

Technique de Big Data Analytics avec Python

Cours conçu par **Razvan BIZOÏ**

Razvan.BIZOI@laposte.net

Table des matières

MODULE 1 : L'INTRODUCTION.....	1-1
Un peu d'histoire	1-2
Présentation du langage Python	1-3
Installation sous Linux	1-5
Installation sous Windows.....	1-6
L'écosystème Python scientifique.....	1-7
PyCharm	1-10
MODULE 2 : L'INTRODUCTION A PYTHON.....	2-1
Le nom de variable	2-2
L'affectation.....	2-3
La réaffectation.....	2-5
La fonction input	2-6
Les types de données	2-7
Les types numériques	2-8
Le type integer	2-9
Le type float.....	2-11
Les types booléens.....	2-13
Les instructions conditionnelles	2-14
Les blocs d'instructions.....	2-16
L'instruction while	2-17
Les listes.....	2-19
Les tuples	2-26
Les dictionnaires	2-28
Les chaînes de caractères	2-32

Manipulation d'une chaîne.....	2-34
Formatage d'une chaîne	2-37
Le type bytes et la page de code	2-39
L'instruction for	2-41
Les expressions régulières	2-42
Les symboles simples.....	2-43
Les symboles de répétition.....	2-46
Les symboles de regroupement	2-48
Les fonctions et objets de re	2-50
L'écriture simplifiée des fonctions.....	2-51
La fonction map.....	2-53
La fonction filter.....	2-54
MODULE 3 : LES OUTILS INDISPENSABLES	3-1
Les tableaux numériques	3-2
La création de tableaux.....	3-4
L'affichage des tableaux	3-6
L'accès aux composantes d'un tableau	3-7
Lecture et écriture d'un tableau	3-8
Les opérations simples sur les tableaux.....	3-11
La librairie Matplotlib	3-14
Visualiser l'incertitude	3-16
La personnalisation et sous-graphes.....	3-18
La librairie Seaborn	3-27
MODULE 4 : LES DATAFRAMES.....	4-1
Les structures de données.....	4-2
Lecture et écriture de DataFrame	4-5
La projection et la restriction.....	4-8
La restriction	4-13
L'union	4-14
La jointure	4-16
L'agrégation.....	4-19
MODULE 5 : UN PROBLEME DE DATA SCIENCE	5-1
MODULE 6 : LES ALGORITHMES DE REGRESSION	6-1
La régression linéaire univariée	6-8

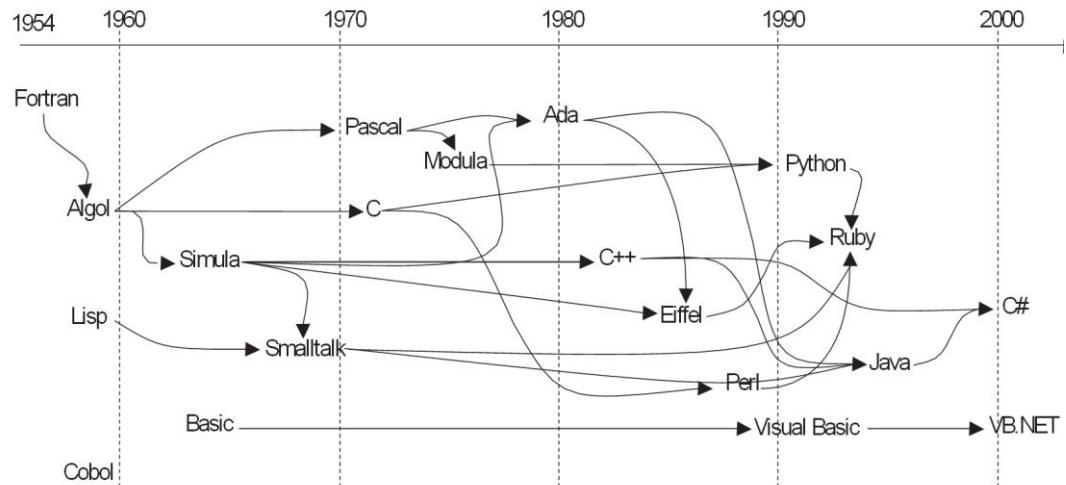
MODULE 7 : LES ESPACES DE GRANDE DIMENSION.....	7-1
MODULE 8 : L'ANALYSE FACTORIELLE	8-1
L'analyse factorielle.....	8-2
L'analyse en composante principale	8-3
Le centrage et réduction des données	8-4
L'étude du nuage des individus	8-5
L'étude du nuage des individus	8-7
L'analyse en Composante Principale ACP.....	8-9
L'analyse des correspondances multiples ACM	8-14
L'analyse factorielle des correspondances AFC	8-16
L'analyse des correspondances multiples ACM	8-19
MODULE 9 : LES ALGORITHMES DE CLASSIFICATIONS	9-1

1

L'introduction

Un peu d'histoire

Un peu d'histoire



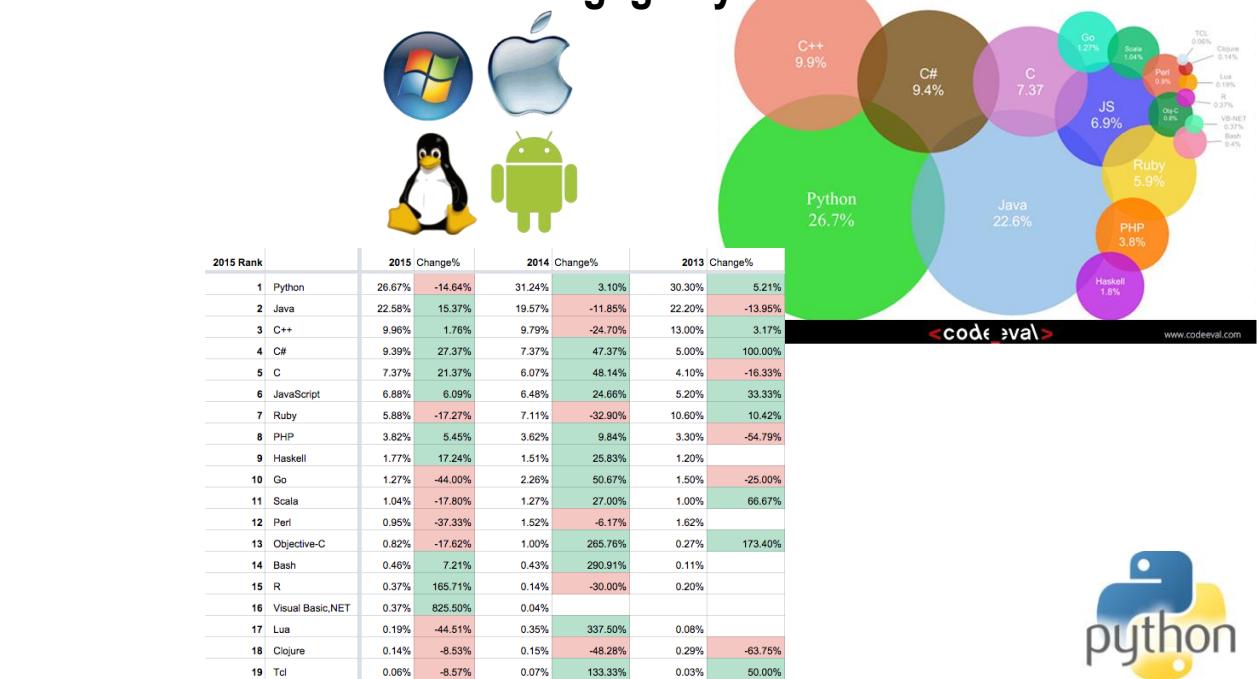
Les langages de programmation ont ouvert la voie de l'abstraction, principe fondamental qui cherche à éloigner autant que possible le programmeur des contingences matérielles, pour le rapprocher des concepts, de la "chose" concrète ou abstraite qu'il doit traiter.

C'est ainsi qu'au fil du temps différents ensembles de règles et de principes de conception de programmation se sont développés. Programmation structurée, fonctionnelle, orientée objet... À chaque évolution, l'objectif est toujours le même : permettre la meilleure représentation informatique possible, la plus intuitive possible, de l'idée sur laquelle opère le programme.

Il ne s'agit là que d'une vue vraiment très simplifiée, presque caricaturale. Mais vous pouvez constater que les langages s'inspirent fortement les uns des autres. Par ailleurs, la plupart de ceux présentés sur ce schéma sont toujours utilisés et continuent d'évoluer – c'est, par exemple, le cas du vénérable ancêtre Fortran, notamment dans les applications très gourmandes en calculs scientifiques.

Présentation du langage Python

Présentation du langage Python



Détaillons un peu les principales caractéristiques de Python, plus précisément, du langage et de ses deux implantations actuelles.

- Python est portable, non seulement sur les différentes variantes d'Unix, mais aussi sur les OS propriétaires.
- Python est gratuit, mais on peut l'utiliser sans restriction dans des projets commerciaux.
- Python convient aussi bien à des scripts d'une dizaine de lignes qu'à des projets complexes de plusieurs dizaines de milliers de lignes.
- La syntaxe de Python est très simple et, combinée à des types de données évolués (listes, dictionnaires...), conduit à des programmes à la fois très compacts et très lisibles. À fonctionnalités égales, un programme Python (abondamment commenté et présenté selon les canons standards) est souvent de 3 à 5 fois plus court qu'un programme C ou C++ (ou même Java) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.
- Python gère ses ressources (mémoire, descripteurs de fichiers...) sans intervention du programmeur, par un mécanisme de comptage de références (proche, mais différent, d'un garbage collector).
- Python est orienté-objet. Il supporte l'héritage multiple et la surcharge des opérateurs. Dans son modèle objets, et en reprenant la terminologie de C++, toutes les méthodes sont virtuelles.
- Python intègre, comme Java ou les versions récentes de C++, un système d'exceptions, qui permettent de simplifier considérablement la gestion des erreurs.
- Python est **dynamique** (l'interpréteur peut évaluer des chaînes de caractères représentant des expressions ou des instructions Python), **orthogonal** (un petit nombre de concepts suffit à engendrer des constructions très riches), **réfléctif** (il supporte la métaprogrammation, par exemple la capacité pour un objet de se

rajouter ou de s'enlever des attributs ou des méthodes, ou même de changer de classe en cours d'exécution) et **introspectif** (un grand nombre d'outils de développement, comme le debugger ou le profiler, sont implantés en Python lui-même).

- Comme Scheme ou SmallTalk, Python est dynamiquement typé. Tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance.
- Python possède actuellement deux implémentations. L'une, interprétée, dans laquelle les programmes Python sont compilés en instructions portables, puis exécutés par une machine virtuelle (comme pour Java, avec une différence importante : Java étant statiquement typé, il est beaucoup plus facile d'accélérer l'exécution d'un programme Java que d'un programme Python). L'autre génère directement du bytecode Java.
- Python est extensible : comme Tcl ou Guile, on peut facilement l'interfacer avec des bibliothèques C existantes. On peut aussi s'en servir comme d'un langage d'extension pour des systèmes logiciels complexes.
- La bibliothèque standard de Python, et les paquetages contribués, donnent accès à une grande variété de services : chaînes de caractères et expressions régulières, services UNIX standards (fichiers, pipes, signaux, sockets, threads...), protocoles Internet (Web, News, FTP, CGI, HTML...), persistance et bases de données, interfaces graphiques.
- Python est un langage qui continue à évoluer, soutenu par une communauté d'utilisateurs enthousiastes et responsables, dont la plupart sont des supporters du logiciel libre. Parallèlement à l'interpréteur principal, écrit en C et maintenu par le créateur du langage, un deuxième interpréteur, écrit en Java, est en cours de développement.

Installation sous Linux

Installation sous Linux



```
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tgz
tar xzf Python-3.6.1.tgz
cd Python-3.6.1
./configure
make
make install

which python3
python3 -v
```



```
wget \
https://repo.continuum.io/archive/Anaconda3-4.3.1-Linux-x86_64.sh

bash Anaconda3-4.3.1-Linux-x86_64.sh
```

Python est souvent préinstallé sur la plupart des distributions GNU/Linux. Vous pouvez contrôler sa présence en tapant la commande `python` dans un terminal.



```
[root@ocean Python-3.6.1]# which python3
/usr/local/bin/python3
[root@ocean Python-3.6.1]# python3 -v
Python 3.6.1
[root@ocean Python-3.6.1]# which python
/usr/bin/python
[root@ocean Python-3.6.1]# python -v
Python 2.6.6
```

S'il vous est nécessaire d'installer Python ou de mettre à jour une version existante, vous pouvez le faire par le biais du système de package de la distribution ou le recompiler manuellement. Les manipulations d'installation se font en tant que super-utilisateur, ou root.



```
[root@mercure ~]# yum -y install gcc make zlib-devel
[root@mercure ~]# wget
https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tgz
...
[root@mercure ~]# tar xzf Python-3.6.1.tgz
[root@mercure ~]# cd Python-3.6.1
[root@mercure Python-3.6.1]# ./configure
...
[root@mercure Python-3.6.1]# make
...
Successfully installed pip-9.0.1 setuptools-28.8.0
[root@mercure Python-3.6.1]# which python3
/usr/local/bin/python3
[root@mercure Python-3.6.1]# python3 -v
Python 3.6.1
```

Installation sous Windows

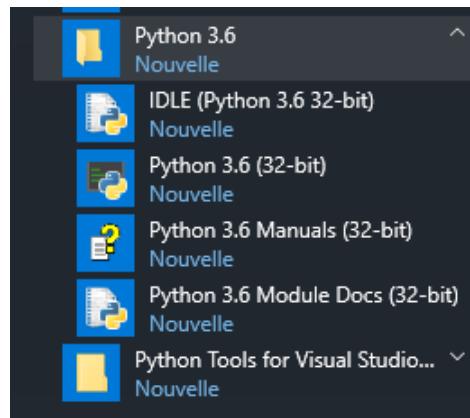
Installation sous Windows



The screenshot shows a web browser displaying the Python Releases for Windows page at <https://www.python.org/downloads/windows/>. The page features the Python logo and navigation links for About, Downloads, Documentation, Community, Success Stories, News, and Events. Below the navigation bar, there are links for Python 2 and Python 3 releases, and a section for Python 3.6.1 from March 2017, which includes links for various installer types. To the right of the main content, a file download dialog box is open, showing the file "python-3.6.1-amd64.exe" with a size of 29.9 Mo. It asks if the user wants to save the file, with "Enregistrer le fichier" (Save) and "Annuler" (Cancel) buttons.

Les plates-formes Windows bénéficient d'un installateur graphique automatique, présenté sous la forme d'un assistant. Si vous n'avez pas les droits administrateurs sur la machine, il est possible de sélectionner dans les options avancées une installation en tant que non-administrateur. L'installation de Python par ce biais ne présente aucune difficulté.

Une fois l'installation achevée, une nouvelle entrée Python apparaît dans le menu Démarrer>Programmes.



The screenshot shows a terminal window titled "Python 3.6.1 Shell". The window title bar includes "File Edit Shell Debug Options Window Help". The main area of the window displays a Python shell session. The output shows the Python version (3.6.1), the date and time of compilation (Mar 21 2017, 18:41:36), the build number (MSC r1900 64 bit (AMD64)), and the platform (win32). The user then types "print('Bonjour !')", followed by three greater-than signs (>>>), and the response "Bonjour !" is shown. At the bottom right of the window, there is a status bar with "Ln: 5 Col: 4".

L'écosystème Python scientifique

L'écosystème Python scientifique



[Download Anaconda Now...](https://www.continuum.io/downloads)

Download for Windows Download for macOS Download for Linux

Anaconda 4.3.1

For Windows

Anaconda is BSD licensed which gives you permission to use Anaconda commercially and for redistribution.

[Changelog](#)

1. Download the installer
2. Optional: Verify data integrity with [MD5](#) or [SHA-256](#) [More info](#)
3. Double-click the .exe file to install Anaconda and follow the instructions on the screen

Behind a firewall? Use these [zipped Windows installers](#)

Python 3.6 version

64-BIT INSTALLER (422M)

32-BIT INSTALLER (348M)

Python 2.7 version

64-BIT INSTALLER (414M)

Au fil des années Python est devenu un outil du quotidien pour les ingénieurs et chercheurs de toutes les disciplines scientifiques. Il est devenu un des outils incontournables des Data Scientists!

Pourquoi utiliser Python pour le calcul scientifique ?

Python est devenu une alternative viable aux solutions propriétaires leader du marché, comme MatLab, Maple, Mathematica, Statistica, ...

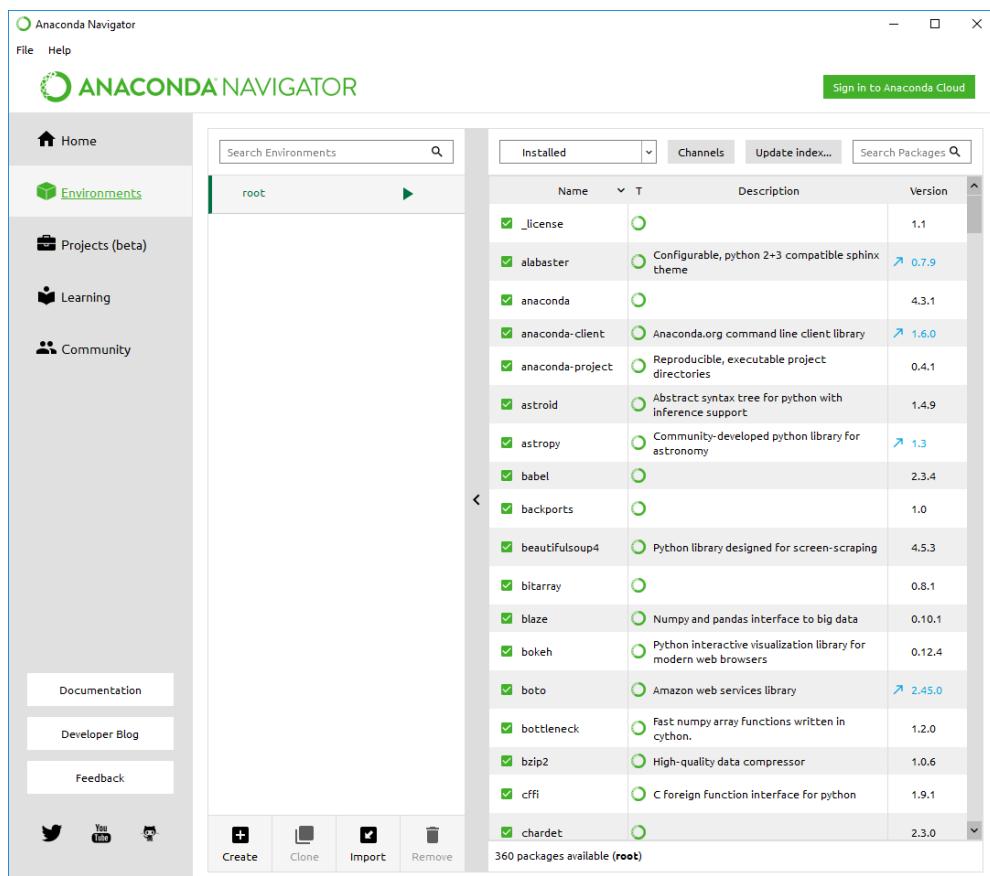
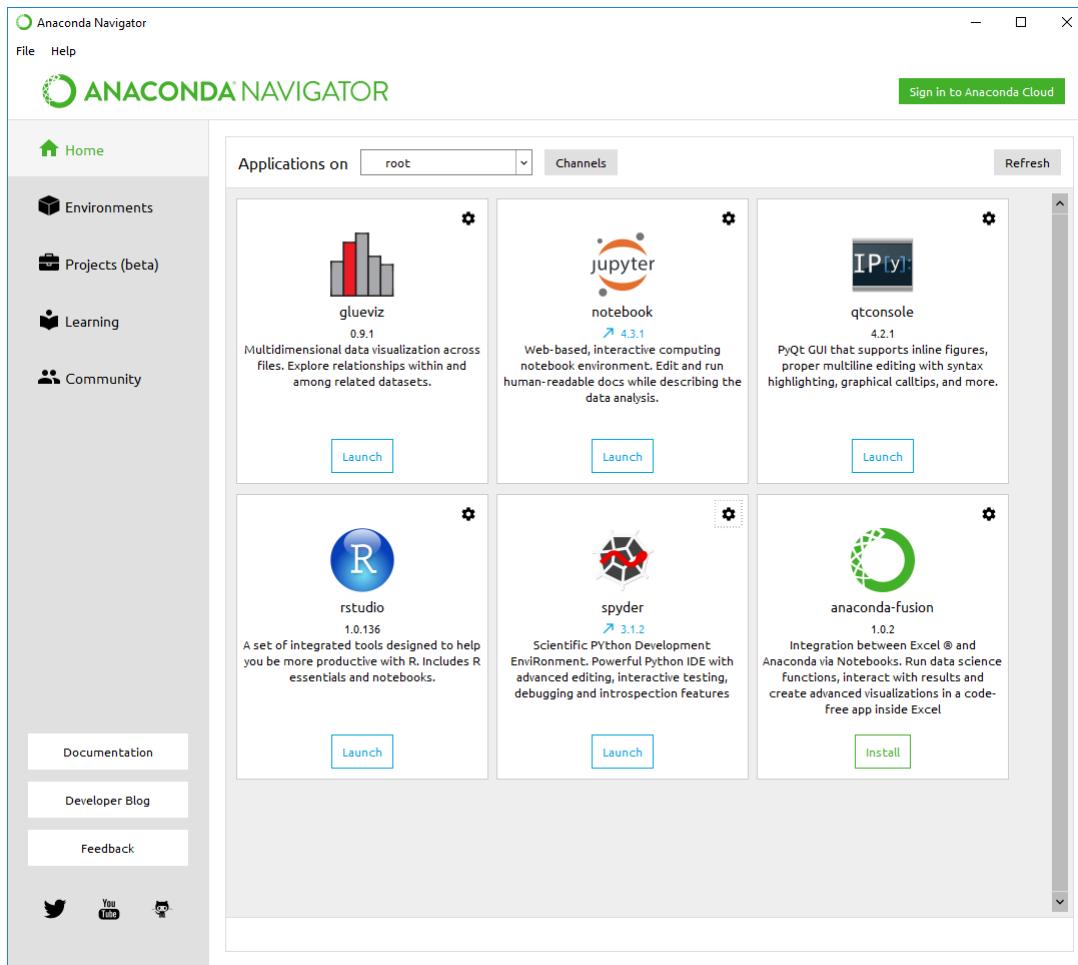
Anaconda

Anaconda est peut-être la distribution la plus répandue en raison de la diversité des plateformes qu'elle supporte et d'une plus grande ouverture des outils sur lesquels elle se base.

Elle propose une distribution communautaire et partiellement optimisée ainsi qu'une version payante mieux optimisée offrant de nombreux outils complémentaires.

Elle est « cross-platform » et s'appuie sur l'outil de virtualisation d'environnement « conda »

Elle inclut plus de 100 librairies pré-installées et en propose 600 de plus en téléchargement. Enfin, elle est compatible avec « pip ».



Jupyter Notebook est un projet open source dérivé d'iPython et fournit une interface web riche dans le cadre d'une programmation interactive.

Jupyter Notebook permet à ses utilisateurs de travailler sur des documents associant du code, du texte, des équations, des images, des vidéos et des graphiques.

The screenshot shows a Jupyter Notebook interface with the title "Afficher le résultat d'un calcul". The notebook contains the following content:

Premier essai de script Python

```
Un petit programme simple affichant une suite de Fibonacci, une suite de nombres dont chaque terme est égal à la somme des deux précédents.
```

```
In [3]: print("Suite de Fibonacci :")
print("chaque terme est égal à la somme des deux précédents.")

Suite de Fibonacci :
chaque terme est égal à la somme des deux précédents.

a & b servent au calcul des termes successifs c est un simple compteur
```

```
In [2]: a,b,c = 1,1,1

affichage du premier terme
```

```
In [5]: print(b)

1

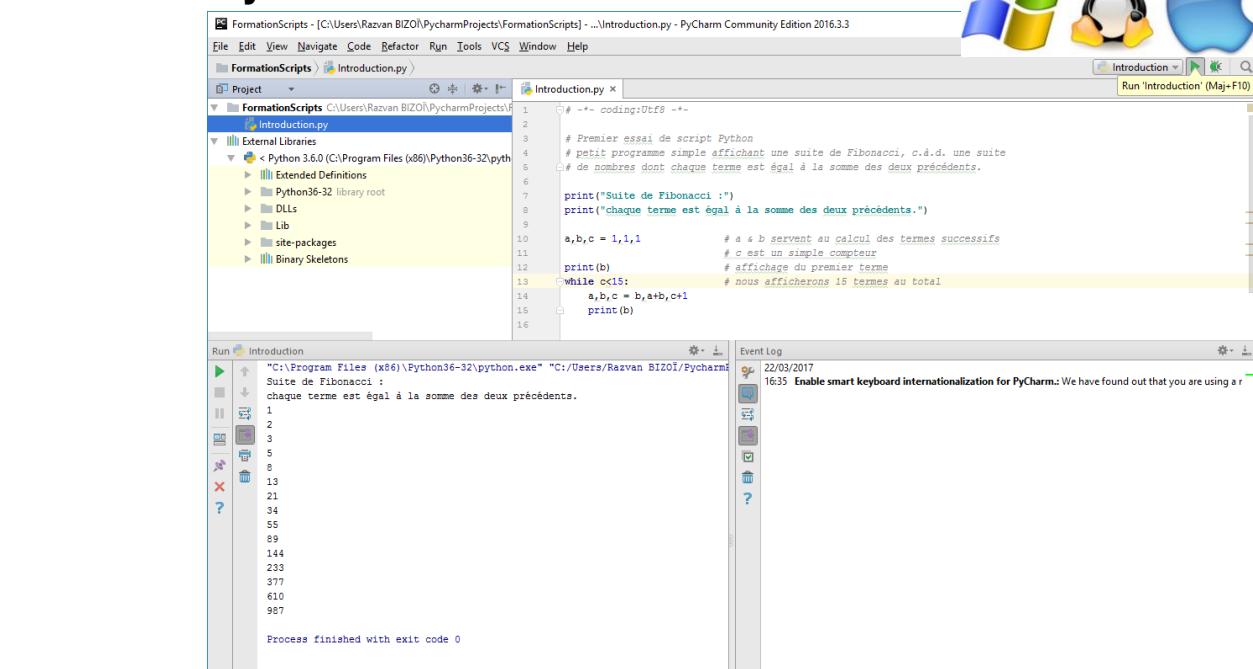
nous afficherons 15 termes au total
```

```
In [6]: while c < 15:
    a,b,c = b,a+b,c+1
    print(b)

2
3
5
8
13
21
34
55
89
144
233
377
610
987
```

PyCharm

PyCharm

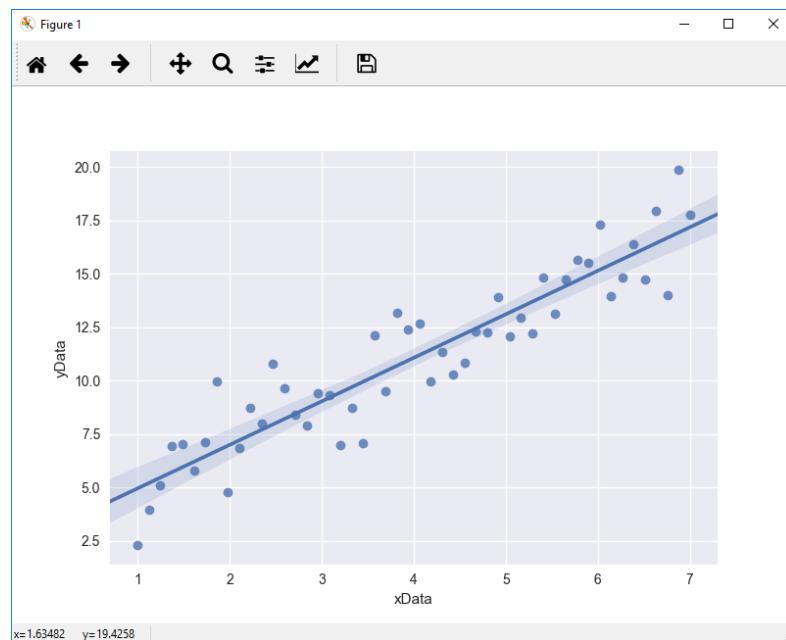


```

import numpy as np
import matplotlib.pyplot as plt

import pandas as pd
import seaborn as sns
x = np.linspace(1, 7, 50)
y = 3 + 2*x + 1.5*np.random.randn(len(x))
df = pd.DataFrame({'xData':x, 'yData':y})
sns.regplot('xData', 'yData', data=df)
plt.show()

```



2

L'introduction à Python

Le nom de variable

Pour pouvoir accéder aux données, le programme d'ordinateur fait abondamment usage d'un grand nombre de variables de différents types. Une variable est une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive.

Les noms de variables doivent obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (**a** ÷ **z**, **A** ÷ **Z**) et de chiffres (**0** ÷ **9**), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que **\$**, **#**, **@**, etc. sont interdits, à l'exception du caractère **_** (souligné).
- La casse est significative (les caractères majuscules et minuscules sont distingués). Attention : **Variable**, **variable**, **VARIABLE** sont donc des variables différentes.

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules. Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans **tableDesMatieres**.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser les mots réservés ci-dessous :

```
and      as          assert    break     class     continue  def
del      elif        else      except    False     finally   for
from     global       if        import   in        is         lambda
None     nonlocal    not      or        pass     raise    return
True     try         while    with     yield
```

Pour leur première initialisation, une variable reçoit dans la plupart des cas une constante. Les constantes sont le contraire des variables, ce sont toutes les valeurs numériques, chaînes de caractères, ..., tout ce qui n'est pas désigné par un nom. Les constantes possèdent un type mais pas d'identificateur.

L'affectation

L'affectation désigne l'opération par laquelle on établit un lien entre le nom de la variable et sa valeur. En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe égale « = ».

Python fait la distinction entre l'opérateur d'affectation « = » et l'opérateur de comparaison « == ».



```
>>> varInt = 12
>>> varStr = 'chaîne de caractères'
>>> varFloat = 15.5
>>> varInt == varFloat
False
>>> varInt == varStr
False
```

Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- créer et mémoriser un nom de variable ;
- lui attribuer un type bien déterminé ;
- créer et mémoriser une valeur particulière ;
- établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

Les trois noms de variables sont des références, mémorisées dans une zone particulière de la mémoire que l'on appelle **espace de noms**, alors que les valeurs correspondantes sont situées ailleurs, dans des emplacements parfois fort éloignés les uns des autres.

L'affichage de la valeur d'une variable

Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable. Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. À l'intérieur d'un programme, vous utiliserez toujours la fonction **print()**.



```
>>> varInt,varStr,varFloat
(12, 'chaîne de caractères', 15.5)
>>> varStr
'chaîne de caractères'
>>> print(varStr)
chaîne de caractères
```

La fonction **print()** n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode affiche aussi des apostrophes afin de vous rappeler que la variable traitée est du type « **chaîne de caractères** ».

Le typage des variables

Dans Python, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour définir le type des variables avant de pouvoir les utiliser. On dira à ce sujet que le typage des variables sous Python est un typage dynamique, par opposition au typage statique qui est de règle par exemple en C++ ou en Java.

Le typage dynamique quant à lui permet d'écrire plus aisément des constructions logiques de niveau élevé, en particulier dans le contexte de la programmation orientée objet. Il facilite également l'utilisation de structures de données très riches telles que les listes et les dictionnaires.



```
>>> varInt,varStr,varFloat
(12, 'chaîne de caractères', 15.5)
>>> varInt = varStr = varFloat
>>> varInt,varStr,varFloat
(15.5, 15.5, 15.5)
>>> varStr = 'chaîne de caractères'
>>> varInt,varStr,varFloat
(15.5, 'chaîne de caractères', 15.5)
>>> varInt == 20                      # ce n'est pas une affectation
False
>>> varInt,varStr,varFloat
(15.5, 'chaîne de caractères', 15.5)
>>> varInt = 20
>>> varInt,varStr,varFloat
(20, 'chaîne de caractères', 15.5)
```

Les affectations multiples

Dans Python, on peut assigner une valeur à plusieurs variables simultanément. On peut aussi effectuer des affectations parallèles à l'aide d'un seul opérateur.



```
>>> varInt,varStr,varFloat
(12, 'chaîne de caractères', 15.5)
>>> varInt = varStr = varFloat
>>> varInt,varStr,varFloat
(15.5, 15.5, 15.5)
>>> varInt,varStr,varFloat = (30, "l'autre chaîne", 3.14)
>>> varInt,varStr,varFloat
(30, "l'autre chaîne", 3.14)
>>> r , pi = 12, 3.14159
>>> s = pi * r**2
>>> print(s)
452.38896
>>> print(type(r), type(pi), type(s))
<class 'int'> <class 'float'> <class 'float'>
>>> h, m, s = 15, 27, 34
>>> print("secondes écoulées depuis minuit = ", h*3600 + m*60 + s)
secondes écoulées depuis minuit = 55654
```

La virgule, est très généralement utilisée pour séparer différents éléments comme on le voit dans notre exemple, pour les variables elles-mêmes ainsi que pour les valeurs qu'on leur attribue.

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de règles de priorité. Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique. Vous pouvez les mémoriser aisément à l'aide d'un acronyme mnémotechnique, **PEMDAS** (parenthèses, exposants, multiplications, divisions, additions et soustractions).

La réaffectation

L'effet d'une réaffectation est de remplacer l'ancienne valeur d'une variable par une nouvelle.



```
>>> altitude = 320
>>> print(altitude)
320
>>> altitude = 375
>>> print(altitude)
375
>>> a = 5
>>> b = a      # a et b contiennent des valeurs égales
>>> b = 2      # a et b sont maintenant différentes
>>> print("a = ",a,"b = ",b)
a = 5 b = 2
```

Rappelons ici que Python permet d'affecter leurs valeurs à plusieurs variables simultanément.



```
>>> a, b, c, d = 3, 4, 5, 7
>>> print("a = ",a,"b = ",b,"c = ",c,"d = ",d)
a = 3 b = 4 c = 5 d = 7
```

Supposons par exemple que nous voulions maintenant échanger les valeurs des variables **a** et **b**.

```
>>> a,b = b,a
>>> print("a = ", a,"b = ", b)
a = 4 b = 3
```

On pourrait bien entendu échanger d'autres variables en même temps, dans la même instruction.

La fonction `input`

La fonction intégrée « `input` » provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec « `Enter` ».



```
>>> prenom = input("Entrez votre prénom : ")
Entrez votre prénom : Razvan
>>> print("Bonjour,", prenom)
Bonjour, Razvan
>>> print("Veuillez entrer un nombre positif quelconque : ")
Veuillez entrer un nombre positif quelconque :
>>> 10
10
>>> print("Veuillez entrer un nombre positif quelconque : ",
... end=" ")
Veuillez entrer un nombre positif quelconque :
>>> 10
10
>>> ch = input()
20
>>> nn = int(ch)      # conversion de la chaîne en un nombre entier
>>> print("Le carré de", nn, "vaut", nn**2)
Le carré de 20 vaut 400
>>> a = input("Entrez une donnée numérique : ")
Entrez une donnée numérique : 25
>>> print(a)
25
```

Soulignons que la fonction « `input` » renvoie toujours une chaîne de caractères. Si vous souhaitez que l'utilisateur entre une valeur numérique, vous devrez donc convertir la valeur entrée en une valeur numérique du type qui vous convient, par l'intermédiaire des fonctions intégrées « `int` » ou « `float` ».

Atelier 2 - Exercice 1.1

Les types de données

Les types incontournables en Python sont classés le plus souvent en deux catégories : types immuables ou modifiables. Tous les types du langage Python sont également des objets, c'est pourquoi on retrouve dans ce chapitre certaines formes d'écriture similaires à celles présentées plus tard dans le chapitre concernant les classes.

Types immuables

Une variable de type immuable ne peut être modifiée. Une opération sur une variable de ce type entraîne nécessairement la création d'une autre variable du même type, même si cette dernière est temporaire.

Autrement dit, la simple instruction `x + = 3` qui consiste à ajouter à la variable `x` la valeur `3` crée une seconde variable dont la valeur est celle de `x` augmentée de `3` puis à en recopier le contenu dans celui de la variable `x`. Les nombres sont des types immuables tout comme les chaînes de caractères et les tuple qui sont des tableaux d'objets. Il n'est pas possible de modifier une variable de ce type, il faut en recréer une autre du même type qui intégrera la modification.

Type « rien »

Python propose un type `None` pour signifier qu'une variable ne contient rien. La variable est de type `None` et est égale à `None`.

```
>>> s = None
>>> print(s) # affiche None
None
```

Certaines fonctions utilisent cette convention lorsqu'il leur est impossible de retourner un résultat. Ce n'est pas la seule option pour gérer cette impossibilité : il est possible de générer une exception, de retourner une valeur par défaut ou encore de retourner `None`. Il n'y a pas de choix meilleur, il suffit juste de préciser la convention choisie.

Les types numériques

Il existe deux types de nombres en Python, les nombres réels « **float** » et les nombres entiers « **int** ». L'instruction **x = 3** crée une variable de type « **int** » initialisée à **3** tandis que **y = 3.0** crée une variable de type « **float** » initialisée à **3.0**. Le programme suivant permet de vérifier cela en affichant pour les variables **x** et **y**, leurs valeurs et leurs types respectifs grâce à la fonction **type**.

```
>>> x = 3
>>> y = 3.0
>>> print ("x =", x, type(x))
x = 3 <class 'int'>
>>> print ("y =", y, type(y))
y = 3.0 <class 'float'>
```

La liste des opérateurs qui s'appliquent aux nombres réels et entiers.

opérateur	signification	exemple
<< >>	décalage à gauche, à droite	x = 8 << 1 (résultat = 16)
 	opérateur logique ou bit à bit	x = 8 1 (résultat = 9)
&	opérateur logique et bit à bit	x = 11&2 (résultat = 2)
+ -	addition, soustraction	x = y + z
+= -=	addition ou soustraction puis affectation	x += 3
* /	multiplication, division le résultat est de type réel si l'un des nombres est réel	x = y * z
//	division entière	x = y // 3
%	reste d'une division entière (modulo)	x = y % 3
*= /=	multiplication ou division puis affectation	x *= 3
**	puissance (entière ou non, racine carrée = ** 0.5)	x = y ** 3

Les fonctions « **int** » et « **float** » permettent de convertir un nombre quelconque ou une chaîne de caractères respectivement en un entier (arrondi) et en un nombre réel.

```
>>> x = int (3.5)
>>> y = float (3)
>>> z = int ("3")
>>> print ("x:", type(x), " y:", type(y), " z:", type(z))
x: <class 'int'> y: <class 'float'> z: <class 'int'>
```

Il existe un cas pour lequel cet opérateur cache un sens implicite : lorsque la division opère sur deux nombres entiers ainsi que le montre l'exemple suivant.

```
>>> x = 11
>>> y = 2
>>> z = x // y
>>> z #le résultat est 5 et non 5.5 car la division est entière
5
>>> z = x / y
>>> z #le résultat est 5.5 car c'est une division entre deux réels
5.5
```

Le type integer

Le type integer en Python, est un entier représenté sous forme décimale, binaire, octale ou hexadécimale. Il n'y a pas de limite de représentation pour les entiers longs mise à part la mémoire virtuelle disponible de l'ordinateur.



```
>>> a, b, c = 1, 1, 1
>>> while c < 80 :
...     print(c,":",b,type(b))
...     a, b, c = b, a+b, c+1
...
1 : 1 <class 'int'>
2 : 2 <class 'int'>
...
74 : 2111485077978050 <class 'int'>
75 : 3416454622906707 <class 'int'>
76 : 5527939700884757 <class 'int'>
77 : 8944394323791464 <class 'int'>
78 : 14472334024676221 <class 'int'>
79 : 23416728348467685 <class 'int'>
```

Python est capable de traiter des nombres entiers de taille illimitée. La fonction **type()** nous permet de vérifier à chaque itération que le type de la variable **b** reste bien en permanence de ce type.



```
>>> a, b, c = 3, 2, 1
>>> while c < 15:
...     print(c, ": ", b)
...     a, b, c = b, a*b, c+1
...
1 : 2
...
12 :
64880030544660752790736837369104977695001034284228042891827649456186
234582611607420928
13 :
70056698901118320029237641399576216921624545057972697917383692313271
75488362123506443467340026896520469610300883250624900843742470237847
552
14 :
45452807645626579985636294048249351205168239870722946151401655655658
39864222761633581512382578246019698020614153674711609417355051422794
79530059170096950422693079038247634055829175296831946224503933501754
776033004012758368256
```

Vous pouvez donc effectuer avec Python des calculs impliquant des valeurs entières comportant un nombre de chiffres significatifs quelconque. Ce nombre n'est limité en effet que par la taille de la mémoire disponible sur l'ordinateur utilisé. Il va de soi cependant que les calculs impliquant de très grands nombres devront être décomposés par l'interpréteur en calculs multiples sur des nombres plus simples, ce qui pourra nécessiter un temps de traitement considérable dans certains cas.

La forme binaire est obtenue avec le préfixe « **0b** » ou « **0B** ». La fonction « **bin** » permet d'afficher la représentation binaire d'un entier.

```
>>> 0b0101101001
361
>>> bin(14)
```

```
'0b1110'
```

La forme octale est obtenue par une séquence de chiffres de 0 à 7, préfixée d'un « **0o** » ou « **0O** ». La fonction « **oct** » permet d'afficher la représentation octale d'un entier.

```
>>> 0o76453
```

```
32043
```

```
>>> oct(543)
```

```
'0o1037'
```

La forme hexadécimale est obtenue par une séquence de chiffres et de lettres de A à F, préfixée par la séquence « **0x** » ou « **0X** ».

```
>>> 0x3ef7b66
```

```
66026342
```

```
>>> hex(43676)
```

```
'0xaa9c'
```

Le type float

La représentation de valeurs à virgule flottante, que l'on notera littéraux réels, permet de décrire des valeurs réelles. Les parties entière et fractionnelle de la valeur réelle sont séparées par le signe « . », chaque partie étant composée de chiffres. Si le premier chiffre de la partie entière est 0, le nombre représenté ne sera néanmoins pas considéré comme un octal et restera traité en base 10. Les nombres à virgule flottante utilisés pour représenter des réels sont tous à double précision en Python, soit des nombres codés sur 64 bits.



```
>>> a, b, c = 1., 2., 1
>>> while c < 18:
...     print(c, ":", b)
...     a, b, c = b, a*b, c+1
...
1 : 2.0
2 : 2.0
3 : 4.0
4 : 8.0
5 : 32.0
6 : 256.0
7 : 8192.0
8 : 2097152.0
9 : 17179869184.0
10 : 3.602879701896397e+16
11 : 6.189700196426902e+26
12 : 2.2300745198530623e+43
13 : 1.3803492693581128e+70
14 : 3.078281734093319e+113
15 : 4.249103942534137e+183
16 : 1.307993905256674e+297
17 : inf
```

Au dixième terme, Python passe automatiquement à la notation scientifique « `e+n` » signifie en fait : « fois dix à l'exposant n ». Après le seizième terme, nous assistons à nouveau à un dépassement de capacité, ainsi les nombres vraiment trop grands sont tout simplement notés « `inf` ».

Ce type autorise les calculs sur de très grands ou très petits nombres, avec un degré de précision constant. Pour qu'une donnée numérique soit considérée par Python comme étant du type « **float** », il suffit qu'elle contienne dans sa formulation un élément tel qu'un point décimal ou un exposant de 10.

Attention à la conversion des valeurs de type « **integer** » en type « **float** », car il est effectué avec une perte de précision.



Les nombres complexes

Les nombres complexes sont formés d'un couple de nombres à virgule flottante et subissent donc les mêmes contraintes.



```
>>> varComplex = 1 + 20j
>>> print(varComplex)
(1+20j)
>>> varComplex.real
1.0
>>> varComplex.imag
20.0
>>> varComplex + 10
(11+20j)
>>> print(varComplex)
(1+20j)
```

Les types booléens

Les booléens sont le résultat d'opérations logiques et ont deux valeurs possibles : « **True** » ou « **False** ». Voici la liste des opérateurs qui s'appliquent aux booléens.

opérateur	signification	exemple
and or	et, ou logique	<code>x = True or False (résultat = True)</code>
not	négation logique	<code>x = not x</code>
< >	inférieur, supérieur	<code>x = 5 < 5</code>
<= >=	inférieur ou égal, supérieur ou égal	<code>x = 5 <= 5</code>
== !=	égal, différent	<code>x = 5 == 5</code>

```
>>> x = 4 < 5
>>> print (x) #affiche True
True
>>> print (not x) #affiche False
False
>>> x
True
```

Python accepte l'écriture résumée qui enchaîne des comparaisons : `3 < x and x < 7` est équivalent à `3 < x < 7`.

Les instructions conditionnelles

La plus simple de ces instructions conditionnelles est l'instruction if avec la syntaxe suivante :

```
if condition :  
    suite  
[elif condition :  
    suite , ...]  
[else :  
    suite]
```



```
>>> varInt = 150  
>>> if (varInt > 100):  
...     print("varInt dépasse la centaine")  
...
```

Frappez encore une fois « **Enter** » et le programme s'exécute, et vous obtenez :

```
varInt dépasse la centaine  
>>> varInt = 150  
>>> if (varInt > 100):  
...     print("varInt dépasse la centaine")  
... else:  
...     print("varInt ne dépasse pas cent")  
...  
varInt dépasse la centaine  
>>> varInt = 10  
>>> if (varInt > 100):  
...     print("varInt dépasse la centaine")  
... else:  
...     print("varInt ne dépasse pas cent")  
...  
varInt ne dépasse pas cent
```

On peut faire mieux encore en utilisant aussi l'instruction « **elif** ».



```
>>> varInt = 0  
>>> if varInt > 0 :  
...     print("varInt est positif")  
... elif varInt < 0 :  
...     print("varInt est négatif")  
... else:  
...     print("varInt est égal à zéro")  
...  
varInt est égal à zéro  
>>> varInt = -1  
>>> if varInt > 0 :  
...     print("varInt est positif")  
... elif varInt < 0 :  
...     print("varInt est négatif")  
... else:  
...     print("varInt est égal à zéro")
```

```
...
varInt est négatif
```

Les opérateurs de comparaison

La condition évaluée après l'instruction if peut contenir les opérateurs de comparaison suivants :

<code>x == y</code>	# x est égal à y
<code>x != y</code>	# x est différent de y
<code>x > y</code>	# x est plus grand que y
<code>x < y</code>	# x est plus petit que y
<code>x >= y</code>	# x est plus grand que, ou égal à y
<code>x <= y</code>	# x est plus petit que, ou égal à y
and or not	



```
>>> varInt = 4
>>> if (varInt % 2 == 0):
...     print("varInt est pair")
...     print("parce que le reste de sa division par 2 est nul")
... else:
...     print("varInt est impair")
...
varInt est pair
parce que le reste de sa division par 2 est nul
```

Passer, instruction pass

Dans certains cas, aucune instruction ne doit être exécutée même si un test est validé. En Python, le corps d'un test ne peut être vide, il faut utiliser l'instruction « **pass** ». Lorsque celle-ci est manquante, Python affiche un message d'erreur.

```
>>> signe = 0
>>> x = 0
>>> if x < 0 :
...     signe = -1
... elif x == 0:
...     pass #signe est déjà égal à 0
... else :
...     signe = 1
>>> print(signe)
0
```

Les blocs d'instructions

La construction que vous avez utilisée avec l'instruction « **if** » est votre premier exemple de blocs d'instructions. Sous Python, les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête.

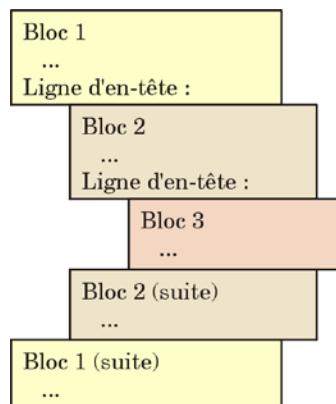
Ligne d'en-tête :

première instruction du bloc

...

dernière instruction du bloc

Ces instructions indentées constituent ce que nous appellerons désormais un bloc d'instructions. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête.



Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (**if**, **elif**, **else**, **while**, **def**, etc.) se terminant par un double point.

Les blocs sont délimités par l'indentation : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière. Le nombre d'espaces à utiliser pour l'indentation est quelconque.

Notez que le code du bloc le plus externe ne peut pas lui-même être écarté de la marge de gauche, il n'est imbriqué dans rien.

Les espaces et les commentaires sont normalement ignorés à part ceux qui servent à l'indentation, en début de ligne, les espaces placés à l'intérieur des instructions et des expressions sont presque toujours ignorés. Il en va de même pour les commentaires : ceux-ci commencent toujours par un caractère dièse « **#** » et s'étendent jusqu'à la fin de la ligne courante.

Attention



Vous pouvez aussi indenter à l'aide de tabulations, mais alors vous devrez faire très attention à ne pas utiliser tantôt des espaces, tantôt des tabulations pour indenter les lignes d'un même bloc. En effet, même si le résultat paraît identique à l'écran, espaces et tabulations sont des codes binaires distincts : Python considérera donc que ces lignes indentées différemment font partie de blocs différents. Il peut en résulter des erreurs difficiles à déboguer.

L'instruction while

L'instruction tant que commence par évaluer la validité de la condition et si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant le corps de la boucle. Voici la syntaxe de boucle tant que :

```
while expression :
    suite
[else :
    suite]
```



```
>>> varInt = 0
>>> while (varInt < 7):          # ( n'oubliez pas le double point ! )
...     varInt = varInt + 1      # ( n'oubliez pas l'indentation ! )
...     print("La valeur de varInt est : ",varInt)
...
La valeur de varInt est :  1
La valeur de varInt est :  2
La valeur de varInt est :  3
La valeur de varInt est :  4
La valeur de varInt est :  5
La valeur de varInt est :  6
La valeur de varInt est :  7
```

La variable évaluée dans la condition doit exister au préalable. Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté. Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment.



```
>>> i = 0
>>> while i < 2:
...     varInt = int(input("Entrez une valeur pour varInt :"))
...     while varInt < 8:
...         varInt = varInt +1
...         print(varInt , varInt**2 , varInt**3)
...     else:
...         print('Valeur de varInt >= 8')
...     i = i + 1
...
Entrez une valeur pour varInt :8
Valeur de varInt >= 8
Entrez une valeur pour varInt :4
5 25 125
6 36 216
7 49 343
8 64 512
Valeur de varInt >= 8
```

Le petit programme ci-dessous permet d'afficher les dix premiers termes d'une suite de Fibonacci. Il s'agit d'une suite de nombres dont chaque terme est égal à la somme des deux termes qui le précèdent.



```
>>> a, b, c = 1, 1, 1
>>> while c < 11 :
...     print(b, end =" ")
...     a, b, c = b, a+b, c+1
```

```
... else :  
...     print(" ")  
...  
1 2 3 5 8 13 21 34 55 89
```

La fonction **print()** ajoute en effet un caractère de saut à la ligne à toute valeur qu'on lui demande d'afficher. L'argument « **end = " "** » signifie que vous souhaitez remplacer le saut à la ligne par un simple espace. Si vous supprimez cet argument, les nombres seront affichés les uns en-dessous des autres.

 **Note**

La variable évaluée dans la condition doit exister au préalable.

Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté.

Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment (tout au moins tant que Python lui-même continue à fonctionner).

L'instruction break pour sortir d'une boucle

Dans une boucle « **while** » on peut interrompre le bouclage indépendamment de la condition de continuation en faisant appel à l'instruction **break**.

```
while <condition 1> :  
    --- instructions diverses ---  
    if <condition 2> :  
        break  
    --- instructions diverses ---
```

```
>>> a, b, c = 3, 2, 1  
>>> while c < 15:  
    print(c, ":", b)  
    a, b, c = b, a*b, c+1  
    if b > 9999999999 : break  
  
1 : 2  
2 : 6  
3 : 12  
4 : 72  
5 : 864  
6 : 62208  
7 : 53747712
```

Les listes

Les listes sont des collections ordonnées d'objets. Comme les chaînes de caractères, les listes font partie d'un type général que l'on appelle séquences sous Python. Comme les caractères dans une chaîne, les objets placés dans une liste sont rendus accessibles par l'intermédiaire d'un index.

Dans une liste on peut combiner des données de n'importe quel type, y compris des listes, des **dictionnaires** et des **tuples**. Pour accéder aux éléments d'une liste, on utilise les mêmes méthodes (index, découpage en tranches) que pour accéder aux caractères d'une chaîne.

```
>>> nombres = [5, 38, 10, 25]
>>> mots = ["jambon", "fromage", "confiture", "chocolat"]
>>> stuff = [5000, "Brigitte", 3.1416, ["Albert", "René", 1947]]
>>> print(nombres[2])
10
>>> print(nombres[1:3])
[38, 10]
>>> print(nombres[2:3])
[10]
>>> print(nombres[2:])
[10, 25]
>>> print(nombres[:2])
[5, 38]
>>> print(nombres[-1])
25
>>> print(nombres[-2])
10
```

Une tranche découpée dans une liste est toujours elle-même une liste, même s'il s'agit d'une tranche qui ne contient qu'un seul élément, comme dans notre troisième exemple, alors qu'un élément isolé peut contenir n'importe quel type de donnée.

Les listes sont modifiables

Contrairement aux chaînes de caractères, les listes sont des séquences modifiables. Cela nous permettra de construire plus tard des listes de grande taille, morceau par morceau, d'une manière dynamique.

```
>>> nombres[0] = 17
>>> nombres
[17, 38, 10, 25]
>>> stuff[3][1] = "Isabelle"
>>> stuff
[5000, 'Brigitte', 3.1416, ['Albert', 'Isabelle', 1947]]
>>> nombres[0] = nombres[2:]
>>> nombres
[[10, 25], 38, 10, 25]
```

Les listes sont des objets

Les listes sont des objets à part entière, et vous pouvez donc leur appliquer un certain nombre de méthodes particulièrement efficaces.

x in l	vrai si x est un des éléments de l
x not in l	réciproque de la ligne précédente
l + t	concaténation de l et t
l * n	concatène n copies de l les unes à la suite des autres
len(l)	nombre d'éléments de l
min(l)	plus petit élément de l, résultat difficile à prévoir lorsque les types des éléments sont différents
max(l)	plus grand élément de l
sum(l)	retourne la somme de tous les éléments
del l[i:j]	supprime les éléments d'indices entre i et j exclu. Cette instruction est équivalente à l[i:j] = [] .
list(x)	convertit x en une liste quand cela est possible
l.count(x)	Retourne le nombre d'occurrences de l'élément x.
l.index(x)	Retourne l'indice de la première occurrence de l'élément x dans la liste l.
l.append(x)	Ajoute l'élément x à la fin de la liste l. Si x est une liste, cette fonction ajoute la liste x en tant qu'élément, au final, la liste l ne contiendra qu'un élément de plus.
l.extend(k)	Ajoute tous les éléments de la liste k à la liste l. La liste l aura autant d'éléments supplémentaires qu'il y en a dans la liste k.
l.insert(i,x)	Insère l'élément x à la position i dans la liste l.
l.remove(x)	Supprime la première occurrence de l'élément x dans la liste l. S'il n'y a aucune occurrence de x, cette méthode déclenche une exception.
l.pop([i])	Retourne l'élément l[i] et le supprime de la liste. Le paramètre i est facultatif, s'il n'est pas précisé, c'est le dernier élément qui est retourné puis supprimé de la liste.
l.reverse()	Retourne la liste, le premier et dernier élément échangent leurs places, le second et l'avant dernier, et ainsi de suite.
l.sort([key=None, reverse=False])	Cette fonction trie la liste par ordre croissant. Le paramètre key est facultatif, il permet de préciser la fonction qui précise clé de comparaison qui doit être utilisée

```

>>> nombres = [17, 38, 10, 25, 72]
>>> nombres.sort()
>>> nombres
[10, 17, 25, 38, 72]
>>> nombres.append(12)
>>> nombres
[10, 17, 25, 38, 72, 12]
>>> nombres.reverse()
>>> nombres
[12, 72, 38, 25, 17, 10]
>>> nombres.index(17)
4
    
```

```
>>> nombres.remove(38)
>>> nombres
[12, 72, 25, 17, 10]
>>> del nombres[2]
>>> nombres
[12, 72, 17, 10]
>>> del nombres[1:3]
>>> nombres
[12, 10]
```

Techniques de slicing avancé pour modifier une liste

Il est possible d'utiliser à la place des « `del` » ou « `append` » pour ajouter ou supprimer des éléments dans une liste l'opérateur « `[]` ». L'utilisation de cet opérateur est un peu plus délicate que celle d'instructions ou de méthodes dédiées, mais elle permet davantage de souplesse.

