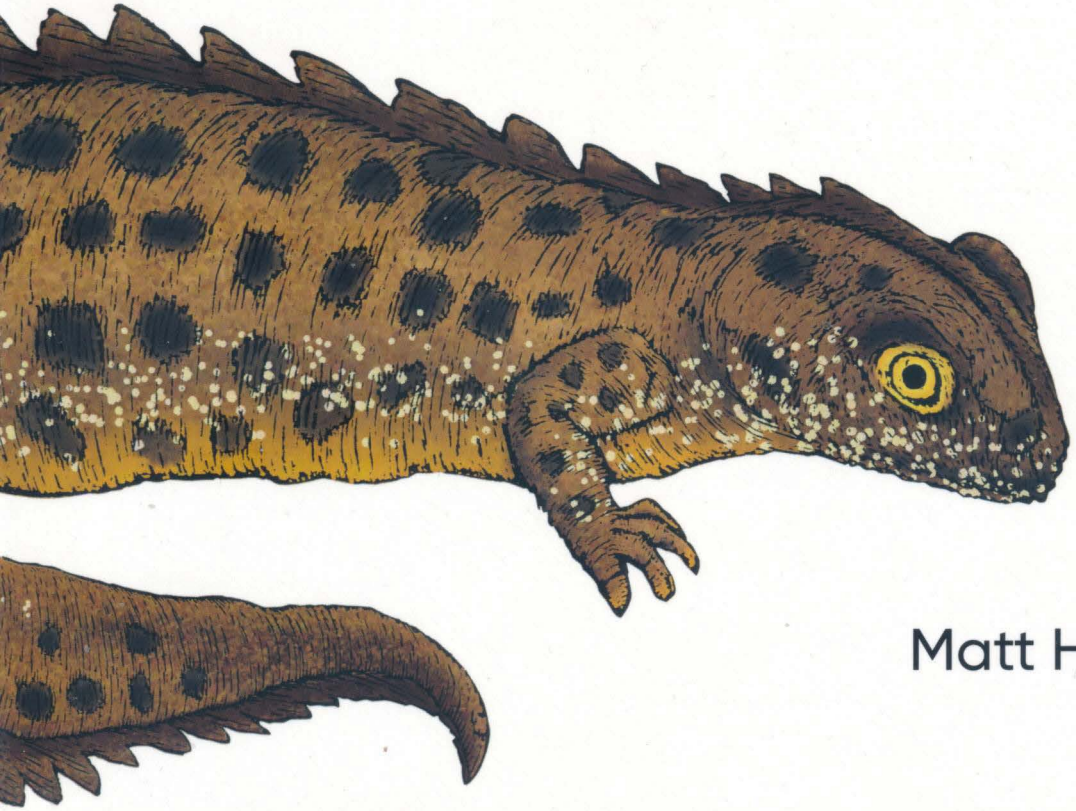
O'REILLY®

# Machine learning

## Les fondamentaux

### Exploiter des données structurées en Python



Matt Harrison

# Machine learning : les fondamentaux

*Matt Harrison*

# Table des matières

# Préface

Les domaines de l'apprentissage machine et de la datalogie (*data science*) sont devenus très recherchés et connaissent des changements à rythme soutenu. J'ai programmé en Python l'essentiel de ma carrière et j'ai ressenti le besoin d'un livre papier qui constituerait une sorte de référence de toutes les techniques que j'ai appliquées chez mes clients et que j'ai enseignées dans mes sessions de formation et ateliers de résolution de problèmes d'apprentissage machine.

Mon intention avec ce livre est de proposer une bonne collection de ressources et d'exemples pour résoudre un projet de modélisation prédictive en partant de données structurées. Un grand nombre de librairies sont disponibles pour réaliser certaines étapes de la chaîne de traitement ; j'ai essayé de mettre à contribution toutes celles qui me sont apparues utiles dans mes missions de consultant en datalogie.

Certains seront étonnés de ne pas voir abordé l'apprentissage profond (*deep learning*). Rappelons que ce domaine mérite un livre à lui seul. Je préfère exploiter des techniques plus simples, et mon point de vue semble assez partagé. Réservons l'apprentissage profond aux données non structurées (vidéo, audio, images) et profitons de puissants outils tels que XGBoost pour les données structurées.

J'espère que ce livre deviendra pour vous aussi un guide de référence utile qui vous aidera à résoudre vos défis.

## Contenu du livre

J'ai tenu à fournir de nombreux exemples de résolution des principaux problèmes relatifs à l'exploitation de données structurées. Je fais usage de dizaines de librairies spécialisées et de modèles en rappelant leurs avantages et inconvénients, comment les optimiser et comment interpréter les résultats produits.

Les extraits de code ont été dimensionnés afin d'être facilement réutilisables et adaptables dans vos projets.

## À qui s'adresse ce livre

Que vous fassiez vos premiers pas en datalogie ou ayez une certaine expérience, ce livre se veut une référence pragmatique. Je suppose un minimum de connaissances du langage Python, car je n'en rappelle pas la syntaxe. Pour l'essentiel, je montre comment bien exploiter des librairies pour résoudre des problèmes du monde réel.

Ce livre ne cherche pas à remplacer une formation dédiée ; il donne cependant les grandes lignes de la façon dont une telle formation au mécapprentisssage devrait se dérouler. (Note : l'auteur se sert de ce livre comme référence durant les formations d'analyse de données et de datalogie qu'il délivre.)

## Conventions typographiques

Pour rendre plus aisée l'exploitation du contenu du livre, nous avons pris soin de mettre en valeur certains termes par un codage visuel à portée sémantique, lorsqu'ils apparaissent dans le corps du texte :

`NomMotClé`

Nous imprimons ainsi toutes les entités fondamentales définies par le langage : noms des types, attributs, paramètres, méthodes et fonctions.

*`NomVariable`*

Noms des variables et d'objets choisis par le développeur.

**`Librairies`** et **`menus`**

Noms de librairies, de commandes et d'options de menus.

*NomFichier*

Noms de fichiers et de répertoires (dossiers).

`Lien`

Adresses Web et FTP.

*Concept*

Mise en valeur de la première apparition d'un terme essentiel.

Cette icône marque un paragraphe contenant une astuce ou une précision.

Cette icône marque une mise en garde ou bien souligne un concept pouvant être à l'origine d'une équivoque.

Cette icône marque une précision technique.

Cette icône marque un complément spécifique à la version française.

## Fichiers source des exemples

Pour chaque chapitre, tous les codes source des exemples sont réunis dans un fichier calepin (*notebook*). Les fichiers sont réunis dans une archive au format ZIP que nous vous invitons à télécharger depuis la page dédiée au livre sur le site de l'éditeur :

```
www.editionsfirst.fr
```

Recherchez le titre du livre grâce à la zone de recherche, puis dans la fiche du livre, cliquez sur **`Contenu additionnel`**, et enfin sur **`Télécharger`**.

N'hésitez pas à consulter l'éventuel fichier *LISEZMOI.txt* que vous trouverez dans ou à côté de l'archive des exemples si des remarques complémentaires ou des points de mise à jour rendent l'existence de ce fichier justifiée.

Le secteur de la datalogie étant en ébullition, il est probable que les exemples bénéficient de retouches après la sortie de la version française. N'hésitez donc pas à vous rendre sur la page de référence anglaise des exemples pour y récupérer les plus récentes versions :

```
https://github.com/mattharrison/ml_pocket_reference
```

Le code source du livre est à votre disposition ; vous pouvez l'incorporer à vos projets sans autorisation préalable. En revanche, vous ne pouvez pas citer *in extenso* ce code source dans un document papier ou numérique (DVD, PDF, *etc.*) sans avoir obtenu notre accord au préalable.

Si vous désirez sourcer une citation du code par exemple dans le cadre d'une formation, voici la formule à ajouter :

```
« Machine Learning Pocket Reference by Matt Harrison (O'Reilly). (C) 2019 Matt Har-
rison, 978-1-492-04754-4. »
```

Les données de test des exemples sont toutes extraites de plusieurs bases de données de test qui sont incorporées dans des librairies standard. Elles ne sont donc pas francisées. Nous utilisons principalement les données du naufrage du *Titanic* et des données immobilières de la région de Boston.

## À propos de l'auteur

Matt Harrison dirige une société de formation et de services autour de Python et de la datalogie nommée *MetaSnake*. Il utilise le langage Python depuis le début des années 2000 dans divers domaines : datalogie, gestion d'entreprise, stockage de données, tests et automatisation, gestion de piles logicielles open source, finances et recherche.

## Terminologie française

Plus encore que dans le domaine de la programmation informatique, la terminologie mérite un soin particulier en datalogie. Voici nos principales décisions.

| Terme utilisé | Terme anglais | Commentaires si nécessaire |
|---|---|---|
| *1 sur n* | one-hot | Encodage binaire dont tous les bits sauf un sont forcés à 0. On rencontre aussi « 1 parmi n ». |
| *aberrant* | outlier | Une valeur anormalement éloignée des autres observations. |
| *analyse en composantes principales* | Principal Component Analysis | Nous conservons le sigle anglais PCA. |
| *binning* | binning | Répartition de valeurs uniquement numériques en plusieurs bacs (*bins*). |
| *calepin (de Jupyter)* | notebook | Le terme *notebook* sert d'abord à désigner un ordinateur portable. Le terme calepin permet de bien distinguer le format de fichier hybride qui combine des blocs de code source et des commentaires et des graphiques. |
| *centroïde* | centroïd | Nous préférons *centroïde* à *centre de masse*. |

| Terme utilisé | Terme anglais | Commentaires si nécessaire |
|---|---|---|
| *datalogie* | data science | L'expression *science des données* tente de remplacer « datascience ». *Datalogie* utilise le même mécanisme de construction que *technologie*. |
| *datalogue* | data scientist | Personne produisant des informations essentielles (des discours, *logos*) par distillation probabiliste de données brutes. |
| *ensachage* | bagging | Notez que le terme anglais proviendrait d'une contraction de Boostrap AGGregation-INGg. |
| *erreur quadratique moyenne* | Mean Squared Error | S'il est nécessaire, nous conservons le sigle anglais MSE. |
| *exactitude* | accuracy | Ne pas confondre avec précision. |
| *grappe* | cluster | Résultat de l'opération de partitionnement ou regroupement (*clustering*) |
| *mécapprentissage* | machine learning (ML) | L'expression « apprentissage machine » va devenir tellement usitée qu'un terme plus concis est la bienvenue. |
| *moindres carrés ordinaires* | Ordinary Least Squares | Nous conservons le sigle anglais OLS. |
| *recherche grille* | grid search | Ou recherche systématique |
| *regroupement* | clustering | Technique de classification non supervisée appelée aussi partitionnement. |
| *surajustement* | overfitting | On utilise aussi *surapprentissage*. |

# Colophon

L'animal présenté sur la couverture du livre est un *triton crêté* (*Triturus cristatus*). Cet amphibien vit près et dans les eaux douces stagnantes dans l'Europe septentrionale et dans l'ouest de la Russie.

Son dos est brun-gris avec des taches plus sombres et son ventre est jaune-orangé parsemé de taches blanches. Les mâles arborent une belle crête crénelée allant jusqu'à la queue pendant la période de reproduction. Les femelles ont toujours une bande orange sur la queue.

Le triton crêté hiberne dans la boue ou sous un rocher. Dans l'eau, il se nourrit d'autres tritons, de têtards, de vers, de larves et d'escargots d'eau douce ; sur la terre ferme, il chasse des insectes, des vers et autres invertébrés. Il peut vivre jusqu'à 27 ans et mesure jusqu'à 15 cm.

La femelle pond un œuf à la fois sur une feuille de myosotis des marais. Une fois l'œuf collé, elle replie avec sa queue le bout de la feuille sur l'œuf pour le rendre moins visible (sans l'écraser !). Elle répète cette opération pour plusieurs centaines d'œufs !

Le statut de conservation du triton crêté est Préoccupation mineure LC (*Least Concern*), mais la plupart des espèces présentées sur les couvertures de cette collection sont menacées ; toutes sont pour autant importantes pour la planète.

# CHAPITRE 1
# Introduction

Il ne s'agit pas ici d'un manuel pédagogique, mais plutôt d'une collection de notes techniques, de tableaux et d'exemples autour de l'apprentissage machine. L'auteur a constitué cet aide-mémoire papier au long de ses sessions de formateur pour qu'il serve de ressource écrite stable, et il l'a distribué aux participants de ses formations. Ceux des participants qui appréciaient la possibilité d'annoter leur exemplaire (quoi que l'on pense des arbres sacrifiés pour produire le papier) sont ainsi repartis des formations avec un document personnalisé et une série d'exemples directement applicables.

Nous allons découvrir les actions de classification avec des données structurées, celles de régression pour prédire des valeurs continues, la création de clusters, la réduction de dimensionnalité, et d'autres techniques. Le livre ne décrit pas les techniques de deep learning qui conviennent plutôt aux données non structurées, alors que les techniques que nous présentons s'appliquent aux données structurées.

Nous supposons que vous connaissez un peu le langage Python. Il n'est pas inutile de savoir exploiter des données avec la librairie **`pandas`** (`https://pandas.pydata.org`), car nous l'utilisons dans de nombreux exemples du fait qu'elle convient bien aux données structurées. Certaines opérations d'indexation vous paraîtront mystérieuses si vous ne connaissez pas la librairie *numpy*. Notez que la description complète de *pandas* occuperait un livre entier.

## Les librairies utilisées

Nous allons utiliser progressivement un grand nombre de librairies dans ce livre, ce qui offre des avantages et des inconvénients. En effet, certaines librairies vont peut-être créer des conflits de versions lorsque vous allez les ajouter. Ne vous en alarmez pas outre mesure ; certaines librairies ne servent qu'à un ou deux exemples dans un seul chapitre. Vous pouvez donc passer la section récalcitrante pour y revenir plus tard. Adoptez une approche d'installation « à la demande »

lorsque la tentative d'exécuter un exemple provoque l'apparition d'un message indiquant qu'une librairie manque.

Voici en guise d'aperçu la liste complète des librairies auxquelles nous ferons appel avec les numéros de version qui étaient en vigueur lors de l'écriture du livre. N'exécutez cependant pas cette commande :

```
>>> import autosklearn, catboost,
category_encoders, dtreeviz, eli5, fancyimpute, fastai, featuretools,
glmnet_py, graphviz, hdbscan, imblearn, janitor, lime, matplotlib,
missingno, mlxtend, numpy, pandas, pdpbox, phate, pydotplus, rfpimp,
scikitplot, scipy, seaborn, shap, sklearn, statsmodels, tpot,
treeinterpreter, umap, xgbfir, xgboost, yellowbrick
>>> for lib in [
...     autosklearn,
...     catboost,
...     category_encoders,
...     dtreeviz,
...     eli5,
...     fancyimpute,
...     fastai,
...     featuretools,
...     glmnet_py,
...     graphviz,
...     hdbscan,
...     imblearn,
...     lime,
...     janitor,
...     matplotlib,
...     missingno,
...     mlxtend,
...     numpy,
...     pandas,
...     pandas_profiling,
...     pdpbox,
...     phate,
...     pydotplus,
...     rfpimp,
...     scikitplot,
...     scipy,
...     seaborn,
...     shap,
...     sklearn,
...     statsmodels,
...     tpot,
...     treeinterpreter,
...     umap,
...     xgbfir,
...     xgboost,
...     yellowbrick,
... ]:
...     try:
```

```
...         print(lib.__name__, lib.__version__)
...     except:
...         print("ABSENTE ", lib.__name__)
catboost 0.11.1
category_encoders 2.0.0
ABSENTE dtreeviz
eli5 0.8.2
fancyimpute 0.4.2
fastai 1.0.28
featuretools 0.4.0
ABSENTE glmnet_py
graphviz 0.10.1
hdbscan 0.8.22
imblearn 0.4.3
janitor 0.16.6
ABSENTE lime
matplotlib 2.2.3
missingno 0.4.1
mlxtend 0.14.0
numpy 1.15.2
pandas 0.23.4
ABSENTE pandas_profiling
pdpbox 0.2.0
phate 0.4.2
ABSENTE pydotplus
rfpimp
scikitplot 0.3.7
scipy 1.1.0
seaborn 0.9.0
shap 0.25.2
sklearn 0.21.1
statsmodels 0.9.0
tpot 0.9.5
treeinterpreter 0.1.0
umap 0.3.8
xgboost 0.81
yellowbrick 0.9
```

Pour installer ces librairies, vous pouvez utiliser l'un des deux outils **`pip`** ou **`conda`**. L'acronyme **`pip`** vient de *Pip Installs Python* ; l'outil est installé d'office avec le langage Python. L'outil **`conda`** fait partie de l'atelier Anaconda (`https://anaconda.org`).

Certaines librairies réclament un traitement spécifique :

- **`fastai`** demande une option : `pip install --no-deps fastai`
- **`umap`** s'installe avec `pip install umap-learn`
- **`janitor`** s'installe avec `pip install pyjanitor`
- **`autosklearn`** s'installe avec `pip install auto-sklearn`

Pour effectuer mes analyses, je me sers en général de l'atelier « notebook » Jupyter, mais vous pouvez en choisir un autre. Sachez cependant que par exemple Google Colab est fourni avec de nombreuses librairies préinstallées mais éventuellement dans des versions obsolètes.

## Installation avec pip

Avant d'installer les librairies, nous allons créer un environnement secondaire virtuel qui servira de bac à sable. Ainsi, les nouvelles librairies ne risqueront pas de perturber celles qui sont éventuellement déjà en place pour d'autres projets. Voici comment créer cet environnement virtuel appelé *env* :

```
$ python -m venv env
```

Sous Linux et macOS, vous utiliserez **python** et sous Windows, **python3**. Si Windows ne reconnaît pas cette commande, il faudra peut-être réinstaller ou réparer l'application en vous assurant que l'option d'ajout de Python au chemin PATH (**Add Python to my PATH**) est activée.

Une fois cet environnement créé, il reste à l'activer en y entrant.

Voici d'abord la commande applicable sous Linux et macOS :

```
$ source env/bin/activate
```

Vous constatez que l'invite système affiche dorénavant le nom de l'environnement, ce qui confirme que vous y êtes entré :

```
(env) $ which python
env/bin/python
```

Sous Windows, vous activez l'environnement ainsi :

```
C:> env\Scripts\activate.bat
```

L'invite confirme l'activation de l'environnement virtuel :

```
(env) C:> where python
env\Scripts\python.exe
```

Vous pouvez maintenant installer une librairie avec **pip**, par exemple **pandas** :

```
(env) $ pip install pandas
```

Comme vu plus haut, pour certaines librairies, le fichier de paquetage qui les contient ne porte pas le même nom que la librairie. Voici comment trouver si une librairie est installée :

```
(env) $ pip search nomlibrairie
```

Une fois toutes les librairies en place, vous créez un fichier texte contenant les noms et numéros de versions avec l'outil **pip** :

```
(env) $ pip freeze > requirements.txt
```

Ce fichier, nommé par exemple *requirements.txt*, permettra ensuite de simplifier l'implantation de tous les paquetages dans un nouvel environnement virtuel :

```
(autre_env) $ pip install -r requirements.txt
```

## Installation avec conda

L'outil nommé **conda** fourni avec Anaconda permet lui aussi de créer des environnements et d'installer des paquetages. Il s'utilise de la même façon dans les trois systèmes Linux, macOS et Windows.

Pour créer un environnement virtuel nommé *env* :

```
$ conda create --name env python=3.6
```

Pour activer cet environnement :

```
$ conda activate env
```

L'invite confirme le basculement dans le bac à sable. Nous pouvons chercher une librairie :

```
(env) $ conda search nomlibrairie
```

Pour installer par exemple la librairie **pandas** :

```
(env) $ conda install pandas
```

Pour générer un fichier d'index des dépendances de tous les paquetages trouvés :

```
(env) $ conda env export > environnement.yml
```

Pour utiliser ce fichier afin d'installer tout en un geste dans un nouvel environnement :

```
(autre_env) $ conda create -f environnement.yml
```

Certaines des librairies utilisées dans ce livre ne sont pas présentes dans le référentiel Anaconda. Ne vous en inquiétez pas : utilisez dans ce cas l'outil **pip** depuis l'environnement **conda** sans créer un nouvel environnement virtuel emboîté.

CHAPITRE 2

# Le processus de mécapprentissage

Les activités relevant du minage de données ont fait l'objet d'une formalisation sous la forme d'un processus standard nommé CRISP-DM (*Cross-Industry Standard Process for Data Mining*). Il définit les grandes étapes successives permettant de profiter d'une amélioration continue. Voici ces six étapes :

- Compréhension du métier
- Compréhension des données
- Préparation des données
- Modélisation
- Évaluation
- Déploiement

La Figure 2.1 détaille le processus de travail que j'ai adopté pour disposer d'un modèle prédictif en me basant sur cette méthodologie CRISP-DM. Ces étapes seront parcourues au long du prochain chapitre.

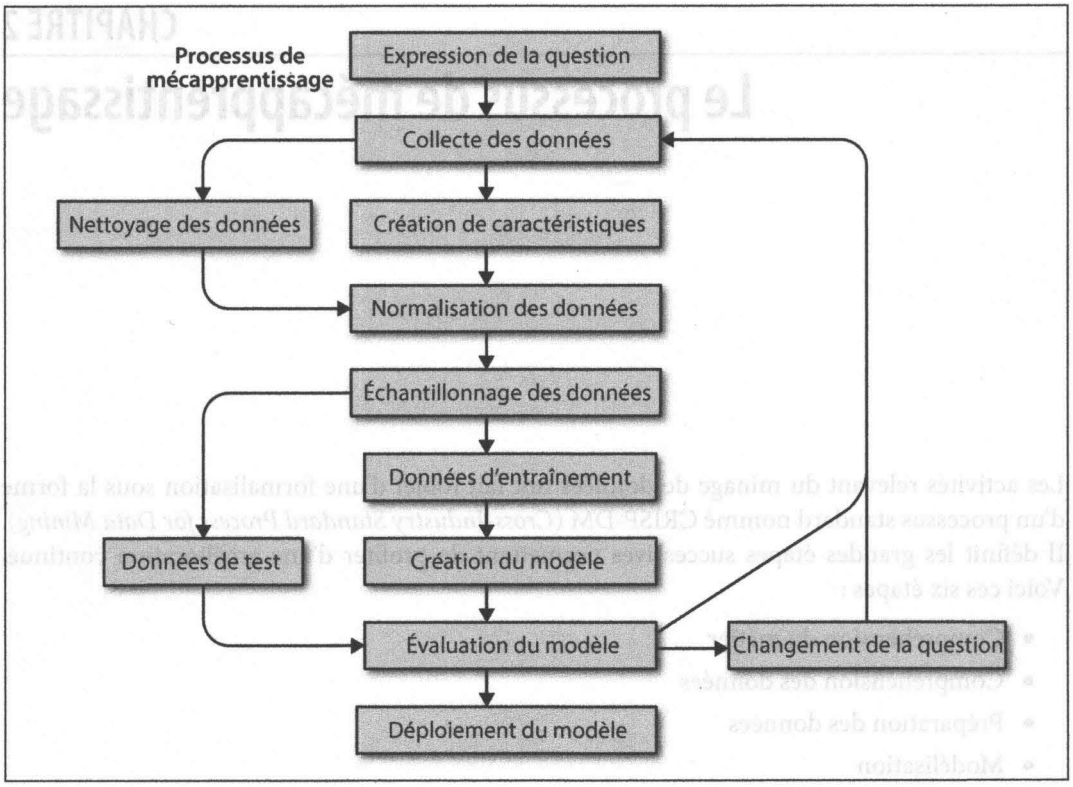



*Figure 2.1 : Processus classique de mécapprentissage.*

CHAPITRE 3

# Classification avec les données Titanic

Ce chapitre présente les étapes successives d'un problème de classification habituel, en travaillant sur le jeu de données Titanic (`https://oreil.ly/Pjce0`). Nous entrerons dans les détails des différentes étapes de l'analyse dans les chapitres ultérieurs.

## Suggestion de structure du projet

Un bon outil pour réaliser des analyses de données exploratoires se nomme **`Jupyter`** (`https://jupyter.org`). C'est un environnement open source basé sur des calepins (*notebooks*) qui sait gérer le langage Python et quelques autres. Son principe est de créer des cellules contenant soit du code, soit des commentaires selon le format Markdown.

Je me sers en général de Jupyter dans deux modes. Le premier sert aux analyses exploratoires pour tester rapidement quelques idées. Dans l'autre mode, je m'intéresse en particulier à la finition en vue d'une livraison, et j'ajoute des cellules d'explication avec un format de stylisation Markdown et d'autres cellules de code pour expliciter des points importants ou des découvertes. Si vous n'y prêtez pas garde, vos fichiers de calepin auront besoin d'être reconstruits en vue de mieux appliquer les bonnes pratiques de développement logiciel (suppression des éléments globaux, utilisation de fonctions et de classes, *etc.*)

Un paquetage dédié à la datalogie nommé **`cookiecutter`** (`https://oreil.ly/86jL3`) propose une structure type qui permet de créer une analyse permettant aisément d'utiliser et de partager du code source.

### Imports

L'exemple de ce chapitre utilise principalement trois librairies :

- **pandas** (`http://pandas.pydata.org/`) réunit des outils pour faciliter la consommation des données.
- **scikit-learn** (`https://scikit-learn.org/`) offre des fonctions de modélisation prédictive de haut niveau.
- **yellowbrick** (`http://www.scikit-yb.org/`) sert à créer des visualisations pour évaluer vos modèles.

```
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> from sklearn import (
...     ensemble,
...     preprocessing,
...     tree,
... )
>>> from sklearn.metrics import (
...     auc,
...     confusion_matrix,
...     roc_auc_score,
...     roc_curve,
... )
>>> from sklearn.model_selection import (
...     train_test_split,
...     StratifiedKFold,
... )
>>> from yellowbrick.classifier import (
...     ConfusionMatrix,
...     ROCAUC,
... )
>>> from yellowbrick.model_selection import (
...     LearningCurve,
... )
```

Il vous arrivera sans doute de lire des documentations avec des exemples qui utilisent des importations systématiques basées sur le signe astérisque, ainsi :

```
from pandas import *
```

N'utilisez pas cette technique globale. En indiquant explicitement ce que vous voulez importer, vous rendez votre code plus lisible.

## Poser les termes de la question

Dans l'exemple, nous voulons obtenir un modèle prédictif qui va répondre à une question. Nous allons créer une classification pour deviner les chances de survie d'un individu dans la catastrophe du *Titanic* en nous basant sur des caractéristiques de cette personne et de ses conditions de présence à bord. Il s'agit d'un exemple de test, ce qui n'en réduit pas les qualités pédagogiques pour illustrer les nombreuses étapes d'une bonne modélisation. Notre modèle va exploiter les informations des passagers pour prédire la survie d'un passager du naufrage du *Titanic*.

Il s'agit ici d'un projet de classification, puisque nous allons prédire des noms (des labels) pour la survie : soit la personne survit, soit elle décède.

## Terminologie des données

Pour effectuer l'entraînement du modèle, on utilise en général une matrice de données. Personnellement, j'ai une préférence pour la structure nommée `Dataframe` de la librairie **pandas** parce qu'elle permet d'avoir des labels pour les colonnes, mais ceci dit, les tableaux de **numpy** conviennent aussi.

Il s'agit ici d'un apprentissage supervisé, c'est-à-dire soit une régression, soit une classification. Il nous faut une fonction capable de produire des labels à partir des caractéristiques. Si nous écrivions ce processus sous forme d'une formule algébrique, elle prendrait l'aspect suivant :

```
y = f(X)
```

*X* correspond à la matrice. Chaque ligne est un échantillon de données concernant un passager. Chaque colonne est une caractéristique. Le résultat de la fonction est stocké dans *y*, qui est un vecteur contenant des labels dans le cas d'une classification, mais ce serait des valeurs dans le cas d'une régression (Figure 3.1).

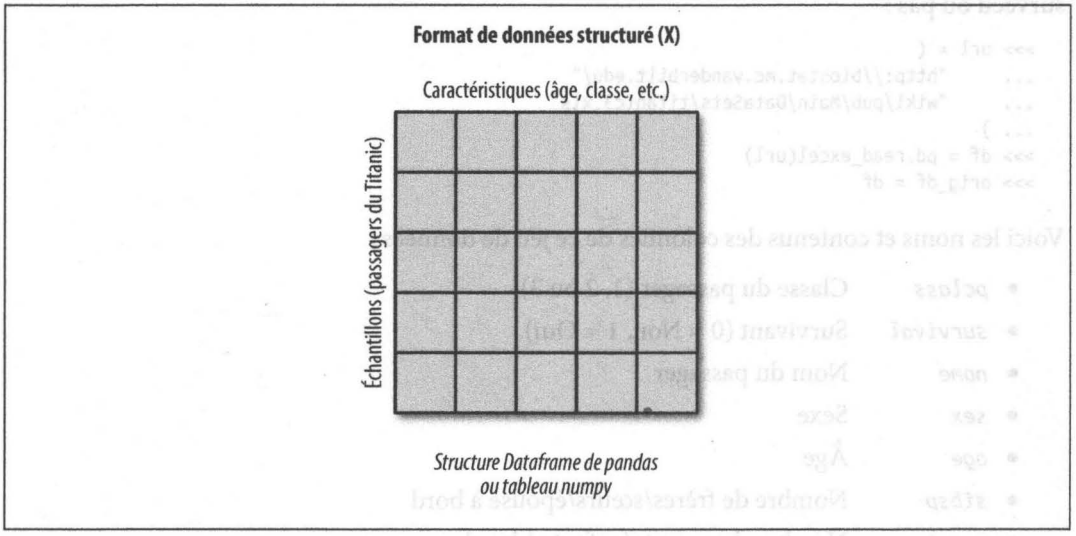



*Figure 3.1 : Format de données structuré.*

Ce sont les termes standard utilisés pour les données en entrée et celles en sortie. Vous les retrouverez dans les analyses des chercheurs et dans les documentations des librairies. En langage Python, nous avons l'habitude d'utiliser le nom de variable *X* pour les données d'entrée, bien que les majuscules contreviennent aux règles de nommage des variables Python (PEP 8). Ne vous

en souciez pas, car tout le monde suit cette convention. Vous seriez même l'objet de quolibets si vous choisissiez de nommer votre variable en minuscule, *x*. Quant à la variable *y*, elle va recevoir les labels ou plus généralement les cibles.

Le Tableau 3.1 montre un jeu de données d'entrée minimal, constitué de deux échantillons avec trois caractéristiques pour chacun.

*Tableau 3.1 : Échantillons (lignes) et caractéristiques (colonnes).*

| class | age | sibsp |
|---|---|---|
| 1 | 29 | 0 |
| 1 | 2 | 1 |

# Collecte des données

Nous allons charger le contenu d'un fichier de tableur Excel. Vérifiez que vous avez bien installé les deux librairies **pandas** et **xlrd**. (Nous n'appelons pas directement cette seconde librairie, mais elle est utilisée par **pandas** pour charger le fichier.) Le fichier de données du *Titanic* comporte un grand nombre de colonnes, et notamment une colonne *survived* qui indique si le passager a survécu ou pas :

```
>>> url = (
...     "http://biostat.mc.vanderbilt.edu/"
...     "wiki/pub/Main/DataSets/titanic3.xls"
... )
>>> df = pd.read_excel(url)
>>> orig_df = df
```

Voici les noms et contenus des colonnes de ce jeu de données :

- *pclass* Classe du passager (1, 2 ou 3)
- *survival* Survivant (0 = Non, 1 = Oui)
- *name* Nom du passager
- *sex* Sexe
- *age* Âge
- *sibsp* Nombre de frères/sœurs/épouse à bord
- *parch* Nombre de parents/enfants à bord
- *ticket* Numéro du billet
- *fare* Prix du billet
- *cabin* Cabine
- *embarked* Port d'embarquement (C = Cherbourg, Q = Queenstown, S = Southampton)

- *boat* Canot de sauvetage
- *body* N° d'identification du corps
- *home.dest* Domicile/destination

La librairie **pandas** va lire le contenu de ce fichier et le convertir directement en une structure de type `Dataframe`. Il nous faut maintenant vérifier l'état des données d'entrée pour garantir qu'elles conviennent à l'analyse.

## Nettoyage des données

Une fois que nous avons accès aux données, il reste à vérifier qu'elles se présentent dans un format exploitable pour la création du modèle. La plupart des modèles adaptés à la librairie **scikit-learn** demandent des caractéristiques de type numérique (entiers ou à virgule flottante). De plus, nombreux sont les modèles qui échouent en tombant sur des valeurs manquantes (correspondant à NaN dans **pandas** ou **numpy**). Certains modèles seront plus efficaces si les données sont standardisées, avec une valeur moyenne forcée à zéro et un écart-type borné à 1. Nous verrons ces différents aspects aussi bien pour **pandas** que pour **scikit**. Précisons que le jeu de données d'exemple Titanic comporte des caractéristiques divulgâcheuses.

Le terme anglais « *spoil* » a trouvé son équivalent français : *divulgâcher*, puisqu'il s'agit de gâcher l'effet d'une situation finale en divulguant des indices.

Une caractéristique divulgâcheuse (*leaky*) est une variable qui contient des informations permettant de directement prédire un élément futur ou une cible. Cela n'est pas un problème en soi, et nous vous allons souvent rencontrer de telles données pendant la création des modèles. Cependant, si ces variables, disponibles dans l'entraînement ne le sont plus lors de la génération d'une prédiction sur un nouvel échantillon, il faut les supprimer du modèle, car elles faussent le traitement en donnant des indices sur les résultats escomptés, mais pas pour tous les échantillons.

Le nettoyage ou préparation des données peut prendre un certain temps. Il est souvent utile de pouvoir faire intervenir un expert métier (SME) qui donnera des conseils essentiels quant à la façon de gérer les valeurs aberrantes et manquantes.

Voyons la liste des types de données de notre jeu :

```
>>> df.dtypes
pclass      int64
survived    int64
name       object
sex        object
age       float64
sibsp       int64
```

```
parch          int64
ticket        object
fare         float64
cabin         object
embarked      object
boat          object
body         float64
home.dest     object
dtype:        object
```

Nous découvrons les types de données `int64`, `float64` et `object`. Ce sont les types que la librairie **pandas** attribue aux différentes colonnes de données pour leur stockage. Les deux types `int64` et `float64` sont numériques et le type `object` sert en général à stocker des chaînes de caractères, ou éventuellement une combinaison d'une chaîne et d'un autre type.

Lors du chargement du contenu d'un fichier CSV, **pandas** fait tout son possible pour attribuer les types appropriés, mais choisit le type `object` s'il ne trouve rien de mieux. Il y a en général moins d'erreurs d'attribution de types lorsque la source des données est une base ou un autre système de stockage structuré et le résultat stocké dans la structure `DataFrame` sera meilleur. Dans tous les cas, il est toujours conseillé de parcourir les données afin de vérifier que les types attribués sont cohérents.

En général, les types entiers ne posent pas de problème, et les types à virgule flottante acceptent même quelques valeurs manquantes. Les types chaîne et date doivent être convertis ou alors considérés comme des types numériques spécialisés définis par le programmeur. Lorsque le contenu d'un type chaîne ne peut prendre qu'un nombre limité de valeurs (petite cardinalité), cela correspond à une colonne catégorielle. Il est en général utile de créer des colonnes factices à partir de celle-ci, ce que permet de réaliser la fonction nommée `pd.get_dummies`.

Jusqu'à la version 0.23 de **pandas**, pour le type `int64`, vous étiez assuré qu'il n'y avait pas de valeurs manquantes. Pour le type `float64`, les valeurs pouvaient toutes être à virgule flottante, ou certaines de type entier avec des valeurs manquantes. La librairie **pandas** convertit en effet les valeurs entières qui comportent des valeurs manquantes en valeurs flottantes, car ce type accepte les valeurs manquantes. Quant au type `object`, il correspond normalement au type chaîne ou à une combinaison de chaîne et de numérique.

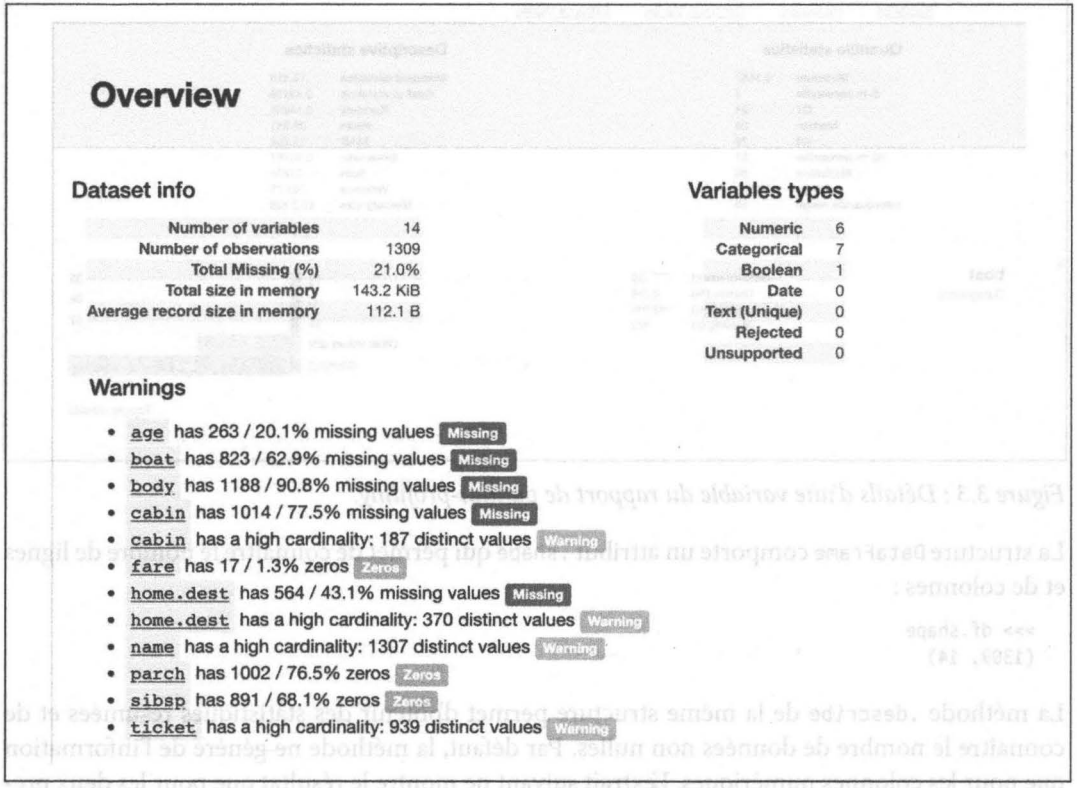
À partir de la version 0.24 de **pandas**, vous disposez d'un nouveau type qui s'écrit `Int64` (remarquez bien la majuscule). Ce n'est pas le type entier par défaut, mais vous pouvez forcer vers ce type pour permettre les valeurs manquantes dans les entiers.

La librairie nommée **`pandas-profiling`** permet de générer un rapport de profil directement dans votre outil Notebook. Cela vous permet d'obtenir les types de toutes les colonnes et d'accéder aux détails statistiques au niveau des quantiles, aux descriptions, à des histogrammes, et aux valeurs les plus fréquentes et extrêmes (Figures 3.2 et 3.3) :

```
>>> import pandas_profiling
>>> pandas_profiling.ProfileReport(df)
```

Vous aurez peut-être à installer la librairie mentionnée ainsi :

```
pip install pandas-profiling
```



## Overview

### Dataset info

| | |
|---|---|
| Number of variables | 14 |
| Number of observations | 1309 |
| Total Missing (%) | 21.0% |
| Total size in memory | 143.2 KiB |
| Average record size in memory | 112.1 B |

### Variables types

| | |
|---|---|
| Numeric | 6 |
| Categorical | 7 |
| Boolean | 1 |
| Date | 0 |
| Text (Unique) | 0 |
| Rejected | 0 |
| Unsupported | 0 |

### Warnings

- `age` has 263 / 20.1% missing values Missing
- `boat` has 823 / 62.9% missing values Missing
- `body` has 1188 / 90.8% missing values Missing
- `cabin` has 1014 / 77.5% missing values Missing
- `cabin` has a high cardinality: 187 distinct values Warning
- `fare` has 17 / 1.3% zeros Zeros
- `home.dest` has 564 / 43.1% missing values Missing
- `home.dest` has a high cardinality: 370 distinct values Warning
- `name` has a high cardinality: 1307 distinct values Warning
- `parch` has 1002 / 76.5% zeros Zeros
- `sibsp` has 891 / 68.1% zeros Zeros
- `ticket` has a high cardinality: 939 distinct values Warning

*Figure 3.2 : Résumé du rapport de pandas-profiling.*

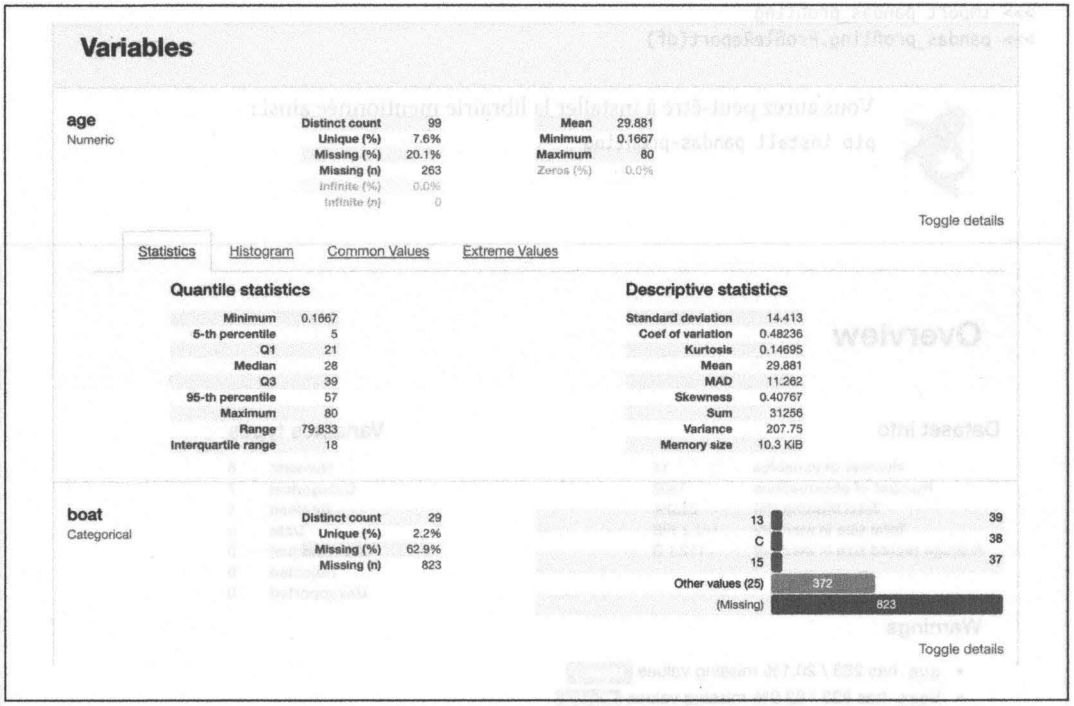



*Figure 3.3 : Détails d'une variable du rapport de pandas-profiling.*

La structure `DataFrame` comporte un attribut `.shape` qui permet de connaître le nombre de lignes et de colonnes :

```
>>> df.shape
(1309, 14)
```

La méthode `.describe` de la même structure permet d'obtenir des statistiques résumées et de connaître le nombre de données non nulles. Par défaut, la méthode ne génère de l'information que pour les colonnes numériques. L'extrait suivant ne montre le résultat que pour les deux premières colonnes :

```
>>> df.describe().iloc[:, :2]

            pclass     survived
count  1309.000000  1309.000000
mean      2.294882     0.381971
std       0.837836     0.486055
min       1.000000     0.000000
25%       2.000000     0.000000
50%       3.000000     0.000000
75%       3.000000     1.000000
max       3.000000     1.000000
```

La première statistique, count, ne dénombre que les valeurs différentes de NaN (non-nombre). Elle sert donc à vérifier s'il manque des données dans une colonne. Pensez également à vérifier les valeurs pour `min` et `max` afin de traquer les valeurs aberrantes. Cette traque des aberrants est également possible en demandant de créer un histogramme ou une boîte à moustaches, mais nous verrons cela plus tard.

Il faut traiter les données manquantes. Pour trouver les lignes ou les colonnes concernées, nous pouvons utiliser la méthode `.isnull` en l'appelant sur une structure `DataFrame` afin de récupérer une nouvelle structure `DataFrame` dont chaque cellule contient soit True, soit False. En Python, ces valeurs correspondent à 1 et 0, ce qui nous permet facilement d'obtenir la somme ou le pourcentage de manquants à partir de la moyenne.

L'exemple suivant indique le nombre de données manquantes dans chaque colonne :

```
>>> df.isnull().sum()
pclass         0
survived       0
name           0
sex            0
age          263
sibsp          0
parch          0
ticket         0
fare           1
cabin       1014
embarked       2
boat         823
body        1188
home.dest    564
dtype: int64
```

Pour obtenir le pourcentage de valeurs manquantes, vous remplacez `.sum` par `.mean`. Par défaut, ces méthodes s'appliquent aux valeurs selon l'axe 0, celui d'index (des lignes). Pour obtenir le nombre de caractéristiques manquantes pour chaque échantillon, donc chaque ligne, vous appliquez la méthode selon l'axe 1 qui est celui des colonnes :

```
>>> df.isnull().sum(axis=1).loc[:10]
0    1
1    1
2    2
3    1
4    2
5    1
6    1
7    2
8    1
9    2
dtype: int64
```

Si nécessaire, faites appel à un expert métier (un « SME », *Subject Matter Expert*) pour savoir quoi faire des données manquantes. La colonne *age* pourrait être utile au modèle, aussi vaut-il mieux la conserver en ajoutant si nécessaire des interpolations. En revanche, les colonnes avec beaucoup de manquants (*cabin*, *boat* et *body*) n'apportent pas de valeur et peuvent être abandonnées.

La colonne *body* qui reçoit le numéro d'identification du corps est par exemple fréquemment vide. Nous pouvons l'abandonner sans regret parce qu'elle divulgâche. En effet, le fait qu'il y ait un numéro de cadavre prouve que le passager n'a pas survécu. Le modèle pourrait se servir de cette information pour tricher dans ses prédictions. Nous ne voulons pas que notre modèle puisse profiter de ce genre d'information, mais qu'il se base uniquement sur les autres colonnes. Il en va de même pour la colonne *boat* qui correspond au numéro du canot de sauvetage, prouvant que le passager a pu embarquer.

Voyons maintenant quelques lignes avec des données manquantes. Procédons à la création d'un tableau de valeurs booléennes True/False pour repérer les lignes contenant des données manquantes :

```
>>> mask = df.isnull().any(axis=1)

>>> mask.head() # rows
0     True
1     True
2     True
3     True
4     True
dtype: bool

>>> df[mask].body.head()
0      NaN
1      NaN
2      NaN
3    135.0
4      NaN
Name: body, dtype: float64
```

Pour les valeurs manquantes dans la colonne *age*, nous créerons des valeurs dérivées par imputation plus tard.

Les colonnes de type `object` sont normalement catégorielles, même si elles peuvent contenir des chaînes avec beaucoup de valeurs différentes ou une combinaison de plusieurs types. Pour chacune des colonnes de type `object` que nous pensons être purement catégorielle, nous pouvons utiliser la méthode `.values_counts` pour connaître le nombre de valeurs :

```
>>> df.sex.value_counts(dropna=False)
male       843
female     466
Name: sex, dtype: int64
```

Rappelons que la librairie **pandas** ignore normalement les valeurs nulles ou NaN. Pour les prendre en compte, il faut ajouter l'option `dropna=False` ce qui permet de voir également les pseudo-valeurs NaN :

```
>>> df.embarked.value_counts(dropna=False)
S      914
C      270
Q      123
NaN      2
Name: embarked, dtype: int64
```

Pour les manquants de la colonne *embarked*, nous avons plusieurs options. Il pourrait sembler logique de forcer la valeur *S* puisque c'est la plus fréquente. En étudiant la situation plus en détail, nous pourrions trouver une autre option. Nous pourrions même ignorer les deux valeurs manquantes. Mais puisque c'est une colonne catégorielle, nous pouvons les ignorer et demander à **pandas** de créer des colonnes factices si les deux échantillons n'ont aucune entrée pour chacune des colonnes. C'est le choix que nous allons faire dans cet exemple.

# Création de caractéristiques

Nous pouvons bien sûr abandonner les colonnes contenant une constante. Dans le jeu qui nous intéresse, il n'y en a pas. S'il y avait une colonne *EstHumain* contenant systématiquement la valeur 1 pour chaque ligne, la caractéristique n'apporterait aucune information.

Sauf dans les projets comportant un volet de traitement du langage naturel NLP (*Natural Language Processing*) ou devant extraire des données des colonnes de texte, un modèle ne peut pas tirer profit d'une colonne dont les valeurs sont toutes différentes. C'est par exemple le cas de la colonne *name*. Certains ont d'ailleurs supprimé la partie titre du contenu (M. ou M^me^) puis traité la colonne comme catégorielle.

Comme déjà dit, il nous faut également abandonner les colonnes qui divulgâchent des informations, ce qui est le cas des colonnes *boat* et *body* qui permettent de deviner si un passager a survécu.

Utilisons donc maintenant la méthode `.drop` de **pandas** pour éliminer des lignes ou des colonnes :

```
>>> name = df.name
>>> name.head(3)
0     Allen, Miss. Elisabeth Walton
1     Allison, Master. Hudson Trevor
2     Allison, Miss. Helen Loraine
Name: name, dtype: object

>>> df = df.drop(
...     columns=[
...         "name",
...         "ticket",
```

```
...         "home.dest",
...         "boat",
...         "body",
...         "cabin",
...     ]
... )
```

Il nous faut créer des colonnes factices pour celles contenant du texte. Dans l'exemple, cela concerne les colonnes *sex* et *embarked*. Profitons de la fonction `get_dummies` de **pandas** :

```
>>> df = pd.get_dummies(df)

>>> df.columns
Index(['pclass', 'survived', 'age', 'sibsp', 'parch', 'fare', 'sex_female', 'sex_male',
'embarked_C', 'embarked_Q', 'embarked_S'], dtype='object')
```

Pour le moment, les deux colonnes *sex_male* et *sex_female* sont en corrélation inverse parfaite, ce qui n'a rien d'étonnant (si on ne gère pas le non-genré). Normalement, nous pouvons supprimer les colonnes avec une corrélation positive ou négative parfaite ou presque. En effet, cette colinéarité peut avoir un impact négatif sur la pondération des caractéristiques et les coefficients dans certains modèles. Voici par exemple comment supprimer la colonne *sex_male* :

```
>>> df = df.drop(columns="sex_male")
```

Nous pouvons également ajouter le paramètre `drop_first=True` lors de l'appel à `get_dummies` :

```
>>> df = pd.get_dummies(df, drop_first=True)

>>> df.columns
Index(['pclass', 'survived', 'age', 'sibsp', 'parch', 'fare', 'sex_male', 'embarked_Q',
   'embarked_S'], dtype='object')
```

*sex_female* et *embarked_C* ont disparu.

Nous créons une structure `DataFrame` *X* avec les caractéristiques, et une série *y* avec les labels. Nous pourrions à cet effet également utiliser des tableaux **numpy**, mais nous ne disposerions pas dans ce cas des noms des colonnes :

```
>>> y = df.survived
>>> X = df.drop(columns="survived")
```

Vous pourriez éventuellement utiliser la librairie **pyjanitor** (`https://oreil.ly/_IWbA`) à la place des deux lignes précédentes ainsi :

```
>>> import janitor as jn
>>> X, y = jn.get_features_targets(df, target_columns="survived")
```

Vous devez dans ce cas installer la librairie ainsi :

```
pip install janitor
```

## Sélection d'un échantillon de données

Il faut toujours appliquer l'entraînement et les tests sur des données différentes. C'est le seul moyen de vraiment savoir si le modèle peut bien s'appliquer à des données qu'il ne connaît pas encore. Nous nous servons de la librairie **scikit-learn** pour prélever 30 % du volume de données pour les tests. Nous utilisons l'option `random_state=42` pour désactiver la sélection aléatoire, ce qui permettra de comparer plusieurs modèles :

```
>>> X_train, X_test, y_train, y_test = model_selection.train_test_split(
...     X, y, test_size=0.3, random_state=42)
```

### Imputation de données

Il y a des valeurs manquantes dans la colonne *age*. Nous devons donc imputer des valeurs à partir des valeurs numériques. Nous ne voulons réaliser ces imputations que sur les données d'entraînement. Nous nous servirons de l'imputateur pour ajouter ensuite les données dans le jeu de test. Ce n'est qu'ainsi que nous pourrons éviter de divulgâcher les données, c'est-à-dire fournir des indices au modèle pour ses prédictions.

Pour imputer les valeurs manquantes sur le jeu d'entraînement puis utiliser les imputations de l'entraînement dans le jeu de test, nous pouvons profiter de la librairie **fancyimpute** (`https://oreil.ly/Vlf9e`) qui offre plusieurs algorithmes à cet effet. Le souci est que la plupart d'entre eux ne sont pas implémentés de façon *inductive*. Vous ne pouvez pas utiliser d'abord `.fit` puis `.transform`, ce qui signifie que vous ne pouvez pas imputer de nouvelles données à partir de la façon dont le modèle a été entraîné.

La classe nommée `IterativeImputer` faisait partie de **fancyimpute** mais a été rapatriée dans la librairie **scikit-learn**. Elle permet un mode inductif. Pour pouvoir l'utiliser, il nous faut ajouter un import expérimental spécial (tout du moins cela était nécessaire dans la version 0.21.2 de **scikit-learn**) :

```
>>> from sklearn.experimental import (
...     enable_iterative_imputer,
... )
>>> from sklearn import impute
>>> num_cols = [
...     "pclass",
...     "age",
...     "sibsp",
...     "parch",
...     "fare",
...     "sex_female",
... ]

>>> imputer = impute.IterativeImputer()
>>> imputed = imputer.fit_transform(
```

```
...     X_train[num_cols]
... )
>>> X_train.loc[:, num_cols] = imputed
>>> imputed = imputer.transform(X_test[num_cols])
>>> X_test.loc[:, num_cols] = imputed
```

Pour effectuer une imputation avec la médiane, nous pouvons nous servir de **pandas** :

```
>>> meds = X_train.median()
>>> X_train = X_train.fillna(meds)
>>> X_test = X_test.fillna(meds)
```

### Normalisation des données

De nombreux modèles fonctionneront mieux si vous réalisez d'abord une normalisation des données ou un prétraitement. C'est notamment le cas des modèles qui cherchent à obtenir une similarité à partir de distances. (Les modèles arborescents qui gèrent chaque caractéristique indépendamment n'ont pas cette contrainte.)

Nous allons donc standardiser les données pour permettre le prétraitement. Cela consiste à convertir les données pour que la valeur moyenne soit égale à zéro, avec un écart-type à 1. Nous évitons ainsi au modèle d'être perturbé par les variables exprimées avec une plus grande échelle que d'autres. Dans mon exemple, je vais réinjecter le résultat (un tableau **numpy**) dans une structure `DataFrame` de **pandas** pour simplifier la suite de la manipulation et maintenir les noms des colonnes.

Normalement, je ne standardise pas les colonnes factices ; je peux donc les ignorer :

```
>>> cols = "pclass,age,sibsp,fare".split(",")
>>> sca = preprocessing.StandardScaler()
>>> X_train = sca.fit_transform(X_train)
>>> X_train = pd.DataFrame(X_train, columns=cols)
>>> X_test = sca.transform(X_test)
>>> X_test = pd.DataFrame(X_test, columns=cols)
```

## Reformulation (*refactor*)

C'est en ce point de la progression que je décide en général de reformuler mon code source. Normalement, je définis deux fonctions. La première se charge du nettoyage et l'autre de la répartition des données en un jeu d'entraînement et un jeu de test, en réalisant les modifications qui doivent s'appliquer différemment sur chacun des deux jeux :

```
>>> def tweak_titanic(df):
...     df = df.drop(
...         columns=[
...             "name",
```

```
...             "ticket",
...             "home.dest",
...             "boat",
...             "body",
...             "cabin",
...         ]
...     ).pipe(pd.get_dummies, drop_first=True)
...     return df

>>> def get_train_test_X_y(
...     df, y_col, size=0.3, std_cols=None
... ):
...     y = df[y_col]
...     X = df.drop(columns=y_col)
...     X_train, X_test, y_train, y_test = model_selection.train_test_split(
...         X, y, test_size=size, random_state=42
...     )
...     cols = X.columns
...     num_cols = [
...         "pclass",
...         "age",
...         "sibsp",
...         "parch",
...         "fare",
...     ]
...     fi = impute.IterativeImputer()
...     X_train.loc[
...         :, num_cols
...     ] = fi.fit_transform(X_train[num_cols])
...     X_test.loc[:, num_cols] = fi.transform(
...         X_test[num_cols]
...     )
...
...     if std_cols:
...         std = preprocessing.StandardScaler()
...         X_train.loc[
...             :, std_cols
...         ] = std.fit_transform(
...             X_train[std_cols]
...         )
...         X_test.loc[
...             :, std_cols
...         ] = std.transform(X_test[std_cols])
...
...     return X_train, X_test, y_train, y_test

>>> ti_df = tweak_titanic(orig_df)
>>> std_cols = "pclass,age,sibsp,fare".split(",")
>>> X_train, X_test, y_train, y_test = get_train_test_X_y(
...     ti_df, "survived", std_cols=std_cols
... )
```

### Création du modèle de référence

En créant un modèle de référence qui effectue un traitement vraiment simple, nous disposons de quelque chose à quoi comparer notre modèle. Soyez prévenu que l'utilisation du résultat par défaut de `.score` fournit une précision qui peut être trompeuse. Lorsqu'il n'y a qu'une valeur positive sur 10 000, vous obtenez facilement 99 % de précision en faisant toujours une prédiction négative !

```
>>> from sklearn.dummy import DummyClassifier
>>> bm = DummyClassifier()
>>> bm.fit(X_train, y_train)
>>> bm.score(X_test, y_test) # accuracy
0.5292620865139949

>>> from sklearn import metrics
>>> metrics.precision_score(
...     y_test, bm.predict(X_test)
... )
0.4027777777777778
```

## Familles d'algorithmes

Notre exemple va tester plusieurs familles d'algorithmes. Le théorème selon lequel il n'y a rien de gratuit en ce bas monde rappelle qu'aucun algorithme ne peut être le meilleur pour toutes les données. Cela dit, un algorithme peut s'avérer bien adapté pour un jeu de données de volume fini. (L'apprentissage structuré est de nos jours souvent réalisé avec un algorithme à arbre amplifié tel que **`XGBoost`**.)

Nous allons donc utiliser plusieurs algorithmes et comparer le score AUC et l'écart-type STD en utilisant une validation croisée à k-plis. Nous privilégierons un algorithme s'il offre un écart-type plus resserré, même s'il a un score de moyenne AUC légèrement plus faible.

