Nous voulons récupérer la décomposition du signal ainsi mixé en ses différentes composantes. Nous supposons que nous avons de nombreuses solutions pour observer notre signal composite (disons, 100 dispositifs), chacune nous fournissant une série de mesures :

**In[37]:**

```
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("Shape of measurements: {}".format(X.shape))
```

**Out[37]:**

```
Shape of measurements: (2000, 100)
```

Nous pouvons faire appel à NMF pour retrouver les trois signaux :

**In[38]:**

```
nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
print("Recovered signal shape: {}".format(S_.shape))
```

**Out[38]:**

```
Recovered signal shape: (2000, 3)
```

Appliquons également PCA à titre de comparaison :

**In[39]:**

```
pca = PCA(n_components=3)
H = pca.fit_transform(X)
```

La Figure 3.19 montre l'activité des signaux tels qu'ils sont découverts par les modèles NMF et PCA :

**In[40]:**

```
models = [X, S, S_, H]
names = ['Observations (first three measurements)',
         'True sources',
         'NMF recovered signals',
         'PCA recovered signals']

fig, axes = plt.subplots(4, figsize=(8, 4), gridspec_kw={'hspace': .5},
                         subplot_kw={'xticks': (), 'yticks': ()})

for model, name, ax in zip(models, names, axes):
    ax.set_title(name)
    ax.plot(model[:, :3], '-')
```

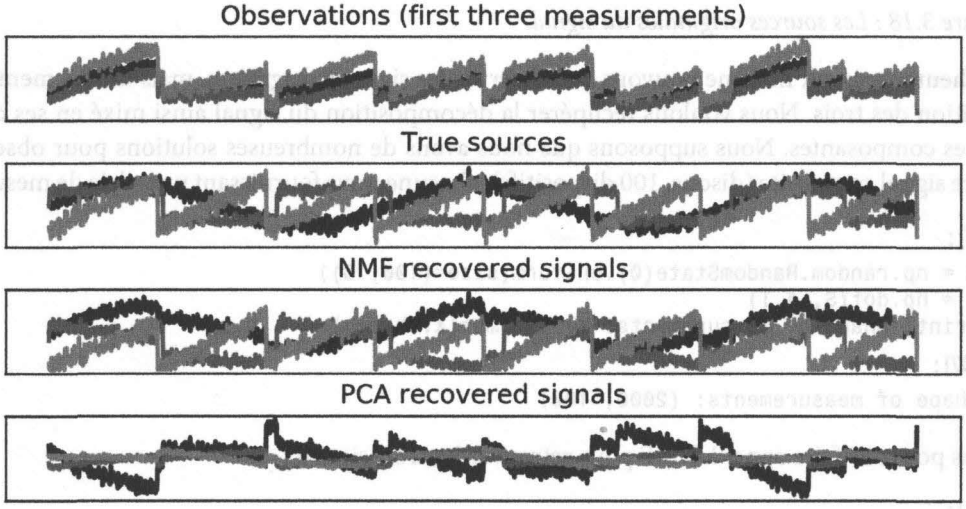



*Figure 3.19 : Récupération de sources mélangées en utilisant NMF et PCA.*

La figure inclut 3 des 100 mesures effectuées à titre de référence. Comme vous pouvez le voir, NMF a fait un travail raisonnablement bon pour découvrir les sources originales, alors que PCA a échoué et utilisé la première composante pour expliquer la majorité des variations dans les données. N'oubliez pas que les composantes produites par NMF n'ont pas d'ordre naturel. Dans

cet exemple, l'ordre des composantes NMF est le même que dans le signal original, mais il s'agit d'un résultat purement accidentel.

Il existe nombre d'autres algorithmes qui pourraient être utilisés pour décomposer chaque point de donnée en une somme pondérée d'une quantité fixée de composantes, comme le font PCA et NMF. Discuter de tous ces algorithmes déborderait largement du cadre de ce livre, et décrire les contraintes apportées aux composantes et aux coefficients nécessite de solides connaissances en théorie des probabilités. Si le sujet vous intéresse, nous vous conseillons vivement de vous intéresser de près aux sections du guide utilisateur de `scikit-learn` qui concernent l'analyse en composantes indépendantes (*independant component analysis*, ou ICA), l'analyse en facteurs (*factor analysis*, ou FA) et le codage parcimonieux (*sparse coding*). Rendez-vous à ce sujet sur la page `http://scikit-learn.org/stable/modules/decomposition.html`, évidemment en anglais.

## Apprentissage de variétés avec t-SNE

Même si PCA constitue souvent une bonne approche de départ pour transformer vos données de manière à pouvoir les visualiser en utilisant un nuage de points, la nature de la méthode (appliquer une rotation puis éliminer certaines directions) en limite l'utilité. C'est ce que nous avons pu constater plus haut avec le jeu d'images de visages. Il existe une classe d'algorithmes appelée *manifold learning*, soit littéralement *apprentissage de variétés* en français, qui permet de réaliser des cartographies bien plus complexes et offre souvent de meilleures visualisations. L'un des algorithmes particulièrement utiles de cette classe s'appelle t-SNE (pour *t-distributed stochastic neighbor embedding*).

Les algorithmes d'apprentissage de variétés sont principalement destinés à la visualisation, et ils sont rarement utilisés pour générer plus de deux nouvelles caractéristiques. Certains d'entre eux, dont t-SNE, calculent une nouvelle représentation des données d'apprentissage, mais ils ne permettent pas d'effectuer des transformations de nouvelles données. Cela implique que ces algorithmes ne peuvent pas être appliqués aux données de test. En fait, ils ne peuvent transformer que les données pour lesquelles ils ont été formés. Dit autrement, les algorithmes de *manifold learning* trouvent leur utilité dans l'analyse exploratoire de données, mais ils sont rarement employés lorsque le but final est un apprentissage supervisé. L'idée sous-jacente à t-SNE est de trouver une représentation bidimensionnelle des données qui préserve le mieux possible les distances entre les points. Cet algorithme débute par une représentation aléatoire en deux dimensions pour chaque point de donnée, puis il essaie de rapprocher les points qui sont proches dans l'espace des caractéristiques d'origine, et les points qui sont éloignés l'un de l'autre dans l'espace des caractéristiques initiales. t-SNE met davantage l'accent sur les points qui sont proches, plutôt que de préserver les distances entre des points éloignés. En d'autres termes, cet algorithme s'efforce de préserver les informations qui indiquent quels points sont proches les uns des autres.

Nous allons appliquer l'algorithme t-SNE sur un jeu de données formé de chiffres écrits à la main. Cet exemple est inclus dans l'installation de `scikit-learn` (mais il ne doit pas être confondu avec le jeu de données plus vaste MNIST). Chaque point du jeu de données est une image en niveaux de gris dans un format de 8 x 8 d'un chiffre tracé à la main (donc de 0 jusqu'à 9). La Figure 3.20 montre un échantillon d'image pour chaque classe :

**In[41]:**

```
from sklearn.datasets import load_digits
digits = load_digits()

fig, axes = plt.subplots(2, 5, figsize=(10, 5),
                         subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img)
```

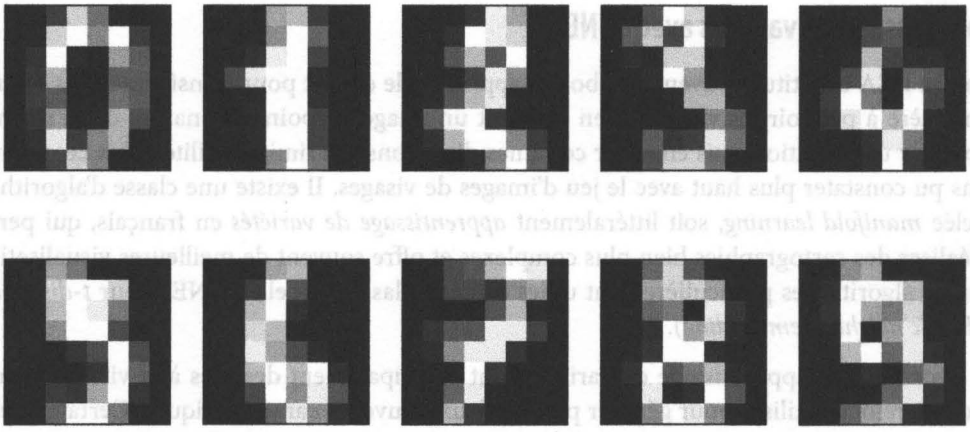


*Figure 3.20 : Exemples d'images du jeu de données* `digits`.

Servons-nous de PCA pour visualiser les données réduites à deux dimensions. Nous traçons les deux premières composantes principales, et nous représentons chaque échantillon avec un chiffre correspondant à sa classe (voir la Figure 3.21) :

**In[42]:**

```
pca = PCA(n_components=2)
pca.fit(digits.data)
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
```

```
            color = colors[digits.target[i]],
            fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

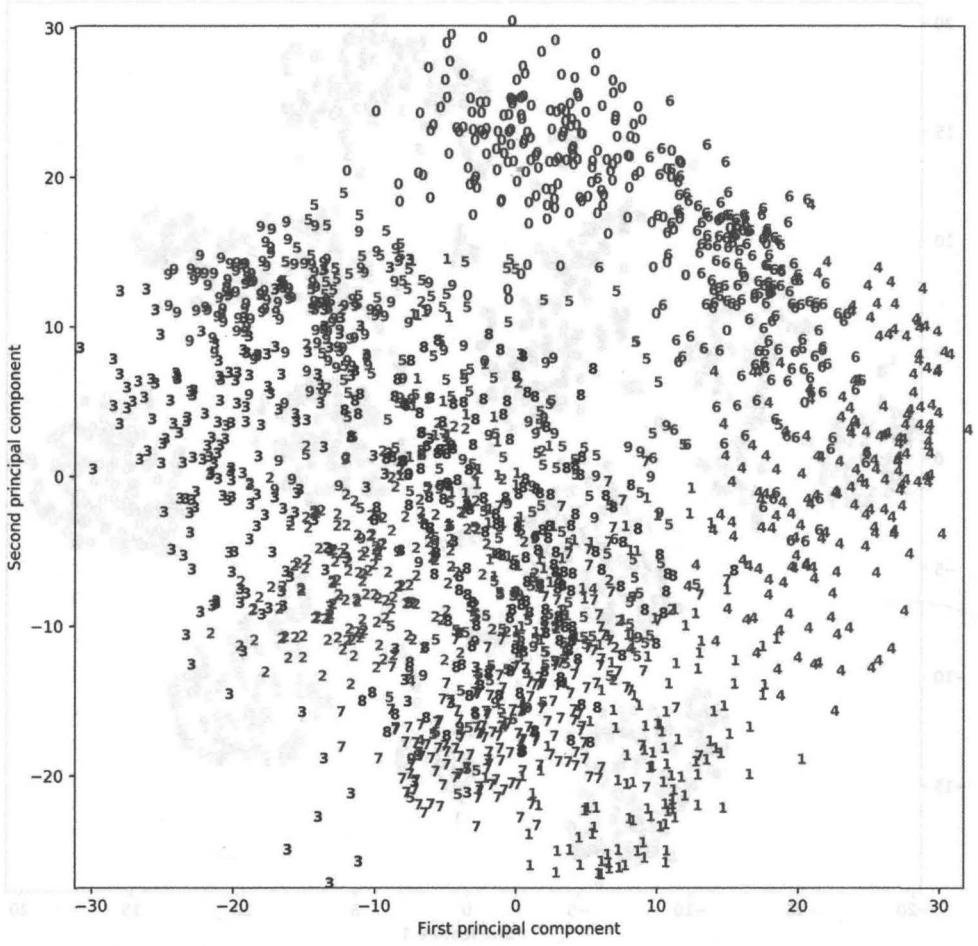



*Figure 3.21 : Nuage de points pour le jeu de données de chiffres en utilisant les deux premières composantes principales.*

Ici, nous avons utilisé en fait les vraies classes des chiffres en tant que glyphes (des représentations graphiques des caractères) afin de montrer de quelles classes il s'agit. Les chiffres zéro, six et quatre sont relativement bien séparés en utilisant les deux premières composantes principales, même s'il y a des recouvrements. Par contre, la plupart des autres chiffres sont plus enchevêtrés.

Appliquons maintenant t-SNE au même jeu de données et comparons les résultats. Du fait que cet algorithme ne supporte pas la transformation de nouvelles données, la classe `TSNE` ne pos-

sède pas de méthode `transform`. À la place, nous pouvons appeler la méthode `fit_transform`, qui va construire le modèle et immédiatement retourner les données transformées (voir la Figure 3.22) :

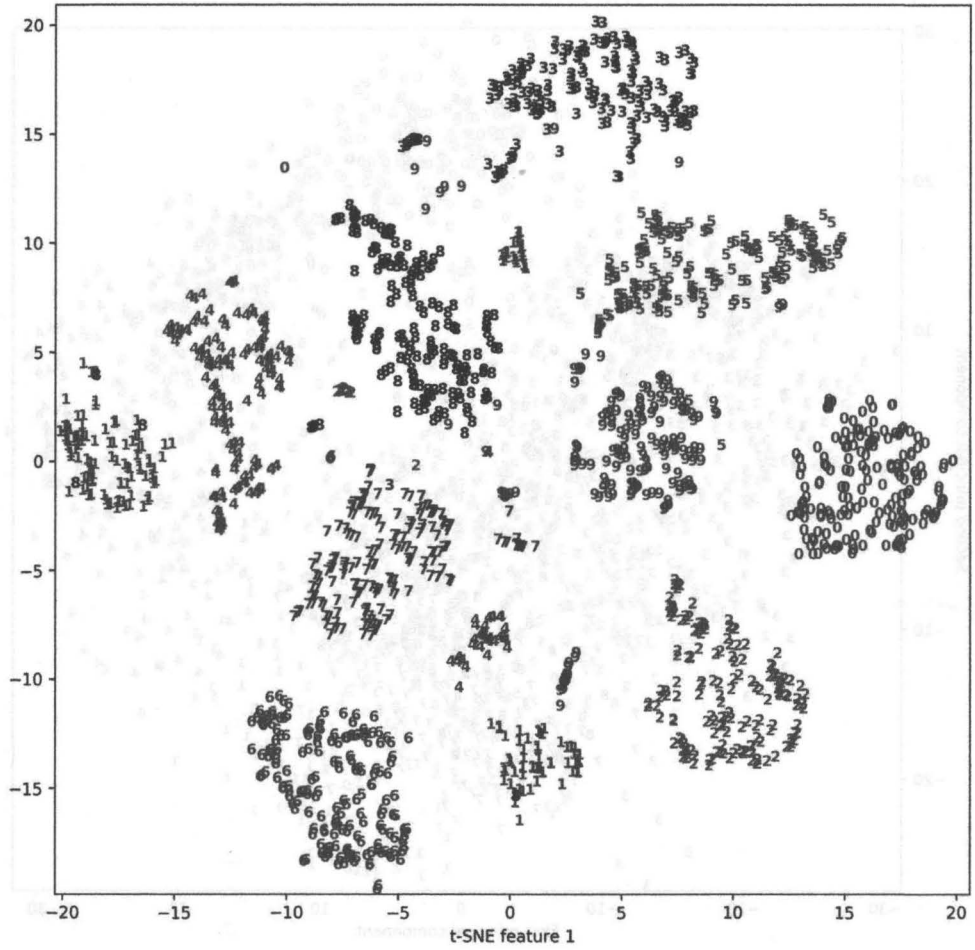



*Figure 3.22 : Nuage de points pour le jeu de données de chiffres en utilisant deux composantes trouvées par t-SNE.*

**In[43]:**

```
from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
digits_tsne = tsne.fit_transform(digits.data)
```

**In[44]:**

```
plt.figure(figsize=(10, 10))
```

```
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("t-SNE feature 0")
plt.xlabel("t-SNE feature 1")
```

Le résultat de t-SNE est assez remarquable. Toutes les classes sont plutôt clairement séparées. Certaines semblent un peu dispersées, comme les 1 et les 0, mais la plupart forment des groupes relativement denses et cohérents. N'oubliez pas que cette méthode ne sait rien des étiquettes des classes : elle est totalement non supervisée. Elle peut donc trouver une représentation des données en deux dimensions qui sépare clairement les classes en se basant uniquement sur la proximité des points dans l'espace d'origine.

L'algorithme t-SNE dispose de plusieurs paramètres qu'il est possible d'ajuster, bien qu'il fonctionne souvent bien avec les réglages par défaut. Vous pouvez par exemple essayer de jouer avec les valeurs de `perplexity` et de `early_exaggeration`, mais les effets produits sont généralement mineurs.

# Clustering

Comme nous l'avons indiqué au début de ce chapitre, le *clustering* consiste à partitionner le jeu de données en groupes appelés des *clusters* (ou *grappes*). Le but est de diviser les données de telle manière que les points qui se trouvent dans un même cluster soient très similaires, et que ceux qui apparaissent dans des clusters différents soient très dissemblables. Comme dans les algorithmes de classification, les algorithmes de partitionnement en clusters assignent (ou prédisent) un numéro à chaque point de donnée, indiquant ainsi à quel cluster un certain point appartient.

## Partitionnement en *k*-moyennes

Le partitionnement en *k*-moyennes (*k-means clustering*) est l'un des algorithmes de clustering parmi les plus simples et les plus couramment utilisés. Il essaie de trouver des « centres » de clusters qui sont représentatifs de certaines régions des données. L'algorithme alterne entre deux étapes : affecter chaque point de donnée au centre du cluster le plus proche, puis définir chaque centre de cluster comme étant la moyenne des points de données qui lui ont été affectés. L'algorithme se termine lorsque l'affectation des instances aux clusters ne varie plus (voyez également l'article K-moyennes sur Wikipédia, à l'adresse `https://fr.wikipedia.org/wiki/K-moyennes`). L'exemple suivant illustre l'algorithme sur un jeu de données synthétique (voir la Figure 3.23) :

In[45]:

```
mglearn.plots.plot_kmeans_algorithm()
```

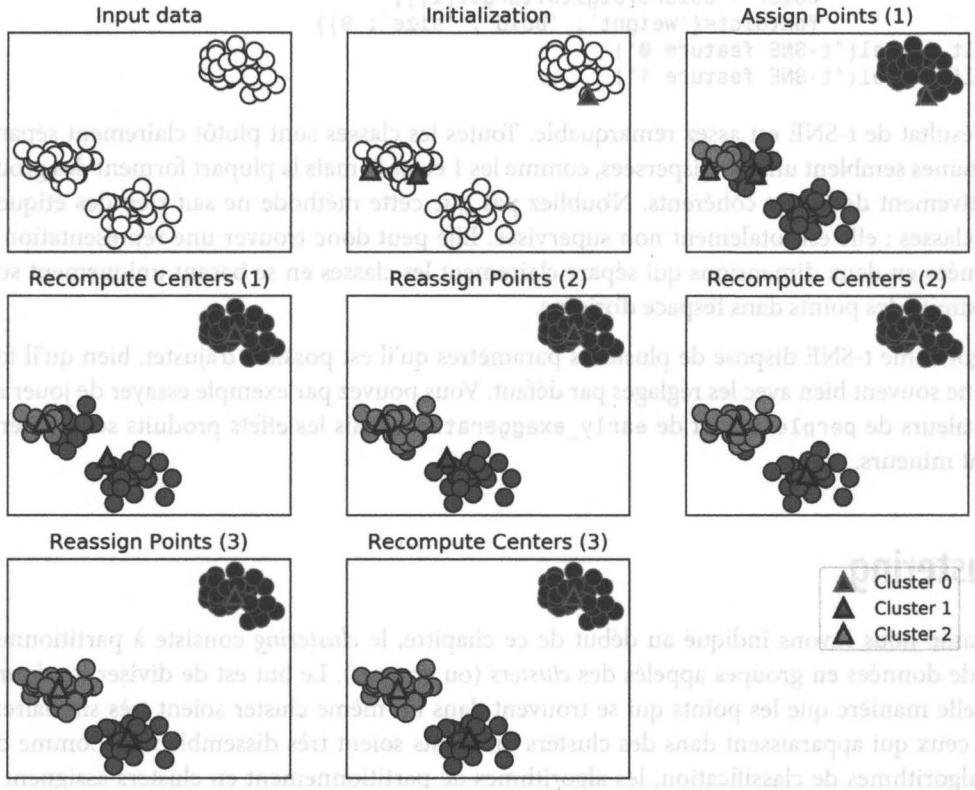



*Figure 3.23 : Données d'entrée et trois étapes de l'algorithme de partitionnement en k-moyennes.*

Les centres des clusters sont représentés par des triangles et les points de données par des cercles. L'intensité des niveaux de gris indique l'appartenance à un certain cluster. Nous avons spécifié que nous recherchions un résultat portant sur trois clusters. De ce fait, l'algorithme a été initialisé en déclarant de façon aléatoire trois points de données comme constituant les centres des clusters (c'est le tracé « Initialization », en haut et au centre). L'algorithme itératif débute ensuite. Tout d'abord, chaque point de donnée est affecté au centre du cluster dont il est le plus proche (voir sur la figure le tracé intitulé « Assign Points (1) »). Les centres des clusters sont alors ajustés pour être la moyenne des points qui lui ont été affectés (voir sur la figure le tracé « Recompute Centers (1) »). Ce processus est répété deux autres fois. Après la troisième itération, l'affectation des points de données aux centres des clusters ne change plus, et donc l'algorithme s'arrête.

Avec de nouveaux points de données, l'algorithme des *k*-moyennes va assigner chacun au centre du cluster le plus proche. L'exemple qui suit montre les frontières des centres des clusters tels qu'ils ont été entraînés sur la Figure 3.23 (voir la Figure 3.24) :

**In[46]:**

```
mglearn.plots.plot_kmeans_boundaries()
```

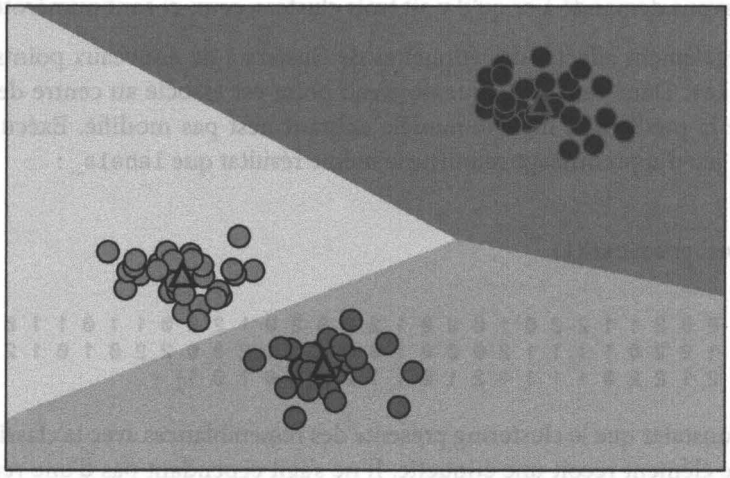


*Figure 3.24 : Centres des clusters et limites correspondantes trouvés par l'algorithme des k-moyennes.*

Appliquer un *clustering* en *k*-moyennes avec `scikit-learn` est assez simple. Ici, nous allons le faire pour les données synthétiques que nous avons utilisées pour les tracés précédents. Nous instancions la classe `KMeans`, et nous définissons le nombre de clusters que nous recherchons (la valeur par défaut de `n_clusters` est de 8, mais il n'y a aucune raison particulière pouvant justifier de conserver ce nombre). Nous appelons ensuite la méthode `fit` avec les données :

**In[47]:**

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

X, y = make_blobs(random_state=1)

kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

Au cours de l'exécution de l'algorithme, chaque donnée du jeu d'apprentissage dans `X` est affectée à l'étiquette d'un cluster. Vous pouvez retrouver ces étiquettes dans l'attribut `kmeans.labels_` :

**In[48]:**

```
print("Cluster memberships:\n{}".format(kmeans.labels_))
```

**Out[48]:**

```
Cluster memberships:
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

Comme nous avons demandé à ce qu'il y ait trois clusters, ceux-ci sont numérotés de 0 à 2.

Vous pouvez également affecter des étiquettes de clusters à de nouveaux points en utilisant la méthode `predict`. Dans ce cas, chaque nouveau point est associé au centre de cluster le plus proche lors de la prédiction, mais le modèle existant n'est pas modifié. Exécuter la méthode `predict` sur le jeu d'apprentissage retourne le même résultat que `labels_` :

**In[49]:**

```
print(kmeans.predict(X))
```

**Out[49]:**

```
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

Vous pouvez constater que le clustering présente des ressemblances avec la classification dans le sens où chaque élément reçoit une étiquette. Il ne s'agit cependant pas d'une réalité de terrain, et donc les étiquettes elles-mêmes n'ont *a priori* aucune signification particulière. Revenons à l'exemple des images de visage déjà étudié plus haut. Avec un algorithme de partitionnement, il se pourrait que, disons, le cluster 3 contienne uniquement des clichés de votre ami Arnold. Mais vous ne pouvez le savoir qu'après avoir regardé les images, et le nombre 3 est totalement arbitraire. La seule information que l'algorithme vous donne est que tous les visages étiquetés 3 sont similaires.

Pour le partitionnement que nous venons de calculer à partir du jeu de données jouet à deux dimensions, cela signifie que nous ne devrions donner aucune signification au fait qu'un groupe est étiqueté 0, et un autre 1. Exécuter une nouvelle fois l'algorithme pourrait se traduire par une numérotation différente des clusters du fait de la nature aléatoire de l'initialisation (et c'est très possiblement ce que vous constaterez en exécutant les lignes de code ci-dessus sur votre propre système).

Voyons un nouveau tracé de ces données (voir la Figure 3.25). Les centres de clusters sont enregistrés dans l'attribut `cluster_centers_` et nous les représentons sous la forme de triangles :

**In[50]:**

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
    markers='^', markeredgewidth=2)
```

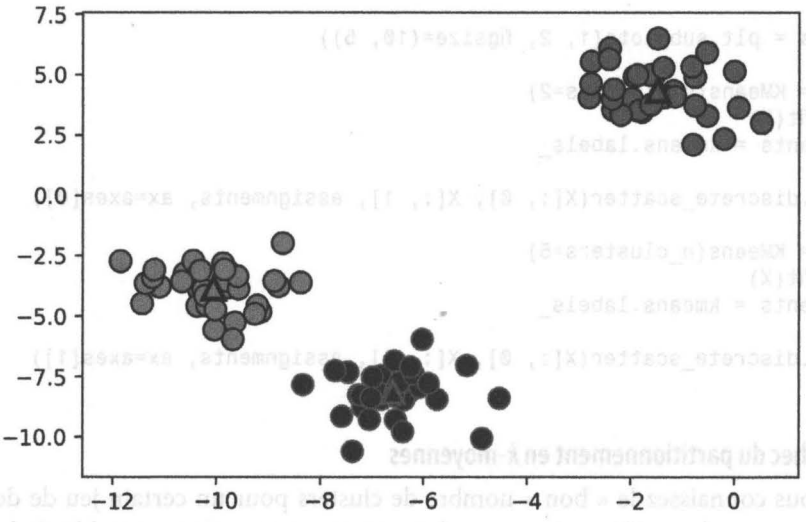



*Figure 3.25 : Affectations et centre des clusters trouvés par l'algorithme des k-moyennes avec trois clusters.*

Nous pouvons également utiliser plus ou moins de centres de clusters (voir la Figure 3.26) :

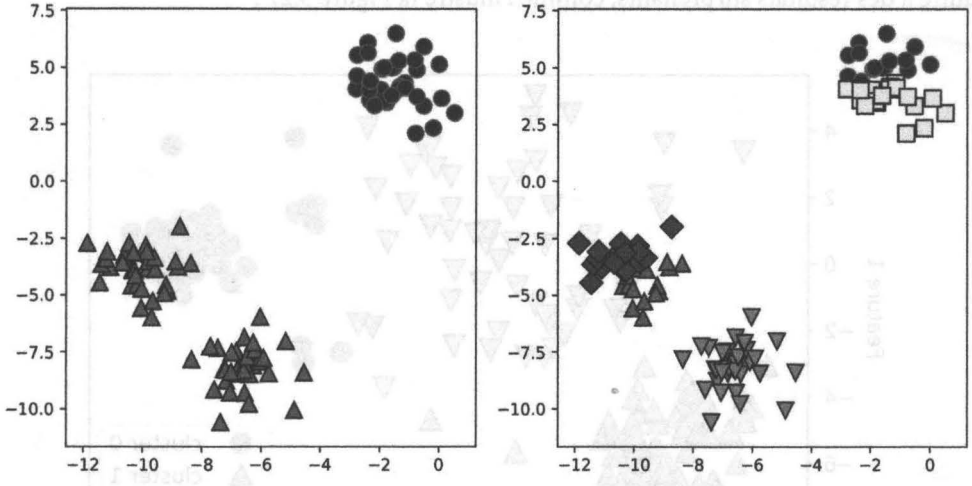



*Figure 3.26 : Application du partitionnement en k-moyennes avec deux clusters (à gauche) et cinq clusters (à droite).*

In[51] :

```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])

kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])
```

### Risques d'échec du partitionnement en *k*-moyennes

Même si vous connaissez le « bon » nombre de clusters pour un certain jeu de données, cela ne signifie pas que le partitionnement en *k*-moyennes soit toujours capable de les retrouver. Chaque cluster est défini uniquement par son centre, ce qui signifie que les clusters ont une forme convexe. De ce fait, cet algorithme ne peut capturer que des formes relativement simples. Il suppose aussi que tous les clusters ont en quelque sorte le même « diamètre » : les frontières entre clusters sont toujours tracées exactement à mi-distance de leurs centres. Cela peut parfois conduire à des résultats surprenants, comme l'illustre la Figure 3.27 :

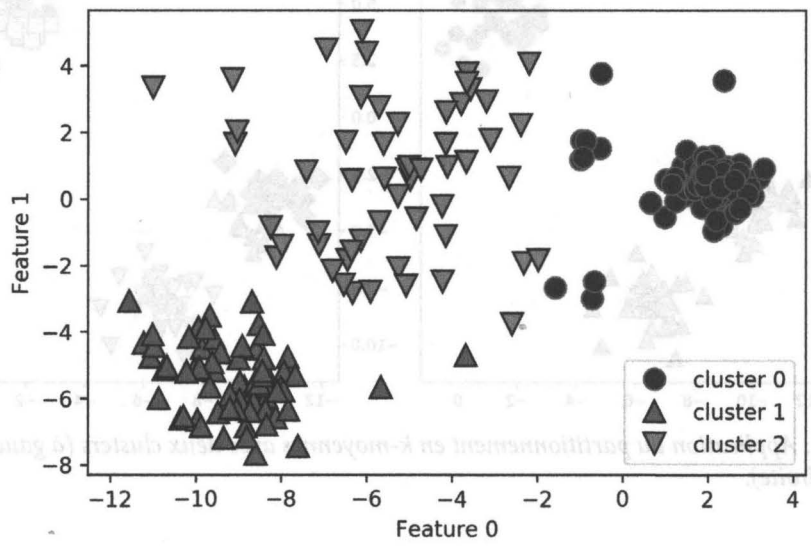



*Figure 3.27 : Clusters trouvés par l'algorithme des k-moyennes lorsque ces clusters ont des densités différentes.*

**In[52]:**

```
X_varied, y_varied = make_blobs(n_samples=200,
                                cluster_std=[1.0, 2.5, 0.5],
                                random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X_varied)

mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["cluster 0", "cluster 1", "cluster 2"], loc='best')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

On pourrait s'attendre à ce que la région dense, en bas et à gauche, soit le premier cluster, l'autre région dense, à droite et vers le haut, soit la seconde, et la région moins dense, au milieu, soit la troisième. Au final, ce n'est pas vraiment le cas et, de plus, les clusters 0 et 1 contiennent des points qui sont éloignés des autres dans ces clusters.

Le partitionnement en *k*-moyennes suppose également que toutes les directions ont la même importance pour chaque cluster. Le tracé suivant montre un jeu de données à deux dimensions où les données sont clairement séparées en trois parties (voir la Figure 3.28). Pourtant, ces groupes sont étirés le long de la diagonale. Du fait que l'algorithme des *k*-moyennes ne considère que les questions de distance par rapport au centre le plus proche, il ne peut pas gérer ce type de données :

**In[53]:**

```
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)

transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)

kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)

mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
    markers='^', markeredgewidth=2)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

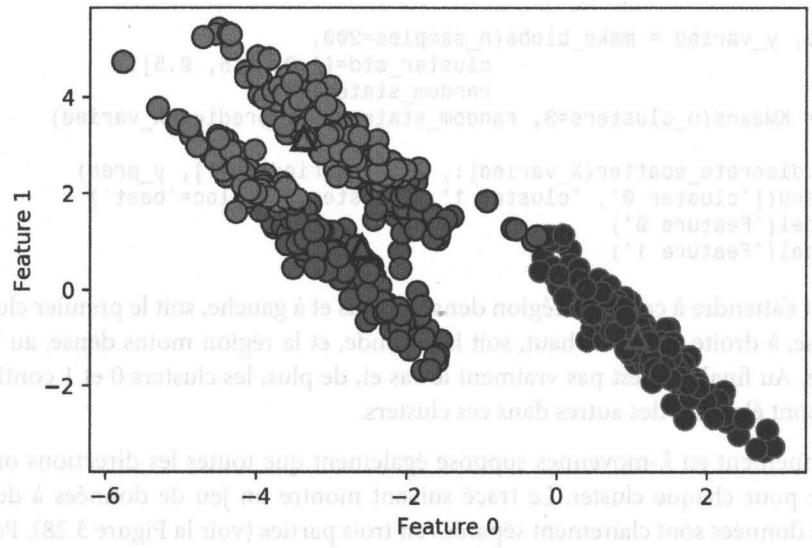



*Figure 3.28 : L'algorithme des k-moyennes échoue à identifier des clusters non sphériques.*

De même, l'algorithme fonctionne mal dans le cas où les clusters ont une forme plus complexe, comme avec le jeu de données `two_moons` déjà rencontré dans le Chapitre 2 (voir la Figure 3.29) :

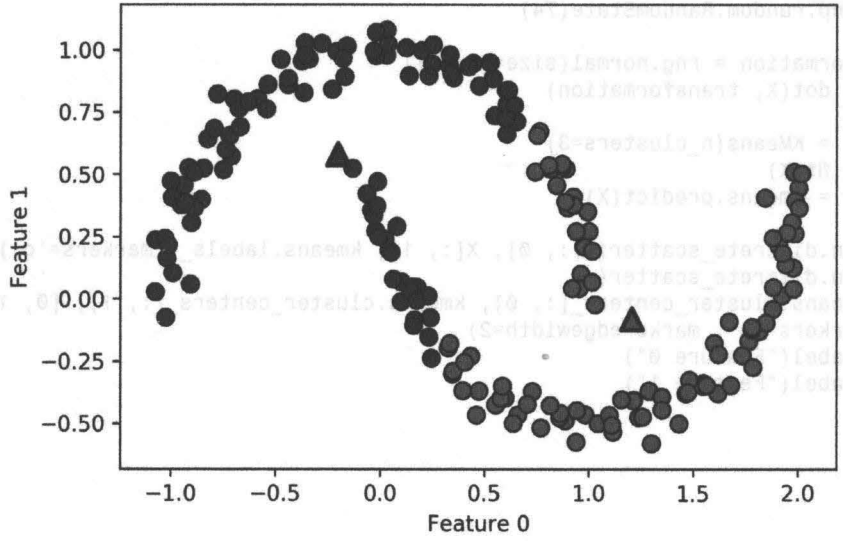



*Figure 3.29 : L'algorithme des k-moyennes échoue à identifier des clusters ayant une forme complexe.*

In[54]:

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm2, s=60, edgecolor='k')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', c=[mglearn.cm2(0), mglearn.cm2(1)], s=100, linewidth=2,
            edgecolor='k')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Nous aurions pu espérer que l'algorithme de partitionnement découvrirait les formes des deux demi-lunes, mais il n'est en fait pas capable de traiter ce type de données.

### Quantification vectorielle

L'algorithme des *k*-moyennes est bien un algorithme de partitionnement, de *clustering*. Mais il existe des parallèles intéressants entre nos *k*-moyennes et les méthodes de décomposition telles que PCA et NMF, traitées plus haut dans ce chapitre.

Vous vous rappelez que PCA essaie de trouver les directions ayant la variance maximale dans les données, tandis que NMF essaie de son côté de trouver des composantes additives, qui correspondent souvent à des « extrémités » ou à un « axe » dans les données (reportez-vous à la Figure 3.13). Ces deux méthodes s'efforcent d'exprimer les points de donnée sous la forme d'une somme pondérée d'un certain nombre de composantes. D'un autre côté, les *k*-moyennes essaient de représenter chaque point de donnée en utilisant un centre de cluster. Vous pouvez vous figurer cette méthode comme s'il s'agissait de représenter chaque point de donnée à l'aide d'une seule composante, fournie en l'occurrence par le centre du cluster. Cette façon de se représenter les *k*-moyennes comme une méthode de décomposition, dans laquelle chaque point est représenté à l'aide d'une seule composante, est appelée *quantification vectorielle* (*vector quantization*).

Comparons l'une à côté de l'autre les méthodes PCA, NMF et *k*-moyennes en montrant les composantes extraites (voir la Figure 3.30), ainsi que la reconstruction des visages à partir du jeu de test en utilisant 100 composantes (voir la Figure 3.31). Dans le cas des *k*-moyennes, la reconstruction est le centre de cluster le plus proche trouvé dans le jeu d'apprentissage :

In[55]:

```
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
nmf = NMF(n_components=100, random_state=0)
nmf.fit(X_train)
pca = PCA(n_components=100, random_state=0)
pca.fit(X_train)
```

```
kmeans = KMeans(n_clusters=100, random_state=0)
kmeans.fit(X_train)

X_reconstructed_pca = pca.inverse_transform(pca.transform(X_test))
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)
```

**In[56]:**

```
fig, axes = plt.subplots(3, 5, figsize=(8, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("Extracted Components")
for ax, comp_kmeans, comp_pca, comp_nmf in zip(
        axes.T, kmeans.cluster_centers_, pca.components_, nmf.components_):
    ax[0].imshow(comp_kmeans.reshape(image_shape))
    ax[1].imshow(comp_pca.reshape(image_shape), cmap='viridis')
    ax[2].imshow(comp_nmf.reshape(image_shape))

axes[0, 0].set_ylabel("kmeans")
axes[1, 0].set_ylabel("pca")
axes[2, 0].set_ylabel("nmf")

fig, axes = plt.subplots(4, 5, subplot_kw={'xticks': (), 'yticks': ()},
                         figsize=(8, 8))
fig.suptitle("Reconstructions")
for ax, orig, rec_kmeans, rec_pca, rec_nmf in zip(
        axes.T, X_test, X_reconstructed_kmeans, X_reconstructed_pca,
        X_reconstructed_nmf):

    ax[0].imshow(orig.reshape(image_shape))
    ax[1].imshow(rec_kmeans.reshape(image_shape))
    ax[2].imshow(rec_pca.reshape(image_shape))
    ax[3].imshow(rec_nmf.reshape(image_shape))

axes[0, 0].set_ylabel("original")
axes[1, 0].set_ylabel("kmeans")
axes[2, 0].set_ylabel("pca")
axes[3, 0].set_ylabel("nmf")
```




*Figure 3.30 : Comparaison entre les centres des clusters des k-moyennes avec les composantes trouvées par PCA et NMF.*




*Figure 3.31 : Comparaison des reconstructions d'images en utilisant les k-moyennes, PCA et NMF avec 100 composantes (ou centres de clusters) – les k-moyennes utilisent un seul centre de cluster par image.*

Un aspect intéressant de la quantification vectorielle en utilisant les *k*-moyennes est que vous pouvez utiliser bien plus de clusters que le nombre de dimensions de l'entrée pour encoder vos données. Revenons au jeu de données `two_moons`. Avec PCA et NMF, on ne peut rien en faire de plus, puisque ce jeu ne comporte que deux dimensions. Le réduire à une seule dimension avec PCA ou NMF ne ferait que détruire complètement la structure des données. Mais nous pouvons

trouver une représentation plus expressive avec les *k*-moyennes en utilisant davantage de centres de clusters (voir la Figure 3.32) :

**In[57]:**

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

kmeans = KMeans(n_clusters=10, random_state=0)
kmeans.fit(X)
y_pred = kmeans.predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=60, cmap='Paired')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=60,
            marker='^', c=range(kmeans.n_clusters), linewidth=2, cmap='Paired')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
print("Cluster memberships:\n{}".format(y_pred))
```

**Out[57]:**

```
Cluster memberships:
[9 2 5 4 2 7 9 6 9 6 1 0 2 6 1 9 3 0 3 1 7 6 8 6 8 5 2 7 5 8 9 8 6 5 3 7 0
 9 4 5 0 1 3 5 2 8 9 1 5 6 1 0 7 4 6 3 3 6 3 8 0 4 2 9 6 4 8 2 8 4 0 4 0 5
 6 4 5 9 3 0 7 8 0 7 5 8 9 8 0 7 3 9 7 1 7 2 2 0 4 5 6 7 8 9 4 5 4 1 2 3 1
 8 8 4 9 2 3 7 0 9 9 1 5 8 5 1 9 5 6 7 9 1 4 0 6 2 6 4 7 9 5 5 3 8 1 9 5 6
 3 5 0 2 9 3 0 8 6 0 3 3 5 6 3 2 0 2 3 0 2 6 3 4 4 1 5 6 7 1 1 3 2 4 7 2 7
 3 8 6 4 1 4 3 9 9 5 1 7 5 8 2]
```

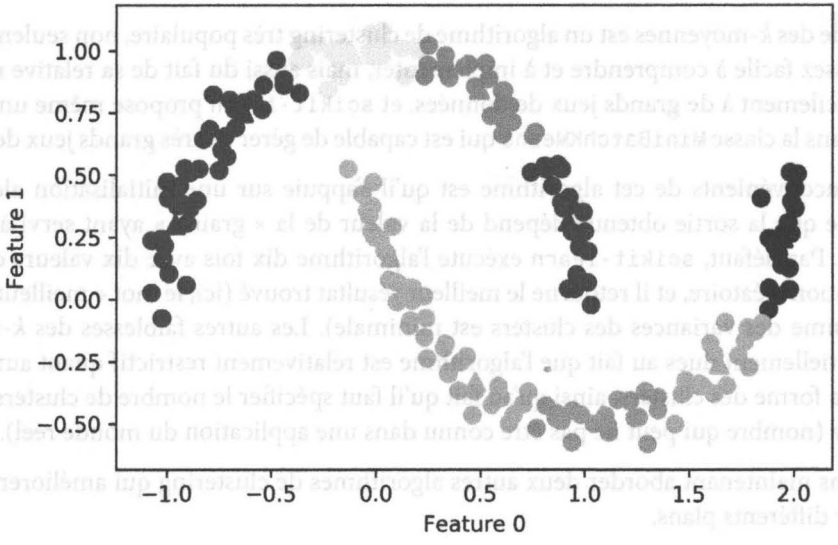



*Figure 3.32 : Utiliser de nombreux clusters k-moyennes pour couvrir les variations d'un jeu de données complexe.*

Nous avons utilisé 10 centres de clusters, ce qui signifie que chaque point est maintenant associé à un nombre entre 0 et 9. Nous pouvons le voir au fait que les données sont représentées par 10 composantes (autrement dit, nous avons 10 nouvelles caractéristiques), toutes les caractéristiques étant nulles sauf celle qui représente le centre du cluster auquel le point est affecté. En utilisant cette représentation en 10 dimensions, nous pourrions maintenant séparer les deux formes de demi-lune en utilisant un modèle linéaire, ce qui n'aurait pas été possible à partir des deux caractéristiques d'origine. Il est également envisageable d'obtenir une représentation encore plus expressive des données en utilisant comme caractéristiques les distances vers chaque centre de cluster. Cela peut être réalisé en utilisant la méthode `transform` de `kmeans` :

**In[58]:**

```
distance_features = kmeans.transform(X)
print("Distance feature shape: {}".format(distance_features.shape))
print("Distance features:\n{}".format(distance_features))
```

**Out[58]:**

```
Distance feature shape: (200, 10)
Distance features:
[[ 0.922  1.466  1.14  ...,  1.166  1.039  0.233]
 [ 1.142  2.517  0.12  ...,  0.707  2.204  0.983]
 [ 0.788  0.774  1.749 ...,  1.971  0.716  0.944]
 ...,
 [ 0.446  1.106  1.49  ...,  1.791  1.032  0.812]
 [ 1.39   0.798  1.981 ...,  1.978  0.239  1.058]
 [ 1.149  2.454  0.045 ...,  0.572  2.113  0.882]]
```

La méthode des *k*-moyennes est un algorithme de clustering très populaire, non seulement parce qu'il est assez facile à comprendre et à implémenter, mais aussi du fait de sa relative rapidité. Il s'adapte facilement à de grands jeux de données, et `scikit-learn` propose même une variante étendue dans la classe `MiniBatchKMeans` qui est capable de gérer de très grands jeux de données.

L'un des inconvénients de cet algorithme est qu'il s'appuie sur une initialisation aléatoire, ce qui signifie que la sortie obtenue dépend de la valeur de la « graine » ayant servi à cette initialisation. Par défaut, `scikit-learn` exécute l'algorithme dix fois avec dix valeurs différentes d'initialisation aléatoire, et il retourne le meilleur résultat trouvé (ici, le mot « meilleur » signifie que la somme des variances des clusters est minimale). Les autres faiblesses des *k*-moyennes sont essentiellement dues au fait que l'algorithme est relativement restrictif quant aux suppositions sur la forme des clusters, ainsi qu'au fait qu'il faut spécifier le nombre de clusters que vous recherchez (nombre qui peut ne pas être connu dans une application du monde réel).

Nous allons maintenant aborder deux autres algorithmes de clustering qui améliorent ces propriétés sur différents plans.

## Clustering agglomératif

Cette appellation fait référence à une collection d'algorithmes de clustering qui sont tous construits selon les mêmes principes : l'algorithme commence par déclarer chaque point comme étant son propre cluster, puis il fusionne les deux clusters les plus similaires (ou les plus proches), et ce jusqu'à ce qu'un certain critère d'arrêt soit satisfait. Le critère d'arrêt implémenté dans `scikit-learn` est le nombre de clusters, et donc les clusters similaires sont fusionnés jusqu'à ce que seul reste ce nombre de clusters. Il y a également plusieurs critères de *lien* (*linkage*) qui spécifient la façon exacte de mesurer ce que représentent les « clusters les plus similaires ». Cette mesure est toujours définie entre deux clusters existants.

Les trois liens suivants sont implémentés dans `scikit-learn` :

`ward`

C'est le choix par défaut. Il sélectionne les deux clusters à fusionner de manière à ce que la variance augmente le moins à l'intérieur de tous les clusters (ce que l'on appelle aussi l'*inertie intraclasse*). Cela conduit souvent à des clusters qui sont de taille relativement égale.

`average`

Ce lien fusionne les deux clusters qui ont la plus petite distance moyenne entre tous leurs points.

`complete`

Ce lien, également appelé « maximum », fusionne les deux clusters qui ont la plus petite distance maximale entre leurs points.

Le critère de `ward` fonctionne avec la plupart des jeux de données, et c'est lui que nous allons utiliser dans nos exemples. Si les clusters ont des nombres très dissemblables de membres (par exemple, si l'un d'entre eux est beaucoup plus grand que tous les autres), les liens `average` ou `complete` pourraient être meilleurs.

Le tracé qui suit illustre la progression de ce type de clustering avec un jeu de données ayant deux dimensions, la recherche portant sur trois clusters (voir la Figure 3.33) :

**In[59]:**

```
mglearn.plots.plot_agglomerative_algorithm()
```

Initialement, chaque point est son propre cluster. Ensuite, les deux clusters les plus proches sont fusionnés lors de chaque étape. Dans les quatre premières étapes, deux clusters ne comportant qu'un seul point sont sélectionnés et groupés pour former des clusters contenant deux points. Dans l'étape 5, un des clusters à deux points est étendu vers un troisième point, et ainsi de suite. Après l'étape, il ne reste plus que trois clusters. Comme il s'agit du nombre de clusters demandé, l'algorithme s'arrête à ce stade.

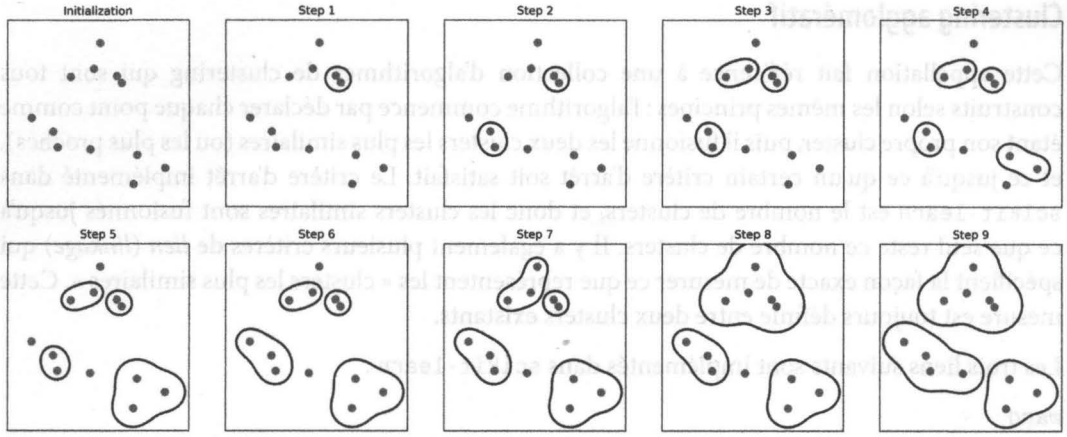



*Figure 3.33 : Le clustering agglomératif joint de manière itérative les deux clusters les plus proches.*

Regardons comment ce mode de clustering fonctionne sur le jeu de données simple, avec trois clusters, que nous avons déjà utilisé à plusieurs reprises. Du fait de la manière dont se déroule l'algorithme, le clustering agglomératif ne peut pas effectuer de prédictions sur de nouveaux points. Par conséquent, la classe `AgglomerativeClustering` ne possède pas de méthode `predict`. Pour construire le modèle et former les clusters sur le jeu d'apprentissage, nous utilisons à la place la méthode `fit_predict` (il serait aussi possible de se servir de l'attribut `labels_`, comme nous l'avons fait pour les *k*-moyennes). Le résultat est montré sur la Figure 3.34 :

**In[60]:**

```
from sklearn.cluster import AgglomerativeClustering
X, y = make_blobs(random_state=1)

agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignment)
plt.legend(["Cluster 0", "Cluster 1", "Cluster 2"], loc="best")
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Comme on pouvait s'y attendre, l'algorithme effectue parfaitement le regroupement en clusters. Bien que l'implémentation de l'algorithme dans `scikit-learn` vous oblige à spécifier le nombre de clusters, il existe des méthodes qui fournissent une certaine aide pour choisir la bonne valeur. C'est ce que nous allons voir maintenant.

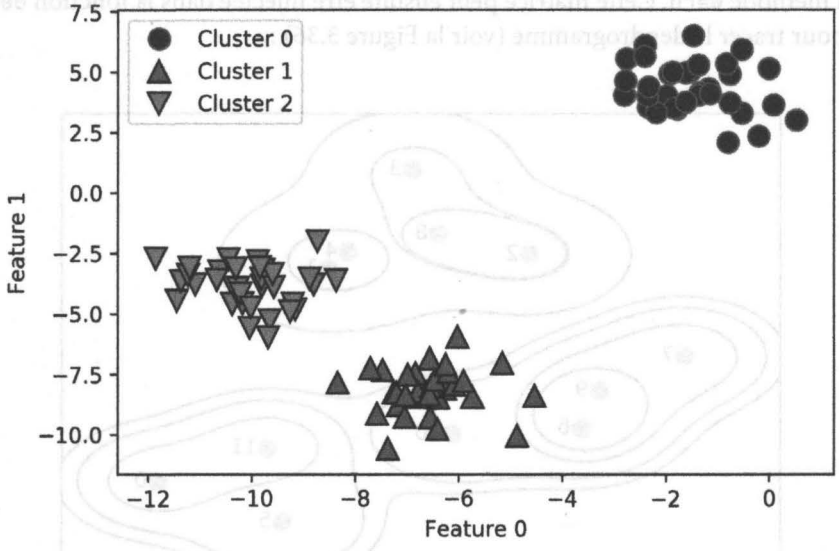



*Figure 3.34 : Détermination des clusters en utilisant le clustering agglomératif avec trois clusters.*

### Clustering hiérarchique et dendrogrammes

Le clustering agglomératif produit une forme dite *ascendante* de ce que l'on appelle un *clustering hiérarchique*. Ce clustering procède de manière itérative, où chaque point est initialement un cluster à lui tout seul, jusqu'à devenir membre d'un certain cluster final. Toutes les étapes intermédiaires fournissent un nouveau partitionnement des données (avec un nombre de clusters différent). Il est parfois utile d'observer conjointement tous les regroupements en clusters possibles. C'est ce que fait l'exemple suivant en partant de la situation illustrée sur la Figure 3.33, ce qui fournit un aperçu de la façon dont chaque cluster est divisible en clusters plus petits (voir la Figure 3.35) :

**In[61]:**

```
mglearn.plots.plot_agglomerative()
```

Bien que cette visualisation fournisse une vue très détaillée du clustering hiérarchique, elle s'appuie sur la nature bidimensionnelle des données, et ne peut donc pas être utilisée avec des jeux de données possédant plus de deux dimensions. Cependant, il existe un autre outil qui corrige cette lacune et que l'on appelle un *dendrogramme*.

Malheureusement, `scikit-learn` ne sait pas à l'heure actuelle dessiner des dendrogrammes. Par contre, vous pouvez facilement les générer grâce à `SciPy`. Les algorithmes de clustering de `SciPy` ont une interface légèrement différente de leurs équivalents dans `scikit-learn`. `SciPy` dispose d'une fonction qui prend un tableau de données X, et qui calcule une matrice de liens en

utilisant la méthode `ward`. Cette matrice peut ensuite être injectée dans la fonction `dendrogram` de `SciPy` pour tracer le dendrogramme (voir la Figure 3.36) :

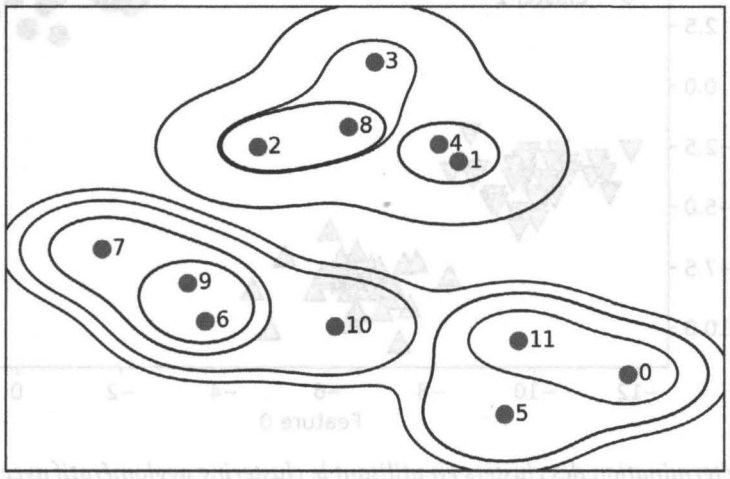



*Figure 3.35 : Clustering hiérarchique (illustré par les lignes) avec numérotation des points de données (voir la Figure 3.36).*

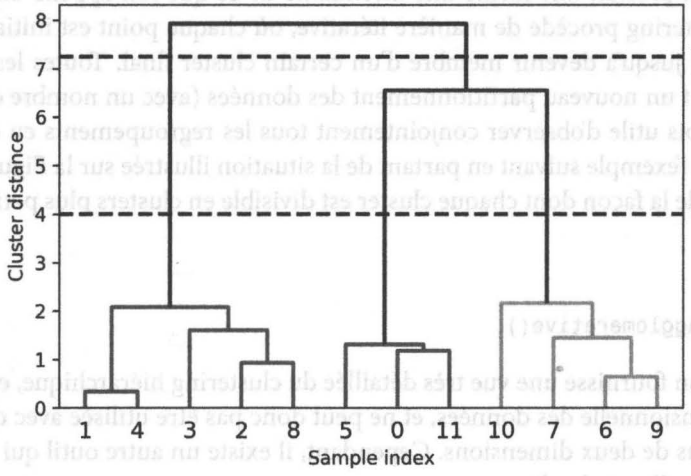



*Figure 3.36 : Dendrogramme du clustering montré sur la Figure 3.35, avec des lignes indiquant les divisions en deux et en trois clusters.*

In[62]:

```
from scipy.cluster.hierarchy import dendrogram, ward

X, y = make_blobs(random_state=0, n_samples=12)
linkage_array = ward(X)
dendrogram(linkage_array)

ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')

ax.text(bounds[1], 7.25, ' two clusters', va='center', fontdict={'size': 15})
ax.text(bounds[1], 4, ' three clusters', va='center', fontdict={'size': 15})
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```

Le dendrogramme montre les points de données en bas du graphique (ils sont numérotés de 0 à 11, comme sur la Figure 3.35). Un arbre est ensuite tracé en partant de ces points (qui sont donc les clusters de départ), un nouveau nœud parent étant ajouté chaque fois que deux clusters sont fusionnés.

En regardant le dendrogramme de bas en haut, on voit que les points 1 et 4 sont fusionnés en premier (comme on peut le constater sur la Figure 3.33). Les points 6 et 9 sont ensuite fusionnés pour produire un nouveau cluster, et ainsi de suite. Au niveau le plus haut, il y a deux branches, l'une formée des points 5, 0, 11, 10, 7, 6 et 9 (partie droite du tracé), et l'autre formée des points 1, 4, 3, 2 et 8 (partie gauche du tracé). Nous retrouvons bien les deux grands clusters représentés sur la Figure 3.35.

L'axe des ordonnées dans le dendrogramme ne spécifie pas uniquement à quel moment deux clusters sont fusionnés dans l'algorithme. La longueur de chaque branche montre également la distance qui sépare les clusters fusionnés. Les branches les plus longues dans ce dendrogramme sont les trois lignes qui sont traversées par le trait pointillé qui a pour légende « three clusters ». Elles indiquent que passer de trois à deux clusters signifie fusionner certains points très éloignés. Nous retrouvons cela en haut du graphique, où la fusion des deux clusters restant en un seul forme comme un pont qui couvre une distance relativement grande.

Malheureusement, le clustering agglomératif échoue également à séparer des formes complexes, comme dans le cas du jeu de données `two_moons`. Mais cela n'est pas vrai pour l'algorithme que nous allons voir maintenant, DBSCAN.

## DBSCAN

Voyons un autre algorithme de clustering très utile, DBSCAN (qui est l'abréviation de *density-based spatial clustering of applications with noise*). Les principaux bénéfices apportés par DBSCAN sont le fait qu'il ne demande pas à ce que l'utilisateur définisse *a priori* le nombre

de clusters, qu'il est capable de capturer des clusters de forme complexe, et qu'il peut identifier des points qui ne font partie d'aucun cluster. Cet algorithme est un peu plus lent que les autres méthodes de clustering que nous avons étudiées plus haut, mais il s'adapte bien à des jeux de données relativement grands.

DBSCAN fonctionne en identifiant des points qui se trouvent dans des régions « encombrées » de l'espace des caractéristiques, c'est-à-dire là où de nombreux points sont proches les uns des autres. Ces régions sont dites *denses*. L'idée sous-jacente est donc que les clusters forment des régions de données denses qui sont séparées par des régions relativement vides.

Les points qui se trouvent dans une région dense sont dits *core points* (ou *points intérieurs*). Ils se définissent de la manière suivante. Il y a deux paramètres dans DBSCAN : `min_samples` et `eps` (pour *epsilon*). S'il y a au moins `min_samples` points se trouvant à une distance inférieure à `eps` d'un certain point de donnée, alors celui-ci est classifié comme étant un point intérieur. Les points intérieurs qui sont plus proches les uns des autres que la distance `eps` sont placés dans le même cluster par DBSCAN.

L'algorithme choisit un point arbitraire pour débuter. Il trouve ensuite tous les points qui en sont à une distance d'`eps` ou moins. Si le nombre de points trouvés de cette manière est plus petit que `min_samples`, alors le point de départ est caractérisé comme étant du *bruit* (*noise*), ce qui signifie qu'il n'appartient à aucun cluster. Dans le cas inverse, il s'agit d'un *core point*, et il est affecté à une nouvelle étiquette de cluster. Tous ses voisins (dans les limites d'`eps`) sont ensuite visités. S'ils n'ont pas encore été placés dans un cluster, ils sont affectés au nouveau cluster qui vient d'être créé. Si des points intérieurs sont trouvés lors de cette étape, leurs voisins sont visités à leur tour, et ainsi de suite. Le cluster croît jusqu'à ce qu'il n'y ait plus de point à traiter dans les limites de la distance *eps* (ou *epsilon-voisinage*, ou encore $\epsilon$*-voisinage*). L'algorithme choisit ensuite un autre point qui n'a pas encore été visité, et la même procédure est répétée.

À la fin, il y a trois types de points : les points intérieurs (*core points*), ceux qui se trouvent dans l'*epsilon-voisinage* de points intérieurs (*boundary points*, ou points frontière) et le bruit. Lorsque l'algorithme DBSCAN est exécuté de multiples fois sur le même jeu de données, les points intérieurs ainsi que le bruit seront inchangés. Par contre, les autres points peuvent être voisins de points intérieurs dans plusieurs clusters. Dans ce cas, leur appartenance à tel ou tel cluster dépend de l'ordre dans lequel ils ont été visités. Pour autant, le nombre de situations de ce type est généralement faible, et cette légère dépendance à l'ordre des points n'est pas importante.

Appliquons DBSCAN sur le jeu de données synthétique que nous avons utilisé plus haut pour le clustering agglomératif. Là encore, DBSCAN ne permet pas d'effectuer des prédictions sur de nouvelles données de test, et donc nous allons utiliser la méthode `fit_predict` pour réaliser la division en clusters et la récupération des étiquettes de ces clusters, le tout en une seule étape :

**In[63]:**

```
from sklearn.cluster import DBSCAN
X, y = make_blobs(random_state=0, n_samples=12)
```

```
dbscan = DBSCAN()
clusters = dbscan.fit_predict(X)
print("Cluster memberships:\n{}".format(clusters))
```

**Out[63]:**

```
Cluster memberships:
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

Comme vous pouvez le constater, tous les points de données ont pour étiquette -1, ce qui signifie qu'ils sont tous considérés comme étant du bruit. C'est une conséquence des valeurs par défaut pour `eps` et `min_samples`, qui ne sont pas adaptées aux petits jeux de données comme celui de notre exemple. Les affectations des clusters pour différentes valeurs de `min_samples` et d'`eps` sont montrées ci-après, et visualisées sur la Figure 3.37 :

**In[64]:**

```
mglearn.plots.plot_dbscan()
```

**Out[64]:**

```
min_samples: 2 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]
min_samples: 2 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 2 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 2 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 3 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]
min_samples: 3 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 3 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 3 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 5 eps: 1.000000  cluster: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
min_samples: 5 eps: 1.500000  cluster: [-1  0  0  0  0 -1 -1 -1  0 -1 -1 -1]
min_samples: 5 eps: 2.000000  cluster: [-1  0  0  0  0 -1 -1 -1  0 -1 -1 -1]
min_samples: 5 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
```

Sur ce tracé, les points qui appartiennent à un cluster sont pleins tandis que les bruits sont représentés par un cercle creux. Les points intérieurs sont un peu plus grands que les autres, ce qui permet de les distinguer facilement. Plus la valeur du paramètre `eps` augmente (en allant de gauche à droite sur la figure), et plus le nombre de points inclus dans un cluster augmente également. Cela fait grossir les clusters, mais pourrait aussi conduire à ce que de multiples clusters finissent par fusionner en un seul. En augmentant la valeur du paramètre `min_samples` (de haut en bas), on diminue le nombre de points intérieurs, et on augmente donc en même temps la quantité de bruit.