Insertion d'un ou plusieurs éléments n'importe où dans une liste

```
>>> mots = ['jambon', 'fromage', 'confiture', 'chocolat']
>>> mots[2:2] =["miel"]
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat']
>>> mots[5:5] =[ 'saucisson', 'ketchup']
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat', 'saucisson',
'ketchup']
>>> mots[1:2]
['fromage']
```

Pour utiliser cette technique, vous devez prendre en compte les particularités suivantes :

- Si vous utilisez l'opérateur « `[]` » à la gauche du signe égale pour effectuer une insertion ou une suppression d'élément(s) dans une liste, vous devez obligatoirement y indiquer une tranche dans la liste cible, c'est-à-dire deux index réunis par le symbole « `:` », et non un élément isolé dans cette liste.
- L'élément que vous fournissez à la droite du signe égale doit lui-même être une liste. Si vous n'insérez qu'un seul élément, il vous faut donc le présenter entre crochets pour le transformer d'abord en une liste d'un seul élément.

La même démarche pour les suppressions et le remplacement d'éléments.

```
>>> mots[2:5] = [] # [] désigne une liste vide
>>> mots
['jambon', 'fromage', 'saucisson', 'ketchup']
>>> mots[1:3] = [ 'salade']
>>> mots
['jambon', 'salade', 'ketchup']
>>> mots[1:] = [ 'mayonnaise', 'poulet', 'tomate']
>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']
```

Création d'une liste de nombres

Si vous devez manipuler des séquences de nombres, vous pouvez les créer très aisément à l'aide de la fonction intégrée « `range` ». Elle renvoie une séquence

d'entiers que vous pouvez utiliser directement, ou convertir en une liste avec la fonction « `list` », ou convertir en tuple avec la fonction « `tuple` ».

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,13))
[5, 6, 7, 8, 9, 10, 11, 12]
>>> list(range(3,16,3))
[3, 6, 9, 12, 15]
>>> list(range(10, -10, -3))
[10, 7, 4, 1, -2, -5, -8]
```

La fonction « `range` » génère par défaut une séquence de nombres entiers de valeurs croissantes, et différent d'une unité. Si vous appelez « `range` » avec un seul argument, la liste contiendra un nombre de valeurs égal à l'argument fourni, mais en commençant à partir de zéro. Notez bien que l'argument fourni n'est jamais dans la liste générée. On peut aussi utiliser « `range` » avec deux, ou même trois arguments séparés par des virgules, que l'on pourrait intituler **FROM**, **TO** et **STEP**

Toute boucle « `for` » peut s'appliquer sur un objet muni d'un itérateur tels que les chaînes de caractères, tuples, les listes, les dictionnaires, les ensembles.

```
>>> prov = ['La','raison','du','plus','fort',\
           'est','toujours','la','meilleure']
>>> for mot in prov:
    print(mot, end = ' ')
```

La raison du plus fort est toujours la meilleure

Si vous voulez parcourir une gamme d'entiers, la fonction `range()` s'impose.

```
>>> for n in range(10, 18, 3):
    print(n, n**2, n**3)

10 100 1000
13 169 2197
16 256 4096
```

Il est très pratique de combiner les fonctions « `range` » et « `len` » pour obtenir automatiquement tous les indices d'une séquence (liste ou chaîne).

```
>>> fable = ['Maître','Corbeau','sur','un','arbre','perché']
>>> for index in range(len(fable)):
    print(index, fable[index])

0 Maître
1 Corbeau
2 sur
3 un
4 arbre
5 perché
```

Une conséquence importante du typage dynamique est que le type de la variable utilisée avec l'instruction « `for` » est redéfini continuellement au fur et à mesure du parcours : même si les éléments d'une liste sont de types différents, on peut parcourir cette liste à l'aide de `for` sans qu'il ne s'ensuive une erreur, car le type de la variable de parcours s'adapte automatiquement à celui de l'élément en cours de lecture.

```
>>> divers = [3, 17.25, [5, 'Jean'], 'Linux is not Windoze']
>>> for item in divers :
    print(item, type(item))
```

```

3 <class 'int'>
17.25 <class 'float'>
[5, 'Jean'] <class 'list'>
Linux is not Windoze <class 'str'>

```

Les opérations sur des listes

On peut appliquer aux listes les opérateurs « + » (concaténation) et « * » (multiplication). L'opérateur « * » est particulièrement utile pour créer une liste de n éléments identiques.

```

>>> fruits = ['orange','citron']
>>> legumes = ['poireau','oignon','tomate']
>>> fruits + legumes
['orange', 'citron', 'poireau', 'oignon', 'tomate']
>>> fruits * 3
['orange', 'citron', 'orange', 'citron', 'orange', 'citron']
>>> sept_zeros = [0]*7
>>> sept_zeros
[0, 0, 0, 0, 0, 0, 0]

```

Supposons par exemple que vous voulez créer une liste B qui contienne le même nombre d'éléments qu'une autre liste A.

```

>>> deuxieme = [10]*len(sept_zeros)
>>> deuxieme
[10, 10, 10, 10, 10, 10, 10]

```

Le test d'appartenance

Vous pouvez aisément déterminer si un élément fait partie d'une liste à l'aide de l'instruction « `in` » (cette instruction puissante peut être utilisée avec toutes les séquences).

```

>>> v = 'tomate'
>>> if v in legumes:
    print('OK')

OK

```

La copie d'une liste

Attention une simple affectation ne créez pas une véritable copie d'une liste que vous souhaitez recopier dans une nouvelle variable. À la suite de cette instruction, il n'existe toujours qu'une seule liste dans la mémoire de l'ordinateur. Ce que vous avez créé est seulement une nouvelle référence vers cette liste.

```

>>> fable = ['Je','plie','mais','ne','romps','point']
>>> phrase = fable
>>> fable[4] = 'casse'
>>> phrase
['Je', 'plie', 'mais', 'ne', 'casse', 'point']

```

Si la variable phrase contenait une véritable copie de la liste, cette copie serait indépendante de l'original et ne devrait donc pas pouvoir être modifiée par une instruction telle que celle de la troisième ligne, qui s'applique à la variable fable.

En fait, les noms fable et phrase désignent tous deux un seul et même objet en mémoire. Ainsi l'objet phrase est une référence de l'objet fable.

```
>>> phrase2 = phrase[0:len(phrase)]
>>> phrase[4] ='romps'
>>> phrase
['Je', 'plie', 'mais', 'ne', 'romps', 'point']
>>> phrase2
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Python vous autorise à « étendre » une longue instruction sur plusieurs lignes, si vous continuez à encoder quelque chose qui est délimité par une paire de parenthèses, de crochets ou d'accolades. Vous pouvez traiter ainsi des expressions parenthèses, ou encore la définition de longues listes, de grands tuples ou de grands. Le niveau d'indentation n'a pas d'importance : l'interpréteur détecte la fin de l'instruction là où la paire syntaxique est refermée. Cette fonctionnalité vous permet d'améliorer la lisibilité de vos programmes.

```
>>> couleurs = ['noir', 'brun', 'rouge',
   'orange', 'jaune', 'vert',
   'bleu', 'violet', 'gris', 'blanc']
```

Les nombres aléatoires – histogrammes

Le module « **random** », Python propose une série de fonctions permettant de générer des nombres aléatoires qui suivent différentes distributions mathématiques.

Les fonctions les plus couramment utilisées sont :

- **choice(sequence)** : renvoie un élément au hasard de la séquence fournie.
- **randint(a, b)** : renvoie un nombre entier compris entre a et b.
- **randrange(a,b,c)** : renvoie un nombre entier tiré au hasard d'une série limitée d'entiers entre a et b, séparés les uns des autres par un certain intervalle, défini par c.
- **random()** : renvoie un réel compris entre 0.0 et 1.0.
- **sample(sequence, k)** : renvoie k éléments uniques de la séquence.
- **seed([salt])** : initialise le générateur aléatoire.
- **shuffle(sequence[, random])** : mélange l'ordre des éléments de la séquence (dans l'objet lui-même). Si random est fourni, c'est un callable qui renvoie un réel entre 0.0 et 1.0. « **random** » est pris par défaut.
- **uniform(a, b)** : renvoie un réel compris entre a et b.

```
>>> from random import *
>>> def list_aleat(n):
    s = [0]*n
    for i in range(n):
        s[i] =random()
    return s

>>> list_aleat(3)
[0.6735559443377152, 0.09987607185190805, 0.6063188589461471]
>>> list_aleat(3)
[0.5182167354579681, 0.21973841027737828, 0.36650995653460494]

>>> for i in range(15):
```

```
print(randrange(3, 13, 3), end = ' ')
6 12 3 6 9 6 6 3 6 6 12 3 12 6 12
```

Vous pouvez constater que nous avons pris le parti de construire d'abord une liste de zéros de taille n, et ensuite de remplacer les zéros par des nombres aléatoires.

```
>>> import random
>>> good_work = ['Excellent travail!',
   'Très bonne analyse',
   'Les résultats sont là !']
>>> bad_work = ["J'ai gratté la copie pour mettre des points",
   'Vous filez un mauvais coton',
   'Que se passe-t-il ?']
>>> ok_work = ['Bonne première partie mais soignez la présentation',
   'Petites erreurs, dommage !',
   'Des progrès']

>>> def auto_corrector(student):
    note = random.randint(1, 20)
    if note < 8:
        appreciation = random.choice(bad_work)
    elif note < 14:
        appreciation = random.choice(ok_work)
    else:
        appreciation = random.choice(good_work)
    return '%s: %s, %s' %(student,
                          note, appreciation)

>>> students = ['Bernard', 'Robert', 'René', 'Gaston',
   'Églantine', 'Aimé', 'Robertine']

>>> for student in students :
    print(auto_corrector(student))

Bernard: 16, Très bonne analyse
Robert: 2, Que se passe-t-il ?
René: 11, Petites erreurs, dommage !
Gaston: 12, Petites erreurs, dommage !
Églantine: 3, J'ai gratté la copie pour mettre des points
Aimé: 18, Excellent travail!
Robertine: 2, J'ai gratté la copie pour mettre des points
```

Atelier 5 - Exercice 2 - Exercice 3

Les tuples

Python propose un type de données appelé « **tuple** », qui est assez semblable à une liste mais qui, comme les chaînes, n'est pas modifiable.

Du point de vue de la syntaxe, un « **tuple** » est une collection d'éléments séparés par des virgules comprise entre parenthèses. Ce terme n'est pas un mot anglais ordinaire : il s'agit d'un néologisme informatique.

```
>>> tup = (5000, "Brigitte",
           3.1416, ["Albert", "René", 1947])
>>> print(tup)
(5000, 'Brigitte', 3.1416, ['Albert', 'René', 1947])
>>> tup2 = 5000, "Brigitte", \
           3.1416, ["Albert", "René", 1947]
>>> tup2
(5000, 'Brigitte', 3.1416, ['Albert', 'René', 1947])
>>> tup2[0]
5000
>>> tup2[0] = 1
Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    tup2[0] = 1
TypeError: 'tuple' object does not support item assignment
```

Bien que cela ne soit pas nécessaire, il est vivement conseillé de mettre le « **tuple** » en évidence en l'enfermant dans une paire de parenthèses, comme la fonction « **print** » de Python le fait elle-même. Il s'agit simplement d'améliorer la lisibilité du code, mais vous savez que c'est important.

x in s	vrai si x est un des éléments de s
x not in s	réciproque de la ligne précédente
s + t	concaténation de s et t
s * n	concatène n copies de s les unes à la suite des autres
s[i]	retourne le i ^{ème} élément de s
s[i:j]	retourne un « tuple » contenant une copie des éléments de s d'indices i à j exclu
s[i:j:k]	retourne un « tuple » contenant une copie des éléments de s dont les indices sont compris entre i et j exclu, ces indices sont espacés de k : i, i + k, i + 2k, i + 3k, ...
len(s)	nombre d'éléments de s
min(s)	plus petit élément de s, résultat difficile à prévoir lorsque les types des éléments sont différents
max(s)	plus grand élément de s
sum(s)	retourne la somme de tous les éléments

Les « **tuples** » composés d'un seul élément ont une écriture un peu particulière puisqu'il est nécessaire d'ajouter une virgule après l'élément, sans quoi l'analyseur syntaxique de Python ne le considérera pas comme un « **tuples** » mais comme l'élément lui-même, et supprimera les parenthèses qu'il analyserait comme superflues.

```
>>> tuple()
```

```
( )
>>> tuple('a')
('a',)
>>> color_and_note = ('rouge', 12, 'vert', 14, 'bleu', 9)
>>> colors = color_and_note[::2]
>>> print(colors)
('rouge', 'vert', 'bleu')
>>> notes = color_and_note[1::2]
>>> print(notes)
(12, 14, 9)
>>> color_and_note = color_and_note + ('violet',)
>>> print(color_and_note)
('rouge', 12, 'vert', 14, 'bleu', 9, 'violet')
>>> ('violet')
'violet'
>>> ('violet',)
('violet',)
```

Les « **tuples** » sont préférables aux listes partout où l'on veut être certain que les données transmises ne soient pas modifiées par erreur au sein d'un programme. En outre, les « **tuples** » occupent moins de place en mémoire, et peuvent être traités plus rapidement par l'interpréteur.

Les dictionnaires

Les types de données composites que nous avons abordés jusqu'à présent « **chaînes** », « **listes** » et « **tuples** » étaient tous des séquences, c'est-à-dire des suites ordonnées d'éléments. Dans une séquence, il est facile d'accéder à un élément quelconque à l'aide d'un index (un nombre entier), mais à la condition expresse de connaître son emplacement.

Les « **dictionnaires** » que nous découvrons ici constituent un autre type composite. Ils ressemblent aux listes dans une certaine mesure, ils sont modifiables comme elles, mais ce ne sont pas des séquences. Les éléments que nous allons y enregistrer ne seront pas disposés dans un ordre immuable. En revanche, nous pourrons accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appellera une clé, laquelle pourra être alphabétique, numérique, ou même d'un type composite sous certaines conditions.

Comme dans une liste, les éléments mémorisés dans un « **dictionnaire** » peuvent être de n'importe quel type. Ce peuvent être des valeurs numériques, des « **chaînes** », des « **listes** », des « **tuples** », des « **dictionnaires** », et même aussi des « **fonctions** », des « **classes** » ou des « **instances** ».

La création d'un dictionnaire

Puisque le type « **dictionnaire** » est un type modifiable, nous pouvons commencer par créer un « **dictionnaire** » vide, puis le remplir petit à petit. Du point de vue de la syntaxe, on reconnaît un « **dictionnaire** » au fait que ses éléments sont enfermés dans une paire d'accolades « { } ».

```
>>> dico = {}
>>> dico['computer'] = 'ordinateur'
>>> dico['mouse'] = 'souris'
>>> dico['keyboard'] = 'clavier'
>>> print(dico)
{'computer': 'ordinateur', 'mouse': 'souris', 'keyboard': 'clavier'}
>>> print(dico['mouse'])
souris
```

Un « **dictionnaire** » apparaît dans la syntaxe Python sous la forme d'une série d'éléments séparés par des virgules, le tout étant enfermé entre deux accolades. Chacun de ces éléments est lui-même constitué d'une paire d'objets : un index et une valeur, séparés par un double point. Dans un « **dictionnaire** », les index s'appellent des clés, et les éléments peuvent donc s'appeler des paires clé-valeur.

Remarquez aussi que contrairement à ce qui se passe avec les listes, il n'est pas nécessaire de faire appel à une méthode particulière (telle que « **append** ») pour ajouter de nouveaux éléments à un dictionnaire : il suffit de créer une nouvelle paire clé-valeur.

Les opérations sur les dictionnaires

Tout comme les listes, les objets de type dictionnaire proposent un certain nombre de méthodes.

Nom	Description
<code>clear()</code>	Supprime tous les éléments du dictionnaire.
<code>copy()</code>	Renvoie une copie par références du dictionnaire. Lire la remarque sur les copies un peu plus bas.
<code>items()</code>	Renvoie sous la forme d'une liste de « tuples », es couples (clé, valeur) du dictionnaire. Les objets représentant les valeurs sont es copies complètes et non des références.
<code>keys()</code>	Renvoie sous la forme d'une liste l'ensemble des clés du dictionnaire. L'ordre de renvoi des éléments n'a aucune signification ni constance et peut varier à chaque modification du dictionnaire.
<code>values()</code>	Renvoie sous forme de liste les valeurs du dictionnaire. L'ordre de renvoi n'a ici non plus aucune signification mais sera le même que pour « keys » si la liste n'est pas modifiée entre-temps, ce qui permet de faire des manipulations avec les deux listes.
<code>get(cle,default)</code>	Renvoie la valeur identifiée par la clé. Si la clé n'existe pas, renvoie la valeur default fournie. Si aucune valeur n'est fournie, renvoie « None ».
<code>pop(cle,default)</code>	Renvoie la valeur identifiée par la clé et retire l'élément du dictionnaire. Si la clé n'existe pas, pop se contente de renvoyer la valeur default. Si le paramètre default n'est pas fourni, une erreur est levée.
<code>popitem()</code>	Renvoie le premier couple (clé, valeur) du dictionnaire et le retire. Si le dictionnaire est vide, une erreur est renvoyée. L'ordre de retrait des éléments correspond à l'ordre des clés retournées par « keys » si la liste n'est pas modifiée entre-temps.
<code>update(dic,**dic)</code>	Update permet de mettre à jour le dictionnaire avec les éléments du dictionnaire dic. Pour les clés existantes dans la liste, les valeurs sont mises à jour, sinon créées. Le deuxième argument est aussi utilisé pour mettre à jour les valeurs.
<code>setdefault(cle,default)</code>	Fonctionne comme « get » mais si clé n'existe pas et default est fourni, le couple (cle, default) est ajouté à la liste.
<code>fromkeys(seq,default)</code>	Génère un nouveau dictionnaire et y ajoute les clés fournies dans la séquence seq. La valeur associée à ces clés est default si le paramètre est fourni, « None » le cas échéant.

Voici quelque exemples de ces syntaxes.

```
>>> invent ={"oranges":274, "poires":137, "bananes":312}
>>> for clef in invent:
        print(clef)

oranges
poires
bananes
>>> dicol = {'a':1,'b':2}
>>> dicol.clear()
>>> dicol
```

```

    []
>>> dico = {'1': 'r', '2': [1,2]}
>>> dico2 = dico.copy()
>>> dico2
{'1': 'r', '2': [1, 2]}
>>> dico['2'].append('E')
>>> dico2['2']
[1, 2, 'E']
>>> dico = {'a': 1, 'b': 2}
>>> 'a' in dico
True
>>> 'c' not in dico
True
>>> a = {'a': 1, 'b': 1}
>>> a.items()
dict_items([(‘a’, 1), (‘b’, 1)])
>>> a = {(1, 3): 3, ‘Q’: 4}
>>> a.keys()
dict_keys([(1, 3), ‘Q’])
>>> a = {(1, 3): 3, ‘Q’: 4}
>>> a.values()
dict_values([3, 4])
>>> l = {1: ‘a’, 2: ‘b’, 3: ‘c’}
>>> for i,j,k in l.items(),l.keys(),l.values() :
    print(i,j,k)

(1, ‘a’) (2, ‘b’) (3, ‘c’)
1 2 3
a b c
>>> l.get(1)
‘a’
>>> l.get(13)

>>> l.get(13, 7)
7
>>> l
{1: ‘a’, 2: ‘b’, 3: ‘c’}
>>> l.pop(1)
‘a’
>>> l
{2: ‘b’, 3: ‘c’}
>>> l.pop(13, 6)
6
>>> l
{2: ‘b’, 3: ‘c’}
>>> l = {1: ‘a’, 2: ‘b’, 3: ‘c’}
>>> l.popitem()
(3, ‘c’)
>>> l.popitem()
(2, ‘b’)
>>> l.popitem()
(1, ‘a’)
>>> l
{}

```

```

>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> l2 = {3: 'ccc', 4: 'd'}
>>> l.update(l2)
>>> l
{1: 'a', 2: 'b', 3: 'ccc', 4: 'd'}
>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> l.setdefault(4, 'd')
'd'
>>> l
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> l = {}

>>> l.fromkeys([1, 2, 3], 0)
{1: 0, 2: 0, 3: 0}

```

Les clés peuvent être de n'importe quel type de données non modifiables : des entiers, des réels, des chaînes de caractères, et même des tuples.

```

>>> arb = {}
>>> arb[(1,2)] = 'Peuplier'
>>> arb[(3,4)] = 'Platane'
>>> arb[6,5] = 'Palmier'
>>> arb[5,1] = 'Cycas'
>>> arb[7,3] = 'Sapin'
>>> print(arb)
{(1, 2): 'Peuplier', (3, 4): 'Platane', (6, 5): 'Palmier', (5, 1): 'Cycas', (7, 3): 'Sapin'}
>>> print(arb[(6,5)])
Palmier
>>> print(arb[1,2])
Peuplier
>>> print(arb[2,1])
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    print(arb[2,1])
TypeError: tuple indices must be integers or slices, not tuple
>>> arb.get((2,1), 'néant')
'néant'
>>> print(arb[1:3])
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    print(arb[1:3])
TypeError: unhashable type: 'slice'

```

Les chaînes de caractères

Le terme "chaîne de caractères" ou string en anglais signifie une suite finie de caractères, autrement dit, du texte. Ce texte est compris entre deux guillemets ou deux apostrophes, ces deux symboles sont interchangeables.

```
>>> t = "string = texte"
>>> print(type(t), t)
<class 'str'> string = texte
>>> t = 'string = texte, initialisation avec apostrophes'
>>> print(type(t), t)
<class 'str'> string = texte, initialisation avec apostrophes
>>> t = "morceau 1" \
    "morceau 2"
>>> #second morceau ajouté au premier par l'ajout du symbole \,
>>> #il ne doit rien y avoir après le symbole \,
>>> #pas d'espace ni de commentaire
>>> print(t)
morceau 1morceau 2
>>> t = """première ligne
seconde ligne"""
>>> # chaîne de caractères qui s'étend sur deux lignes
>>> print(t)
première ligne
seconde ligne
>>> t = '''première ligne
    seconde ligne'''
>>> t
'première ligne\n                seconde ligne'
>>> print(t)
première ligne
    seconde ligne
```

Python offre la possibilité de couper le texte en deux chaînes de caractères recollées à l'aide du symbole \ à condition que ce symbole soit le dernier de la ligne sur laquelle il apparaît. De même, lorsque le texte contient plusieurs lignes, il suffit de les encadrer entre deux symboles « """ » ou « ''' » pour que l'interpréteur Python considère l'ensemble comme une chaîne de caractères et non comme une série d'instructions.

opérateur	signification
\"	guillemet
\'	apostrophe
\n	passage à la ligne
\f	saut de ligne
\\"	insertion du symbole \
\%	pourcentage, ce symbole est aussi un caractère spécial
\t	Tabulation
\v	tabulation verticale
\r	retour à la ligne, peu usité, il a surtout son importance lorsqu'on passe d'un système Windows à Linux car Windows l'ajoute automatiquement à tous ses fichiers textes

Il peut être fastidieux d'avoir à doubler tous les symboles \ d'un nom de fichier. Il est plus simple dans ce cas de préfixer la chaîne de caractères par « `r` » de façon à éviter que l'utilisation du symbole \ ne désigne un caractère spécial.

```
>>> s1 = "C:\\\\Users\\\\exemple.txt"
>>> s2 =r"C:\\\\Users\\\\exemple.txt"
>>> s1 == s2
True
```

Triple quotes

Pour insérer plus aisément des caractères spéciaux ou « exotiques » dans une chaîne, sans faire usage de l'antislash, ou pour faire accepter l'antislash lui-même dans la chaîne, on peut encore délimiter la chaîne à l'aide de triples guillemets ou de triples apostrophes :

```
>>> a1 = """
... Exemple de texte préformaté, c'est-à-dire
... dont les indentations et les
... caractères spéciaux \ ' " sont
... conservés sans
... autre forme de procès."""
>>> print(a1)
... Exemple de texte préformaté, c'est-à-dire
... dont les indentations et les
... caractères spéciaux \ ' " sont
... conservés sans
... autre forme de procès.
```

Manipulation d'une chaîne

Une chaîne de caractères est semblable à un tableau et certains opérateurs qui s'appliquent aux tableaux s'appliquent également aux chaînes de caractères. La fonction « `str` » permet de convertir un nombre, un tableau, un objet en chaîne de caractères afin de pouvoir l'afficher. La fonction « `len` » retourne la longueur de la chaîne de caractères.

```
>>> x = 5.567
>>> s = str(x)
>>> print(type(s),s)
<class 'str'> 5.567
>>> print(len(s))
5
```

Opérations applicables aux chaînes de caractères.

opérateur	signification	exemple
<code>+</code>	concaténation de chaînes de caractères	<code>t="abc "+"def"</code>
<code>+=</code>	concaténation puis affectation	<code>t+="abc"</code>
<code>in, not in</code>	une chaîne en contient-elle une autre ?	<code>"ed" in "med"</code>
<code>*</code>	répétition d'une chaîne de caractères	<code>t="abc" * 4</code>
<code>[n]</code>	obtention du n ^{ième} caractère, le premier caractère a pour indice 0	<code>t="abc" print(t[0])</code>
<code>[i : j]</code>	obtention des caractères compris entre les indices i et j – 1 inclus, le premier caractère a pour indice 0	<code>t="abc" print t[0:2]</code>

Il existe d'autres fonctions qui permettent de manipuler les chaînes de caractères.

opérateur	signification
<code>count(sub[,start[,end]])</code>	Retourne le nombre d'occurrences de la chaîne de caractères « <code>sub</code> », les paramètres par défaut « <code>start</code> » et « <code>end</code> » permettent de réduire la recherche entre les caractères d'indice « <code>start</code> » et « <code>end</code> » exclu.
<code>find(sub[,start[,end]])</code>	Recherche une chaîne de caractères « <code>sub</code> », les paramètres par défaut ont la même signification que ceux de la fonction count.
<code>index(car)</code>	Retrouve l'indice de la première occurrence du caractère « <code>car</code> » dans la chaîne
<code>isalpha()</code>	Retourne True si tous les caractères sont des lettres, False sinon.
<code>isdigit()</code>	Retourne True si tous les caractères sont des chiffres, False sinon.
<code>replace(old,new[,count])</code>	Retourne une copie de la chaîne de caractères en remplaçant toutes les occurrences de la chaîne « <code>old</code> » par « <code>new</code> ». Si le paramètre optionnel « <code>count</code> » est renseigné, alors seules les premières occurrences seront remplacées.
<code>split(sep=None,maxsplit=1)</code>	Découpe la chaîne de caractères en se servant de la chaîne « <code>sep</code> » comme délimiteur. Si le paramètre « <code>maxsplit</code> » est renseigné, au plus « <code>maxsplit</code> » coupures seront effectuées.
<code>strip([s])</code>	Supprime les espaces au début et en fin de chaîne. Si le paramètre « <code>s</code> » est renseigné, tous les caractères qui font partie de « <code>s</code> » au début et en fin de chaîne sont supprimés.
<code>upper()</code>	Remplace les minuscules par des majuscules

lower()	Remplace les majuscules par des minuscules.
join(words)	Fait la somme d'un tableau de chaînes de caractères (une liste ou un Tuple). La chaîne de caractères sert de séparateur qui doit être ajouté entre chaque élément du tableau words.
title	Convertit en majuscule l'initiale de chaque mot (suivant l'usage des titres anglais)
capitalize	Convertit en majuscule seulement la première lettre de la chaîne
swapcase	Convertit toutes les majuscules en minuscules, et vice-versa

Python considère qu'une chaîne de caractères est un objet de la catégorie des séquences, lesquelles sont des collections ordonnées d'éléments. Cela signifie simplement que les caractères d'une chaîne sont toujours disposés dans un certain ordre. Par conséquent, chaque caractère de la chaîne peut être désigné par sa place dans la séquence, à l'aide d'un index.

```
>>> phrase1 = 'les oeufs durs.'
>>> phrase2 = '"Oui", répondit-il,'
>>> phrase3 = "j'aime bien"
>>> print(phrase2, phrase3, phrase1)
"Oui", répondit-il, j'aime bien les oeufs durs.
>>> phrase = phrase2 + phrase3 + ' ' + phrase1
>>> print(phrase)
"Oui", répondit-il, j'aime bien les oeufs durs.
>>> phrase.count('o')
2
>>> phrase.find('oeuf')
35
>>> phrase[35:40]
'oeufs'
>>> phrase.replace('"', '')
"Oui, répondit-il, j'aime bien les oeufs durs."
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> phrase.split(',', maxsplit=1)
[['Oui', " répondit-il, j'aime bien les oeufs durs."]]
>>> phrase.split(',')
[['Oui', ' répondit-il', ' j'aime bien les oeufs durs.']]
>>> s = '#..... Section 3.2.1 Issue #32 .....'
>>> s.strip('.#! ')
'Section 3.2.1 Issue #32'
```

Attention les chaînes constituent un type de données non-modifiables il est impossible de changer les éléments individuels d'une chaîne.

```
>>> s="abcd"
>>> s[1]='c'
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    s[1]='c'
TypeError: 'str' object does not support item assignment
```

À toutes fins utiles, rappelons également ici que l'on peut aussi appliquer aux chaînes un certain nombre de fonctions intégrées dans le langage :

len(ch) renvoie la longueur de la chaîne ch, ou en d'autres termes, son nombre de caractères.

float(ch) convertit la chaîne ch en un nombre réel « **float** » (bien entendu, cela ne pourra fonctionner que si la chaîne représente bien un nombre, réel ou entier)

int(ch) convertit la chaîne ch en un nombre entier (avec des restrictions similaires)

str(obj) convertit (ou représente) l'objet **obj** en une chaîne de caractères. **obj** peut être une donnée d'à peu près n'importe quel type :

Formatage d'une chaîne

Python offre une manière plus concise de former une chaîne de caractères à l'aide de plusieurs types d'informations en évitant la conversion explicite de ces informations et leur concaténation. Il est particulièrement intéressant pour les nombres réels qu'il est possible d'écrire en imposant un nombre de décimales fixe.

Le format est le suivant :

```
".... %c1 .... %c2 " % (v1,v2)
```

« **c1** » est un code du format dans lequel la variable « **v1** » devra être transcrise. Il en est de même pour le code « **c2** » associé à la variable « **v2** ». Les codes insérés dans la chaîne de caractères seront remplacés par les variables citées entre parenthèses après le symbole « **%** » suivant la fin de la chaîne de caractères. Il doit y avoir autant de codes que de variables, qui peuvent aussi être des constantes.

code	signification
d	entier relatif
e	nombre réel au format exponentiel
f	nombre réel au format décimal
g	nombre réel, format décimal ou exponentiel si la puissance est trop grande ou trop petite
s	chaîne de caractères

La seconde affectation de la variable « **res** » propose une solution équivalente à la première en utilisant l'opérateur de concaténation « **+** ».

```
>>> x,d,s = 5.5,7,"caractères"
>>> res = "un nombre réel %f et un entier %d, une chaîne de %s, \n\"\
    "un réel d'abord converti en chaîne de caractères %s" \
    % (x,d,s, str(x+4))
>>> print(res)
un nombre réel 5.500000 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
>>> res = "un nombre réel " + str (x) + \
    " et un entier " + str (d) + \
    ", une chaîne de " + s + \
    ",\n un réel d'abord converti en chaîne de caractères " \
    + str(x+4)
>>> print(res)
un nombre réel 5.5 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
```

Les deux solutions sont équivalentes, tout dépend des préférences de celui qui écrit le programme. La première option permet néanmoins un formatage plus précis des nombres réels en imposant par exemple un nombre défini de décimal.

```
>>> x = 0.123456789
>>> print (x)
0.123456789
>>> print ("%1.2f"%x)
0.12
>>> print ("%06.2f"%x)
000.12
```

format

La fonction se révèle particulièrement utile dans tous les cas où vous devez construire une chaîne de caractères complexe à partir d'un certain nombre de morceaux, tels que les valeurs de variables diverses. Vous pouvez préparer une chaîne « **patron** » contenant l'essentiel du texte invariable, avec des balises particulières aux endroits où vous souhaitez qu'apparaissent des contenus variables. Vous appliquerez ensuite à cette chaîne la méthode « **format** », à laquelle vous fournirez comme arguments les divers objets à convertir en caractères et à insérer en remplacement des balises. Les balises à utiliser sont constituées d'accolades, contenant ou non des indications de formatage.

```
>>> coul ="verte"
>>> temp =1.347 + 15.9
>>> ch ="La couleur est {} et la température vaut {} °C"
>>> print(ch.format(coul, temp))
La couleur est verte et la température vaut 17.247 °C
```

Si les balises sont vides, la méthode « **format** » devra recevoir autant d'arguments qu'il y aura de balises dans la chaîne. Python appliquera alors la fonction « **str** » à chacun de ces arguments, et les insérera ensuite dans la chaîne à la place des balises, dans le même ordre. Les arguments peuvent être n'importe quel objet ou expression :

```
>>> pi =3.14159265358979323846264338327950288419716939937510582
>>> r =4.7
>>> ch ="L'aire d'un disque de rayon {} est égale à {:.2f}."
>>> print(ch.format(r, pi * r**2))
L'aire d'un disque de rayon 4.7 est égale à 69.3978.
```

Les balises peuvent contenir des numéros d'ordre pour désigner précisément lesquels des arguments transmis à « **format** » devront les remplacer. Cette technique est particulièrement précieuse si le même argument doit remplacer plusieurs balises :

```
>>> phrase ="Le{0} chien{0} aboie{1} et le{0} chat{0} miaule{1}."
>>> print(phrase.format("", ""))
Le chien aboie et le chat miaule.
>>> print(phrase.format("s", "nt"))
Les chiens aboient et les chats miaulent.
```

Le formatage permet d'afficher très facilement divers résultats numériques en notation binaire, octale ou hexadécimale :

```
>>> n =789
>>> txt ="Le nombre {0:d} (décimal) \n"\
           "vaut {0:x} en hexadécimal \n"\
           "et {0:b} en binaire."
>>> print(txt.format(n))
Le nombre 789 (décimal)
vaut 315 en hexadécimal
et 1100010101 en binaire.
```

Le type bytes et la page de code

Comme tous les langages de programmation Python est conçu pour le monde anglophone et l'utilisation des accents ne va pas de soi. Si vous avez rédigé votre script avec un éditeur récent (tels ceux que nous avons déjà indiqués), le script décrit ci-dessus devrait s'exécuter sans problème avec la version actuelle de Python 3. Si votre logiciel est ancien ou mal configuré, il se peut que vous obteniez un message d'erreur similaire à celui-ci :

File "fibo2.py", line 2

```
SyntaxError: Non-UTF-8 code starting with '\xe0' in file fibo2.py on line 2, but no
encoding declared; see http://python.org/dev/peps/pep-0263/ for details
```

Avec les versions de Python antérieures à la version 3.0, comme dans beaucoup d'autres langages, il fallait fréquemment convertir les chaînes de caractères d'une norme d'encodage à une autre. Du fait des conventions et des mécanismes adoptés désormais, vous ne devrez plus beaucoup vous en préoccuper pour vos propres programmes traitant des données récentes.

Afin que Python puisse les interpréter correctement, il vous est conseillé d'y inclure toujours l'un des pseudo-commentaires suivants (obligatoirement à la 1e ou à la 2e ligne).

```
# -*- coding:latin-1 -*-
ou
```

```
# -*- coding:utf-8 -*-
```

Ainsi l'interpréteur Python sait décoder correctement les chaînes de caractères littérales que vous avez utilisées dans le script. Notez que vous pouvez omettre ce pseudo commentaire si vous êtes certain que vos scripts sont encodés en « **utf-8** », car c'est cet encodage qui est désormais la norme par défaut pour les scripts Python.

```
>>> import locale
>>> import sys
>>> print (sys.getdefaultencoding ())
utf-8
>>> locale.getdefaultlocale()
('fr_FR', 'cp1252')
>>> import encodings
>>> print (''.join('- ' + e + '\n' \
    for e in sorted(set(encodings.aliases.aliases.values()))))
- ascii
- base64_codec
- big5
- big5hkscs
- bz2_codec
- cp037
...
- cp1254
...
- iso8859_16
...
- latin_1
...
- utf_16
```

```
- utf_16_be
- utf_16_le
- utf_32
- utf_32_be
- utf_32_le
- utf_7
- utf_8
- uu_codec
- zlib_codec
```

Le type « **bytes** » représente un tableau d'octets. Il fonctionne quasiment pareil que le type « **str** ». Les opérations qu'on peut faire dessus sont quasiment identiques. Les deux méthodes suivantes de la classe « **str** » permettent de convertir une chaîne de caractères en « **bytes** » et l'invers.

<code>encode(enc)</code>	Cette fonction permet de passer d'un jeu de caractères, celui de la variable, au jeu de caractères précisé par enc à moins que ce ne soit le jeu de caractères par défaut. Cette fonction retourne un type « bytes ».
<code>decode(enc)</code>	Cette fonction est la fonction inverse de la fonction encode. Avec les mêmes paramètres, elle effectue la transformation inverse.

Le type « **bytes** » est très utilisé quand il s'agit de convertir une chaîne de caractères d'une page de code à une autre.

```
>>> b = b"345"
>>> print(b, type(b))
b'345' <class 'bytes'>
>>> b = bytes.fromhex('2Ef0 F1f2 ')
>>> print(b, type(b))
b'.\xf0\xf1\xf2' <class 'bytes'>
>>> b = "abc".encode("utf-8")
>>> s = b.decode("ascii")
>>> print(b, s)
b'abc' abc
>>> print(type(b), type(s))
<class 'bytes'> <class 'str'>
>>> varStr = '圖形碼常用字次常用字'
>>> varBytes = varStr.encode()
>>> print(type(varStr), type(varBytes))
<class 'str'> <class 'bytes'>
>>> print(varStr,'\\n',varBytes)
圖形碼常用字次常用字

b'\xe5\x9c\x96\xe5\xbd\xad\x97\xe6\xac\x91\xe5\xb8\x94\x94\xad\x97'
```

L'instruction for

Le parcours d'une séquence est une opération très fréquente en programmation. Pour en faciliter l'écriture, Python vous propose une structure de boucle plus appropriée que la boucle « **while** », basée sur le couple d'instructions « **for...in...** ».

```
>>> nom = "Cléopâtre"
>>> for car in nom : print(car + ' *', end = ' ')
C * l * é * o * p * â * t * r * e *
```

L'instruction for permet donc d'écrire des boucles, dans lesquelles l'itération traite successivement tous les éléments d'une séquence donnée.

```
>>> liste = ['chien', 'chat', 'crocodile', 'éléphant']
>>> for animal in liste:
    print('longueur de la chaîne', animal, '=', len(animal))

longueur de la chaîne chien = 5
longueur de la chaîne chat = 4
longueur de la chaîne crocodile = 9
longueur de la chaîne éléphant = 8
```

Lors de l'affichage d'une liste, les éléments n'apparaissent pas triés, le langage Python propose néanmoins la fonction « **sorted** ».

```
>>> liste = [3, 2, 1, 6, 4, 9, 7, 8, 5]
>>> for x in liste: print (x)

3
2
1
6
4
9
7
8
5
>>> for x in sorted(liste): print (x)

1
2
3
4
5
6
7
8
9
```

La boucle la plus répandue est celle qui parcourt des indices entiers compris entre 0 et $n - 1$. On utilise pour cela la boucle for et la fonction « **range** ».

```
range (début, fin [,marche])
```

Retourne une liste incluant tous les entiers compris entre « **début** » et « **fin** » exclu. Si le paramètre facultatif **marche** est renseigné, la liste contient tous les entiers compris « **début** » et « **fin** » exclu et tels que « **n - début** » soit un multiple de « **marche** ».

Les expressions régulières

Les expressions régulières sont prises en charge par le module « `re` ». Ainsi, comme avec tous les modules en Python, nous avons seulement besoin de l'importer pour commencer à les utiliser.

Même si les expressions régulières ne sont pas propres à un langage, chaque implémentation introduit généralement des spécificités pour leur notation. L'antislash « `\` » tient un rôle particulier dans la syntaxe des expressions régulières puisqu'il permet d'introduire des caractères spéciaux. Comme il est également interprété dans les chaînes de caractères, il est nécessaire de le doubler pour ne pas le perdre dans l'expression.

```
>>> expression = "\btest\b"
>>> print(expression)
test
>>> expression = "\\btest\\b"
>>> print(expression)
\btest\b
>>> expression = r"\btest\b"
>>> print(expression)
\btest\b
```

Les chaînes de caractères peuvent éventuellement être précédées d'une lettre « `r` » ou « `R` ». Ces chaînes sont appelées chaînes brutes et traitent l'antislash « `\` » comme un caractère littéral.

La syntaxe des expressions régulières

La syntaxe des expressions régulières peut se regrouper en trois groupes de symboles :

- les symboles simples ;
- les symboles de répétition ;
- les symboles de regroupement.

Les symboles simples

Les symboles simples sont des caractères spéciaux qui permettent de définir des règles de capture pour un caractère du texte et sont réunis dans le tableau ci-dessous.

Symbole	Fonction
.	Remplace tout caractère sauf le saut de ligne.
^	Symbolise le début d'une ligne.
\$	Symbolise la fin d'une ligne.
\A	Symbolise le début de la chaîne.
\b	Symbolise le caractère d'espacement. Intercepté seulement au début ou à la fin d'un mot. Un mot est ici une séquence de caractères alphanumériques ou espace souligné.
\B	Comme « \b » mais uniquement lorsque ce caractère n'est pas au début ou à la fin d'un mot.
\d	Intercepte tout chiffre.
\D	Intercepte tout caractère sauf les chiffres.
\s	Intercepte tout caractère d'espacement : horizontale « \t », verticale « \v », saut de ligne « \n », retour à la ligne « \r », form feed « \f ».
\S	Symbol inverse de \s
\w	Intercepte tout caractère alphanumérique et espace souligné.
\W	Symbol inverse de « \w ».
\z	Symbolise la fin de la chaîne.

Voici quelques exemples :

```
>>> import re
>>> re.findall(r'\.', ' test ')
[ ' ', 't', 'e', 's', 't', ' ', '*' ]
>>> re.findall(r'\.', 'test\n')
[ 't', 'e', 's', 't' ]
>>> re.findall(r'\.', '\n')
[ ]
>>> re.findall(r'^le', "c'est le début")
[ ]
>>> re.findall(r'^le', "le début")
[ 'le' ]
>>> re.findall(r'mot$', 'mot mot mot')
[ 'mot' ]
>>> re.findall(r'\Aparoles', 'paroles, paroles, paroles,\nparoles,
encore des parooooles')
[ 'paroles' ]
>>> re.findall(r'\bpar\b', 'parfaitement')
[ ]
>>> re.findall(r'\bpar\b', 'par monts et par veaux')
[ 'par', 'par' ]
>>> re.findall(r'\Bpar\B', "imparfait")
[ 'par' ]
```

```
>>> re.findall(r'\Bpar\B', "parfait")
[ ]
>>> re.findall(r'\d', '1, 2, 3, nous irons au bois (à 12:15h)')
['1', '2', '3', '1', '2', '1', '5']
>>> print(''.join(re.findall(r'\D', '1, 2, 3, nous irons au bois (à
12:15h)')))
, , , nous irons au bois (à :h)
>>> len(re.findall(r'\s', "combien d'espaces dans la phrase ?"))
5
>>> len(re.findall(r'\s', "latoucheespaceestbloquée"))
0
>>> phrase = """Lancez vous!"""
>>> len(re.findall(r'\s', phrase))
1
>>> len(re.findall(r'\S', "combien de lettres dans la phrase ?"))
29
>>> ''.join(re.findall(r'\w', '*!mot-clé_*'))
'motclé_'
>>> ''.join(re.findall(r'\W', '*!mot-clé_*'))
'*!-*'
>>> re.findall(r'end\Z', 'The end will come')
[ ]
>>> re.findall(r'end\Z', 'This is the end')
['end']
```

Le fonctionnement de chacun de ces symboles est affecté par les options suivantes :

- **(A)SCII** : les symboles « \w », « \W », « \b », « \B », « \d », « \D », « \s » et « \S » se basent uniquement sur le code ascii.
 - **S ou DOTALL** : le saut de ligne est également intercepté par le symbole « \b ».
 - **(M)ULTILINE** : dans ce mode, les symboles « ^ » et « \$ » interceptent le début et la fin de chaque ligne.
 - **(U)NICODE** : les symboles « \w », « \W », « \b », « \B », « \d », « \D », « \s » et « \S » se basent sur de l'unicode.
 - **(I)gnorecase** : rend les symboles insensibles à la casse du texte.
 - **X ou VERBOSE** : autorise l'insertion d'espaces et de commentaires en fin de ligne, pour une mise en page de l'expression régulière plus lisible.

```
>>> re.findall(r"\w+", "这是一个 例子 是一", re.UNICODE)
['这是一个', '例子', '是一']
>>> re.findall(r"\w+", "这是一个 例子 是一")
['这是一个', '例子', '是一']
```

Les symboles de répétition

Les symboles simples peuvent être combinés et répétés par le biais de symboles de répétition.

Symbol	Fonction
*	Répète le symbole précédent de 0 à n fois (autant que possible).
+	Répète le symbole précédent de 1 à n fois (autant que possible).
?	Répète le symbole précédent 0 ou 1 fois (autant que possible).
{n}	Répète le symbole précédent n fois.
{n,m}	Répète le symbole précédent entre n et m fois inclus. n ou m peuvent être omis comme pour les tranches de séquences. Dans ce cas ils sont remplacés respectivement par 0 et *.
{n,m}?	Équivalent à {n,m} mais intercepte le nombre minimum de caractères.
e1 e2	Intercepte l'expression e1 ou e2. (OR)
[]	Regroupe des symboles et caractères en un jeu.

Voici quelques exemples :

```
>>> import re
>>> re.findall(r'pois*', 'poisson pois poilant poi')
['poiss', 'pois', 'poi', 'poi']
>>> re.findall(r'pois+', 'poisson pois poilant poi')
['poiss', 'pois']
>>> re.findall(r'pois?', 'poisson pois poilant poi')
['pois', 'pois', 'poi', 'poi']
>>> re.findall(r'pois{2}', 'poisson pois poilant poi')
['poiss']
>>> re.findall(r'pois{2,4}', 'poisssssssssson pois poilant poi')
['poissss']
>>> re.findall(r'pois{,4}', 'poisssssssssson pois poilant poi')
['poissss', 'pois', 'poi', 'poi']
>>> re.findall(r'pois{2,}', 'poisssssssssson pois poilant poi')
['poisssssssssssss']
>>> re.findall(r'pois{2,4}?', 'poisssssssssson pois poilant poi')
['poiss']
>>> re.findall(r'pois{2,}?', 'poisssssssssson pois poilant poi')
['poiss']
>>> re.findall(r'pois{,4}?', 'poisssssssssson pois poilant poi')
['poi', 'poi', 'poi', 'poi']
>>> re.findall(r'Mr|Mme', 'Mr et Mme')
['Mr', 'Mme']
>>> re.findall(r'Mr|Mme', 'Mr Untel')
['Mr']
>>> re.findall(r'Mr|Mme', 'Mme Unetelle')
['Mme']
>>> re.findall(r'Mr|Mme', 'Mlle Unetelle')
[]
>>> re.findall(r'[abc]def', 'adef bdef cdef')
```

```
[ 'adef' , 'bdef' , 'cdef' ]
```

Le regroupement de caractères accepte aussi des caractères d'abréviation, à savoir :

- - : définit une plage de valeurs. « **[a-z]** » représente par exemple toutes les lettres de l'alphabet en minuscules.
- ^ : placé en début de jeu, définit la plage inverse. « **[^a-z]** » représente par exemple tous les caractères sauf les lettres de l'alphabet en minuscules.

Les symboles de répétition « ? », « * » et « + » sont dits gloutons ou greedy : comme ils répètent autant de fois que possible le symbole précédent, des effets indésirables peuvent survenir.

```
>>> telRegex = re.compile(r'''(\d{2})[ ]\((\d+)\)\d
                           [\.]\d{2}[\.]\d{2}
                           [\.]\d{2}[\.]\d{2}''', re.VERBOSE)
>>> tel = telRegex.search(
    'Mobile 33 (0)6.85.20.70.68 Tél 33 (0)3.88.27.13.34')
>>> tel.group()
'33 (0)6.85.20.70.68'
>>> telRegex.findall(
    'Mobile 33 (0)6.85.20.70.68 Tél 33 (0)3.88.27.13.34')
['33 (0)6.85.20.70.68', '33 (0)3.88.27.13.34']
```

Dans l'exemple suivant, l'expression régulière tente d'extraire les balises html du texte sans succès : le texte complet est intercepté car il correspond au plus grand texte possible pour le motif. La solution est d'ajouter un symbole « ? » après le symbole greedy, pour qu'il n'intercepte que le texte minimum.

```
>>> chaine = '<div><span>le titre</span></div>'
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search(chaine)
>>> mo.group()
'<div>'
>>> nongreedyRegex.findall(chaine)
['<div>', '<span>', '</span>', '</div>']
>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search(chaine)
>>> mo.group()
'<div><span>le titre</span></div>'
>>> greedyRegex.findall(chaine)
['<div><span>le titre</span></div>']
```

Les symboles de regroupement

Les symboles de regroupement offrent des fonctionnalités qui permettent de combiner plusieurs expressions régulières, au-delà des jeux de caractères « [] » et de la fonction « OR », et d'associer à chaque groupe un identifiant unique. Certaines d'entre elles permettent aussi de paramétriser localement le fonctionnement des expressions.

Symbol	Fonction
(e)	Forme un groupe avec l'expression e. Si les caractères « (» ou «) » sont utilisés dans e, ils doivent être préfixés de « \ »
(?FLAGS)	Insère directement des flags d'options dans l'expression. S'applique à l'expression complète quel que soit son positionnement.
(?:e)	Similaire à (e) mais le groupe intercepté n'est pas conservé.
(?P<name>e)	Associe l'étiquette name au groupe. Ce groupe peut ensuite être manipulé par ce nom par le biais des API de « re », ou même dans la suite de l'expression régulière.
(?#comment)	Insère un commentaire, qui sera ignoré. Le mode « verbose » est plus souple pour l'ajout direct de commentaires en fin de ligne.
(?=e)	Similaire à (e) mais le groupe n'est pas consommé.
(?!e)	Le groupe n'est pas consommé et est intercepté uniquement si le pattern (le motif) n'est pas e. (?!e) est le symbole inverse de (?=e)
(?<=e1)e2	Intercepte e2 à condition qu'elle soit prefixée d'e1.
(?<!e1)e2	Intercepte e2 à condition qu'elle ne soit pas prefixée d'e1.
(?(id/name)e1 e2)	Rend l'expression conditionnelle : si le groupe d'identifiant id ou name existe, e1 est utilisée, sinon e2. e2 peut être omise, dans ce cas e1 ne s'applique que si le groupe id ou name existe. Dans l'exemple <123> et 123 sont interceptés mais pas <123>

Voici quelques exemples.

```
>>> re.findall(r'((\d{3}))(\d{2})(\d{4})', '(03)80666666')
[('03', '80', '666666')]
>>> re.findall(r'(?i)AAZ*', 'aaZZzRr')
['aaZZz']
>>> re.findall(r'(:\d{3})(?:\d{2})(.*?)(\d{4})', '(03)80666666')
['666666']
>>> match = re.search(r'(\d{3})(\d{2})(\d{4})', '0380666666')
>>> match.group('numero')
'666666'
>>> re.findall(r'(# récupération des balises)<.*?>', \
    '<h2><span>hopla</span></h2>')
['<h2>', '<span>', '</span>', '</h2>']
>>> re.findall(r'John(= Doe)', 'John Doe')
['John']
>>> re.findall(r'John(= Doe)', 'John Minor')
[]
>>> re.findall(r'John(?! Doe)', 'John Doe')
```

```
[ ]  
>>> re.findall(r'John(?! Doe)', 'John Minor')  
[ 'John' ]  
>>> re.findall(r'(?=<=John )Doe', 'John Doe')  
[ 'Doe' ]  
>>> re.findall(r'(?<=John )Doe', 'John Minor')  
[ ]  
>>> re.findall(r'(?<!John )Doe', 'John Doe')  
[ ]  
>>> re.findall(r'(?<!John )Doe', 'Juliette Doe')  
[ 'Doe' ]  
>>> re.match(r'(?P<one><)?(\\d+)(?(one)>)', '<123')  
>>> match = re.match(r'(?P<one><)?(\\d+)(?(one)>)', '123')  
>>> match.group()  
'123'  
>>> match = re.match(r'(?P<one><)?(\\d+)(?(one)>)', '<123>')  
>>> match.group()  
'<123>'
```

Les fonctions et objets de re

Le module « `re` » contient un certain nombre de fonctions qui permettent de manipuler des motifs et les exécuter sur des chaînes :

Fonction	Description
<code>compile(pattern[, flags])</code>	compile le motif pattern et renvoie un objet de type « <code>SRE_Pattern</code> ».
<code>escape(string)</code>	ajoute un antislash « \ » devant tous les caractères non alphanumériques contenus dans string. Permet d'utiliser la chaîne dans les expressions régulières.
<code>findall(pattern, string[, flags])</code>	renvoie une liste des éléments interceptés dans la chaîne string par le motif pattern. Lorsque le motif est composé de groupes, chaque élément est un tuple composé de chaque groupe.
<code>finditer(pattern, string[, flags])</code>	équivalente à « <code>findall</code> », mais un itérateur sur les éléments est renvoyé. flags est un entier contenant d'éventuels flags, appliqués au motif complet.
<code>match(pattern, string[, flags])</code>	renvoie un objet de type « <code>MatchObject</code> » si le début de la chaîne string correspond au motif. flags est un entier contenant d'éventuels flags, appliqués au motif complet.
<code>search(pattern, string[, flags])</code>	équivalente à « <code>match</code> » mais recherche le motif dans toute la chaîne.
<code>split(pattern, string[, maxsplit=0])</code>	équivalente au « <code>split</code> » de l'objet string. Renvoie une séquence de chaînes délimitées par le motif pattern. Si maxsplit est fourni, limite le nombre d'éléments à maxsplit, le dernier élément regroupant la fin de la chaîne lorsque maxsplit est atteint.
<code>sub(pattern, repl, string[, count])</code>	remplace les occurrences du motif pattern de string par repl. repl peut être une chaîne ou un objet « <code>callable</code> » qui reçoit un objet « <code>MatchObject</code> » et renvoie une chaîne. Si count est fourni, limite le nombre de remplacements.
<code>subn(pattern, repl, string[, count])</code>	équivalente à « <code>sub</code> » mais renvoie un tuple (nouvelle chaîne, nombre de remplacements) au lieu de la chaîne.

```
>>> import re
>>> motif = re.compile('(\bMr\b|\bMme\b|\bMlle\b)\s([\A-Za-z]+)\s([\A-Za-z]+)')
>>> print(motif.sub(r'Nom: \3, Prénom: \2', 'Mr John Doe'))
Nom: Doe, Prénom: John
>>> print(motif.sub(r'Mon nom est \g<3>, \g<2> \g<3>\'\
                   , 'Mr Jean Bon'))
Mon nom est Bon, Jean
```

L'écriture simplifiée des fonctions

Lorsque le code d'une fonction tient en une ligne et est le résultat d'une expression, il est possible de condenser son écriture à l'aide du mot-clé « **lambda** ». Cette syntaxe est issue de langages fonctionnels comme le Lisp.

```
nom_fonction = lambda param_1, ..., param_n : expression
```

L'exemple suivant utilise cette écriture pour définir la fonction min retournant le plus petit entre deux nombres positifs.

```
>>> min = lambda x,y : (abs (x+y) - abs (x-y)) / 2
>>> print(min(1,2))
1.0
>>> print(min(5,4))
4.0
>>> def min(x,y):
...     return (abs (x+y) - abs (x-y))/2

>>> print(min(1,2))
1.0
>>> print(min(5,4))
4.0
```

La fonction lambda considère le contexte de fonction qui la contient comme son contexte. Il est possible de créer des fonctions lambda mais celle-ci utiliseront le contexte dans l'état où il est au moment de son exécution et non au moment de sa création.

```
>>> fs = []
>>> for a in range (0,5) :
...     f = lambda x : x + a
...     fs.append(f)
...     print(a)
0
1
2
3
4
>>> print(a)
4
>>> for f in fs :
...     print('a = ', a,' lambda = ',f(1))
a = 4 lambda = 5
```

Pour que le programme affiche les entiers de 1 à 5, il faut préciser à la fonction lambda une variable y égale à a au moment de la création de la fonction et qui sera intégrée au contexte de la fonction lambda.

```
>>> fs = []
>>> for a in range (0,5) :
...     f = lambda x,y=a : x + y
...     fs.append(f)
```

```
...
>>> for f in fs :
...     print('a = ', a,' lambda = ',f(1))
...
a = 4 lambda = 1
a = 4 lambda = 2
a = 4 lambda = 3
a = 4 lambda = 4
a = 4 lambda = 5
```

La fonction map

La fonction « `map` » renvoie une liste correspondant à l'ensemble des éléments de la séquence.

```
map : map(fonction, séquence[, séquence...]) -> liste
```

Avant d'être inséré dans la liste, chaque élément est passé à la fonction fournie. Cette dernière doit donc être de la forme :

```
fonction(element) -> element
```

Lorsque plusieurs séquences sont fournies, la fonction reçoit une liste d'arguments correspondants à un élément de chaque séquence. Si les séquences ne sont pas de la même longueur, elles sont complétées avec des éléments à la valeur « `None` ».