Vous devez installer auparavant la librairie **`XGBoost`** ainsi :

```
pip install XGBoost
```

Puisque nous réalisons une validation croisée à k-plis, nous fournissons la totalité de *X* et de *y* au modèle :

```
>>> X = pd.concat([X_train, X_test])
>>> y = pd.concat([y_train, y_test])
>>> from sklearn import model_selection
>>> from sklearn.dummy import DummyClassifier
>>> from sklearn.linear_model import (
...     LogisticRegression,
... )
```

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import (
...     KNeighborsClassifier,
... )
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.svm import SVC
>>> from sklearn.ensemble import (
...     RandomForestClassifier,
... )
>>> import xgboost

>>> for model in [
...     DummyClassifier,
...     LogisticRegression,
...     DecisionTreeClassifier,
...     KNeighborsClassifier,
...     GaussianNB,
...     SVC,
...     RandomForestClassifier,
...     xgboost.XGBClassifier,
... ]:
...     cls = model()
...     kfold = model_selection.KFold(
...         n_splits=10, random_state=42
...     )
...     s = model_selection.cross_val_score(
...         cls, X, y, scoring="roc_auc", cv=kfold
...     )
...     print(
...         f"{model.__name__:22} AUC: "
...         f"{s.mean():.3f} STD: {s.std():.2f}"
...     )
```

```
DummyClassifier         AUC: 0.511  STD: 0.04
LogisticRegression       AUC: 0.843  STD: 0.03
DecisionTreeClassifier  AUC: 0.761  STD: 0.03
KNeighborsClassifier    AUC: 0.829  STD: 0.05
GaussianNB               AUC: 0.818  STD: 0.04
SVC                      AUC: 0.838  STD: 0.05
RandomForestClassifier  AUC: 0.829  STD: 0.04
XGBClassifier           AUC: 0.864  STD: 0.04
```

## Empilement (stacking)

Si vous voulez privilégier les performances au détriment de la facilité d'interprétation en prenant la route Kaggle, vous pouvez opter pour la technique d'apprentissage ensembliste nommée *stacking*. Un classifieur de ce type prend les prédictions produites par d'autres modèles pour faire sa propre prédiction d'une cible ou d'un label. Nous allons nous servir dans l'exemple des résultats des autres modèles en les combinant pour voir si le classifieur à empilement donne de meilleurs résultats.

Vous devez d'abord installer la librairie mlxtend ainsi :

```
pip install mlxtend
```

```
>>> from mlxtend.classifier import (
...     StackingClassifier,
... )
>>> clfs = [
...     x()
...     for x in [
...         LogisticRegression,
...         DecisionTreeClassifier,
...         KNeighborsClassifier,
...         GaussianNB,
...         SVC,
...         RandomForestClassifier,
...     ]
... ]
>>> stack = StackingClassifier(
...     classifiers=clfs,
...     meta_classifier=LogisticRegression(),
... )
>>> kfold = model_selection.KFold(
...     n_splits=10, random_state=42
... )
>>> s = model_selection.cross_val_score(
...     stack, X, y, scoring="roc_auc", cv=kfold
... )
>>> print(
...     f"{stack.__class__.__name__} "
...     f"AUC: {s.mean():.3f} STD: {s.std():.2f}"
... )

StackingClassifier AUC: 0.804 STD: 0.06
```

Il semble que les performances ont un peu décru, ainsi que l'écart-type.

## Création d'un modèle

Je vais maintenant créer un modèle basé sur un classifieur de forêt aléatoire. J'obtiens ainsi un modèle souple et capable de fournir dès le départ des résultats satisfaisants. Il ne faut pas oublier de réaliser son entraînement par un appel à .fit sur les données d'entraînement, telles que nous les avions mises de côté en créant un jeu d'entraînement et un jeu de test :

```
>>> rf = ensemble.RandomForestClassifier(
...     n_estimators=100, random_state=42 ... )
>>> rf.fit(X_train, y_train)

RandomForestClassifier(bootstrap=True, class_weight=None,
    criterion='gini', max_depth=None, max_features='auto',
```

```
    max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    min_samples_leaf=1, min_samples_split=2,
    min_weight_fraction_leaf=0.0, n_estimators=10,
    n_jobs=1, oob_score=False, random_state=42,
    verbose=0, warm_start=False)
```

# Évaluation du modèle

Nous pouvons maintenant faire travailler notre modèle sur le jeu de données de test pour voir comment il sait généraliser avec des données qu'il ne connaît pas encore. Nous nous servons de la méthode `.score` du classifieur pour récupérer la moyenne de l'exactitude (*accuracy*) de prédiction. Il faut bien vous assurer d'appeler cette méthode avec les données de test, car elle va se montrer bien trop excellente avec les données d'entraînement :

```
>>> rf.score(X_test, y_test)
0.7964376590330788
```

Nous pouvons nous intéresser à d'autres métriques, et notamment à la précision :

```
>>> metrics.precision_score(
...     y_test, rf.predict(X_test)
... )
0.8013698630136986
```

Les modèles basés sur une arborescence ont l'avantage de permettre d'inspecter l'importance des caractéristiques, c'est-à-dire leur poids relatif. Vous pouvez ainsi savoir si une caractéristique contribue beaucoup ou pas au modèle. Sachez que le fait d'enlever une caractéristique ne provoque pas nécessairement une baisse du score, car certaines caractéristiques peuvent être colinéaires. Dans notre exemple, vous pourriez tout à fait supprimer soit la colonne ***sex_male***, soit la colonne ***sex_female***, puisqu'elles sont en corrélation inverse exacte :

```
>>> for col, val in sorted(
...     zip(
...         X_train.columns,
...         rf.feature_importances_,
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
age         0.277
fare        0.265
sex_female  0.240
pclass      0.092
sibsp       0.048
```

L'importance des caractéristiques est générée en surveillant l'augmentation de l'erreur. Lorsque le fait d'enlever une caractéristique augmente l'erreur du modèle, c'est que cette caractéristique est plus importante.

J'apprécie beaucoup la librairie SHAP pour découvrir quelles caractéristiques sont considérées comme importantes par le modèle, et donc pour expliquer ses prédictions. C'est une librairie qui fonctionne avec des modèles en format « boîte noire » ; nous la découvrirons plus loin.

## Optimisation par les hyperparamètres

Les modèles disposent d'une série d'hyperparamètres permettant de contrôler leur comportement. En jouant sur les valeurs de ces paramètres, nous influons sur les performances. La librairie **`sklearn`** possède une classe `GridSearchCV` qui permet d'évaluer un modèle en essayant plusieurs combinaisons de paramètres et en renvoyant les résultats. Nous pouvons nous servir de ces paramètres pour procéder à l'instanciation de la classe du modèle :

```
>>> rf4 = ensemble.RandomForestClassifier() >>> params = {
...     "max_features": [0.4, "auto"],
...     "n_estimators": [15, 200],
...     "min_samples_leaf": [1, 0.1],
...     "random_state": [42],
... }
>>> cv = model_selection.GridSearchCV(
...     rf4, params, n_jobs=-1
... ).fit(X_train, y_train)
>>> print(cv.best_params_)
{'max_features': 'auto', 'min_samples_leaf': 0.1,
'n_estimators': 200, 'random_state': 42}

>>> rf5 = ensemble.RandomForestClassifier(
...     **{
...         "max_features": "auto",
...         "min_samples_leaf": 0.1,
...         "n_estimators": 200,
...         "random_state": 42,
...     }
... )
>>> rf5.fit(X_train, y_train)
>>> rf5.score(X_test, y_test)

0.7888040712468194
```

Nous transmettons un paramètre nommé `scoring` à la classe `GridSearchCV` pour demander une optimisation selon plusieurs métriques. Nous présenterons la liste des métriques avec leur description dans le Chapitre 12.

## Matrices de confusion

Une matrice de confusion permet de vérifier la qualité d'un modèle en visualisant les classifications correctes ainsi que les faux positifs et faux négatifs. Nous pouvons en effet vouloir optimiser dans le sens des faux positifs ou des faux négatifs, ce qui est possible en choisissant le modèle ou en jouant sur les paramètres. Nous nous servons de **`scikit-learn`** pour obtenir une version texte, et de **`YellowBrick`** pour obtenir un diagramme (Figure 3.4) :

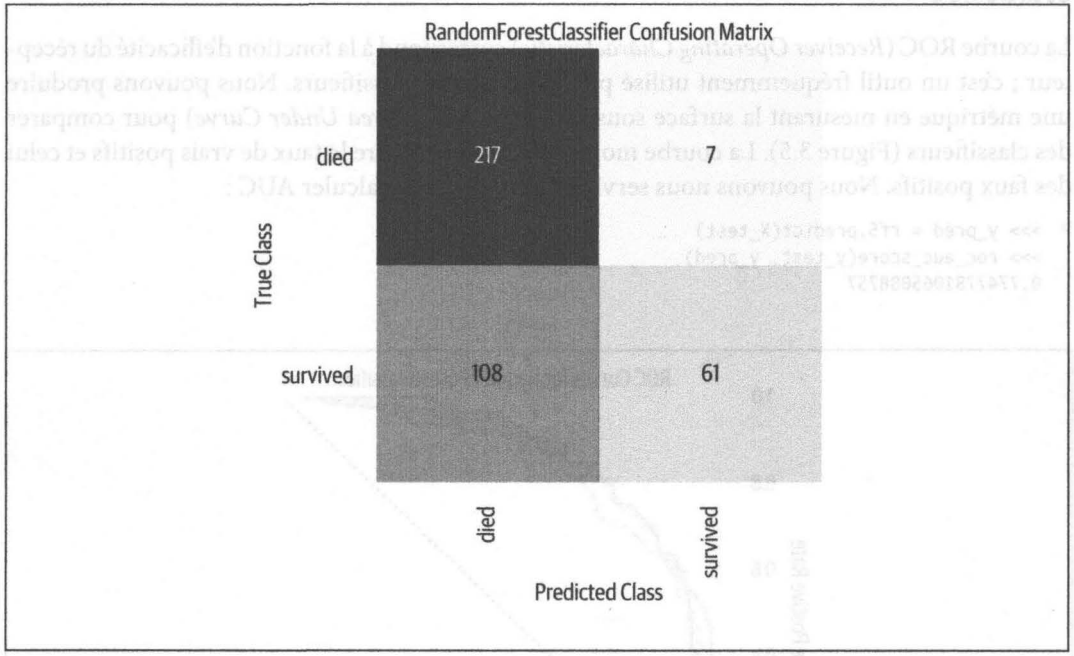



*Figure 3.4 : Matrice de confusion visualisée par YellowBrick. La classe prédite correspond à l'axe horizontal en bas et la classe réelle au sens vertical. Un bon classifieur montrerait toutes les valeurs dans la diagonale, avec des zéros dans les autres cellules.*

```
>>> from sklearn.metrics import confusion_matrix
>>> y_pred = rf5.predict(X_test)
>>> confusion_matrix(y_test, y_pred)
array([[196, 28],
       [ 55, 114]])
```

```
>>> mapping = {0: "died", 1: "survived"}
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cm_viz = ConfusionMatrix(
...     rf5,
...     classes=["died", "survived"],
...     label_encoder=mapping,
... )
```

```
>>> cm_viz.score(X_test, y_test)
>>> cm_viz.poof()
>>> fig.savefig(
...     "images/mlpr_0304.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

## Courbe ROC

La courbe ROC (*Receiver Operating Characteristic*) correspond à la fonction d'efficacité du récepteur ; c'est un outil fréquemment utilisé pour évaluer des classifieurs. Nous pouvons produire une métrique en mesurant la surface sous la courbe AUC (*Area Under Curve*) pour comparer des classifieurs (Figure 3.5). La courbe montre les relations entre le taux de vrais positifs et celui des faux positifs. Nous pouvons nous servir de **`sklearn`** pour calculer AUC :

```
>>> y_pred = rf5.predict(X_test)
>>> roc_auc_score(y_test, y_pred)
0.7747781065088757
```

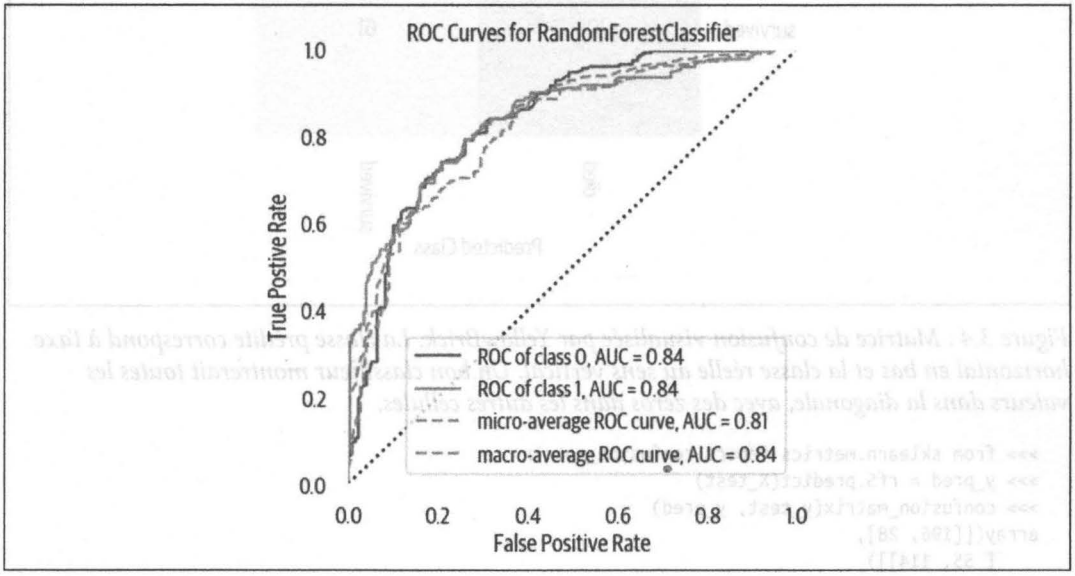



*Figure 3.5 : Courbe AUC. En général, plus la courbe est cintrée vers l'angle supérieur gauche, mieux c'est. La mesure AUC produit une seule valeur, qui doit se rapprocher de 1. Un modèle faible montre une valeur inférieure à 0.5.*

Pour obtenir une visualisation graphique, nous utilisons **`Yellowbrick`** :

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> roc_viz = ROCAUC(rf5)
>>> roc_viz.score(X_test, y_test)
0.8279691030696217

>>> roc_viz.poof()
>>> fig.savefig("images/mlpr_0305.png")
```

## Courbe d'apprentissage

La courbe d'apprentissage permet de vérifier que vous disposez d'assez de données d'entraînement. Le principe est d'entraîner le modèle en lui fournissant une proportion de plus en plus importante du volume de données et en mesurant le score (Figure 3.6). Si le score de validation croisée ne cesse d'augmenter, c'est qu'il faut sans doute rassembler un plus grand volume de données. Voici un exemple montré avec **`Yellowbrick`** :

```
>>> import numpy as np
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> cv = StratifiedKFold(12)
>>> sizes = np.linspace(0.3, 1.0, 10)
>>> lc_viz = LearningCurve(
...     rf5,
...     cv=cv,
...     train_sizes=sizes,
...     scoring="f1_weighted",
...     n_jobs=4,
...     ax=ax,
... )
>>> lc_viz.fit(X, y)
>>> lc_viz.poof()
>>> fig.savefig("images/mlpr_0306.png")
```

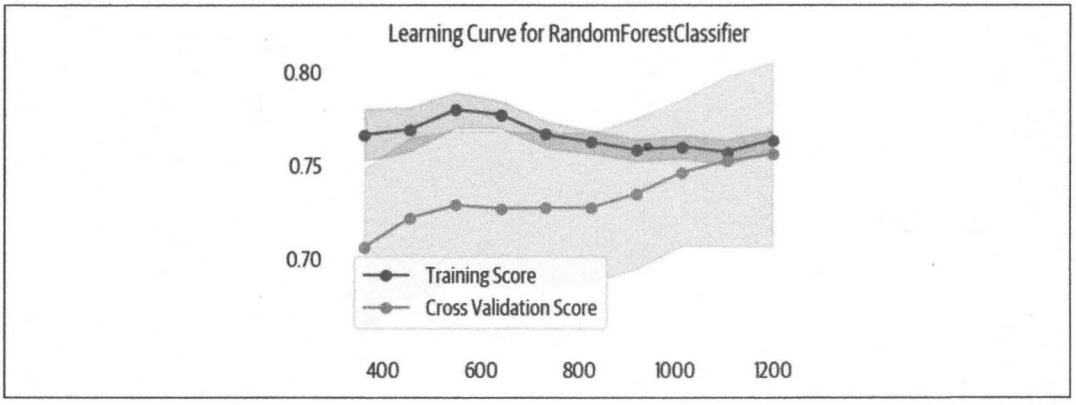



*Figure 3.6 : Courbe d'apprentissage qui montre qu'en ajoutant des données d'entraînement, le score de validation croisée du test semble augmenter.*

## Déploiement du modèle

Le module **pickle** de Python permet de rendre persistants les modèles et de les charger. Une fois que nous avons un modèle, nous pouvons appeler la méthode `.predict` pour générer un résultat de classification ou de régression :

```
>>> import pickle
>>> pic = pickle.dumps(rf5)
>>> rf6 = pickle.loads(pic)
>>> y_pred = rf6.predict(X_test)
>>> roc_auc_score(y_test, y_pred)
0.7747781065088757
```

Un outil souvent utilisé pour déployer un service Web de prédiction correspond à Flask (`https://palletsprojects.com/p/flask`). D'autres outils sont apparus aussi bien en tant que logiciels commerciaux qu'open source pour supporter un déploiement, parmi lesquels Clipper (`http://clipper.ai/`), Pipeline (`https://oreil.ly/UfHdP`) et Google's Cloud Machine Learning Engine (`https://oreil.ly/1qYkH`).

CHAPITRE 4

# Données manquantes

Nous devons absolument gérer les données manquantes. Nous avons eu un aperçu de la technique dans le précédent chapitre. Entrons plus dans les détails. En effet, lorsqu'il y a des manquants, la plupart des algorithmes ne fonctionnent pas ou mal. Une famille de librairies qui fait exception à cette contrainte est celle des librairies à amplification boosting apparues récemment : **`XGBoost`**, **`CatBoost`** et **`LightGBM`**.

Comme souvent dans le domaine du mécapprentissage (*machine learning*), il n'y a pas de réponse tranchée pour la gestion des données manquantes, car elles correspondent à des situations variées. Prenons le cas des données d'un recensement dans lequel la caractéristique *age* reste vide pour certaines personnes. Est-ce dû au fait que ces personnes ne veulent pas avouer leur âge ? Est-ce qu'elles ne savent pas leur âge exact ? Est-ce que l'enquêteur a oublié de poser la question ? Peut-on détecter des corrélations entre les âges manquants ? Est-ce que cette absence est liée à une autre caractéristique ou bien est-elle totalement aléatoire ?

La façon de gérer les manquants est elle aussi multiple :

- suppression de toutes les lignes contenant des manquants ;
- suppression des colonnes contenant des manquants ;
- imputation de valeurs de substitution pour les manquants ;
- ajout d'une colonne d'indication pour informer des données manquantes.

## Étude des manquants

Revenons au jeu de données du naufrage du *Titanic*. Le langage Python considère les deux valeurs booléennes True et False comme étant égales à 1 et à 0, respectivement. Nous nous servons de cette équivalence dans **`pandas`** pour produire les pourcentages de données manquantes :

```
>>> df.isnull().mean() * 100
pclass       0.000000
survived     0.000000
name         0.000000
sex          0.000000
age         20.091673
sibsp        0.000000
parch        0.000000
ticket       0.000000
fare         0.076394
cabin       77.463713
embarked     0.152788
boat        62.872422
body        90.756303
home.dest   43.086325
dtype: float64
```

Vous avez besoin d'installer la librairie **`missingno`** comme ceci :

```
pip install missingno
```

Pour détecter des motifs remarquables dans les manquants, nous utilisons la librairie nommée **`missingno`** (`https://oreil.ly/rgYJG`). Elle permet de bien visualiser des zones dans lesquelles se regroupent les manquants, ce qui permet d'en déduire que ces trous ne se présentent pas au hasard (Figure 4.1). La fonction `matrix` utilisée dans la figure présente sur le bord droit une ligne brisée de style sismographe qui, si elle contient des motifs, permet également d'en déduire que les données ne sont pas aléatoires. Il est possible qu'il faille limiter le nombre d'échantillons pour mieux voir les motifs éventuels :

```
>>> import missingno as msno
>>> ax = msno.matrix(orig_df.sample(500))
>>> ax.get_figure().savefig("images/mlpr_0401.png")
```

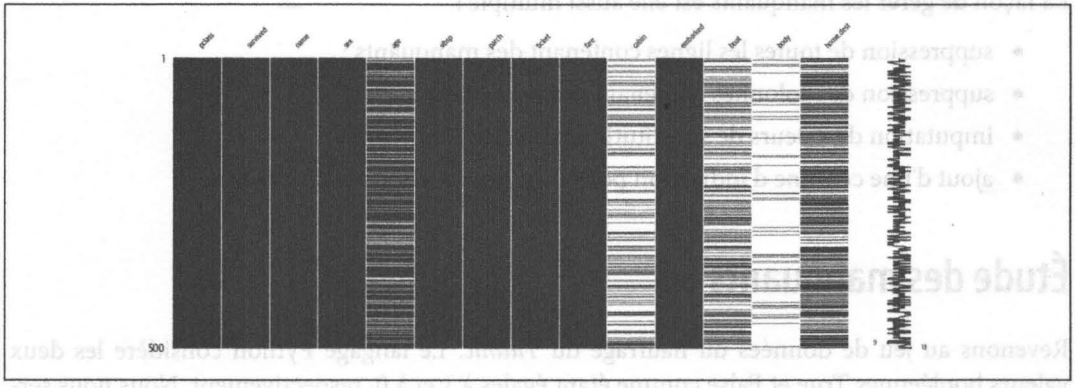



*Figure 4.1 : Visualisation des manquants. Aucun motif remarquable n'apparaît dans cet exemple.*

Avec **pandas**, nous pouvons demander la production d'un histogramme des manquants avec leur nombre (Figure 4.2) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> (1 - df.isnull().mean()).abs().plot.bar(ax=ax)
>>> fig.savefig("images/mlpr_0402.png", dpi=300)
```

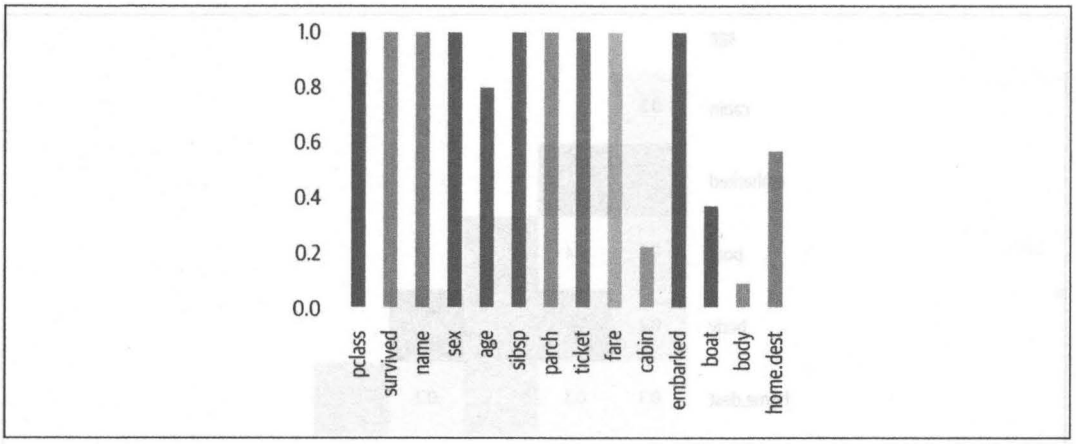



*Figure 4.2 : Pourcentage des données non manquantes avec pandas. Nous ignorons les colonnes boat et body qui sont divulgâcheuses. Nous remarquons que quelques valeurs pour l'âge manquent.*

Nous pouvons obtenir le même histogramme avec la librairie **`missingno`** (Figure 4.3) :

```
>>> ax = msno.bar(orig_df.sample(500))
>>> ax.get_figure().savefig("images/mlpr_0403.png")
```

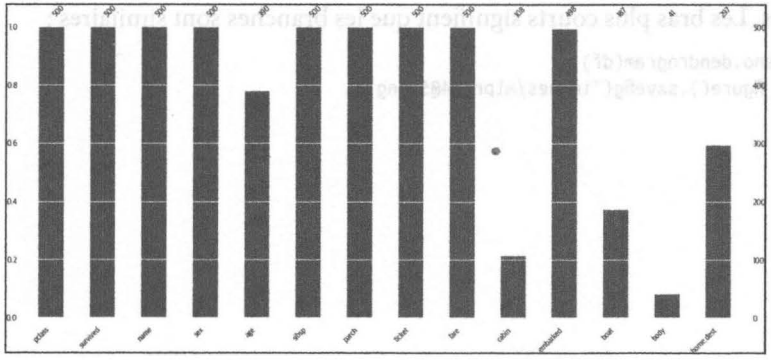


*Figure 4.3 : Pourcentage de données non manquantes avec missingno.*

Nous pouvons même créer une carte d'intensité de fréquentation (*heatmap*) pour détecter des corrélations dans les manquants (Figure 4.4). Dans l'exemple, il ne semble pas qu'il y ait corrélation entre les positions des manquants :

```
>>> ax = msno.heatmap(df, figsize=(6, 6))
>>> ax.get_figure().savefig("/tmp/mlpr_0404.png")
```

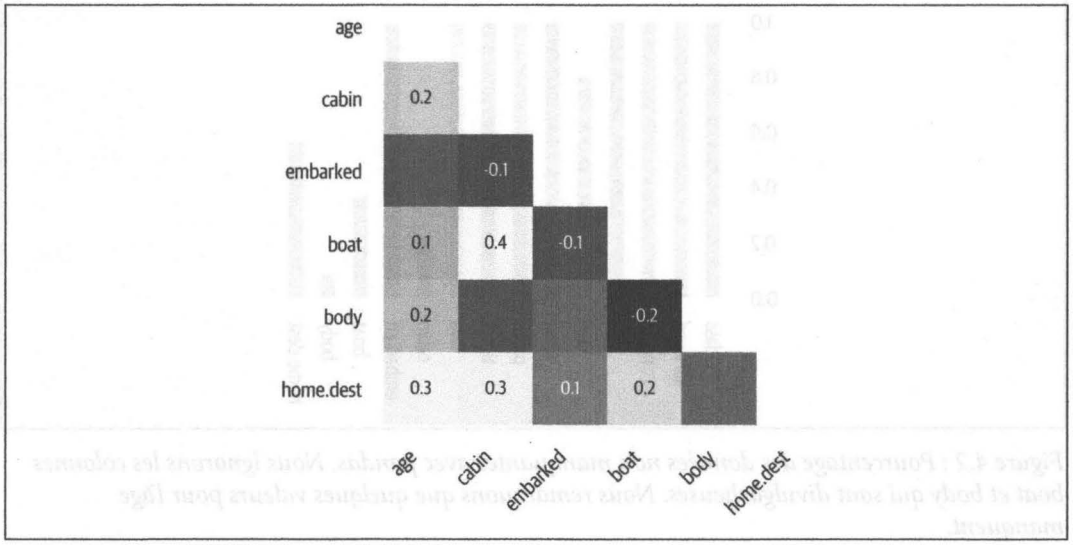



*Figure 4.4 : Corrélation des manquants avec missingno.*

Nous pouvons enfin créer un dendrogramme qui met en valeur les regroupements des manquants (Figure 4.5). Quand deux feuilles sont au même niveau, cela prédit la présence de pleins ou de vides d'une autre caractéristique. Les traits verticaux visualisent le degré de différence entre groupes. Les bras plus courts signifient que les branches sont similaires :

```
>>> ax = msno.dendrogram(df)
>>> ax.get_figure().savefig("images/mlpr_@4@5.png")
```

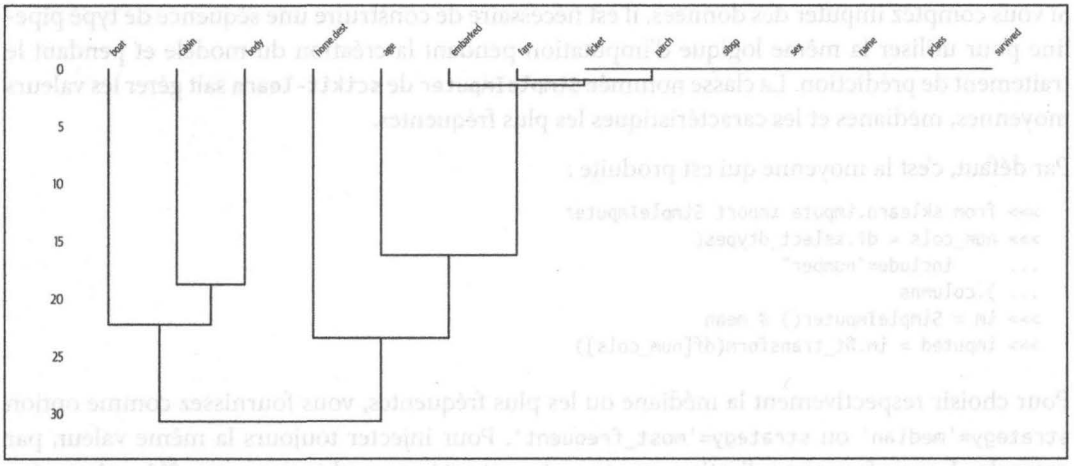



*Figure 4.5 : Dendrogramme de données manquantes avec missingno. Les colonnes sans manquants sont en haut à droite.*

# Abandon des données manquantes

Avec la librairie **pandas**, vous pouvez abandonner toutes les lignes contenant des données manquantes au moyen de la méthode `.dropna` :

```
>>> df1 = df.dropna()
```

Pour abandonner des colonnes, il faut d'abord les repérer et utiliser la méthode `.drop`. Il suffit de lui transmettre une liste de noms de colonnes ou un seul nom :

```
>>> df1 = df.drop(columns="cabin")
```

Nous obtenons le même résultat en utilisant la méthode `.dropna` mais en ajoutant l'option `axis=1` pour la faire travailler selon l'axe des colonnes :

```
>>> df1 = df.dropna(axis=1)
```

Réfléchissez bien avant de décider d'abandonner des données. Personnellement, je considère que c'est une action en dernier ressort.

# Imputation de données

Puisque vous disposez d'un outil pour prédire des données, vous pouvez vous en servir pour prédire les manquants. L'opération consistant à ajouter des valeurs en remplacement des manquants se nomme l'imputation.

Si vous comptez imputer des données, il est nécessaire de construire une séquence de type pipeline pour utiliser la même logique d'imputation pendant la création du modèle et pendant le traitement de prédiction. La classe nommée `SimpleImputer` de **`scikit-learn`** sait gérer les valeurs moyennes, médianes et les caractéristiques les plus fréquentes.

Par défaut, c'est la moyenne qui est produite :

```
>>> from sklearn.impute import SimpleImputer
>>> num_cols = df.select_dtypes(
...     include="number"
... ).columns
>>> im = SimpleImputer() # mean
>>> imputed = im.fit_transform(df[num_cols])
```

Pour choisir respectivement la médiane ou les plus fréquentes, vous fournissez comme option `strategy='median'` ou `strategy='most_frequent'`. Pour injecter toujours la même valeur, par exemple -1, vous fournissez l'option `strategy='constant'` en combinaison avec `fill_value=-1`.

La méthode `.fillna` de **`pandas`** déjà rencontrée permet aussi d'injecter des valeurs pour les manquants. Vous devez cependant veiller à ne pas provoquer un divulgâchage. Si vous injectez la moyenne, assurez-vous d'utiliser la même valeur pendant la création du modèle et pendant la prédiction.

Pour les valeurs de type numérique ou chaîne, vous utiliserez la plus grande fréquence et la constante. Pour la moyenne et la médiane, il est nécessaire d'utiliser des valeurs numériques.

La librairie nommée **`fancyimpute`** contient plusieurs algorithmes et obéit à la même interface que **`scikit-learn`**. Le souci est que la plupart de ces algorithmes sont transductifs : vous ne pouvez pas appeler directement la méthode `.transform` après avoir ajusté l'algorithme. En revanche, la méthode `IterativeImputer` est inductive et autorise les transformations après ajustement. Elle a d'ailleurs été migrée de **`fancyimpute`** vers **`scikit-learn`**.

## Ajout de colonnes indicatrices

L'absence de données peut fournir une information à un modèle. La librairie **`pandas`** permet d'ajouter une colonne pour indiquer qu'une valeur manquait :

```
>>> def add_indicator(col):
...     def wrapper(df):
...         return df[col].isna().astype(int)
...
...     return wrapper

>>> df1 = df.assign(
...     cabin_missing=add_indicator("cabin")
... )
```

CHAPITRE 5

# Nettoyage des données

Pour vous aider à bien nettoyer les données, nous pouvons utiliser des outils génériques, comme ceux de **pandas** ou des outils spécialisés comme **pyjanitor**.

## Renommage des colonnes

Il n'est pas inutile de rendre les noms des colonnes compatibles avec les conventions Python pour permettre un accès plus facile aux attributs avec **pandas**. Voyons la fonction `clean_names` de **pyjanitor** qui renvoie une structure `DataFrame` après avoir forcé les noms des colonnes et remplacé tous les espaces par des caractères de soulignement :

```
>>> import janitor as jn
>>> Xbad = pd.DataFrame(
...     {
...     "A": [1, None, 3],
...     " sales numbers ": [20.0, 30.0, None],
...     }
... )
>>> jn.clean_names(Xbad)

     a   _sales_numbers_
0  1.0              20.0
1  NaN              30.0
2  3.0               NaN
```

Vous devez installer, si ce n'est pas encore fait, la librairie **janitor** ainsi :

```
pip install janitor
```

Il est possible que vous rencontriez un problème au niveau de l'identifiant `ConfigParser` en cas de conflit de versions. Voyez dans ce cas le fichier *LISEZMOI.txt* fourni avec les fichiers d'exemple sur le site de l'éditeur.

Je conseille de procéder à la mise à jour des colonnes en utilisant une affectation par index, ou bien la méthode `.assign`, ou encore une affectation par `.loc` ou `.iloc`. Je déconseille d'utiliser les affectations d'attributs pour la mise à jour des colonnes dans **pandas**. Vous risquez en effet d'écraser une méthode existante portant le même nom que celui d'une colonne ; l'affectation par attributs n'a pas un fonctionnement certain.

La librairie **pyjanitor** est pratique, mais elle ne sait pas supprimer correctement les espaces en début et fin de noms de colonnes. Avec la librairie **pandas**, nous contrôlons mieux le traitement du nom des colonnes :

```
>>> def clean_col(name):
...     return (
...         name.strip().lower().replace(" ", "_")
...     )

>>> Xbad.rename(columns=clean_col)
     a   sales_numbers
0  1.0            20.0
1  NaN            30.0
2  3.0             NaN
```

## Remplacement des manquants

La fonction `coalesce` de **pyjanitor** traite une structure `DataFrame` avec une liste de colonnes, ce qui ressemble à la façon dont travaillent le tableur Excel et les bases SQL. La fonction renvoie la première valeur non nulle pour chaque ligne :

```
>>> jn.coalesce(
...     Xbad,
...     columns=["A", " sales numbers "],
...     new_column_name="val",
... )
     val
0    1.0
1   30.0
2    3.0
```

Pour injecter une valeur dans toutes les cellules vides, nous utilisons la méthode `.fillna` de `DataFrame` :

```
>>> Xbad.fillna(10)
      A  sales numbers
0   1.0           20.0
1  10.0           30.0
2   3.0           10.0
```

ou bien la fonction `fill_empty` de **pyjanitor** qui donne le même résultat :

```
>>> jn.fill_empty(
```

```
...     Xbad,
...     columns=["A", " sales numbers "],
...     value=10,
... )
      A  sales numbers
0   1.0           20.0
1  10.0           30.0
2   3.0           10.0
```

En général, nous utilisons les possibilités de remplacement des valeurs nulles par colonnes plus précises offertes par **pandas**, **scikit-learn** ou **fancyimpute**.

Un dernier contrôle qualité avant de créer le modèle consiste à utiliser **pandas** pour garantir que vous avez traité tous les manquants. L'instruction suivante renvoie une valeur booléenne dès qu'une cellule est un manquant dans la structure `DataFrame` :

```
>>> df.isna().any().any()
True
```

# CHAPITRE 6
# Exploration

On dit qu'il est plus simple de former un expert métier à la datalogie (science des données) qu'injecter des compétences métier dans un modèle de datalogie. Je ne dirais pas que je suis entièrement d'accord avec cette déclaration, mais il est vrai qu'il y a bien des nuances dans les données, et que seul un expert métier est capable de les distinguer. Sa connaissance des activités et des données correspondantes lui permet de produire des modèles de meilleure qualité avec un impact plus positif sur l'activité.

Avant de créer mon modèle, je réalise une analyse exploratoire des données pour mieux sentir ce qu'elles veulent dire. C'est l'occasion pour moi d'entrer en contact avec les différents départements qui sont les producteurs de ces données dans l'entreprise.

## Volumétrie des données

Nous reprenons le jeu de données Titanic. La propriété `.shape` de **pandas** permet de récupérer un tuple correspondant au nombre de lignes et de colonnes :

```
>>> X.shape
(1309, 13)
```

Nous constatons que le jeu comporte 1 309 lignes de 13 colonnes.

## Statistiques globales

La librairie **pandas** nous permet d'obtenir des statistiques générales. La méthode `.describe` indique le nombre de valeurs différentes de NaN. Voyons les résultats pour les première et dernière colonnes :

```
>>> X.describe().iloc[:, [0, -1]]
            pclass    embarked_S
count  1309.000000  1309.000000
mean     -0.012831     0.698243
std       0.995822     0.459196
min      -1.551881     0.000000
25%      -0.363317     0.000000
50%       0.825248     1.000000
75%       0.825248     1.000000
max       0.825248     1.000000
```

La ligne qui commence par `count` nous confirme que ces deux colonnes n'ont aucun manquant. Nous disposons également de la moyenne `mean`, de l'écart-type `std`, des valeurs minimale et maximale et des trois quartiles intermédiaires.

La structure `DataFrame` possède un attribut nommé `iloc` qui permet de travailler avec les index. Nous pouvons ainsi sélectionner des lignes et des colonnes en fonction de leur index. Nous fournissons d'abord la position de ligne sous forme d'un scalaire, d'une liste ou d'une tranche, une virgule puis la position de colonne, également sous forme d'un scalaire, d'une liste ou d'une tranche.

Voici comment récupérer la deuxième et la cinquième ligne avec les trois dernières colonnes :

```
>>> X.iloc[[1, 4], -3:]
     sex_male  embarked_Q  embarked_S
677       1.0           0           1
864       0.0           0           1
```

Vous disposez également de l'attribut `.loc` pour sélectionner des lignes et des colonnes en fonction de leur nom. Voici comment récupérer ainsi le même extrait de la structure :

```
>>> X.loc[[677, 864], "sex_male":]
sex_male embarked_Q embarked_S
     sex_male  embarked_Q  embarked_S
677       1.0           0           1
864       0.0           0           1
```

# Histogrammes

Les histogrammes sont d'excellents outils pour les données numériques. Ils permettent de voir le nombre de modes ainsi que la distribution (Figure 6.1). Nous pouvons nous servir à cet effet de la méthode `.plot` de **pandas** :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> X.fare.plot(kind="hist", ax=ax)
>>> fig.savefig("images/mlpr_0601.png", dpi=300)
```

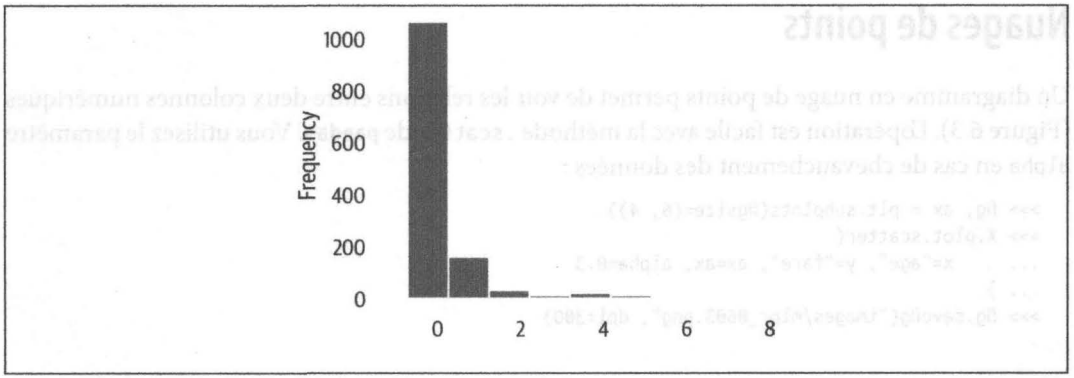



*Figure 6.1 : Histogramme produit avec pandas.*

Note qu'il existe une autre librairie, **seaborn**, pour produire un histogramme de valeurs continues par rapport à une cible (Figure 6.2) :

```
fig, ax = plt.subplots(figsize=(12, 8))
mask = y_train == 1
ax = sns.distplot(X_train[mask].fare, label='survived')
ax = sns.distplot(X_train[~mask].fare,label='died')
ax.set_xlim(-1.5, 1.5)
ax.legend()
fig.savefig('images/mlpr_0602.png', dpi=300, bbox_inches='tight')
```

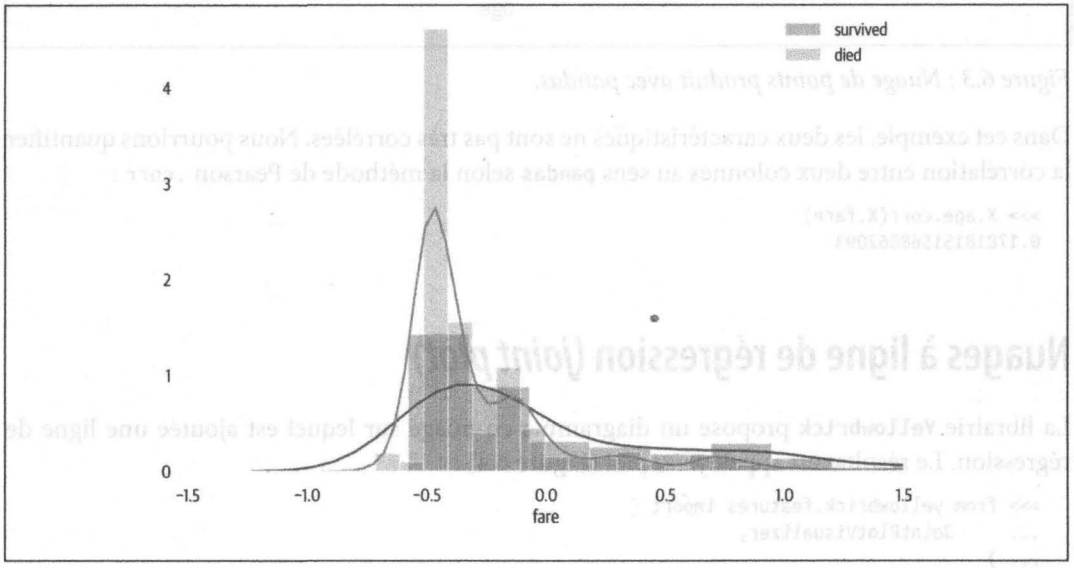



*Figure 6.2 : Histogramme produit avec seaborn.*

## Nuages de points

Un diagramme en nuage de points permet de voir les relations entre deux colonnes numériques (Figure 6.3). L'opération est facile avec la méthode `.scatter` de **pandas**. Vous utilisez le paramètre `alpha` en cas de chevauchement des données :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> X.plot.scatter(
...     x="age", y="fare", ax=ax, alpha=0.3
... )
>>> fig.savefig("images/mlpr_0603.png", dpi=300)
```

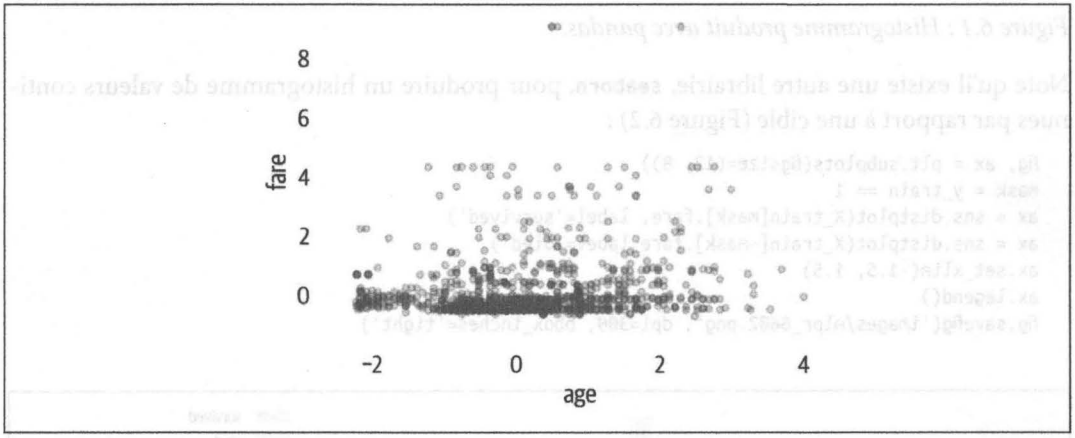



*Figure 6.3 : Nuage de points produit avec pandas.*

Dans cet exemple, les deux caractéristiques ne sont pas très corrélées. Nous pourrions quantifier la corrélation entre deux colonnes au sens **pandas** selon la méthode de Pearson `.corr` :

```
>>> X.age.corr(X.fare)
0.17818151568062093
```

## Nuages à ligne de régression (*joint plot*)

La librairie **`Yellowbrick`** propose un diagramme en nuage sur lequel est ajoutée une ligne de régression. Le résultat est appelé *joint plot* (Figure 6.4) :

```
>>> from yellowbrick.features import (
...     JointPlotVisualizer,
... )
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> jpv = JointPlotVisualizer(
...     feature="age", target="fare"
```

```
... )
>>> jpv.fit(X["age"], X["fare"])
>>> jpv.poof()
>>> fig.savefig("images/mlpr_0604.png", dpi=300)
```

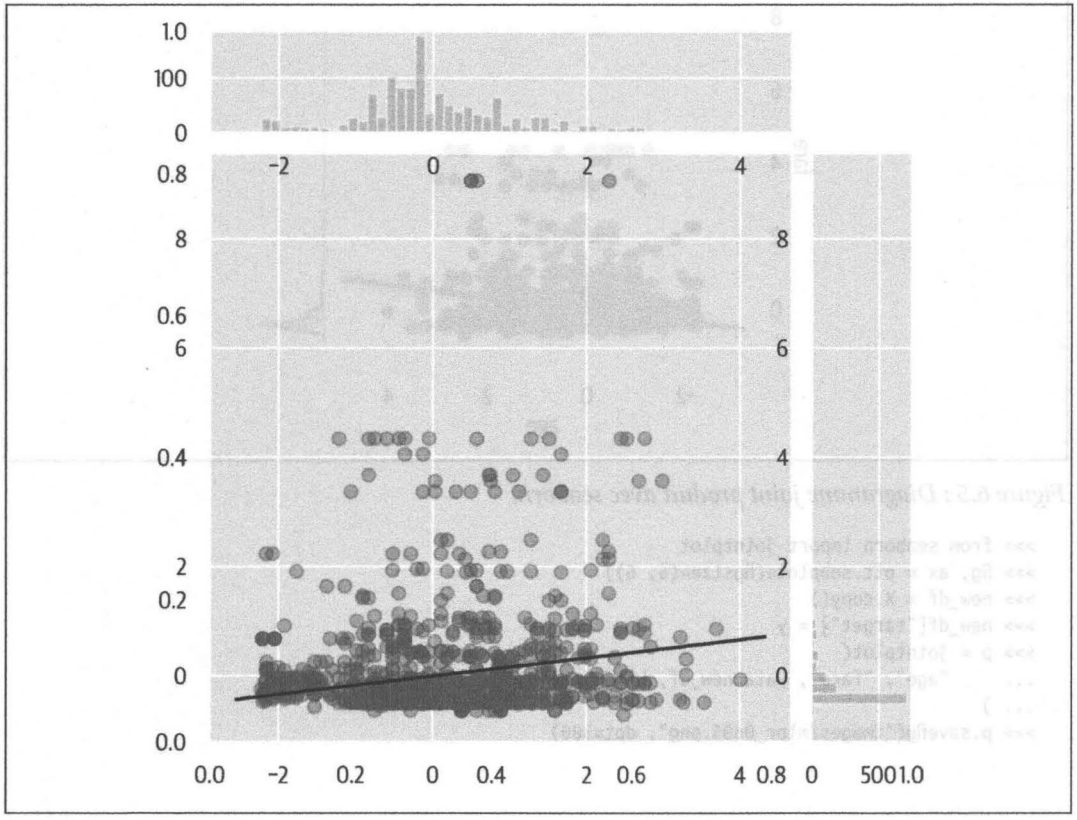



*Figure 6. 4 : Diagramme en nuage de points à régression de Yellowbrick.*

Dans l'utilisation de la méthode `.fit`, *X* est également une colonne, alors qu'en général, *X* est un objet de structure `DataFrame` et non une série.

Nous pourrions également utiliser la librairie **seaborn** (`https://seaborn.pydata.org`) pour créer le même genre de diagramme (Figure 6.5) :

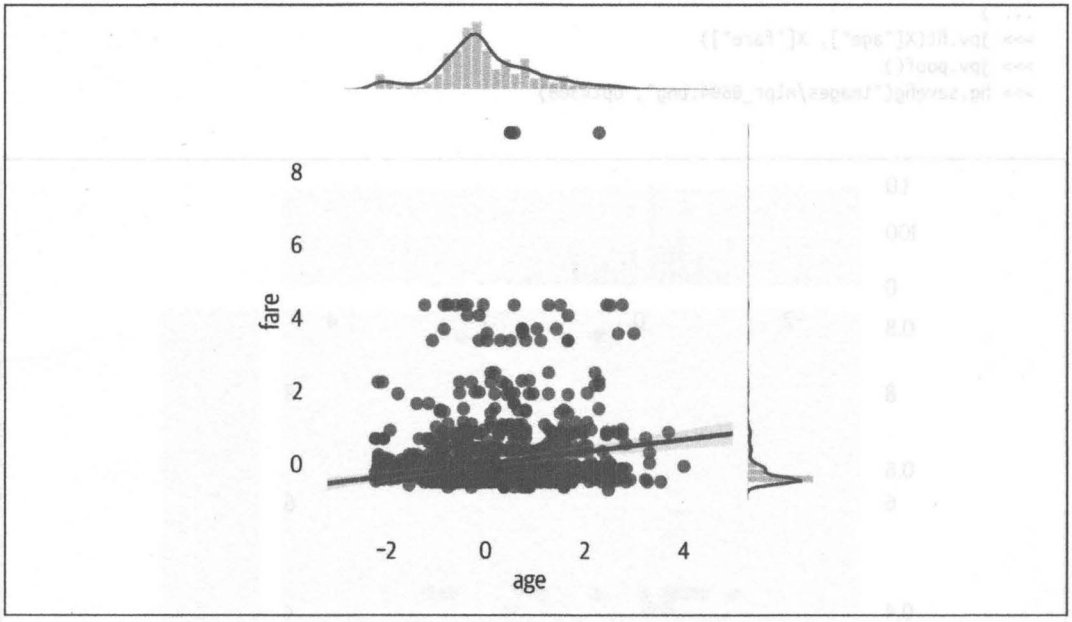



*Figure 6.5 : Diagramme joint produit avec seaborn.*

```
>>> from seaborn import jointplot
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> p = jointplot(
...     "age", "fare", data=new_df, kind="reg"
... )
>>> p.savefig("images/mlpr_0605.png", dpi=300)
```

# Grille de paires

La librairie **seaborn** permet de créer une grille de paires (Figure 6.6). Il s'agit d'une matrice de colonnes avec des estimations de densités de noyau. Pour définir une couleur en fonction d'une colonne d'une structure `DataFrame`, vous vous servez du paramètre `hue`. Pour connaître l'effet des caractéristiques sur la cible, il suffit de faire colorier en fonction de cette cible :

```
>>> from seaborn import pairplot
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> vars = ["pclass", "age", "fare"]
>>> p = pairplot(
...     new_df, vars=vars, hue="target", kind="reg"
```

```
... )
>>> p.savefig("images/mlpr_0606.png", dpi=300)
```

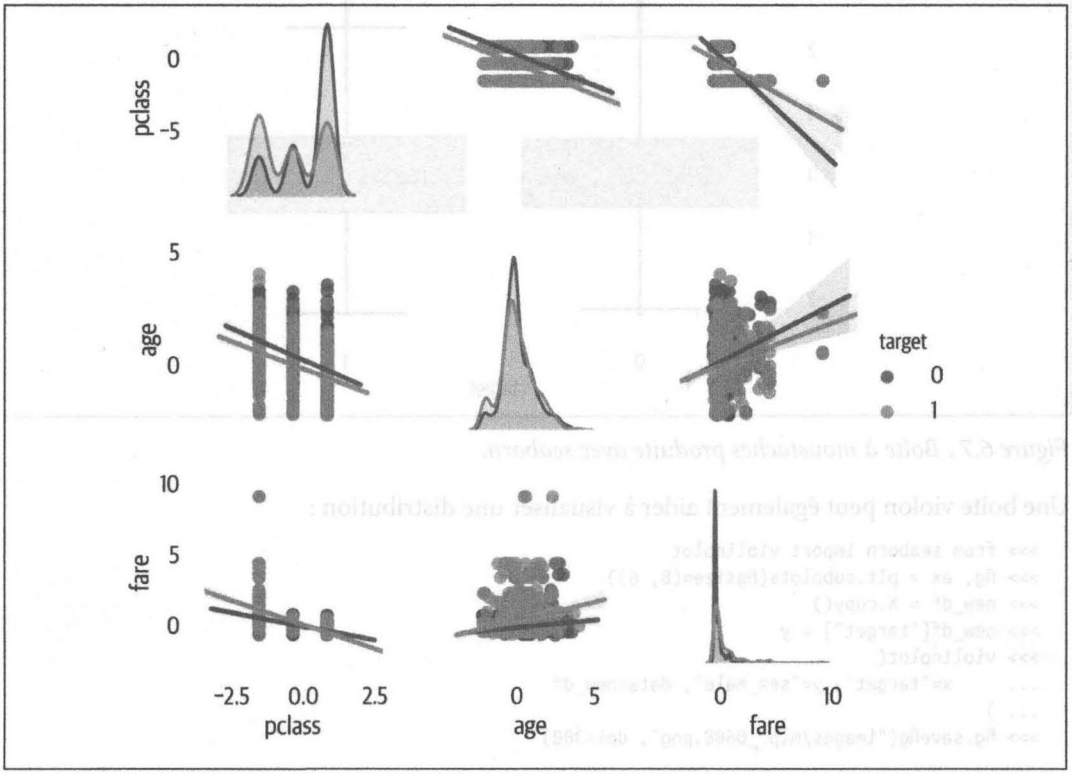



*Figure 6.6 : Grille de paires produite avec seaborn.*

## Boîtes à moustaches et boîtes violon

La librairie **seaborn** propose plusieurs types de diagrammes pour visualiser les distributions. Voici par exemple une boîte à moustaches et une boîte violon dans les Figures 6.7 et 6.8. Ces genres de diagrammes permettent de visualiser une caractéristique par rapport à une cible :

```
>>> from seaborn import box plot
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> boxplot(x="target", y="age", data=new_df)
>>> fig.savefig("images/mlpr_0607.png", dpi=300)
```

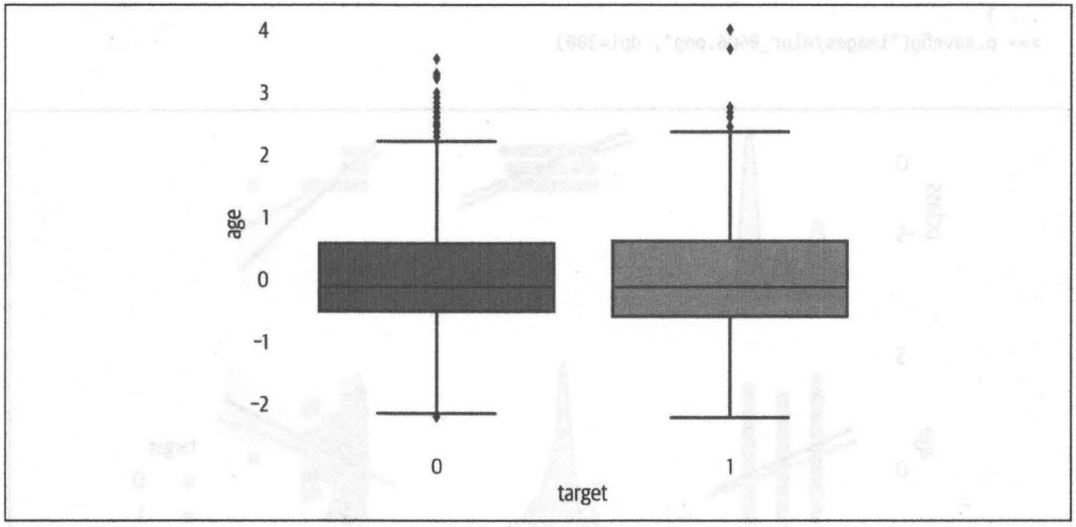



*Figure 6.7 : Boîte à moustaches produite avec seaborn.*

Une boîte violon peut également aider à visualiser une distribution :

```
>>> from seaborn import violinplot
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> violinplot(
...     x="target", y="sex_male", data=new_df
... )
>>> fig.savefig("images/mlpr_0608.png", dpi=300)
```

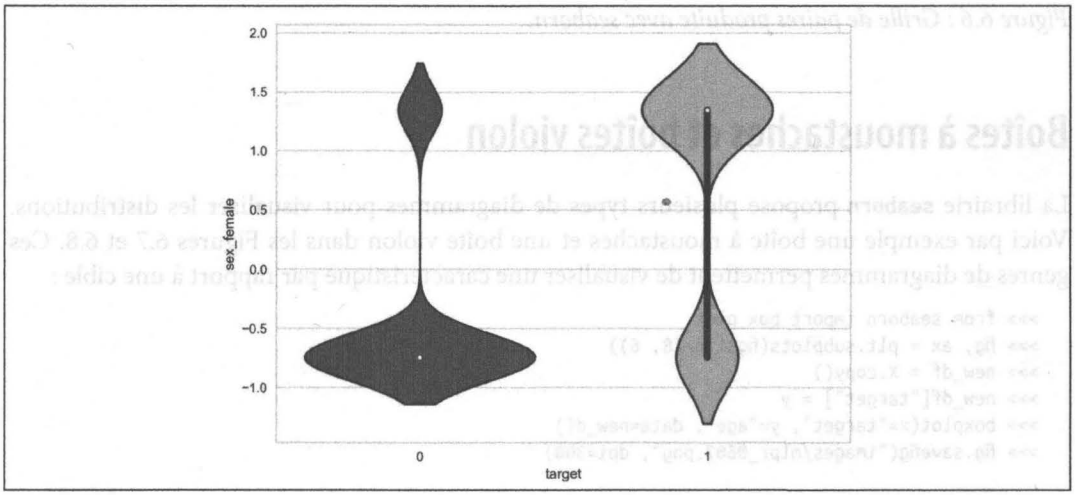



*Figure 6.8 : Boîte violon produite avec seaborn.*

## Comparaison de deux valeurs ordinales

Le code utilisant **pandas** ci-dessous compare deux catégories ordinales. Je distribue les valeurs de *age* en dix déciles (opération de *binning*) et les valeurs de `pclass` en trois groupes. Le diagramme est normalisé pour remplir tout l'espace dans le sens vertical, ce qui permet facilement de voir que la plupart des tickets du cinquième décile (valeur 4), étaient des tickets de troisième classe (Figure 6.9) :

```
>>> fig, ax = plt.subplots(figsize=(8, 6))
(
...     X.assign(
...         age_bin=pd.qcut(
...             X.age, q=10, labels=False
...         ),
...         class_bin=pd.cut(
...             X.pclass, bins=3, labels=False
...         ),
...     )
...     .groupby(["age_bin", "class_bin"])
...     .size()
...     .unstack()
...     .pipe(lambda df: df.div(df.sum(1), axis=0))
...     .plot.bar(
...         stacked=True,
...         width=1,
...         ax=ax,
...         cmap="viridis",
...     )
...     .legend(bbox_to_anchor=(1, 1))
...)
...#fig.savefig(
...#     "image/mlpr_0609.png",
...#     dpi=300,
...#     bbox_inches="tight",
...# )
```

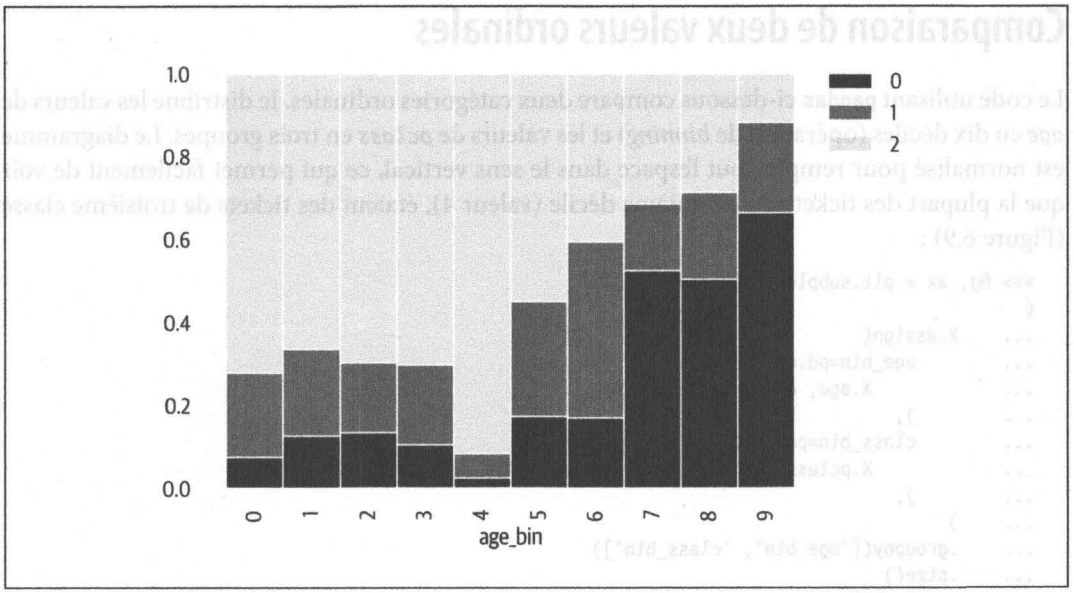



*Figure 6.9 : Comparaison de valeurs ordinales.*

Les trois lignes suivantes :

```
.groupby(["age_bin", "class_bin"])
.size()
.unstack()
```

peuvent être remplacées par :

```
.pipe(lambda df: pd.crosstab(
      df.age_bin, df.class_bin)
)
```

Vous pouvez souvent arriver au même résultat de plusieurs manières dans **pandas**. Vous disposez de fonctions de support qui donnent accès à d'autres possibilités, et notamment `pd.crosstab`.

# Corrélations

La librairie **`Yellowbrick`** permet de créer des comparaisons par paires de caractéristiques (Figure 6.10). Notre exemple montre une corrélation de Pearson. Sachez que le paramètre `algorithm` peut accepter les deux valeurs `'spearman'` et `'covariance'` :

```
>>> from yellowbrick.features import Rank2D
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> pcv = Rank2D(
...     features=X.columns, algorithm="pearson"
... )
```

```
>>> pcv.fit(X, y)
>>> pcv.transform(X)
>>> pcv.poof()
>>> fig.savefig(
...     "images/mlpr_0610.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

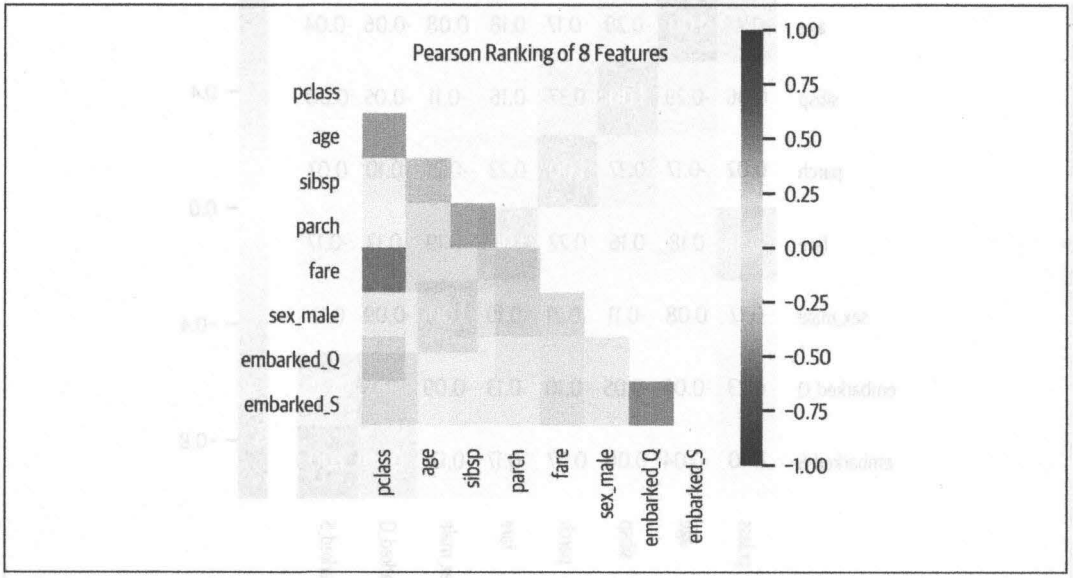



*Figure 6.10 : Corrélation de covariance créée avec Yellowbrick.*

La librairie **seaborn** permet de produire une carte d'intensité *heatmap* (Figure 6.11). Pour les données d'entrée, il faut fournir une structure `DataFrame` de corrélation. Le souci est que la gamme de couleurs ne s'étend pas de -1 à +1, sauf si c'est aussi le cas des valeurs de la matrice. Mais il est également possible d'ajouter des paramètres de bornage `vmin` et `vmax` :

```
>>> from seaborn import heatmap
>>> fig, ax = plt.subplots(figsize=(8, 8))
>>> ax = heatmap(
...     X.corr(),
...     fmt=".2f",
...     annot=True,
...     ax=ax,
...     cmap="RdBu_r",
...     vmin=-1,
...     vmax=1,
... )
```

```
>>> fig.savefig(
...     "images/mlpr_0611.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

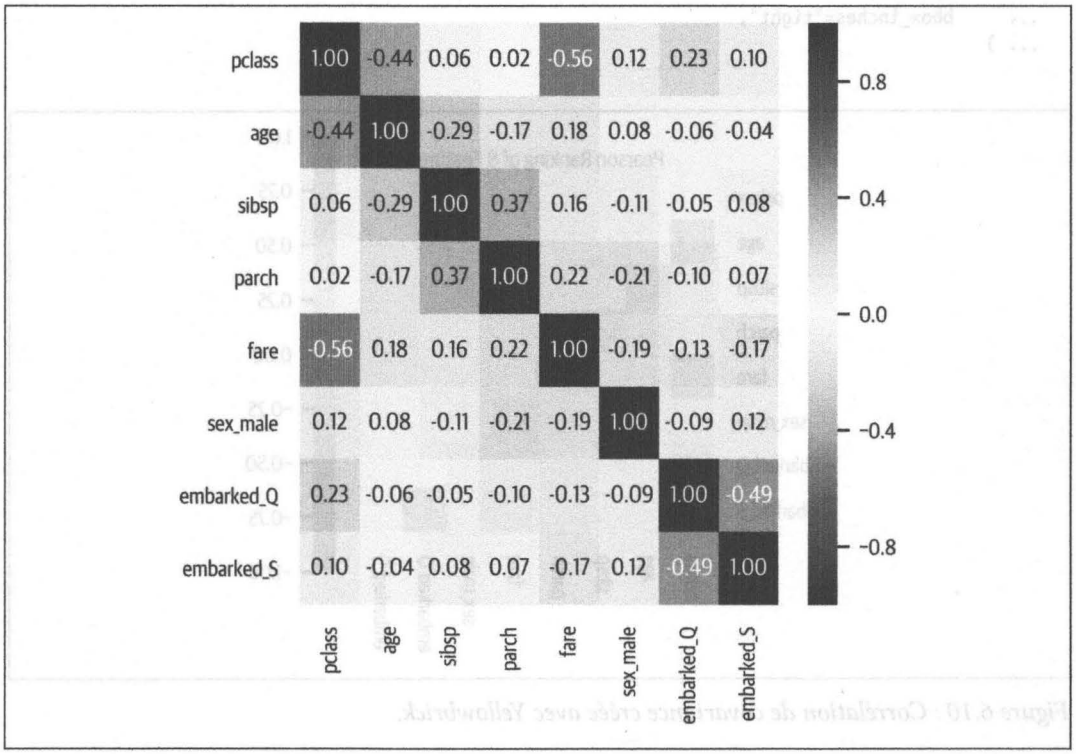



*Figure 6.11 : Carte d'intensité produite avec seaborn.*

La librairie **pandas** permet aussi d'obtenir une corrélation entre plusieurs colonnes d'une structure `DataFrame`. Limitons-nous aux deux premières colonnes. La méthode appliquée par défaut est Pearson, mais vous pouvez également choisir pour le paramètre `method` la valeur `'kendall'` ou `'spearman'` ou bien fournir une fonction spécifique qui sera appelée pour renvoyer une valeur flottante à partir de deux colonnes :

```
>>> X.corr().iloc[:, :2]
              pclass       age
pclass      1.000000 -0.439704
age        -0.439704  1.000000
sibsp       0.060832 -0.292056
parch       0.018322 -0.176447
fare       -0.558827  0.177200
sex_male    0.124617  0.065004
embarked_Q  0.230491 -0.053904
embarked_S  0.096335 -0.045361
```

Notez que les colonnes qui sont fortement corrélées n'apportent pas de valeur et risquent même de perturber la pondération des caractéristiques et l'interprétation des coefficients de régression. L'exemple qui suit cherche à trouver ces colonnes corrélées. Dans l'exemple, aucune des colonnes n'est trop corrélée, mais n'oubliez pas que nous avons supprimé la colonne *sex_male*.

S'il y avait des colonnes corrélées, nous aurions pu choisir d'éliminer soit celles de `level_0`, soit celles de `level_1` dans les données des caractéristiques :

```
>>> def correlated_columns(df, threshold=0.95):
...     return (
...         df.corr()
...         .pipe(
...             lambda df1: pd.DataFrame(
...                 np.tril(df1, k=-1),
...                 columns=df.columns,
...                 index=df.columns,
...             )
...         )
...         .stack()
...         .rename("pearson")
...         .pipe(
...             lambda s: s[
...                 s.abs() > threshold
...             ].reset_index()
...         )
...         .query("level_0 not in level_1")
...     )

>>> correlated_columns(X)
Empty DataFrame
Columns: [level_0, level_1, pearson]
Index: []
```

En utilisant un plus grand nombre de colonnes, nous constatons que plusieurs sont assez corrélées :

```
>>> c_df = correlated_columns(agg_df)
>>> c_df.style.format({"pearson": "{:.2f}"})
```

| | level_0 | level_1 | pearson |
|---|---|---|---|
| 3 | pclass_mean | pclass | 1.00 |
| 4 | pclass_mean | pclass_min | 1.00 |
| 5 | pclass_mean | pclass_max | 1.00 |
| 6 | sibsp_mean | sibsp_max | 0.97 |
| 7 | parch_mean | parch_min | 0.95 |
| 8 | parch_mean | parch_max | 0.96 |
| 9 | fare_mean | fare | 0.95 |
| 10 | fare_mean | fare_max | 0.98 |
| 17 | body_sum | body | 1.00 |
| 18 | body_sum | body_min | 1.00 |
| 19 | body_sum | body_max | 1.00 |
| 20 | body_sum | body_mean | 1.00 |

## RadViz

Un diagramme de visualisation radiale ou RadViz présente les caractéristiques sur la circonférence et les valeurs dans un nuage de points (Figure 6.12). Les valeurs sont normalisées. Vous pouvez vous les imaginer comme des ressorts entre les valeurs et les caractéristiques. Il s'agit d'une technique permettant de voir le degré de séparation entre cibles.