D'une certaine manière, le paramètre `eps` est le plus important, car il détermine le sens du mot « proche ». Si `eps` est très petit, la conséquence peut être qu'il n'y a aucun point intérieur, et donc que le jeu d'apprentissage semble ne contenir que du bruit. Inversement, si `eps` est trop grand, tous les points finissent par ne former qu'un seul cluster.

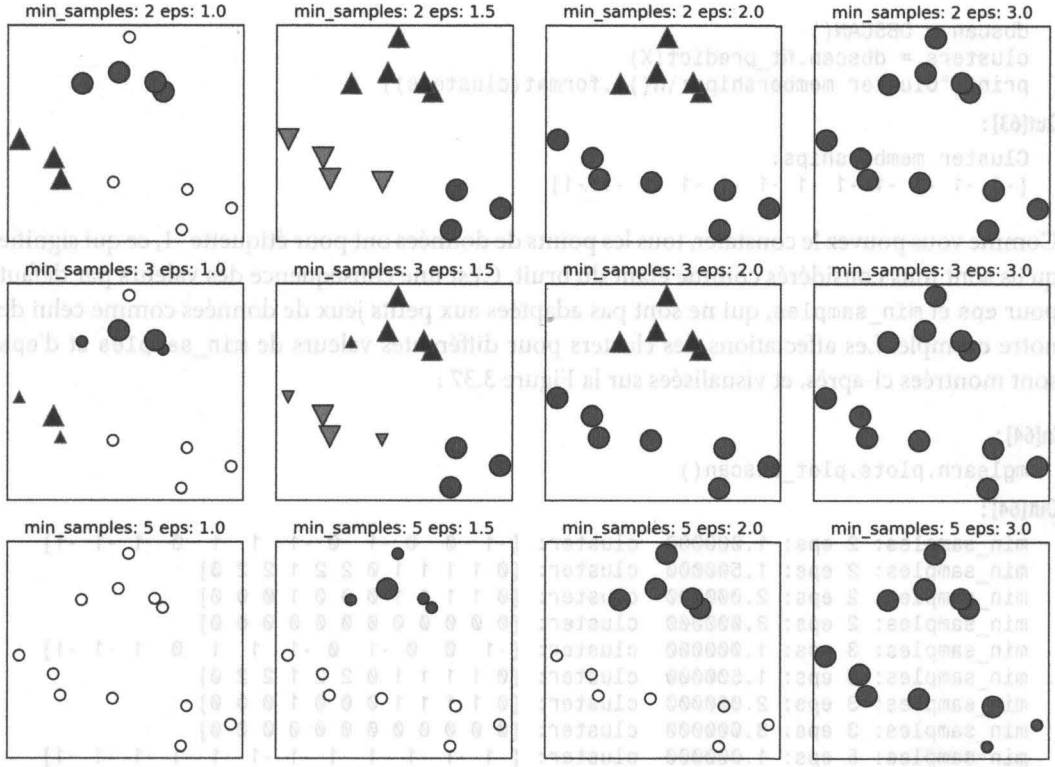



*Figure 3.37 : Partitionnements en clusters trouvés par DBSCAN avec différentes valeurs pour les paramètres* `min_samples` *et* `eps`.

Le paramètre `min_samples` détermine surtout le statut des points qui se trouvent dans les régions moins denses (points dits atypiques, anormaux ou encore aberrants, ou *outliers*, ou bien formant leur propre cluster). Si vous augmentez la valeur de `min_samples`, donc la quantité de points « proches » nécessaires pour former un cluster, vous rendez les conditions de fabrication de ceux-ci plus draconiennes, et vous augmentez donc la quantité de bruit. Autrement dit, `min_samples` détermine la taille minimale des clusters. Vous pouvez voir clairement cela sur la Figure 3.37 en passant de `min_samples=3` à `min_samples=5` avec `eps=1.5`. Dans le premier cas, il y a trois clusters contenant respectivement trois, quatre et cinq points. Avec `min_samples=5`, les deux clusters les plus petits (ceux qui contenaient trois et quatre points) sont maintenant considérés comme étant du bruit, seul restant le plus grand cluster.

Même si DBSCAN n'oblige pas à définir explicitement le nombre de clusters à chercher, `eps` contrôle implicitement celui-ci. Trouver une bonne valeur pour `eps` est parfois plus facile une fois les données recalibrées à l'aide de `StandardScaler` ou de `MinMaxScaler`, du fait que cela permet de s'assurer que toutes les caractéristiques sont dans des plages similaires.

La Figure 3.38 montre le résultat produit en appliquant DBSCAN au jeu de données `two_moons`. L'algorithme trouve parfaitement les deux demi-cercles et les sépare en utilisant simplement les paramètres par défaut :

**In[65]:**

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

dbscan = DBSCAN()
clusters = dbscan.fit_predict(X_scaled)
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mglearn.cm2, s=60)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

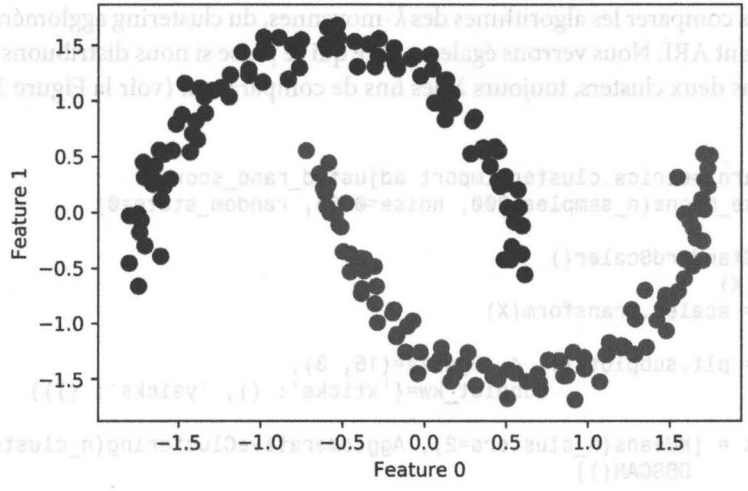



*Figure 3.38 : Regroupements en clusters trouvés par DBSCAN avec la valeur par défaut d'*`eps`*, 0.5.*

Ici, l'algorithme produit le nombre désiré de clusters, deux, et donc la valeur par défaut du paramètre `eps` (`0.5`) semble fonctionner parfaitement. Si nous diminuons `eps` jusqu'à `0.2`, nous obtiendrons huit clusters, ce qui est clairement beaucoup trop. Par contre, une valeur de `0.7` ne donnerait qu'un unique cluster.

Lorsque vous utilisez DBSCAN, vous devez faire très attention à la manière dont les clusters sont retournés. Une valeur de -1 indique un bruit, ce qui est très bien pour écarter des valeurs aberrantes, mais pourrait aussi produire des effets inattendus lors de l'utilisation des étiquettes des clusters dans l'indexation d'un autre tableau.

## Comparer et évaluer les algorithmes de clustering

L'un des défis posés par l'application des algorithmes de clustering est qu'il est très difficile d'évaluer leur travail et de comparer les sorties produites par deux algorithmes différents. Après avoir étudié les *k*-moyennes, le clustering agglomératif et DBSCAN, nous allons essayer de les comparer sur des jeux de données provenant du monde réel.

### Évaluer le clustering à partir de la réalité de terrain

Il existe des métriques qui peuvent être utilisées pour mesurer le degré de correspondance entre la sortie produite par un algorithme de clustering et une partition connue des données. Les plus importantes de ces métriques sont les indices ARI (*ajusted rand index*, ou *indice de Rand ajusté*) et NMI (*normalized mutual information*, ou *information mutuelle normalisée*) qui fournissent toutes les deux une mesure quantitative avec une valeur optimum de 1, et une valeur de 0 si aucune correspondance n'a été trouvée (quoiqu'ARI puisse devenir négatif).

Ici, nous allons comparer les algorithmes des *k*-moyennes, du clustering agglomératif et de DBSCAN en utilisant ARI. Nous verrons également ce qui se passe si nous distribuons aléatoirement des points dans deux clusters, toujours à des fins de comparaison (voir la Figure 3.39) :

**In[66]:**

```
from sklearn.metrics.cluster import adjusted_rand_score
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                         subplot_kw={'xticks': (), 'yticks': ()})

algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
                cmap=mglearn.cm3, s=60)
axes[0].set_title("Random assignment - ARI: {:.2f}".format(
        adjusted_rand_score(y, random_clusters)))

for ax, algorithm in zip(axes[1:], algorithms):
    clusters = algorithm.fit_predict(X_scaled)
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters,
               cmap=mglearn.cm3, s=60)
    ax.set_title("{} - ARI: {:.2f}".format(algorithm.__class__.__name__,
                                           adjusted_rand_score(y, clusters)))
```

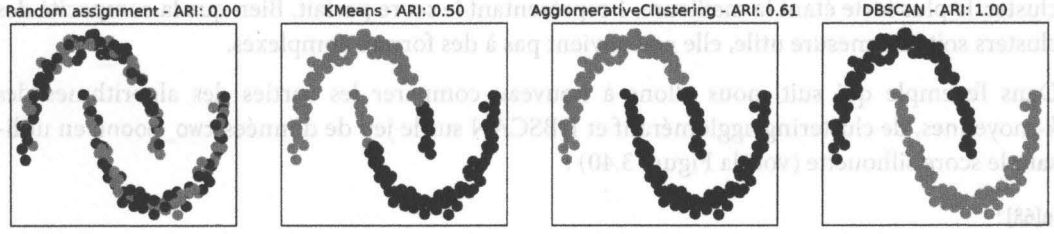



*Figure 3.39 : Comparaison entre distribution aléatoire, algorithmes des k-moyennes, de clustering agglomératif et DBSCAN appliqués au jeu de données* `two_moons` *en utilisant le score ARI supervisé.*

L'indice de Rand ajusté produit des résultats intuitifs, avec une affectation aléatoire qui obtient un score de 0, et à l'autre bout de 1 pour DBSCAN, qui retrouve parfaitement les clusters désirés.

Une erreur courante lors de l'évaluation d'un clustering de cette manière consiste à utiliser `accuracy_score` au lieu d'`adjusted_rand_score`, `normalized_mutual_info_score` ou une autre métrique. Le problème posé par le fait d'utiliser l'exactitude est que cela nécessite une correspondance exacte entre étiquettes des clusters affectées et données de terrain. Pour autant, les étiquettes des clusters elles-mêmes n'ont aucune signification particulière. La seule chose qui compte, c'est de savoir quels points se trouvent dans le même cluster :

**In[67]:**

```
from sklearn.metrics import accuracy_score

clusters1 = [0, 0, 1, 1, 0]
clusters2 = [1, 1, 0, 0, 1]
print("Accuracy: {:.2f}".format(accuracy_score(clusters1, clusters2)))
print("ARI: {:.2f}".format(adjusted_rand_score(clusters1, clusters2)))
```

**Out[67]:**

```
Accuracy: 0.00
ARI: 1.00
```

### Évaluer le clustering indépendamment de la réalité de terrain

Nous venons de voir une manière d'évaluer les algorithmes de clustering. Mais, en pratique, il y a un sérieux problème lorsque l'on utilise des mesures comme ARI. Lorsque l'on applique des algorithmes de clustering, il n'existe généralement pas de données provenant du monde réel avec lesquelles comparer les résultats. Sinon, cela impliquerait vraisemblablement que nous serions à même de construire un modèle supervisé comme un classifieur. C'est pourquoi des métriques comme ARI et NMI servent essentiellement dans le cadre du développement d'algorithmes, et non pas à évaluer le degré de réussite d'une application.

Il existe donc aussi des métriques applicables au clustering et qui ne nécessitent pas la présence de mesures de terrain, comme le *coefficient de silhouette*. Ce coefficient mesure la compacité d'un

cluster, la plus forte étant la meilleure, 1 représentant le score parfait. Bien que la compacité des clusters soit une mesure utile, elle ne convient pas à des formes complexes.

Dans l'exemple qui suit, nous allons à nouveau comparer les sorties des algorithmes des *k*-moyennes, de clustering agglomératif et DBSCAN sur le jeu de données `two_moons` en utilisant le score silhouette (voir la Figure 3.40) :

**In[68] :**

```
from sklearn.metrics.cluster import silhouette_score

X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                         subplot_kw={'xticks': (), 'yticks': ()})

random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
                cmap=mglearn.cm3, s=60)
axes[0].set_title("Random assignment: {:.2f}".format(
    silhouette_score(X_scaled, random_clusters)))

algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

for ax, algorithm in zip(axes[1:], algorithms):
    clusters = algorithm.fit_predict(X_scaled)
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mglearn.cm3,
               s=60)
    ax.set_title("{} : {:.2f}".format(algorithm.__class__.__name__,
                                      silhouette_score(X_scaled, clusters)))
```

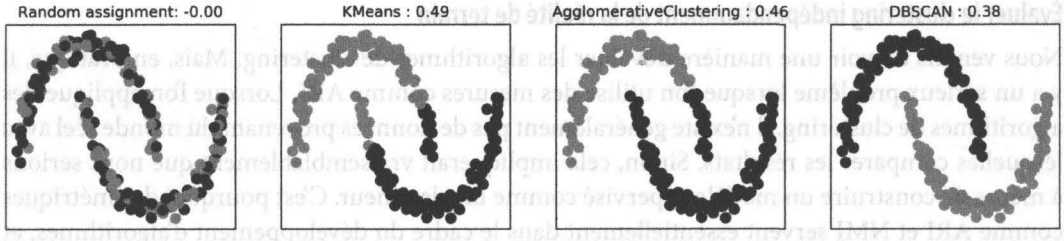



*Figure 3.40 : Comparaison entre distribution aléatoire, algorithmes des k-moyennes, de clustering agglomératif et DBSCAN appliqués au jeu de données* `two_moons` *en utilisant le score silhouette non supervisé – le résultat plus intuitif de DBSCAN obtient un score plus faible que les k-moyennes du fait de la forme complexe du jeu.*

Comme vous pouvez le constater, c'est *k*-moyennes qui obtient la valeur la plus élevée pour le coefficient de silhouette, même si nous pourrions tout à fait préférer le résultat produit par DBSCAN. Une stratégie un peu meilleure pour l'évaluation des clusters consiste à utiliser des métriques basées sur la *stabilité*. La méthode consiste maintenant à relancer un algorithme après avoir ajouté aux données une certaine quantité de bruit, ou bien après avoir modifié certains paramètres, puis à comparer les résultats obtenus. L'idée sous-jacente est que, si de nombreux changements de paramètres et/ou perturbations dans les données renvoient le même résultat, alors il est probable que l'algorithme est digne de confiance. Malheureusement, cette stratégie n'est pas implémentée dans `scikit-learn` au moment où ce livre est écrit.

Même si nous obtenons un clustering très stable, ou un score de silhouette très élevé, cela ne nous dit toujours pas s'il est possible d'en tirer une signification sémantique, ou même si ce clustering reflète un aspect des données qui puisse nous intéresser. Revenons à l'exemple des photos de visages. Nous voudrions retrouver des groupes de visages similaires, qu'il s'agisse d'hommes ou de femmes, de personnes jeunes ou âgées, d'individus avec ou sans barbe et ainsi de suite. Supposons que nous voulions répartir les données en deux clusters, et que tous les algorithmes soient d'accord sur la manière dont les points devraient être regroupés en clusters. Pour autant, cela ne nous indique pas si les clusters trouvés correspondent d'une manière ou d'une autre aux concepts qui nous intéressent. Il se pourrait par exemple que des positions décalées vers la droite ou vers la gauche soient prises pour des vues de face, que des images de nuit et de jour soient confondues, ou encore qu'il y ait une distinction entre photos prises avec un iPhone ou avec un appareil Android. La seule manière de savoir si le clustering correspond à quelque chose qui nous intéresse est d'analyser manuellement les clusters.

### Comparer les algorithmes sur le jeu de données des visages

Voyons comment appliquer les *k*-moyennes, DBSCAN et le clustering agglomératif aux images de visages déjà étudiées, et si cela nous permet de trouver des structures intéressantes. Pour cela, nous allons nous servir d'une représentation sous la forme de vecteurs propres des données produite par `PCA(whiten=True)` avec cent composantes :

**In[69]:**

```
from sklearn.decomposition import PCA
pca = PCA(n_components=100, whiten=True, random_state=0)
pca.fit_transform(X_people)
X_pca = pca.transform(X_people)
```

Nous avons vu plus haut qu'il s'agit d'une représentation sémantiquement plus significative des visages que ce que nous proposent les images raw elles-mêmes. Cela rend d'ailleurs plus rapides les calculs. Un bon exercice consisterait à exécuter les différentes expérimentations qui suivent sur les données originales, sans PCA, pour voir si vous trouvez des clusters similaires.

#### Analyser le jeu de photos de visages avec DBSCAN

Nous allons commencer par appliquer DBSCAN, que nous venons d'étudier :

**In[70]:**

```
dbscan = DBSCAN()
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

**Out[70]:**

```
Unique labels: [-1]
```

Nous pouvons constater que toutes les étiquettes retournées sont égales à -1, et donc que toutes les données ont été considérées comme étant du bruit par DBSCAN. Il y a deux choses qu'il est possible de modifier pour corriger cela : nous pouvons augmenter la valeur d'`eps` afin d'étendre le voisinage de chaque point, ou abaisser celle de `min_samples` pour considérer que des groupes de points plus petits forment également des clusters. Commençons par changer la valeur de `min_samples` :

**In[71]:**

```
dbscan = DBSCAN(min_samples=3)
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

**Out[71]:**

```
Unique labels: [-1]
```

Même en considérant des groupes de trois points, chacun est étiqueté comme étant du bruit. Essayons alors d'augmenter `eps` :

**In[72]:**

```
dbscan = DBSCAN(min_samples=3, eps=15)
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

**Out[72]:**

```
Unique labels: [-1  0]
```

Avec une valeur d'`eps` égale à 15, nous obtenons un unique cluster et des points de bruit. Nous pouvons alors utiliser ce résultat pour déterminer ce à quoi « ressemble » le bruit en comparaison du reste des données. Pour mieux comprendre ce qui se passe, regardons combien de points constituent du bruit, et combien de points se trouvent dans le cluster :

**In[73]:**

```
print("Number of points per cluster: {}".format(np.bincount(labels + 1)))
```

**Out[73]:**

```
Number of points per cluster: [  32 2031]
```

Il y a très peu de points concernés (seulement 32). Nous pouvons donc tous les observer simultanément (voir la Figure 3.41) :

**In[74]:**

```
noise = X_people[labels==-1]

fig, axes = plt.subplots(3, 9, subplot_kw={'xticks': (), 'yticks': ()},
                         figsize=(12, 4))
for image, ax in zip(noise, axes.ravel()):
    ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
```



*Figure 3.41 : Exemples du jeu de données de visages étiquetés comment étant du bruit par DBSCAN.*

En comparant ces images avec les exemples aléatoires de la Figure 3.7, nous pouvons nous faire une idée des raisons pour lesquelles elles ont été classées comme étant du bruit. Ainsi, sur la cinquième image de la seconde ligne, la personne est en train de boire dans un verre. Il y a aussi des sujets qui portent une casquette, et, sur la troisième image de la seconde ligne, il y a une main devant le visage. Les autres clichés présentent des angles plus ou moins étranges, et certains cadrages sont faits de trop près ou encore sont trop larges.

Ce type d'analyse, qui consiste à rechercher « ce qui ne va pas », est appelé *détection des données aberrantes* (*outlier detection* en anglais, le mot *outliers* étant parfois conservé tel quel). Pour une application du monde réel, nous devrions essayer de recadrer les images pour obtenir des données plus homogènes. Il n'y a pas grand-chose que l'on puisse faire dans le cas de photos où la personne porte une coiffure, ou tient quelque chose devant son visage, mais il est bon de savoir qu'il s'agit de problèmes dans les données que tout algorithme que nous pourrions appliquer devra gérer.

Si nous voulons trouver des résultats plus intéressants qu'un seul gros cluster, nous devons diminuer la valeur du paramètre `eps`, quelque part entre 15 et 0.5 (la valeur par défaut). Regardons ce qui se passe pour différentes configurations :

**In[75]:**

```
for eps in [1, 3, 5, 7, 9, 11, 13]:
    print("\neps={}".format(eps))
    dbscan = DBSCAN(eps=eps, min_samples=3)
    labels = dbscan.fit_predict(X_pca)
    print("Number of clusters: {}".format(len(np.unique(labels))))
    print("Cluster sizes: {}".format(np.bincount(labels + 1)))
```

**Out[75]:**

```
eps=1
Number of clusters: 1
Cluster sizes: [2063]

eps=3
Number of clusters: 1
Cluster sizes: [2063]

eps=5
Number of clusters: 1
Cluster sizes: [2063]

eps=7
Number of clusters: 14
Cluster sizes: [2004    3   14    7    4    3    3    4    4    3    3    5
  3    3]

eps=9
Number of clusters: 4
Cluster sizes: [1308  749    3    3]

eps=11
Number of clusters: 2
Cluster sizes: [ 415 1648]

eps=13
Number of clusters: 2
Cluster sizes: [ 120 1943]
```

Pour de petites valeurs d'eps, tous les points sont étiquetés comme étant du bruit. Pour eps=7, nous obtenons de multiples points de bruit, ainsi que de nombreux petits clusters. Pour eps=9, nous avons toujours beaucoup de bruit, mais avec cette fois un gros cluster et quelques petits. À partir de 11, nous n'obtenons plus qu'un grand cluster et du bruit.

Ce qui est intéressant à noter ici, c'est qu'il n'y a jamais plus d'un grand cluster. Au plus, il y a un grand cluster qui contient la plupart des points, et quelques petits clusters. Cela indique que nous n'avons pas dans les données deux ou trois types d'images de visages qui soient très distincts, mais plutôt que toutes les images sont plus ou moins similaires au (ou dissemblables du) reste.

Le résultat obtenu avec `eps=7` semble être le plus intéressant avec ses nombreux petits clusters. Nous pouvons étudier ce cas plus en détail en visualisant tous les points de chacun des 13 petits clusters (voir la Figure 3.42) :

**In[76]:**

```
dbscan = DBSCAN(min_samples=3, eps=7)
labels = dbscan.fit_predict(X_pca)

for cluster in range(max(labels) + 1):
    mask = labels == cluster
    n_images =  np.sum(mask)
    fig, axes = plt.subplots(1, n_images, figsize=(n_images * 1.5, 4),
                             subplot_kw={'xticks': (), 'yticks': ()})
    for image, label, ax in zip(X_people[mask], y_people[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1])
```

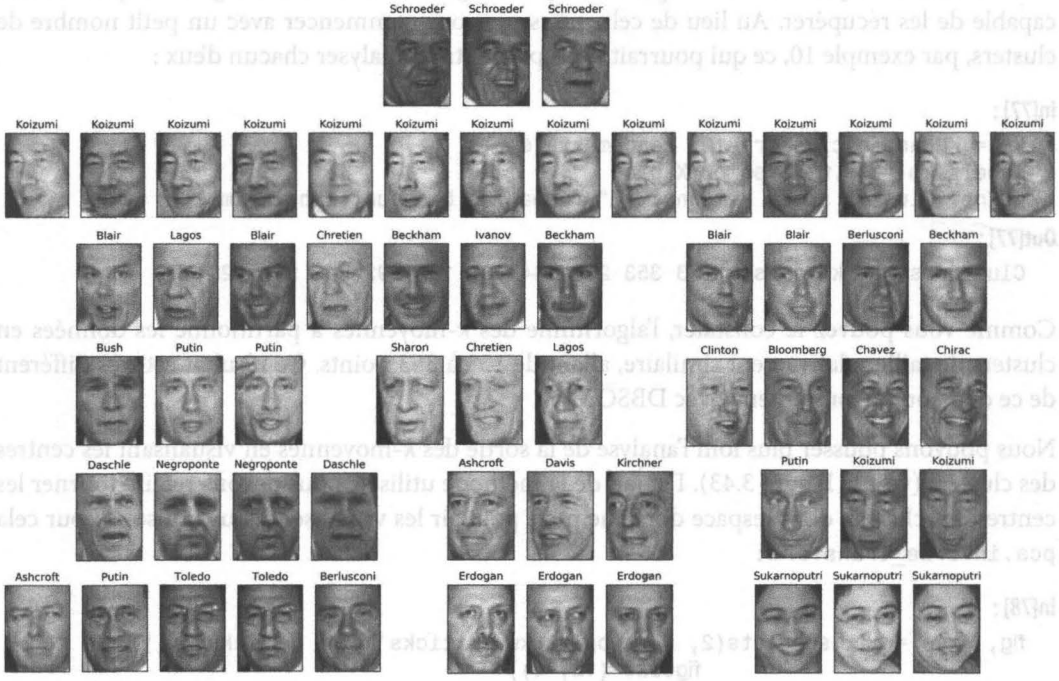



*Figure 3.42 : Clusters trouvés par DBSCAN avec* `eps=7`.

Certains des clusters correspondent à des personnes avec des visages bien distincts, du moins dans ce jeu de données, comme Koizumi ou Erdogan. À l'intérieur de chaque cluster, l'orientation du visage est relativement fixée, de même que l'expression faciale. Autrement dit, on trouve

un certain nombre de clusters qui contiennent des visages de personnes différentes, mais avec une orientation et une expression similaires.

Ainsi se termine notre analyse de l'algorithme DBSCAN appliqué au jeu de données photographiques. Comme vous pouvez le voir, nous faisons ici une analyse manuelle, différente de l'approche bien plus automatisée que nous aurions pu mener avec un apprentissage supervisé basé sur le score $R^2$ ou l'exactitude.

Voyons maintenant ce qui se passe avec l'algorithme des *k*-moyennes et le clustering agglomératif.

#### Analyser le jeu de photos de visages avec les *k*-moyennes

Nous avons vu qu'il n'était pas possible de créer plus d'un grand cluster avec DBSCAN. Le clustering agglomératif et les *k*-moyennes sont probablement plus à même de créer des clusters de taille uniforme, mais nous devons dans ce cas spécifier le nombre de clusters à rechercher. Nous pourrions définir ce paramètre en lui affectant le nombre de personnes connues dans le jeu de données, bien qu'il soit très improbable qu'un algorithme de clustering non supervisé soit capable de les récupérer. Au lieu de cela, nous pouvons commencer avec un petit nombre de clusters, par exemple 10, ce qui pourrait nous permettre d'analyser chacun d'eux :

**In[77]:**

```
km = KMeans(n_clusters=10, random_state=0)
labels_km = km.fit_predict(X_pca)
print("Cluster sizes k-means: {}".format(np.bincount(labels_km)))
```

**Out[77]:**

```
Cluster sizes k-Means: [133 353 232 144 185 136 292 249 211 128]
```

Comme vous pouvez le constater, l'algorithme des *k*-moyennes a partitionné les données en clusters de taille relativement similaire, allant de 133 à 353 points. Ce résultat est très différent de ce que nous avions obtenu avec DBSCAN.

Nous pouvons pousser plus loin l'analyse de la sortie des *k*-moyennes en visualisant les centres des clusters (voir la Figure 3.43). Du fait de la méthode utilisée, nous devons refaire tourner les centres des clusters dans l'espace d'origine pour pouvoir les visualiser. Nous utilisons pour cela `pca.inverse_transform` :

**In[78]:**

```
fig, axes = plt.subplots(2, 5, subplot_kw={'xticks': (), 'yticks': ()},
                         figsize=(12, 4))
for center, ax in zip(km.cluster_centers_, axes.ravel()):
    ax.imshow(pca.inverse_transform(center).reshape(image_shape),
              vmin=0, vmax=1)
```

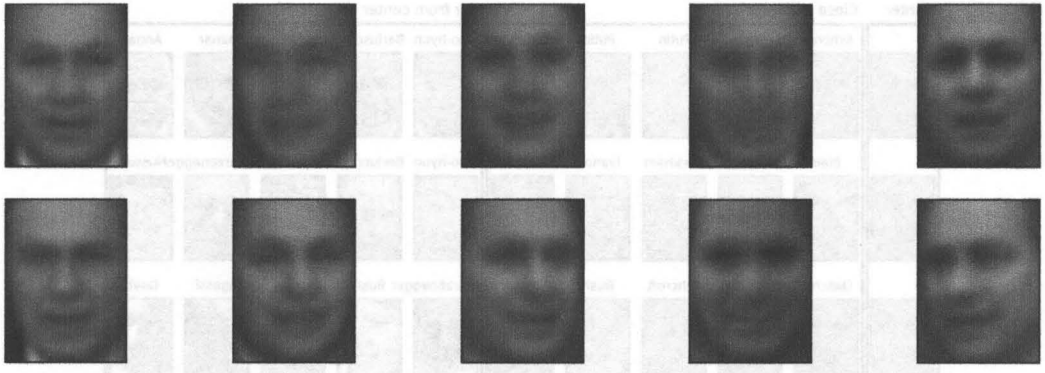


*Figure 3.43 : Centres des clusters trouvés par les k-moyennes en définissant un nombre de clusters égal à 10.*

Les centres des clusters trouvés par l'algorithme des *k*-moyennes sont des versions très « lissées » des visages. Ce n'est pas vraiment surprenant, car chaque centre est une moyenne de 133 à 353 images de visages. Travailler avec une représentation PCA réduite ajoute au lissage des images (par comparaison avec la reconstruction des images utilisant 100 composantes, comme nous l'avons vu avec la Figure 3.11). Le clustering semble avoir capturé différentes orientations des visages, différentes expressions, ainsi que la présence de cols de chemise (sur les deux clichés de gauche).

Pour avoir une vue plus détaillée, nous montrons sur la Figure 3.44 les cinq images les plus typiques pour chaque centre de cluster (autrement dit, celles qui sont classées comme étant les plus proches des centres de ces clusters), ainsi que les cinq images les plus atypiques (donc celles qui, dans chaque cluster, sont les plus éloignées du centre) :

**In[79]:**

```
mglearn.plots.plot_kmeans_faces(km, pca, X_pca, X_people,
                                y_people, people.target_names)
```

La Figure 3.44 confirme notre intuition quant à l'orientation des visages, ou encore la reconnaissance de cols de chemise. Par contre, les points « atypiques » ne ressemblent pas trop aux centres des clusters, et leur affectation semble quelque peu arbitraire. Cela peut être attribué au fait que l'algorithme des *k*-moyennes partitionne tous les points de données, et qu'il ne possède pas de concept de « bruit », à l'instar de DBSCAN. Avec un nombre plus élevé de clusters, l'algorithme pourrait trouver des distinctions plus fines. Cependant, cela rendrait également l'inspection manuelle encore plus difficile à effectuer.




*Figure 3.44 : Exemples d'images pour chaque cluster trouvé par les k-moyennes. Les centres des clusters se trouvent à gauche, suivis par les cinq points les plus proches de chaque centre et par les cinq points qui en sont les plus éloignés.*

#### Analyser le jeu de photos de visages avec le clustering agglomératif

Regardons maintenant les résultats que peut fournir le clustering agglomératif :

**In[80] :**

```
agglomerative = AgglomerativeClustering(n_clusters=10)
labels_agg = agglomerative.fit_predict(X_pca)
print("cluster sizes agglomerative clustering: {}".format(
      np.bincount(labels_agg)))
```

**Out[80] :**

```
cluster sizes agglomerative clustering: [169 660 144 329 217 85 18 261 31 149]
```

L'algorithme produit là encore des clusters d'une taille relativement équilibrée, avec tout de même un minimum de 18 et un maximum de 660 qui en agrandissent l'échelle. Le résultat est plus disparate qu'avec les *k*-moyennes, mais beaucoup moins qu'avec DBSCAN.

Nous pouvons calculer l'indice ARI afin de mesurer le niveau de similarité entre les deux partitions des données fournies par le clustering agglomératif et les *k*-moyennes :

**In[81] :**

```
print("ARI: {:.2f}".format(adjusted_rand_score(labels_agg, labels_km)))
```

**Out[81] :**

```
ARI: 0.07
```

Cette valeur ARI de 0.07 indique que les deux partitionnements `labels_agg` et `labels_km` ont très peu en commun. Ce n'est pas réellement surprenant étant donné que les points éloignés des centres des clusters semblent n'avoir que peu de rapport avec ceux-ci dans le cas des *k*-moyennes.

Voyons maintenant ce que donne le tracé du dendrogramme (voir la Figure 3.45). Nous allons limiter ici la profondeur de l'arbre, car descendre jusqu'aux 2 063 points de donnée individuels rendrait la figure tellement dense qu'elle serait illisible :

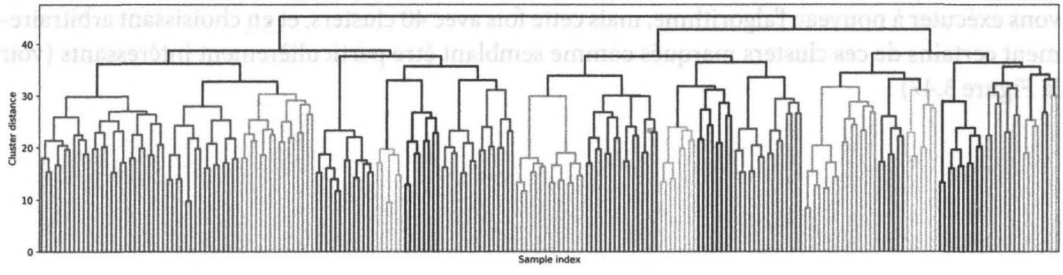



*Figure 3.45 : Dendrogramme du clustering agglomératif appliqué au jeu de données des visages.*

**In[82] :**

```
linkage_array = ward(X_pca)
plt.figure(figsize=(20, 5))
```

```
dendrogram(linkage_array, p=7, truncate_mode='level', no_labels=True)
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```

En créant dix clusters, nous coupons l'arbre en son sommet, là où se trouvent dix lignes verticales. Dans le dendrogramme pour le jeu de données jouet illustré sur la Figure 3.36, nous pouvions voir grâce à la longueur des branches que deux ou trois clusters étaient capables de capturer les données de manière appropriée. Dans le cas de nos visages, cette interprétation ne semble plus du tout ni évidente, ni naturelle. Certaines branches représentent des groupes plus distincts, mais il ne semble pas y avoir un nombre particulier de clusters qui soit bien « ajusté ». Ce n'est pas étonnant au vu des résultats de DBSCAN qui a essayé de regrouper tous les points.

Essayons maintenant de visualiser les dix clusters comme nous l'avons fait avec les *k*-moyennes (voir la Figure 3.46). Notez bien qu'il n'y a pas de notion de centre dans le clustering agglomératif (bien que nous pourrions calculer la moyenne). Nous montrons simplement quelques exemples d'images dans les clusters. Le nombre total de points dans chaque cluster est indiqué à gauche de la première image de la rangée correspondante :

**In[83]:**

```
n_clusters = 10
for cluster in range(n_clusters):
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 10, subplot_kw={'xticks': (), 'yticks': ()},
                             figsize=(15, 8))
    axes[0].set_ylabel(np.sum(mask))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1],
                     fontdict={'fontsize': 9})
```

Bien que certains clusters semblent proposer un thème sémantique, la plupart d'entre eux sont trop grands pour être réellement homogènes. Pour améliorer cet aspect des choses, nous pouvons exécuter à nouveau l'algorithme, mais cette fois avec 40 clusters, et en choisissant arbitrairement certains de ces clusters marqués comme semblant être particulièrement intéressants (voir la Figure 3.47) :




*Figure 3.46 : Images aléatoires provenant des clusters générés par le code In[83], chaque ligne correspondant à un cluster.*




*Figure 3.47 : Sélection d'images trouvées par le clustering agglomératif avec 40 clusters. Le texte de gauche montre l'indice du cluster ainsi que le nombre total de points qu'il contient.*

**In[84]:**

```
agglomerative = AgglomerativeClustering(n_clusters=40)
labels_agg = agglomerative.fit_predict(X_pca)
print("cluster sizes agglomerative clustering: {}".format(np.bincount(labels_
  agg)))

n_clusters = 40
for cluster in [16, 21, 23, 26, 39]: # hand-picked "interesting" clusters
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 15, subplot_kw={'xticks': (), 'yticks': ()},
                             figsize=(15, 8))
    cluster_size = np.sum(mask)
    axes[0].set_ylabel("#{}: {}".format(cluster, cluster_size))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1],
                     fontdict={'fontsize': 9})
    for i in range(cluster_size, 15):
        axes[i].set_visible(False)
```

**Out[84]:**

```
cluster sizes agglomerative clustering:
[ 43 120 100 194  56  58 127  22   6  37  65  49  84  18 168  44  47  31
  78  30 166  20  57  14  11  29  23   5   8  84  67  30  57  16  22  12
  29   2  26   8]
```

Ici, même avec certaines incohérences, le clustering semble s'être focalisé sur certains types de peau, certaines orientations de visage, d'yeux, ou encore certaines personnalités au visage très caractéristique… Pour autant, si nous voulions pousser l'analyse plus loin, nous pourrions aussi juger de la réalité de ces interprétations ainsi que d'autres similarités en traçant un dendrogramme.

### Résumons les méthodes de clustering

Cette section a montré qu'appliquer et évaluer des méthodes de clustering est une procédure extrêmement qualitative, et souvent surtout utile dans la phase exploratoire de l'analyse des données. Nous avons étudié trois algorithmes de clustering : *k*-moyennes, DBSCAN et le clustering agglomératif. Tous les trois ont leur propre manière de contrôler la granularité du clustering. Le clustering agglomératif et les *k*-moyennes vous demandent de préciser le nombre de clusters voulu, tandis que DBSCAN vous permet de spécifier une proximité en utilisant le paramètre `eps`, qui influe indirectement sur la taille des clusters. Les trois méthodes peuvent être utilisées sur de grands jeux de données du monde réel et sont relativement faciles à comprendre, tout en permettant de gérer de nombreux clusters.

Chaque algorithme a ses propres avantages (et donc aussi ses propres inconvénients). Par exemple, les *k*-moyennes permettent une caractérisation des clusters à partir de leurs moyennes. Cet algorithme peut également être perçu comme étant une méthode de décomposition, dans laquelle chaque point de donnée est représenté par son centre de cluster. DBSCAN permet de détecter des « points de bruit », qui ne sont affectés à aucun cluster, et il peut aussi aider à déterminer automatiquement le nombre voulu de clusters. Par contraste avec les deux autres méthodes, il autorise le traitement de formes de clusters complexes, comme nous l'avons vu dans l'exemple `two_moons`. DBSCAN produit également parfois des clusters de taille très différente, ce qui est selon le cas une qualité ou un défaut. Le clustering agglomératif peut fournir de son côté une hiérarchie des partitions possibles des données qu'il est possible d'explorer facilement *via* des dendrogrammes.

## Résumé et perspectives

Ce chapitre vous a présenté divers algorithmes d'apprentissage non supervisés qui peuvent être appliqués pour une analyse exploratoire et un prétraitement des données. Avoir la bonne représentation des données est souvent crucial pour un comportement efficace des techniques d'apprentissage automatique supervisées ou non supervisées, et les méthodes de prétraitement et de décomposition jouent un rôle important dans la préparation des données.

La décomposition, l'apprentissage des variétés ou encore le clustering sont des outils essentiels pour pousser plus loin la compréhension de vos données, et ces méthodes peuvent même

être les seules manières de donner du sens à celles-ci en l'absence d'informations supervisées. Et y compris dans un environnement supervisé, les outils d'exploration sont importants pour mieux comprendre les propriétés des données. Il est souvent difficile de quantifier l'utilité d'un algorithme non supervisé, mais cela ne devrait pas vous dissuader de faire appel à ce type de méthode pour recueillir des informations sur vos données. Avec ces outils dans votre besace, vous êtes maintenant équipé de tous les algorithmes essentiels dont les praticiens de l'apprentissage automatique se servent quotidiennement.

Nous vous encourageons à essayer des méthodes de clustering et de décomposition aussi bien sur des jeux de données jouet en deux dimensions que sur les jeux de données du monde réel inclus dans `scikit-learn`, comme `digits`, `iris` et `cancer`.

## Résumé de l'interface Estimator

Résumons rapidement l'API que nous avons étudiée dans les Chapitres 2 et 3. Tous les algorithmes de `scikit-learn`, qu'il s'agisse de prétraitement, d'apprentissage supervisé ou d'apprentissage non supervisé, sont implémentés sous forme de classes. Ces classes sont appelées des *estimators* (*estimateurs*) dans `scikit-learn`. Pour appliquer un algorithme, vous devez tout d'abord instancier un objet de la classe correspondante. Par exemple :

**In[85]:**

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
```

La classe de l'estimateur contient l'algorithme, et elle enregistre également le modèle qui est appris des données en utilisant cet algorithme.

Vous devriez spécifier tous les paramètres du modèle lorsque vous construisez l'objet de ce modèle. Ces paramètres peuvent être variés : régularisation, contrôle de la complexité, nombre de clusters, *etc.* Tous les estimateurs ont une méthode `fit`, qui sert à construire le modèle. La méthode `fit` nécessite toujours de spécifier comme premier argument les données X, présentées sous forme d'un tableau NumPy ou d'une matrice creuse SciPy, chaque ligne représentant un unique point de donnée. X est également toujours supposé être un tableau NumPy ou une matrice creuse SciPy dont les entrées sont continues (en virgule flottante). Les algorithmes supervisés prennent également un argument y, qui est un tableau NumPy avec une seule dimension, et qui contient des valeurs cibles pour la régression ou la classification (par exemple des étiquettes ou des réponses connues).

Il y a essentiellement deux manières d'appliquer un modèle entraîné dans `scikit-learn`. Pour créer une prédiction sous la forme d'une nouvelle sortie comme y, vous utilisez la méthode `predict`. Pour créer une nouvelle représentation des données d'entrée X, vous utilisez la méthode `transform`. Le Tableau 3.1 résume les situations dans lesquelles s'appliquent les méthodes `predict` et `transform`.

*Tableau 3.1 : Résumé de l'API de* **scikit-learn**.

| `estimator.fit(X_train, [y_train])` | |
|---|---|
| `estimator.predict(X_test)` | `estimator.transform(X_test)` |
| Classification | Prétraitement |
| Régression | Réduction de dimension |
| Clustering | Extraction de caractéristiques<br>Sélection de caractéristiques |

De plus, tous les modèles supervisés ont une méthode `score(X_test, y_test)` qui fournit une évaluation du modèle. Dans le Tableau 3.1, X_train et y_train font référence respectivement aux données et aux étiquettes d'apprentissage, tandis que `X_test` représente les données de test, et (si c'est applicable) les étiquettes de test.

# CHAPITRE 4

# Représentation des données et ingénierie des caractéristiques

Jusqu'ici, nous avons supposé que nos données se présentaient sous la forme d'un tableau en deux dimensions de nombres en virgule flottante, dans lequel chaque colonne est une *caractéristique continue* décrivant les points de données. Mais, pour de multiples applications, ce n'est pas ainsi que les données sont collectées. Certaines caractéristiques particulières sont dites *catégorielles*, ou encore *discrètes*. Il ne s'agit d'ailleurs généralement pas de valeurs numériques. La distinction entre caractéristiques catégorielles et caractéristiques continues est analogue à ce qui se passe pour la classification et la régression, du moins du côté des données d'entrée. Comme exemples de caractéristiques continues parmi ce que nous avons vu, nous pourrions citer l'intensité des pixels d'une image, ou encore la mesure de certaines dimensions dans des fleurs. Et voici quelques exemples de données catégorielles : la marque d'un produit, sa couleur, ou encore le rayon dans lequel il est vendu dans une boutique (librairie, vêtements, matériel de jardinage, et ainsi de suite). Toutes ces propriétés sont à même de décrire un produit, mais elles ne varient pas d'une façon continue. Ainsi, un certain produit peut se trouver dans le rayon « Librairie » *ou* dans le rayon « Vêtements ». Il n'y a (normalement) pas de point de rencontre entre les deux, ni d'ordre naturel pour les différentes catégories (un livre n'est pas en soi supérieur ou inférieur à un vêtement, l'équipement en matériel de jardinage ne se situe pas quelque part *entre* les livres et l'habillement, *etc.*).

Indépendamment du type des caractéristiques de vos données, la manière de les représenter peut avoir un impact énorme sur les performances des modèles d'apprentissage automatique. Nous avons vu dans les Chapitres 2 et 3 que le calibrage des données est important. En d'autres termes, si vous ne recalibrez pas correctement vos données (par exemple pour obtenir une variance unitaire), une certaine mesure fera une grande différence selon qu'elle est exprimée en centimètres ou bien en pouces. Nous avons aussi vu dans le Chapitre 2 qu'il peut être utile d'*aug-*

*menter* vos données en leur ajoutant des caractéristiques, comme des interactions (produits) ou des polynômes plus généraux.

La question consistant à déterminer comment représenter le mieux possible vos données pour une application particulière relève de ce que l'on appelle l'*ingénierie de caractéristiques* (*feature engineering*), et c'est l'une des tâches principales que doivent se fixer les *data scientists* et les praticiens de l'apprentissage automatique lorsqu'il s'agit de résoudre des problèmes du monde réel. Bien représenter les données peut avoir une influence plus grande sur les performances d'un modèle supervisé qu'un choix approprié de paramètres.

Dans ce chapitre, nous allons d'abord passer en revue le cas important et très courant des caractéristiques catégorielles, puis nous donnerons quelques exemples de transformations utiles pour des combinaisons spécifiques de caractéristiques et de modèles.

# Variables catégorielles

À titre d'exemple, nous allons utiliser le jeu de données appelé `adult`, et qui concerne les revenus de citoyens américains à partir des informations fournies par le recensement de 1994. L'objectif consiste à essayer de prédire si une personne qui travaille a un revenu annuel qui se situe au-dessus ou en dessous d'une barrière fixée à 50 000 $. Les caractéristiques de ce jeu de données comprennent l'âge des personnes, leur statut (artisans ou commerçants, employés de l'industrie privée ou du secteur public, *etc.*), leur niveau de formation, leur sexe, le nombre d'heures travaillées par semaine, la nature de leur poste de travail, et ainsi de suite. Le Tableau 4.1 montre les premières entrées de ce jeu de données.

*Tableau 4.1 : Les premières entrées du jeu de données* `adult`.

| | age | workclass | education | gender | hours-per-week | occupation | income |
|---|---|---|---|---|---|---|---|
| 0 | 39 | State-gov | Bachelors | Male | 40 | Adm-clerical | <=50K |
| 1 | 50 | Self-emp-not-inc | Bachelors | Male | 13 | Exec-managerial | <=50K |
| 2 | 38 | Private | HS-grad | Male | 40 | Handlers-cleaners | <=50K |
| 3 | 53 | Private | 11th | Male | 40 | Handlers-cleaners | <=50K |
| 4 | 28 | Private | Bachelors | Female | 40 | Prof-specialty | <=50K |
| 5 | 37 | Private | Masters | Female | 40 | Exec-managerial | <=50K |
| 6 | 49 | Private | 9th | Female | 16 | Other-service | <=50K |
| 7 | 52 | Self-emp-not-inc | HS-grad | Male | 45 | Exec-managerial | >50K |
| 8 | 31 | Private | Masters | Female | 50 | Prof-specialty | >50K |
| 9 | 42 | Private | Bachelors | Male | 40 | Exec-managerial | >50K |
| 10 | 37 | Private | Some-college | Male | 80 | Exec-managerial | >50K |

La tâche se définit comme étant une classification, avec deux classes correspondant au seuil des 50 000 $ de revenus (*income*). Il serait également possible de prédire un revenu exact, et donc d'en faire une tâche de régression. Mais le problème deviendrait alors beaucoup plus difficile, et la division proposée est déjà intéressante à comprendre en soi.

Dans ce jeu de données, les colonnes `age` et `hours-per-week` sont des caractéristiques continues que nous savons comment traiter. Par contre, `workclass`, `education`, `gender` et `occupation` sont des données catégorielles. Elles prennent toutes leur contenu dans une liste fixée de valeurs possibles, à la différence d'une plage ou d'un intervalle de valeurs, et elles dénotent une propriété *qualitative*, par opposition à une propriété *quantitative*.

Comme point de départ, disons que nous voulons entraîner comme classifieur une régression logistique sur ces données. Nous avons vu dans le Chapitre 2 qu'une régression logistique effectue des prédictions, ŷ, en utilisant la formule suivante :

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b > 0$$

Ici, *w[i]* et *b* sont les coefficients entraînés à partir du jeu d'apprentissage, et *x[i]* sont les caractéristiques récupérées en entrée. Cette formule fait sens lorsque les valeurs *x[i]* sont des nombres, mais pas si *x[2]*, par exemple, vaut `"Masters"` ou `"Bachelors"`. Clairement, nous devons donc représenter nos données d'une autre manière pour pouvoir appliquer une régression logistique. La section qui suit va expliquer comment surmonter ce problème.

## Encodage one-hot (variables indicatrices)

La manière de loin la plus courante pour représenter des variables catégorielles consiste à utiliser l'*encodage one-hot* (ou encore *encodage one-out-of-N*), ce que l'on appelle aussi des *variables indicatrices* (*dummy variables* en anglais). L'idée sous-jacente est de remplacer une variable catégorielle par une ou plusieurs nouvelles caractéristiques qui peuvent avoir comme valeur 0 et 1. Dans ce cas, ces chiffres font sens pour la formule ci-dessus (comme d'ailleurs pour tous les autres modèles de `scikit-learn`), et nous pouvons représenter un nombre quelconque de catégories en introduisant une nouvelle caractéristique par catégorie, comme nous allons le voir ici.

Supposons que, pour la caractéristique `workclass`, nous ayons quatre entrées possibles : `"Government Employee"`, `"Private Employee"`, `"Self Employed"`, et `"Self Employed Incorporated"`. Pour encoder ces valeurs possibles, nous créons autant de nouvelles caractéristiques, également appelées `"Government Employee"`, `"Private Employee"`, `"Self Employed"` et `"Self Employed Incorporated"`. Une de ces nouvelles caractéristiques vaut 1 si la personne occupe le type d'emploi correspondant, et 0 dans le cas contraire. Pour chaque point de donnée, une et une seule de ces caractéristiques vaudra donc 1, et les trois autres 0.

Ce principe est illustré sur le Tableau 4.2. On y voit bien qu'une seule caractéristique est encodée à l'aide de quatre nouvelles caractéristiques. Dans un algorithme d'apprentissage automatique,

il suffit ensuite d'éliminer la colonne `workclass` d'origine, et de garder uniquement les caractéristiques binaires.

*Tableau 4.2 : Transformer la caractéristique* `workclass` *en utilisant un encodage one-hot.*

| workclass | Government Employee | Private Employee | Self Employed | Self Employed Incorporated |
|---|---|---|---|---|
| Government Employee | 1 | 0 | 0 | 0 |
| Private Employee | 0 | 1 | 0 | 0 |
| Self Employed | 0 | 0 | 1 | 0 |
| Self Employed Incorporated | 0 | 0 | 0 | 1 |



L'encodage one-hot que nous utilisons est assez semblable, mais pas identique, au codage « dummy » utilisé en statistiques. Pour simplifier l'analyse (plus techniquement, pour éviter un problème de déficience de rang dans la matrice des données), nous encodons chaque catégorie avec une caractéristique binaire différente. En statistiques, il est courant par contre d'encoder une caractéristique catégorielle avec *k* valeurs différentes possibles dans *k*-1 caractéristiques (la dernière étant représentée uniquement par des zéros).

Il y a deux façons de convertir vos données en un encodage one-hot des variables catégorielles : en utilisant soit `pandas` soit `scikit-learn`. Au moment où ce livre est écrit, le chemin est un peu plus facile avec `pandas`, et c'est donc lui que nous allons suivre. Nous allons commencer par charger les données à partir d'un fichier au format CSV (provenant du package `mglearn`) en nous servant de `pandas` :

**In[1]:**

```
from IPython.display import display
import os
adult_path = os.path.join(mglearn.datasets.DATA_PATH, "adult.data")
data = pd.read_csv(
    adult_path, header=None, index_col=False,
    names=['age', 'workclass', 'fnlwgt', 'education', 'education-num',
           'marital-status', 'occupation', 'relationship', 'race', 'gender',
           'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',
           'income'])
data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week',
             'occupation', 'income']]
display(data.head())
```

Le Tableau 4.3 montre les résultats obtenus.

*Tableau 4.3 : Les cinq premières lignes du jeu de données* `adult`.

| | age | workclass | education | gender | hours-per-week | occupation | income |
|---|---|---|---|---|---|---|---|
| 0 | 39 | State-gov | Bachelors | Male | 40 | Adm-clerical | <=50K |

| | age | workclass | education | gender | hours-per-week | occupation | income |
|---|---|---|---|---|---|---|---|
| 1 | 50 | Self-emp-not-inc | Bachelors | Male | 13 | Exec-managerial | <=50K |
| 2 | 38 | Private | HS-grad | Male | 40 | Handlers-cleaners | <=50K |
| 3 | 53 | Private | 11th | Male | 40 | Handlers-cleaners | <=50K |
| 4 | 28 | Private | Bachelors | Female | 40 | Prof-specialty | <=50K |

### Vérifier les chaînes de caractères dans les données catégorielles

Une fois que vous avez lu un jeu de données tel que celui-ci, il est souvent bon de vérifier tout de suite si une colonne contient des données catégorielles significatives. Si vous travaillez avec des données qui ont été saisies par des humains (comme des utilisateurs sur un site Web), il peut ne pas y avoir un nombre bien fixé de catégories. Ainsi, des différences dans l'orthographe et la capitalisation peuvent nécessiter un prétraitement des données. Par exemple, il se pourrait que certaines personnes aient renseigné le champ Sexe sous la forme « mâle » ou « homme », ou plus simplement « M » ou « H », et nous devons prendre en compte ces variations pour les replacer dans la même catégorie. Une bonne manière de s'y prendre pour contrôler le contenu d'une colonne consiste à faire appel à la méthode `value_counts` d'un tableau `Series` de `pandas` (le type d'une seule colonne dans un `DataFrame`). Cela nous permet de voir quelles sont les valeurs uniques et combien de fois elles apparaissent :

**In[2]:**

```
print(data.gender.value_counts())
```

**Out[2]:**

```
 Male      21790
 Female    10771
Name: gender, dtype: int64
```

Nous pouvons constater qu'il y a exactement deux valeurs pour le sexe, `Male` et `Female`, ce qui signifie que cette donnée est déjà dans un bon format pour être représentée *via* l'encodage one-hot. Dans une application réelle, vous devriez analyser le contenu de toutes les colonnes afin de vérifier leurs valeurs. Nous allons laisser cette question de côté pour éviter de surcharger le livre.

Il y a un moyen très simple d'encoder les données dans `pandas` grâce à la fonction `get_dummies`. Elle transforme automatiquement toutes les colonnes qui ont un certain type d'objet (comme des chaînes de caractères) ou qui sont catégorielles (un concept `pandas` qui n'a pas encore été abordé) :

**In[3]:**

```
print("Original features:\n", list(data.columns), "\n")
data_dummies = pd.get_dummies(data)
print("Features after get_dummies:\n", list(data_dummies.columns))
```

**Out[3]:**

```
Original features:
 ['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation',
  'income']

Features after get_dummies:
 ['age', 'hours-per-week', 'workclass_ ?', 'workclass_ Federal-gov',
  'workclass_ Local-gov', 'workclass_ Never-worked', 'workclass_ Private',
  'workclass_ Self-emp-inc', 'workclass_ Self-emp-not-inc', 'workclass_
  State-gov', 'workclass_ Without-pay', 'education_ 10th', 'education_ 11th',
  'education_ 12th', 'education_ 1st-4th', 'education_ 5th-6th', 'education_
  7th-8th', 'education_ 9th', 'education_ Assoc-acdm', 'education_ Assoc-
  voc', 'education_ Bachelors', 'education_ Doctorate', 'education_ HS-grad',
  'education_ Masters', 'education_ Preschool', 'education_ Prof-school',
  'education_ Some-college', 'gender_ Female', 'gender_ Male', 'occupation_
  ?', 'occupation_ Adm-clerical', 'occupation_ Armed-Forces', 'occupation_
  Craft-repair', 'occupation_ Exec-managerial', 'occupation_ Farming-
  fishing', 'occupation_ Handlers-cleaners', 'occupation_ Machine-op-inspct',
  'occupation_ Other-service', 'occupation_ Priv-house-serv', 'occupation_
  Prof-specialty', 'occupation_ Protective-serv', 'occupation_ Sales',
  'occupation_ Tech-support', 'occupation_ Transport-moving', 'income_ <=50K',
  'income_ >50K']
```

Vous pouvez constater que les caractéristiques continues `age` et `hours-per-week` n'ont pas été touchées, tandis que les caractéristiques catégorielles ont été développées afin de créer une nouvelle caractéristique pour chaque valeur possible :

**In[4]:**

```
display(data_dummies.head(n=5))
```

**Out[4]:**

| | age | hours-per-week | workclass ? | workclass Federal-gov | workclass Local-gov | ... | occupation Tech-support | occupation Transport-moving | income <=50K | income >50K |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 39 | 40 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |
| 1 | 50 | 13 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |
| 2 | 38 | 40 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |
| 3 | 53 | 40 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |
| 4 | 28 | 40 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |

Nous pouvons maintenant utiliser l'attribut `values` pour convertir le `DataFrame` `data_dummies` en un tableau NumPy, puis entraîner sur lui un modèle d'apprentissage automatique. Faites bien attention à séparer la variable cible (qui est maintenant encodée dans deux colonnes `income`) des autres données avant d'entraîner un modèle. Inclure la variable de sortie, ou une propriété dérivée de celle-ci, dans la représentation des caractéristiques est une erreur très courante lors de la construction de modèles d'apprentissage automatique supervisés.



L'indexation des colonnes dans `pandas` inclut la fin de l'intervalle. Autrement dit, l'intervalle `'age':'occupation_ Transport-moving'` comprend `occupation_ Transport-moving`. Cela est différent de la manière dont fonctionnent les tableaux NumPy. Par exemple, `np.arange(11)[0:10]` n'inclut pas l'entrée d'indice 10.

Dans ce cas, nous extrayons uniquement les colonnes qui contiennent des caractéristiques, c'est-à-dire en allant de `age` jusqu'à `occupation_ Transport-moving`. Cette plage contient toutes les caractéristiques, mais pas la cible :

**In[5]:**

```
features = data_dummies.loc[:, 'age':'occupation_ Transport-moving']
X = features.values
y = data_dummies['income_ >50K'].values
print("X.shape: {}  y.shape: {}".format(X.shape, y.shape))
```

**Out[5]:**

```
X.shape: (32561, 44)  y.shape: (32561,)
```

Les données sont maintenant représentées sous une forme avec laquelle `scikit-learn` peut travailler, et nous pouvons donc procéder comme d'habitude :

**In[6]:**

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Test score: {:.2f}".format(logreg.score(X_test, y_test)))
```

**Out[6]:**

```
Test score: 0.81
```



Dans cet exemple, nous avons appelé `get_dummies` sur un `DataFrame` contenant les données d'apprentissage, mais aussi de test. Cela est important pour s'assurer que les valeurs catégorielles sont représentées de la même manière dans ces deux jeux de données.

Imaginez que les jeux d'apprentissage et de test soient dans deux `DataFrame` différents. Si, par exemple, la valeur `"Private Employee"` pour la caractéristique `workclass` n'apparaît pas dans le jeu de test, `pandas` supposera qu'il n'existe que trois possibilités pour cette caractéristique, et il ne créera donc que trois nouvelles variables indicatrices. Maintenant, nos jeux d'apprentissage et de test ont un nombre différent de caractéristiques, et nous ne pouvons donc plus appliquer au jeu de test le modèle qui a été entraîné sur le jeu d'apprentissage. Pire même, imaginez que la caractéristique `workclass` possède les valeurs `"Government Employee"` et `"Private Employee"` dans le jeu d'apprentissage, et `"Self Employed"` ainsi que `"Self Employed Incorporated"` dans le jeu de test. Dans ce cas, `pandas` va créer

deux nouvelles variables indicatrices de chaque côté, et donc les `DataFrame` encodés auront le même nombre de caractéristiques. Par contre, ces indicatrices auront des significations totalement différentes dans les jeux d'apprentissage et de test. La colonne qui indiquerait `"Government Employee"` pour le jeu d'apprentissage encoderait la valeur `"Self Employed"` dans le jeu de test.

Si nous construisions un modèle d'apprentissage automatique à partir de ces données, il fonctionnerait très mal puisqu'il supposerait que les colonnes auraient la même signification (du fait qu'elles se trouvent à la même position), alors que ce n'est pas du tout le cas. Pour réparer cela, il faut soit appeler `get_dummies` sur un `DataFrame` qui contienne à la fois les points de données du jeu d'apprentissage et ceux du jeu de test, soit s'assurer que les noms des colonnes sont les mêmes dans les deux jeux après avoir appelé `get_dummies` afin de vérifier que les sémantiques en sont identiques.