La fonction peut être définie à « `None` », et dans ce cas tous les éléments des séquences fournies sont conservés.

```
>>> a = lambda x: x**2
>>> b = list(map(a, (2,3,4)))
>>> b
[4, 9, 16]
>>> list(map(pow, (7, 5, 3), (2, 3, -2)))
[49.0, 125.0, 0.1111111111111111]
>>> a, c = [1,2,3,4,5], [2, -3, 5,7, -0.5]
>>> mul = lambda x, y: x*y
>>> d = sum(map(mul, a,c))
>>> d
36.5

>>> sq = lambda x: x**2
>>> cu = lambda y: y**3
>>> fc = (sq,cu)
>>> #Retour carré & cube de r - utilisation de 'map'
>>> def vv(r): return list(map(lambda z: z(r), fc))
>>> res1 = vv(5)
>>> res1
[25, 125]

>>> def meva(dd):
    'Avec dd comme sequence de nombres,\n\
     retrouvez leur moyenne et variance'
    bb = len(dd)
    med = sum(dd)/bb
    vr = sum(list(map(sq,dd)))/bb - med**2
    return med, vr

>>> seq = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, \
       6.0, 7.0, 8.0, 9.0, 10.0, 11.0, \
       12.0, 13.0, 14.0]
>>> res1 = meva(seq)
>>> res1
(7.0, 18.66666666666667)
```

La fonction filter

La fonction « **filter** » renvoie une liste correspondant à l'ensemble des éléments de la séquence pour lesquels la fonction fournie retourne vrai.

```
filter(fonction, séquence[, séquence...]) -> liste
```

Avant d'être inséré dans la liste, chaque élément est passé à la fonction fournie. Cette dernière doit donc être de la forme :

```
fonction(element) -> booléen
```

```
>>> symbols = '€¢£¤ç'
>>> code_car = [ord(s) for s in symbols if ord(s) > 160]
>>> code_car
[8364, 163, 167, 231]
>>>
>>> code_car = list(filter(lambda c: c > 160, map(ord, symbols)))
>>> code_car
[8364, 163, 167, 231]

>>> def factorial(n):
...     '''returns n!'''
...     return 1 if n < 2 else n * factorial(n-1)
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> map(factorial, range(11))
<map object at 0x0000023169057CC0>
>>> list(map(factorial, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
>>> list(map(factorial, filter(lambda n: n % 2, range(6))))
[1, 6, 120]
```

3

Les outils indispensables

Les tableaux numériques

Ce type ne fait pas partie du langage python standard mais il est couramment utilisé. Il permet de convertir des listes en une structure plus appropriée au calcul qui sont nettement plus rapides. En contrepartie, il n'est pas aussi rapide d'ajouter ou supprimer des éléments.

Le type de base dans « **NumPy** » est le tableau unidimensionnel ou multidimensionnel composé d'éléments de même type, et est indexé par un « **tuple** » d'entiers non négatifs. La classe correspondante est « **ndarray** », à ne pas confondre avec la classe Python « **array.array** » qui gère seulement des tableaux unidimensionnels et présente des fonctionnalités comparativement limitées.

ndarray.ndim	dimension du tableau (nombre d'axes)
ndarray.shape	tuple d'entiers indiquant la taille dans chaque dimension ; une matrice à n lignes et m colonnes : (n,m)
ndarray.size	nombre total d'éléments du tableau
ndarray.dtype	type de (tous) les éléments du tableau ; il est possible d'utiliser les types prédéfinis comme numpy.int64 ou numpy.float64 ou définir de nouveaux types
ndarray.data	les données du tableau ; en général, pour accéder aux données d'un tableau on passe plutôt par les indices

L'emploi de raccourcis « **np** » plutôt que « **numpy** » permet de faciliter l'écriture des appels des fonctions de la librairie.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> print(type(a))
<class 'numpy.ndarray'>
>>> print(a.shape)
(3,)
>>> print(a[0], a[1], a[2])
1 2 3
>>> a[0] = 5
>>> print(a)
[5 2 3]
>>> b = np.array([[1,2,3],[4,5,6]])
>>> print(b.shape)
(2, 3)
>>> print(b[0, 0], b[0, 1], b[1, 0])
1 2 4
```

La librairie « **numpy** » fournit également de nombreuses fonctions pour créer des tableaux :

```
>>> import numpy as np
>>> a = np.zeros((2,2))
>>> print(a)
[[ 0.  0.]
 [ 0.  0.]]
>>> b = np.ones((1,2))
>>> print(b)
[[ 1.  1.]]
```

```
>>> c = np.full((2,2), 7)
>>> print(c)
[[ 7.  7.]
 [ 7.  7.]]
>>> d = np.eye(2)
>>> print(d)
[[ 1.  0.]
 [ 0.  1.]]
>>> e = np.random.random((2,2))
>>> print(e)
[[ 0.72843251  0.10508965]
 [ 0.84919262  0.75706259]]
```

Vous pouvez également mélanger l'indexation des entiers avec l'indexation des tranches. Cependant, cela produira un tableau de rang inférieur à celui du tableau original.

```
>>> import numpy as np
>>> a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> row_r1 = a[1, :]
>>> row_r2 = a[1:2, :]
>>> print(row_r1, row_r1.shape)
[5 6 7 8] (4,)
>>> print(row_r2, row_r2.shape)
[[5 6 7 8]] (1, 4)
>>> col_r1 = a[:, 1]
>>> col_r2 = a[:, 1:2]
>>> print(col_r1, col_r1.shape)
[ 2   6  10] (3,)
>>> print(col_r2, col_r2.shape)
[[ 2]
 [ 6]
 [10]] (3, 1)

>>> a.ndim
2
>>> a.shape
(3, 4)
>>> a.size
12
>>> a.dtype
dtype('int32')
```

La création de tableaux

De nombreuses méthodes de création de tableaux sont disponibles. D'abord, un tableau peut être créé à partir d'une « **liste** » ou d'un « **tuple** », à condition que tous les éléments soient de même type, le type des éléments du tableau est déduit du type des éléments de la « **liste** » ou « **tuple** ».

```
>>> import numpy as np
>>> ti = np.array([1, 2, 3, 4])
>>> ti
array([1, 2, 3, 4])
>>> ti.dtype
dtype('int32')
>>> tf = np.array([1.5, 2.5, 3.5, 4.5])
>>> tf.dtype
dtype('float64')
```

À partir des listes simples sont produits des tableaux unidimensionnels, à partir des listes de listes (de même taille) des tableaux bidimensionnels, et ainsi de suite.

```
>>> tf2d = np.array([[1.5, 2, 3], [4, 5, 6]])
>>> tf2d
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

Le type du tableau peut être indiqué explicitement à la création, des conversions sont effectuées pour les valeurs fournies.

```
>>> tfi = np.array([[1.5, 2, 3], [4, 5, 6]], dtype=int)
>>> tfi
array([[1, 2, 3],
       [4, 5, 6]])
>>> tfi.dtype
dtype('int32')
>>> tfi.shape
(2, 3)
>>> tfi.ndim
2
>>> tfi.size
6
```

Il est souvent nécessaire de créer des tableaux remplis de 0, de 1, ou dont le contenu n'est pas initialisé. Par défaut, le type des tableaux ainsi créés est float64.

```
>>> tz2d = np.zeros((3,4))
>>> tz2d
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> tu2d = np.ones((3,4))
>>> tu2d
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> id2d = np.eye(5)
>>> id2d
array([[ 1.,  0.,  0.,  0.,  0.],
```

```

[ 0.,  1.,  0.,  0.,  0.],
[ 0.,  0.,  1.,  0.,  0.],
[ 0.,  0.,  0.,  1.,  0.],
[ 0.,  0.,  0.,  0.,  1.]])
>>> tni2d = np.empty((3,4))
>>> tni2d
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])

```

Des tableaux peuvent être initialisés aussi par des séquences générées.

```

>>> ts1d = np.arange(0, 40, 5)
>>> ts1d
array([ 0,  5, 10, 15, 20, 25, 30, 35])
>>> ts1d2 = np.linspace(0, 35, 8)
>>> ts1d2
array([ 0.,  5., 10., 15., 20., 25., 30., 35.])
>>> ta2d = np.random.rand(3,5)
>>> ta2d
array([[ 0.57132596,  0.00819932,  0.17252526,  0.03082183,  0.53830712],
       [ 0.83042098,  0.75446994,  0.25991065,  0.62847979,  0.15797572],
       [ 0.08598435,  0.00754915,  0.50282216,  0.72654331,  0.90316578]])

```

Les tableaux peuvent être redimensionnés en utilisant « **reshape** ».

```

>>> tr = np.arange(20)
>>> tr
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
>>> tr.reshape(4,5)
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> tr.reshape(2,10)
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
>>> tr.reshape(20)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])

```

L'affichage des tableaux

Les tableaux unidimensionnels sont affichés comme des listes, les tableaux bidimensionnels comme des matrices et les tableaux tridimensionnels comme des listes de matrices.

```
>>> ta1d = np.random.rand(5)
>>> ta1d
array([ 0.79064584,  0.06775449,  0.39452303,  0.01723293,  0.25219518])
>>> ta3d = np.random.rand(2,3,5)
>>> ta3d
array([[[ 0.48491945,  0.64296218,  0.36891503,  0.32513357,  0.67010718],
       [ 0.25877942,  0.54814236,  0.76000793,  0.80090898,  0.49214619],
       [ 0.25499884,  0.54651908,  0.33886573,  0.52616233,  0.35187091]],
      [[ 0.32749216,  0.49832111,  0.27457405,  0.48603902,  0.10054472],
       [ 0.0191687 ,  0.06256129,  0.1715306 ,  0.60250356,  0.2672456 ],
       [ 0.78437914,  0.72383496,  0.16868848,  0.84315508,  0.0267777 ]]])
```

Si un tableau est considéré trop grand pour être affiché en entier, « NumPy » affiche le début et la fin, avec des « ... » au milieu.

```
>>> ta2d = np.random.rand(30,50)
>>> ta2d
array([[ 0.6835294 ,  0.46886275,  0.81712639, ...,  0.01835169,
         0.89688112,  0.37007707],
       [ 0.5791199 ,  0.30927608,  0.14415233, ...,  0.92076745,
         0.57043746,  0.42868106],
       [ 0.66825711,  0.84835774,  0.43771497, ...,  0.24745712,
         0.26247765,  0.74588404],
       ...,
       [ 0.35933294,  0.94859926,  0.77803165, ...,  0.71668177,
         0.74483146,  0.44250986],
       [ 0.69740862,  0.07109011,  0.86203099, ...,  0.58584055,
         0.74100104,  0.32189666],
       [ 0.38432312,  0.39479927,  0.34965807, ...,  0.37925167,
         0.47014018,  0.68201961]])
```

L'accès aux composantes d'un tableau

```
>>> tr = np.arange(20)
>>> tr
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
>>> a = tr[:2]
>>> a
array([0, 1])
>>> a[0] = 3
>>> tr
array([ 3,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

Les données ne sont pas copiées de « `tr` » vers un nouveau tableau « `a` », la modification d'un élément de `a` avec « `a[0] = 3` » change aussi le contenu de « `tr[0]` ». Pour obtenir une copie il faut utiliser « `copy` ».

```
>>> a = tr[:2].copy()
>>> a
array([3, 1])
>>> a[0] = 0
>>> a
array([0, 1])
>>> tr
array([ 3,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

Pour les tableaux multidimensionnels, lors des itérations c'est le dernier indice qui change le plus vite, ensuite l'avant-dernier, et ainsi de suite. Par exemple, pour les tableaux bidimensionnels c'est l'indice de colonne qui change d'abord et ensuite celui de ligne ainsi le tableau est lu ligne après ligne.

```
>>> ta2d = np.random.rand(3,5)
>>> ta2d
array([[ 0.99327185,  0.85869692,  0.87930205,  0.03911094,  0.93273251],
       [ 0.86430312,  0.16671027,  0.61613967,  0.84354217,  0.1950944 ],
       [ 0.40074433,  0.52467501,  0.71025502,  0.55148182,  0.0599687 ]])
>>> ta2d[0,0]
0.99327184504277644
>>> ta2d[0,:]
array([ 0.99327185,  0.85869692,  0.87930205,  0.03911094,  0.93273251])
>>> ta2d[0]
array([ 0.99327185,  0.85869692,  0.87930205,  0.03911094,  0.93273251])
>>> ta2d[:,0]
array([ 0.99327185,  0.86430312,  0.40074433])
>>> ta2d[:,2,:]
array([[ 0.99327185,  0.85869692],
       [ 0.86430312,  0.16671027]])
>>> for row in ta2d:
...     print(row)
...
[[ 0.99327185  0.85869692  0.87930205  0.03911094  0.93273251]
 [ 0.86430312  0.16671027  0.61613967  0.84354217  0.1950944 ]
 [ 0.40074433  0.52467501  0.71025502  0.55148182  0.0599687 ]]
```

Lecture et écriture d'un tableau

Les fonctions de lecture / écriture de tableaux depuis / dans des fichiers sont variées, nous regarderons rapidement deux des plus simples et plus rapides car les fichiers de données ont en général des formats assez simples.

La fonction :

```
numpy.loadtxt(fname, dtype=<type 'float'>,
              comments='#', delimiter=None, converters=None,
              skiprows=0, usecols=None, unpack=False, ndmin=0),
```

qui retourne un **ndarray**, réalise une lecture à partir d'un fichier texte et est bien adaptée aux tableaux bidimensionnels ; chaque ligne de texte doit contenir un même nombre de valeurs. Les principaux paramètres sont :

- fname** : fichier ou chaîne de caractères ; si le fichier a une extension .gz ou .bz2, il est d'abord décompressé.
- dtype** : type, optionnel, float par défaut.
- comments** : chaîne de caractères, optionnel, indique une liste de caractères employée dans le fichier pour précéder des commentaires à ignorer lors de la lecture.
- delimiter** : chaîne de caractères, optionnel, indique la chaîne de caractères employée pour séparer des valeurs, par défaut l'espace.
- converters** : dictionnaire, optionnel, pour permettre des conversions.
- skiprows** : entier, optionnel, pour le nombre de lignes à sauter en début de fichier par défaut 0.
- usecols** : séquence, optionnel, indique les colonnes à lire ; par ex. **usecols** = [1,4,5] extrait la 2ème, 5ème et 6ème colonne ; par défaut toutes les colonnes sont extraites.
- unpack** : booléen, optionnel, false par défaut ; si true, le tableau est transposé.
- ndmin** : entier, optionnel, le tableau a au moins **ndmin** dimensions ; par défaut 0.

```
>>> from io import StringIO
>>> import numpy as np
>>> nutriage = np.loadtxt('donnees/nutriage.csv',
...                      delimiter=';', skiprows=1)
>>> nutriage.shape
(226, 13)
>>> nutriage[:10,:6]
array([[ 2.,    1.,    0.,    0.,   151.,   58.],
       [ 2.,    1.,    1.,    1.,   162.,   60.],
       [ 2.,    1.,    0.,    4.,   162.,   75.],
       [ 2.,    1.,    0.,    0.,   154.,   45.],
       [ 2.,    1.,    2.,    1.,   154.,   50.],
       [ 2.,    1.,    2.,    0.,   159.,   66.],
       [ 2.,    1.,    2.,    0.,   160.,   66.],
       [ 2.,    1.,    0.,    2.,   163.,   66.],
       [ 2.,    1.,    0.,    3.,   154.,   60.]])
```

```
[ 2., 1., 0., 2., 160., 77.]])
```

La fonction :

```
numpy.savetxt(fname, x, fmt='%.18e', delimiter=' ',
newline='\n', header='',
footer='', comments='# ')
```

permet d'écrire un tableau dans un fichier texte. Les paramètres sont :

- fname** : fichier ; si le fichier a une extension .gz ou .bz2, il est compressé.
- x** : le tableau à écrire dans le fichier texte.
- fmt** : chaîne de caractères, optionnel ; indique le formatage du texte écrit.
- delimiter** : chaîne de caractères, optionnel, indique la chaîne de caractères employée pour séparer des valeurs, par défaut `` `` (l'espace).
- newline** : chaîne de caractères, optionnel, indique le caractère à employer pour séparer des lignes.
- header** : chaîne de caractères, optionnel, indique le commentaire à ajouter au début du fichier.
- footer** : chaîne de caractères, optionnel, indique le commentaire à ajouter à la fin du fichier.
- comments** : caractère à ajouter avant header et footer pour en faire des commentaires ; par défaut #.

```
>>> np.savetxt('donnees/nutriage.txt', nutriage, delimiter=', ')
>>> nutriage = np.loadtxt('donnees/nutriage.txt', delimiter=', ')
>>> nutriage.shape
(226, 13)
>>> nutriage[:10,:6]
array([[ 2., 1., 0., 0., 151., 58.],
       [ 2., 1., 1., 1., 162., 60.],
       [ 2., 1., 0., 4., 162., 75.],
       [ 2., 1., 0., 0., 154., 45.],
       [ 2., 1., 2., 1., 154., 50.],
       [ 2., 1., 2., 0., 159., 66.],
       [ 2., 1., 2., 0., 160., 66.],
       [ 2., 1., 0., 2., 163., 66.],
       [ 2., 1., 0., 3., 154., 60.],
       [ 2., 1., 0., 2., 160., 77.]])
```

Autres opérations d'entrée et sortie :

fromfile(file[, dtype, count, sep]) : Construction d'un tableau à partir d'un fichier texte ou binaire.

fromregex(file, regexp, dtype) : Construction d'un tableau à partir d'un fichier texte, avec un parseur d'expressions régulières.

genfromtxt() : Fonction plus flexible pour la construction d'un tableau à partir d'un fichier texte, avec une gestion des valeurs manquantes.

load(file[, mmap_mode, allow_pickle, ...]) : Lecture de tableaux (ou autres objets) à partir de fichiers .npy, .npz ou autres fichiers de données serialisées.

loadtxt(fname[, dtype, comments, delimiter, ...]): Lecture de données à partir d'un fichier texte.

ndarray.tofile(fid[, sep, format]): Ecriture d'un tableau dans un fichier texte ou binaire (par défaut).

save(file, arr[, allow_pickle, fix_imports]): Ecriture d'un tableau dans un fichier binaire de type .npy.

savetxt(fname, X[, fmt, delimiter, newline, ...]): Ecriture d'un tableau dans un fichier texte.

savez(file, *args, **kwds): Ecriture de plusieurs tableaux dans un fichier de type .npz sans compression.

savez_compressed(file, *args, **kwds): Ecriture de plusieurs tableaux dans un fichier de type .npz avec compression.

Les opérations simples sur les tableaux

Concaténation de tableaux bidimensionnels

```
>>> tu2d = np.ones((2,2))
>>> tu2d
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> tb2d = np.ones((2,2))*2
>>> tb2d
array([[ 2.,  2.],
       [ 2.,  2.]])
>>> tcl = np.concatenate((tu2d,tb2d))
>>> tcl
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 2.,  2.],
       [ 2.,  2.]])
>>> tcl = np.concatenate((tu2d,tb2d),axis=0)    # ou np.vstack
>>> tcl
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 2.,  2.],
       [ 2.,  2.]])
>>> tcl = np.concatenate((tu2d,tb2d),axis=1)    # ou np.hstack
>>> tcl
array([[ 1.,  1.,  2.,  2.],
       [ 1.,  1.,  2.,  2.]])
```

Ajouter un tableau unidimensionnel comme colonne à un tableau bidimensionnel.

```
>>> from numpy import newaxis
>>> tuld = np.ones(2)
>>> tuld
array([ 1.,  1.])
>>> tuld[:,newaxis]
array([[ 1.],
       [ 1.]])
>>> np.column_stack((tuld[:,newaxis],tb2d))
array([[ 1.,  2.,  2.],
       [ 1.,  2.,  2.]])
>>> np.hstack((tuld[:,newaxis],tb2d))
array([[ 1.,  2.,  2.],
       [ 1.,  2.,  2.]])
```

Opérations arithmétiques élément par élément

```
>>> tsomme = tb2d - tuld
>>> tsomme
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> tb2d*5
array([[ 10.,  10.],
       [ 10.,  10.]])
>>> tb2d**2
array([[ 4.,  4.],
```

```

        [ 4.,  4.]])
>>> tc = np.hstack((tb2d,tu1d[:,newaxis]))
>>> tc
array([[ 2.,  2.,  1.],
       [ 2.,  2.,  1.]])
>>> tc > 1
array([[ True,  True, False],
       [ True,  True, False]], dtype=bool)
>>> tb2d * tb2d
array([[ 4.,  4.],
       [ 4.,  4.]])
>>> tb2d *= 3
>>> tb2d
array([[ 6.,  6.],
       [ 6.,  6.]])
>>> tb2d += tu2d
>>> tb2d
array([[ 7.,  7.],
       [ 7.,  7.]])
>>> tb2d.sum()
28.0
>>> nutriage = np.loadtxt('donnees/nutriage.csv',
...                         delimiter=';',skiprows=1)
>>> nutriage.shape
(226, 13)
>>> nutriage[:10,:6]
array([[ 2.,  1.,  0.,  0., 151.,  58.],
       [ 2.,  1.,  1.,  1., 162.,  60.],
       [ 2.,  1.,  0.,  4., 162.,  75.],
       [ 2.,  1.,  0.,  0., 154.,  45.],
       [ 2.,  1.,  2.,  1., 154.,  50.],
       [ 2.,  1.,  2.,  0., 159.,  66.],
       [ 2.,  1.,  2.,  0., 160.,  66.],
       [ 2.,  1.,  0.,  2., 163.,  66.],
       [ 2.,  1.,  0.,  3., 154.,  60.],
       [ 2.,  1.,  0.,  2., 160.,  77.]])
>>> nutriage.min()
0.0
>>> nutriage.max()
188.0
>>> nutriage.sum()
75004.0
>>> nutriage.sum(axis=0)
array([ 367.,   363.,   161.,   366., 37055., 15025., 16832.,
       847.,   592., 1014.,   991.,   529.,    862.])
>>> nutriage[:20,:].sum(axis=1)
array([ 307.,  317.,  342.,  306.,  298.,  332.,  332.,  330.,  337.,
       346.,  364.,  344.,  331.,  320.,  342.,  346.,  313.,  311.,
       329.,  349.])

```

Algèbre linéaire

```

>>> n0 = nutriage[:2,5:7]
>>> n0
array([[ 58.,  72.],
       [ 60.,  68.]])

```

```
>>> n0.transpose()
array([[ 58.,  60.],
       [ 72.,  68.]])
>>> np.linalg.inv(n0)
array([[-0.18085106,  0.19148936],
       [ 0.15957447, -0.15425532]])
>>> n0.dot(tu2d)      # ou np.dot(n0,tu2d)
array([[ 130.,  130.],
       [ 128.,  128.]])
>>> tu2d
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> n0.dot(np.eye(2))
array([[ 58.,  72.],
       [ 60.,  68.]])
>>> n0.trace()
126.0
```

Résolution de systèmes linéaires

```
>>> y = np.array([[5.], [7.]])
>>> np.linalg.solve(n0, y)
array([[ 0.43617021],
       [-0.28191489]])
```

Valeurs et vecteurs propres

```
>>> vpvp = np.linalg.eig(n0)
>>> vpvp
(array([-2.91661399, 128.91661399]), array([[-0.76342008, -0.71244657],
       [ 0.64590231, -0.70172636]]))
```

Vectorisation de fonctions

Des fonctions Python qui travaillent sur des scalaires peuvent être vectorisées, c'est à dire travailler sur des tableaux, élément par élément.

```
>>> def addsubtract(a,b):
...     if a > b:
...         return a - b
...     else:
...         return a + b
...
>>> addsubtract(2,3)
5
>>> vec_addsubtract = np.vectorize(addsubtract)
>>> tu2d
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> n0
array([[ 58.,  72.],
       [ 60.,  68.]])
>>> vec_addsubtract(n0,tu2d)
array([[ 57.,  71.],
       [ 59.,  67.]])
```

La librairie Matplotlib

La librairie Matplotlib a vu le jour pour permettre de générer directement des graphiques à partir de Python. Au fil des années, Matplotlib est devenu une librairie puissante, compatible avec beaucoup de plateformes, et capable de générer des graphiques dans beaucoup de formats différents.

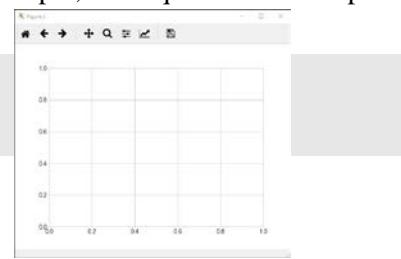
Pour commencer, mettons en place l'environnement de travail.

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use('seaborn-whitegrid')
>>> import numpy as np
```

Réaliser des graphiques simples

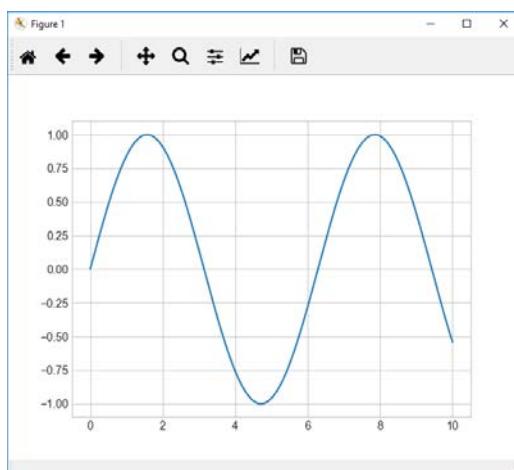
Commençons par étudier un cas simple, comme tracer la courbe d'une fonction. Nous avons vu un exemple de cette utilisation dans le chapitre 3 de la première partie de ce cours. Ici, nous allons le faire d'une manière moins simple, mais qui nous donne plus de possibilités.

```
>>> fig = plt.figure()
>>> ax = plt.axes()
>>> plt.show()
```



La variable **fig** correspond à un conteneur qui contient tous les objets (axes, labels, données, etc). Les axes correspondent au carré que l'on voit au-dessus, et qui contiendra par la suite les données du graphe.

```
>>> fig = plt.figure()
>>> ax = plt.axes()
>>> x = np.linspace(0, 10, 1000)
>>> ax.plot(x, np.sin(x));
[<matplotlib.lines.Line2D object at 0x0000022885641860>]
>>> plt.show()
```



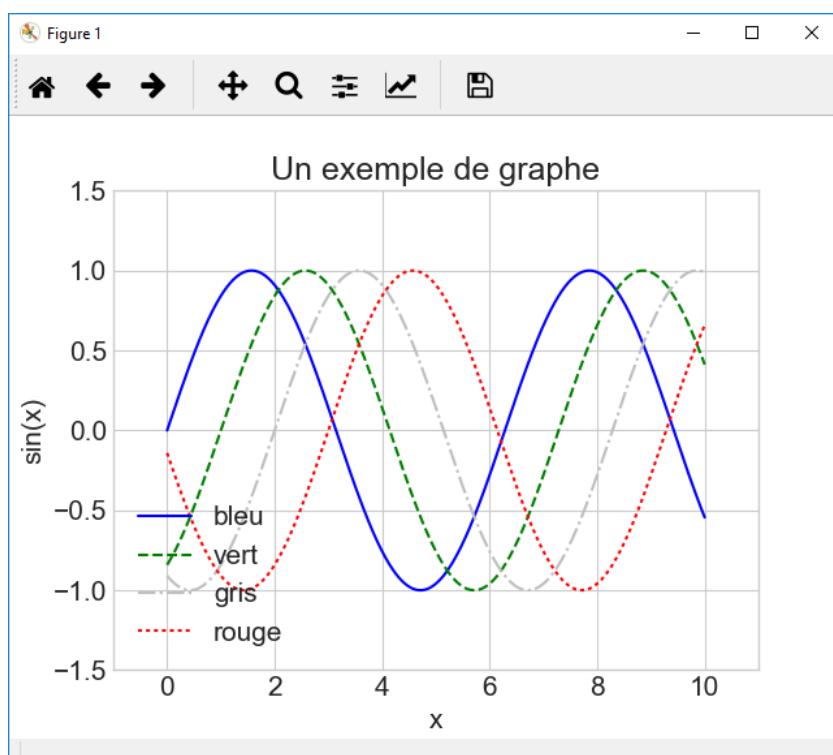
On aurait pu simplement taper `plt.plot(x, np.sin(x))`. Maintenant, voyons un exemple un peu plus poussé.

```
>>> # Changer la taille de police par défaut
... plt.rcParams.update({'font.size': 15})
```

```

>>> fig = plt.figure()
>>> ax = plt.axes()
>>> # Couleur spécifiée par son nom, ligne solide
>>> plt.plot(x, np.sin(x - 0), color='blue', linestyle='solid',
label='bleu')
[<matplotlib.lines.Line2D object at 0x00000228865DD198>]
>>> # Nom court pour la couleur, ligne avec des traits
>>> plt.plot(x, np.sin(x - 1), color='g', linestyle='dashed',
label='vert')
[<matplotlib.lines.Line2D object at 0x00000228865DD358>]
>>> # Valeur de gris entre 0 et 1, des traits et des points
>>> plt.plot(x, np.sin(x - 2), color='0.75', linestyle='dashdot',
label='gris')
[<matplotlib.lines.Line2D object at 0x00000228865E6320>]
>>> # Couleur spécifié en RGB, avec des points
>>> plt.plot(x, np.sin(x - 3), color='#FF0000', linestyle='dotted',
label='rouge')
[<matplotlib.lines.Line2D object at 0x00000228865E6AC8>]
>>> # Les limites des axes, essayez aussi les arguments 'tight' et
'equal' pour voir leur effet
>>> plt.axis([-1, 11, -1.5, 1.5]);
[-1, 11, -1.5, 1.5]
>>> # Les labels
>>> plt.title("Un exemple de graphe")
<matplotlib.text.Text object at 0x000002288643DCF8>
>>> # La légende est générée à partir de l'argument label de la
fonction plot. L'argument loc spécifie le placement de la légende
>>> plt.legend(loc='lower left');
<matplotlib.legend.Legend object at 0x00000228865E6B38>
>>> # Titres des axes
>>> ax = ax.set(xlabel='x', ylabel='sin(x)')
>>> plt.show()

```



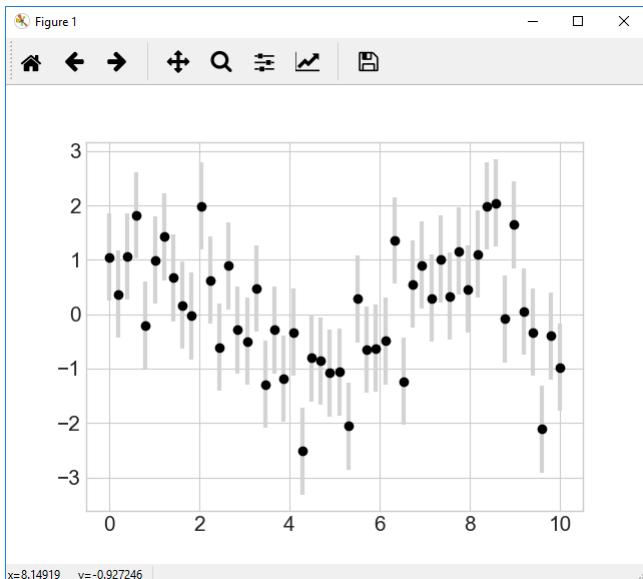
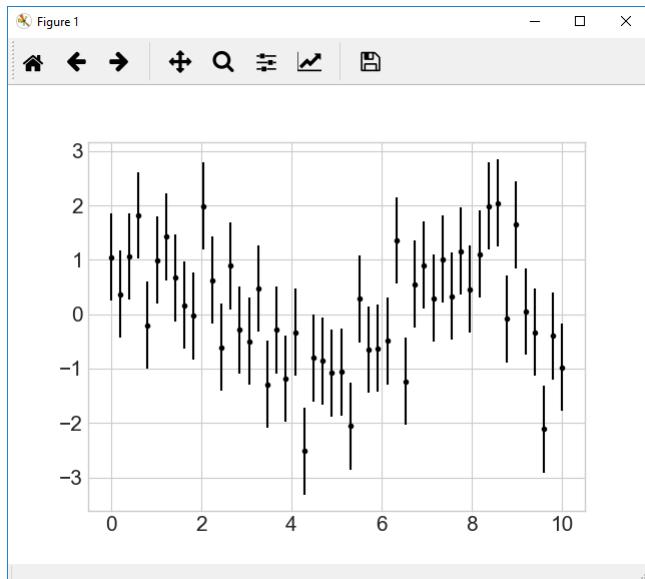
Visualiser l'incertitude

Dans la vie réelle, les données que nous sommes amenés à analyser sont souvent bruitées, c'est-à-dire qu'il existe une part d'incertitude sur leur valeur réelle. Il est extrêmement important d'en tenir compte non seulement lors de l'analyse des données, mais aussi quand on veut les présenter.

Données discrètes

Dans le cas de données discrètes (des points), nous utilisons souvent les barres d'erreur pour représenter, pour chaque point, l'incertitude quant à sa valeur exacte. Souvent la longueur des barres correspond à l'écart type des observations empiriques. C'est chose aisée avec **Matplotlib**.

```
>>> x = np.linspace(0, 10, 50)
>>> dy = 0.8
>>> y = np.sin(x) + dy * np.random.randn(50)
>>> plt.errorbar(x, y, yerr=dy, fmt='.k');
<Container object of 3 artists>
>>> plt.show()
```



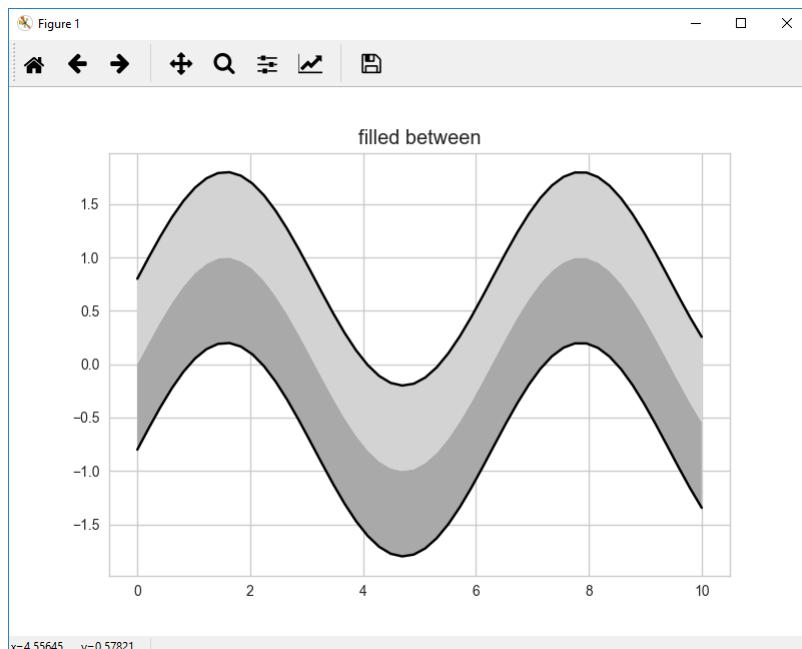
Errorbar prend en argument les abscisses **x**, les coordonnées **y** et les longueurs de chaque barre (une barre par point) **yerr**. Notez l'argument **fmt**. Il permet de choisir, de façon courte, la couleur (ici noir ou black) et la forme des marqueurs du graphe. **Errorbar** permet aussi de personnaliser encore plus l'apparence du graphe.

```
>>> plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
...                 ecolor='lightgray', elinewidth=3, capsize=0);
<Container object of 3 artists>
>>> plt.show()
```

Données continues

Parfois, comme quand on essaie d'appliquer la régression par processus gaussien, nous avons besoin de représenter une incertitude sur une fonction continue. On peut faire ceci en utilisant la fonction **plot** conjointement avec la fonction **fill_between**.

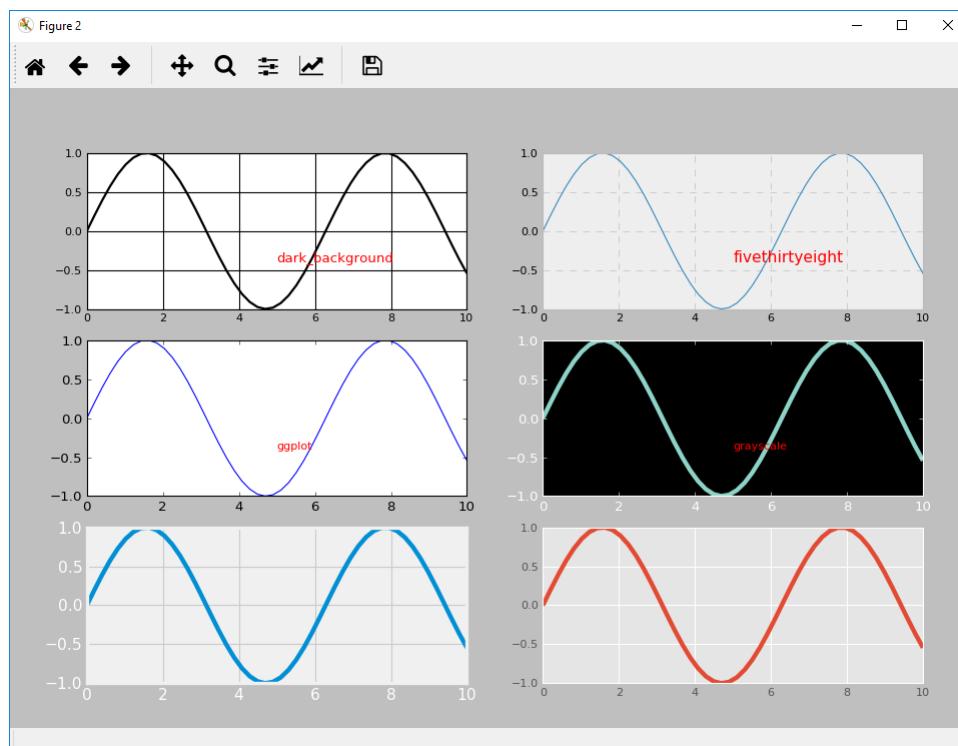
```
>>> x = np.linspace(0, 10, 50)
>>> dy = 0.8
>>> y = np.sin(x)
>>> y1 = y - dy
>>> y2 = y + dy
>>> plt.plot(x, y1, x, y2, color='black')
[<matplotlib.lines.Line2D object at 0x0000022887D49278>,
<matplotlib.lines.Line2D object at 0x0000022887D49BE0>]
>>> plt.fill_between(x, y, y1, where=y>=y1,
... facecolor='darkgray', interpolate=True)
<matplotlib.collections.PolyCollection object at 0x0000022887F24E80>
>>> plt.fill_between(x, y, y2, where=y<=y2,
... facecolor='lightgray', interpolate=True)
<matplotlib.collections.PolyCollection object at 0x0000022887DA2BE0>
>>> plt.title("filled between")
<matplotlib.text.Text object at 0x0000022887E5BF98>
>>> plt.show()
```



La personnalisation et sous-graphes

Matplotlib est très flexible. Quasiment tous les aspects d'une figure peuvent être configurés par l'utilisateur soit pour y ajouter des données, soit pour améliorer l'aspect esthétique. Plutôt que de vous faire une liste des fonctions qui permettent de faire ces actions, j'ai plutôt décidé de vous montrer des exemples. A l'avenir, n'hésitez pas à revenir vers cette partie pour vous remémorer comment réaliser une opération spécifique.

```
>>> print(plt.style.available[:6])
['bmh', 'classic', 'dark_background', 'fivethirtyeight', 'ggplot',
'grayscale']
>>> # Notez la taille de la figure
... fig = plt.figure(figsize=(12,8))
>>> for i in range(6):
...     # On peut ajouter des sous graphes ainsi
...     fig.add_subplot(3,2,i+1)
...     plt.style.use(plt.style.available[i])
...     plt.plot(x, y)
...     # Pour ajouter du texte
...     plt.text(s=plt.style.available[i], x=5, y=2, color='red')
...
>>> plt.show()
```

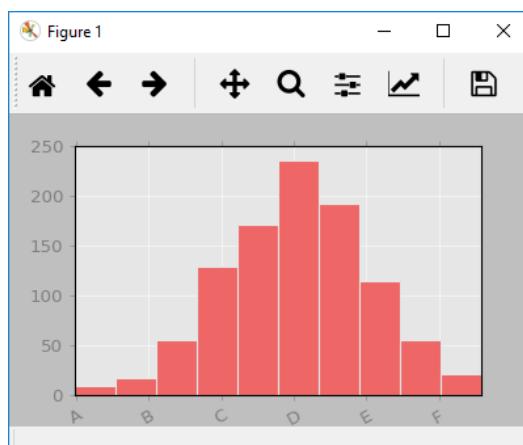


Le premier argument de la fonction **add_subplot** est le nombre de lignes de notre tableau de graphes 3. Le deuxième est le nombre de colonnes 2. Le troisième est le numéro du graphe, parmi les graphes de ce tableau, que nous voulons dessiner.

Pour des raisons historiques, les sous-graphes sont numérotés à partir de 1, au lieu de 0. Le graphe tout en haut à gauche est donc le graphe numéro 1.

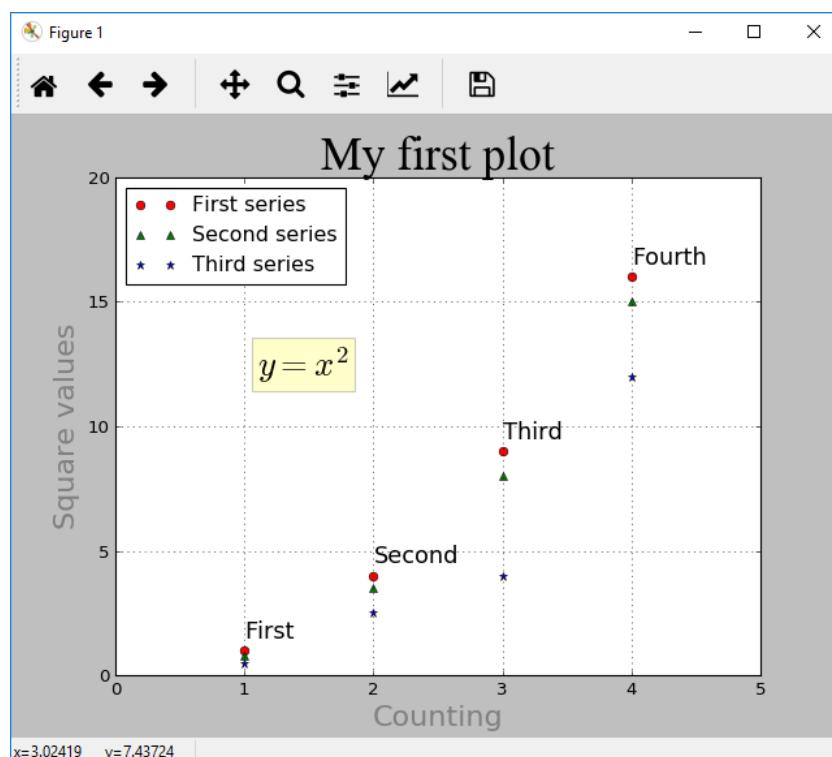
Nous pouvons aussi tout personnaliser à la main.

```
>>> # On peut aussi tout personnaliser à la main
... x = np.random.randn(1000)
>>> plt.style.use('classic')
>>> fig = plt.figure(figsize=(5,3))
>>> ax = plt.axes(facecolor='#E6E6E6')
>>> # Afficher les ticks en dessous de l'axe
>>> ax.set_axisbelow(True)
>>> # Cadre en blanc
>>> plt.grid(color='w', linestyle='solid')
>>> # Cacher le cadre
>>> # ax.spines contient les lignes qui entourent la zone où les
>>> # données sont affichées.
>>> for spine in ax.spines.values():
...     spine.set_visible(False)
...
...
>>> # Cacher les marqueurs en haut et à droite
>>> ax.xaxis.tick_bottom()
>>> ax.yaxis.tick_left()
>>> # Nous pouvons personnaliser les étiquettes des marqueurs
>>> # et leur appliquer une rotation
>>> marqueurs = [-3, -2, -1, 0, 1, 2, 3]
>>> xtick_labels = ['A', 'B', 'C', 'D', 'E', 'F']
>>> plt.xticks(marqueurs, xtick_labels, rotation=30)
>>> # Changer les couleur des marqueurs
>>> ax.tick_params(colors='gray', direction='out')
>>> for tick in ax.get_xticklabels():
...     tick.set_color('gray')
...
...
>>> for tick in ax.get_yticklabels():
...     tick.set_color('gray')
...
...
>>> # Changer les couleur des barres
>>> ax.hist(x, edgecolor='#E6E6E6', color='#EE6666');
>>> plt.show()
```

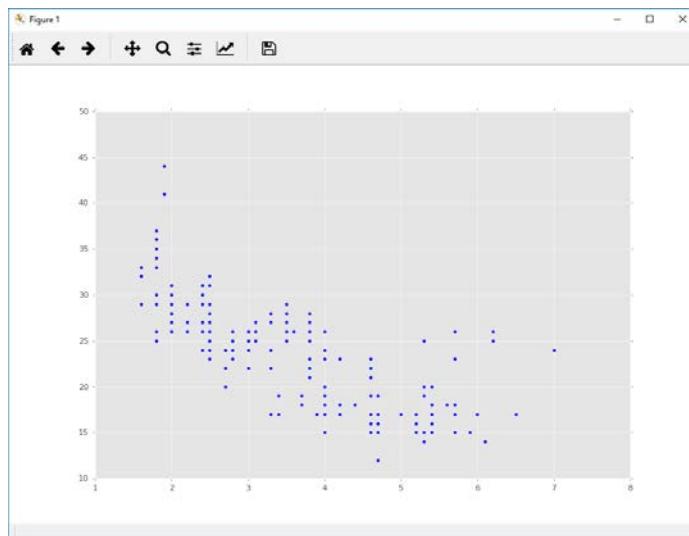


```
>>> plt.axis([0,5,0,20])
[0, 5, 0, 20]
>>> plt.title('My first plot', fontsize=32, fontname='Times New
Roman')
```

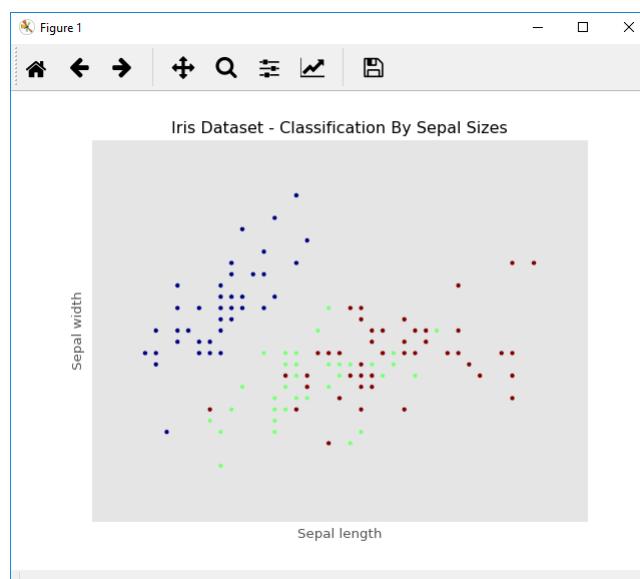
```
>>> plt.xlabel('Counting',color='gray',fontsize=20)
>>> plt.ylabel('Square values',color='gray',fontsize=20)
>>> plt.text(1,1.5,'First',fontsize=16)
>>> plt.text(2,4.5,'Second',fontsize=16)
>>> plt.text(3,9.5,'Third',fontsize=16)
>>> plt.text(4,16.5,'Fourth',fontsize=16)
>>> plt.text(1.1,12,r'$y ='
x^2$',fontsize=24,bbox={'facecolor':'yellow','alpha':0.2})
>>> plt.grid(True)
>>> plt.plot([1,2,3,4],[1,4,9,16],'ro')
>>> plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
>>> plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
>>> plt.legend(['First series','Second series','Third
series'],loc=2)
>>> plt.show()
```



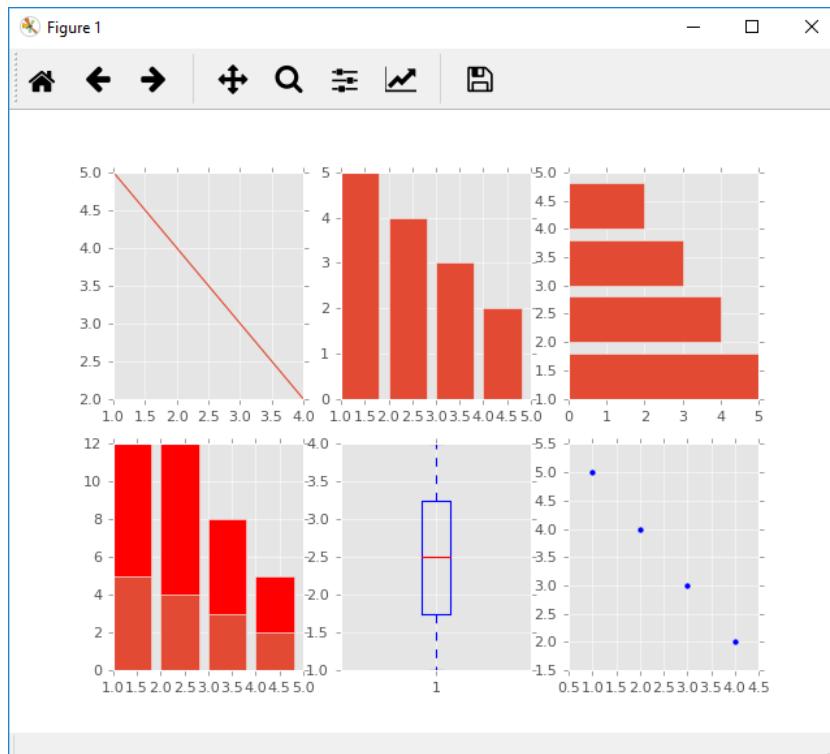
```
>>> df = pd.read_csv('donnees/mpg.csv')
>>> df.head()
   manufacturer model  displ  year  cyl      trans  drv  cty  hwy  fl  class
1          audi    a4    1.8  1999    4  auto(15)   f    18   29  p  compact
2          audi    a4    1.8  1999    4  manual(m5)   f    21   29  p  compact
3          audi    a4    2.0  2008    4  manual(m6)   f    20   31  p  compact
4          audi    a4    2.0  2008    4  auto(av)    f    21   30  p  compact
5          audi    a4    2.8  1999    6  auto(15)   f    16   26  p  compact
>>> plt.style.use('ggplot')
>>> plt.figure(figsize=(12, 8))
<matplotlib.figure.Figure object at 0x0000022889C647F0>
>>> plt.scatter(df.displ,df.hwy)
<matplotlib.collections.PathCollection object at 0x0000022887D68D30>
>>> plt.show()
```



```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> x = iris.data[:,0] #X-Axis - sepal length
>>> y = iris.data[:,1] #Y-Axis - sepal length
>>> species = iris.target #Species
>>> x_min, x_max = x.min() - .5,x.max() + .5
>>> y_min, y_max = y.min() - .5,y.max() + .5
>>> plt.figure()
>>> plt.title('Iris Dataset - Classification By Sepal Sizes')
>>> plt.scatter(x,y, c=species)
>>> plt.xlabel('Sepal length')
>>> plt.ylabel('Sepal width')
>>> plt.xlim(x_min, x_max)
(3.79999999999998, 8.400000000000004)
>>> plt.ylim(y_min, y_max)
(1.5, 4.900000000000004)
>>> plt.xticks(())
([], <a list of 0 Text xticklabel objects>)
>>> plt.yticks(())
([], <a list of 0 Text yticklabel objects>)
>>> plt.show()
```



```
>>> from matplotlib.pyplot import *
>>> x = [1,2,3,4]
>>> y = [5,4,3,2]
>>> figure()
>>> subplot(231)
>>> plot(x, y)
>>> subplot(232)
>>> bar(x, y)
>>> subplot(233)
>>> barh(x, y)
>>> subplot(234)
>>> bar(x, y)
>>> y1 = [7,8,5,3]
>>> bar(x, y1, bottom=y, color = 'r')
>>> subplot(235)
>>> boxplot(x)
>>> subplot(236)
>>> scatter(x,y)
>>> show()
```

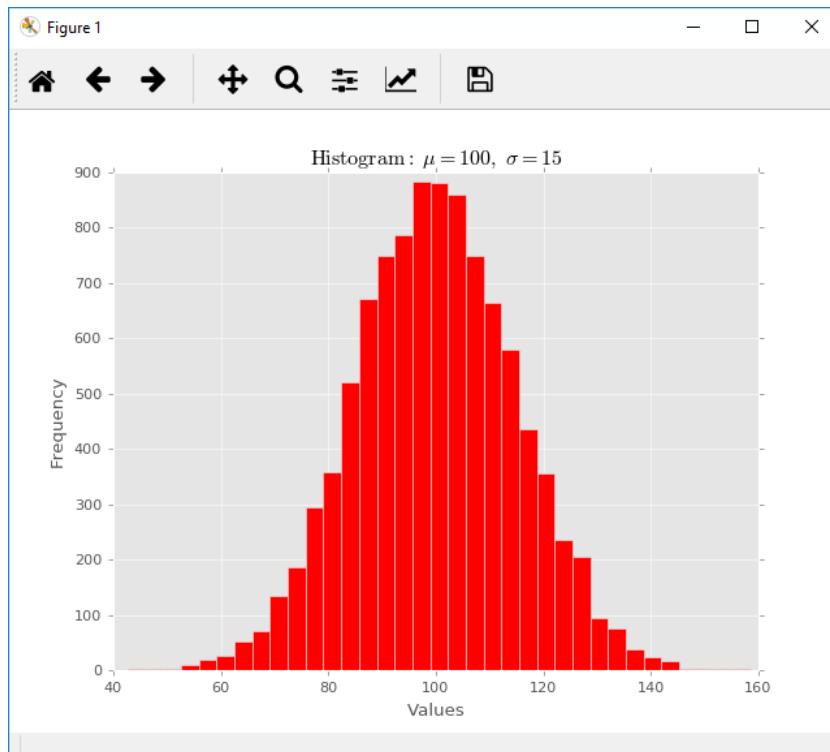


```
>>> mu = 100
>>> sigma = 15
>>> x = np.random.normal(mu, sigma, 10000)
>>> ax = plt.gca()
>>> ax.hist(x, bins=35, color='r')
(array([ 2.,  2.,  3.,  10.,  19.,  27.,  53.,  71.,  134.,
        187.,  295.,  358.,  520.,  672.,  748.,  786.,  884.,  881.,
        860.,  749.,  664.,  580.,  437.,  357.,  237.,  205.,  94.,
        75.,  38.,  24.,  16.,  2.,  3.,  3.,  4.]), array([
42.68588348, 45.99367319, 49.3014629 , 52.6092526 ,
55.91704231, 59.22483202, 62.53262173, 65.84041143,
69.14820114, 72.45599085, 75.76378056, 79.07157027,
82.37935997, 85.68714968, 88.99493939, 92.3027291 ,
```

```

95.6105188 , 98.91830851, 102.22609822, 105.53388793,
108.84167764, 112.14946734, 115.45725705, 118.76504676,
122.07283647, 125.38062617, 128.68841588, 131.99620559,
135.3039953 , 138.611785 , 141.91957471, 145.22736442,
148.53515413, 151.84294384, 155.15073354, 158.45852325]), <a list
of 35 Patch objects>
>>> ax.set_xlabel('Values')
>>> ax.set_ylabel('Frequency')
>>> ax.set_title(r'$\mathrm{Histogram:} \mu=%d, \sigma=%d$' % (mu,
... sigma))
>>> plt.show()

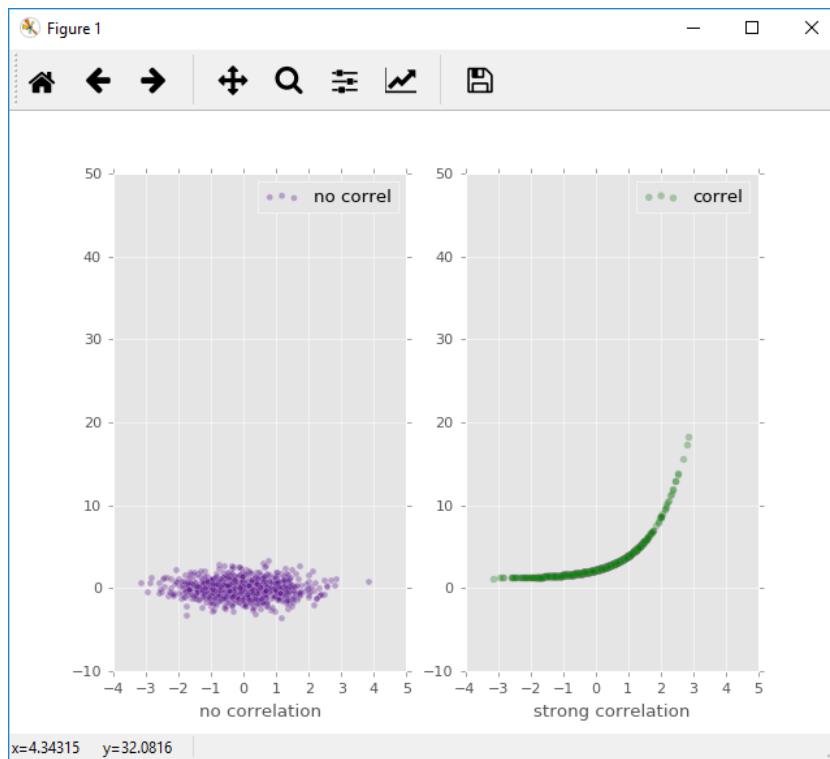
```



```

>>> x = np.random.randn(1000)
>>> y1 = np.random.randn(len(x))
>>> y2 = 1.2 + np.exp(x)
>>> ax1 = plt.subplot(121)
>>> plt.scatter(x, y1, color='indigo', alpha=0.3,
...                 edgecolors='white', label='no correl')
>>> plt.xlabel('no correlation')
>>> plt.grid(True)
>>> plt.legend()
>>> ax2 = plt.subplot(122, sharey=ax1, sharex=ax1)
>>> plt.scatter(x, y2, color='green', alpha=0.3, edgecolors='grey',
...                 label='correl')
>>> plt.xlabel('strong correlation')
>>> plt.grid(True)
>>> plt.legend()
>>> plt.show()