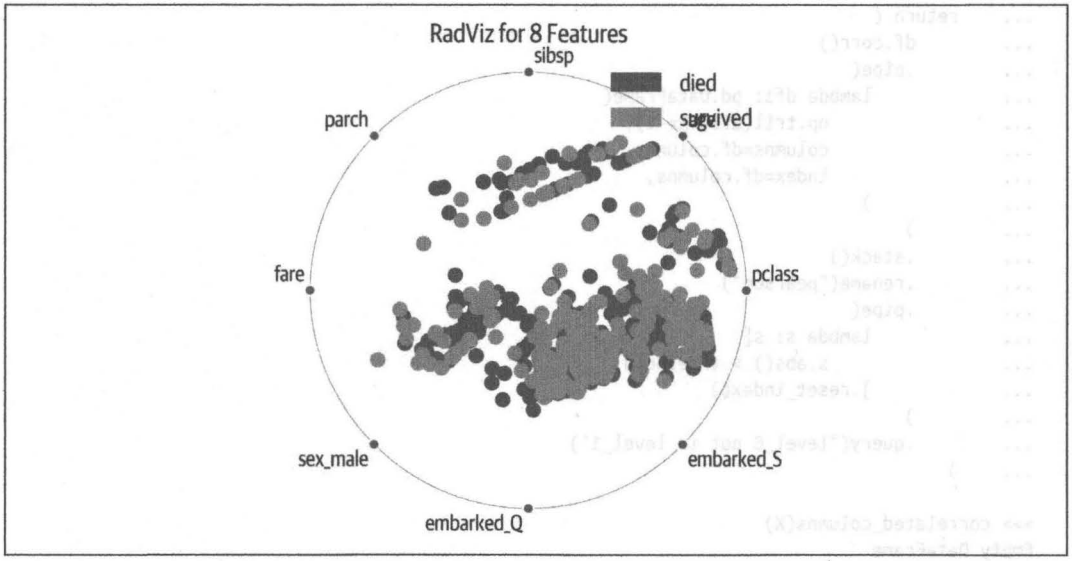



*Figure 6.12 : Diagramme circulaire RadViz avec Yellowbrick.*

Voici comment produire un RadViz avec **`Yellowbrick`** :

```
>>> from yellowbrick.features import RadViz
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> rv = RadViz(
...     classes=["died", "survived"],
...     features=X.columns,
... )
>>> rv.fit(X, y)
>>> _ = rv.transform(X)
>>> rv.poof()
>>> fig.savefig("images/mlpr_0612.png", dpi=300)
```

La librairie **pandas** peut aussi générer un tel diagramme (Figure 6.13) :

```
>>> from pandas.plotting import radviz
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> radviz(
```

```
...     new_df, "target", ax=ax, colormap="PiYG"
... )
>>> fig.savefig("images/mlpr_0613.png", dpi=300)
```

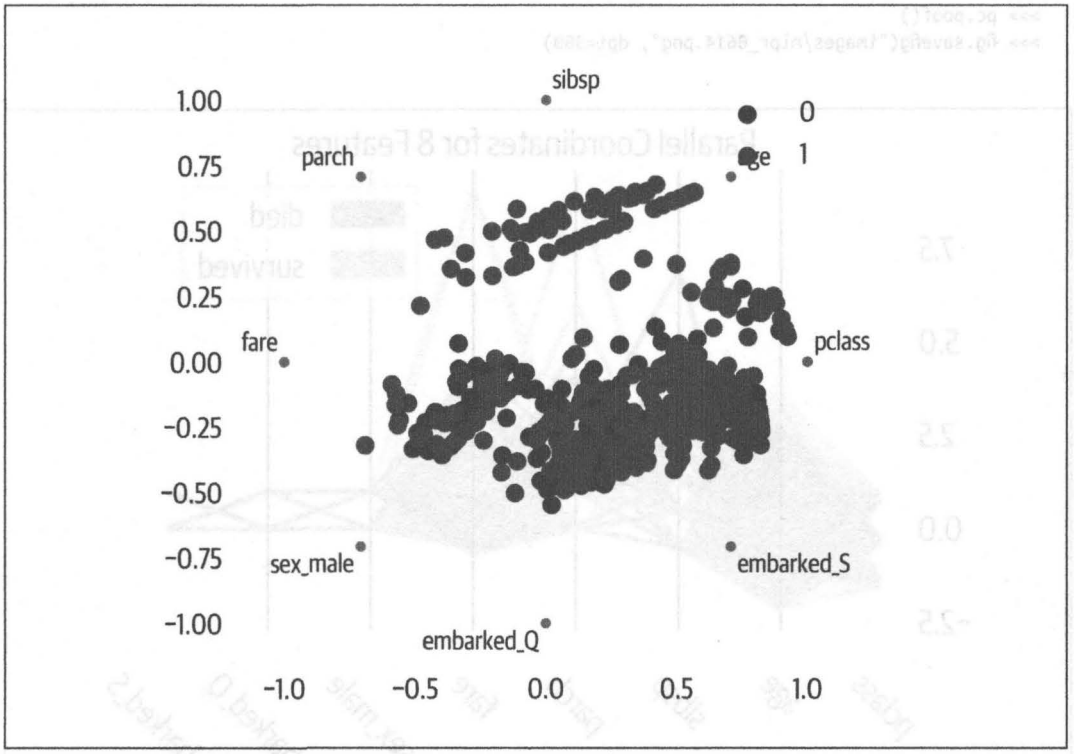



*Figure 6.13 : Diagramme circulaire RadViz avec pandas.*

# Coordonnées parallèles

Si vous traitez des données multivariantes, vous pouvez faire tracer un diagramme en coordonnées parallèles pour voir les regroupements (Figures 6.14 et 6.15). Voici comment l'obtenir avec **`Yellowbrick`** :

```
>>> from yellowbrick.features import (
...     ParallelCoordinates,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pc = ParallelCoordinates(
...     classes=["died", "survived"],
...     features=X.columns,
... )
```

```
>>> pc.fit(X, y)
>>> pc.transform(X)
>>> ax.set_xticklabels(
...     ax.get_xticklabels(), rotation=45
... )
>>> pc.poof()
>>> fig.savefig("images/mlpr_0614.png", dpi=300)
```

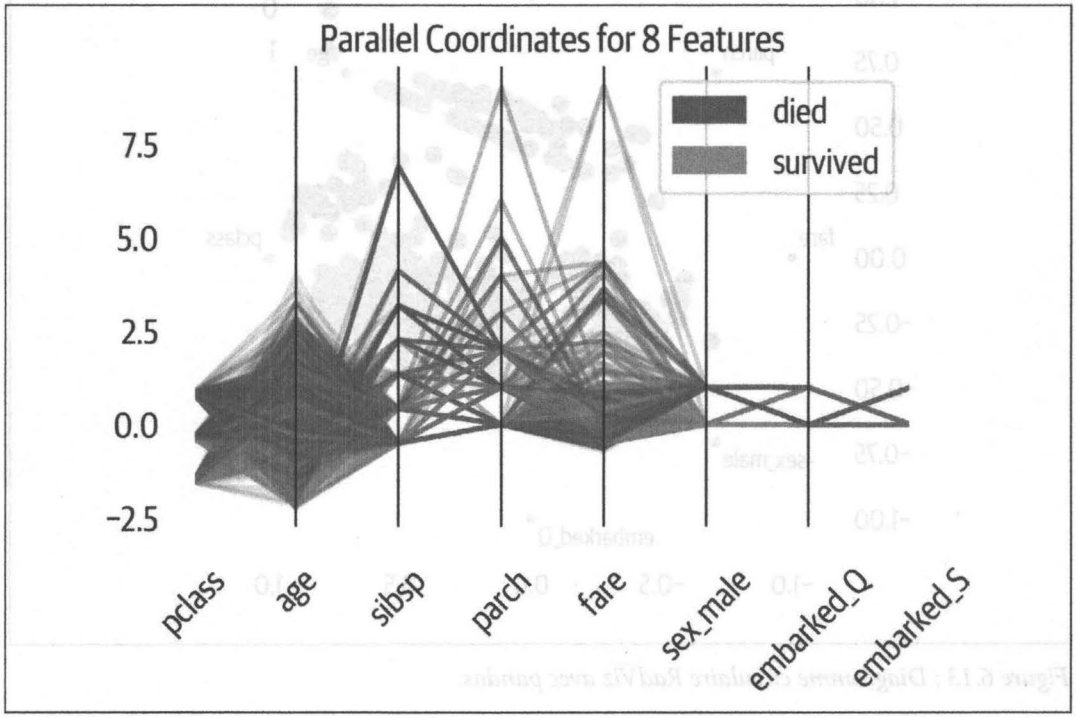



*Figure 6.14 : Diagramme à coordonnées parallèles avec Yellowbrick.*

Et voici le même diagramme produit avec **pandas** :

```
>>> from pandas.plotting import (
...     parallel_coordinates,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> parallel_coordinates(
...     new_df,
...     "target",
...     ax=ax,
...     colormap="viridis",
...     alpha=0.5,
... )
```

```
>>> ax.set_xticklabels(
...     ax.get_xticklabels(), rotation=45
... )
>>> fig.savefig(
...     "images/mlpr_0615.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

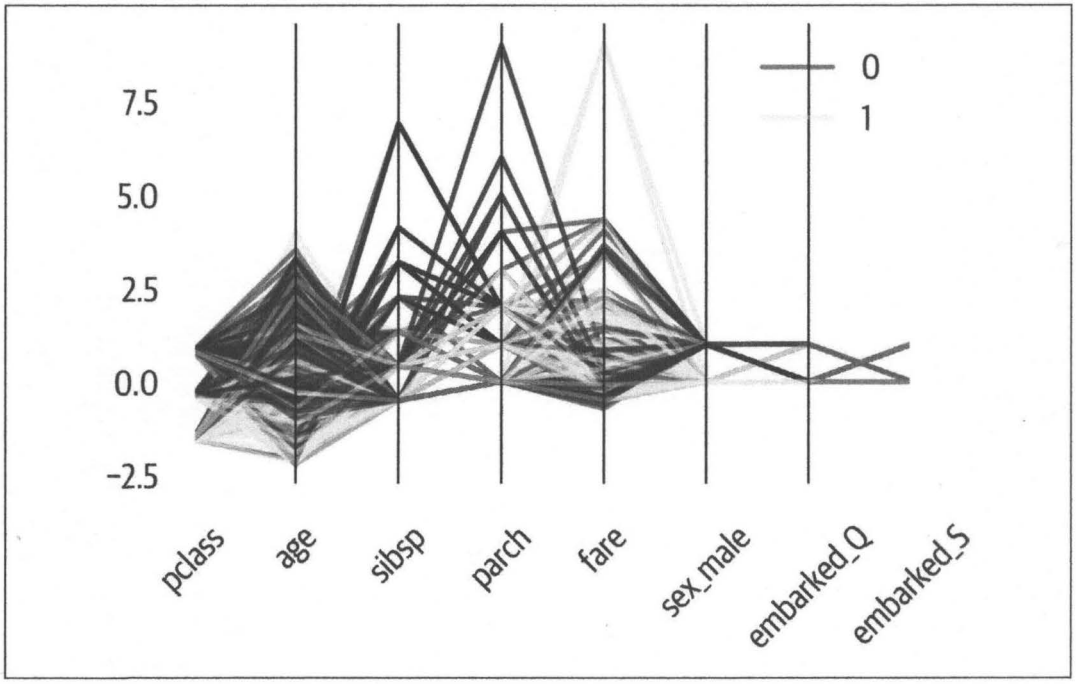



*Figure 6.15 : Diagramme à coordonnées parallèles avec pandas.*

CHAPITRE 7

# Prétraitement des données

Nous allons dans ce chapitre découvrir quelques techniques habituelles de prétraitement en utilisant le jeu de données suivant :

```
>>> X2 = pd.DataFrame(
...     {
...         "a": range(5),
...         "b": [-100, -50, 0, 200, 1000],
...     }
... )

>>> X2
   a     b
0  0  -100
1  1   -50
2  2     0
3  3   200
4  4  1000
```

## Standardisation

La standardisation des données permet à certains algorithmes de mieux travailler, notamment SVM. Il s'agit pour chacune des colonnes de contenir une moyenne égale à zéro, avec un écart-type égal à 1. La librairie **sklearn** propose la méthode `.fit_transform` qui combine `.fit` et `.transform` :

```
>>> from sklearn import preprocessing
>>> std = preprocessing.StandardScaler()
>>> std.fit_transform(X2)
array([[-1.41421356, -0.75995002],
       [-0.70710678, -0.63737744],
       [ 0.        , -0.51480485],
       [ 0.70710678, -0.02451452],
       [ 1.41421356,  1.93664683]])
```

Une fois l'ajustement fait, nous pouvons inspecter plusieurs attributs :

```
>>> std.scale_
array([ 1.41421356, 407.92156109])
>>> std.mean_
array([ 2., 210.])
>>> std.var_
array([2.000e+00, 1.664e+05])
```

Voici la même technique avec la librairie **pandas**. N'oubliez pas qu'il faut mémoriser la moyenne et l'écart-type utilisés au départ si vous voulez faire du prétraitement avec. En effet, tous les échantillons ultérieurs que vous voudrez utiliser pour vos prédictions devront avoir été standardisés avec les mêmes valeurs :

```
>>> X_std = (X2 - X2.mean()) / X2.std()
>>> X_std
          a         b
0 -1.264911 -0.679720
1 -0.632456 -0.570088
2  0.000000 -0.460455
3  0.632456 -0.021926
4  1.264911  1.732190
>>> X_std.mean()
a   4.440892e-17
b   0.000000e+00
dtype: float64
>>> X_std.std()
a   1.0
b   1.0
dtype: float64
```

Pour information, le même résultat peut être obtenu avec la librairie **fastai** :

```
>>> X3 = X2.copy()
>>> from fastai.structured import scale_vars
>>> scale_vars(X3, mapper=None)
>>> X3.std()
a    1.118034
b    1.118034
dtype: float64

>>> X3.mean()
a    0.000000e+00
b    4.440892e-17
dtype: float64
```

Il est fort possible que la librairie **fastai** soit difficile à installer. L'exemple précédent n'a pas besoin d'être exécuté pour poursuivre la pratique des exemples suivants du chapitre.

## Confinement (*scale to range*)

L'opération de confinement consiste à convertir les données pour que toutes les valeurs se situent entre 0 et 1, bornes comprises. Ce bornage peut s'avérer utile, mais vous devrez l'utiliser avec précaution s'il y a des valeurs aberrantes (très éloignées des autres) :

```
>>> from sklearn import preprocessing
>>> mms = preprocessing.MinMaxScaler()
>>> mms.fit(X2)
>>> mms.transform(X2)
array([[0.     , 0. ],
       [0.25   , 0.04545],
       [0.5    , 0.09091],
       [0.75   , 0.27273],
       [1.     , 1.     ]])
```

Voici la version utilisant **pandas** :

```
>>> (X2 - X2.min()) / (X2.max() - X2.min())
      a         b
0  0.00  0.000000
1  0.25  0.045455
2  0.50  0.090909
3  0.75  0.272727
4  1.00  1.000000
```

## Variables factices (*dummy*)

Nous pouvons utiliser la librairie **pandas** pour obtenir des variables factices à partir des données catégorielles, ce qui correspond à l'encodage 1 parmi n *1-hot* ou encore encodage d'indicateurs. Ce genre de variables factices est particulièrement intéressant pour des données nominales non triées. La fonction de **pandas** nommée `get_dummies` permet de créer plusieurs colonnes à partir d'une colonne catégorielle, chacune contenant soit 1, soit 0, en fonction de la valeur contenue dans la colonne de départ :

```
>>> X_cat = pd.DataFrame(
...     {
...         "name": ["George", "Paul"],
...         "inst": ["Bass", "Guitar"],
...     }
... )
>>> X_cat
     name    inst
0  George    Bass
1    Paul  Guitar
```

Vous pouvez ajouter l'option `drop_first` pour éliminer une colonne (une des colonnes factices est par exemple une combinaison linéaire des autres) :

```
>>> pd.get_dummies(X_cat, drop_first=True)
      name_Paul   inst_Guitar
0             0             0
1             1             1
```

Vous pouvez également subdiviser des colonnes au moyen de la librairie **`pyjanitor`** grâce à sa fonction `expand_column` :

```
>>> X_cat2 = pd.DataFrame(
...     {
...         "A": [1, None, 3],
...         "names": [
...             "Fred,George",
...             "George",
...             "John,Paul",
...         ],
...     }
... )
>>> jn.expand_column(X_cat2, "names", sep=",")
    A         names  Fred  George  John  Paul
0  1.0  Fred,George     1       1     0     0
1  NaN       George     0       1     0     0
2  3.0    John,Paul     0       0     1     1
```

Avec des données nominales ayant beaucoup de valeurs différentes (une forte cardinalité), nous pouvons adopter un encodage de labels, ce que nous voyons maintenant.

# Encodage de labels

Au lieu d'encoder des variables factices, vous pouvez encoder des labels, ce qui consiste à affecter une valeur numérique à chaque valeur catégorielle, opération très pratique s'il y a beaucoup de valeurs différentes. Notez que cet encodeur suppose des données ordonnées, ce qui n'est pas toujours désirable. Il occupe moins d'espace que l'encodage 1 parmi n (*1-hot*), et plusieurs algorithmes arborescents peuvent en exploiter les résultats.

L'encodeur de labels ne peut traiter qu'une colonne à la fois :

```
>>> from sklearn import preprocessing
>>> lab = preprocessing.LabelEncoder()
>>> lab.fit_transform(X_cat)
array([0,1])
```

À partir de valeurs ainsi encodées, vous pouvez les décoder avec la méthode `.inverse_transform` :

```
>>> lab.inverse_transform([1, 1, 0])
array(['Guitar', 'Guitar', 'Bass'], dtype=object)
```

Vous obtenez le même résultat avec **pandas**. Il faut d'abord convertir la colonne vers un type de colonne catégorielle puis générer la valeur numérique à partir du contenu.

L'extrait suivant produit une série de données numériques à partir d'une série **pandas**. Nous garantissons que la catégorie est ordonnée au moyen de la méthode `.as_ordered` :

```
>>> X_cat.name.astype(
...     "category"
... ).cat.as_ordered().cat.codes + 1
0    1
1    2
dtype: int8
```

# Encodage fréquentiel

Pour des données catégorielles à forte cardinalité, une autre solution consiste à effectuer un encodage fréquentiel. Cela consiste à remplacer le nom de la catégorie par la quantité trouvée dans les données d'entraînement. Nous pouvons y parvenir avec **pandas**. Nous commençons par utiliser la méthode `.value_counts` de **pandas** pour créer une correspondance entre les chaînes et le dénombrement. Nous pouvons ensuite utiliser la méthode `.map` pour réaliser l'encodage :

```
>>> mapping = X_cat.name.value_counts()
>>> X_cat.name.map(mapping)
0    1
1    1
Name: name, dtype: int64
```

Pensez à bien sauvegarder la procédure de mise en correspondance pour pouvoir encoder les données futures dans les mêmes conditions.

# Des catégories à partir des chaînes

Une technique permettant d'augmenter l'exactitude du modèle Titanic consiste à extraire les titres dans les noms. La classe `Counter` permet rapidement de trouver les titres les plus fréquents :

```
>>> from collections import Counter
>>> c = Counter()
>>> def triples(val):
...     for i in range(len(val)):
...         c[val[i : i + 3]] += 1
>>> df.name.apply(triples)
>>> c.most_common(10)
[(', M', 1282),
 (' Mr', 954),
 ('r. ', 830),
 ('Mr.', 757),
```

```
('s. ', 460),
('n, ', 320),
(' Mi', 283),
('iss', 261),
('ss.', 261),
('Mis', 260)]
```

Nous repérons ainsi les titres les plus fréquents.

Il est également possible d'utiliser une expression régulière pour récupérer une lettre majuscule suivie de lettres minuscules et d'un point :

```
>>> df.name.str.extract(
...     "([A-Za-z]+)\.", expand=False
... ).head()
0      Miss
1    Master
2      Miss
3        Mr
4       Mrs
Name: name, dtype: object
```

Pour obtenir la fréquence, nous appliquons `.value_counts` :

```
>>> df.name.str.extract(
...     "([A-Za-z]+)\.", expand=False
... ).value_counts()
Mr          757
Miss        260
Mrs         197
Master       61
Dr            8
Rev           8
Col           4
Mlle          2
Ms            2
Major         2
Dona          1
Don           1
Lady          1
Countess      1
Capt          1
Sir           1
Mme           1
Jonkheer      1
Name: name, dtype: int64
```

Ce livre ne peut pas donner tous les détails à propos des expressions régulières. Dans notre exemple, l'expression sélectionne un ou plusieurs caractères alphabétiques suivis d'un signe point.

Ces différentes manipulations avec la librairie **pandas** permettent de créer des variables factices et de combiner des colonnes ayant peu d'occurrences en d'autres catégories ou de les abandonner.

## Autres encodages catégoriels

Il existe une librairie nommée **categorical_encoding** (`https://oreil.ly/JbxWG`) qui contient plusieurs transformateurs **scikit-learn** servant à convertir des données catégorielles en données numériques. Un des avantages de cette librairie est qu'elle permet de générer des structures `DataFrame` à la norme **pandas**, alors que **scikit-learn** convertit les sorties en tableaux **numpy**.

Un des algorithmes proposés est un encodeur à hachage. Il pourra vous servir si vous ne savez pas combien de catégories vous allez traiter ou bien lorsque vous utilisez un sac de mots (*Bag of words*) pour symboliser le texte. Le processus crée des valeurs de hachage à partir des colonnes catégorielles pour produire des éléments `n_components`. Il peut s'avérer très pratique si vous faites de l'apprentissage en ligne (avec des modèles qui peuvent être mis à jour).

```
>>> import category_encoders as ce
>>> he = ce.HashingEncoder(verbose=1)
>>> he.fit_transform(X_cat)
  col_0 col_1 col_2 col_3 col_4 col_5 col_6 col_7
0     0     0     0     1     0     1     0     0
1     0     2     0     0     0     0     0     0
```

Vous disposez aussi d'un encodeur ordinal pour convertir des colonnes catégorielles ordonnées vers une seule colonne de nombres. Voyons comment convertir la colonne de taille *`size`* ainsi. En cas de valeurs manquantes dans le dictionnaire, nous attribuons la valeur par défaut -1 :

```
>>> size_df = pd.DataFrame(
...     {
...         "name": ["Fred", "John", "Matt"],
...         "size": ["small", "med", "xxl"],
...     }
... )
>>> ore = ce.OrdinalEncoder(
...     mapping=[
...         {
...             "col": "size",
...             "mapping": {
...                 "small": 1,
...                 "med": 2,
...                 "lg": 3,
...             },
...         }
...     ]
... )
>>> ore.fit_transform(size_df)
   name  size
```

```
0  Fred    1.0
1  John    2.0
2  Matt   -1.0
```

Les algorithmes de la librairie `categorical_encoding` sont décrits plus en détail dans une page Web (`https://oreil.ly/JUtYh`).

En cas de forte cardinalité, c'est-à-dire d'un grand nombre de valeurs différentes, vous pouvez opter pour un des encodeurs bayésiens produisant une seule colonne par colonne catégorielle : `TargetEncoder`, `LeaveOneOutEncoder`, `WOEEncoder`, `JamesSteinEncoder` ou `MEstimateEncoder`.

Voici par exemple comment convertir la colonne *`survival`* du jeu Titanic vers une fusion entre probabilité postérieure de cible et probabilité antérieure utilisant l'information catégorielle du titre. La probabilité postérieure correspond au degré d'exactitude une fois tenu compte des caractéristiques :

```
>>> def get_title(df):
...     return df.name.str.extract(
...         "([A-Za-z]+)\.", expand=False
...     )
>>> te = ce.TargetEncoder(cols="Title")
>>> te.fit_transform(
...     df.assign(Title=get_title), df.survived
... )["Title"].head()
0    0.676923
1    0.508197
2    0.676923
3    0.162483
4    0.786802
Name: Title, dtype: float64
```

# Caractéristiques temporelles

La librairie **`fastai`** possède la fonction `add_datepart` qui permet de générer des colonnes d'attributs temporels à partir d'une colonne de date et heure. Cette fonction est la bienvenue car la plupart des algorithmes de mécapprentissage ne sont pas capables d'inférer ce type de signal utile à partir d'une représentation de date sous forme numérique :

```
>>> from fastai.tabular.transform import (
...     add_datepart,
... )
>>> dates = pd.DataFrame(
...     {
...         "A": pd.to_datetime(
...             ["9/17/2001", "Jan 1, 2002"]
...         )
...     }
... )
```

```
>>> add_datepart(dates, "A")
>>> dates.T
                          0           1
AYear                  2001        2002
AMonth                    9           1
AWeek                    38           1
ADay                     17           1
ADayofweek                0           1
ADayofyear              260           1
AIs_month_end         False       False
AIs_month_start       False        True
AIs_quarter_end       False       False
AIs_quarter_start     False        True
AIs_year_end          False       False
AIs_year_start        False        True
AElapsed         1000684800  1009843200
```

`add_datepart` provoque une mutation de la structure `DataFrame`. **Pandas** peut le faire aussi, mais pas en utilisation normale !

# Ajout d'une caractéristique col_na

La librairie **fastai** offrait une fonction pour créer une colonne pour recevoir une valeur pour un manquant (la médiane) informant ainsi de l'existence du manquant. En effet, le fait qu'une valeur manque peut être un signal utile. À titre d'information, voici une copie de cette fonction avec un exemple d'utilisation :

```
>>> from pandas.api.types import is_numeric_dtype
>>> def fix_missing(df, col, name, na_dict):
...     if is_numeric_dtype(col):
...         if pd.isnull(col).sum() or (
...             name in na_dict
...         ):
...             df[name + "_na"] = pd.isnull(col)
...             filler = (
...                 na_dict[name]
...                 if name in na_dict
...                 else col.median()
...             )
...             df[name] = col.fillna(filler)
...             na_dict[name] = filler
...         return na_dict

>>> data = pd.DataFrame({"A": [0, None, 5, 100]})
>>> fix_missing(data, data.A, "A", {})
{'A': 5.0}
>>> data
     A     A_na
```

```
0   0.0    False
1   5.0     True
2   5.0    False
3 100.0    False
```

Voici la version correspondante de **pandas** :

```
>>> data = pd.DataFrame({"A": [0, None, 5, 100]})
>>> data["A_na"] = data.A.isnull()
>>> data["A"] = data.A.fillna(data.A.median())
```

# Création manuelle de caractéristiques

Avec **pandas**, vous pouvez créer de nouvelles caractéristiques. Dans le cas du jeu Titanic, nous allons créer des données agrégées de cabine (âge maximal par cabine, âge moyen par cabine, *etc.*). Pour créer ces données puis les fusionner, nous nous servons de la méthode de **pandas** nommée `.groupby` puis nous réalignons les données par rapport aux données originales avec la méthode `.merge` :

```
>>> agg = (
...     df.groupby("cabin")
...     .agg("min,max,mean,sum".split(","))
...     .reset_index()
... )
>>> agg.columns = [
...     "_".join(c).strip("_")
...     for c in agg.columns.values
... ]
>>> agg_df = df.merge(agg, on="cabin")
```

Si vous avez besoin d'obtenir la somme des bonnes et des mauvaises colonnes, vous pouvez créer une nouvelle colonne correspondant à l'addition des colonnes agrégées (ou appliquer une autre opération mathématique). Mais ceci est une opération un peu plus délicate qui suppose d'avoir étudié les données encore plus en détail. (N.d.T. : Prenez note de la variable *agg_df* que nous allons réutiliser.)

CHAPITRE 8

# Sélection de caractéristiques

La sélection de caractéristiques permet de se concentrer sur celles qui sont utiles au modèle. En effet, les caractéristiques non pertinentes peuvent avoir un effet négatif. Par exemple, les caractéristiques corrélées rendent instables ou difficiles à interpréter les coefficients dans les régressions ou l'importance des caractéristiques dans les modèles arborescents.

Il faut également craindre la *malédiction des dimensions*. En augmentant le nombre de dimensions dans vos données, vous les rendez moins denses, ce qui rend de plus en plus difficile l'extraction de signaux intéressants, sauf à ajouter encore des données. Les calculs de voisinage deviennent de moins en moins fructueux.

Il faut enfin tenir compte du temps d'entraînement qui est en général proportionnel au nombre de colonnes (et parfois de façon vraiment non linéaire !). Vous obtiendrez donc un modèle plus rapidement et de meilleure qualité si vous pouvez rester concis et précis au niveau du nombre de colonnes. Nous allons découvrir plusieurs exemples qui repartent du jeu de données *agg_df* que nous avons produit à la fin du chapitre précédent. Rappelons qu'il s'agit du jeu Titanic auquel nous avons ajouté quelques colonnes pour des données au sujet des cabines. Du fait que le jeu a agrégé des valeurs numériques pour chaque cabine, il va contenir de nombreuses corrélations. Nous verrons donc aussi la technique PCA et l'élément `.feature_importances_` d'un classifieur arborescent.

## Colonnes colinéaires

Pour repérer les colonnes dont le coefficient de corrélation est au moins égal à 0.95, nous pouvons nous servir de la fonction `correlated_columns` que nous venons de définir ou bien utiliser le code source suivant :

```
>>> limit = 0.95
>>> corr = agg_df.corr()
```

```
>>> mask = np.triu(
...     np.ones(corr.shape), k=1
... ).astype(bool)
>>> corr_no_diag = corr.where(mask)
>>> coll = [
...     c
...     for c in corr_no_diag.columns
...     if any(abs(corr_no_diag[c]) > threshold)
... ]
>>> coll
['pclass_min', 'pclass_max', 'pclass_mean',
'sibsp_mean', 'parch_mean', 'fare_mean',
'body_max', 'body_mean', 'sex_male', 'embarked_S']
```

L'outil de visualisation *Yellowbrick Rank2* permet de produire une carte d'intensité des corrélations.

Les colinéarités multiples peuvent être rendues visibles par le paquetage **`rfpimp`** (`https://oreil.ly/MsnXc`), à installer si nécessaire. Il devient alors possible d'entraîner une forêt aléatoire avec la fonction `plot_dependence_heatmap` pour chaque colonne numérique à partir des autres colonnes d'un jeu d'entraînement. La valeur de dépendance correspond au score R2 à partir des estimations de prédiction *out-of-bag* (OOB) de la colonne (Figure 8.1).

Pour exploiter cette situation, nous cherchons les valeurs proches de 1. Le label dans l'axe X correspond à la caractéristique qui prédit le label dans l'axe Y. Lorsqu'une caractéristique en prédit une autre, vous pouvez enlever la caractéristique prédite (celle de l'axe Y). Dans notre exemple, *`fare`* permet de prédire quatre autres caractéristiques (*`pclass`*, *`sibsp`*, *`parch`* et *`embarked_Q`*). Nous conservons *`fare`* et supprimons les autres sans rien perdre en performances :

```
>>> rfpimp.plot_dependence_heatmap(
...     rfpimp.feature_dependence_matrix(X_train),
...     value_fontsize=12,
...     label_fontsize=14,
...     figsize=(8, 8),sn
... )
>>> fig = plt.gcf()
>>> fig.savefig(
...     "images/mlpr_0801.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

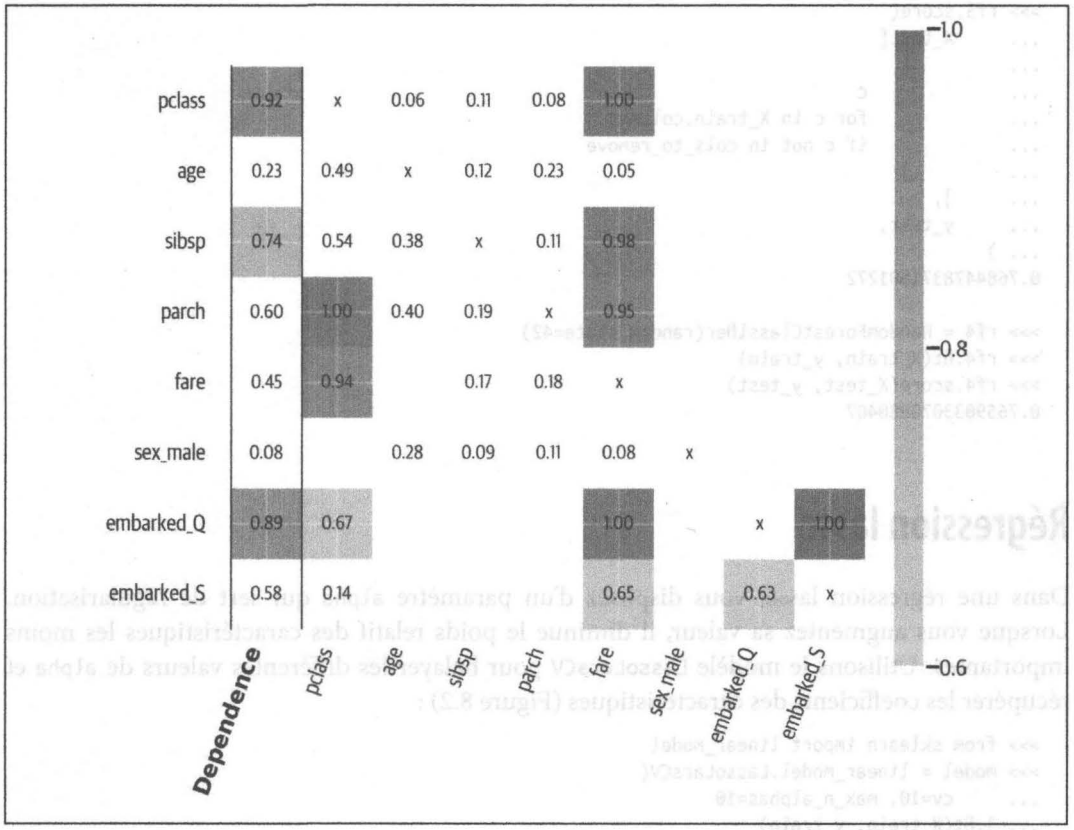



*Figure 8.1 : Carte d'intensité des dépendances (**heat map**). Les colonnes pclass, sibsp, parch et embarked_Q pouvant être prédites de la colonne fare, nous les supprimons.*

Voici le code prouvant que nous obtenons le même score après avoir supprimé ces colonnes :

```
>>> cols_to_remove = [
...     "pclass",
...     "sibsp",
...     "parch",
...     "embarked_Q",
... ]
>>> rf3 = RandomForestClassifier(random_state=42)
>>> rf3.fit(
...     X_train[
...         [
...             c
...             for c in X_train.columns
...             if c not in cols_to_remove
...         ]
...     ],
...     y_train,
... )
```

```
>>> rf3.score(
...     X_test[
...         [
...             c
...             for c in X_train.columns
...             if c not in cols_to_remove
...         ]
...     ],
...     y_test,
... )
0.7684478371501272

>>> rf4 = RandomForestClassifier(random_state=42)
>>> rf4.fit(X_train, y_train)
>>> rf4.score(X_test, y_test)
0.7659033078880407
```

# Régression lasso

Dans une régression lasso, vous disposez d'un paramètre `alpha` qui sert de régularisation. Lorsque vous augmentez sa valeur, il diminue le poids relatif des caractéristiques les moins importantes. Utilisons le modèle `LassoLarsCV` pour balayer les différentes valeurs de `alpha` et récupérer les coefficients des caractéristiques (Figure 8.2) :

```
>>> from sklearn import linear_model
>>> model = linear_model.LassoLarsCV(
...     cv=10, max_n_alphas=10
... ).fit(X_train, y_train)
>>> fig, ax = plt.subplots(figsize=(12, 8))
>>> cm = iter(
...     plt.get_cmap("tab20")(
...         np.linspace(0, 1, X.shape[1])
...     )
... )
>>> for i in range(X.shape[1]):
...     c = next(cm)
...     ax.plot(
...         model.alphas_,
...         model.coef_path_.T[:, i],
...         c=c,
...         alpha=0.8,
...         label=X.columns[i],
...     )
>>> ax.axvline(
...     model.alpha_,
...     linestyle="-",
...     c="k",
...     label="alphaCV",
... )
```

```
>>> plt.ylabel("Regression Coefficients")
>>> ax.legend(X.columns, bbox_to_anchor=(1, 1))
>>> plt.xlabel("alpha")
>>> plt.title(
...     "Regression Coefficients Progression for Lasso Paths"
... )
>>> fig.savefig(
...     "images/mlpr_0802.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

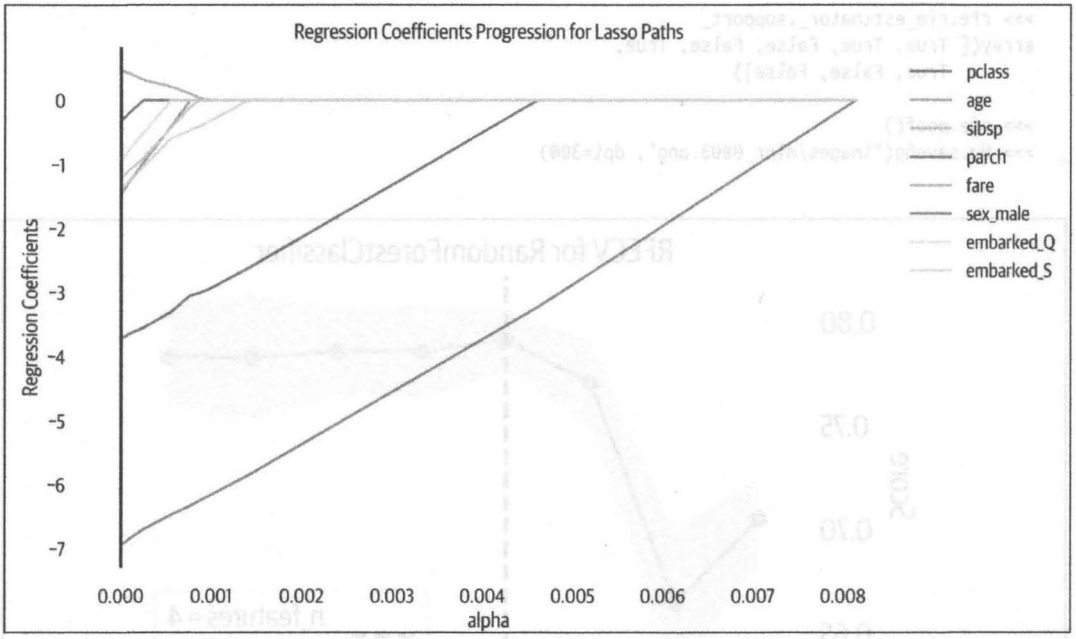



*Figure 8.2 : Variation des coefficients de caractéristiques en fonction de celle de alpha dans une régression lasso.*

# Élimination récursive de caractéristiques

Cette élimination récursive va enlever les caractéristiques les plus faibles, puis ajuster un modèle (Figure 8.3). L'opération consiste à transmettre un modèle **`scikit-learn`** avec un attribut `.coef` ou `.feature_importances_` :

```
>>> from yellowbrick.features import RFECV
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> rfe = RFECV(
```

```
...     ensemble.RandomForestClassifier(
...         n_estimators=100
...     ),
...     cv=5,
... )
>>> rfe.fit(X, y)

>>> rfe.rfe_estimator_.ranking_
array([1, 1, 2, 3, 1, 1, 5, 4])

>>> rfe.rfe_estimator_.n_features_
4

>>> rfe.rfe_estimator_.support_
array([ True, True, False, False, True,
        True, False, False])

>>> rfe.poof()
>>> fig.savefig("images/mlpr_0803.png", dpi=300)
```

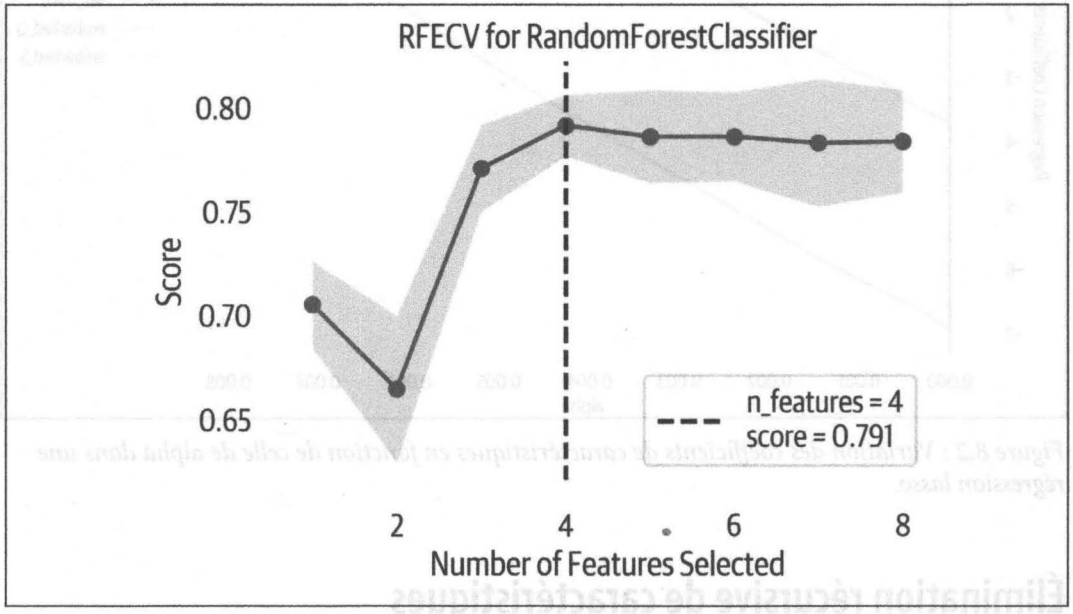



*Figure 8.3 : Élimination récursive de caractéristiques.*

Nous utilisons cette élimination récursive pour détecter les dix caractéristiques les plus importantes (dans ce jeu agrégé, nous remarquons que nous avons divulgâché la colonne *survival*) :

```
>>> from sklearn.feature_selection import RFE
>>> model = ensemble.RandomForestClassifier(
...     n_estimators=100
... )
>>> rfe = RFE(model, 4)
>>> rfe.fit(X, y)
>>> agg_X.columns[rfe.support_]
Index(['pclass', 'age', 'fare', 'sex_male'], dtype='object')
```

## Informations mutuelles

La librairie **sklearn** propose des tests non paramétriques qui se servent des k-voisins les plus proches pour détecter les informations mutuelles entre les caractéristiques et la cible. Par informations mutuelles, on désigne la quantité d'informations recueillies par observation d'une autre variable.

La valeur est donc égale à zéro ou plus. Si elle est nulle, c'est qu'il n'y a pas de relation (Figure 8.4). La valeur n'est pas liée et représente le nombre de bits partagés entre caractéristique et cible :

```
>>> from sklearn import feature_selection

>>> mic = feature_selection.mutual_info_classif(
...     X, y
... )
>>> fig, ax = plt.subplots(figsize=(10, 8))
>>> (
...     pd.DataFrame(
...         {"feature": X.columns, "vimp": mic}
...     )
...     .set_index("feature")
...     .plot.barh(ax=ax)
... )
>>> fig.savefig("images/mlpr_0804.png")
```

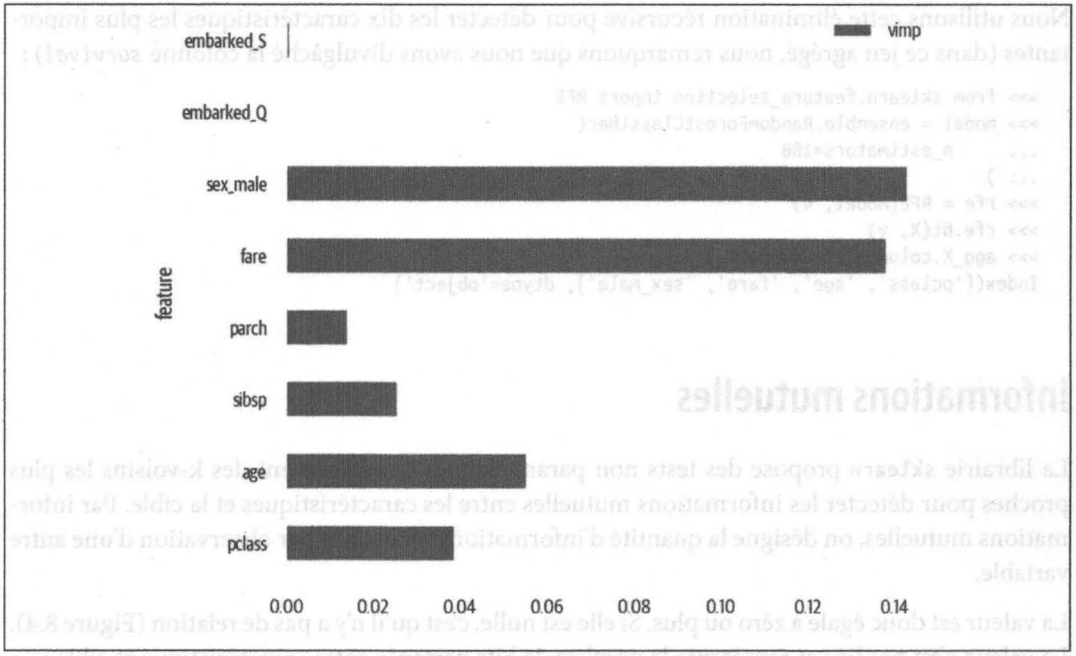



*Figure 8.4 : Diagramme d'informations mutuelles.*

# Analyse par composantes principales PCA

Vous pouvez également sélectionner des caractéristiques grâce à une analyse par composantes principales. Une fois que vous avez détecté ces composantes, vous pouvez chercher les caractéristiques qui ont contribué le plus à celles-ci. Ce sont celles qui ont la plus grande variance. Rappelons que c'est un algorithme non supervisé qui ne prend pas la variable y en compte.

Nous reviendrons sur la PCA dans le Chapitre 17.

# Importance des caractéristiques

La plupart des modèles arborescents permettent d'exploiter un attribut `.feature_importances_` après l'entraînement. Une valeur élevée signifie en général qu'il y aura plus d'erreur si la caractéristique est supprimée du modèle. Pour tout détail, voyez les chapitres traitant des modèles arborescents.

CHAPITRE 9

# Classes non équilibrées

Lorsque vous classifiez des données, si les classes ne sont pas équilibrées au niveau de leur taille, les plus présentes vont influer sur le modèle. Si vous avez par exemple un cas positif pour 99 cas négatifs, vous atteignez aisément 99 % de précision en considérant tous les cas comme négatifs. Plusieurs options sont disponibles pour gérer les classes déséquilibrées.

## Changement de métrique

Une solution consiste à utiliser autre chose pour le calibrage que l'exactitude (*accuracy*), par exemple la valeur AUC. Lorsque les tailles cibles sont différentes, la précision et le rappel constituent de meilleures options, mais il y en a d'autres.

## Algorithmes arborescents et ensembles

Les performances des modèles arborescents dépendent de la distribution des petites classes. Lorsqu'il y a tendance au regroupement, la classification est plus facile.

Les méthodes d'ensemble peuvent aider également à extraire les classes minoritaires. Les modèles arborescents tels que les forêts aléatoires et les amplifications à gradient extrême (*Extreme Gradient Boosting, XGBoost*) offrent des options telles que le *bagging* et le *boosting*.

## Pénalisation du modèle

Plusieurs modèles de classification de **`scikit-learn`** acceptent un paramètre nommé `class_weight`. En lui fournissant la valeur `'balanced'`, vous demandez d'essayer de régulariser les

classes minoritaires en incitant le modèle à les classer correctement. Vous pouvez également lancer une recherche *grid search* en stipulant les options de pondération ; il suffit de transmettre un dictionnaire qui met en correspondance les classes et les poids (les petites classes ayant les poids les plus forts).

La librairie **XGBoost** (`https://xgboost.readthedocs.io`) comporte le paramètre `max_delta_step` dont la valeur peut aller de 1 à 10 pour rendre les tables de mises à jour plus conservatrices. Elle offre également le paramètre `scale_pos_weight` pour définir la proportion entre échantillons positifs et négatifs pour les classes binaires. Enfin, le paramètre `eval_metric` doit recevoir la valeur `'auc'` à la place de la valeur par défaut `'error'` pour les classifications.

Le modèle KNN possède le paramètre `weights` avec lequel les voisins les plus proches peuvent être biaisés (décalés). En choisissant la valeur `'distance'` pour ce paramètre, vous pouvez augmenter les performances si les échantillons des classes minoritaires sont proches les uns des autres.

## Suréchantillonnage des minoritaires

Plusieurs techniques permettent de suréchantillonner les classes minoritaires. Voici comment faire avec **`scikit-learn`** :

```
>>> from sklearn.utils import resample
>>> mask = df.survived == 1
>>> surv_df = df[mask]
>>> death_df = df[~mask]
>>> df_upsample = resample(
...     surv_df,
...     replace=True,
...     n_samples=len(death_df), random_state=42,)
>>> df2 = pd.concat([death_df, df_upsample])

>>> df2.survived.value_counts()
1    809
0    809
Name: survived, dtype: int64
```

Il est également possible d'échantillonner avec remplacement de façon aléatoire au moyen de la librairie **`imblearn`** :

```
>>> from imblearn.over_sampling import (RandomOverSampler,)
>>> ros = RandomOverSampler(random_state=42)
>>> X_ros, y_ros = ros.fit_sample(X, y)
>>> pd.Series(y_ros).value_counts()
1    809
0    809
dtype: int64
```

Pour cet exemple, il faut installer la librairie **imbalanced-learn** :

```
pip install imblearn
```

## Génération de données minoritaires

La librairie **imblearn** peut générer de nouveaux échantillons de classes minoritaires avec l'un des deux algorithmes à échantillonnage SMOTE (*Synthetic Minority Oversampling TEchnique*) et ADASYN (*Adaptive Synthetic*). L'algorithme SMOTE sélectionne un des k-voisins les plus proches, établit une ligne entre les deux échantillons puis choisit un point sur cette ligne. ADASYN y ressemble, mais génère plus d'échantillons à partir de ceux qui sont les plus difficiles à apprendre. Les deux classes correspondantes dans **imblearn** portent les noms `over_sampling.SMOTE` et `over_sampling.ADASYN`.

## Sous-échantillonnage des majoritaires

Une autre approche pour rééquilibrer des classes consiste à sous-échantillonner les classes majoritaires comme dans cet exemple, avec **sklearn** :

```
>>> from sklearn.utils import resample
>>> mask = df.survived == 1
>>> surv_df = df[mask]
>>> death_df = df[~mask]
>>> df_downsample = resample(
...     death_df,
...     replace=False,
...     n_samples=len(surv_df),
...     random_state=42,)
>>> df3 = pd.concat([surv_df, df_downsample])

>>> df3.survived.value_counts()
1    500
0    500
Name: survived, dtype: int64
```

N'utilisez pas le paramètre de remplacement `replace` pour un sous-échantillonnage.

La librairie **imblearn** propose plusieurs algorithmes de sous-échantillonnage :

`ClusterCentroids`

Utilise les k-moyennes pour synthétiser des données avec les centroïdes.

`RandomUnderSampler`

Sélectionne les échantillons de façon aléatoire.

`NearMiss`

Utilise les voisins les plus proches pour sous-échantillonner.

`TomekLink`

Sous-échantillonne en supprimant les échantillons proches les uns des autres.

`EditedNearestNeighbours`

Supprime les échantillons dont les voisins ne sont soit pas dans la majorité, soit sont tous dans la même classe.

`RepeatedNearestNeighbours`

Appelle de façon répétée `EditedNearestNeighbours`.

`AllKNN`

Classe similaire à la précédente, en augmentant le nombre de voisins proches pendant les différentes étapes du sous-échantillonnage.

`CondensedNearestNeighbour`

Sélectionne un échantillon de la classe à sous-échantillonner, puis itère parmi les autres échantillons de la même classe. Ajoute l'échantillon si KNN ne se trompe pas dans la classification.

`OneSidedSelection`

Supprime les échantillons bruités.

`NeighbourhoodCleaningRule`

Utilise les résultats de `EditedNearestNeighbours` puis applique KNN.

`InstanceHardnessThreshold`

Entraîne un modèle puis supprime les échantillons à faible probabilité.

Toutes ces classes supportent la méthode `.fit_sample`.

## Sur échantillonnage puis sous-échantillonnage

La librairie **`imblearn`** implémente également les algorithmes SMOTEEN et SMOTE Tomek. Tous deux suréchantillonnent puis sous-échantillonnent pour nettoyer les données.

# CHAPITRE 10
# Classification

La classification est un processus d'apprentissage supervisé qui consiste à attribuer des labels (étiquettes) à des échantillons en se basant sur les caractéristiques. Dans cette approche supervisée, le but est que l'algorithme apprenne à exploiter les labels dans une classification ou les valeurs numériques dans une régression.

Nous allons faire un tour d'horizon des différents modèles de classification les plus usités. Pratiquement tous sont disponibles dans la librairie **sklearn**. Nous en verrons d'autres qui n'en font pas partie, mais les librairies correspondantes ont pris soin de conserver la même interface de programmation API. Il devient ainsi facile de tester plusieurs familles de modèles pour en comparer les performances.

L'approche **sklearn** suppose de commencer par créer une instance du modèle, puis de lui appliquer la méthode `.fit` avec les données et les labels d'entraînement. Nous appelons ensuite la méthode `.predict` ou une de ses variantes (`.predict_proba` ou `.predict_log_proba`) en fournissant le modèle ajusté. Il reste ensuite à évaluer le modèle avec la méthode `.score` à laquelle nous fournissons les données et les labels de test.

Le principal défi est en général l'organisation des données sous une forme exploitable par **sklearn**. Les données d'entrée qui correspondent par convention à $X$ doivent se présenter sous forme d'un tableau de $m$ par $n$ dans **numpy** ou d'une structure `DataFrame` dans **pandas**. Le tableau contient autant de lignes $m$ que d'échantillons, chacune possédant $n$ colonnes pour les caractéristiques. Le label $y$ est un vecteur ou une série **pandas** de taille $m$ possédant une valeur (une classe) pour chaque échantillon.

La méthode `.score` calcule l'exactitude (*accuracy*) moyenne, mais celle-ci ne suffit en général pas à évaluer un classifieur. Nous verrons donc d'autres métriques possibles.

Nous allons découvrir de nombreux modèles en présentant leur efficacité, les techniques de prétraitement requises, les solutions pour éviter le surajustement et nous dirons si chaque modèle permet une interprétation aisée des résultats.

Voici les cinq méthodes que l'approche de type **`sklearn`** implémente :

`fit(X, y[, sample_weight])`

Ajuste un modèle.

`predict(X)`

Prédit des classes.

`predict_log_proba(X)`

Prédit une probabilité log.

`predict_proba(X)`

Prédit une probabilité.

`score(X, y[, sample_weight])`

Renvoie une exactitude (*accuracy*).

# Régression logistique

Une régression logistique permet d'obtenir des probabilités en se basant sur une fonction logistique. Méfiez-vous : malgré le nom, il s'agit d'un outil de classification et pas de régression. C'est le modèle de classification standard dans la plupart des domaines scientifiques.

Voici quelques propriétés essentielles du modèle que nous fournirons pour chacun des modèles suivants :

## Propriétés

*Efficacité d'exécution*

Sait utiliser `n_jobs` sauf avec le solveur `'liblinear'`.

*Prétraitement des données*

Si `solver` vaut `'sag'` ou `'saga'`, standardiser pour que la convergence fonctionne. Sait gérer des entrées éparses (creuses).

*Prévention du surajustement*

Le paramètre `C` contrôle la régularisation. Une valeur faible implique plus de régularisation et inversement. Le paramètre `penalty` peut valoir `'l1'` ou `'l2'` (valeur par défaut).

*Interprétation des résultats*

L'attribut `.coef_` du modèle ajusté permet de connaître les coefficients de la fonction de décision. Un changement de $x$ d'une unité provoque un changement du log du rapport des chances (*odds*) selon le coefficient. L'attribut `.intercept_` correspond au log du rapport des chances inverse de la condition de référence (*baseline*).

### Exemple d'utilisation du modèle

```
>>> from sklearn.linear_model import (
...     LogisticRegression,
... )
>>> lr = LogisticRegression(random_state=42)
>>> lr.fit(X_train, y_train)
LogisticRegression(C=1.0, class_weight=None,
    dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100,
    multi_class='ovr', n_jobs=1, penalty='l2',
    random_state=42, solver='liblinear',
    tol=0.0001, verbose=0, warm_start=False)
>>> lr.score(X_test, y_test)
0.8040712468193384

>>> lr.predict(X.iloc[[0]])
array([1])
>>> lr.predict_proba(X.iloc[[0]])
array([[0.08698937, 0.91301063]])
>>> lr.predict_log_proba(X.iloc[[0]])
array([[-2.4419694 , -0.09100775]])
>>> lr.decision_function(X.iloc[[0]])
array([2.35096164])
```

### Paramètres d'instance

`penalty='l2'`

Norme de pénalisation, L1 ou L2.

`dual=False`

Utiliser la formulation duelle (seulement avec `'l2'` et `'liblinear'`).

`C=1.0`

Valeur flottante positive pour la force de la régularisation inverse. Une petite valeur donne une régularisation plus forte.

`fit_intercept=True`

Ajoute un biais à la fonction de décision.

`intercept_scaling=1`

Si fit_intercept et `'liblinear'`, rééchelonnement de l'Interception.

`max_iter=100`

Nombre maximal d'itérations.

`multi_class='ovr'`

Utiliser une contre toutes pour chaque classe, ou si multinomial, entraîner une classe.

`class_weight=None`

Un dictionnaire ou `'balanced'`.

`solver= liblinear`

`'liblinear'` convient aux petits jeux de données. `'newton-cg'`, `'sag'`, `'saga'` et `'lbfgs'` s'appliquent aux données multiclasses. `'liblinear'` et `'saga'` ne fonctionnent qu'avec la pénalité `'l1'`, les autres avec `'l2'`.

`tol=0.0001`

Tolérance d'arrêt.

`verbose=0`

Afficher tous les messages si valeur non nulle.

`warm_start=False`

Si valeur True, réutiliser l'ajustement précédent.

`njobs=1`

Nombre de processeurs CPU à utiliser. Tous si *-1*. Ne fonctionne que si `multi_class='over'` et si le solveur est autre que `'liblinear'`.

### Attributs après ajustement

`coef_`

Coefficients des caractéristiques dans la fonction de décision.

`intercept_`

Interception de la fonction de décision.

`n_iter_`

Nombre d'itérations.

L'interception correspond au *log odds* de la condition de référence. Nous pouvons la reconvertir en un pourcentage d'exactitude (une proportion) :

```
>>> lr.intercept_
array([-0.62386001])
```

La fonction logit inverse permet de constater que la référence pour le taux de survie est de 34 % :

```
>>> def inv_logit(p):
...     return np.exp(p) / (1 + np.exp(p))

>>> inv_logit(lr.intercept_)
array([0.34890406])
```

Nous pouvons inspecter les coefficients. Leur logit inverse permet d'obtenir la proportion de cas positifs. Dans l'exemple, si la valeur du prix du billet (*fare*) monte, le taux de survie augmente. Si la caractéristique de sexe est mâle, le taux de survie diminue :

```
>>> cols = X.columns
>>> for col, val in sorted(
...     zip(cols, lr.coef_[0]),
...     key=lambda x: x[1],
...     reverse=True,
... ):
...     print(
...         f"{col:10}{val:10.3f} {inv_logit(val): 10.3f}"
...     )
fare          0.104    0.526
parch        -0.062    0.485
sibsp        -0.274    0.432
age          -0.296    0.427
embarked_Q   -0.504    0.377
embarked_S   -0.507    0.376
pclass       -0.740    0.323
sex_male     -2.400    0.083
```

La librairie **`Yellowbrick`** permet aussi de visualiser les coefficients. Elle possède un paramètre `relative=True` qui force la plus grande valeur à 100 ou -100, en calculant toutes les autres valeurs comme pourcentages de ce plafond (Figure 10.1) :

```
>>> from yellowbrick.features.importances import (
...     FeatureImportances,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(lr)
>>> fi_viz.fit(X, y)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1001.png", dpi=300)
```

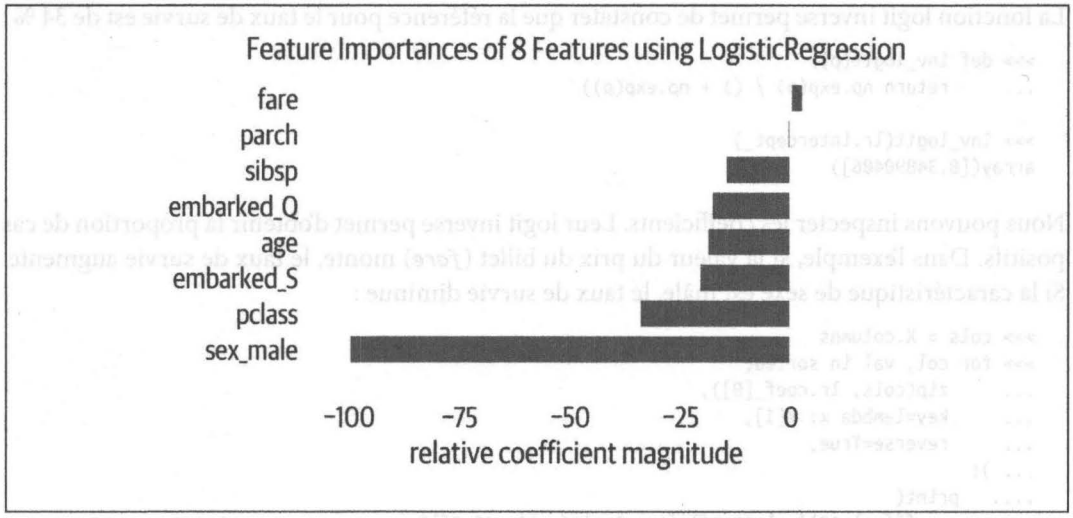



*Figure 10.1 : Importance relative des caractéristiques par rapport au plus grand coefficient de régression absolue.*

# Bayésien naïf

Le classifieur probabiliste bayésien naïf suppose que les caractéristiques des données sont indépendantes. Il est très utilisé pour la manipulation du texte, par exemple pour intercepter les pourriels. Un des avantages de ce modèle est qu'il permet un entraînement avec un petit nombre d'échantillons, puisque les caractéristiques ne dépendent pas les unes des autres. Un des inconvénients du modèle est qu'il ne sait pas capturer les interactions entre caractéristiques. Ce modèle simple fonctionne malgré tout lorsque les données ont beaucoup de colonnes, et constitue donc un bon modèle de référence.

Trois classes lui correspondent dans **`sklearn`** : `GaussianNB`, `MultinomialNB` et `BernoulliNB`. La première suppose une distribution gaussienne, avec des caractéristiques continues et une distribution normale. La deuxième sert aux valeurs numériques dénombrables (discrètes) et la troisième aux caractéristiques booléennes.

### Propriétés

*Efficacité d'exécution*

Pour l'entraînement, `O(Nd)`, avec `N` nombre d'exemples d'entraînement et `d` les dimensions. Pour les tests, `O(cd)`, avec `c` le nombre de classes.

*Prétraitement des données*

Les données sont supposées indépendantes. Le fonctionnement est meilleur si les colonnes colinéaires sont supprimées. Pour des données numériques continues, il peut être avisé de les distribuer par *binning*. Le profil gaussien supposant une distribution normale, il peut être nécessaire de transformer les données en ce sens.

*Prévention du surajustement*

Montre un biais important et une variance faible (les ensembles ne réduisent pas la variance).

*Interprétation des résultats*

Le pourcentage dénote la probabilité qu'un échantillon appartienne à une classe en se basant sur les antécédents sans tenir compte des caractéristiques.

### Exemple d'utilisation

```
>>> from sklearn.naive_bayes import GaussianNB
>>> nb = GaussianNB()
>>> nb.fit(X_train, y_train)
GaussianNB(priors=None, var_smoothing=1e-09)
>>> nb.score(X_test, y_test)
0.7837150127226463

>>> nb.predict(X.iloc[[0]])
array([1])
>>> nb.predict_proba(X.iloc[[0]])
array([[2.17472227e-08, 9.99999978e-01]])
>>> nb.predict_log_proba(X.iloc[[0]])
array([[-1.76437798e+01, -2.17472227e-08]])
```

### Paramètres d'instance

`priors=None`

Probabilité préalable des classes.

`var_smoothing=1e-9`

Valeur ajoutée à la variance pour stabiliser les calculs.

### Attributs après ajustement

`class_prior_`

Probabilité des classes.

`class_count_`

Nombre de classes.

`theta_`

Moyenne de chaque colonne par classe.

`sigma_`

Variance de chaque colonne par classe.

`epsilon_`

Valeur additive pour chaque variance.

Ce genre de modèle peut subir le problème de probabilité nulle. En effet, lorsque vous tentez de classifier un échantillon pour lequel il n'y a pas de données d'entraînement, la probabilité sera égale à zéro. Une solution consiste à utiliser un lissage de Laplace. Dans **`sklearn`**, vous contrôlez cela au moyen du paramètre `alpha` qui vaut 1 par défaut. Ce lissage est possible pour les deux modèles `MultinomialNB` et `BernoulliNB`.

## Machine à vecteurs de support (SVM)

La machine SVM (*Support Vector Machine*) est un algorithme qui tente de positionner une ligne (ou un plan ou un hyperplan) entre les différentes classes de sorte de maximiser la distance entre cette ligne et les points des classes. L'algorithme tente donc de trouver la meilleure séparation entre les classes. Les vecteurs de support correspondant sous des points sur le bord de l'hyperplan de division.

Plusieurs implémentations de SVM sont disponibles dans **`sklearn`**. `SVC` réutilise la librairie **`libsvm`** alors que `LinearSVC` réutilise la librairie **`liblinear`**.

Vous disposez également de `linear_model.SGDClassifier`, qui implémente une machine SVM en utilisant le paramètre par défaut `loss`. Nous ne verrons dans ce chapitre que la première implémentation.

L'algorithme SVM s'en sort généralement bien ; il est capable de supporter les espaces linéaires et non linéaires, grâce à une *astuce du noyau*. Cette astuce consiste à considérer qu'il est possible de créer une frontière de décision dans une nouvelle dimension en réduisant une formule dont le calcul est plus simple que la mise en correspondance réelle de tous les points dans la nouvelle dimension. Le noyau utilisé par défaut correspond à la fonction de base radiale (`'rbf'`) que vous contrôlez grâce au paramètre `gamma`. Elle est en mesure de mapper un espace d'entrée dans un espace à forte dimensionnalité.

### Propriétés

*Efficacité d'exécution*

Les performances de la version **scikit-learn** valent `O(n4)`, ce qui la rend mal adaptée aux gros volumes. Les performances peuvent être améliorées au détriment de l'exactitude avec un noyau linéaire ou le modèle `LinearSVC`. En optimisant le paramètre `cache_size`, on peut atteindre le niveau de performances `O(n3)`.

*Prétraitement des données*

L'algorithme n'est pas insensible au changement d'échelle. Il est fortement recommandé de standardiser les données.

*Prévention du surajustement*

Le paramètre de pénalité `C` contrôle la régularisation. Une petite valeur se traduit par une petite marge dans l'hyperplan. Une grande valeur pour `gamma` aura tendance à surajuster les données d'entraînement. Le modèle `LinearSVC` accepte les paramètres `loss` et `penalty` pour gérer la régularisation.

*Interprétation des résultats*

Il s'agit d'inspecter les vecteurs *via* `.support_vectors_`, mais ils ne sont pas faciles à expliquer. Avec un noyau linéaire, vous pouvez étudier `.coef_`.

### Exemple d'utilisation

```
>>> from sklearn.svm import SVC
>>> svc = SVC(random_state=42, probability=True)
>>> svc.fit(X_train, y_train)
SVC(C=1.0, cache_size=200, class_weight=None,
  coef0=0.0, decision_function_shape='ovr',
  degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=True, random_state=42,
  shrinking=True, tol=0.001, verbose=False)
>>> svc.score(X_test, y_test)
0.8015267175572519

>>> svc.predict(X.iloc[[0]])
array([1])
>>> svc.predict_proba(X.iloc[[0]])
array([[0.15344656, 0.84655344]])
>>> svc.predict_log_proba(X.iloc[[0]])
array([[-1.87440289, -0.16658195]])
```

Pour obtenir des probabilités, il faut ajouter `probability=True`, mais vous allez ralentir l'ajustement.

Le résultat ressemble à un *perceptron*, mais en cherchant la marge maximale. L'erreur sera minimisée si les données ne sont pas séparables de façon linéaire. Un autre noyau peut être envisagé.

### Paramètres d'instance

`C=1.0`

Paramètre de pénalité. Plus la valeur est faible, plus la frontière de décision est serrée, avec plus de surajustement.

`cache_size=200`

Taille du cache (Mo). Une valeur plus forte peut améliorer les temps d'entraînement pour de grands jeux de données.

`class_weight=None`

Un dictionnaire ou `'balanced'`. Utilisez un dictionnaire pour ajuster `C` pour chacune des classes.

`coef0=0.0`

Terme indépendant pour les noyaux poly et sigmoïdes.

`decision_function_shape='ovr'`

Choix entre une contre le reste (*one versus rest*, `'ovr'`) et une contre une (*one versus one*).

`degree=3`

Degré pour un noyau polynomial.

`gamma='auto'`

Coefficient du noyau qui peut être un nombre, la valeur `'scale'` (par défaut `0.22, 1 / (nbre caract * X.std() )`) ou la valeur `'auto'` (par défaut prior, `1 / nbre caract`). Une valeur faible pousse au surajustement des données d'entraînement.

`kernel='rbf'`

Type de noyau : `'linear'`, `'poly'`, `'rbf'` (par défaut), `'sigmoid'`, `'precomputed'` ou le nom d'une fonction.

`max_iter=-1`

Nombre maximal d'itérations du solveur. *-1* pour ne pas déterminer de limite.

`probability=False`

Active l'estimation de probabilités, mais ralentit l'entraînement.

`random_state=None`

Graine du générateur aléatoire.

`shrinking=True`

Applique une heuristique rétrécissante.

`tol=0.001`

Tolérance d'arrêt.

`verbose=False`

Verbosité.

### Attributs après ajustement

`support_`

Indices des vecteurs de support.

`support_vectors_`

Vecteurs de support.

`n_support_vectors_`

Nombre de vecteurs de support par classe.

`coef_`

Coefficient pour noyau linéaire

## K-plus proches voisins (KNN)

L'algorithme KNN (*K-nearest neighbors*) classifie en fonction de la distance par rapport à un nombre `k` d'échantillons d'entraînement. Toute la famille d'algorithmes est dite *basée instance* car l'apprentissage se fait sans aucun paramètre. Le modèle suppose que la distance suffit à réussir une inférence. Il n'utilise aucune hypothèse quant aux données sous-jacentes ou à leur distribution.

Le défi est ici de bien choisir la valeur de `k`. En outre, la malédiction des dimensions peut gêner la mesure des distances parce qu'il y a peu de différence en cas de dimensions importantes entre les voisins les plus proches et les plus éloignés.

### Propriétés

*Efficacité d'exécution*

Pour l'entraînement, `O(1)`, mais les données doivent être stockées. Pour le test, `O(Nd)` avec `N` le nombre d'exemples d'entraînement et `d` la dimensionnalité.

*Prétraitement des données*

Oui, les calculs basés sur les distances sont plus efficaces après standardisation.

*Prévention du surajustement*

Augmenter `n_neighbors`. Changer `p` pour une métrique `L1` ou `L2`.

**Interprétation des résultats**

Interprétation des k-voisins les plus proches de l'échantillon avec la méthode `.kneighbors`. Si ces voisins peuvent être expliqués, ils expliquent le résultat.

### Exemple d'utilisation

```
>>> from sklearn.neighbors import (
...     KNeighborsClassifier,
... )
>>> knc = KNeighborsClassifier()
>>> knc.fit(X_train, y_train)
KNeighborsClassifier(algorithm='auto',
  leaf_size=30, metric='minkowski',
  metric_params=None, n_jobs=1, n_neighbors=5,
  p=2, weights='uniform')
>>> knc.score(X_test, y_test)
0.7837150127226463

>>> knc.predict(X.iloc[[0]])
array([1])

>>> knc.predict_proba(X.iloc[[0]])
array([[0., 1.]])
```

### Paramètres d'instance

`algorithm='auto'`

Autres options : `'brute'`, `'ball_tree'` ou `'kd_tree'`.

`leaf_size=30`

Paramètre servant aux algorithmes d'arbres.

`metric='minkowski'`

Métrique de distance.

`metric_params=None`

Dictionnaire complémentaire de paramètres pour la fonction de métrique personnalisée.

`n_jobs=1`

Nombre d'unités centrales CPU.

`n_neighbors=5`

Nombre de voisins.

`p=2`

Paramètre de puissance de Minkowski : 1 = manhattan (L1), 2 = euclidien (L2).

`weights='uniform'`

Une autre valeur possible est `'distance'`, auquel cas les points les plus proches ont plus d'effet.

Vous pouvez spécifier une fonction appelable ou une des métriques de distance suivantes :

```
'euclidean',      'manhattan',    'chebyshev',     'minkowski',   'wminkowski',
'seuclidean',     'mahalanobis',  'haversine',     'hamming',     'canberra',
'braycurtis',     'jaccard',      'matching',      'dice',
'rogerstanimoto', 'russellrao',   'sokalmichener', 'sokalsneath'
```

Si k est une valeur paire et si les voisins sont divisés, le résultat va dépendre de l'ordre des données d'entraînement.

# Arbre de décision

Un arbre de décision peut se comparer au questionnaire que parcourt le docteur lorsqu'il tente de trouver les causes de vos soucis. Il est possible de créer un tel arbre de décision comportant une série de questions pour prédire une classe cible. Ce modèle a plusieurs avantages, et notamment le support des données non numériques dans certaines implémentations, un travail de préparation des données limité, pas de besoin de remise à l'échelle, le support des relations non linéaires, une mise en lumière de l'importance des caractéristiques et une simplicité de description.

L'algorithme utilisé par défaut pour créer un tel arbre porte le nom CART (*Classification And Regression Tree*). Les décisions sont bâties au moyen de la valeur d'impureté ou coefficient de Gini. Il s'agit de parcourir les caractéristiques pour trouver la valeur qui donne le plus faible risque d'erreur de classement.

En utilisant les valeurs par défaut, vous aboutissez à un arbre totalement déployé, donc surajusté. Vous vous servez de `max_depth` et de la validation croisée pour contrôler sa croissance.

## Propriétés

*Efficacité d'exécution*

Pour la création, itération parmi toutes les caractéristiques `m` puis tri des échantillons `n`, selon `O(mn log n)`. Pour la prédiction, parcours de l'arbre selon `O(hauteur)`.

*Prétraitement des données*

Pas de remise à l'échelle nécessaire. Les valeurs manquantes doivent être éliminées et conversion en numérique.

*Prévention du surajustement*

Donner une valeur plus faible à `max_depth` et augmenter `min_impurity_decrease`.

*Interprétation des résultats*

Parcours possible des étapes de l'arbre de décision. En raison de ces étapes, un arbre gère mal les relations linéaires (un petit changement d'un nombre fait emprunter un chemin différent). Les arbres sont également très dépendants des données d'entraînement. Un tout petit changement peut radicalement modifier l'arbre.

### Exemple d'utilisation

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> dt = DecisionTreeClassifier(
...     random_state=42, max_depth=3
... )
>>> dt.fit(X_train, y_train)
DecisionTreeClassifier(class_weight=None,
  criterion='gini', max_depth=None,
  max_features=None, max_leaf_nodes=None,
  min_impurity_decrease=0.0,
  min_impurity_split=None,
  min_samples_leaf=1, min_samples_split=2,
  min_weight_fraction_leaf=0.0, presort=False,
  random_state=42, splitter='best')

>>> dt.score(X_test, y_test)
0.8142493638676844

>>> dt.predict(X.iloc[[0]])
array([1])
>>> dt.predict_proba(X.iloc[[0]])
array([[0.02040816, 0.97959184]])
>>> dt.predict_log_proba(X.iloc[[0]])
array([[-3.8918203 , -0.02061929]])
```

### Paramètres d'instance

`class_weight=None`

Poids des classes du dictionnaire. `'balanced'` produit des valeurs inversement proportionnelles aux fréquences des classes. Par défaut, chaque classe reçoit la valeur un en cas de multiclasse, il faut une liste du dictionnaire, en une contre toutes (OVR) pour chaque classe.