## Les nombres peuvent encoder des catégories

Dans l'exemple du jeu de données `adult`, les variables catégorielles ont été encodées sous forme de chaînes de caractères. D'un côté, cela ouvre la possibilité de rencontrer des fautes d'orthographe ou de saisie ; mais, d'un autre côté, cela marque clairement le fait qu'une variable est catégorielle. Souvent, soit pour des raisons de facilité d'enregistrement, ou bien parce que les données ont été collectées ainsi, les variables catégorielles sont encodées sous la forme d'entiers. Supposons par exemple que les informations de recensement du jeu de données `adult` aient été collectées sur la base d'un questionnaire, et que les réponses pour `workclass` soient indiquées en cochant une case (codée 0 pour la première réponse, 1 pour la seconde, et ainsi de suite). Maintenant, la colonne va donc contenir des chiffres allant de 0 à 8 au lieu de chaînes telles que `"Private"`, et il ne sera pas évident du premier coup d'œil pour une personne qui regarderait le tableau représentant le jeu de données de savoir si cette variable doit être considérée comme étant continue ou catégorielle. Savoir qu'un chiffre correspond à une certaine catégorie d'emploi, cependant, permet de comprendre qu'il s'agit d'états très distincts qui ne peuvent donc pas être modélisés à l'aide d'une seule variable continue.



Les caractéristiques catégorielles sont souvent encodées en utilisant des entiers. Le fait qu'il s'agisse de nombres ne signifie pas qu'elles doivent pour autant être traitées comme des caractéristiques continues. La distinction entre variable continue et variable discrète (ou résultant d'un encodage one-hot) n'est pas toujours évidente. S'il n'y a pas d'ordre particulier entre les sémantiques qui sont encodées (comme dans l'exemple de `workclass`), la caractéristique doit être traitée comme étant discrète. Dans d'autres situations, par exemple des appréciations par étoiles, le meilleur encodage dépend à la fois des données, de la tâche particulière à réaliser et de l'algorithme d'apprentissage automatique qui est utilisé.

La fonction `get_dummies` de `pandas` traite tous les nombres comme étant continus, et ne crée donc pas d'indicatrices pour eux. Pour contourner le problème, vous pouvez soit utiliser la

méthode `OneHotEncoder` de `scikit-learn`, qui vous permet de spécifier quelles variables sont continues et lesquelles sont discrètes, soit convertir les colonnes numériques du `DataFrame` en chaînes. Pour illustrer cela, créons un objet `DataFrame` avec deux colonnes, une contenant des chaînes et l'autre des entiers :

**In[7]:**

```
demo_df = pd.DataFrame({'Integer Feature': [0, 1, 2, 1],
                        'Categorical Feature': ['socks', 'fox', 'socks',
  'box']})
display(demo_df)
```

Le Tableau 4.4 illustre le résultat.

*Tableau 4.4 : DataFrame contenant une caractéristique chaîne et une autre formée d'entiers.*

| | Categorical Feature | Integer Feature |
|---|---|---|
| 0 | socks | 0 |
| 1 | fox | 1 |
| 2 | socks | 2 |
| 3 | box | 1 |

Utiliser `get_dummies` va uniquement encoder la caractéristique chaîne, et laisser de côté les nombres, ce qu'illustre le Tableau 4.5 :

**In[8]:**

```
display(pd.get_dummies(demo_df))
```

*Tableau 4.5 : Encodage one-hot des données du Tableau 4.4, laissant les entiers inchangés.*

| | Integer Feature | Categorical Feature_box | Categorical Feature_fox | Categorical Feature_socks |
|---|---|---|---|---|
| 0 | 0 | 0.0 | 0.0 | 1.0 |
| 1 | 1 | 0.0 | 1.0 | 0.0 |
| 2 | 2 | 0.0 | 0.0 | 1.0 |
| 3 | 1 | 1.0 | 0.0 | 0.0 |

Si vous voulez demander à ce que soient créées des variables indicatrices pour la colonne remplie d'entiers, vous pouvez la lister explicitement en utilisant le paramètre `columns`. Dans ce cas, les deux caractéristiques seront traitées comme étant catégorielles (voir le Tableau 4.6) :

**In[9]:**

```
demo_df['Integer Feature'] = demo_df['Integer Feature'].astype(str)
display(pd.get_dummies(demo_df, columns=['Integer Feature', 'Categorical
  Feature']))
```

*Tableau 4.6 : Encodage one-hot des deux colonnes de données du Tableau 4.4.*

| | Integer Feature_0 | Integer Feature_1 | Integer Feature_2 | Categorical Feature_box | Categorical Feature_fox | Categorical Feature_socks |
|---|---|---|---|---|---|---|
| 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| 1 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 2 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 |
| 3 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 |

# Binning, discrétisation, modèles linéaires et arbres

La meilleure façon de représenter des données dépend non seulement de leur sémantique, mais aussi du type de modèle que vous utilisez. Les modèles linéaires et ceux qui sont basés sur des arbres (arbres de décision, gradient boosting ou encore forêts aléatoires), deux familles vastes et couramment employées, ont des propriétés très différentes lorsque l'on s'intéresse de près à la manière dont elles fonctionnent avec des représentations également différentes des caractéristiques. Reprenons l'exemple du jeu de données wave que nous avons déjà utilisé dans le Chapitre 2. Il ne contient qu'une seule caractéristique d'entrée. Voici une comparaison entre un modèle de régression linéaire et une tâche de régression avec un arbre de décision (voir la Figure 4.1) :

**In[10]:**

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

X, y = mglearn.datasets.make_wave(n_samples=100)
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)

reg = DecisionTreeRegressor(min_samples_split=3).fit(X, y)
plt.plot(line, reg.predict(line), label="decision tree")

reg = LinearRegression().fit(X, y)
plt.plot(line, reg.predict(line), label="linear regression")

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

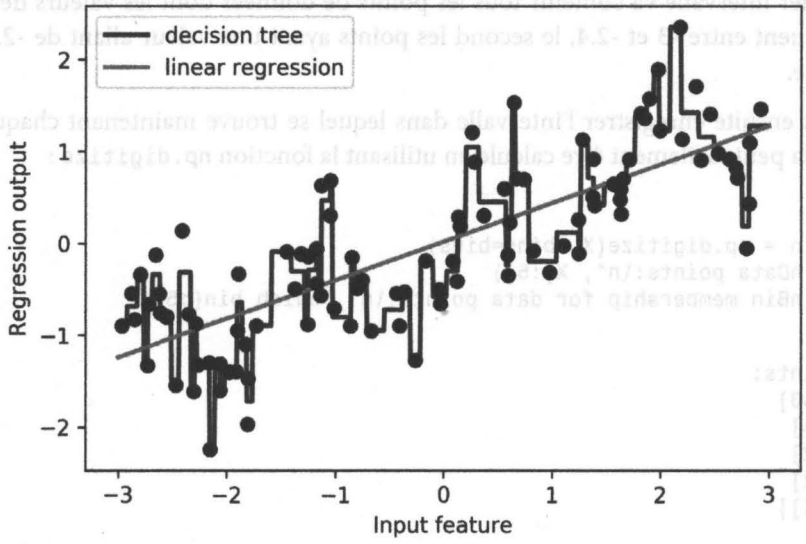



*Figure 4.1 : Comparaison entre régression linéaire et arbre de décision pour le jeu de données* wave.

Comme vous le savez, les modèles linéaires ne peuvent modéliser que des relations linéaires qui, dans le cas d'une caractéristique unique, sont des lignes. L'arbre de décision peut construire un modèle bien plus complexe. Cependant, cela dépend fortement de la représentation des données. Une manière de rendre les modèles linéaires plus puissants avec des données continues consiste à utiliser la *discrétisation* (ou encore *binning*) de la caractéristique, autrement dit son partitionnement en un certain nombre de nouvelles caractéristiques, non continues, comme nous allons le voir dans ce qui suit.

Nous imaginons une partition de la plage d'entrée de la caractéristique (dans ce cas, les nombres allant de -3 à 3) en un certain nombre d'intervalles de même taille (appelés *bins*), disons 10. Un point de donnée sera alors représenté par la classe, ou la « boîte », dans laquelle il a été placé. Avant cela, nous devons définir ces classes. En l'occurrence, nous allons effectuer une division en dix intervalles également espacés entre -3 et 3. Pour cela, nous allons faire appel à la fonction `np.linespace`, qui va créer 11 entrées, soit 10 intervalles successifs :

**In[11]:**

```
bins = np.linspace(-3, 3, 11)
print("bins: {}".format(bins))
```

**Out[11]:**

```
bins: [-3.  -2.4 -1.8 -1.2 -0.6  0.   0.6  1.2  1.8  2.4  3. ]
```

Ici, le premier intervalle va contenir tous les points de données dont les valeurs de caractéristiques se situent entre -3 et -2.4, le second les points ayant une valeur allant de -2.4 à -1.8, et ainsi de suite.

Nous allons ensuite enregistrer l'intervalle dans lequel se trouve maintenant chaque point de donnée. Cela peut facilement être calculé en utilisant la fonction `np.digitize` :

**In[12]:**

```
which_bin = np.digitize(X, bins=bins)
print("\nData points:\n", X[:5])
print("\nBin membership for data points:\n", which_bin[:5])
```

**Out[12]:**

```
Data points:
 [[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]

Bin membership for data points:
 [[ 4]
 [10]
 [ 8]
 [ 6]
 [ 2]]
```

Ce que nous avons fait ici, c'est de transformer une unique caractéristique d'entrée du jeu de données `wave`, de type continu, en une caractéristique catégorielle, discrète, qui encode les *bins*, donc les intervalles dans lesquels est placé chacun des points. Pour utiliser un modèle `scikit-learn` sur ces données, nous transformons maintenant cette caractéristique discrète en un encodage one-hot à l'aide de la méthode `OneHotEncoder` du module `preprocessing`. Cette méthode fait en réalité le même encodage que `pandas.get_dummies`, si ce n'est qu'elle ne fonctionne à l'heure actuelle que sur des variables catégorielles qui sont des entiers :

**In[13]:**

```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse=False)
encoder.fit(which_bin)
X_binned = encoder.transform(which_bin)
print(X_binned[:5])
```

**Out[13]:**

```
[[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

Du fait que nous avons défini ici dix intervalles, le jeu de données transformé `X_binned` est maintenant constitué de dix caractéristiques :

**In[14] :**

```
print("X_binned.shape: {}".format(X_binned.shape))
```

**Out[14] :**

```
X_binned.shape: (100, 10)
```

Nous allons maintenant construire un nouveau modèle de régression linéaire et un nouveau modèle d'arbre de décision sur les données avec l'encodage one-hot. Le résultat est visualisé sur la Figure 4.2, ainsi que les limites des intervalles (les lignes verticales) :

**In[15] :**

```
line_binned = encoder.transform(np.digitize(line, bins=bins))

reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='linear regression binned')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='decision tree binned')
plt.plot(X[:, 0], y, 'o', c='k')
plt.vlines(bins, -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")
plt.ylabel("Regression output")
plt.xlabel("Input feature")
```

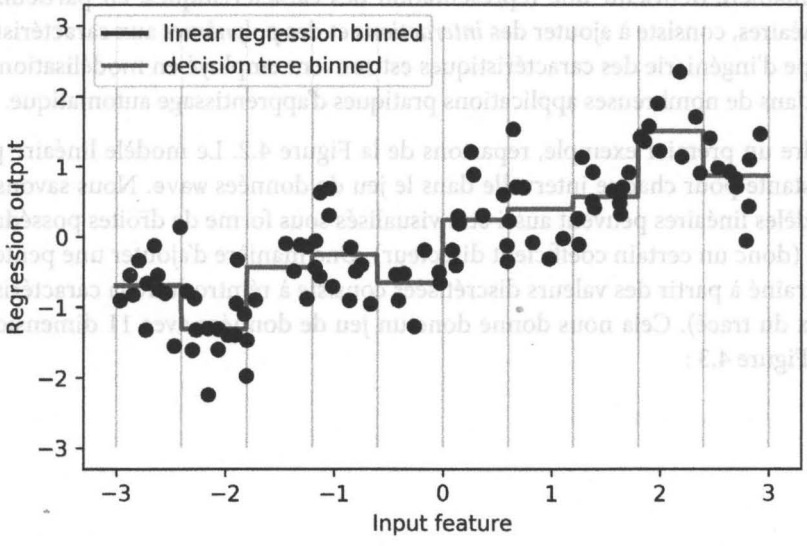



*Figure 4.2 : Comparaison entre régression linéaire et arbre de décision pour le jeu de données transformé.*

La ligne solide et la ligne tiretée se superposent parfaitement, ce qui signifie que le modèle de régression linéaire et l'arbre de décision font exactement les mêmes prédictions. Pour chaque *bin*, ils prédisent une valeur constante. Du fait que les caractéristiques sont constantes dans chaque intervalle, tous les modèles doivent prédire la même valeur pour tous les points qui se trouvent à l'intérieur d'un même *bin*. En comparant les modèles avant discrétisation et après, nous voyons que le modèle linéaire devient bien plus flexible, car il a maintenant une valeur différente pour chaque « bin », tandis qu'à l'inverse l'arbre de décision perd nettement en souplesse. La discrétisation n'apporte généralement pas de bénéfices particuliers aux modèles arborescents, car ceux-ci n'en ont pas besoin pour apprendre à diviser les données. En un sens, cela signifie que les arbres de décision peuvent apprendre quelle stratégie de *binning* (ou de *regroupement*) est la plus utile pour effectuer des prédictions sur ces données. De surcroît, ils sont capables de traiter de multiples caractéristiques en même temps, alors que la discrétisation est généralement appliquée sur une base individuelle. Cependant, le modèle linéaire gagne nettement en qualité avec cette transformation des données.

S'il y a de bonnes raisons pour utiliser un modèle linéaire sur un jeu de données particulier (par exemple parce qu'il est très gros et avec beaucoup de dimensions, mais avec certaines caractéristiques ayant des relations non linéaires avec la sortie), alors la discrétisation peut être un excellent moyen pour accroître la puissance du modèle.

## Interactions et polynômes

Une autre manière d'enrichir une représentation des caractéristiques, en particulier pour les modèles linéaires, consiste à ajouter des *interactions* et des *polynômes* aux caractéristiques d'origine. Ce type d'ingénierie des caractéristiques est souvent employé en modélisation statistique, mais aussi dans de nombreuses applications pratiques d'apprentissage automatique.

Pour prendre un premier exemple, repartons de la Figure 4.2. Le modèle linéaire produit une valeur constante pour chaque intervalle dans le jeu de données `wave`. Nous savons cependant que les modèles linéaires peuvent aussi être visualisés sous forme de droites possédant une certaine pente (donc un certain coefficient directeur). Une manière d'ajouter une pente au modèle linéaire entraîné à partir des valeurs discrétisées consiste à réintroduire la caractéristique d'origine (l'axe x du tracé). Cela nous donne donc un jeu de données avec 11 dimensions, comme l'illustre la Figure 4.3 :

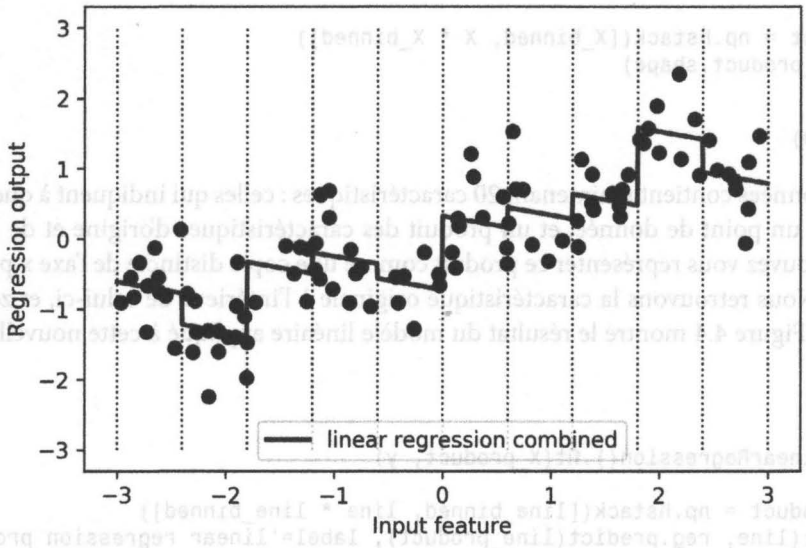



*Figure 4.3 : Régression linéaire utilisant les caractéristiques discrétisées et une seule pente globale.*

**In[16]:**

```
X_combined = np.hstack([X, X_binned])
print(X_combined.shape)
```

**Out[16]:**

```
(100, 11)
```

**In[17]:**

```
reg = LinearRegression().fit(X_combined, y)

line_combined = np.hstack([line, line_binned])
plt.plot(line, reg.predict(line_combined), label='linear regression combined')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k', linewidth=1)
plt.legend(loc="best")
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.plot(X[:, 0], y, 'o', c='k')
```

Dans cet exemple, la pente produite par le modèle est descendante et partagée entre tous les intervalles – nous avons ici une seule caractéristique pour l'axe x, et donc une seule valeur pour la pente. De ce fait, elle ne semble pas très utile en l'état. Il serait préférable de disposer d'une pente adaptée à chaque intervalle. Nous pouvons atteindre cet objectif en ajoutant une interaction, ou un produit, indiquant dans quel intervalle se trouve un point de donnée *et* où il se situe sur l'axe x. Cette caractéristique est un produit de l'indicateur *bin* et de la caractéristique originale. Créons ce jeu de données :

**In[18]:**

```
X_product = np.hstack([X_binned, X * X_binned])
print(X_product.shape)
```

**Out[18]:**

```
(100, 20)
```

Le jeu de données contient maintenant 20 caractéristiques : celles qui indiquent à quel intervalle correspond un point de donnée, et un produit des caractéristiques d'origine et de l'indicateur *bin*. Vous pouvez vous représenter ce produit comme une copie distincte de l'axe x pour chaque intervalle. Nous retrouvons la caractéristique originale à l'intérieur de celui-ci, et zéro partout ailleurs. La Figure 4.4 montre le résultat du modèle linéaire appliqué à cette nouvelle représentation :

**In[19]:**

```
reg = LinearRegression().fit(X_product, y)

line_product = np.hstack([line_binned, line * line_binned])
plt.plot(line, reg.predict(line_product), label='linear regression product')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k', linewidth=1)

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

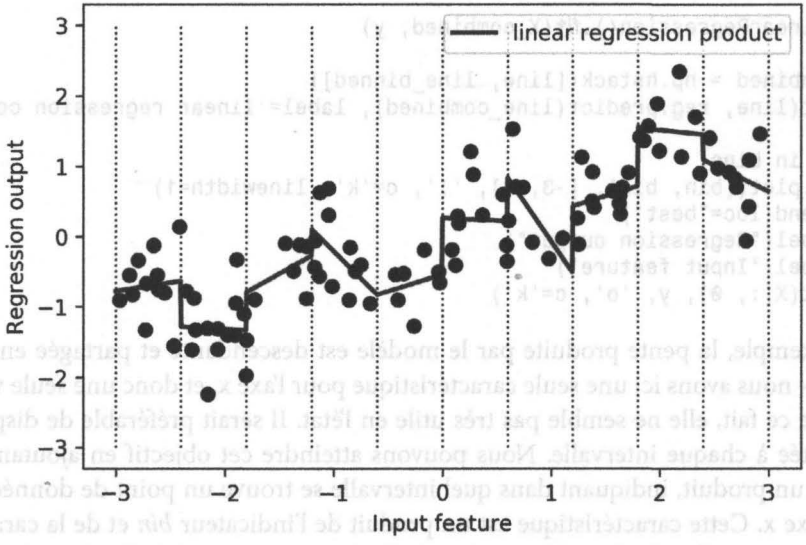



*Figure 4.4 : Régression linéaire avec une pente distincte pour chaque intervalle.*

Comme vous pouvez le constater, chaque intervalle a maintenant son propre décalage et sa propre pente dans ce modèle.

Une autre manière d'étendre une caractéristique continue consiste à utiliser des *polynômes*. Pour une caractéristique donnée `x`, nous pourrions vouloir considérer `x ** 2`, `x ** 3`, `x ** 4` et ainsi de suite. Cela est implémenté dans la classe `PolynomialFeatures` du module `preprocessing` de `scikit-learn` :

**In[20]:**

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=10, include_bias=False)
poly.fit(X)
X_poly = poly.transform(X)
```

Utiliser un degré de 10 produit 10 caractéristiques :

**In[21]:**

```
print("X_poly.shape: {}".format(X_poly.shape))
```

**Out[21]:**

```
X_poly.shape: (100, 10)
```

Comparons les entrées de `X_poly` à celles de `X` :

**In[22]:**

```
print("Entries of X:\n{}".format(X[:5]))
print("Entries of X_poly:\n{}".format(X_poly[:5]))
```

**Out[22]:**

```
Entries of X:
[[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]
Entries of X_poly:
[[    -0.753      0.567     -0.427      0.321     -0.242      0.182
      -0.137      0.103     -0.078      0.058]
 [     2.704      7.313     19.777     53.482    144.632    391.125
    1057.714   2860.36    7735.232  20918.278]
 [     1.392      1.938      2.697      3.754      5.226      7.274
      10.125     14.094     19.618     27.307]
 [     0.592      0.35       0.207      0.123      0.073      0.043
       0.025      0.015      0.009      0.005]
 [    -2.064      4.26      -8.791     18.144    -37.448     77.289
    -159.516    329.222   -679.478   1402.367]]
```

Vous pouvez afficher l'exposant correspondant à chaque caractéristique en appelant la méthode `get_feature_name` :

In[23]:

```
print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

Out[23]:

```
Polynomial feature names:
['x0', 'x0^2', 'x0^3', 'x0^4', 'x0^5', 'x0^6', 'x0^7', 'x0^8', 'x0^9', 'x0^10']
```

Vous pouvez constater que la première colonne de `X_poly` correspond exactement à X, tandis que les autres colonnes sont des puissances de la première entrée. Il est intéressant de voir quelle est la taille que peuvent atteindre certaines valeurs. La seconde ligne a des entrées au-dessus de 20 000, soit des ordres de magnitude différents du reste.

Utiliser des caractéristiques polynomiales en même temps qu'un modèle de régression linéaire conduit au modèle classique de *régression polynomiale* (voir la Figure 4.5) :

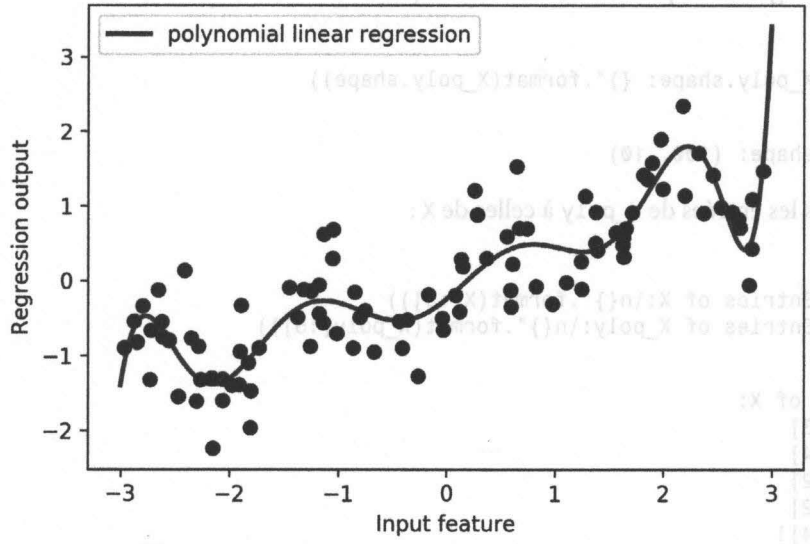



*Figure 4.5 : Régression linéaire avec des caractéristiques polynomiales d'ordre 10.*

In[24]:

```
reg = LinearRegression().fit(X_poly, y)

line_poly = poly.transform(line)
plt.plot(line, reg.predict(line_poly), label='polynomial linear regression')
plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

Comme vous pouvez le voir, nous obtenons ici un tracé très lissé sur ces données unidimensionnelles. Cependant, les polynômes d'ordre élevé ont tendance à se comporter de manière extrême aux limites ou dans des régions ne comportant que peu de données.

À titre de comparaison, voyons ce que donne un modèle de SVM à noyau entraîné sur le jeu de données d'origine, sans aucune transformation (voir la Figure 4.6) :

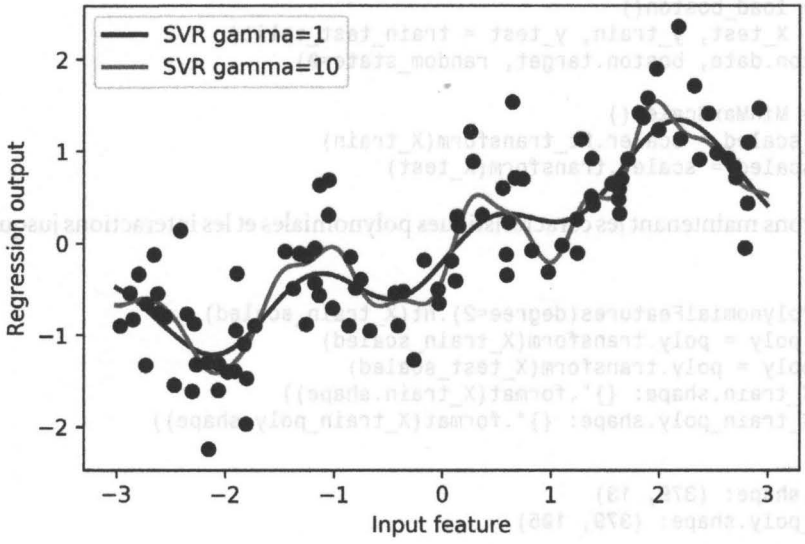



*Figure 4.6 : Comparaison de plusieurs paramètres* `gamma` *pour un SVM avec noyau RBF.*

**In[25]:**

```
from sklearn.svm import SVR

for gamma in [1, 10]:
    svr = SVR(gamma=gamma).fit(X, y)
    plt.plot(line, svr.predict(line), label='SVR gamma={}'.format(gamma))

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

En utilisant un modèle plus complexe, ici un SVM à noyau, nous pouvons réaliser une prédiction aussi bonne que dans le cas de la régression polynomiale, mais sans transformation explicite des caractéristiques.

Pour une application plus réaliste de ces méthodes, revenons au jeu de données Boston Housing. Nous avons déjà fait appel à des caractéristiques polynomiales sur ce jeu de données dans le Chapitre 2. Regardons maintenant de plus près comment ces caractéristiques ont été construites, et

à quel point les fonctions polynomiales peuvent aider. Nous allons commencer par charger les données, puis par les recalibrer entre 0 et 1 avec `MinMaxScaler` :

**In[26] :**

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(
    boston.data, boston.target, random_state=0)

scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Nous extrayons maintenant les caractéristiques polynomiales et les interactions jusqu'au degré 2 :

**In[27] :**

```
poly = PolynomialFeatures(degree=2).fit(X_train_scaled)
X_train_poly = poly.transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_poly.shape: {}".format(X_train_poly.shape))
```

**Out[27] :**

```
X_train.shape: (379, 13)
X_train_poly.shape: (379, 105)
```

Les données originales comportaient 13 caractéristiques, qui ont été étendues en 105 interactions. Ces nouvelles caractéristiques représentent toutes les interactions possibles entre deux caractéristiques d'origine, ainsi que le carré de celles-ci. Définir `degree=2` signifie que nous regardons toutes les caractéristiques qui sont le produit de deux caractéristiques de départ. La correspondance exacte entre caractéristiques en entrée et en sortie peut être trouvée en utilisant la méthode `get_feature_names` :

**In[28] :**

```
print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

**Out[28] :**

```
Polynomial feature names:
['1', 'x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10', 'x11',
 'x12', 'x0^2', 'x0 x1', 'x0 x2', 'x0 x3', 'x0 x4', 'x0 x5', 'x0 x6', 'x0 x7',
 'x0 x8', 'x0 x9', 'x0 x10', 'x0 x11', 'x0 x12', 'x1^2', 'x1 x2', 'x1 x3',
 'x1 x4', 'x1 x5', 'x1 x6', 'x1 x7', 'x1 x8', 'x1 x9', 'x1 x10', 'x1 x11',
 'x1 x12', 'x2^2', 'x2 x3', 'x2 x4', 'x2 x5', 'x2 x6', 'x2 x7', 'x2 x8',
 'x2 x9', 'x2 x10', 'x2 x11', 'x2 x12', 'x3^2', 'x3 x4', 'x3 x5', 'x3 x6',
 'x3 x7', 'x3 x8', 'x3 x9', 'x3 x10', 'x3 x11', 'x3 x12', 'x4^2', 'x4 x5',
 'x4 x6', 'x4 x7', 'x4 x8', 'x4 x9', 'x4 x10', 'x4 x11', 'x4 x12', 'x5^2',
 'x5 x6', 'x5 x7', 'x5 x8', 'x5 x9', 'x5 x10', 'x5 x11', 'x5 x12', 'x6^2',
```

```
'x6 x7', 'x6 x8', 'x6 x9', 'x6 x10', 'x6 x11', 'x6 x12', 'x7^2', 'x7 x8',
'x7 x9', 'x7 x10', 'x7 x11', 'x7 x12', 'x8^2', 'x8 x9', 'x8 x10', 'x8 x11',
'x8 x12', 'x9^2', 'x9 x10', 'x9 x11', 'x9 x12', 'x10^2', 'x10 x11',
'x10 x12', 'x11^2', 'x11 x12', 'x12^2']
```

La première nouvelle caractéristique est une constante appelée ici `'1'`. Les 13 suivantes sont les caractéristiques d'origine (dénommées `'x0'` à `'x12'`). Elles viennent après le carré de la première caractéristique (`'x0^2'`), puis les combinaisons de cette première caractéristique et des suivantes, et ainsi de suite.

Comparons les performances obtenues avec une régression ridge appliquée sur ce jeu de données avec et sans interactions :

**In[29]:**

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train_scaled, y_train)
print("Score without interactions: {:.3f}".format(
    ridge.score(X_test_scaled, y_test)))
ridge = Ridge().fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(
    ridge.score(X_test_poly, y_test)))
```

**Out[29]:**

```
Score without interactions: 0.621
Score with interactions: 0.753
```

Clairement, les interactions et les fonctions polynomiales améliorent ici sensiblement les résultats. En revanche, l'histoire est assez différente si nous effectuons le même travail avec un modèle plus complexe comme une forêt aléatoire :

**In[30]:**

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100).fit(X_train_scaled, y_train)
print("Score without interactions: {:.3f}".format(
    rf.score(X_test_scaled, y_test)))
rf = RandomForestRegressor(n_estimators=100).fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(rf.score(X_test_poly, y_test)))
```

**Out[30]:**

```
Score without interactions: 0.796
Score with interactions: 0.764
```

On peut constater que, même sans caractéristiques supplémentaires, la forêt aléatoire bat la performance de Ridge. Et le fait d'ajouter des interactions et des fonctions polynomiales fait même légèrement baisser les performances.

## Transformations non linéaires univariées

Nous venons de voir qu'ajouter des caractéristiques carrées (ou cubiques) peut aider à améliorer les performances des modèles linéaires pour la régression. Il y a également d'autres transformations qui peuvent souvent se révéler bénéfiques, en particulier l'application de fonctions mathématiques telles que `log`, `exp` ou `sin`. Alors que les modèles arborescents ne se soucient que de l'ordre des caractéristiques, les modèles linéaires et les réseaux de neurones sont très liés à l'échelle et à la distribution de chaque caractéristique, et par le fait de savoir s'il existe une relation non linéaire entre la caractéristique et la cible, ce qui devient difficile à modéliser, en particulier dans le cas de la régression. Les fonctions `log` et `exp` peuvent aider en ajustant les échelles relatives des données de façon à ce qu'elles puissent être plus facilement capturées par un modèle linéaire ou un réseau de neurones. Nous en avons vu une application dans le Chapitre 2, avec l'exemple du prix des mémoires. Les fonctions `sin` et `cos` peuvent également être intéressantes dans le cas de données qui encodent par exemple des signaux périodiques.

La plupart des modèles fonctionnent mieux lorsque chaque caractéristique (et aussi la cible dans le cas de la régression) est distribuée de manière à peu près gaussienne. En d'autres termes, un histogramme de chaque caractéristique devrait, dans le meilleur des mondes, donner une représentation ayant à peu près la forme bien connue d'une « courbe en cloche ». Utiliser des transformations comme `log` et `exp` est une technique un peu spéciale, mais simple et efficace pour obtenir ce résultat. Une situation particulièrement courante dans laquelle une telle transformation peut être utile est celle où il s'agit de dénombrer des entiers. Il peut par exemple s'agir de traiter une question du genre : « Combien de fois un utilisateur A s'est-il connecté ? ». Ces décomptes ne sont jamais négatifs, et ils suivent souvent des schémas statistiques particuliers. Ici, nous allons nous servir d'un jeu de données synthétique ayant des propriétés semblables à celles que vous pouvez trouver dans la nature. Les caractéristiques sont toutes des valeurs entières, tandis que la réponse est continue :

**In[31]:**

```
rnd = np.random.RandomState(0)
X_org = rnd.normal(size=(1000, 3))
w = rnd.normal(size=3)

X = rnd.poisson(10 * np.exp(X_org))
y = np.dot(X_org, w)
```

Regardons les dix premières entrées de la première caractéristique. Ce sont toutes des valeurs entières et positives, mais il est difficile d'y trouver un schéma particulier.

Si nous comptons le nombre d'occurrences de chaque valeur, la distribution en devient plus claire :

**In[32]:**

```
print("Number of feature appearances:\n{}".format(np.bincount(X[:, 0])))
```

**Out[32]:**

```
Number of feature appearances:
[28 38 68 48 61 59 45 56 37 40 35 34 36 26 23 26 27 21 23 23 18 21 10  9 17
  9  7 14 12  7  3  8  4  5  5  3  4  2  4  1  1  3  2  5  3  8  2  5  2  1
  2  3  3  2  2  3  3  0  1  2  1  0  0  3  1  0  0  0  1  3  0  1  0  2  0
  1  1  0  0  0  0  1  0  0  2  2  0  1  1  0  0  0  0  1  1  0  0  0  0  0
  0  0  1  0  0  0  0  0  1  1  0  0  1  0  0  0  0  0  0  0  1  0  0  0  0
  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1]
```

La valeur 2 semble être la plus répandue, avec 68 apparitions (la fonction `bincount` partant toujours de zéro). Le compte des valeurs les plus élevées chute ensuite rapidement. Cependant, il y a quelques très hautes valeurs, comme 84 et 85 qui apparaissent deux fois. Nous visualisons ces décomptes sur la Figure 4.7 :

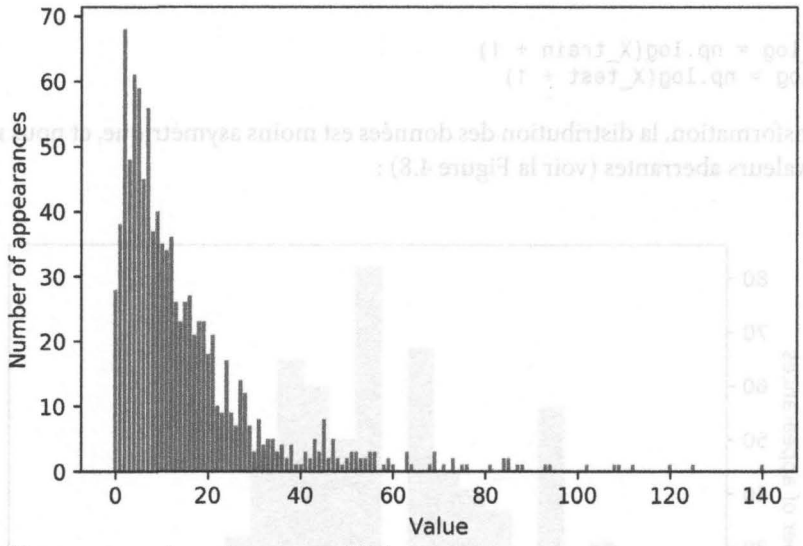



*Figure 4.7 : Histogramme des valeurs pour X[0].*

**In[33]:**

```
bins = np.bincount(X[:, 0])
plt.bar(range(len(bins)), bins, color='grey')
plt.ylabel("Number of appearances")
plt.xlabel("Value")
```

Les caractéristiques `X[:, 1]` et `X[:, 2]` ont des propriétés similaires. Ce type de distribution de valeurs (beaucoup de petites et peu de grandes) est très courant en pratique (il s'agit ici d'une distribution de Poisson, qui est assez fondamentale dans le comptage de données). Essayons de voir ce que donne une régression ridge sur ce modèle :

**In[34] :**

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
score = Ridge().fit(X_train, y_train).score(X_test, y_test)
print("Test score: {:.3f}".format(score))
```

**Out[34] :**

```
Test score: 0.622
```

Comme on peut le constater sur ce score $R^2$ relativement bas, `ridge` n'a pas été capable de capturer réellement la relation entre `X` et `y`. Appliquer un algorithme de transformation peut cependant aider. Comme la valeur 0 apparaît dans les données (et que le logarithme n'est pas défini pour une valeur nulle), nous ne pouvons pas appliquer directement la fonction `log`. Nous allons donc calculer à la place `log(X + 1)` :

**In[35] :**

```
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```

Après la transformation, la distribution des données est moins asymétrique, et nous n'avons plus de grandes valeurs aberrantes (voir la Figure 4.8) :

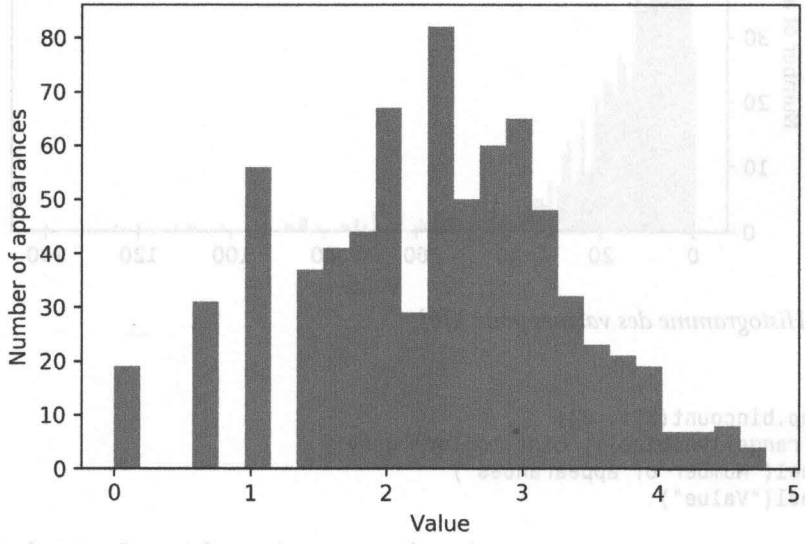



*Figure 4.8 : Histogramme des valeurs pour X[0] après transformation logarithmique.*

In[36]:

```
plt.hist(X_train_log[:, 0], bins=25, color='gray')
plt.ylabel("Number of appearances")
plt.xlabel("Value")
```

Reprenons notre régression ridge avec ces nouvelles données :

In[37]:

```
score = Ridge().fit(X_train_log, y_train).score(X_test_log, y_test)
print("Test score: {:.3f}".format(score))
```

Out[37]:

```
Test score: 0.875
```

Nous pouvons constater une nette amélioration des performances.

Trouver la transformation qui fonctionne le mieux pour chaque combinaison de jeu de données et de modèle est tout un art. Dans cet exemple, toutes les caractéristiques ont les mêmes propriétés. C'est rarement le cas en pratique et, en général, seul un sous-ensemble des caractéristiques devrait être transformé. Il peut aussi arriver que chaque caractéristique ait besoin d'être transformée d'une manière différente. Comme nous l'avons mentionné plus haut, ces types de transformations ne sont pas adaptés aux modèles arborescents. Par contre, ils peuvent être essentiels pour les modèles linéaires. Parfois, c'est aussi une bonne idée de transformer la variable cible y dans une régression. Essayer de prédire des comptages (comme un nombre de commandes) est une tâche tout à fait courante, et utiliser la transformation `log(y + 1)` y aide souvent (c'est une approximation très grossière de l'utilisation de la régression de Poisson, qui serait la bonne solution d'un point de vue probabiliste).

Comme nous l'avons vu dans les exemples précédents, la discrétisation, les polynômes et les interactions peuvent avoir une grande influence sur les performances des modèles en partant d'un certain jeu de données. Cela est particulièrement vrai pour les modèles les moins complexes, comme les modèles linéaires ou bayésiens naïfs. D'un autre côté, les modèles arborescents sont souvent capables de découvrir par eux-mêmes des interactions importantes, et ils ne nécessitent pas la plupart du temps une transformation explicite des données. D'autres types de modèles, comme les SVM, les plus proches voisins ou les réseaux de neurones, peuvent parfois bénéficier de ces manipulations, mais les implications en sont généralement bien moins claires que dans le cas des modèles linéaires.

# Sélection automatique de caractéristiques

Avec autant de façons de créer de nouvelles caractéristiques, vous pourriez être tenté d'augmenter la dimensionnalité des données bien au-delà du nombre de caractéristiques d'origine. Cependant, cela rend tous les modèles plus complexes et augmente les risques de surapprentissage.

D'une manière générale, et en particulier dans le cas de jeux de données comportant un grand nombre de dimensions, une bonne idée consiste souvent à réduire le nombre de caractéristiques pour ne conserver que les plus utiles et laisser les autres de côté. Cela peut permettre d'obtenir des modèles plus simples et qui se généralisent mieux. Mais comment pouvez-vous savoir à quel point chaque caractéristique est « bonne » ? Pour répondre à cette question, vous disposez de trois stratégies de base que nous allons étudier dans ce qui suit : les *statistiques univariées*, la *sélection basée sur le modèle*, et la *sélection itérative*. Toutes ces méthodes sont supervisées, ce qui signifie qu'elles ont besoin de la cible pour l'ajustement du modèle. Nous devons donc diviser les données en un jeu d'apprentissage et un jeu de test, et n'opérer la sélection des caractéristiques que sur le jeu d'apprentissage.

## Statistiques univariées

Dans les statistiques univariées, nous calculons s'il existe une relation statistiquement signifiante entre chaque caractéristique et la cible. Ensuite, les caractéristiques qui sont liées avec le plus haut degré de confiance sont sélectionnées. Dans le cas de la classification, on appelle cela l'*analyse de variance*, ou ANOVA (qui est un raccourci pour *ANalysis Of VAriance*). Une propriété essentielle de ces tests est qu'ils sont *univariés*, c'est-à-dire qu'ils considèrent uniquement chaque caractéristique de manière individuelle. Par conséquent, une caractéristique sera rejetée si elle est seulement informative lorsqu'elle est combinée avec une autre. Les calculs de tests univariés sont souvent très rapides, et ils ne nécessitent pas la construction d'un modèle. D'un autre côté, ils sont complètement indépendants du modèle que vous pourriez vouloir appliquer une fois les caractéristiques sélectionnées.

Pour utiliser cette sélection des caractéristiques dans `scikit-learn`, vous devez choisir un test, généralement *via* la méthode `f_classif` (par défaut) pour la classification ou `f_regression` pour la régression, ainsi qu'une méthode pour rejeter les caractéristiques en fonction des valeurs *p-values* déterminées par le test. Toutes les méthodes de rejet se servent d'un seuil pour enlever les caractéristiques pour lesquelles la valeur de *p-value* est trop élevée (ce qui signifie qu'elles sont peu susceptibles d'être liées à la cible). Elles diffèrent dans la manière de calculer ce seuil, entre la plus simple, `SelectKBest`, qui sélectionne un nombre fixé ($k$) de caractéristiques, ou encore `SelectPercentile`, qui sélectionne un pourcentage fixé de ces caractéristiques. Essayons d'appliquer ce mode de sélection de caractéristiques pour la classification sur le jeu de données `cancer`. Pour rendre la tâche un peu plus difficile, nous allons ajouter du bruit aux données. Nous voulons que la sélection des caractéristiques soit capable d'identifier celles qui n'apportent pas d'informations pour les éliminer :

**In[38]:**

```
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectPercentile
from sklearn.model_selection import train_test_split
```

```
cancer = load_breast_cancer()

rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))
X_w_noise = np.hstack([cancer.data, noise])

X_train, X_test, y_train, y_test = train_test_split(
    X_w_noise, cancer.target, random_state=0, test_size=.5)
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
X_train_selected = select.transform(X_train)

print("X_train.shape: {}".format(X_train.shape))
print("X_train_selected.shape: {}".format(X_train_selected.shape))
```

**Out[38]:**

```
X_train.shape: (284, 80)
X_train_selected.shape: (284, 40)
```

Comme vous pouvez le voir, le nombre de caractéristiques a été réduit de 80 à 40, soit une diminution de 50 %. Nous pouvons découvrir les caractéristiques qui ont été sélectionnées en utilisant la méthode `get_support`, qui retourne le masque booléen tel qu'il est visualisé sur la Figure 4.9 :

**In[39]:**

```
mask = select.get_support()
print(mask)
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
plt.yticks(())
```

**Out[39]:**

```
 True  True  True  True  True  True  True  True  True False  True False
  True  True  True  True  True  True False False  True  True  True  True
  True  True  True  True  True  True False False False  True False  True
 False False  True False False False False  True False False  True False
 False  True False  True False False False False False False  True False
  True False False False False  True False  True False False False False
  True  True False  True False False False False]
```

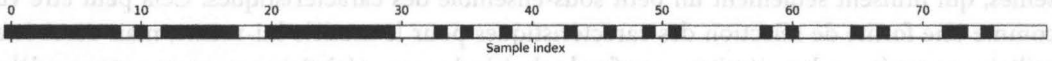



*Figure 4.9 : Caractéristiques sélectionnées par SelectPercentile.*

Vous constatez que la plupart des caractéristiques sélectionnées sont celles qui étaient déjà présentes à l'origine, et que, pour l'essentiel, le bruit a été supprimé. Pour autant, la récupération des caractéristiques originales n'est pas parfaite. Comparons les performances d'une régression logistique appliquée à l'ensemble des caractéristiques avec celles qui résultent du même travail effectué uniquement avec les caractéristiques sélectionnées :

In[40] :

```
from sklearn.linear_model import LogisticRegression

X_test_selected = select.transform(X_test)

lr = LogisticRegression()
lr.fit(X_train, y_train)
print("Score with all features: {:.3f}".format(lr.score(X_test, y_test)))
lr.fit(X_train_selected, y_train)
print("Score with only selected features: {:.3f}".format(
    lr.score(X_test_selected, y_test)))
```

Out[40] :

```
Score with all features: 0.930
Score with only selected features: 0.940
```

Dans ce cas, supprimer le bruit améliore les performances, même si certaines des caractéristiques d'origine sont perdues. Il s'agit bien sûr ici d'un exemple synthétique très simple, et les sorties obtenues sur des données réelles sont généralement bien plus complexes. Pour autant, la sélection univariée peut tout de même être très utile s'il y a tellement de caractéristiques que bâtir un modèle sur elles soit infaisable, ou si vous suspectez que de nombreuses caractéristiques ne comportent aucune information utile.

## Sélection de caractéristiques basée sur le modèle

Cette méthode de sélection utilise un modèle d'apprentissage automatique supervisé pour juger de l'importance de chaque caractéristique afin de ne conserver que les plus intéressantes. Le modèle supervisé servant à cette sélection n'a pas besoin d'être le même que celui qui va servir à la modélisation finale du jeu de données. En revanche, il doit fournir une certaine mesure de l'importance de chaque caractéristique afin de pouvoir effectuer un classement en fonction de cette mesure. Les arbres de décision, ainsi que les autres modèles de décision arborescents, possèdent un attribut `feature_importances_` qui encode directement l'importance de chaque attribut. Les modèles linéaires disposent de coefficients qui peuvent aussi être utilisés pour capturer l'importance des caractéristiques en considérant les valeurs absolues. Comme nous l'avons vu dans le Chapitre 2, les modèles linéaires avec une pénalité L1 génèrent des coefficients clairsemés, qui utilisent seulement un petit sous-ensemble des caractéristiques. Cela peut être vu comme une forme de sélection des caractéristiques pour le modèle lui-même, mais aussi être utilisé comme étape de prétraitement afin de choisir des caractéristiques pour un autre modèle. Par contraste avec la sélection univariée, cette méthode considère toutes les caractéristiques de manière globale, et elle peut donc en capturer les interactions (du moins si le modèle en est capable). Pour utiliser la sélection basée sur le modèle, vous avez besoin de faire appel à la classe `SelectFromModel` de `scikit-learn` :

**In[41]:**

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
    threshold="median")
```

La classe `SelectFromModel` sélectionne toutes les caractéristiques qui ont une mesure de leur importance (telle qu'elle est fournie par le modèle supervisé) plus grande que le seuil spécifié. Pour obtenir un résultat comparable à celui que nous avions trouvé avec la sélection de caractéristiques univariée, nous avons défini un seuil médian (`threshold="median"`). De ce fait, nous sélectionnons la moitié des caractéristiques. Pour calculer l'importance de celles-ci, nous avons utilisé comme classifieur une forêt aléatoire avec 100 arbres. Ce modèle est plutôt complexe, et beaucoup plus puissant que les tests univariés. Voyons maintenant ce que va donner l'entraînement du modèle :

**In[42]:**

```
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_l1.shape: {}".format(X_train_l1.shape))
```

**Out[42]:**

```
X_train.shape: (284, 80)
X_train_l1.shape: (284, 40)
```

Nous pouvons à nouveau jeter un coup d'œil sur les caractéristiques qui ont été sélectionnées (voir la Figure 4.10) :

**In[43]:**

```
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
plt.yticks(())
```

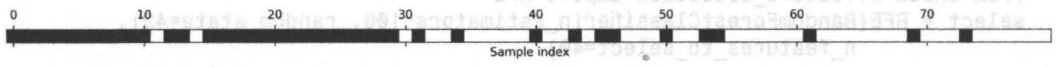



*Figure 4.10 : Caractéristiques sélectionnées par* `SelectFromModel` *en utilisant* `RandomForestClassifier`.

Cette fois, toutes les caractéristiques d'origine sauf deux ont été sélectionnées. Du fait que nous avions demandé une sélection médiane (soit 40 caractéristiques sur 80), il reste également un peu de bruit. Voyons ce que donnent les performances :

In[44]:

```
X_train_rfe = select.transform(X_train)
X_test_rfe = select.transform(X_test)

score = LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
print("Test score: {:.3f}".format(score))
```

Out[44]:

```
Test score: 0.951
```

Avec cette sélection des caractéristiques, nous avons légèrement amélioré les résultats.

## Sélection itérative de caractéristiques

Dans notre test univarié, nous n'avons utilisé aucun modèle. Dans la sélection basée sur le modèle, nous avons fait appel à un modèle unique pour sélectionner des caractéristiques. Avec la sélection itérative, on construit une série de modèles avec un nombre variable de caractéristiques. Il y a deux méthodes de base : commencer sans caractéristique puis les ajouter une par une jusqu'à ce qu'un certain critère d'arrêt soit atteint, ou inversement commencer avec toutes les caractéristiques et les supprimer une par une jusqu'à ce qu'un certain critère d'arrêt soit atteint. Du fait que nous construisons une série de modèles, ces méthodes sont bien plus gourmandes en termes de traitement informatique que les précédentes. Une méthode particulière de ce type est l'algorithme RFE (*recursive feature elimination*) qui débute avec toutes les caractéristiques, construit un modèle, et rejette la caractéristique la moins importante pour ce modèle. Un nouveau modèle est ensuite construit à partir des caractéristiques restantes, et ainsi de suite jusqu'à ce qu'il ne reste qu'un nombre prédéfini de caractéristiques. Pour que cela puisse fonctionner, il faut que le modèle utilisé pour la sélection soit capable de fournir une méthode pour déterminer l'importance des caractéristiques, comme dans le cas de la sélection basée sur le modèle. Ici, nous allons nous servir de la même forêt aléatoire que ci-dessus, ce qui va nous donner les résultats illustrés sur la Figure 4.11 :

In[45]:

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42),
             n_features_to_select=40)

select.fit(X_train, y_train)
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
plt.yticks(())
```

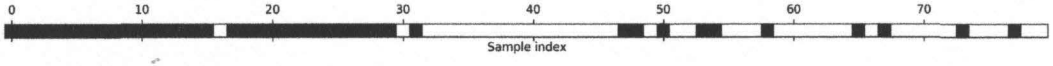



*Figure 4.11 : Caractéristiques sélectionnées par RFE en utilisant* `RandomForestClassifier`.

La sélection des caractéristiques est meilleure que celle obtenue par la méthode univariée ou basée sur le modèle, mais une caractéristique a encore été manquée. L'exécution de ce code prend aussi nettement plus de temps que la sélection basée sur le modèle, puisque la forêt aléatoire est entraînée 40 fois. Testons ensuite l'exactitude de la régression logistique lors de l'utilisation de RFE pour la sélection des caractéristiques :

**In[46]:**

```
X_train_rfe = select.transform(X_train)
X_test_rfe = select.transform(X_test)

score = LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
print("Test score: {:.3f}".format(score))
```

**Out[46]:**

```
Test score: 0.951
```

Nous pouvons également nous servir du modèle utilisé dans RFE pour faire des prédictions. Nous n'avons besoin pour cela que du jeu de caractéristiques qui a été sélectionné :

**In[47]:**

```
print("Test score: {:.3f}".format(select.score(X_test, y_test)))
```

**Out[47]:**

```
Test score: 0.951
```

Ici, la performance de la forêt aléatoire utilisée dans RFE est la même que celle obtenue en entraînant un modèle de régression logistique à partir des caractéristiques sélectionnées. En d'autres termes, une fois que nous avons sélectionné les « bonnes » caractéristiques, le modèle linéaire est aussi efficace que la forêt aléatoire.

Si vous n'êtes pas sûr de ce qu'il convient d'utiliser en entrée pour vos algorithmes d'apprentissage automatique, la sélection automatique de caractéristiques peut être assez utile. Elle est aussi très bien adaptée pour réduire le nombre des caractéristiques nécessaires, par exemple pour accélérer les prédictions ou pour rendre les modèles plus faciles à interpréter. Dans la plupart des applications du monde réel, il est peu probable que la sélection des caractéristiques apporte des gains de performances significatifs. Par contre, cela reste un outil de valeur pour l'ingénierie des caractéristiques.

## Savoir utiliser l'expertise

L'ingénierie des caractéristiques est souvent un domaine important pour utiliser les *connaissances d'experts* (ou encore l'expertise métier) dans le cadre d'une application particulière. Bien que le but de l'apprentissage automatique soit dans de nombreux cas d'éviter d'avoir à créer un ensemble de règles conçues par des experts, cela ne signifie pas que la connaissance préalable de

l'application ou du domaine devrait être écartée. Souvent, un œil d'expert peut aider à identifier des caractéristiques utiles qui fournissent plus d'informations que la représentation initiale des données. Imaginez par exemple que vous travaillez dans une agence de voyages et que vous vouliez prédire le prix de billets d'avion. Supposons donc que vous disposez de nombreux enregistrements de prix, avec les dates, les compagnies, les villes de départ et les villes de destination. Un modèle d'apprentissage automatique pourrait être capable de construire un modèle décent à partir de ces informations. En revanche, certains facteurs importants ne peuvent pas être pris en compte de cette manière. Par exemple, les vols sont souvent plus chers lors des périodes de vacances ou de grandes fêtes. Les dates de certaines fêtes, comme Noël, sont fixes et leur effet peut donc être assez facilement pris en compte. Par contre, d'autres événements ont des dates variables, comme Pâques ou encore la Pentecôte. D'autres encore dépendent de décisions prises par les autorités, comme pour les congés scolaires. Ces événements ne peuvent pas être déduits des données si chaque vol est enregistré uniquement sur la base des dates. Cependant, il est facile d'ajouter une caractéristique qui permettra d'encoder le fait de savoir si un vol a lieu juste avant, pendant ou juste après une période fériée amovible ou des jours de vacances scolaires. De cette manière, une connaissance préalable quant à la nature de la tâche peut être encodée dans certaines caractéristiques de manière à venir en aide à l'algorithme d'apprentissage automatique. Ajouter une caractéristique ne force pas un algorithme à l'utiliser, et même si une information sur des dates de congés se révélait sans utilité particulière, cette augmentation du volume des données ne ferait aucun mal.

Nous allons nous intéresser dans ce qui suit à un cas particulier d'expertise, qui devrait en fait plutôt être appelé ici le « sens commun ». La tâche consiste à prédire des locations de vélo devant la maison d'Andreas.

À New York, Citi Bike gère un réseau de stations de vélos en libre-service avec ou sans abonnement. Ces stations couvrent tout le centre-ville et ce système permet, comme dans beaucoup d'autres grandes agglomérations, de se déplacer facilement pour un coût limité. Les données concernant la location des vélos sont rendues publiques dans des fichiers rendus anonymes et qui ont été analysés de multiples manières (voyez à ce sujet l'adresse `https://www.citibikenyc.com/system-data`). La tâche à laquelle nous nous attelons consiste à prédire pour un certain jour et à une certaine heure le nombre de personnes qui vont louer un vélo devant la maison d'Andreas (afin qu'il puisse savoir s'il va en rester un pour lui).

Nous commençons par charger en tant que `DataFrame` `pandas` les données correspondant à cette station spécifique pour le mois d'août 2015. Nous rééchantillonnons les données par tranches de trois heures afin d'obtenir les principales tendances pour chaque journée :

**In[48]:**
```
citibike = mglearn.datasets.load_citibike()
```
**In[49]:**
```
print("Citibike data:\n{}".format(citibike.head()))
```

**Out[49]:**

```
Citibike data:
starttime
2015-08-01 00:00:00     3.0
2015-08-01 03:00:00     0.0
2015-08-01 06:00:00     9.0
2015-08-01 09:00:00    41.0
2015-08-01 12:00:00    39.0
Freq: 3H, Name: one, dtype: float64
```

L'exemple qui suit montre une visualisation des fréquences de location pour la totalité du mois (voir la Figure 4.12) :

**In[50]:**

```
plt.figure(figsize=(10, 3))
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.max(),
                       freq='D')
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")
plt.plot(citibike, linewidth=1)
plt.xlabel("Date")
plt.ylabel("Rentals")
```

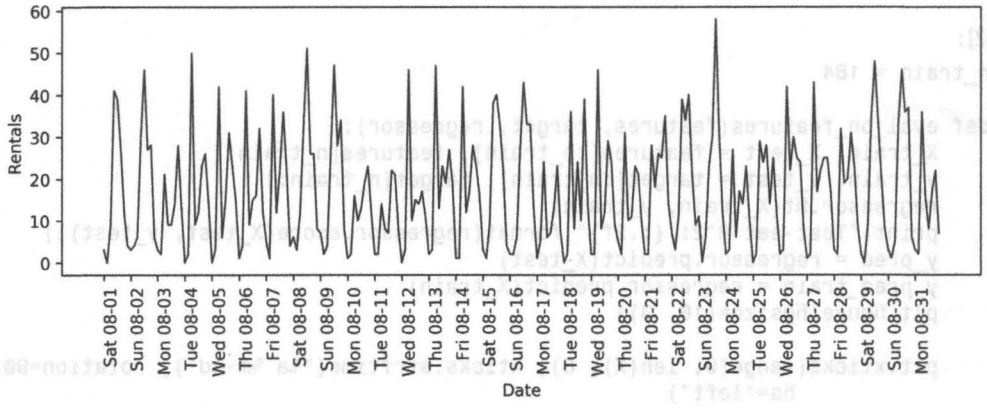



*Figure 4.12 : Nombre de locations de vélos pour une station Citi Bike donnée et pour un mois donné.*

En observant les données, nous pouvons clairement distinguer le jour et la nuit pour chacun des intervalles de 24 heures. Les valeurs pour les jours de la semaine et les week-ends semblent aussi assez différentes. Lorsqu'il s'agit d'évaluer une tâche de prédiction pour une série temporelle comme celle-ci, nous voulons apprendre quelque chose du passé et prédire quelque chose pour le futur. En d'autres termes, nous voulons utiliser toutes les données jusqu'à une certaine date pour le jeu d'apprentissage, et toutes les données après cette date pour le jeu de test. Et dit encore autrement, nous *savons* quelque chose sur le passé (en l'occurrence de simples locations), et nous *croyons* que

quelque chose va se passer dans le futur (en l'occurrence, d'autres locations). Dans cet exemple, nous allons utiliser les 184 premiers points de données, correspondant aux 23 premiers jours, comme jeu d'apprentissage, et les 64 points de données restants, soit 8 jours, comme jeu de test.

La seule caractéristique que nous utilisons dans notre tâche de prédiction est la date et l'heure d'une ou plusieurs locations de vélos à l'endroit choisi. La caractéristique d'entrée est donc formée de la date et de l'heure, par exemple le 1er août 2015 à zéro heure (soit `2015-08-01 00-00-00` en notation anglo-saxonne). La sortie est le nombre de locations dans les heures qui suivent (soit trois heures pour le `DataFrame` choisi).

Une façon courante d'enregistrer les dates dans les ordinateurs consiste à utiliser le temps dit POSIX (une norme UNIX essentielle), qui est le nombre de secondes écoulées depuis le tout début de l'année 1970 (ce que l'on appelle aussi le début du temps UNIX). Pour un premier essai, nous pouvons utiliser cette caractéristique entière particulière pour représenter nos données :

**In[51]:**

```
y = citibike.values
X = citibike.index.astype("int64").values.reshape(-1, 1) // 10**9
```

Nous définissons d'abord une fonction pour diviser les données, entre jeux d'apprentissage et de test, nous construisons ensuite le modèle et nous visualisons le résultat :

**In[52]:**

```
n_train = 184

def eval_on_features(features, target, regressor):
    X_train, X_test = features[:n_train], features[n_train:]
    y_train, y_test = target[:n_train], target[n_train:]
    regressor.fit(X_train, y_train)
    print("Test-set R^2: {:.2f}".format(regressor.score(X_test, y_test)))
    y_pred = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)
    plt.figure(figsize=(10, 3))

    plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotation=90,
               ha="left")

    plt.plot(range(n_train), y_train, label="train")
    plt.plot(range(n_train, len(y_test) + n_train), y_test, '-', label="test")
    plt.plot(range(n_train), y_pred_train, '--', label="prediction train")

    plt.plot(range(n_train, len(y_test) + n_train), y_pred, '--',
             label="prediction test")
    plt.legend(loc=(1.01, 0))
    plt.xlabel("Date")
    plt.ylabel("Rentals")
```

Nous avons déjà vu que les forêts aléatoires ne nécessitent que très peu de prétraitement des données, ce qui semble en faire un bon modèle pour commencer. Nous utilisons la caractéris-

tique de temps POSIX X et nous passons une forêt aléatoire en tant que *régresseur* à notre fonction `eval_on_features`. La Figure 4.13 illustre le résultat :

**In[53]:**