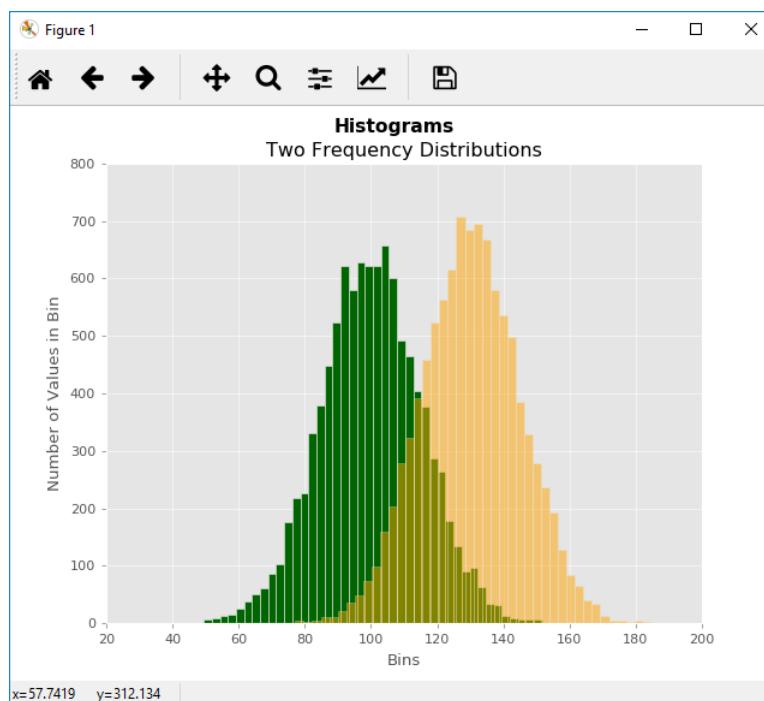
```



```
>>> plt.style.use('ggplot')
>>> customers = ['ABC', 'DEF', 'GHI', 'JKL', 'MNO']
>>> customers_index = range(len(customers))
>>> sale_amounts = [127, 90, 201, 111, 232]
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1,1,1)
>>> ax1.bar(customers_index, sale_amounts, align='center',
color='darkblue')
>>> ax1.xaxis.set_ticks_position('bottom')
>>> ax1.yaxis.set_ticks_position('left')
>>> plt.xticks(customers_index, customers, rotation=0,
fontsize='small')
>>> plt.xlabel('Customer Name')
>>> plt.ylabel('Sale Amount')
>>> plt.title('Sale Amount per Customer')
>>> plt.savefig('bar_plot.png', dpi=400, bbox_inches='tight')
>>> plt.show()
```

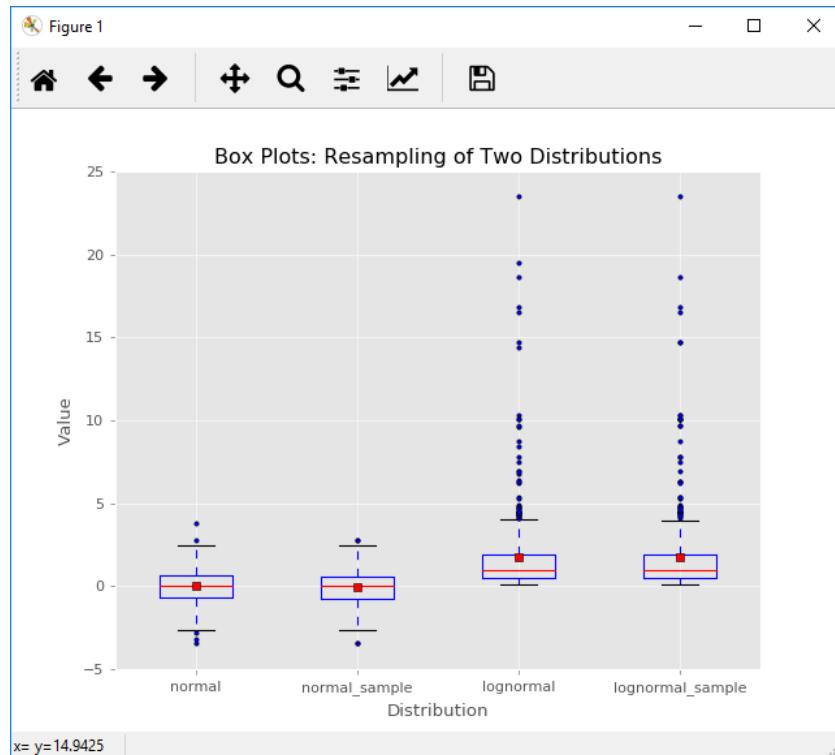


```
>>> plt.style.use('ggplot')
>>> mu1, mu2, sigma = 100, 130, 15
>>> x1 = mu1 + sigma*np.random.randn(10000)
>>> x2 = mu2 + sigma*np.random.randn(10000)
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1,1,1)
>>> n, bins, patches = ax1.hist(x1, bins=50, normed=False,
color='darkgreen')
>>> n, bins, patches = ax1.hist(x2, bins=50, normed=False,
color='orange', alpha=0.5)
>>> ax1.xaxis.set_ticks_position('bottom')
>>> ax1.yaxis.set_ticks_position('left')
>>> plt.xlabel('Bins')
>>> plt.ylabel('Number of Values in Bin')
>>> fig.suptitle('Histograms', fontsize=14, fontweight='bold')
>>> ax1.set_title('Two Frequency Distributions')
>>> plt.savefig('histogram.png', dpi=400, bbox_inches='tight')
>>> plt.show()
```



```
>>> N = 500
>>> normal = np.random.normal(loc=0.0, scale=1.0, size=N)
>>> lognormal = np.random.lognormal(mean=0.0, sigma=1.0, size=N)
>>> index_value = np.random.randint(low=0, high=N-1, size=N)
>>> normal_sample = normal[index_value]
>>> lognormal_sample = lognormal[index_value]
>>> box_plot_data =
[normal,normal_sample,lognormal,lognormal_sample]
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1,1,1)
>>> box_labels =
['normal','normal_sample','lognormal','lognormal_sample']
>>> ax1.boxplot(box_plot_data, notch=False, sym='.', vert=True, \
... whis=1.5, showmeans=True, labels=box_labels)
>>> ax1.xaxis.set_ticks_position('bottom')
```

```
>>> ax1.yaxis.set_ticks_position('left')
>>> ax1.set_title('Box Plots: Resampling of Two Distributions')
>>> ax1.set_xlabel('Distribution')
>>> ax1.set_ylabel('Value')
>>> plt.savefig('box_plot.png', dpi=400, bbox_inches='tight')
>>> plt.show()
```



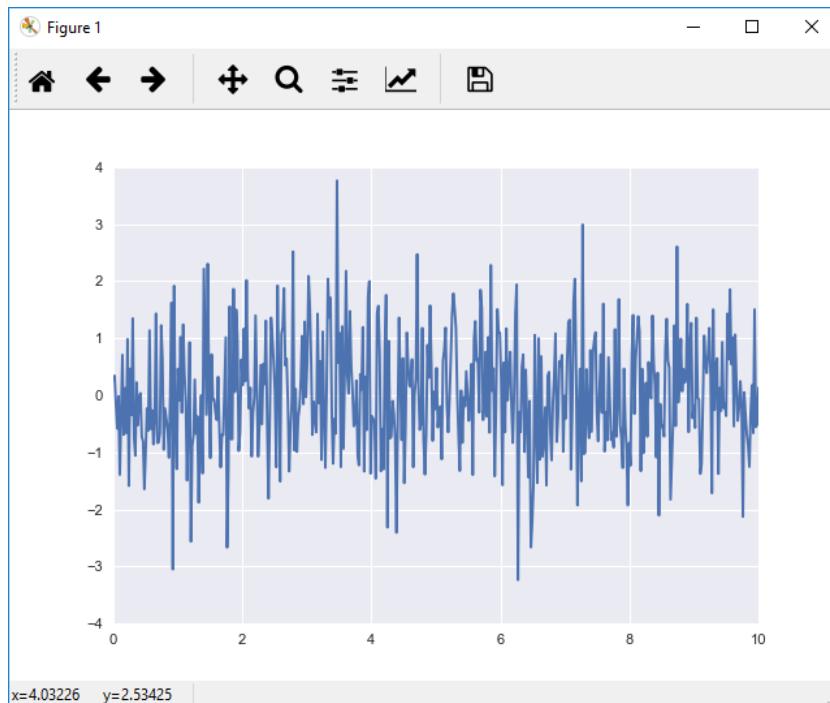
La librairie Seaborn

Seaborn est une librairie qui vient s'ajouter à **Matplotlib**, remplace certains réglages par défaut et fonction, et lui ajoute de nouvelles fonctionnalités. **Seaborn** vient corriger trois défauts de **Matplotlib** :

- **Matplotlib**, surtout dans les versions avant la 2.0, ne génère pas des graphiques d'une grande qualité esthétique.
- **Matplotlib** ne possède pas de fonctions permettant de créer facilement des analyses statistiques sophistiquées.
- Les fonctions de **Matplotlib** ne sont pas faites pour interagir avec les **Dataframes** de **Panda**.

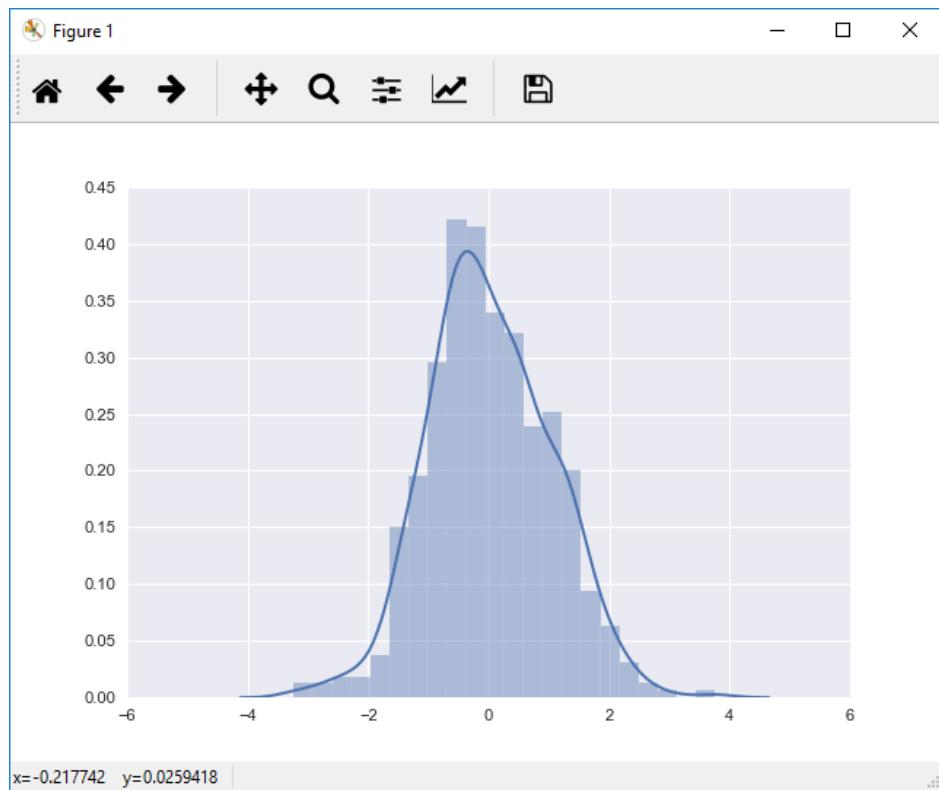
Seaborn fournit une interface qui permet de palier ces problèmes. Il utilise toujours **Matplotlib**, mais le fait en exposant des fonctions plus intuitives. Pour commencer à l'utiliser, rien de plus simple.

```
>>> import seaborn as sns
>>> sns.set()
>>> x = np.linspace(0, 10, 500)
>>> y = np.random.randn(500)
>>> plt.plot(x,y)
>>> plt.show()
```

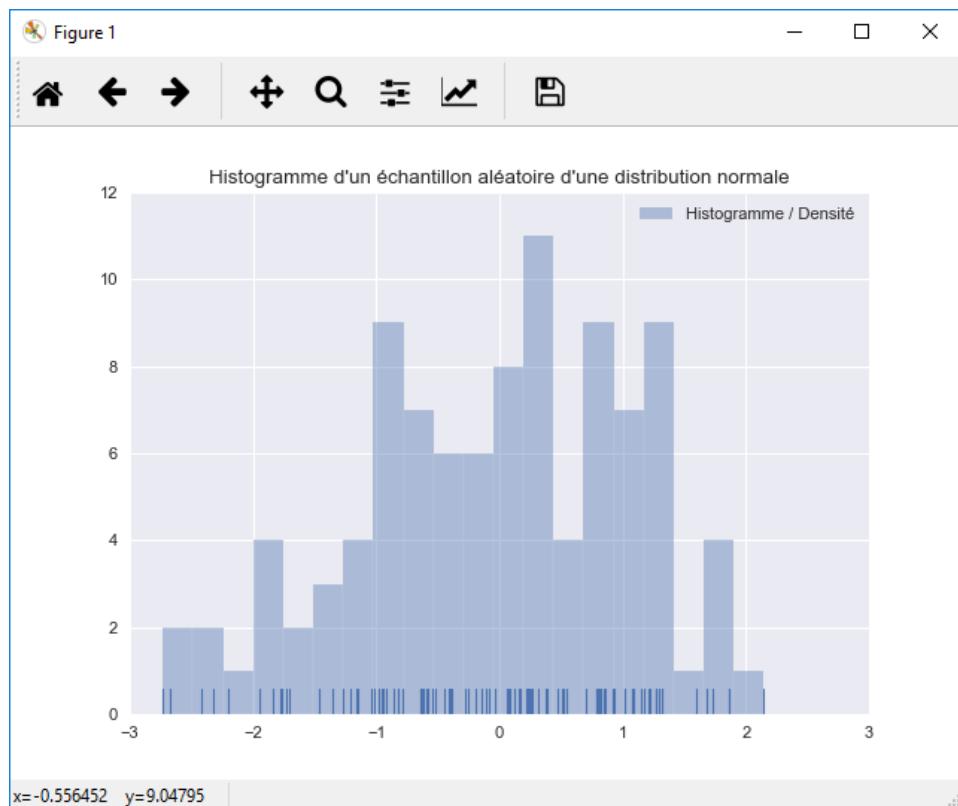


Seaborn nous fournit aussi des fonctions pour des graphiques utiles pour l'analyse statistique. Par exemple, la fonction **distplot** permet non seulement de visualiser l'histogramme d'un échantillon, mais aussi d'estimer la distribution dont l'échantillon est issu.

```
>>> sns.distplot(y, kde=True);
>>> plt.show()
```



```
>>> sns.set(color_codes=True)
>>> x = np.random.normal(size=100)
>>> sns.distplot(x, bins=20, kde=False, rug=True, label="Histogramme / Densité")
>>> plt.title("Histogramme d'un échantillon aléatoire d'une distribution normale")
>>> plt.legend()
>>> plt.show()
```



Imaginons que nous voulons travailler sur un ensemble de données provenant du jeu de données « **Iris** », qui contient des mesures de la longueur et la largeur des sépales et des pétales de trois espèces d'iris.

```
>>> iris = pd.read_csv('donnees/Iris.csv')
>>> sns.pairplot(iris)
>>> sns.pairplot(iris, hue='Species', size=2.5);
>>> plt.show()
```

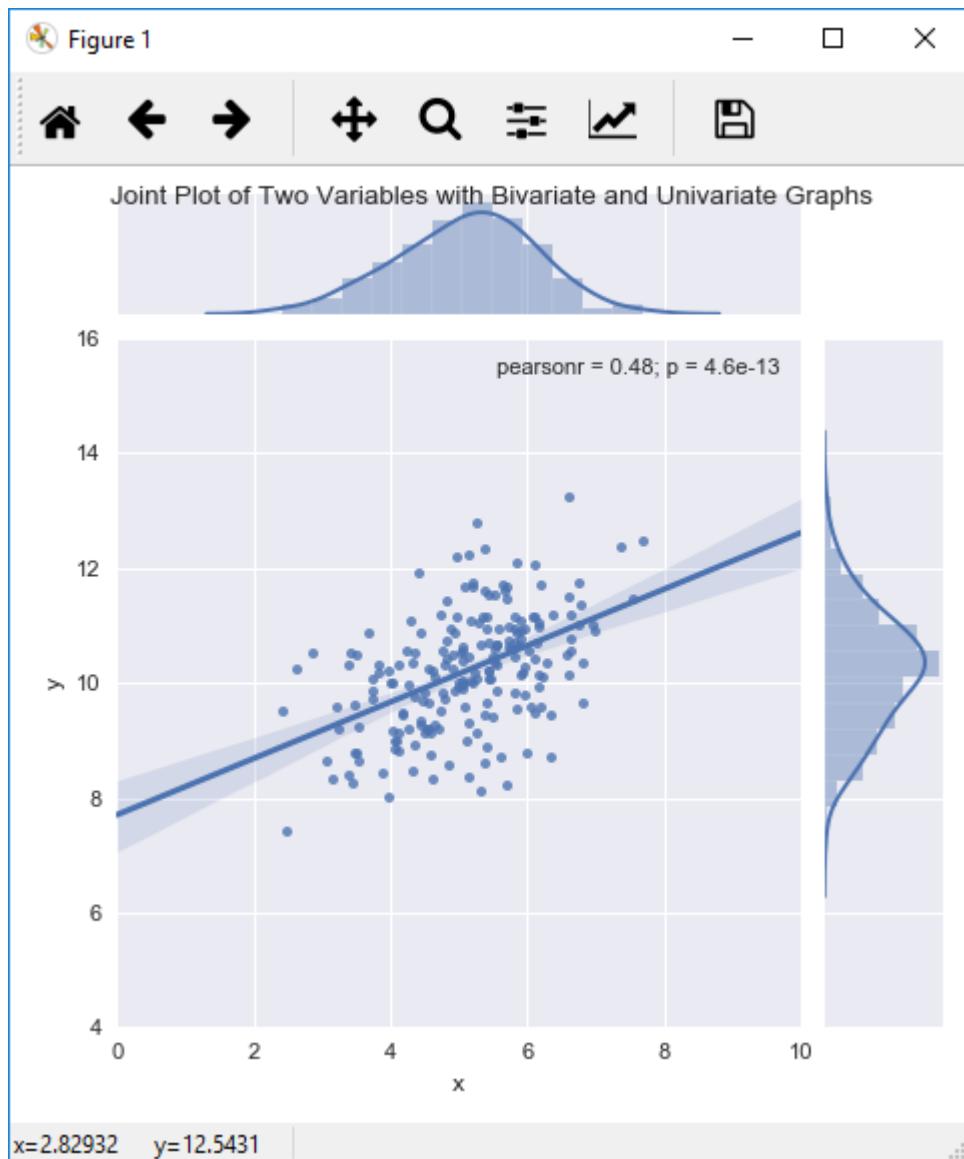


La diagonale est traitée différemment, car tracer une variable en fonction d'elle-même n'aurait aucun intérêt. À la place, **sns.pairplot** trace un histogramme des données en fonction de la variable en question pour chaque classe de données.

Nous pouvons aussi voir la distribution jointe de deux caractéristiques :

```
>>> mean, cov = [5, 10], [(1, .5), (.5, 1)]
>>> data = np.random.multivariate_normal(mean, cov, 200)
>>> data_frame = pd.DataFrame(data, columns=["x", "y"])
>>> sns.jointplot(x="x", y="y", data=data_frame, kind="reg")\
...     .set_axis_labels("x", "y")
>>> plt.suptitle("Joint Plot of Two Variables with Bivariate and Univariate Graphs")
```

```
>>> plt.show()
```



```
>>> mpg = pd.read_csv('donnees/mpg.csv')
>>> sns.factorplot(x="year", y="hwy", \
...                  col="class", data=mpg, kind="box", size=4, aspect=.5)
>>> plt.show()
```



4

Les *DataFrames*

Les structures de données

La librairie **Pandas** fournit deux structures de données fondamentales, la « **Series** » et le « **DataFrame** ». On peut voir ces structures comme une généralisation des tableaux et des matrices de **Numpy**. La différence fondamentale entre ces structures et les versions de **Numpy** est que les objets **Pandas** possèdent des indices explicites. Là où on ne pouvait se référer à un élément d'un tableau **Numpy** que par sa position dans le tableau, chaque élément d'une « **Series** » ou d'un « **DataFrame** » peut avoir un indice explicitement désigné par l'utilisateur.

L'indice explicite est optionnel. On peut très bien utiliser une « **Series** » par exemple comme on utiliserait un tableau « **Numpy** », en se contentant des indices générés automatiquement en fonction de la position de chaque élément.

Commençons par voir comment créer ces structures et nous en servir pour quelques opérations de base.

```
>>> import numpy as np
>>> import pandas as pd
>>> # On peut créer une Series à partir d'une list
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0])
>>> print("data ressemble à un tableau Numpy: ", data)
data ressemble à un tableau Numpy:  0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
>>> # On peut spécifier des indices à la main
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0],
...                  index=['a', 'b', 'c', 'd'])
>>> print("data ressemble à un dict en Python: ", data)
data ressemble à un dict en Python:  a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
>>> print(data['b'])
0.5
>>> # On peut même créer une Serie directement à partir d'une dict
... population_dict = {'California': 38332521,
...                     'Texas': 26448193,
...                     'New York': 19651127,
...                     'Florida': 19552860,
...                     'Illinois': 12882135}
>>> area_dict = {'California': 423967,
...                 'Texas': 695662,
...                 'New York': 141297,
...                 'Florida': 170312,
...                 'Illinois': 149995}
>>> population = pd.Series(population_dict)
>>> area = pd.Series(area_dict)
>>> print(population)
California    38332521
```

```

Florida      19552860
Illinois     12882135
New York     19651127
Texas        26448193
dtype: int64
>>> # Que pensez vous de cette ligne?
>>> print(population['California':'Florida'])
California   38332521
Florida      19552860
dtype: int64

```

De la même façon que les opérations sur les tableaux **Numpy** sont plus rapides que celles sur les « **list** » en Python, les opérations sur les « **Series** » sont plus rapides que celles sur les « **dict** ».

Les « **DataFrame** » permettent de combiner plusieurs « **Series** » en colonnes, un peu comme dans un tableau SQL.

```

>>> # A partir d'une Series
>>> df = pd.DataFrame(population, columns=['population'])
>>> print(df)
           population
California   38332521
Florida      19552860
Illinois     12882135
New York     19651127
Texas        26448193
>>> # A partir d'une list de dict
>>> data = [{ 'a': i, 'b': 2 * i}
...          for i in range(3)]
>>> df = pd.DataFrame(data)
>>> print(df)
   a   b
0  0   0
1  1   2
2  2   4
>>> # A partir de plusieurs Series
>>> df = pd.DataFrame({'population': population, 'area': area})
>>> print(df)
           area  population
California  423967      38332521
Florida     170312      19552860
Illinois    149995      12882135
New York    141297      19651127
Texas       695662      26448193
>>> # A partir d'un tableau Numpy de dimension 2
>>> df = pd.DataFrame(np.random.rand(3, 2),
...                      columns=['foo', 'bar'],
...                      index=[ 'a', 'b', 'c'])
>>> print(df)
         foo      bar
a  0.379015  0.789917
b  0.713045  0.660162
c  0.527456  0.634284
>>> # Une fonction pour générer facilement des DataFrame.
>>> # Elle nous sera utile dans la suite de ce chapitre.

```

```
>>> def make_df(cols, ind):
...     """Crée rapidement des DataFrame"""
...     data = {c: [str(c) + str(i) for i in ind]
...             for c in cols}
...     return pd.DataFrame(data, ind)
...
>>> # exemple
>>> make_df('ABC', range(3))
   A    B    C
0  A0  B0  C0
1  A1  B1  C1
2  A2  B2  C2

>>> raw_data = {'first_name': ['Jason', 'Molly', 'Tina',
...                             'Jake', 'Amy'],
...               'last_name': ['Miller', 'Jacobson', ".",
...                            'Milner', 'Cooze'],
...               'age': [42, 52, 36, 24, 73],
...               'preTestScore': [4, 24, 31, ".", "."],
...               'postTestScore': ["25,000", "94,000", 57, 62, 70]}
>>> df = pd.DataFrame(raw_data, columns = ['first_name',
...                                           'last_name','age','preTestScore','postTestScore'])
>>> df
   first_name last_name  age preTestScore postTestScore
0      Jason    Miller    42           4       25,000
1      Molly  Jacobson    52          24       94,000
2       Tina        .     36          31          57
3       Jake    Milner    24          ..          62
4       Amy     Cooze    73          ..          70
```

Lecture et écriture de DataFrame

Aujourd’hui, on n’a plus besoin de réécrire soi-même une fonction de lecture ou d’écriture de données présentées sous forme de tables. Il existe des fonctions plus génériques qui gère un grand nombre de cas.

```
>>> import pandas as pd
>>> l = [ { "date":"2017-06-22", "prix":220.0, "devise":"euros" },
...           { "date":"2017-06-23", "prix":221.0, "devise":"euros" }, ]
>>> df = pd.DataFrame(l)
>>> # écriture au format texte
>>> df.to_csv("donnees/exemple.txt",sep="\t",
...             encoding="utf-8", index=False)
>>> # on regarde ce qui a été enregistré
... with open("donnees/exemple.txt", "r", encoding="utf-8") as f:
...     text = f.read()
...
>>> print(text)
date    devise   prix
2017-06-22    euros  220.0
2017-06-23    euros  221.0

>>> # on enregistre au format Excel
>>> df.to_excel("donnees/exemple.xlsx", index=False)
```

La librairie **Pandas** fournit un ensemble de fonctions de lecture écriture de haut niveau pour accéder aux fichiers. Le fonctions de lecture renvoient généralement un objet DataFrame.

Format Type	Data Description	Reader	Writer
text	CSV	<code>read_csv</code>	<code>to_csv</code>
text	JSON	<code>read_json</code>	<code>to_json</code>
text	HTML	<code>read_html</code>	<code>to_html</code>
text	Local clipboard	<code>read_clipboard</code>	<code>to_clipboard</code>
binary	MS Excel	<code>read_excel</code>	<code>to_excel</code>
binary	HDF5 Format	<code>read_hdf</code>	<code>to_hdf</code>
binary	Feather Format	<code>read_feather</code>	<code>to_feather</code>
binary	Parquet Format	<code>read_parquet</code>	<code>to_parquet</code>
binary	Msgpack	<code>read_msgpack</code>	<code>to_msgpack</code>
binary	Stata	<code>read_stata</code>	<code>to_stata</code>
binary	SAS	<code>read_sas</code>	
binary	Python Pickle Format	<code>read_pickle</code>	<code>to_pickle</code>
SQL	SQL	<code>read_sql</code>	<code>to_sql</code>
SQL	Google Big Query	<code>read_gbq</code>	<code>to_gbq</code>

```

>>> import pandas as pd
>>> villes = pd.read_csv("donnees/temperatures.csv",
...                      index_col='Ville')
>>> villes.head()
      Janv Févr Mars Avri Mai Juin Juil Août Sept \
Ville
Abbeville    4.27  4.96  7.27  9.94 13.17 15.89 17.73 17.95 15.47
Lille-Lesquin 3.80  4.73  7.40 10.55 14.09 16.95 18.77 18.74 15.90
Pte De La Hague 7.89  7.66  8.54 10.03 12.38 14.79 16.72 17.48 16.63
Caen-Carpiquet 5.38  5.94  7.80 10.02 13.21 16.19 17.94 18.18 15.85
Rouen-Boos     3.95  4.66  7.28  9.99 13.38 16.38 18.11 18.11 15.33

      Octo Nove Déce   Lat   Long Alt Moy Amp \
Ville
Abbeville    11.90  7.80  4.92 50.136000 1.834000 69 10.94 13.68
Lille-Lesquin 11.81  7.53  4.52 50.570000 3.097500 47 11.23 14.97
Pte De La Hague 14.38 11.31  8.96 49.725167 -1.939833 6 12.23 9.82
Caen-Carpiquet 12.59  8.75  5.96 49.180000 -0.456167 67 11.48 12.80
Rouen-Boos     11.60  7.40  4.46 49.383000 1.181667 151 10.89 14.16

      Zone
Ville
Abbeville    NO
Lille-Lesquin NE
Pte De La Hague NO
Caen-Carpiquet NO
Rouen-Boos     NO

>>> villes.describe()
      Janv Févr Mars Avri Mai Juin \
count 42.000000 42.000000 42.000000 42.000000 42.000000 42.000000 \
mean  5.335714 5.959762 8.814524 11.577381 15.236190 18.755952
std   2.286357 2.015944 1.705363 1.614030 1.781203 2.179185
min   1.110000 1.610000 4.980000 7.970000 12.100000 14.790000
25%  3.837500 4.662500 7.425000 10.282500 14.107500 17.270000
50%  5.210000 5.860000 8.550000 11.225000 14.955000 18.525000
75%  7.397500 7.607500 10.167500 12.790000 16.567500 20.140000
max  9.270000 9.510000 12.010000 14.680000 18.880000 23.150000

      Juil Août Sept Octo Nove Déce \
count 42.000000 42.000000 42.000000 42.000000 42.000000 42.000000 \
mean 20.566905 20.473571 17.319286 13.578095 8.947381 5.977857
std  2.342368 2.248508 1.991061 2.079219 2.215485 2.279859
min 16.720000 16.990000 13.520000 9.890000 4.900000 1.830000
25% 19.172500 18.907500 15.905000 11.877500 7.545000 4.555000
50% 20.155000 20.115000 16.740000 13.075000 8.380000 5.530000
75% 21.607500 21.640000 18.437500 14.737500 10.535000 8.015000
max 25.320000 25.000000 21.350000 17.740000 13.350000 10.220000

      Lat Long Alt Moy Amp
count 42.000000 42.000000 42.000000 42.000000 42.000000
mean 46.251996 2.421921 174.476190 12.711667 15.316429
std  2.450608 3.419851 211.239459 1.836500 2.406873
min  41.918000 -4.412000 2.000000 9.120000 9.530000
25% 43.962000 0.027542 43.250000 11.440000 14.420000
50% 46.320333 2.372083 101.500000 12.205000 15.630000
75% 48.445167 4.991625 231.000000 13.715000 16.665000
max 50.570000 9.485167 871.000000 16.470000 19.180000

>>> villes.axes
[Index(['Abbeville', 'Lille-Lesquin', 'Pte De La Hague', 'Caen-Carpiquet',
       'Rouen-Boos', 'Reims-Prunay', 'Brest-Guipavas', 'Ploumanac'h',
       'Rennes-St Jacques', 'Alencon', 'Orly', 'Troyes-Barberey',
       'Nancy-Ochey', 'Strasbourg-Entzheim', 'Belle Ile-Le Talut',
       'Nantes-Bouguenais', 'Tours', 'Bourges', 'Dijon-Longvic',
       'Bale-Mulhouse', 'Pte De Chassiron', 'Poitiers-Biard'],
      dtype='object')]

```

```
'Limoges-Bellegarde', 'Clermont-Fd', 'Le Puy-Loudes', 'Lyon-St Exupery',
'Bordeaux-Merignac', 'Gourdon', 'Millau', 'Montelimar', 'Embrun',
'Mont-De-Marsan', 'Tarbes-Ossun', 'St Girons', 'Toulouse-Blagnac',
'Montpellier', 'Marignane', 'Cap Cépet', 'Nice', 'Perpignan', 'Ajaccio',
'Bastia'],
dtype='object', name='Ville'), Index(['Janv', 'Févr', 'Mars', 'Avri', 'Mai',
'Juin', 'Juil', 'Août', 'Sept',
'Octo', 'Nove', 'Déce', 'Lat', 'Long', 'Alt', 'Moy', 'Amp', 'Zone'],
dtype='object')]
>>> villes.dtypes
Janv      float64
Févr     float64
Mars     float64
Avri     float64
Mai      float64
Juin     float64
Juil     float64
Août     float64
Sept     float64
Octo     float64
Nove     float64
Déce     float64
Lat      float64
Long     float64
Alt      int64
Moy      float64
Amp      float64
Zone    object
dtype: object
```

Il est possible de sélectionner les colonnes qui doivent être chargées et changer les noms par défaut.

```
>>> commandes = pd.read_csv("donnees/stagiaire/commandes.csv",
...                 sep=';', header=0, usecols=[0,1,2,3,4,5],
...                 names=['No','Client','Employe','Commande','Envoi','Port'])
>>> commandes.head()
      No Client   Employe   Commande       Envoi  Port
0  215650  LONEP      84  2010-02-02  2010-03-08  50.1
1  215653  PERIC      78  2010-02-02  2010-03-14  97.6
2  215652  BOTTM      72  2010-02-02  2010-03-02  89.3
3  215674  SPECD     111  2010-02-02  2010-03-01  86.2
4  215672  WELLI      39  2010-02-02  2010-02-12  71.9
```

La projection et la restriction

L'algèbre relationnelle est une théorie permettant de manipuler des données disposées sous forme de tableau ; et ça tombe bien : un « **DataFrame** », c'est justement un tableau !

On peut référencer aux éléments des objets **Pandas** en utilisant soit leurs index implicites, de la même façon que les tableaux **Numpy**, soit les index explicites comme dans les « **dict** ». Pour éviter toute confusion, il est conseillé d'utiliser les attributs « **loc** » qui référence par l'index et « **iloc** » qui référence par la position de chaque objet.

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0],
...                   index=['a', 'b', 'c', 'd'])
>>> print(data)
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
>>> # On peut désigner un élément d'une Series par son index
... print(data.loc['b'])
0.5
>>> # Ou bien par sa position
... print(data.iloc[1])
0.5
```

La différence entre les deux devrait être claire après avoir exécuté ces lignes. Effectuer ces mêmes opérations sur les **DataFrame** se fait de manière analogue.

```
>>> data = pd.DataFrame({'area':area, 'pop':population})
>>> print(data)
           area      pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
New York    141297  19651127
Texas       695662  26448193
>>> data.loc[:'Illinois', :'pop']
           area      pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
```

Il est possible d'accéder à une colonne a l'aide de plusieurs syntaxes.

```
>>> villes.head(3)
           Janv   Févr   Mars   Avri   Mai   Juin   Juil   Août   Sept   \
Ville
Abbeville    4.27   4.96   7.27   9.94  13.17  15.89  17.73  17.95  15.47
Lille-Lesquin  3.80   4.73   7.40  10.55  14.09  16.95  18.77  18.74  15.90
Pte De La Hague  7.89   7.66   8.54  10.03  12.38  14.79  16.72  17.48  16.63
                                         Octo   Nove   Déce   Lat   Long   Alt   Moy   Amp   \
Ville
Abbeville    11.90   7.80   4.92  50.136000  1.834000   69  10.94  13.68
Lille-Lesquin  11.81   7.53   4.52  50.570000  3.097500   47  11.23  14.97
Pte De La Hague  14.38  11.31   8.96  49.725167 -1.939833    6  12.23   9.82
```

```

Zone
Ville
Abbeville      NO
Lille-Lesquin   NE
Pte De La Hague NO
>>> villes.head(3).Janv
Ville
Abbeville      4.27
Lille-Lesquin   3.80
Pte De La Hague 7.89
Name: Janv, dtype: float64
>>> villes.head(3)[ 'Janv' ]
Ville
Abbeville      4.27
Lille-Lesquin   3.80
Pte De La Hague 7.89
Name: Janv, dtype: float64

```

L'objet que renvoie `villes.head(3)['Janv']` est de type `pandas.Series`. Pour obtenir les valeurs de la colonne `Janv` au format `numpy`, il faut saisir `villes.head(3)['Janv'].values`.

Accédons maintenant aux données de la ville Abbeville, d'abord par sa position 0, puis par son nom. Le résultat retourné est exactement le même dans les 2 cas.

```

>>> villes.iloc[0,0:3]
Janv    4.27
Févr    4.96
Mars    7.27
Name: Abbeville, dtype: object
>>> villes.loc[['Abbeville'],
...             [ 'Janv','Févr','Mars']]
          Janv  Févr  Mars
Ville
Abbeville  4.27  4.96  7.27

>>> villes.iloc[0:3,0:3]
          Janv  Févr  Mars
Ville
Abbeville    4.27  4.96  7.27
Lille-Lesquin 3.80  4.73  7.40
Pte De La Hague 7.89  7.66  8.54

```

On désigne généralement une colonne ou variable par son nom. Les lignes peuvent être désignées par un entier.

```

>>> import pandas as pd
>>> villes = pd.read_csv("donnees/temperatures.csv")
>>> villes.head().iloc[:, :5]
          Ville  Janv  Févr  Mars  Avri
0        Abbeville  4.27  4.96  7.27  9.94
1    Lille-Lesquin  3.80  4.73  7.40 10.55
2  Pte De La Hague  7.89  7.66  8.54 10.03
3   Caen-Carpiquet  5.38  5.94  7.80 10.02
4     Rouen-Boos  3.95  4.66  7.28  9.99
>>> villes.iloc[2]
          Ville  Pte De La Hague
Janv                7.89

```

```

Févr           7.66
Mars          8.54
Avri          10.03
Mai           12.38
Juin          14.79
Juil          16.72
Août          17.48
Sept          16.63
Octo          14.38
Nove          11.31
Déce          8.96
Lat            49.7252
Long         -1.93983
Alt             6
Moy            12.23
Amp            9.82
Zone           NO
Name: 2, dtype: object
>>> villes.iloc[1,2]
4.7300000000000004
>>> villes.iloc[:,2]
Ville   Janv
0      Abbeville  4.27
1      Lille-Lesquin  3.80
2  Pte De La Hague  7.89

```

On extrait une valeur en indiquant sa position dans la table avec des entiers

```

>>> villes.head().loc[1,'Janv']
3.7999999999999998
>>> villes.head().iloc[:,3]
Ville   Janv  Févr
0      Abbeville  4.27  4.96
1      Lille-Lesquin  3.80  4.73
2  Pte De La Hague  7.89  7.66
3  Caen-Carpiquet  5.38  5.94
4      Rouen-Boos  3.95  4.66
>>> villes.head().iloc[:,[1,3,5,12]]
Janv  Mars    Mai  Déce
0  4.27  7.27  13.17  4.92
1  3.80  7.40  14.09  4.52
2  7.89  8.54  12.38  8.96
3  5.38  7.80  13.21  5.96
4  3.95  7.28  13.38  4.46
>>> villes.head().loc[:,3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...

```

Avec loc, il faut préciser le nombre de la colonne

```

>>> villes.columns
Index(['Ville', 'Janv', 'Févr', 'Mars', 'Avri', 'Mai', 'Juin', 'Juil', 'Août',
       'Sept', 'Octo', 'Nove', 'Déce', 'Lat', 'Long', 'Alt', 'Moy', 'Amp',
       'Zone'],
      dtype='object')
>>> villes.head().loc[:,['Janv','Mai','Déce']]
Janv    Mai  Déce

```

```

0  4.27  13.17  4.92
1  3.80  14.09  4.52
2  7.89  12.38  8.96
3  5.38  13.21  5.96
4  3.95  13.38  4.46

```

Mais il est possible d'utiliser une colonne ou plusieurs colonnes comme index à l'aide de la fonction **set_index**.

```

>>> villesI = villes.set_index('Ville')
>>> villesI.head()
          Janv  Févr  Mars  Avri  Mai  Juin  Juil  Août  Sept
\\
Ville
Abbeville      4.27  4.96  7.27  9.94  13.17  15.89  17.73  17.95  15.47
Lille-Lesquin   3.80  4.73  7.40  10.55  14.09  16.95  18.77  18.74  15.90
Pte De La Hague 7.89  7.66  8.54  10.03  12.38  14.79  16.72  17.48  16.63
Caen-Carpiquet  5.38  5.94  7.80  10.02  13.21  16.19  17.94  18.18  15.85
Rouen-Boos       3.95  4.66  7.28  9.99  13.38  16.38  18.11  18.11  15.33

          Octo  Nove  Déce    Lat    Long  Alt  Moy  Amp
\\
Ville
Abbeville     11.90  7.80  4.92  50.136000  1.834000  69  10.94  13.68
Lille-Lesquin   11.81  7.53  4.52  50.570000  3.097500  47  11.23  14.97
Pte De La Hague 14.38 11.31  8.96  49.725167 -1.939833  6  12.23  9.82
Caen-Carpiquet 12.59  8.75  5.96  49.180000 -0.456167  67  11.48  12.80
Rouen-Boos       11.60  7.40  4.46  49.383000  1.181667  151  10.89  14.16

Zone
Ville
Abbeville      NO
Lille-Lesquin   NE
Pte De La Hague NO
Caen-Carpiquet NO
Rouen-Boos       NO

>>> villesI.head().iloc[:,[1,3,5,12]]
          Févr  Avri  Juin    Lat
Ville
Abbeville     4.96  9.94  15.89  50.136000
Lille-Lesquin  4.73 10.55  16.95  50.570000
Pte De La Hague 7.66 10.03  14.79  49.725167
Caen-Carpiquet 5.94 10.02  16.19  49.180000
Rouen-Boos       4.66  9.99  16.38  49.383000

>>> villesI.head().iloc[:,['Janv','Mars','Mai','Déce']]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...
>>> villesI.head().loc[:,['Janv','Mars','Mai','Déce']]
          Janv  Mars  Mai  Déce
Ville
Abbeville     4.27  7.27 13.17  4.92
Lille-Lesquin   3.80  7.40 14.09  4.52
Pte De La Hague 7.89  8.54 12.38  8.96
Caen-Carpiquet  5.38  7.80 13.21  5.96
Rouen-Boos       3.95  7.28 13.38  4.46

```

Il est possible d'utiliser plusieurs colonnes comme index

```
>>> commandes = pd.read_csv("donnees/stagiaire/commandes.csv",
```

```

...           sep=';', header=0, usecols=[0,1,2,3,4,5],
...           names=['No', 'Client', 'Employe', 'Commande', 'Envoi', 'Port'])
>>> commandes.head()
   No Client Employe     Commande      Envoi  Port
0  215650  LONEP       84  2010-02-02  2010-03-08  50.1
1  215653  PERIC       78  2010-02-02  2010-03-14  97.6
2  215652  BOTTM       72  2010-02-02  2010-03-02  89.3
3  215674  SPEC'D      111 2010-02-02  2010-03-01  86.2
4  215672  WELL'I      39  2010-02-02  2010-02-12  71.9
>>> commandesI =
commandes.set_index(['Client', 'Employe', 'Commande'])
>>> commandes.dtypes
No          int64
Client      object
Employe    int64
Commande    object
Envoi      object
Port       float64
dtype: object
>>> commandesI.dtypes
No          int64
Envoi      object
Port       float64
dtype: object
>>> commandesI.iloc[1]
No          215653
Envoi      2010-03-14
Port        97.6
Name: (PERIC, 78, 2010-02-02), dtype: object
>>> commandesI.loc["PERIC", 78, "2010-02-02"]
No          215653
Envoi      2010-03-14
Port        97.6
Name: (PERIC, 78, 2010-02-02), dtype: object

```

Si on veut changer l'index ou le supprimer il faut utiliser la fonction « `reset_index` ». Le mot-clé « `drop` » est utilisé pour garder ou non les colonnes servant d'index et « `inplace` » signifie qu'on modifie l'instance et non qu'une copie est modifiée.

```

>>> commandesI.reset_index(drop=False, inplace=True)
>>> commandesI.set_index(['No'], inplace=True)
>>> commandesI.dtypes
Client      object
Employe    int64
Commande    object
Envoi      object
Port       float64
dtype: object
>>> commandesI.head(3)
   Client Employe     Commande      Envoi  Port
No
215650  LONEP       84  2010-02-02  2010-03-08  50.1
215653  PERIC       78  2010-02-02  2010-03-14  97.6
215652  BOTTM       72  2010-02-02  2010-03-02  89.3

```

Les index sont particulièrement utiles lorsqu'il s'agit de fusionner deux tables. Pour des petites tables, la plupart du temps, il est plus facile de s'en passer.

La restriction

Filter consiste à sélectionner un sous-ensemble de lignes du **DataFrame**. Pour filter sur plusieurs conditions, il faut utiliser les opérateurs logique & (et), | (ou), ~ (non).

```
>>> villes[villes.Janv > 7].Janv
Ville
Pte De La Hague      7.89
Brest-Guipavas       7.09
Ploumanac'h          7.79
Belle Ile-Le Talut   8.16
Pte De Chassiron     7.69
Montpellier          7.50
Marignane            7.54
Cap Cepet             9.08
Nice                  8.81
Perpignan            8.78
Ajaccio                9.14
Bastia                 9.27
Name: Janv, dtype: float64
>>> villes[(villes.Janv > 7) &
...           (villes.Alt > 50)].loc[:,,
...           ['Janv','Lat','Long','Alt']]
              Janv      Lat      Long   Alt
Ville
Brest-Guipavas    7.09  48.444167 -4.412000   94
Ploumanac'h       7.79  48.825833 -3.473167   55
Cap Cepet          9.08  43.079333  5.940833  115
```

Les dernières versions de pandas ont introduit la méthode query qui permet de réduire encore l'écriture.

```
>>> villes.query('(Janv > 7) & (Alt > 50)').loc[:,,
...           ['Janv','Lat','Long','Alt']]
              Janv      Lat      Long   Alt
Ville
Brest-Guipavas    7.09  48.444167 -4.412000   94
Ploumanac'h       7.79  48.825833 -3.473167   55
Cap Cepet          9.08  43.079333  5.940833  115

>>> villes.query('((Janv < 5) & (Moy > 10 ) & (Amp < 15)) | (Alt >
500)').loc[:,,
...           ['Janv','Moy','Amp','Alt']]
              Janv      Moy      Amp   Alt
Ville
Abbeville        4.27  10.94  13.68   69
Lille-Lesquin     3.80  11.23  14.97   47
Rouen-Boos        3.95  10.89  14.16  151
Alencon           4.37  11.29  14.35  143
Le Puy-Loudes    1.11   9.12  16.84  833
Millau            3.15  10.99  16.61  712
Embrun            1.57  10.89  19.18  871
```

L'union

Une des opérations les plus simples en algèbre relationnelle est l'union de données. Dans notre cas, nous allons nous intéresser à l'union de « **Series** » ou de « **DataFrame** ». Cette opération consiste en l'assemblage de plusieurs structures pour en créer une nouvelle. Avec Pandas, cette opération s'accomplit grâce à la fonction « **pd.concat** ».

```
>>> ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
>>> ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
>>> pd.concat([ser1, ser2])
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

Pour une Series, cela paraît facile. Mais pour un DataFrame ?

```
>>> def make_df(cols, ind):
...     """Crée rapidement des DataFrame"""
...     data = {c: [str(c) + str(i) for i in ind]
...             for c in cols}
...     return pd.DataFrame(data, ind)
...
>>> df1 = make_df('AB', [1, 2])
>>> df2 = make_df('AB', [3, 4])
>>> df1
      A    B
1  A1  B1
2  A2  B2
>>> df2
      A    B
3  A3  B3
4  A4  B4
>>> pd.concat([df1, df2])
      A    B
1  A1  B1
2  A2  B2
3  A3  B3
4  A4  B4
>>> df1 = make_df('AB', [1, 2])
>>> df2 = make_df('CD', [3, 4])
>>> pd.concat([df1, df2])
      A    B    C    D
1  A1  B1  NaN  NaN
2  A2  B2  NaN  NaN
3  NaN  NaN  C3  D3
4  NaN  NaN  C4  D4
```

La concaténation préserve les index ! Par exemple, si les deux **DataFrames** donnés en arguments ont des index en commun, le résultat final aura des index dupliqués.

Pour accéder à un élément d'un objet Pandas avec un index hiérarchique, il suffit de spécifier plusieurs index.

```
>>> x = make_df('AB', [0, 1])
>>> y = make_df('AB', [2, 3])
>>> y.index = x.index # Rend les index identiques
>>> # Nous avons alors des index dupliqués
>>> print(pd.concat([x, y]))
      A    B
0   A0   B0
1   A1   B1
0   A2   B2
1   A3   B3
>>> # Nous pouvons spécifier des index hiérarchiques
>>> hdf = pd.concat([x, y], keys=['x', 'y'])
>>> print(hdf)
      A    B
x 0   A0   B0
    1   A1   B1
y 0   A2   B2
    1   A3   B3
```

La jointure

Une autre fonction très utile pour manipuler les **Dataframe** est « `pd.merge` ». Elle permet de réaliser des opérations différentes en fonction des arguments qu'elle reçoit.

Jointure un-à-un

Imaginons que nous disposons de deux **Dataframe**, un contenant une liste d'employés et leurs dates d'entrée dans l'entreprise, et l'autre le nom des départements dans lesquels ils travaillent. La fonction « `pd.merge` » nous permet de transformer ces deux **Dataframes** en un seul contenant les deux informations.

```
>>> df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
...                      'department': ['Accounting',
...                                     'Engineering', 'Engineering', 'HR']})
>>> df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
...                      'date': [2004, 2008, 2012, 2014]})
>>> df3 = pd.merge(df1, df2)
>>> df3
   department employee    date
0  Accounting        Bob  2008
1  Engineering       Jake  2012
2  Engineering       Lisa  2004
3            HR        Sue  2014
```

La fonction « `pd.merge` » a automatiquement reconnu que la colonne `employee` était commune aux deux **Dataframe**, et l'a utilisée comme clé de jointure.

Jointure plusieurs-à-un

Maintenant nous voulons ajouter une autre colonne. Chaque département a un chef. Cette information est contenue dans un **Dataframe**. Nous voulons ajouter une colonne à `df3` pour y ajouter le chef de chaque employé.

```
>>> df4 = pd.DataFrame({'department': ['Accounting',
...                                      'Engineering', 'HR'],
...                      'supervisor': ['Carly', 'Guido', 'Steve']})
>>> pd.merge(df3, df4)
   department employee    date supervisor
0  Accounting        Bob  2008      Carly
1  Engineering       Jake  2012      Guido
2  Engineering       Lisa  2004      Guido
3            HR        Sue  2014      Steve
```

Remarquez que Guido apparaît plusieurs fois dans le résultat.

Jointure plusieurs-à-plusieurs

Les jointures plusieurs-à-plusieurs sont un peu compliquées à expliquer, mais elles sont quand même bien définies et très utiles. Pour continuer avec notre exemple, supposons que nous disposions d'un autre **Dataframe** contenant les compétences nécessaires pour travailler dans chaque département. Maintenant, nous souhaitons associer à chaque employé les compétences qu'il doit posséder pour travailler dans son département.

```
>>> df5 = pd.DataFrame({'department': ['Accounting','Accounting',
...                                     'Engineering','Engineering','HR','HR'],
...                      'competence': ['math','spreadsheets','coding',
...                                     'linux','spreadsheets','organization']})
>>> pd.merge(df1, df5)
   department  employee  competence
0  Accounting      Bob        math
1  Accounting      Bob  spreadsheets
2  Engineering     Jake       coding
3  Engineering     Jake      linux
4  Engineering     Lisa       coding
5  Engineering     Lisa      linux
6          HR       Sue  spreadsheets
7          HR       Sue  organization
```

Quand la colonne utilisée comme clé de jointure possède des entrées dupliquées, comme c'est le cas pour `df5`, le résultat de « `pd.merge` » est une jointure plusieurs-à-plusieurs.

Le produit cartésien

Nous pouvons utiliser les jointures plusieurs-à-plusieurs pour réaliser une autre opération d'algèbre relationnelle, le produit cartésien.

```
>>> # Nous ajoutons une nouvelle colonne à df1 et df2
... # , qui contient toujours la même valeur, ici 0.
>>> df1['key'] = 0
>>> df2['key'] = 0
>>> # La jointure plusieurs-à-plusieurs
>>> produit_cartesien = pd.merge(df1, df2, how='left', on='key')
>>> produit_cartesien
   department  employee_x  key  date  employee_y
0  Accounting      Bob    0  2004      Lisa
1  Accounting      Bob    0  2008      Bob
2  Accounting      Bob    0  2012     Jake
3  Accounting      Bob    0  2014      Sue
4  Engineering     Jake    0  2004      Lisa
5  Engineering     Jake    0  2008      Bob
6  Engineering     Jake    0  2012     Jake
7  Engineering     Jake    0  2014      Sue
8  Engineering     Lisa    0  2004      Lisa
9  Engineering     Lisa    0  2008      Bob
10  Engineering    Lisa    0  2012     Jake
11  Engineering    Lisa    0  2014      Sue
12          HR       Sue    0  2004      Lisa
13          HR       Sue    0  2008      Bob
14          HR       Sue    0  2012     Jake
15          HR       Sue    0  2014      Sue
>>> # Effaçons la colonne key qui n'est plus utile
>>> produit_cartesien.drop('key',1, inplace=True)
>>> produit_cartesien.dtypes
department    object
employee_x    object
date        int64
employee_y    object
dtype: object
```

L'argument optionnel « `on` » permet de dire explicitement à « `pd.merge` » quelle colonne utiliser comme clé de jointure. L'argument « `how` » spécifie le type de jointure, parmi « `inner` », « `outer` », « `left` » et « `right` ».

```
>>> employes = pd.read_csv("donnees/stagiaire/employes.csv",
...                         sep=';', header=0, na_values=["null"],
...                         usecols=['No', 'Manager', 'Nom', 'Prenom'],
...                         names=['No', 'Manager', 'Nom', 'Prenom'])
>>> managers = employes.copy()
>>> employes.dtypes
No           int64
Manager      float64
Nom          object
Prenom       object
dtype: object
>>> managers.dtypes
No           int64
Manager      float64
Nom          object
Prenom       object
dtype: object
>>> empman = pd.merge(employes, managers, left_on='Manager',
...                      right_on='No', how='left')
>>> empman.head()
   No_x  Manager_x     Nom_x    Prenom_x  No_y  Manager_y     Nom_y \
0    37        NaN  Giroux  Jean-Claude    NaN        NaN      NaN
1    14       37.0  Fuller    Andrew  37.0        NaN  Giroux
2    18       37.0 Brasseur   Hervé  37.0        NaN  Giroux
3    24      14.0 Buchanan  Steven  14.0       37.0  Fuller
4    95      18.0     Leger   Pierre  18.0       37.0 Brasseur

   Prenom_y
0        NaN
1  Jean-Claude
2  Jean-Claude
3    Andrew
4    Hervé
```

L'agrégation

Comme les tableaux **Numpy**, nous pouvons facilement effectuer des opérations sur l'ensemble des éléments d'une **Series** ou un **Dataframe**.

```
>>> import numpy as np
>>> rng = np.random.RandomState(42)
>>> # Une Series avec cinq nombres aléatoires
>>> ser = pd.Series(rng.rand(5))
>>> print(ser.sum())
2.811925491708157
>>> print(ser.mean())
0.5623850983416314
```

Pour un **Dataframe**, par défaut le calcul est fait par colonne.