`criterion='gini'`

Fonction de bifurcation *splitting* : `'gini'` ou `'entropy'`.

`max_depth=None`

Profondeur de l'arbre. Par défaut, fait croître l'arbre jusqu'à ce que les feuilles contiennent moins de `min_samples_split`.

`max_features=None`

Nombre de caractéristiques à examiner pour la bifurcation. Toutes par défaut.

`max_leaf_nodes=None`

Nombre de feuilles, illimité par défaut.

`min_impurity_decrease=0.0`

Division du nœud si cela diminue impurity >= value.

`min_impurity_split=None`

Déprécié.

`min_samples_leaf=1`

Nombre minimal d'échantillons par feuille.

`min_samples_split=2`

Nombre minimal d'échantillons requis pour diviser un nœud.

`min_weight_fraction_leaf=0.0`

Minimum de la somme des poids requis pour les nœuds des feuilles.

`presort=False`

Si valeur True, peut accélérer l'entraînement pour de petits volumes de données ou une profondeur limitée.

`random_state=None`

Graine du générateur aléatoire.

`splitter='best'`

Choix entre `'random'` et `'best'`.

### Attributs après ajustement

`classes_`

Labels des classes.

`feature_importances_`

Tableau d'importances de Gini.

`n_classes_`

Nombre de classes.

`n_features_`

Nombre de caractéristiques.

`tree_`

Objet arbre sous-jacent.

Exemple pour visualiser un arbre (Figure 10.2) :

```
>>> import pydotplus
>>> from io import StringIO
>>> from sklearn.tree import export_graphviz
>>> dot_data = StringIO()
>>> tree.export_graphviz(
...     dt,
...     out_file=dot_data,
...     feature_names=X.columns,
...     class_names=["Died", "Survived"],
...     filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
...     dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1002.png")
```

Pour Jupyter, ajoutez ceci :

```
from IPython.display import Image
Image(g.create_png())
```

Vous devez installer la librairie **`pydotplus`** ainsi :

```
pip install pydotplus
```

Le paquetage **`dtreeviz`** (`https://github.com/parrt/dtreeviz`) peut aider à comprendre le fonctionnement de l'arbre de décision. Il produit un arbre avec des histogrammes labellisés, ce qui aide à comprendre le traitement (Figure 10.3). Avec l'outil Jupyter, nous pouvons directement afficher l'objet `viz`. Depuis un script, il s'agit d'appeler la méthode `.save` pour produire un fichier PDF, SVG ou PNG :

```
>>> viz = dtreeviz.trees.dtreeviz(
...     dt,
...     X,
...     y,
...     target_name="survived",
...     feature_names=X.columns,
...     class_names=["died", "survived"],)
>>> viz
```

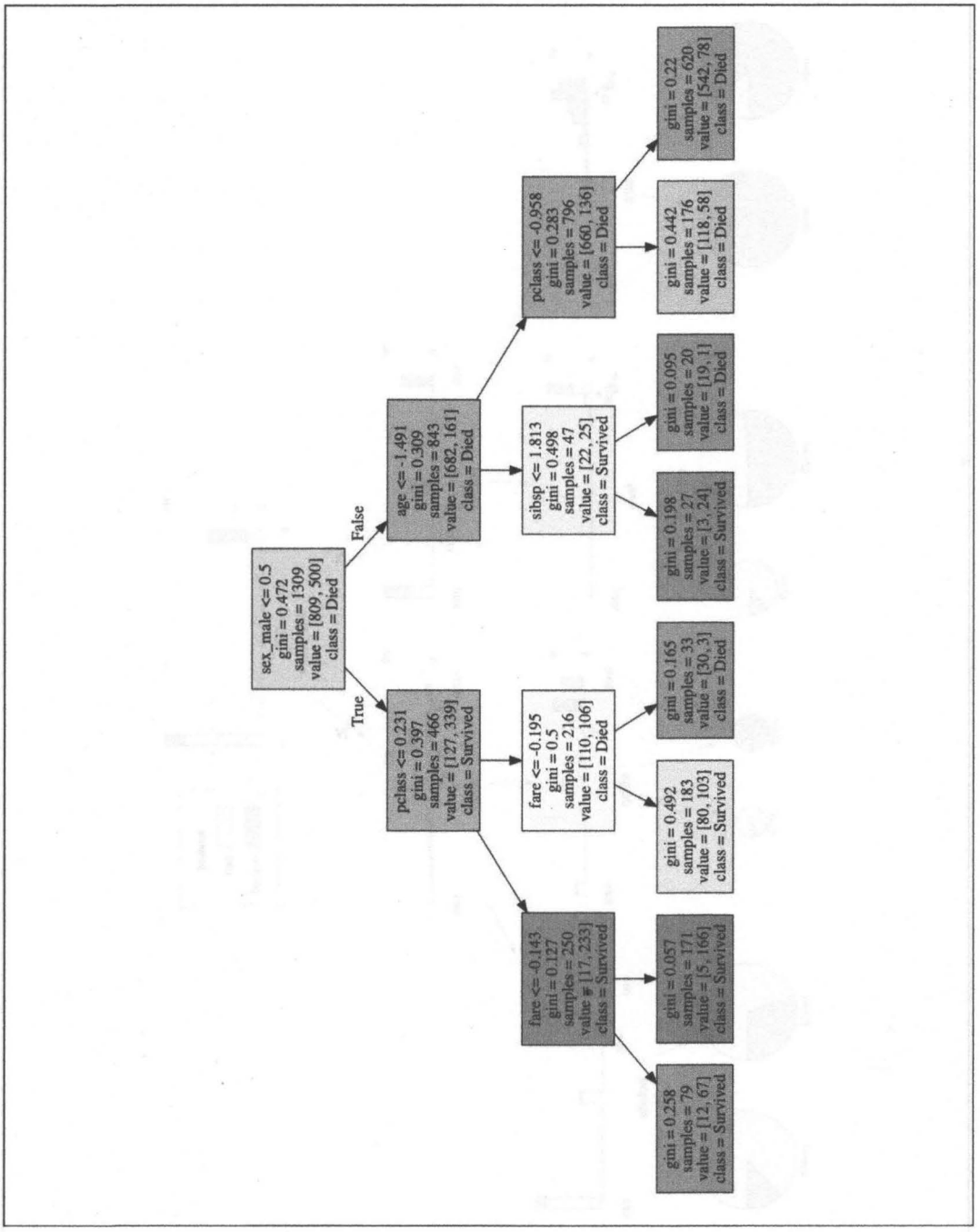



*Figure 10.2 : Arbre de décision.*

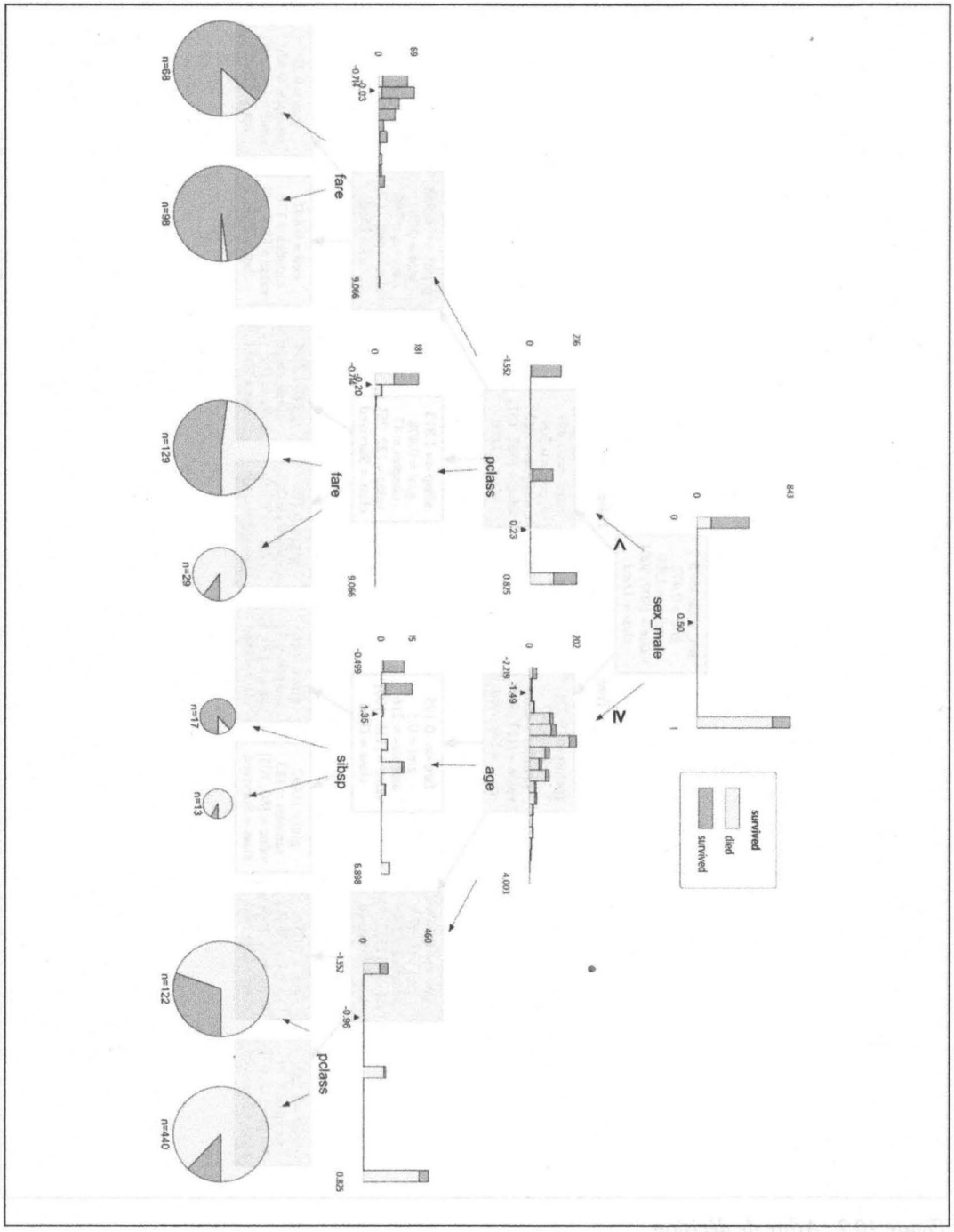



*Figure 10.3 : Visualisation avec dtreeviz.*

Recherche de l'importance des caractéristiques pour montrer l'importance de Gini (réduction d'erreur en utilisant cette possibilité) :

```
>>> for col, val in sorted(
...     zip(X.columns, dt.feature_importances_),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
sex_male  0.607
pclass    0.248
sibsp     0.052
fare      0.050
age       0.043
```

L'importance des caractéristiques peut aussi être visualisée avec **`Yellowbrick`** (Figure 10.4) :

```
>>> from yellowbrick.features.importances import (
...     FeatureImportances,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(dt)
>>> fi_viz.fit(X, y)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1004.png", dpi=300)
```

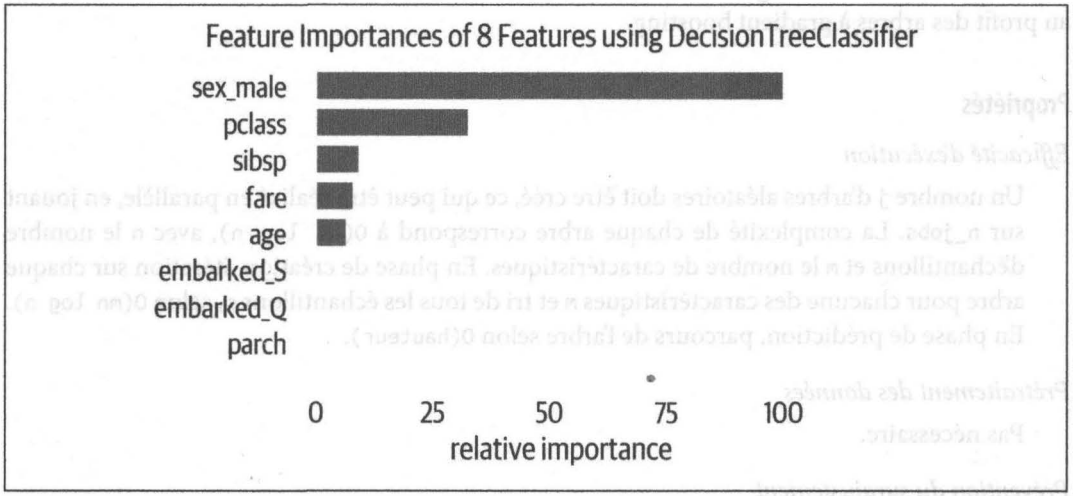



*Figure 10.4 : Importance de caractéristiques en coefficient de Gini pour un arbre de décision (normalisé selon l'importance de « male »).*

## Forêt aléatoire

Une forêt aléatoire est un ensemble d'arbres de décision dans lequel la tendance au surajustement des arbres est corrigée par l'utilisation de la technique de *bagging*. Le principe consiste à créer de nombreux arbres et à les entraîner sur des sous-échantillons et des caractéristiques des données choisies au hasard, afin de réduire la variance.

Du fait que l'entraînement se fait sur des sous-échantillons des données, les forêts aléatoires sont en mesure d'évaluer les erreurs hors sac OOB (*Out Of Bag*) et d'évaluer les performances. Elles permettent également de suivre l'importance des caractéristiques en faisant la moyenne de leur importance parmi tous les arbres.

La technique de *bagging* a été décrite à l'origine dans un essai du marquis de Condorcet en 1785. Le principe est que lorsque vous créez un jury, vous augmentez le nombre de membres en ajoutant tous ceux qui ont une chance supérieure à 50 % de fournir le bon verdict, puis vous faites une moyenne des décisions. Vous améliorez donc les résultats chaque fois que vous ajoutez un nouveau membre et que son processus de sélection est indépendant de celui des autres.

Le principe de la forêt aléatoire consiste à créer une forêt d'arbres qui sont entraînés sur des colonnes différentes de données. Dès que chaque arbre a plus de 50 % de chances de bien classifier, vous pouvez prendre en compte sa prédiction. Ce genre de forêt a beaucoup servi en classification comme en régression, même si les forêts aléatoires ont été récemment délaissées au profit des arbres à gradient boosting.

### Propriétés

*Efficacité d'exécution*

Un nombre `j` d'arbres aléatoires doit être créé, ce qui peut être réalisé en parallèle, en jouant sur `n_jobs`. La complexité de chaque arbre correspond à `O(mn log n)`, avec `n` le nombre d'échantillons et `m` le nombre de caractéristiques. En phase de création, itération sur chaque arbre pour chacune des caractéristiques `m` et tri de tous les échantillons `n`, selon `O(mn log n)`. En phase de prédiction, parcours de l'arbre selon `O(hauteur)`.

*Prétraitement des données*

Pas nécessaire.

*Prévention du surajustement*

Ajouter des arbres (`n_estimators`). Exploiter `lower max_depth`.

*Interprétation des résultats*

Importance des caractéristiques accessible, mais pas de parcours possibles d'un arbre de décision isolé. En revanche, inspection possible des arbres isolés depuis l'ensemble.

### Exemple d'utilisation

```
>>> from sklearn.ensemble import (
...     RandomForestClassifier,
... )
>>> rf = RandomForestClassifier(random_state=42)
>>> rf.fit(X_train, y_train)
RandomForestClassifier(bootstrap=True,
  class_weight=None, criterion='gini',
  max_depth=None, max_features='auto',
  max_leaf_nodes=None, min_impurity_decrease=0.0,
  min_impurity_split=None, min_samples_leaf=1,
  min_samples_split=2, min_weight_fraction_leaf=0.0,
  n_estimators=10, n_jobs=1, oob_score=False,
  random_state=42, verbose=0, warm_start=False)
>>> rf.score(X_test, y_test)
0.7862595419847328

>>> rf.predict(X.iloc[[0]])
array([1])
>>> rf.predict_proba(X.iloc[[0]])
array([[0., 1.]])
>>> rf.predict_log_proba(X.iloc[[0]])
array([[-inf,   0.]])
```

### Paramètres d'instance

Ces options reprennent celles des arbres de décision.

`bootstrap=True`

Construction des arbres par échantillonnage interne bootstrap.

`class_weight=None`

Un dictionnaire indiquant les poids des classes ou `'balanced'`. Avec `'balanced'`, les valeurs sont inversement proportionnelles aux fréquences des classes. La valeur par défaut pour chaque classe est 1. En multiclasse, il faut une liste de dictionnaires (OVR) par classe.

`criterion='gini'`

Fonction de bifurcation *splitting* : `'gini'` ou `'entropy'`.

`max_depth=None`

Profondeur de l'arbre. Par défaut, fait croître l'arbre jusqu'à ce que les feuilles contiennent moins de `min_samples_split`.

`max_features='auto'`

Nombre de caractéristiques à examiner pour la bifurcation. Toutes par défaut.

`max_leaf_nodes=None`

Nombre de feuilles, illimité par défaut.

`min_impurity_decrease=0.0`

Division du nœud si cela diminue impurity >= value.

`min_impurity_split=None`

Déprécié.

`min_samples_leaf=1`

Nombre minimal d'échantillons par feuille.

`min_samples_split=2`

Nombre minimal d'échantillons requis pour diviser un nœud.

`min_weight_fraction_leaf=0.0`

Minimum de la somme des poids requis pour les nœuds des feuilles.

`* n_estimators=10`

Nombre d'arbres dans la forêt.

`n_jobs=1`

Nombre de processus d'ajustement et prédiction.

`oob_score=False`

Demande d'estimation de `oob_score`.

`random_state=None`

Graine du générateur aléatoire.

`verbose=0`

Verbosité.

`warm_start=False`

Ajuste une nouvelle forêt ou utilise celle qui existe.

### Attributs après ajustement

`classes_`

Labels des classes.

`feature_importances_`

Tableau d'importances de Gini.

`n_classes_`

Nombre de classes.

`n_features_`

Nombre de caractéristiques.

`oob_score_`

Score OOB. Exactitude (*accuracy*) moyenne de chaque observation non utilisée dans les arbres.

Exemple d'importance des caractéristiques avec importance de Gini (ce qui permet de réduire les erreurs) :

```
>>> for col, val in sorted(
...     zip(X.columns, rf.feature_importances_),
...     key=lambda x:   x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
age        0.285
fare       0.268
sex_male   0.232
pclass     0.077
sibsp      0.059
```

Les classifieurs à forêt aléatoire calculent l'importance des caractéristiques en cherchant la *diminution moyenne d'impureté* de chacune d'elles, autrement dit l'importance de Gini. Une caractéristique qui diminue l'incertitude de classification obtient un meilleur score.

La valeur peut être faussée lorsque les caractéristiques sont très variables au niveau de l'échelle ou de la cardinalité pour des colonnes catégorielles. Un score plus fiable est dans ce cas l'*importance des permutations*, dans laquelle les valeurs de chaque colonne sont permutées pour mesurer la chute d'exactitude. Une approche encore plus fiable est l'*importance de réduction de colonnes* dans laquelle chaque colonne est abandonnée pour évaluer le modèle. Le souci est que cela suppose de créer un nouveau modèle pour chaque colonne écartée. Voyez à ce sujet la fonction `importance` dans le paquetage **`rfpimp`** (nous avons vu ce paquetage dans le Chapitre 8) :

```
>>> import rfpimp
>>> rf = RandomForestClassifier(random_state=42)
>>> rf.fit(X_train, y_train)
>>> rfpimp.importances(
...     rf, X_test, y_test
... ).Importance

Feature
sex_male     0.155216
fare         0.043257
age          0.033079
pclass       0.027990
parch        0.020356
embarked_Q   0.005089
```

```
sibsp        0.002545
embarked_S   0.000000
Name: Importance, dtype: float64
```

# XGBoost

La librairie **sklearn** propose le classifieur `GradientBoostedClassifier`, mais vous obtiendrez de meilleurs résultats avec une implémentation qui utilise la technique d'amplification extrême *Extreme Boosting*.

C'est la raison d'être de la librairie **XGBoost** (`https://oreil.ly/WBo0g`) très usitée. Elle commence par créer un arbre pauvre puis produit par réplication de nouveaux arbres amplifiés afin de réduire l'erreur résiduelle. La fonction cherche à capturer et corriger tout motif remarquable dans les erreurs jusqu'à ce que les erreurs restantes semblent aléatoires.

## Propriétés

*Efficacité d'exécution*

**XGBoost** peut être parallélisée, en jouant sur la valeur de l'option `n_jobs` qui stipule le nombre de processeurs. Vous améliorez encore les performances en exploitant les coprocesseurs graphiques GPU.

*Prétraitement des données*

Aucun redimensionnement nécessaire pour les modèles arborescents. Les données catégorielles doivent être encodées.

*Prévention du surajustement*

Pour stopper l'entraînement s'il n'y a plus d'amélioration après `N` tours, il suffit d'utiliser le paramètre `early_stopping_rounds=N`. Les régularisations L1 et L2 sont contrôlées respectivement par `reg_alpha` et `reg_lambda`. Plus la valeur est forte, plus le traitement est conservateur.

*Interprétation des résultats*

Donne accès à l'importance des caractéristiques.

**XGBoost** propose un paramètre supplémentaire pour la méthode `.fit`, `early_stopping_rounds`, qui peut être combiné à `eval_set` pour obliger **XGBoost** à arrêter de créer des arbres s'il n'y a pas d'amélioration de la métrique après un certain nombre de tours d'amplification. De plus, le paramètre `eval_metric` peut recevoir l'une des valeurs suivantes : `'rmse'`, `'mae'`, `'logloss'`, `'error'` (par défaut), `'auc'`, `'aucpr'`, ou bien le nom d'une fonction spécifique.

### Exemple d'utilisation

```
>>> import xgboost as xgb
>>> xgb_class = xgb.XGBClassifier(random_state=42)
>>> xgb_class.fit(
...     X_train,
...     y_train,
...     early_stopping_rounds=10,
...     eval_set=[(X_test, y_test)],
... )
XGBClassifier(base_score=0.5, booster='gbtree',
  colsample_bylevel=1, colsample_bytree=1, gamma=0,
  learning_rate=0.1, max_delta_step=0, max_depth=3,
  min_child_weight=1, missing=None,
  n_estimators=100, n_jobs=1, nthread=None,
  objective='binary:logistic', random_state=42,
  reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
  seed=None, silent=True, subsample=1)

>>> xgb_class.score(X_test, y_test)
0.7862595419847328

>>> xgb_class.predict(X.iloc[[0]])
array([1])
>>> xgb_class.predict_proba(X.iloc[[0]])
array([[0.06732017, 0.93267983]], dtype=float32)
```

### Paramètres d'instance

`max_depth=3`

Profondeur maximale.

`learning_rate=0.1`

Taux d'apprentissage du boosting (*eta*), avec une valeur entre zéro et un. Après chaque pas d'amplification, les poids venant d'être ajoutés sont remis à l'échelle selon ce facteur. Plus la valeur est faible, plus le traitement est conservateur, mais il requiert plus d'arbres pour finir par converger. Dans les appels à `.train`, vous pouvez fournir un paramètre `learning_rates`, qui est une liste de taux pour chaque tour (par exemple `[.1]*100 + [. 05]*100`).

`n_estimators=100`

Nombre de tours ou d'arbres amplifiés.

`silent=True`

Option inverse de `verbose`. Contrôle l'affichage des messages pendant le traitement de boosting.

`objective='binary:logistic'`

Choix d'une tâche d'apprentissage aux fournitures d'une fonction appelable pour la classification.

`booster= gbtree`

Peut être `'gbtree'`, `'gblinear'` ou `'dart'`.

`nthread=None`

Déprécié.

`n_jobs=1`

Nombre d'exétrons à lancer (*threads*).

`gamma=0`

Contrôle l'élagage (*pruning*) avec une valeur entre zéro et l'infini. Une réduction de perte minimale est requise pour subdiviser plus encore une feuille. Une valeur de gamma importante est plus conservatrice. Lorsque les scores d'entraînement et de test divergent, fournir une valeur élevée (environ 10). Choisir une valeur plus faible si les scores convergents.

`min_child_weight=1`

Valeur minimale de la somme hessienne pour un enfant.

`max_delta_step=0`

Rend l'actualisation plus conservatrice. Valeur entre 1 et 10 pour les classes déséquilibrées.

`subsample=1`

Fraction des échantillons à utiliser pour le prochain tour.

`colsample_bytree=1`

Fraction des colonnes à utiliser pour le tour.

`colsample_bylevel=1`

Fraction des colonnes à utiliser pour le niveau.

`colsample_bynode=1`

Fraction des colonnes à utiliser pour le nœud.

`reg_alpha=0`

Régularisation L1 (moyenne des poids) qui favorise la rareté. Une augmentation rend plus conservateur.

`reg_lambda=1`

Régularisation L2 (racine des poids quadratique) qui favorise les petits poids. Une augmentation rend plus conservateur.

`scale_pos_weight=1`

Proportion entre poids négatifs et positifs.

`base_score=.5`

Prédiction initiale.

`seed=None`

Déprécié.

`random_state=0`

Graine du générateur aléatoire.

`missing=None`

Valeur pour interpréter les manquants *missings*. La valeur `None` signifie `np.nan`.

`importance_type='gain'`

Type d'importance des caractéristiques : `'gain'`, `'weight'`, `'cover'`, `'total_gain'` ou `'total_cover'`.

### Attributs

`coef_`

Coefficient pour les apprenants `gblinear` (*learners*).

`feature_importances_`

Importance des caractéristiquess pour les apprenants `gbtree`.

L'importance de caractéristiques correspond au gain moyen pour tous les nœuds dans lesquels la caractéristique est utilisée :

```
>>> for col, val in sorted(
...     zip(
...         X.columns,
...         xgb_class.feature_importances_,
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")

fare       0.420
age        0.309
pclass     0.071
sex_male   0.066
sibsp      0.050
```

Vous pouvez avec **`XGBoost`** visualiser cette importance (Figure 10.5). Le paramètre `importance_type` possède comme valeur par défaut `'weight'` qui indique le nombre de fois qu'une carac-

téristique apparaît dans un arbre. Vous pouvez aussi choisir la valeur `'gain'` pour voir le gain moyen d'utilisation de la caractéristique ou encore `'cover'` pour voir le nombre d'échantillons impactés par une division :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> xgb.plot_importance(xgb_class, ax=ax)
>>> fig.savefig("images/mlpr_1005.png", dpi=300)
```

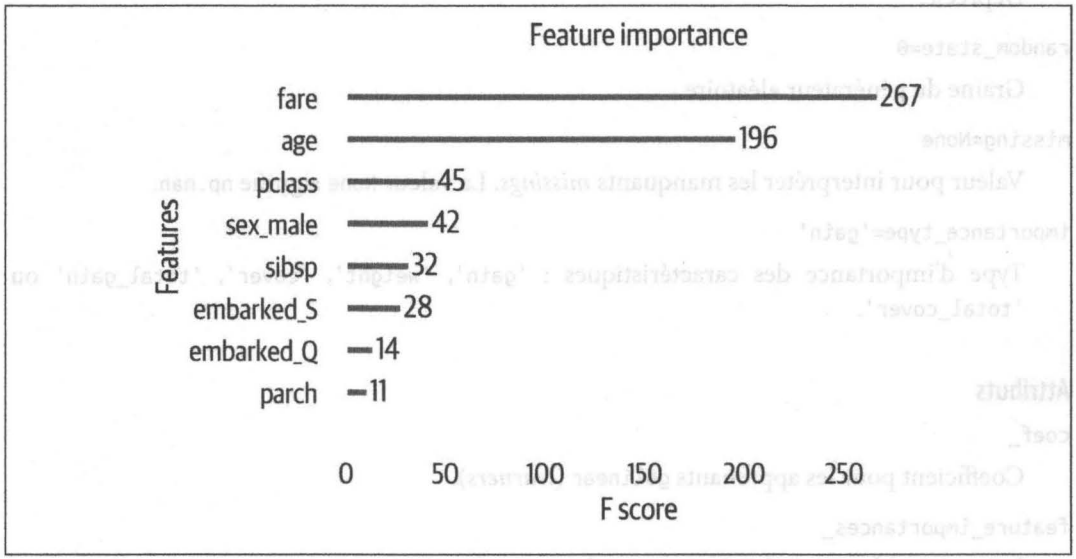



*Figure 10.5 : Importance des caractéristiques visualisant les poids (le nombre d'occurrences d'une caractéristique dans les arbres).*

La visualisation avec **`Yellowbrick`** réalise une normalisation des valeurs (Figure 10.6) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(xgb_class)
>>> fi_viz.fit(X, y)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1006.png", dpi=300)
```

**`XGBoost`** propose une représentation des arbres sous forme texte et sous forme graphique. Voici celle sous forme texte :

```
>>> booster = xgb_class.get_booster()
>>> print(booster.get_dump()[0])
0:[sex_male<0.5] yes=1,no=2,missing=1
  1:[pclass<0.23096] yes=3,no=4,missing=3
    3:[fare<-0.142866] yes=7,no=8,missing=7
      7:leaf=0.132530
      8:leaf=0.184
    4:[fare<-0.19542] yes=9,no=10,missing=9
```

```
    9:leaf=0.024598
    10:leaf=-0.1459
  2:[age<-1.4911] yes=5,no=6,missing=5
    5:[sibsp<1.81278] yes=11,no=12,missing=11
      11:leaf=0.13548
      12:leaf=-0.15000
    6:[pclass<-0.95759] yes=13,no=14,missing=13
      13:leaf=-0.06666
      14:leaf=-0.1487
```

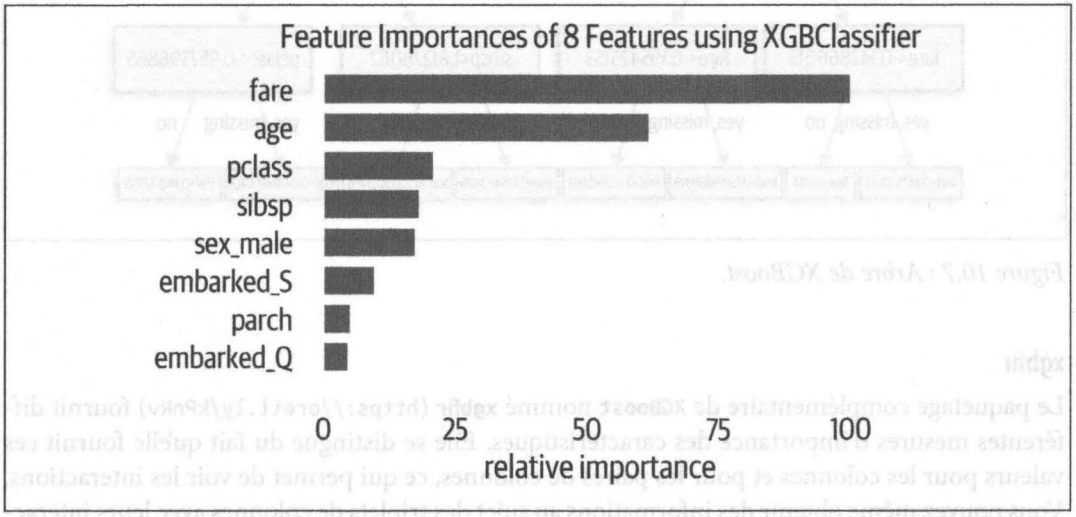



*Figure 10.6 : Importance des caractéristiques pour XGBoost avec Yellowbrick (normalisation à 100).*

La valeur dans chaque feuille correspond au score pour la classe 1. Vous pouvez la convertir en probabilité grâce à la fonction logistique. Lorsque les décisions descendent jusqu'à la feuille 7, la probabilité de la classe 1 est de 53 %, score pour un seul arbre. Si le modèle exploite 100 arbres, il faut faire la somme de toutes les valeurs de feuilles pour obtenir la probabilité avec la fonction logistique :

```
>>> # score de premier arbre, feuille 7
>>> 1 / (1 + np.exp(-1 * 0.1238))
0.5309105310475829
```

Voici la version graphique du premier arbre du modèle (Figure 10.7) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> xgb.plot_tree(xgb_class, ax=ax, num_trees=0)
>>> fig.savefig("images/mlpr_1007.png", dpi=300)
```

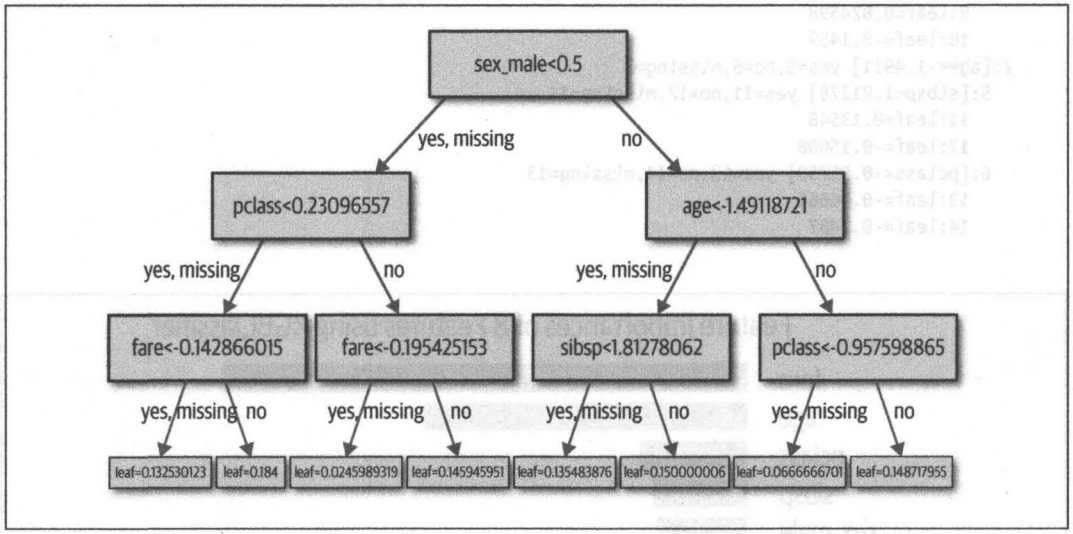



*Figure 10.7 : Arbre de XGBoost.*

### xgbfir

Le paquetage complémentaire de **`XGBoost`** nommé **`xgbfir`** (`https://oreil.ly/kPnRv`) fournit différentes mesures d'importance des caractéristiques. Elle se distingue du fait qu'elle fournit ces valeurs pour les colonnes et pour les paires de colonnes, ce qui permet de voir les interactions. Vous pouvez même obtenir des informations au sujet des triplets de colonnes avec leurs interactions. Voici les mesures disponibles :

`Gain`

: Gain total de chaque caractéristique ou de chaque interaction.

`FScore`

: Quantité de divisions possibles pour une caractéristique ou une interaction.

`wFScore`

: Nombre de divisions possibles pour une caractéristique ou une interaction, pondéré selon la probabilité des divisions envisageables.

`Average wFScore`

: `wFScore` divisé par `FScore`.

`Average Gain`

: `Gain` divisé par `FScore`.

`Expected Gain`

Total de chaque caractéristique ou interaction pondérée par la probabilité de recueillir ce gain.

L'interface propose un export vers le format tableur, et nous pouvons donc utiliser **pandas** pour récupérer les données. Voici l'importance des colonnes :

```
>>> import xgbfir
>>> xgbfir.saveXgbFI(
...     xgb_class,
...     feature_names=X.columns,
...     OutputXlsxFile="fir.xlsx",
... )
>>> pd.read_excel("/tmp/surv-fir.xlsx").head(3).T
                          0         1         2
Interaction         sex_male    pclass      fare
Gain                1311.44   585.794   544.884
FScore                   42        45       267
wFScore              39.2892   21.5038    128.33
Average wFScore     0.935458 0.477861 0.480636
Average Gain         31.2247   13.0177   2.04076
Expected Gain        1307.43   229.565   236.738
Gain Rank                  1         2         3
FScore Rank                4         3         1
wFScore Rank               3         4         1
Avg wFScore Rank           1         5         4
Avg Gain Rank              1         2         4
Expected Gain Rank         1         3         2
Average Rank         1.83333   3.16667       2.5
Average Tree Index   32.2381   20.9778   51.9101
Average Tree Depth  0.142857   1.13333   1.50562
```

La lecture du tableau permet de constater que *sex_male* offre des valeurs élevées pour le gain, le `wFScore` moyen, les gains moyen et espéré, alors que *fare* se distingue au niveau de `FScore` et de `wFScore`.

Étudions les interactions pour une paire de colonnes :

```
>>> pd.read_excel(
... "fir.xlsx",
... sheet_name="Interaction Depth 1",
... ).head(2).T
Interaction     pclass|sex_male  age|sex_male
Gain                    2090.27       964.046
FScore                       35            18
wFScore                 14.3608       9.65915
Average wFScore        0.410308      0.536619
Average Gain             59.722       53.5581
Expected Gain            827.49        616.17
Gain Rank                     1             2
FScore Rank                   5            10
wFScore Rank                  4             8
```

```
Avg wFScore Rank              8           5
Avg Gain Rank                 1           2
Expected Gain Rank            1           2
Average Rank            3.33333     4.83333
Average Tree Index      18.9714     38.1111
Average Tree Depth            1     1.11111
```

Nous voyons que les deux interactions principales concernent *sex_male* en combinaison avec *pclass* et *age*. Si vous pouviez créer un modèle avec seulement deux caractéristiques, vous choisiriez sans doute *pclass* et *sex_male*.

Voyons pour terminer un triplet :

```
>>> pd.read_excel(
... "fir.xlsx",
... sheet_name="Interaction Depth 2",
... ).head(1).T
                                      0
Interaction          fare|pclass|sex_male
Gain                              2973.16
FScore                                 44
wFScore                          8.92572
Average wFScore                 0.202857
Average Gain                     67.5719
Expected Gain                    549.145
Gain Rank                              1
FScore Rank                            1
wFScore Rank                           4
Avg wFScore Rank                      21
Avg Gain Rank                          3
Expected Gain Rank                     2
Average Rank                     5.33333
Average Tree Index               16.6591
Average Tree Depth                     2
```

Nous ne montrons que le premier triplet par manque de place, mais le tableau en contient bien d'autres :

```
>>> pd.read_excel(
...     "/tmp/surv-fir.xlsx",
...     sheet_name="Interaction Depth 2",
... )[["Interaction", "Gain"]].head()
           Interaction         Gain
0  fare|pclass|sex_male  2973.162529
1   age|pclass|sex_male  1621.945151
2    age|sex_male|sibsp  1042.320428
3     age|fare|sex_male   366.860828
4    fare|fare|sex_male   196.224791
```

# Gradient Boosted avec LightGBM

**LightGBM** de Microsoft utilise un mécanisme d'échantillonnage pour prendre en charge les valeurs continues, ce qui permet une création plus rapide des arbres qu'avec, par exemple, **XGBoost**, tout en limitant l'empreinte mémoire.

**LightGBM** commence par faire croître les arbres dans le sens de la profondeur (par feuilles plutôt que par niveaux). Pour contrôler le surajustement, vous n'utilisez donc pas `max_depth`, mais `num_leaves` (avec une valeur inférieure à `2^(max_depth)`).

Au moment d'écrire ceci, il est nécessaire d'avoir un compilateur pour installer cette librairie, et la procédure est un peu plus complexe qu'un simple `pip install`.

## Propriétés

*Efficacité d'exécution*

Sait exploiter plusieurs processeurs. Peut se montrer 15 fois plus rapide que **XGBoost** en utilisant le *binning*.

*Prétraitement des données*

Permet l'encodage des colonnes catégorielles comme valeurs entières (ou comme type `Categorical` de **pandas**). Cela dit, la valeur AUC est alors dégradée en comparaison d'un encodage One-hot.

*Prévention du surajustement*

Diminuer `num_leaves`. Augmenter `min_data_in_leaf`. Utiliser `min_gain_to_split` avec `lambda_l1` ou `lambda_l2`.

*Interprétation des résultats*

Accès possible à l'importance des caractéristiques. Les arbres individuels sont faibles et souvent difficiles à interpréter.

## Exemple d'utilisation

```
>>> import lightgbm as lgb
>>> lgbm_class = lgb.LGBMClassifier(
...     random_state=42
... )
>>> lgbm_class.fit(X_train, y_train)
LGBMClassifier(boosting_type='gbdt',
  class_weight=None, colsample_bytree=1.0,
  learning_rate=0.1, max_depth=-1,
  min_child_samples=20, min_child_weight=0.001,
```

```
    min_split_gain=0.0, n_estimators=100,
    n_jobs=-1, num_leaves=31, objective=None,
    random_state=42, reg_alpha=0.0, reg_lambda=0.0,
    silent=True, subsample=1.0,
    subsample_for_bin=200000, subsample_freq=0)

>>> lgbm_class.score(X_test, y_test)
0.7964376590330788

>>> lgbm_class.predict(X.iloc[[0]])
array([1])
>>> lgbm_class.predict_proba(X.iloc[[0]])
array([[0.01637168, 0.98362832]])
```

Nous ne précisons plus que si la librairie est introuvable, il suffit de l'installer avec `pip`.

### Paramètres d'instance

`boosting_type='gbdt'`

Valeurs possibles : `'gbdt'` (*gradient boosting*), `'rf'` (*random forest*), `'dart'` (*Dropouts meet multiple Additive Regression Trees*) ou `'goss'` (*Gradient-based, One-Sided Sampling*).

`class_weight=None`

Un dictionnaire ou `'balanced'`. Utilisez un dictionnaire pour définir les poids de chaque label de classe dans les problèmes multiclasses. Pour les problèmes binaires, utilisez `is_unbalance` ou `scale_pos_weight`.

`colsample_bytree=1.0`

Plage `(0, 1.0]`. Sélection du pourcentage de caractéristiques dans chaque tour de boost.

`importance_type='split'`

Mode de calcul de l'importance des caractéristiques. `'split'` pour le nombre d'utilisations d'une caractéristique. `'gain'` pour la somme des gains des divisions pour une caractéristique.

`learning_rate=0.1`

Plage `(0, 1.0]`. Taux d'apprentissage du boosting. Une valeur faible ralentit le surajustement du fait que chaque tour de boosting a de moins en moins d'impact. Les performances en sont améliorées, mais cela requiert un plus grand nombre d'itérations `num_iterations`.

`max_depth=-1`

Profondeur d'arbre maximale. -1 correspond à illimité. Une grande profondeur a tendance à surajuster encore plus.

`min_child_samples=20`

Nombre d'échantillons pour une feuille. Une valeur faible entraîne plus de surajustement.

`min_child_weight=0.001`

Somme des poids hessiens requis pour une feuille.

`min_split_gain=0.0`

Réduction des pertes requises pour diviser une feuille.

`n_estimators=100`

Nombre d'arbres ou de tours de boosting.

`n_jobs=-1`

Nombre d'exétrons (*threads*).

`num_leaves=31`

Nombre maximal de feuilles de l'arbre.

`objective=None`

Valeur `'binary'` ou `'multiclass'` pour un classifieur. Peut être une chaîne ou un nom de fonction.

`random_state=42`

Graine du générateur aléatoire.

`reg_alpha=0`

Régularisation L1 (moyenne des poids). Une augmentation rend plus conservateur.

`reg_lambda=1`

Régularisation L2 (racine des poids quadratique). Une augmentation rend plus conservateur.

`silent=True`

Mode d'affichage verbeux.

`subsample=1.0`

Fraction du total d'échantillons à utiliser dans le prochain tour.

`subsample_for_bin=200000`

Nombre d'échantillons requis pour créer les bacs *bins*.

`subsample_freq=0`

Fréquence de sous-échantillonnage. Valeur 1 pour activer.

L'importance des caractéristiques se fonde sur la valeur de `'splits'` (nombre d'utilisations d'un produit) :

```
>>> for col, val in sorted(
...     zip(cols, lgbm_class.feature_importances_),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
fare         1272.000
age          1182.000
sibsp         118.000
pclass        115.000
sex_male      110.000
```

La librairie **`LightGBM`** permet de créer un diagramme d'importance des caractéristiques (Figure 10.8). Par défaut, le traitement se fonde sur le nombre d'utilisations d'une caractéristique, `'splits'`. Vous pouvez spécifier `'importance_type'` si vous voulez basculer vers `'gain'` :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lgb.plot_importance(lgbm_class, ax=ax)
>>> fig.tight_layout()
>>> fig.savefig("images/mlpr_1008.png", dpi=300)
```

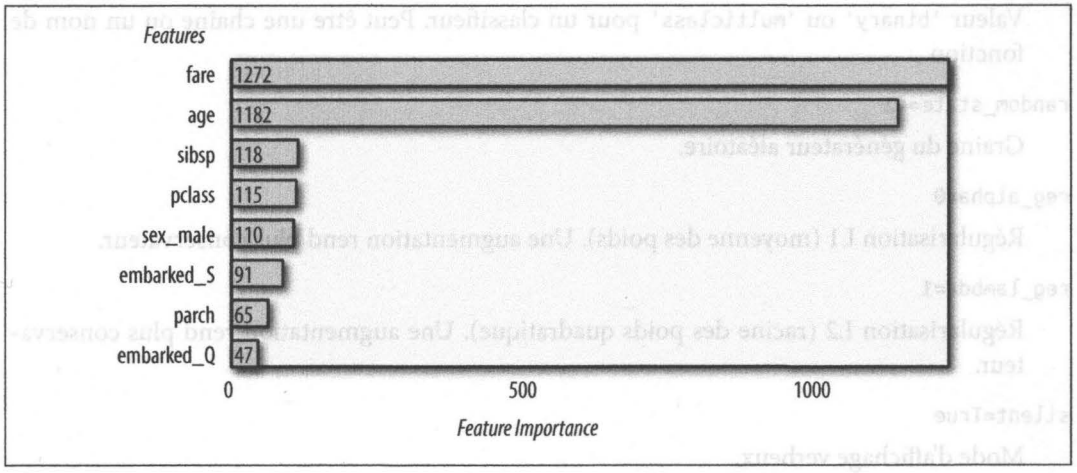



*Figure 10.8 : Divisions de l'importance des caractéristiques avec LightGBM.*

Dans la version 0.9, on ne pouvait pas utiliser **`Yellowbrick`** pour visualiser les importances de caractéristiques de **`LightGBM`**.

Nous pouvons également produire un arbre de décisions (Figure 10.9) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lgb.plot_tree(lgbm_class, tree_index=0, ax=ax)
>>> fig.savefig('images/mlpr_1009.png', dpi=300)
```

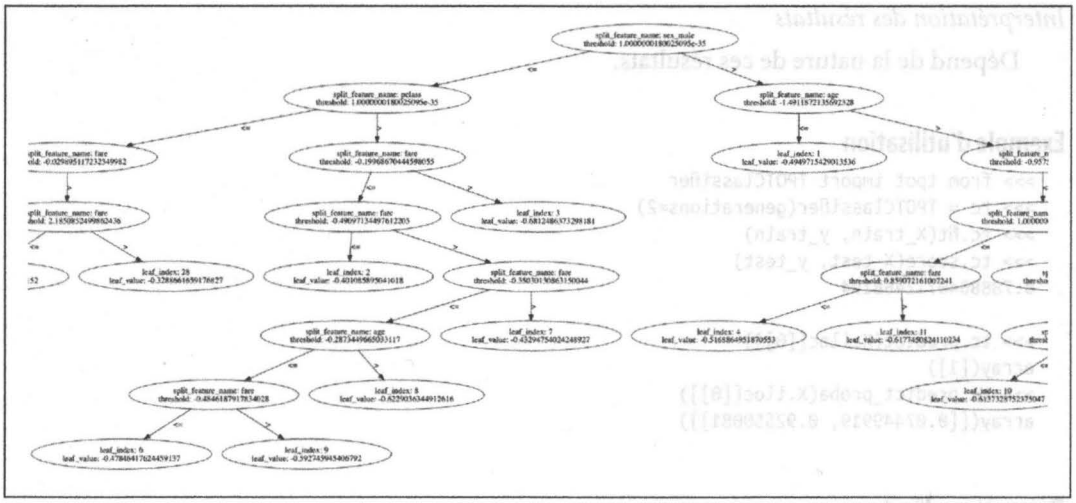


*Figure 10.9 : Arbre LightGBM.*

Pour visualiser l'arbre dans Jupyter, utilisez cette commande :

```
lgb.create_tree_digraph(lgbm_class)
```

# TPOT

La librairie TPOT (`https://oreil.ly/NFJvl`) se fonde sur un algorithme génétique afin d'essayer plusieurs modèles et ensembles. L'exécution peut de ce fait réclamer des heures ou des jours entiers ! En conséquence, en raison de tous ces modèles et des étapes de prétraitement, sans compter les hyperparamètres associés et les options d'ensemble, une génération prendra au minimum cinq vraies minutes.

## Propriétés

*Efficacité d'exécution*

L'exécution peut durer des heures ou des jours. Avec `n_jobs=-1`, vous demandez d'utiliser toutes les CPU.

*Prétraitement des données*

Les valeurs `NaN` et données catégorielles doivent être écartées.

*Prévention du surajustement*

Dans l'idéal, les résultats doivent se fonder sur une validation croisée pour limiter le surajustement.

*Interprétation des résultats*

Dépend de la nature de ces résultats.

### Exemple d'utilisation

```
>>> from tpot import TPOTClassifier
>>> tc = TPOTClassifier(generations=2)
>>> tc.fit(X_train, y_train)
>>> tc.score(X_test, y_test)
0.7888040712468194

>>> tc.predict(X.iloc[[0]])
array([1])
>>> tc.predict_proba(X.iloc[[0]])
array([[0.07449919, 0.92550081]])
```

### Paramètres d'instance

`generations=100`

Nombre d'itérations de l'exécution.

`population_size=100`

Taille de la population en programmation génétique. Une grande valeur donne de meilleures performances mais consomme de la mémoire et du temps.

`offspring_size=None`

Progéniture par génération. Valeur par défaut égale à `population_size`.

`mutation_rate=.9`

Taux de mutation de l'algorithme [0, 1]. Par défaut 0.9.

`crossover_rate=.1`

Taux de cross-over (nombre de pipelines à alimenter par génération). Plage [0, 1]. Par défaut 0.1.

`scoring='accuracy'`

Mécanisme de scoring. Se fonde sur des chaînes **`sklearn`**.

`cv=5`

Plis de validation croisée.

`subsample=1`

Instances d'entraînement de sous-échantillons. Plage [0, 1]. Par défaut 1.

`n_jobs=1`

Nombre de CPU à utiliser, -1 pour toutes.

`max_time_mins=None`

Durée d'exécution maximale en minutes.

`max_eval_time_mins=5`

Durée maximale en minutes pour évaluation d'un pipeline.

`random_state=None`

Graine du générateur aléatoire.

`config_dict`

Options de configuration d'optimisation.

`warm_start=False`

Réutilisation des appels précédents à `.fit` si `True`.

`memory=None`

Mise en cache possible des pipelines. Persistance dans un répertoire avec `'auto'` ou un chemin d'accès.

`use_dask=False`

Utilisation de `dask`.

`periodic_checkpoint_folder=None`

Chemin d'accès vers un répertoire dans lequel le meilleur pipeline sera sauvé périodiquement.

`early_stop=None`

Arrêt après un nombre stipulé de générations sans autre amélioration.

`verbosity=0`

0 = rien, 1 = minimal, 2 = beaucoup ou 3 = tout. Jauge de progression ajoutée à partir du niveau 2.

`disable_update_check=False`

Désactiver le contrôle de version.

### Attributs

`evaluated_individuals_`

Dictionnaire avec tous les pipelines qui ont été évalués.

`fitted_pipeline_`

Meilleur pipeline

Une fois que vous êtes arrivé à vos fins, vous pouvez exporter le pipeline :

```
>>> tc.export("tpot_exported_pipeline.py")
```

Voici à quoi pourrait ressembler le résultat :

```
import numpy as np
import pandas as pd
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import \
    train_test_split
from sklearn.pipeline import make_pipeline, \
    make_union
from sklearn.preprocessing import Normalizer
from tpot.builtins import StackingEstimator

# NOTE: La classe doit être marquée 'target'
# dans le fichier de données
tpot_data = pd.read_csv('PATH/TO/DATA/FILE',
    sep='COLUMN_SEPARATOR', dtype=np.float64)
features = tpot_data.drop('target', axis=1).values
training_features, testing_features, \
  training_target, testing_target = \
  train_test_split(features,
    tpot_data['target'].values, random_state=42)

# Score sur jeu d'entraînement était: 0.8122535043953432
exported_pipeline = make_pipeline(
  Normalizer(norm="max"),
  StackingEstimator(
    estimator=ExtraTreesClassifier(bootstrap=True,
      criterion="gini", max_features=0.85,
      min_samples_leaf=2, min_samples_split=19,
      n_estimators=100)),
  ExtraTreesClassifier(bootstrap=False,
    criterion="entropy", max_features=0.3,
    min_samples_leaf=13, min_samples_split=9,
    n_estimators=100)
)

exported_pipeline.fit(training_features,training_target)
results = exported_pipeline.predict(testing_features)
```

# CHAPITRE 11
# Sélection de modèle

Ce chapitre s'intéresse à l'optimisation des hyperparamètres. Nous verrons également s'il est nécessaire d'ajouter encore des données pour améliorer le fonctionnement du modèle.

## Courbe de validation

La création d'une courbe de validation est un des outils permettant de trouver la bonne valeur pour un hyperparamètre. Il s'agit d'un graphique montrant la réponse des performances du modèle aux variations de la valeur de l'hyperparamètre (Figure 11.1). Le diagramme montre les données d'entraînement et celles de validation. Les scores de validation permettent de déduire la façon dont le modèle va réagir à des données qu'il ne connaît pas encore. En temps normal, il faut choisir une valeur d'hyperparamètre qui offre le meilleur score de validation.

Dans le prochain exemple, nous profitons de **`Yellowbrick`** pour voir si un changement de `max_depth` influe sur les performances du modèle pour une forêt aléatoire. Il est possible de spécifier un paramètre `scoring` relié à une métrique de modèle **`scikit-learn`**. Pour les classifications, la valeur par défaut est `accuracy` :

Pour accélérer le traitement et utiliser plusieurs processeurs, servez-vous du paramètre `n_jobs`. La valeur -1 fait utiliser toutes les CPU disponibles.

```
>>> from yellowbrick.model_selection import (
...     ValidationCurve,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> vc_viz = ValidationCurve(
...     RandomForestClassifier(n_estimators=100),
...     param_name="max_depth",
```

```
...         param_range=np.arange(1, 11),
...         cv=10,
...         n_jobs=-1,
... )
>>> vc_viz.fit(X, y)
>>> vc_viz.poof()
>>> fig.savefig("images/mlpr_1101.png", dpi=300)
```

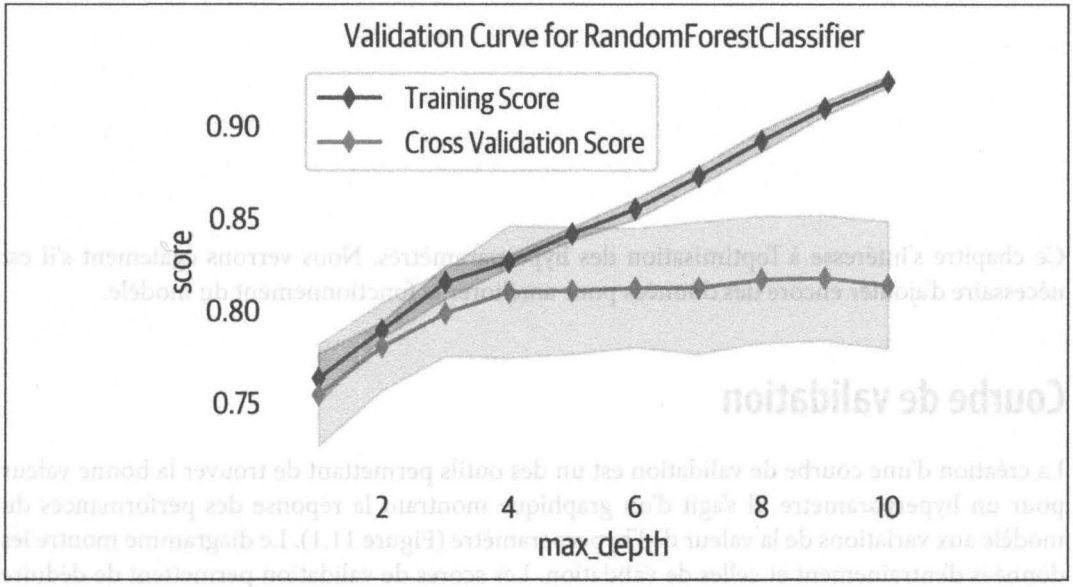



*Figure 11.1 : Rapport de courbe de validation.*

La classe `ValidationCurve` reconnaît le paramètre `scoring` qui peut être selon la tâche à réaliser une fonction spécifique ou une des options suivantes.

Voici les options de `scoring` pour la classification :

```
'accuracy',       'average_precision',
'f1', 'f1_micro', 'f1_macro',    'f1_weighted', 'f1_samples',
'neg_log_loss',   'precision',   'recall',      'roc_auc'.
```

Voici les options de `scoring` pour le clustering :

```
'adjusted_mutual_info_score',   'adjusted_rand_score',
'completeness_score',           'fowlkesmallows_score',
'homogeneity_score',            'mutual_info_score',
'normalized_mutual_info_score', 'v_measure_score'
```

Voici les options de `scoring` pour la régression :

```
'explained_variance',
'neg_mean_absolute_error',    'neg_mean_squared_error';
'neg_mean_squared_log_error', 'neg_median_absolute_error','r2'
```

## Courbe d'apprentissage

Quel volume de données faut-il prévoir pour pouvoir sélectionner le meilleur modèle d'un projet ? La réponse peut être facilitée par une courbe d'apprentissage qui visualise les scores d'entraînement et de validation croisée pendant la création du modèle avec de plus en plus d'échantillons. Si le score de validation croisée continue à augmenter, cela suggère que le modèle bénéficierait de plus de données.

L'exemple de courbe de validation qui suit permet d'étudier le biais et la variance du modèle (Figure 11.2). La zone grisée pour le score d'entraînement correspond à la variabilité. Si elle est grande, cela indique que le modèle souffre d'erreur de biais et qu'il est trop simple, donc sous-ajusté. Si la variabilité concerne le score de validation croisée, c'est que le modèle souffre d'erreur de variance et qu'il trop complexe, surajusté. Le fait que le jeu de test (de validation) montre des performances bien moindres que celles du jeu d'entraînement est un autre indice de surajustement du modèle.

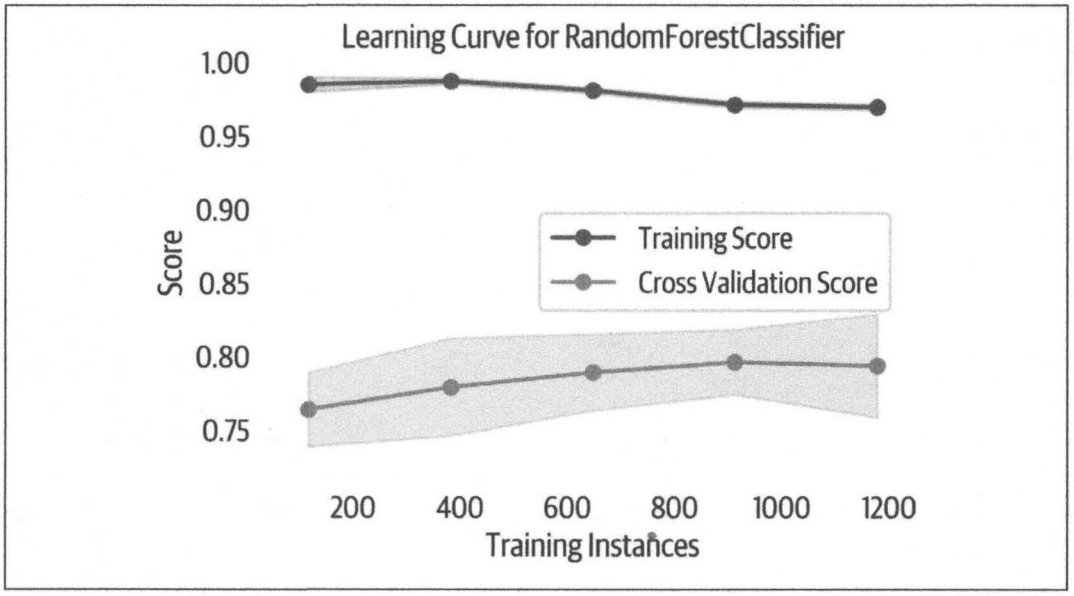



*Figure 11.2 : Diagramme de courbe d'apprentissage. Lorsque le score de validation donne une horizontale, c'est que le modèle ne peut pas être amélioré en ajoutant des données.*

Voici un exemple de création d'une courbe d'apprentissage avec **`Yellowbrick`** :

```
>>> from yellowbrick.model_selection import (
...     LearningCurve,
... )
```

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lc3_viz = LearningCurve(
...     RandomForestClassifier(n_estimators=100),
...     cv=10,
... )
>>> lc3_viz.fit(X, y)
>>> lc3_viz.poof()
>>> fig.savefig("images/mlpr_1102.png", dpi=300)
```

En modifiant les options d'évaluation, la même visualisation peut être utilisée pour une régression ou un regroupement clustering.