```
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators=100, random_state=0)
eval_on_features(X, y, regressor)
```

**Out[53]:**

```
Test-set R^2: -0.04
```

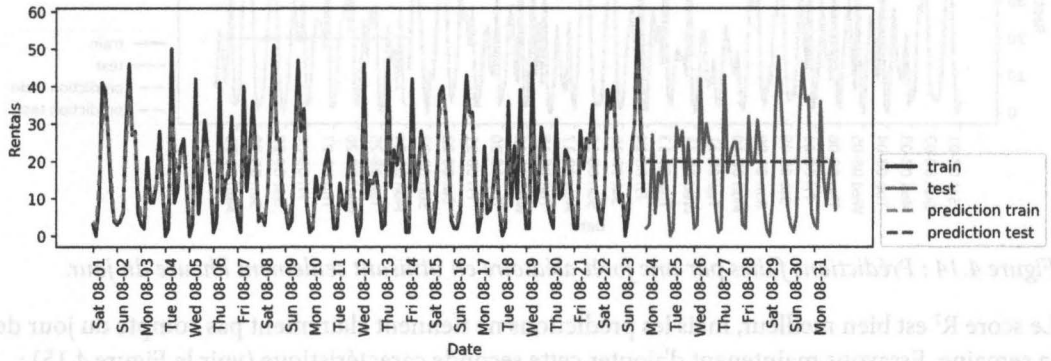



*Figure 4.13 : Prédictions effectuées par une forêt aléatoire en utilisant uniquement le temps POSIX.*

Les prédictions sur le jeu d'apprentissage sont assez bonnes, comme d'ailleurs d'habitude avec les forêts aléatoires. Cependant, pour le jeu de test, la prédiction donne une ligne droite horizontale. Le score $R^2$ est de -0.04, ce qui signifie que nous n'avons rien appris ? Que s'est-il passé ?

Le problème tient à la combinaison de notre caractéristique et de la forêt aléatoire. La valeur de la caractéristique de temps POSIX pour le jeu de test se trouve en dehors de la plage des valeurs disponibles dans le jeu d'apprentissage. En effet, toutes les dates du jeu de test sont postérieures à celles du jeu d'apprentissage. Les arbres, et donc les forêts aléatoires, ne peuvent pas *extrapoler* des caractéristiques au-delà des points contenus dans le jeu d'apprentissage. Le résultat est que le modèle prédit simplement la valeur cible correspondant au point le plus proche dans le jeu d'apprentissage, qui est la dernière heure pour laquelle les données ont été observées.

Nous pouvons clairement faire mieux. Et c'est là où la question de l'expertise entre en jeu. En observant les variations de location dans le jeu d'apprentissage, deux facteurs semblent être très importants : l'heure du jour et le jour de la semaine. Ajoutons ces deux caractéristiques. Nous ne pouvons réellement rien apprendre du temps POSIX, et donc nous laissons de côté cette caractéristique. Commençons alors par utiliser l'heure du jour. Comme l'illustre la Figure 4.14, les prédictions ont maintenant une forme semblable pour chaque jour de la semaine :

**In[54]:**

```
X_hour = citibike.index.hour.values.reshape(-1, 1)
eval_on_features(X_hour, y, regressor)
```

**Out[54]:**

```
Test-set R^2: 0.60
```

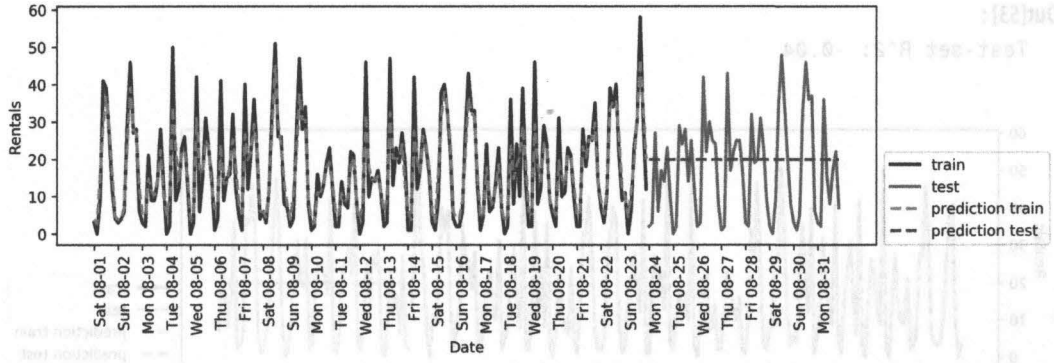



*Figure 4.14 : Prédictions faites par une forêt aléatoire en utilisant seulement l'heure du jour.*

Le score $R^2$ est bien meilleur, mais les prédictions ne tiennent clairement pas compte du jour de la semaine. Essayons maintenant d'ajouter cette seconde caractéristique (voir la Figure 4.15) :

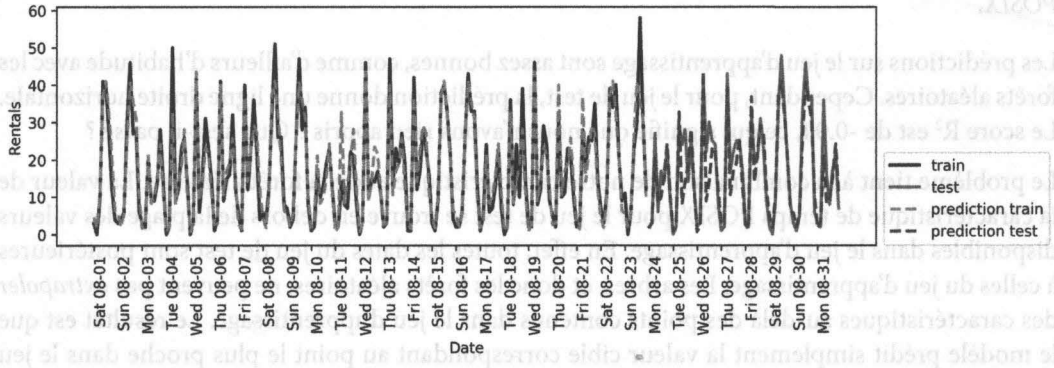



*Figure 4.15 : Prédictions faites par une forêt aléatoire en utilisant le jour de la semaine et l'heure du jour.*

**In[55]:**

```
X_hour_week = np.hstack([citibike.index.dayofweek.values.reshape(-1, 1),
                         citibike.index.hour.values.reshape(-1, 1)])
eval_on_features(X_hour_week, y, regressor)
```

**Out[55]:**

```
Test-set R^2: 0.84
```

Nous disposons maintenant d'un modèle qui capture le comportement périodique en considérant le jour de la semaine et l'heure du jour. Son score $R^2$ est de 0.84, ce qui montre des performances prédictives plutôt bonnes. Ce que ce modèle apprend vraisemblablement, c'est le nombre moyen de locations de vélos pour chaque combinaison de jours et de tranches horaires pour les 23 premiers jours d'août. Cela ne nécessite pas en réalité un modèle complexe comme une forêt aléatoire. Nous allons donc essayer de construire un modèle plus simple, en l'occurrence une régression linéaire (voir la Figure 4.16) :

**In[56]:**

```
from sklearn.linear_model import LinearRegression
eval_on_features(X_hour_week, y, LinearRegression())
```

**Out[56]:**

```
Test-set R^2: 0.13
```

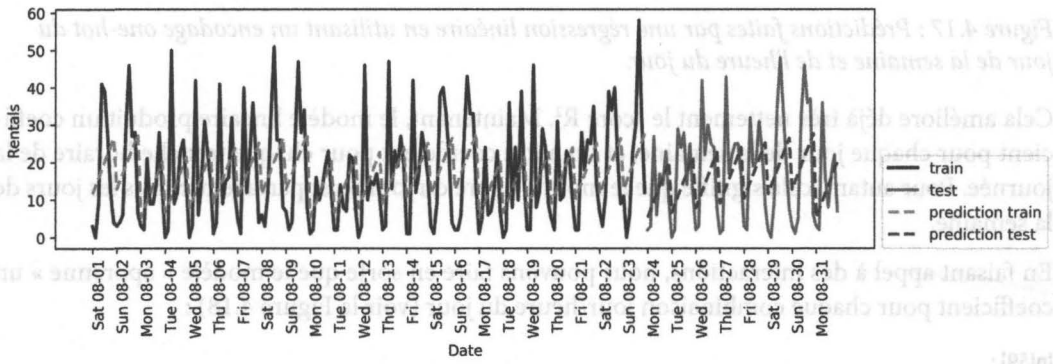



*Figure 4.16 : Prédictions faites par une régression linéaire en utilisant le jour de la semaine et l'heure du jour.*

Cette régression linéaire fait un travail bien plus mauvais, et le motif périodique a une forme bizarre. La raison tient au fait que nous avons encodé le jour de la semaine et l'heure du jour en utilisant des entiers, qui sont interprétés comme des variables continues. Par conséquent, ce modèle ne peut s'appuyer que sur une fonction linéaire de l'heure du jour, et il a découvert qu'il y a davantage de locations plus tard dans la journée. Cependant, la trame des données est bien plus complexe que cela. Nous pouvons essayer de capturer ces informations en interprétant les entiers sous forme de variables catégorielles, c'est-à-dire en les transformant à l'aide de `OneHotEncoder` (voir la Figure 4.17) :

**In[57]:**

```
enc = OneHotEncoder()
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()
```

**In[58]:**

```
eval_on_features(X_hour_week_onehot, y, Ridge())
```

**Out[58]:**

```
Test-set R^2: 0.62
```

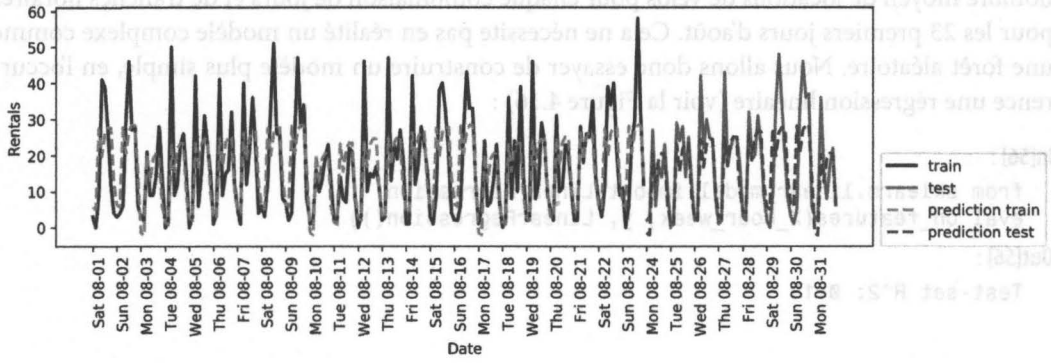



*Figure 4.17 : Prédictions faites par une régression linéaire en utilisant un encodage one-hot du jour de la semaine et de l'heure du jour.*

Cela améliore déjà très nettement le score $R^2$. Maintenant, le modèle linéaire produit un coefficient pour chaque jour de la semaine, et un autre coefficient pour chaque tranche horaire de la journée. Pour autant, cela signifie que le motif « heure du jour » est partagé par tous les jours de la semaine.

En faisant appel à des interactions, nous pouvons faire en sorte que le modèle « apprenne » un coefficient pour chaque combinaison jour/heure du jour (voir la Figure 4.18) :

**In[59]:**

```
poly_transformer = PolynomialFeatures(degree=2, interaction_only=True,
                                      include_bias=False)
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_onehot)
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```

**Out[59]:**

```
Test-set R^2: 0.85
```

Cette transformation conduit finalement à un modèle presque aussi performant que la forêt aléatoire. Un des grands bénéfices de cette méthode est qu'elle fournit des éléments très clairs, à savoir un coefficient pour chaque jour et chaque plage horaire. Nous pouvons donc simplement tracer tous ces coefficients, chose qui ne serait pas possible avec la forêt aléatoire.

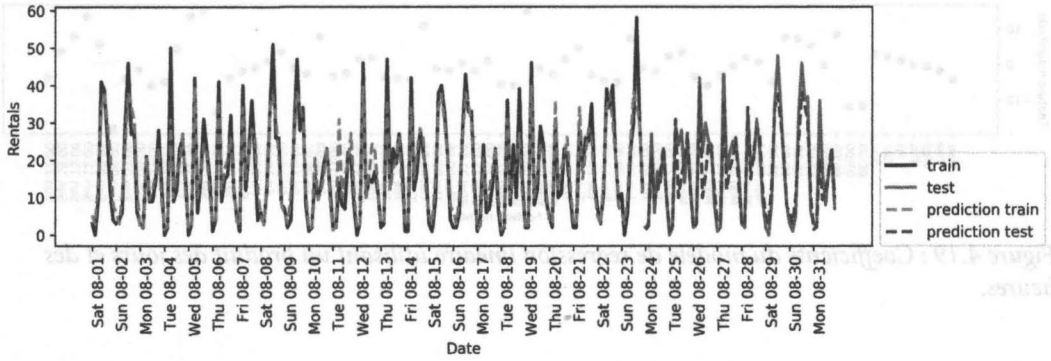



*Figure 4.18 : Prédictions faites par une régression linéaire en utilisant un produit du jour de la semaine et de l'heure du jour.*

Tout d'abord, nous créons des noms de caractéristiques pour les jours et les heures :

**In[60]:**

```
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features = day + hour
```

Nous nommons ensuite toutes les interactions extraites par `PolynomialFeatures` en utilisant la méthode `get_feature_names`, et nous ne conservons que les caractéristiques avec des coefficients non nuls :

**In[61]:**

```
features_poly = poly_transformer.get_feature_names(features)
features_nonzero = np.array(features_poly)[lr.coef_ != 0]
coef_nonzero = lr.coef_[lr.coef_ != 0]
```

Nous pouvons maintenant visualiser les coefficients obtenus par le modèle linéaire, comme l'illustre la Figure 4.19 :

**In[62]:**

```
plt.figure(figsize=(15, 2))
plt.plot(coef_nonzero, 'o')
plt.xticks(np.arange(len(coef_nonzero)), features_nonzero, rotation=90)
plt.xlabel("Feature name")
plt.ylabel("Feature magnitude")
```

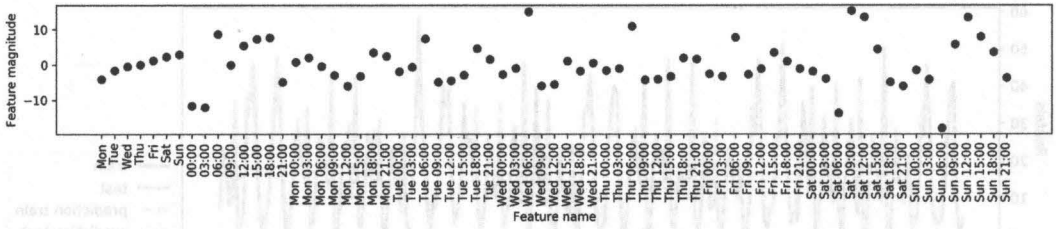



*Figure 4.19 : Coefficients du modèle de régression linéaire utilisant un produit des jours et des heures.*

# Résumé et perspectives

Dans ce chapitre, nous avons étudié diverses manières de travailler avec différents types de données (en particulier avec les variables catégorielles). Nous avons mis l'accent sur l'importance de la représentation des données d'une façon qui soit adaptée à l'algorithme d'apprentissage automatique, par exemple *via* un encodage one-hot des variables catégorielles. Nous avons aussi discuté du rôle de l'ingénierie des caractéristiques, et de la possibilité de faire appel à une expertise particulière afin de créer des caractéristiques dérivées de vos données. En particulier, les modèles linéaires peuvent grandement bénéficier de la génération des nouvelles caractéristiques *via* la discrétisation ou encore l'ajout de polynômes et d'interactions, ce qui n'est pas forcément le cas de modèles plus complexes non linéaires, comme les forêts aléatoires ou les SVM. En pratique, les caractéristiques qui sont utilisées (et la correspondance entre caractéristiques et méthode) sont souvent la pièce la plus importante pour construire un moteur d'apprentissage automatique efficace.

Maintenant que vous avez une bonne idée de la manière de représenter vos données de façon appropriée et de choisir un algorithme adapté en fonction de la tâche à accomplir, nous allons nous intéresser dans le prochain chapitre à l'évaluation des performances des modèles d'apprentissage automatique et à la sélection des meilleurs réglages des paramètres.

# CHAPITRE 5
# Évaluation et amélioration du modèle

Jusqu'ici, nous avons discuté de sujets de fond concernant l'apprentissage supervisé et non supervisé, et nous avons exploré divers algorithmes d'apprentissage. Nous allons maintenant nous intéresser de plus près à l'évaluation du modèle et à la sélection des paramètres.

Nous nous attacherons dans ce qui suit aux méthodes supervisées, régression et classification, du fait que l'évaluation et la sélection de modèles dans un apprentissage non supervisé sont souvent un processus très qualitatif, comme nous l'avons vu au Chapitre 3.

Pour évaluer nos modèles supervisés, nous avons jusqu'ici divisé notre jeu de données en deux parties à l'aide de la fonction `train_test_split` : les données d'apprentissage, et les données de test. Nous avons ensuite construit un modèle à partir du jeu d'apprentissage en appelant la fonction `fit`, puis lancé une évaluation sur le jeu de test en utilisant la méthode `score`, qui calcule la fraction des échantillons correctement classifiés. Voici un exemple de ce processus :

**In[1]:**

```
from preamble import *
%matplotlib inline

from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X, y = make_blobs(random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
logreg = LogisticRegression().fit(X_train, y_train)
print("Test set score: {:.2f}".format(logreg.score(X_test, y_test)))
```

**Out[1]:**

```
Test set score: 0.88
```

Souvenez-vous que la raison pour laquelle nous partageons nos données en deux jeux est que nous voulons mesurer la qualité de la *généralisation* dont est capable notre modèle lorsqu'on lui

transmet de nouvelles données, différentes de celles qu'il connaît déjà. La manière dont notre modèle se débrouille avec le jeu de données d'apprentissage n'est pas réellement notre sujet. Ce qui nous intéresse en fait, c'est de savoir s'il est capable de faire des prédictions correctes à partir de données qu'il n'a pas observées lors de l'apprentissage.

Dans ce chapitre, nous allons nous attacher à deux aspects de cette évaluation. Nous présenterons d'abord la validation croisée, qui est une manière plus robuste d'accéder aux performances de généralisation, et nous discuterons des méthodes permettant d'évaluer les performances de classification et de régression qui dépassent les capacités des mesures par défaut de l'exactitude et du $R^2$ (erreur d'ajustement) fournies par la méthode `score`.

Nous discuterons aussi de la recherche sur grille, une méthode efficace pour ajuster les paramètres des modèles supervisés afin d'atteindre les meilleures performances de généralisation possibles.

# Validation croisée

La *validation croisée* est une méthode statistique d'évaluation des performances de généralisation qui est plus stable et approfondie que le simple partage des données en un jeu d'apprentissage et un jeu de test. Dans la validation croisée, les données sont divisées de manière répétée (au lieu de deux jeux, l'idée consiste à partager les données en *k* séries d'échantillons, chacune servant à son tour de jeu de test). La méthode la plus couramment employée est dite *k-fold cross validation* (littéralement *validation croisée à k-plis*, forme que nous conserverons dans ce qui suit), où *k* est un nombre spécifié par l'utilisateur, généralement 5 ou 10. Lors d'une validation croisée à 5 plis, les données sont partagées en cinq parties, de taille approximativement égale, appelées *folds* (*plis*). Une suite de modèles est alors entraînée. Le premier modèle prend le premier pli comme jeu de test, les autres plis (de 2 à 5) formant le jeu d'apprentissage. Ce modèle est donc construit avec les données des plis 2 à 5, et l'exactitude de la prédiction est évaluée sur le pli 1. Un autre modèle est ensuite construit, cette fois en prenant le pli 2 comme jeu de test, et les plis 1, 3, 4 et 5 comme jeu d'apprentissage. Ce même processus est répété jusqu'à ce que chaque pli ait servi de jeu de test, l'exactitude de la prédiction étant évaluée à chaque fois. À la fin du processus, nous disposons donc de cinq indicateurs d'exactitude. Ce processus est illustré sur la Figure 5.1 :

**In[2]:**

```
mglearn.plots.plot_cross_validation()
```

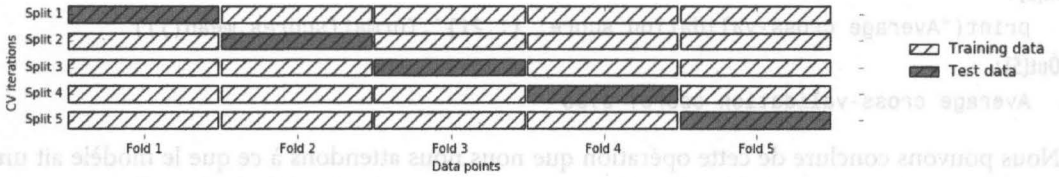



*Figure 5.1 : Partage des données dans une validation croisée à 5 plis.*

En règle générale, le premier cinquième des données constitue le pli 1, le second cinquième le pli 2, et ainsi de suite.

## Validation croisée dans `scikit-learn`

La validation croisée est implémentée dans `scikit-learn` *via* la fonction `cross_val_score` du module `model_selection`. Les arguments de cette fonction sont le modèle que nous voulons évaluer, les données d'apprentissage et les étiquettes cibles. Évaluons par exemple le modèle `LogisticRegression` sur le jeu de données `iris` :

**In[3]:**

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
logreg = LogisticRegression()

scores = cross_val_score(logreg, iris.data, iris.target)
print("Cross-validation scores: {}".format(scores))
```

**Out[3]:**

```
Cross-validation scores: [ 0.961  0.922  0.958]
```

Par défaut, `cross_val_score` effectue une validation croisée à 3 plis, et retourne donc trois indicateurs d'exactitude. Nous pouvons modifier le nombre de plis en changeant la valeur du paramètre `cv` :

**In[4]:**

```
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
print("Cross-validation scores: {}".format(scores))
```

**Out[4]:**

```
Cross-validation scores: [ 1.     0.967  0.933  0.9    1.   ]
```

Une manière courante de résumer l'exactitude de la validation croisée consiste à calculer la moyenne des valeurs obtenues :

In[5]:

```
print("Average cross-validation score: {:.2f}".format(scores.mean()))
```

Out[5]:

```
Average cross-validation score: 0.96
```

Nous pouvons conclure de cette opération que nous nous attendons à ce que le modèle ait un taux moyen d'exactitude de 96 %. De plus, si nous observons la série des résultats obtenus par la validation croisée à 5 plis, nous pouvons aussi conclure que la variation est relativement élevée d'un pli à un autre, puisque l'évaluation de l'exactitude va de 100 % à 90 %. Cela pourrait impliquer que le modèle est très dépendant des plis particuliers utilisés pour l'apprentissage, mais une autre conclusion possible est que le jeu de données utilisé ici est trop restreint.

## Bénéfices de la validation croisée

La validation croisée apporte plusieurs bénéfices par rapport au simple partage en deux jeux de données, un pour l'apprentissage et l'autre pour les tests. Tout d'abord, souvenez-vous que `train_test_split` effectue une division aléatoire des données. Si vous êtes « chanceux », tous les échantillons qui sont difficiles à classifier finiront dans le jeu d'apprentissage. Dans ce cas, le jeu de test ne contiendra que des échantillons « faciles », et l'exactitude obtenue sera surévaluée de façon irréaliste. Inversement, si vous êtes « malchanceux », il se peut que vous ayez aléatoirement stocké les données les plus difficiles à classifier dans le jeu de test, et que vous obteniez en conséquence un score anormalement bas. Avec la validation croisée, par contre, chaque échantillon apparaîtra exactement une fois dans le jeu d'apprentissage. En d'autres termes, chaque échantillon se trouve dans l'un des plis, et chaque pli joue une seule fois le rôle du jeu d'apprentissage. Par conséquent, le modèle doit se généraliser correctement pour tous les échantillons du jeu de données de manière à ce que les scores produits par la validation croisée (et leur moyenne) soient élevés.

Avoir de multiples divisions des données fournit également certaines informations sur le niveau de sensibilité de notre modèle par rapport à la sélection du jeu d'apprentissage. Dans le cas du jeu de données `iris`, nous avons noté des niveaux d'exactitude se situant entre 90 % et 100 %. Cet intervalle est assez important, ce qui nous donne une idée sur la manière dont le modèle pourrait se comporter dans le pire ou dans le meilleur des cas lorsqu'il est appliqué à de nouvelles données.

Un autre bénéfice apporté par la validation croisée, par comparaison avec la simple division en deux des données, est que celles-ci sont employées plus efficacement. Lorsque nous utilisons `train_test_split`, nous prenons généralement 75 % des données pour le jeu d'apprentissage, et les 25 % restants pour l'évaluation. Avec une validation croisée à 5 plis, chaque itération se sert des quatre cinquièmes, soit 80 %, des données pour remplir le modèle. Avec une validation croisée à 10 plis, on passerait à 90 % des données pour chaque itération. Plus on utilise de données, et plus le modèle gagne généralement en exactitude.

Le principal désavantage de la validation croisée est l'augmentation des coûts de traitement qu'elle entraîne. Puisque nous devons maintenant réaliser l'entraînement sur *k* modèles au lieu d'un seul, la validation croisée sera à peu près *k* fois plus lente que la simple division des données.



Il est important de garder présent à l'esprit le fait que la validation croisée n'est pas une autre manière de construire un modèle qui pourrait être appliqué à de nouvelles données. Elle ne retourne pas un modèle. Lors de l'appel à `cross_val_score`, de multiples modèles sont construits en interne, mais le but de la validation croisée consiste uniquement à évaluer la manière dont un algorithme donné va se généraliser lorsqu'il sera appliqué à un jeu de données spécifique.

## Validation croisée à *k*-plis stratifiée et autres stratégies

Diviser le jeu de données en *k*-plis en partant de la première des *k*-parties des données, comme cela a été décrit dans la précédente section, peut ne pas toujours être une bonne idée. Reprenons par exemple le jeu de données `iris` :

**In[6]:**

```
from sklearn.datasets import load_iris
iris = load_iris()
print("Iris labels:\n{}".format(iris.target))
```

**Out[6]:**

```
Iris labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Comme vous pouvez le constater, le premier tiers des données appartient à la classe 0, le second tiers à la classe 1 et le troisième tiers à la classe 2. Imaginez alors que vous réalisiez une validation croisée à trois plis sur ce jeu de données. Le premier pli ne correspondrait qu'à la classe 0, puisque les données d'apprentissage (les deux derniers tiers) contiendraient la totalité des classes 1 et 2. Du fait que les classes des jeux de données d'apprentissage et de test seraient différentes lors des trois partages, l'exactitude de cette validation croisée serait nulle pour cet échantillon. Cela ne serait de toute façon pas très pas utile, puisque nous savons qu'il est possible d'obtenir de bien meilleurs résultats sur le jeu `iris`.

Du fait que cette stratégie de validation croisée échoue dans ce cas, `scikit-learn` ne l'utilise pas pour la classification. Il fait alors appel à la *validation croisée à k-plis stratifiée*. Ici, nous divisons les données de manière à ce que la proportion entre les classes soit la même dans chaque pli que dans le jeu de données tout entier, comme l'illustre la Figure 5.2 :

In[7]:

```
mglearn.plots.plot_stratified_cross_validation()
```

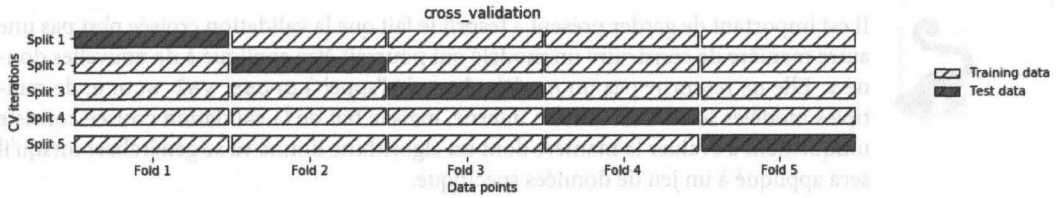



*Figure 5.2 : Comparaison entre la validation croisée standard et la validation croisée stratifiée lorsque les données sont ordonnées selon les étiquettes des classes.*

Par exemple, si 90 % de nos échantillons appartiennent à la classe A, et que les 10 % qui restent appartiennent à la classe B, alors la validation croisée stratifiée s'assurera que chaque pli contient 90 % d'éléments provenant de la classe A et 10 % provenant de la classe B.

Il est généralement préférable d'utiliser une validation croisée à *k*-plis stratifiée, plutôt qu'une simple validation croisée à *k*-plis, pour évaluer un classifieur, car elle fournit une estimation plus fiable des performances attendues d'une généralisation. Dans le cas où seulement 10 % des échantillons appartiendraient à une classe B, une validation croisée à *k*-plis pourrait facilement engendrer un ou plusieurs plis qui ne contiendraient que des données provenant de la classe A. Utiliser alors un tel pli ne donnerait aucune information significative sur les performances globales du classifieur.

Pour une régression, `scikit-learn` utilise par défaut la validation croisée à *k*-plis standard. Il serait aussi possible d'essayer de rendre chaque pli représentatif des différentes valeurs de l'objectif cible, mais cette stratégie n'est pas courante et elle surprendrait la plupart des utilisateurs.

### Plus de contrôle sur la validation croisée

Nous avons vu plus haut qu'il est possible d'ajuster le nombre de plis utilisés par `cross_val_score` en utilisant le paramètre `cv`. Cependant, `scikit-learn` offre un contrôle bien plus fin sur ce qui se passe au cours du processus en fournissant un « séparateur de validation croisée » (*cross-validation splitter*) en tant que paramètre `cv`. Dans la plupart des cas d'utilisation, les traitements par défaut de validation croisée à *k*-plis pour la régression et la validation croisée à *k*-plis stratifiée pour la classification fonctionnent bien, mais il existe certaines situations où vous pourriez vouloir utiliser une stratégie différente. Supposons par exemple que nous voulions employer la validation à *k*-plis standard sur un jeu de données de classification afin de reproduire les résultats obtenus par quelqu'un d'autre. Pour cela, nous devons tout d'abord importer la classe `KFold` (définie en tant que `K-Folds cross-validator`, soit « validateur croisé à *k*-plis », dans la documentation de `scikit-learn`) du module `model_selection` et l'instancier avec le nombre de plis que nous voulons utiliser :

**In[8]:**

```
from sklearn.model_selection import KFold
kfold = KFold(n_splits=5)
```

Nous passons ensuite l'objet `kfold` en tant que paramètre `cv` à `cross_val_score` :

**In[9]:**

```
print("Cross-validation scores:\n{}".format(
      cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

**Out[9]:**

```
Cross-validation scores:
[ 1.     0.933  0.433  0.967  0.433]
```

De cette manière, nous pouvons vérifier qu'utiliser une validation croisée à 3 plis (non stratifiée) sur le jeu de données `iris` est réellement une mauvaise idée :

**In[10]:**

```
kfold = KFold(n_splits=3)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

**Out[10]:**

```
Cross-validation scores:
[ 0.  0.  0.]
```

Souvenez-vous : chaque pli correspond à une des classes du jeu de données `iris`, et il n'est donc pas possible d'apprendre quoi que ce soit. Une autre manière de résoudre ce problème consiste à mélanger les données au lieu de stratifier les plis afin de changer l'ordre des échantillons. Nous pouvons appliquer cette stratégie en définissant le paramètre `shuffle` de `KFold` avec la valeur `True`. Dans ce cas, nous devons aussi spécifier la valeur de `random_state` de manière à obtenir un résultat reproductible. Sinon, chaque exécution de `cross_val_score` fournirait une réponse différente du fait que la division des données changerait à chaque fois (ce qui n'est pas forcément un problème, mais peut surprendre). Mélanger les données avant de les diviser donne de bien meilleurs résultats :

**In[11]:**

```
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

**Out[11]:**

```
Cross-validation scores:
[ 0.9   0.96  0.96]
```

### Validation croisée *leave-one-out*

Une autre méthode de validation croisée fréquemment utilisée est dite *leave-one-out*, ou LOO. Vous pouvez vous figurer cette méthode comme étant une validation croisée à *k*-plis dans laquelle chaque pli est un échantillon unique. Pour chaque division, vous choisissez un point de donnée et un seul afin de constituer le jeu de test. Cela peut être très coûteux en termes de temps de traitement, en particulier pour les jeux de données importants. Par contre, cette technique fournit parfois de meilleurs résultats sur des jeux de données de petite taille :

**In[12]:**

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("Number of cv iterations: ", len(scores))
print("Mean accuracy: {:.2f}".format(scores.mean()))
```

**Out[12]:**

```
Number of cv iterations:  150
Mean accuracy: 0.95
```

### Validation croisée *shuffle-split*

Une autre stratégie très souple de validation croisée est dite *validation croisée shuffle-split* (dite aussi parfois à *échantillonnage répété*). Dans ce cas, on échantillonne de manière aléatoire un nombre prédéfini de données pour le jeu d'apprentissage (*via* la valeur de `train_size`), et un autre pour le jeu de test (*via* la valeur de `test_size`). Cette procédure est répétée `n_splits` fois. La Figure 5.3 illustre l'exécution de cette stratégie sur un jeu comportant dix points de données, avec un jeu d'apprentissage de cinq points et des jeux de test de deux points chacun (vous pouvez utiliser des entiers pour `train_size` et `test_size` afin de définir des valeurs absolues pour ces jeux, ou bien des nombres en virgule flottante afin d'utiliser des fractions de l'ensemble du jeu de données) :

**In[13]:**

```
mglearn.plots.plot_shuffle_split()
```

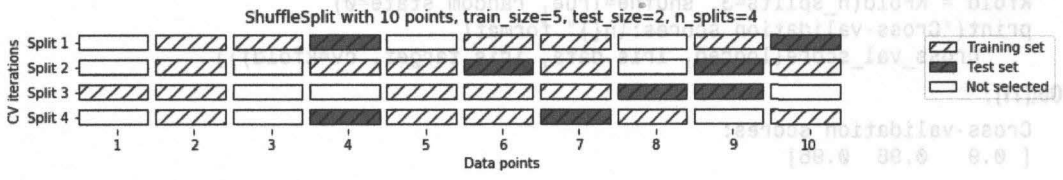



*Figure 5.3 : Exemple d'échantillonnage répété avec 10 points,* `train_size` *= 5,* `test_size` *= 2 et* `n_splits` *= 4.*

Le code qui suit partage le jeu de données en parts égales (50 %) pour l'apprentissage et le test, avec 10 itérations :

**In[14]:**

```
from sklearn.model_selection import ShuffleSplit
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Cross-validation scores:\n{}".format(scores))
```

**Out[14]:**

```
Cross-validation scores:
[ 0.92   0.987  0.92   0.96   0.987  0.947  0.92   0.853  0.88   0.96 ]
```

Cette méthode de validation croisée permet de contrôler le nombre d'itérations indépendamment de la taille des jeux d'apprentissage et de test, ce qui peut parfois être utile. Elle permet également de n'utiliser qu'une partie des données lors de chaque itération, en fournissant pour `train_size` et `test_size` des valeurs qui ne couvrent pas la totalité des échantillons. Cela peut en particulier apporter une certaine aide pour réaliser des expérimentations sur des jeux de données de grande taille.

Il existe également une variante stratifiée de `ShuffleSplit`, judicieusement appelée `StratifiedShuffleSplit`, qui peut fournir des résultats plus fiables pour des tâches de classification.

### Validation croisée avec groupes

Une autre stratégie très courante de validation croisée se présente lorsqu'il existe parmi les données des groupes qui offrent d'importantes similarités. Supposons que vous vouliez construire un système servant à reconnaître des émotions dans des images de visages. Vous collectez un jeu de données de photographies de cent personnes, chacune étant capturée de multiples fois en montrant des émotions différentes. Le but est de produire un classifieur capable d'identifier correctement des émotions pour des personnes qui ne figurent pas dans le jeu de données. Vous pourriez utiliser la validation croisée stratifiée par défaut afin de mesurer les performances d'un classifieur. Cependant, il est probable que les photos de la même personne risquent de se retrouver aussi bien dans le jeu d'apprentissage que dans le jeu de test. Il sera bien plus facile pour un classifieur de détecter des émotions dans un visage qui se trouve déjà dans le jeu d'apprentissage, plutôt que sur un visage totalement nouveau. Pour évaluer avec précision la généralisation à de nouveaux visages, vous devez par conséquent vous assurer que les jeux d'apprentissage et de test contiennent des images de personnes différentes.

Pour obtenir ce résultat, nous pouvons utiliser la classe `GroupKFold`, qui agit sur un tableau de groupes (`groups`) que nous pouvons utiliser pour spécifier quelle personne apparaît sur une image. Le tableau `groups` indique ici des groupes de données qui ne devraient pas être séparées lors de la création des jeux d'apprentissage et de test.

Cet exemple de groupement de données est courant dans les applications médicales, où vous pouvez avoir des multiples échantillons provenant du même patient, mais qu'il est intéressant de généraliser à de nouveaux patients. Dans le cas de la reconnaissance vocale, vous pourriez avoir dans vos échantillons de multiples enregistrements de la même personne, votre objectif étant alors de détecter la voix d'une nouvelle personne.

Le code qui suit est un exemple d'utilisation d'un jeu de données synthétique avec un regroupement déterminé par le tableau `groups`. Le jeu contient 12 points de données. Pour chaque point, le tableau `groups` spécifie à quel groupe (songez par exemple à un patient) ce point appartient. Ici, nous allons utiliser quatre groupes. Les trois premiers échantillons appartiendront au premier groupe, les quatre suivants au second groupe, et ainsi de suite :

**In[15]:**

```
from sklearn.model_selection import GroupKFold
X, y = make_blobs(n_samples=12, random_state=0)
groups = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=3))
print("Cross-validation scores:\n{}".format(scores))
```

**Out[15]:**

```
Cross-validation scores:
[ 0.75   0.8    0.667]
```

Les échantillons n'ont pas besoin d'être triés par groupe. Nous avons suivi cet ordre uniquement pour des raisons de simplicité. Les divisions calculées de cette manière sont illustrées sur la Figure 5.4. Comme vous pouvez le constater, chaque groupe se trouve en totalité soit dans le jeu d'apprentissage, soit dans le jeu de test :

**In[16]:**

```
mglearn.plots.plot_group_kfold()
```

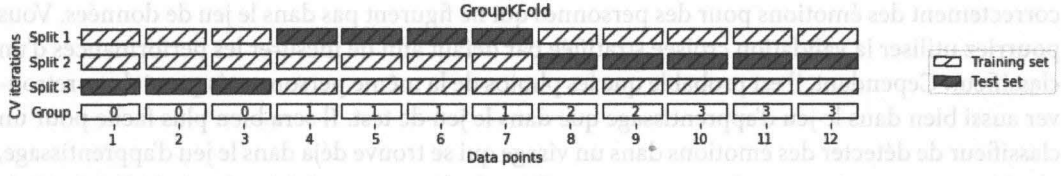



*Figure 5.4 : Validation croisée avec GroupKFold.*

Vous pouvez découvrir dans `scikit-learn` d'autres stratégies de division pour la validation croisée. Voyez ce qui est expliqué à ce sujet dans la documentation de `scikit-learn` (`http://scikit-learn.org/stable/modules/cross_validation.html`). Cependant, les méthodes standard KFold, StratifiedKFold et GroupKFold sont de loin les plus couramment utilisées.

# Recherche sur grille

Maintenant que nous savons comment évaluer la qualité des capacités de généralisation d'un modèle, nous pouvons passer à l'étape suivante et améliorer les performances du modèle en affinant ses paramètres. Nous avons discuté de la configuration des paramètres de nombreux algorithmes de `scikit-learn` dans les Chapitres 2 et 3, et il est important de comprendre le sens de ces paramètres avant d'essayer de les ajuster. Trouver les valeurs des paramètres importants d'un modèle (ceux qui fournissent les meilleures performances de généralisation) est une tâche délicate, mais qui est nécessaire pour pratiquement tous les modèles et tous les jeux de données. Du fait que cette tâche est tellement courante, `scikit-learn` propose à cet effet des méthodes standard. La plus couramment utilisée est dite *grid search* (ou *recherche sur grille*, d'autres équivalents pouvant se rencontrer dans la littérature spécialisée), qui représente d'un point de vue basique le fait d'essayer toutes les combinaisons possibles des paramètres intéressants.

Considérons le cas d'un SVM avec un noyau RBF (*Radial Basis Function*), tel qu'implémenté dans la classe `SVC`. Comme nous l'avons vu au Chapitre 2, il y a ici deux paramètres importants : `gamma` (qui indique l'influence d'un échantillon d'apprentissage sur les autres) et `C`, qui est le paramètre de régularisation. Supposons que nous désirions utiliser la série de valeurs `0.001`, `0.01`, `0.1`, `1`, `10` et `100` pour ces deux paramètres. Du fait que nous voulons essayer six valeurs différentes pour `C` comme pour `gamma`, cela nous donne 36 combinaisons possibles. Regarder toutes ces combinaisons crée un tableau (ou grille) de configuration de paramètres pour le SVM, comme illustré ici :

| | C = 0.001 | C = 0.01 | ... | C = 10 |
|---|---|---|---|---|
| gamma=0.001 | SVC(C=0.001, gamma=0.001) | SVC(C=0.01, gamma=0.001) | ... | SVC(C=10, gamma=0.001) |
| gamma=0.01 | SVC(C=0.001, gamma=0.01) | SVC(C=0.01, gamma=0.01) | ... | SVC(C=10, gamma=0.01) |
| ... | ... | ... | ... | ... |
| gamma=100 | SVC(C=0.001, gamma=100) | SVC(C=0.01, gamma=100) | ... | SVC(C=10, gamma=100) |

## Exemple simple de recherche sur grille

Nous pouvons implémenter un exemple simple sous la forme de boucles `for` pour les deux paramètres, en formant un jeu d'apprentissage et en évaluant un classifieur pour chaque combinaison :

**In[17]:**

```
from sklearn.svm import SVC
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    random_state=0)
print("Size of training set: {}   size of test set: {}".format(
      X_train.shape[0], X_test.shape[0]))
```

```
best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        score = svm.score(X_test, y_test)
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

print("Best score: {:.2f}".format(best_score))
print("Best parameters: {}".format(best_parameters))
```

**Out[17]:**

```
Size of training set: 112   size of test set: 38
Best score: 0.97
Best parameters: {'C': 100, 'gamma': 0.001}
```

## Le danger du surapprentissage des paramètres et du jeu de validation

Partant de ce résultat, nous pourrions être tentés de déclarer que nous avons trouvé un modèle qui atteint une exactitude de 97 % sur notre jeu de données. Cependant, cette proclamation pourrait être excessivement optimiste (ou même tout simplement fausse), et ce pour la raison suivante : nous avons essayé de nombreux paramètres différents, et sélectionné celui qui offre la meilleure exactitude sur le jeu de test, mais celle-ci ne va pas nécessairement valoir pour de nouvelles données. Comme nous avons utilisé les données de test pour ajuster les paramètres, nous ne pouvons plus nous en servir pour évaluer la qualité de notre modèle. C'est en fait la même raison que celle pour laquelle nous avons décomposé nos données en un jeu d'apprentissage et un jeu de test. Nous avons besoin d'un jeu de données indépendant pour réaliser une évaluation correcte, autrement dit un jeu qui n'a pas été utilisé pour créer le modèle.

Une manière de résoudre ce problème consiste à diviser à nouveau les données afin d'avoir trois jeux : les données d'apprentissage pour construire le modèle, un jeu de validation (ou de développement) pour sélectionner les paramètres du modèle, et enfin le jeu de test permettant d'évaluer les performances des paramètres sélectionnés. La Figure 5.5 illustre ce mécanisme :

**In[18]:**

```
mglearn.plots.plot_threefold_split()
```




*Figure 5.5 : Division en trois parties des données pour former un jeu d'apprentissage, un jeu de validation et un jeu de test.*

Après avoir choisi les meilleurs paramètres à l'aide du jeu de validation, nous pouvons reconstruire un modèle en utilisant les paramètres trouvés, mais en effectuant maintenant un entraînement à partir des deux jeux, celui déjà employé pour l'apprentissage et le jeu de validation. De cette manière, nous pouvons utiliser autant de données que possible pour construire notre modèle. Cela nous amène à l'implémentation suivante :

**In[19]:**

```
from sklearn.svm import SVC
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)
print("Size of training set: {}   size of validation set: {}   size of test
  set:"
      " {}\n".format(X_train.shape[0], X_valid.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        score = svm.score(X_valid, y_valid)
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
test_score = svm.score(X_test, y_test)
print("Best score on validation set: {:.2f}".format(best_score))
print("Best parameters: ", best_parameters)
print("Test set score with best parameters: {:.2f}".format(test_score))
```

**Out[19]:**

```
Size of training set: 84   size of validation set: 28   size of test set: 38

Best score on validation set: 0.96
Best parameters:  {'C': 10, 'gamma': 0.001}
Test set score with best parameters: 0.92
```

Le meilleur score obtenu sur la validation est de 96 %. C'est un peu plus bas que précédemment, probablement parce que nous avons utilisé moins de données pour l'apprentissage du modèle (`X_train` est en effet maintenant plus petit puisque nous avons partagé deux fois nos données). Cependant, le score sur le jeu de test (donc celui qui nous indique en réalité la manière dont se passe la généralisation) est encore plus bas à 92 %. Nous pouvons donc seulement prétendre classifier nos nouvelles données avec un taux de réussite de 92 %, et non de 97 % comme auparavant !

La distinction entre jeux d'apprentissage, de validation et de test est fondamentalement importante pour appliquer les méthodes de l'apprentissage automatique de manière concrète. Tous les choix faits en se basant sur l'exactitude du jeu de test font « fuiter » des informations de ce jeu vers le modèle. Il est donc important de conserver un jeu de test séparé, qui sera uniquement utilisé pour l'évaluation finale. Une bonne pratique consiste à effectuer tout le travail exploratoire d'analyse et de sélection du modèle en utilisant la combinaison d'un jeu d'apprentissage et d'un jeu de validation, et de réserver le jeu de test pour une évaluation finale (ce qui vaut également pour tout ce qui est visualisation). Strictement parlant, évaluer plus d'un modèle sur le jeu de test et choisir le meilleur des deux va produire une estimation exagérément optimiste quant à l'exactitude de ce modèle.

## Recherche sur grille avec validation croisée

Bien que la méthode de division des données en trois jeux (apprentissage, validation et test) soit tout à fait réalisable, et même couramment utilisée, elle est assez sensible à la précision avec laquelle le partage des données est effectué. Dans la sortie produite par le dernier morceau de code, nous pouvons constater que la recherche sur grille sélectionne `'C': 10, 'gamma': 0.001` comme meilleurs paramètres, tandis que le code précédent retenait de son côté `'C': 100, 'gamma': 0.001` comme meilleurs paramètres. Pour mieux estimer les performances de généralisation, au lieu d'utiliser une unique division entre jeu d'apprentissage et jeu de validation, nous pouvons faire appel à la validation croisée afin d'évaluer les performances de chaque combinaison de paramètres. Cette méthode peut être codée de la manière suivante :

**In[20]:**

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        svm = SVC(gamma=gamma, C=C)
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        score = np.mean(scores)
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
```

**Out[20]:**

```
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

Pour évaluer l'exactitude du SVM sur un paramétrage particulier de `C` et de `gamma` en utilisant une validation croisée à 5 plis, nous avons besoin d'entraîner 36 * 5 = 180 modèles. Comme vous pouvez l'imaginer, le principal inconvénient de l'emploi de la validation croisée est le temps nécessaire à l'entraînement de tous ces modèles.

Le graphisme de la Figure 5.6 illustre la manière dont la meilleure configuration de paramètres est sélectionnée par le code précédent :

**In[21]:**

```
mglearn.plots.plot_cross_val_selection()
```

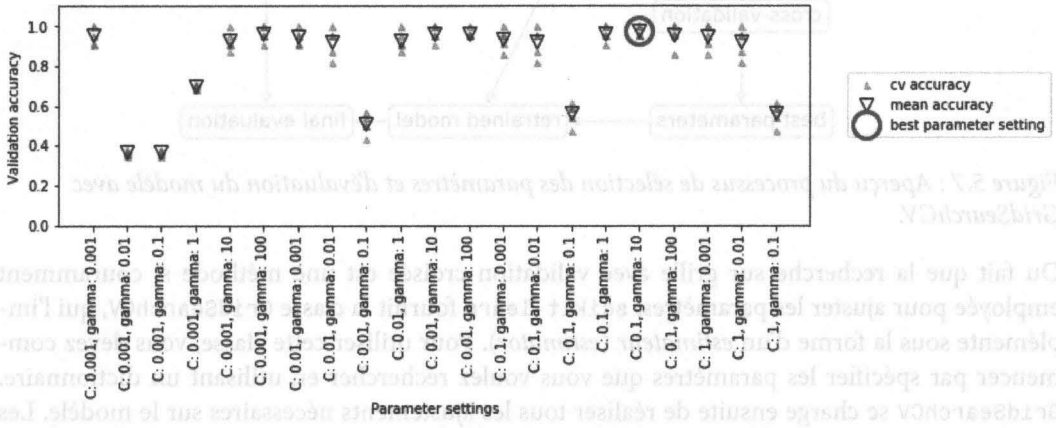



*Figure 5.6 : Résultats obtenus lors d'une recherche sur grille avec validation croisée.*

Pour chaque configuration de paramètres (seule une partie en est montrée ici), cinq valeurs d'exactitude sont calculées, une par division dans la validation croisée. On calcule ensuite la moyenne des valeurs pour chacune de ces configurations. La meilleure combinaison est enfin sélectionnée. Elle est signalée par le cercle sur la figure.



Comme nous l'avons mentionné plus haut, la validation croisée est une manière d'évaluer un certain algorithme sur un jeu de données spécifique. Cependant, elle est souvent utilisée en conjonction avec des méthodes de recherche de paramètres, comme la recherche sur grille. Pour cette raison, de nombreuses personnes utilisent familièrement l'expression *validation croisée* pour faire référence à la recherche sur grille avec validation croisée.

Le processus de division des données, d'exécution de la recherche sur grille et d'évaluation finale des paramètres est illustré sur la Figure 5.7 à l'aide du code ci-après :

**In[22]:**

```
mglearn.plots.plot_grid_search_overview()
```

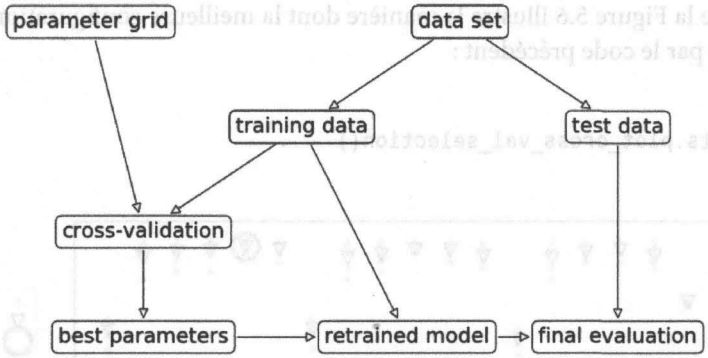



*Figure 5.7 : Aperçu du processus de sélection des paramètres et d'évaluation du modèle avec GridSearchCV.*

Du fait que la recherche sur grille avec validation croisée est une méthode si couramment employée pour ajuster les paramètres, `scikit-learn` fournit la classe `GridSearchCV`, qui l'implémente sous la forme d'un *estimateur* (*estimator*). Pour utiliser cette classe, vous devez commencer par spécifier les paramètres que vous voulez rechercher en utilisant un dictionnaire. `GridSearchCV` se charge ensuite de réaliser tous les ajustements nécessaires sur le modèle. Les clés du dictionnaire sont les noms des paramètres que nous voulons ajuster (dans ce cas, `C` et `gamma`), et ses valeurs sont celles que nous voulons tester. Ainsi, essayer les valeurs `0.001`, `0.01`, `0.1`, `1`, `10` et `100` pour `C` et `gamma` va se traduire par le dictionnaire suivant :

**In[23]:**

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parameter grid:\n{}".format(param_grid))
```

**Out[23]:**

```
Parameter grid:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Nous pouvons maintenant instancier la classe `GridSearchCV` avec le modèle (`SVC`), la grille à parcourir (`param_grid`) et la stratégie de validation croisée que nous voulons utiliser (par exemple, une validation croisée stratifiée à 5 plis) :

**In[24]:**

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

`GridSearchCV` va se servir de la validation croisée à la place de la division des données en un jeu d'apprentissage et un jeu de validation utilisée précédemment. Cependant, nous devons toujours nous charger du découpage des données en un jeu d'apprentissage et un jeu de test pour éviter de provoquer un phénomène de surajustement sur les paramètres :

In[25]:

```
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
```

L'objet `grid_search` créé un peu plus haut se comporte comme un classifieur. Nous pouvons donc appeler sur lui les méthodes standard `fit`, `predict` et `score`. Lorsque nous appelons `fit`, il exécute maintenant une validation croisée pour chaque combinaison de paramètres spécifiée dans `param_grid` :

In[26]:

```
grid_search.fit(X_train, y_train)
```



Un estimateur `scikit-learn` qui est créé en utilisant un autre estimateur est dit un *méta-estimateur* (*meta-estimator*). `GridSearchCV` est le méta-estimateur le plus couramment employé, mais ce n'est pas le seul.

L'objet `GridSearchCV` ne se contente pas de rechercher les meilleurs paramètres. Il produit de surcroît automatiquement un nouveau modèle basé sur l'ensemble du jeu d'apprentissage pour les paramètres qui ont fourni les meilleures performances de validation croisée. Ce qui se passe avec `fit` est donc équivalent au résultat obtenu avec le code du listing In[21] que nous avons vu au début de cette section. La classe `GridSearchCV` propose ainsi une interface très commode pour accéder au modèle réentraîné en utilisant les méthodes `predict` et `score`. Pour évaluer la manière dont les meilleurs paramètres trouvés se généralisent, nous pouvons appeler la méthode `score` sur le jeu de test :

In[27]:

```
print("Test set score: {:.2f}".format(grid_search.score(X_test, y_test)))
```

Out[27]:

```
Test set score: 0.97
```

En choisissant les paramètres avec cette technique de validation croisée, nous avons en fait trouvé un modèle qui atteint 97 % d'exactitude sur le jeu de test. La chose importante ici, c'est que *nous n'avons pas utilisé le jeu de test* pour choisir les paramètres. Ceux qui ont été trouvés sont enregistrés dans l'attribut `best_params_`, et la meilleure exactitude de validation croisée (l'exactitude moyenne des différents partages pour cette configuration de paramètres) est enregistrée dans `best_score_` :

In[28]:

```
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

Out[28]:

```
Best parameters: {'C': 100, 'gamma': 0.01}
Best cross-validation score: 0.97
```



Ne confondez pas `best_score_` avec les performances de généralisation du modèle telles qu'elles sont calculées par la méthode `score` appliquée au jeu de test. Utiliser la méthode `score` (ou évaluer la sortie de la méthode `predict`) emploie un modèle *entraîné à partir de l'ensemble du jeu d'apprentissage*. L'attribut `best_score_` mémorise l'exactitude de validation croisée moyenne, avec *une validation croisée effectuée sur le jeu d'apprentissage*.

Il est parfois utile de pouvoir accéder au modèle réel qui a été trouvé, par exemple pour vérifier certains coefficients ou certaines importances des caractéristiques. Vous pouvez accéder au modèle correspondant aux meilleurs paramètres obtenus sur la totalité du jeu d'apprentissage en utilisant l'attribut `best_estimator_` :

**In[29]:**

```
print("Best estimator:\n{}".format(grid_search.best_estimator_))
```

**Out[29]:**

```
Best estimator:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

Du fait que `grid_search` lui-même possède des méthodes `predict` et `score`, utiliser `best_estimator_` n'est pas nécessaire pour effectuer des prédictions ou évaluer le modèle.

### Analyser le résultat de la validation croisée

Il est souvent utile de visualiser les résultats d'une validation croisée pour comprendre en quoi la généralisation du modèle dépend des paramètres que nous recherchons. Du fait que les recherches sur grille sont potentiellement coûteuses en termes de calculs, commencer par une grille un peu grossière et de petite taille est souvent une bonne idée. Nous pouvons alors inspecter les résultats obtenus, et voir s'il est intéressant d'étendre notre recherche. Les résultats d'une recherche sur grille peuvent être trouvés dans l'attribut `cv_results_`, qui est un dictionnaire enregistrant tous les aspects de la recherche. Il contient de nombreux détails, ainsi que vous pouvez le constater sur la sortie suivante, et il est plus facile de le consulter après l'avoir converti en un `DataFrame` pandas :

**In[30]:**

```
import pandas as pd
results = pd.DataFrame(grid_search.cv_results_)
display(results.head())
```

**Out[30]:**

| | param_C | param_gamma | params | mean_test_score |
|---|---|---|---|---|
| 0 | 0.001 | 0.001 | {'C': 0.001, 'gamma': 0.001} | 0.366 |
| 1 | 0.001 | 0.01 | {'C': 0.001, 'gamma': 0.01} | 0.366 |
| 2 | 0.001 | 0.1 | {'C': 0.001, 'gamma': 0.1} | 0.366 |

```
3   0.001        1          {'C': 0.001, 'gamma': 1}       0.366
4   0.001       10          {'C': 0.001, 'gamma': 100}     0.366

   rank_test_score   split0_test_score   split1_test_score   split2_test_score
0               22               0.375               0.347               0.363
1               22               0.375               0.347               0.363
2               22               0.375               0.347               0.363
3               22               0.375               0.347               0.363
4               22               0.375               0.347               0.363

    split3_test_score  split4_test_score   std_test_score
0               0.363              0.380            0.011
1               0.363              0.380            0.011
2               0.363              0.380            0.011
3               0.363              0.380            0.011
4               0.363              0.380            0.011
```

Chaque ligne de `results` correspond à une combinaison de paramètres particulière. À chaque fois, les résultats de toutes les divisions formées lors de la validation croisée sont enregistrés, ainsi que la moyenne et la déviation standard correspondantes. Comme nous effectuons une recherche dans une grille bidimensionnelle de paramètres (`C` et `gamma`), une visualisation sous forme de carte de chaleur est tout à fait appropriée (voir la Figure 5.8). Nous commençons par extraire les scores de validation moyens, puis nous réorganisons ces scores pour que les axes correspondent à `C` et à `gamma` :

**In[31]:**

```
scores = np.array(results.mean_test_score).reshape(6, 6)

mglearn.tools.heatmap(scores, xlabel='gamma', xticklabels=param_grid['gamma'],
                      ylabel='C', yticklabels=param_grid['C'], cmap="viridis")
```

Chaque point de la carte correspond à une exécution de la validation croisée avec une configuration particulière de paramètres. L'intensité de la couleur (ou plutôt ici du niveau de gris) décrit l'exactitude de la validation croisée. Les teintes les plus claires correspondent aux niveaux d'exactitude les plus élevés, et les teintes les plus sombres aux taux les plus bas. Vous pouvez constater que SVC est très sensible à la configuration des paramètres. Dans de nombreux cas, l'exactitude se situe autour de 40 %, ce qui est plutôt mauvais. Pour d'autres valeurs, on atteint 96 %, ce qui est évidemment bien meilleur. Nous pouvons déduire plusieurs choses de ce graphique. Tout d'abord, les paramètres que nous avons ajustés sont *très importants* pour obtenir de bonnes performances. `C` et `gamma` jouent un rôle essentiel, puisque le fait de les ajuster peut faire passer le degré d'exactitude de 40 % à 96 %. De plus, les plages choisies pour les paramètres se traduisent par des modifications substantielles dans le résultat. Il est également important de noter que ces plages sont suffisamment étendues : les valeurs optimales pour chaque paramètre ne se trouvent pas sur les bords du tracé.

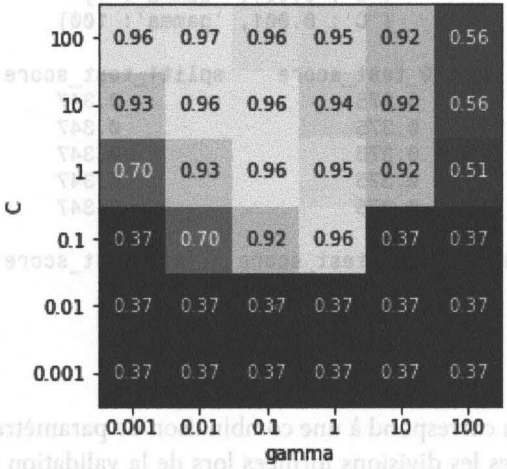



*Figure 5.8 : Carte de chaleur des scores moyens de validation croisée en fonction de* `C` *et de* `gamma`.

Regardez maintenant la Figure 5.9. Elle nous montre un comportement différent, avec un résultat bien moins bon, car les plages n'ont pas été correctement choisies pour la recherche :

**In[32] :**

```
fig, axes = plt.subplots(1, 3, figsize=(13, 5))

param_grid_linear = {'C': np.linspace(1, 2, 6),
                     'gamma':  np.linspace(1, 2, 6)}

param_grid_one_log = {'C': np.linspace(1, 2, 6),
                      'gamma':  np.logspace(-3, 2, 6)}

param_grid_range = {'C': np.logspace(-3, 2, 6),
                    'gamma':  np.logspace(-7, -2, 6)}

for param_grid, ax in zip([param_grid_linear, param_grid_one_log,
                           param_grid_range], axes):
    grid_search = GridSearchCV(SVC(), param_grid, cv=5)
    grid_search.fit(X_train, y_train)
    scores = grid_search.cv_results_['mean_test_score'].reshape(6, 6)

    scores_image = mglearn.tools.heatmap(
        scores, xlabel='gamma', ylabel='C', xticklabels=param_grid['gamma'],
        yticklabels=param_grid['C'], cmap="viridis", ax=ax)

plt.colorbar(scores_image, ax=axes.tolist())
```

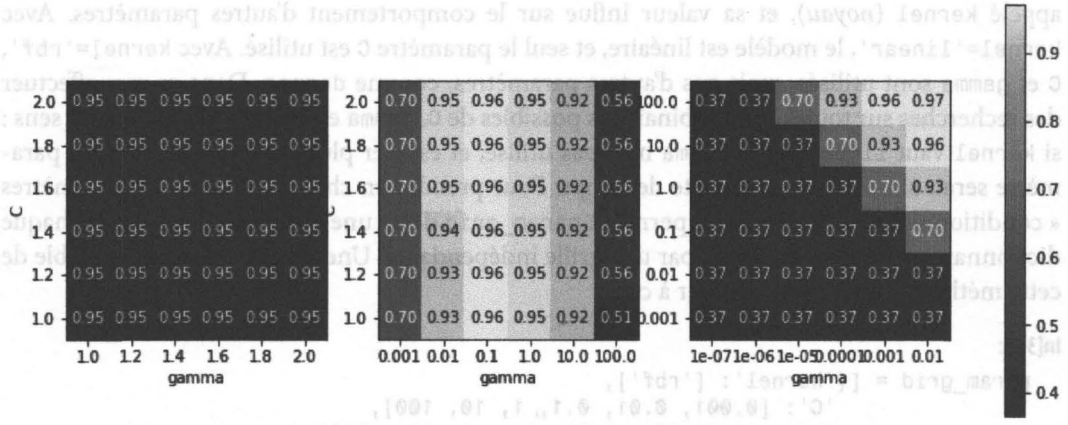



*Figure 5.9 : Cartes de chaleur correspondant à des grilles de recherche mal spécifiées.*

Sur la première carte, toutes les valeurs sont identiques et représentées par la même intensité de gris. Ici, cela signifie que l'échelle et l'étendue de la plage choisie pour les paramètres `C` et `gamma` sont incorrectes. Cependant, une telle situation pourrait également indiquer qu'au moins un des paramètres n'a en réalité aucune importance. Il est généralement bon d'essayer en premier des valeurs extrêmes afin de voir si le fait de changer un paramètre se traduit par des modifications dans l'exactitude.