```
>>> df = pd.DataFrame({'A': rng.rand(5),
...                     'B': rng.rand(5)})
>>> df
       A        B
0  0.155995  0.020584
1  0.058084  0.969910
2  0.866176  0.832443
3  0.601115  0.212339
4  0.708073  0.181825
>>> # Par colonne
... print(df.mean())
A    0.477888
B    0.443420
dtype: float64
>>> # Par ligne
>>> print(df.mean(axis='columns'))
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

Pandas nous permet aussi d'accomplir une agrégation par groupe, semblable à ce qu'on peut obtenir en utilisant le mot clé « **GROUP BY** » en SQL.

```
>>> df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
...                     'data': range(6)}, columns=['key', 'data'])
>>> print(df)
   key  data
0    A     0
1    B     1
2    C     2
3    A     3
4    B     4
5    C     5
>>> df.groupby('key').sum()
      data
key
```

A	3
B	5
C	7

Dans Pandas, cette opération se fait en deux étapes. Nous allons d'abord créer un objet de type **DataFrame**. **GroupBy** c'est une sorte de vue sur notre **DataFrame**. Ensuite, nous pouvons appliquer les opérations que nous souhaitons sur ce nouvel objet. Le résultat sera agrégé !

```
>>> employes = pd.read_csv("donnees/stagiaire/employes.csv",
...     sep=';', header=0, index_col='No', na_values=["null"],
...     usecols=['No', 'Nom', 'Fonction', 'Pays', 'Salaire', 'Commission'],
...     names=['No', 'Manager', 'Nom', 'Prenom', 'Fonction', 'Titre',
...     'Naissance', 'Embauche', 'Salaire', 'Commission', 'Pays', 'Region'])
>>> employes.head()
      Nom        Fonction   Salaire  Commission Pays
No
37  Giroux      Président  150000      NaN  NaN
14  Fuller  Vice-Président  96000      NaN  NaN
18  Brasseur  Vice-Président  147000      NaN  NaN
24 Buchanan  Chef des ventes  13000  12940.0  NaN
95   Leger  Chef des ventes  19000  11150.0  NaN
>>> employes.Commission = employes.apply(lambda x: 0
...     if np.isnan(x['Commission']) else x['Commission'], axis=1)
>>> employes.Pays = employes.apply(lambda x: 'NonAff'
...     if pd.isnull(x['Pays']) else x['Pays'], axis=1)
>>> employes.head()
      Nom        Fonction   Salaire  Commission      Pays
No
37  Giroux      Président  150000      0.0  NonAff
14  Fuller  Vice-Président  96000      0.0  NonAff
18  Brasseur  Vice-Président  147000      0.0  NonAff
24 Buchanan  Chef des ventes  13000  12940.0  NonAff
95   Leger  Chef des ventes  19000  11150.0  NonAff
>>> employes.groupby('Fonction').sum()
              Salaire  Commission
Fonction
Assistante commerciale  16540      0.0
Chef des ventes          83000  68790.0
Président                 150000      0.0
Représentant(e)          692900  88900.0
Vice-Président            243000      0.0
>>> employes.groupby('Fonction').mean()
              Salaire  Commission
Fonction
Assistante commerciale  1654.000000  0.000000
Chef des ventes          13833.333333  11465.000000
Président                 150000.000000  0.000000
Représentant(e)          7531.521739  966.304348
Vice-Président            121500.000000  0.000000
>>> employes.groupby(['Fonction', 'Pays']).sum()
              Salaire  Commission
Fonction      Pays
Assistante commerciale NonAff  16540      0.0
Chef des ventes      NonAff  83000  68790.0
Président           NonAff  150000      0.0
Représentant(e)      Allemagne  51200  9660.0
                           Argentine  38900  5640.0
```

	Autriche	25600	1960.0
	Belgique	27000	2930.0
	Brésil	23100	1090.0
	Canada	35500	5840.0
	Danemark	27500	3510.0
	Espagne	36700	4860.0
	Finlande	29800	2200.0
	France	32200	3610.0
	Irlande	36400	4040.0
	Italie	29000	1590.0
	Mexique	28900	3860.0
	Norvège	35300	6160.0
	Pologne	32400	5030.0
	Portugal	31600	3860.0
	Royaume-Uni	42900	4570.0
	Suisse	36700	5560.0
	Suède	31300	2790.0
	Venezuela	37800	6510.0
	États-Unis	23100	3630.0
Vice-Président	NonAff	243000	0.0
>>> employes.groupby(['Fonction','Pays']).mean()			
		Salaire	Commission
Fonction	Pays		
Assistante commerciale	NonAff	1654.000000	0.000000
Chef des ventes	NonAff	13833.333333	11465.000000
Président	NonAff	150000.000000	0.000000
Représentant(e)	Allemagne	7314.285714	1380.000000
	Argentine	7780.000000	1128.000000
	Autriche	6400.000000	490.000000
	Belgique	6750.000000	732.500000
	Brésil	7700.000000	363.333333
	Canada	7100.000000	1168.000000
	Danemark	9166.666667	1170.000000
	Espagne	7340.000000	972.000000
	Finlande	7450.000000	550.000000
	France	8050.000000	902.500000
	Irlande	7280.000000	808.000000
	Italie	7250.000000	397.500000
	Mexique	7225.000000	965.000000
	Norvège	7060.000000	1232.000000
	Pologne	8100.000000	1257.500000
	Portugal	7900.000000	965.000000
	Royaume-Uni	8580.000000	914.000000
	Suisse	7340.000000	1112.000000
	Suède	7825.000000	697.500000
	Venezuela	7560.000000	1302.000000
	États-Unis	7700.000000	1210.000000
Vice-Président	NonAff	121500.000000	0.000000

5

Un problème de data science

Introduction

Nous avons tous des problèmes... tout le temps... notre voiture ne démarre pas, notre patron nous fatigue, notre enfant ne fait pas ses nuits, etc. Hélas ! Au risque de vous décevoir, sachez que le *machine learning* ne permet pas de résoudre tous les problèmes. En revanche, il permet d'apporter des éléments de réponse à certains d'entre eux. Par exemple, aucun algorithme ne changera pour vous le démarreur de votre automobile. Néanmoins, si vous disposez de suffisamment de données concernant le fonctionnement de votre véhicule, un algorithme pourrait détecter une panne et vous suggérer de changer le démarreur. Peut-être même pourrait-il vous le suggérer avant même que le démarreur ne rende l'âme ! Mais pour cela, il est nécessaire de traduire votre problème humain en éléments qui puissent être analysés par un algorithme de *machine learning*. Autrement dit, vous devez être capable de poser votre problème sous la forme d'un problème de *data science*. Globalement, la démarche est simple : (1) il vous faut des données ; (2) vous devez savoir ce que vous voulez en faire, puis (3) comment le faire. Ce chapitre va vous donner des éléments pratiques pour répondre à ces questions. Après l'avoir lu, vous saurez mieux qualifier et rechercher des données. Vous saurez aussi poser votre problème de *data science* selon un vocabulaire précis. Enfin, vous saurez comment structurer vos données pour qu'un algorithme de *machine learning* puisse leur être appliqué.

Préliminaire : qu'est-ce que le machine learning ?

Même s'il est actuellement dopé par les nouvelles technologies et de nouveaux usages, le *machine learning* n'est pas un domaine d'étude récent. On en trouve une première définition dès 1959, due à Arthur Samuel, l'un des pionniers de l'intelligence artificielle, qui définit le *machine learning* comme le champ d'étude visant à donner la capacité à une machine d'apprendre sans être explicitement programmée. En 1997, Tom Mitchell, de l'université de Carnegie Mellon, propose une définition plus précise :

« A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E ».

Tom Mitchell illustre ensuite cette définition par quelques exemples (tableau 1-1).

Tableau 1-1. Quelques exemples de machine learning proposés par Mitchell

Cas d'application	Checkers learning	Handwriting recognition	Robot driving learning
Tasks T	Playing checkers	Recognizing and classifying handwritten words within images	Driving on public four-lane highways using vision sensors
Performance measure P	Percent of games won against opponents	Percent of words correctly classified	Average distance traveled before an error (as judged by human overseer)
Training experience E	Playing practice games against itself	A database of handwritten words with given classifications	A sequence of images and steering commands recorded while observing a human driver

Un robot qui apprend à conduire ? Cela peut paraître un peu loin de vos préoccupations... mais illustrons brièvement ce que peut faire le *machine learning* avec un cas simple, sans doute plus proche de votre quotidien : un filtre antispam. Dans un premier temps, on peut imaginer que la « machine » (votre service de messagerie) va « analyser » la façon dont vous allez classer vos mails entrants en spam ou pas. Grâce à cette période d'« apprentissage », la machine va déduire quelques grands critères de classification. Par exemple, la probabilité que la machine classe un mail en spam va augmenter si le mail contient des termes tels qu'« argent », « rencontre facile » ou « viagra » et si l'expéditeur du mail n'est pas dans votre carnet d'adresses. A contrario, la probabilité de classement en spam va baisser si l'expéditeur est connu et que les mots du mail sont plus « classiques ».

On laissera le lecteur choisir la définition du *machine learning* avec laquelle il est à l'aise. Le tout est de bien comprendre la métaphore de l'apprentissage de la machine (à opposer à une programmation impérative de type règle « IF... THEN... ELSE... »), qui permettra à la machine d'apprendre à partir de données réelles. Avec le *machine learning*, on passe d'une informatique impérative basée sur des hypothèses à une informatique probabiliste basée sur des informations réelles. Mais pour se lancer dans cette aventure, il nous faut avant tout des données !

Au commencement était la donnée...

Un prérequis indispensable

La *data science* est une démarche empirique qui se base sur des données pour apporter une réponse à des problèmes. Donc, avant toute chose, assurez-vous d'avoir des données... Cela peut paraître idiot de le préciser, mais nous voyons souvent dans nos projets que l'accès aux données utiles à l'analyse n'est pas toujours aisé. Tantôt elles ne sont pas disponibles ou certaines manipulations techniques (jointure, etc.) ne sont pas faisables, ou encore interdites pour des raisons de confidentialité, tantôt le service informatique n'est pas capable de les extraire, tantôt elles n'existent tout simplement pas.

De plus, même si l'objectif de ce livre n'est pas de parler de *big data*, ayez en tête qu'en *machine learning*, la plupart du temps, plus il y a de données, mieux c'est ! En effet, le *machine learning* s'appuie souvent sur des algorithmes de *data-mining* connus depuis longtemps, souvent développés avant les années 2000. Hélas ! Leurs performances ont bien souvent été limitées par le manque de données disponibles ou de capacités informatiques pour traiter de larges volumes de données. C'est que certains problèmes de *data science* ne donnent parfois des résultats probants qu'à partir de plusieurs centaines de millions d'observations (on parle parfois d'analyse de signaux faibles, car non observables sur des ensembles de données réduits tels que ceux traités par les études statistiques classiques). Aujourd'hui, la hausse des capacités de stockage et la profusion de données numériques qu'elles engendrent, couplées à des moyens de calcul informatique de plus en plus puissants, font revenir ces algorithmes sur le devant de la scène. Ayez donc conscience que les gros volumes de données ne sont pas un problème pour votre travail, mais au contraire une formidable opportunité d'y trouver des informations précieuses !

Très bien, vous savez maintenant qu'il vous faut des données, peut-être même beaucoup de données... mais il se peut que ce terme soit un peu abstrait pour vous, alors clarifions-le dès à présent.

Que sont les données ?

Un bon vieux dictionnaire de statistique indique qu'une donnée est « *le résultat d'une observation faite sur une population ou sur un échantillon¹* » (Dodge, 2007). Une donnée est donc un nombre, une caractéristique, qui m'apporte une information sur un individu, un objet ou une observation. Par exemple, 33 est un nombre sans intérêt², mais si quelqu'un vous dit « J'ai 33 ans », 33 devient une donnée qui vous permettra d'en savoir un peu plus sur lui.

Généralement, on lie les données à des variables parce que le nombre/la caractéristique varie si on observe plusieurs objets/individus/observations. En effet, si on s'intéresse à l'âge de tous les lecteurs de ce livre, on sait qu'il est défini par un nombre compris entre, disons, 15 et 90, et qu'il variera d'un lecteur à l'autre. Ainsi, si l'on nomme $X_{\text{âge}}$ la variable « âge du lectorat », les données mesurant cet âge sont égales à $x_{1\text{âge}}, x_{2\text{âge}}, \dots, x_{m\text{âge}}$, où $x_{1\text{âge}}$ est l'âge du lecteur 1, $x_{2\text{âge}}$ l'âge du lecteur 2, et ainsi de suite jusqu'à $x_{m\text{âge}}$, où m représente le nombre total de lecteurs.

Tel est le matériau brut que va manipuler le *data scientist* : des variables exprimées concrètement par des données et qui lui permettent de décrire un ensemble d'objets/individus/observations. Ces données peuvent prendre diverses formes que nous allons désormais détailler.

Les principaux types de données

On distingue généralement les données quantitatives des données qualitatives.

Les données quantitatives sont des valeurs qui décrivent une quantité mesurable, sous la forme de nombres sur lesquels on peut faire des calculs (moyenne, etc.) et des comparaisons (égalité/différence, infériorité/supériorité, etc.). Elles répondent typiquement à des questions du type « combien ». On fait parfois la différence entre :

- les données quantitatives continues, qui peuvent prendre n'importe quelle valeur dans un ensemble de valeurs : la température, le PIB, le taux de chômage, en sont des exemples ;
- et les données quantitatives discrètes, qui ne peuvent prendre qu'un nombre limité de valeurs dans un ensemble de valeurs : le nombre d'enfants par famille, le nombre de pièces d'un logement, etc.

Les données qualitatives décrivent quant à elles des qualités ou des caractéristiques. Elles répondent à des questions de la forme « quel type » ou « quelle catégorie ». Ces valeurs ne sont plus des nombres, mais un ensemble de modalités. On ne peut pas faire de calcul sur ces valeurs, même dans l'éventualité où elles prendraient l'apparence d'une série numérique. Elles peuvent toutefois être comparées entre elles et éventuellement triées. On distingue :

- les données qualitatives nominales (ou catégorielles), dont les modalités ne peuvent être ordonnées. Par exemple : la couleur des yeux (bleu, vert, marron, etc.), le sexe (homme, femme), la région d'appartenance (68, 38, etc.) ;
- et les données qualitatives ordinaires, dont les modalités sont ordonnées selon un ordre « logique ». Par exemple : les tailles de vêtements (S, M, L, XL), le degré d'accord à un test d'opinion (fortement d'accord, d'accord, pas d'accord, fortement pas d'accord).

Le tableau 1-2 résume ces différents types de données ainsi que les opérations qu'ils supportent.

Tableau 1-2. Les opérations supportées par chaque type de données

Type de données	Opérations supportées
Quantitatives continues	Calculs, égalité/différence, infériorité/supériorité
Quantitatives discrètes	Calculs, égalité/différence, infériorité/supériorité
Qualitatives nominales	Égalité/différence
Qualitatives ordinaires	Égalité/différence, infériorité/supériorité

Maintenant que vous savez la différence entre les diverses données, vous vous demandez peut-être où les chercher... Bonne question en effet ! Voici quelques pistes qui pourront vous aider.

D'où viennent les données ?

La réponse à cette question n'est pas bien difficile : elles viennent de partout ! C'est d'ailleurs bien pour cela qu'on observe de nos jours un tel engouement pour la *data science*, le *machine learning* et l'analyse de données en général. Précisons tout de même un peu. En premier lieu, distinguons les données dites privées des données publiques.

Les données privées sont tout simplement les données qui en théorie n'appartiennent qu'à vous ou à votre organisation. Si vous travaillez en entreprise, ce pourrait être les bases de données internes de votre société, les divers documents électroniques ou numérisés disponibles (e-mails, fichiers informatiques, documents scannés, images et vidéos, etc.). Ce pourrait être aussi les fichiers de *logs*³ générés par vos machines (serveurs informatiques, équipements de production, etc.) et par vos applications informatiques (logiciels, applications web, etc.), les informations remontées par les capteurs et objets connectés, et bien d'autres. Il y a déjà beaucoup de connaissances à en tirer.

Toutefois, les plus ambitieux d'entre vous pourraient aussi avoir envie d'aller plus loin en exploitant des données publiques, c'est-à-dire accessibles à tous. Dans ce cas, vous disposez d'une source de données quasi infinie : Internet. Pour cela, trois modes de collecte de données existent.

- Les *open data*, qui correspondent à la mise à disposition gratuite de données de la société civile, sur des sites tels que www.data.gov, www.data.gouv.fr, <http://opendata.alsace.fr>, etc.
- Les *open API (Application Programming Interface)*, qui sont des technologies permettant d'accéder à des données sur Internet. Elles vous permettent de récupérer par exemple des données mises à disposition par Google, Twitter, etc. Pour en savoir plus sur les API disponibles, consultez par exemple l'annuaire <http://www.programmableweb.com>.
- Et bien sûr, le Web en tant que tel est lui aussi directement source de données. Pour cela, il faut un minimum d'expertise en programmation pour être capable de faire ce que l'on nomme du *web scraping*, qui consiste à récupérer des données directement à partir des pages des sites Internet.

Nous avons également parlé d'images, de vidéos. Vous vous demandez peut-être ce que cela a à voir avec les types de données qui vous ont été présentés précédemment. En fait, lorsqu'on traite informatiquement des objets de types images et vidéos, on leur applique des traitements visant à les réduire à une suite de nombres interprétables par un algorithme de *machine learning*. Vous en verrez des exemples dans la suite de ce livre. Ces objets, assez complexes à manipuler, sont dits non structurés. À l'opposé, les données les plus faciles à traiter sont celles qui proviennent directement des bases de données : indexées⁴, prêtes à être traitées, elles sont dites structurées. Sachez qu'il existe un niveau de structuration intermédiaire, dit semi-structuré. Celui-ci correspond à divers formats de fichiers informatiques simples à traiter, mais qui ne bénéficient pas de l'indexation

propre aux données extraites des bases de données informatiques. Le tableau 1-3 résume ces différents niveaux de structuration.

Tableau 1-3. Les différents niveaux de structuration des données

Niveau de structuration	Modèle de données	Exemples	Facilité de traitement
Structuré	Système de données relationnel objet/colonne	Base de données d'entreprise...	Facile (indexé)
Semi-structuré	XML, JSON, CSV, logs	API Google, API Twitter, web, logs...	Facile (non indexé)
Non structuré	Texte, image, vidéo	web, e-mails, documents...	Complexé

Voilà, vous savez identifier la matière première du *data scientist* et vous savez où la chercher. À présent, vous brûlez sans doute d'impatience de pouvoir l'exploiter grâce aux nombreux algorithmes conçus pour cela. Mais avant de les passer en revue, laissez-nous vous donner une idée générale de ce que savent faire tous ces algorithmes.

Les algorithmes : pour faire quoi ?

Sous les données, des liens... plus ou moins certains !

Quel que soit l'algorithme qu'il utilise, le *data scientist* n'a qu'une seule idée en tête : découvrir des liens dans ses données (on parle souvent de *pattern*). Dans le cadre de l'emploi de méthodes de *machine learning*, on suppose donc qu'il existe un lien au sein des données et que nos algorithmes vont nous aider à le trouver.

Attention, il s'agit d'une hypothèse forte : par défaut, il n'y a généralement pas de lien dans les données. Cela peut paraître surprenant, car nous avons plutôt tendance à en voir partout ! Ce phénomène bien connu en recherche s'appelle le biais de publication (Cucherat *et al.*, 1997). En science, il désigne le fait que les chercheurs et les revues scientifiques ont plus tendance à publier des expériences ayant obtenu un résultat positif (statistiquement significatif) que des expériences ayant obtenu un résultat négatif. Ce biais de publication donne aux lecteurs une perception biaisée (vers le positif) de l'état de la recherche.

En plus du biais de publication, un ensemble de phénomènes bien connus en statistiques peuvent aussi amener à conclure artificiellement à des liens dans les données.

- La corrélation fallacieuse, pour désigner des séries de données corrélées entre elles alors qu'elles n'ont a priori aucune raison de l'être. Deux explications peuvent être apportées à ce type de corrélation :
 - soit les séries n'ont aucun lien entre elles, mais sont toutes deux corrélées à une troisième variable cachée (Pearl, 1998) ;
 - soit de grands jeux de données génèrent naturellement des corrélations, sans aucune relation de causalité : si l'on considère un grand nombre de variables totalement indépendantes, on observera toujours un petit nombre de corrélations significatives (Granville, 2013) !

- La corrélation fallacieuse au sens de K. Pearson, utilisée dans le contexte très spécifique des données de composition. Dans un article de 1897, appliquée à la mesure d'organes humains, il montre que deux variables indépendantes X et Y peuvent sembler corrélées si rapportées à une troisième variable Z . La taille du fémur est indépendante de la taille du tibia, mais le ratio taille du fémur/taille de l'humérus est corrélé au ratio taille du tibia/taille de l'humérus ! L'analyse des données de compositions est un champ assez spécifique des statistiques, c'est pourquoi c'est la dernière fois que nous en parlerons dans ce livre.
- La régression fallacieuse, dans le cadre de l'estimation d'une régression linéaire simple ou multiple. On peut observer des modèles qui semblent très explicatifs, mais faux du fait de l'influence du temps sur le modèle (Granger et Newbold, 1974).
- Le paradoxe de Bertrand, qui nous montre que les résultats issus de nos analyses peuvent être influencés par les méthodes et les mécanismes qui produisent les variables (Bertrand, 2010).
- Le mauvais usage de la p -value, indicateur souvent utilisé sans précaution par certains statisticiens pour évaluer une significativité statistique et qui aboutit très facilement à rejeter une hypothèse nulle de façon totalement artificielle lorsque le nombre d'observations est grand (Lin *et al.*, 2013).

Les précautions d'usage ayant été exposées, nous supposerons un lien dans les données dans la suite de cet ouvrage, complexe peut-être, mais bel et bien présent. D'ailleurs, ces écueils sont plutôt la conséquence de mauvaises applications des méthodes statistiques classiques dans des contextes inappropriés, notamment de larges volumes de données. Néanmoins, voyez-y une invitation à la prudence : on gagne toujours à réfléchir à deux fois aux résultats de ses analyses et à remettre en question ses conclusions ! De toute manière, une approche *machine learning* est toujours une prise de position probabiliste qui implique de renoncer à une vision déterministe et figée du monde. Elle va inférer des règles, avec des marges d'incertitude et potentiellement changeantes.

Ces considérations métaphysiques étant faites, parlons plus concrètement de la façon dont les liens entre les données peuvent être découverts.

Une taxinomie des algorithmes

Les algorithmes ne sont pas tous destinés aux mêmes usages. On les classe usuellement selon deux composantes⁵ :

- le mode d'apprentissage : on distingue les algorithmes supervisés des algorithmes non supervisés ;
- le type de problème à traiter : on distingue les algorithmes de régression de ceux de classification.

Tous les algorithmes qui seront présentés dans ce livre sont définis selon ces deux axes, comme l'indique le tableau 1-4.

Tableau 1-4. Taxinomie des algorithmes présentés dans ce livre

Algorithmme	Mode d'apprentissage	Type de problème à traiter
Régression linéaire univariée	Supervisé	Régression
Régression linéaire multivariée	Supervisé	Régression
Régression polynomiale	Supervisé	Régression
Régression régularisée	Supervisé	Régression
Naïve Bayes	Supervisé	Classification
Régression logistique	Supervisé	Classification
Clustering hiérarchique	Non supervisé	-
Clustering non hiérarchique	Non supervisé	-
Arbres de décision	Supervisé	Régression ou classification
Random forest	Supervisé	Régression ou classification
Gradient boosting	Supervisé	Régression ou classification
Support Vector Machine	Supervisé	Régression ou classification
Analyse en composantes principales	Non supervisé	-

Donnons un peu plus de détails sur la signification de cette taxinomie.

Algorithmes supervisés et non supervisés

La différence entre algorithmes supervisés et non supervisés est fondamentale. Les algorithmes supervisés extraient de la connaissance à partir d'un ensemble de données contenant des couples entrée-sortie. Ces couples sont déjà « connus », dans le sens où les sorties sont définies a priori. La valeur de sortie peut être une indication fournie par un expert : par exemple, des valeurs de vérité de type OUI/NON ou MALADE/SAIN. Ces algorithmes cherchent à définir une représentation compacte des associations entrée-sortie, par l'intermédiaire d'une fonction de prédiction.

A contrario, les algorithmes non supervisés n'intègrent pas la notion d'entrée-sortie. Toutes les données sont équivalentes (on pourrait dire qu'il n'y a que des entrées). Dans ce cas, les algorithmes cherchent à organiser les données en groupes⁶. Chaque groupe doit comprendre des données similaires et les données différentes doivent se retrouver dans des groupes distincts. Dans ce cas, l'apprentissage ne se fait plus à partir d'une indication qui peut être préalablement fournie par un expert, mais uniquement à partir des fluctuations observables dans les données.

Le petit exemple qui suit illustre les principes de ces deux familles d'algorithmes. Imaginons un ensemble d'individus décrits par deux variables d'entrée, X_1 et X_2 . Dans le cas d'un apprentissage supervisé, il faudra leur adjoindre une variable de sortie Y , qui pourra par exemple prendre deux valeurs {O, X}. L'algorithme proposera alors une fonction de prédiction de la forme

$Y = f(X_1, X_2)$. Dans le cas de l'apprentissage non supervisé, plus de Y : l'algorithme va trouver tout seul, comme un grand, deux groupes d'individus distincts, juste à partir des positions dans le plan défini par X_1 et X_2 , et ce sans aucune autre indication. La figure 1-1 illustre ces deux formes d'apprentissage.

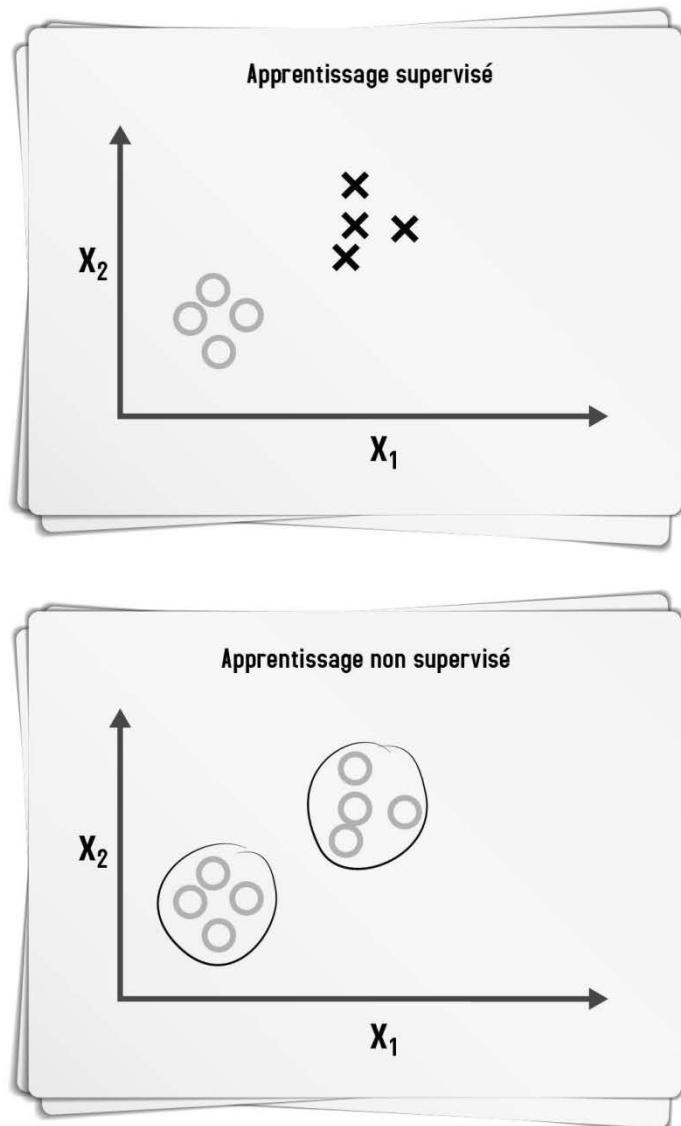


Figure 1-1 – Apprentissages supervisé et non supervisé

Généralement, les algorithmes supervisés sont plus performants, mais pour les utiliser, il faut être capable de spécifier une valeur de sortie, et cette information d'expert n'est pas toujours disponible.

Insistons sur ce point. À l'ère des *big data*, la donnée ne coûte pas cher. Vous avez sans doute, à force de le lire partout dans la presse ou en écoutant les consultants passer dans votre bureau, retenu les fameux « 3V » des *big data* (Volume – Variété – Vélocité), faible descripteur de ce que sont vraiment les *big data*. Alors oui, bien évidemment, le volume de données disponibles explose. Mais ces données ont-elles une valeur métier pour vous, êtes-vous capable de les valoriser ? C'est moins sûr. Nous y reviendrons plus tard, mais songez que pour obtenir un résultat probant pour un problème relativement simple de classification binaire équilibré (autant de 0 que de 1), il faudra déjà quelques dizaines de milliers de lignes. Chacune de ces lignes devra être parfaitement labélisée d'un point de vue métier. Disposez-vous des moyens humains pour constituer vos bases de données supervisées ? C'est une question non négligeable. Rassurez-vous toutefois avec cette information : une grande partie des efforts de la sphère académique se concentre aujourd'hui sur les algorithmes non supervisés, qui cherchent précisément à vous épargner de douloureuses et ennuyeuses journées de travail à labéliser à la main plusieurs millions de lignes.

Pour finir, notez qu'il existe une catégorie bien moins courante d'algorithmes hybrides basés sur une approche dite semi-supervisée. C'est une approche intermédiaire qui se base à la fois sur des observations de type entrée-sortie et sur des observations sans variable de sortie, mais elle ne sera pas développée dans ce livre.

Algorithmes de régression et de classification

La distinction régression/classification se fait au sujet des algorithmes supervisés. Elle distingue deux types de valeurs de sorties qu'on peut chercher à traiter. Dans le cadre d'un problème de régression, Y peut prendre une infinité de valeurs dans l'ensemble continu des réels (noté $Y \in \mathbb{R}$). Ce peut être des températures, des tailles, des PIB, des taux de chômage, ou tout autre type de mesure n'ayant pas de valeurs finies a priori.

Dans le cadre d'un problème de classification, Y prend un nombre fini k de valeurs ($Y = \{1, \dots, k\}$). On parle alors d'étiquettes attribuées aux valeurs d'entrée. C'est le cas des valeurs de vérité de type OUI/NON ou MALADE/SAIN évoqués précédemment.

Voici à nouveau un petit exemple illustratif, à partir de la figure 1-2.

- L'image du haut répond au problème suivant : quel est le prix d'une maison en fonction de sa taille ? Ce prix peut prendre une infinité de valeurs dans \mathbb{R} , c'est un problème de régression.
- L'image du bas s'intéresse quant à elle à un autre problème : selon sa taille, une tumeur est-elle dangereuse ou bénigne ? Ici, on va chercher à classer les observations en fonction de valeurs de réponse possibles en nombre limité : OUI, NON (éventuellement PEUT-ÊTRE). C'est un problème de classification.

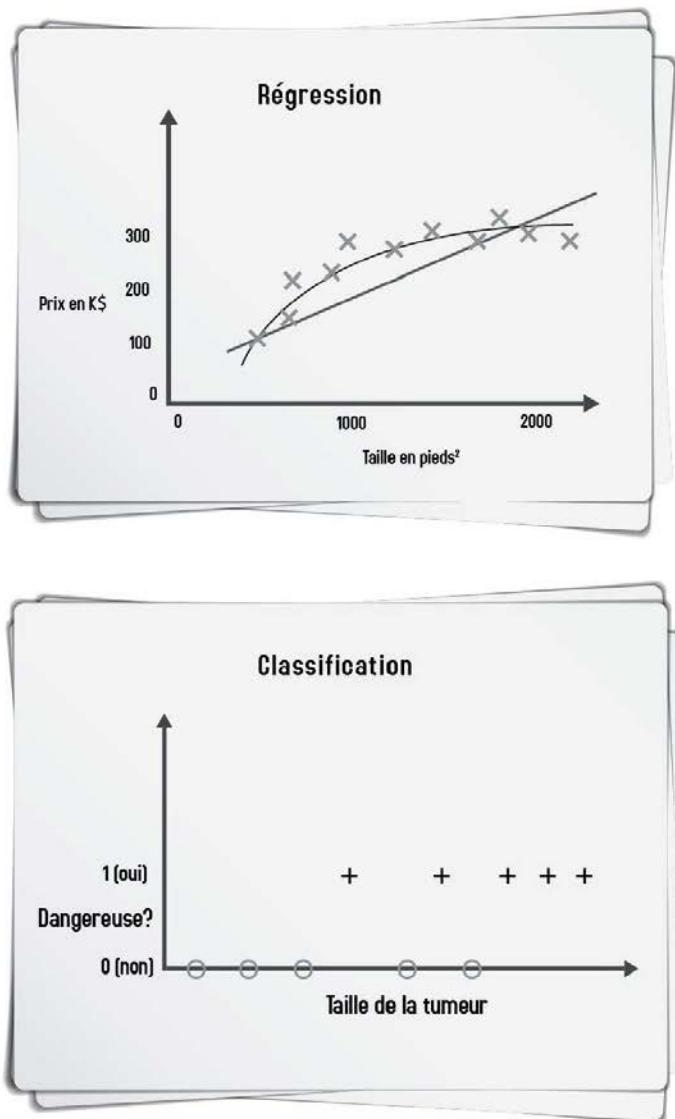


Figure 1-2 – Algorithmes de régression et algorithmes de classification

Algorithmes et structures de données

Représentation matricielle des données

Nous avons vu que dans le monde du *machine learning*, tout objet, individu, observation, est décrit par un ensemble de variables X_1, X_2, \dots, X_n , j allant de 1 à n . Évidemment, tout l'intérêt du *machine learning* sera de trouver des régularités dans les données grâce à l'observation d'un grand nombre i d'individus⁷, allant de 1 à m . La valeur de la variable X_j de l'individu i se note x_{ij} . Le cas général se note ainsi : x_{ij} , c'est-à-dire la valeur de la variable X_j de l'individu x_i .

Ces n variables décrivant m individus sont représentés dans ce qu'on appelle une matrice X de dimensions (m,n) . On la représente comme suit :

$$X = \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix}$$

Chaque colonne correspond à une variable et chaque ligne correspond à un individu. Pour être encore plus clair, cette matrice est directement équivalente à la manière dont vous pourriez organiser vos données dans un tableau (figure 1-4).

	VARIABLES		
	x_1	...	x_n
INDIVIDUS	$x_{1,1}$	X	$x_{1,n}$
...			
m	$x_{m,1}$		$x_{m,n}$

Figure 1-4 – Une matrice, c'est un tableau, tout simplement !

Si le tableau n'a qu'une colonne (soit une seule variable) on ne parle plus de matrice, mais de vecteur. Le *machine learning* a pour seul objectif la manipulation de vecteurs et de matrices pour en dégager des représentations synthétiques des données observées.

Que font les algorithmes ?

Quel que soit l'algorithme utilisé, le but sera toujours le même. Les algorithmes non supervisés vont chercher à produire des représentations compactes des données, en regroupant les individus similaires compte tenu des valeurs de la matrice X . Pour cela, la matrice X est suffisante. Les algorithmes ont simplement besoin de moyens de mesurer les proximités entre les individus. Nous reparlerons de cela en détail lors de la présentation des algorithmes.

Pour les algorithmes supervisés, c'est un peu différent. X va représenter les valeurs d'entrée (on parle aussi d'attributs ou de *features*). Il faudra leur adjoindre, pour chaque individu, un vecteur Y représentant les valeurs de sortie ; il sera donc de dimension $(m, 1)$. Le but du jeu est de décrire une relation liant X à Y . Pour cela, nous avons besoin d'un deuxième vecteur, nommé Θ (dans le cas d'un modèle linéaire, mais nous préciserons tout cela dans les chapitres à venir). Les valeurs de Θ , qu'on nomme paramètres ou poids du modèle, sont associées à celles de X , pour expliquer Y . Il faut ainsi une valeur de Θ pour chaque variable, Θ sera donc de dimension $(n, 1)$. Grossièrement, cela revient à écrire :

$$\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix} \begin{pmatrix} \Theta_1 \\ \vdots \\ \Theta_n \end{pmatrix}$$

ou, sous une forme condensée : $Y=X\Theta$. Tout l'art du *machine learning* sera de trouver les bonnes valeurs de Θ , qui ne sont pas données mais inférées à partir des données. Lorsqu'on recherche ces valeurs Θ , on parle d'entraînement ou d'apprentissage du modèle. Pour un problème donné, différents algorithmes donneront des valeurs de Θ plus ou moins efficaces pour modéliser les données.

Mesurer des distances, apprendre les paramètres d'un modèle, voilà tout ce que le *data scientist* cherche à faire. Mais il existe de multiples façons de le faire. Nous allons vous les présenter dans la partie suivante, par une petite visite guidée des principaux algorithmes rencontrés dans le monde du *machine learning*.

Avant d'entrer dans le vif du sujet, nous aborderons un dernier point préliminaire dans le chapitre suivant : la question des outils informatiques. En effet, l'ordinateur est le meilleur allié du *data scientist* pour manipuler les données, avoir accès à des codes d'algorithmes prêts à l'emploi, etc.

6

Les algorithmes de régression

La régression linéaire univariée

Introduction

Régression linéaire univariée... Sous ce nom un peu barbare se cache en réalité un algorithme d'apprentissage supervisé très basique qui vise à trouver la meilleure droite possible à l'aide d'une seule variable explicative (on parle aussi d'un seul degré de liberté). Du fait de cette unique variable explicative, on parle de modèle univarié, par opposition aux modèles multivariés, qui font appel à plusieurs variables.

L'utilité de ce chapitre est plus pédagogique que scientifique, car dans les faits, vous utiliserez rarement des modèles univariés. Toutefois, leur simplicité permet d'introduire des notions qui peuvent devenir complexes lorsque l'on se place dans le cadre général. Ainsi, nous introduirons notamment la construction de la fonction de coût, qui mesure l'erreur que l'on fait en approximant nos données. Nous verrons que trouver les meilleurs paramètres de notre modèle équivaut à minimiser cette fonction de coût. Enfin, nous introduirons une méthode de résolution numérique dont l'intuition se trouve déjà dans les travaux du très vénérable Isaac Newton : la descente de gradient.

Ce chapitre va donc expliciter les trois étapes nécessaires pour passer des données brutes au meilleur modèle de régression linéaire univarié :

- la définition d'une fonction hypothèse ;
- puis la construction d'une fonction de coût ;
- et enfin la minimisation de cette fonction de coût.

Une fois ces notions acquises, nous pourrons introduire des algorithmes linéaires plus performants.

Définition de la fonction hypothèse

Dans le cas de la régression linéaire univariée, on prend une hypothèse simplificatrice très forte : le modèle dépend d'une unique variable explicative, nommée X . Cette variable est un entrant de notre problème, une donnée que nous connaissons. Par exemple, le nombre de pièces d'un appartement, une pression atmosphérique, le revenu de vos clients, ou n'importe quelle grandeur qui vous paraît influencer une cible, qu'on nommera Y et qui, pour continuer nos exemples, serait respectivement, le prix d'un appartement, le nombre de millilitres de pluie tombés, ou encore le montant du panier moyen d'achat de vos clients.

On cherche alors à trouver la meilleure fonction hypothèse, qu'on nommera h et qui aura pour rôle d'approximer les valeurs de sortie Y :

$$\begin{array}{ccc} \text{hypothèse } h & & \\ \text{valeur d'entrée } x & \longrightarrow & \text{valeur de sortie } y \end{array}$$

Dans le cas de la régression linéaire à une variable, la fonction hypothèse h sera de la forme :

$$h(X) = \theta_0 + \theta_1 X$$

En représentation matricielle, θ_0 et θ_1 forment un vecteur Θ tel que :

$$\Theta = \begin{pmatrix} \theta_0 \\ \theta_1 \end{pmatrix}$$

Notre problème revient donc à trouver le meilleur couple (θ_0, θ_1) tel que $h(x)$ soit « proche » de Y pour les couples (x, y) de notre base de données, que l'on peut commencer à nommer base d'apprentissage puisque les données dont vous disposez serviront à entraîner la fonction hypothèse h .

Qui dit approximation dit erreur

La notion de proximité qui vient d'être introduite incite assez naturellement à calculer une fonction d'erreur. En effet, pour chaque point x_i , la fonction hypothèse associe une valeur définie par $h(x_i)$ qui est plus ou moins proche de y_i . On définit ainsi « l'erreur unitaire » pour x_i par $(h(x_i) - y_i)^2$ (on élève au carré pour s'assurer que la contribution de l'erreur viendra bien pénaliser une fonction de coût). On résume cela dans la figure 3-1.

L'erreur unitaire pour x_i étant définie, on peut ensuite sommer les erreurs pour l'ensemble des points :

$$\sum_{i=1}^m (h(x_i) - y_i)^2$$

La fonction de coût est alors définie en normant cette somme par le nombre m de points dans la base d'apprentissage⁸ :

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h(x_i) - y_i)^2$$

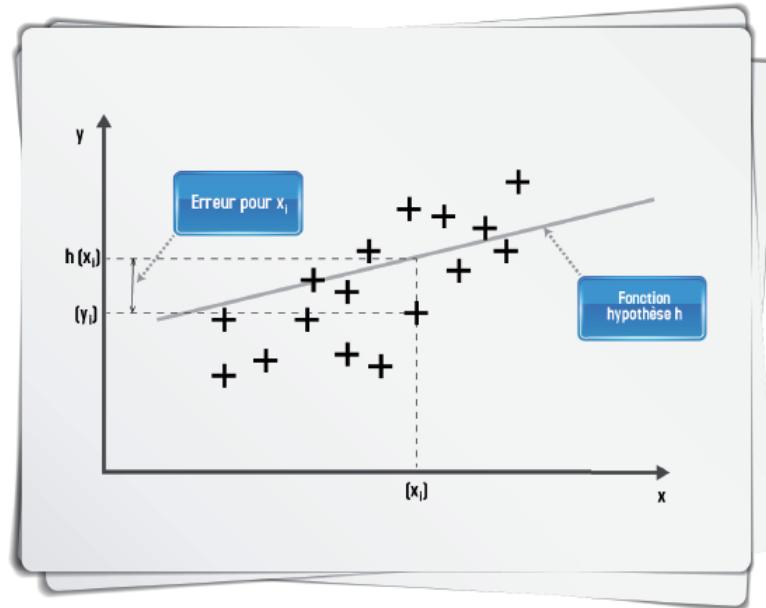


Figure 3-1 – La notion d'erreur unitaire

Si on remplace h par son expression, on obtient finalement l'expression suivante :

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_i (\theta_0 + \theta_1 x_i - y_i)^2$$

Il convient ici de comprendre une chose très importante : bien que h soit une fonction de x , J ne l'est pas ! En effet, les valeurs de x et de y sont données et J est par construction une fonction des paramètres de h , à savoir (θ_0, θ_1) .

Rappelons pour l'exercice la signification des paramètres de J , qui définissent une droite affine telle qu'on nous l'a enseignée au collège :

- θ_0 est l'ordonnée à l'origine de la fonction h , soit la valeur de h quand x est nulle ;
- θ_1 est la pente (ou coefficient directeur) de h dans un repère cartésien orthonormé, soit la variation de h quand x augmente d'une unité.

À chaque point dans l'espace (θ_0, θ_1) , correspond donc une unique droite h avec son ordonnée à l'origine et sa pente.

8. Intuitivement, cette normalisation permet d'« annuler » l'effet de la taille de la population observée sur la fonction de coût.

Minimiser la fonction de coût

Ainsi, trouver les meilleurs paramètres (θ_0, θ_1) de h – et donc la meilleure droite pour modéliser notre problème – équivaut exactement à trouver le minimum de la fonction J .

Pour comprendre la fonction J , remarquons que :

- pour θ_0 donné, J est une fonction de θ_1^2 ;
- pour θ_1 donné, J est une fonction de θ_0^2 .

Pour l'une de ces deux dimensions, c'est-à-dire en fixant l'un de ces deux paramètres, la fonction J est une parabole. En tenant compte des deux dimensions, c'est-à-dire en se plaçant dans l'espace (θ_0, θ_1) , J aura donc une forme de bol très caractéristique, comme le montre la figure 3-2.

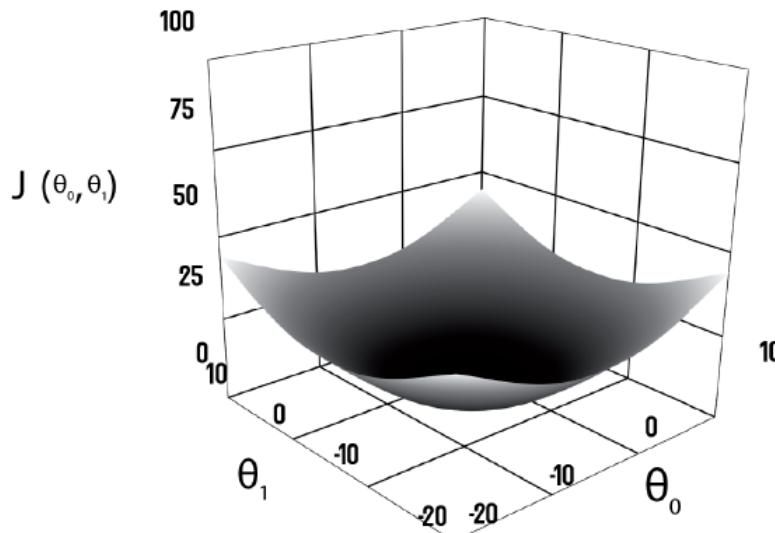


Figure 3-2 – La fonction de coût J dans l'espace (θ_0, θ_1)

Avec cette forme de bol, J possède une propriété mathématique très intéressante : elle est convexe. Pour rappel, une fonction f est convexe quand elle vérifie la propriété suivante :

$$\forall (x_1, x_2) \text{ et } \forall t \in [0,1] \\ f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

En français courant, si x_1 et x_2 sont deux points du graphe de la fonction f , alors le segment $[x_1, x_2]$ est au-dessus de ce même graphe.

L'ensemble des points situés au-dessus du graphe de la fonction f définissent à leur tour un ensemble convexe, objet géométrique très simple à comprendre comme l'illustre la figure 3-3.

L'importance de la convexité de la fonction de coût tient dans la méthode utilisée pour la minimiser.

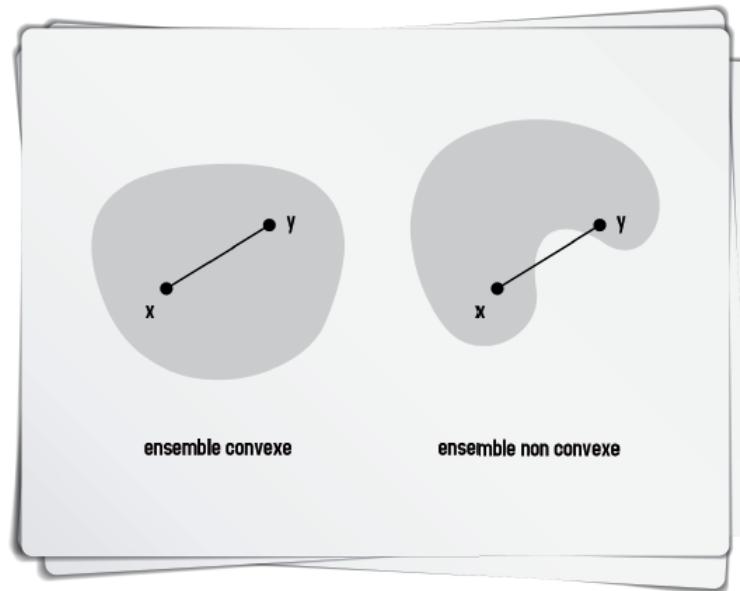


Figure 3-3 – Ensemble convexe à gauche, non convexe à droite

En effet, une façon de trouver le meilleur couple (θ_0, θ_1) , est la descente de gradient, méthode itérative dont le principe est assez intuitif : que ferait une balle lâchée assez haut dans notre bol ? Elle prendrait, à chaque instant, la meilleure pente jusqu'au point bas du bol. La formulation mathématique de cette intuition est :

- itération 0 : initialisation d'un couple (θ_0, θ_1) ;
- itérer jusqu'à convergence :

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \text{ pour } j=1 \text{ et } 0$$

À chaque itération, on choisit la meilleure « pente » sur notre fonction J pour se diriger itération après itération vers le minimum de notre fonction.

La vitesse de convergence dépendra notamment de l'initialisation plus ou moins heureuse de l'itération 0. Mais là n'est pas le vrai problème. Dans le cas d'une fonction non convexe, une initialisation malheureuse peut conduire à trouver un minimum local pour J , comme l'illustre la figure 3-4 (ici en une seule dimension).

À un couple (θ_0, θ_1) trouvé grâce un minimum local de J correspond une fonction h qui est loin d'être optimale. La convexité de J résout cet épineux problème puisque pour une fonction convexe, tout minimum local est aussi le minimum global.

Vous avez sans doute remarqué la présence d'un facteur α devant la dérivée partielle de J dans la formulation de la descente de gradient. Ce facteur α , qu'on appelle le *learning rate* représente physiquement la vitesse à laquelle nous souhaitons terminer nos itérations. Plus α est grand, plus le pas est grand entre deux itérations, mais plus la probabilité de passer outre le minimum, voire de diverger, est grande. À l'inverse, plus α est petit et plus on a de chance de trouver le minimum, mais plus longue sera la convergence.

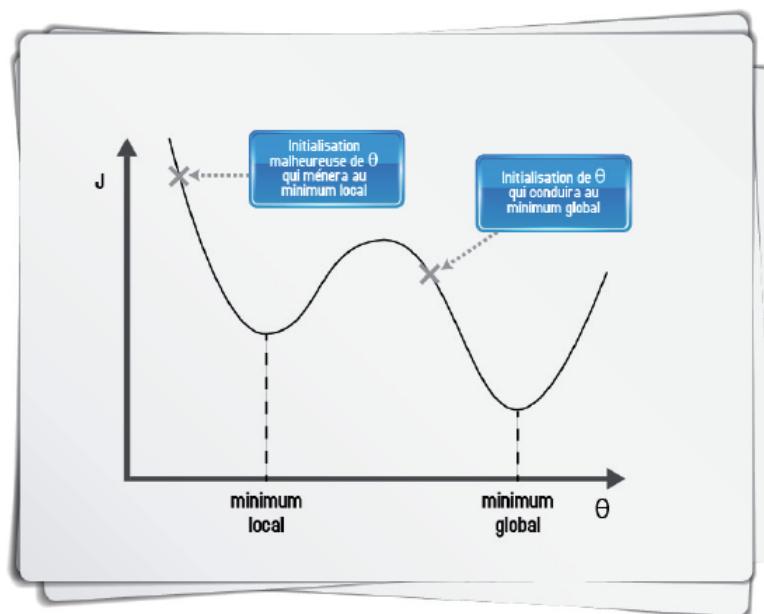


Figure 3-4 – Minima locaux et minima globaux d'une fonction non convexe

Que le lecteur se rassure, il n'aura jamais à coder sa propre descente de gradient car tous les outils de *machine learning* qui utilisent cette méthode d'optimisation rendent transparente cette étape. En revanche, il lui appartiendra souvent de choisir les méta-paramètres comme α (nous en verrons d'autres dans la suite de ce livre).

À RETENIR Régression linéaire univariée

Apprentissage supervisé – régression

L'approximation des données s'effectue selon la démarche suivante :

- on construit une fonction de coût correspondant à l'erreur induite par la modélisation ;
- la meilleure modélisation est celle qui minimise l'erreur ;
- une façon de minimiser l'erreur est la descente de gradient, méthode itérative qui vise à prendre la meilleure pente jusqu'au minimum de la fonction de coût.

Références

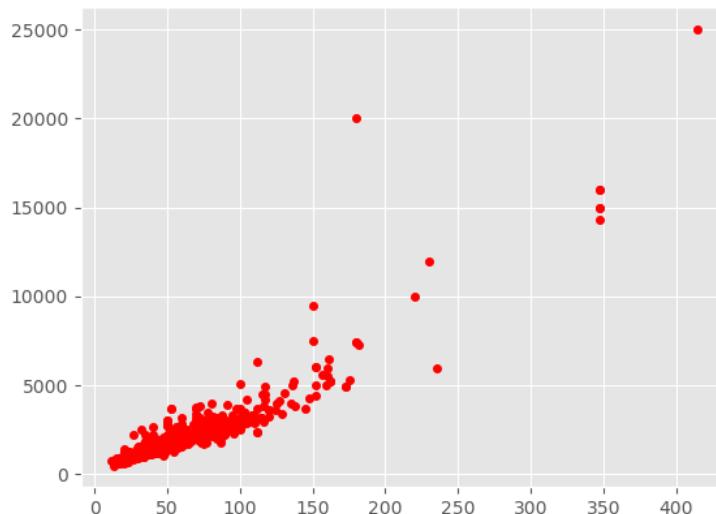
À propos de l'optimisation des fonctions convexes et de la descente de gradient :

- Cauchy A. 1847. Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes-rendus de l'Académie des Sciences de Paris*, série A, vol. 25, p. 536-538.
- Snyman JA. 2005. *Practical mathematical optimization: an introduction to basic optimization theory and classical and new gradient-based algorithms*. Springer.

La régression linéaire univariée

Commençons par charger et afficher les données d'entraînement, juste pour avoir une meilleure idée de ce à quoi on a affaire. Ce sont des données réelles de location, pour quelques arrondissements parisiens. On peut donc se concentrer sur la modélisation.

```
>>> # On importe les librairies dont on aura besoin pour ce tp
... import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> # On charge le dataset
>>> house_data = pd.read_csv('donnees/house.csv')
>>> # On affiche le nuage de points dont on dispose
>>> plt.plot(house_data['surface'], house_data['loyer'], 'ro',
markersize=4)
>>> plt.show()
>>>
>>> # On importe les librairies dont on aura besoin pour ce tp
... import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> plt.style.use('ggplot')
>>>
>>> # On charge le dataset
... house_data = pd.read_csv('donnees/house.csv')
>>>
>>> # On affiche le nuage de points dont on dispose
>>> plt.plot(house_data['surface'], house_data['loyer'], 'ro',
markersize=4)
>>> plt.show()
```



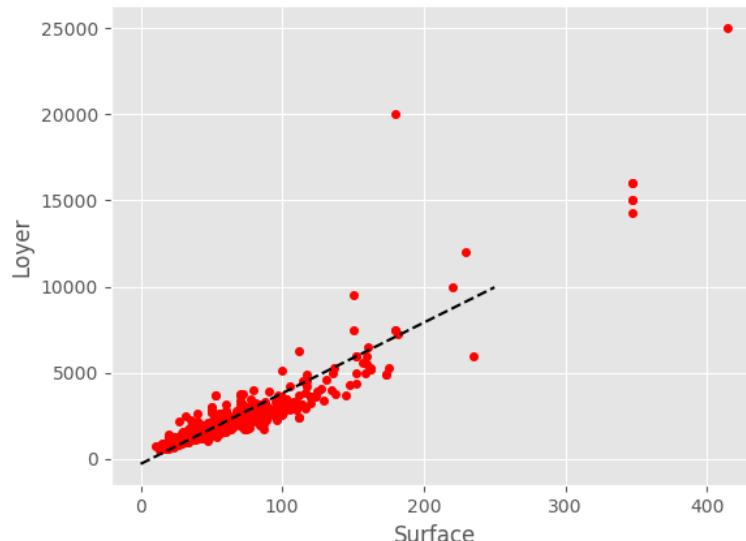
Clairement d'après la visualisation, on peut se dire que le montant du loyer dépend de manière linéaire de la surface du logement. On peut donc émettre une hypothèse de modélisation qui est que le phénomène possède la forme d'une droite. Bon en tout cas, on peut maintenant calculer les paramètres directement

```
>>> # On décompose le dataset et on le transforme en matrices
```

```
>>> #pour pouvoir effectuer notre calcul
>>> X = np.matrix([np.ones(house_data.shape[0]),
house_data['surface'].as_matrix()]).T
>>> y = np.matrix(house_data['loyer']).T
>>>
>>> # On effectue le calcul exact du paramètre theta
... theta = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
>>>
>>> print(theta)
[[ -283.37836117]
 [ 40.97116431]]
```

On peut représenter graphiquement la droite qu'on a trouvée pour vérifier qu'elle colle bien aux données

```
>>> plt.xlabel('Surface')
>>> plt.ylabel('Loyer')
>>> plt.plot(house_data['surface'], house_data['loyer'], 'ro',
markersize=4)
>>> # On affiche la droite entre 0 et 250
... plt.plot([0,250], [theta.item(0),theta.item(0) + 250 *
theta.item(1)], linestyle='--', c='#000000')
>>> plt.show()
```



Utiliser le modèle pour effectuer des prédictions

Maintenant qu'on a notre paramètre θ , c'est à dire qu'on a trouvé la droite qui correspond le mieux nos données d'entraînement, on peut effectuer des prédictions sur de nouvelles données, c'est à dire prédire le loyer en fonction de la surface qu'on nous donne en entrée, en appliquant directement la formule du modèle dessus.