CHAPITRE 12

# Métriques et évaluation des classifications

Dans ce chapitre, nous allons nous intéresser à plusieurs techniques de mesure et outils d'évaluation : les matrices de confusion, les principales métriques, les rapports de classification et quelques techniques de visualisation.

Ces différentes techniques seront mises en pratique pour évaluer un modèle d'arbre de décision servant à prédire la survie à bord du *Titanic*.

## Matrices de confusion

Les matrices de confusion servent à évaluer les performances d'un classifieur.

Un classifieur binaire ne permet que quatre résultats de classification : vrais positifs TP (*True Positives*), vrais négatifs (*True Negatives*), faux positifs FP (*False Positives*) et faux négatifs FN (*False Negatives*). Seuls les deux premiers correspondent à des classifications correctes, donc vraies.

Voici quelques éclaircissements sur les deux autres résultats possibles. Nous supposons qu'un résultat positif signifie « être enceinte » et qu'un résultat négatif signifie ne pas l'être. Un faux positif revient à prétendre qu'un homme est enceint et un faux négatif qu'une femme n'est pas enceinte alors qu'elle l'est en réalité (Figure 12.1). Ces deux résultats qui sont des erreurs correspondent respectivement aux erreurs de type 1 et de type 2 (Tableau 12.1).

Une astuce mnémotechnique consiste à se souvenir que la lettre P de faux positif n'a qu'un trait vertical (donc type 1), alors que la lettre N de faux négatif comporte deux traits verticaux (donc type 2).

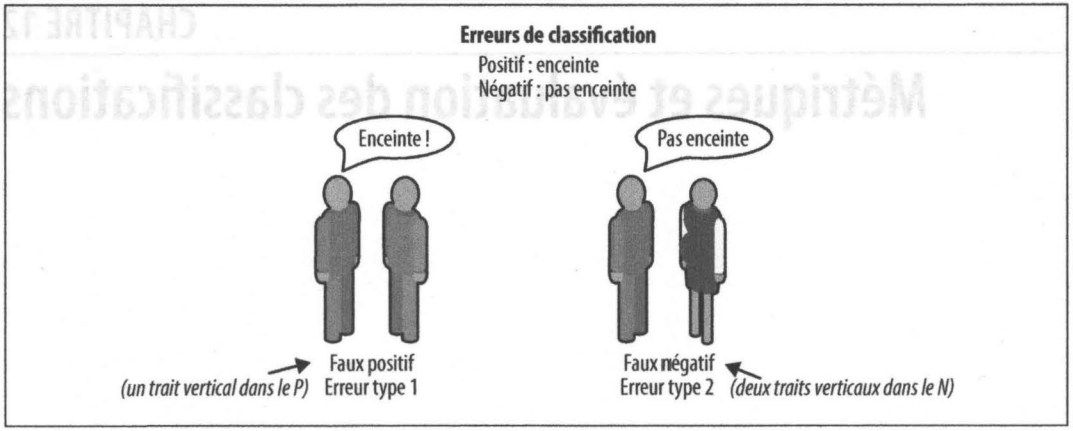



*Figure 12.1 : Erreurs de classification.*

*Tableau 12.1 : Résultat d'une classification binaire par matrice de confusion.*

| Réalité | Prédiction négative | Prédiction positive |
|---|---|---|
| Négatif réel | Vrai négatif | Faux positif (type 1) |
| Positif réel | Faux négatif (type 2) | Vrai positif |

Voici le code source utilisant **pandas** pour calculer ces résultats de classification. Nous allons nous servir dans les exemples suivants des mêmes variables :

```
>>> y_predict = dt.predict(X_test)
>>> tp = (
...     (y_test == 1) & (y_test == y_predict)
... ).sum() # 123
>>> tn = (
...     (y_test == 0) & (y_test == y_predict)
... ).sum() # 199
>>> fp = (
...     (y_test == 0) & (y_test != y_predict)
... ).sum() # 25
>>> fn = (
...     (y_test == 1) & (y_test != y_predict)
... ).sum() # 46
```

Un classifieur de bonne qualité montre un grand nombre de valeurs dans la diagonale reliant vrais positifs et vrais négatifs. Voici comment créer une structure `DataFrame` avec la fonction de **`sklearn`** nommée `confusion_matrix` :

```
>>> from sklearn.metrics import confusion_matrix
>>> y_predict = dt.predict(X_test)
>>> pd.DataFrame(
...     confusion_matrix(y_test, y_predict),
...     columns=[
```

```
...         "Predict died",
...         "Predict Survive",
...     ],
...     index=["True Death", "True Survive"],
... )
              Predict died   Predict Survive
True Death             199                25
True Survive            46               123
```

Avec **`Yellowbrick`**, nous pouvons créer un diagramme à partir de cette matrice (Figure 12.2):

```
>>> import matplotlib.pyplot as plt
>>> from yellowbrick.classifier import (
...     ConfusionMatrix,
... )
>>> mapping = {0: "died", 1: "survived"}
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cm_viz = ConfusionMatrix(
...     dt,
...     classes=["died", "survived"],
...     label_encoder=mapping,
... )
>>> cm_viz.score(X_test, y_test)
>>> cm_viz.poof()
>>> fig.savefig("images/mlpr_1202.png", dpi=300)
```

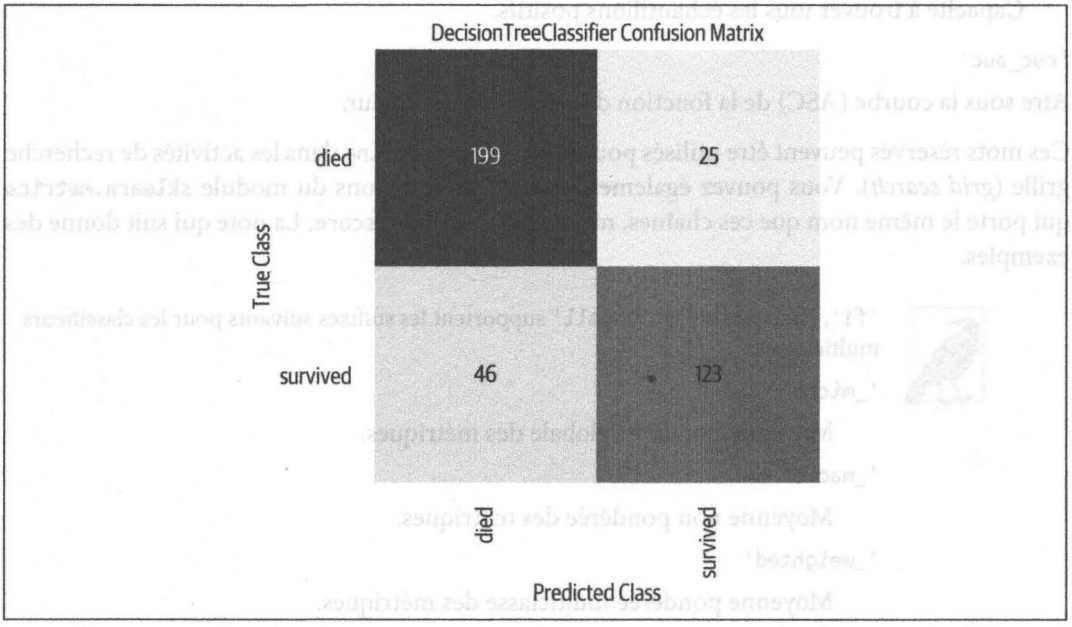



*Figure 12.2 : Matrice de confusion. Les deux résultats corrects correspondent aux carrés supérieur gauche et inférieur droit, les faux négatifs en bas à gauche et les faux positifs en haut à droite.*

## Métriques

Le module nommé **`sklearn.metrics`** regroupe la plupart des métriques principales, et notamment :

`'accuracy'`

Pourcentage de prédictions correctes (positives ou négatives).

`'average_precision'`

Synthèse de la courbe précision/rappel.

`'f1'`

Moyenne harmonique de précision et rappel.

`'neg_log_loss'`

Perte logistique ou cross-entropie (le modèle doit accepter `predict_proba`).

`'precision'`

Capacité à ne trouver que les échantillons corrects (à ne pas prédire comme positif un négatif).

`'recall'`

Capacité à trouver tous les échantillons positifs.

`'roc_auc'`

Aire sous la courbe (ASC) de la fonction d'efficacité du récepteur.

Ces mots réservés peuvent être utilisés pour le paramètre `scoring` dans les activités de recherche grille (*grid search*). Vous pouvez également utiliser les fonctions du module **`sklearn.metrics`** qui porte le même nom que ces chaînes, mais se termine par `_score`. La note qui suit donne des exemples.

`'f1'`, `'precision'` et `'recall'` supportent les suffixes suivants pour les classifieurs multiclasses :

`'_micro'`

Moyenne pondérée globale des métriques.

`'_macro'`

Moyenne non pondérée des métriques.

`'_weighted'`

Moyenne pondérée multiclasse des métriques.

`'_samples'`

Métrique par échantillon.

## Exactitude (*accuracy*)

L'exactitude (qu'il ne faut pas confondre avec la précision) correspond au pourcentage de classifications correctes :

```
>>> (tp + tn) / (tp + tn + fp + fn)
0.8142493638676844
```

L'exactitude n'est pas tout. Si le projet consiste à prédire des cas de fraude (qui sont des événements rares, par exemple 1 sur 10 000), vous obtiendrez aisément une grande exactitude en prédisant systématiquement qu'il n'y a aucune fraude, mais cela ne vous aide nullement. Il faut vous intéresser à d'autres métriques et évaluer le coût de prédiction d'un faux positif et d'un faux négatif, avant de décider qu'un modèle est acceptable.

Voici comment calculer l'exactitude avec **`sklearn`** :

```
>>> from sklearn.metrics import accuracy_score
>>> y_predict = dt.predict(X_test)
>>> accuracy_score(y_test, y_predict)
0.8142493638676844
```

## Rappel (*recall*)

La mesure de rappel (ou sensibilité) correspond au pourcentage de valeurs positives qui le sont réellement. Autrement dit, c'est le nombre de résultats pertinents renvoyé :

```
>>> tp / (tp + fn)
0.7159763313609467

>>> from sklearn.metrics import recall_score
>>> y_predict = dt.predict(X_test)
>>> recall_score(y_test, y_predict)
0.7159763313609467
```

## Précision

La précision mesure le pourcentage de prédictions positives qui sont réellement positives, soit `TP / (TP + FP)`. C'est le degré de pertinence des résultats.

```
>>> tp / (tp + fp)
0.8287671232876712

>>> from sklearn.metrics import precision_score
>>> y_predict = dt.predict(X_test)
>>> precision_score(y_test, y_predict)
0.8287671232876712
```

# f1

Cette métrique correspond à la moyenne harmonique du rappel et de la précision :

```
>>> pre = tp / (tp + fp)
>>> rec = tp / (tp + fn)
>>> (2 * pre * rec) / (pre + rec)
0.7682539682539683
```

```
>>> from sklearn.metrics import f1_score
>>> y_predict = dt.predict(X_test)
>>> f1_score(y_test, y_predict)
0.7682539682539683
```

# Rapports de classification

La librairie **`Yellowbrick`** propose un rapport de classification qui permet de voir la précision, le rappel et la mesure f1 pour les valeurs positives et négatives (Figure 12.3). La Figure est en couleur ; plus une cellule est rouge, plus elle est proche de un et plus le score est élevé :

```
>>> import matplotlib.pyplot as plt
>>> from yellowbrick.classifier import (
...     ClassificationReport,
... )
>>> fig, ax = plt.subplots(figsize=(6, 3))
>>> cm_viz = ClassificationReport(
...     dt,
...     classes=["died", "survived"],
...     label_encoder=mapping,
... )
>>> cm_viz.score(X_test, y_test)
>>> cm_viz.poof()
>>> fig.savefig("images/mlpr_1203.png", dpi=300)
```

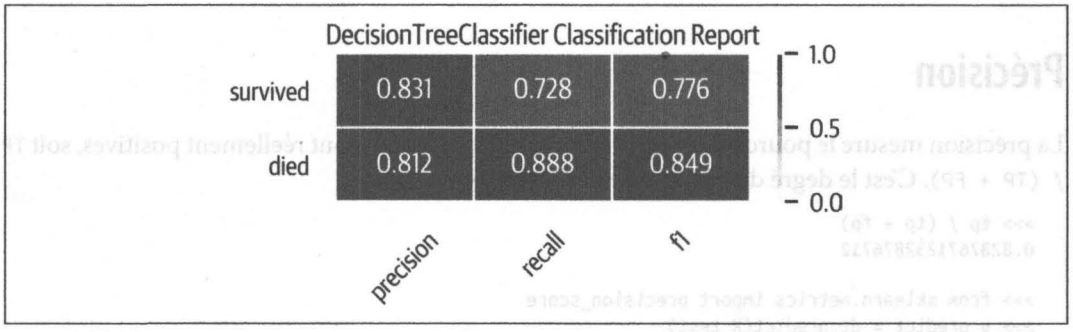



*Figure 12.3 : Rapport de classification.*

# Courbe ROC

Ce que l'on appelle courbe ROC permet de voir les performances d'un classifieur en s'intéressant au taux de vrais positifs (rappel/sensibilité) en fonction des variations du taux de faux positifs (spécificité inversée), comme le montre la Figure 12.4.

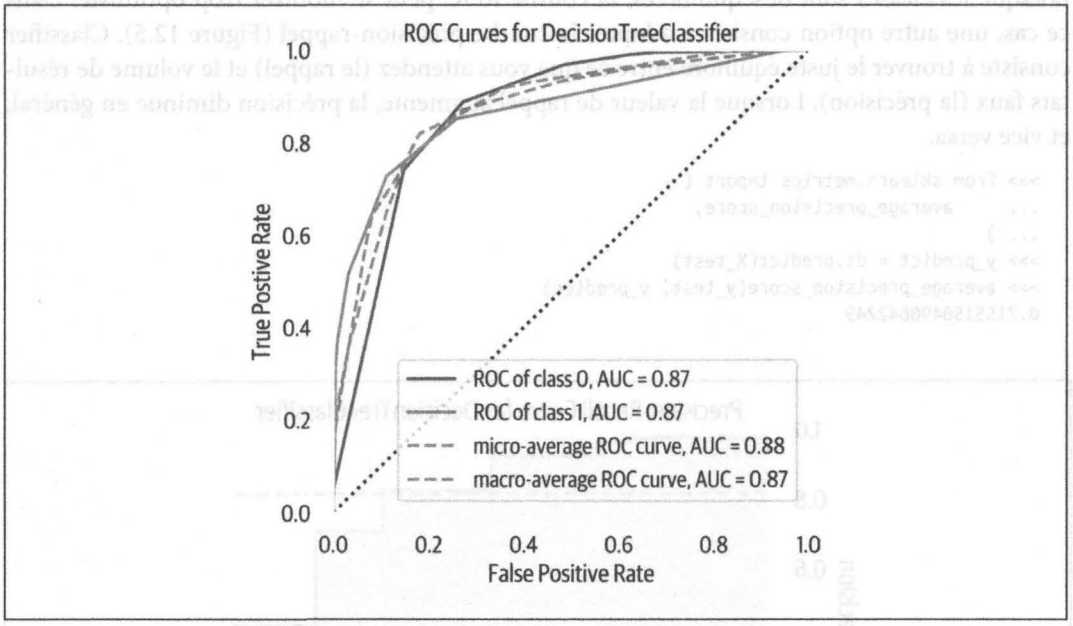



*Figure 12.4 : Courbe ROC.*

En règle générale, la courbe doit être attirée vers le coin supérieur gauche. Chaque point situé à gauche et au-dessus d'un autre correspond à des performances meilleures. La ligne en diagonale rappelle les performances d'un classifieur purement aléatoire. Vous pouvez mesurer les performances en étudiant l'aire sous cette courbe, AUC :

```
>>> from sklearn.metrics import roc_auc_score
>>> y_predict = dt.predict(X_test)
>>> roc_auc_score(y_test, y_predict)
0.8706304346418559
```

Voici une visualisation de cette courbe avec **`Yellowbrick`** :

```
>>> from yellowbrick.classifier import ROCAUC
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> roc_viz = ROCAUC(dt)
>>> roc_viz.score(X_test, y_test)
0.8706304346418559
```

```
>>> roc_viz.poof()
>>> fig.savefig("images/mlpr_1204.png", dpi=300)
```

## Courbe précision-rappel

Lorsque les classes sont déséquilibrées, la courbe ROC peut se montrer trop optimiste. Dans ce cas, une autre option consiste à adopter la courbe précision-rappel (Figure 12.5). Classifier consiste à trouver le juste équilibre entre ce que vous attendez (le rappel) et le volume de résultats faux (la précision). Lorsque la valeur de rappel augmente, la précision diminue en général, et vice versa.

```
>>> from sklearn.metrics import (
...     average_precision_score,
... )
>>> y_predict = dt.predict(X_test)
>>> average_precision_score(y_test, y_predict)
0.7155150490642249
```

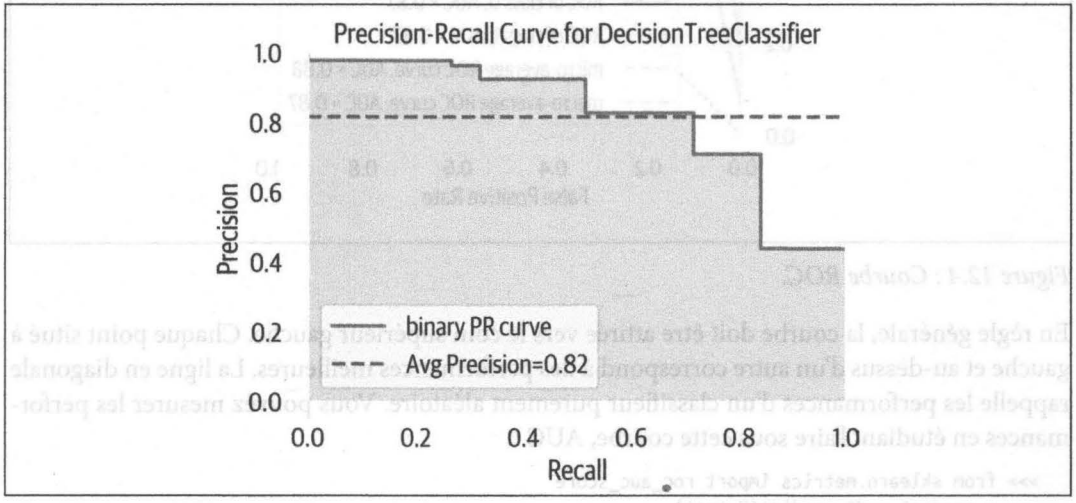



*Figure 12.5 : Courbe précision-rappel.*

Voici la visualisation **`Yellowbrick`** d'une telle courbe :

```
>>> from yellowbrick.classifier import (
...     PrecisionRecallCurve,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> viz = PrecisionRecallCurve(
...     DecisionTreeClassifier(max_depth=3)
... )
```

```
>>> viz.fit(X_train, y_train)
>>> print(viz.score(X_test, y_test))
>>> viz.poof()
>>> fig.savefig("images/mlpr_1205.png", dpi=300)
```

# Diagramme de gains cumulés

Un diagramme de gains cumulés permet d'évaluer la qualité d'un classifieur binaire en confrontant le taux de vrais positifs (sensibilité) au taux de support (la fraction de prédictions positives). L'objectif final est de trier les classifications selon leurs probabilités de prédiction. En théorie, il devrait être possible de constater une frontière nette entre échantillons positifs et négatifs. Si les premiers 10 % de prédiction englobent 30 % des échantillons positifs, il suffit de tirer une ligne entre (0,0) et (0.1, 0.3). Ce processus est ensuite répété pour les autres échantillons (Figure 12.6).

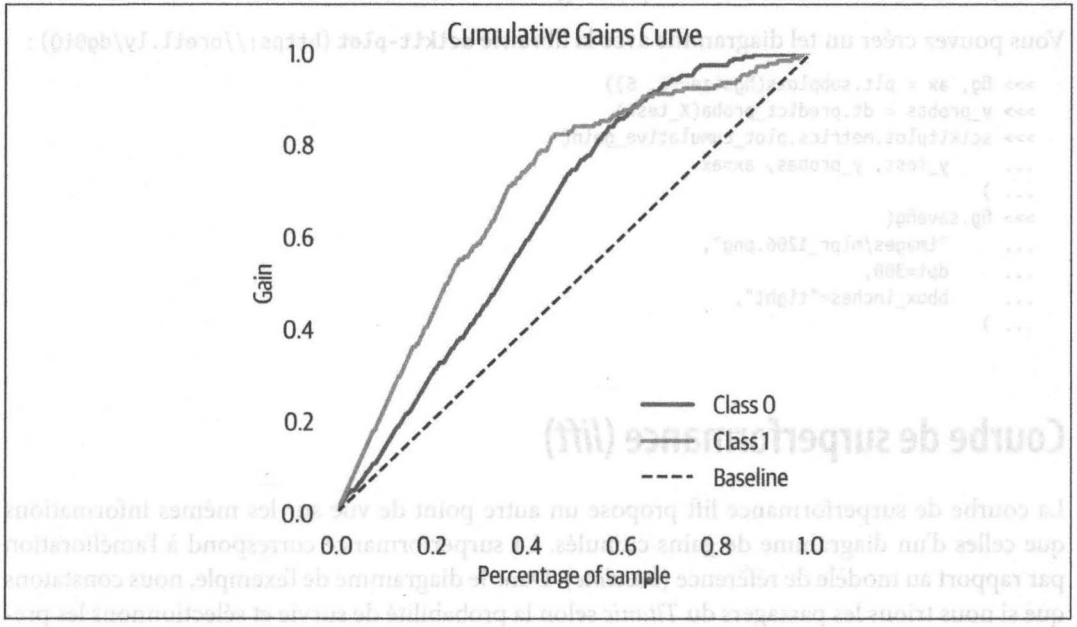



*Figure 12.6 : Diagramme de gains cumulés. Si nous trions les passagers du **Titanic** selon ce modèle, nous trouvons 40 % des survivants avec seulement 20 % des échantillons.*

Ce genre de diagramme est souvent utilisé pour décider d'une stratégie de réponse à des clients. Le diagramme de gains cumulés montre le taux de positifs prédits sur l'axe x. Dans notre figure, cela correspond au pourcentage d'échantillons. La sensibilité ou taux de vrais positifs correspond à l'axe des ordonnées y avec le label `Gain` dans la figure.

Si vous décidez de prendre contact avec 90 % des clients qui répondraient (sensibilité), il suffit de tracer un trait horizontal depuis la position .9 de l'axe y jusqu'à rencontrer la courbe. Le point de l'axe x que vous touchez permet de connaître le nombre de clients à contacter pour obtenir ces 90 % de taux de support.

Dans notre exemple, nous ne cherchons pas à contacter des clients, mais à prédire la survie à bord du *Titanic*. Si nous demandons de trier les passagers selon notre modèle en fonction de leurs chances de survivre, nous trouverions 90 % de survivants avec seulement 65 % d'échantillons. Dans une campagne de prospection commerciale, si vous connaissez le coût pour chaque contact et le chiffre d'affaires moyen prévu par réponse, vous pouvez décider finement de la meilleure valeur à sélectionner.

En règle générale, un modèle est meilleur s'il se situe à gauche et au-dessus d'un autre. Les modèles parfaits ou presque sont ceux qui montent jusqu'en haut, puis bifurquent vers la droite (si 10 % des échantillons sont positifs, le trait monterait jusqu'en (0.1, 1)) puis irait directement à droite). Si la courbe est moins bonne que la ligne de référence, autant choisir les labels au hasard.

Vous pouvez créer un tel diagramme avec la librairie **`scikit-plot`** (`https://oreil.ly/dg0iQ`) :

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> y_probas = dt.predict_proba(X_test)
>>> scikitplot.metrics.plot_cumulative_gain(
...     y_test, y_probas, ax=ax
... )
>>> fig.savefig(
...     "images/mlpr_1206.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

## Courbe de surperformance (*lift*)

La courbe de surperformance lift propose un autre point de vue sur les mêmes informations que celles d'un diagramme de gains cumulés. La surperformance correspond à l'amélioration par rapport au modèle de référence (*baseline*). Dans le diagramme de l'exemple, nous constatons que si nous trions les passagers du *Titanic* selon la probabilité de survie et sélectionnons les premiers 20 %, nous profitons d'une surperformance d'environ 2,2 fois par rapport à une sélection au hasard (gain divisé par pourcentage d'échantillons). La Figure 12.7 montre que nous obtiendrons 2,2 fois plus de survivants. Voici comment créer une courbe d'élévation avec la librairie **`scikit-plot`** :

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> y_probas = dt.predict_proba(X_test)
>>> scikitplot.metrics.plot_lift_curve(
...     y_test, y_probas, ax=ax
... )
```

```
>>> fig.savefig(
...     "images/mlpr_1207.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

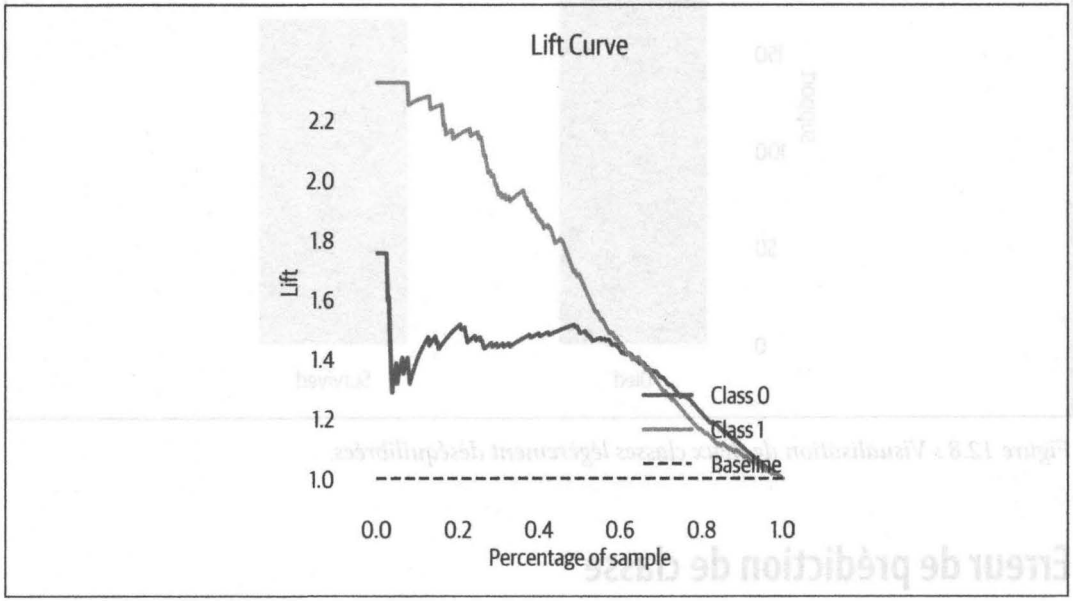



*Figure 12.7 : Courbe de surperformance.*

# Équilibre des classes (*balance*)

**`Yellowbrick`** propose un diagramme en barres verticales pour comparer les tailles des classes. En effet, lorsque les tailles sont assez différentes, l'exactitude n'est plus une métrique très utile (Figure 12.8). Pour la répartition des données entre jeu d'entraînement et jeu de test, il faut utiliser un échantillonnage stratifié pour que les deux jeux conservent les mêmes proportions entre classes. (La fonction `test_train_split` s'en charge en mettant en correspondance le paramètre `stratify` et les labels.)

```
>>> from yellowbrick.classifier import ClassBalance
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cb_viz = ClassBalance(
...     labels=["Died", "Survived"]
... )
>>> cb_viz.fit(y_test)
>>> cb_viz.poof()
>>> fig.savefig("images/mlpr_1208.png", dpi=300)
```

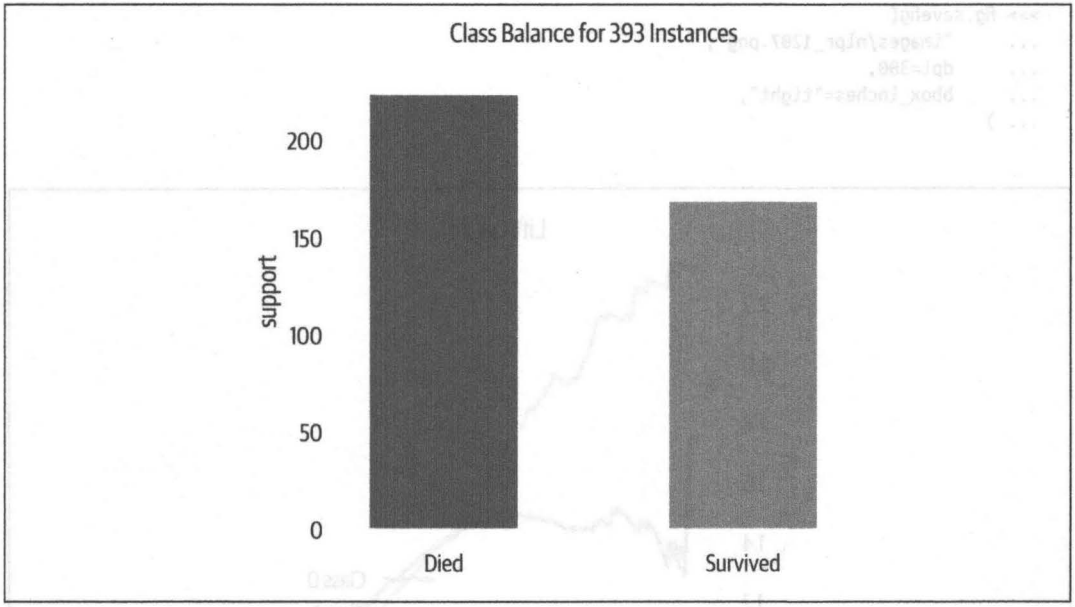



*Figure 12.8 : Visualisation de deux classes légèrement déséquilibrées.*

# Erreur de prédiction de classe

Le diagramme d'erreur de prédiction de classe de **`Yellowbrick`** est un histogramme qui montre en fait une matrice de confusion (Figure 12.9).

```
>>> from yellowbrick.classifier import (
...     ClassPredictionError,
... )
>>> fig, ax = plt.subplots(figsize=(6, 3))
>>> cpe_viz = ClassPredictionError(
...     dt, classes=["died", "survived"]
...)
>>> cpe_viz.score(X_test, y_test)
>>> cpe_viz.poof()
>>> fig.savefig("images/mlpr_1209.png", dpi=300)
```

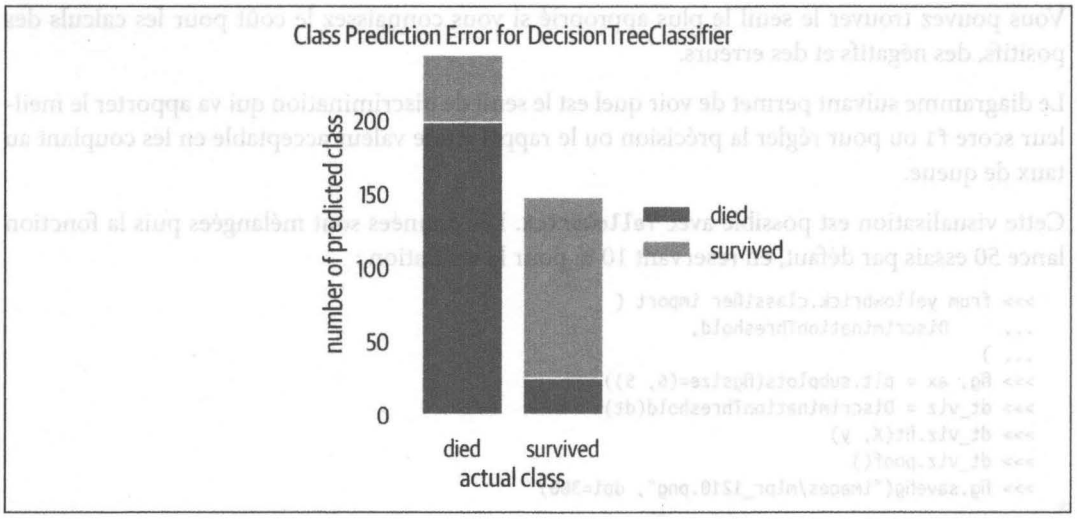



*Figure 12.9 : Visualisation des erreurs de prédiction de classe. La partie supérieure de la barre de gauche correspond aux personnes décédées que nous avions prétendues survivantes, donc des faux positifs. Le bas de la barre de droite correspond aux survivants que nous avions prétendus avoir péri, donc des faux négatifs.*

## Seuil de discrimination

La plupart des classifieurs binaires qui prédisent des probabilités travaillent avec un seuil de discrimination de 50 %. Autrement dit, le classifieur attribue un label positif dès que la probabilité prédite est supérieure à 50 %. Dans la Figure 12.10, le seuil varie entre 0 et 100, ce qui permet de montrer l'impact sur la précision, le rappel, la moyenne et le taux de queue.

Le *taux de queue* correspond au pourcentage de prédiction supérieur au seuil. C'est le pourcentage de cas qu'il faut voir en détail, par exemple, lorsqu'il s'agit de traquer des fraudes.

Le diagramme permet de visualiser l'arbitrage réalisé entre précision et rappel. Si nous recherchons des fraudes, et décidons que la fraude est une classe positive, pour obtenir un bon rappel, c'est-à-dire trouver toutes les fraudes, nous pourrions tout classer comme fraude. Dans une situation bancaire réelle, cela ne serait pas praticable car cela réclamerait un travail de contrôle énorme. Pour atteindre une bonne précision, c'est-à-dire ne détecter comme fraude que ce qui l'est vraiment, nous avons besoin d'un modèle qui ne se déclenche qu'en cas de fraude extrême. Le problème est que des cas de fraude moins évidents vont alors passer au travers. Il y a donc bien un arbitrage à réaliser.

Vous pouvez trouver le seuil le plus approprié si vous connaissez le coût pour les calculs des positifs, des négatifs et des erreurs.

Le diagramme suivant permet de voir quel est le seuil de discrimination qui va apporter le meilleur score `f1` ou pour régler la précision ou le rappel à une valeur acceptable en les couplant au taux de queue.

Cette visualisation est possible avec **`Yellowbrick`**. Les données sont mélangées puis la fonction lance 50 essais par défaut, en réservant 10 % pour la validation :

```
>>> from yellowbrick.classifier import (
...     DiscriminationThreshold,
... )
>>> fig, ax = plt.subplots(figsize=(6, 5))
>>> dt_viz = DiscriminationThreshold(dt)
>>> dt_viz.fit(X, y)
>>> dt_viz.poof()
>>> fig.savefig("images/mlpr_1210.png", dpi=300)
```

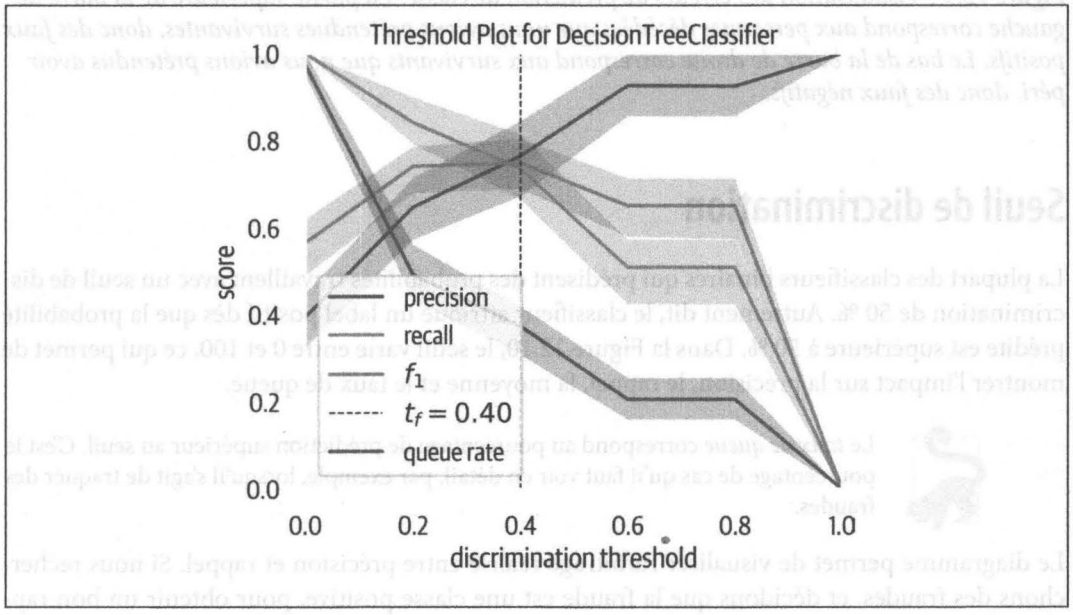



*Figure 12.10 : Visualisation d'un seuil de discrimination.*

CHAPITRE 13

# Explication des modèles

Chaque modèle prédictif a des propriétés spécifiques. Certains sont consacrés à la gestion de données linéaires, alors que d'autres s'adaptent mieux à des données d'entrée complexes. Certains modèles sont faciles à interpréter, alors que d'autres sont des boîtes noires qui ne permettent pas de savoir exactement comment leurs prédictions sont générées.

Voyons donc comment interpréter différents modèles. Nous nous servirons en guise d'exemple des données du jeu Titanic :

```
>>> dt = DecisionTreeClassifier(
...     random_state=42, max_depth=3
... )
>>> dt.fit(X_train, y_train)
```

## Coefficient de régression

Le coefficient de régression et celui d'interception permettent de comprendre la valeur espérée et de voir en quoi les caractéristiques impactent la prédiction. Lorsque le coefficient est positif, si la valeur de la caractéristique augmente, la prédiction augmente aussi.

## Importance des caractéristiques

Les modèles basés arbre dans la librairie **`scikit-learn`** disposent d'un attribut nommé `.feature_importances_` qui sert à inspecter la façon dont les caractéristiques d'un jeu de données impactent un modèle. Nous pouvons les inspecter et en produire une visualisation graphique.

## LIME

LIME (`https://oreil.ly/shCR`) permet d'en apprendre plus sur les modèles de type boîte noire en réalisant une interprétation locale et non pas globale. Cela permet notamment d'expliquer un échantillon isolé.

L'approche LIME permet pour un point de donnée ou un échantillon de découvrir quelles caractéristiques ont été prépondérantes dans le résultat. La technique consiste à perturber l'échantillon et à créer un ajustement de modèle linéaire approprié qui constitue une approximation du modèle proche de l'échantillon (Figure 13.1).

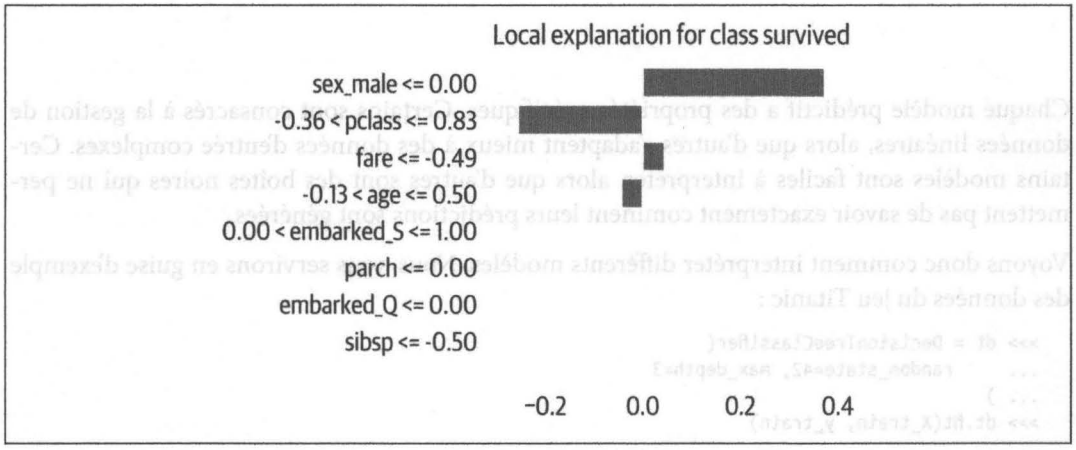



*Figure 13.1 : Explication LIME pour le jeu Titanic. Les caractéristiques concernant l'échantillon déplacent la prédiction vers la droite pour la survie ou vers la gauche pour le décès.*

Voici un exemple qui explique le dernier échantillon du jeu d'entraînement, celui que notre prédiction considère comme devant survivre :

```
>>> from lime import lime_tabular
>>> explainer = lime_tabular.LimeTabularExplainer(
...     X_train.values,
...     feature_names=X.columns,
...     class_names=["died", "survived"],
... )
>>> exp = explainer.explain_instance(
...     X_train.iloc[-1].values, dt.predict_proba
...)
```

Vous devrez éventuellement installer cette librairie ainsi :

```
pip install lime
```

Notez que LIME n'apprécie pas les structures `DataFrame` en entrée. C'est la raison pour laquelle nous avons converti les données vers des tableaux **`numpy`** avec `.values`.

Si vous faites l'exercice dans Jupyter, vous devez ajouter cette instruction :

```
exp.show_in_notebook()
```

Vous obtiendrez ainsi une version HTML de l'explication.

Vous pouvez créer une figure **`matplotlib`** si vous avez besoin d'exporter les explications (ou si vous n'utilisez pas Jupyter) :

```
>>> fig = exp.as_pyplot_figure()
>>> fig.tight_layout()
>>> fig.savefig("images/mlpr_1301.png")
```

Si vous faites des essais, vous verrez que vous allez impacter les résultats si vous intervertissez les sexes. Voyons cela avec l'avant-dernière ligne du jeu d'entraînement. Sa prédiction est de 48 % de décès et de 52 % de survie. Si nous échangeons les sexes, la prédiction monte à 88 % de décès :

```
>>> data = X_train.iloc[-2].values.copy()
>>> dt.predict_proba(
...     [data]
... ) # Prédiction de survie de femme
[[0.48062016 0.51937984]]

>>> data[5] = 1 # Change en homme
>>> dt.predict_proba([data])
array([[0.87954545, 0.12045455]])
```

La méthode `.predict_proba` renvoie une probabilité pour chaque label.

# Interprétation d'un arbre

Avec les modèles arborescents de **`sklearn`** (arbres de décision, forêts aléatoires et autres modèles en arbre), vous pouvez profiter du paquetage nommé **`treeinterpreter`** (`https://oreil.ly/vN1Bl`). Il permet de connaître le biais et la contribution de chaque caractéristique. Rappelons que le biais est la moyenne pour le jeu d'entraînement.

Chaque contribution montre l'impact sur chacun des labels. (L'addition du biais et des contributions doit donner la valeur de la prédiction.) Il n'y a que deux valeurs puisqu'il s'agit d'une

classification binaire. Nous voyons que la caractéristique essentielle est *sex_male*, suivie de *age* puis de *fare* :

```
>>> from treeinterpreter import (
...     treeinterpreter as ti,
... )
>>> instances = X.iloc[:2]
>>> prediction, bias, contribs = ti.predict(
...     rf5, instances
... )
>>> i = 0
>>> print("Instance", i)
>>> print("Prediction", prediction[i])
>>> print("Bias (trainset mean)", bias[i])
>>> print("Contributions:")
>>> for c, feature in zip(
...     contribs[i], instances.columns
... ):
...     print(" {} {}".format(feature, c))
Instance 0
Prediction [0.98571429 0.01428571]
Bias (trainset mean) [0.63984716 0.36015284]
Contributions:
pclass [ 0.03588478 -0.03588478]
age [ 0.08569306 -0.08569306]
sibsp [ 0.01024538 -0.01024538]
parch [ 0.0100742 -0.0100742]
fare [ 0.06850243 -0.06850243]
sex_male [ 0.12000073 -0.12000073]
embarked_Q [ 0.0026364 -0.0026364]
embarked_S [ 0.01283015 -0.01283015]
```

Cet exemple concerne la classification, mais le module supporte aussi la régression.

## Diagrammes de dépendance partielle

La mesure d'importance des caractéristiques dans les arbres permet de détecter qu'une caractéristique a un impact, mais cela ne suffit pas à savoir dans quelle mesure cet impact varie en fonction du changement de valeur de la caractéristique. Un diagramme de dépendance partielle permet de connaître les relations entre les changements d'une caractéristique et le résultat. Nous nous servons du module **pdpbox** (`https://oreil.ly/09zY2`) pour voir par exemple comment l'âge affecte le taux de survie (Figure 13.2).

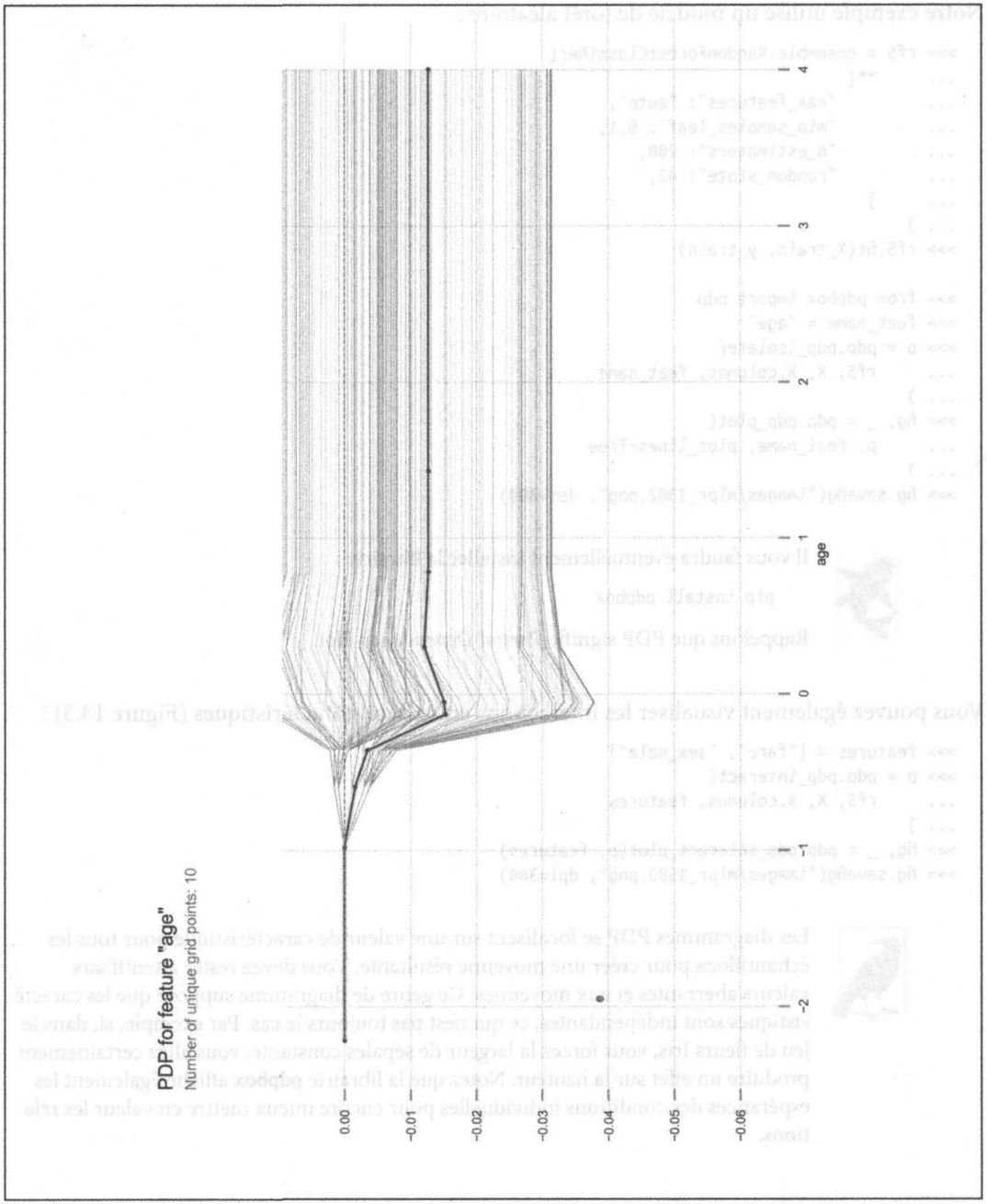



*Figure 13.2 : Diagramme de dépendance partielle montrant l'impact sur la cible des changements de l'âge.*

Notre exemple utilise un modèle de forêt aléatoire :

```
>>> rf5 = ensemble.RandomForestClassifier(
...     **{
...         "max_features": "auto",
...         "min_samples_leaf": 0.1,
...         "n_estimators": 200,
...         "random_state": 42,
...     }
... )
>>> rf5.fit(X_train, y_train)

>>> from pdpbox import pdp
>>> feat_name = "age"
>>> p = pdp.pdp_isolate(
...     rf5, X, X.columns, feat_name
... )
>>> fig, _ = pdp.pdp_plot(
...     p, feat_name, plot_lines=True
... )
>>> fig.savefig("images/mlpr_1302.png", dpi=300)
```

Il vous faudra éventuellement installer la librairie :

```
pip install pdpbox
```

Rappelons que PDP signifie *Partial Dependency Plot.*

Vous pouvez également visualiser les interactions entre deux caractéristiques (Figure 13.3) :

```
>>> features = ["fare", "sex_male"]
>>> p = pdp.pdp_interact(
...     rf5, X, X.columns, features
... )
>>> fig, _ = pdp.pdp_interact_plot(p, features)
>>> fig.savefig("images/mlpr_1303.png", dpi=300)
```

Les diagrammes PDP se focalisent sur une valeur de caractéristique pour tous les échantillons pour créer une moyenne résultante. Vous devez rester attentif aux valeurs aberrantes et aux moyennes. Ce genre de diagramme suppose que les caractéristiques sont indépendantes, ce qui n'est pas toujours le cas. Par exemple, si, dans le jeu de fleurs Iris, vous forcez la largeur de sépales constante, vous allez certainement produire un effet sur la hauteur. Notez que la librairie **pdpbox** affiche également les espérances des conditions individuelles pour encore mieux mettre en valeur les relations.

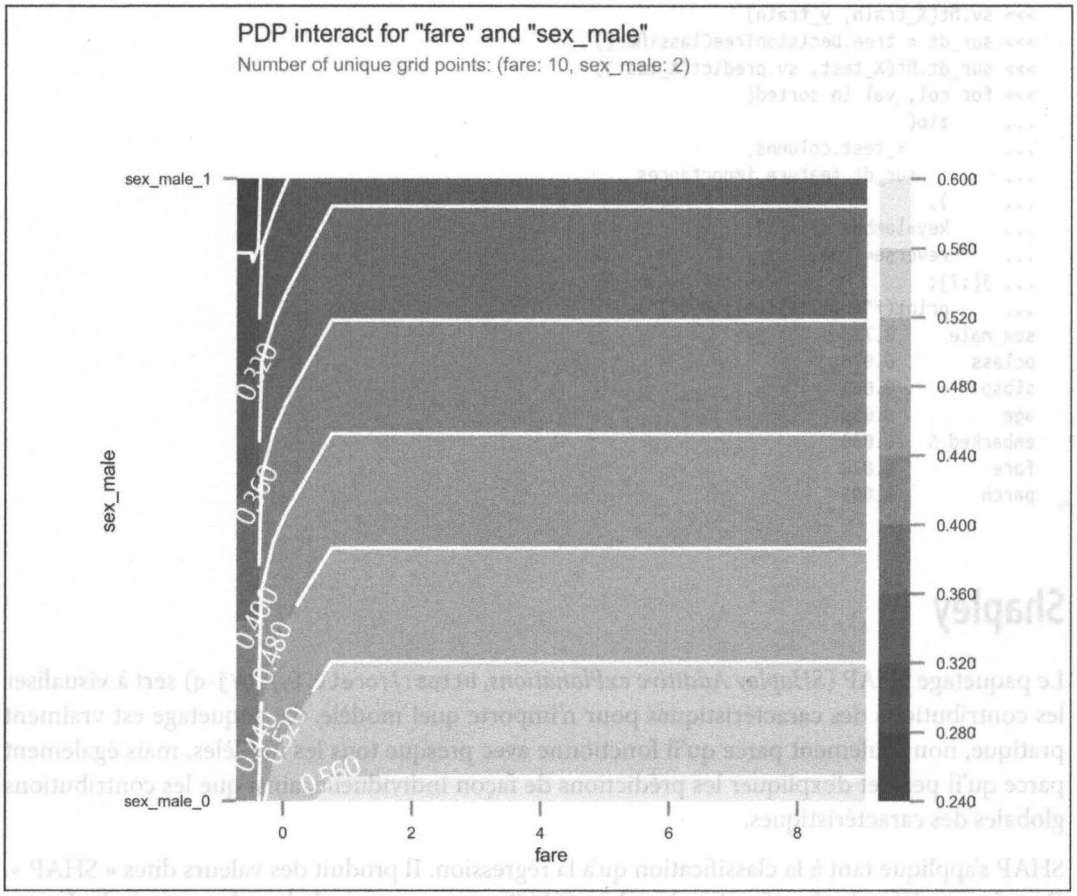



*Figure 13.3 : Diagramme de dépendance partielle avec deux caractéristiques. Quand fare augmente et que sex passent de male à female, le taux de survie augmente.*

# Modèles substituts

Certains modèles ne peuvent pas être interprétés, par exemple SVM et les réseaux neuronaux. Vous pouvez néanmoins ajuster un modèle interprétable (un arbre de décision) à ce genre de modèle. Ce substitut permet ensuite d'examiner l'importance des caractéristiques.

Voyons comment créer un modèle SVC (*Support Vector Classifier*), tout en faisant l'entraînement avec un arbre de décision. Nous ne limitons pas le surajustement pour pouvoir bien capturer ce qui se passe dans le modèle. Cela nous permet d'expliquer le modèle SVC :

```
>>> from sklearn import svm
>>> sv = svm.SVC()
```

```
>>> sv.fit(X_train, y_train)
>>> sur_dt = tree.DecisionTreeClassifier()
>>> sur_dt.fit(X_test, sv.predict(X_test))
>>> for col, val in sorted(
...     zip(
...         X_test.columns,
...         sur_dt.feature_importances_,
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:7]:
...     print(f"{col:10}{val:10.3f}")
sex_male     0.723
pclass       0.076
sibsp        0.061
age          0.056
embarked_S   0.050
fare         0.028
parch        0.005
```

# Shapley

Le paquetage SHAP (*SHapley Additive exPlanations*, `https://oreil.ly/QYj-q`) sert à visualiser les contributions des caractéristiques pour n'importe quel modèle. Ce paquetage est vraiment pratique, non seulement parce qu'il fonctionne avec presque tous les modèles, mais également parce qu'il permet d'expliquer les prédictions de façon individuelle, ainsi que les contributions globales des caractéristiques.

SHAP s'applique tant à la classification qu'à la régression. Il produit des valeurs dites « SHAP ». Dans les modèles de classification, la valeur SHAP est une somme de *logs de rapports de chances (odds)* pour classification binaire. Pour les modèles de régression, les valeurs SHAP donnent une somme qui est la prédiction cible.

Pour certains diagrammes, vous devez utiliser Jupyter pour l'interactivité avec JavaScript. (Certains diagrammes peuvent créer une image statique avec **`matplotlib`**.) Voici en guise d'exemple l'échantillon 20 que nous avons prédit comme devant périr[*] :

```
>>> rf5.predict_proba(X_test.iloc[[20]])
array([[0.59223553, 0.40776447]])
```

Dans le diagramme de force de l'échantillon 20 (Figure 13.4), nous voyons la valeur de base. Elle correspond à une femme dont nous avons prédit le décès. Nous allons utiliser la valeur 1 pour l'index de survie, ce qui permet de faire correspondre le côté droit à la survie. Les caractéristiques vont décaler la valeur vers la gauche ou vers la droite. Plus la caractéristique est grande, plus elle aura d'impact. Dans notre exemple, un faible prix de billet *`fare`* et le choix de la troisième classe poussent en direction du décès (la valeur de sortie est inférieure à 0.5) :

```
>>> import shap
>>> s = shap.TreeExplainer(rf5)
>>> shap_vals = s.shap_values(X_test)
>>> target_idx = 1
>>> shap.force_plot(
...     s.expected_value[target_idx],
...     shap_vals[target_idx][20, :],
...     feature_names=X_test.columns,
... )
```

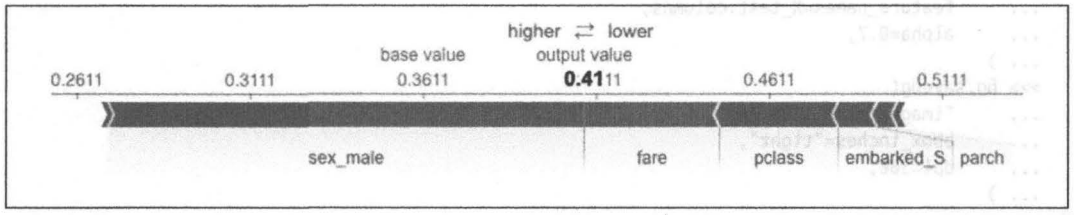



*Figure 13.4 : Contribution des caractéristiques selon Shapley pour l'échantillon 20. Le diagramme montre la valeur de base et les caractéristiques qui tirent la valeur vers la gauche (vers le décès).*

Nous pouvons aussi visualiser les explications pour la totalité du jeu de données en les tournant de 90 ° pour les présenter le long de l'axe x (Figure 13.5) :

```
>>> shap.force_plot(
...     s.expected_value[1],
...     shap_vals[1],
...     feature_names=X_test.columns,
... )
```

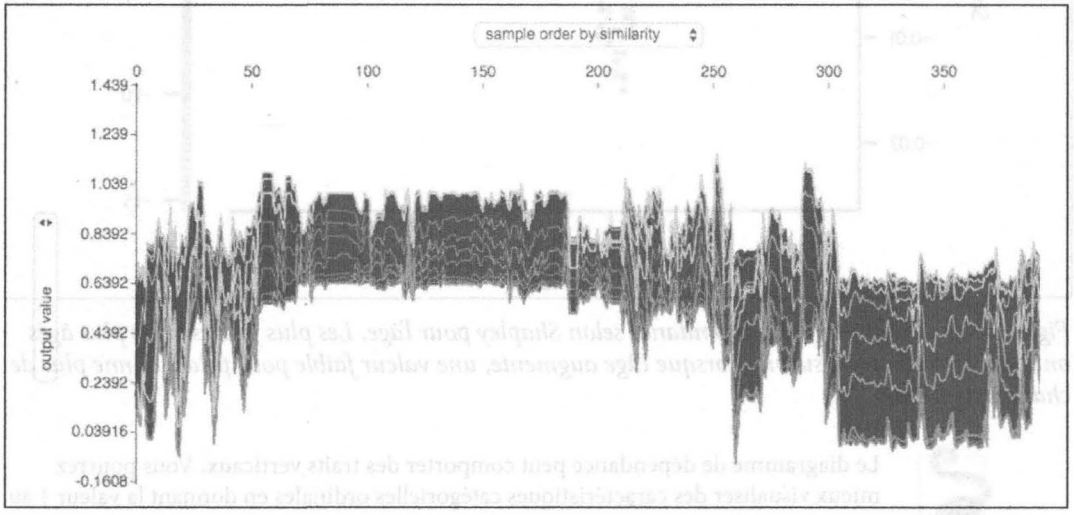



*Figure 13.5 : Contribution des caractéristiques selon Shapley pour tout le jeu de données.*

La librairie SHAP permet également de générer des diagrammes de dépendance. Le diagramme suivant (Figure 13.6) montre les relations entre l'âge et la valeur SHAP. Les couleurs sont choisies en fonction de *pclass* qui a été sélectionné d'office par SHAP ; pour choisir une autre colonne, vous l'indiquez dans le paramètre `interaction_index` :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> res = shap.dependence_plot(
...     "age",
...     shap_vals[target_idx],
...     X_test,
...     feature_names=X_test.columns,
...     alpha=0.7,
... )
>>> fig.savefig(
...     "images/mlpr_1306.png",
...     bbox_inches="tight",
...     dpi=300,
... )
```

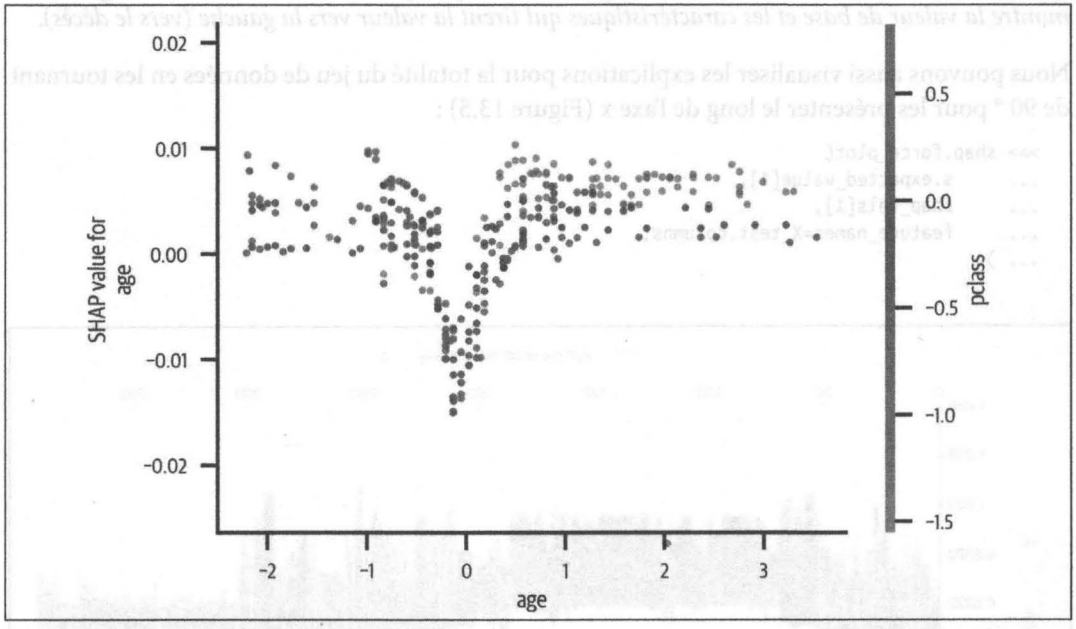



*Figure 13.6 : Diagramme de dépendance selon Shapley pour l'âge. Les plus jeunes et les plus âgés ont plus de chances de survie. Lorsque l'âge augmente, une valeur faible pour pclass donne plus de chances de survie.*

Le diagramme de dépendance peut comporter des traits verticaux. Vous pourrez mieux visualiser des caractéristiques catégorielles ordinales en donnant la valeur 1 au paramètre `x_jitter`.

Nous pouvons enfin demander une synthèse de toutes les caractéristiques, ce qui constitue un diagramme très intéressant car il montre l'impact global ainsi que les impacts individuels. Les caractéristiques sont triées par importance, les principales étant en haut.

En outre, les caractéristiques sont colorées d'après leur valeur. Nous voyons aisément qu'un score faible pour *sex_male* (donc une femme) pousse fortement vers la survie alors qu'un score élevé pousse moins vers le décès. La caractéristique *age* est moins simple à interpréter parce que les personnes jeunes et âgées poussent vers la survie alors que les âges moyens poussent vers le décès.

Si vous combinez le diagramme de synthèse avec le diagramme de dépendance, vous obtenez une bonne explication du comportement de votre modèle (Figure 13.7) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.summary_plot(shap_vals[0], X_test)
>>> fig.savefig("images/mlpr_1307.png", dpi=300)
```

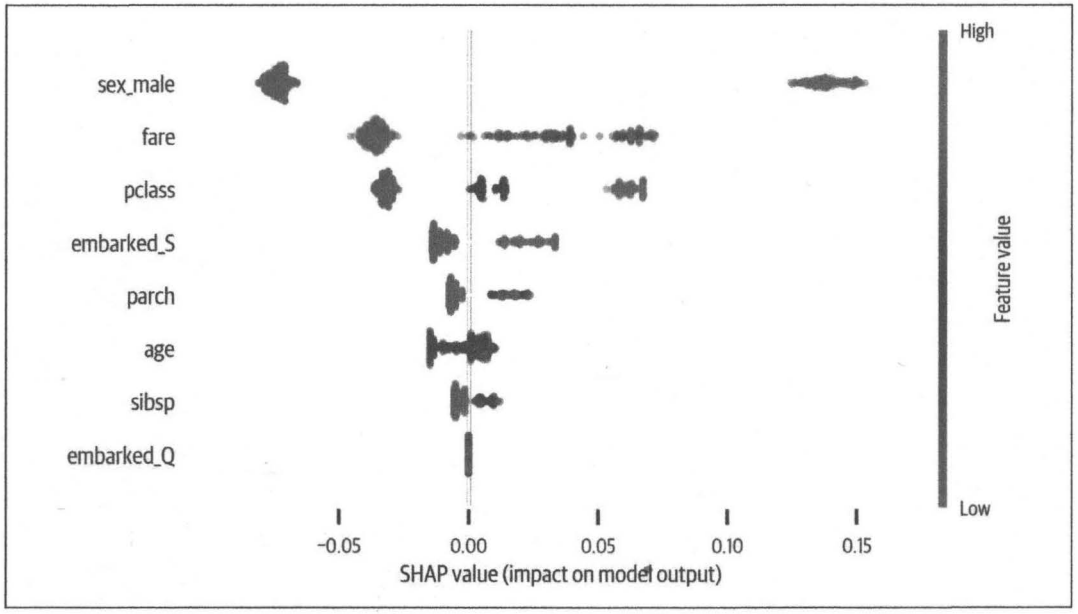



*Figure 13.7 : Diagramme de synthèse Shapley montrant les caractéristiques les plus importantes en haut. Les couleurs expliquent comment les valeurs de la caractéristique impactent la cible.*

# CHAPITRE 14
# Régressions

La régression est un processus d'apprentissage machine supervisé qui s'apparente à la classification, sauf qu'au lieu de prédire des labels, il s'agit de prédire des valeurs numériques. Dès que vous avez besoin de prédire un nombre, vous adopterez la régression.

La librairie **`sklearn`** permet de réaliser la plupart des modèles de classification sous forme de régression. D'ailleurs, l'interface API est la même que pour la classification avec les méthodes `.fit`, `.score` et `.predict`. Cette compatibilité reste vraie pour la prochaine génération de librairies de boosting, dont font partie **`XGBoost`** et **`LightGBM`**.

Les modèles de régression ressemblent à ceux de classification et leurs hyperparamètres, mais c'est au niveau des métriques d'évaluation que les choses divergent. Nous allons dans ce chapitre passer en revue la plupart des types de modèles de régression. Nous nous servirons comme jeu de données de la base de prix immobiliers de la région de Boston (`https://oreil.ly/b2bKQ`).