La seconde carte montre une disposition qui forme des bandes verticales. Cela indique que seules les valeurs affectées au paramètre `gamma` produisent des différences Cela pourrait signifier que ces valeurs sont effectivement intéressantes, mais pas celles du paramètre `C`, ou bien que celui-ci n'est pas important.

Sur la troisième carte, les valeurs de `C` et de `gamma` sont modifiées. Cependant, vous pouvez constater qu'il ne se passe rien d'intéressant dans le coin inférieur gauche du tracé. Nous pouvons donc vraisemblablement exclure les très petites valeurs pour de futures recherches. Du fait que les résultats optimaux se trouvent sur le bord de la carte, il est aussi possible de penser qu'il pourrait y avoir des valeurs encore meilleures au-delà, et donc nous pourrions modifier en ce sens notre plage de recherche afin d'y inclure davantage de paramètres dans cette région.

Affiner la grille de paramètres en se basant sur les scores de la validation croisée est une excellente démarche, et un bon moyen pour explorer l'importance de divers paramètres. Cependant, vous ne devriez pas tester différentes plages de paramètres sur le jeu de test final. Comme cela a déjà été indiqué plus haut, l'évaluation du jeu de test ne devrait intervenir que lorsque nous savons exactement quel modèle nous voulons utiliser.

### Effectuer des recherches sur des espaces qui ne sont pas des grilles

Dans certains cas, essayer toutes les combinaisons possibles de tous les paramètres, ce que fait généralement `GridSearchCV`, n'est pas une bonne idée. Par exemple, SVC a un paramètre

appelé `kernel` (*noyau*), et sa valeur influe sur le comportement d'autres paramètres. Avec `kernel='linear'`, le modèle est linéaire, et seul le paramètre `C` est utilisé. Avec `kernel='rbf'`, `C` et `gamma` sont utilisés, mais pas d'autres paramètres, comme `degree`. Dans ce cas, effectuer des recherches sur toutes les combinaisons possibles de `C`, `gamma` et `kernel` n'aurait aucun sens : si `kernel` vaut `linear`, alors `gamma` n'est pas utilisé, et essayer plusieurs valeurs pour ce paramètre serait simplement une perte de temps. Pour prendre en charge ces types de paramètres « conditionnels », `GridSearchCV` permet à `param_grid` d'être une *liste de dictionnaires*. Chaque dictionnaire de la liste se traduit par une grille indépendante. Une mise en pratique possible de cette méthode pourrait ressembler à ceci :

**In[33]:**

```
param_grid = [{'kernel': ['rbf'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100],
               'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
              {'kernel': ['linear'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
print("List of grids:\n{}".format(param_grid))
```

**Out[33]:**

```
List of grids:
[{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001,
  0.01, 0.1, 1, 10, 100]}, {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1,
  10, 100]}]
```

Dans la première grille, le paramètre `kernel` est toujours défini avec la valeur `'rbf'` (notez que l'entrée pour `kernel` est une liste de longueur 1). Dans ce cas, on fait varier les deux paramètres `C` et `gamma`. Dans la seconde grille, le paramètre `kernel` est cette fois défini avec la valeur `'linear'`, et seul `C` varie. Appliquons maintenant cette recherche plus complexe :

**In[34]:**

```
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

**Out[34]:**

```
Best parameters: {'C': 100, 'gamma': 0.01, 'kernel': 'rbf'}
Best cross-validation score: 0.97
```

Regardons alors les résultats enregistrés dans `cv_results_`. Comme attendu, seul `C` varie si `kernel` vaut `'linear'` :

**In[35]:**

```
results = pd.DataFrame(grid_search.cv_results_)
display(results.T)
```

**Out[35] :**

| | 0 | 1 | 2 | 3 | ... | 38 | 39 | 40 | 41 |
|---|---|---|---|---|---|---|---|---|---|
| param_C | 0.001 | 0.001 | 0.001 | 0.001 | ... | 0.1 | 1 | 10 | 100 |
| param_gamma | 0.001 | 0.01 | 0.1 | 1 | ... | NaN | NaN | NaN | NaN |
| param_kernel | rbf | rbf | rbf | rbf | ... | linear | linear | linear | linear |
| params | {C: 0.001, kernel: rbf, gamma: 0.001} | {C: 0.001, kernel: rbf, gamma: 0.01} | {C: 0.001, kernel: rbf, gamma: 0.1} | {C: 0.001, kernel: rbf, gamma: 1} | ... | {C: 0.1, kernel: linear} | {C: 1, kernel: linear} | {C: 10, kernel: linear} | {C: 100, kernel: linear} |
| mean_test_score | 0.37 | 0.37 | 0.37 | 0.37 | ... | 0.95 | 0.97 | 0.96 | 0.96 |
| rank_test_score | 27 | 27 | 27 | 27 | ... | 11 | 1 | 3 | 3 |
| split0_test_score | 0.38 | 0.38 | 0.38 | 0.38 | ... | 0.96 | 1 | 0.96 | 0.96 |
| split1_test_score | 0.35 | 0.35 | 0.35 | 0.35 | ... | 0.91 | 0.96 | 1 | 1 |
| split2_test_score | 0.36 | 0.36 | 0.36 | 0.36 | ... | 1 | 1 | 1 | 1 |
| split3_test_score | 0.36 | 0.36 | 0.36 | 0.36 | ... | 0.91 | 0.95 | 0.91 | 0.91 |
| split4_test_score | 0.38 | 0.38 | 0.38 | 0.38 | ... | 0.95 | 0.95 | 0.95 | 0.95 |
| std_test_score | 0.011 | 0.011 | 0.011 | 0.011 | ... | 0.033 | 0.022 | 0.034 | 0.034 |

### Utiliser différentes stratégies de validation croisée avec la recherche sur grille

De manière semblable à `cross_val_score`, `GridSearchCV` utilise par défaut une validation croisée stratifiée à *k*-plis pour la classification, et une validation croisée à *k*-plis pour la régression. Cependant, nous avons vu dans la section « Plus de contrôle sur la validation croisée » que vous pouviez redéfinir le paramètre `cv` de `GridSearchCV`. En particulier, pour obtenir uniquement une simple division en un jeu d'apprentissage et un jeu de validation, vous pouvez vous servir de `ShuffleSplit` ou de `StratifiedShuffleSplit` avec `n_splits` égal à 1. Cela pourrait être utile pour des jeux de données très grands, ou des modèles très lents.

### Validation croisée imbriquée

Dans les exemples précédents, nous sommes partis d'un découpage entre jeux d'apprentissage, de validation et de test pour obtenir une division des données entre jeux d'apprentissage et de test, puis pour effectuer une validation croisée sur le jeu d'apprentissage. Mais, en utilisant `GridSearchCV` comme nous l'avons décrit ci-dessus, nous n'avons toujours au final qu'une seule partition des données en un jeu d'apprentissage et un jeu de test, ce qui peut rendre nos résultats instables et nous rendre trop dépendant de cette division simple des données. Nous pouvons pousser les choses plus loin, en effectuant de nouvelles séparations entre jeux d'apprentissage et de test au sein d'autres validations croisées. C'est ce que l'on appelle une *validation croisée imbriquée*. Dans ce cas, il y a une boucle externe qui vient « envelopper » les divisions de données

entre jeux d'apprentissage et jeux de test. Pour chaque configuration, une recherche sur grille est exécutée (ce qui pourrait se traduire par différentes configurations de meilleurs paramètres dans la boucle extérieure). Pour chaque passage dans la boucle extérieure, le score obtenu par le jeu de test en utilisant les meilleurs réglages est alors enregistré.

Le résultat de cette procédure est une liste de scores, et non un modèle ou une configuration de paramètres. Ces scores nous informent sur la capacité de généralisation d'un modèle, et ce en partant des meilleurs paramètres trouvés par la grille. Comme le résultat n'est pas un modèle qui pourrait être utilisé sur de nouvelles données, la validation croisée imbriquée est rarement employée pour la recherche d'un modèle prédictif susceptible d'être appliqué à de futurs échantillons. Cependant, cette méthode peut être utile pour évaluer la manière dont un certain modèle fonctionne sur un jeu de données particulier.

Implémenter une stratégie de validation croisée imbriquée avec `scikit-learn` est simple. Nous appelons `cross_val_score` avec une instance de `GridSearchCV` en tant que modèle :

**In[36]:**

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                         iris.data, iris.target, cv=5)
print("Cross-validation scores: ", scores)
print("Mean cross-validation score: ", scores.mean())
```

**Out[36]:**

```
Cross-validation scores:  [ 0.967  1.     0.967  0.967  1.   ]
Mean cross-validation score:  0.98
```

Le résultat de notre validation croisée imbriquée pourrait être résumé ainsi : « SVC peut atteindre une exactitude de validation croisée moyenne de 98 % sur le jeu de données `iris` ». Rien de plus, et rien de moins.

Ici, nous avons utilisé une validation croisée stratifiée à 5 plis aussi bien dans la boucle intérieure que dans la boucle extérieure. Comme notre grille `param_grid` contient 36 combinaisons de paramètres, cela génère une énorme quantité de modèles à construire (précisément 36 * 5 * 5, soit 900 modèles), ce qui rend la validation croisée imbriquée extrêmement lourde en termes de traitements. Dans cet exemple, nous avons défini la même méthode de validation croisée pour la boucle intérieure et la boucle extérieure. Mais ce n'est absolument pas nécessaire, et vous pouvez choisir une tout autre stratégie de validation croisée. Il peut cependant être assez difficile de comprendre ce qui se passe dans la ligne de code donnée ci-dessus, et il est alors utile de visualiser le traitement sous la forme de boucles `for`, comme dans l'implémentation simplifiée suivante :

**In[37]:**

```
def nested_cv(X, y, inner_cv, outer_cv, Classifier, parameter_grid):
    outer_scores = []
    for training_samples, test_samples in outer_cv.split(X, y):
        best_parms = {}
        best_score = -np.inf
        for parameters in parameter_grid:
            cv_scores = []
            for inner_train, inner_test in inner_cv.split(
                    X[training_samples], y[training_samples]):
                clf = Classifier(**parameters)
                clf.fit(X[inner_train], y[inner_train])
                score = clf.score(X[inner_test], y[inner_test])
                cv_scores.append(score)
            mean_score = np.mean(cv_scores)
            if mean_score > best_score:
                best_score = mean_score
                best_params = parameters
        clf = Classifier(**best_params)
        clf.fit(X[training_samples], y[training_samples])
        outer_scores.append(clf.score(X[test_samples], y[test_samples]))
    return np.array(outer_scores)
```

Exécutons maintenant cette fonction sur le jeu de données `iris` :

**In[38]:**

```
from sklearn.model_selection import ParameterGrid, StratifiedKFold
scores = nested_cv(iris.data, iris.target, StratifiedKFold(5),
                   StratifiedKFold(5), SVC, ParameterGrid(param_grid))
print("Cross-validation scores: {}".format(scores))
```

**Out[38]:**

```
Cross-validation scores: [ 0.967  1.     0.967  0.967  1.   ]
```

### Paralléliser validation croisée et recherche sur grille

Exécuter une recherche sur grille avec de nombreux paramètres et sur de larges jeux de données peut être extrêmement gourmand en termes de traitement informatique. Mais, en même temps, c'est aussi quelque chose que l'on dit être *embarrassingly parallel*. Cela signifie que construire un modèle utilisant un paramétrage particulier sur une certaine configuration de validation croisée peut être réalisé de manière complètement indépendante par rapport aux autres modèles et configurations de paramètres. Cela fait des stratégies de validation croisée et de recherche sur grille des candidates idéales pour la parallélisation sur une série de processeurs multicœurs ou sur un cluster d'ordinateurs. Vous pouvez faire appel à de multiples cœurs dans `GridSearchCV` et `cross_val_score` en affectant au paramètre `n_jobs` le nombre de cœurs de CPU que vous voulez utiliser. Pour mobiliser la totalité des cœurs disponibles, définissez `n_jobs` à la valeur -1.

Vous devez prendre en considération le fait que `scikit-learn` *ne permet pas l'imbrication d'opérations parallèles*. Par conséquent, si vous utilisez l'option `n_jobs` pour votre modèle (par exemple, une forêt aléatoire), vous ne pouvez pas l'employer dans `GridSearchCV` pour effectuer des recherches sur ce modèle. Si votre jeu de données et votre modèle sont très gros, utiliser de nombreux cœurs pourrait entraîner une consommation trop importante de mémoire. Vous devriez donc surveiller ce problème lorsque vous construisez des modèles importants en parallèle.

Il est aussi possible de paralléliser des recherches sur grille et des validations croisées dans un cluster de machines, bien que `scikit-learn` ne supporte pas ce type de traitement au moment où ce livre est écrit. Il est cependant possible d'utiliser l'environnement de développement de IPython pour réaliser des recherches sur grille en parallèle. Enfin, les utilisateurs de Spark peuvent se tourner vers le package `spark-sklearn`, qui permet d'exécuter une recherche sur grille sur un cluster Spark déjà constitué.

# Métriques d'évaluation et scoring

Jusqu'ici, nous avons évalué les performances de classification en utilisant l'exactitude (la fraction des échantillons correctement classifiés), et les performances de régression en utilisant $R^2$. Cependant, il s'agit uniquement de deux des multiples façons de juger du comportement d'un modèle supervisé sur un jeu de données particulier. En pratique, il est possible que ces métriques d'évaluation ne soient pas appropriées pour votre application, et il est important de choisir la bonne métrique lors de la sélection entre modèles et ajustement des paramètres.

## Ne jamais oublier le but final

Lorsque vous sélectionnez une métrique, vous devriez toujours garder présent à l'esprit le but final recherché dans l'application d'apprentissage automatique. En pratique, vous n'êtes généralement pas seulement intéressé par le fait de réaliser des prédictions exactes, mais aussi d'utiliser ces prédictions dans le cadre d'un processus décisionnel plus large. Avant de choisir une métrique d'évaluation, vous devriez réfléchir au but principal, de plus haut niveau, de l'application, souvent appelé la métrique « business ». Les conséquences induites par le fait de choisir un algorithme particulier pour une application d'apprentissage automatique sont également parfois appelées *impact business* (mais, quelle que soit l'expression utilisée, l'essentiel est de ne jamais perdre de vue le but final). Par exemple, l'objectif principal peut être d'éviter des accidents de la circulation, ou bien de diminuer le nombre d'admissions à l'hôpital. Mais il peut aussi s'agir de gagner plus d'utilisateurs pour votre site Web, ou encore de faire dépenser davantage d'argent par vos clients dans votre boutique. Lorsque vous choisissez un modèle ou que vous ajustez des paramètres, vous devriez sélectionner les valeurs qui ont l'influence la plus positive sur cette métrique. C'est souvent difficile, car déterminer l'impact d'un modèle particulier peut impliquer le fait de le mettre en production dans un système réel.

Dans les premiers stades du développement, et pour ajuster des paramètres, il est souvent impossible de mettre des modèles en production simplement dans un but de test, car cela pourrait engendrer des risques considérables, aussi bien matériels qu'humains. Imaginez par exemple la question de l'évaluation des capacités d'évitement des piétons par un véhicule autonome. Si vous laissez simplement le véhicule se promener sans avoir effectué par avance toutes les vérifications indispensables, les piétons risquent d'avoir des tas d'ennuis ! C'est pourquoi nous avons souvent besoin de trouver une procédure de substitution, en utilisant une métrique d'évaluation qui soit plus simple à traiter. Par exemple, nous pourrions tester la classification d'images de piétons et de non-piétons, et mesurer l'exactitude des résultats obtenus. Mais n'oubliez pas qu'il s'agit uniquement d'une procédure de substitution, et qu'il est clairement préférable de trouver la métrique qui se rapproche le plus de l'objectif original et dont l'évaluation soit réalisable. Cette métrique devrait être utilisée chaque fois que possible pour l'évaluation et la sélection du modèle. Le résultat de cette évaluation n'est pas forcément un nombre unique (par exemple, la conséquence de votre algorithme pourrait être que vous gagnez 10 % de clients, mais que chaque client va dépenser 15 % de moins), mais il devrait en tout état de cause capturer l'impact attendu du choix d'un modèle plutôt qu'un autre.

Dans cette section, nous allons commencer par nous intéresser au cas important de la classification binaire, avant de nous pencher sur la classification multiclasse et finalement sur la régression.

## Métriques d'évaluation pour la classification binaire

La classification binaire est presque sans aucun doute l'application la plus courante et la plus simple conceptuellement de l'apprentissage automatique d'un point de vue pratique. Cependant, évaluer cette tâche, même simple, implique certaines mises en garde. Avant de se plonger dans des métriques alternatives, il est utile de regarder en quoi la manière dont l'exactitude est mesurée peut être trompeuse. N'oubliez pas que, dans le cas d'une classification binaire, nous parlons souvent d'une classe *positive* et d'une classe *négative*, avec comme schéma de compréhension le fait que la classe positive est celle que nous recherchons.

### Types d'erreurs

Souvent, l'exactitude n'est pas une bonne mesure quant aux performances de prédiction, du fait que le nombre d'erreurs que nous pouvons commettre ne contient pas toutes les informations qui nous intéressent. Imaginez ainsi une application dédiée à la détection précoce d'un cancer en utilisant un test automatisé. Si le test est négatif, le patient ou la patiente est réputé(e) être en bonne santé. S'il est positif, il ou elle va devoir subir une autre batterie de tests. Ici, un test positif (une indication d'un possible cancer) correspond donc à la classe positive, et un test négatif à la casse négative. Nous ne pouvons pas supposer que notre modèle va toujours fonctionner de

façon parfaite, et il va donc commettre des erreurs. Dans une véritable application, nous devons nous demander quelles conséquences ces erreurs pourraient avoir dans le monde réel.

Une des erreurs possibles, c'est qu'un patient en bonne santé soit classifié comme étant positif, ce qui amène à des analyses supplémentaires. Tout cela induira un coût inutile, ainsi qu'une gêne pour le patient (sans même parler d'un important désordre mental). Une prédiction positive incorrecte est appelée un *faux positif*. L'autre erreur possible est qu'un patient malade soit classé dans la catégorie négative, et ne reçoive donc aucun traitement ou test supplémentaire. Le cancer non diagnostiqué pourrait donc se traduire par de sérieux problèmes de santé, voire même avoir des conséquences fatales. Une erreur de ce type (une prédiction négative incorrecte) est appelée un *faux négatif*. Dans le domaine des statistiques, un faux positif est également dit *erreur de type I*, et un faux négatif *erreur de type II*. Nous retenons les termes « faux négatif » et « faux positif » car ils sont plus explicites et plus faciles à mémoriser. Dans le cas du diagnostic d'un cancer, il est évident que nous voulons éviter les faux négatifs autant que possible, tandis que les faux positifs seront perçus comme une moindre nuisance.

Même si cet exemple est particulièrement radical, les conséquences des faux positifs et des faux négatifs sont rarement les mêmes. Dans le cas d'applications commerciales, il serait possible d'assigner des valeurs en dollars (ou en euros) aux deux types d'erreurs, ce qui conduirait à mesurer l'erreur sur une prédiction particulière en termes de monnaie et non d'exactitude. Cela pourrait être plus significatif pour prendre des décisions d'ordre opérationnel quant au modèle à utiliser.

### Jeux de données déséquilibrés

Les types d'erreurs jouent un rôle important lorsque l'une de deux classes est plus fréquente que l'autre. En réalité, cette situation est très courante. Un bon exemple est représenté par la prédiction dite *click-through*, dans laquelle chaque point de donnée représente une « impression », en d'autres termes un élément qui a été présenté à un utilisateur. Il pourrait s'agir d'une publicité, d'une certaine histoire personnelle, ou encore d'un(e) « ami(e) » à suivre sur un réseau social. Le but consiste alors à prédire si, en montrant un certain élément, un utilisateur va cliquer dessus (indiquant ainsi qu'il est intéressé). Dans la plupart des cas, ce que l'utilisateur voit sur l'Internet (en l'occurrence et en particulier des publicités) ne va pas conduire qui que ce soit à spontanément cliquer. Vous devez donc montrer à un certain utilisateur 100 publicités ou articles avant que la personne trouve quelque chose d'intéressant sur lequel cliquer. Vous vous retrouvez alors avec un jeu de données comportant 99 % de points « sans clic » et 1 % « avec clic ». En d'autres termes, 99 % des échantillons appartiennent à la classe « pas de clic ». Les jeux de données pour lesquels une classe est bien plus fréquente qu'une autre sont souvent dits *déséquilibrés*. En fait, il s'agit de la norme, et rares sont les événements pour lesquels on trouve dans les données des fréquences égales ou même similaires.

Supposons maintenant que vous construisez un classifieur qui offre une exactitude de 99 % sur la tâche de prédiction du clic. Qu'est-ce que cela nous apprend ? Bien sûr, une exactitude de

99 % semble être extrêmement impressionnante, mais cela ne prend pas en compte la question du déséquilibre. Vous pouvez atteindre une exactitude de 99 % sans construire un modèle d'apprentissage automatique, et cela en effectuant toujours une prédiction « pas de clic ». D'un autre côté, même avec des données déséquilibrées, un modèle avec une exactitude de 99 % pourrait être de bonne qualité. Pour autant, cette exactitude ne nous permet pas de distinguer le modèle « pas de clic » constant d'un modèle potentiellement bon.

Pour illustrer cela, nous allons créer un jeu déséquilibré « 9:1 » depuis le jeu de données `digits` en classifiant le chiffre 9 contre les neuf autres classes :

**In[39]:**

```
from sklearn.datasets import load_digits

digits = load_digits()
y = digits.target == 9

X_train, X_test, y_train, y_test = train_test_split(
    digits.data, y, random_state=0)
```

Nous pouvons utiliser le classifieur DummyClassifier pour prédire la classe la plus fréquente (soit ici « non neuf ») afin de voir combien l'exactitude peut ne pas être une donnée informative :

**In[40]:**

```
from sklearn.dummy import DummyClassifier
dummy_majority = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
pred_most_frequent = dummy_majority.predict(X_test)
print("Unique predicted labels: {}".format(np.unique(pred_most_frequent)))
print("Test score: {:.2f}".format(dummy_majority.score(X_test, y_test)))
```

**Out[40]:**

```
Unique predicted labels: [False]
Test score: 0.90
```

Nous obtenons près de 90 % d'exactitude sans aucun apprentissage. Cela peut paraître assez frappant, mais réfléchissez-y une minute. Imaginez que quelqu'un vous explique que son modèle offre une exactitude de 90 %. Vous pourriez alors vous dire qu'il a fait un très bon travail. Mais, pour autant, il se pourrait aussi que, selon le problème posé, la prédiction ne concerne qu'une seule classe ! Comparons cela avec un vrai classifieur :

**In[41]:**

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)
pred_tree = tree.predict(X_test)
print("Test score: {:.2f}".format(tree.score(X_test, y_test)))
```

**Out[41]:**

```
Test score: 0.92
```

Le résultat obtenu avec `DecisionTreeClassifier` est seulement légèrement meilleur que dans l'exemple précédent. Cela pourrait indiquer que quelque chose ne va pas dans la manière dont nous avons utilisé DecisionTreeClassifier, ou bien que l'exactitude n'est en fait pas une bonne mesure ici.

Pour des raisons de comparaison, évaluons deux classifieurs de plus, LogisticRegression et DummyClassifier. Ce dernier effectue des prédictions en suivant des règles simples (et donc évidemment à n'utiliser que pour ce genre de comparaison) mais produit par défaut des classes selon les mêmes proportions que dans le jeu d'apprentissage :

**In[42]:**

```
from sklearn.linear_model import LogisticRegression

dummy = DummyClassifier().fit(X_train, y_train)
pred_dummy = dummy.predict(X_test)
print("dummy score: {:.2f}".format(dummy.score(X_test, y_test)))

logreg = LogisticRegression(C=0.1).fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)
print("logreg score: {:.2f}".format(logreg.score(X_test, y_test)))
```

**Out[42]:**

```
dummy score: 0.82
logreg score: 0.98
```

Le classifieur DummyClassifier, qui produit une sortie aléatoire, est clairement le pire du lot (du point de vue de l'exactitude), tandis que `LogisticRegression` donne de très bons résultats. Cependant, même le classifieur aléatoire offre une exactitude supérieure à 80 %. Il est donc très difficile de dire lequel de ces résultats est véritablement utile. Le problème ici, c'est que l'exactitude est une mesure inadéquate pour quantifier les performances prédictives dans cette configuration déséquilibrée. Pour la suite de ce chapitre, nous allons explorer des métriques alternatives qui fournissent un meilleur guidage pour la sélection de modèles. En particulier, nous aimerions avoir des métriques qui nous disent à quel point un modèle est meilleur que juste faire des prédictions aléatoires ou du genre « plus fréquent », telles qu'elles sont calculées dans `pred_dummy` et `pred_most_frequent`. Si nous utilisons une métrique pour évaluer nos modèles, elle devrait définitivement être capable d'éliminer ces prédictions absurdes.

### Matrices de confusion

L'une des manières les plus complètes pour représenter le résultat de l'évaluation d'une classification binaire consiste à utiliser des matrices de confusion. Inspectons par exemple les prédictions de `LogisticRegression` (reportez-vous à la section précédente) en utilisant la fonction `confusion_matrix`. Nous avons déjà enregistré les prédictions sur le jeu de test dans `pred_logreg` :

**In[43]:**

```
from sklearn.metrics import confusion_matrix

confusion = confusion_matrix(y_test, pred_logreg)
print("Confusion matrix:\n{}".format(confusion))
```

**Out[43]:**

```
Confusion matrix:
[[401   2]
 [  8  39]]
```

La sortie produite par `confusion_matrix` est un tableau de deux lignes et deux colonnes. Les lignes correspondent aux classes réelles, tandis que les colonnes correspondent aux classes prédites. Chaque entrée dénombre la quantité d'échantillons appartenant à la classe correspondant à la ligne (ici, « neuf » et « non neuf ») et qui ont été classifiés dans la colonne de prédiction. Le tracé de la Figure 5.10 illustre cela :

**In[44]:**

```
mglearn.plots.plot_confusion_matrix_illustration()
```

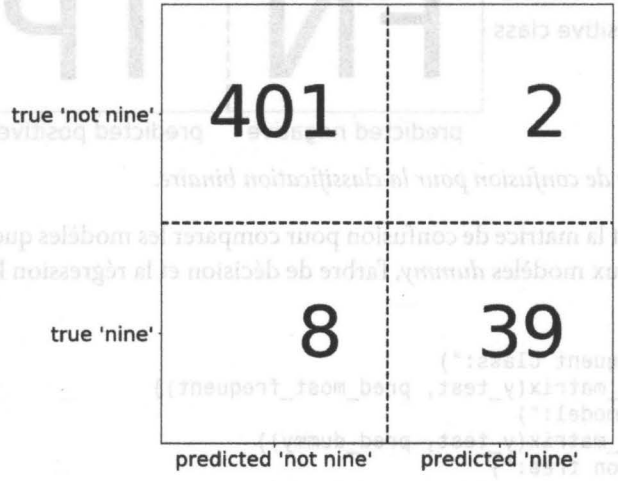



*Figure 5.10 : Un exemple de matrice de confusion.*

Les entrées de la diagonale principale de la matrice de confusion correspondent aux classifications correctes, tandis que les autres entrées nous indiquent combien d'échantillons d'une classe ont été mal classifiés dans l'autre classe.



La diagonale principale d'un tableau ou d'une matrice à deux dimensions A est A[i, i].

Si nous déclarons qu'« un neuf » est la classe positive, nous pouvons relier les entrées de la matrice de confusion avec les expressions *faux positif* et *faux négatif*, déjà rencontrées. Pour compléter le schéma, nous appelons les échantillons correctement classifiés dans la classe positive des *vrais positifs*, et les échantillons correctement classifiés dans la classe négative des *vrais négatifs*. Ces expressions sont généralement abrégées en FP, FN, VP (ou TP pour *true positive*) et VN (ou TN pour *true negative*). De ce point de vue, la matrice de confusion prend la forme illustrée sur la Figure 5.11 :

**In[45] :**

```
mglearn.plots.plot_binary_confusion_matrix()
```

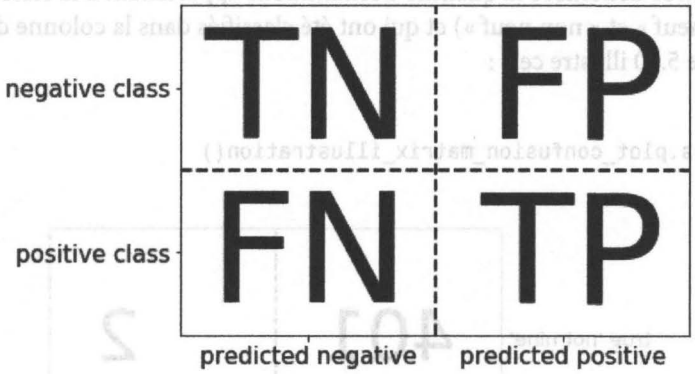



*Figure 5.11 : Matrice de confusion pour la classification binaire.*

Utilisons maintenant la matrice de confusion pour comparer les modèles que nous avons entraînés plus haut (les deux modèles *dummy*, l'arbre de décision et la régression logistique) :

**In[46] :**

```
print("Most frequent class:")
print(confusion_matrix(y_test, pred_most_frequent))
print("\nDummy model:")
print(confusion_matrix(y_test, pred_dummy))
print("\nDecision tree:")
print(confusion_matrix(y_test, pred_tree))
print("\nLogistic Regression")
print(confusion_matrix(y_test, pred_logreg))
```

**Out[46] :**

```
Most frequent class:
[[403   0]
 [ 47   0]]

Dummy model:
[[361  42]
 [ 39   8]]
```

```
Decision tree:
[[390  13]
 [ 24  23]]

Logistic Regression
[[401   2]
 [  8  39]]
```

Il est assez clair, quand on regarde la matrice de confusion, que quelque chose ne va pas avec `pred_most_frequent`, car elle prédit toujours la même classe. D'un autre côté, `pred_dummy` a un très petit nombre de vrais positifs (8), surtout si on la compare avec les faux négatifs et les vrais négatifs (il y a beaucoup plus de faux positifs que de vrais positifs !). Les prédictions effectuées par l'arbre de décision sont bien plus cohérentes, même si l'exactitude n'est pas si différente. Finalement, nous pouvons voir que la régression logistique l'emporte sur tous les aspects : elle a plus de vrais positifs et de vrais négatifs, tout en ayant moins de faux positifs et de faux négatifs. Dans cet exemple, il est clair que seuls l'arbre de décision et la régression logistique fournissent des résultats raisonnables, et que la régression logistique fonctionne systématiquement mieux que l'arbre. Cependant, inspecter la matrice de confusion entière est un peu pénible, et tout en offrant une meilleure vue sur l'ensemble des aspects de cette matrice, ce processus reste très manuel et qualitatif. Il existe plusieurs manières de résumer les informations contenues dans la matrice de confusion, comme nous allons le voir maintenant.

**Relation avec l'exactitude.** Une manière de résumer les résultats obtenus dans la matrice de confusion consiste à calculer l'exactitude, c'est-à-dire le rapport entre le nombre de prédictions correctes et le nombre total d'échantillons traités (le total des entrées de la matrice de confusion) :

$$Exactitude = \frac{TP + TN}{TP+TN + FP + FN}$$

**Précision, rappel et *f*-mesure.** Il existe plusieurs méthodes permettant de résumer la matrice de confusion, les plus courantes étant la précision et le rappel. La *précision* mesure la proportion des échantillons prédits comme étant positifs et qui le sont effectivement :

$$Précision = \frac{TP}{TP + FP}$$

La précision est utilisée comme une métrique de performance lorsque le but est de limiter le nombre de faux positifs. Imaginez par exemple un modèle clinique dans lequel il s'agit de prédire si un nouveau médicament sera efficace dans le traitement d'une certaine maladie. Les essais cliniques sont notoirement chers, et une entreprise pharmaceutique ne se lancera dans une expérimentation que si elle est très sûre que le médicament va réellement marcher. Il est donc important que le modèle ne produise pas de nombreux faux positifs, en d'autres termes

qu'il possède une précision élevée. La précision est également connue sous le nom de *valeur prédictive positive* (*positive predictive value*, ou PPV, en anglais).

Le *rappel*, d'un autre côté, mesure la quantité d'échantillons positifs capturés par les prédictions positives :

$$Rappel = \frac{TP}{TP + FP}$$

Le rappel est utilisé comme métrique de performance lorsque nous avons besoin d'identifier tous les échantillons positifs, autrement dit quand il est important d'éviter les faux négatifs. Le cas du diagnostic d'un cancer, déjà évoqué, en est un bon exemple. Il est en effet important de trouver toutes les personnes qui sont malades, en incluant le cas échéant des personnes bien portantes dans la prédiction. Le rappel est également appelé *sensibilité*, ou encore *taux de vrais positifs* (*true positive rate*, ou TPR, en anglais).

Il y a un compromis entre optimiser le rappel et optimiser la précision. Vous pouvez trivialement obtenir un rappel parfait si vous prédisez que tous les échantillons appartiennent à la classe positive. Il n'y aurait en effet alors pas de faux négatifs, ni d'ailleurs de vrais négatifs. Cependant, cela générerait aussi de nombreux faux positifs, et la précision serait donc très basse. D'un autre côté, si vous trouvez un modèle qui prédit qu'un seul point de donnée est un faux positif, le reste étant négatif, la précision sera parfaite tandis que le rappel sera très mauvais.



La précision et le rappel ne sont que deux des nombreuses mesures de classification dérivées de VP, FP, VN et FN. Dans la communauté de l'apprentissage automatique, la précision et le rappel sont incontestablement les mesures les plus couramment employées pour la classification binaire, mais d'autres communautés pourraient utiliser des métriques différentes. Voyez notamment à ce sujet ce qui est dit dans Wikipédia sur le sujet « Sensibilité et spécificité », et, surtout, en anglais, l'article « Sensitivity and specificity ».

Bien que la précision et le rappel soient si importants, ne s'intéresser qu'à une de ces mesures ne peut pas suffire à brosser tout le tableau. Une manière de les combiner consiste à utiliser la mesure de compromis appelée *f-mesure*, ou encore *f-score*, qui est basée sur la moyenne harmonique de la précision et du rappel :

$$F = 2 \bullet \frac{précision \bullet rappel}{précision + rappel}$$

Comme cette formule prend en compte aussi bien la précision que le rappel, elle peut constituer une meilleure mesure que la précision dans le cas d'une classification binaire avec un jeu de données déséquilibré. Voyons ce qui se passe avec notre jeu de données précédent. Ici, nous allons supposer que la classe « neuf » est la classe positive, donc en l'occurrence minoritaire (elle est étiquetée comme étant `True` – vrai – tandis que le reste est étiqueté comme `False` – faux) :

**In[47]:**

```
from sklearn.metrics import f1_score
print("f1 score most frequent: {:.2f}".format(
    f1_score(y_test, pred_most_frequent)))
print("f1 score dummy: {:.2f}".format(f1_score(y_test, pred_dummy)))
print("f1 score tree: {:.2f}".format(f1_score(y_test, pred_tree)))
print("f1 score logistic regression: {:.2f}".format(
    f1_score(y_test, pred_logreg)))
```

**Out[47]**

```
f1 score most frequent: 0.00
f1 score dummy: 0.16
f1 score tree: 0.55
f1 score logistic regression: 0.89
```

Ici, nous pouvons noter deux choses. Tout d'abord, nous obtenons un message d'erreur pour la prédiction `most_frequent`, puisqu'il n'y a aucune prédiction pour la classe positive (ce qui rend le dénominateur de *f*-mesure nul). De plus, il y a un saut très important entre les prédictions *dummy* et arborescentes, ce qui n'était pas aussi clair en ne regardant que l'exactitude. En utilisant *f*-mesure pour l'évaluation, nous avons à nouveau condensé les performances prédictives en un seul nombre. Cependant, *f*-mesure semble capturer notre intuition de ce qui constitue un bon modèle, bien mieux que ne le faisait l'exactitude. Par contre, un inconvénient de cette mesure est qu'elle est plus difficile à interpréter et à expliquer que l'exactitude.

Si nous voulons un rapport plus compréhensible sur la précision, le rappel et la *f*-mesure, nous pouvons utiliser la fonction `classification_report` afin de calculer les trois à la fois puis de les afficher très proprement :

**In[48]:**

```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred_most_frequent,
                            target_names=["not nine", "nine"]))
```

**Out[48]:**

```
              precision    recall  f1-score   support

    not nine       0.90      1.00      0.94       403
        nine       0.00      0.00      0.00        47

 avg/total         0.80      0.90      0.85       450
```

La fonction `classification_report` produit une ligne par classe (ici, `True` et `False`) et fournit les valeurs de précision, rappel et *f*-mesure pour la classe positive (en l'occurrence, la classe minoritaire « nine » choisie plus haut). Si nous inversions les deux classes, nous pourrions constater sur la sortie de `classification_report` que nous obtenons une *f*-mesure de 0.94 avec le modèle `most_frequent`. En outre, nous avons un rappel de 1 pour la classe « non neuf » puisque nous avons classifié tous les échantillons comme étant « non neuf ». La dernière colonne fournit le

*support* de chaque classe, soit tout simplement le nombre d'échantillons de cette classe selon la réalité du terrain.

La dernière ligne du rapport de classification montre une moyenne pondérée (par le nombre d'échantillons dans la classe) du remplissage de chaque classe. Voici deux autres rapports, un pour le classifieur *dummy* et l'autre pour la régression logistique :

**In[49]:**

```
print(classification_report(y_test, pred_dummy,
                            target_names=["not nine", "nine"]))
```

**Out[49]:**

```
              precision    recall  f1-score   support

    not nine       0.90      0.90      0.90       403
        nine       0.16      0.17      0.16        47

 avg/total         0.82      0.82      0.82       450
```

**In[50]:**

```
print(classification_report(y_test, pred_logreg,
                            target_names=["not nine", "nine"]))
```

**Out[50]:**

```
              precision    recall  f1-score   support

    not nine       0.98      1.00      0.99       403
        nine       0.95      0.83      0.89        47

 avg/total         0.98      0.98      0.98       450
```

Comme vous pouvez le constater ici, les différences entre les modèles *dummy* et un très bon modèle ne sont plus aussi claires. Décider de la classe qui est déclarée comme étant positive a un grand impact sur les métriques. Alors que la *f*-mesure (`f1-score`) donne 0.16 pour la classification *dummy* (contre 0.89 pour la régression logistique) pour la classe « neuf », les valeurs obtenues avec la classe « non neuf » sont respectivement de 0.90 et 0.99, qui semblent tous les deux être des résultats raisonnables. Observer tous ces nombres en même temps donne un tableau assez précis de ce qui se passe, ce qui nous permet de voir clairement la supériorité du modèle de régression logistique.

### Prendre en compte l'incertitude

La matrice de confusion et le rapport de classification fournissent une analyse très détaillée pour un jeu particulier de prédictions. Cependant, les prédictions elles-mêmes éliminent déjà certaines informations qui sont contenues dans le modèle. Comme nous l'avons vu dans le Chapitre 2, la plupart des classifieurs fournissent une fonction `decision_function` ou une méthode `predict_proba` pour évaluer les degrés de certitude sur les prédictions. Faire des prédictions peut être vu comme un seuillage de la sortie produite par `decision_function` ou `predict_`

`proba` en un certain point fixé. Dans la classification binaire, nous utilisons 0 pour la fonction de décision et 0.5 pour `predict_proba`.

Dans l'exemple qui suit, nous avons une tâche de classification binaire déséquilibrée, avec 400 points classifiés dans la classe négative contre 50 dans la classe positive. Les données d'apprentissage sont illustrées sur la gauche de la Figure 5.12. Nous lançons un entraînement sur ces données avec un modèle SVM. Les tracés de droite illustrent les valeurs de la fonction de décision sous la forme de cartes de chaleur. Vous pouvez voir un cercle noir en haut et vers le centre qui signale la position pour laquelle le seuil de la fonction de décision (`decision_function`) est exactement égal à zéro. Les points à l'intérieur de ce cercle seront classifiés comme formant la classe positive, et ceux qui sont à l'extérieur comme constituant la classe négative :

**In[51]:**

```
from mglearn.datasets import make_blobs
X, y = make_blobs(n_samples=(400, 50), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
```

**In[52]:**

```
mglearn.plots.plot_decision_threshold()
```

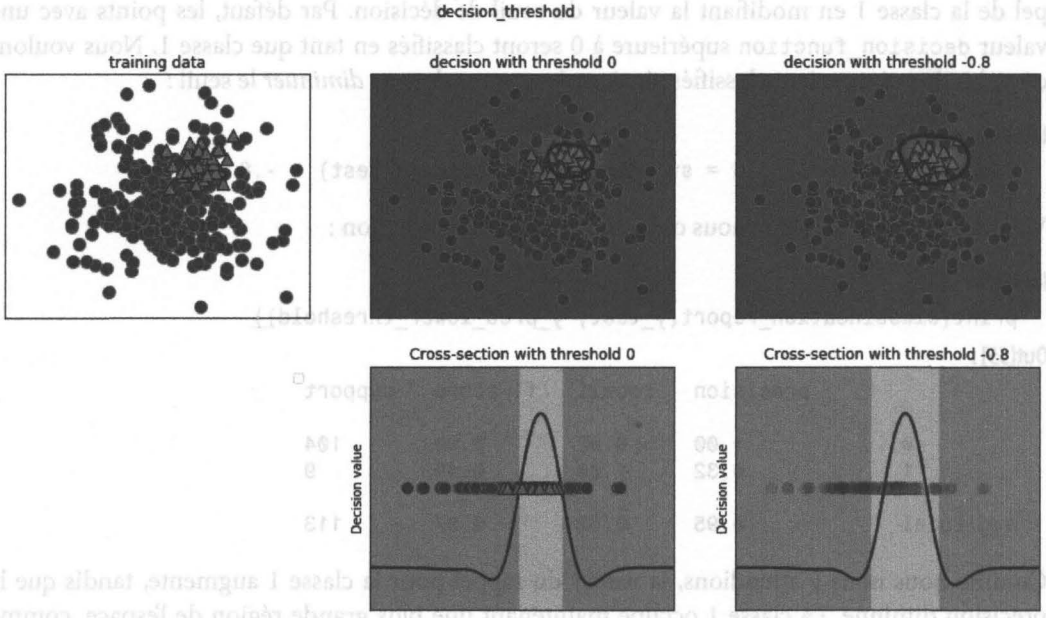



*Figure 5.12 : Carte de chaleur de la fonction de décision, et impact produit par un changement dans le seuil de décision.*

Nous pouvons utiliser la fonction `classification_report` pour évaluer la précision et le rappel pour les deux classes :

**In[53]:**

```
print(classification_report(y_test, svc.predict(X_test)))
```

**Out[53]:**

```
             precision    recall  f1-score   support

          0       0.97      0.89      0.93       104
          1       0.35      0.67      0.46         9

avg/total         0.92      0.88      0.89       113
```

Pour la classe 1, nous obtenons une précision assez faible, et le rappel est moyen. Du fait que la classe 0 est beaucoup plus importante, le classifieur se concentre sur les résultats qui la concernent, et non sur la plus petite classe 1.

Supposons que, dans notre application, il soit plus important d'avoir un rappel élevé pour la classe 1, comme par exemple dans le cas déjà évoqué du dépistage d'un cancer. Cela signifie que nous acceptons de risquer davantage de faux positifs (faux classe 1) en échange de plus de vrais positifs (ce qui augmente le rappel). Les prédictions générées par `scv.predict` ne remplissent réellement pas cette condition, mais nous pouvons ajuster les prédictions pour améliorer le rappel de la classe 1 en modifiant la valeur du seuil de décision. Par défaut, les points avec une valeur `decision_function` supérieure à 0 seront classifiés en tant que classe 1. Nous voulons que *plus* de points soient classifiés ainsi, et donc nous devons *diminuer* le seuil :

**In[54]:**

```
y_pred_lower_threshold = svc.decision_function(X_test) > -.8
```

Voyons maintenant ce que nous obtenons pour cette prédiction :

**In[55]:**

```
print(classification_report(y_test, y_pred_lower_threshold))
```

**Out[55]:**

```
             precision    recall  f1-score   support

          0       1.00      0.82      0.90       104
          1       0.32      1.00      0.49         9

avg/total         0.95      0.83      0.87       113
```

Comme nous nous y attendions, la valeur du rappel pour la classe 1 augmente, tandis que la précision diminue. La classe 1 occupe maintenant une plus grande région de l'espace, comme l'illustre le panneau en haut et à droite de la Figure 5.12. Si la précision est plus importante que le rappel, ou l'inverse, ou si vos données sont trop déséquilibrées, changer le seuil de décision est la manière la plus simple d'améliorer les résultats. Du fait que `decision_function` peut jouer avec

des intervalles arbitraires, il est cependant difficile de fournir une règle un tant soit peu générale quant au choix d'un seuil.



Pour plus de simplicité, nous avons changé dans le code ci-dessus la valeur du seuil en fonction des résultats obtenus sur le jeu de test. En pratique, vous devez utiliser un jeu de validation simple (*hold-out*), et non le jeu de test. Comme pour tout autre paramètre, définir un seuil de décision à partir du jeu de test est susceptible de conduire à des résultats exagérément optimistes. Servez-vous à la place d'un jeu de validation, ou d'une validation croisée.

Choisir un seuil pour des modèles qui implémentent la méthode `predict_proba` peut être plus facile (la sortie produite est comprise entre 0 et 1 et représente une probabilité). Par défaut, la valeur de seuil de 0.5 signifie que, si le modèle est « sûr » à plus de 50 % qu'un point est dans la classe positive, alors il est classifié comme tel. Augmenter la valeur du seuil signifie que le modèle a besoin d'améliorer son taux de certitude pour prendre une décision positive (et donc qu'il a besoin de moins de certitude pour prendre une décision négative). Bien que travailler avec des probabilités puisse être plus intuitif qu'avec des seuils arbitraires, tous les modèles ne fournissent pas une vue réaliste des phénomènes d'incertitude (par exemple, un arbre de décision qui a atteint toute sa profondeur est toujours sûr à 100 % de ses décisions, même s'il pourrait souvent se tromper à ce sujet). Cela conduit au concept de *calibration*, sujet qui dépasse largement le cadre de ce livre. Disons simplement qu'un modèle calibré est un modèle qui fournit une mesure précise de son incertitude. Les lecteurs anglicistes et assez férus de mathématiques pourront lire avec profit l'article « Predicting Good Probabilities With Supervised Learning » d'Alexandru Niculescu-Mizil et Rich Caruana (`http://www.machinelearning.org/proceedings/icml2005/papers/079_GoodProbabilities_NiculescuMizilCaruana.pdf`).

### Courbes précision-rappel et courbes ROC

Comme nous venons de le voir, changer le seuil utilisé pour prendre une décision de classification dans un modèle est une manière d'ajuster le compromis entre précision et rappel pour une classe donnée. Peut-être voulez-vous manquer moins de 10 % d'échantillons positifs, et donc vouloir un rappel d'au moins 90 %. Cette décision dépend de l'application, et elle devrait être prise en fonction des objectifs recherchés. Une fois un certain but atteint, disons une valeur de rappel ou de précision particulière pour une classe, un seuil peut être défini de manière appropriée. Il est toujours possible de choisir un seuil pour remplir un objectif spécifique, comme obtenir un rappel de 90 %. La partie difficile de l'affaire, c'est de développer un modèle qui conserve une précision raisonnable avec ce seuil. Si vous classifiez tout comme étant positif, vous obtiendrez certes un rappel de 100 %, mais votre modèle n'aura aucune utilité.

Poser une exigence sur un classifieur, comme un rappel de 90 %, est souvent appelé définir le *point de fonctionnement* (*operating point*). Fixer un point de fonctionnement est utile dans certains domaines, par exemple pour apporter des garanties de performances à des clients ou à d'autres groupes dans votre organisation.

En règle générale, lors du développement d'un nouveau modèle, ce que sera ce point de fonctionnement n'est pas quelque chose de totalement clair. Pour cette raison, et pour mieux comprendre un problème de modélisation, il est instructif de regarder tous les seuils possibles, ou tous les compromis possibles entre précision et rappel *d'un seul coup*. Cela est réalisable en faisant appel à un outil appelé la *courbe précision-rappel*. Vous pouvez trouver la fonction qui calcule cette courbe dans le module `sklearn.metrics`. Elle réclame en particulier en argument ce qui est renvoyé par `decision_function` ou `predict_proba` :

```
In [56] :
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
```

La fonction `precision_recall_curve` retourne une liste de valeurs de précision et de rappel pour tous les seuils possibles (toutes les valeurs qui apparaissent dans la fonction de décision) selon un ordre de tri permettant de définir les axes de la courbe, comme l'illustre la Figure 5.13 :

**In[57]:**

```
X, y = make_blobs(n_samples=(4000, 500), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
close_zero = np.argmin(np.abs(thresholds))
plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
         label="threshold zero", fillstyle="none", c='k', mew=2)

plt.plot(precision, recall, label="precision recall curve")
plt.xlabel("Precision")
plt.ylabel("Recall")
plt.legend(loc="best")
```

Chaque point de la courbe de la Figure 5.13 correspond à un seuil possible de `decision_function`. Nous voyons par exemple que nous pouvons obtenir un rappel de 0.4 pour une précision d'environ 0.75. Le cercle noir marque le point qui correspond à un seuil de zéro, autrement dit la valeur par défaut de celui-ci pour `decision_function`. Ce point est le compromis qui est choisi lors de l'appel à la méthode `predict`.

Plus une courbe se rapproche du coin supérieur droit, meilleur est le classifieur. En effet, un point dans cette zone implique une haute précision *et* un rappel élevé pour le même seuil. La courbe débute en haut et à gauche, ce qui correspond à un seuil très bas, classifiant tout comme faisant partie de la classe positive. Augmenter le seuil déplace la courbe vers des précisions plus élevées, mais aussi vers des rappels plus bas. Élever encore plus le seuil conduit à une situation dans laquelle la plupart des points classifiés comme étant positifs sont des vrais positifs, ce qui amène à une précision élevée, mais aussi à un rappel plus petit. Plus le modèle conserve un rappel d'un haut niveau lorsque la précision s'accroît, meilleur il est.

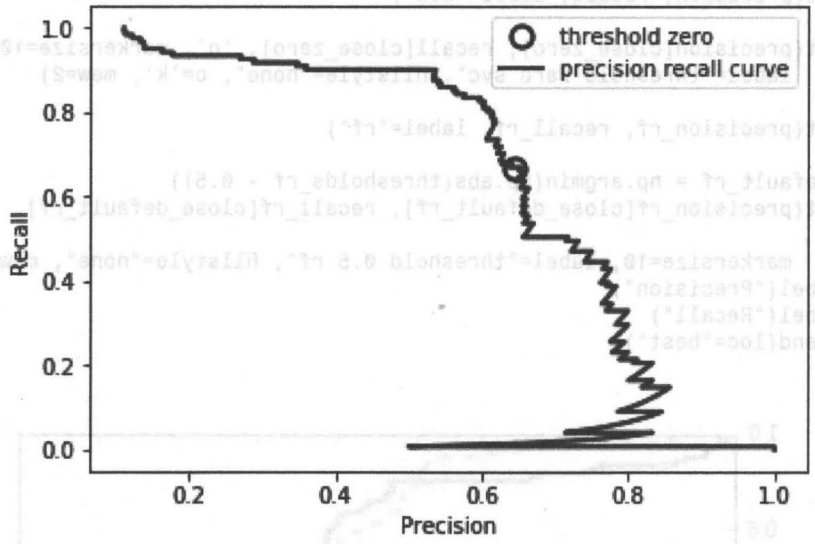



*Figure 5.13 : Courbe précision-rappel pour SVC (`gamma` = 0.05).*

En observant cette courbe encore plus, nous pouvons voir qu'avec ce modèle il est possible d'obtenir une précision allant jusqu'à environ 0.5 avec une valeur très élevée du rappel. Si nous voulons obtenir une précision supérieure, il faut sacrifier une partie du rappel. En d'autres termes, la partie gauche de la courbe est relativement plate, ce qui signifie que le rappel ne diminue pas beaucoup lorsque la précision augmente. Par contre, dès que celle-ci dépasse 0.5, le rappel baisse rapidement.

Divers classifieurs peuvent fonctionner correctement sur différentes parties de la courbe, autrement dit en différents points de fonctionnement. Comparons le SVM précédent avec une forêt aléatoire entraînée sur le même jeu de données. Rappelez-vous que `RandomForestClassifier` ne possède pas de fonction `decision_function`, seulement `predict_proba`. La fonction precision_recall_curve attend comme second argument une mesure de certitude pour la classe positive (classe 1), et donc nous lui passons la probabilité pour un échantillon d'appartenir à cette classe, soit ici `rf.predict_proba(X_test)[:, 1]`. Le seuil par défaut pour `predict_proba` dans la classification binaire est 0.5, ce qui correspond au point marqué sur la courbe (voir la Figure 5.14) :

**In[58] :**

```
rom sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=100, random_state=0, max_features=2)
rf.fit(X_train, y_train)

precision_rf, recall_rf, thresholds_rf = precision_recall_curve(
    y_test, rf.predict_proba(X_test)[:, 1])
```

```
plt.plot(precision, recall, label="svc")

plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
         label="threshold zero svc", fillstyle="none", c='k', mew=2)

plt.plot(precision_rf, recall_rf, label="rf")

close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(precision_rf[close_default_rf], recall_rf[close_default_rf], '^',
  c='k',
         markersize=10, label="threshold 0.5 rf", fillstyle="none", mew=2)
plt.xlabel("Precision")
plt.ylabel("Recall")
plt.legend(loc="best")
```

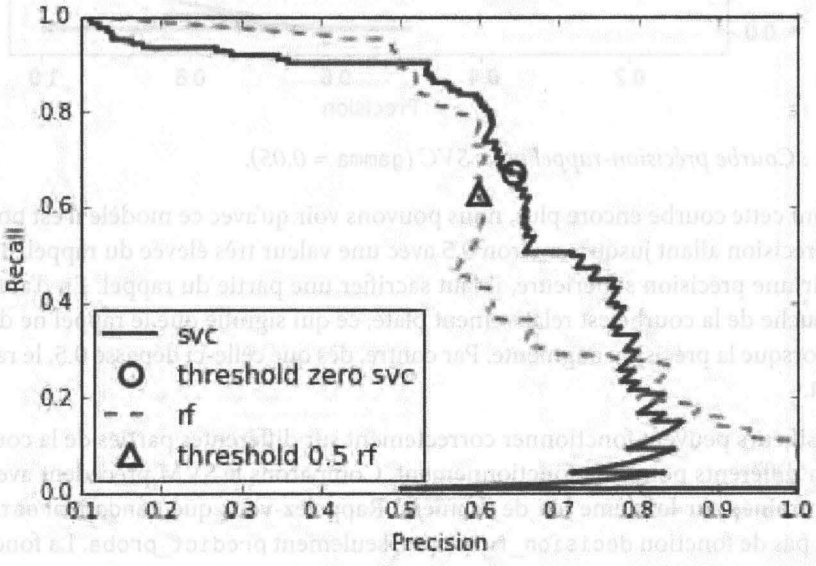



*Figure 5.14 : Comparaison entre deux courbes précision-rappel (SVM et forêt aléatoire).*

Sur le tracé de la Figure 5.14, nous pouvons constater que le résultat obtenu avec la forêt aléatoire est meilleur aux extrêmes, autrement dit là où les valeurs du rappel ou bien de la précision sont les plus élevées. Vers le milieu (pour une précision d'approximativement 0.7), c'est le SVM qui l'emporte. Si nous nous en étions tenus à la *f*-mesure pour comparer les performances globales, nous serions passés à côté de ces subtilités. En effet, une *f*-mesure capture uniquement un point sur la courbe précision-rappel, celui qui correspond au seuil par défaut :

**In[59]:**

```
print("f1_score of random forest: {:.3f}".format(
    f1_score(y_test, rf.predict(X_test))))
print("f1_score of svc: {:.3f}".format(f1_score(y_test, svc.predict(X_test))))
```

**Out[59]:**

```
f1_score of random forest: 0.610
f1_score of svc: 0.656
```

Comparer deux courbes précision-rappel fournit nombre de renseignements détaillés, mais c'est un processus essentiellement manuel. Pour une comparaison automatique entre modèles, nous devrions donc essayer de résumer les informations contenues dans la courbe, et ce sans nous limiter à un seuil ou à un point de fonctionnement spécifique. Une technique particulière consiste à calculer l'intégrale ou l'aire de la partie qui se trouve sous la courbe précision-rappel, également appelée *précision moyenne*. Vous pouvez utiliser la fonction `average_precision_score` pour évaluer la précision moyenne. Comme nous avons besoin de calculer la courbe précision-rappel et de considérer de multiples seuils, le résultat de `decision_function` ou de `predict_proba` doit être passé à `average_precision_score`, et non celui de `predict` :



Il y a quelques différences techniques mineures entre aire sous la courbe précision-rappel et précision moyenne. Cependant, cette explication donne l'idée générale.

**In[60]:**

```
from sklearn.metrics import average_precision_score
ap_rf = average_precision_score(y_test, rf.predict_proba(X_test)[:, 1])
ap_svc = average_precision_score(y_test, svc.decision_function(X_test))
print("Average precision of random forest: {:.3f}".format(ap_rf))
print("Average precision of svc: {:.3f}".format(ap_svc))
```

**Out[60]:**

```
Average precision of random forest: 0.660
Average precision of svc: 0.666
```

Nous voyons que les résultats fournis par SVC et la forêt aléatoire sont très proches les uns des autres, l'avantage étant ici légèrement pour SVC (mais l'inverse serait également totalement possible). Cela peut sensiblement différer de ce que nous avions obtenu plus haut avec la *f*-mesure. Comme nous nous intéressons à une courbe dont les extrémités se situent entre 0 et 1, la précision moyenne renvoie toujours une valeur comprise entre 0 (le pire) et 1 (le meilleur). La précision moyenne d'un classifieur qui attribue `decision_function` au hasard est la fraction d'échantillons positifs dans le jeu de données.

### Courbe ROC (*Receiver operating characteristic*) et AUC

Il existe un autre outil qui est couramment employé pour analyser le comportement de classifieurs en fonction de différents seuils. Il s'agit de la courbe ROC, abréviation de *Receiver operating characteristic*. Comme la courbe précision-rappel, la courbe ROC considère tous les seuils possibles pour un classifieur donné, mais au lieu de fournir un lien entre précision et rappel, elle montre le *taux de faux positifs* (TFP) par rapport au *taux de vrais positifs* (TVP). Souvenez-vous que le taux de vrais positifs est simplement un autre nom du rappel (ou sensibilité), tandis que le

taux de faux positifs est la fraction de faux positifs qui sont détectés dans l'ensemble des échantillons négatifs :

$$TFP = \frac{FP}{FP + TN}$$

La courbe ROC peut être calculée en utilisant la fonction roc_curve (voir la Figure 5.15) :

**In[61]:**

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, svc.decision_function(X_test))

plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
close_zero = np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
         label="threshold zero", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
```

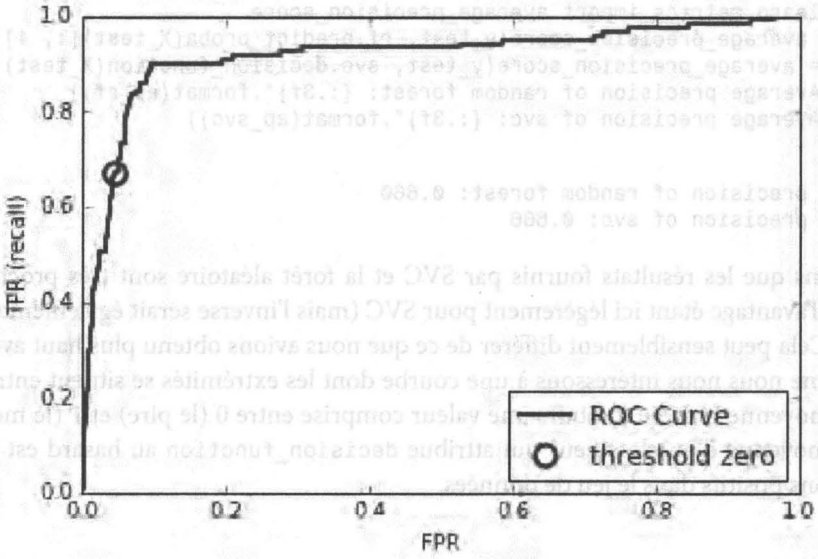



*Figure 5.15 : Courbe ROC pour SVM.*

La courbe idéale est dans ce cas proche de la région en haut et à gauche : vous voulez un classifieur qui produise un *rappel élevé* tout en conservant un *taux de faux positifs bas*. Partant de la valeur de seuil par défaut, zéro, la courbe montre que nous pouvons atteindre un rappel très significativement élevé (autour de 0.9) en n'augmentant que légèrement la valeur du TFP. Le point le plus proche de la partie en haut et à gauche de la courbe pourrait être un meilleur point

de fonctionnement que celui choisi par défaut. Une nouvelle fois, souvenez-vous que le seuil devrait être déterminé sur un jeu de validation séparé, et non sur le jeu de test.