Par exemple, si on l'applique pour une surface de 35m carré :

```
>>> theta.item(0) + theta.item(1) * 35
1150.612389740233
```

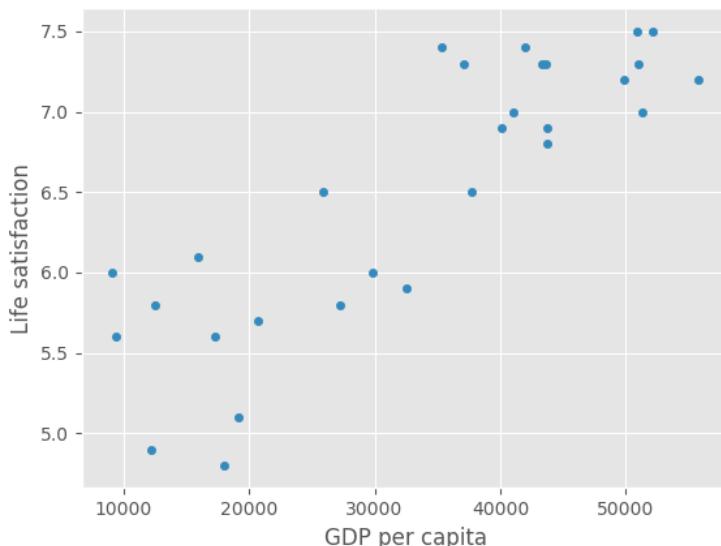
On vient ici de décomposer l'entraînement de la régression linéaire. En réalité, vous vous doutez bien que cette régression est déjà implémentée dans le package **scikit-learn**. On peut l'utiliser directement de la manière suivante :

```
>>> import matplotlib
>>> import matplotlib.pyplot as plt
```

```

>>> import numpy as np
>>> import pandas as pd
>>> import sklearn
>>>
>>> def prepare_country_stats(oecd_bli, gdp_per_capita):
...     oecd_bli = oecd_bli[oecd_bli["INEQUALITY"]=="TOT"]
...     oecd_bli = oecd_bli.pivot(index="Country",
...                               columns="Indicator", values="Value")
...     gdp_per_capita.rename(columns={"2015": "GDP per capita"}, inplace=True)
...     gdp_per_capita.set_index("Country", inplace=True)
...     full_country_stats = pd.merge(left=oecd_bli,
...                                   right=gdp_per_capita,
...                                   left_index=True, right_index=True)
...     full_country_stats.sort_values(by="GDP per capita",
...                                   inplace=True)
...     remove_indices = [0, 1, 6, 8, 33, 34, 35]
...     keep_indices = list(set(range(36)) - set(remove_indices))
...     return full_country_stats[["GDP per capita",
...                               'Life satisfaction"]].iloc[keep_indices]
...
>>> # Chargement des données
>>> oecd_bli = pd.read_csv("donnees/oecd_bli_2015.csv",
thousands=',')
>>> gdp_per_capita =
pd.read_csv("donnees/gdp_per_capita.csv",thousands=',',delimiter='\t',
',
encoding='latin1', na_values="n/a")
>>> # Préparation des données
... country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
>>> X = np.c_[country_stats["GDP per capita"]]
>>> y = np.c_[country_stats["Life satisfaction"]]
>>> # Visualisation des données
... country_stats.plot(kind='scatter', x="GDP per capita", y='Life
satisfaction')
>>> plt.show()

```



```
>>> # Sélection d'un modèle linéaire
```

```
... lin_reg_model = sklearn.linear_model.LinearRegression()
>>> # Entraînement du modèle
>>> lin_reg_model.fit(X, y)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)
>>> # Réalisation d'une prédiction pour Chypre
>>> X_new = [[22587]] # PIB par habitant pour Chypre
>>> print(lin_reg_model.predict(X_new)) # imprime [[ 5.96242338]]
[[ 5.96242338]]
```

La régression linéaire multivariée

Introduction

Si vous venez de lire le chapitre précédent, vous verrez que ce chapitre n'est qu'une généralisation des principes évoqués dans le cas univarié.

Nous profiterons de ce chapitre pour introduire deux notions nouvelles : la normalisation des données et une méthode de résolution analytique pour la recherche des meilleurs paramètres du modèle.

Le modèle en détail

Comme pour le cas de la régression linéaire univariée, on recherche la meilleure fonction hypothèse qui approximera les données d'entrée :

$$\begin{array}{ccc} \text{hypothèse } h & & \\ \text{valeur d'entrée } x & \longrightarrow & \text{valeur de sortie } y \end{array}$$

En revanche, on offre plus de variables en entrée du problème, qui constituent autant de degrés de liberté à la fonction h pour approximer au mieux les données d'entrée (rappelons que l'hypothèse d'approximer une grandeur avec seulement une variable est une hypothèse extrêmement forte et qui est rarement efficiente).

Compte tenu de ces nouvelles hypothèses, h prend une forme plus générale pour n variables d'entrée :

$$h(X) = \theta_0 + \theta_1 X_1 + \theta_2 X_2 + \dots + \theta_n X_n$$

Nos données d'entrée se représentent aisément sous forme d'une matrice de dimension (m,n) , telle que présentée dans le chapitre « Le B.A.-ba du *data scientist* » :

$$X = \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix}$$

x_{ij} correspond à la valeur prise par la variable j de l'observation i .

La fonction de coût est encore une fois une simple généralisation de ce que nous avons vu précédemment. Elle est de la forme :

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h(x_i) - y_i)^2$$

Et pour minimiser J , nous pouvons utiliser la méthode générale de la descente de gradient que nous rappelons :

- itération 0 : initialisation d'un vecteur $(\theta_0, \theta_1, \theta_2, \dots, \theta_n)$
- itérer jusqu'à convergence :

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ pour } j = 0, \dots, n$$

J étant toujours convexe, nous n'aurons pas de problème de minima locaux. En revanche, et c'est la nouveauté de ce chapitre, nous risquons de rencontrer un problème avec cette méthode de résolution si nos variables évoluent avec des échelles très différentes.

Normalisation

Prenons un exemple afin de mieux comprendre. Supposons que nous souhaitions prédire les prix d'appartements parisiens (vous n'y échapperez pas, c'est un des grands classiques de la « régression pédagogique » !). Vous disposez donc pour cela d'une base de données avec plusieurs milliers d'appartements et, pour simplifier, de seulement deux variables explicatives : le nombre de pièces et la superficie, qu'on nomme X_1 et X_2 . Les figures 4-1 et 4-2 montrent les distributions respectives de ces deux variables.

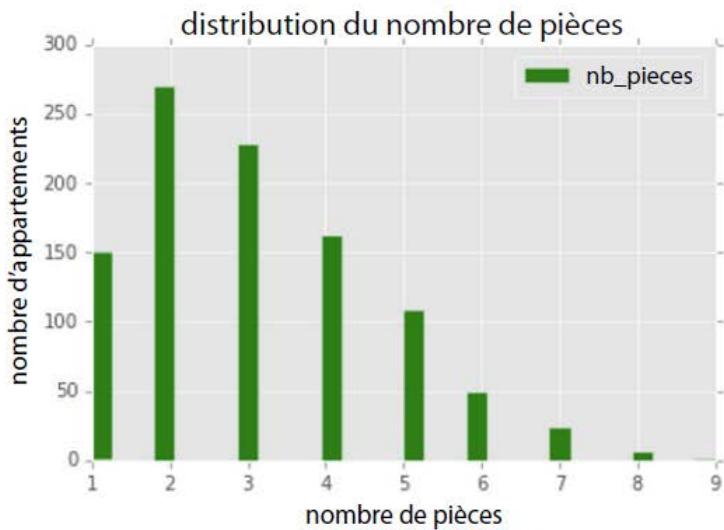


Figure 4-1 – Distribution du nombre de pièces

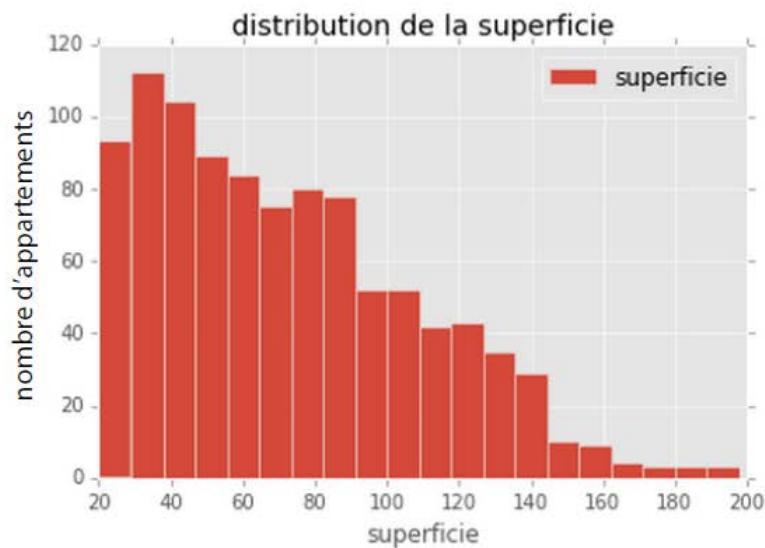


Figure 4-2 – Distribution de la superficie

On remarque que ces deux variables évoluent dans des proportions qui sont très différentes, X_1 variant de 1 à 8 et X_2 de 20 à 200.

Cette différence d'échelle posera problème pour certains algorithmes, ou tout du moins dans leur recherche de minima. En effet, on retrouvera ce rapport d'échelle entre les variables (environ 20 dans notre cas) sur la fonction J et l'algorithme de la descente de gradient pourra alors peiner à converger.

Remarquons simplement pour cela que θ_1 étant le poids accordé à la variable X_1 et θ_2 celui accordé à la variable X_2 , le facteur multiplicatif de θ_1 sera relativement petit par rapport à celui de θ_2 , par construction de la fonction de coût J . Dans un tel cas de figure, si l'on trace les contours de la fonction J dans l'espace (θ_1, θ_2) , on obtient une fonction semblable à celle de la figure 4-3.

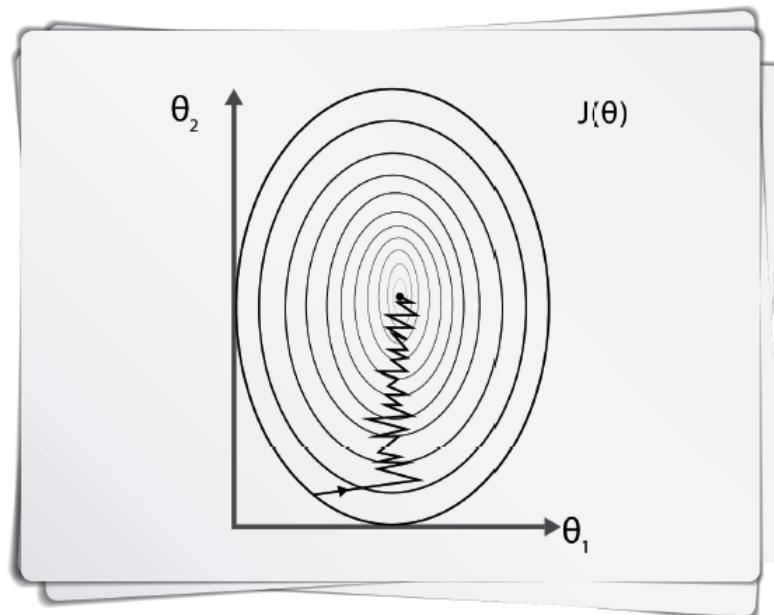


Figure 4-3 – Influence d'une mauvaise normalisation des variables

La descente de gradient demandera bien plus d'itérations dans ce cas : la topologie de la fonction de coût est en forme de « vallée », bien loin d'être optimale pour la convergence de la descente de gradient.

Pour résoudre ce problème, il convient de mettre les variables à la même échelle, par exemple entre -1 et 1 : on parle de normalisation (ou *scaling*). Pas la peine de coder vos propres fonctions de *scaling* ! La plupart des outils de *machine learning* offrent bon nombre de fonctions pour le faire. Par exemple, scikit-learn en offre nativement plusieurs.

- `sklearn.preprocessing.StandardScaler`

Ce module normalise les variables de sorte qu'elles aient une moyenne nulle et une variance égale à 1 (de même que l'écart-type). Pour une variable, cela correspond à retrancher à chaque observation la moyenne de la variable et à diviser chaque observation par l'écart-type de la variable. Ce procédé est nommé « centrage réduction » en statistique usuelle¹.

Revoyons la distribution de la superficie de nos appartements après normalisation :

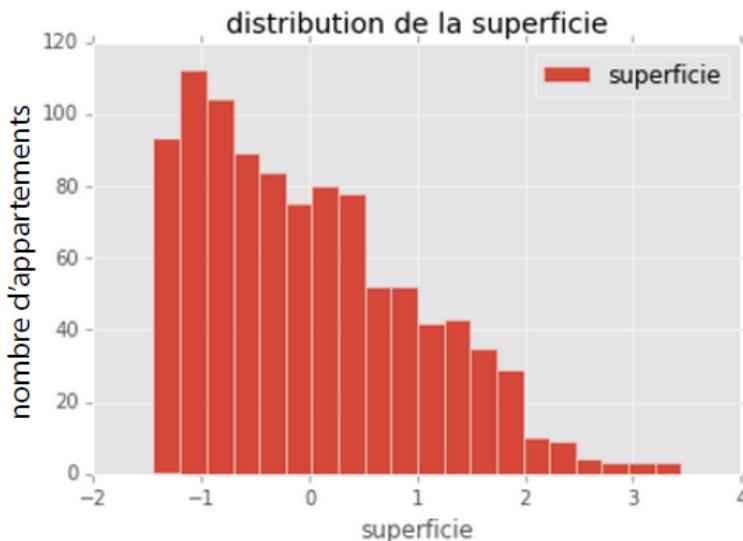


Figure 4-4 – Superficie normalisée, $\mu = 0$, $\sigma = 1$

- `sklearn.preprocessing.MinMaxScaler`

Ce module normalise les variables de sorte qu'elles évoluent entre 0 et 1², pratique si vous avez besoin de probabilité !

La standardisation est la suivante :

$$X_{std} = \frac{X - \min(X)}{\max(X) - \min(X)}$$

Voici la distribution de la superficie de nos appartements après ce *scaling* (StandardScaler est en arrière-plan pour comparaison) :

1. Il peut également être réalisé sous R avec la fonction `scale`, avec les arguments `center = TRUE` et `scale = TRUE`.

2. [0,1] est la valeur de sortie par défaut. Vous pouvez spécifier n'importe quel intervalle [a,b].

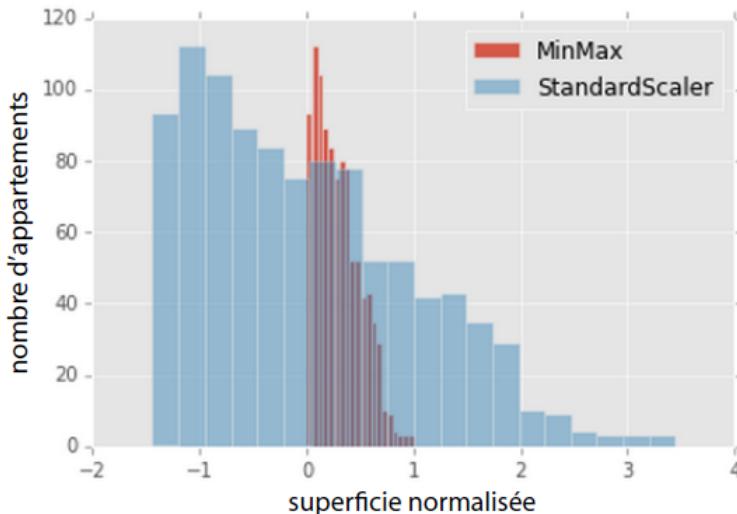


Figure 4-5 – Superficie normalisée, min = 0, max = 1

- `sklearn.preprocessing.Normalizer`

À titre indicatif, il existe d'autres façons de normaliser les données, comme ce module un peu spécial qui permet d'obtenir une norme unité (au sens l_1 ou l_2) pour chacune des observations. Il travaille donc en lignes en non pas en colonnes.

Ce *scaler* n'est pas utile pour réaliser la descente de gradient, mais il peut s'avérer utile dès que les distances relatives entre les observations sont en jeu, comme pour de la classification de textes ou du *clustering*. Plus de détails sur cette notion de norme unité sont donnés dans le chapitre 6.

Pour finir, voilà comment utiliser les différents *scalers* dans scikit-learn :

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
scaler = StandardScaler()
# house est le data frame contenant les variables appartements
new_house = scaler.fit_transform(house)
```

Dans le cadre de la descente de gradient appliquée à la régression linéaire multivariée, n'importe lequel des deux premiers *scalers* (le normalizer étant à part) fera l'affaire car il s'agit grossièrement de donner la même échelle aux données.

Résolution analytique

La méthode de la descente de gradient présentée ici est une approche numérique qui permet de trouver une solution au problème de modélisation. Elle est généralement utilisée en *machine learning* car particulièrement adaptée aux gros volumes de données et lâche en terme d'hypothèses de modélisation. Toutefois, sachez qu'il existe d'autres approches, analytiques quant à

elles, qui permettent aussi de résoudre la régression linéaire multivariée. Ce sont elles qui sont généralement mises en avant dans les ouvrages classiques de statistiques. Le paragraphe qui suit donne un aperçu de cette façon de faire.

Réécrivons tout d'abord quelques-unes de nos équations et hypothèses.

Si on pose $x_{i0} = 1 \forall i \in \{1, \dots, m\}$, on peut écrire notre hypothèse h sous cette forme :

$$h(x_i) = \theta_0 x_{i0} + \theta_1 x_{i1} + \theta_2 x_{i2} + \dots + \theta_n x_{in}$$

X est alors de dimension $(m, n+1)$, sous cette forme :

$$X = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \ddots & \vdots \\ 1 & x_{m1} & \cdots & x_{mn} \end{pmatrix}$$

Posons enfin Θ , vecteur de paramètres de dimension $n + 1$, ainsi que le vecteur Y de dimension $(m, 1)$ des valeurs à prédire :

$$\Theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix} \quad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

Avec ces notations, on peut définir l'erreur pour la i -ième observation, aussi appelée résidu³ par les « stateux ». On appellera ce terme r^4 .

$$r_i = y_i - \sum_{j=0}^n x_{ij} \theta_j$$

J devient alors :

$$J = \sum_{i=1}^m r_i^2$$

3. D'où la célèbre boutade : « Je suis un modèle, tous ceux qui s'éloignent de moi sont des résidus ! »

4. Il est aussi souvent désigné par le symbole ε dans les ouvrages de statistiques.

Et sa dérivée par rapport à θ_j :

$$\frac{\partial J}{\partial \theta_j} = 2 \sum_{i=1}^m r_i \frac{\partial r_i}{\partial \theta_j}$$

$$= 2 \sum_{i=1}^m \left(y_i - \sum_{k=0}^n x_{ik} \theta_k \right) (-x_{ij})$$

Notons $\hat{\Theta}$ le vecteur que nous recherchons. Il vérifie :

$$2 \sum_{i=1}^m \left(y_i - \sum_{k=0}^n x_{ik} \hat{\theta}_k \right) (-x_{ij}) = 0$$

Si les notations et les indices ne vous ont toujours pas donné mal à la tête, vous noterez qu'on peut arranger cette expression sous cette forme :

$$\sum_{i=1}^m \sum_{k=0}^n x_{ij} x_{ik} \hat{\theta}_k = \sum_{i=1}^m x_{ij} y_i$$

Cette expression importante, appelée « équation normale », est plus connue sous sa forme matricielle :

$$(X^T X) \hat{\Theta} = X^T Y$$

Si $(X^T X)$ est non singulière⁵, autrement dit, si elle est inversible, on dispose d'une solution analytique à notre problème, donnée par :

$$\hat{\Theta} = (X^T X)^{-1} X^T Y$$

Cette méthode est dite des « moindres carrés ».

Géométriquement, elle peut s'expliquer comme suit (Johnston et Dinardo, 1997). Soit une matrice X formant l'espace des observations et un vecteur Y de variables à expliquer. Y est généralement hors de X (sinon Y serait parfaitement explicable par X). Notons $\hat{Y} = X\hat{\theta}$ toute combinaison linéaire arbitraire des colonnes de X . Y peut alors être exprimé sous la forme $Y = \hat{Y} + r$. Le principe des moindres carrés consiste à choisir \hat{Y} de façon à minimiser la longueur du vecteur r . Elle est minimale lorsque \hat{Y} et r sont orthogonaux. La figure 4-6 résume cette géométrie.

5. Rappel : en algèbre linéaire, une matrice est dite carrée (ce qui est le cas de $X^T X$) si elle possède le même nombre de lignes que de colonnes. De plus, elle est dite singulière si elle n'est pas inversible, ou encore, ce qui est parfaitement équivalent, si ses colonnes ne sont pas linéairement indépendantes. La dépendance linéaire signifie qu'une des variables est une combinaison linéaire d'autres variables (nous proposerons des pistes pour éviter ce type de problème au Chapitre 15).

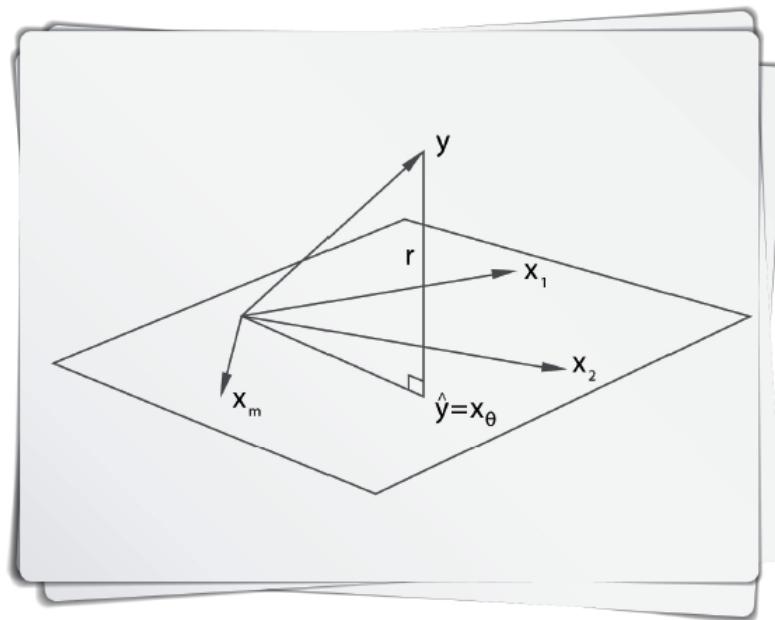


Figure 4-6 – La géométrie des moindres carrés

Pour une telle résolution analytique, les statisticiens font peser des hypothèses fortes sur les caractéristiques du vecteur r (entre autres : un modèle statistique repose également sur d'autres hypothèses de modélisation). Il doit en effet se comporter comme une suite de variables indépendantes de même loi normale $N(0, \sigma^2)$. Cette hypothèse est fondamentale pour l'interprétation probabiliste qu'ils ont de la régression. Elle implique des pratiques qui n'intéressent pas toujours les *machine learners* : analyse de la variance, calcul de la loi des estimateurs, calcul des intervalles de confiance et de prévision d'une valeur prédictive, etc.

Lorsque les hypothèses concernant r ne peuvent être respectées (non normalité des résidus, non indépendance – comme c'est souvent le cas pour les séries temporelles –, etc.), on emploie des alternatives aux moindres carrés ordinaires telles que les moindres carrés généralisés ou le maximum de vraisemblance, pour les plus connues d'entre elles. Elles permettent d'assouplir certaines hypothèses et/ou de construire une plus large variété de modèles. Pour en savoir plus, n'hésitez pas à vous jeter sur un manuel de statistique !

Pour finir, remarquons que la méthode analytique a une complexité de type $O(n^3)$. Ce qu'il faut comprendre, c'est qu'elle peut devenir compliquée à mettre en œuvre pour des problèmes de grandes dimensions (c'est-à-dire avec beaucoup de variables). Mais à l'ère des *big data*, tout le monde a un gros ordinateur, n'est-ce pas ? Sinon, rappelons alors que la méthode de la descente de gradient reste une bonne alternative, généralisable et très appropriée à des analyses à grande échelle.

La régression polynomiale

Introduction

Ce chapitre présente une forme particulière de régression linéaire multivariée : la régression polynomiale. L'intérêt de cette méthode est de pouvoir introduire de la non-linéarité dans un modèle pourtant linéaire. Notez toutefois qu'il existe des méthodes non linéaires bien plus performantes, que nous évoquerons plus loin dans ce livre.

Plus important encore, ce chapitre permet surtout de mettre en exergue de façon intuitive deux problématiques clés en *machine learning* : le surapprentissage et le compromis biais-variance. Toutefois, retenez bien que ces problématiques ne sont pas propres à la régression polynomiale. Il vous faudra les envisager quel que soit l'algorithme de modélisation que vous emploierez !

Principes généraux de la régression polynomiale

La régression polynomiale est une extension de la régression linéaire multivariée. Elle permet de lier les variables par un polynôme de degré k . Un polynôme est tout simplement la somme de plusieurs expressions de la forme ax^k , où a est un nombre réel (ou complexe) et k un entier naturel. On ne dirait pas comme ça, mais un polynôme est un objet mathématique drôlement pratique puisqu'il permet d'introduire de la non-linéarité dans les relations entre variables. Voici par exemple un polynôme dit du premier degré (c'est-à-dire que $n = 1$) : $y = -10 - 3x$. Il correspond à une droite affine simple (figure 5-1).

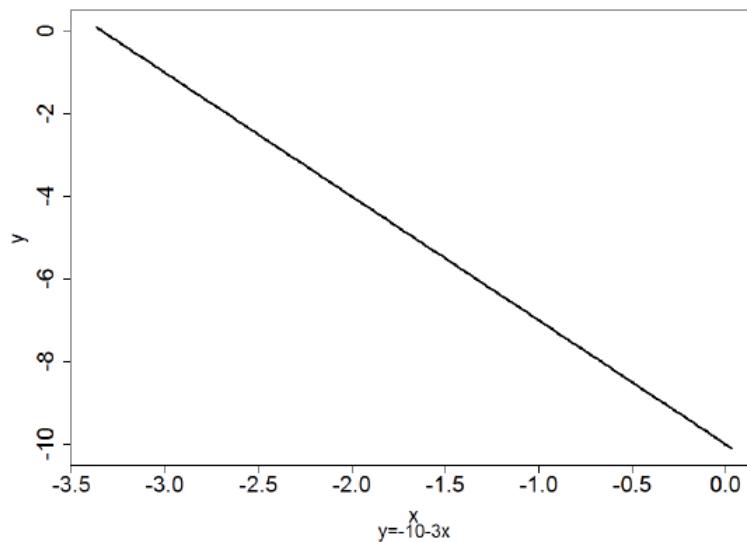


Figure 5-1 – $y = -10 - 3x$: un polynôme de degré 1

Rajoutons-y un terme en $k = 2$, par exemple $y = -10 - 3x + x^2$, ou en $k = 3$, par exemple $y = -10 - 3x + x^2 + 5x^3$. Nous obtenons la figure 5-2.

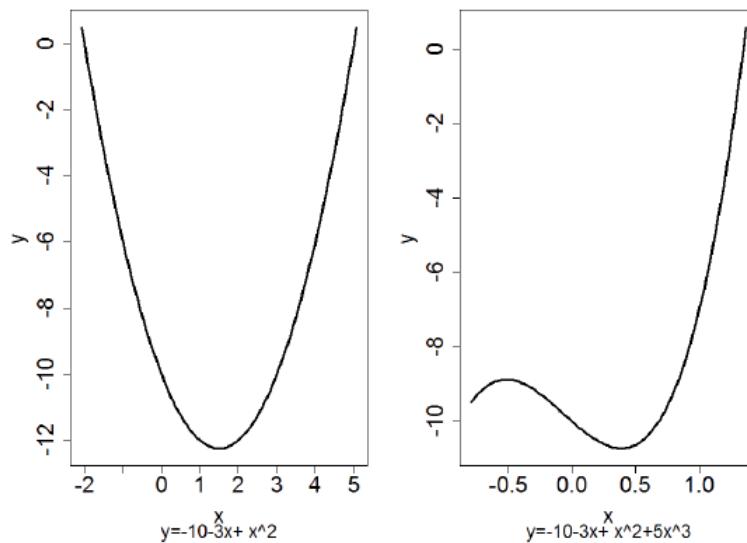


Figure 5-2 – Polynômes de degrés 2 (gauche) et 3 (droite) : des relations non linéaires entre x et y

Intuitivement, chaque nouvel ordre va permettre d'ajouter un « pli » à la courbe.

Selon la notation déjà employée, pour calculer $h(x)$, on évalue chaque variable en l'associant à tous les degrés polynomiaux de 1 à k. Chacun de ces polynômes a son propre coefficient. Par exemple, un modèle polynomial de degré 2 à deux variables explicatives s'écrira :

$$h(X) = \theta_0 + \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_1^2 + \theta_4 X_2^2$$

En pratique, les calculs sont les mêmes que dans le cas de la régression multiple traditionnelle (la régression linéaire n'est en fait qu'une régression polynomiale de degré 1). L'introduction de termes polynomiaux dans un modèle de régression permet donc de modéliser simplement des relations potentiellement très complexes.

Voici dans la figure 5-3 un exemple tout simple à partir du jeu de données R cars. On souhaite expliquer la distance d nécessaire à l'arrêt d'un véhicule en fonction de sa vitesse v .

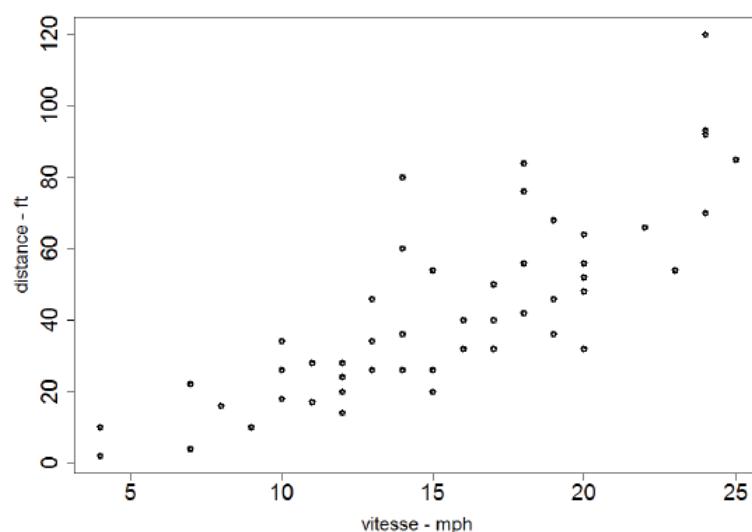


Figure 5-3 – Données cars : la distance de freinage en fonction de la vitesse d'un véhicule

Ne vous effrayez pas à la vue du jeu de données cars, qui provient d'enregistrements des années 1920. Vous êtes bien plus en sécurité avec votre véhicule actuel ! Les lecteurs férus de physique sauront que cette distance dépend du carré de la vitesse, selon la formule $d = \frac{v^2}{2F}$, où F est une constante que nous ne détaillerons pas et qui dépend du coefficient de friction et du temps de réaction du conducteur. Visuellement, le graphique ci-dessus semble confirmer cette non-linéarité. Cherchons donc un modèle polynomial d'ordre 2, grâce au code R suivant :

```
model1 <- lm(dist ~ poly(speed, 2, raw = TRUE), data = cars)
```

On peut aussi utiliser une autre syntaxe, qui donnera exactement le même résultat :

```
model2 <- lm(dist ~ speed + I(speed^2), data = cars)
```

Cela nous permet d'obtenir un ajustement illustré par la figure 5-4 suivante (l'ajustement obtenu par régression linéaire classique est indiqué à titre de comparaison).

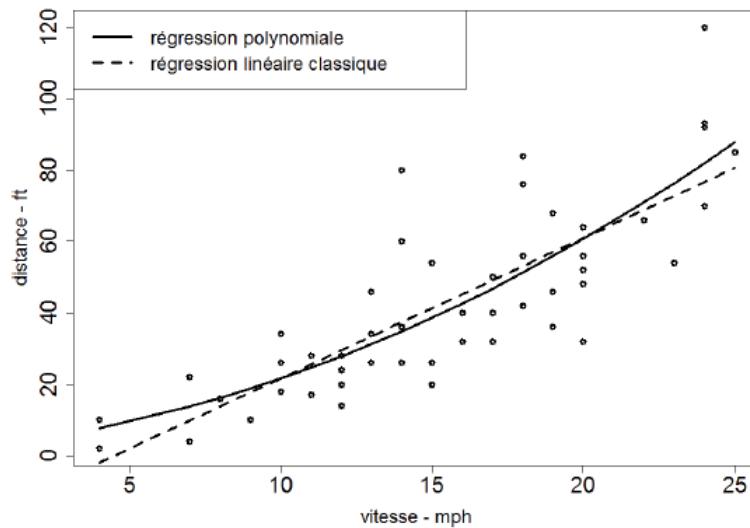


Figure 5-4 – Données cars : la régression polynomiale d'ordre 2

La fonction `poly` est un peu plus pratique à utiliser, puisqu'en cas d'ordre polynomial élevé, tous les degrés seront automatiquement spécifiés, contrairement à la seconde syntaxe qui nécessite qu'elles soient toutes indiquées explicitement. De plus, si l'on passe l'argument `raw = FALSE` à la fonction `poly`, on évalue un polynôme orthogonal, c'est-à-dire qu'on va limiter la corrélation entre les différents degrés polynomiaux. Ceci aura pour effet de fiabiliser les résultats de l'estimation (on ne développera pas les détails de cet algorithme particulier dans ce livre, mais pour plus de détails concernant les effets de la corrélation entre variables explicatives, lisez le chapitre 15).

La notion de surapprentissage

Attention, même si elle peut formidablement améliorer l'ajustement d'un modèle, la régression polynomiale n'est pas sans risque. En effet, si l'on choisit un degré de polynôme trop grand, on risque de construire une fonction qui passera par un nombre élevé de points des données d'apprentissage, mais selon une forme très oscillante. Une fonction oscillant trop fortement ne permettra pas de généraliser le modèle et donnera des résultats en prévision très mauvais. La petite simulation R ci-après va illustrer cette proposition. Simulons un jeu de données très simple de 10 points¹ :

- soit un vecteur x allant de 1 à 10 ;
- et un vecteur y tel que : 0,5 ; 2,5 ; 2,5 ; 4,5 ; 4,5...

Ajustons un modèle pour expliquer y en fonction de x selon trois formes de modèles différentes :

- une régression linéaire simple ;
- une régression polynomiale d'ordre 3 ;
- et une régression polynomiale d'ordre 9.

Utilisons ensuite ces trois modèles pour réaliser des prédictions de y à partir d'un vecteur z contenant 250 points répartis à intervalles réguliers entre 1 et 10 (1,00 ; 1,03 ; 1,07). Voici le code de la simulation :

```

x <- 1:10
y <- x + c(-0.5, 0.5)

plot(x, y)

## Modèles
model1 <- lm(y~x) # Régression linéaire simple
model2 <- lm(y~poly(x, 3)) # Régression polynomiale d'ordre 3
model3 <- lm(y~poly(x, 9)) # Régression polynomiale d'ordre 9

## Prévisions pour 250 points allant de 1 à 10
z <- seq(1, 10, length.out = 250)
lines(z, predict(model1, data.frame(X = z)), lty = 1)
lines(z, predict(model2, data.frame(X = z)), lty = 2, lwd = 2)
lines(z, predict(model3, data.frame(X = z)), lty = 2, )

legend('bottom', c('régression linéaire classique',
                   'régression polynomiale - ordre 3',
                   'régression polynomiale - ordre 9',
                   'données d\'apprentissage'),
       lty = c(1, 2, 2, NA),
       lwd = c(1, 2, 1, NA),
       pch = c(NA, NA, NA, 1))

```

1. Simulation inspirée par cette discussion : <http://stackoverflow.com/questions/3822535/fitting-polynomial-model-to-data-in-r>

Le résultat montré dans la figure 5-5 est assez parlant.

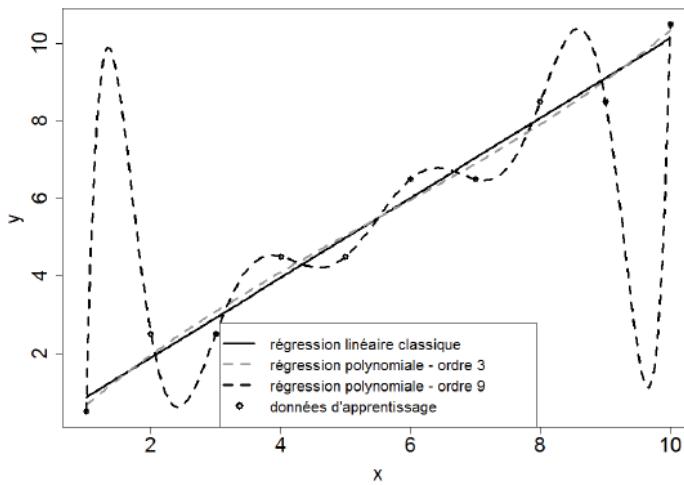


Figure 5-5 – La notion de surapprentissage : résultat de la simulation 1

Si l'on se fie uniquement aux données d'apprentissage, la régression polynomiale d'ordre 9 est celle qui présente le meilleur ajustement, puisqu'elle passe exactement par tous les points (x, y) . Par contre, pour l'ensemble des prévisions, elle fournit des valeurs plus que douteuses. Un tout petit changement de valeur au niveau des x va entraîner d'énormes écarts dans les prévisions des y . Par rapport à la forme générale du nuage de points $y = f(x)$, tout laisse à croire que cette régression polynomiale d'ordre 9 est hasardeuse. A contrario, les régressions linéaire et polynomiale d'ordre 3 semblent plus fiables. Elles ne passent pas parfaitement par les données d'apprentissage, mais elles fournissent une représentation globale « moyenne » crédible.

En outre, un petit changement dans les données d'apprentissage va engendrer des ajustements très différents pour la régression polynomiale d'ordre 9 : le modèle est instable. Modifions le vecteur x de notre simulation :

```
x <- c(1:5, 10:15)
```

Nous obtenons les résultats de simulation montrés dans la figure 5-6.

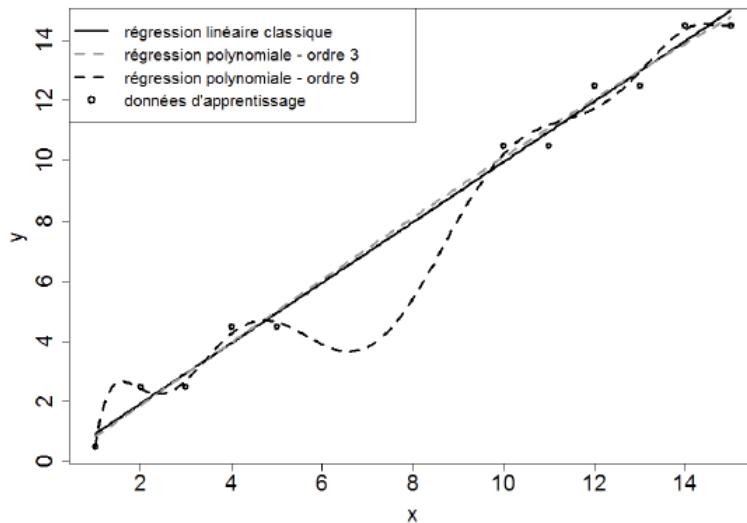


Figure 5-6 – La notion de surapprentissage : résultat de la simulation 2

Les prévisions de la simulation 2 sont proches de celles de la simulation 1 pour les modèles de régression linéaire classique et de régression polynomiale d'ordre 3. En revanche, les résultats obtenus pour la régression polynomiale d'ordre 9 sont totalement différents dans les deux simulations. Cette très forte variabilité, soumise aux variations des données d'apprentissage, entrave fortement la confiance qu'on peut accorder à ce modèle.

Le cas de la régression polynomiale d'ordre 9 se nomme surapprentissage (ou overfitting en anglais). Il caractérise un modèle qui décrit très bien les données disponibles, mais qui s'extrapolate très mal à d'autres données non utilisées pour le calcul du modèle, tout en étant fortement instable. Il correspond à une sorte d'apprentissage « bête et méchant » des données utilisées pour la définition du modèle, sans possibilité de généralisation. Un peu comme un étudiant qui apprendrait des annales d'examen par cœur, sans savoir appliquer les raisonnements dans d'autres contextes d'application.

Le compromis biais-variance

Le surapprentissage est la bête noire de tout *data scientist*. Dans la réalité, la distinction entre un bon modèle et un modèle overfitté n'est pas toujours évidente. Le *data scientist* a besoin d'un minimum d'outillage pour qualifier le surapprentissage. Pour cela, on s'appuie souvent sur ce qu'on nomme le compromis biais-variance. Ce compromis va permettre de trouver un juste équilibre entre la complexité du modèle et sa capacité à se généraliser.

Divisons pour cela notre jeu de données en deux. Une partie sera utilisée pour calculer le modèle, une autre pour effectuer des prévisions (voir le chapitre 14 pour plus de détails sur les différentes stratégies de division des données). On va alors mesurer deux types d'erreurs : l'erreur de modélisation d'une part et l'erreur de prévision d'autre part. Calculons ces erreurs avec différents degrés de complexité de modèles, en augmentant par exemple progressivement les degrés d'une régression polynomiale. Comme nous l'avons montré plus haut, l'erreur de modélisation va diminuer avec la complexité du modèle (la régression polynomiale d'ordre 9 passe par tous les points d'apprentissage de la simulation 1). L'erreur de prévision se comportera par contre différemment : elle va diminuer tant que l'augmentation de la complexité améliorera également les résultats en prévision, mais elle va ensuite augmenter à partir d'un certain degré de complexité. Elle augmente en fait lorsque le modèle commence à surapprendre, c'est-à-dire quand il perd sa capacité de généralisation et qu'il devient instable. Le modèle optimal est celui qui minimise les deux erreurs, car il décrit au mieux les données d'apprentissage et il est capable de faire les meilleures prévisions possibles tout en restant robuste. La figure 5-7 résume cette situation.

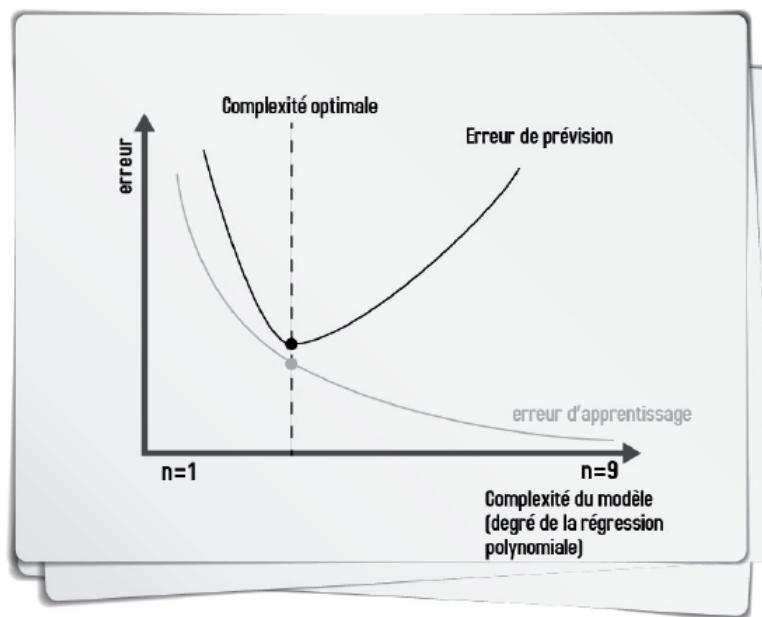


Figure 5-7 – Degré de complexité optimal d'un modèle en fonction des erreurs d'apprentissage de prévision

Ce degré de complexité optimale correspond à ce que l'on appelle le meilleur compromis biais-variance. Les modèles à gauche de la figure correspondent en général à des modèles à fort biais, mais à faible variance : l'écart entre les données modélisées et les vraies valeurs est fort. En contrepartie, la prévision est stable : de petits changements dans les données feront peu varier la prédiction pour un point donné. C'est une situation de sous-apprentissage : le modèle pourrait être amélioré en augmentant sa complexité. À l'inverse, les modèles à droite de la figure sont ceux à faible biais, mais à forte variance : ils décrivent très bien les données, mais sont très instables en prévision. C'est la situation de surapprentissage déjà évoquée, et il faudra simplifier le modèle.

Notez que cette problématique a été introduite à partir de l'algorithme de régression polynomiale, mais en réalité, elle apparaît dans toute situation de modélisation. Tous les autres algorithmes présentés dans ce livre sont concernés et il convient de toujours s'assurer de trouver le meilleur compromis biais-variance.

À RETENIR Régression polynomiale

Apprentissage supervisé – régression

La régression polynomiale est un cas particulier de modèle linéaire où de la non linéarité est introduite avant l'apprentissage d'un modèle linéaire. Elle permet de modéliser des relations complexes, pouvant conduire à de l'*overfitting*. L'*overfitting*, bête noire du *data scientist*, se caractérise par un modèle qui apprend par cœur les données d'entrées, mais qui est peu stable et n'a aucune capacité de généralisation.

Le biais d'un modèle se caractérise par son erreur sur les données d'apprentissage. La variance d'un modèle est une mesure de sa stabilité et donc de l'écart qu'on observera entre l'erreur d'apprentissage et l'erreur de prévision.

Le compromis biais-variance permet de trouver un juste équilibre entre la complexité du modèle et sa capacité à généraliser.

Référence

Un beau post sur le compromis biais-variance :

- <http://scott.fortmann-roe.com/docs/BiasVariance.html>

La régression régularisée

Introduction

Nous avons vu que le modèle de régression linéaire multivarié se base sur la minimisation de la variance résiduelle pour l'estimation de ses coefficients. Mais la fonction de coût présentée dans le chapitre précédent peut engendrer de l'instabilité dans les résultats de l'estimation. Pourquoi ? Parce qu'aucune contrainte n'est appliquée aux paramètres du modèle. On peut donc obtenir des paramètres avec de grandes valeurs qui vont former un espace de solutions très étendu. Plus particulièrement, ce type d'espace peut générer des « arêtes » ou des « vallées » dans sa topologie. Ces arêtes sont problématiques. En effet, plusieurs vecteurs d'estimations peuvent aboutir à des solutions proches en termes de minimisation de l'erreur, comme le montre la figure 6-1 qui suit.

En conséquence, de faibles changements dans les données (même un changement d'arrondi !) peuvent produire des modèles très différents.

Pour limiter ces problèmes, on a recours aux méthodes de régularisation. Ces méthodes vont en quelque sorte « distordre » l'espace des solutions pour empêcher les valeurs trop grandes. Ce « rétrécissement » (on parle d'ailleurs aussi de méthodes de *shrinkage*) va resserrer l'espace des solutions autour de zéro (les données ont bien entendu été préalablement normalisées) et, en conséquence, les vallées d'instabilité vont disparaître (remarque : le terme constant θ_0 du modèle n'est pas concerné). Pour opérer cette transformation spatiale, on va modifier un petit peu la fonction de coût du problème de régression en la complétant par une fonction de pénalité, telle que :

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h(x_i) - y_i)^2 + P(\lambda, \Theta)$$

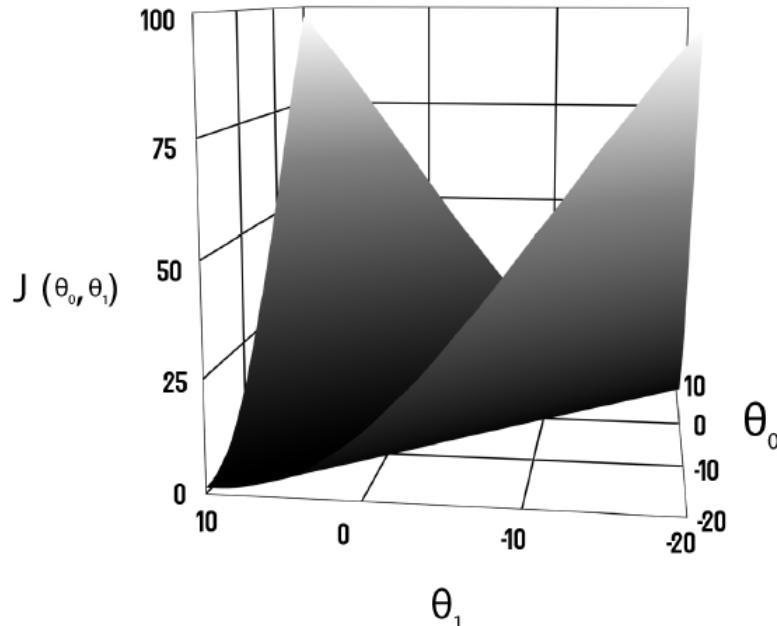


Figure 6-1 – Le risque d'instabilité dans les résultats d'une régression linéaire classique

(Cette figure a été inspirée par cette discussion : <http://stats.stackexchange.com/questions/118712/why-does-ridge-estimate-become-better-than-ols-by-adding-a-constant-to-the-diago/120073#120073>)

C'est la fonction $P(\lambda, \Theta)$ qui va gérer la pénalité, selon un paramètre λ que l'on fixe empiriquement de façon à obtenir les meilleurs résultats (à ce sujet, voir le chapitre sur la validation croisée).

Nous présentons dans ce chapitre trois méthodes s'appuyant sur ce principe : la régression *ridge*, le *LASSO* et *ElasticNet*.

La régression ridge

Différentes fonctions de pénalité existent. La plus intuitive est la pénalisation *ridge*, initialement proposée pour limiter l'instabilité liée à des variables explicatives trop corrélées entre elles (voir le chapitre 15 pour plus de détails). Cette fonction se base sur la norme dite l_2 du vecteur des paramètres, que l'on va noter $\|\Theta\|_{l_2}$. En mathématiques, la norme est une fonction qui permet d'assigner une longueur à un vecteur. La norme l_2 est intuitive, parce qu'elle correspond à la distance euclidienne que nous avons l'habitude d'utiliser dans notre quotidien, donnée par la relation suivante :

$$\|\Theta\|_{l_2} = \sqrt{\Theta_1^2 + \dots + \Theta_n^2}$$

Cette mesure de distance est celle que l'on apprend dès le collège. Pour mesurer la distance dans un plan entre deux points A et B, de coordonnées respectives (x_a, y_a) et (x_b, y_b) , on calcule $\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$.

De vieux souvenirs, n'est-ce pas ?

Donc, pour en revenir à notre problème de régularisation, la pénalisation *ridge* va diminuer la distance entre les solutions possibles, sur la base de la mesure euclidienne. Plus précisément, pour des raisons théoriques que nous ne détaillerons pas, c'est le carré de la norme l_2 qui est utilisé. La régression *ridge* revient donc à minimiser la fonction de coût suivante :

$$J(\theta)_{\text{ridge}} = \frac{1}{2m} \sum_{i=1}^m (h(x_i) - y_i)^2 + \lambda \sum_{j=1}^n \Theta_j^2$$

En résultat, on obtiendra un espace des solutions semblable à celui de la figure 6-2.

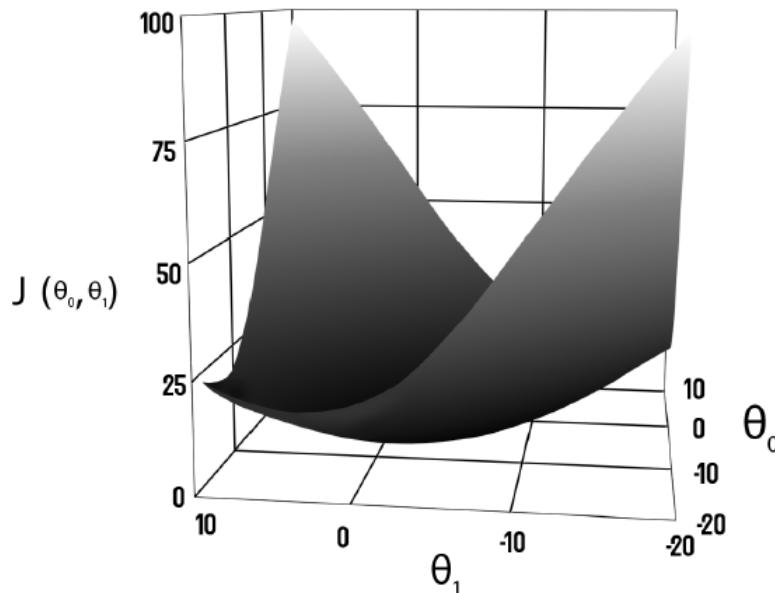


Figure 6-2 – La régression ridge stabilise les résultats du modèle linéaire

(Cette figure a été inspirée par cette discussion : <http://stats.stackexchange.com/questions/118712/why-does-ridge-estimate-become-better-than-ols-by-adding-a-constant-to-the-diago/120073#120073>)

Comme vous pouvez le voir, on n'a plus de vallée : la solution converge avec stabilité vers un optimum global. Le paramètre λ est supérieur à zéro. Quand il est proche de zéro, on s'approche de la solution « classique », non pénalisée, et quand λ tend vers l'infini, la pénalisation est telle que tous les paramètres valent zéro. En augmentant λ , on augmente le biais de la solution, mais on diminue sa variance. Plus de détails sur ce compromis biais-variance ont été donnés dans le chapitre précédent.

Tout comme la régression linéaire classique, la régression *ridge* peut être résolue par descente de gradient en itérant jusqu'à convergence pour $J(\theta)_{\text{ridge}}$, ou analytiquement. Dans ce dernier cas, cela revient à la solution suivante :

$$\hat{\Theta} = (X^T X + \lambda I)^{-1} X^T Y$$

où I représente la matrice identité. I est de dimension $(n+1, n+1)$ et ceci permet d'associer un coefficient égal à λ à chaque paramètre du modèle.

Le LASSO

Sous ce nom très western (qui signifie Least Absolute Shrinkage and Selection Operator), on retrouve le principe de la pénalisation à l'aide d'une fonction $P(\lambda, \Theta)$. Simplement, elle est définie par une autre norme. On va cette fois-ci utiliser la norme l_1 , ce qui nous amène à sortir de notre intuition courante des distances. En effet, en mathématiques, une distance peut être donnée par un ensemble d'applications, dont la distance euclidienne standard n'est qu'un cas particulier. La formulation générale de la distance dans un espace de n dimensions, pour tout p , est donnée par la relation $\left(|x_1|^p + \dots + |x_n|^p \right)^{1/p}$. Pour $p = 2$, nous retrouvons la norme l_2 de la régression *ridge*. Dans le cas du LASSO, la pénalisation va diminuer la distance entre les solutions possibles sur la base d'une norme l_1 , c'est-à-dire pour $p = 1$. Nous aurons donc :

$$\|\Theta\|_{l_1} = |\Theta_1| + \dots + |\Theta_n|$$

Cette norme est aussi appelée distance de Manhattan (ou *city-block* ou *taxi-distance* : voir également le chapitre concernant le *clustering*). Elle correspond à un déplacement à angle droit sur un damier, contrairement à la distance euclidienne qui correspond à un déplacement en ligne droite, comme le montre la figure 6-3.

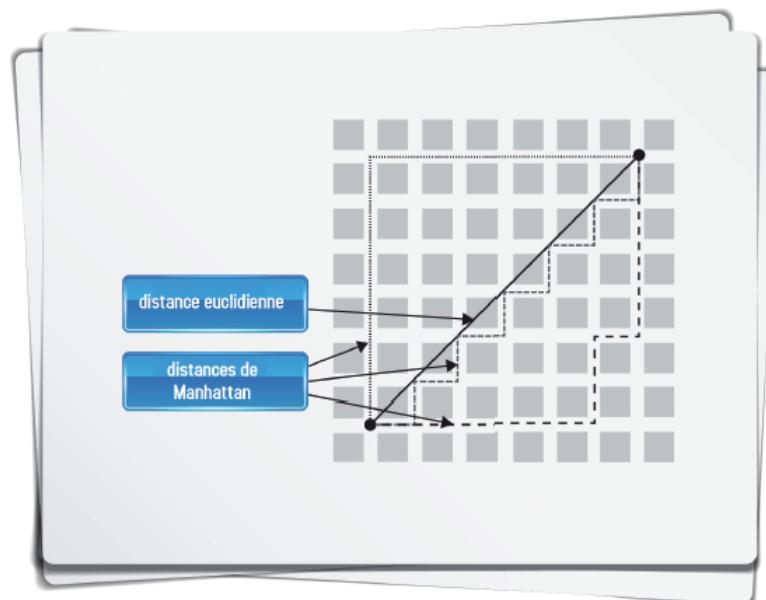


Figure 6-3 – Distance entre deux points : distance euclidienne et distance de Manhattan (plusieurs chemins possibles pour cette dernière)

La fonction de coût à minimiser dans le cas du LASSO est donc :

$$J(\theta)_{LASSO} = \frac{1}{2m} \sum_{i=1}^m (h(x_i) - y_i)^2 + \lambda \sum_{j=1}^n |\Theta_j|$$

Il n'y pas de solution analytique pour le LASSO. On pourra donc utiliser la descente de gradient bien connue, ou d'autres algorithmes itératifs tels que la *least angle regression*, qui est hors du périmètre de ce livre (voir Efron *et al.*, 2004). Néanmoins, le LASSO a une propriété intéressante : c'est une forme de pénalisation qui a la capacité de fixer des coefficients à zéro (contrairement à la régression *ridge*, qui pourra aboutir à des coefficients très proches de 0, mais jamais strictement nuls). Il participe ainsi à la simplification du modèle en éliminant des variables. Pour une explication détaillée et géométrique de cette différence fondamentale entre *ridge* et LASSO, référez-vous à l'article de Tibshirani (1996). Il propose notamment une figure indiquant la forme de ces fonctions de pénalisation. On en tire une bonne intuition de l'effet des types de régularisation sur les coefficients du modèle, comme le montre la figure 6-4.

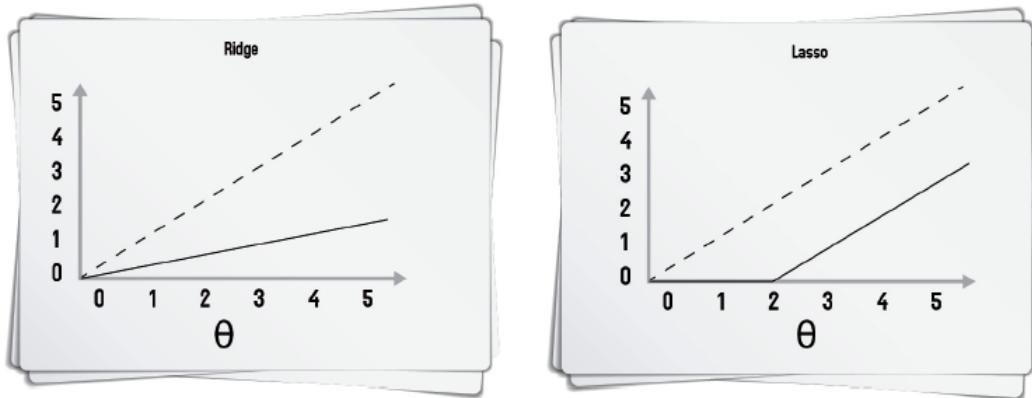


Figure 6-4 – Les effets des régularisations ridge et LASSO sur les paramètres du modèle

Le trait plein représente la fonction de régularisation, le trait pointillé représente une droite non régularisée. On voit que la régression *ridge* redimensionne les coefficients en les divisant par un facteur constant, tandis que le LASSO soustrait un facteur constant, en tronquant à 0 en-deçà d'une certaine valeur.

Ridge + LASSO = ElasticNet

En pratique, la régression *ridge* donne des meilleurs résultats que le LASSO lorsque les variables sont corrélées entre elles (à la base, cette méthode a été proposée pour faire face à ce cas particulier). Mais si on utilise la régression *ridge*, on ne peut pas réduire le nombre de variables. Pour sortir de ce dilemme, la régularisation *ElasticNet* a été proposée par Zou et Hastie en 2005, qui combine les deux approches. La fonction de coût devient :

$$J(\theta)_{\text{ElasticNet}} = \frac{1}{2m} \sum_{i=1}^m (h(x_i) - y_i)^2 + \lambda \sum_{j=1}^n \left[\frac{1}{2}(1-\alpha)\theta_j^2 + \alpha |\theta_j| \right]$$

Le paramètre α est compris entre 0 et 1 et va permettre de définir l'équilibre entre *ridge* et LASSO. Pour $\alpha = 1$, la fonction de coût correspondra à celle du LASSO et pour $\alpha = 0$, à celle de la régression *ridge*. Ainsi, on pourra régler la pénalisation en fonction du cas d'application. Avec α proche de 1, on pourra avoir un comportement proche du LASSO, tout en éliminant les problèmes liés aux fortes corrélations entre variables. Quand α augmente de 0 à 1, pour une valeur de λ fixe, le nombre de variables retirées du modèle (donc de coefficients nuls) augmente de façon monotone, de zéro jusqu'au modèle le plus réduit obtenu par LASSO.

À RETENIR Régression régularisée

Apprentissage supervisé – régression

La régularisation permet de « rétrécir » l'espace formé par les solutions du problème de modélisation. Pour ce faire, on injecte dans la fonction un terme qui pénalisera les coefficients. Minimiser la fonction de coût minimisera de facto les coefficients de la régression.

Il existe trois standards de régularisation :

- *ridge*, basée sur le carré de la norme l_2 ; elle rapproche globalement l'ensemble des solutions autour de 0 ;
- LASSO, basée sur la norme l_1 ; plus sévère que *ridge*, elle a pour effet d'annuler certains coefficients ;
- et *ElasticNet*, qui est une combinaison des deux régularisations précédentes.

7

Les espaces de grande dimension

Introduction

La dimension d'un problème de *machine learning* correspond au nombre n de variables de la matrice X . Un n grand peut poser de nombreux pièges que le *data scientist* doit savoir surmonter. Le traitement de ces espaces fait en effet partie de son travail. En général, on cherche à sélectionner judicieusement un sous-espace pertinent, pour améliorer la modélisation¹. L'objectif est de conserver le maximum de l'information contenue dans les données, avec un minimum de variables. Pour cela, deux approches sont envisageables.

- On peut sélectionner un nombre restreint des variables les plus importantes. Certains algorithmes intègrent directement cette opération au travers de leurs paramètres de modélisation comme le lambda de la régression *ridge*. Sinon, on intègre une phase préliminaire de sélection de variables, avant application d'un algorithme. Plusieurs méthodes existent pour cela et nous en présenterons quelques-unes dans ce chapitre.
- Ou bien on peut créer des variables « synthétiques » à partir des variables initiales. En conséquence, on n'utilisera plus les variables observées, mais on reconstruira un nouvel espace de dimension réduite, en manipulant les variables d'origine. Il existe de nombreuses méthodes pour cela. Nous en aborderons les grands principes dans ce livre en présentant l'approche la plus connue : l'analyse en composantes principales. Notez que, outre la réduction de dimension,

ce type d'analyse peut aussi avoir pour objectif à part entière de faciliter l'interprétation des mécanismes générateurs des données observées.

Mais avant cela, commençons par une brève introduction aux problèmes posés par les espaces de grande dimension.

Les problèmes liés à la grande dimension

La malédiction de la dimension

Rassurez-vous : sous ce nom, nulle sorcellerie. C'est Richard Bellman qui a suggéré cette appellation ésotérique en 1961, pour désigner un ensemble de problèmes qui se posent uniquement lorsque l'on analyse des données dans des espaces de grandes dimensions. L'idée qui se cache derrière ce concept, c'est que le volume de données nécessaire à l'apprentissage statistique augmente très vite – exponentiellement, même – avec le nombre de dimensions. En effet, l'augmentation du nombre de dimensions entraîne une rapide augmentation du volume de l'espace dans lequel se trouvent les données. Les données se retrouvent donc un peu « perdues » dans ce grand espace (on parle de données éparées) et, en conséquence, les méthodes de *machine learning* perdent en efficacité si elles ne sont pas adaptées à ce contexte particulier d'application.

Dans le cas de la régression, le problème des hautes dimensions se comprend assez intuitivement. En fait, plus le nuage de point est épars, plus la qualité de l'estimation va diminuer. Il sera en effet possible d'expliquer les observations par plusieurs modèles pouvant être fort différents, et pourtant tous vraisemblables (figure 15-1).

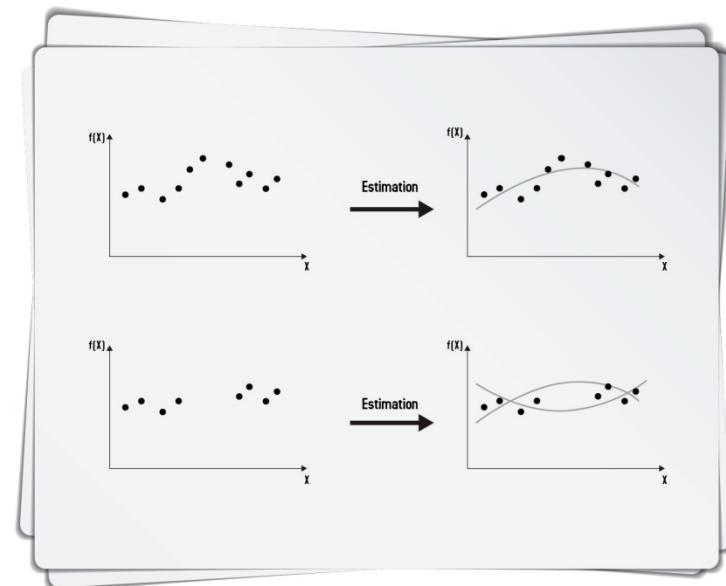


Figure 15-1 – Plus un nuage de points est épars, plus nombreux seront les modèles qui pourront l'expliquer

De plus, la modélisation devient très instable : le déplacement d'un seul des points peut changer fortement les résultats de l'estimation. Imaginons un autre exemple encore plus pathologique où l'on aurait un nombre d'observations inférieur au nombre de dimensions. Dans ce cas, le nuage de points serait explicable par une infinité de modèles ! Prenez par exemple deux points dans un espace à trois dimensions : on peut y faire passer une infinité de plans !

En ce qui concerne la classification, le problème de la haute dimension est un peu moins intuitif, mais non moins problématique. En fait, on peut montrer géométriquement que plus la dimension augmente, plus le volume de l'espace considéré tend à se concentrer dans la « bordure » de cet espace (imaginez la peau d'une orange pour vous représenter ce que serait une telle bordure dans un espace sphérique à trois dimensions). Puisque dans un hypercube tous les points de la bordure sont à égale distance du centre, la majorité des points sont donc à presque égale distance du centre, car concentrés autour de cette bordure (figure 15-2). Cela a des conséquences fâcheuses sur le calcul des distances entre vecteurs, qui tendent vers une constante et ne permettent donc plus d'identifier des groupes distincts.

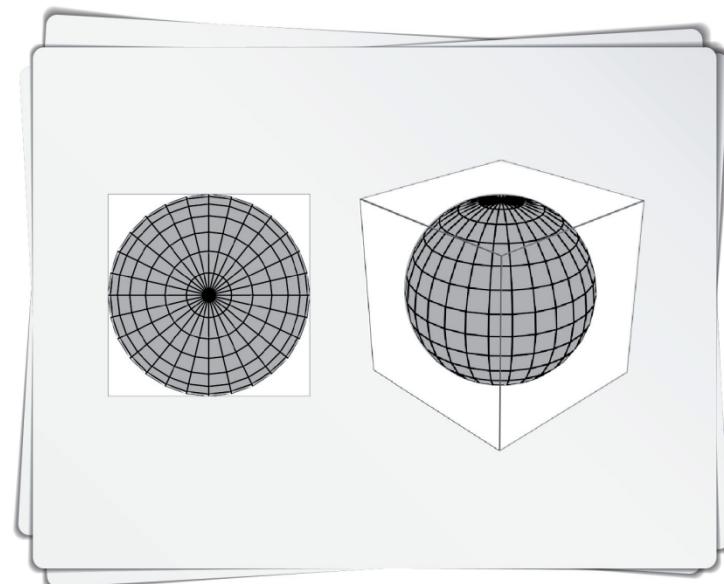


Figure 15-2 – De deux dimensions à trois dimensions : le volume « dans les coins » de l'espace augmente.
Ce phénomène s'accroît avec le nombre de dimensions.

Cette image vous permet par ailleurs de comprendre une raison pour laquelle ce problème croît dramatiquement avec le nombre de dimensions : le nombre de coins augmente exponentiellement avec la dimension. Quatre coins à gauche en 2D, huit en 3D, etc. comme l'illustre le tableau 15-1.

Tableau 15-1. Le nombre de coins en fonction du nombre de dimensions

Nombre de dimensions	2	3	10	16	20	40	50	100
Nombre de coins	4	8	1024	65 536	$\approx 10^6$	$\approx 10^{12}$	$\approx 10^{15}$	$\approx 10^{30}$

Par conséquent, en haute dimension, on aura plein de coins contenant un petit nombre de points qui seront difficiles à discriminer, car à égale distance du centre. Bon courage pour en faire un partitionnement de qualité !

La multicolinéarité

Ce problème n'est pas directement lié à la grande dimension, mais a plus de chance d'apparaître lorsque l'on multiplie les *features* d'une modélisation. On parle de colinéarité ou de multicolinéarité lorsque l'on observe une ou plusieurs relation(s) linéaire(s) parfaite(s) entre les variables explicatives du modèle. En pratique, cette distinction n'est pas vraiment faite et l'on parle de multicolinéarité dans les deux cas de figure. De plus, on parle aujourd'hui de multicolinéarité même lorsque les relations ne sont pas parfaites, c'est-à-dire lorsque les coefficients de corrélation entre certaines variables explicatives sont inférieurs à 1, mais restent élevés. Ce phénomène peut poser problème pour la construction de modèles de régression. Tout d'abord, il va rendre l'interprétation d'un modèle difficile, en confondant les effets de plusieurs variables explicatives. Ceci est surtout gênant pour le statisticien, qui va chercher à isoler les influences respectives des différents régresseurs de son modèle. Pour le *machine-learner* pur et dur, ce problème est relativement trivial, puisque son objectif est rarement l'explication du modèle.

Néanmoins, dans certains cas plus graves, lorsque les relations linéaires sont fortes, la multicolinéarité peut poser des problèmes plus fâcheux et nuire à l'estimation des modèles. Rappelez-vous la résolution analytique de l'équation normale de la régression linéaire multivariée. Dans le cas d'une multicolinéarité parfaite (c'est-à-dire quand on observe des relations linéaires parfaites entre variables explicatives), la matrice $X^T X$ n'est pas inversible et il n'existe donc pas de solution théorique au problème. Par exemple, dans un espace en trois dimensions, on va pouvoir proposer plusieurs plans solutions.

Dans le cas moins pathologique de la colinéarité imparfaite, l'inversion de la matrice $X^T X$ sera possible, mais le plan de régression sera mal déterminé. Un petit changement dans les données pourra en effet engendrer un fort changement dans les coefficients. Ce genre de modèle est fortement instable et non souhaitable. La résolution analytique est instable, mais la descente de gradient aussi. On obtient en effet des espaces de solutions caractérisés par des « vallées », déjà présentés dans le chapitre sur la régularisation.

Autres problèmes liés aux grandes dimensions

Tout d'abord, nous savons qu'en marge des difficultés liées à la malédiction de la dimension et de la multicolinéarité, l'augmentation du nombre de variables va mécaniquement engendrer une surdétermination du modèle. C'est ce que nous avons montré dans le chapitre concernant la régression polynomiale. À nouveau, un modèle ainsi mal posé semblera correct, mais sera en fait non généralisable.

Par ailleurs, d'autres complications non mathématiques sont liées aux espaces de grandes dimensions. D'une part, multiplier les dimensions revient à multiplier le volume de données à traiter, notamment si l'on cherche à « remplir » l'espace vide pour faire face à la malédiction de la dimension. En conséquence, les contraintes techniques à gérer seront beaucoup plus fortes : volume de stockage, temps de traitement, éventuellement parallélisation des algorithmes de modélisation², etc.

D'autre part, il est cognitivement difficile d'appréhender des espaces en très hautes dimensions, et encore plus dur de les visualiser. Faites l'exercice : imaginer un carré dans un plan est chose facile, de même qu'un cube dans un espace à trois dimensions... mais qu'est-il d'un hypercube en quatre dimensions ? Réduire le nombre de dimensions permet donc de donner plus facilement du sens aux données (même si, répétons-le, ceci est plus une préoccupation liée au monde de la statistique que du *machine learning*).

La sélection de variables

Régression pas à pas

L'idée de cette méthode de sélection de variables est simple. Parmi les n variables explicatives, on va rechercher le plus petit sous-ensemble de variables qui explique au mieux la variabilité du vecteur Y . On peut pour cela tester des modèles pour tous les arrangements de modèles possibles (voir par exemple la procédure *leaps and bounds* de Furnival et Wilson, 1974), mais cela devient rapidement infaisable dès que l'on dépasse les 40 variables. C'est pourquoi on utilise plutôt la méthode de la régression pas à pas (*stepwise*). C'est une méthode itérative qui peut se décliner en deux variantes.

La variante d'*« introduction progressive »* (*forward stepwise selection*) initialise un modèle constitué uniquement du terme constant, puis enrichit séquentiellement le modèle en rajoutant, prédicteur après prédicteur, ceux qui améliorent le mieux l'ajustement en terme de réduction de la variabilité résiduelle. Cette sélection se fait en calculant une statistique F . Soient $\hat{\Theta}$ les paramètres estimés pour un sous-ensemble de n *features* et $\tilde{\Theta}$ les paramètres estimés pour un sous-ensemble de $n+1$ *features*. La statistique F permet de comparer la somme des carrés des résidus (SCR) de ces différents modèles, telle que :

$$F = \frac{SCR(\hat{\Theta}) - SCR(\tilde{\Theta})}{S(\tilde{\Theta}) / (m - n - 2)}$$

On enrichit alors itérativement le modèle avec la variable qui produit la plus grande valeur de F . Pour stopper l'enrichissement du modèle, on compare F à la distribution théorique de Fisher-Snedecor $F_{1,m-n-2}$ ³ : on s'arrête lorsqu'aucune nouvelle variable ne permet de générer une statistique F supérieure au quantile 90 ou 95 de la distribution théorique⁴.

L'autre variante est celle de l'élimination progressive (*backward stepwise selection*). C'est le même principe que la sélection *forward*, mais « à l'envers ». On initialise la procédure avec un modèle complet comprenant toutes les variables, que l'on va retirer séquentiellement. Pour cela, on retire successivement les variables qui produisent les plus petites valeurs de F et l'on s'arrête dès que toutes les variables restantes génèrent des valeurs supérieures au quantile 90 ou 95 de la distribution théorique de Fisher-Snedecor.

Il existe de nombreux enrichissements de ces approches, mais nous ne les développerons pas ici pour les raisons suivantes. D'un côté, les statisticiens se montrent toujours assez réservés face à la sélection automatique de variables. Ils utiliseront toujours des procédures plus complexes intégrant plusieurs algorithmes d'analyse, couplés avec des étapes d'interprétations fonctionnelles. D'un autre côté, les *machine learners* jugent les approches *stepwise* et leurs variantes trop statistiques à leur goût et préfèrent employer les méthodes que nous allons présenter dès à présent.

Approches machine learning

Outre la lourdeur statistique, les approches décrites ci-dessus se basent sur des tests statistiques univariés incapables de capturer les interactions entre variables pourtant si porteuses d'informations. Une approche plus typée *machine learning*, qu'on peut qualifier de sélection non linéaire, consiste à utiliser la capacité des modèles non linéaires à trouver ces interactions entre variables.

Vous vous rappelez sans doute que les modèles ensemblistes à base d'arbres comme le *random forest*, *extra trees* ou le *gradient boosting* disposent tous d'un paramètre `compute_importance`. Ce paramètre permet de calculer les variables qui concourent le plus à la construction de l'ensemble. Ce précieux paramètre est la base de la sélection non linéaire de variables. On voit par exemple, dans la figure 15-3, l'importance des variables dans un problème où nous devions prédire le prix de véhicules d'occasions (ici, des BMW) :