Puisque nous n'avons pas encore utilisé ce jeu de données, nous commençons par charger les données puis à préparer un sous-jeu d'entraînement et un autre de test, ainsi qu'une autre paire de jeux après standardisation des données :

```
>>> import pandas as pd
>>> from sklearn.datasets import load_boston
>>> from sklearn import (
...     model_selection,
...     preprocessing,
... )

>>> b = load_boston()
>>> bos_X = pd.DataFrame(
...     b.data, columns=b.feature_names
... )
>>> bos_y = b.target

>>> bos_X_train, bos_X_test, bos_y_train,
bos_y_test = model_selection.train_test_split(
```

```
...     bos_X,
...     bos_y,
...     test_size=0.3,
...     random_state=42,
... )

>>> bos_sX = preprocessing.StandardScaler().fit_transform(
...     bos_X
... )

>>> bos_sX_train, bos_sX_test, bos_sy_train,
bos_sy_test = model_selection.train_test_split(
...     bos_sX,
...     bos_y,
...     test_size=0.3,
...     random_state=42,
... )
```

Voici en outre la description abrégée de la signification de chacune des caractéristiques de ce jeu de données immobilières :

| | |
|---|---|
| CRIM | Taux d'homicides par ville |
| ZN | Proportion de zones résidentielles loties d'au minimum 2 500 m² par lot |
| INDUS | Proportion de surface d'activités B2B par ville |
| CHAS | Variable témoin Charles River (= 1 si bien sur rive ; 0 sinon) |
| NOX | Taux d'oxyde d'azote (PP 10 millions) |
| RM | Moyenne de pièces par logement |
| AGE | Proportion de logements avec propriétaire occupant construits avant 1940 |
| DIS | Distances pondérées de cinq agences pour l'emploi de Boston |
| RAD | Index d'accessibilité aux autoroutes sortantes |
| TAX | Taux taxe foncière pour 10 000 $ |
| PTRATIO | Taux enseignants/élèves par ville |
| B | `1000(Bk - 0.63)^2` avec *Bk* la proportion d'Afro-Américains par ville |
| LSTAT | Pourcentage de la classe à faibles revenus |
| MEDV | Valeur médiane des maisons de propriétaires occupants en milliers de $ |

## Modèle de référence (*baseline*)

Un modèle de régression de référence vous sera indispensable pour y confronter les autres modèles. Avec la librairie **sklearn**, la valeur par défaut de la méthode `.score` correspond au coefficient de détermination ($r^2$ ou $R^2$). Cette valeur explique le pourcentage de variation des données d'entrée qui est capturé par la prédiction. C'est une valeur normalement entre zéro et un (mais elle peut être négative lorsque le modèle est particulièrement défectueux).

Le régresseur `DummyRegressor` propose comme stratégie par défaut la prédiction de la valeur moyenne du jeu d'entraînement. Nous pouvons ainsi constater que ce premier modèle n'est pas très pertinent :

```
>>> from sklearn.dummy import DummyRegressor
>>> dr = DummyRegressor()
>>> dr.fit(bos_X_train, bos_y_train)
>>> dr.score(bos_X_test, bos_y_test)
-0.03469753992352409
```

## Régression linéaire

Les régressions linéaires simples sont celles qui sont enseignées en cours de mathématiques et en première année de statistiques. Le but est de réussir à ajuster la formule `y = mx + b` tout en cherchant à réduire le carré des erreurs. Lorsque la formule est résolue, nous obtenons un intercepteur et un coefficient. L'intercepteur fournit une valeur de base pour la prédiction qui est modifiée en y ajoutant le produit du coefficient par la valeur d'entrée.

Cette forme peut être généralisée à un plus grand nombre de dimensions, auquel cas il y a un coefficient pour chaque caractéristique. Plus la valeur absolue est élevée pour le coefficient, plus la caractéristique impacte la cible.

Dans ce modèle, il est supposé que la prédiction reste une combinaison linéaire des entrées. Ce n'est pas suffisamment souple pour certains jeux de données. Vous pouvez dans ce cas ajouter de la complexité en transformant les caractéristiques : le transformeur de **sklearn** nommé `preprocessing.PolynomialFeatures` est en mesure de créer des combinaisons polynomiales de caractéristiques. Si ce traitement entraîne un surajustement, vous pouvez appliquer une régression de type *ridge* ou *lasso* pour régulariser l'estimateur.

Ce modèle est sensible aux problèmes d'*hétéroscédasticité*. Ce phénomène désigne la situation dans laquelle un changement de taille d'une valeur d'entrée provoque une variation de l'erreur de prédiction ou des résidus. Si vous demandez un diagramme entre l'entrée et les résidus, vous obtiendrez une forme de cône. Nous en verrons des exemples plus loin.

Un autre problème à surveiller et la multicolinéarité. Lorsque plusieurs colonnes ont une forte corrélation, les coefficients sont plus difficiles à interpréter. En général, cela n'a pas d'impact sur le modèle, mais cela en a sur la signification des coefficients.

### Propriétés

*Efficacité d'exécution*

Augmentation des performances en jouant sur `n_jobs`.

*Prétraitement des données*

Standardiser les données avant l'entraînement du modèle.

*Prévention du surajustement*

Simplifiez le modèle en n'utilisant et n'ajoutant pas de caractéristiques polynomiales.

*Interprétation des résultats*

Les résultats peuvent être interprétés sous forme de pouvoirs de contribution des caractéristiques, mais le modèle suppose la normalisation et l'indépendance des caractéristiques. Vous pouvez améliorer l'interprétation en supprimant les caractéristiques qui sont colinéaires. La métrique $R^2$ pourra vous aider à savoir à quel point la variance totale du résultat peut s'expliquer par le modèle.

### Exemple d'utilisation

Avec les données par défaut.

```
>>> from sklearn.linear_model import (
...     LinearRegression,
... )
>>> lr = LinearRegression()
>>> lr.fit(bos_X_train, bos_y_train)
LinearRegression(copy_X=True, fit_intercept=True,
n_jobs=1, normalize=False)

>>> lr.score(bos_X_test, bos_y_test)
0.7109203586326287

>>> lr.coef_
array([-1.32774155e-01, 3.57812335e-02,
4.99454423e-02, 3.12127706e+00,
-1.54698463e+01, 4.04872721e+00,
-1.07515901e-02, -1.38699758e+00,
2.42353741e-01, -8.69095363e-03,
-9.11917342e-01, 1.19435253e-02,
-5.48080157e-01])
```

### Paramètres d'instance

`n_jobs=None`

Nombre de processeurs CPU à utiliser. -1 pour tous.

### Attributs après ajustement

`coef_`

Coefficient de régression linéaire.

`intercept_`

Intercepteur du modèle linéaire.

La valeur `intercept_` correspond à la valeur moyenne espérée. Pour voir comment le changement d'échelle des données impacte les coefficients, observez leur signe : il explique le sens de la relation entre caractéristiques et cible. Une valeur positive correspond à une augmentation de la valeur cible en proportion de celle de la caractéristique. Une valeur négative diminue la valeur cible quand la caractéristique augmente. Plus la valeur absolue du coefficient est élevée, plus il a d'impact :

```
>>> lr2 = LinearRegression()
>>> lr2.fit(bos_sX_train, bos_sy_train)
LinearRegression(copy_X=True, fit_intercept=True,
n_jobs=1, normalize=False)
>>> lr2.score(bos_sX_test, bos_sy_test)
0.7109203586326278
>>> lr2.intercept_
22.50945471291039
>>> lr2.coef_
array([-1.14030209, 0.83368112, 0.34230461,
0.792002, -1.7908376, 2.84189278, -0.30234582,
-2.91772744, 2.10815064, -1.46330017,
-1.97229956, 1.08930453, -3.91000474])
```

Vous pouvez visualiser les coefficients avec **`Yellowbrick`** (Figure 14.1). Puisque nous fournissons les données Boston remises à l'échelle dans un tableau **`numpy`**, et non dans une structure DataFrame de **`pandas`**, nous prenons soin de transmettre le paramètre `labels` pour pouvoir utiliser les noms des colonnes :

```
>>> from yellowbrick.features import (
...     FeatureImportances,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(
...     lr2, labels=bos_X.columns
... )
>>> fi_viz.fit(bos_sX, bos_sy)
>>> fi_viz.poof()
>>> fig.savefig(
```

```
...     "images/mlpr_1401.png",
...     bbox_inches="tight",
...     dpi=300,
... )
```

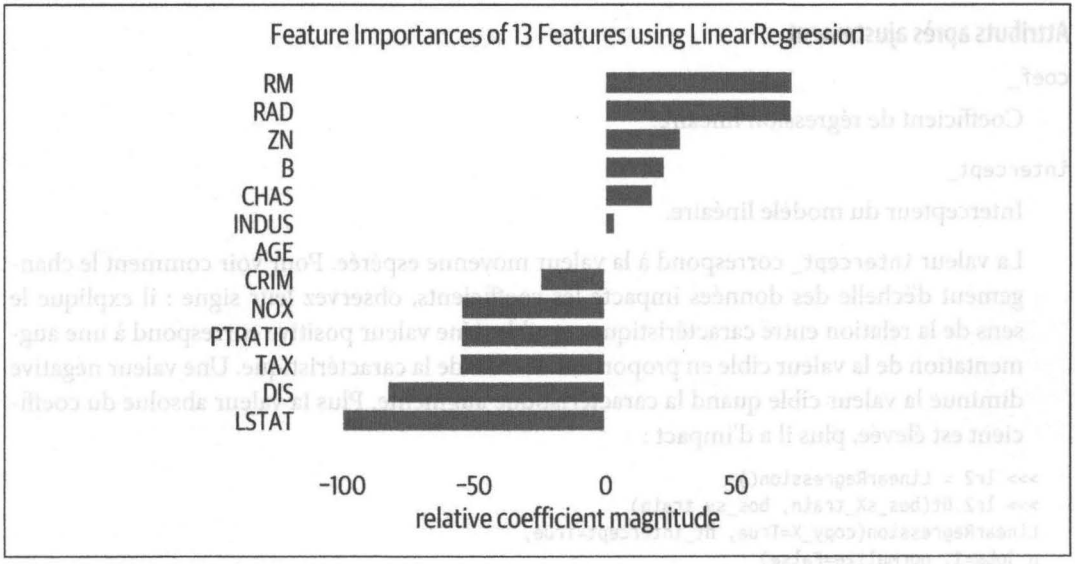



*Figure 14.1 : Visualisation de l'importance des caractéristiques. Par exemple, le nombre de pièces RM augmente en même temps que le prix ; la vétusté AGE n'a pas d'impact conséquent et le pourcentage de faibles revenus LSTAT fait logiquement baisser le prix.*

# SVM

Les machines à vecteurs de support permettent de réaliser des régressions.

## Propriétés

*Efficacité d'exécution*

L'implémentation par **`scikit-learn`** est de niveau `O(n4)`. La montée en volume peut donc être difficile. Un noyau linéaire ou le modèle `LinearSVR` peut améliorer les performances d'exécution en faisant chuter l'exactitude. En optimisant le paramètre `cache_size`, la durée peut être ramenée à `O(n3)`.

*Prétraitement des données*

L'algorithme n'est pas invariant pour l'échelle ; il est donc très recommandé de standardiser les données d'abord.

*Prévention du surajustement*

La régularisation est contrôlée par le paramètre de pénalité `C`. Une petite valeur procure une petite marge dans l'hyperplan. Une valeur de gamma élevée aura tendance à faire surajuster les données d'entraînement. Le modèle `LinearSVR` permet de régulariser avec les paramètres `loss` et `penalty`. Le paramètre `epsilon` peut être augmenté (la valeur zéro doit vous faire redouter un surajustement).

*Interprétation des résultats*

Inspectez `.support_vectors_`, mais l'opération n'est pas simple. Pour un noyau linéaire, vous pouvez inspecter `.coef_`.

### Exemple d'utilisation

```
>>> from sklearn.svm import SVR
>>> svr = SVR()
>>> svr.fit(bos_sX_train, bos_sy_train)
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3,
 epsilon=0.1, gamma='auto', kernel='rbf',
 max_iter=-1, shrinking=True, tol=0.001,
 verbose=False)

>>> svr.score(bos_sX_test, bos_sy_test)
0.6555356362002485
```

### Paramètres d'instance

`C=1.0`

Paramètre de pénalité. Plus la valeur est faible, plus la frontière de décision est serrée (plus de surajustement).

`cache_size=200`

Taille du cache en Mo. La durée d'entraînement peut être réduite pour un grand volume de données en augmentant cette valeur.

`coef0=0.0`

Terme indépendant pour les noyaux polynomiaux et sigmoïdes.

`epsilon=0.1`

Marge de tolérance dans laquelle aucune pénalité n'est attribuée aux erreurs. La valeur doit être faible pour les grands volumes de données.

`degree=3`

Degré de noyau polynomial.

`gamma='auto'`

Coefficient de noyau. Peut être un nombre, ou bien `'scale'` (par défaut en V.0.22, `1 / (num features * X.std())`) ou encore `'auto'` (par défaut avant, `1 / num_features`). Une valeur faible entraîne un surajustement des données d'entraînement.

`kernel='rbf'`

Type de noyau : `'linear'`, `'poly'`, `'rbf'` (par défaut), `'sigmoid'`, `'precomputed'` ou une fonction.

`max_iter=-1`

Nombre maximal d'itérations pour le solveur et -1 pour ne pas imposer de limite.

`probability=False`

Active l'estimation de probabilité mais ralentit l'entraînement.

`random_state=None`

Graine du générateur aléatoire.

`shrinking=True`

Utiliser l'heuristique *shrinking*.

`tol=0.001`

Tolérance d'arrêt.

`verbose=False`

Verbosité d'affichage.

### Attributs après ajustement

`support_`

Indices des vecteurs de support.

`support_vectors_`

Vecteurs de support.

`coef_`

Coefficients de noyau (linéaire).

`intercept_`

Constante pour la fonction de décision.

# K-plus proches voisins (KNN)

Le modèle KNN (*K-nearest neighbors*) permet la régression en cherchant les cibles les plus proches voisines de l'échantillon pour lequel vous désirez une prédiction. Dans le cas des régressions, le modèle fait la moyenne des cibles entre elles pour trouver sa prédiction.

## Propriétés

*Efficacité d'exécution*

La durée d'entraînement s'établit selon `O(1)`, mais les données des échantillons doivent être stockées. La durée de test s'élève à `O(Nd)`, avec $N$ le nombre d'exemples d'entraînement et $d$ la dimensionnalité.

*Prétraitement des données*

Oui, les calculs basés distance sont toujours meilleurs lorsqu'il y a standardisation.

*Prévention du surajustement*

Augmentez `n_neighbors`. Changez `p` pour une métrique `L1` ou `L2`.

*Interprétation des résultats*

Il faut interpréter les k-plus proches voisins de l'échantillon au moyen de la méthode `.kneighbors`. Si vous pouvez expliquer ces voisins, ils expliquent le résultat.

## Exemple d'utilisation

```
>>> from sklearn.neighbors import (
...     KNeighborsRegressor,
... )
>>> knr = KNeighborsRegressor()
>>> knr.fit(bos_sX_train, bos_sy_train)
KNeighborsRegressor(algorithm='auto',
  leaf_size=30, metric='minkowski',
  metric_params=None, n_jobs=1, n_neighbors=5,
  p=2, weights='uniform')

>>> knr.score(bos_sX_test, bos_sy_test)
0.747112767457727
```

## Attributs

`algorithm='auto'`

`'brute'`, `'ball_tree'` ou `'kd_tree'`.

`leaf_size=30`

Sert aux algorithmes en arbre.

`metric='minkowski'`

Métrique de distance.

`metric_params=None`

Dictionnaire de paramètres complémentaires pour une fonction métrique spécifique.

`n_jobs=1`

Nombre de CPU.

`n_neighbors=5`

Nombre de voisins.

`p=2`

Paramètre de puissance de Minkowski. 1 = manhattan (L1). 2 = euclidien (L2).

`weights='uniform'`

Peut être `'distance'`, auquel cas les points les plus proches ont plus d'influence.

# Arbre de décision

Les arbres de décision permettent la classification et la régression. Sont évaluées les différentes divisions réalisées à chaque niveau de l'arbre sur les caractéristiques. La division qui produit la plus petite erreur ou impureté est alors sélectionnée. La mesure d'impureté peut être réglée grâce au paramètre `criterion`.

## Propriétés

*Efficacité d'exécution*

Dans la phase de création, il faut boucler parmi chacune des caractéristiques $m$, pour trier tous les échantillons $n$, donc selon `O(mn log n)`. Pour la phase de prédiction, il faut parcourir l'arbre selon `O(hauteur)`.

*Prétraitement des données*

La remise à l'échelle n'est pas nécessaire. Il faut se débarrasser des manquants et convertir vers des types numériques.

*Prévention du surajustement*

Réduire `max_depth` et augmenter `min_impurity_decrease`.

**Interprétation des résultats**

Possibilité de parcourir pas à pas l'arbre des décisions. En raison de ces étapes, les arbres ne gèrent pas correctement les relations linéaires : un petit changement de valeur d'une caracté-

ristique peut entraîner la création d'un arbre totalement différent. Les arbres sont également très dépendants des données d'entraînement. Le moindre changement peut radicalement changer l'arbre entier.

### Exemple d'utilisation

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> dtr = DecisionTreeRegressor(random_state=42)
>>> dtr.fit(bos_X_train, bos_y_train)
DecisionTreeRegressor(criterion='mse',
  max_depth=None, max_features=None,
  max_leaf_nodes=None, min_impurity_decrease=0.0,
  min_impurity_split=None, min_samples_leaf=1,
  min_samples_split=2,
  min_weight_fraction_leaf=0.0, presort=False,
  random_state=42, splitter='best')

>>> dtr.score(bos_X_test, bos_y_test)
0.8426751288675483
```

### Paramètres d'instance

`criterion='mse'`

Fonction de division (split). Par défaut, c'est l'erreur quadratique moyenne (L2 loss), les autres choix étant `'friedman_mse'` ou `'mae'` (L1 loss).

`max_depth=None`

Profondeur de l'arbre. Par défaut, des branches sont créées jusqu'à n'avoir plus que `min_samples_split` dans les feuilles.

`max_features=None`

Nombre de caractéristiques à examiner pour les divisions. Par défaut, toutes.

`max_leaf_nodes=None`

Nombre maximal de feuilles. Par défaut, sans limites.

`min_impurity_decrease=0.0`

Division du nœud si la division va réduire l'impureté d'au minimum cette valeur.

`min_impurity_split=None`

Déprécié.

`min_samples_leaf=1`

Nombre minimal d'échantillons dans chaque feuille.

`min_samples_split=2`

Nombre minimal d'échantillons requis pour diviser un nœud.

`min_weight_fraction_leaf=0.0`

Somme de poids minimale requise pour les nœuds feuilles.

`presort=False`

Peut accélérer l'entraînement pour un petit jeu de données ou une faible profondeur si vous indiquez la valeur True.

`random_state=None`

Graine du générateur aléatoire.

`splitter='best'`

Choix entre `'random'` et `'best'`.

### Attributs après ajustement

`feature_importances_`

Tableau d'importance de Gini.

`max_features_`

Valeur calculée de `max_features`.

`n_outputs_`

Nombre de sorties.

`n_features_`

Nombre de caractéristiques.

`tree_`

Objet arbre sous-jacent.

Visualisation de l'arbre (Figure 14.2) :

```
>>> import pydotplus
>>> from io import StringIO
>>> from sklearn.tree import export_graphviz
>>> dot_data = StringIO()
>>> tree.export_graphviz(
...     dtr,
...     out_file=dot_data,
...     feature_names=bos_X.columns,
...     filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
...     dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1402.png")
```

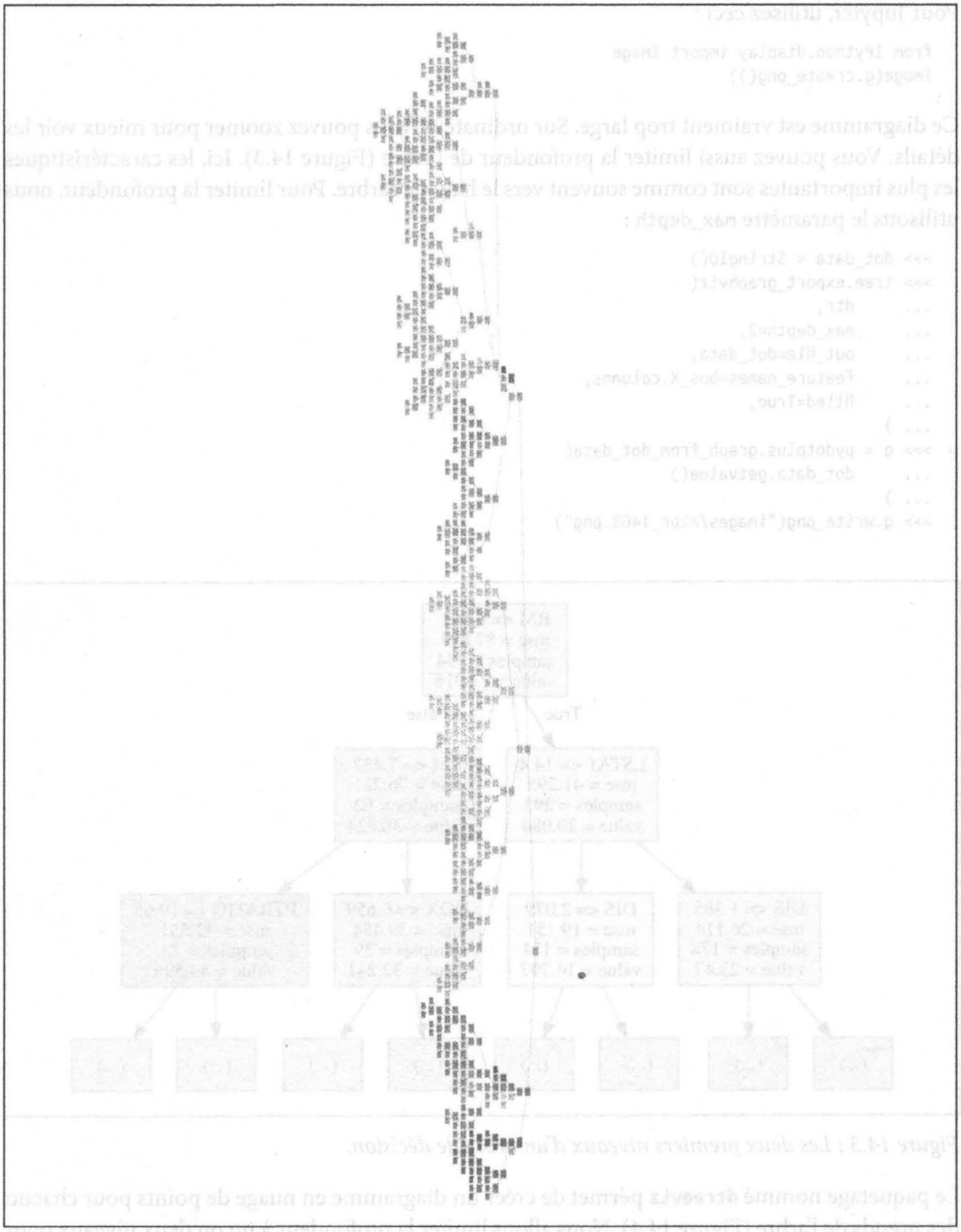


*Figure 14.2 : Visualisation d'un arbre de décision.*

Pour Jupyter, utilisez ceci :

```
from IPython.display import Image
Image(g.create_png())
```

Ce diagramme est vraiment trop large. Sur ordinateur, vous pouvez zoomer pour mieux voir les détails. Vous pouvez aussi limiter la profondeur de l'arbre (Figure 14.3). Ici, les caractéristiques les plus importantes sont comme souvent vers le haut de l'arbre. Pour limiter la profondeur, nous utilisons le paramètre `max_depth` :

```
>>> dot_data = StringIO()
>>> tree.export_graphviz(
...     dtr,
...     max_depth=2,
...     out_file=dot_data,
...     feature_names=bos_X.columns,
...     filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
...     dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1403.png")
```

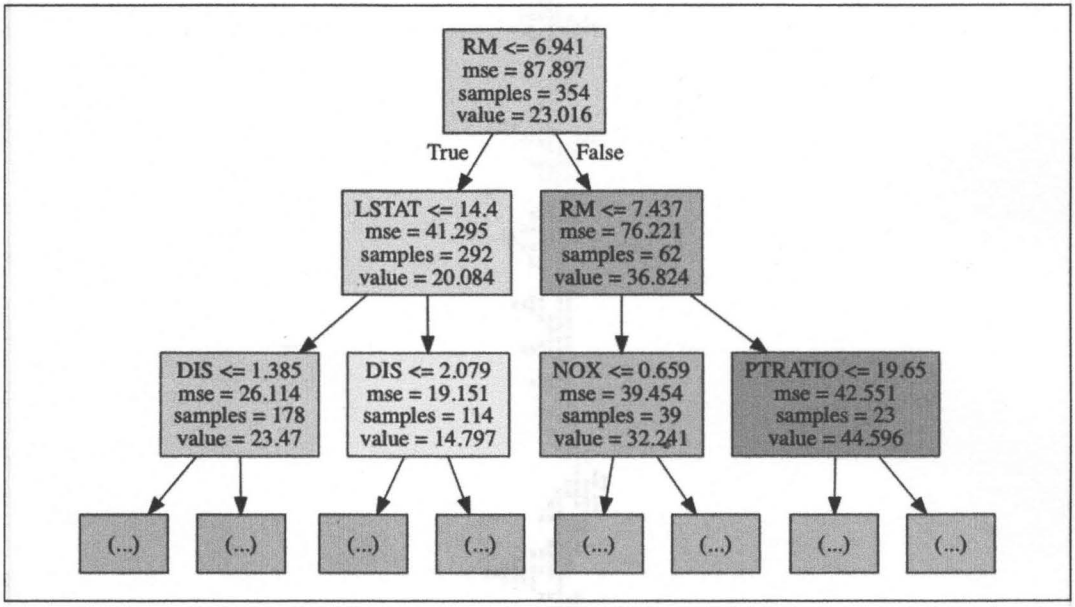



*Figure 14.3 : Les deux premiers niveaux d'un arbre de décision.*

Le paquetage nommé **`dtreeviz`** permet de créer un diagramme en nuage de points pour chacun des nœuds de l'arbre (Figure 14.4). Nous allons limiter la profondeur à un ou deux niveaux pour pouvoir voir les détails :

```
>>> dtr3 = DecisionTreeRegressor(max_depth=2)
>>> dtr3.fit(bos_X_train, bos_y_train)
>>> viz = dtreeviz.trees.dtreeviz(
...     dtr3,
...     bos_X,
...     bos_y,
...     target_name="price",
...     feature_names=bos_X.columns,
... )
>>> viz
```

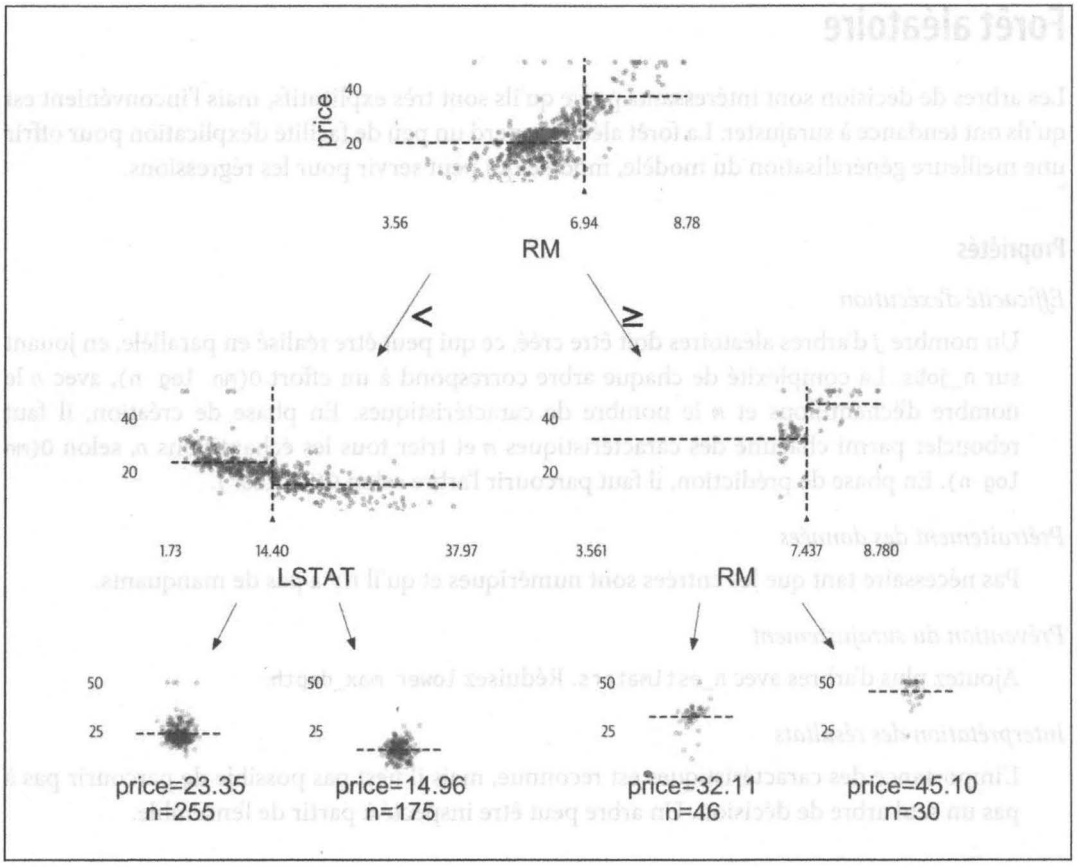



*Figure 14.4 : Visualisation d'une régression avec dtreeviz.*

Importance des caractéristiques:

```
>>> for col, val in sorted(
...     zip(
...         bos_X.columns, dtr.feature_importances_
...     ),
```

```
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
RM         0.574
LSTAT      0.191
DIS        0.110
CRIM       0.061
RAD        0.018
```

# Forêt aléatoire

Les arbres de décision sont intéressants parce qu'ils sont très explicatifs, mais l'inconvénient est qu'ils ont tendance à surajuster. La forêt aléatoire perd un peu de facilité d'explication pour offrir une meilleure généralisation du modèle, modèle qui peut servir pour les régressions.

## Propriétés

*Efficacité d'exécution*

Un nombre $j$ d'arbres aléatoires doit être créé, ce qui peut être réalisé en parallèle, en jouant sur `n_jobs`. La complexité de chaque arbre correspond à un effort `O(mn log n)`, avec $n$ le nombre d'échantillons et $m$ le nombre de caractéristiques. En phase de création, il faut reboucler parmi chacune des caractéristiques $m$ et trier tous les échantillons $n$, selon `O(mn log n)`. En phase de prédiction, il faut parcourir l'arbre selon `O(hauteur)`.

*Prétraitement des données*

Pas nécessaire tant que les entrées sont numériques et qu'il n'y a pas de manquants.

*Prévention du surajustement*

Ajoutez plus d'arbres avec `n_estimators`. Réduisez `lower max_depth`.

*Interprétation des résultats*

L'importance des caractéristiques est reconnue, mais il n'est pas possible de parcourir pas à pas un seul arbre de décision. Un arbre peut être inspecté à partir de l'ensemble.

## Exemple d'utilisation

```
>>> from sklearn.ensemble import (
...     RandomForestRegressor,
... )
>>> rfr = RandomForestRegressor(
...     random_state=42, n_estimators=100
... )
>>> rfr.fit(bos_X_train, bos_y_train)
```

```
RandomForestRegressor(bootstrap=True,
  criterion='mse', max_depth=None,
  max_features='auto', max_leaf_nodes=None,
  min_impurity_decrease=0.0,
  min_impurity_split=None,_samples_leaf=1,
  min_samples_split=2,
  min_weight_fraction_leaf=0.0,
  n_estimators=100, n_jobs=1,
  oob_score=False, random_state=42,
  verbose=0, warm_start=False)

>>> rfr.score(bos_X_test, bos_y_test)
0.8641887615545837
```

### Paramètres d'instance

(Ces options reflètent celles des arbres de décision.)

`bootstrap=True`

Utilise un autorenforcement *bootstrap* pour construire les arbres.

`criterion='mse'`

Fonction de division, `'mae'`.

`max_depth=None`

Profondeur de l'arbre. Par défaut, des branches sont créées jusqu'à n'avoir plus que `min_samples_split` dans les feuilles.

`max_features='auto'`

Nombre de caractéristiques à examiner pour les divisions. Par défaut, toutes.

`max_leaf_nodes=None`

Nombre maximal de feuilles. Par défaut, sans limites.

`min_impurity_decrease=0.0`

Division du nœud si la division va réduire l'impureté d'au minimum cette valeur.

`min_impurity_split=None`

Déprécié.

`min_samples_leaf=1`

Nombre minimal d'échantillons dans chaque feuille.

`min_samples_split=2`

Nombre minimal d'échantillons requis pour diviser un nœud.

`min_weight_fraction_leaf=0.0`

Somme de poids minimale requise pour les nœuds feuilles.

`n_estimators=10`

Nombre d'arbres dans la forêt.

`n_jobs=None`

Nombre de processus exétrons pour l'ajustement et la prédiction (`None` vaut 1.)

`oob_score=False`

Utilisation ou pas d'échantillons hors sac OOB pour estimer un score pour des données inconnues.

`random_state=None`

Générateur aléatoire.

`verbose=0`

Verbosité.

`warm_start=False`

Ajuste une nouvelle forêt ou utilise celle qui existe.

### Attributs après ajustement

`estimators_`

Collection d'arbres.

`feature_importances_`

Tableau d'importance de Gini.

`n_classes_`

Nombre de classes.

`n_features_`

Nombre de caractéristiques

`oob_score_`

Score du jeu d'entraînement avec estimation OOB.

**Importance des caractéristiques :**

```
>>> for col, val in sorted(
...     zip(
...         bos_X.columns, rfr.feature_importances_
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
RM         0.505
```

```
LSTAT     0.283
DIS       0.115
CRIM      0.029
PTRATIO   0.016
```

# Régression XGBoost

La librairie **`XGBoost`** permet la régression en commençant par construire un arbre de décision simple, puis en l'amplifiant (*boosting*) par ajout d'autres arbres. Chaque nouvel arbre cherche à corriger les résidus de la précédente sortie. En pratique, cela fonctionne bien avec des données structurées.

## Propriétés

*Efficacité d'exécution*

**`XGBoost`** est parallélisable. Vous contrôlez le nombre de processeurs à utiliser au moyen de l'option `n_jobs`. Profitez si possible des coprocesseurs graphiques GPU.

*Prétraitement des données*

Pas de remise à l'échelle nécessaire avec les arbres. Les données catégorielles doivent être encodées. Les données manquantes sont supportées !

*Prévention du surajustement*

Le paramètre `early_stopping_rounds=N` peut être réglé pour stopper l'entraînement s'il n'y a plus d'amélioration après *N* tours. Les régularisations L1 et L2 sont contrôlées par les deux paramètres `reg_alpha` et `reg_lambda`, respectivement. Plus la valeur est grande, plus le traitement est conservateur.

*Interprétation des résultats*

Permet d'évaluer l'importance des caractéristiques.

## Exemple d'utilisation

```
>>> xgr = xgb.XGBRegressor(random_state=42)
>>> xgr.fit(bos_X_train, bos_y_train)
XGBRegressor(base_score=0.5, booster='gbtree',
  colsample_bylevel=1, colsample_bytree=1,
  gamma=0, learning_rate=0.1, max_delta_step=0,
  max_depth=3, min_child_weight=1, missing=None,
  n_estimators=100, n_jobs=1, nthread=None,
  objective='reg:linear', random_state=42,
  reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
  seed=None, silent=True, subsample=1)
>>> xgr.score(bos_X_test, bos_y_test)
```

```
0.871679473122472

>>> xgr.predict(bos_X.iloc[[0]])
array([27.013563], dtype=float32)
```

### Paramètres d'instance

`max_depth=3`

Profondeur maximale.

`learning_rate=0.1`

Vitesse d'apprentissage (eta) entre 0 et 1 pour le boosting. Après chaque étape d'amplification, les poids venant d'être ajoutés sont remis à l'échelle selon ce facteur. Plus la valeur est faible, plus le traitement est conservateur, mais également plus il va avoir besoin d'arbres pour converger. Vous pouvez spécifier dans l'appel à `.train` le paramètre `learning_rates`, sous forme d'une liste de vitesses pour chaque tour, par exemple `[.1]*100 + [.05]*100`.

`n_estimators=100`

Nombre de tours ou d'arbres amplifiés.

`silent=True`

Affichage ou pas de messages pendant le traitement de boosting.

`objective="reg:linear"`

Tâche d'apprentissage ou fonction appelable pour la classification.

`booster="gbtree"`

Valeur `'gbtree'`, `'gblinear'` ou `'dart'`. L'option `dart` ajoute du délestage, consistant à écarter des arbres au hasard pour éviter le surajustement. L'option `gblinear` produit un modèle linéaire régularisé (proche d'une régression lasso).

`nthread=None`

Déprécié.

`n_jobs=1`

Nombre d'exétrons CPU à utiliser.

`gamma=0`

Réduction de perte minimale requise pour continuer à diviser une feuille.

`min_child_weight=1`

Valeur minimale de la somme hessienne pour un nœud fils.

`max_delta_step=0`

Rend l'actualisation plus conservatrice. Valeur entre 1 et 10 pour les classes déséquilibrées.

`subsample=1`

Pourcentage d'échantillons à utiliser pour le prochain tour de boosting.

`colsample_bytree=1`

Pourcentage de colonnes à utiliser pour le prochain tour de boosting.

`colsample_bylevel=1`

Pourcentage de colonnes à utiliser pour le niveau courant dans l'arbre.

`colsample_bynode=1`

Pourcentage de colonnes à utiliser pour la division d'un nœud dans l'arbre.

`reg_alpha=0`

Régularisation L1 (moyenne des poids). À augmenter pour être plus conservateur.

`reg_lambda=1`

Régularisation L2 (racine quadratique des poids). À augmenter pour être plus conservateur.

`base_score=.5`

Prédiction initiale.

`seed=None`

Déprécié.

`random_state=0`

Graine du générateur aléatoire.

`missing=None`

Valeurs à utiliser pour les manquants. `None` signifie `np.nan`.

`importance_type='gain'`

Type de l'importance des caractéristiques : `'gain'`, `'weight'`, `'cover'`, `'total_gain'` ou `'total_cover'`.

### Attributs

`coef_`

Coefficients pour learner `gblinear` (`booster = 'gblinear'`)

`intercept_`

Intercepteur pour learner `gblinear`

`feature_importances_`

Importance des caractéristiques pour learner `gbtree`

L'importance des caractéristiques est le gain moyen parmi tous les nœuds dans lesquels la caractéristique est utilisée :

```
>>> for col, val in sorted(
...     zip(
...         bos_X.columns, xgr.feature_importances_
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
DIS         0.187
CRIM        0.137
RM          0.137
LSTAT       0.134
AGE         0.110
```

**XGBoost** offre des capacités de tracé de graphique pour l'importance des caractéristiques. Le paramètre `importance_type` influe sur les valeurs dans le diagramme (Figure 14.5). Par défaut, l'importance des caractéristiques est déterminée par les poids :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> xgb.plot_importance(xgr, ax=ax)
>>> fig.savefig("images/mlpr_1405.png", dpi=300)
```

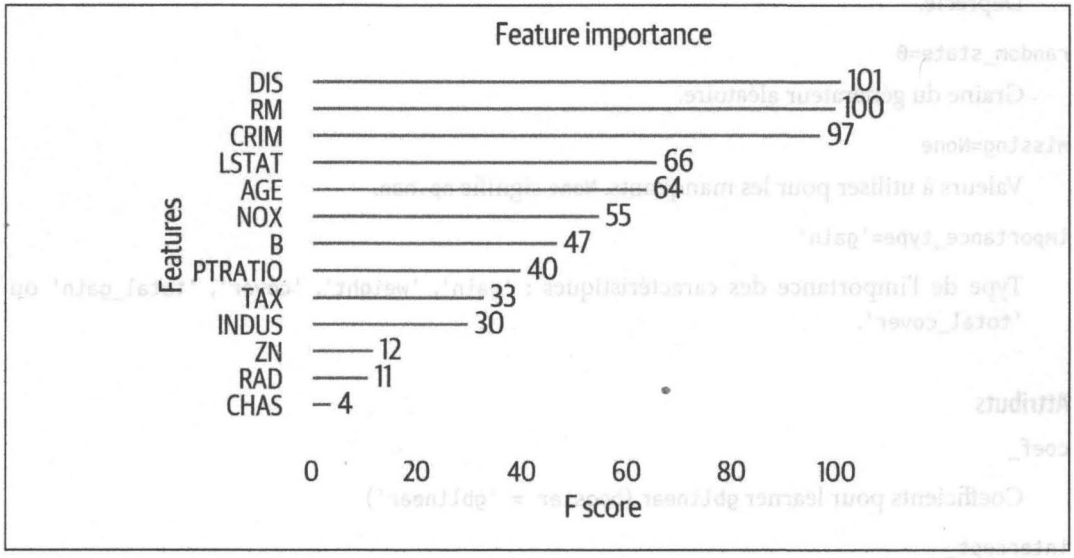



*Figure 14.5 : Importance des caractéristiques à partir des poids (nombre de divisions d'une caractéristique dans les arbres).*

**Yellowbrick** permet aussi de visualiser l'importance des caractéristiques en normalisant l'attribut `feature_importances_` (Figure 14.6) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(xgr)
>>> fi_viz.fit(bos_X_train, bos_y_train)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1406.png", dpi=300)
```

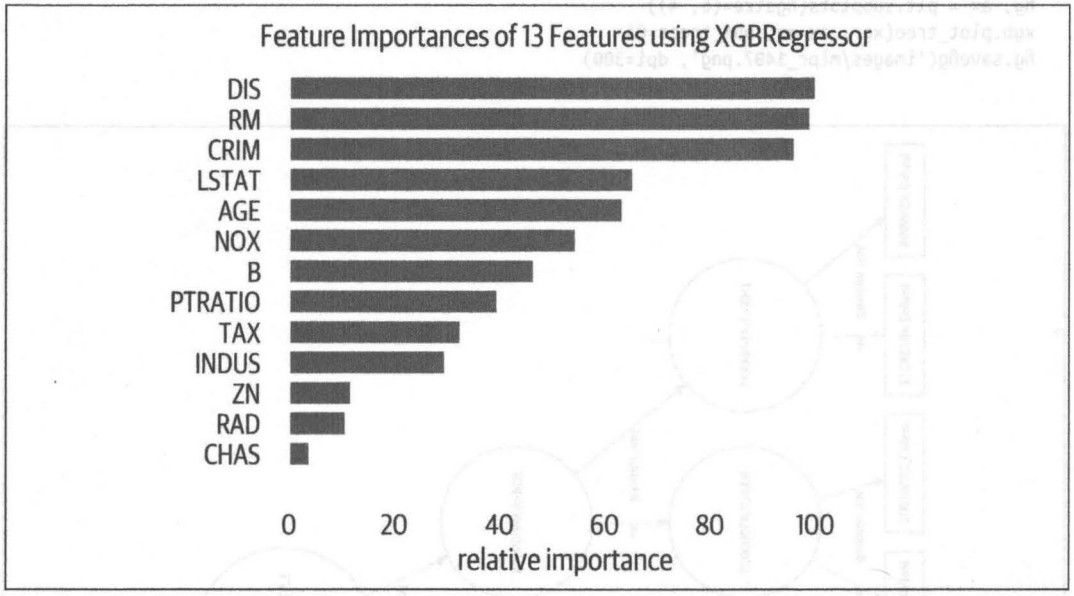



*Figure 14.6 : Importance des caractéristiques à partir de l'importance relative des gains (importance en pourcentage de la caractéristique principale).*

**XGBoost** propose une représentation des arbres sous forme de texte, comme sous forme graphique. Voici celle sous forme de texte :

```
>>> booster = xgr.get_booster()
>>> print(booster.get_dump()[0])
0:[LSTAT<9.72500038] yes=1,no=2,missing=1
 1:[RM<6.94099998] yes=3,no=4,missing=3
  3:[DIS<1.48494995] yes=7,no=8,missing=7
   7:leaf=3.9599998
   8:leaf=2.40158272
  4:[RM<7.43700027] yes=9,no=10,missing=9
   9:leaf=3.22561002
   10:leaf=4.31580687
 2:[LSTAT<16.0849991] yes=5,no=6,missing=5
  5:[B<116.024994] yes=11,no=12,missing=11
   11:leaf=1.1825
   12:leaf=1.99701393
  6:[NOX<0.603000045] yes=13,no=14,missing=13
   13:leaf=1.6868
   14:leaf=1.18572915
```

Les valeurs des feuilles correspondent à la somme de `base_score` et de la feuille. (Vous validez cette interprétation en appelant `.predict` avec le paramètre `ntree_limit=1` afin d'obliger le modèle à n'utiliser les résultats que du premier arbre.)

Version graphique de l'arbre (Figure 14.7) :

```
fig, ax = plt.subplots(figsize=(6, 4))
xgb.plot_tree(xgr, ax=ax, num_trees=0)
fig.savefig('images/mlpr_1407.png', dpi=300)
```

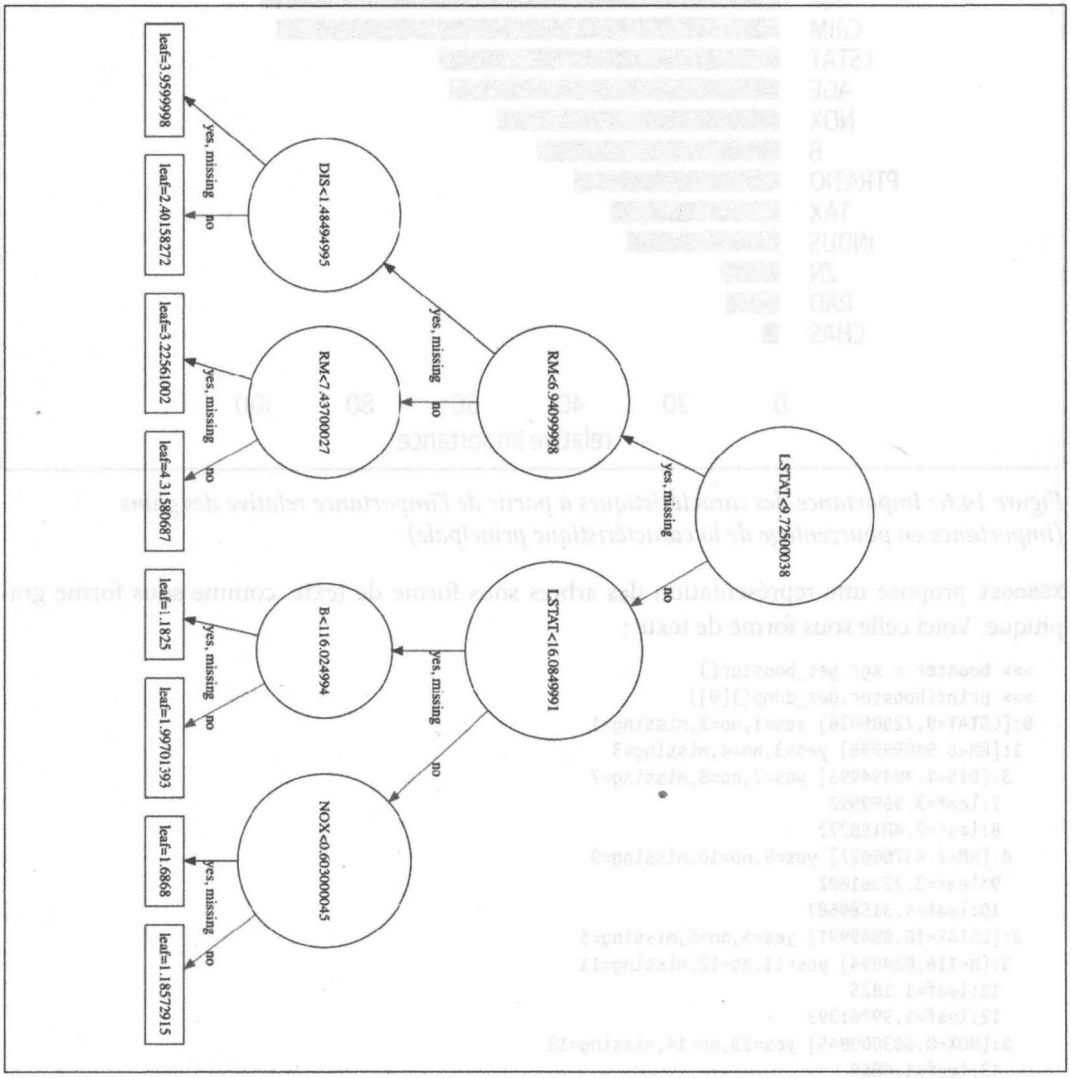



*Figure 14.7 : Arbre XGBoost.*

# Régression LightGBM

La librairie d'arbre à gradient boosting, **`LightGBM`**, permet la régression. Nous avons vu dans le chapitre sur la classification qu'elle pouvait se montrer plus rapide que **`XGBoost`** dans la création des arbres en raison du mécanisme d'échantillonnage qui sert à décider des divisions des nœuds.

Du fait que cette technique fait croître les arbres dans le sens de la profondeur d'abord, vous risquez d'endommager le modèle si vous imposez une limite à cette profondeur.

## Propriétés

*Efficacité d'exécution*

Sait tirer profit d'un système multiprocesseur. Peut être jusqu'à 15 fois plus rapide que **`XGBoost`** si vous pouvez utiliser le binning.

*Prétraitement des données*

Dispose d'un support partiel pour l'encodage des colonnes catégorielles sous forme d'entiers ou du type pandas `Categorical`. Il semble que AUC en souffre, en comparaison d'un encodage One Hot.

*Prévention du surajustement*

Réduisez `num_leaves`. Augmentez `min_data_in_leaf`. Utilisez `min_gain_to_split` avec `lambda_l1` ou `lambda_l2`.

*Interprétation des résultats*

L'importance des caractéristiques est accessible. Chaque arbre est pauvre et n'est pas facile à interpréter.

## Exemple d'utilisation

```
>>> import lightgbm as lgb
>>> lgr = lgb.LGBMRegressor(random_state=42)
>>> lgr.fit(bos_X_train, bos_y_train)
LGBMRegressor(boosting_type='gbdt',
  class_weight=None, colsample_bytree=1.0,
  learning_rate=0.1, max_depth=-1,
  min_child_samples=20, min_child_weight=0.001,
  min_split_gain=0.0, n_estimators=100,
  n_jobs=-1, num_leaves=31, objective=None,
  random_state=42, reg_alpha=0.0,
  reg_lambda=0.0, silent=True, subsample=1.0,
  subsample_for_bin=200000, subsample_freq=0)

>>> lgr.score(bos_X_test, bos_y_test)
0.847729219534575
```

```
>>> lgr.predict(bos_X.iloc[[0]])
array([30.31689569])
```

### Paramètres d'instance

`boosting_type='gbdt'`

Choix entre `'gbdt'` (*Gradient Boosting*), `'rf'` (*Random Forest*), `'dart'` (*Dropouts Additive Regression Trees*) et `'goss'` (*Gradient-based One-Sided Sampling*).

`num_leaves=31`

Nombre maximum de feuilles par arbre.

`max_depth=-1`

Profondeur maximale de l'arbre. Sans limites avec -1. Une profondeur plus grande a tendance à favoriser le surajustement.

`learning_rate=0.1`

Plage (0, 1.0]. Taux d'apprentissage pour le boosting. Une valeur faible ralentit le surajustement car les tours de boosting ont moins d'impact, ce qui offre donc de meilleures performances, mais réclame un plus grand nombre de tours avec `num_iterations`.

`n_estimators=100`

Nombre d'arbres ou de tours de boosting.

`subsample_for_bin=200000`

Nombre d'échantillons requis pour créer les bacs bins.

`objective=None`

La valeur `None` demande une régression par défaut. Peut être une fonction ou une chaîne de caractères.

`min_split_gain=0.0`

Réduction de perte requise pour partitionner une feuille.

`min_child_weight=0.001`

Somme hessienne requise pour une feuille. Une forte valeur est plus conservatrice.

`min_child_samples=20`

Nombre d'échantillons requis pour une feuille. Une valeur faible entraîne plus de surajustement.

`subsample=1.0`

Fraction des échantillons à utiliser pour le tour suivant.

`subsample_freq=0`

Fréquence de sous-échantillonnage. Valeur 1 pour activer.

`colsample_bytree=1.0`

Plage (0, 1.0]. Pourcentage des caractéristiques pour chaque tour de boosting.

`reg_alpha=0`

Régularisation L1 (moyenne des poids). À augmenter pour être plus conservateur.

`reg_lambda=1`

Régularisation L2 (racine quadratique des poids). À augmenter pour être plus conservateur.

`random_state=42`

Graine du générateur aléatoire.

`n_jobs=-1`

Nombre d'exétrons (*threads*).

`silent=True`

Mode d'affichage verbeux.

`importance_type='split'`

Contrôle le mode de calcul des importances : `'split'` (nombre de fois qu'une caractéristique a été utilisée) ou bien `'gain'` (total des divisions dans lesquelles une caractéristique a été utilisée).

**`LightGBM`** rend accessible l'importance des caractéristiques. Le mode de calcul est déterminé par le paramètre `importance_type`. Par défaut, il se base sur le nombre d'utilisations effectives d'une caractéristique :

```
>>> for col, val in sorted(
...     zip(
...         bos_X.columns, lgr.feature_importances_
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
LSTAT      226.000
RM         199.000
DIS        172.000
AGE        130.000
B          121.000
```

L'importance des caractéristiques peut être visualisée dans un diagramme montrant le nombre d'utilisations d'une caractéristique (Figure 14.8) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lgb.plot_importance(lgr, ax=ax)
>>> fig.tight_layout()
>>> fig.savefig("images/mlpr_1408.png", dpi=300)
```

Pour visualiser un arbre dans Jupyter, il faut ajouter la commande suivante :

```
lgb.create_tree_digraph(lgbr)
```

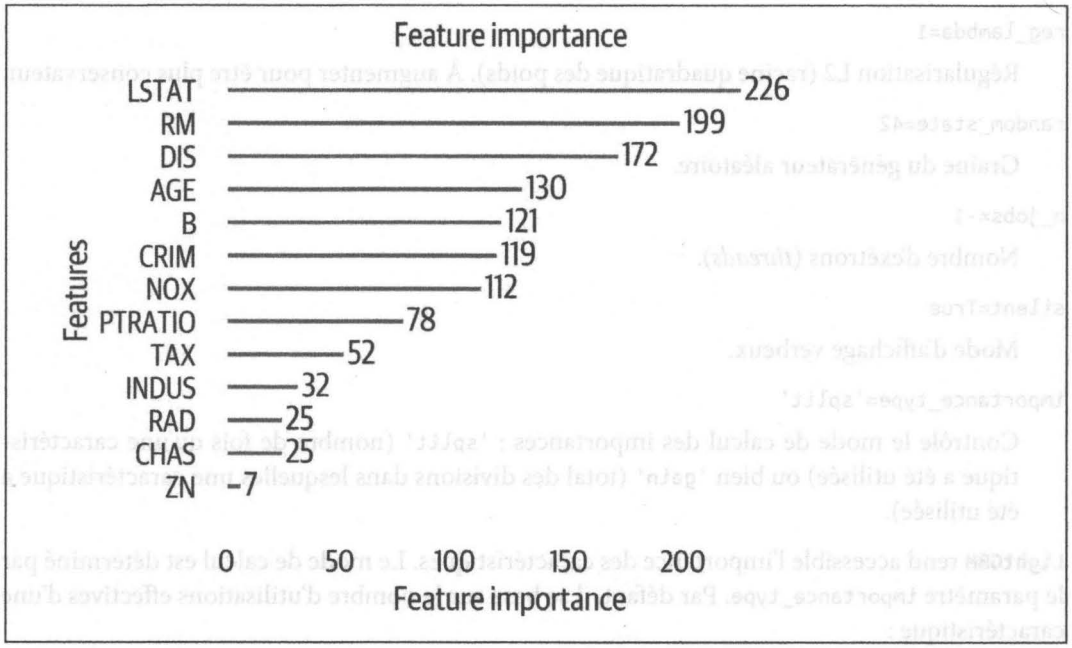



*Figure 14.8 : Visualisation de l'importance des caractéristiques montrant le nombre d'utilisations d'une caractéristique.*

CHAPITRE 15

# Métriques et évaluation des régressions



Ce chapitre montre comment évaluer les résultats d'une régression de forêt aléatoire. Les exemples se fondent sur le jeu de données des prix de l'immobilier dans la région de Boston :

```
>>> rfr = RandomForestRegressor(
...     random_state=42, n_estimators=100
... )
>>> rfr.fit(bos_X_train, bos_y_train)
```

## Métriques

Le module **`sklearn.metrics`** contient des moyens de mesure pour évaluer les modèles de régression. Toutes les fonctions métriques qui se terminent par `loss` ou `error` doivent être minimisées alors que celles qui se terminent par `score` doivent être maximisées.

Une métrique de régression habituelle est le *coefficient de détermination* ($r^2$) variant entre zéro et un. Il désigne le pourcentage de variance de la cible à laquelle les caractéristiques ont contribué. Plus la valeur est forte, mieux c'est, mais il n'est normalement pas facile d'évaluer un modèle uniquement avec cette métrique. Dans certains cas, une valeur de 0.7 peut constituer un bon score, mais pas toujours. Dans un autre cas, une valeur de 0.5 peut s'avérer satisfaisante, alors que dans un autre cas 0.9, il faudrait sera insuffisant. Il faut donc en général combiner cette valeur avec d'autres métriques ou des visualisations pour pouvoir évaluer un modèle.

Il est par exemple assez facile de créer un modèle pour prédire le niveau des actions du lendemain avec un coefficient $r^2$ de 0.99. Personnellement, je n'investirais pas mes économies sur ce genre de conseil. Il suffit que le modèle soit un peu trop pessimiste ou optimiste pour que le cours des actions s'en écarte fortement.

La métrique $r^2$ est celle utilisée par défaut dans les recherches grille. Pour demander d'autres métriques, vous utilisez le paramètre `scoring`.

Avec la méthode `.score`, vous pouvez demander le calcul pour les modèles de régression :

```
>>> from sklearn import metrics
>>> rfr.score(bos_X_test, bos_y_test)
0.8721182042634867

>>> metrics.r2_score(bos_y_test, bos_y_test_pred)
0.8721182042634867
```

Vous disposez également d'une *métrique de variance expliquée* (`'explained_variance'` dans les recherches grille). Si la moyenne des erreurs de prédiction que sont les résidus est égale à zéro (dans les modèles OLS, *Ordinary Least Square*), alors la variance expliquée équivaut au coefficient de détermination :

```
>>> metrics.explained_variance_score(
...     bos_y_test, bos_y_test_pred
... )
0.8724890451227875
```

L'*erreur moyenne absolue* (`'neg_mean_absolute_error'` dans les recherches grille) exprime la moyenne de l'erreur de prédiction du modèle absolue. Si le modèle était parfait, la valeur serait égale à zéro, mais il n'y a pas de limite supérieure à cette métrique, à la différence du coefficient de détermination. Elle est en revanche plus facile à interpréter parce qu'elle est exprimée en unités de la cible. C'est une bonne métrique lorsque vous voulez ignorer les valeurs aberrantes.

La mesure ne permet pas de savoir à quel point le modèle est défectueux, mais elle permet de comparer deux modèles. Dans ce cas, celui ayant le score le plus faible est le meilleur.

Dans l'exemple, cette métrique montre que l'erreur moyenne s'écarte d'environ deux unités au-dessus ou au-dessous de la valeur réelle :

```
>>> metrics.mean_absolute_error(
...     bos_y_test, bos_y_test_pred
... )
2.0839802631578945
```

La *racine de l'erreur quadratique moyenne* (`'neg_mean_squared_error'` dans les recherches grille) mesure aussi les erreurs du modèle en termes de cible, mais cette mesure cherche d'abord la moyenne des erreurs quadratiques avant d'en extraire la racine carrée, ce qui pénalise les erreurs importantes. C'est la métrique à utiliser si c'est votre but, par exemple lorsque vous voulez considérer qu'un écart de huit est plus de deux fois plus grave qu'un écart de quatre.

Comme l'erreur moyenne absolue, il n'est pas possible de déterminer à quel point le modèle est défectueux, mais il est possible de comparer deux modèles. C'est une bonne métrique si vous supposez que les erreurs sont distribuées normalement.

Dans l'exemple, la moyenne des erreurs quadratiques donne un résultat d'environ 9.5 :

```
>>> metrics.mean_squared_error(
...     bos_y_test, bos_y_test_pred
... )
9.52886846710526
```

L'*erreur quadratique logarithmique moyenne* (`'neg_mean_squared_log_error'` dans les recherches grille) pénalise plus la sous-prédiction que la surprédiction. C'est une métrique intéressante lorsque vous ciblez pour une croissance exponentielle (recensement de population, inventaire de stock, *etc.*).

Si vous partez du log de l'erreur et l'élevez au carré, la moyenne des résultats vaudra 0.021 :

```
>>> metrics.mean_squared_log_error(
...     bos_y_test, bos_y_test_pred
... )
0.02128263061776433
```

## Diagrammes des résidus

Un bon modèle (avec des scores $r^2$) est dit *homoscédastique*. Cela signifie que la variance reste la même pour toutes les valeurs des cibles quelles que soient les entrées. Dans un diagramme, les valeurs apparaissent réparties au hasard dans un diagramme de résidus. Si des motifs remarquables apparaissent, c'est qu'il y a un problème dans le modèle ou dans les données.

Les diagrammes de résidus montrent également les aberrants, qui peuvent avoir un impact important sur l'ajustement (Figure 15.1).

Voici comment produire un diagramme des résidus avec **`Yellowbrick`** :

```
>>> from yellowbrick.regressor import ResidualsPlot
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> rpv = ResidualsPlot(rfr)
>>> rpv.fit(bos_X_train, bos_y_train)
>>> rpv.score(bos_X_test, bos_y_test)
>>> rpv.poof()
>>> fig.savefig("images/mlpr_1501.png", dpi=300)
```

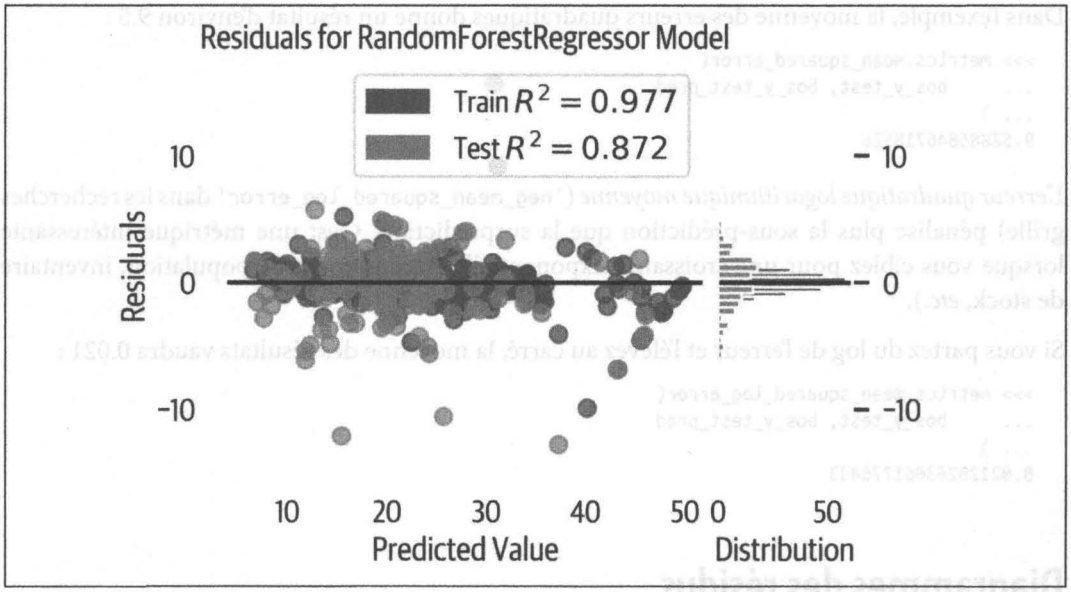



*Figure 15.1 : Diagramme des résidus. Des tests plus poussés montreront une hétéroscédasticité.*

# Hétéroscédasticité

Pour traquer une éventuelle *hétéroscédaticité*, vous disposez dans la librairie **`statsmodel`** (`https://oreil.ly/HtIi5`) d'un *test de Breusch-Pagan*. Ce défaut signifie que la variance des résidus n'est pas la même pour les différentes valeurs prédites. Dans le test de Breusch-Pagan, si les p-valeurs sont significatives, c'est-à-dire si `p-value` devient inférieure à 0.05, l'hypothèse de nullité de l'homoscédasticité est rejetée. Cela confirme que les résidus sont hétéroscédastiques et que les prédictions sont biaisées.

Voici comment confirmer une hétéroscédasticité :

```
>>> import statsmodels.stats.api as sms
>>> hb = sms.het_breuschpagan(resids, bos_X_test)
>>> labels = [
...     "Lagrange multiplier statistic",
...     "p-value",
...     "f-value",
...     "f p-value",
... ]
>>> for name, num in zip(name, hb):
...     print(f"{name}: {num:.2}")
Lagrange multiplier statistic: 3.6e+01
p-value: 0.00036
f-value: 3.3
f p-value: 0.00022
```

## Résidus normaux

La librairie **scipy** propose un diagramme de probabilités ainsi qu'un test de Kolmogorov-Smirnov. Tous deux permettent d'apprécier la normalité des résidus.