La Figure 5.16 propose une comparaison entre forêt aléatoire et SVM en utilisant pour cela des courbes ROC :

**In[62]:**

```
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, rf.predict_proba(X_test)[:,
  1])

plt.plot(fpr, tpr, label="ROC Curve SVC")
plt.plot(fpr_rf, tpr_rf, label="ROC Curve RF")

plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
      label="threshold zero SVC", fillstyle="none", c='k', mew=2)
close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(fpr_rf[close_default_rf], tpr[close_default_rf], '^', markersize=10,
      label="threshold 0.5 RF", fillstyle="none", c='k', mew=2)

plt.legend(loc=4)
```

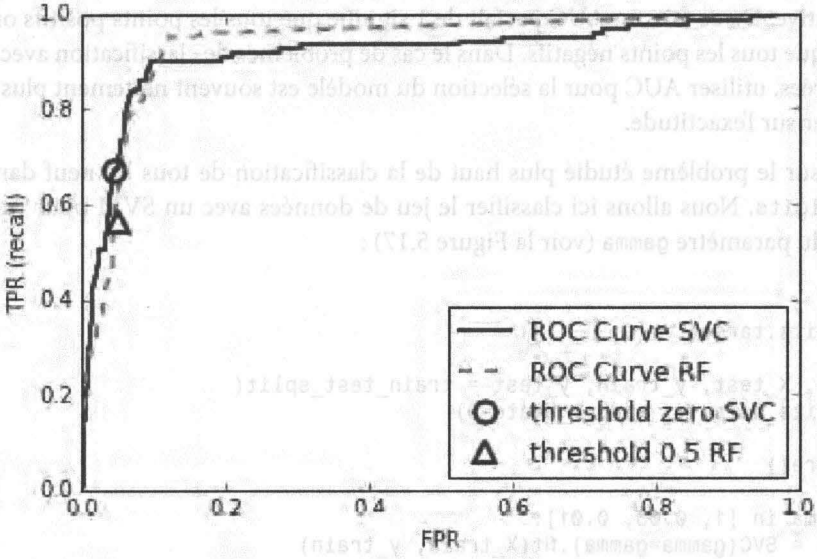



*Figure 5.16 : Comparaison entre courbes ROC pour classifieurs SVM et forêt aléatoire.*

Comme pour la courbe précision-rappel, nous avons souvent besoin de condenser la courbe ROC en utilisant un nombre unique, qui est l'aire sous cette courbe. C'est ce que l'on désigne par

l'abréviation AUC (*Area Under the Curve*), étant entendu que cette aire est bien celle déterminée par la courbe ROC. Nous pouvons calculer cette aire en faisant appel à la fonction `roc_auc_score` :

**In[63]:**

```
from sklearn.metrics import roc_auc_score
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[:, 1])
svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))
print("AUC for Random Forest: {:.3f}".format(rf_auc))
print("AUC for SVC: {:.3f}".format(svc_auc))
```

**Out[63]:**

```
AUC for Random Forest: 0.937
AUC for SVC: 0.916
```

En comparant les méthodes forêt aléatoire et SVM, nous constatons ici que la première donne un résultat un peu meilleur que la seconde. Souvenez-vous que, puisque la valeur AUC calcule l'aire sous une courbe comprise entre 0 et 1, son résultat est également compris entre 0 (le pire) et 1 (le meilleur). Une prédiction aléatoire renvoie toujours une valeur de AUC de 0.5, et ce quel que soit le déséquilibrage des classes dans le jeu de données. De ce fait, le calcul de l'AUC est une métrique bien meilleure que l'exactitude dans les problèmes de classification déséquilibrés. Il peut être interprété comme une évaluation du *ranking* (ou *ordonnancement*) des échantillons positifs. Il est équivalent à la probabilité qu'un point de la classe positive sélectionné de manière aléatoire ait un score plus élevé pour le classifieur qu'un point sélectionné aléatoirement dans la classe négative. De ce fait, un AUC parfait de 1 signifie que tous les points positifs ont un score plus élevé que tous les points négatifs. Dans le cas de problèmes de classification avec des classes déséquilibrées, utiliser AUC pour la sélection du modèle est souvent nettement plus significatif que se baser sur l'exactitude.

Revenons sur le problème étudié plus haut de la classification de tous les neuf dans le jeu de données `digits`. Nous allons ici classifier le jeu de données avec un SVM pour trois réglages différents du paramètre `gamma` (voir la Figure 5.17) :

**In[64]:**

```
y = digits.target == 9

X_train, X_test, y_train, y_test = train_test_split(
    digits.data, y, random_state=0)

plt.figure()

for gamma in [1, 0.05, 0.01]:
    svc = SVC(gamma=gamma).fit(X_train, y_train)
    accuracy = svc.score(X_test, y_test)
    auc = roc_auc_score(y_test, svc.decision_function(X_test))
    fpr, tpr, _ = roc_curve(y_test , svc.decision_function(X_test))
    print("gamma = {:.2f}  accuracy = {:.2f}  AUC = {:.2f}".format(
        gamma, accuracy, auc))
    plt.plot(fpr, tpr, label="gamma={:.3f}".format(gamma))
```

```
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.xlim(-0.01, 1)
plt.ylim(0, 1.02)
plt.legend(loc="best")
```

**Out[64]:**

```
gamma = 1.00  accuracy = 0.90  AUC = 0.50
gamma = 0.05  accuracy = 0.90  AUC = 0.90
gamma = 0.01  accuracy = 0.90  AUC = 1.00
```

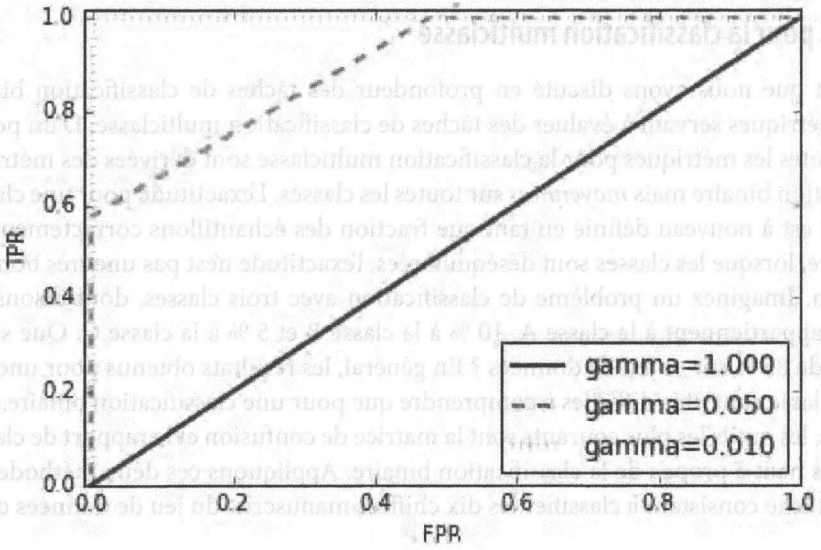



*Figure 5.17 : Comparaison de courbes ROC de SVM avec différentes valeurs de* `gamma`.

Notez que l'exactitude des trois valeurs de `gamma` est la même, soit 90 %. Cela pourrait provenir d'une performance déterminée au hasard, ou non. En regardant les valeurs AUC et les courbes correspondantes, nous pouvons cependant constater une distinction claire entre les trois modèles. Avec `gamma = 1.0`, la courbe AUC se situe en fait au niveau du hasard, ce qui signifie que la sortie de `decision_function` est aussi bonne qu'une valeur aléatoire. Avec `gamma = 0.5`, la performance fait un bond en avant très significatif jusqu'à une valeur AUC de 0.9. Enfin, avec `gamma = 0.01`, AUC atteint la valeur parfaite de 1.0. Cela signifie donc que tous les points positifs sont classés au-dessus de tous les points négatifs conformément à la fonction de décision. En d'autres termes, avec le bon seuil, ce modèle peut classifier les données de façon parfaite ! Sachant cela, nous pouvons ajuster le seuil pour ce modèle et obtenir d'excellentes prédictions. Si nous n'avions utilisé que la notion d'exactitude, nous n'aurions jamais fait cette découverte.



Si vous regardez en détail la partie de la courbe correspondant à `gamma = 0.01`, vous pourrez voir comme un petit défaut près de l'angle en haut et à gauche. Cela signifie qu'un moins un point n'a pas été classé correctement. La valeur d'AUC de 1.0 est une conséquence de l'arrondi sur le second chiffre décimal.

Pour cette raison, nous recommandons vivement d'utiliser AUC chaque fois qu'il s'agit d'évaluer des modèles à partir de données déséquilibrées. N'oubliez pas, cependant, qu'AUC n'utilise pas le seuil par défaut, et donc qu'ajuster le seuil de décision pourrait être nécessaire afin d'obtenir des résultats de classification utiles à partir d'un modèle pour lequel la valeur AUC est élevée.

## Métriques pour la classification multiclasse

Maintenant que nous avons discuté en profondeur des tâches de classification binaire, passons aux métriques servant à évaluer des tâches de classification multiclasse. D'un point de vue basique, toutes les métriques pour la classification multiclasse sont dérivées des métriques pour la classification binaire mais *moyennées* sur toutes les classes. L'exactitude pour une classification multiclasse est à nouveau définie en tant que fraction des échantillons correctement classifiés. Et, là encore, lorsque les classes sont déséquilibrées, l'exactitude n'est pas une très bonne mesure d'évaluation. Imaginez un problème de classification avec trois classes, dont disons 85 % des points qui appartiennent à la classe A, 10 % à la classe B et 5 % à la classe C. Que signifie une exactitude de 85 % sur ce jeu de données ? En général, les résultats obtenus pour une classification multiclasse sont plus difficiles à comprendre que pour une classification binaire. Mis à part l'exactitude, les outils les plus courants sont la matrice de confusion et le rapport de classification étudiés plus haut à propos de la classification binaire. Appliquons ces deux méthodes d'évaluation sur la tâche consistant à classifier les dix chiffres manuscrits du jeu de données `digits` :

**In[65]:**

```
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target, random_state=0)
lr = LogisticRegression().fit(X_train, y_train)
pred = lr.predict(X_test)
print("Accuracy: {:.3f}".format(accuracy_score(y_test, pred)))
print("Confusion matrix:\n{}".format(confusion_matrix(y_test, pred)))
```

**Out[65]:**

```
Accuracy: 0.953
Confusion matrix:
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 39  0  0  0  0  2  0  2  0]
 [ 0  0 41  3  0  0  0  0  0  0]
 [ 0  0  1 43  0  0  0  0  0  1]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  1  0  0  0 47  0  0  0  0]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  1  0  1  1  0  0 45  0  0]
```

```
 [ 0  3  1  0  0  0  0  0 43  1]
 [ 0  0  0  1  0  1  0  0  1 44]]
```

Le modèle présente une exactitude de 95.3 %, ce qui nous indique déjà que la situation est plutôt bonne. La matrice de confusion fournit quelques détails supplémentaires. Comme dans le cas binaire, chaque ligne correspond à une étiquette vraie, et chaque colonne à une étiquette prédite. La Figure 5.18 propose un tracé visuellement plus explicite de ce résultat :

**In[66]:**

```
scores_image = mglearn.tools.heatmap(
    confusion_matrix(y_test, pred), xlabel='Predicted label',
    ylabel='True label', xticklabels=digits.target_names,
    yticklabels=digits.target_names, cmap=plt.cm.gray_r, fmt="%d")
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```

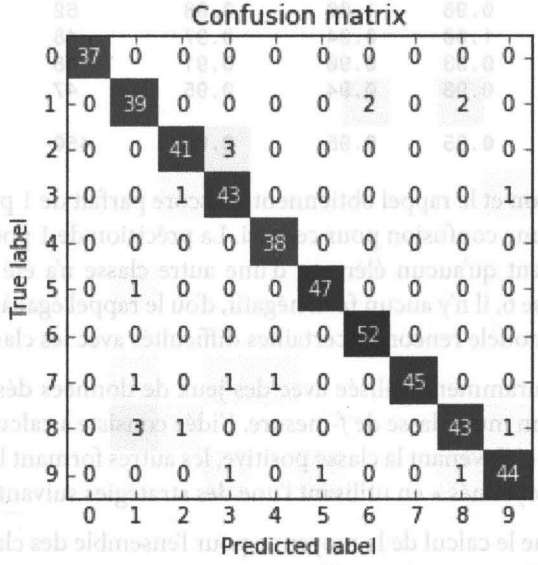



*Figure 5.18 : Matrice de confusion pour une tâche de classification multiclasse.*

Pour la première classe, le chiffre 0, nous trouvons 37 échantillons, tous classifiés en tant que classe 0 (il n'y a pas de faux négatifs pour cette classe). Nous pouvons le constater car toutes les autres entrées de la première ligne de la matrice de confusion sont nulles. Nous voyons également qu'aucun autre chiffre n'a été par erreur classifié dans la classe 0, puisque toutes les autres entrées de la première colonne sont également nulles (il n'y a pas de faux positifs pour la classe 0). Certains chiffres ont cependant été confondus avec d'autres (par exemple, le chiffre 2, soit la troisième ligne, dont trois échantillons ont été classifiés comme appartenant à la classe 3 dans la quatrième colonne). On trouve également un chiffre 3 classifié comme étant un 2

(troisième colonne, quatrième ligne), ou encore le chiffre 8 classifié en 2 (troisième colonne, neuvième ligne).

Avec la fonction `classification_report`, nous pouvons calculer la précision, le rappel et la *f*-mesure pour chaque classe :

**In[67]:**

```
print(classification_report(y_test, pred))
```

**Out[67]:**

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        37
           1       0.89      0.91      0.90        43
           2       0.95      0.93      0.94        44
           3       0.90      0.96      0.92        45
           4       0.97      1.00      0.99        38
           5       0.98      0.98      0.98        48
           6       0.96      1.00      0.98        52
           7       1.00      0.94      0.97        48
           8       0.93      0.90      0.91        48
           9       0.96      0.94      0.95        47

   avg / total       0.95      0.95      0.95       450
```

Sans surprise, la précision et le rappel obtiennent un score parfait de 1 pour la classe 0, puisqu'il n'y a effectivement aucune confusion pour celle-ci. La précision de 1 pour la classe 7, d'un autre côté, indique simplement qu'aucun élément d'une autre classe n'a été classifié par erreur ici, tandis que, pour la classe 6, il n'y aucun faux négatif, d'où le rappel égal à 1. Nous pouvons également constater que le modèle rencontre certaines difficultés avec les classes 8 et 3.

La métrique la plus couramment utilisée avec des jeux de données déséquilibrés dans ce type de situation est la version multiclasse de *f*-mesure. L'idée consiste à calculer une valeur *f*-mesure binaire par classe, celle-ci devenant la classe positive, les autres formant les classes négatives. Ces scores sont ensuite « moyennés » en utilisant l'une des stratégies suivantes :

- « macro » : effectue le calcul de la moyenne pour l'ensemble des classes, celles-ci prenant un poids égal quelle que soit leur taille.
- « pondéré » : calcule la moyenne des *f*-mesures pour toutes les classes, pondérée par leur support. C'est ce qui figure dans le rapport de classification.
- « micro » : effectue le calcul de la moyenne pour le total de faux positifs, de faux négatifs et de vrais positifs pour l'ensemble des classes, puis recalcule précision, rappel et *f*-mesure en utilisant ces résultats.

Si vous voulez que chaque *échantillon* soit traité à l'identique, il est recommandé d'utiliser la stratégie « micro ». Si vous voulez que toutes les *classes* soient considérées à l'identique, il est recommandé d'utiliser la stratégie « macro » :

**In[68]:**

```
print("Micro average f1 score: {:.3f}".format(
    f1_score(y_test, pred, average="micro")))
print("Macro average f1 score: {:.3f}".format(
    f1_score(y_test, pred, average="macro")))
```

**Out[68]:**

```
Micro average f1 score: 0.953
Macro average f1 score: 0.954
```

## Métriques de régression

L'évaluation de la régression peut être effectuée de manière similaire à celle de la classification, par exemple en analysant la surestimation de la cible contre une sous-estimation de celle-ci. Cependant, dans la plupart des applications que nous avons vues, la valeur $R^2$ par défaut utilisée dans la méthode `score` de tous les *régresseurs* est suffisante. Parfois, des décisions sont prises sur la base de l'erreur quadratique moyenne ou de l'erreur absolue moyenne, ce qui peut inciter à affiner les modèles en utilisant ces métriques. En général, cependant, la pratique nous a montré que le coefficient de détermination $R^2$ est une métrique plus intuitive pour évaluer les modèles de régression.

## Utiliser des métriques d'évaluation dans la sélection de modèles

Nous avons discuté en détail de nombreuses méthodes d'évaluation, ainsi que de leur application à la réalité de terrain et à un modèle. Cependant, nous voulons souvent utiliser des métriques comme AUC lors d'une sélection de modèle en utilisant `GridSearchCV` ou `cross_val_score`. Heureusement, `scikit-learn` fournit un moyen très simple d'y parvenir *via* l'argument `scoring` qui est disponible aussi bien dans `GridSearchCV` que dans `cross_val_score`. Il vous suffit de passer dans cet argument une chaîne de caractères décrivant la métrique d'évaluation que vous souhaitez appliquer. Supposons par exemple que vous vouliez évaluer le classifieur SVM sur la tâche « neuf vs. le reste » appliquée au jeu de données `digits` en utilisant comme `scoring` AUC. Il suffit pour cela de remplacer le paramètre par défaut (`accuracy`) par la valeur `roc_auc` :

**In[69]:**

```
print("Default scoring: {}".format(
    cross_val_score(SVC(), digits.data, digits.target == 9)))
explicit_accuracy =  cross_val_score(SVC(), digits.data, digits.target == 9,
                                     scoring="accuracy")
print("Explicit accuracy scoring: {}".format(explicit_accuracy))
roc_auc = cross_val_score(SVC(), digits.data, digits.target == 9,
                          scoring="roc_auc")
print("AUC scoring: {}".format(roc_auc))
```

**Out[69]:**

```
Default scoring: [ 0.9  0.9  0.9]
Explicit accuracy scoring: [ 0.9  0.9  0.9]
AUC scoring: [ 0.994  0.99   0.996]
```

Le principe est semblable dans `GridSearchCV` :

**In[70]:**

```
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target == 9, random_state=0)

param_grid = {'gamma': [0.0001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(SVC(), param_grid=param_grid)
grid.fit(X_train, y_train)
print("Grid-Search with accuracy")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (accuracy)): {:.3f}".format(grid.best_
  score_))
print("Test set AUC: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Test set accuracy: {:.3f}".format(grid.score(X_test, y_test)))
```

**Out[70]:**

```
Grid-Search with accuracy
Best parameters: {'gamma': 0.0001}
Best cross-validation score (accuracy)): 0.970
Test set AUC: 0.992
Test set accuracy: 0.973
```

**In[71]:**

```
grid = GridSearchCV(SVC(), param_grid=param_grid, scoring="roc_auc")
grid.fit(X_train, y_train)
print("\nGrid-Search with AUC")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (AUC): {:.3f}".format(grid.best_score_))
print("Test set AUC: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Test set accuracy: {:.3f}".format(grid.score(X_test, y_test)))
```

**Out[71]:**

```
Grid-Search with AUC
Best parameters: {'gamma': 0.01}
Best cross-validation score (AUC): 0.997
Test set AUC: 1.000
Test set accuracy: 1.000
```

Lorsque nous utilisons l'exactitude, le paramètre `gamma=0.0001` est sélectionné, tandis que nous obtenons `0.01` pour l'AUC. Dans les deux cas, le résultat est cohérent. Cependant, utiliser AUC a permis de trouver un meilleur réglage de paramètre en termes d'AUC et même d'exactitude (en fait, il s'agit probablement d'une conséquence du fait que l'exactitude est une mauvaise mesure pour les performances du modèle dans le cas de données déséquilibrées).

Les valeurs les plus importantes du paramètre `scoring` pour la classification sont `accurary` (par défaut), `roc_auc` pour l'aire sous la courbe ROC, `average_precision` pour l'aire sous la courbe précision-rappel, ainsi que `f1`, `f1_macro`, `f1_micro` et `f1_weighted` pour les scores *f*-mesure binaires et les variantes pondérées. Dans le cas de la régression, les valeurs les plus couramment utilisées sont `r2` pour le score $R^2$, `mean_squared_error` pour l'erreur quadratique moyenne et `mean_absolute_error` pour l'erreur absolue moyenne. Vous pouvez trouver une liste complète d'arguments dans la documentation de `scikit-learn` (voyez la section 3.3 du Guide utilisateur) ou en vous servant du dictionnaire `SCORER` défini dans le module `metrics.scorer` :

**In[72]:**

```
from sklearn.metrics.scorer import SCORERS
print("Available scorers:\n{}".format(sorted(SCORERS.keys())))
```

**Out[72]:**

```
Available scorers:
['accuracy', 'adjusted_mutual_info_score', 'adjusted_rand_score', 'average_
  precision', 'completeness_score', 'f1', 'f1_macro', 'f1_micro', 'f1_samples',
  'f1_weighted', 'fowlkes_mallows_score', 'homogeneity_score', 'log_loss',
  'mean_absolute_error', 'mean_squared_error', 'median_absolute_error',
  'mutual_info_score', 'neg_log_loss', 'neg_mean_absolute_error', 'neg_mean_
  squared_error', 'neg_mean_squared_log_error', 'neg_median_absolute_error',
  'normalized_mutual_info_score', 'precision', 'precision_macro', 'precision_
  micro', 'precision_samples', 'precision_weighted', 'r2', 'recall', 'recall_
  macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc',
  'v_measure_score']
```

# Résumé et perspectives

Dans ce chapitre, nous avons discuté de la validation croisée, de la recherche sur grille et des métriques d'évaluation, qui sont les pierres angulaires de l'évaluation et de l'amélioration des algorithmes d'apprentissage automatique. Les outils décrits dans ce chapitre, plus les algorithmes présentés dans les Chapitres 2 et 3, sont le pain et le beurre (certains appellent aussi cela un gagne-pain…) de tout praticien de l'apprentissage automatique.

Il y a deux points particuliers de ce chapitre qui méritent d'être répétés, car les néophytes les négligent trop souvent. Le premier concerne la validation croisée. La validation croisée, ou l'usage d'un jeu de test, nous permet d'évaluer un modèle d'apprentissage automatique tel qu'il se comportera dans le futur. Cependant, se servir du jeu de test pour sélectionner un modèle ou les paramètres de celui-ci se traduit par un « épuisement » des données de test, et le fait d'utiliser les mêmes données pour évaluer le comportement de notre modèle dans le futur va engendrer des estimations exagérément optimistes. Nous devons donc reprendre la division entre données d'apprentissage pour produire le modèle, données de validation pour affiner celui-ci et sélectionner ses paramètres, et enfin jeu de test pour l'évaluation proprement dite du modèle. Au lieu d'une simple division, nous pouvons remplacer tous ces partages par une validation croisée.

La manière la plus courante de procéder, comme nous l'avons vu plus haut dans ce chapitre, consiste à se servir d'une division apprentissage/test pour l'évaluation, et d'utiliser la validation croisée sur le jeu d'apprentissage pour la sélection du modèle et des paramètres.

Le second point concerne l'importance de la métrique d'évaluation ou de la fonction de scoring servant à la sélection du modèle et à son évaluation. La question des théories concernant la prise de décisions à partir des prédictions faites par un modèle d'apprentissage automatique déborde du cadre de ce livre. Nous conseillons vivement à ce sujet aux lecteurs lisant l'anglais de se procurer l'ouvrage de Foster Provost et Tom Fawcett, *Data Science for Business*, paru aux éditions O'Reilly (`http://shop.oreilly.com/product/0636920028918.do`). Cependant, il est rare que le but final d'une tâche d'apprentissage automatique soit de construire un modèle possédant un haut niveau d'exactitude. Vous devez donc vous assurer que la métrique que vous choisissez pour évaluer et sélectionner un modèle est parfaitement adaptée au modèle réel, tel qu'il sera effectivement employé pour répondre à l'objectif posé. En fait, les problèmes de classification ont rarement des classes équilibrées. Les faux positifs et les faux négatifs ont donc souvent des conséquences très différentes. Assurez-vous que vous comprenez ce que sont ces conséquences, et choisissez une métrique d'évaluation en accord avec ces questions.

L'évaluation du modèle et les techniques de sélection décrites dans ce qui précède sont les outils les plus importants dans la trousse du *data scientist*. La recherche sur grille et la validation croisée, telles que nous les avons décrites dans ce chapitre, ne peuvent s'appliquer qu'à un modèle supervisé unique. Cependant, nous avons vu auparavant que de nombreux modèles nécessitent un prétraitement et que, dans certaines applications telles que l'exemple de reconnaissance facial étudié au Chapitre 3, extraire une représentation différente des données peut être utile. Dans le prochain chapitre, nous allons vous présenter la classe `Pipeline`, qui nous permet d'utiliser la recherche sur grille et la validation croisée sur ces chaînes complexes d'algorithmes.

# CHAPITRE 6

# Chaînage d'algorithmes et pipelines

Pour de nombreux algorithmes d'apprentissage automatique, la manière particulière de représenter les données que vous fournissez est très importante, ainsi que nous l'avons vu au Chapitre 4. Cela commence par un recalibrage des données et la combinaison manuelle de caractéristiques, et va jusqu'à l'entraînement des caractéristiques en utilisant l'apprentissage automatique non supervisé, ainsi que nous l'avons étudié dans le Chapitre 3. Par conséquent, la plupart des applications d'apprentissage automatique ne se contentent pas d'un seul algorithme, mais impliquent le chaînage de nombreuses étapes de traitement différentes ainsi que de modèles d'apprentissage automatique variés. Dans ce chapitre, nous allons étudier l'utilisation de la classe `Pipeline` pour simplifier le processus de construction de chaînes de transformation et de modèles. En particulier, nous verrons comment combiner `Pipeline` et `GridSearchCV` pour rechercher des paramètres dans toutes les étapes de traitement simultanément.

À titre d'exemple de l'importance du chaînage de modèles, nous avons remarqué que nous pouvons grandement améliorer les performances d'un SVM à noyau sur le jeu de données `cancer` en utilisant `MinMaxScaler` pour le prétraitement. Le code suivant se charge de diviser les données, de calculer le minimum et le maximum, d'effectuer le calibrage des données et de réaliser l'entraînement du SVM :

**In[1]:**

```
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

scaler = MinMaxScaler().fit(X_train)
```

In[2]:

```
X_train_scaled = scaler.transform(X_train)

svm = SVC()
svm.fit(X_train_scaled, y_train)
X_test_scaled = scaler.transform(X_test)
print("Test score: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

Out[2]:

```
Test score: 0.95
```

## Sélection de paramètres avec prétraitement

Supposons maintenant que nous voulions trouver de meilleurs paramètres pour le SVC en utilisant `GridSearchCV`, comme nous l'avons vu dans le Chapitre 5. Une approche naïve pourrait ressembler à ceci :

In[3]:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=5)
grid.fit(X_train_scaled, y_train)
print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))
print("Best parameters: ", grid.best_params_)
print("Test set accuracy: {:.2f}".format(grid.score(X_test_scaled, y_test)))
```

Out[3]:

```
Best cross-validation accuracy: 0.98
Best parameters:  {'C': 1, 'gamma': 1}
Test set accuracy: 0.97
```

Ici, nous avons exécuté la recherche sur grille pour les paramètres du SVC en utilisant les données calibrées. Cependant, il y a alors un piège un peu subtil. Lorsque nous avons recalibré les données, nous avons utilisé *toutes les données du jeu d'apprentissage* pour en calculer le minimum et le maximum. Nous nous sommes ensuite servis de ces *données d'apprentissage recalibrées* pour lancer notre recherche sur grille en utilisant une validation croisée. Pour chaque division dans cette validation croisée, une certaine partie du jeu d'apprentissage original va être déclaré comme formant le jeu d'apprentissage de celle-ci, et une autre partie comme formant le jeu de test de cette division. Ce jeu de test sert à mesurer les performances d'un modèle entraîné à partir de cette partie « apprentissage » lorsqu'il est appliqué à de nouvelles données. Cependant, nous avons déjà utilisé les informations contenues dans la partie « test » de la division lors du recalibrage des données. Rappelez-vous que la partie « test » de chaque division dans la validation croisée provient du jeu d'apprentissage, alors que nous avons ici utilisé les informations de la totalité de ce jeu d'apprentissage pour trouver le bon calibrage des données. *C'est fondamen-*

*talement différent de la manière dont les nouvelles données se présentent devant le modèle.* Si nous observons de toutes nouvelles données, celles-ci n'auront pas été utilisées pour le recalibrage des données d'apprentissage, et elles pourront donc avoir un minimum et un maximum différents par rapport à ces dernières. L'exemple qui suit illustre comment le traitement des données au cours de la validation croisée et lors de l'évaluation finale diffère (voir la Figure 6.1) :

**In[4] :**

```
mglearn.plots.plot_improper_processing()
```

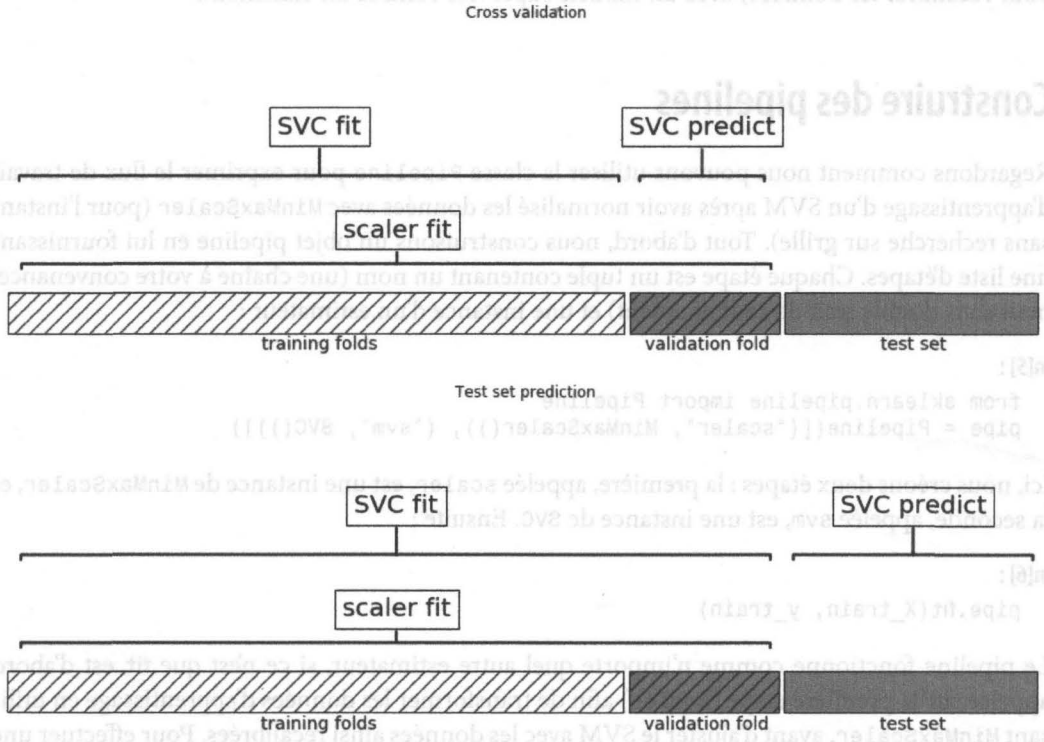



*Figure 6.1 : Utilisation des données lors d'un prétraitement en dehors de la boucle de validation croisée.*

Les divisions dans la validation croisée ne reflètent plus correctement la manière dont les nouvelles données vont se présenter au processus de modélisation. Nous avons déjà fait « fuiter » des informations depuis ces parties de données dans notre processus de modélisation. Cela va conduire à des résultats exagérément optimistes lors de la validation croisée, et possiblement à la sélection de paramètres pas du tout optimaux.

Pour éviter ce problème, la division du jeu de données au cours de la validation croisée devrait être effectuée *avant tout prétraitement*. Tout processus qui extrait des connaissances à partir du

jeu de données devrait être entraîné uniquement depuis la partie « apprentissage » de celui-ci, et donc être contenu dans la boucle de validation croisée.

Pour obtenir ce résultat dans `scikit-learn` avec les fonctions `cross_val_score` et `GridSearchCV`, nous pouvons utiliser la classe `Pipeline`. Cette classe permet en quelque sorte de « coller » bout à bout de multiples étapes de traitement dans un seul estimateur `scikit-learn`. La classe `Pipeline` elle-même possède des méthodes `fit`, `predict` et `score`, et elle se comporte exactement comme n'importe quel autre modèle dans `scikit-learn`. L'utilisation la plus courante de la classe `Pipeline` consiste à chaîner des étapes de prétraitement (par exemple pour recalibrer les données) avec un modèle supervisé comme un classifieur.

## Construire des pipelines

Regardons comment nous pouvons utiliser la classe `Pipeline` pour exprimer le flux de travail d'apprentissage d'un SVM après avoir normalisé les données avec `MinMaxScaler` (pour l'instant sans recherche sur grille). Tout d'abord, nous construisons un objet pipeline en lui fournissant une liste d'étapes. Chaque étape est un tuple contenant un nom (une chaîne à votre convenance, mais sans double trait de soulignement) et une instance d'un estimateur :

**In[5]:**

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])
```

Ici, nous créons deux étapes : la première, appelée `scaler`, est une instance de `MinMaxScaler`, et la seconde, appelée `svm`, est une instance de `SVC`. Ensuite :

**In[6]:**

```
pipe.fit(X_train, y_train)
```

Le pipeline fonctionne comme n'importe quel autre estimateur, si ce n'est que `fit` est d'abord appelée sur la première étape (`scaler`) afin de transformer les données d'apprentissage en utilisant `MinMaxScaler`, avant d'ajuster le SVM avec les données ainsi recalibrées. Pour effectuer une évaluation sur les données de test, nous appelons simplement `pipe.score` :

**In[7]:**

```
print("Test score: {:.2f}".format(pipe.score(X_test, y_test)))
```

**Out[7]:**

```
Test score: 0.95
```

Appeler la méthode `score` sur le pipeline effectue d'abord la transformation des données de test avec `scaler`, puis celles-ci sont passées à la méthode `score` du SVM. Comme vous pouvez le constater, le résultat est identique à celui obtenu manuellement au début de ce chapitre. En utilisant le pipeline, nous avons réduit le code nécessaire pour notre processus « prétraitement + clas-

sification ». Cependant, le principal bénéfice apporté par le pipeline est que nous pouvons maintenant nous servir de cet estimateur unique dans `cross_val_score` ou `GridSearchCV`.

## Utiliser des pipelines dans des recherches sur grille

Utiliser un pipeline dans une recherche sur grille fonctionne de la même manière que pour n'importe quel autre estimateur. Nous définissons une grille de paramètres à rechercher, et nous construisons un `GridSearchCV` à partir du pipeline ainsi que de cette grille. Par contre, il y a tout de même ici une légère modification de la procédure. En effet, nous devons maintenant préciser pour chaque paramètre à quelle étape du pipeline il correspond. Les deux paramètres que nous voulons ajuster, `C` et `gamma`, concernent `SVC`, donc la seconde étape. Nous avons donné à cette étape comme nom `svm`. La syntaxe à employer pour définir une grille dans un pipeline consiste à spécifier pour chaque paramètre le nom de l'étape, suivi d'un double trait de soulignement (`__`), puis du nom du paramètre concerné. Ainsi, pour le paramètre `C`, nous devons utiliser `"svm__C"` comme clé dans le dictionnaire de la grille, le même principe valant pour `gamma` :

**In[8]:**

```
param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Nous pouvons maintenant utiliser `GridSearchCV` comme d'habitude :

**In[9]:**

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))
print("Test set score: {:.2f}".format(grid.score(X_test, y_test)))
print("Best parameters: {}".format(grid.best_params_))
```

**Out[9]:**

```
Best cross-validation accuracy: 0.98
Test set score: 0.97
Best parameters: {'svm__C': 1, 'svm__gamma': 1}
```

Par contraste avec la recherche sur grille effectuée auparavant, pour chaque division dans la validation croisée, `MinMaxScaler` est maintenant réajusté uniquement à partir des données d'apprentissage, et donc aucune information ne « fuite » du jeu de test. Cette situation est illustrée sur la Figure 6.2, à comparer avec la Figure 6.1 :

**In[10] :**

```
mglearn.plots.plot_proper_processing()
```

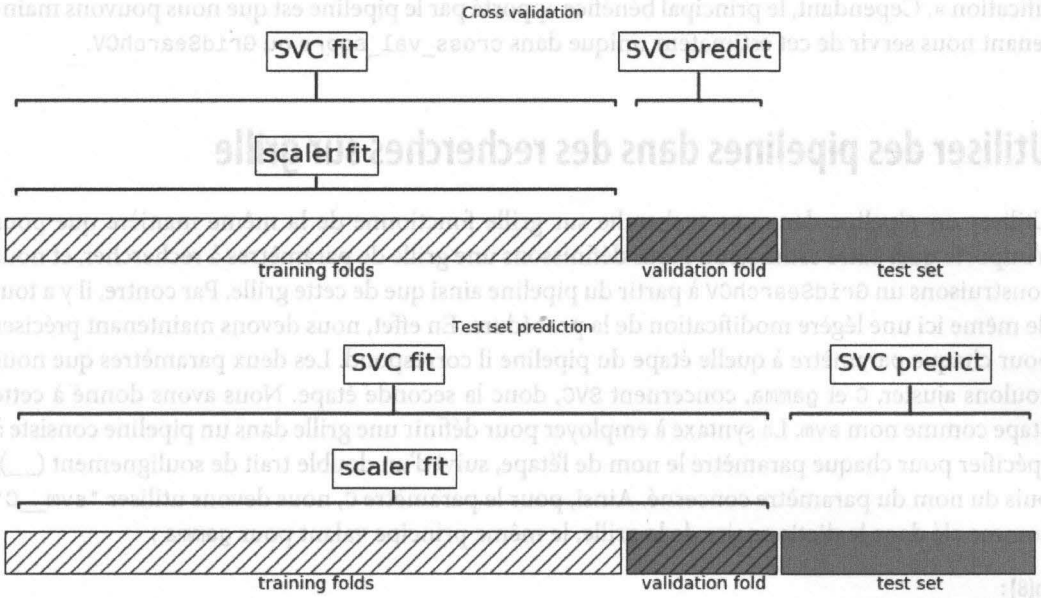



*Figure 6.2 : Utilisation des données lors d'un prétraitement à l'intérieur de la boucle de validation croisée avec un pipeline.*

L'impact produit par une « fuite » d'informations dans la validation croisée varie en fonction de la nature de l'étape de prétraitement. Estimer le calibrage des données en utilisant le pli de test n'a généralement pas un impact terrible, tandis qu'utiliser ce pli dans une extraction et une sélection de caractéristiques peut conduire à des différences substantielles dans les sorties.

## La « fuite » d'informations illustrée

Un excellent exemple de « fuite » d'informations dans une validation croisée est donné dans l'ouvrage *The Elements of Statistical Learning*, par Hastie, Tidshirani et Friedman, déjà cité dans ce livre. Nous en reproduisons ici une version adaptée. Considérons une tâche de régression synthétique avec 100 échantillons et 10 000 caractéristiques qui sont obtenues de manière indépendante à partir d'une distribution gaussienne. Nous échantillonnons également la réponse depuis une distribution gaussienne :

**In[11]:**

```
rnd = np.random.RandomState(seed=0)
X = rnd.normal(size=(100, 10000))
y = rnd.normal(size=(100,))
```

Étant donnée la manière dont est créé le jeu de données, il n'y a pas de relation entre X et la cible y, qui sont totalement indépendants. Par conséquent, il ne devrait pas être possible d'apprendre quoi que ce soit de ce jeu de données. Nous allons maintenant commencer par sélectionner les éléments réputés être les plus significatifs des 10 000 caractéristiques en utilisant la fonction de sélection `SelectPercentile`, avant d'évaluer une régression ridge en utilisant une validation croisée :

**In[12]:**

```
from sklearn.feature_selection import SelectPercentile, f_regression

select = SelectPercentile(score_func=f_regression, percentile=5).fit(X, y)
X_selected = select.transform(X)
print("X_selected.shape: {}".format(X_selected.shape))
```

**Out[12]:**

```
X_selected.shape: (100, 500)
```

**In[13]:**

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import Ridge
print("Cross-validation accuracy (cv only on ridge): {:.2f}".format(
      np.mean(cross_val_score(Ridge(), X_selected, y, cv=5))))
```

**Out[13]:**

```
Cross-validation accuracy (cv only on ridge): 0.91
```

Le score $R^2$ calculé par la validation est de 0.91, indiquant un très bon modèle. Cela ne peut clairement pas être vrai, puisque nos données sont totalement aléatoires. Ce qui s'est passé ici, c'est que notre sélection a tiré parmi les 10 000 valeurs possibles certaines caractéristiques qui sont (par chance) très bien corrélées avec la cible. Comme la sélection a été faite *en dehors* de la validation croisée, elle a pu trouver des caractéristiques qui sont en corrélation à la fois avec les plis d'apprentissage et ceux de test. Les éléments qui ont « fuité » à partir des plis de test se sont donc révélés très « informatifs », conduisant ainsi à des résultats hautement irréalistes. Comparons cela avec une validation croisée correcte utilisant un pipeline :

**In[14]:**

```
pipe = Pipeline([("select", SelectPercentile(score_func=f_regression,
                                             percentile=5)),
                 ("ridge", Ridge())])
print("Cross-validation accuracy (pipeline): {:.2f}".format(
      np.mean(cross_val_score(pipe, X, y, cv=5))))
```

**Out[14]:**

```
Cross-validation accuracy (pipeline): -0.25
```

Cette fois, nous obtenons un score $R^2$ *négatif*, ce qui indique un modèle très pauvre. Le fait d'utiliser un pipeline place maintenant la phase de sélection *à l'intérieur* de la validation croisée. Cela signifie que seuls les plis d'apprentissage peuvent servir à la sélection, pas ceux de test. Il est donc possible de trouver des caractéristiques qui sont corrélées avec la cible sur le jeu d'apprentissage, mais pas avec le jeu de test du fait que les données sont totalement aléatoires. Dans cet exemple, rectifier le problème des fuites de données fait toute la différence entre conclure qu'un modèle fonctionne très bien et conclure qu'il ne fonctionne pas du tout…

## Interface générale de la classe Pipeline

La classe `Pipeline` n'est pas restreinte au prétraitement et à la classification. Elle peut en fait relier un nombre quelconque d'estimateurs. Par exemple, vous pourriez parfaitement construire un pipeline proposant un total de quatre étapes : extraction de caractéristiques, sélection de caractéristiques, recalibrage et classification (cette dernière étape pouvant également être une régression ou un *clustering*).

La seule condition pour les estimateurs d'un pipeline, c'est que tous, sauf la dernière étape, doivent posséder une méthode `transform` de façon à ce qu'ils soient capables de produire une nouvelle représentation des données qui pourra être utilisée dans l'étape suivante.

En interne, au cours de l'appel à `Pipeline.fit`, le pipeline appelle `fit` puis `transform` (ou simplement `fit_transform`) sur chaque étape l'une après l'autre, l'entrée d'une certaine étape étant fournie par la sortie produite par la méthode `transform` de l'étape précédente. Lors de la dernière étape du pipeline, seul l'appel à `fit` est effectué.
En laissant de côté quelques détails plus subtils, cette démarche est implémentée comme ci-après. Rappelez-vous que `pipeline.steps` est une liste de tuples, autrement dit que `pipeline.steps[0][1]` est le premier estimateur, `pipeline.steps[1][1]` le second, et ainsi de suite :

**In[15]:**
```
def fit(self, X, y):
    X_transformed = X
    for name, estimator in self.steps[:-1]:
        X_transformed = estimator.fit_transform(X_transformed, y)
    self.steps[-1][1].fit(X_transformed, y)
    return self
```

Lorsque nous réalisons une prédiction en utilisant `Pipeline`, nous transformons de la même manière les données en passant par toutes les étapes sauf la dernière, puis en appelant `predict` lors de celle-ci :

**In[16]:**

```
def predict(self, X):
    X_transformed = X
    for step in self.steps[:-1]:
        X_transformed = step[1].transform(X_transformed)
    return self.steps[-1][1].predict(X_transformed)
```

Ce processus est illustré sur la Figure 6.3 pour deux transformateurs, T1 et T2, et un classifieur (appelé `Classifier`) :

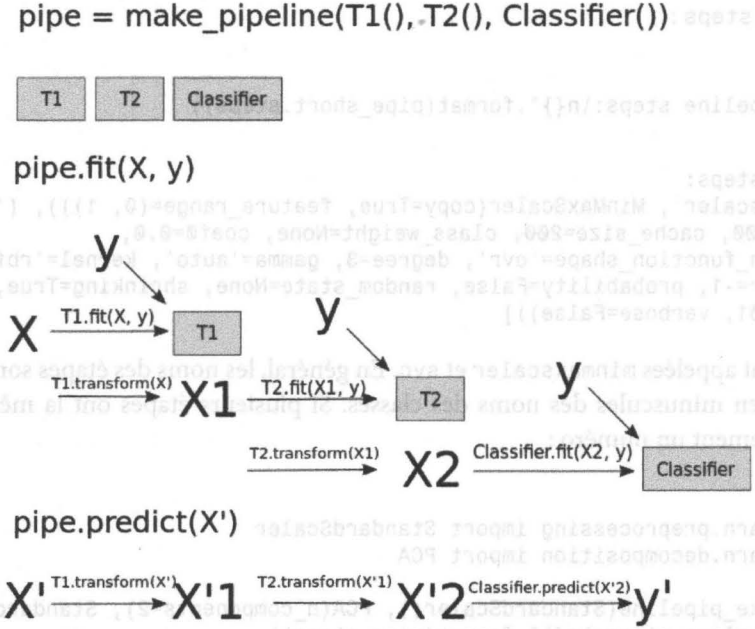



*Figure 6.3 : Vue d'ensemble du processus d'apprentissage et de prédiction du pipeline.*

Le pipeline est en fait bien plus général que cela. Il n'y a aucune raison particulière pour que la dernière étape dispose d'une fonction `predict`. Par exemple, si la dernière étape est un `PCA` (Principal Component Analysis) possédant une méthode `transform`, nous pourrions appeler cette méthode sur le pipeline pour obtenir la sortie de `PCA.transform` appliquée aux données qui ont été traitées lors de l'étape précédente. La dernière étape d'un pipeline a juste besoin de posséder une méthode `fit`.

## Créer facilement un pipeline avec make_pipeline

Créer un pipeline avec la syntaxe proposée plus haut est parfois assez pénible. En fait, bien souvent, nous n'avons pas besoin de faire appel à des noms fournis par l'utilisateur pour chaque étape. Il existe en réalité une fonction très pratique, `make_pipeline`, qui va créer un pipeline à

notre place en fournissant automatiquement un nom pour chaque étape à partir de sa classe. La syntaxe de `make_pipeline` se présente ainsi :

**In[17]:**

```
from sklearn.pipeline import make_pipeline
pipe_long = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100))])
pipe_short = make_pipeline(MinMaxScaler(), SVC(C=100))
```

Les objets `pipe_long` et `pipe_short` font exactement la même chose, mais `pipe_short` a des étapes qui sont automatiquement nommées. Nous pouvons voir le nom de ces étapes en regardant l'attribut `steps` :

**In[18]:**

```
print("Pipeline steps:\n{}".format(pipe_short.steps))
```

**Out[18]:**

```
Pipeline steps:
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('svc',
  SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False))]
```

Les étapes sont appelées `minmaxscaler` et `svc`. En général, les noms des étapes sont simplement des versions en minuscules des noms des classes. Si plusieurs étapes ont la même classe, on rajoute simplement un numéro :

**In[19]:**

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

pipe = make_pipeline(StandardScaler(), PCA(n_components=2), StandardScaler())
print("Pipeline steps:\n{}".format(pipe.steps))
```

**Out[19]:**

```
Pipeline steps:
[('standardscaler-1', StandardScaler(copy=True, with_mean=True, with_
  std=True)), ('pca', PCA(copy=True, iterated_power='auto', n_components=2,
  random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)), ('standardscaler-2',
  StandardScaler(copy=True, with_mean=True, with_std=True))]
```

Comme vous pouvez le constater, la première étape `StandardScaler` a été renommée `standardscaler-1`, et la seconde `standardscaler-2`. Cependant, dans ce type de situation, il pourrait être préférable d'utiliser la construction `Pipeline` avec des noms explicites afin de rendre chaque étape plus explicite.

## Accéder aux attributs des étapes

Souvent, vous voudriez inspecter les attributs d'une des étapes du pipeline, comme par exemple les coefficients d'un modèle linéaire ou les composants extraits par PCA. La manière la plus simple de s'y prendre consiste à faire appel à l'attribut named_steps, par exemple comme ceci :

**In[20]:**

```
pipe.fit(cancer.data)
components = pipe.named_steps["pca"].components_
print("components.shape: {}".format(components.shape))
```

**Out[20]:**

```
components.shape: (2, 30)
```

## Accéder aux attributs d'un pipeline depuis GridSearchCV

Comme nous l'avons vu plus haut dans ce chapitre, l'une des raisons principales de l'utilisation des pipelines est la recherche sur des grilles. Une tâche courante consiste à accéder à certaines des étapes d'un pipeline depuis une recherche sur grille. Voyons ce que cela peut donner avec un classifieur LogisticRegression appliqué au jeu de données cancer en utilisant la classe Pipeline ainsi que StandardScaler pour normaliser les données avant de les passer à la régression logistique. Nous allons tout d'abord créer un pipeline en utilisant la fonction make_pipeline :

**In[21]:**

```
from sklearn.linear_model import LogisticRegression

pipe = make_pipeline(StandardScaler(), LogisticRegression())
```

Nous créons ensuite une grille de paramètres. Comme cela a été expliqué dans le Chapitre 2, le paramètre de régularisation pour la régression logistique est C. Nous utilisons une grille logarithmique pour ce paramètre, avec une recherche comprise entre 0.01 et 100. Comme nous avons fait appel à la fonction make_pipeline, le nom de l'étape LogisticRegression dans le pipeline est le nom de la classe mis en minuscules, suivi d'un double trait de soulignement et du nom du paramètre, soit logisticregression__C :

**In[22]:**

```
param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10, 100]}
```

Comme d'habitude, nous divisons le jeu de données cancer en un jeu d'apprentissage et un jeu de test, et nous effectuons une recherche sur grille :

**In[23]:**

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=4)
```

```
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

Comment pouvons-nous accéder aux coefficients du meilleur modèle `LogisticRegression` trouvé par `GridSearchCV` ? Nous avons vu dans le Chapitre 5 que cette information était disponible via l'attribut `grid.best_estimator_` :

**In[24]:**

```
print("Best estimator:\n{}".format(grid.best_estimator_))
```

**Out[24]:**

```
Best estimator:
Pipeline(memory=None,
     steps=[('standardscaler', StandardScaler(copy=True, with_mean=True,
  with_std=True)), ('logisticregression', LogisticRegression(C=0.1, class_
  weight=None, dual=False, fit_intercept=True,
            intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
            penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
            verbose=0, warm_start=False))])
```

Dans notre cas, `best_estimator_` est un pipeline avec deux étapes, `standardscaler` et `logisticregression`. Pour accéder à l'étape `logisticregression`, nous pouvons utiliser l'attribut `named_steps` du pipeline, comme cela a été expliqué plus haut :

**In[25]:**

```
print("Logistic regression step:\n{}".format(
      grid.best_estimator_.named_steps["logisticregression"]))
```

**Out[25]:**

```
Logistic regression step:
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='
```

Maintenant que nous disposons de l'instance `LogisticRegression` entraînée, nous pouvons accéder aux coefficients (poids) associés à chaque caractéristique d'entrée :

**In[26]:**

```
print("Logistic regression coefficients:\n{}".format(
      grid.best_estimator_.named_steps["logisticregression"].coef_))
```

**Out[26]:**

```
Logistic regression coefficients:
[[-0.389 -0.375 -0.376 -0.396 -0.115  0.017 -0.355 -0.39  -0.058  0.209
  -0.495 -0.004 -0.371 -0.383 -0.045  0.198  0.004 -0.049  0.21   0.224
  -0.547 -0.525 -0.499 -0.515 -0.393 -0.123 -0.388 -0.417 -0.325 -0.139]]
```

Cette expression peut évidemment paraître longue, mais elle est souvent commode pour mieux comprendre les modèles.

# Étapes de prétraitement et paramètres du modèle

En utilisant des pipelines, nous pouvons encapsuler toutes les étapes de prétraitement de notre flux d'apprentissage automatique dans un seul estimateur `scikit-learn`. Un autre bénéfice apporté par cette technique est que nous pouvons maintenant *ajuster les paramètres du prétraitement* en une utilisant la sortie d'une tâche supervisée, comme une régression ou une classification. Dans de précédents chapitres, nous avons utilisé des caractéristiques polynomiales sur le jeu de données `boston` avant d'appliquer la régression ridge. Modélisons cela en faisant appel à un pipeline. Ce pipeline va contenir trois étapes : mise à l'échelle des données, calcul des caractéristiques polynomiales et régression ridge :

**In[27]:**

```
from sklearn.datasets import load_boston
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target,
                                                    random_state=0)

from sklearn.preprocessing import PolynomialFeatures
pipe = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(),
    Ridge())
```

Comment savoir quels degrés de polynômes choisir, ou même s'il faut choisir des polynômes ou des interactions ? Dans l'idéal, nous voulons sélectionner le paramètre `degree` en nous basant sur la sortie de la classification. En utilisant notre pipeline, nous pouvons effectuer une recherche sur le paramètre `degree` conjointement avec le paramètre `alpha` de ridge. Pour cela, nous définissons un `param_grid` qui contient les deux, préfixés comme nous l'avons déjà vu par les noms des étapes :

**In[28]:**

```
param_grid = {'polynomialfeatures__degree': [1, 2, 3],
              'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Nous pouvons maintenant relancer notre recherche sur grille :

**In[29]:**

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)
```

Il est possible de visualiser la sortie de la validation croisée en utilisant une carte de chaleur, comme nous l'avons fait dans le Chapitre 5 (voir la Figure 6.4) :

**In[30]:**

```
mglearn.tools.heatmap(grid.cv_results_['mean_test_score'].reshape(3, -1),
                      xlabel="ridge__alpha", ylabel="polynomialfeatures__
```

```
degree",
                    xticklabels=param_grid['ridge__alpha'],
                    yticklabels=param_grid['polynomialfeatures__degree'],
vmin=0)
```

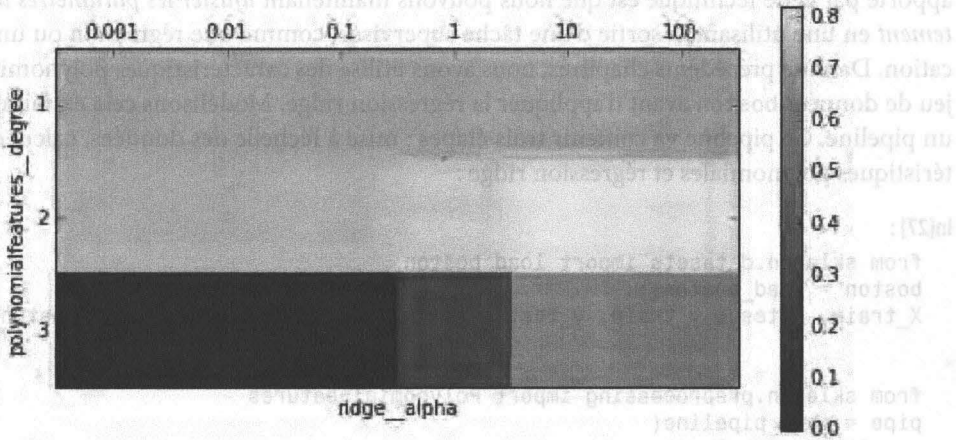



*Figure 6.4 : Carte de chaleur du score de validation moyen sous forme d'une fonction du degré des caractéristiques polynomiales et du paramètre* `alpha` *de la régression ridge.*

En regardant les résultats produits par la validation croisée, nous pouvons voir qu'utiliser des polynômes de degré deux aide, mais que les polynômes de degré trois sont ce qu'il y a de pire. Cette situation est reflétée dans les meilleurs paramètres trouvés :

**In[31]:**

```
print("Best parameters: {}".format(grid.best_params_))
```

**Out[31]:**

```
Best parameters: {'polynomialfeatures__degree': 2, 'ridge__alpha': 10}
```

Ce qui conduit au score suivant :

**In[32]:**

```
print("Test-set score: {:.2f}".format(grid.score(X_test, y_test)))
```

**Out[32]:**

```
Test-set score: 0.77
```

À titre de comparaison, lançons une recherche sur grille sans caractéristique polynomiale :

**In[33]:**

```
param_grid = {'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = make_pipeline(StandardScaler(), Ridge())
grid = GridSearchCV(pipe, param_grid, cv=5)
```

```
grid.fit(X_train, y_train)
print("Score without poly features: {:.2f}".format(grid.score(X_test, y_test)))
```

**Out[33]:**

```
Score without poly features: 0.63
```

Comme nous pouvions nous y attendre au vu des résultats affichés sur la Figure 6.4, nous n'obtenons de cette manière que des scores décidément pires.

Rechercher en même temps des paramètres de prétraitement et des paramètres du modèle est une stratégie très puissante. Par contre, il ne faut pas oublier que `GridSearchCV` essaye *toutes les combinaisons possibles* des paramètres spécifiés. Cela peut donc se traduire par une augmentation exponentielle du nombre de modèles à construire.

## Effectuer une recherche sur grille pour trouver quel modèle utiliser

Vous pouvez aller encore plus loin en combinant `GridSearchCV` et `Pipeline`. En effet, il est aussi possible d'effectuer une recherche sur les étapes du pipeline (disons pour savoir s'il faut utiliser `StandardScaler` ou `MinMaxScaler`). Cela conduit à un espace de recherche encore plus grand et devrait donc être évalué soigneusement. Essayer toutes les solutions possibles n'est généralement pas une stratégie d'apprentissage automatique viable. Voyons cependant un exemple comparant un `RandomForestClassifier` et un `SVC` sur le jeu de données `iris`. Nous savons que le `SVC` pourrait avoir besoin d'un recalibrage des données, et nous effectuons donc aussi une recherche pour opter entre utilisation de `StandardScaler` ou pas de prétraitement. Pour `RandomForestClassifier`, nous savons qu'un prétraitement n'est pas nécessaire. Nous commençons par définir le pipeline. Ici, nous nommons les étapes de manière explicite. Nous voulons deux étapes, une première pour le prétraitement, suivie d'un classifieur. Nous pouvons instancier cela en utilisant `SVC` et `StandardScaler` :

**In[34]:**

```
pipe = Pipeline([('preprocessing', StandardScaler()), ('classifier', SVC())])
```

Nous pouvons maintenant définir `param_grid` pour notre recherche. Nous voulons que le classifieur soit `RandomForestClassifier` ou `SVC`. Comme ils doivent ajuster des paramètres différents, et qu'ils ont besoin d'un prétraitement différent, nous pouvons nous servir de ce que nous avons vu dans le Chapitre 5, à la section « Effectuer des recherches sur des espaces qui ne sont pas des grilles ». Pour affecter un estimateur à une étape, nous utilisons le nom de celle-ci comme nom de paramètre. Lorsque nous voulons sauter une étape dans le pipeline (par exemple parce que nous n'avons pas besoin de prétraitement pour RandomForest), nous pouvons définir cette étape comme étant `None` :

**In[35]:**

```
from sklearn.ensemble import RandomForestClassifier

param_grid = [
    {'classifier': [SVC()], 'preprocessing': [StandardScaler(), None],
     'classifier__gamma': [0.001, 0.01, 0.1, 1, 10, 100],
     'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100]},
    {'classifier': [RandomForestClassifier(n_estimators=100)],
     'preprocessing': [None], 'classifier__max_features': [1, 2, 3]}]
```

Nous pouvons maintenant instancier et exécuter la recherche sur grille comme d'habitude, ici sur le jeu de données `cancer` :

**In[36]:**

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)

print("Best params:\n{}\n".format(grid.best_params_))
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Test-set score: {:.2f}".format(grid.score(X_test, y_test)))
```

**Out[36]:**

```
Best params:
{'classifier': SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False), 'classifier__C': 10, 'classifier__gamma': 0.01,
  'preprocessing': StandardScaler(copy=True, with_mean=True, with_std=True)}

Best cross-validation score: 0.99
Test-set score: 0.98
```

Notre pipeline montre que SVC avec le prétraitement `StandardScaler`, `C = 10` et `gamma = 0.01` fournit le meilleur résultat.

# Résumé et perspectives

Dans ce chapitre, nous vous avons présenté la classe `Pipeline`, qui est un outil généraliste servant à chaîner plusieurs étapes de traitement dans un flux d'apprentissage automatique. Les applications pratiques, concrètes, de l'apprentissage automatique ne se servent que rarement d'un modèle isolé. Elles sont pour l'essentiel des séquences d'étapes de traitement. L'utilisation de pipelines nous permet d'encapsuler de multiples étapes dans un seul objet Python qui se conforme à l'interface `scikit-learn` avec ses méthodes `fit`, `predict` et `transform`. En particulier, lors de l'évaluation d'un modèle par validation croisée et la sélection de paramètres par

recherche sur grille, l'utilisation de la classe `Pipeline` pour capturer toutes les étapes du traitement est essentielle afin de réaliser une évaluation correcte. La classe `Pipeline` permet également d'écrire du code de manière plus succincte, ce qui réduit la probabilité de commettre des erreurs, comme cela pourrait arriver lors de l'enchaînement de traitements sans cette classe (par exemple, en oubliant d'appliquer tous les transformateurs sur le jeu de test, ou en ne les appliquant pas dans le bon ordre). Choisir la bonne combinaison d'extraction des caractéristiques, de prétraitements et de modèles est presque un art, et cela nécessite souvent une démarche empirique faite d'essais et d'erreurs. En revanche, avec des pipelines, ces « essais » de multiples étapes de traitement différentes deviennent relativement simples. Faites cependant attention à ne pas rendre vos processus trop compliqués, et assurez-vous de bien évaluer si chaque composant que vous incluez dans votre modèle est réellement nécessaire.

Avec ce chapitre, nous avons terminé notre tour d'horizon des outils et des algorithmes généraux fournis par `scikit-learn`. Vous possédez maintenant tous les savoir-faire nécessaires et vous connaissez les mécanismes indispensables pour mettre en pratique l'apprentissage automatique. Dans le prochain chapitre, nous allons nous pencher plus en détail sur un type particulier de données que l'on rencontre fréquemment dans la pratique, mais qui demande une expertise particulière pour être traité correctement : les données textuelles.