```

1. feature : is_X5 (0.155824)
2. feature : dimen_empattement (0.124637)
3. feature : deltadays_1mec_till_vente (0.092594)
4. feature : vehicule_mt_100000 (0.087163)
5. feature : deltamonths_1mec_till_vente (0.086386)
6. feature : millesime (0.072133)
7. feature : deltamonths_1mec_till_today (0.057463)
8. feature : deltadays_1mec_till_today (0.055810)
9. feature : deltadays_deb_com_till_today (0.046614)
10. feature : year_dt_1mec (0.046408)
11. feature : deltamonths_deb_com_till_today (0.034887)
12. feature : dimen_longueur (0.021863)
13. feature : poids_charge_utile (0.015639)
14. feature : norm_famille_km (0.013480)
15. feature : norm_produit_km (0.013205)
16. feature : km_cat (0.008357)
17. feature : vehicule_ancien (0.008279)
18. feature : split_destination_PARTICULIER (0.003630)
19. feature : km_sqrt (0.002633)
20. feature : norm_energie_km (0.002622)
21. feature : dimen_largeur (0.002611)
22. feature : split_departement_CHER (0.002293)
23. feature : norm_produit_prix_neuf_opt (0.002238)
24. feature : norm_famille_prix_neuf_opt (0.002192)
25. feature : km (0.002092)

```

Figure 15-3 – Prédiction du prix de véhicules d’occasion : importance des variables

La présence d’une variable en haut de classement veut dire qu’elle est souvent utilisée par les arbres et qu’elle s’y positionne relativement haut, parce qu’un *split* sur cette variable minimisera les critères d’entropie ou de Gini. On utilisera donc cette information pour sélectionner les meilleures variables. Dans la pratique, la façon la plus rigoureuse de mettre en œuvre cette méthode est la suivante :

- séparation des données en trois groupes : $(X_{\text{élection}}, Y_{\text{élection}})$, $(X_{\text{entraînement}}, Y_{\text{entraînement}})$, $(X_{\text{test}}, Y_{\text{test}})$;
- apprentissage d’un modèle ensembliste sur le couple $(X_{\text{élection}}, Y_{\text{élection}})$;
- sélection des K meilleures variables issues du modèle entraîné ;
- filtrage de $X_{\text{entraînement}}$ et X_{test} , en ne conservant que les K meilleures variables. Vous pouvez désormais travailler avec ces nouvelles matrices de dimensions réduites.

Cette méthode est très puissante, mais elle est à manipuler avec précaution. En effet, elle privilie les variables continues à grande dispersion, tout simplement parce que les arbres arriveront toujours à y trouver des « poches » pour faire leur *split* et les utiliseront donc régulièrement pour construire leurs arbres. Ainsi, une variable continue pourtant peu explicative peut se retrouver au-dessus d’une variable binaire bien plus porteuse d’information d’un point de vue métier. On peut illustrer ce caractère quelque peu dangereux en prenant un peu d’avance sur la partie 3-2

où nous prédirons les survivants du Titanic. Prenons un modèle qui dispose de variables très orientées métier comme la classe, l'âge, le prix du ticket, etc. Injectons maintenant tout bêtement une variable *random* à grande dispersion dans notre modèle.

```
import random
X['random'] = [random.randint(0, 1000) for i in xrange(X.shape[0])]
```

Puis, faisons l'apprentissage de ce modèle :

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
rf.fit(X, y)
```

Et vérifions nos meilleures variables :

```
1. feature : Fare (0.200859)
2. feature : split_Sex_female (0.200355)
3. feature : random (0.184521)
4. feature : Age (0.171788)
5. feature : split_Sex_male (0.076643)
6. feature : split_Pclass_3 (0.047177)
7. feature : split_Pclass_1 (0.036631)
8. feature : Parch (0.024891)
9. feature : SibSp (0.024589)
10. feature : split_Pclass_2 (0.017957)
11. feature : is_child (0.014590)
```

Figure 15-4 – Prédiction des survivants du Titanic :
importance des variables avec insertion d'une variable random

Notre variable *random* se hisse sur le podium, tout simplement parce qu'elle présente une grande dispersion :

```
for col in X.columns :
    print col, std(X[col]) # std : calcul de l'écart-type

Fare 49.6655344448
split_Sex_female 0.477721763229
random 288.190029495
split_Sex_male 0.477721763229
split_Pclass_3 0.497385405301
split_Pclass_1 0.428549564355
Parch 0.805604761245
SibSp 1.10212443509
split_Pclass_2 0.404800382654
is_child 0.230146988265
```

Au-delà de cette limite, vous remarquerez que cette méthode utilise des observations qui ne seront ensuite plus utilisées dans le modèle final. On réservera donc cette approche à des problèmes où le nombre de lignes est suffisamment grand (au moins 100 000 lignes pour une classification binaire par exemple).

Réduction de dimensions : l'analyse en composantes principales

Objectif

Il existe de nombreuses méthodes numériques pour effectuer des réductions de dimensions, voir par exemple la synthèse de Fodor (2002). Dans le cadre de cet ouvrage, on se limitera à évoquer la plus largement utilisée : l'Analyse en Composantes principales (ACP), dont l'idée de base a été posée dès 1901 par Karl Pearson. C'est une « *technique de représentation des données ayant un caractère optimal selon certains critères algébriques et géométriques et que l'on utilise en général sans référence à des hypothèses de nature statistique ni à un modèle particulier* »⁵ (Lebart *et al.*, 2006). Cette méthode permet à la fois d'identifier les individus qui se ressemblent (notion de proximité) et de résumer les relations entre les variables. Cette méthode est ancrée dans un corpus théorique important. Nous n'en donnerons qu'une vue d'ensemble, en nous focalisant sur la question des relations entre variables, qui permet d'aboutir à la réduction dimensionnelle recherchée.

L'ACP s'applique dans le cadre de notre bien connue matrice X , constituée de valeurs numériques continues⁶, dans laquelle chaque individu est décrit par n variables dans \mathbb{R}^n . L'ACP va permettre de construire k variables à partir de n , tel que $k < n$. Ces k variables sont les composantes principales de la matrice de données, aussi nommées axes factoriels. Elles sont définies comme suit.

- Ce sont des combinaisons linéaires des n variables initiales.
- Chacune d'elles restitue une partie de l'information des n variables initiales : elles sont caractérisées par la quantité d'information qu'elles restituent et ordonnées en fonction de cette quantité.
- Elles sont indépendantes les unes des autres (pas de corrélation linéaire).

Les individus sont ainsi projetés dans un sous-espace défini dans \mathbb{R}^P , qui synthétise les relations entre les n variables initiales.

Husson *et al.* (2008) donne une illustration intuitive de cette approche, synthétisée dans l'image 15-5.

On admet communément que les positions des planètes de notre système solaire sont situées dans un espace à trois dimensions. On peut illustrer ces distances dans un système à deux dimensions, ou même dans un système à une dimension : on projette ainsi successivement \mathbb{R}^3 dans \mathbb{R}^2 (représentation dans le plan), puis dans \mathbb{R}^1 (représentation axiale), en résumant de plus en plus l'information initiale. Pour ce faire, il faut minimiser l'erreur de projection dans le sous-espace considéré (en effet, une projection dans un sous-espace réduit va distordre les données. Il faut donc trouver la meilleure projection qui limitera au maximum la déformation). Voici comment s'opère ce numéro de gymnastique spatiale, en deux étapes principales.

Les grandes étapes de l'ACP

Étape 1 : matrice carrée et inertie

Tout d'abord, pour déterminer les composantes principales de m observations définies par n variables, on utilise la matrice de variance-covariance des variables. Soit s_1^2 la variance d'une variable X_1 et $cov_{1,2}$ la covariance de deux variables X_1 et X_2 , ces grandeurs sont définies par :

$$s_1^2 = \frac{1}{m} \sum_{i=1}^m (x_{i1} - \bar{X}_1)^2$$

$$cov_{1,2} = \frac{1}{m} \sum_{i=1}^m (x_{i1} - \bar{X}_1)(x_{i2} - \bar{X}_2)$$

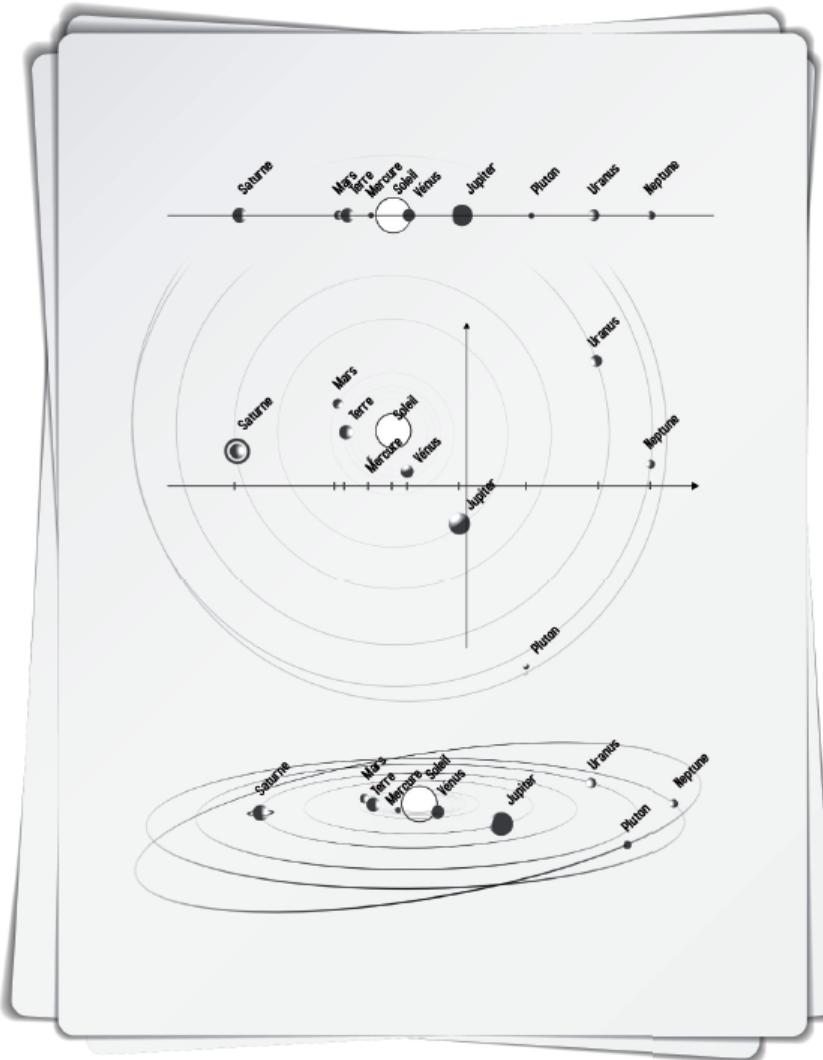


Figure 15-5 – Projection du système solaire (3D) sur un axe (1D) et dans le plan (2D)
Inspiré de Husson et al. (2009)

La barre au-dessus d'une variable (\bar{X}) représente sa moyenne. La variance mesure donc une dispersion autour de la valeur moyenne⁷. Dans le cas de la covariance, on mesure une variation simultanée :

- $\text{cov}_{1,2} > 0$ si X_1 croît quand X_2 croît ;
- $\text{cov}_{1,2} < 0$ si X_1 croît quand X_2 décroît ;
- $\text{cov}_{1,2} = 0$ si X_1 et X_2 sont indépendantes.

Au final, la matrice carrée sera définie telle que :

$$\begin{matrix} & \boxed{} & 1 & j & \dots & n \\ 1 & s_1^2 & cov_{1,j} & cov_{1,\dots} & cov_{1,n} \\ j & cov_{j,1} & s_j^2 & cov_{j,\dots} & cov_{j,n} \\ \dots & cov_{\dots,1} & cov_{\dots,j} & s_{\dots}^2 & cov_{\dots,n} \\ n & cov_{n,1} & cov_{n,j} & cov_{n,\dots} & s_n^2 \end{matrix}$$

Si les variables mesurées sont d'ordres de grandeur hétérogènes (cas fréquent), on va préalablement centrer-réduire les données. Calculer une matrice de variance-covariance sur des données centrées réduites revient à calculer une matrice des corrélations. Toutes les valeurs de la diagonale de la matrice seront égales à 1, et la mesure de covariance sera remplacée par une mesure de corrélation (le coefficient de corrélation de Pearson⁸). Cette mesure comprise entre -1 et 1 exprime l'intensité et le sens d'une relation linéaire :

- si nulle, pas de relation linéaire ;
- si proche de 1, forte relation linéaire croissante ;
- si proche de -1, forte relation linéaire décroissante.

Toutes les moyennes calculées définissent par ailleurs un point moyen g du nuage des individus, donné par le vecteur de la valeur moyenne de chacune des j variables.

Grâce à la matrice de variance-covariance, on peut calculer l'inertie du nuage de points considéré. L'inertie correspond à la somme des variances de n variables, soit la trace de la matrice de variance-covariance (la trace d'une matrice est la somme de ses coefficients diagonaux). L'inertie permet de mesurer la dispersion totale du nuage de points, qui correspond à la somme pondérée des carrés des distances des individus au centre de gravité g . Si on travaille sur la matrice de corrélation, la variance de chaque variable vaut 1 et l'inertie totale est donc égale au nombre n de variables.

Étape 2 : définition et choix des composantes principales

Dans un second temps, il va être possible de définir les composantes principales. Pour cela, on va rechercher des axes portant le maximum d'inertie par la construction de nouvelles variables de variance maximale. Cela revient à effectuer un changement de repère dans \mathbb{R}^n de façon à se placer dans un nouveau système de représentation. Dans le nouveau système, le premier axe apporte le plus possible de l'inertie totale du nuage, le deuxième axe le plus possible de l'inertie non prise en compte par le premier axe, et ainsi de suite.

Ces composantes sont déterminées par calcul matriciel, en diagonalisant la matrice carrée calculée préalablement. Nous vous épargnons les détails mathématiques, mais sachez qu'une matrice est diagonale si tous ses coefficients en dehors de la diagonale sont nuls et que diagonaliser une matrice... c'est la rendre semblable à une matrice diagonale ! Cette opération permet de définir les axes principaux d'inertie (souvent notés u_1, \dots, u_k), qui représentent les n axes de direction des

vecteurs propres de la matrice carrée. À chacun de ces axes est associée une nouvelle variable : c'est la composante principale tant recherchée ! C'est un vecteur renfermant les coordonnées des projections des individus sur l'axe considéré. Ces coordonnées sont obtenues par combinaison linéaire des variables initiales. Si c_1, \dots, c_k sont les k composantes principales de l'analyse, on a :

$$\begin{aligned}c_1 &= u_{1,1}X_1 + u_{1,2}X_2 + \dots + u_{1,n}X_n, \\c_2 &= u_{2,1}X_1 + u_{2,2}X_2 + \dots + u_{2,n}X_n, \\&\dots \\c_k &= u_{k,1}X_1 + u_{k,2}X_2 + \dots + u_{k,n}X_n\end{aligned}$$

Chaque axe de direction a également une valeur propre λ_k , associée à la composante considérée. Cette valeur propre est égale à la somme des coefficients de corrélation au carré de chaque variable initiale avec la composante. Tout ceci est un peu abstrait. Retenez juste que la valeur propre mesure l'inertie apportée par l'axe ou, autrement dit, la part de variance totale des variables initiales pour la composante principale (la somme des valeurs propres correspond donc à la variance totale du nuage de points). Lors de la réalisation de l'ACP, on classe les composantes principales par valeur propre décroissante. Pour mieux appréhender le niveau d'information restitué par chaque composante, on donne la proportion de la valeur propre de chacune d'elles par rapport à la somme de toutes les valeurs propres.

Dans le cadre d'une analyse statistique multidimensionnelle classique, on visualise les résultats de l'ACP à travers deux graphiques finement analysés.

- La représentation des individus dans les plans principaux : on constitue des plans factoriels à partir des couples des composantes qui portent le plus d'information, et on y projette les individus grâce à l'équation de la composante (le meilleur plan est constitué des deux premières composantes, mais il est souvent utile d'aller plus loin et d'en visualiser d'autres pour avoir une représentation plus fine des individus).
- La représentation des variables, la plupart du temps grâce au cercle des corrélations : celui-ci permet de caractériser les composantes principales par leurs « proximités » avec les variables initiales. Les composantes synthétiques peuvent ainsi être interprétées à partir de leurs corrélations avec les variables mesurées et un sens fonctionnel peut donc être inféré.

Dans un usage *machine learning*, on ne s'arrête généralement pas sur ces représentations. L'objectif est simplement de retenir les k composantes principales qui permettront de mieux résumer l'espace initial des variables. On dispose de nombreuses règles empiriques pour déterminer le nombre de composantes à retenir, par exemple :

- interprétabilité des axes : on ne garde que les composantes pour lesquelles on saura donner un sens métier, grâce à leurs corrélations aux variables initiales ;
- règle de Cattell (ou règle du coude) : on trace l'information cumulée apportée par les composantes et on « coupe » au coude. Cela revient à négliger les composantes portant peu d'informations additionnelles ;
- règle de Kaiser : on conserve les composantes ayant des valeurs propres supérieures à 1, c'est-à-dire dont la variance est supérieure à celle des variables d'origine.

Il n'y a pas de règle universelle. À vous de voir en pratique le choix le plus adapté au problème traité.

Exemple d'application

Ce petit exemple provient d'un travail réalisé pour une usine de production de semi-conducteurs. Dans le cadre d'une mission plus générale, il a fallu intégrer un travail de réduction dimensionnelle pour appréhender l'activité applicative du système d'information. En effet, pour chaque application, des dizaines, voire des centaines de transactions différentes peuvent être appelées quotidiennement. L'objectif est donc de résumer le volume d'activité transactionnelle journalier. Les jours de la période d'observation seront les individus de l'analyse, le volume d'activité de chacune des transactions en définira les variables.

Pour cela, nous chargeons dans R un fichier CSV qui contient :

- une colonne `PERIOD` : les 248 jours observés (qui peuvent être vus comme les « individus » de l'étude) ;
- une colonne `JOUR` : une variable qualitative qui définit le jour considéré (lundi, mardi, etc.) ;
- et 106 autres colonnes qui représentent le nombre de transactions appelées par jour. Ces 106 variables correspondent à des transactions propres à la production microélectronique (`ComputeWaferForEqp`, `FetchCompleteSeries_LITHO`, `GetRecipeTroughput...`).

Nous chargeons les données, définissons les individus en tant que noms de lignes et supprimons les samedis et dimanches⁹ de l'analyse avec le code suivant :

```
## Définition du nom des lignes
rownames(transactions) <- transactions$PERIOD
transactions$PERIOD <- NULL

## Retrait des week-ends
transactionsWeek <- transactions[!transactions$JOUR%in%c("samedi", "dimanche"),]

## Retrait de la variable JOUR qui ne sera plus utilisée
transactionsWeek$JOUR <- NULL
```

Ces données sont analysées par ACP. La librairie R `stats` (par défaut dans toute installation de R) permet de la réaliser :

```
acpTransactions <- princomp(x = transactionsWeek, cor = TRUE)
```

Avec `cor = TRUE`, nous précisons que nous réalisons l'analyse à partir de la matrice de corrélation (ordres de grandeur hétérogènes) et non à partir de la matrice de variance-covariance¹⁰. Et voilà, l'analyse est terminée ! La commande suivante permet de résumer la qualité de l'analyse :

```
summary(acpTransactions)
```

Ce qui donne, pour les quatre premières composantes, les résultats présentés à la figure 15-6.

Importance of components:

	Comp.1	Comp.2	Comp.3	Comp.4
Standard deviation	7.6101129	4.0456567	2.2467486	1.7877852
Proportion of Variance	0.5463568	0.1544089	0.0476215	0.0301526
Cumulative Proportion	0.5463568	0.7007656	0.7483871	0.7785397

Figure 15-6 – Résultats de l'analyse en ACP

Standard deviation donne $\sqrt{\lambda_k}$. Proportion of variance indique l'inertie apportée par l'axe par rapport à l'inertie totale : en cumulé, on constate que les quatre premières composantes expliquent près de 78 % de la variabilité totale des 106 variables initiales. C'est donc une compression des données de départ plutôt efficace !

Pour interpréter les composantes, on peut tracer le cercle les corrélations avec la fonction `biplot`. On peut utiliser le code suivant, qui permet de visualiser les corrélations entre une composante et les variables initiales :

```
## Composante à analyser
score <- "Comp.2"

## Corrélation minimum à visualiser
cormin <- 0.6

## Données correspondant à cette corrélation minimum
corel <- cbind(acpTransactions $scores[,score], transactionsWeek)
corel <- cor(corel)
corel <- corel[1,]
corel <- corel[corel > cormin | corel < -cormin]
corel <- sort(corel)

## Visualisation
mycol <- colorRampPalette(c("white","black"))

par(mfrow = c(1,1))
par(oma=c(2, 20, 0, 0), las = 2, cex = 0.6, las = 1)

barplot(corel, xlim = c(-1, 1), col = mycol(length(corel)), horiz = TRUE, space = 0.4,
main = "Corrélations variables actives & axe factoriel")
box()

abline(v=c(-0.8, 0.8), lty = 2, lwd = 2)
abline(v=c(-0.5, 0.5), lty = 2, lwd = 1)
```

Avec `cor = TRUE`, nous précisons que nous réalisons l'analyse à partir de la matrice de corrélation (ordres de grandeur hétérogènes) et non à partir de la matrice de variance-covariance¹⁰. Et voilà, l'analyse est terminée ! La commande suivante permet de résumer la qualité de l'analyse :

```
summary(acpTransactions)
```

Ce qui donne, pour les quatre premières composantes, les résultats présentés à la figure 15-6.

Importance of components:

	Comp.1	Comp.2	Comp.3	Comp.4
Standard deviation	7.6101129	4.0456567	2.2467486	1.7877852
Proportion of Variance	0.5463568	0.1544089	0.0476215	0.0301526
Cumulative Proportion	0.5463568	0.7007656	0.7483871	0.7785397

Figure 15-6 – Résultats de l'analyse en ACP

Standard deviation donne $\sqrt{\lambda_k}$. Proportion of variance indique l'inertie apportée par l'axe par rapport à l'inertie totale : en cumulé, on constate que les quatre premières composantes expliquent près de 78 % de la variabilité totale des 106 variables initiales. C'est donc une compression des données de départ plutôt efficace !

Pour interpréter les composantes, on peut tracer le cercle les corrélations avec la fonction `biplot`. On peut utiliser le code suivant, qui permet de visualiser les corrélations entre une composante et les variables initiales :

```
## Composante à analyser
score <- "Comp.2"

## Corrélation minimum à visualiser
cormin <- 0.6

## Données correspondant à cette corrélation minimum
corel <- cbind(acpTransactions $scores[,score], transactionsWeek)
corel <- cor(corel)
corel <- corel[1,]
corel <- corel[corel > cormin | corel < -cormin]
corel <- sort(corel)

## Visualisation
mycol <- colorRampPalette(c("white","black"))

par(mfrow = c(1,1))
par(oma=c(2, 20, 0, 0), las = 2, cex = 0.6, las = 1)

barplot(corel, xlim = c(-1, 1), col = mycol(length(corel)), horiz = TRUE, space = 0.4,
main = "Corrélations variables actives & axe factoriel")
box()

abline(v=c(-0.8, 0.8), lty = 2, lwd = 2)
abline(v=c(-0.5, 0.5), lty = 2, lwd = 1)
```

- et les méthodes de classification (attention, on parle bien ici de classification au sens statistique du terme, et non *machine learning* ; voir la remarque faite sur ce sujet au chapitre sur le *clustering*), qui produisent des classes d'individus.

Husson *et al.* (2009) synthétisent les parentés entre ces méthodes dans l'arbre 15-8.

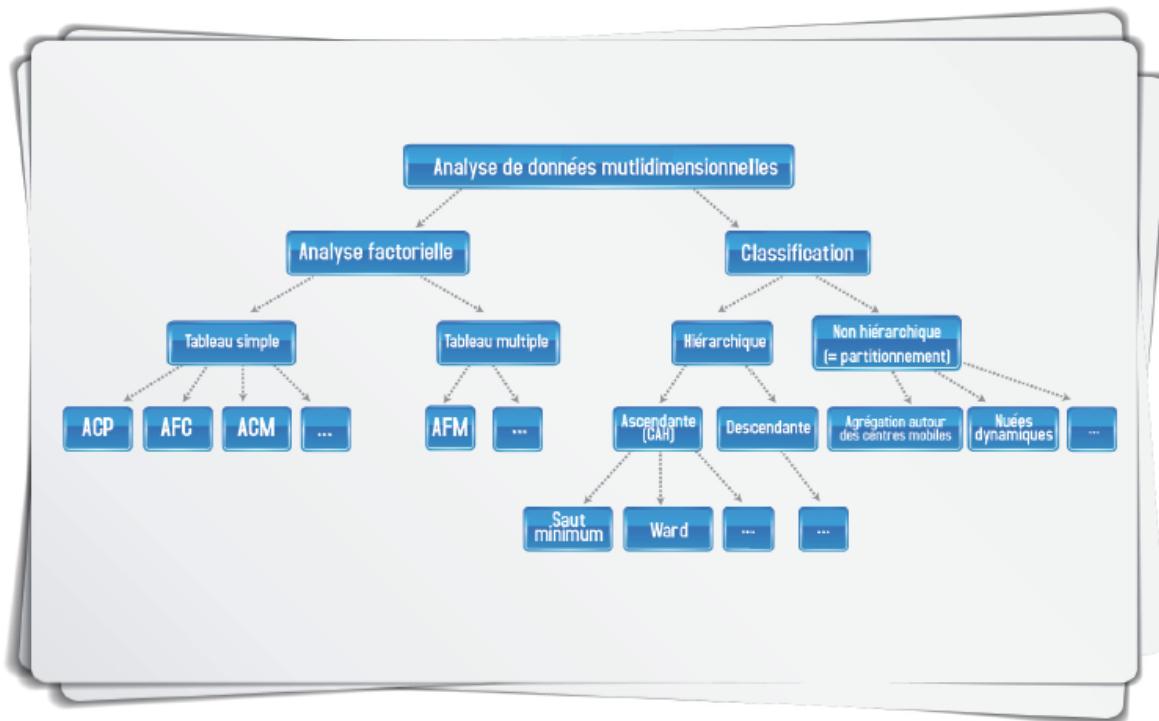


Figure 15-8 – Les principales méthodes d'analyse exploratoire multidimensionnelle
(dans Husson *et al.*, 2009)

Les statisticiens ont l'habitude d'utiliser ces techniques conjointement, pour aboutir à plus de richesse d'analyse. Par exemple, en réalisant une analyse factorielle préalable à une classification hiérarchique, pour ne conserver qu'un certain nombre de facteurs dans la classification. Cela permet de réduire le bruit lié aux composantes à faible inertie ou à ne conserver que les composantes fonctionnellement interprétables. D'autres approches sont possibles. Nous recommandons au lecteur intéressé de se tourner vers un ouvrage de statistique exploratoire pour plus de détails (Lebart *et al.*, 2006, par exemple, déjà largement cité dans ce livre).

À RETENIR Les espaces de grandes dimensions

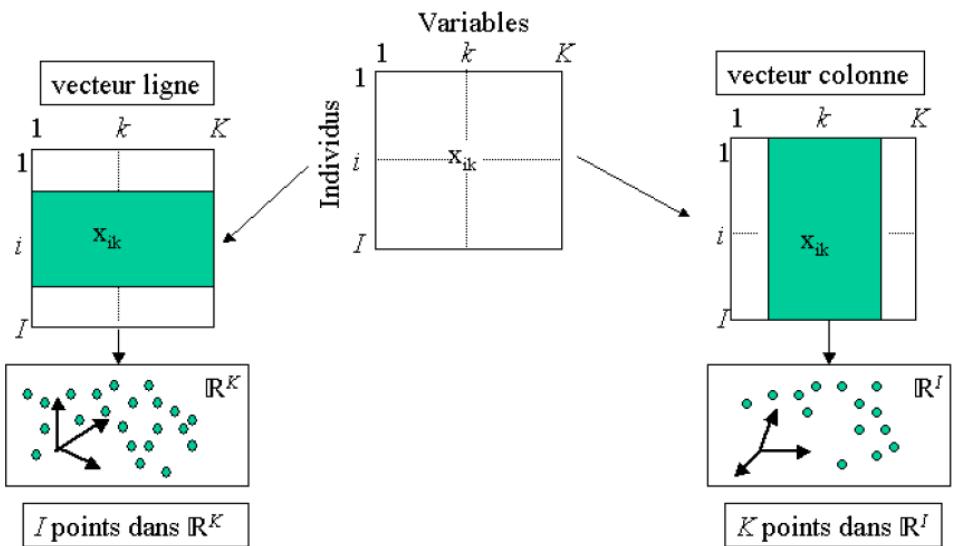
Les espaces de grandes dimensions (avec beaucoup de colonnes) peuvent nuire à un bon apprentissage et/ou complexifier inutilement le modèle. Pour pallier ce problème, deux méthodes peuvent être employées afin de réduire le nombre de variables :

- sélectionner un nombre réduit de variables, les approches *machine learning* permettent d'effectuer cette opération avec facilité et efficacité ;
- ou créer des variables synthétiques résument l'information initiale à l'aide de méthodes factorielles de type ACP.

8

L'analyse factorielle

L'analyse factorielle



Les techniques factorielles sont très appréciées des statisticiens, auxquels elles permettent, à la fois de représenter en deux ou trois dimensions, le plus fidèlement possible, les individus d'une population, et aussi de détecter les liaisons entre les variables ainsi que les variables séparant le mieux les individus. Elles font appel à l'algèbre linéaire et à un outil très bien adapté à la classification et à la reconnaissance des formes.

Les techniques factorielles sont aussi un puissant outil de réduction des dimensions d'un problème, qui permet de diminuer le nombre de variables étudiées en perdant le moins possible d'information.

La transformation, par l'analyse des correspondances multiples, de variables qualitatives en variables continues est assez régulièrement utilisée, notamment dans l'analyse discriminante sur variables qualitatives. Enfin, l'analyse en composantes principales avec rotation permet de constituer des groupes de variables en s'appuyant sur leurs corrélations et est à la base d'un algorithme efficace de classification des variables.

Il existe plusieurs techniques d'analyse factorielle dont les plus courantes sont :

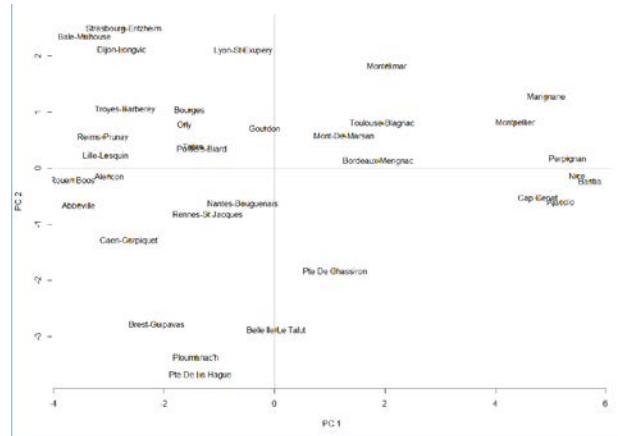
- l'Analyse en Composante Principale (ACP) portant sur des variables quantitatives,
- l'Analyse Factorielle des Correspondances (AFC) portant sur deux variables qualitatives et
- l'Analyse des Correspondances Multiples (ACM) portant sur plusieurs variables qualitatives (il s'agit d'une extension de l'AFC).

Pour combiner des variables à la fois quantitatives et qualitatives, on pourra avoir recours à l'analyse mixte de Hill et Smith.

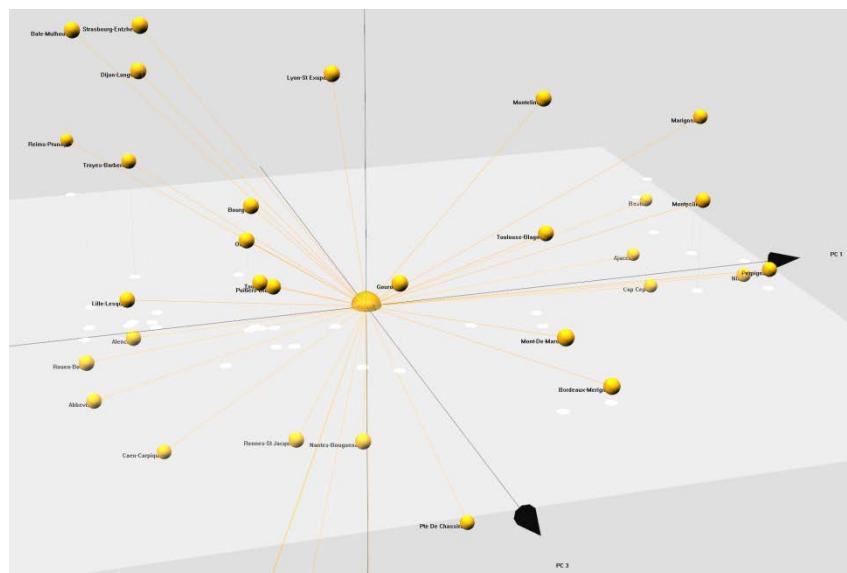
L'analyse en composante principale

L'analyse en composantes principales (ACP), que nous allons étudier dans cette section, est une méthode qui permet de projeter le nuage d'individus sur des sous-espaces de dimension inférieure en respectant au mieux les distances entre individus.

Quand les k variables décrivant les i individus d'une population sont toutes numériques, chaque individu peut être représenté par un point dans un espace \mathbb{R}^k à k dimensions. L'ensemble des individus est un « **nuage de points** ».



Quand $p \leq 2$, l'observation du nuage montre bien les distances entre individus ; cette observation devient plus difficile quand $p = 3$, mais elle est impossible dès que le nombre de variables excède 3. Il est donc naturel de vouloir réduire l'espace \mathbb{R}^p à \mathbb{R}^2 ou \mathbb{R}^3 — on parle de projection des variables de \mathbb{R}^p sur \mathbb{R}^2 ou \mathbb{R}^3 .



Le problème est que le choix de deux ou trois variables, par exemple « âge, salaire, ancienneté », est nécessairement arbitraire, et peut faire perdre beaucoup d'information sur les données, car rien n'assure a priori qu'il s'agisse des variables les plus discriminantes.

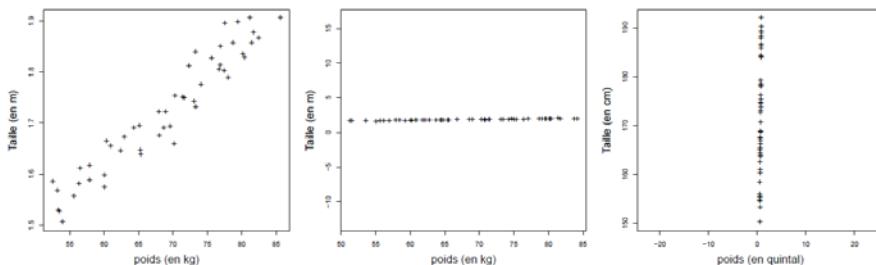
La projection du nuage des individus de l'espace initial à p dimensions dans un espace de dimension inférieure diminue systématiquement les distances entre individus : il faut évidemment chercher à les diminuer le moins possible, si l'on veut bien distinguer les individus et comprendre ce qui les unit et les oppose.

Le centrage et réduction des données

Centrage – Réduction des données

Centrer les données ne modifie pas la forme du nuage (toujours centrer)

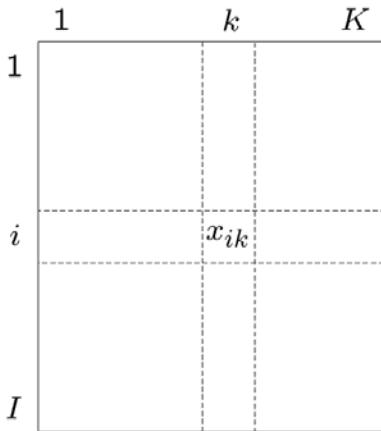
$$x_{ik} \hookrightarrow x_{ik} - \bar{x}_k$$



Réduire les données est indispensable si les unités de mesure sont différentes d'une variable à l'autre

$$x_{ik} \hookrightarrow \frac{x_{ik}}{s_k} \quad x_{ik} \hookrightarrow \frac{x_{ik} - \bar{x}_k}{s_k}$$

L'ACP s'intéresse à des tableaux de données rectangulaires avec des individus en lignes et des variables quantitatives en colonnes



Pour la variable k , on note :

$$\text{la moyenne : } \bar{x}_k = \frac{1}{I} \sum_{i=1}^I x_{ik}$$

$$\text{l'écart-type : } s_k = \sqrt{\frac{1}{I} \sum_{i=1}^I (x_{ik} - \bar{x}_k)^2}$$

Nous commençons par centrer systématiquement toutes les variables, en leur soustrayant leur moyenne, de façon à travailler sur des variables de moyenne nulle. Cela simplifie les calculs et les représentations géométriques, puisque le centre de gravité du nuage d'individus coïncide alors avec l'origine 0 des axes et des sous-espaces.

L'étude du nuage des individus

L'étude du nuage des individus¹

1 individu = 1 ligne du tableau) 1 point dans un espace à K dim

Si $K = 1$: Représentation axiale

Si $K = 2$: Nuage de points

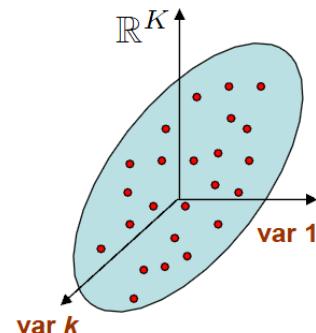
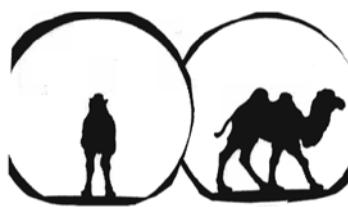
Si $K = 3$: Représentation + difficile en 3D

Si $K = 4$: Impossible à représenter MAIS le concept est simple 

Notion de ressemblance : distance (au carré) entre individus i et i' :

II

$$d^2(i, i') = \sum_{k=1}^K (x_{ik} - x_{i'k})^2$$

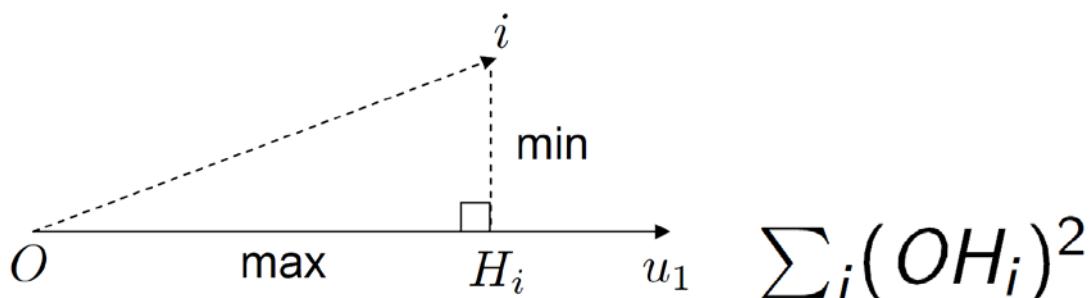


La distance entre deux individus mesure donc la différence existant entre eux. Analyser la variabilité entre les individus revient donc à étudier l'ensemble des distances inter-individus. Ainsi, on peut voir la somme des distances inter-individus comme une mesure de l'information portée par le nuage. En effet, la somme des distances inter-individus quantifie en quelque sorte la forme du nuage. Si les points sont tous proches les uns des autres, cette quantité sera faible alors que des points très éloignés des autres auront tendance à l'augmenter.