Il est possible de visualiser les résidus pour vérifier leur normalité en traçant un histogramme (Figure 15.2) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> resids = bos_y_test - rfr.predict(bos_X_test)
>>> pd.Series(resids, name="residuals").plot.hist(
...     bins=20, ax=ax, title="Residual Histogram"
... )
>>> fig.savefig("images/mlpr_1502.png", dpi=300)
```

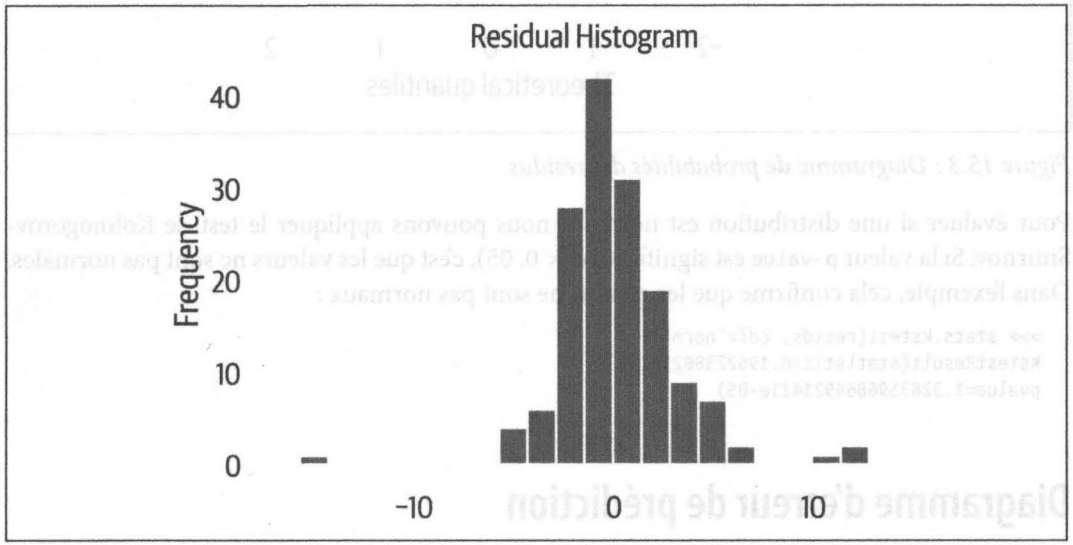



*Figure 15.2 : Histogramme de résidus.*

La Figure 15.3 propose un diagramme de probabilités. Les résidus sont considérés comme normaux si les échantillons tracés au long des quantiles sont alignés. Dans l'exemple, nous voyons que ce n'est pas le cas :

```
>>> from scipy import stats
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> _ = stats.probplot(resids, plot=ax)
>>> fig.savefig("images/mlpr_1503.png", dpi=300)
```

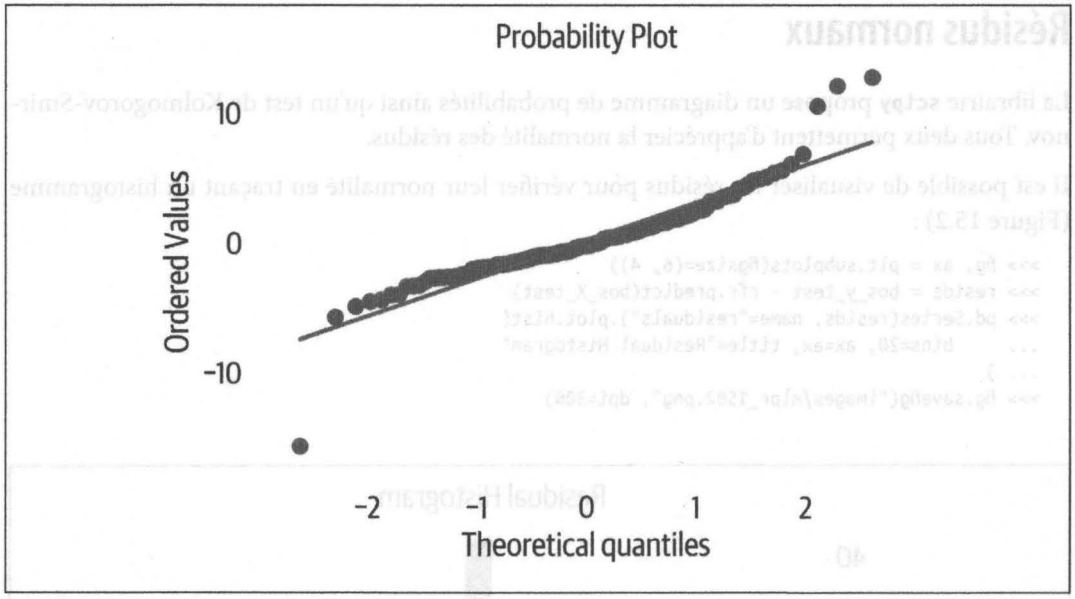



*Figure 15.3 : Diagramme de probabilités des résidus.*

Pour évaluer si une distribution est normale, nous pouvons appliquer le test de Kolmogorov-Smirnov. Si la valeur `p-value` est significative (< 0. 05), c'est que les valeurs ne sont pas normales. Dans l'exemple, cela confirme que les résidus ne sont pas normaux :

```
>>> stats.kstest(resids, cdf="norm")
KstestResult(statistic=0.1962230021010155,
pvalue=1.3283596864921421e-05)
```

# Diagramme d'erreur de prédiction

Un diagramme d'erreur de prédiction visualise les cibles réelles par rapport aux valeurs prédites. Elles sont parfaitement alignées en diagonale à 45 ° si le modèle est parfait.

Notre modèle d'exemple semble prédire des valeurs plus faibles pour la partie supérieure de y ; le modèle a donc des problèmes de performances. Le diagramme des résidus le confirme (Figure 15.4).

Voici la visualisation avec **`Yellowbrick`** :

```
>>> from yellowbrick.regressor import (
...     PredictionError,
... )
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> pev = PredictionError(rfr)
```

```
>>> pev.fit(bos_X_train, bos_y_train)
>>> pev.score(bos_X_test, bos_y_test)
>>> pev.poof()
>>> fig.savefig("images/mlpr_1504.png", dpi=300)
```

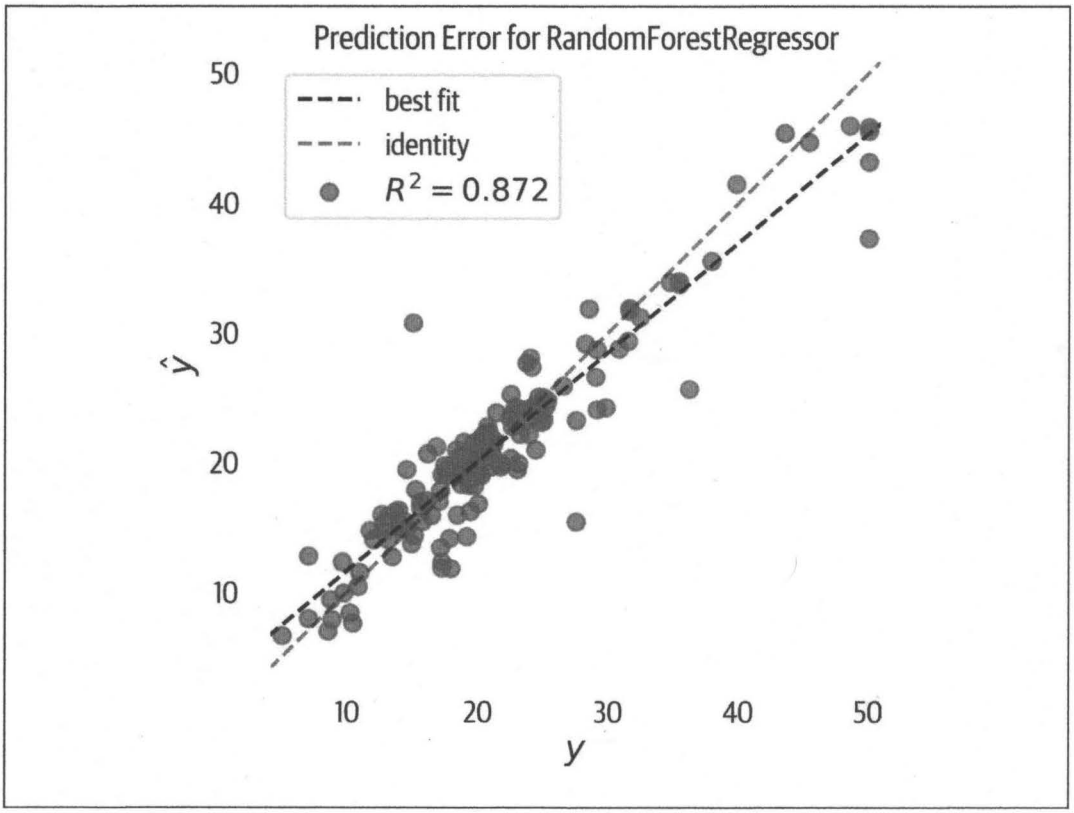



*Figure 15.4 : Diagramme d'erreur de prédiction. Points prédits en y (y-hat) par rapport aux y réels.*

CHAPITRE 16

# Explication des modèles de régression

La plupart des techniques qui servent à expliquer les modèles de classification s'appliquent aux modèles de régression. Nous allons voir dans ce chapitre comment utiliser la librairie SHAP pour interpréter un modèle de régression.

Pour la mise en pratique, nous allons interpréter un modèle **`XGBoost`** travaillant sur le jeu de données immobilières Boston. Nous commençons par préparer ces données :

```
>>> import xgboost as xgb
>>> xgr = xgb.XGBRegressor(
...     random_state=42, base_score=0.5
... )
>>> xgr.fit(bos_X_train, bos_y_train)
```

## Shapley

J'apprécie particulièrement l'approche Shapley parce qu'elle ne privilégie pas un modèle plutôt qu'un autre et procure une vision globale du modèle tout en permettant d'expliquer les prédictions individuelles. La librairie est tout à fait appropriée à un modèle de type boîte noire.

Commençons par nous intéresser à la prédiction pour l'indice 5. Notre modèle lui prédit la valeur 27.26 :

```
>>> sample_idx = 5
>>> xgr.predict(bos_X.iloc[[sample_idx]])
array([27.269186], dtype=float32)
```

Pour pouvoir utiliser le modèle, nous devons créer un composant explicateur `TreeExplainer` à partir du modèle puis estimer les valeurs SHAP des échantillons. Si vous utilisez Jupyter avec une interface interactive, il faut penser à appeler la fonction `initjs` :

```
>>> import shap
>>> shap.initjs()

>>> exp = shap.TreeExplainer(xgr)
>>> vals = exp.shap_values(bos_X)
```

Une fois que nous avons l'élément explicateur ainsi que les valeurs SHAP, nous pouvons demander la création d'un *diagramme de force* pour expliquer les prédictions (Figure 16.1). Nous pouvons ainsi voir que la prédiction de base était de 23. Le niveau de vie de la population LSTAT et le taux de taxe foncière TAX font monter le prix alors que le nombre de pièces RM le fait descendre :

```
>>> shap.force_plot(
...     exp.expected_value,
...     vals[sample_idx],
...     bos_X.iloc[sample_idx],
... )
```

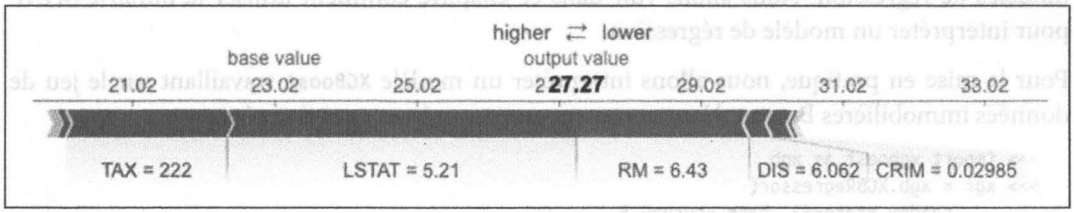



*Figure 16.1 : Diagramme de force pour une régression. La valeur passe de 23 à 27 sous l'influence du niveau de vie et du taux de taxe.*

Pour une appréciation globale du comportement du modèle, nous pouvons visualiser le diagramme de force pour tous les échantillons dans le mode interactif JavaScript de Jupyter ; nous pouvons même balayer les différents échantillons avec la souris et voir quelles caractéristiques ont un impact sur le résultat (Figure 16.2) :

```
>>> shap.force_plot(
...     exp.expected_value, vals, bos_X
... )
```

Le diagramme de l'échantillon isolé nous a permis de constater que LSTAT avait un impact important. Pour le confirmer, nous créons un diagramme de dépendance. La librairie va automatiquement choisir une caractéristique pour l'afficher en couleur (vous pouvez contrôler la couleur au moyen de `interaction_index`).

Le diagramme de dépendance de LSTAT (Figure 16.3) nous apprend que lorsque LSTAT augmente (pourcentage de ménages à faibles revenus), la valeur SHAP diminue, et fait diminuer celle de la cible. Une valeur de LSTAT très faible fait croître SHAP. La couleur de TAX montre que lorsque le taux de taxe diminue (devient plus bleu), la valeur SHAP augmente :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.dependence_plot("LSTAT", vals, bos_X)
>>> fig.savefig(
...     "images/mlpr_1603.png",
...     bbox_inches="tight",
...     dpi=300,
... )
```

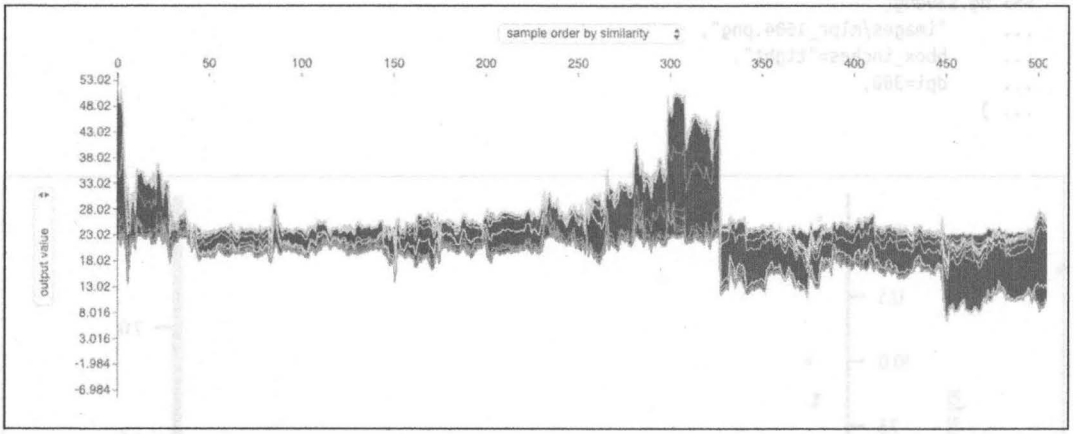



*Figure 16.2 : Diagramme de force de régression pour tous les échantillons.*

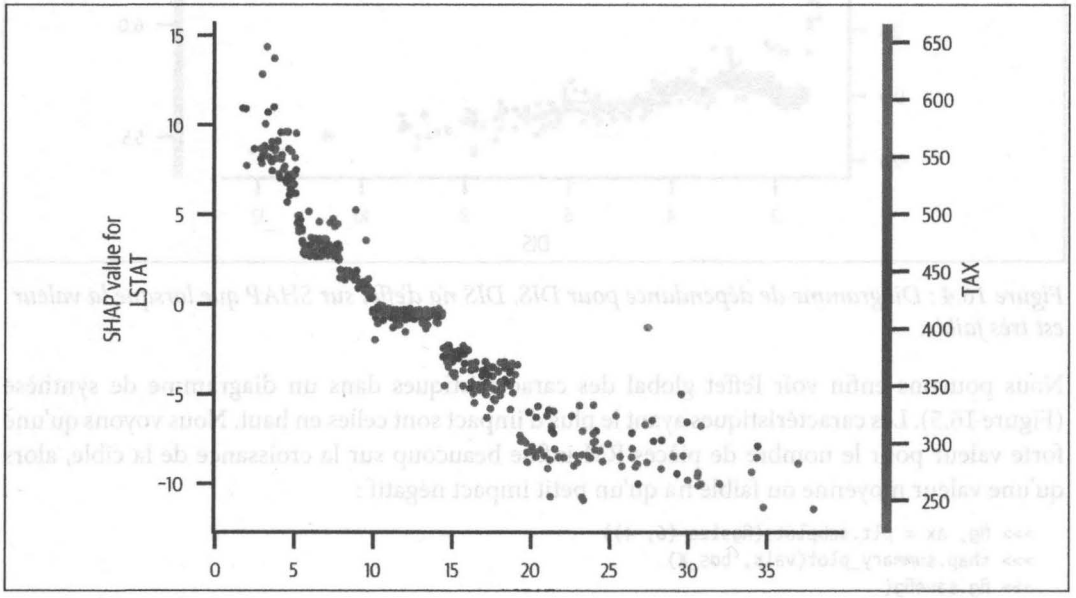



*Figure 16.3 : Diagramme de dépendance pour LSTAT. La valeur diminue lorsque LSTAT augmente.*

La Figure 16.4 montre un autre diagramme de dépendance pour la distance aux agences pour l'emploi, DIS. Nous constatons que cette caractéristique n'a un impact sensible que lorsque la valeur est très faible :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.dependence_plot(
...     "DIS", vals, bos_X, interaction_index="RM"
... )
>>> fig.savefig(
...     "images/mlpr_1604.png",
...     bbox_inches="tight",
...     dpi=300,
... )
```

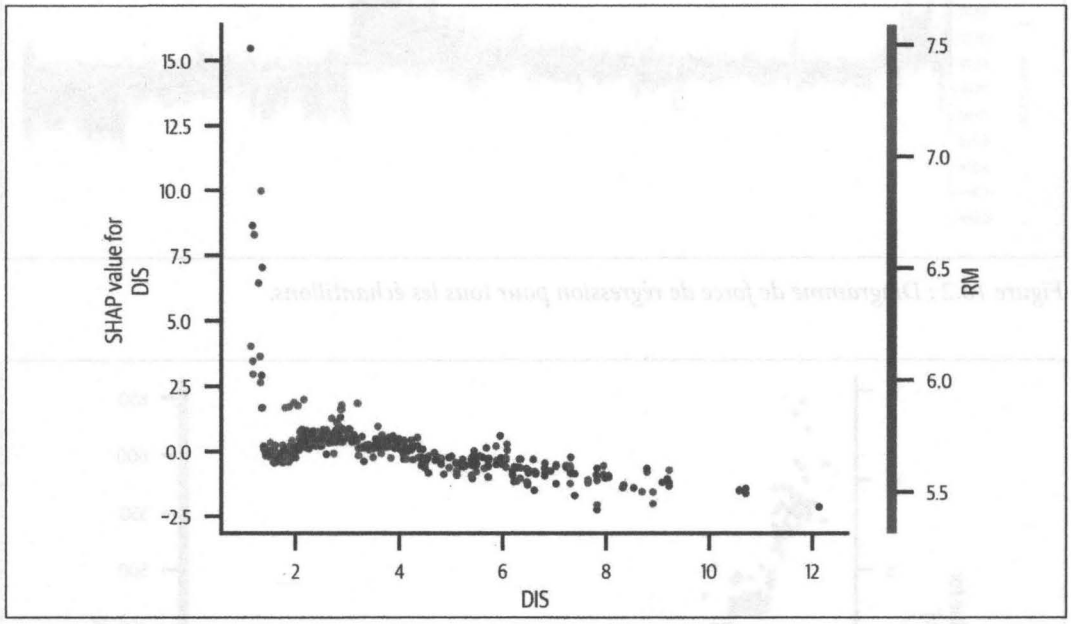



*Figure 16.4 : Diagramme de dépendance pour DIS. DIS n'a d'effet sur SHAP que lorsque la valeur est très faible.*

Nous pouvons enfin voir l'effet global des caractéristiques dans un diagramme de synthèse (Figure 16.5). Les caractéristiques ayant le plus d'impact sont celles en haut. Nous voyons qu'une forte valeur pour le nombre de pièces RM influe beaucoup sur la croissance de la cible, alors qu'une valeur moyenne ou faible n'a qu'un petit impact négatif :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.summary_plot(vals, bos_X)
>>> fig.savefig(
...     "images/mlpr_1605.png",
...     bbox_inches="tight",
```

```
...     dpi=300,
... )
```

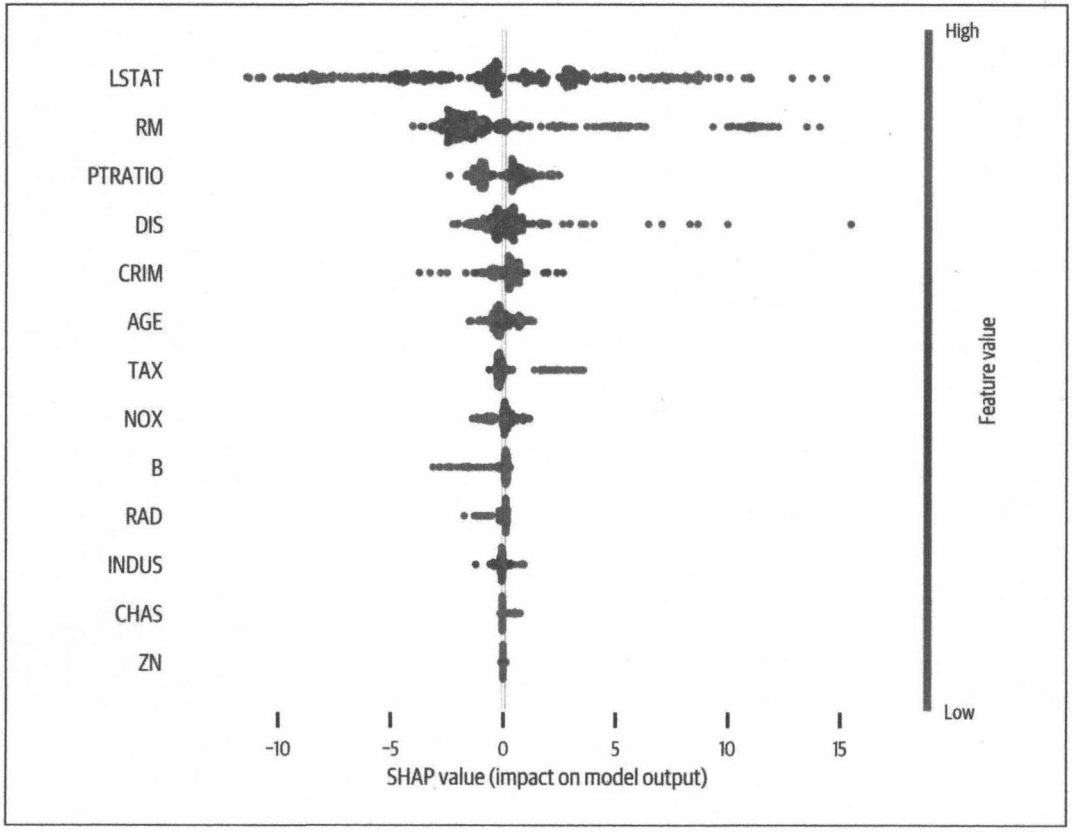



*Figure 16.5 : Diagramme de synthèse. Les caractéristiques les plus impactantes sont en haut.*

La librairie SHAP est donc un excellent outil à garder sous la main. Elle aide à mieux comprendre l'impact global des caractéristiques tout en permettant d'étudier les prédictions individuelles.

CHAPITRE 17

# Réduction de la dimensionnalité

Plusieurs techniques sont disponibles pour décomposer des caractéristiques afin d'obtenir un sous-ensemble moins grand. Ce genre de traitement est utile dans les analyses exploratoires de données, dans la visualisation, dans la création de modèles prédictifs et dans le partitionnement ou regroupement (*clustering*).

Nous allons dans ce chapitre travailler sur le jeu de données Titanic pour découvrir plusieurs techniques : PCA, UMAP, t-SNE et PHATE.

Voici d'abord les données de travail :

```
>>> ti_df = tweak_titanic(orig_df)
>>> std_cols = "pclass,age,sibsp,fare".split(",")
>>> X_train, X_test, y_train, y_test = get_train_test_X_y(
...     ti_df, "survived", std_cols=std_cols
... )
>>> X = pd.concat([X_train, X_test])
>>> y = pd.concat([y_train, y_test])
```

## PCA

L'analyse par composantes principales PCA (*Principal Component Analysis*) traite une matrice X constituée de lignes pour les échantillons et de colonnes pour les caractéristiques. La PCA renvoie une autre matrice dont les colonnes sont des combinaisons linéaires des colonnes de départ, ces combinaisons linéaires produisant une variance maximale.

Chaque colonne est à angle droit, donc orthogonale des autres. Les colonnes sont triées dans l'ordre des variances décroissantes.

Ce modèle est disponible dans la librairie **`scikit-learn`**. Il est conseillé de standardiser les données lors du prétraitement. Après avoir appelé la méthode `.fit`, vous accédez à l'attribut nommé

`.explained_variance_ratio_` qui donne la liste des pourcentages de variance dans chaque colonne.

PCA permet de visualiser des données dans deux ou trois dimensions et de filtrer le bruit aléatoire dans les données pendant le prétraitement. Elle permet bien de trouver des structures globales, mais pas des motifs locaux ; elle convient bien aux données linéaires.

Nous allons appliquer PCA aux caractéristiques du jeu Titanic. La classe nommée `PCA` est une des classes transformatrices de la librairie **`scikit-learn`**. Vous appelez d'abord la méthode `.fit` pour lui apprendre comment accéder aux composantes principales, puis vous appelez `.transform` pour convertir la matrice de départ vers la matrice de composantes principales :

```
>>> from sklearn.decomposition import PCA
>>> from sklearn.preprocessing import (
...     StandardScaler,
... )
>>> pca = PCA(random_state=42)
>>> X_pca = pca.fit_transform(
...     StandardScaler().fit_transform(X) ... )
>>> pca.explained_variance_ratio_
array([0.23917891, 0.21623078, 0.19265028,
0.10460882, 0.08170342, 0.07229959,
0.05133752, 0.04199068])

>>> pca.components_[0]
arrayarray([-0.63368693, 0.39682566,
0.00614498, 0.11488415, 0.58075352,
-0.19046812, -0.21190808, -0.09631388])
```

### Paramètres d'instance

`n_components=None`

Nombre de composantes à générer. Avec la valeur `None`, renvoie le même nombre que le nombre de colonnes. Si vous fournissez une valeur float (0,1), seront créées autant de composantes que nécessaire pour atteindre ce ratio de variance.

`copy=True`

Réalise une mutation des données sur `.fit` si True.

`whiten=False`

Blanchit les données après la transformation pour garantir que les composantes sont non corrélées.

`svd_solver='auto'`

La valeur `'auto'` fait exécuter SVD en `'randomized'` si `n_components` vaut moins de 80 % de la plus petite dimension (plus rapide, mais avec approximation). Sinon, exécution en mode `'full'`.

`tol=0.0`

Tolérance pour les valeurs singulières.

`iterated_power='auto'`

Nombre d'itérations pour le solveur randomisé `svd_solver`.

`random_state=None`

Statut aléatoire pour le solveur randomisé `svd_solver`.

### Attributs

`components_`

Composantes principales. La ligne 0 donne les poids pour PC1, la ligne 1 pour PC2, etc. Chaque colonne est le poids de la colonne de départ correspondante. Plus la valeur absolue du poids est grande, plus la colonne de départ a d'impact sur la composante principale de cette ligne..

`explained_variance_`

Variance par composante.

`explained_variance_ratio_`

Variance par composante après normalisation (total de 1).

`singular_values_`

Valeurs singulières pour chaque composante.

`mean_`

Moyenne de chaque caractéristique.

`n_components_`

Si `n_components` contient une valeur `float`, ceci est la taille des composantes.

`noise_variance_`

Covariance de bruit estimée.

Le fait de créer un tracé de la somme cumulée du ratio de variance expliquée correspond à un diagramme *scree* (Figure 17.1). Ce diagramme permet de voir combien d'informations sont stockées dans les composantes. La méthode du coude permet de décider du nombre de composantes à utiliser :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> ax.plot(pca.explained_variance_ratio_)
>>> ax.set(
...     xlabel="Composante",
```

```
...     ylabel="Pourcentage de variance expliquée",
...     title="Scree Plot",
...     ylim=(0, 1),
... )
>>> fig.savefig(
...     "images/mlpr_1701.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

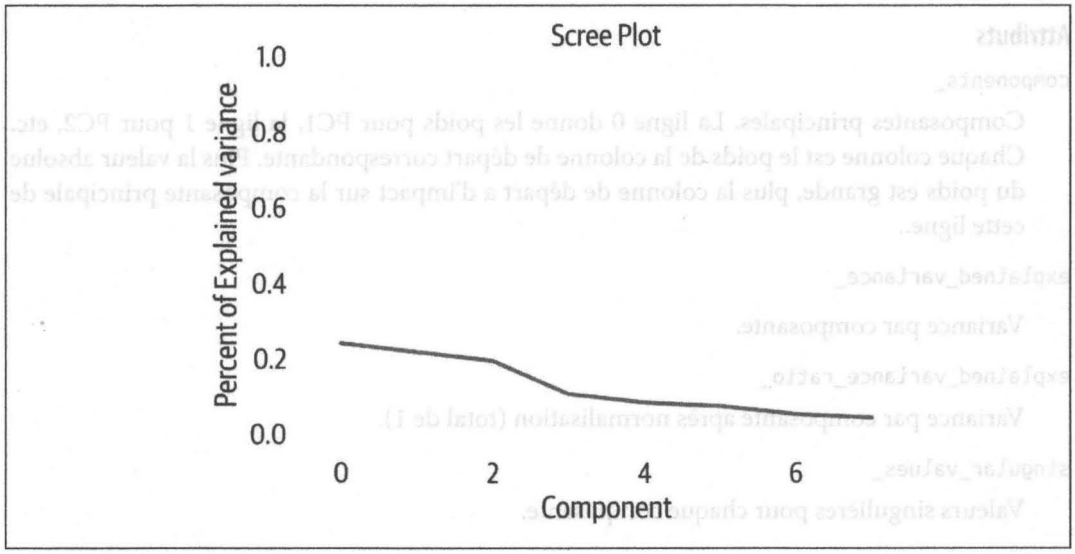



*Figure 17.1 : Diagramme scree de PCA.*

Un autre point de vue sur les données correspond au diagramme cumulatif (Figure 17.2). Les données de départ possèdent huit colonnes, mais le diagramme confirme que nous préservons environ 90 % de la variance en n'utilisant que quatre composantes PCA :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> ax.plot(
...     np.cumsum(pca.explained_variance_ratio_)
... )
>>> ax.set(
...     xlabel="Composante",
...     ylabel="Pourcentage de variance expliquée",
...     title="Variance cumulée",
...     ylim=(0, 1),
... )
>>> fig.savefig("images/mlpr_1702.png", dpi=300)
```

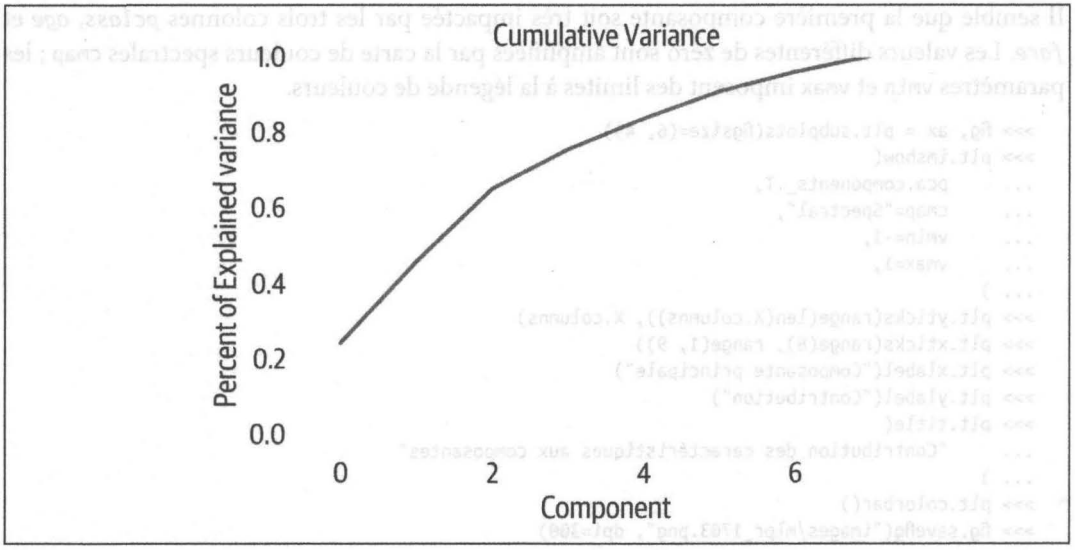



*Figure 17.2 : Diagramme cumulatif de la variance expliquée PCA.*

Pour savoir à quel point les caractéristiques ont un impact sur les composantes, nous pouvons utiliser la fonction de **`matplotlib`** nommée `imshow` qui va tracer les composantes le long de l'axe x et les caractéristiques de départ le long de l'axe y (Figure 17.3). Plus la couleur est sombre, plus la colonne de départ contribue à la composante.

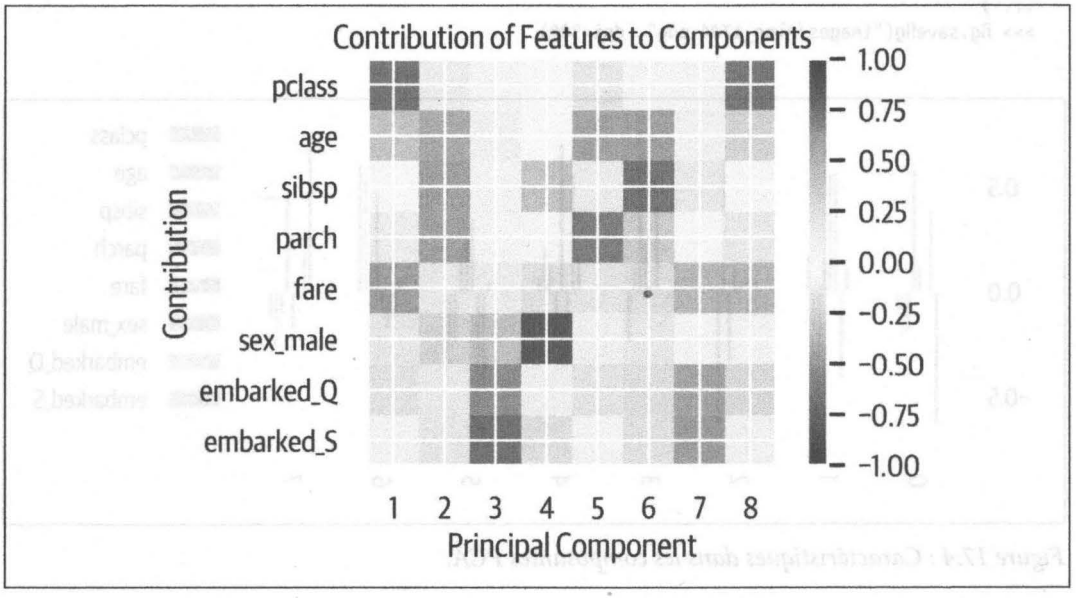



*Figure 17.3 : Impact des caractéristiques sur les composantes PCA.*

Il semble que la première composante soit très impactée par les trois colonnes *pclass*, *age* et *fare*. Les valeurs différentes de zéro sont amplifiées par la carte de couleurs spectrales `cmap` ; les paramètres `vmin` et `vmax` imposent des limites à la légende de couleurs.

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> plt.imshow(
...     pca.components_.T,
...     cmap="Spectral",
...     vmin=-1,
...     vmax=1,
... )
>>> plt.yticks(range(len(X.columns)), X.columns)
>>> plt.xticks(range(8), range(1, 9))
>>> plt.xlabel("Composante principale")
>>> plt.ylabel("Contribution")
>>> plt.title(
...     "Contribution des caractéristiques aux composantes"
... )
>>> plt.colorbar()
>>> fig.savefig("images/mlpr_1703.png", dpi=300)
```

Une autre approche consiste à demander un diagramme en barres (Figure 17.4). Dans ce cas, chaque composante est visualisée avec les contributions des données de départ :

```
>>> fig, ax = plt.subplots(figsize=(8, 4))
>>> pd.DataFrame(
...     pca.components_, columns=X.columns
... ).plot(kind="bar", ax=ax).legend(
...     bbox_to_anchor=(1, 1)
... )
>>> fig.savefig("images/mlpr_1704.png", dpi=300)
```

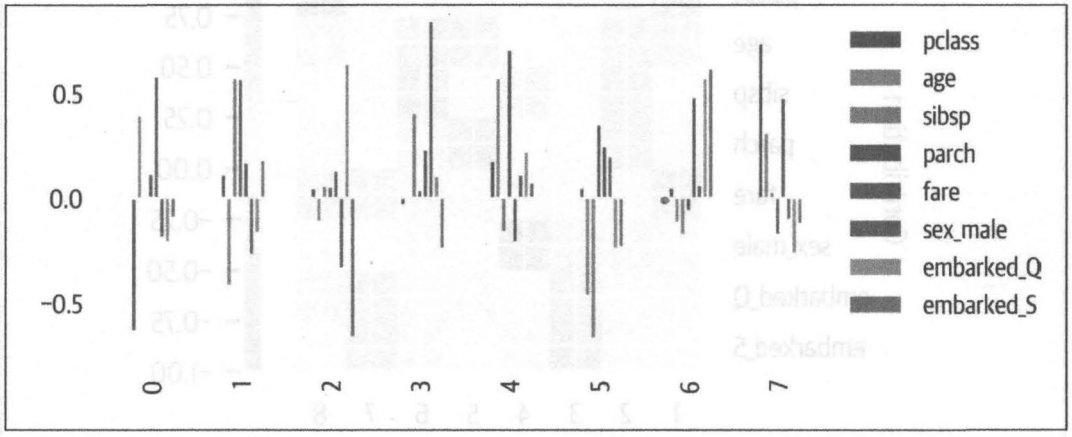



*Figure 17.4 : Caractéristiques dans les composantes PCA.*

Dans le cas d'un grand nombre de caractéristiques, il n'est pas inutile de limiter l'affichage à celles qui ont un poids minimal. Le code source suivant cherche toutes les caractéristiques dans les deux premières composantes dont la valeur absolue est au moins égale à 0.5 :

```
>>> comps = pd.DataFrame(
...     pca.components_, columns=X.columns
... )
>>> min_val = 0.5
>>> num_components = 2
>>> pca_cols = set()
>>> for i in range(num_components):
...     parts = comps.iloc[i][
...         comps.iloc[i].abs() > min_val
...     ]
...     pca_cols.update(set(parts.index))
>>> pca_cols
{'fare', 'parch', 'pclass', 'sibsp'}
```

La technique PCA est souvent utilisée pour visualiser des jeux de données à forte dimensionnalité dans deux composantes. Voyons par exemple comment visualiser les caractéristiques de Titanic en 2D. Nous les faisons colorer selon le statut de survie. Parfois, des regroupements apparaissent. Dans l'exemple, il ne semble pas y avoir de regroupement des survivants (Figure 17.5).

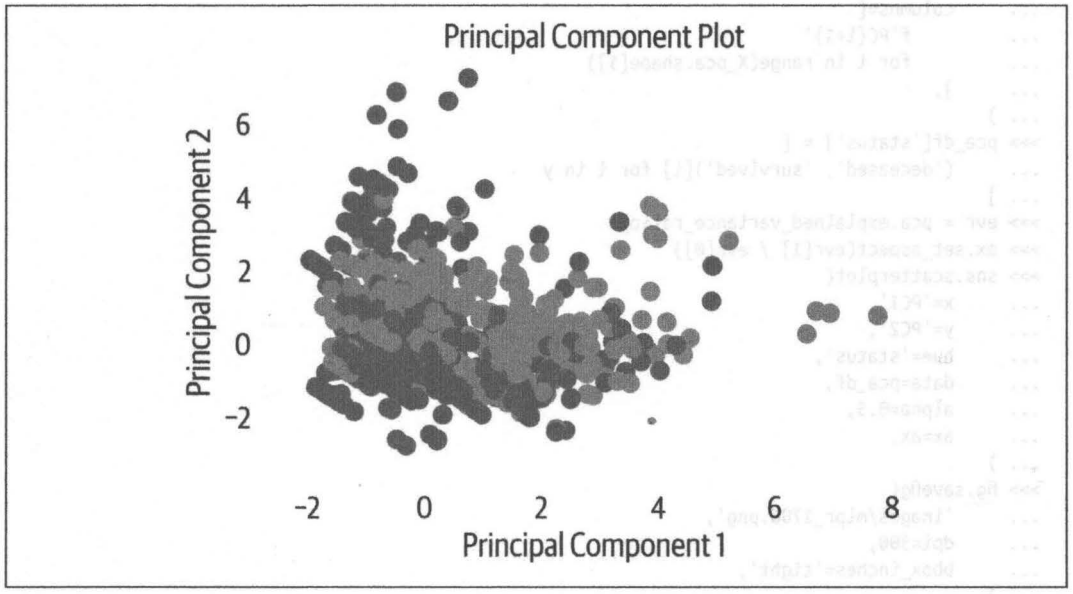



*Figure 17.5 : Diagramme PCA avec Yellowbrick.*

Nous obtenons cette visualisation grâce à **Yellowbrick** :

```
>>> from yellowbrick.features.pca import (
...     PCADecomposition,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> colors = ["rg"[j] for j in y]
>>> pca_viz = PCADecomposition(color=colors)
>>> pca_viz.fit_transform(X, y)
>>> pca_viz.poof()
>>> fig.savefig("images/mlpr_1705.png", dpi=300)
```

Si vous voulez colorier le diagramme en nuage de points selon une colonne et ajouter une légende au lieu d'une barre de couleurs, il faut parcourir les différentes couleurs en boucle pour tracer le groupe de façon individuelle dans **pandas** ou **matplotlib** (ou bien utiliser **seaborn**). Dans l'exemple, nous en profitons pour régler le ratio d'aspect à celui des variances expliquées pour les composantes que nous visualisons (Figure 17.6). Notez que la seconde composante est plus courte, car elle ne correspond qu'à 90 % de la première.

Voici donc la version avec **seaborn** :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pca_df = pd.DataFrame(
...     X_pca,
...     columns=[
...         f'PC{i+1}'
...         for i in range(X_pca.shape[1])
...     ],
... )
>>> pca_df['status'] = [
...     ('deceased', 'survived')[i] for i in y
... ]
>>> evr = pca.explained_variance_ratio_
>>> ax.set_aspect(evr[1] / evr[0])
>>> sns.scatterplot(
...     x='PC1',
...     y='PC2',
...     hue='status',
...     data=pca_df,
...     alpha=0.5,
...     ax=ax,
... )
>>> fig.savefig(
...     'images/mlpr_1706.png',
...     dpi=300,
...     bbox_inches='tight',
... )
```

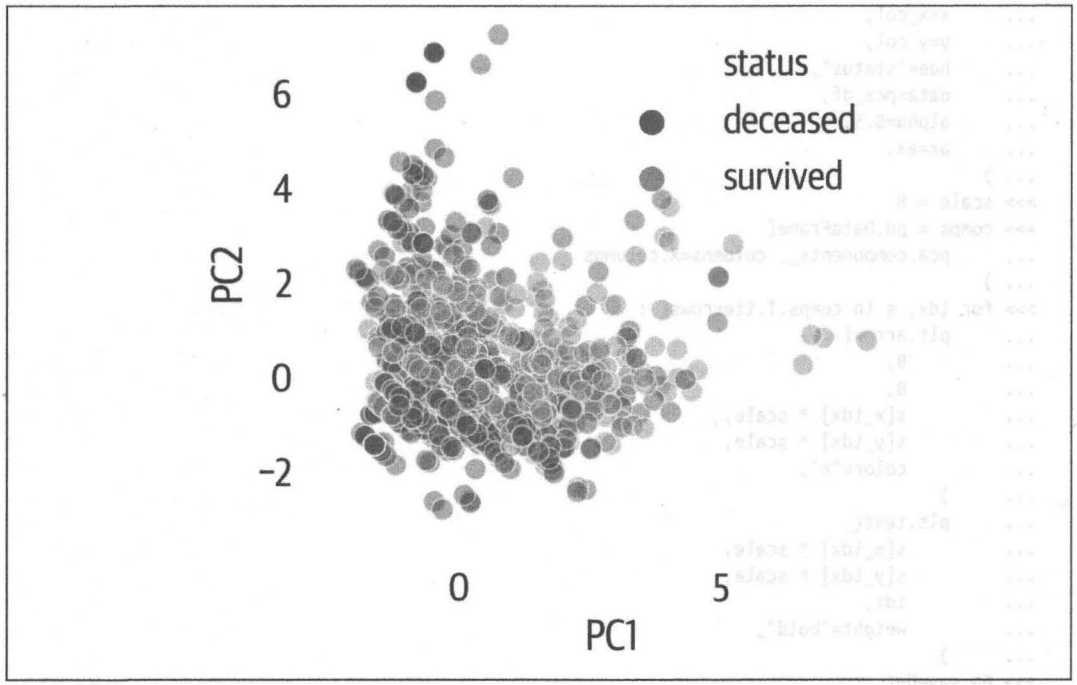



*Figure 17.6 : PCA avec légendes et aspect relatif (avec seaborn).*

Nous pouvons enrichir notre nuage en y ajoutant un diagramme de charge, ce qui en fait un diagramme combo (*biplot*). Il combine en effet un nuage de points et un diagramme de charge (Figure 17.7). La charge permet de connaître la force des caractéristiques et en quoi elles sont corrélées. Les corrélations sont fortes lorsque les angles sont aigus. Avec un angle droit, la corrélation est peu probable et dans le cas d'un alignement à 180 °, c'est une corrélation négative :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pca_df = pd.DataFrame(
...     X_pca,
...     columns=[
...         f"PC{i+1}"
...         for i in range(X_pca.shape[1])
...     ],
... )
>>> pca_df["status"] = [
...     ("deceased", "survived")[i] for i in y
... ]
>>> evr = pca.explained_variance_ratio_
>>> x_idx = 0 # x_pc
>>> y_idx = 1 # y_pc
>>> ax.set_aspect(evr[y_idx] / evr[x_idx])
>>> x_col = pca_df.columns[x_idx]
>>> y_col = pca_df.columns[y_idx]
>>> sns.scatterplot(
```

```
...     x=x_col,
...     y=y_col,
...     hue="status",
...     data=pca_df,
...     alpha=0.5,
...     ax=ax,
... )
>>> scale = 8
>>> comps = pd.DataFrame(
...     pca.components_, columns=X.columns
... )
>>> for idx, s in comps.T.iterrows():
...     plt.arrow(
...         0,
...         0,
...         s[x_idx] * scale,
...         s[y_idx] * scale,
...         color="k",
...     )
...     plt.text(
...         s[x_idx] * scale,
...         s[y_idx] * scale,
...         idx,
...         weight="bold",
...     )
>>> fig.savefig(
...     "images/mlpr_1707.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

Les diagrammes en arbres déjà réalisés nous ont permis d'apprendre que les trois colonnes *age*, *fare* et *sex* avaient un fort impact sur la survie d'un passager. La première composante principale est influencée par *pclass*, *age* et *fare* alors que la quatrième est influencée par *sex*. Visualisons ces deux composantes.

Comme le précédent, ce diagramme utilise comme ratio d'aspect les ratios des variances des composantes (Figure 17.8).

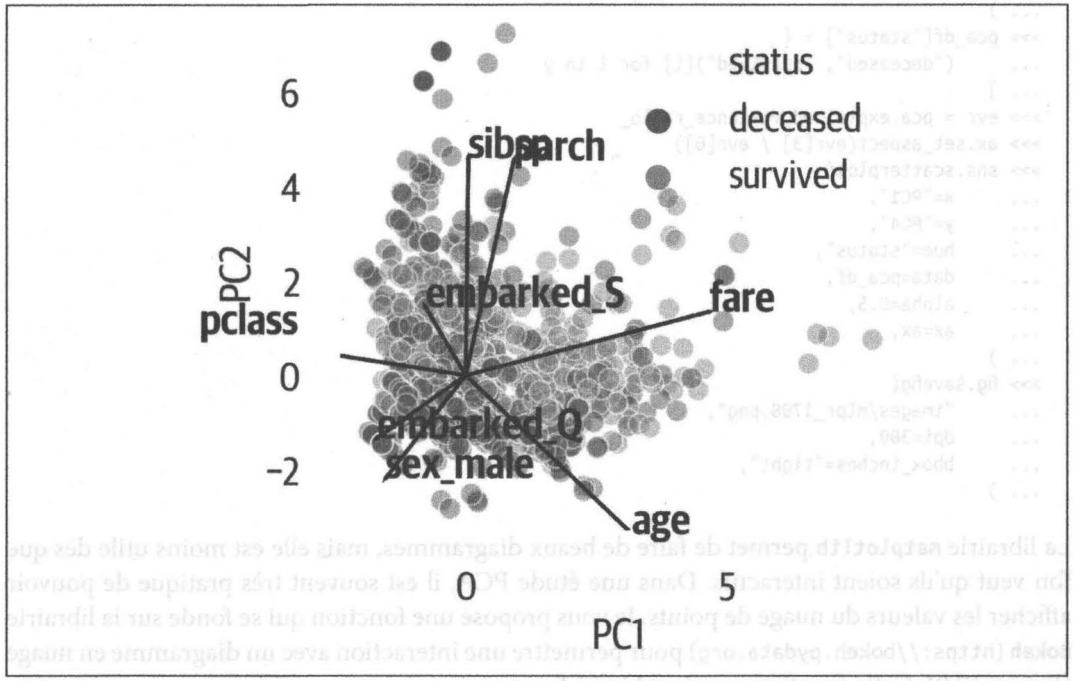



*Figure 17.7 : Diagramme combiné nuage et charges avec seaborn.*

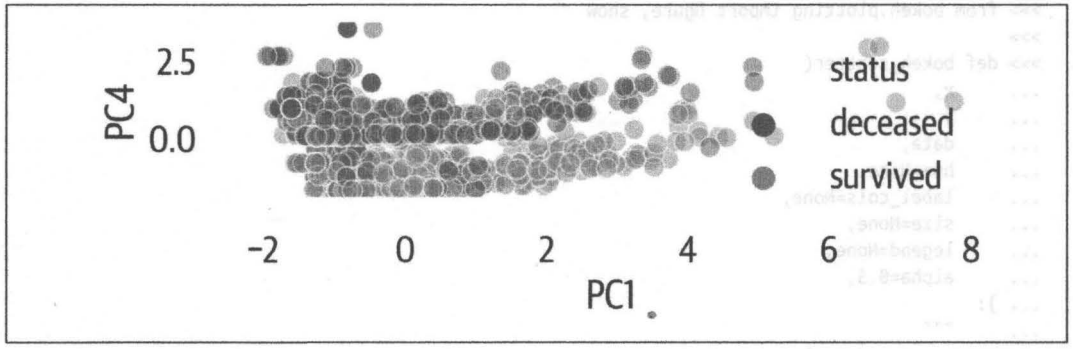



*Figure 17.8 : Diagramme PCA pour la composante 1 confrontée à la 4.*

Il semble que le diagramme sépare de façon plus exacte les survivants :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pca_df = pd.DataFrame(
...     X_pca,
...     columns=[
...         f"PC{i+1}"
...         for i in range(X_pca.shape[1])
...     ],
```

```
... )
>>> pca_df["status"] = [
...     ("deceased", "survived")[i] for i in y
... ]
>>> evr = pca.explained_variance_ratio_
>>> ax.set_aspect(evr[3] / evr[0])
>>> sns.scatterplot(
...     x="PC1",
...     y="PC4",
...     hue="status",
...     data=pca_df,
...     alpha=0.5,
...     ax=ax,
... )
>>> fig.savefig(
...     "images/mlpr_1708.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

La librairie **`matplotlib`** permet de faire de beaux diagrammes, mais elle est moins utile dès que l'on veut qu'ils soient interactifs. Dans une étude PCA, il est souvent très pratique de pouvoir afficher les valeurs du nuage de points. Je vous propose une fonction qui se fonde sur la librairie **`Bokeh`** (`https://bokeh.pydata.org`) pour permettre une interaction avec un diagramme en nuage (Figure 17.9). Cette fonction convient bien à Jupyter :

```
>>> from bokeh.io import output_notebook
>>> from bokeh import models, palettes, transform
>>> from bokeh.plotting import figure, show
>>>
>>> def bokeh_scatter(
...     x,
...     y,
...     data,
...     hue=None,
...     label_cols=None,
...     size=None,
...     legend=None,
...     alpha=0.5,
... ):
...     """
...     x - nom de colonne en x
...     y - nom de colonne en y
...     data - DataFrame pandas
...     hue - numéro de colonne pour couleur
...     legend - nom de colonne pour label
...     label_cols - colonnes pour tooltip
...     (None toutes dans DataFrame)
...     size - taille de points en unités écran
...     alpha - transparence
...     """
...     output_notebook()
```

```
...     circle_kwargs = {}
...     if legend:
...         circle_kwargs["legend"] = legend
...     if size:
...         circle_kwargs["size"] = size
...     if hue:
...         color_seq = data[hue]
...         mapper = models.LinearColorMapper(
...             palette=palettes.viridis(256),
...             low=min(color_seq),
...             high=max(color_seq),
...         )
...         circle_kwargs[
...             "fill_color"
...         ] = transform.transform(hue, mapper)
...     ds = models.ColumnDataSource(data)
...     if label_cols is None:
...         label_cols = data.columns
...     tool_tips = sorted(
...         [
...             (x, "@{}".format(x))
...             for x in label_cols
...         ],
...         key=lambda tup: tup[0],
...     )
...     hover = models.HoverTool(
...         tooltips=tool_tips
...     )
...     fig = figure(
...         tools=[
...             hover,
...             "pan",
...             "zoom_in",
...             "zoom_out",
...             "reset",
...         ],
...         toolbar_location="below",
...     )
...     fig.circle(
...         x,
...         y,
...         source=ds,
...         alpha=alpha,
...         **circle_kwargs
...     )
...     show(fig)
...     return fig
>>> res = bokeh_scatter(
...     "PC1",
...     "PC2",
...     data=pca_df.assign(
...         surv=y.reset_index(drop=True)
...     ),
```

```
...     hue="surv",
...     size=10,
...     legend="surv",
... )
```

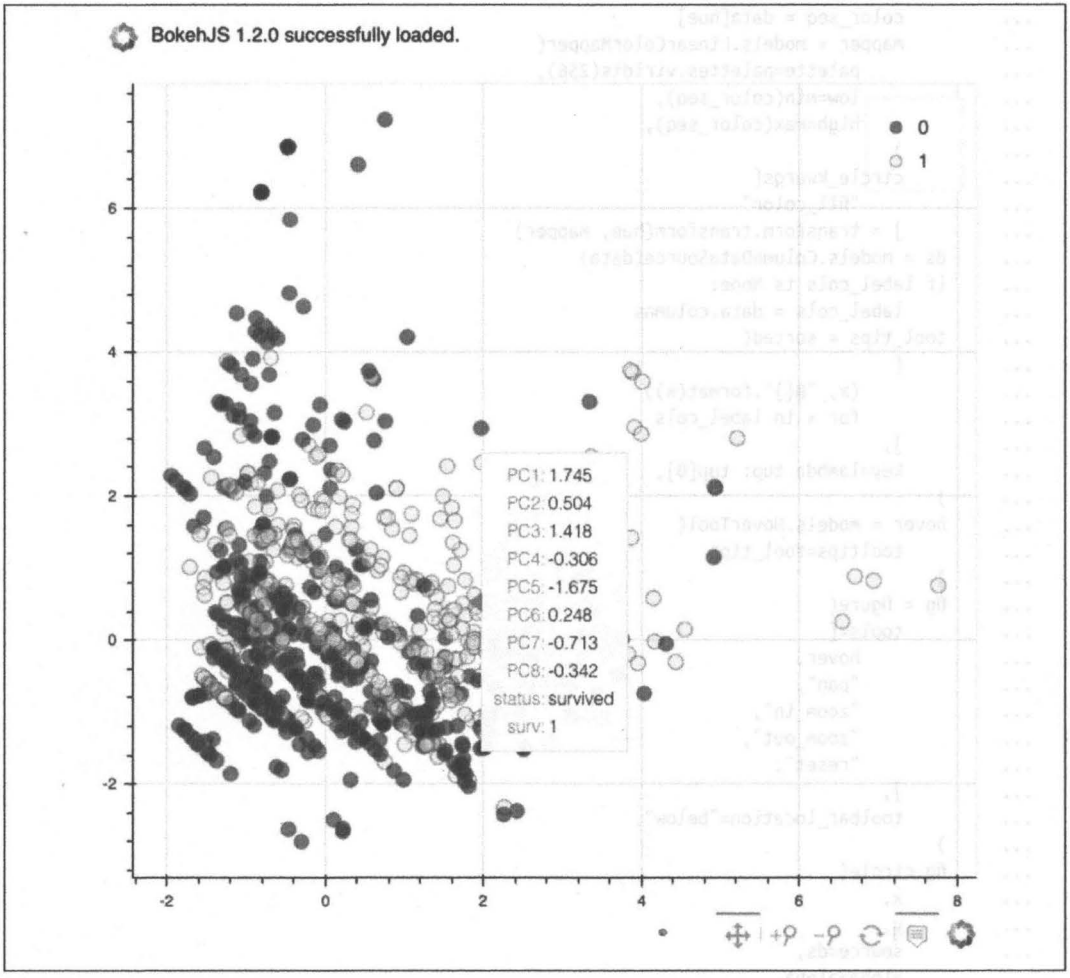



*Figure 17.9 : Diagramme en nuage dans Bokeh avec fenêtre d'aide.*

**`Yellowbrick`** est capable aussi de tracer en trois dimensions (Figure 17.10) :

```
>>> from yellowbrick.features.pca import (
...     PCADecomposition,
... )
>>> colors = ["rg"[j] for j in y]
>>> pca3_viz = PCADecomposition(
...     proj_dim=3, color=colors
```

```
... )
>>> pca3_viz.fit_transform(X, y)
>>> pca3_viz.finalize()
>>> fig = plt.gcf()
>>> plt.tight_layout()
>>> fig.savefig(
...     "images/mlpr_1710.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

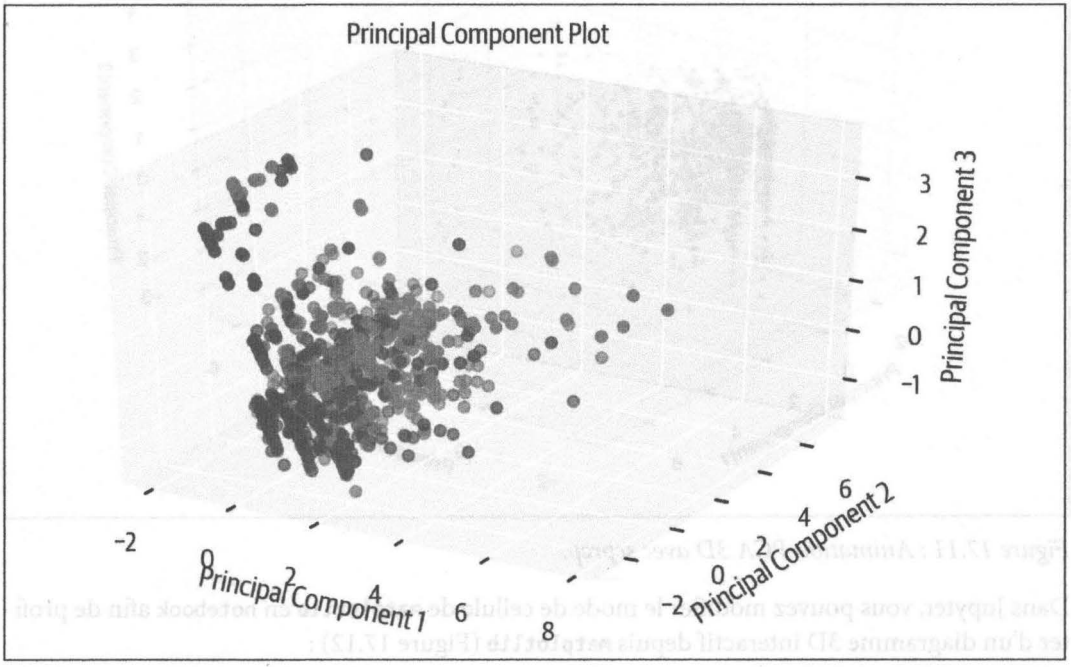



*Figure 17.10 : PCA 3D avec Yellowbrick.*

La librairie nommée **scprep** (`https://oreil.ly/Jdq1s`) est une dépendance pour la librairie PHATE que nous allons voir sous peu. Elle propose une fonction de tracé bien utile, `rotate_scatter3d`, pour produire un diagramme interactif dans Jupyter (Figure 17.11). Cela peut faciliter la compréhension d'un diagramme en 3D.