CHAPITRE 7

# Travailler avec des données textuelles

Dans le Chapitre 4, nous avons vu deux types de caractéristiques qui peuvent représenter des propriétés de la donnée : des caractéristiques continues, qui représentent une quantité, et des caractéristiques catégorielles (ou qualitatives), qui sont des éléments d'une certaine liste. Il existe une troisième catégorie qui se rencontre dans de nombreuses applications : le texte. Par exemple, si nous voulons classifier un message électronique pour le ranger dans l'un des dossiers Boîte de réception ou Courrier indésirable, le contenu de cet e-mail va certainement contenir des informations importantes pour cette tâche. Dans le cas d'un service apporté aux clients, nous voulons très souvent détecter automatiquement si un message reçu est une réclamation ou une demande de renseignement. Nous pouvons utiliser la ligne Objet et le contenu d'un message pour déterminer automatiquement l'intention du client, ce qui nous permet ensuite de rétrocéder ce message au service concerné, ou encore de répondre d'une manière totalement automatisée.

Les données textuelles se présentent ordinairement sous la forme de chaînes formées de séries de caractères. Dans les quelques exemples mentionnés ci-dessus, la longueur des données de texte varie dans des proportions plus ou moins importantes. C'est clairement quelque chose de très différent des données numériques que nous avons étudiées jusqu'ici. Nous allons donc devoir traiter ces données avant de pouvoir leur appliquer nos algorithmes d'apprentissage automatique.

## Types de données représentés sous forme de chaînes de caractères

Avant de plonger au cœur des processus qui concernent la représentation des données textuelles pour l'apprentissage automatique, il est nécessaire d'aborder brièvement la question des différents types que vous pourriez rencontrer pour ces données. Les textes se présentent en général simplement sous la forme de chaînes de caractères dans les jeux de données, mais cela ne signifie

pas que tout ce qui ressemble à une chaîne doive être traité comme du texte. Une chaîne pourrait par exemple représenter des variables qualitatives, comme nous en avons déjà rencontré. Il n'est pas possible de savoir comment traiter une chaîne quelconque avant d'avoir étudié son contenu.

Vous pouvez rencontrer quatre types de données se présentant sous la forme de chaînes de caractères :

- données catégorielles ;
- chaînes libres qui peuvent être associées d'un point de vue sémantique à des catégories ;
- chaînes structurées ;
- textes.

Les *données catégorielles* proviennent d'une liste fixée. Supposons que vous collectiez vos données dans le cadre d'une enquête où vous demandez à des personnes leur couleur préférée. Pour cela, elles doivent cliquer sur un menu déroulant qui leur propose une série de réponses parmi lesquelles vos cobayes font un choix, disons entre « rouge », « vert », « bleu », « jaune », « noir », « blanc », « violet » et « marron ». Vous obtenez ainsi un jeu de données comportant exactement huit valeurs possibles, ce qui correspond très clairement à une variable catégorielle. Vous pouvez vérifier si c'est le cas pour vos données en les observant de près. Ainsi, si vous constatez qu'il y a un très grand nombre de chaînes différentes, il est peu probable que vous ayez à faire à une variable catégorielle. Vous pouvez le confirmer en déterminant le nombre de valeurs uniques dans le jeu de données, et peut-être aussi en calculant un histogramme montrant le nombre d'apparitions de chacune d'entre elles. Vous pourriez également vouloir vérifier si chaque variable correspond en réalité à une catégorie qui fasse sens pour votre application. Par exemple, à mi-course de votre enquête, quelqu'un s'aperçoit qu'il y a une faute d'orthographe dans votre liste (disons que « marron » avait par erreur été orthographié « maron »). La faute est alors corrigée, mais votre jeu de données contient maintenant les deux mots « marron » et « maron », qui peuvent être associés d'un point de vue sémantique et devraient donc être consolidés.

Imaginez maintenant qu'au lieu de fournir un menu déroulant, vous vous contentiez d'un champ de texte dans lequel les utilisateurs doivent saisir eux-mêmes le nom de leur couleur préférée. De nombreuses personnes vont répondre en tapant effectivement un nom de couleur, comme « vert » ou « bleu ». D'autres pourraient faire des fautes de frappe en saisissant par exemple « jeune » au lieu de « jaune », utiliser une orthographe un peu différente, comme « verte » ou « verts » au lieu de « vert », ou bien encore choisir des dénominations plus évocatrices et spécifiques, comme « bleu nuit ». Mais vous pourriez également rencontrer des entrées bien plus étranges, du genre « comme la peau d'un vélociraptor », ou encore « moi, c'est comme la couleur du papier peint dans le cabinet de mon dentiste. Un genre d'orange. Je m'en souviens bien parce que je le regardais fixement pendant qu'il s'approchait avec sa roulette. ». Voilà des réponses qu'il est difficile, voire impossible, de classer automatiquement selon une certaine couleur. Les réponses que vous pouvez obtenir à partir du contenu d'un champ de texte relèvent de la seconde catégorie, celle des *chaînes libres qui peuvent être associées d'un point de vue sémantique à des catégories*. Il sera probablement préférable d'encoder ces données sous forme d'une

variable catégorielle, afin de pouvoir sélectionner les catégories soit en utilisant les entrées les plus courantes, soit en définissant des catégories qui vont capturer les réponses d'une manière qui fasse sens pour votre application. Vous pourriez alors disposer de plusieurs catégories pour les couleurs « standard », peut-être une catégorie « multicolore » pour les réponses du style « des bandes rouges et vertes », ou encore une catégorie « autres » pour tout ce qui ne peut pas être classé ailleurs. Ce type de prétraitement des chaînes peut demander d'importants efforts manuels et n'est pas facile à automatiser. Si vous êtes dans une position où vous avez la possibilité d'influencer la manière dont les données sont collectées, nous vous recommandons vivement d'éviter toute méthode de saisie à la main pour les valeurs qui seront bien mieux capturées en faisant appel à des variables catégorielles.

Souvent, des valeurs entrées manuellement n'appartiennent pas à des catégories fixées à l'avance, mais ont tout de même une *structure* sous-jacente, comme une adresse, un nom de lieu ou de personne, une date, un numéro de téléphone et ainsi de suite. Ces types de chaînes sont souvent très difficiles à analyser, et leur traitement dépend essentiellement du contexte et du domaine concerné. Un traitement sémantique de ces situations déborde largement du cadre de ce livre.

La dernière catégorie regroupe les textes, disons libres, formés de successions de phrases. En nous en tenant évidemment aux textes accessibles par voie électronique, nous pourrions citer les messages postés sur Facebook ou encore les tweets, les appréciations laissées par des clients à propos d'une location, d'un restaurant, *etc.*, mais aussi le contenu de Wikipédia ou bien le projet Gutenberg et ses (environ) 54 000 livres électroniques en accès libre. Toutes ces collections de données contiennent des informations se présentant essentiellement sous la forme de phrases composées de mots (quoique, sans aucun doute, les sites Web dont les liens apparaissent dans des tweets contiennent beaucoup plus d'informations que les tweets eux-mêmes). De plus, pour des raisons pratiques et de facilité, nous allons dans ce qui suit nous appuyer sur des documents tous écrits dans une seule langue, l'anglais (même si, bien entendu, les sujets abordés dans ce chapitre valent pour toute autre langue romane, comme le français). Dans le contexte de l'analyse des textes, le jeu de données est généralement appelé le *corpus*, et chaque point de donnée, en l'occurrence un certain texte, est appelé un *document*. Ces termes sont ceux qui sont employés notamment, et pour ce qui nous concerne ici, dans le domaine du *traitement du langage naturel* (TLN).

## Exemple d'application : analyse de sentiment dans des commentaires de films

Comme fil rouge dans ce chapitre, nous allons utiliser un jeu de données formé de commentaires (ou de critiques, au choix) de films provenant du site Web IMDb (*Internet Movie Database*) et qui ont été compilées par un chercheur de l'université de Standford, en Californie, Andrew Maas. Le jeu de données est accessible à l'adresse suivante :

```
http://ai.stanford.edu/~amaas/data/sentiment/
```

Le jeu de données contient les textes des commentaires, ainsi qu'une mention indiquant si une critique est positive ou négative. Le site IMDb lui-même affiche pour chaque film une notation comprise entre 1 et 10. Pour simplifier la modélisation, cette annotation est condensée sous forme d'un jeu de données pour la classification avec deux classes, les scores de sept ou plus étant considérés comme étant « positifs », et les scores inférieurs comme étant « négatifs ». La question des avis neutres est laissée de côté, de même que celle de savoir s'il s'agit ou non d'une bonne représentation de ces informations. Nous utiliserons simplement dans ce qui suit les données telles qu'elles sont fournies par Andrew Maas.

Une fois les données téléchargées et décompressées, le jeu de données se présente sous la forme de fichiers de texte placés dans deux dossiers principaux distincts, `train` pour les données d'apprentissage et `test` pour les données de test. Chacun de ces dossiers possède à son tour deux sous-dossiers appelés respectivement *pos* et *neg* :

**In[1]:**
```
!tree -dL 2 data/aclImdb
```
**Out[1]:**
```
data/aclImdb
ÐÐÐ test
Ð ÐÐÐ neg
Ð ÐÐÐ pos
ÐÐÐ train
    ÐÐÐ neg
    ÐÐÐ pos
    ÐÐÐ unsup
6 directories
```

Le dossier *pos* contient toutes les critiques positives, chacune dans un fichier de texte distinct. Le même principe s'applique au dossier *neg* (critiques négatives). Le sous-dossier *unsup* contient les données non caractérisées que nous n'allons pas utiliser. On peut donc le supprimer :

**In[2]:**
```
!rm -r data/aclImdb/train/unsup
```

Il existe dans `scikit-learn` une fonction permettant de charger des fichiers stockés dans ce type de structure de dossiers, `load_files`. Nous appliquons d'abord la fonction `load_files` aux données d'apprentissage :

**In[3]:**
```
from sklearn.datasets import load_files
reviews_train = load_files("data/aclImdb/train/")
text_train, y_train = reviews_train.data, reviews_train.target
print("type of text_train: {}".format(type(text_train)))
print("length of text_train: {}".format(len(text_train)))
print("text_traIn[6]:\n{}".format(text_traIn[6]))
```

**Out[3]:**

```
type of text_train: <class 'list'>
length of text_train: 25000
text_traIn[6]:
b"This movie has a special way of telling the story, at first i found it
  rather odd as it jumped through time and I had no idea whats happening.<br
  /><br />Anyway the story line was although simple, but still very real and
  touching. You met someone the first time, you fell in love completely, but
  broke up at last and promoted a deadly agony. Who hasn't go through this?
  but we will never forget this kind of pain in our life. <br /><br />I would
  say i am rather touched as two actor has shown great performance in showing
  the love between the characters. I just wish that the story could be a happy
  ending."
```

Vous pouvez constater que `text_train` est une liste de 25 000 documents, chaque entrée étant une chaîne de caractères correspondant à une critique. L'exemple ci-dessus montre le contenu de l'élément d'indice 6. Vous pouvez également remarquer que le texte contient des balises de saut de ligne HTML (`<br />`). Comme ces balises n'ont probablement aucun impact sur nos modèles d'apprentissage automatique, il est préférable de nettoyer les données et de supprimer ces codes avant de faire quoi que ce soit d'autre :

**In[4]:**

```
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

Le type des entrées de `text_train` dépend de votre version de Python. Dans Python 3, ces entrées sont des `bytes` qui représentent un encodage binaire des données de type chaîne. Dans Python 2, `text_train` contient des chaînes de caractères. Discuter ici des détails concernant les différents types de chaînes de caractères dans Python dépasse notre propos, mais nous vous conseillons de vous reporter à ce que dit la documentation de Python à propos des chaînes et de l'Unicode (voyez à ce sujet les adresses `https://docs.python.org/2/howto/unicode.html` et/ou `https://docs.python.org/3/howto/unicode.html`).

Le jeu de données a été collecté de manière à ce qu'il y ait autant d'éléments dans les deux classes, positive et négative, et donc autant de chaînes positives que de chaînes négatives :

**In[5]:**

```
print("Samples per class (training): {}".format(np.bincount(y_train)))
```

**Out[5]:**

```
Samples per class (training): [12500 12500]
```

Nous chargeons de la même manière le jeu de test :

**In[6]:**

```
reviews_test = load_files("data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Number of documents in test data: {}".format(len(text_test)))
print("Samples per class (test): {}".format(np.bincount(y_test)))
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]
```

**Out[6]:**

```
Number of documents in test data: 25000
Samples per class (test): [12500 12500]
```

La tâche que nous voulons nous assigner est la suivante : étant donnée une critique, l'objectif est d'affecter une étiquette « positif » ou « négatif » basée sur le contenu du texte de cette critique. Il s'agit d'une tâche standard de classification binaire. Cependant, les données textuelles ne sont pas présentées dans un format qu'un modèle d'apprentissage automatique puisse traiter. Nous avons donc besoin de convertir ce format en une représentation numérique à laquelle nous pourrons appliquer nos algorithmes d'apprentissage automatique.

## Représenter les données textuelles sous forme de sacs de mots

L'une des méthodes les plus simples, mais aussi les plus efficaces et les plus courantes, servant à transformer les textes afin de leur donner un format reconnaissable pour l'apprentissage automatique, consiste à utiliser une représentation dite par *sacs de mots* (*bag of words* en anglais). Lorsque nous utilisons cette représentation, nous éliminons l'essentiel de la structure du texte présenté en entrée, comme les chapitres, les paragraphes, les phrases et tout ce qui concerne le formatage, et nous comptons uniquement *le nombre d'apparitions de chaque mot dans chaque texte* du corpus. D'où la vision mentale que l'on peut se faire de ce processus, comme si l'on jetait ces quantités de mots dans une sorte de sac.

Calculer la représentation par sac de mots d'un corpus de documents consiste à suivre une succession de trois étapes :

- *Tokenization.* Ce terme, difficilement traduisible, désigne le fait de partager un document dans les mots qui le composent (ou *tokens*, soit jetons ou encore signes, symboles), par exemple pour distinguer les espaces ou encore les signes de ponctuation. Un texte devient alors une séquence de *tokens*.
- *Construction du vocabulaire.* Cela consiste à former un vocabulaire en collectant tous les mots qui apparaissent dans les documents afin de constituer un dictionnaire, et à les numéroter (par exemple dans l'ordre alphabétique).
- *Encodage.* Pour chaque document, cette étape consiste à dénombrer le nombre d'occurrences de chaque mot du vocabulaire.

Les étapes 1 et 2 impliquent quelques subtilités sur lesquelles nous reviendrons plus en détail dans ce chapitre. Pour l'instant, voyons comment nous pouvons appliquer cette technique de sac de mots en utilisant `scikit-learn`. La Figure 7.1 illustre ce processus sur une chaîne de caractères spécifique.

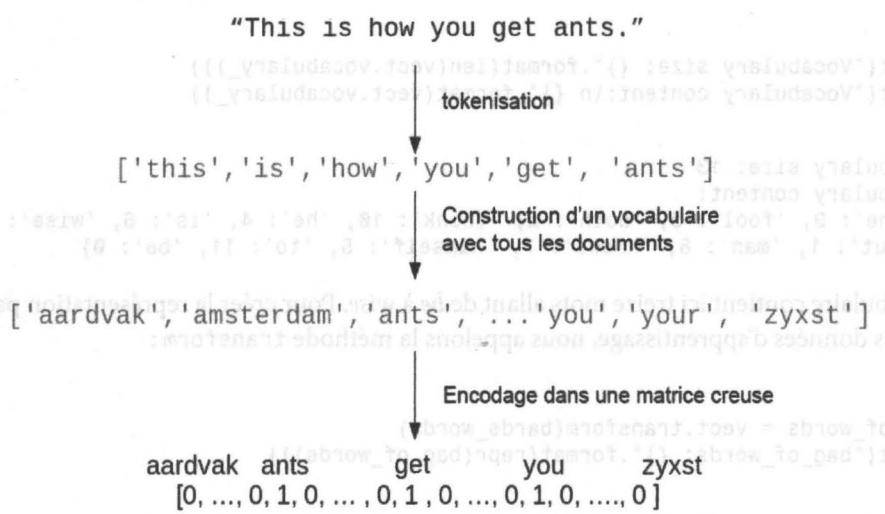



*Figure 7.1 : Processus de représentation par sac de mots.*

La sortie est un vecteur de dénombrement de mots pour chaque document. Pour chaque mot du vocabulaire, nous obtenons un décompte du nombre de fois où il apparaît dans chaque document. Cela signifie que notre représentation numérique possède une caractéristique pour chaque mot unique de l'ensemble du jeu de données. Remarquez en passant que l'ordre des mots dans la chaîne d'origine n'a absolument aucun rapport avec la représentation par sac de mots.

## Appliquer une représentation par sac de mots à un jeu de données jouet

La représentation par sac de mots est implémentée dans `CountVectorizer`, qui est un transformateur. Commençons par l'appliquer à un jeu de données jouet (ou *toy dataset*) consistant en deux échantillons afin de voir comment cela fonctionne :

**In[7]:**

```
bards_words =["The fool doth think he is wise,",
              "but the wise man knows himself to be a fool"]
```

Nous importons et instancions `CountVectorizer`, puis nous appliquons la fonction `fit` à notre jeu de données de la manière suivante :

**In[8]:**

```
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
```

De cette manière, nous « *tokenisons* » les données d'apprentissage et nous construisons le vocabulaire auquel nous pouvons ensuite accéder *via* l'attribut `vocabulary_` :

**In[9]:**

```
print("Vocabulary size: {}".format(len(vect.vocabulary_)))
print("Vocabulary content:\n {}".format(vect.vocabulary_))
```

**Out[9]:**

```
Vocabulary size: 13
Vocabulary content:
 {'the': 9, 'fool': 3, 'doth': 2, 'think': 10, 'he': 4, 'is': 6, 'wise': 12,
  'but': 1, 'man': 8, 'knows': 7, 'himself': 5, 'to': 11, 'be': 0}
```

Le vocabulaire contient ici treize mots allant de *be* à *wise*. Pour créer la représentation par sac de mots des données d'apprentissage, nous appelons la méthode `transform` :

**In[10]:**

```
bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))
```

**Out[10]:**

```
bag_of_words: <2x13 sparse matrix of type '<class 'numpy.int64'>'
   with 16 stored elements in Compressed Sparse Row format>
```

La représentation par sac de mots est placée dans une matrice creuse SciPy qui n'enregistre que les entrées non nulles (voyez à ce sujet le Chapitre 1). Cette matrice a une taille de 2 x 13, avec une ligne pour chacun des deux points de donnée et une colonne pour chaque mot du vocabulaire. On utilise une matrice creuse, car la plupart des documents ne contiendront qu'un petit sous-ensemble des mots du vocabulaire, ce qui implique que l'essentiel des colonnes sera nul. Songez au nombre possible de mots rencontrés dans une critique de film comparé à la taille d'un dictionnaire entier (qui est ce que le vocabulaire modélise) ! Stocker tous ces zéros serait évidemment prohibitif et un immense gaspillage de mémoire. Pour observer le contenu réel de la matrice creuse, nous pouvons la convertir en un tableau NumPy « dense » (qui va donc aussi contenir toutes les entrées nulles) à l'aide de la méthode `toarray` (ce qui n'est possible que parce que notre jeu ne contient que 13 mots, un grand nombre de données risquant de se traduire par une erreur `MemoryError`) :

**In[11]:**

```
print("Dense representation of bag_of_words:\n{}".format(
    bag_of_words.toarray()))
```

**Out[11]:**

```
Dense representation of bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

Nous pouvons voir que le décompte du nombre de mots vaut 0 ou 1. Autrement dit, aucune des deux chaînes de `bards_words` ne contient deux fois le même mot. La première chaîne est représentée sur la première ligne, et elle contient le premier mot du vocabulaire, *be*, zéro fois. Elle contient également zéro fois le second mot du vocabulaire, *but*. Le troisième mot, *doth*, est

trouvé une fois, et ainsi de suite. En continuant nos observations, nous remarquons également que le quatrième mot, *fool*, ainsi que le dixième, *the*, et le treizième, *wise*, apparaissent dans les deux chaînes.

## La représentation par sac de mots appliquée aux critiques de films

Maintenant que nous avons vu en détail le processus de construction des sacs de mots, nous pouvons l'appliquer à notre tâche d'*analyse de sentiment* dans les appréciations de films (le *sentiment* n'est pas à prendre ici dans le sens de sentimental, mais en tant qu'opinion, avis ou encore ressenti). Nous avons plus haut chargé nos données d'apprentissage et de test provenant de la base IMDb dans des listes de chaînes (`text_train` et `text_test`), que nous allons maintenant traiter :

**In[12]:**

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))
```

**Out[12]:**

```
X_train:
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'
   with 3431196 stored elements in Compressed Sparse Row format>
```

La taille de `X_train`, la représentation par sac de mots des données d'apprentissage, est de 25 000 x 74 849, ce qui indique que le vocabulaire contient 74 849 entrées. À nouveau, les données sont enregistrées sous forme d'une matrice creuse SciPy. Regardons d'un peu plus près ce vocabulaire. Une autre manière d'y accéder consiste à utiliser la méthode `get_feature_name` du *vectorizer* (littéralement *vectorisateur*), qui retourne une liste dans laquelle chaque entrée correspond à une caractéristique :

**In[13]:**

```
feature_names = vect.get_feature_names()
print("Number of features: {}".format(len(feature_names)))
print("First 20 features:\n{}".format(feature_names[:20]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 2000th feature:\n{}".format(feature_names[::2000]))
```

**Out[13]:**

```
Number of features: 74849
First 20 features:
['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830',
  '006', '007', '0079', '0080', '0083', '0093638', '00am', '00pm', '00s', '01',
  '01pm', '02']
Features 20010 to 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw',
  'drawback', 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings', 'drawl',
  'drawled', 'drawling', 'drawn', 'draws', 'draza', 'dre', 'drea']
```

```
Every 2000th feature:
['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bête', 'chicanery',
 'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit',
 'freezer', 'goldman', 'hasan', 'huitieme', 'intelligible', 'kantrowitz',
 'lawful', 'maars', 'megalunged', 'mostey', 'norrland', 'padilla', 'pincher',
 'promisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse',
 'subset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

Comme vous pouvez le remarquer, les dix premières entrées sont toutes des nombres, ce qui pourrait peut-être paraître un peu surprenant. En fait, tous ces nombres apparaissent quelque part dans les critiques, et ils sont donc extraits en tant que mots dans le vocabulaire. La plupart de ces mots n'ont aucune signification sémantique évidente, sauf sans doute *007*, qui, dans ce contexte cinématographique, fait vraisemblablement référence au personnage de James Bond (une rapide analyse des données confirmerait cette hypothèse, mais nous vous laissons le soin d'effectuer vous-même cette recherche). Trouver (ou non) un sens à des « mots » qui n'en ont *a priori* pas est quelque chose qui peut être très difficile. En continuant à observer notre vocabulaire, nous trouvons aussi une collection de mots anglais qui commencent par *dra*. Vous pouvez remarquer que, par exemple, les formes au singulier et au pluriel de *draught*, *drawback* ou *drawer* sont enregistrées en tant que mots distincts. Or ces formes sont évidemment très proches sur le plan sémantique, et les compter en tant que termes différents dans le vocabulaire, et donc en tant que caractéristiques différentes, n'est certainement pas idéal.

Avant d'améliorer notre extraction de caractéristiques, nous allons chercher une mesure quantitative des performances en construisant un classifieur. Les étiquettes des données d'apprentissage sont enregistrées dans `y_train` et la représentation par sac de mots des données du jeu d'apprentissage se trouvent dans `X_train`. Nous sommes dans un schéma classique. Dans le cas, comme ici, de données creuses de grande taille, les modèles linéaires tels que LogisticRegression sont souvent ce qui fonctionne le mieux.

Commençons par évaluer `LogisticRegression` en utilisant une validation croisée :



Le lecteur attentif aura sans doute remarqué que nous violons ici ce qui a été expliqué dans le Chapitre 6 sur la validation croisée avec prétraitement. En utilisant les valeurs par défaut de `CountVectorizer`, elle ne collecte en réalité aucune statistique, ce qui fait que nos résultats sont valides. Utiliser dès le début un pipeline serait un meilleur choix pour le développement d'applications, mais nous laissons ici cette question de côté pour faciliter l'exposé.

**In[14]:**

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
print("Mean cross-validation accuracy: {:.2f}".format(np.mean(scores)))
```

**Out[14]:**

```
Mean cross-validation accuracy: 0.88
```

Nous obtenons un score de validation croisée moyen de 88 %, ce qui correspond à des performances raisonnables pour une tâche de classification binaire équilibrée. Nous savons que `LogisticRegression` a un paramètre de régularisation, `C`, que nous pouvons essayer d'améliorer *via* la validation croisée :

**In[15]:**

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters: ", grid.best_params_)
```

**Out[15]:**

```
Best cross-validation score: 0.89
Best parameters:  {'C': 0.1}
```

Nous obtenons un score de validation croisée de 89 % en utilisant `C = 0.1`. Nous pouvons maintenant accéder aux performances de généralisation pour cette valeur de paramètre sur le jeu de test :

**In[16]:**

```
X_test = vect.transform(text_test)
print("Test score: {:.2f}".format(grid.score(X_test, y_test))
```

**Out[16]:**

```
Test score: 0.88
```

Voyons maintenant si nous pouvons améliorer l'extraction des mots. `CountVectorizer` extrait les « mots » (les *tokens*) en utilisant une expression régulière. Par défaut, l'expression régulière utilisée est `"\b\w\w+\b"`. Si vous n'êtes pas familiarisé avec les expressions régulières, sachez que celle-ci trouve toutes les séquences de caractères qui comportent au moins deux lettres ou chiffres («\w») et qui sont séparées du reste par des délimiteurs de mots («\b»). Les mots formés d'une seule lettre sont donc laissés de côté, des termes ou expressions tels que « doesn't » ou « bit.ly » sont scindés en deux, tandis que quelque chose du genre « h8ter » sera traité comme un mot unique. `CountVectorizer` convertit ensuite tous les mots en lettres minuscules, de manière par exemple à ce que *soon*, *Soon* ou encore *sOon* soient traités comme un même mot (et donc comme une même caractéristique). Ce mécanisme simple fonctionne assez bien en pratique, mais, comme nous l'avons vu plus haut, cela conserve de multiples caractéristiques ne fournissant pas d'information (par exemple les nombres). Une manière de réduire ce problème consiste à n'utiliser que des « mots » (ou des *tokens*) qui apparaissent dans au moins deux documents (ou dans au moins cinq documents, et ainsi de suite). En effet, un mot qui ne figurerait que dans un seul document a une faible probabilité d'apparaître dans le jeu de test, et n'est donc pas particulièrement utile. Nous pouvons définir le nombre minimum de documents requis avec le paramètre `min_df` :

**In[17]:**

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train with min_df: {}".format(repr(X_train)))
```

**Out[17]:**

```
X_train with min_df: <25000x27271 sparse matrix of type '<class 'numpy.int64'>'
   with 3354014 stored elements in Compressed Sparse Row format>
```

En exigeant au moins cinq apparitions de chaque token dans des documents différents, nous diminuons nettement le nombre de caractéristiques (27 271), soit environ un tiers de la sortie obtenue précédemment. Refaisons un passage :

**In[18]:**

```
feature_names = vect.get_feature_names()

print("First 50 features:\n{}".format(feature_names[:50]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 700th feature:\n{}".format(feature_names[::700]))
First 50 features:
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08',
  '09', '10', '100', '1000', '100th', '101', '102', '103', '104', '105', '107',
  '108', '10s', '10th', '11', '110', '112', '116', '117', '11th', '12', '120',
  '12th', '13', '135', '13th', '14', '140', '14th', '15', '150', '15th', '16',
  '160', '1600', '16mm', '16s', '16th']
Features 20010 to 20030:
['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions',
  'repetitious', 'repetitive', 'rephrase', 'replace', 'replaced',
  'replacement', 'replaces', 'replacing', 'replay', 'replayable', 'replayed',
  'replaying', 'replays', 'replete', 'replica']
Every 700th feature:
['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered',
  'cheese', 'commitment', 'courts', 'deconstructed', 'disgraceful', 'dvds',
  'eschews', 'fell', 'freezer', 'goriest', 'hauser', 'hungary', 'insinuate',
  'juggle', 'leering', 'maelstrom', 'messiah', 'music', 'occasional',
  'parking', 'pleasantville', 'pronunciation', 'recipient', 'reviews', 'sas',
  'shea', 'sneers', 'steiger', 'swastika', 'thrusting', 'tvs', 'vampyre',
  'westerns']
```

Il y a clairement nettement moins de nombres, et certains mots les plus obscurs ou mal orthographiés se sont évaporés. Voyons donc maintenant quelles sont les performances de notre modèle en effectuant à nouveau une recherche sur grille :

**In[19]:**

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

**Out[19]:**

```
Best cross-validation score: 0.89
```

Le résultat est toujours de 89 %, donc inchangé par rapport à ce que nous avons déjà vu. Nous n'avons pas amélioré notre modèle, mais, en revanche, nous avons moins de fonctionnalités susceptibles de ralentir le traitement, et nous avons aussi éliminé bon nombre de caractéristiques inutiles, ce qui pourrait rendre le modèle plus interprétable.



Si la méthode `transform` de `CountVectorizer` est appelée sur un document contenant des mots qui ne se retrouvent pas dans les données d'apprentissage, ceux-ci sont ignorés puisqu'ils ne font pas partie du dictionnaire. Cela n'est généralement pas un problème pour la classification du fait qu'il n'est pas possible d'apprendre quoi que ce soit avec des mots qui ne se trouvent pas dans les données d'apprentissage. Avec certaines applications, par exemple pour la détection de spams, il pourrait cependant être utile d'ajouter une fonctionnalité servant à mesurer le nombre de mots « hors vocabulaire » trouvés dans un document particulier. Jusqu'à présent, `scikit-learn` n'implémente pas ce type de fonction, mais il n'est pas très dur de l'écrire soi-même. Vous devez par contre vous assurer que vous restreigniez bien le vocabulaire d'une certaine manière. Sinon, aucun mot ne sera « hors vocabulaire » lors de l'entraînement.

## Mots vides (*stop words*)

Une autre manière d'éliminer des mots sans intérêt consiste à supprimer ceux qui sont tellement courants qu'ils n'apportent aucune information utile (par exemple les articles définis). On rencontre principalement deux approches : utiliser une liste de mots vides spécifiques à un langage (voir par exemple l'adresse `http://www.ranks.nl/stopwords/french`), ou éliminer des mots qui apparaissent trop fréquemment. En fait, `scikit-learn` possède une liste de mots vides pour l'anglais dans son module `feature_extraction.text` (la liste que vous obtenez peut se présenter différemment du résultat ci-après, mais cela ne change rien au contenu de la base de mots vides de `scikit-learn`) :

**In[20]:**

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Number of stop words: {}".format(len(ENGLISH_STOP_WORDS)))
print("Every 10th stopword:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))
```

**Out[20]:**

```
Number of stop words: 318
Every 10th stopword:
['they', 'of', 'who', 'found', 'none', 'co', 'full', 'otherwise', 'never',
 'have', 'she', 'neither', 'whereby', 'one', 'any', 'de', 'hence',
 'wherever', 'whose', 'him', 'which', 'nine', 'still', 'from', 'here', 'what',
 'everything', 'us', 'etc', 'mine', 'find', 'most']
```

Clairement, supprimer les mots vides du vocabulaire ne peut que réduire le nombre de caractéristiques de la longueur de la liste, soit ici 318. En revanche, cette technique peut conduire à une amélioration des performances. Voyons cela de plus près :

In[21]:

```
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("X_train with stop words:\n{}".format(repr(X_train)))
```

Out[21]:

```
X_train with stop words:
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'
   with 2149958 stored elements in Compressed Sparse Row format>
```

Le jeu de données contient maintenant 305 caractéristiques de moins (27 271 – 26 966), ce qui signifie que la plupart des mots vides de la liste, mais pas tous, apparaissent dans nos documents. Lançons à nouveau la recherche sur grille :

In[22]:

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

Out[22]:

```
Best cross-validation score: 0.88
```

La performance obtenue lors de la recherche sur grille a très légèrement diminué avec la suppression des mots vides, mais pas assez pour s'en inquiéter. En fait, étant donné qu'exclure 305 caractéristiques sur plus de 27 000 ne risque pas vraiment de modifier les performances ou l'interprétation des résultats, l'utilisation de cette liste ne semble pas vraiment valoir la peine ici. Ce genre de liste est surtout utile avec de petits jeux de données, qui pourraient ne pas contenir suffisamment d'informations pour que le modèle soit capable de déterminer quels sont les mots vides à partir des données elles-mêmes. À titre d'exercice, vous pouvez essayer l'autre approche, en éliminant les mots qui apparaissent trop fréquemment, en définissant l'option `max_df` de `CountVectorizer` afin de voir en quoi cela influence le nombre de caractéristiques et les performances.

## Pondérer les données avec tf-idf

Au lieu de rejeter des caractéristiques qui sont considérées comme n'étant pas importantes, une autre approche consiste à les pondérer en fonction du niveau d'informations que nous en attendons. L'une des techniques les plus courantes consiste à utiliser la méthode dite tf-idf (pour *term frequency-inverse document frequency*, soit *fréquence de terme – fréquence inverse de document*). L'intuition formulée par cette méthode est de donner un poids élevé aux termes qui apparaissent souvent dans un document particulier, mais pas dans de nombreux documents du corpus. Si un mot apparaît souvent dans un certain document, donc s'il y a une fréquence élevée, mais très rarement ailleurs, il est probablement très représentatif du contenu de *ce* document. `scikit-learn` implémente la méthode tf-idf dans deux classes : `TfidfTransformer`, qui prend la matrice

creuse produite par `CountVectorizer` et la transforme, et `TfidfVectorizer`, qui prend les données textuelles et effectue l'extraction de la représentation par sac de mots ainsi que la transformation tf-idf. Il existe plusieurs variantes de ce schéma. Vous pouvez consulter à ce sujet les pages Wikipédia consacrées à la méthode tf-idf : `https://fr.wikipedia.org/wiki/TF-IDF` (en français) et/ou `https://en.wikipedia.org/wiki/Tf-idf` (en anglais, mais plus détaillée).

Le score tf-idf pour le mot *w* dans le document *d* tel qu'il est implémenté dans les deux classes `TfidfTransformer` et `TfidfVectorizer` est donné par la formule (indiquée uniquement pour être assez complets sur le sujet) :

$$\text{tfidf}(w,d) = \text{tf} * \log\left(\frac{N+1}{N_w+1}\right)+1$$

Ici, $N$ est le nombre de documents dans le jeu d'apprentissage, $N_w$ le nombre de documents du jeu d'apprentissage dans lesquels le mot *w* apparaît, et *tf* le nombre d'occurrences (la fréquence du terme) du mot *w* dans le document considéré *d* (celui que nous voulons transformer ou encoder). Les deux classes appliquent aussi une normalisation L2 après avoir calculé la représentation tf-idf (cette question déborde du cadre de ce livre, mais indiquons simplement que l'application de cette norme euclidienne implique que la longueur d'un document, c'est-à-dire le nombre de mots qui le composent, ne change pas la représentation vectorisée).

Du fait que tf-idf utilise en réalité les propriétés statistiques du jeu d'apprentissage, nous allons nous servir d'un pipeline (étudié au Chapitre 6) pour nous assurer que les résultats de notre recherche sur grille sont valides. Cela nous conduit au code suivant :

**In[23]:**

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5, norm=None),
                     LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

**Out[23]:**

```
Best cross-validation score: 0.89
```

Dans ce cas, tf-idf n'a pas d'impact. Nous pouvons aussi inspecter quels mots cette méthode a déterminés comme étant les plus importants. N'oubliez pas que tf-idf a pour but de trouver des mots qui permettent de distinguer des documents, mais il s'agit d'une technique purement non supervisée. Par conséquent, le terme « importants » n'a pas nécessairement de rapport avec les notions d'appréciation positive ou négative, donc celles qui nous intéressent en fait. Commençons par extraire `TfidfVectorizer` du pipeline :

**In[24]:**

```
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
X_train = vectorizer.transform(text_train)
max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
feature_names = np.array(vectorizer.get_feature_names())

print("Features with lowest tfidf:\n{}".format(
      feature_names[sorted_by_tfidf[:20]]))

print("Features with highest tfidf: \n{}".format(
      feature_names[sorted_by_tfidf[-20:]]))
```

**Out[24]:**

```
Features with lowest tfidf:
['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred'
 'currently' 'altogether' 'nearby' 'undoubtedly' 'directs' 'fond' 'stinker'
 'avoided' 'emphasis' 'commented' 'disappoint' 'realizing' 'downhill'
 'inane']
Features with highest tfidf:
['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur'
 'vargas' 'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria'
 'khouri' 'zizek' 'rob' 'timon' 'titanic']
```

Les mots avec une basse valeur de tf-idf sont soit très courants dans l'ensemble des documents, soit utilisés seulement de manière modérée et uniquement dans de très longs documents. De manière intéressante, la plupart des résultats correspondant à une valeur tf-idf élevée identifient en fait des films précis. Ces termes n'apparaissent que dans des critiques ou commentaires laissés pour un certain film, ou une certaine franchise de films, mais ils ont par contre tendance à apparaître fréquemment dans ces documents. Cela est très clair, par exemple, pour les termes *homer*, *bridget* ou encore *titanic*. Ces mots ne vont probablement pas nous aider dans notre tâche de classification des avis (sauf si les opinions sont universellement uniformes), mais ils fournissent certainement des tas d'informations spécifiques sur les critiques concernées.

Nous pouvons également trouver les mots qui ont une faible fréquence inverse de document (idf), c'est-à-dire ceux qui apparaissent fréquemment et sont donc considérés comme étant moins importants. Les valeurs de fréquence inverse de document trouvées sur le jeu d'apprentissage sont mémorisées dans l'attribut `idf_` :

**In[25]:**

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("Features with lowest idf:\n{}".format(
      feature_names[sorted_by_idf[:100]]))
```

**Out[25]:**

```
Features with lowest idf:
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'
 'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'
 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'
```

```
'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'
'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'
'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'
'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'
'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'
'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'
'him']
```

Comme on pouvait s'y attendre, on trouve ici essentiellement des mots vides en anglais, comme *the* ou *no*. Mais certains relèvent aussi du champ cinématographique, comme *movie*, *film*, *story* et ainsi de suite. Il est aussi intéressant de constater que les mots *good*, *great* et *bad* font partie des plus fréquents, donc aussi des moins pertinents selon la mesure tf-idf, même si nous pouvons les considérer comme étant très importants pour la tâche que nous nous sommes fixée.

# Investiguer les coefficients du modèle

Finalement, regardons un peu plus en détail ce que notre modèle de régression logistique a pu apprendre des données. Comme le nombre de caractéristiques est très élevé (27 271 une fois retirées les moins fréquentes), nous ne pouvons clairement pas observer tous les coefficients en même temps. Cependant, il est possible de regarder les principaux coefficients, et voir à quels mots ils correspondent. Nous utiliserons pour cela le dernier modèle que nous avons entraîné, basé sur la détermination de la valeur tf-idf.

Le graphique de la Figure 7.2 montre les 25 coefficients les plus grands, et les 25 plus petits du modèle de régression logistique, les barres montrant la taille de chaque coefficient :

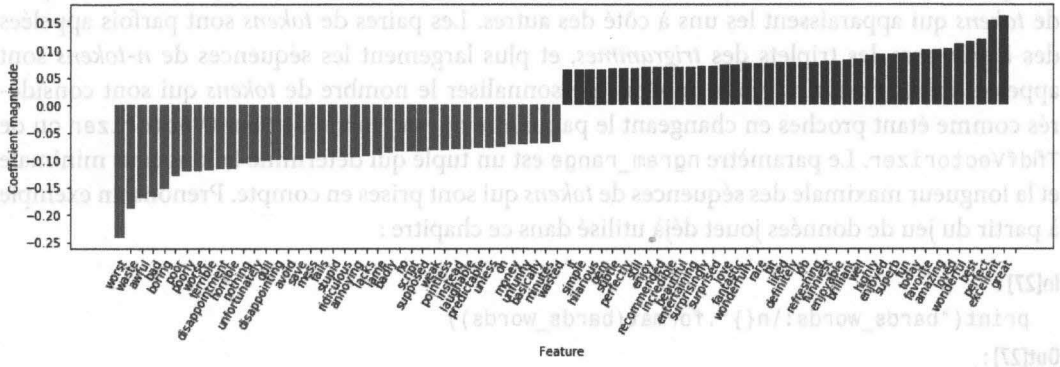



*Figure 7.2 : Plus grands et plus petits coefficients de la régression logistique entraînée sur les caractéristiques tf-idf.*

**In[26]:**

```
mglearn.tools.visualize_coefficients(
    grid.best_estimator_.named_steps["logisticregression"].coef_,
    feature_names, n_top_features=40)
```

Les coefficients négatifs, à gauche, correspondent à des mots qui, selon le modèle, indiquent des critiques négatives. De même, les coefficients positifs, à droite, correspondent à des mots qui, selon le modèle, indiquent des critiques positives. La plupart des termes sont assez intuitifs. Par exemple, *worst* (pire), *waste* (gaspillage, pour être gentil), *disappointment* (déception) ou encore *laughable* (ridicule) indiquent sans aucun doute des appréciations négatives, tandis que *excellent*, *wonderful* (formidable), *enjoyable* (agréable) ou bien *refreshing* (rafraîchissant) dénotent certainement des avis positifs. Certains mots sont un peu moins clairs, comme *bit* (un peu), *job* (travail) ou *today* (aujourd'hui), mais il est tout à fait possible qu'ils fassent partie d'expressions comme *good job* ou encore *best today*.

## Des sacs avec plusieurs mots (*n*-grammes)

L'un des principaux désavantages de l'utilisation d'une représentation par sac de mots, c'est que l'ordre de ceux-ci est totalement laissé de côté. Prenons un petit exemple, cette fois en français, en comparant deux chaînes : « c'est mal, pas bien du tout » et « c'est bien, pas mal du tout ». La représentation par sac de mots sera exactement la même, alors que le sens des deux phrases est totalement opposé. Placer « pas » devant un autre mot, même s'il s'agit d'un exemple un peu extrême, montre la façon dont le contexte intervient. Heureusement, il existe une manière de capturer ce contexte dans une représentation par sac de mots qui consiste à prendre en compte non seulement la fréquence d'un certain *token*, mais aussi celle de paires ou de triplets de *tokens* qui apparaissent les uns à côté des autres. Les paires de *tokens* sont parfois appelées des *bigrammes*, les triplets des *trigrammes*, et plus largement les séquences de *n-tokens* sont appelées des *n-grammes*. Nous pouvons personnaliser le nombre de *tokens* qui sont considérés comme étant proches en changeant le paramètre `ngram_range` de `CountVectorizer` ou de `TfidfVectorizer`. Le paramètre `ngram_range` est un tuple qui détermine la longueur minimale et la longueur maximale des séquences de *tokens* qui sont prises en compte. Prenons un exemple à partir du jeu de données jouet déjà utilisé dans ce chapitre :

**In[27]:**

```
print("bards_words:\n{}".format(bards_words))
```

**Out[27]:**

```
bards_words:
['The fool doth think he is wise,', 'but the wise man knows himself to be a
  fool']
```

Par défaut, il s'agit de créer une caractéristique par séquence comportant au moins un *token*, et au plus aussi un *token*, soit au final exactement un *token* (ce que l'on appelle aussi un *unigramme*) :

**In[28]:**

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

**Out[28]:**

```
Vocabulary size: 13
Vocabulary:
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the',
  'think', 'to', 'wise']
```

Si nous voulons uniquement regarder des *bigrammes*, donc des séquences de deux *tokens* qui se suivent, nous pouvons définir la valeur de `ngram_range` avec la valeur `(2, 2)` :

**In[29]:**

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

**Out[29]:**

```
Vocabulary size: 14
Vocabulary:
['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to', 'is
  wise', 'knows himself', 'man knows', 'the fool', 'the wise', 'think he', 'to
  be', 'wise man']
```

Utiliser des séquences plus longues fournit généralement bien davantage de caractéristiques, et notamment des caractéristiques plus spécifiques. Il n'y a pas de bigrammes communs entre les deux phrases de `bards_words` :

**In[30]:**

```
print("Transformed data (dense):\n{}".format(cv.transform(bards_words).
  toarray()))
```

**Out[30]:**

```
Transformed data (dense):
[[0 0 1 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

Dans la plupart des applications, le nombre minimum de *tokens* devrait être un, puisque des mots uniques capturent souvent beaucoup de sens. Ajouter des bigrammes aide dans la plupart des cas. Ajouter également des séquences plus longues (jusqu'à 5-grammes) pourrait aussi aider, mais cela conduirait à une explosion du nombre de caractéristiques et entraînerait un phénomène de surapprentissage, puisqu'il y aurait beaucoup de caractéristiques très spécifiques.

En principe, le nombre de bigrammes pourrait être égal au nombre d'unigrammes au carré, et le nombre de trigrammes pourrait être égal au nombre d'unigrammes à la puissance trois, conduisant évidemment à de très grands espaces de caractéristiques. En pratique, le nombre de *n*-grammes de niveau supérieur est nettement plus petit que l'estimation précédente du fait de la structure du langage analysé (l'anglais en l'occurrence, mais cela vaut également pour le français).

Voici à quoi ressemble l'utilisation d'unigrammes, bigrammes et trigrammes sur le texte `bards_words` :

**In[31]:**

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

**Out[31]:**

```
Vocabulary size: 39
Vocabulary:
['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think', 'doth
  think he', 'fool', 'fool doth', 'fool doth think', 'he', 'he is', 'he is
  wise', 'himself', 'himself to', 'himself to be', 'is', 'is wise', 'knows',
  'knows himself', 'knows himself to', 'man', 'man knows', 'man knows himself',
  'the', 'the fool', 'the fool doth', 'the wise', 'the wise man', 'think',
  'think he', 'think he is', 'to', 'to be', 'to be fool', 'wise', 'wise man',
  'wise man knows']
```

Essayons `TfidfVectorizer` sur le jeu de données provenant du site IMDb, et trouvons la meilleure valeur possible pour la plage des *n*-grammes en utilisant une recherche sur grille :

**In[32]:**

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10, 100],
              "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters:\n{}".format(grid.best_params_))
```

**Out[32]:**

```
Best cross-validation score: 0.91
Best parameters:
{'logisticregression__C': 100, 'tfidfvectorizer__ngram_range': (1, 3)}
```

Comme vous le constatez, nous avons amélioré les performances d'un peu plus d'un pour cent en ajoutant les bigrammes et les trigrammes. Nous pouvons alors visualiser une carte de chaleur de l'exactitude de la validation croisée sous la forme d'une fonction des paramètres `ngram_range` et `C` comme nous l'avons fait dans le Chapitre 5 (voir la Figure 7.3) :

In[33]:

```
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
heatmap = mglearn.tools.heatmap(
    scores, xlabel="C", ylabel="ngram_range", cmap="viridis", fmt="%.3f",
    xticklabels=param_grid['logisticregression__C'],
    yticklabels=param_grid['tfidfvectorizer__ngram_range'])
plt.colorbar(heatmap)
```

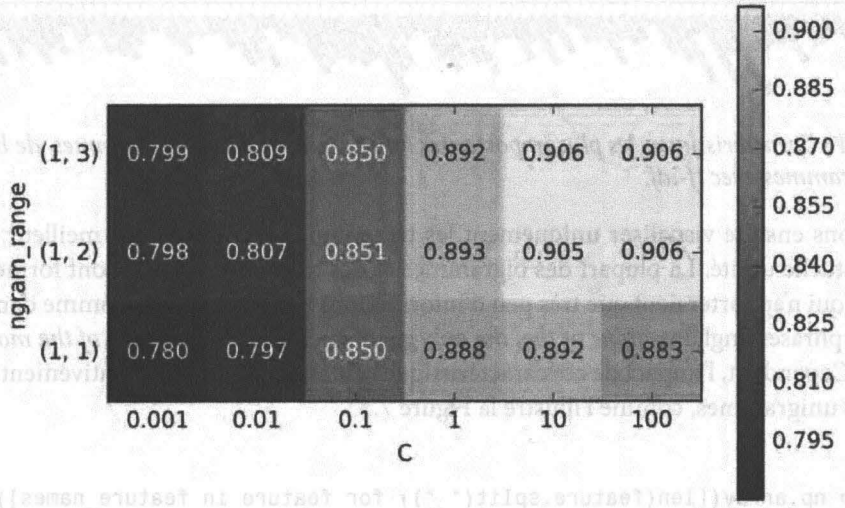



*Figure 7.3 : Carte de chaleur de l'exactitude moyenne de la validation croisée en tant que fonction des paramètres* `ngram_range` *et* `C`.

La carte de chaleur révèle que l'utilisation des bigrammes améliore visiblement les performances, tandis que les trigrammes n'apportent qu'un très petit profit en termes d'exactitude. Pour mieux comprendre ce mécanisme, nous pouvons visualiser les coefficients importants pour le meilleur modèle, qui inclut des unigrammes, des bigrammes et des trigrammes (voir la Figure 7.4) :

In[34]:

```
vect = grid.best_estimator_.named_steps['tfidfvectorizer']
feature_names = np.array(vect.get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
mglearn.tools.visualize_coefficients(coef, feature_names, n_top_features=40)
```

On constate un phénomène particulièrement intéressant concernant les caractéristiques qui contiennent le mot *worth* (valeur) et qu'on ne retrouve pas dans le modèle unigramme. Ainsi, *not worth* indique que le film ne vaut pas la peine, ce qui est un jugement négatif, tandis qu'à l'inverse *definitely worth* indique que le film vaut incontestablement le coup, ce qui est une appréciation éminemment positive. Il s'agit d'un excellent exemple de la manière dont le contexte peut influencer le sens d'un mot.

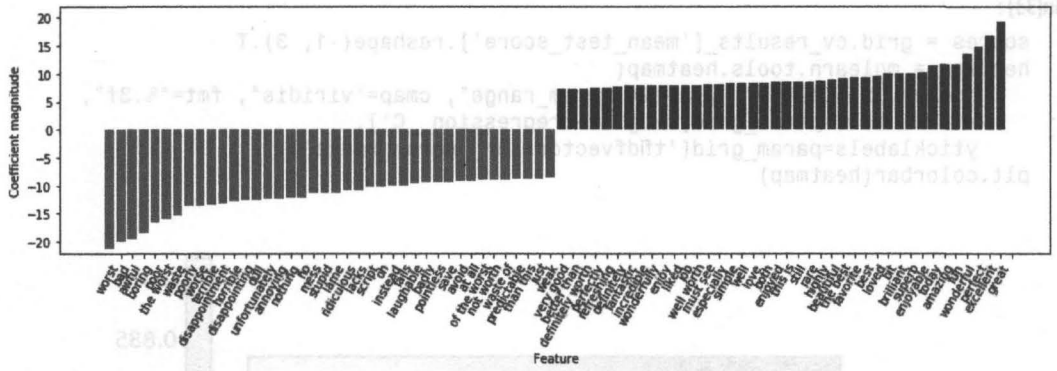



*Figure 7.4 : Caractéristiques les plus importantes lors de l'utilisation d'unigrammes, de bigrammes et de trigrammes avec tf-idf.*

Nous allons ensuite visualiser uniquement les trigrammes afin d'avoir un meilleur aperçu de leur éventuelle utilité. La plupart des bigrammes et des trigrammes utiles sont formés de mots courants qui n'apporteraient que très peu d'informations par eux-mêmes, comme dans les morceaux de phrases anglaises *none of the*, *the only good*, *on and on*, *this is one*, *of the most*, et ainsi de suite. Cependant, l'impact de ces caractéristiques est assez limité comparativement à l'importance des unigrammes, comme l'illustre la Figure 7.5 :

**In[35]:**

```
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
mglearn.tools.visualize_coefficients(coef.ravel()[mask],
                                     feature_names[mask], n_top_features=40)
```

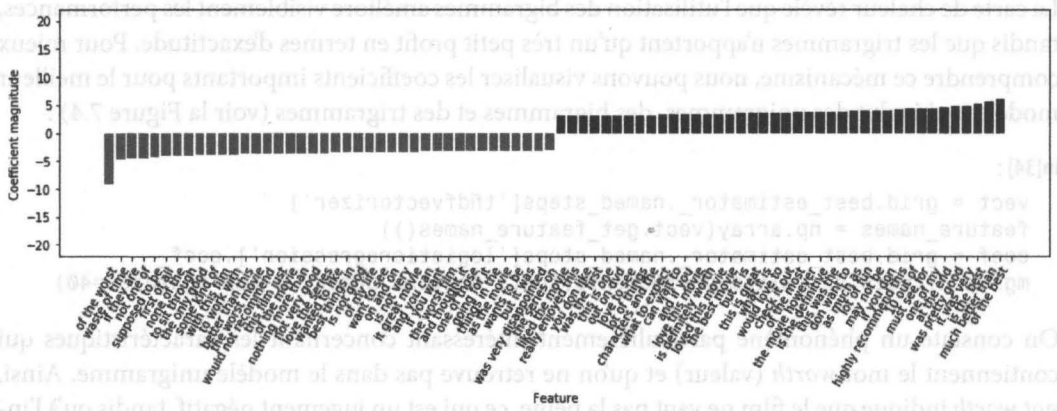



*Figure 7.5 : Visualisation des seuls trigrammes importants du modèle.*

## Tokenisation avancée, racinisation et lemmatisation

Comme nous l'avons déjà mentionné, l'extraction des caractéristiques par `CountVectorizer` et `TfidfVectorizer` est relativement simple, et des méthodes bien plus élaborées sont possibles. Une étape en particulier qui est souvent améliorée dans des applications de fouille de texte plus sophistiquées est la première du modèle de représentation par sac de mots, celle précisément du découpage en « mots », ou *tokenisation*. Cette étape définit ce qui constitue un mot en vue d'une extraction future.

Nous avons vu plus haut que le vocabulaire contient souvent des mots aussi bien au singulier qu'au pluriel. Du point de vue du modèle de représentation par sac de mots, des formes comme par exemple *drawing* (dessin) et *drawings* (dessins) sont si proches que les distinguer ne ferait qu'accroître le surapprentissage et ne permettrait pas au modèle d'exploiter pleinement les données d'apprentissage. De la même manière, nous avons trouvé dans le vocabulaire des mots comme *replace*, *replaced*, *replacement* et *replacing*, marquant tous une notion de remplacement, et qui sont différentes formes verbales et un nom liés au verbe *to replace*. Traiter ces variations comme des mots distincts est évidemment désavantageux pour construire un modèle qui se généralise bien.

Ce problème peut être surmonté en représentant chaque mot à partir de sa *racine* afin de repérer les éléments d'une même « famille ». Si cela est réalisé en utilisant une méthode heuristique, éliminant par exemple les préfixes et suffixes courants, on parle de *stemming* ou *racinisation* (voyez par exemple à ce sujet la page de Wikipédia `https://fr.wikipedia.org/wiki/Racinisation`). Si on se sert d'un dictionnaire de formes connues (un système explicite et non automatisé) et que le rôle du mot dans la phrase est pris en compte, le processus est appelé *lemmatisation*, et la forme standardisée, *canonique*, du mot un *lemme* (reportez-vous aussi à la page de Wikipédia qui présente ce sujet `https://fr.wikipedia.org/wiki/Lemmatisation`). Les deux types de traitement, lemmatisation et racinisation, sont des formes de *normalisation* qui essaient d'extraire une certaine forme normale à partir d'un mot. Un autre cas intéressant de normalisation du langage est la correction orthographique, domaine fort utile en pratique, mais qui déborde du cadre de ce livre.

Pour nous faire une meilleure idée de ces techniques, nous allons comparer une méthode très courante de racinisation (l'algorithme, ou *stemmer*, de Porter, conçu pour la langue anglaise, et importé ici du package `nltk`) avec une méthode de lemmatisation implémentée dans le package `spacy` (nous n'entrerons pas ici dans les détails de ces outils, mais vous pouvez bien entendu vous reporter à leur documentation) :

**In[36]:**

```
import spacy
import nltk

en_nlp = spacy.load('en')
stemmer = nltk.stem.PorterStemmer()
```

```
def compare_normalization(doc):
    doc_spacy = en_nlp(doc)
    print("Lemmatization:")
    print([token.lemma_ for token in doc_spacy])
    print("Stemming:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

Notez en passant que si l'un, l'autre ou les deux de ces packages n'est pas encore installé sur votre système, vous pouvez vous référer pour réparer cet oubli aux pages Web `https://pypi.python.org/pypi/spacy` pour l'un et `https://pypi.python.org/pypi/nltk` pour l'autre. Dans le cas de spacy, il est indispensable de vérifier la bonne procédure d'installation sur la page `https://spacy.io/docs/usage/`, et si nécessaire de télécharger et d'installer Visual C++ Build Tools 2015 depuis la page `http://landinghub.visualstudio.com/visual-cpp-build-tools`.

Nous voulons effectuer la comparaison entre lemmatisation et racinisation selon la méthode de Porter en se basant sur une phrase conçue pour montrer certaines différences :

**In[37]:**

```
compare_normalization(u"Our meeting today was worse than yesterday, "
                      "I'm scared of meeting the clients tomorrow.")
```

**Out[37]:**

```
Lemmatization:
['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be',
  'scar', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
Stemming:
['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m",
  'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
```

La racinisation est toujours limitée à l'ajustement du mot à sa racine, ce qui fait qu'ici *was* devient *wa*, alors que la lemmatisation trouve la forme verbale de base correcte, *be*. De même, la lemmatisation est capable de normaliser le mot *worse* (pire) en *bad* (mauvais), alors que la racinisation produit *wors*. Une autre différence majeure est que la racinisation réduit les deux occurrences de *meeting* en *meet*, alors que la lemmatisation reconnaît la première des deux comme étant un nom qu'elle conserve, et la seconde comme un verbe qui est réduit à sa forme *meet*. En général, la lemmatisation est bien plus complexe que la racinisation, mais elle produit aussi la plupart du temps de meilleurs résultats lorsqu'elle est utilisée pour normaliser des *tokens* dans un système d'apprentissage automatique.

Même si `scikit-learn` n'implémente aucune de ces formes de normalisation, `CountVectorizer` vous permet de spécifier votre propre méthode afin de convertir chaque document en une liste de *tokens* à l'aide du paramètre `tokenizer`. Nous pouvons ainsi nous servir de la lemmatisation proposée par spacy afin de créer un élément qui va prendre une chaîne et produire une liste de lemmes :

**In[38]:**

```
import re
regexp = re.compile('(?u)\\b\\w\\w+\\b')
```

```
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(
    regexp.findall(string))

def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)
```

Transformons maintenant les données et inspectons la taille du vocabulaire :

**In[39]:**

```
X_train_lemma = lemma_vect.fit_transform(text_train)
print("X_train_lemma.shape: {}".format(X_train_lemma.shape))

vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train.shape: {}".format(X_train.shape))
```

**Out[39]:**

```
X_train_lemma.shape: (25000, 21637)
X_train.shape: (25000, 27271)
```

Comme vous pouvez le voir sur cette sortie, la lemmatisation réduit le nombre de caractéristiques de 27 271 (avec le traitement standard `CountVectorizer`) à 21 637. La lemmatisation peut être perçue comme une forme de régularisation, puisqu'elle associe en quelque sorte certaines caractéristiques. Par conséquent, nous en attendons un effet maximal lorsque le jeu de données est de taille réduite. Pour illustrer en quoi la lemmatisation peut aider, nous allons utiliser `StratifiedShuffleSplit` pour la validation croisée, en ne retenant que 1 % des données pour l'apprentissage, le reste servant de données de test :

**In[40]:**

```
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_splits=5, test_size=0.99,
                            train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(), param_grid, cv=cv)
grid.fit(X_train, y_train)
print("Best cross-validation score "
      "(standard CountVectorizer): {:.3f}".format(grid.best_score_))
grid.fit(X_train_lemma, y_train)
print("Best cross-validation score "
      "(lemmatization): {:.3f}".format(grid.best_score_))
```

**Out[40]:**

```
Best cross-validation score (standard CountVectorizer): 0.721
Best cross-validation score (lemmatization): 0.731
```

Dans ce cas, l'amélioration des performances apportée par la lemmatisation est modeste. Comme bon nombre de techniques différentes d'extraction de caractéristiques, le résultat varie en fonction du jeu de données. La lemmatisation et la racinisation peuvent parfois aider à construire des modèles qui sont meilleurs (ou du moins plus compacts) et nous vous conseillons donc de les essayer lorsque vous tentez de tirer quelques parcelles supplémentaires d'amélioration des performances pour une tâche particulière.

## Modèles de sujets et partitionnement de document

Une technique particulière qui est souvent appliquée aux données textuelles est appelée *modélisation de sujet* ou encore *de thématiques* (*topic modeling*). Ce terme est générique, et il sert à décrire la tâche qui consiste à affecter chaque document à un ou plusieurs sujets, ou thèmes, généralement sans supervision. Un bon exemple est fourni par les articles d'informations quotidiens, qui pourraient être de manière assez évidente classés dans des rubriques comme « politique », « sport », « finance » et ainsi de suite. Si chaque document est assigné à un unique thème, cette tâche relève d'une problématique d'apprentissage non supervisé dont nous avons discuté au Chapitre 3. Si chaque document peut correspondre à plusieurs sujets, la tâche relève des méthodes d'extraction, ou de décomposition, également abordées au Chapitre 3. Chacun des composants appris correspond alors à un sujet, et les coefficients des composants dans la représentation d'un document nous indiquent le degré d'attachement de ce document avec un sujet spécifique. Souvent, lorsque l'on parle de modélisation de sujet, on fait en réalité référence à une méthode de décomposition, ou d'extraction, particulière appelée *Allocation de Dirichlet latente* (ou en anglais LDA, pour *Latent Dirichlet Allocation*). En passant, signalons qu'il ne faut pas confondre la LDA traitée ici avec l'analyse discriminante linéaire (en anglais *Linear Discriminant Analysis*)…

### L'allocation de Dirichlet latente (LDA)

Intuitivement, le modèle LDA essaie de trouver des groupes de mots (les sujets, ou topics) qui apparaissent fréquemment ensemble. Ce modèle nécessite aussi que chaque document puisse être compris comme un « mélange » d'un sous-ensemble du sujet. Il est indispensable avant tout de bien comprendre que, du point de vue du modèle d'apprentissage automatique, un sujet peut parfaitement être quelque chose de différent de ce que nous, humains, pourrions appeler ainsi dans la vie quotidienne. En l'occurrence, cela ressemble davantage aux composants extraits par les méthodes PCA ou NMF (voir le Chapitre 3) et qui peuvent ou non avoir une signification sémantique. De plus, même s'il existe une telle signification pour un « sujet » LDA, il pourrait ne pas s'agir pour autant de quelque chose que nous appellerions généralement un *sujet*. Revenons à l'exemple des articles d'informations quotidiens. Supposons que nous disposons d'une série d'articles concernant le sport, la politique et la finance, mais écrits par deux journalistes

différents. Dans un article politique, on peut s'attendre à trouver des mots comme *président, vote, parti, etc.* De même, dans un article de sport, on peut s'attendre à trouver des mots comme *équipe, score, saison, etc.* Des mots de chacun de ces groupes devraient probablement apparaître à proximité les uns des autres, tandis que ce genre de rencontre pour les mots *président* et *saison* est nettement moins vraisemblable. Cependant, nos deux journalistes peuvent aussi avoir leurs préférences dans la manière de tourner les phrases ou de choisir les mots. Ainsi, l'un d'entre eux pourrait aimer placer dans ses articles le verbe « démarquer », et l'autre le verbe « polariser ». Dans ce cas, des sujets *autres* pourraient être « mots souvent employés par le journaliste A » et « mots souvent employés par le journaliste B », qui ne sont bien sûr pas pour nous des sujets au sens usuel du terme.

Essayons d'appliquer le modèle LDA à notre jeu de données filmographique afin de voir en pratique comment il fonctionne. Pour ce type de modèle non supervisé appliqué à des documents textuels, il est souvent bon d'éliminer les mots très courants du fait qu'ils pourraient avoir un poids bien trop élevé lors de l'analyse. Nous allons supprimer les mots qui apparaissent dans au moins 15 % des documents, et nous limiterons le modèle de représentation par sac de mots aux 10 000 termes qui sont les plus courants une fois enlevés les premiers 15 % :

**In[41]:**

```
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
```

Nous allons entraîner un modèle avec 10 sujets, ce qui est suffisamment restreint pour que nous puissions tous les observer simultanément. De manière semblable aux composants dans NMF, les sujets n'ont pas un ordre inhérent, et modifier leur nombre change tous les sujets (en fait, NMF et LDA résolvent des problèmes assez similaires, et nous pourrions aussi nous servir de NMF pour extraire des sujets). Nous utiliserons ici la méthode d'apprentissage `"batch"`, qui est un peu plus lente que la méthode par défaut (`"online"`), mais qui fournit généralement de meilleurs résultats, et nous augmenterons également `"max_iter"`, ce qui peut aussi améliorer les modèles :

**In[42]:**

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch",
                                max_iter=25, random_state=0)
document_topics = lda.fit_transform(X)
```

Comme les méthodes d'extraction que nous avons vues dans le Chapitre 3, `LatentDirichletAllocation` possède un attribut `components_` qui mémorise l'importance de chaque mot pour chaque sujet. La taille de `components_` est `(n_topics, n_words)` :

**In[43]:**

```
print("lda.components_.shape: {}".format(lda.components_.shape))
```

**Out[43]:**

```
lda.components_.shape: (10, 10000)
```

Pour mieux comprendre la signification des différents sujets, nous allons regarder quels sont les mots les plus importants pour chacun d'entre eux. La fonction `print_topics` permet de présenter très clairement ces caractéristiques :

**In[44]:**

```
sorting = np.argsort(lda.components_, axis=1)[:,::-1]
feature_names = np.array(vect.get_feature_names())
```

**In[45]:**

```
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=10)
```

**Out[45]:**

```
topic 0       topic 1       topic 2       topic 3       topic 4
--------      --------      --------      --------      --------
between       war           funny         show          didn
family        world         comedy        series        saw
young         us            guy           episode       thought
real          american      laugh         tv            am
us            our           jokes         episodes      thing
director      documentary   fun           shows         got
work          history       humor         season        10
both          years         re            new           want
beautiful     new           hilarious     years         going
each          human         doesn         television    watched

topic 5       topic 6       topic 7       topic 8       topic 9
--------      --------      --------      --------      --------
action        kids          role          performance   horror
effects       action        cast          role          house
nothing       animation     john          john          killer
budget        children      version       actor         gets
script        game          novel         cast          woman
minutes       disney        director      plays         dead
original      fun           both          jack          girl
director      old           played        michael       around
least         10            mr            oscar         goes
doesn         kid           young         father        wife
```

En fondant son jugement sur les mots importants, le sujet 1 paraît concerner des films de guerre et historiques, le sujet 2 de mauvaises comédies, le sujet 3 des séries télé. Le sujet 4 semble capturer des mots très courants, tandis que le sujet 5 paraît concerner des aspects techniques, le sujet 6 étant manifestement dédié aux films pour enfants. Le sujet 8 a trait aux films, actrices et acteurs qui ont obtenu des récompenses. Quant au sujet 9, il se rapporte très vraisemblablement aux

films d'horreur et violents. Puisque nous n'utilisons que dix sujets, chacun doit représenter un champ très large, de manière à ce qu'ils puissent tous ensemble couvrir la totalité des critiques de films de notre jeu de données.

Nous allons ensuite entraîner un autre modèle, cette fois avec 100 sujets. Cela rend l'analyse plus complexe, mais aussi plus précise quant à l'adéquation des sujets avec nos données :

**In[46]:**

```
lda100 = LatentDirichletAllocation(n_topics=100, learning_method="batch",
                                   max_iter=25, random_state=0)
document_topics100 = lda100.fit_transform(X)
```

Regarder tous les résultats serait une tâche évidemment accablante. Nous avons donc sélectionné plusieurs sujets intéressants et représentatifs :

**In[47]:**

```
topics = np.array([7, 16, 24, 25, 28, 36, 37, 41, 45, 51, 53, 54, 63, 89, 97])

sorting = np.argsort(lda100.components_, axis=1)[:,::-1]
feature_names = np.array(vect.get_feature_names())
mglearn.tools.print_topics(topics=topics, feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=20)
```

**Out[47]:**

```
topic 7       topic 16      topic 24      topic 25      topic 28
--------      --------      --------      --------      --------
thriller      worst         german        car           beautiful
suspense      awful         hitler        gets          young
horror        boring        nazi          guy           old
atmosphere    horrible      midnight      around        romantic
mystery       stupid        joe           down          between
house         thing         germany       kill          romance
director      terrible      years         goes          wonderful
quite         script        history       killed        heart
bit           nothing       new           going         feel
de            worse         modesty       house         year
performances  waste         cowboy        away          each
dark          pretty        jewish        head          french
twist         minutes       past          take          sweet
hitchcock     didn          kirk          another       boy
tension       actors        young         getting       loved
interesting   actually      spanish       doesn         girl
mysterious    re            enterprise    now           relationship
murder        supposed      von           night         saw
ending        mean          nazis         right         both
creepy        want          spock         woman         simple
```

| topic 36 | topic 37 | topic 41 | topic 45 | topic 51 |
|---|---|---|---|---|
| performance | excellent | war | music | earth |
| role | highly | american | song | space |
| actor | amazing | world | songs | planet |
| cast | wonderful | soldiers | rock | superman |
| play | truly | military | band | alien |
| actors | superb | army | soundtrack | world |
| performances | actors | tarzan | singing | evil |
| played | brilliant | soldier | voice | humans |
| supporting | recommend | america | singer | aliens |
| director | quite | country | sing | human |
| oscar | performance | americans | musical | creatures |
| roles | performances | during | roll | miike |
| actress | perfect | men | fan | monsters |
| excellent | drama | us | metal | apes |
| screen | without | government | concert | clark |
| plays | beautiful | jungle | playing | burton |
| award | human | vietnam | hear | tim |
| work | moving | ii | fans | outer |
| playing | world | political | prince | men |
| gives | recommended | against | especially | moon |

| topic 53 | topic 54 | topic 63 | topic 89 | topic 97 |
|---|---|---|---|---|
| scott | money | funny | dead | didn |
| gary | budget | comedy | zombie | thought |
| streisand | actors | laugh | gore | wasn |
| star | low | jokes | zombies | ending |
| hart | worst | humor | blood | minutes |
| lundgren | waste | hilarious | horror | got |
| dolph | 10 | laughs | flesh | felt |
| career | give | fun | minutes | part |
| sabrina | want | re | body | going |
| role | nothing | funniest | living | seemed |
| temple | terrible | laughing | eating | bit |
| phantom | crap | joke | flick | found |
| judy | must | few | budget | though |
| melissa | reviews | moments | head | nothing |
| zorro | imdb | guy | gory | lot |
| gets | director | unfunny | evil | saw |
| barbra | thing | times | shot | long |
| cast | believe | laughed | low | interesting |
| short | am | comedies | fulci | few |
| serial | actually | isn | re | half |

Les sujets que nous avons extraits semblent cette fois être plus spécifiques, quoique nombre d'entre eux soient difficiles à interpréter. Par exemple, le sujet 7 paraît être associé aux films policiers et d'horreur. Les sujets 16 et 54 concernent manifestement des critiques négatives, alors que le sujet 63 paraît relever de comédies appréciées comme étant positives. Si nous voulons obtenir des inférences supplémentaires en utilisant les sujets que nous venons de découvrir,

nous devrions confirmer nos premières intuitions en regardant les mots qui ont obtenu le classement le plus élevé pour chaque sujet, et ce en observant les documents associés à ces sujets. Prenons par exemple le sujet 45. La première intuition est évidemment qu'il concerne des créations musicales. Regardons alors quels types de commentaitres ou de critiques sont assignés à ce sujet :

**In[48]:**

```
music = np.argsort(document_topics100[:, 45])[::-1]
for i in music[:10]:
    print(b".".join(text_traIn[i].split(b".")[:2]) + b".\n")
```

**Out[48]:**

```
b'I love this movie and never get tired of watching. The music in it is
  great.\n'
b"I enjoyed Still Crazy more than any film I have seen in years. A successful
  band from the 70's decide to give it another try.\n"
b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for
  Warner Bros. His directing style had changed or evolved to the point that
  this film does not contain his signature overhead shots or huge production
  numbers with thousands of extras.\n'
b"What happens to washed up rock-n-roll stars in the late 1990's? They launch a
  comeback / reunion tour. At least, that's what the members of Strange Fruit,
  a (fictional) 70's stadium rock group do.\n"
b'As a big-time Prince fan of the last three to four years, I really can\'t
  believe I\'ve only just got round to watching "Purple Rain". The brand new
  2-disc anniversary Special Edition led me to buy it.\n'
b"This film is worth seeing alone for Jared Harris' outstanding portrayal of
  John Lennon. It doesn't matter that Harris doesn't exactly resemble Lennon;
  his mannerisms, expressions, posture, accent and attitude are pure Lennon.\n"
b"The funky, yet strictly second-tier British glam-rock band Strange Fruit
  breaks up at the end of the wild'n'wacky excess-ridden 70's. The individual
  band members go their separate ways and uncomfortably settle into lackluster
  middle age in the dull and uneventful 90's: morose keyboardist Stephen Rea
  winds up penniless and down on his luck, vain, neurotic, pretentious lead
  singer Bill Nighy tries (and fails) to pursue a floundering solo career,
  paranoid drummer Timothy Spall resides in obscurity on a remote farm so he
  can avoid paying a hefty back taxes debt, and surly bass player Jimmy Nail
  installs roofs for a living.\n"
b"I just finished reading a book on Anita Loos' work and the photo in TCM
  Magazine of MacDonald in her angel costume looked great (impressive wings),
  so I thought I'd watch this movie. I'd never heard of the film before, so I
  had no preconceived notions about it whatsoever.\n"
b'I love this movie!!! Purple Rain came out the year I was born and it has had
  my heart since I can remember. Prince is so tight in this movie.\n'
b"This movie is sort of a Carrie meets Heavy Metal. It's about a highschool guy
  who gets picked on alot and he totally gets revenge with the help of a Heavy
  Metal ghost.\n"
```

Comme nous pouvons le voir, ce thème couvre une large variété de critiques centrées sur la musique, allant de musiciens jusqu'à des films biographiques, le dernier document quant à lui étant sans doute plus difficile à classer pour un non-spécialiste. Une autre manière intéressante

d'inspecter les sujets consiste à observer leurs poids en sommant l'attribut `document_topics` sur toutes les critiques. Dans ce qui suit, nous nommerons chaque sujet à partir des deux mots les plus courants (voir la Figure 7.6) :

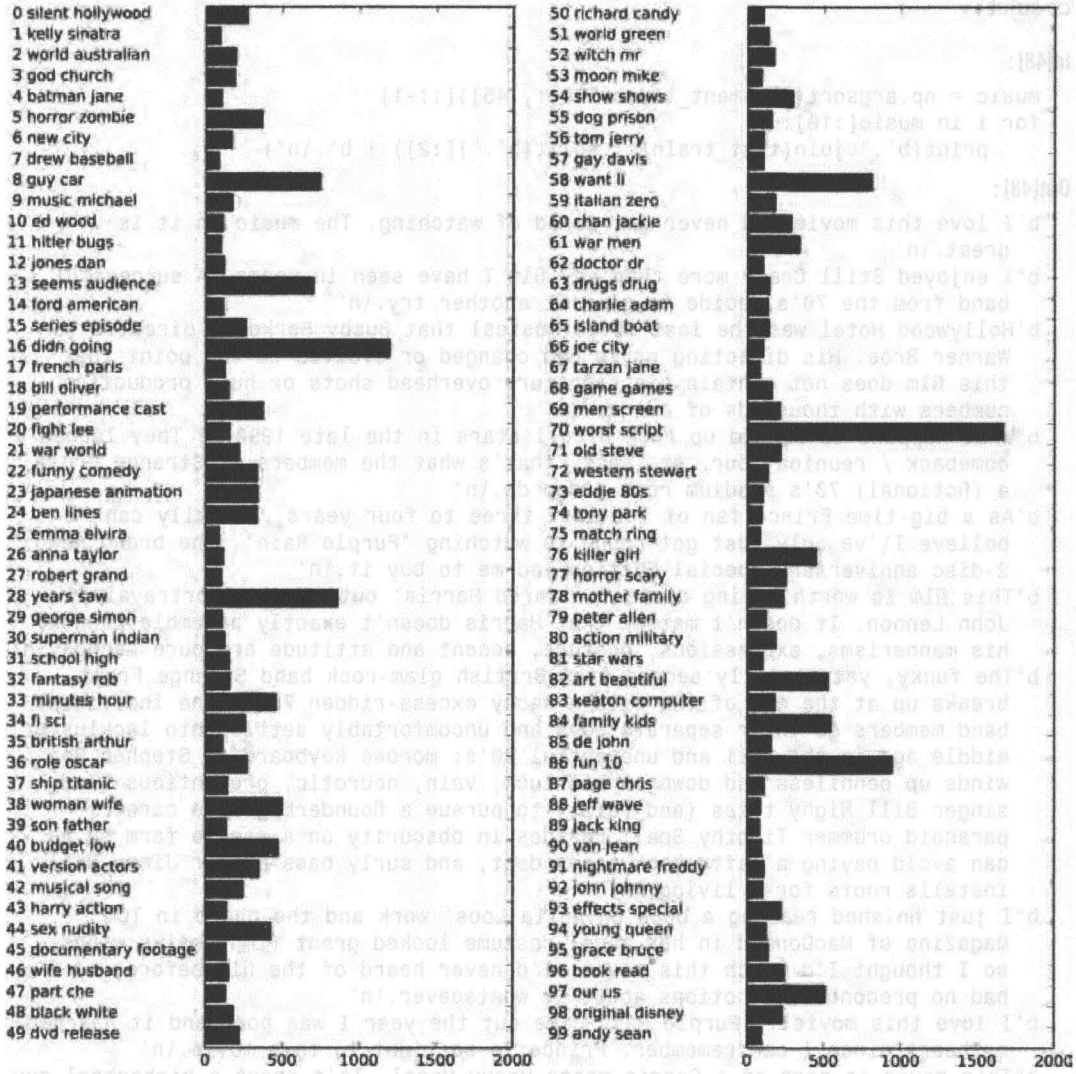



*Figure 7.6 : Poids respectif des sujets avec la méthode LDA.*

**In[49] :**

```
fig, ax = plt.subplots(1, 2, figsize=(10, 10))
topic_names = ["{:>2} ".format(i) + " ".join(words)
               for i, words in enumerate(feature_names[sorting[:, :2]])]
```

```
for col in [0, 1]:
    start = col * 50
    end = (col + 1) * 50
    ax[col].barh(np.arange(50), np.sum(document_topics100, axis=0)[start:end])
    ax[col].set_yticks(np.arange(50))
    ax[col].set_yticklabels(topic_names[start:end], ha="left", va="top")
    ax[col].invert_yaxis()
    ax[col].set_xlim(0, 2000)
    yax = ax[col].get_yaxis()
    yax.set_tick_params(pad=130)
plt.tight_layout()
```

Les sujets les plus importants sont le 70, qui correspond vraisemblablement à des appréciations négatives, les 16, 13 et 58, qui semblent contenir des mots vides, ainsi que le 86, qui est associé à des appréciations positives. Le nombre 10 pour ce sujet correspond vraisemblablement à une note de 10 sur 10 mentionnée dans le commentaire. Ces sujets principaux sont suivis par d'autres plus spécifiques, comme les numéros 8, 38, 40, 44, 76, 82 et 84.

Cela donne l'impression que la méthode LDA a découvert pour l'essentiel deux grands types de sujets : par genre d'une part, et par notation d'autre part, ainsi que plusieurs thèmes moins spécifiques. Il s'agit d'une découverte intéressante, car la plupart des critiques sont composées pour une partie de commentaires spécifiques au film, et pour une autre partie de commentaires qui justifient ou mettent l'accent sur la notation.

Des modèles tels que LDA sont des méthodes intéressantes pour comprendre de larges corpus de texte. L'algorithme LDA est cependant *randomisé*, et donc modifier le paramètre `random_state` peut se traduire par des sorties assez différentes. Bien que l'identification des sujets puisse être utile, toutes les conclusions obtenues à partir d'un modèle non supervisé devraient être prises avec quelques pincettes, et nous vous conseillons vivement de vérifier vos intuitions en consultant les documents associés à un thème spécifique. Les sujets produits par la méthode `LDA.transform` peuvent aussi parfois être utilisés en tant que représentation compacte pour l'apprentissage supervisé. Cela est particulièrement intéressant lorsque la quantité d'échantillons d'apprentissage est réduite.

# Résumé et perspectives

Dans ce chapitre, nous avons abordé les bases du traitement des textes, ou *traitement du langage naturel* (TLN en français, ou *natural language processing*, NLP, en anglais) avec un exemple d'application de classification basé sur des critiques de films. Les outils discutés ici devraient fournir un bon point de départ pour essayer de traiter des données textuelles. En particulier, dans le cas de tâches de classification telles que la détection de spams ou de fraudes, ou encore l'analyse de commentaires ou d'appréciations laissés par des utilisateurs, les représentations par sacs de mots fournissent une solution simple mais puissante. Comme c'est souvent le cas dans l'apprentissage

automatique, la manière de représenter les données est au cœur des applications de TLN, et inspecter les *tokens* comme les *n*-grammes qui sont extraits peut donner nombre d'idées très utiles quant au processus de modélisation. Dans le cas d'applications de traitement du texte, il est souvent possible de réaliser une introspection réellement significative des modèles, aussi bien pour des tâches supervisées que non supervisées. Vous devriez tirer pleinement parti de ces possibilités lorsque vous avez à utiliser en pratique des méthodes de traitement du langage naturel.

Le langage naturel et le traitement des données textuelles (la *fouille de textes*) constituent un vaste champ de recherche, et discuter plus en détail des méthodes avancées dépasse largement le cadre de ce livre. Nous ne pouvons que vous inviter à poursuivre vos recherches sur le sujet dans la littérature spécialisée ainsi que sur le Web. Dans ce chapitre, nous avons vu que les classes `CountVectorizer` et `TfidfVectorizer` n'implémentent que des méthodes de traitement de texte relativement simples. Pour des méthodes plus élaborées, nous vous recommandons de vous tourner vers les packages Python `spacy` (qui est relativement nouveau, pas forcément très simple à installer, mais très efficace et bien conçu, proposant de surcroît un modèle pour le français), `nltk` (une librairie bien établie et complète, mais peut-être un peu datée) et `gensim` (qui met l'accent sur la modélisation de sujet).

Il est aussi important de bien noter que plusieurs et très excitants nouveaux axes de développement sont apparus ces toutes dernières années. Ils dépassent bien sûr le cadre de cet ouvrage. Contentons-nous d'indiquer qu'ils se rapportent aux réseaux de neurones. Citons cependant en particulier l'utilisation de représentations dites distribuées de mots (*distributed word representations*). Voyez notamment à ce sujet ce que vous pouvez trouver sur la librairie `word2vec` de Tomas Mikolov et aussi dans ses écrits.

Une autre direction récente dans le traitement du langage naturel est l'utilisation de *réseaux de neurones récurrents* (*recurrent neutral networks* ou RNN en anglais). Ce type de réseau de neurones est particulièrement puissant car il est capable de produire du texte, à la différence des modèles de classification qui ne peuvent assigner que des étiquettes de classe. Cela rend ces réseaux particulièrement adaptés pour les tâches de traduction, ou encore en reconnaissance automatique de la parole ou de l'écriture manuscrite. Vous pouvez commencer par consulter à ce sujet la page de Wikipédia à l'adresse `https://fr.wikipedia.org/wiki/Réseau_de_neurones_récurrents`. Un tutoriel pratique et qui propose quantité de liens très intéressants concerne le cadre de programmation `tensorflow`, en partant de l'adresse `https://www.tensorflow.org/tutorials/seq2seq`.

# CHAPITRE 8
# Pour conclure

Vous savez maintenant comment appliquer les principaux algorithmes de *machine learning* à des apprentissages supervisés et non supervisés, ce qui vous permet de résoudre une large variété de problèmes de ce type. Avant de vous laisser poursuivre pour explorer toutes les possibilités offertes par ce domaine, nous souhaitons vous proposer quelques derniers conseils, vous signaler quelques ressources supplémentaires, et vous suggérer quelques pistes pour améliorer vos compétences dans les domaines de l'apprentissage automatique et de la science des données (*data science*).

## Aborder un problème d'apprentissage automatique

Partant de toutes les grandes et importantes méthodes que nous vous avons présentées dans ce livre, vous pourriez être tenté(e) de sauter à pieds joints et de commencer à résoudre votre problème de traitement de données en exécutant simplement votre algorithme préféré. Cependant, il ne s'agit généralement pas d'une bonne manière de commencer votre analyse. La question de l'algorithme d'apprentissage automatique ne représente le plus souvent qu'une petite partie d'une réflexion bien plus vaste tant en ce qui concerne l'analyse des données que le processus de prise de décision. Pour employer ces techniques de manière efficace, vous devez prendre un certain recul, et considérer le problème d'un point de vue plus large. Tout d'abord, il vous faut réfléchir aux types de questions auxquelles vous voulez répondre. Avez-vous besoin d'une démarche exploratoire, ou simplement de trouver quelque chose d'intéressant dans les données ? Avez-vous déjà un objectif particulier présent à l'esprit ? Souvent, vous partez avec un certain but en tête, comme détecter des transactions bancaires frauduleuses, recommander des films ou des livres, ou bien encore découvrir de nouvelles exoplanètes. Si vous avez un tel but (ou un autre tout à fait différent), vous devriez réfléchir en premier lieu non pas à construire un système, mais à comment définir et mesurer votre réussite, et à songer à l'impact qu'une solution efficace et réussie aurait sur votre activité ou vos objectifs de recherche.

Prenons comme exemple la détection de fraudes. Voici quelques questions importantes à se poser :

- Comment puis-je mesurer la validité de ma prédiction de fraudes ?
- Est-ce que je dispose des bonnes données pour les faire évaluer par l'algorithme ?
- Si je réussis, quel sera l'impact économique de ma solution ?

Comme nous l'avons vu au Chapitre 5, il est préférable, si possible, de mesurer directement les performances de votre algorithme en faisant appel à une métrique « business », comme l'augmentation du profit ou la réduction des pertes. Cependant, cela est souvent difficile à réaliser. Une question à laquelle il peut être plus facile de répondre est la suivante : « Que va-t-il se passer si je construis le modèle parfait ? ». Si détecter parfaitement toutes les fraudes fait gagner chaque mois 100 €, ces gains possibles ne seront probablement pas suffisants pour justifier les efforts à fournir pour simplement commencer à développer un algorithme. D'un autre côté, si votre solution permet à votre entreprise d'éviter de perdre des dizaines de milliers d'euros chaque mois, étudier le problème peut vraiment valoir la peine.

Supposons maintenant que vous avez défini le problème à résoudre, que vous connaissez une solution qui pourrait avoir un impact significatif sur votre projet, et que vous disposez des bonnes informations pour évaluer la réussite de celui-ci. Les étapes suivantes consistent généralement à acquérir les données et à bâtir un prototype fonctionnel. Dans ce livre, nous avons traité des multiples modèles auxquels vous pouvez faire appel, et des outils qui vous permettent d'évaluer et d'améliorer ces modèles. Mais, ce faisant, n'oubliez jamais qu'il ne s'agit que d'une petite partie d'un champ bien plus large : celui de la science des données. De ce point de vue, la construction d'un modèle n'est souvent qu'un élément dans une démarche circulaire consistant à collecter de nouvelles données, les mettre en forme, fabriquer des modèles, analyser ces modèles, et retour. Analyser les erreurs faites par un modèle fournit souvent de bonnes indications sur ce qui manque dans les données, sur quelles données supplémentaires il faudrait collecter, ou encore sur la manière dont la tâche devrait être reformulée afin de rendre le processus d'apprentissage automatique plus efficace. Collecter davantage de données, ou des données différentes, ou bien modifier légèrement la formulation de la tâche, peut apporter un avantage bien plus grand qu'exécuter à l'infini des recherches sur grille afin d'affiner les paramètres.

## Est-ce qu'il y des humains dans la boucle ?

Vous devriez également prendre en considération la présence d'êtres humains dans la boucle, et la manière d'interagir avec eux. Certains processus (comme la détection des piétons par un véhicule autonome) nécessitent une prise de décision immédiate. Ce n'est pas le cas dans d'autres situations, et il peut donc être possible que des humains soient amenés à valider des décisions incertaines. Des applications médicales, par exemple, nécessitent souvent un très haut niveau d'exactitude qui ne peut probablement pas être obtenu uniquement par un algorithme d'apprentissage automatique. Mais si un algorithme est capable de prendre automatiquement 90 %,

50 %, voire même 10 % des décisions, alors le bénéfice obtenu en termes de réduction du délai de réponse ou de réduction des coûts mérite d'être examiné de près. En fait, la plupart des applications sont dominées par des « cas simples », pour lesquels un algorithme peut prendre une décision, et il y a relativement peu de « cas complexes », pour lesquels une intervention humaine est nécessaire.

## Du prototype à la production

Les outils étudiés dans ce livre sont parfaits pour de multiples applications d'apprentissage automatique, et ils permettent très rapidement d'effectuer des analyses et du prototypage. Python et `scikit-learn` sont également employés dans des systèmes de production par de nombreuses organisations, et même par de très grandes entreprises comme des banques internationales ou des réseaux sociaux. Cependant, de nombreuses sociétés ont des infrastructures complexes, et il n'est pas toujours facile d'inclure Python dans ces systèmes. Mais ce n'est pas nécessairement un problème. Bien souvent, les équipes chargées de l'analyse des données travaillent avec des langages tels que Python et R afin de pouvoir tester rapidement des idées, tandis que les équipes de production se servent de langages de programmation tels que Go, Scala, C++ ou encore Java afin de construire des systèmes solides et évolutifs. L'analyse des données ne répond pas aux mêmes besoins que les autres services de l'entreprise, et utiliser des langages différents pour ces tâches fait sens. Une méthode relativement courante consiste à réimplanter la solution développée par l'équipe en charge de l'analyse des données dans le cadre plus général des systèmes de programmation de l'entreprise en faisant appel à un langage à hautes performances. Cela peut être plus simple que d'essayer d'incorporer toute une librairie ou un langage de programmation, et d'effectuer des conversions entre différents formats de données.

Indépendamment du fait que vous puissiez ou non utiliser `scikit-learn` dans un système de production, il est important de garder présent à l'esprit le fait que ces systèmes ont des besoins différents de ceux des scripts d'analyse ponctuels. Si un algorithme est déployé sur un système plus large, des aspects d'ingénierie logicielle comme la fiabilité, la prédictibilité, les performances d'exécution et les besoins en mémoire gagnent en pertinence. La simplicité est centrale pour fournir des systèmes d'apprentissage automatique capables de bien s'adapter dans de telles situations. Si le sujet vous concerne, ou plus simplement vous intéresse, et si vous maîtrisez suffisamment l'anglais, nous vous conseillons vivement de lire l'article intitulé « Machine Learning : The High Interest Credit Card of Technical Debt », publié par des chercheurs de Google (`https://research.google.com/pubs/pub43146.html`).

## Tester des systèmes de production

Dans ce livre, nous avons vu comment évaluer des prédictions algorithmiques à partir d'un jeu de test collecté à l'avance. C'est ce que l'on appelle aussi une *évaluation hors ligne*. Si votre système d'apprentissage automatique s'interface avec l'utilisateur, il ne s'agit cependant que de la première phase d'évaluation d'un algorithme. L'étape suivante passe généralement par des *tests en ligne*, dans lesquels les conséquences de l'emploi de l'algorithme dans le système global sont évaluées. Changer les recommandations ou le résultat de recherches d'utilisateurs dans un site Web peut dramatiquement modifier leurs comportements et induire des conséquences inattendues. Pour se protéger contre ces surprises, la plupart des services ayant à s'interfacer avec les utilisateurs emploient ce que l'on appelle un *test A/B*, dans lequel plusieurs variantes d'un même critère sont proposées afin de déterminer celle qui fournit les meilleurs résultats auprès des utilisateurs.

Dans ce genre de test, donc, une partie des utilisateurs (évidemment non prévenus à l'avance) se voit proposer une version d'un site Web ou d'un service qui utilise un algorithme A, tandis que le reste de la population est testé avec un algorithme B. Dans les deux groupes, des mesures de réussites pertinentes sont enregistrées pendant un certain laps de temps. Les algorithmes A et B sont ensuite comparés, et un choix entre les deux approches peut être effectué en fonction des résultats obtenus. Utiliser cette technique de test A/B nous permet d'évaluer les algorithmes en situation réelle, ce qui peut également nous aider à découvrir des conséquences inattendues de l'interaction des utilisateurs avec notre modèle. Souvent, A est un nouveau modèle, tandis que B est le système déjà établi. Il existe bien entendu des mécanismes plus élaborés que le test A/B, comme l'algorithme dit du *bandit manchot*, mais cela dépasse notre propos.

## Construire votre propre estimateur

Ce livre a traité d'une variété d'outils et d'algorithmes implémentés dans `scikit-learn` qui peuvent être utilisés pour des tâches très diverses. Cependant, il est courant d'avoir besoin d'effectuer un traitement particulier sur les données d'une manière qui n'est pas implémentée dans `scikit-learn`. Il peut simplement s'agir d'un prétraitement des données avant de passer celles-ci à votre modèle ou pipeline `scikit-learn`. Cependant, si votre prétraitement dépend des données, ou si vous voulez appliquer une recherche sur grille ou une validation croisée, les choses deviennent plus compliquées.

Dans le Chapitre 6, nous avons discuté de l'importance de placer tous les traitements dépendants des données dans la boucle de validation croisée. Comment alors utiliser votre propre procédure de traitement avec les outils de `scikit-learn` ? Voici une solution simple, du moins dans le principe : construire votre propre *estimateur* (ou *estimator*) ! Implémenter un estimateur qui soit compatible avec l'interface de `scikit-learn`, et donc utilisable avec `Pipeline`, `GridSearchCV` et `cross_val_score`, est relativement facile. La documentation de `scikit-learn` fournit des

instructions détaillées à ce sujet (voyez ainsi l'adresse `http://scikit-learn.org/stable/developers/contributing.html#rolling-your-own-estimator`). Pour aller à l'essentiel, la manière la plus simple d'implémenter une telle classe (ou *transformer*) consiste à la faire hériter de `BaseEstimator` et de `TransformerMixin`, puis à implémenter les fonctions `__init__`, `fit` et `predict`, comme ceci :

**In[1]:**

```
from sklearn.base import BaseEstimator, TransformerMixin

class MyTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, first_paramter=1, second_parameter=2):
        self.first_paramter = 1
        self.second_parameter = 2

    def fit(self, X, y=None):

        print("fitting the model right here")
        return self

    def transform(self, X):

        X_transformed = X + 1
        return X_transformed
```

Implémenter un classifieur ou un régresseur fonctionne de la même manière, si ce n'est que vous devez remplacer `TransformerMixin` par `ClassifierMixin` ou `RegressorMixin`. De plus, la définition de `transform` devrait être remplacée par celle de `predict`.

Comme vous pouvez le constater sur cet exemple, cette implémentation ne demande que très peu de code. D'ailleurs, les utilisateurs de `scikit-learn` ont généralement l'habitude de fabriquer leur propre collection de modèles personnalisés au fil du temps et en fonction de leurs besoins.

# Pour aller plus loin

Ce livre propose une introduction à l'apprentissage automatique et vous aide à en devenir un praticien efficace. Cependant, si vous voulez pousser plus loin vos connaissances, nous vous encourageons vivement à vous lancer dans des tas de recherches sur le Web et à vous intéresser à la littérature technique et/ou universitaire sur ce sujet.

## Théorie

Dans ce livre, nous avons essayé de fournir une intuition sur la manière dont fonctionnent les algorithmes d'apprentissage automatique les plus courants, et cela sans faire appel à des notions

de très haut niveau, que ce soit en mathématiques ou en informatique. Cependant, la plupart des modèles discutés ici utilisent des principes issus de la théorie des probabilités, de l'algèbre linéaire et de l'optimisation. Même s'il n'est pas nécessaire de comprendre tous les détails de l'implémentation de ces algorithmes, nous pensons que le fait de connaître au moins une partie de la théorie qui sous-tend ces algorithmes fera de vous un meilleur *data scientist*. À nouveau, faites de recherches sur le Web pour trouver des références de livres, de cours gratuits, ou encore de vidéos de formation qui soient adaptés à votre niveau et à vos propres besoins.

Si votre anglais est d'un niveau suffisant et si vos moyens vous le permettent, voici quelques références d'ouvrages que nous ne pouvons que vous recommander : le déjà nommé *The Elements of Statistical Learning* (de Hastie, Tibshirani et Friedman), *Machine Learning: An Algorithmic Perspective* par Stephen Marsland (Chapman et Hall/CRC), *Pattern Recognition and Machine Learning* par Christopher Bishop (Springer) et le volumineux *Machine Learning : A Probabilistic Perspective* par Kevin Murphy (MIT Press) (de ce point de vue, les travaux d'Augustin Cournot datant du 19e siècle mériteraient certainement d'être réévalués).

## Autres packages et ressources pour l'apprentissage automatique

Vous l'avez bien compris, pour tout ce qui concerne l'apprentissage automatique, Python est notre langage préféré et `scikit-learn` notre *package* favori. Mais il existe bien entendu d'autres options. Il se peut que le couple Python/`scikit-learn` ne soit pas ce qui convient le mieux à votre propre situation. Souvent, Python est très bon pour tester et évaluer des modèles, mais les services Web importants et les applications sont plus couramment écrits en Java ou en C++, et une intégration dans ces systèmes pourrait être nécessaire pour le déploiement de votre modèle. Une autre raison possible tient au fait que votre champ d'intérêt est peut-être plus tourné vers la modélisation statistique et l'inférence que vers la prédiction. Dans ce cas, vous devriez envisager de vous tourner vers le package Python appelé `statsmodels`, qui implémente plusieurs modèles linéaires avec une interface davantage orientée vers les statistiques. Si vous n'êtes pas marié avec Python, vous pouvez aussi prendre en considération le langage R, également très apprécié des *data scientists*. R est un langage conçu spécifiquement pour les analyses statistiques et il est réputé pour ses excellentes capacités de visualisation ainsi que pour la disponibilité de nombreux compléments (souvent extrêmement spécialisés) de modélisation statistique.

Un autre outil populaire dans le domaine de l'apprentissage automatique est `vowpal wabbit` (souvent désigné par ses initiales vw pour éviter les problèmes de diction). Hautement optimisé, il est écrit en C++ et dispose d'une interface en mode ligne de commande. Pour l'exécution d'algorithmes d'apprentissage automatique distribués sur un cluster, l'une des solutions les plus répandues à l'heure actuelle est `mllib`, une librairie Scala construite sur l'environnement de traitement distribué `spark`.

## Ranking, systèmes de recommandation, et autres types d'apprentissage

Comme ce livre se veut être une introduction, nous nous sommes concentrés sur les tâches les plus courantes de l'apprentissage automatique, aussi bien supervisé que non supervisé. Mais il existe bien d'autres types d'apprentissage automatique, avec nombre d'applications importantes. En particulier, deux sujets importants n'ont pas été abordés dans le livre. Le premier est ce que l'on appelle le *ranking*, système dans lequel nous voulons retrouver des réponses à une certaine requête en les classant selon leur pertinence. En fait, vous êtes sans cesse confronté à cette technique, car c'est ainsi que fonctionnent les moteurs de recherche. Vous saisissez une requête, et vous obtenez une liste de réponses classées selon un ordre supposé de pertinence (voyez par exemple à ce sujet le livre *Introduction to Information Retrieval* de Manning, Raghavan et Schütze).

Le second sujet concerne les *systèmes de recommandation* qui fournissent des suggestions aux utilisateurs en fonction de leurs préférences. Là encore, il est plus que probable que vous rencontrez couramment ce type de système, par exemple avec Facebook ou Google+ (avec tous ces amis potentiels qui vous sont aimablement suggérés), ou encore avec Amazon et la plupart des sites marchands (« Les clients ayant acheté cet article ont également acheté », « Vos recommandations en vedette », et ainsi de suite). Une autre application classique est la prédiction d'évolutions dans le temps, comme dans le cas de cours de la Bourse. À nouveau, le domaine de l'apprentissage automatique est encore bien plus vaste que ce que nous pouvons mentionner ici, et nous ne pouvons que vous encourager à rechercher des ouvrages, des cours ou des articles en ligne, ainsi que des communautés de spécialistes ou d'utilisateurs.

## Inférence, programmation et modélisation probabiliste

La plupart des systèmes d'apprentissage automatique fournissent des modèles prédéfinis qui appliquent un algorithme spécifique. Cependant, de nombreux problèmes du monde réel ont une structure particulière qui, lorsqu'elle est incorporée correctement dans le modèle, peut conduire à des prédictions plus efficaces et plus précises. Souvent, la structure d'un problème particulier peut être exprimée en utilisant le langage de la théorie des probabilités. Une telle structure provient d'un modèle mathématique décrivant la situation pour laquelle vous voulez effectuer une prédiction. Pour comprendre ce que nous entendons par problème structuré, considérons l'exemple suivant.

Supposons que vous vouliez développer une application mobile qui fournisse une estimation de position très détaillée dans un espace extérieur afin d'aider les utilisateurs à naviguer dans un site historique. Un appareil mobile contient de nombreux capteurs qui vous permettent de mesurer une position avec précision, comme le GPS, l'accéléromètre, le gyromètre et ainsi de suite. Vous disposez également d'une carte exacte des lieux concernés. Ce problème est hautement structuré. Vous savez où passent les voies et où se trouvent les points d'intérêt. Le GPS vous donne une position approximative, tandis que les autres capteurs du smartphone de l'utilisateur

vous fournissent des mesures relatives très précises. Mais se contenter de jeter toutes ces informations en vrac dans un système $x$ d'apprentissage automatique (donc dans une boîte noire) afin de prédire les positions n'est peut-être pas la meilleure idée. En fait, vous savez mieux que le système la manière dont les choses se passent dans le monde réel. Si les capteurs vous disent qu'un utilisateur se dirige vers le nord, et que le GPS vous indique au contraire qu'il va vers le sud, vous ne pouvez probablement pas vous fier à ce dernier. Si l'estimation de la position vous indique que l'utilisateur vient de traverser un mur, vous avez totalement le droit d'être très sceptique. Il est possible d'exprimer cette situation en utilisant un modèle probabiliste, puis d'utiliser l'apprentissage automatique ou l'inférence probabiliste pour déterminer le niveau de confiance à accorder à chaque mesure, et donc raisonner sur la détermination de la meilleure estimation possible quant à la position de l'utilisateur.

Une fois que vous avez pu exprimer la situation et votre modèle décrivant la manière dont les différents facteurs fonctionnent correctement ensemble, il existe des méthodes permettant de calculer directement des prédictions à partir de ces modèles personnalisés. La plus générale de ces méthodes fait appel à ce que l'on appelle des *langages de programmation probabiliste*. Ceux-ci fournissent une méthode très élégante et très compacte pour exprimer un problème d'apprentissage. Parmi les langages de programmation probabiliste les plus connus, citons `PyMC` (qui peut être intégré dans Python) et `Stan` (un environnement de développement qui peut être utilisé à partir de plusieurs langages, dont Python). Ces outils nécessitent bien sûr un certain niveau de compréhension de la théorie des probabilités, mais retenez qu'ils simplifient la création de nouveaux modèles de manière significative.

## Réseaux de neurones

Nous avons brièvement abordé le sujet des réseaux de neurones dans les Chapitres 2 et 7, mais il s'agit d'un domaine de l'apprentissage automatique qui évolue rapidement, avec des innovations et de nouvelles applications qui sont annoncées presque chaque semaine. Les récentes percées dans l'apprentissage automatique et l'intelligence artificielle, comme la victoire du programme AlphaGo contre le plus grand champion humain actuel du jeu de Go, l'amélioration constante des performances dans la compréhension du langage, ou encore la traduction presque instantanée entre des langues très diverses, sont possibles grâce à ces avancées. Certes, les progrès dans ces domaines sont si rapides que l'état actuel de l'art sera vite dépassé. Mais s'intéresser de près à la question des réseaux de neurones est un atout supplémentaire pour tout *data scientist* en devenir. À titre d'introduction sur ce sujet, nous vous recommandons le livre récent *Deep Learning* d'Ian Goodfellow, Yoshua Bengio et Aaron Courville (MIT Press). Voyez à ce sujet le site `http://www.deeplearningbook.org/`.

## Recalibrer les jeux de données

Dans ce livre, nous avons supposé que les données sur lesquelles nous travaillions pouvaient être placées dans des tableaux NumPy ou des matrices creuses SciPy et enregistrées dans la mémoire vive (la RAM) d'un ordinateur. Même si les serveurs modernes ont souvent une RAM qui se compte en centaines de gigaoctets, cela n'en reste pas moins une restriction fondamentale quant à la taille des données avec lesquelles vous pouvez travailler. Tout le monde ne peut pas s'offrir une machine aussi puissante, ni même en louer une auprès d'un prestataire du *cloud*. Certes, dans la plupart des applications, les données utilisées pour construire le système d'apprentissage automatique sont relativement petites, et finalement peu de jeux de données sont susceptibles d'atteindre une taille qui se chiffrerait en centaines de gigaoctets, voire plus. De ce fait, augmenter la quantité de mémoire vive dont dispose votre propre matériel, ou louer une machine puissamment équipée auprès d'un prestataire, reste une solution viable dans la plupart des cas. En revanche, si vos besoins atteignent des téraoctets de données, ou si vous devez traiter de très grandes quantités de données dans un cadre contraint, vous disposez de deux stratégies de base : l'apprentissage hors mémoire (*out-of-core learning*) et la parallélisation dite sur un cluster (*parallelization over a cluster*).

L'apprentissage hors mémoire décrit un apprentissage à partir de données qui ne peuvent pas être stockées dans la mémoire principale, mais qui se déroule dans un seul ordinateur (voire même dans un seul processeur de l'ordinateur). Les données sont lues à partir d'une source comme un disque dur ou une unité en réseau, à raison d'un exemple à la fois ou bien d'une collection d'exemples, de manière à ce que chaque composant individuel tienne en mémoire. Ce sous-ensemble du jeu de données est alors traité, et le modèle est mis à jour afin de refléter ce qui a été appris à partir des données. Le conglomérat de données courant est éliminé, et une nouvelle collection est chargée en mémoire. Cette technique d'apprentissage hors mémoire est implémentée dans certains des modèles de `scikit-learn`, et vous pourrez en apprendre plus à ce sujet dans son guide utilisateur (`http://scikit-learn.org/stable/modules/scaling_strategies.html#scaling-with-instances-using-out-of-core-learning`). Du fait que cette stratégie nécessite que toutes les données soient traitées par un même ordinateur, cela peut se traduire par des temps d'exécution importants dans le cas de jeux de données de très grande taille. De plus, tous les algorithmes d'apprentissage ne peuvent pas être implémentés de cette manière.

L'autre stratégie consiste à distribuer les données sur de multiples machines disposées en grappe (*cluster*) et à laisser chaque ordinateur traiter une partie de ces données. Cela peut être plus rapide pour certains modèles. De plus, la taille des données susceptibles d'être traitées de cette manière n'est limitée que par l'étendue du cluster. Cependant, cette technique nécessite souvent une infrastructure relativement complexe. L'une des plates-formes d'informatique distribuées les plus populaires à l'heure actuelle est `spark`, qui est bâtie sur `Hadoop`. `Spark` inclut un certain nombre de fonctionnalités d'apprentissage automatique contenues dans la librairie `MLLib`. Si vous utilisez déjà le système de fichiers Hadoop, ou même `spark`, pour le traitement de vos données, il peut s'agir de la solution la plus simple. Si vous ne disposez pas encore d'une telle

infrastructure, il est possible que l'effort à fournir soit trop important. Dans ce cas, la librairie vw, mentionnée plus haut, pourrait constituer une meilleure solution.

### Améliorer vos compétences

Comme pour bien d'autres choses dans l'existence, seule la pratique dans les domaines traités dans ce livre peut vous aider à devenir un(e) expert(e). Des actions comme l'extraction, le prétraitement, la visualisation ou la construction d'un modèle peuvent largement varier selon les buts poursuivis et les jeux de données à traiter. Vous avez peut-être la chance d'avoir présentes à l'esprit toutes une série de tâches à accomplir et de jeux de données sur lesquels travailler. Si ce n'est pas le cas, voyez du côté des ressources universitaires disponibles, et en particulier des sites où s'affrontent des équipes de compétiteurs s'efforçant d'obtenir les meilleures prédictions possibles à partir d'un certain jeu de données proposé à la sagacité des participants. Regardez par exemple comment fonctionne Kaggle (`https://www.kaggle.com/`), où des entreprises proposent de véritables challenges, dotés parfois de prix importants.

Une autre adresse très intéressante est celui de la plate-forme OpenML (`https://www.openml.org/`), sur laquelle vous pouvez notamment trouver des milliers de jeux de données à analyser dans des domaines très divers, et encore plus de projets de tâches accomplir. Tout cela peut vous aider considérablement à progresser et à développer vos compétences. Bien entendu, l'inconvénient de ce mode de fonctionnement est que les jeux de données y sont généralement figés et que l'enjeu consiste bien souvent à tenter d'améliorer une méthode déjà fixée. Mais la compétition peut tout de même être passionnante… N'oubliez pas cependant que définir le problème et collecter les données sont aussi des aspects importants des problèmes qui se posent dans la vraie vie, et que représenter un certain problème de la bonne manière peut être beaucoup plus important que d'améliorer la qualité d'une prédiction de quelques dixièmes.

## Conclusion

Nous espérons vous avoir convaincus de l'utilité de l'apprentissage automatique dans de nombreux champs d'application, et montré que son implémentation n'est pas aussi compliquée qu'on pourrait le penser. Alors, n'hésitez pas plonger dans un océan de données, mais sans jamais perdre de vue l'ensemble du paysage !

# Index


## A

Aire sous la courbe 295, 297, 305
Algorithmes 5
  apprentissage de variétés 167
  apprentissage supervisé 2
  chaînage 307
  de clustering 134, 194, 197
  k plus proches voisins 38
  lasso 56
  modèles linéaires 48, 59, 66
  pour l'apprentissage supervisé 33
  transformations non supervisées 133
Allocation de Dirichlet latente 350
Anaconda 7
Analyse
  de sentiment 327, 333
  de variance 238
ANOVA 238
Apprentissage automatique 1, 2, 359, 365
  analyser le problème 5
  autres ressources 364
  deep learning 107
  du prototype à la production 361
  entraînement des données 16
  hors-mémoire 367
  jeux de données 19
  non supervisé 133
    exemples d'applications 4
    problématiques 134
  supervisé 29
    algorithmes 33
    exemples d'applications 3
    résumé des modèles 129
  tester des systèmes de production 362
  théorie 363
  supervisé et prétraitement 141
Apprentissage de variétés 142, 167
Apprentissage profond 107
Arbres de décision 73
  analyser 78
  complexité 77
  construire 74
  ensembles 86
Arbres de régression 92
ARI 194
Astuce du noyau 100
Attributs 51
AUC 298

## B

Bagging 87
Bigrammes 342
Bin 147, 223
Binning 223
Boosting d'arbres de régression 92
Bootstrap 87
Boundary points 190
Bruit 145, 190
Bunch (objets) 37

## C

Calibration 291
Caractéristiques 5, 17, 18, 80
  catégorielles 213
    nombres entiers 220
  continues 213
  corrélées 143
  élimination récursive 242
  extraction 5, 142, 150
  ingénierie 38, 214
  non linéaires 95
  originales 157

pondération 48
sélection automatique 237
sélection d'après le modèle 240
sélection itérative 242
Chaînage de méthodes 70
Chaînes de caractères 325
Classes 16, 29
positive/négative 30
Classification 16, 29, 34
algorithmes 22
arbres de décision 82
binaire 29
métriques d'évaluation 279
*k* plus proches voisins 38
modèles linéaires 59, 66
multi-classes 16, 29
incertitude 127
méthode un contre tous 66
métriques d'évaluation 300
modèles linéaires 66
régression 30
SVM à noyau 95
SVM linéaire 59
Classifieurs 41
bayésiens naïfs 71
linéaires 59
taux d'incertitude 121
Clustering 133, 171
agglomératif 185, 205
critères d'arrêt 185
coefficient de silhouette 195
comparer les algorithmes 194, 197
DBSCAN 189
hiérarchique 187
*k*-moyennes 171, 176
résumé des méthodes 209
stabilité 197
Clusters 171, 367
données denses 190
Coefficient
de détermination 46
de silhouette 195
directeur 48
Composantes principales 143
Core points 190
Corpus 327
Couche cachée 108
Courbe précision-rappel 292
Courbe ROC 295
Courbes d'apprentissage 55
Cross-validation splitter 258

## D

DataFrame 11
Data science 6
Data scientists 6
DBSCAN 189
appliquer 190
bruit 190
paramètre eps 191
paramètre min_samples 192
Deep learning 107
librairies 119
Dendrogrammes 187
Détection
fraudes 360
données aberrantes 199
Diagramme de dispersion 21
Discrétisation 223
Distribution de Poisson 235
Document 327
Données 5
entraîner et tester 19
jeu d'apprentissage 19
jeu de test 19
notation X 19
sous forme de tableau 4
taille du jeu 33
textuelles 325
catégorielles 326
chaînes libres 326
mots vides 337
pondérer avec tf-idf 338
vérifier 20
visualiser 21
Dummy variables 215

## E

Échantillons 5, 17
Eigenfaces 150
Embarrassingly parallel 277
Encodage one-hot 215
Ensembles 86
Enthought Canopy 7
Epsilon-voisinage 190
Erreur quadratique moyenne 50
Estimateur 23, 211
  construire 362
Étiquette 29
  notation y 19
Évaluation hors-ligne 362
Exactitude 25, 279
Expertise 243
Expression régulière 335

## F

Factorisation en matrices non négatives 159
Faux
  négatif 280
  positif 280
Feature engineering 214
Feuille 74
  pure 76
f-mesure 286
Folds 254
Fonction
  de décision 122
  de perte 59
  linéaire 48
Forêts aléatoires 86
  analyser 88
  construire 86
  parallélisation 91
Forme 18
Fouille de textes 358
Frontière de décision 41, 59
f-score 286

## G

Généralisation 19, 30, 129, 253
  performances 53
GitHub 8
Gradient boosting 86, 92
Grid search 263

## H

Hadoop 367
Hyperplan 49

## I

Importance des caractéristiques 80
Incertitude 288
Ingénierie des caractéristiques 38, 214
Instanciation 23
Interactions 38, 95, 226
  PCA 147
Intercept 59
IPython 7
Iris de Fisher 16

## J

Jeux
  d'apprentissage 19, 29
  de données clairsemés 47
  de données, jouet 331
  de test 19
  de validation 264
  déséquilibrés 280
Jupyter Notebook 7, 8

## K

Kaggle 368
*k*-fold cross validation 254
*k*-moyennes 171, 176
*k* plus proches voisins 22, 38

## L


Langage de programmation 361
  probabiliste 366
Lasso 56
  paramètre alpha 56
  ridge 58
LDA 350
Leave-one-out 260
Lemmatisation 347
Lien 185
Linéaires (modèles) 48, 59, 66
Linkage 185
Liste de dictionnaires 274
LOO 260


## M


Manifold learning 167
MATLAB 6
matplotlib 7, 10
Matrice
  creuse 9, 332
  de confusion 282
    exactitude 285
  de dispersion 21
Métriques de régression 303
Métriques d'évaluation 278
  classification binaire 279
  classification multi-classes 300
  exactitude 279
  modèles 303
mglearn 12
mllib 364
Modèles
  calibrés 125
  chaînage 307
  complexité 32
    taille du jeu de données 33
  complexité et généralisation 42
  compromis entre surapprentissage et sous-apprentissage 32
  construire 22
  d'apprentissage automatique 16
  entraînement 48
  évaluer 25
  évaluer les performances 19
  exactitude 25
  généralisation 19
  *k* plus proches voisins 22, 38
  lasso 56
  linéaires 48, 59, 66
  métriques d'évaluation 303
  prédictions et 24
  recherche sur grille et 321
  résumé 129
  score 25
Modélisation
  de sujet 350
  probabiliste 365
Moindres carrés ordinaires 50
Mots vides 337


## N


nltk 347
NMF 159
  reconnaissances de visages et 161
NMI 194
Noyau
  polynomial 100
  RBF 100
Nuage de points 21
NumPy 7, 9


## O


Objets 16
OpenML 368
Operating point 291
Outlier detection 199
Outliers 136, 192


## P


pandas 11
  DataFrame 11, 217
  fonction get_dummies 217, 220
  tableaux Series 217

Parallélisation 91
PCA 143
  appliquer à la visualisation 145
  interactions 147
  Option whitening 154
  reconnaissances de visages 150
  réduction de la dimension 143
  variance 157
Perceptrons multicouche 107, 110
Pipeline 307, 310, 314
  accéder aux attributs 317
    des étapes 317
  combiner avec GridSearchCV 321
  construire 310
  créer avec make_pipeline 315
  recherche sur grille 311
Plis 254
Points
  aberrants 136, 192
  de données 4
  de fonctionnement 291
  intérieurs 190
Polynômes 226, 229
Post-élagage 77
PPV 286
Précision 285
  moyenne 295
Prédiction 24, 48
Pré-élagage 77
Prétraitement 105, 135, 217, 308
  appliquer 136
  apprentissage supervisé 141
  changer d'algorithme 142
Probabilités (prédire) 124
Python 6
  documentation 329
  librairies 8
  versions 13
  versions des librairies 13
Python(x,y) 8

## Q

Qualitatif/quantitatif 215
Quantification vectorielle 179

## R

$R^2$ (score) 46
Racine 75
Racinisation 347
RandomForestClassifier 86
Ranking 298, 365
Rappel 286
Recalibrage 135
Recherche sur grille 263
  avec validation croisée 266
  choisir un modèle 321
  consulter les résultats 270
  liste de dictionnaires 273
  pipeline 311
  stratégies de validation croisée 275
Reconnaissances de visages 150
  NMF 161
  PCA 150
Réduction de dimension 133, 142, 159
  PCA 143
Régression 29, 35
  arbres de décision 82
  classification 30
  *k* plus proches voisins 44
  lasso 56
  linéaire 50
    coefficients 50
  logistique 59
    paramètre C 61
  métriques 303
  modèles linéaires 48
  polynomiale 230
  ridge 52
  SVM à noyau et 95
Régularisation 52
  L1 56

L1 ou L2 69
L2 52
paramètres alpha et C 69
Relu 108
Réseaux de neurones 107, 358, 366
ajuster 110
algorithmes 120
complexité 120
Feed-forward 107
librairies Python 119
paramètre alpha 114
pénalité L2 114
récurrents 358
RFE 242
Ridge 52
lasso 58
paramètre alpha 53
R (langage) 364
ROC 295

## S

Sacs de mots 330
n-grammes 342
Science des données 360
scikit-learn 6
classes, fonctions, méthodes
AgglomerativeClustering 186
average_precision_score 295
BaseEstimator 363
BernoulliNB 71
classification_report 287
ClassifierMixin 363
confusion_matrix 282
CountVectorizer 331
cross_val_score 255, 303
decision_function 121, 127
DecisionTreeClassifier 77
DecisionTreeRegressor 77, 83
DummyClassifier 281
ElasticNet 58
Estimator 23
f_classif 238
fit 23, 70, 211
fit_predict 186
fit_transform 141
f_regression 238
GaussianNB 71
get_feature_name 333
get_feature_names 251
get_support 239
GradientBoostingClassifier 92
GridSearchCV 268, 274, 303
GroupKFold 261
inverse_transform 157
KFold 258
KMeans 173
KNeighborsClassifier 23, 41, 153
KNeighborsRegressor 44
LatentDirichletAllocation 351
linear_model.Ridge 52
LinearRegression 51
LinearSVC 60
load_breast_cancer 36
load_extended_boston 38
load_files 328
load_iris 16
LogisticRegression 60, 70
make_pipeline 315
metrics.scorer 305
MiniBatchKMeans 184
MinMaxScaler 136
MLPClassifier 110, 114
MultinomialNB 71
Normalizer 136
np.digitize 224
np.linespace 223
OneHotEncoder 221, 249
pca.inverse_transform 202
Pipeline 307, 314
PolynomialFeatures 229
precision_recall_curve 292
predict 24, 41, 211
predict_proba 121, 127
print_topics 352
RandomForestRegressor 86
RegressorMixin 363

RobustScaler 136
roc_auc_score 298
roc_curve 296
score 25, 41, 211
SelectFromModel 240
StandardScaler 135
TfidfTransformer 338
TfidfVectorizer 339
toarray 332
train_test_split 19
transform 137, 211, 337
TransformerMixin 363
TSNE 169
documentation en ligne 7
estimateurs 211
formes 18
installer 7
paramètres des modèles 24
SciPy 7, 9
dendrogrammes 187
Score 25
Séparateur
à vaste marge 95
de validation croisée 258
Série temporelle 245
Shuffle-split 260
Sous-apprentissage 32, 129
spacy 347
spark 278, 367
Stabilité 197
Statistiques univariées 238
statsmodels 364
Support Vector Machines. *Voir* SVM
Surapprentissage 32, 129
éviter 52
SVC 95
SVM 59, 95, 101
à noyau 95, 106
prétraitement des données 105
paramètre C 61, 102
paramètre gamma 102
SVR 95
Systèmes de recommandation 365

## T

Tangente hyperbolique 108
Taux de vrais positifs 286
tensorflow 358
Tests en ligne 362
tf-idf 338
Théorie des probabilités 365
Tokenization 330
TPR 286
Traitement du langage naturel 327
Transformateur 135
Transformation
appliquer 136
min-max 136
non linéaire univariée 234
non supervisée 133
Trigrammes 342
t-SNE 167

## U

Unigramme 343

## V

Valeur prédictive positive 286
Validation 264
croisée 254, 309
à *k*-plis 254, 257
analyser le résultat 270
avec groupes 261
bénéfices 256
contrôler 258
dans scikit-learn 255
désavantages 257
imbriquée 275
leave-one-out 260
recherche sur grille 266
shuffle-split 260, 261
stratégies 275
Variables
catégorielles 214
indicatrices 215

Vecteurs support 101
Vocabulaire 330
Vote 39
  souple 87
vowpal wabbit 364
Vrai
  négatif 284
  positif 284

## W

Weak learners 92
Whitening 154
word2vec 358

O'REILLY®

# Le Machine learning avec Python

Le machine learning (ou apprentissage automatique) est désormais partie intégrante de nombreuses applications commerciales et projets de recherche. Mais ce domaine ne reste pas l'apanage des grandes entreprises dotées d'un département en recherche et développement. Si vous connaissez un minimum le langage de programmation Python, vous apprendrez grâce à ce livre à concevoir vos propres solutions de machine learning. Avec la masse de données qui circulent actuellement, la seule limite que peuvent connaître vos applications de machine learning, c'est votre imagination.

Cet ouvrage énumère les étapes nécessaires à la création d'une application de machine learning réussie avec Python et la librairie scikit-learn. Ses auteurs se sont efforcés de ne pas trop insister sur les aspects mathématiques de l'apprentissage automatique, mais plutôt sur les utilisations pratiques de ces algorithmes. Si vous êtes déjà quelque peu familiarisé avec les librairies NumPy et matplotlib, vous n'en serez que plus à l'aise.

Au programme de ce livre :

- Concepts fondamentaux et applications de machine learning
- Avantages et inconvénients d'utiliser les algorithmes de machine learning les plus courants
- Comment représenter les données traitées par le machine learning, et sur lesquelles se concentrer
- Méthodes avancées d'évaluation de modèle et ajustement des paramètres
- Le concept de pipeline pour le chaînage des modèles et l'encapsulation du flux de travail
- Techniques de traitement des données textuelles
- Suggestions pour améliorer vos compétences en apprentissage automatique et en sciences des données

« Ce livre est une ressource aussi formidable que pratique pour tous ceux qui veulent se mettre au machine learning avec Python. Si seulement un tel ouvrage avait pu exister quand j'ai fait mes premiers pas sur scikit-learn ! »

Hanna Wallach
Chercheuse chez Microsoft Research

**Andreas Müller** *est titulaire d'un doctorat sur le machine learning soutenu à l'université de Bonn. Après avoir exercé comme chercheur en machine learning pour Amazon, il a rejoint le Center for Data Science de l'université de New York. C'est l'un des principaux développeurs et animateurs de scikit-learn.*

**Sarah Guido** *travaille comme data scientist pour différentes startups, comme Bitly. Titulaire d'un master en sciences de l'information de l'université du Michigan, elle est également reconnue pour ses talents de conférencière.*

671555 II-18 35 €
9 782412 034460

editionsfirst.fr

FIRST INTERACTIVE