Cette librairie permet de visualiser n'importe quelles données en 3D, et non seulement avec PHATE :

```
>>> import scprep
>>> scprep.plot.rotate_scatter3d(
...     X_pca[:, :3],
...     c=y,
...     cmap="Spectral",
```

```
...     figsize=(8, 6),
...     label_prefix="Principal Component",
... )
```

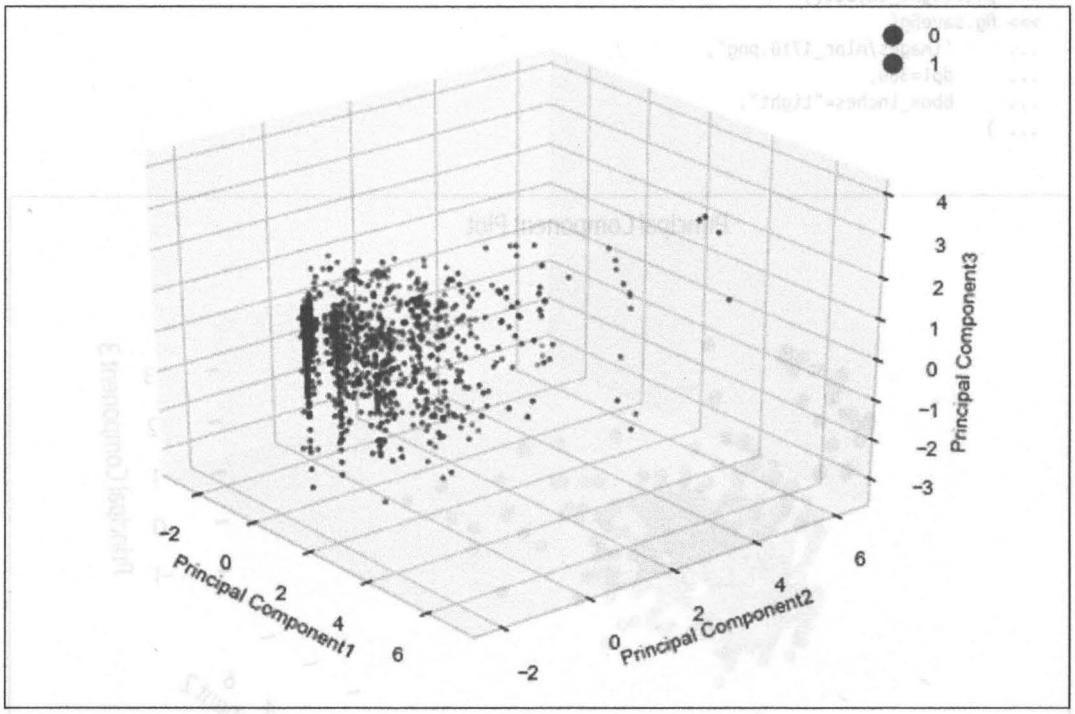



*Figure 17.11 : Animation PCA 3D avec scprep.*

Dans Jupyter, vous pouvez modifier le mode de cellule de **`matplotlib`** en notebook afin de profiter d'un diagramme 3D interactif depuis **`matplotlib`** (Figure 17.12) :

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure(figsize=(6, 4))
>>> ax = fig.add_subplot(111, projection="3d")
>>> ax.scatter(
...     xs=X_pca[:, 0],
...     ys=X_pca[:, 1],
...     zs=X_pca[:, 2],
...     c=y,
...     cmap="viridis",
... )
>>> ax.set_xlabel("PC 1")
>>> ax.set_ylabel("PC 2")
>>> ax.set_zlabel("PC 3")
```

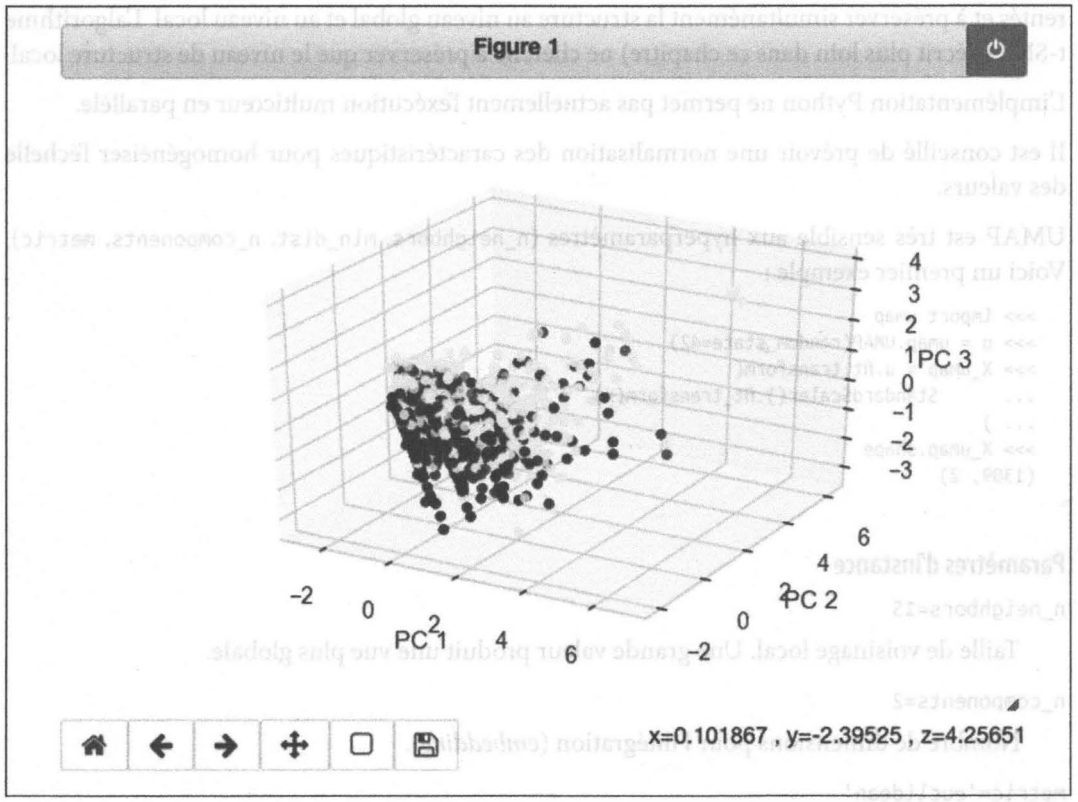



*Figure 17.12 : Diagramme 3D PCA interactif de matplotlib en mode notebook.*

Sachez que le basculement du mode de fonctionnement de cellule dans Jupyter entre :

```
% matplotlib inline
```

et

```
% matplotlib notebook
```

a parfois pour conséquence que Jupyter ne réponde plus. Utilisez cette possibilité avec précaution.

# UMAP

L'algorithme d'approximation et de projection de variétés uniforme UMAP (*Uniform Manifold Approximation and Projection*, `https://oreil.ly/qF8RJ`) propose une solution de réduction de dimensionnalité fondée sur un apprentissage par collecte/intégration (*manifold learning*). UMAP a tendance à maintenir proches en termes topologiques les éléments de données appa-

rentés et à préserver simultanément la structure au niveau global et au niveau local. L'algorithme t-SNE (décrit plus loin dans ce chapitre) ne cherche à préserver que le niveau de structure local.

L'implémentation Python ne permet pas actuellement l'exécution multicœur en parallèle.

Il est conseillé de prévoir une normalisation des caractéristiques pour homogénéiser l'échelle des valeurs.

UMAP est très sensible aux hyperparamètres (`n_neighbors`, `min_dist`, `n_components`, `metric`). Voici un premier exemple :

```
>>> import umap
>>> u = umap.UMAP(random_state=42)
>>> X_umap = u.fit_transform(
...       StandardScaler().fit_transform(X)
... )
>>> X_umap.shape
(1309, 2)
```

### Paramètres d'instance

`n_neighbors=15`

Taille de voisinage local. Une grande valeur produit une vue plus globale.

`n_components=2`

Nombre de dimensions pour l'intégration (*embedding*).

`metric='euclidean'`

Métrique de distance. Peut être une fonction acceptant deux tableaux à une dimension et envoyant une valeur `float`.

`n_epochs=None`

Nombre d'époques d'entraînement. Par défaut 200 pour gros volumes ou 500 pour petits volumes de données.

`learning_rate=1.0`

Taux d'apprentissage pour l'optimisation de l'intégration.

`init='spectral'`

Type d'initialisation. Spectrale par défaut. Peut être `'random'` ou bien un tableau **`numpy`** de localisations.

`min_dist=0.1`

Entre 0 et 1. Distance minimale entre points intégrés (*embedded*). Une valeur faible multiplie les grumeaux, une valeur forte disperse les points.

`spread=1.0`

Distance des points intégrés.

`set_op_mix_ratio=1.0`

Entre 0 et 1 : union floue (1) ou intersection floue (0).

`local_connectivity=1.0`

Nombre de voisins dans la connectivité locale. Une valeur plus haute augmente la création de connexions locales.

`repulsion_strength=1.0`

Force de répulsion. Une valeur haute renforce l'impact des échantillons négatifs.

`negative_sample_rate=5`

Nombre d'échantillons négatifs pour chaque échantillon positif lors de l'optimisation. Une valeur plus haute augmente la répulsion, ce qui alourdit l'optimisation mais améliore l'exactitude (*accuracy*).

`transform_queue_size=4.0`

Agressivité de recherche des plus proches voisins. Une valeur plus haute réduit les performances mais améliore l'exactitude.

`a=None`

Paramètre de contrôle de l'intégration. Si `None`, UMAP déduit un réglage à partir de `min_dist` et de `spread`.

`b=None`

Même usage que `a`.

`random_state=None`

Graine du générateur aléatoire.

`metric_kwds=None`

Dictionnaire de métriques avec paramètres additionnels si une fonction est définie pour les mesures. Plusieurs métriques telles que `minkowsi` peuvent être paramétrées de cette façon.

`angular_rp_forest=False`

Utiliser la projection angulaire aléatoire.

`target_n_neighbors=-1`

Nombre de voisins pour l'ensemble simplicial.

`target_metric='categorical'`

Utilisé avec la réduction supervisée. La valeur peut être `'L1'` ou `'L2'`. Accepte une fonction attendant en entrée deux tableaux de X pour renvoyer la distance entre eux.

`target_metric_kwds=None`

Dictionnaire de métriques à utiliser si une fonction est définie pour `target_metric`.

`target_weight=0.5`

Facteur de pondération entre 0.0 et 1.0. Avec 0, se base seulement sur les données et avec 1, seulement sur la cible.

`transform_seed=42`

Graine du générateur aléatoire.

`verbose=False`

Verbosité (prolixité) d'affichage.

### Attributs

`embedding_`

Résultats de l'intégration.

Visualisons les résultats avec les paramètres par défaut d'UMAP pour le jeu Titanic (Figure 17.13) :

```
>>> fig, ax = plt.subplots(figsize=(8, 4)) >>> pd.DataFrame(X_umap).plot(
...     kind="scatter",
...     x=0,
...     y=1,
...     ax=ax,
...     c=y,
...     alpha=0.2,
...     cmap="Spectral",
... )
>>> fig.savefig("images/mlpr_1713.png", dpi=300)
```

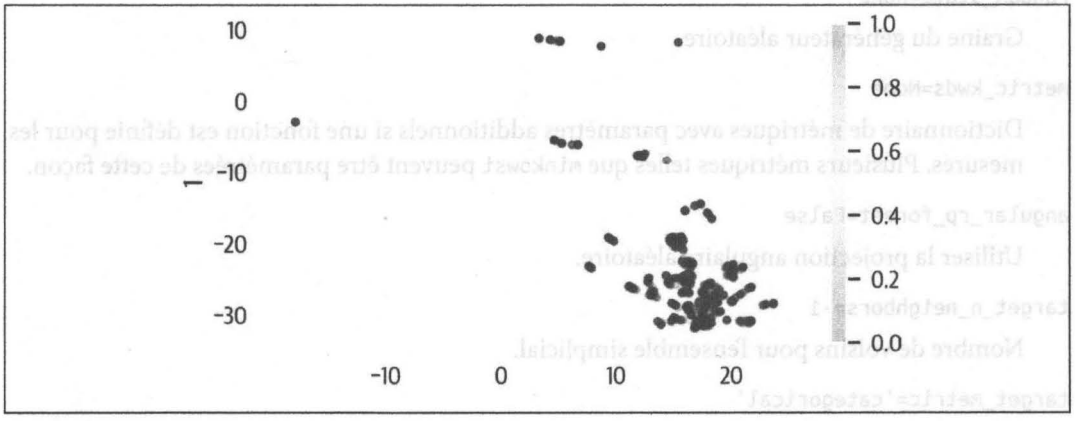



*Figure 17.13 : Résultats d'UMAP.*

Lorsqu'il s'agit d'ajuster UMAP, concentrez-vous sur les deux hyperparamètres `n_neighbors` et `min_dist`. Voici un aperçu de l'impact de ces deux valeurs (Figure 17.14 et 17.15) :

```
>>> X_std = StandardScaler().fit_transform(X)
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> for i, n in enumerate([2, 5, 10, 50]):
...     ax = axes[i]
...     u = umap.UMAP(
...         random_state=42, n_neighbors=n
...     )
...     X_umap = u.fit_transform(X_std)
...     pd.DataFrame(X_umap).plot(
...         kind="scatter",
...         x=0,
...         y=1,
...         ax=ax,
...         c=y,
...         cmap="Spectral",
...         alpha=0.5,
...     )
...     ax.set_title(f"nn={n}")
>>> plt.tight_layout()
>>> fig.savefig("images/mlpr_1714.png", dpi=300)
```

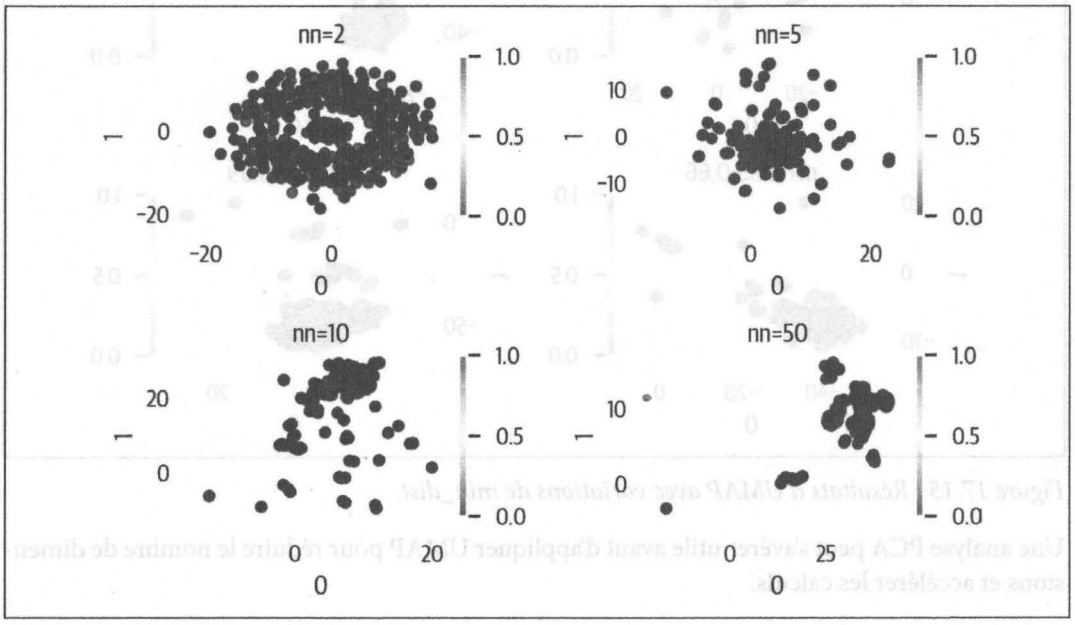



*Figure 17.14 : Résultats d'UMAP avec variations de n_neighbors.*

```
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> for i, n in enumerate([0, 0.33, 0.66, 0.99]):
...     ax = axes[i]
...     u = umap.UMAP(random_state=42, min_dist=n)
...     X_umap = u.fit_transform(X_std)
...     pd.DataFrame(X_umap).plot(
...         kind="scatter",
...         x=0,
...         y=1,
...         ax=ax,
...         c=y,
...         cmap="Spectral",
...         alpha=0.5,
...     )
...     ax.set_title(f"min_dist={n}")
>>> plt.tight_layout()
>>> fig.savefig("images/mlpr_1715.png", dpi=300)
```

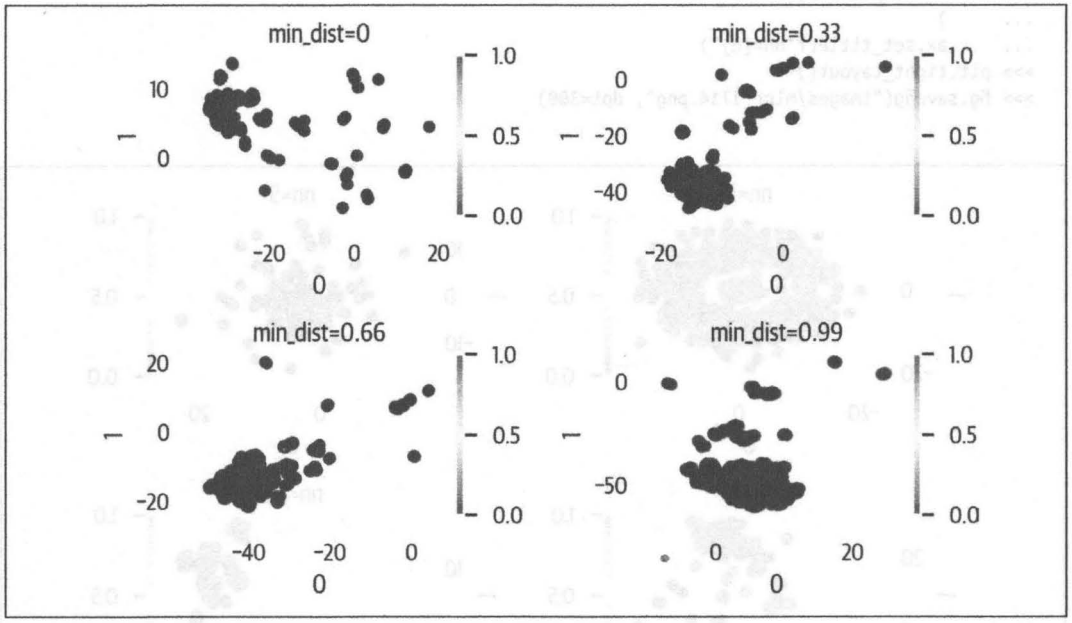



*Figure 17.15 : Résultats d'UMAP avec variations de min_dist.*

Une analyse PCA peut s'avérer utile avant d'appliquer UMAP pour réduire le nombre de dimensions et accélérer les calculs.

# t-SNE

L'algorithme t-SNE (*t-Distributed Stochastic Neighboring Embedding*) est une autre technique de réduction de dimensionnalité et de visualisation. Il convertit les similarités entre points de données en probabilités jointes puis tente de minimiser la divergence de Kullback-Leibler entre les probabilités jointes des embeddings à faible dimensionalité et les données à forte dimensionalité. C'est un traitement intensif qui ne sera parfois pas réalisable si le volume de données est trop important.

t-SNE est assez sensible aux hyperparamètres. Il préserve assez bien les groupes locaux, mais pas l'information globale : les distances entre groupes (clusters) perdent leur signification. Enfin, ce n'est pas un algorithme déterministe ; il peut ne pas converger.

Il est conseillé de standardiser les données dans le prétraitement :

```
>>> from sklearn.manifold import TSNE
>>> X_std = StandardScaler().fit_transform(X)
>>> ts = TSNE()
>>> X_tsne = ts.fit_transform(X_std)
```

### Paramètres d'instance

`n_components=2`

Nombre de dimensions de l'intégration (*embedding*).

`perplexity=30.0`

Valeurs entre 5 et 50 en pratique. Les valeurs basses produisent des grumeaux plus compacts.

`early_exaggeration=12.0`

Contrôle l'étanchéité des clusters et l'écartement entre eux. Une valeur haute demande plus d'écart.

`learning_rate=200.0`

En général entre 10 et 1000. Réduire la valeur si les données montrent un aspect de balle. L'augmenter si elles semblent compressées.

`n_iter=1000`

Nombre d'itérations.

`n_iter_without_progress=300`

Abandonne si aucun progrès après ce nombre d'itérations.

`min_grad_norm=1e-07`

Stoppe l'optimisation si la norme de gradient descend sous cette valeur.

`metric='euclidean'`

Métrique de distance de `scipy.spatial.distance.pdist`, `pairwise.PAIRWISE_DISTANCE_METRIC` ou le nom d'une fonction.

`init='random'`

Initialisation de l'intégration (*embedding*).

`verbose=0`

Verbosité (prolixité) de l'affichage.

`random_state=None`

Graine du générateur aléatoire.

`method='barnes_hut'`

Algorithme de calcul de gradient.

`angle=0.5`

Angle du calcul de gradient. Une valeur inférieure à 0.2 rallonge la durée de traitement, supérieure à 0.8, elle augmente les erreurs.

### Attributs

`embedding_`

Vecteurs de l'intégration.

`kl_divergence_`

Divergence de Kullback-Leibler.

`n_iter_`

Nombre d'itérations.

Voici une visualisation des résultats de t-SNE avec **`matplotlib`** (Figure 17.16) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> colors = ["rg"[j] for j in y]
>>> scat = ax.scatter(
...     X_tsne[:, 0],
...     X_tsne[:, 1],
...     c=colors,
...     alpha=0.5,
... )
>>> ax.set_xlabel("Embedding 1")
>>> ax.set_ylabel("Embedding 2")
>>> fig.savefig("images/mlpr_1716.png", dpi=300)
```

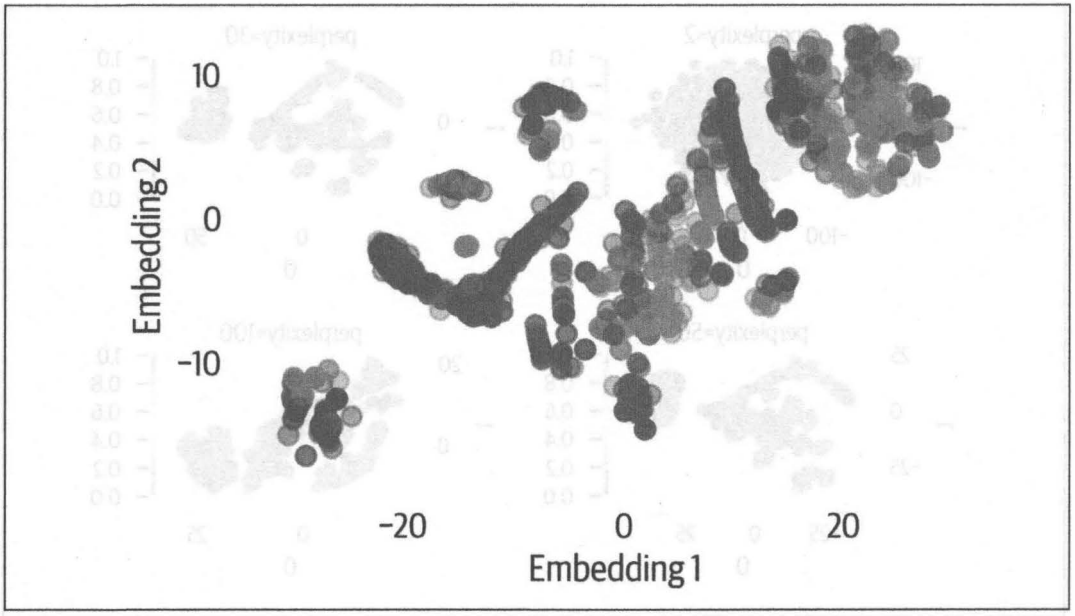



*Figure 17.16 : Résultats de t-SNE avec matplotlib.*

Le changement de la valeur de `perplexity` peut avoir un grand effet (Figure 17.17). Voici un essai avec plusieurs valeurs :

```
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> for i, n in enumerate((2, 30, 50, 100)):
...     ax = axes[i]
...     t = TSNE(random_state=42, perplexity=n)
...     X_tsne = t.fit_transform(X)
...     pd.DataFrame(X_tsne).plot(
...         kind="scatter",
...         x=0,
...         y=1,
...         ax=ax,
...         c=y,
...         cmap="Spectral",
...         alpha=0.5,
...     )
...     ax.set_title(f"perplexity={n}")
... plt.tight_layout()
... fig.savefig("images/mlpr_1717.png", dpi=300)
```

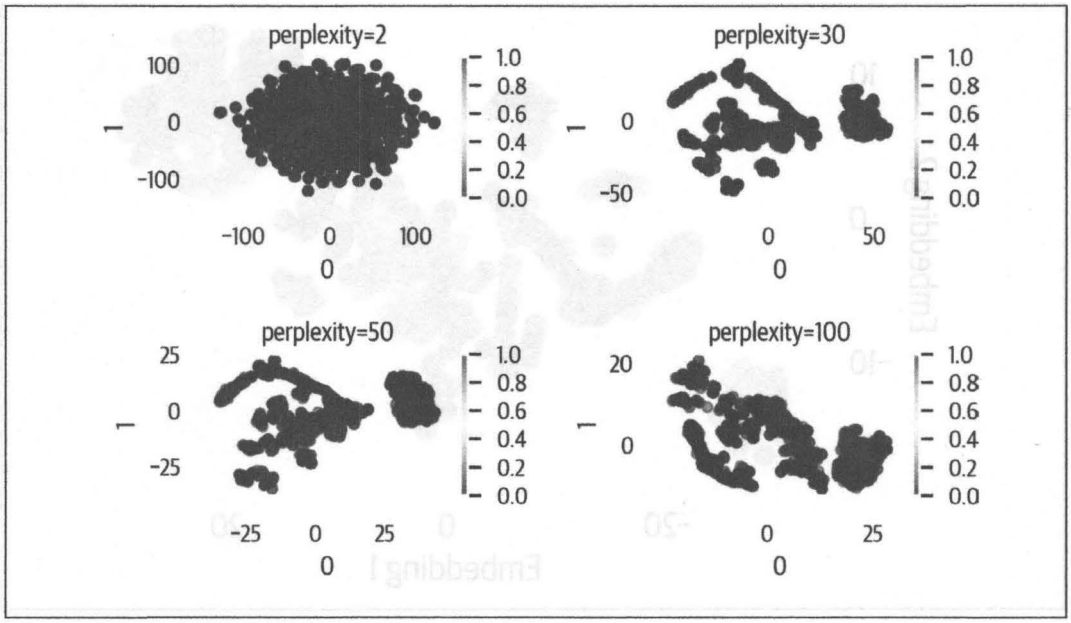



*Figure 17.17 : Variation de perplexity pour t-SNE.*

# PHATE

L'algorithme PHATE (*Potential of Heat-diffusion for Affinity-based Trajectory Embedding*, `https://phate.readthedocs.io`) sert à visualiser des données à grand nombre de dimensions. Il a tendance à préserver tant la structure globale (comme PCA) que les structures locales (comme t-SNE).

PHATE commence par encoder l'information locale (les points voisins doivent le rester) puis utilise la technique de « diffusion » pour trouver les données globales, ce qui lui permet enfin de réduire la dimensionnalité :

```
>>> import phate
>>> p = phate.PHATE(random_state=42)
>>> X_phate = p.fit_transform(X)
>>> X_phate.shape
```

### Paramètres d'instance

`n_components=2`

Nombre de dimensions.

`knn=5`

Nombre de voisins pour le noyau. Augmentez si l'intégration est déconnectée ou si le jeu de données comprend plus de 100 000 échantillons.

`decay=40`

Taux de *decay* de queue de noyau. Une valeur plus basse augmente la connectivité du graphe.

`n_landmark=2000`

Nombre de balises à utiliser.

`t='auto'`

Force de diffusion. Un lissage des données est réalisé. Augmentez la valeur si les intégrations manquent de structure. Réduisez-la si la structure est serrée et trop compacte.

`gamma=1`

Constante de distance informationnelle entre -1 et 1. Si les intégrations se concentrent autour d'un seul point, essayez de ramener cette valeur à 0.

`n_pca=100`

Nombre de composantes principales pour le calcul de voisinage.

`knn_dist='euclidean'`

Métrique KNN.

`mds_dist='euclidean'`

Métrique de *scaling* multidimensionnel (MDS).

`mds='metric'`

Algorithme MDS pour la réduction de dimensions.

`n_jobs=1`

Nombre de CPU à exploiter.

`random_state=None`

Graine du générateur aléatoire.

`verbose=1`

Verbosité.

### Attributs

Notez bien que les noms ne se terminent pas par le signe _ :

`X`

Données d'entrée.

`embedding`

Espace d'intégration.

`diff_op`

Opérateur de diffusion.

`graph`

Graphe KNN construit à partir des entrées.

Voici un exemple d'utilisation de PHATE (Figure 17.18) :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> phate.plot.scatter2d(p, c=y, ax=ax, alpha=0.5)
>>> fig.savefig("images/mlpr_1718.png", dpi=300)
```

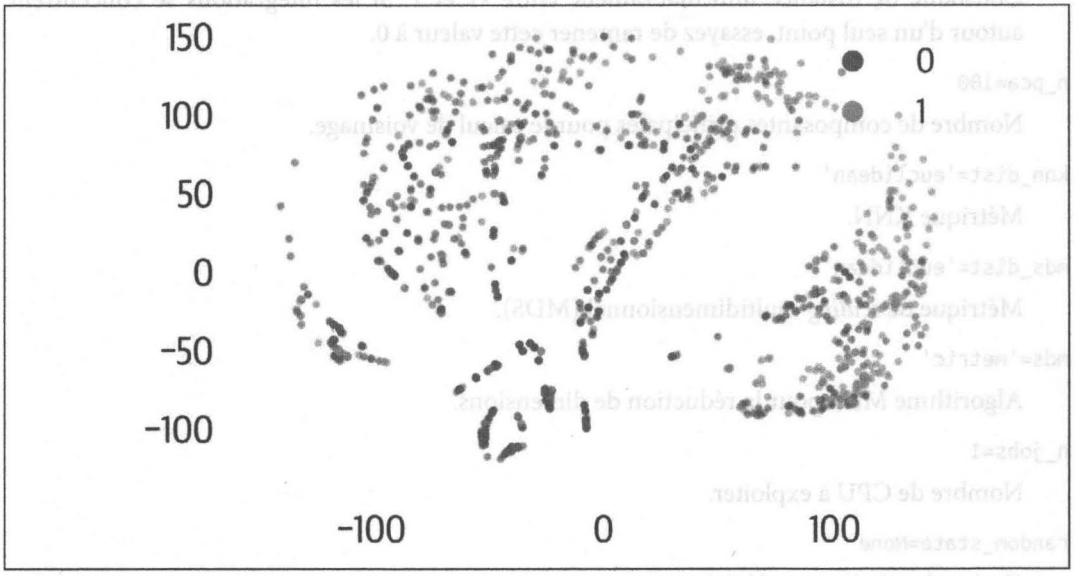



*Figure 17.18 : Résultats de PHATE.*

La description des paramètres nous a appris que certains d'entre eux peuvent être retouchés pour modifier le comportement du modèle. Voici par exemple comment ajuster `knn` (Figure 17.19). Notez qu'en utilisant la méthode `.set_params`, nous accélérons les calculs du fait que cela fait réutiliser le graphe précalculé et l'opérateur de diffusion :

```
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> p = phate.PHATE(random_state=42, n_jobs=-1)
```

```
>>> for i, n in enumerate((2, 5, 20, 100)):
...     ax = axes[i]
...     p.set_params(knn=n)
...     X_phate = p.fit_transform(X)
...     pd.DataFrame(X_phate).plot(
...         kind="scatter",
...         x=0,
...         y=1,
...         ax=ax,
...         c=y,
...         cmap="Spectral",
...         alpha=0.5,
...     )
...     ax.set_title(f"knn={n}")
... plt.tight_layout()
... fig.savefig("images/mlpr_1719.png", dpi=300)
```

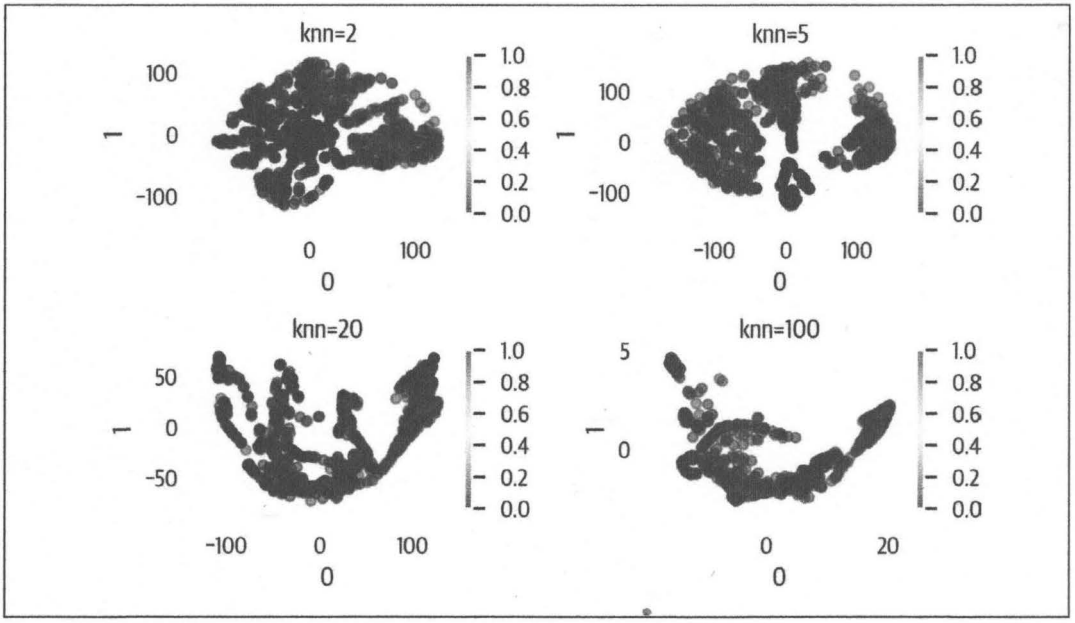



*Figure 17.19 : Variation du paramètre knn dans PHATE.*

# CHAPITRE 18
# Regroupement (*clustering*)

Le *regroupement/clustering* (ou encore partitionnement) est une technique d'apprentissage machine non supervisé qui cherche à distribuer un lot de données en plusieurs cohortes. Cette approche est non supervisée au sens où elle n'a aucun label à disposition dans le modèle ; elle ne peut qu'inspecter les caractéristiques pour en déduire quels échantillons sont similaires et peuvent être regroupés dans la même *grappe* ou paquet (*cluster*). Nous allons découvrir dans ce chapitre, la méthode des k-moyennes et celle des regroupements hiérarchiques. Nous utiliserons comme jeu de données d'exemple le jeu Titanic.

## K-moyennes

Pour créer des grappes avec l'algorithme k-moyennes, l'utilisateur doit choisir un nombre *k* de grappes. Le modèle choisit ensuite un nombre k de centroïdes et affecte chacun des échantillons à une grappe en fonction de son éloignement par rapport à ce centroïde. Une fois la distribution faite, l'algorithme recalcule les centroïdes en se basant sur le centre de chacun des échantillons de la même grappe avec le même label. L'affectation des échantillons à la grappe est ensuite répétée à partir des nouveaux centroïdes. Au bout de quelques tours, le résultat doit converger.

Du fait que cette technique se fonde sur les distances pour regrouper les échantillons, la qualité du résultat dépend de l'échelle des données. Vous pouvez donc avoir intérêt à standardiser ces données pour que toutes les caractéristiques soient exprimées dans la même échelle. Cependant, certains font remarquer que les experts métier déconseillent la standardisation dans certains cas, car les différences d'échelle peuvent prouver que certaines caractéristiques ont plus d'importance que d'autres. Dans notre exemple, nous avons standardisé les données.

Nous allons effectuer un partitionnement pour les passagers du *Titanic*. Nous commencerons avec deux grappes pour voir si le regroupement réussit à prédire quels passagers vont survivre.

Nous prendrons soin de ne pas divulguer la survie en laissant de côté les données qui donnent des indices ; nous n'utilisons que *X* et pas *y*.

Les algorithmes non supervisés disposent d'une méthode `.fit` et d'une méthode `.predict`. Nous prenons soin de ne transmettre que *X* à `.fit` :

```
>>> from sklearn.cluster import KMeans
>>> X_std = preprocessing.StandardScaler().fit_transform(
...     X
... )
>>> km = KMeans(2, random_state=42)
>>> km.fit(X_std)
KMeans(algorithm='auto', copy_x=True,
  init='k-means', max_iter=300,
  n_clusters=2, n_init=10, n_jobs=1,
  precompute_distances='auto',
  random_state=42, tol=0.0001, verbose=0)
```

Une fois que l'entraînement du modèle est fait, nous appelons la méthode `.predict` pour affecter de nouveaux échantillons aux grappes :

```
>>> X_km = km.predict(X)
>>> X_km
array([1, 1, 1, ..., 1, 1, 1], dtype=int32)
```

### Paramètres d'instance

`n_clusters=8`

Nombre grappes (clusters) à créer.

`init='kmeans++'`

Méthode d'initialisation.

`n_init=10`

Nombre de tours d'exécution de l'algorithme avec différents centroïdes. Le meilleur score gagne.

`max_iter=300`

Nombre d'itérations pour une exécution.

`tol=0.0001`

Tolérance avant convergence.

`precompute_distances='auto'`

Précalcul des distances (plus grande empreinte mémoire, mais plus rapide). `auto` précalcule si `n_samples * n_clusters` est inférieur ou égal à 12 millions.

`verbose=0`

Verbosité.

`random_state=None`

Graine du générateur aléatoire.

`copy_x=True`

Copie des données avant calcul.

`n_jobs=1`

Nombre de CPU.

`algorithm='auto'`

Algorithme de k-moyennes. `'full'` convient aux données éparses, mais `'elkan'` est plus efficace. `'auto'` se sert de `'elkan'` si les données sont denses.

### Attributs

`cluster_centers_`

Coordonnées des centroïdes.

`labels_`

Labels des échantillons.

`inertia_`

Somme des distances quadratiques aux centroïdes des grappes.

`n_iter_`

Nombre d'itérations.

Si vous ne savez pas au départ combien de grappes il vous faut, exécutez l'algorithme avec plusieurs tailles et évaluez les métriques. Notez que l'investigation peut être complexe.

Vous pouvez utiliser le calcul de l'inertie avec `.inertia_` pour utiliser la méthode du coude dans le diagramme. Il suffit de chercher à quel endroit la ligne montre une courbe plus prononcée. En général, cela correspond assez bien au nombre de grappes. Dans l'exemple, la courbe n'a pas de coude franc, mais on peut constater qu'il n'y a plus beaucoup d'amélioration au bout de huit tours (Figure 18.1).

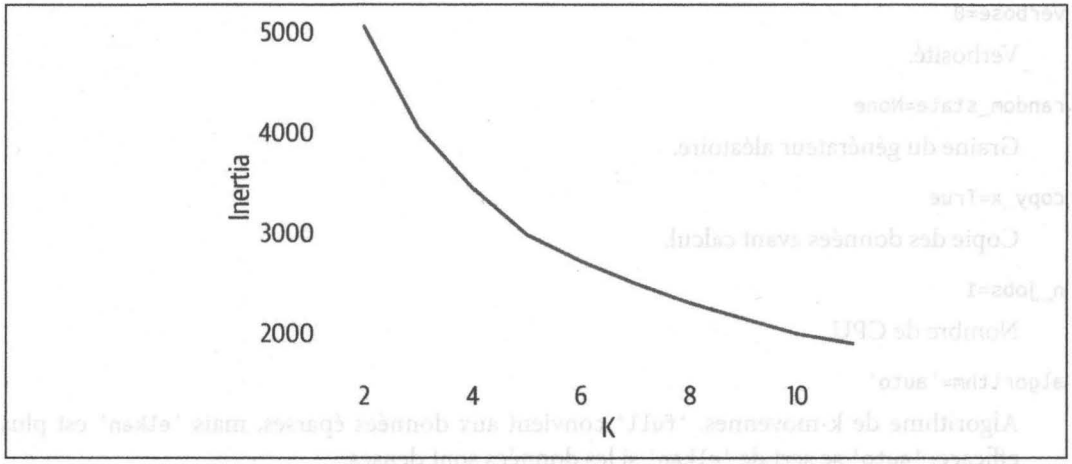



*Figure 18.1 : Un coude peu prononcé.*

Lorsqu'il n'y a pas de coude dans le diagramme, nous avons plusieurs possibilités. Nous pouvons utiliser d'autres métriques, et nous allons en voir quelques-unes plus loin. Nous pouvons aussi demander une visualisation du regroupement pour voir si l'on peut repérer des groupes. Nous pouvons enfin ajouter des caractéristiques et voir si elles aident à l'opération. Voici le code pour obtenir un diagramme avec coude :

```
>>> inertias = []
>>> sizes = range(2, 12)
>>> for k in sizes:
...     k2 = KMeans(random_state=42, n_clusters=k)
...     k2.fit(X)
...     inertias.append(k2.inertia_)
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pd.Series(inertias, index=sizes).plot(ax=ax)
>>> ax.set_xlabel("K")
>>> ax.set_ylabel("Inertie")
>>> fig.savefig("images/mlpr_1801.png", dpi=300)
```

La librairie **`scikit-learn`** propose plusieurs métriques de regroupement pour le cas où les labels véritables ne sont pas connus. Rien ne nous empêche de les calculer et de les visualiser.

- Le *coefficient de silhouette* est une valeur entre -1 et 1. Plus le score est élevé, mieux c'est. La valeur 1 dénote des grappes compactes et la valeur 0 des grappes en chevauchement. Selon cette métrique, dans notre exemple, nous trouvons deux grappes pour le meilleur score.
- L'*indice Calinski-Harabasz* correspond au rapport entre la dispersion entre grappes et la dispersion interne à chaque grappe. Plus le score est élevé, mieux c'est. Ici aussi, la métrique invite à choisir deux grappes.

- L'*indice Davis-Bouldin* correspond à la similarité moyenne entre une grappe et la voisine la plus proche. Les valeurs commencent à zéro, et le meilleur regroupement correspond à cette valeur nulle.

Produisons un diagramme combiné pour montrer l'inertie, le coefficient de silhouette, l'indice Calinski-Harabasz et l'indice Davies-Bouldin en les appliquant à une plage de taille de grappe. Cela nous permettra de voir s'il y a effectivement une taille idéale pour nos données (Figure 18.2). Le résultat confirme que la plupart des métriques invitent à considérer deux grappes :

```
>>> from sklearn import metrics
>>> inertias = []
>>> sils = []
>>> chs = []
>>> dbs = []
>>> sizes = range(2, 12)
>>> for k in sizes:
...     k2 = KMeans(random_state=42, n_clusters=k)
...     k2.fit(X_std)
...     inertias.append(k2.inertia_)
...     sils.append(
...         metrics.silhouette_score(X, k2.labels_)
...     )
...     chs.append(
...         metrics.calinski_harabasz_score(
...             X, k2.labels_
...         )
...     )
...     dbs.append(
...         metrics.davies_bouldin_score(
...             X, k2.labels_
...         )
...     )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> (
...     pd.DataFrame(
...         {
...             "inertia": inertias,
...             "silhouette": sils,
...             "calinski": chs,
...             "davis": dbs,
...             "k": sizes,
...         }
...     )
...     .set_index("k")
...     .plot(ax=ax, subplots=True, layout=(2, 2))
... )
>>> fig.savefig("images/mlpr_1802.png", dpi=300)
```

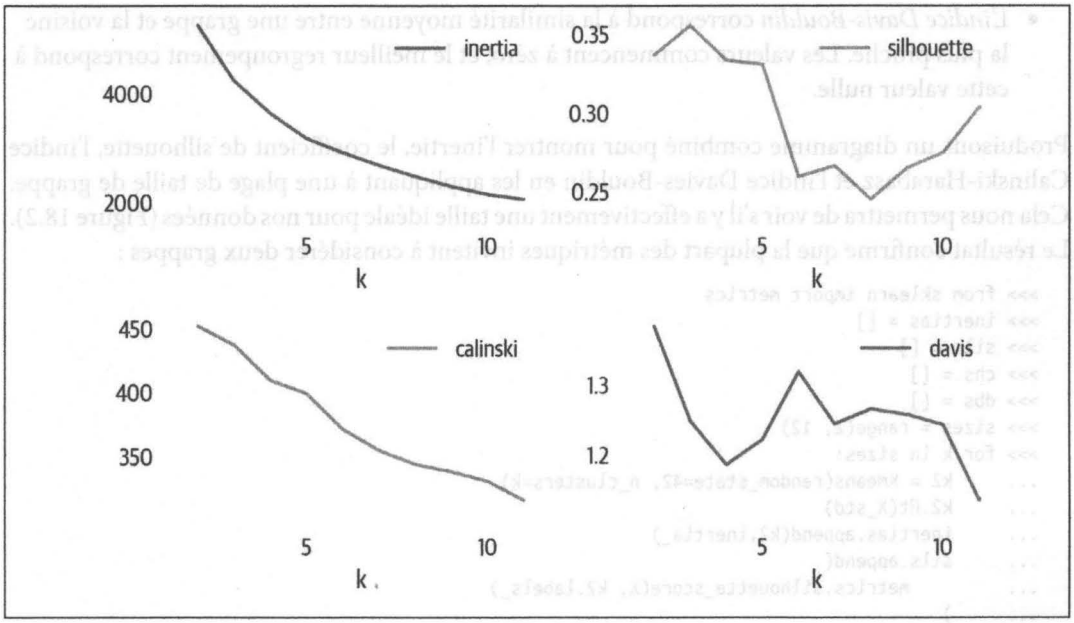



*Figure 18.2 : Plusieurs métriques pour regroupement.*

Une autre technique pour déterminer le nombre de grappes consiste à montrer les scores de silhouette pour chaque grappe. **`Yellowbrick`** propose un outil pour visualiser cela (Figure 18.3).

La ligne pointillée rouge verticale correspond au score moyen. Pour l'interpréter, repérez les grappes qui surnagent au-dessus de la moyenne avec un score satisfaisant. Assurez-vous de bien utiliser les mêmes limites en x (`ax.set_xlim`). D'après ce diagramme, je choisirais une fois de plus deux grappes :

```
>>> from yellowbrick.cluster.silhouette import (
...     SilhouetteVisualizer,
... )
>>> fig, axes = plt.subplots(2, 2, figsize=(12, 8))
>>> axes = axes.reshape(4)
>>> for i, k in enumerate(range(2, 6)):
...     ax = axes[i]
...     sil = SilhouetteVisualizer(
...         KMeans(n_clusters=k, random_state=42),
...         ax=ax,
...     )
...     sil.fit(X_std)
...     sil.finalize()
...     ax.set_xlim(-0.2, 0.8)
>>> plt.tight_layout()
>>> fig.savefig("images/mlpr_1803.png", dpi=300)
```

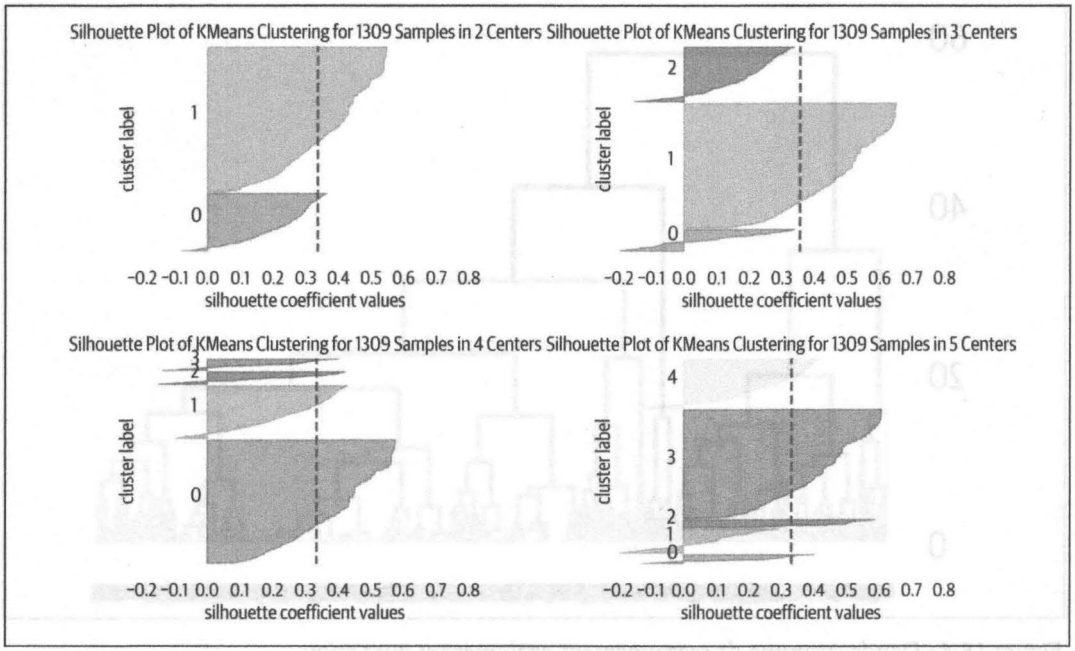



*Figure 18.3 : Visualisation de silhouette avec Yellowbrick.*

# Regroupement agglomérant (hiérarchique)

Une technique similaire est le regroupement agglomérant. Vous commencez par traiter chaque échantillon dans sa grappe individuelle puis vous combinez les grappes les plus proches. Vous répétez l'opération tout en surveillant les tailles des voisines proches.

À la fin de l'opération, vous disposez d'un dendrogramme, une sorte d'arbre qui permet de voir comment les grappes ont été créées les unes à partir des autres et quelles sont les distances utilisées. Vous visualisez ce genre de dendrogramme avec la librairie **`scipy`**.

C'est ce que nous faisons dans la Figure 18.4. Vous remarquez que si le nombre d'échantillons est important, le nombre de feuilles terminales est tel que le résultat devient difficile à lire :

```
>>> from scipy.cluster import hierarchy
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> dend = hierarchy.dendrogram(
...     hierarchy.linkage(X_std, method="ward")
... )
>>> fig.savefig("images/mlpr_1804.png", dpi=300)
```

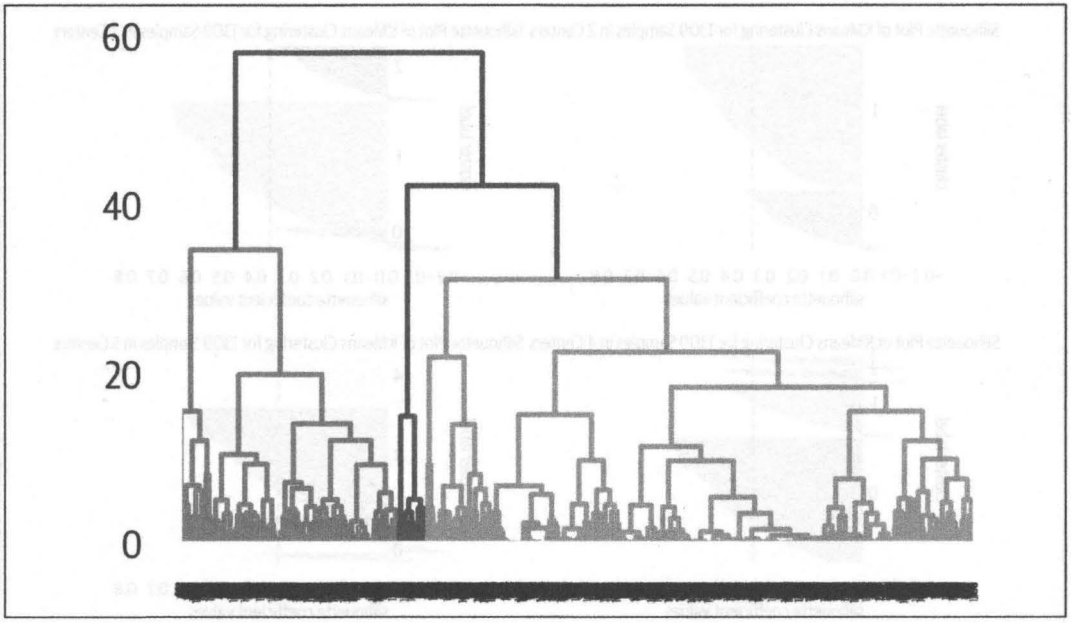



*Figure 18.4 : Dendrogramme de regroupement agglomérant avec scipy.*

Une fois que vous avez obtenu le dendrogramme, vous voyez toutes les grappes, de la principale jusqu'au niveau des échantillons individuels. Les hauteurs correspondent à la similarité entre grappes. Pour déterminer le nombre de grappes à spécifier, il suffit de tirer un trait horizontal au niveau où il va couper les lignes les plus longues.

Dans notre exemple, il semble qu'il en ressorte trois grappes principales.

Le problème de ce diagramme est qu'il est très bruité, puisqu'il descend au niveau des échantillons individuels. Il est possible dans ce cas d'utiliser le paramètre `truncate_mode` pour combiner les feuilles en un nœud (Figure 18.5) :

```
>>> from scipy.cluster import hierarchy
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> dend = hierarchy.dendrogram(
...     hierarchy.linkage(X_std, method="ward"),
...     truncate_mode="lastp",
...     p=20,
...     show_contracted=True,
... )
>>> fig.savefig("images/mlpr_1805.png", dpi=300)
```

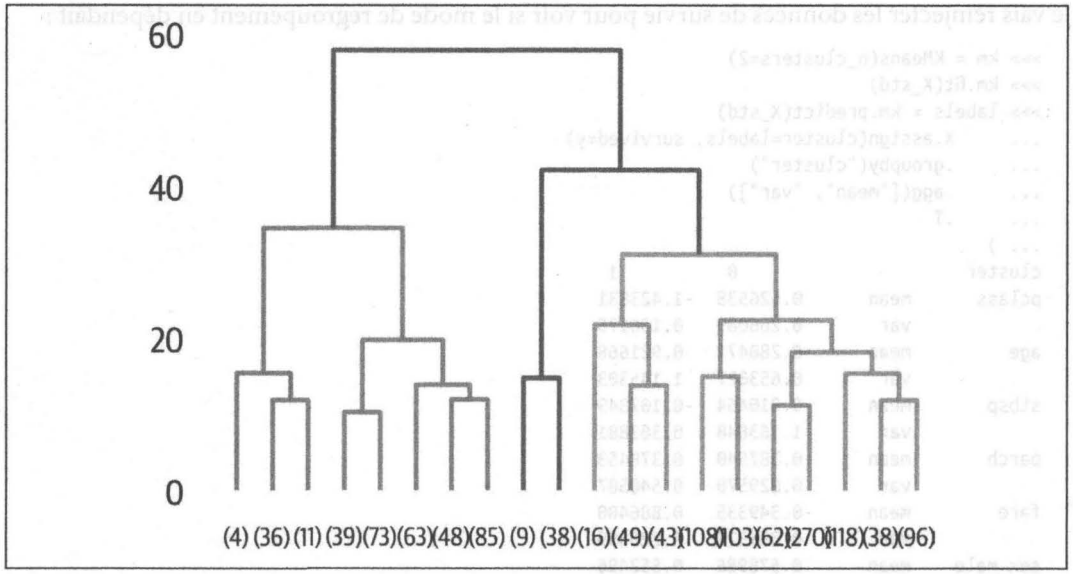



*Figure 18.5 : Dendrogramme tronqué donc allégé. Un trait horizontal coupant les trois lignes les plus longues permet de trouver trois grappes.*

Puisque nous savons maintenant le nombre de grappes à mentionner, nous pouvons créer le modèle avec **`scikit-learn`** :

```
>>> from sklearn.cluster import (
...     AgglomerativeClustering,
... )
>>> ag = AgglomerativeClustering(
...     n_clusters=4,
...     affinity="euclidean",
...     linkage="ward",
... )
>>> ag.fit(X)
```

Le paquetage nommé **`fastcluster`** (`https://oreil.ly/OuNuo`) permet un regroupement agglomérant optimisé, ce qui est utile si vous constatez que la librairie **`scikit-learn`** est trop lente.

# Analyse des grappes

Nous allons partir sur deux grappes en utilisant la technique des k-moyennes avec le jeu Titanic. Nous nous servirons des possibilités de regroupement de **`pandas`** pour étudier les différences entre grappes. L'exemple suivant s'intéresse à la moyenne et la variance de chaque caractéristique. Nous pouvons constater que la valeur moyenne de *`pclass`* varie de façon remarquable.

Je vais réinjecter les données de survie pour voir si le mode de regroupement en dépendait :

```
>>> km = KMeans(n_clusters=2)
>>> km.fit(X_std)
>>> labels = km.predict(X_std)
...     X.assign(cluster=labels, survived=y)
...     .groupby("cluster")
...     .agg(["mean", "var"])
...     .T
... )
cluster                   0         1
pclass      mean      0.526538  -1.423831
            var       0.266089   0.136175
age         mean     -0.280471   0.921668
            var       0.653027   1.145303
sibsp       mean     -0.010464  -0.107849
            var       1.163848   0.303881
parch       mean      0.387540   0.378453
            var       0.829570   0.540587
fare        mean     -0.349335   0.886400
            var       0.056321   2.225399
sex_male    mean      0.678986   0.552486
            var       0.218194   0.247930
embarked_Q  mean      0.123548   0.016575
            var       0.108398   0.016345
embarked_S  mean      0.741288   0.585635
            var       0.191983   0.243339
survived    mean      0.596685   0.299894
            var       0.241319   0.210180
```

Dans Jupyter, vous pouvez ajouter l'instruction suivante en liaison avec une structure `DataFrame` pour mettre en surbrillance les valeurs fortes et faibles dans chaque ligne. Cela vous permet de vérifier visuellement les valeurs remarquables dans la synthèse de grappes précédente :

```
.style.background_gradient(cmap='RdBu', axis=1)
```

La Figure 18.6 est un diagramme en barres des moyennes pour chaque grappe :

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
... (
...     X.assign(cluster=labels, survived=y)
...     .groupby("cluster")
...     .mean()
...     .T.plot.bar(ax=ax)
... )
>>> fig.savefig(
...     "images/mlpr_1806.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

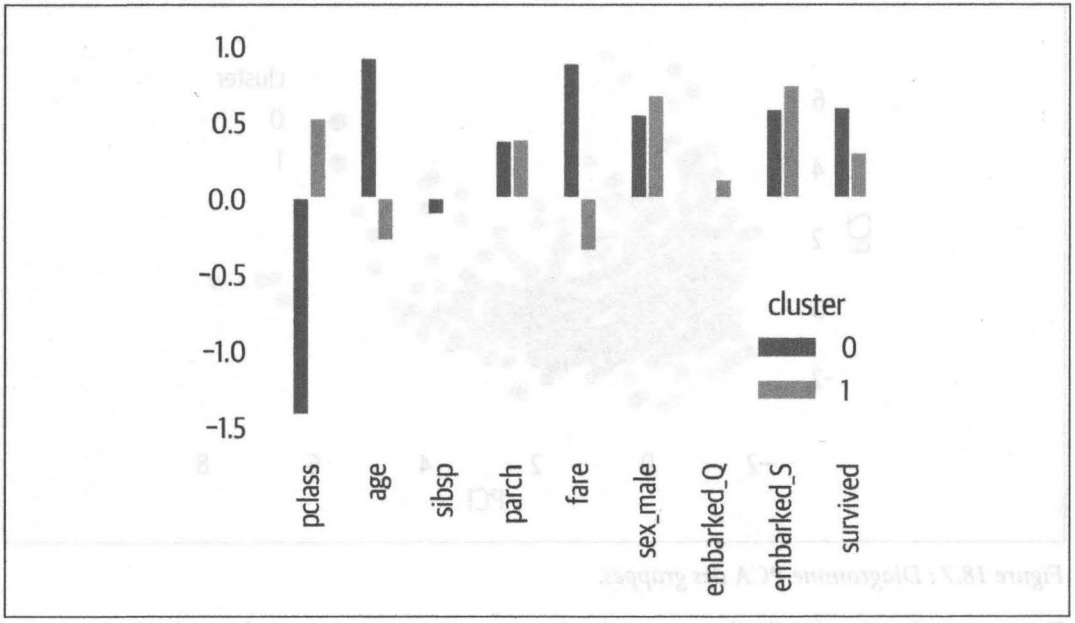



*Figure 18.6 : Valeurs moyennes de chaque grappe.*

Je trouve intéressant de visualiser également les composantes PCA, mais en les colorant en fonction du label de chaque grappe (Figure 18.7). Ici, je me sers de la librairie **seaborn**. N'hésitez pas à jouer sur la valeur du paramètre `hue` pour pouvoir entrer dans les détails des caractéristiques qui sont les constituants des grappes.

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> sns.scatterplot(
...     "PC1",
...     "PC2",
...     data=X.assign(
...         PC1=X_pca[:, 0],
...         PC2=X_pca[:, 1],
...         cluster=labels,
...     ),
...     hue="cluster",
...     alpha=0.5,
...     ax=ax,
... )
>>> fig.savefig(
...     "images/mlpr_1807.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

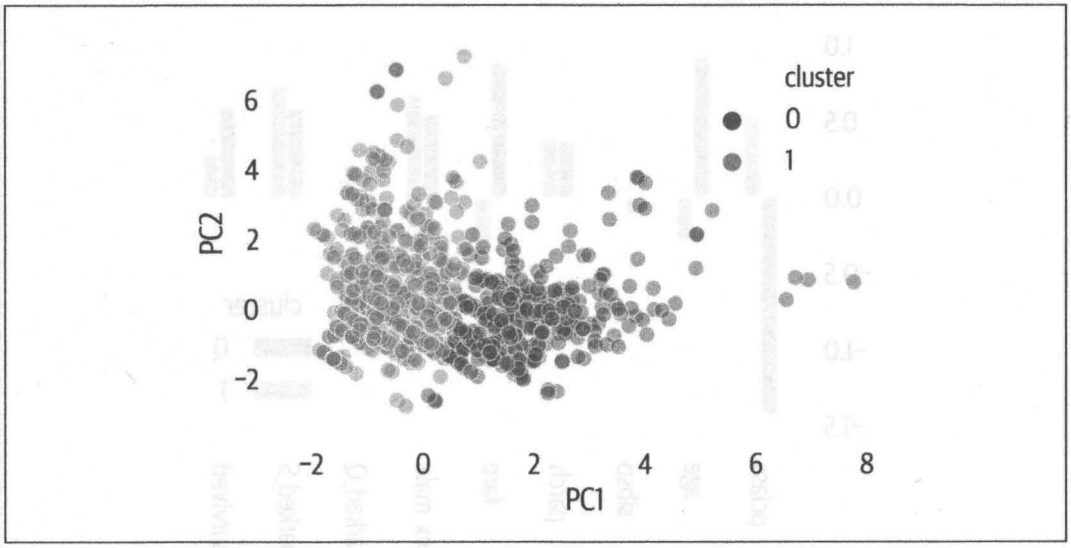



*Figure 18.7 : Diagramme PCA des grappes.*

Pour entrer dans les détails d'une caractéristique, nous nous servons de la méthode de pandas nommée `.describe` :

```
>>> (
...     X.assign(cluster=label)
...     .groupby("cluster")
...     .age.describe()
...     .T
... )
cluster           0           1
count    362.000000  947.000000
mean       0.921668   -0.280471
std        1.070188    0.808101
min       -2.160126   -2.218578
25%        0.184415   -0.672870
50%        0.867467   -0.283195
75%        1.665179    0.106480
max        4.003228    3.535618
```

Nous pouvons en outre produire un modèle substitut pour expliquer nos grappes. Choisissons par exemple un arbre de décision. Il montre que *pclass* (dont la moyenne comporte de grandes différences) est une caractéristique très importante :

```
>>> dt = tree.DecisionTreeClassifier()
>>> dt.fit(X, labels)
>>> for col, val in sorted(
...     zip(X.columns, dt.feature_importances_),
...     key=lambda col_val: col_val[1],
...     reverse=True,
```

```
... ):
...     print(f"{col:10}{val:10.3f}")
pclass        0.902
age           0.074
sex_male      0.016
embarked_S    0.003
fare          0.003
parch         0.003
sibsp         0.000
embarked_Q    0.000
```

Nous pouvons enfin visualiser les décisions (Figure 18.8). Nous confirmons que *pclass* est la première caractéristique étudiée par le substitut pour prendre ses décisions :

```
>>> dot_data = StringIO()
>>> tree.export_graphviz(
...     dt,
...     out_file=dot_data,
...     feature_names=X.columns,
...     class_names=["0", "1"],
...     max_depth=2,
...     filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
...     dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1808.png")
```

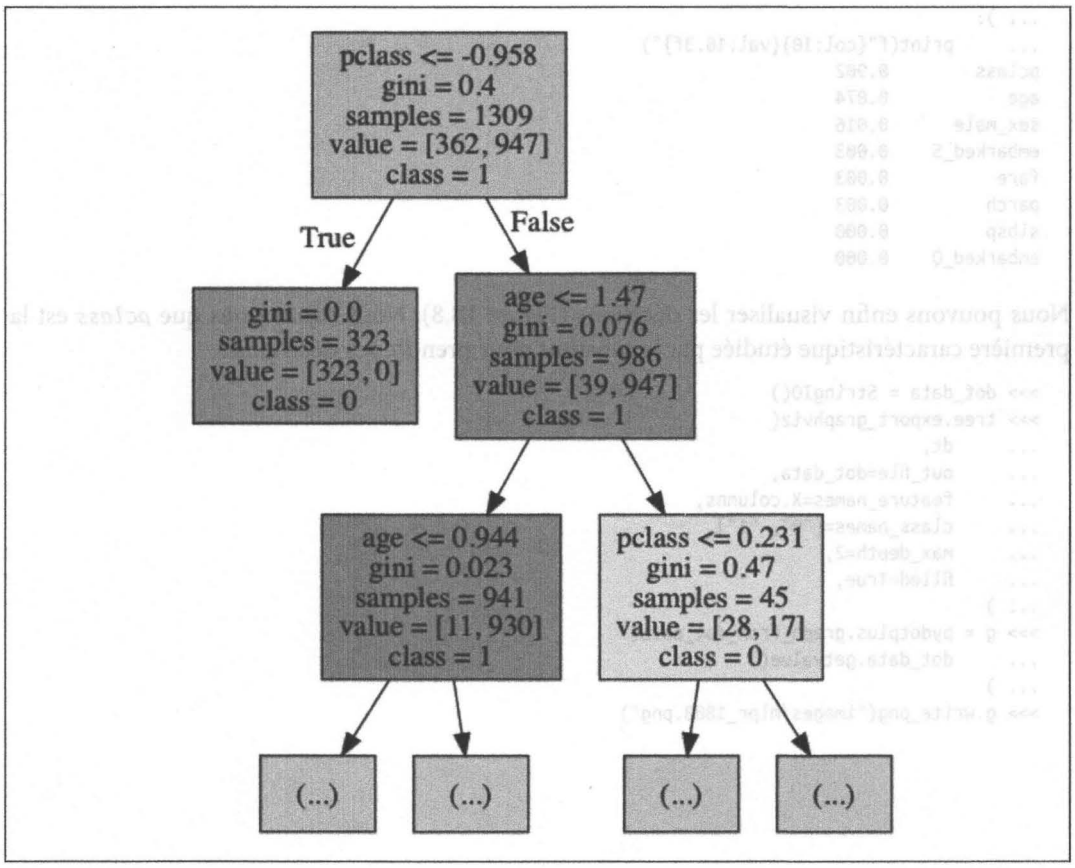



*Figure 18.8 : Arbre de décision expliquant le regroupement.*

# CHAPITRE 19
# Pipelines

La librairie **scikit-learn** connaît le concept de pipeline. Sa classe `Pipeline` permet d'enchaîner des transformeurs et des modèles en considérant le processus global comme un nouveau modèle **scikit-learn**. Vous pouvez bien sûr y ajouter de la logique spécifique.

## Pipeline de classification

Voici un exemple qui définit une fonction nommée *`tweak_titanic`* à l'intérieur d'un pipeline :

```
>>> from sklearn.base import (
...     BaseEstimator,
...     TransformerMixin,
... )
>>> from sklearn.pipeline import Pipeline

>>> def tweak_titanic(df):
...     df = df.drop(
...     columns=[
...         "name",
...         "ticket",
...         "home.dest",
...         "boat",
...         "body",
...         "cabin",
... ...     ]
...     ).pipe(pd.get_dummies, drop_first=True)
...     return df

>>> class TitanicTransformer(
...     BaseEstimator, TransformerMixin
... ):
...     def transform(self, X):
...         # assumes X is output
```

```
...         # from reading Excel file
...         X = tweak_titanic(X)
...         X = X.drop(column="survived")
...         return X
...
...     def fit(self, X, y):
...         return self

>>> pipe = Pipeline(
...     [
...         ("titan", TitanicTransformer()),
...         ("impute", impute.IterativeImputer()),
...         (
...             "std",
...             preprocessing.StandardScaler(),
...         ),
...         ("rf", RandomForestClassifier()),
...     ]
... )
```

Une fois ce pipeline mis en place, nous pouvons lui appliquer `.fit` et `.score` :

```
>>> from sklearn.model_selection import (
...     train_test_split,
... )
>>> X_train2, X_test2, y_train2, y_test2 =
train_test_split(
...     orig_df,
...     orig_df.survived,
...     test_size=0.3,
...     random_state=42,
... )
>>> pipe.fit(X_train2, y_train2)
>>> pipe.score(X_test2, y_test2)
0.7913486005089059
```

Vous pouvez utiliser un pipeline dans une recherche grille. L'élément nommé `param_grid` doit recevoir les paramètres après qualification de leur nom : il faut d'abord indiquer le nom du pipeline suivi de deux caractères de soulignement, comme dans `rf__`. Dans cet exemple, nous spécifions deux paramètres pour l'étape de forêt aléatoire (`rf`) :

```
>>> params = {
...     "rf__max_features": [0.4, "auto"],
...     "rf__n_estimators": [15, 200],
... }

>>> grid = model_selection.GridSearchCV(
...     pipe, cv=3, param_grid=params
... )
>>> grid.fit(orig_df, orig_df.survived)
```

Nous pouvons ensuite demander les meilleurs paramètres et lancer l'entraînement du modèle final. (Dans l'exemple, l'étape de forêt aléatoire n'améliore rien après la recherche grille.)

```
>>> grid.best_params_
{'rf__max_features': 0.4, 'rf__n_estimators': 15}
>>> pipe.set_params(**grid.best_params_)
>>> pipe.fit(X_train2, y_train2)
>>> pipe.score(X_test2, y_test2)
0.7913486005089059
```

Nous pouvons ensuite utiliser ce pipeline comme nous utilisons les autres modèles de **scikit-learn** :

```
>>> metrics.roc_auc_score(
...     y_test2, pipe.predict(X_test2)
... )
0.7813688715131023
```

# Pipeline de régression

Voici un exemple de pipeline pour effectuer une régression linéaire sur le jeu de données immobilières de Boston :

```
>>> from sklearn.pipeline import Pipeline
>>> reg_pipe = Pipeline(
...     [
...         (
...             "std",
...             preprocessing.StandardScaler(),
...         ),
...         ("lr", LinearRegression()),
...     ]
... )

>>> reg_pipe.fit(bos_X_train, bos_y_train)
>>> reg_pipe.score(bos_X_test, bos_y_test)
0.7112260057484934
```

Nous pouvons analyser les différentes étapes du pipeline et leurs propriétés au moyen de l'attribut nommé `.named_steps` :

```
>>> reg_pipe.named_steps['lr'].intercept_
23.01581920903956
>>> reg_pipe.named_steps['lr'].coef_
array([-1.10834602, 0.80843998, 0.34313466, 0.81386426,
       -1.79804295, 2.913858 , -0.29893918, -2.94251148,
        2.09419303, -1.44706731, -2.05232232, 1.02375187,
       -3.88579002])_
```

Le pipeline peut également être utilisé dans les calculs de métriques :

```
>>> from sklearn import metrics
>>> metrics.mean_squared_error(
...     bos_y_test, reg_pipe.predict(bos_X_test)
... )
21.517444231177205
```

# Pipeline PCA

Un pipeline **`scikit-learn`** est également utilisable pour une analyse PCA.

Standardisons le jeu Titanic puis appliquons-lui une PCA :

```
>>> pca_pipe = Pipeline(
...     [
...         (
...             "std",
...             preprocessing.StandardScaler(),
...         ),
...         ("pca", PCA()),
...     ]
... )
>>> X_pca = pca_pipe.fit_transform(X)
```

L'attribut `.named_steps` permet de récupérer les propriétés de la partie PCA du pipeline :

```
>>> pca_pipe.named_steps[
...     "pca"
... ].explained_variance_ratio_
array([0.23917891, 0.21623078, 0.19265028,
       0.10460882, 0.08170342, 0.07229959,
       0.05133752, 0.04199068])
>>> pca_pipe.named_steps["pca"].components_[0]
array([-0.63368693, 0.39682566, 0.00614498,
        0.11488415, 0.58075352, -0.19046812,
       -0.21190808, -0.09631388])
```

# Index

## Symboles



## A



## B



## C

## D

## E



## F



## G



## H

## I



## J



## K



## L

## M



## N



## O

## P



## Q



## R



## S

## T



## U



## V



## W



## X



## Y

O'REILLY®

# Machine learning : les fondamentaux

Avec plus de 200 extraits de code et des dizaines de notes techniques, ce guide de référence pratique se propose de vous aider à tracer votre route dans le domaine de l'apprentissage machine avec des données structurées. Son auteur, Matt Harrison, a produit un guide précieux qui va constituer une ressource utile dans vos prochains projets de datalogie.

Destiné aux programmeurs, aux datalogues et aux ingénieurs en science des données, le livre aborde toutes les techniques actuelles de traitement et de visualisation de données structurées fondées sur l'approche de classification ou sur celle de régression. Il met à contribution des dizaines de librairies spécifiques.

- Techniques de classification avec le jeu de données Titanic
- Nettoyage des données et traitement des manquants
- Analyse exploratoire de données
- Prétraitements, confinement et variables factices
- Sélection de caractéristiques, colinéarité et PCA
- Modèles de classification (bayésien, SVM, KNN, forêts, etc.)
- Métriques et évaluation d'une classification
- Modèles de régression (XGBoost, arbre de décision, SVM, KNN, etc.)
- Métriques et évaluation d'une régression (hétéroscédasticité, résidus)
- Regroupement clustering (k-moyennes, analyse de grappes)
- Réduction de dimensionnalité (PCA, UMAP, T-SNE, PHATE)
- Pipelines de Scikit-learn

« Matt Harrison sait mieux que personne résoudre de façon pratique des problèmes concrets avec les outils de Python dédiés à la science des données. Son ouvrage est un livre de chevet pour tous les programmeurs et datalogues qui exploitent des données structurées en Python. »

Chris Moffit
Practical Business Python

**Matt Harrison** dirige la société de formation et de services autour de Python et de la datalogie MetaSnake. Il utilise le langage Python depuis le début des années 2000 dans divers domaines : datalogie, gestion d'entreprise, stockage de données, tests et automatisation, gestion de piles logicielles open source, finances et recherche.

MIXTE
Papier issu de sources responsables
FSC® C003309
www.fsc.org

editionsfirst.fr

FIRST INTERACTIVE

732298 III-20 25,95 €

9 782412 056028