Trouver le sous-espace qui résume au mieux les données.

Qualité d'une image :

- Restitue fidèlement la forme générale du nuage (animation)
 - Meilleure représentation de la diversité, de la variabilité
 - Ne perturbe pas les distances entre individus



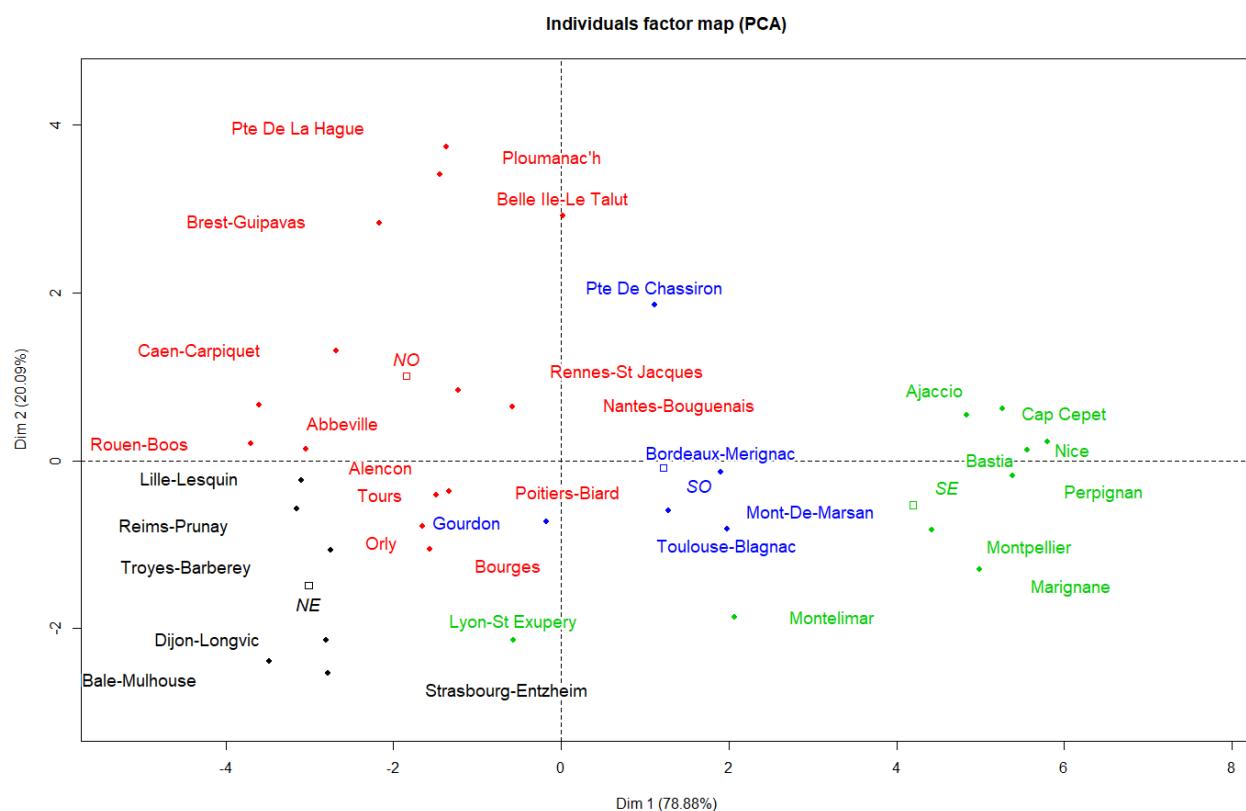
Un objectif de l'ACP sera de décomposer une quantité dérivant de cette somme (l'inertie) en faisant apparaître des individus ou des groupes d'individus y contribuant de manière particulière. On cherchera en particulier à déterminer quelles directions de l'espace y contribuent le plus, autrement-dit, on cherchera à savoir dans quelles directions de l'espace les déformations ou les allongements du nuage sont les plus importants.

Cela revient à minimiser l'écart entre chaque individu et sa projection, c'est-à-dire à allonger au maximum la projection du nuage d'individus sur l'axe. Cela fait, on cherche un deuxième axe qui, parmi tous les axes orthogonaux (perpendiculaires, c'est-à-dire non corrélés) au premier sera celui qui maximise l'inertie projetée sur ce deuxième axe. Cette inertie projetée sur le deuxième axe est par construction inférieure à celle projetée sur le premier axe. Plusieurs axes factoriels seront ainsi successivement déterminés avec des inerties projetées décroissantes. En raison de leur orthogonalité, l'inertie totale du nuage d'individus se décompose en la somme des inerties projetées sur chaque axe.

C'est donc, au coefficient $1/n$ près, la somme des carrés de toutes les cellules de la matrice des données centrées réduites. En cela, il est bien clair que c'est une mesure de l'information portée par les données.

Cependant, on peut également en faire deux interprétations : une en lien avec le nuage des individus et l'autre en lien avec le nuage des variables.

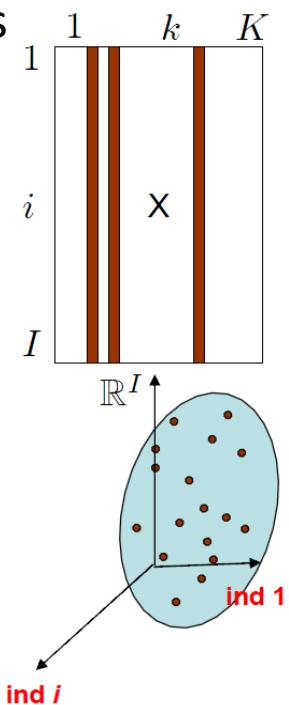
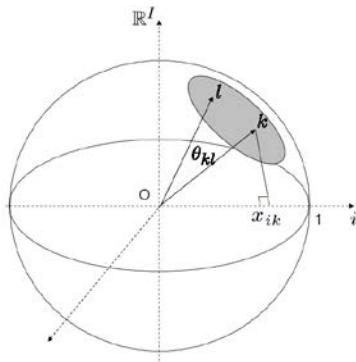
L'inertie peut être vue comme la somme des carrées des distances au centre de gravité pour tous les individus. En cela, l'inertie renseigne sur la "**forme**" du nuage des individus.



L'étude du nuage des individus

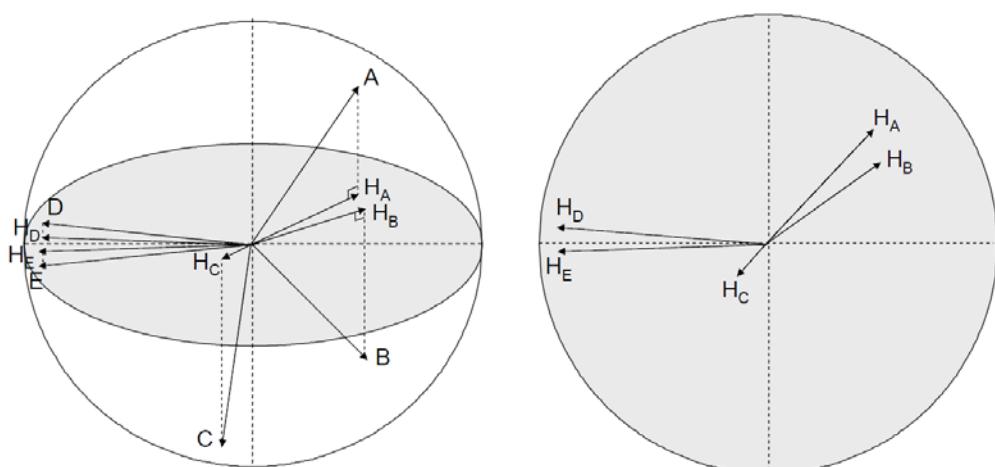
L'étude du nuage des variables

Nuage des variables

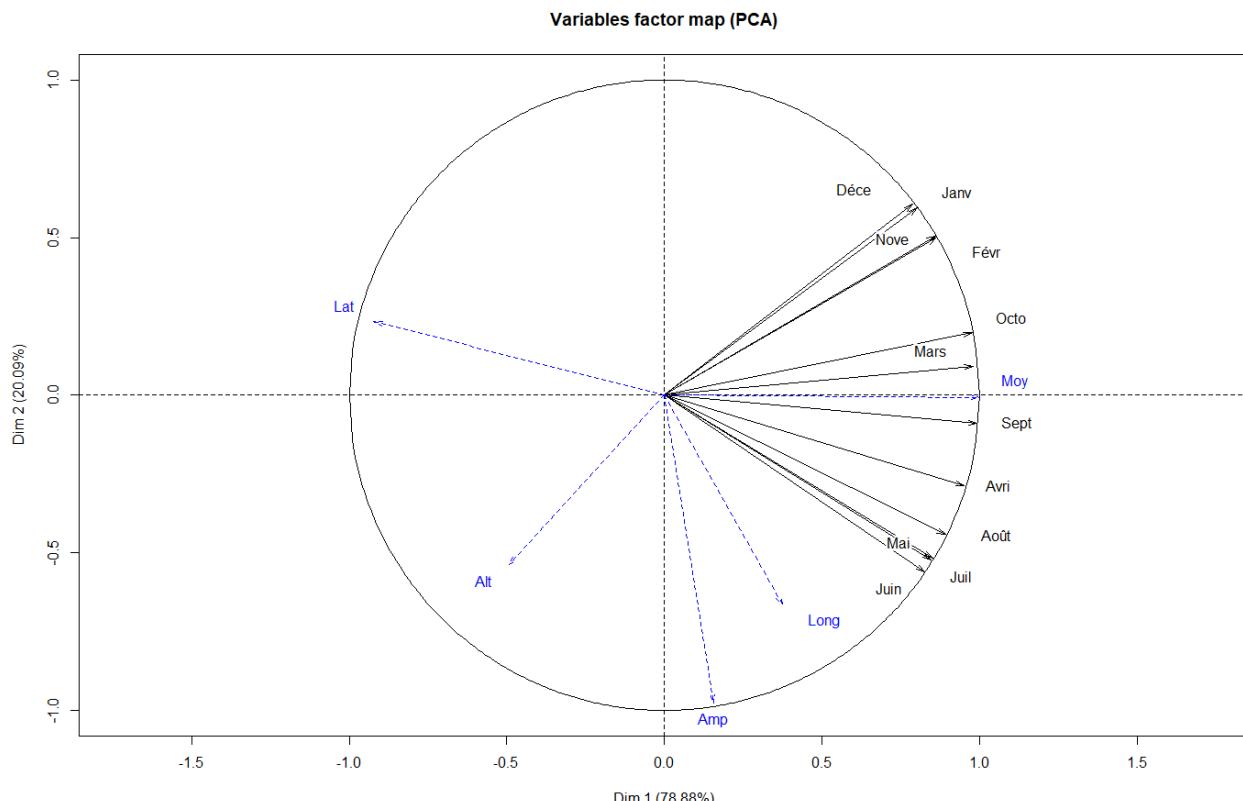


Venons-en à l'espace des variables. Son étude se fait en parallèle de celle de l'espace des individus. En effet, si i individus décris par k variables définissent un nuage de i points dans \mathbb{R}^k (espace des individus), ils définissent aussi un nuage de points dans \mathbb{R}^k (espace des variables). À chaque variable correspond un point qui est l'ensemble de ses valeurs prises pour tous les individus.

Dans le nuage de variables, on cherche, comme dans le nuage d'individus, à maximiser l'inertie projetée. Toutefois, il ne s'agit ici pas de rechercher la direction d'allongement maximal du nuage, puisque toutes les variables ont même norme 1 et sont donc sur l'hypersphère de rayon 1. Il s'agit de maximiser la somme des carrés des coordonnées des projections des variables sur un axe, c'est-à-dire de maximiser les sommes des cosinus carrés des angles que forment l'axe et les variables, autrement dit de maximiser la somme des carrés des coefficients de corrélation des variables et de l'axe cherché, celui donnant la direction d'inertie maximale. L'axe qui vérifie cette propriété est appelé axe factoriel.



Le nuage des variables est plus souvent l'objet d'analyse que le nuage des individus, car, surtout quand ceux-ci sont nombreux (au-delà de plusieurs centaines), il est rare que l'on s'intéresse aux projections des individus sur les axes factoriels et à leurs contributions. Tandis que dans l'espace des individus on s'intéresse aux distances entre points, dans l'espace des variables on s'intéresse aux angles entre variables. Cela vient de la propriété du cosinus de l'angle entre deux variables centrées-réduites est égal à leur coefficient de corrélation linéaire, et que chercher deux variables corrélées positivement revient à chercher deux variables formant un angle aigu.

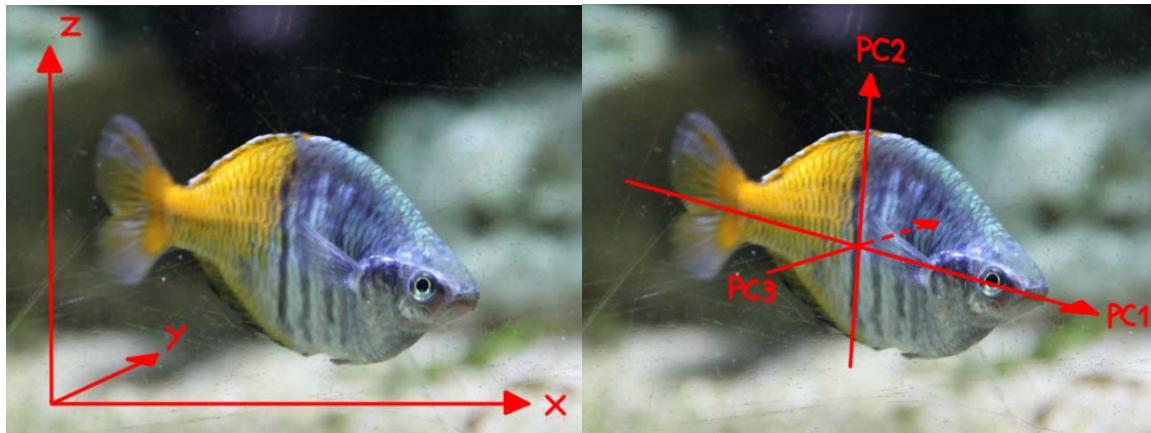


La qualité de la représentation d'une variable sur un axe factoriel est le carré de son cosinus avec cet axe, cette définition se justifiant par son lien avec le coefficient de corrélation.

- la variable est parfaitement représentée sur le plan (**qualité = 1**) => sa projection est sur le cercle des corrélations ;
- la qualité de la représentation de la variable est < 1 et > 0 => sa projection est à l'intérieur du cercle des corrélations ;
- la variable n'est pas du tout représentée sur le plan (**qualité = 0**) => sa projection est au centre du plan et du cercle des corrélations.

L'analyse en Composante Principale ACP

La transformation implique d'abord une rotation des axes de coordonnées : le premier axe suivra la direction la plus "allongée" (imaginez un poisson ; l'axe va de la tête à la queue), le second une direction perpendiculaire (du ventre vers le dos), le troisième perpendiculaire aux deux premiers (l'épaisseur du poisson), etc.



L'important dans tout ça, c'est que si les deux premières composantes cumulent une grande portion de la variabilité totale, on peut ignorer les autres et les opposer dans un graphique.

```
>>> import pandas as pd
>>> import numpy as np
>>> from matplotlib import pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
>>> import seaborn as sns
>>> db = pd.read_csv('donnees/index2016.csv', sep=',')
>>> db.head()
      Country          Region  Population    GDP \
0  Afghanistan  Asia-Pacific      31.3  60.58
1      Albania        Europe      2.8  31.56
2      Algeria  Middle East / North Africa      38.7  551.81
3      Angola  Sub-Saharan Africa      24.4  175.64
4    Argentina  South and Central America / Caribbean      42.0  947.57

  Unemployment rate  Inflation rate    FDI Inflow  Public Debt rate \
0                9.1        4.61     53.56       6.7
1               16.1        1.63    1093.48      72.6
2                9.5        2.92    1488.00       8.8
3                6.8        7.30   -3881.00      38.0
4                8.2       37.60    6612.00      48.6

  Freedom from Corruption
0                      12.0
1                      33.0
2                      36.0
3                      19.0
4                      34.0
>>> #db.tail()
>>> db.dtypes # types de variables
Country          object
Region          object
```

```

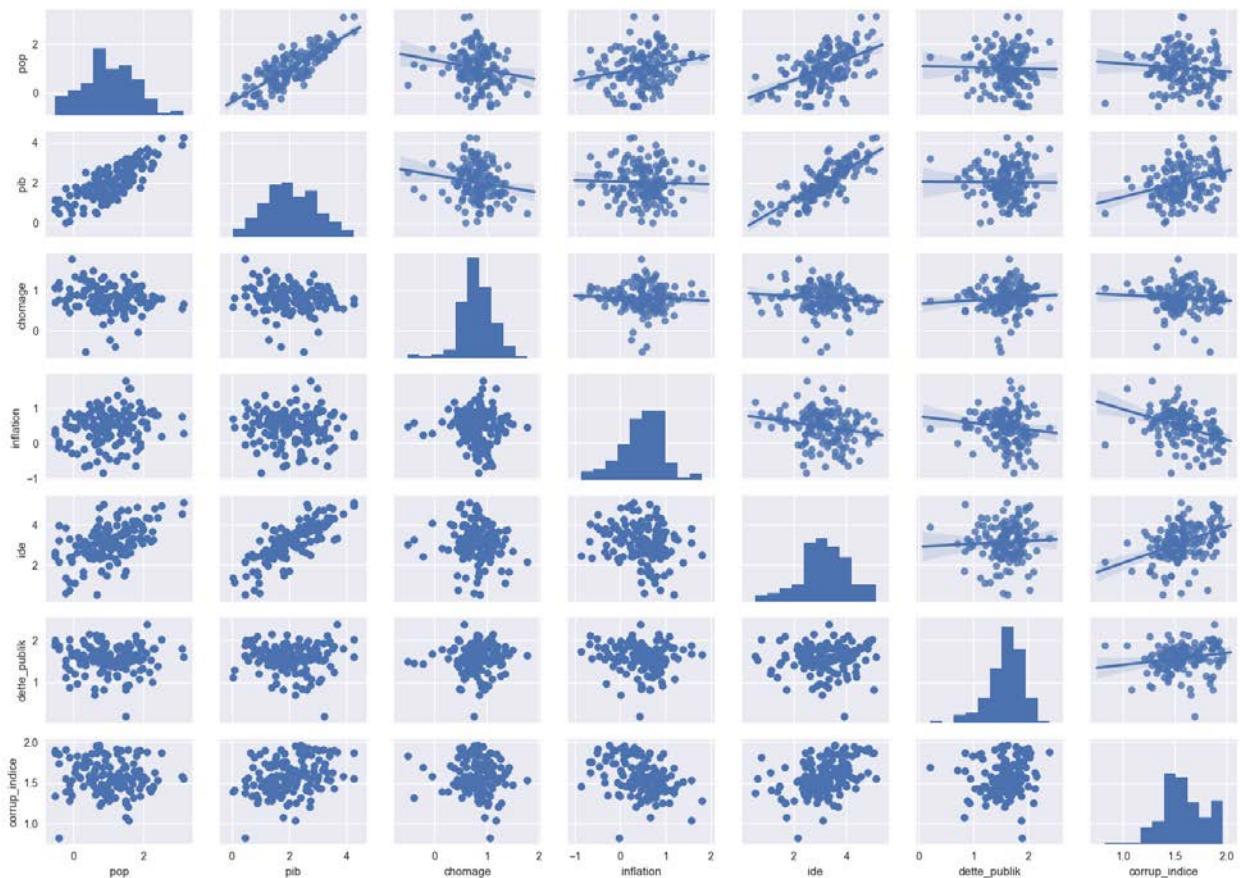
Population           float64
GDP                float64
Unemployment rate float64
Inflation rate    float64
FDI Inflow          float64
Public Debt rate   float64
Freedom from Corruption float64
dtype: object
>>> db.shape
(169, 9)
>>> # Renommer les variables
>>> db.columns = ['Pays','Region','pop','pib','chomage',
...                 'inflation','ide','dette_publik',
...                 'corrup_indice']
>>> # visualisation des mesures de statistique descriptive
>>> db.describe()
              pop        pib      chomage  inflation      ide \
count  169.000000  169.000000  169.000000  169.000000  169.000000
mean   41.968047  638.236864   8.820118   4.419231  7244.549112
std    147.034910  2068.930753   7.209696   6.961893 17993.503001
min    0.300000    1.090000   0.300000  -1.600000  -4956.680000
25%    3.600000   27.130000   4.300000   0.910000   363.270000
50%    9.500000   81.120000   6.900000   2.900000  1200.000000
75%   30.600000  409.330000  10.900000   5.990000  4901.840000
max   1367.800000 17617.320000  60.000000  62.170000 128500.000000

              dette_publik  corrupt_indice
count  169.000000  169.000000
mean   50.195799  42.875148
std    34.436074  19.835297
min    0.000000  6.700000
25%   29.800000  28.000000
50%   41.600000  38.000000
75%   65.000000  55.000000
max   246.400000  92.000000
>>> # log - transformation et exclusion des NAN
>>> logdb = pd.concat([np.log10(db.ix[:,2:9])],
...                      axis=1).replace([np.inf, - np.inf],
...                      np.nan).dropna()
>>> g = sns.PairGrid(logdb)
>>> g.map_upper(sns.regplot)
>>> g.map_lower(plt.scatter)
>>> g.map_diag(plt.hist)
>>> plt.show()
>>> db.ix[:,2:9].corr().round(2)
              pop      pib      chomage  inflation      ide  dette_publik \
pop       1.00    0.77     -0.11      0.03    0.55      0.03
pib       0.77    1.00     -0.10     -0.02    0.71      0.17
chomage   -0.11   -0.10     1.00     -0.05   -0.13      0.15
inflation  0.03   -0.02     -0.05     1.00   -0.08     -0.06
ide        0.55    0.71     -0.13     -0.08    1.00      0.08
dette_publik  0.03    0.17     0.15     -0.06    0.08      1.00
corrup_indice -0.04   0.14     -0.05     -0.31    0.33      0.27

              corrupt_indice
pop            -0.04
pib            0.14

```

chomage	-0.05
inflation	-0.31
ide	0.33
dette_publik	0.27
corrup_indice	1.00



```

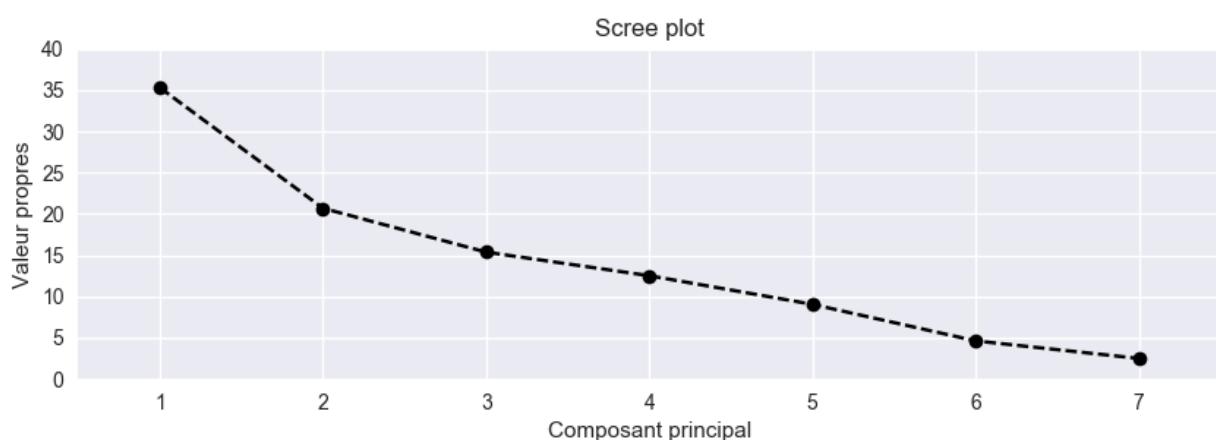
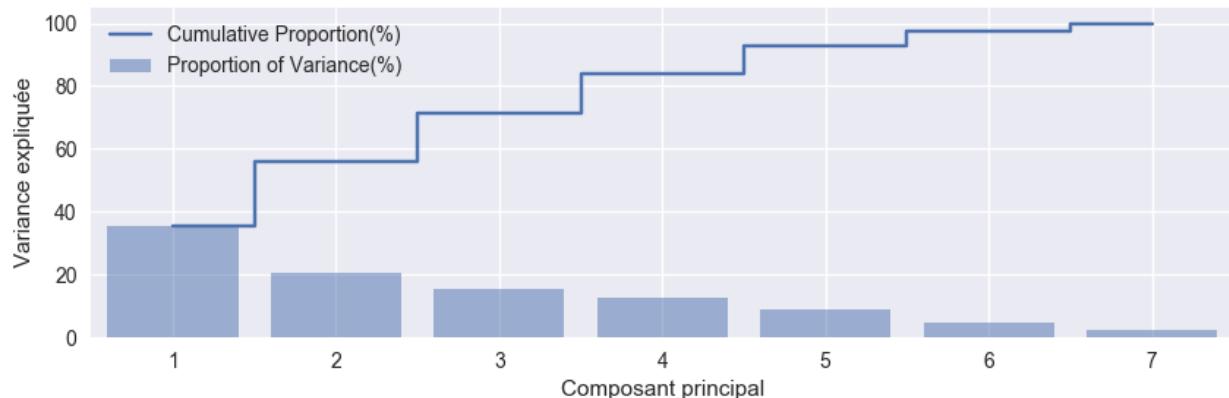
>>> S = np.diag(1/db.ix[:,2:9].std())
>>> mu = np.matrix(db.ix[:,2:9].mean())
>>> U = np.matrix([np.ones(db.ix[:,2:9].shape[0])])
>>> X = pd.DataFrame((db.ix[:,2:9] -
U.T.dot(mu)).dot(S).values,columns = db.columns[2:9])
>>> Var_X = X.cov() # obtenir la matrice variance covariance
>>> val_propres,vec_propres = np.linalg.eig(Var_X)
>>> ordre = val_propres.argsort()[:-1]
>>> val_propres = val_propres[ordre]
>>> vec_propres = vec_propres[:,ordre]
>>> val_propres
array([ 2.47139835,  1.4481121 ,  1.07579038,  0.87588825,  0.63277798,
       0.32250087,  0.17353206])
>>> vec_propres
array([[ 0.51905632,  0.2882542 , -0.13257068,  0.17093657,  0.21782766,
      -0.53914789, -0.51241236],
       [ 0.58653886,  0.11708648, -0.12470379,  0.05594056,  0.06492594,
      -0.04484539,  0.78572009],
      [-0.11658595, -0.22448201, -0.7330986 ,  0.51089215, -0.36917483,
      -0.03559414, -0.00376848],
      [-0.07545644,  0.52547596, -0.32138423, -0.60612799, -0.48618549,
      -0.10539125,  0.00432858],
      [ 0.54983653, -0.02099039,  0.0833643 ,  0.02866106, -0.34878119,
       0.67811177, -0.32860957],
```

```

[ 0.14086425, -0.44646528, -0.49822193, -0.53615785,  0.46196584,
  0.14032646, -0.10968929],
[ 0.21237656, -0.613931 ,  0.26641679, -0.22574925, -0.4905187 ,
 -0.46411971,  0.00534719]])
>>> row = ["Comp." + str(i+1) for i in range(7)]
>>> col = ["Std deviation","Prop.of Variance(%)", "Cumulative Prop.(%)"]
>>> tot = sum(val_propres)
>>> res_1 = pd.DataFrame([[i**0.5,np.round(i*100 /tot,2),
...                         np.round(j*100/tot,2)]
...                         for i,j in zip(val_propres,np.cumsum(val_propres))],
...                         columns= col,index = row)
>>> res_2 = pd.DataFrame(vec_propres,columns = row,
...                         index = db.columns[2:9])
>>> res_1
      Std deviation  Prop.of Variance(%)  Cumulative Prop.(%)
Comp.1        1.572068          35.31           35.31
Comp.2        1.203375          20.69           55.99
Comp.3        1.037203          15.37           71.36
Comp.4        0.935889          12.51           83.87
Comp.5        0.795473           9.04           92.91
Comp.6        0.567892           4.61           97.52
Comp.7        0.416572           2.48           100.00
>>> plt.clf()
>>> plt.subplot(211)
>>> plt.axis([0.5,7.5,0,105])
[0.5, 7.5, 0, 105]
>>> plt.bar(range(1,8), res_1.ix[:,1], alpha=0.5,
...             align='center',
...             label="Proportion of Variance(%)")
>>> plt.step(range(1,8),res_1.ix[:,2], where='mid',
...             label="Cumulative Proportion(%)")
>>> plt.ylabel("Variance expliquée")
>>> plt.xlabel("Composant principal")
>>> plt.legend(loc='best')
>>> plt.subplot(212)
>>> plt.axis([0.5,7.5,0,40])
[0.5, 7.5, 0, 40]
>>> plt.plot(range(1,8),res_1.ix[:,1], linestyle='--',
...             marker='o', color='black')
>>> plt.ylabel("Valeur propres")
>>> plt.xlabel("Composant principal")
>>> plt.title("Scree plot")
>>> plt.tight_layout()
>>> plt.show()
>>> W = res_2.ix[:,0:3] # extraction des vecteurs propres
>>> Z = pd.concat([db.ix[:,0:2],X.dot(W)],axis = 1)
>>> Z.head()
   Pays                    Region     Comp.1     Comp.2 \
0  Afghanistan  Asia-Pacific  -0.936261  1.480023
1    Albania            Europe  -0.599799 -0.525962
2    Algeria  Middle East / North Africa  -0.449635  0.610582
3    Angola       Sub-Saharan Africa  -0.837208  1.129783
4  Argentina  South and Central America / Caribbean  -0.382677  2.837451

```

Comp. 3
0 0.188455
1 -1.024873
2 0.488135
3 -0.079627
4 -1.586402



L'analyse des correspondances multiples ACM

L'analyse factorielle des correspondances a été utilisée en pratique car elle est conçue pour les tableaux de contingence et permet ainsi l'étude des liaisons (dites aussi correspondances) existant entre deux variables nominales. Les domaines d'application de l'AFC sont donc différents de ceux de l'ACP qui est adaptée aux tableaux de mesures hétérogènes ou non.

L'AFC conçue pour les tableaux de contingence (fréquences), peut être appliquée aux tableaux de mesures homogènes (même système d'unités), aux tableaux de notes, de rangs, de préférences, aux tableaux à valeurs logiques (0 ou 1), et encore aux tableaux issus de questionnaires d'enquêtes.

		MODALITÉ DE LA SECONDE VARIABLE	
		1 j J	
MODALITÉ DE LA PREMIÈRE VARIABLE	1	k_{ij}	
	:		:
	:		:
	i	 k_{ij}
	:		:
	:		:
	I		

Les données, à la différence de l'ACP, doivent être organisées en tableaux de contingence. Un tableau de contingence est un tableau d'effectifs obtenus en croisant les modalités de deux variables qualitatives définies sur une même population de n individus.

L'AFC peut également être étendue aux variables quantitatives homogènes en définissant simplement quelques modalités pour ces variables. Par extension, elle s'applique aussi aux tableaux individus-variables pour des variables quantitatives homogènes, dans ce cas les individus sont considérés comme des variables.

		1 j J	marge
		1	$f_{i\bullet}$
marge	1	$f_{ij} = \frac{k_{ij}}{n},$	$f_{i\bullet} = \sum_{j \in J} f_{ij},$
	:		
	:		
	i	 f_{ij}
	:		
	:		
	I		
		$f_{\bullet j}$	
		1	

$$f_{\bullet j} = \sum_{i \in I} f_{ij}.$$

$$\sum_{i \in I} f_{i\bullet} = \sum_{j \in J} f_{\bullet j} = \sum_{i \in I} \sum_{j \in J} f_{ij} = 1.$$

Les objectifs

Les objectifs sont les mêmes que ceux de l'ACP dans le sens où l'AFC cherche donc à obtenir une typologie des lignes et une typologie des colonnes, puis de relier ces deux typologies. Il faut donc faire ressortir un bilan des ressemblances entre lignes (respectivement colonnes) en répondant aux questions du type :

- Quels sont les lignes (respectivement colonnes) qui se ressemblent ?
- Quelles sont celles qui sont différentes ?
- Existe-t-il des groupes homogènes de lignes (respectivement colonnes) ?
- Est-il possible de mettre en évidence une typologie des lignes (respectivement des colonnes) ?

La notion de ressemblance entre deux lignes ou deux colonnes diffère cependant de l'ACP. En effet, deux lignes (respectivement deux colonnes) sont proches si elles s'associent de la même façon à l'ensemble des colonnes (respectivement des lignes), elles s'associent trop ou trop peu par rapport à l'indépendance.

Il faut donc chercher les lignes (respectivement colonnes) dont la répartition s'écarte le plus de l'ensemble de la population, celles qui se ressemblent entre elles et celles qui s'opposent. Afin de relier la typologie des lignes avec l'ensemble des colonnes, chaque groupe de lignes est caractérisé par les colonnes auxquelles ce groupe s'associe peu ou fortement. Par symétrie, chaque groupe de colonnes est caractérisé par les lignes auxquelles ce groupe s'associe peu ou fortement. Ainsi nous pouvons décomposer la liaison entre deux variables en une somme de tendances simples et interprétables et mesurer leur importance respective.

L'analyse factorielle des correspondances AFC

L'exemple qu'on a ici un tableau de données issu d'un questionnaire réalisé sur des Françaises en 1974.

Présentation du tableau de données

1724 femmes ont répondu à différentes questions à propos du travail des femmes, parmi lesquelles :

Quelle est selon vous la famille parfaite ?

- L'homme et la femme travaillent
- L'homme travaille plus que la femme
- Seul l'homme travaille

Quelle activité est la meilleure pour une mère quand les enfants vont à l'école ?

- Rester à la maison
- Travailler à mi-temps
- Travailler à temps complet

Que pensez-vous de la phrase suivante : les femmes qui ne travaillent pas se sentent coupées du monde ?

- Complètement d'accord
- Plutôt d'accord
- Plutôt en désaccord
- Complètement en désaccord

Le tableau de données est formé de deux tableaux de contingence qui croisent les réponses de la première question à celles des deux autres.

Chaque valeur correspond au nombre de femmes ayant donné la réponse en ligne et la réponse en colonne.

	stay.at.home	part-time.work	full-time.work	housewives.cut.from.world.totally.agree	housewives.cut.from.world.quite.agree	housewives.cut.from.world.quite.disagree	housewives.cut.from.world.totally.disagree
both.man.and.woman.work	13	142	106	107	75	40	39
man.works.more	30	408	117	192	175	100	88
only.man.works	241	573	94	140	215	254	299

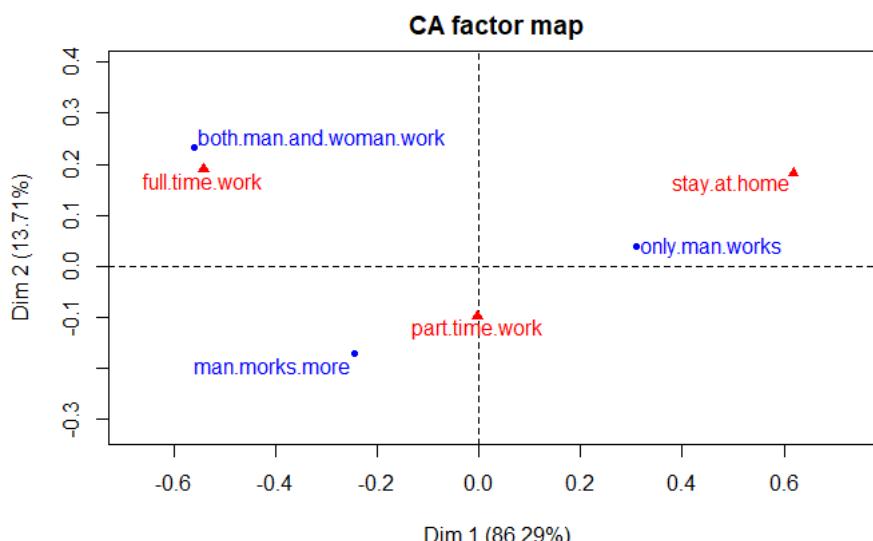
Les objectifs de l'AFC sont similaires à ceux de l'ACP : obtenir une typologie des lignes et des colonnes et étudier le lien entre ces deux typologies.

Cependant, le concept de similarité entre les lignes et les colonnes est différent. Ici, la similarité entre deux lignes ou deux colonnes est complètement symétrique. Deux lignes (resp. colonnes) sont proches l'une de l'autre si elles s'associent aux colonnes (resp. lignes) de la même façon.

On recherche les lignes et respectif les colonnes dont la distribution est la plus différente de celle de la population. Celles qui semblent le plus ou le moins semblables.

Chaque groupe de lignes et respectif des colonnes est caractérisé par les colonnes et respectif les lignes auxquelles il est particulièrement ou particulièrement peu associé.

On utilisera les trois premières colonnes, correspondant aux réponses de la deuxième question, comme variables actives et les quatre dernières, correspondant à la troisième question, comme variables illustratives.



Le nuage des colonnes montre que le premier axe oppose "**Stay at home**" et "**Full-time work**", ce qui signifie qu'il oppose deux profils de femmes.

Les femmes qui ont répondu "**Stay at home**" ont répondu "**Only husband works**" plus souvent que l'ensemble de la population et "**Both husband and wife work**" moins souvent que l'ensemble de la population.

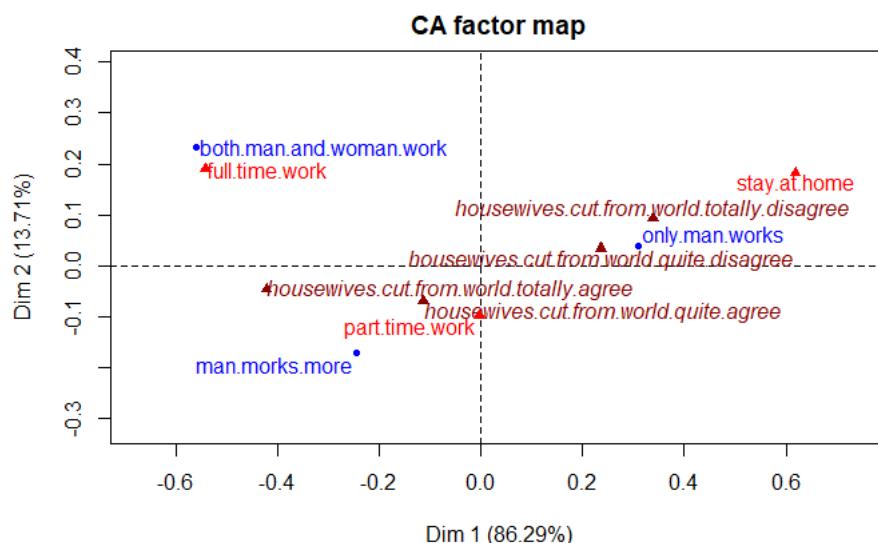
De même, les femmes qui ont répondu "**Full-time work**" ont répondu "**Only husband works**" moins souvent que l'ensemble de la population et "**Both husband and wife work**" plus souvent que l'ensemble de la population. Le premier axe ordonne les modalités de la deuxième question de la moins à la plus en faveur du travail des femmes.

La même interprétation peut être faite pour le premier axe du nuage des lignes. Les modalités sont triées de la moins "**Only husband works**" à la plus "**Both husband and wife work**" en faveur du travail des femmes.

"**Stay at home**" est associé à "**Only husband works**" et peu associé aux deux autres modalités.

"**Both husband and wife work**" est associé à "**Full-time work**" et opposé à "**Stay at home**".

On ajoute les colonnes qui correspondent à la troisième question en tant que variables illustratives.



"**Totally agree**" et "**Quite agree**" pour "**Women who do not work feel cut off from the world**" sont proches des modalités en faveur du travail des femmes.

"**Quite disagree**" et "**Totally "disagree"**" sont proches des modalités opposées au travail des femmes.

L'analyse des correspondances multiples ACM

A titre d'exemple, on utilise ici un tableau de données issu d'un questionnaire sur la consommation de thé. 300 consommateurs de thé ont répondu à un questionnaire sur leur consommation de thé.

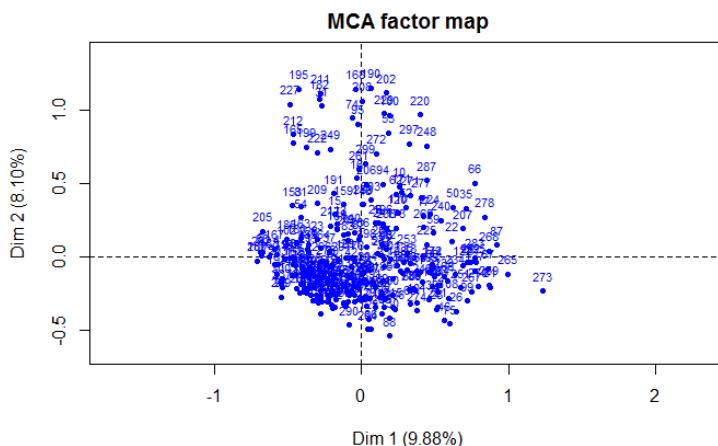
Les questions portaient sur leur façon de consommer le thé et leur image du thé. Le questionnaire comportait également des questions descriptives telles que le sexe, l'âge, la catégorie socio-professionnelle et la pratique régulière d'un sport.

Excepté l'âge, toutes les variables sont qualitatives. Le tableau de données comporte deux variables différentes pour l'âge : une continue et une qualitative.

On étudie les individus, les variables et les modalités.

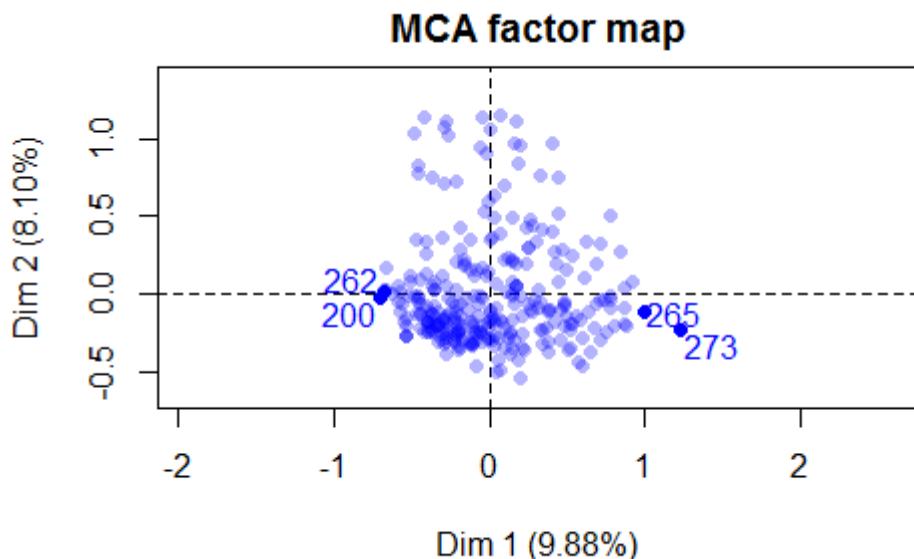
- **Etude des individus** : deux individus sont proches s'ils ont répondu de la même façon aux questions. On ne s'intéressera pas vraiment aux individus en tant que tels mais plutôt aux populations : y a-t-il des groupes d'individus ?
- **Etude des variables et des modalités** : les questions sont les mêmes que pour l'ACP. Premièrement, on veut étudier les relations entre variables et les associations entre modalités. Deux modalités sont proches si elles ont souvent été prises ensemble. Deuxièmement, on recherche une ou plusieurs variable(s) synthétique(s) continue(s) pour résumer les variables qualitatives. Troisièmement, on cherche à caractériser des groupes d'individus par des modalités.

Dans cette étude, les variables sur l'attitude de consommation sont actives et les autres variables sont illustratives.

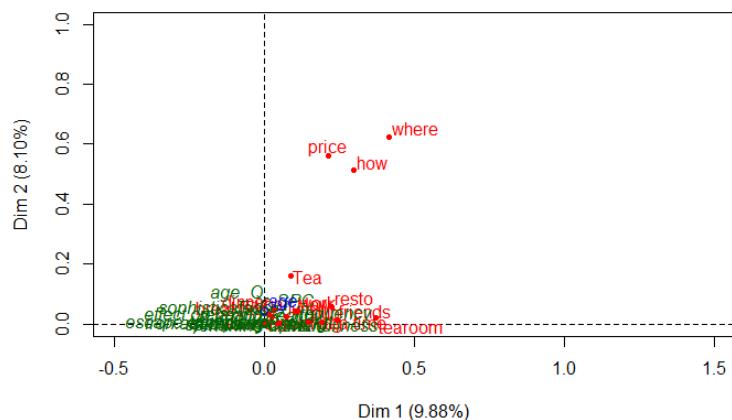


Le nuage des individus ne montre aucun groupe d'individus particulier. Le nuage est assez homogène.

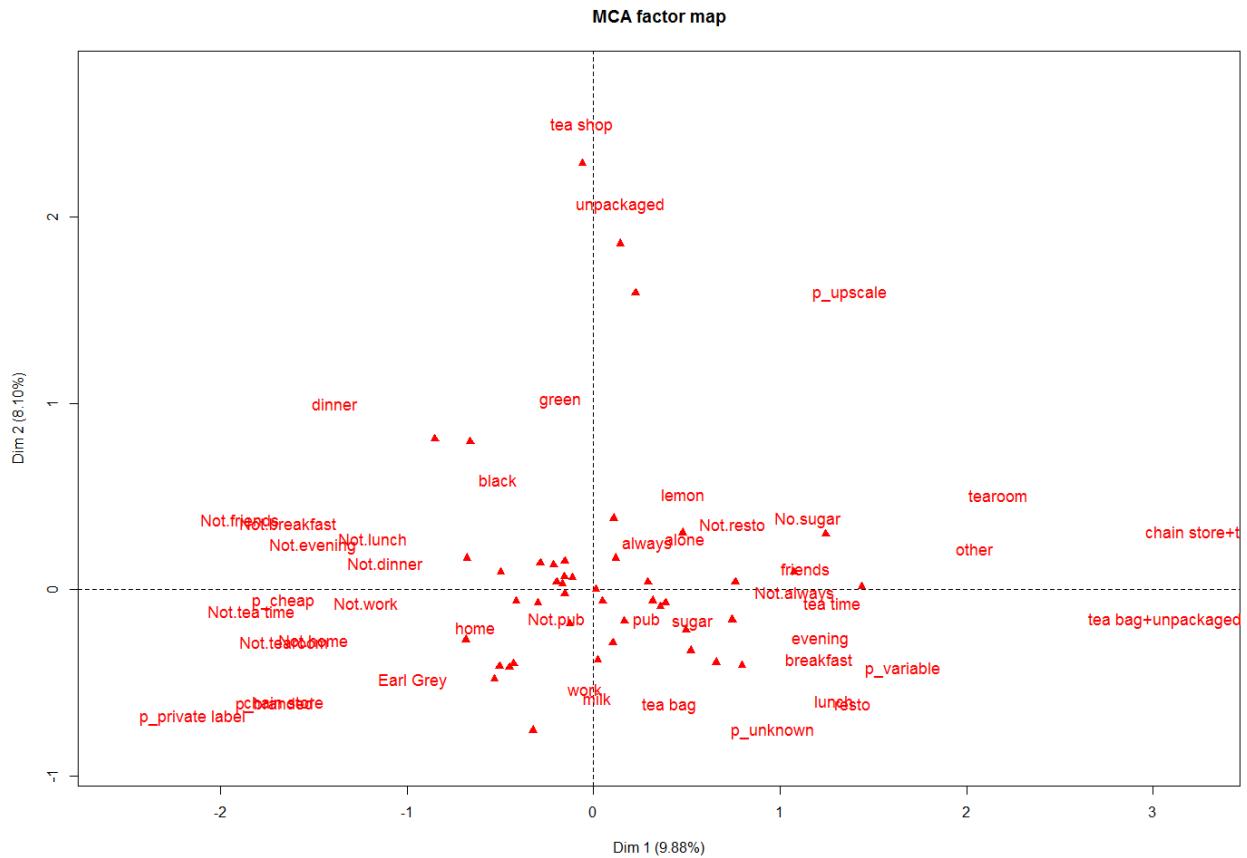
On utilise les individus extrêmes pour interpréter les composantes principales de l'ACM (cela est plus facile que d'utiliser directement les groupes d'individus). Les individus 265 et 273 aiment le thé et en boivent souvent à n'importe quel moment. Les individus 200 et 262 ne boivent du thé que chez eux, au petit-déjeuner ou le soir.



Les individus sont trop nombreux pour qu'on les regarde un par un. C'est pourquoi on a besoin d'une représentation des modalités.

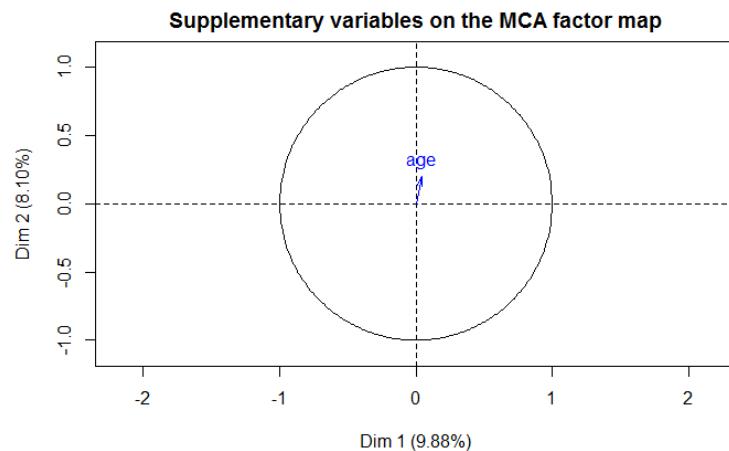


Les variables "price", "where" et "how" sont liées à chacune des deux premières dimensions. On ne peut pas retirer beaucoup plus d'informations de ce graphe. La représentation des modalités va aider à mieux interpréter ces relations.



La première dimension oppose "tea room", "chain store+tea shop", "tea bag+unpacked", "pub", "resto", "work" à "not friends", "not resto", "not work", "not home". Elle oppose les buveurs de thé réguliers aux buveurs occasionnels.

La deuxième dimension oppose "specialized shop", "unpacked" et "upscale price" aux autres modalités.



La variable "âge" n'est pas bien représentée. Cependant, le nombre élevé d'individus entraîne une corrélation significative avec la deuxième dimension (0.204). Les jeunes ont tendance à acheter du thé ailleurs que dans des magasins spécialisés alors que les plus âgés ont tendance à acheter du thé en vrac dans des magasins spécialisés.

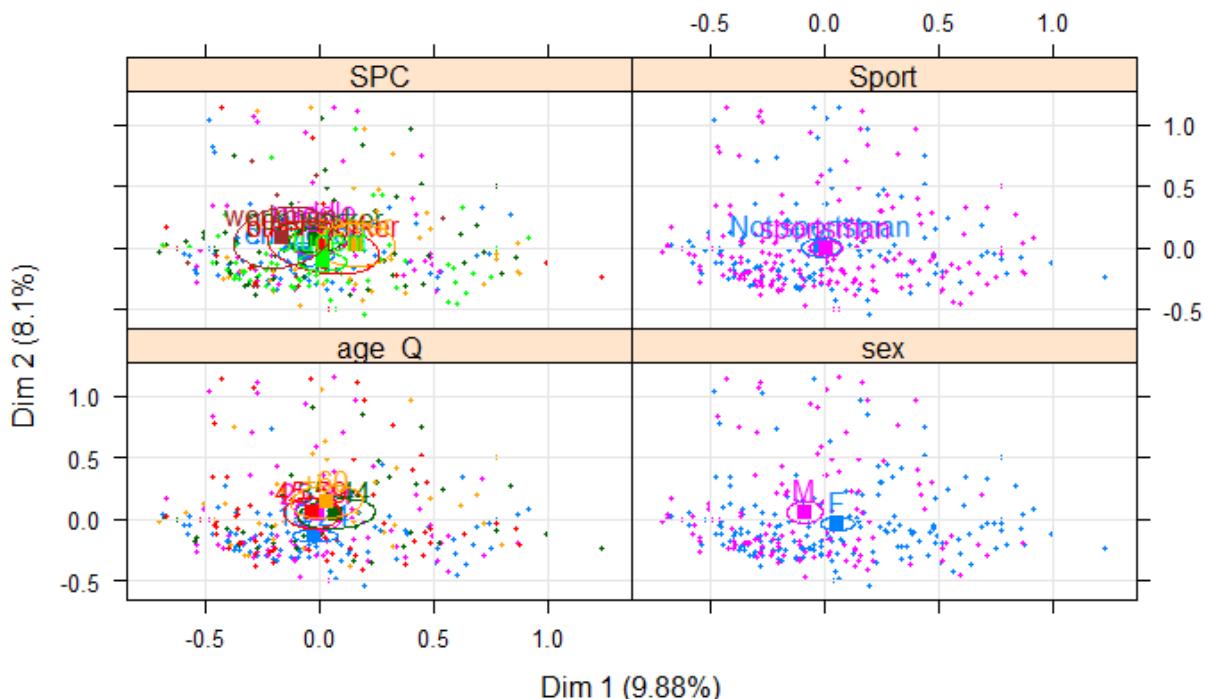
Il est assez difficile de parler des variables qualitatives illustratives car leurs modalités sont localisées au centre du graphique. Néanmoins, il est possible de cacher les

modalités actives pour s'intéresser seulement aux modalités illustratives. On voit alors que les modalités de la variable "age_Q" sont ordonnées de "15-24" à "+60" le long de la deuxième dimension. Ceci est en relation avec la coordonnée positive de la variable "age" sur la deuxième dimension.

La première composante principale est caractérisée par les variables "where", "tea room", etc. Quelques variables qualitatives illustratives lui sont aussi corrélées comme "sex" et "conviviality".

La caractérisation par les modalités est similaire à la caractérisation par les variables mais permet plus de précision. Par exemple, la coordonnée de la modalité "tea room" est positive alors que celle de "not tea room" est négative. Cela signifie que les individus dont la coordonnée sur l'axe 1 est positive ont tendance à fréquenter les salons de thé.

Il est possible de tracer des ellipses de confiance avec la fonction « **plotellipses** ».



9

Les algorithmes de classifications

Naive Bayes

Introduction

Nous voici arrivés à notre premier classifieur. Le *Naive Bayes* est l'un des algorithmes de classification les plus simples à comprendre et donc celui par lequel nous commencerons. Un des domaines d'application classique de ce modèle est la classification de textes en se basant uniquement sur la fréquence d'apparition des mots.

Bien que reposant sur l'hypothèse simpliste (naïve) d'indépendance totale des variables, c'est un algorithme polyvalent dont on retrouve naturellement l'application dans de multiples secteurs d'activité, même quand l'hypothèse de base est violée ! Dans un article de 1997, Domingos et Pazzani se sont même penchés sur les raisons théoriques de cette efficacité contre-intuitive.

Nous avons choisi d'expliquer son fonctionnement par une illustration, plutôt que de passer du temps sur les bases théoriques relativement simples du *Naive Bayes*. Ces bases (théorème de Bayes et notion d'indépendance) sont brièvement rappelées au début du chapitre. Enfin, une formulation générale du modèle termine ces quelques pages consacrées au *Naive Bayes*.

Le théorème de Bayes et la notion d'indépendance

Le théorème de Bayes

En théorie des probabilités, le théorème de Bayes énonce des probabilités conditionnelles. Étant donné deux événements A et B, il permet de déterminer la probabilité de A sachant B, si l'on connaît les probabilités de A, de B et de B sachant A, selon la relation suivante :

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Pour les habitués des probabilités conditionnelles, on écrit parfois :

$$\text{postérieure} = \frac{\text{vraisemblance. antérieure}}{\text{évidence}}$$

Le terme $P(A)$ est la probabilité a priori de A. Elle est « antérieure » au sens qu'elle précède toute information sur B. $P(A)$ est aussi appelée la probabilité marginale de A. Le terme $P(A|B)$ est appelé la probabilité a posteriori de A sachant B (ou encore de A sous condition B). Elle est « postérieure », au sens qu'elle dépend directement de B. Le terme $P(B|A)$, pour un B connu, est appelé la fonction de vraisemblance de A. Et enfin, le terme $P(B)$ est la probabilité marginale ou a priori de B.

La notion d'indépendance

Parlons désormais d'un cas particulier de relation entre A et B : l'indépendance. Deux événements A et B sont dits indépendants quand la probabilité de A ne dépend pas de celle de B. Autrement dit :

$$p(A|B) = p(A)$$

Le *Naive Bayes* repose sur l'hypothèse d'indépendance des variables du problème qu'il traite. C'est une hypothèse forte : supposons que vous soyez en train de modéliser le comportement de vos clients et que vous ayez identifié plusieurs variables discriminantes par rapport à votre problème, notamment leur âge et leur salaire. Il est évident que ces deux variables sont corrélées et donc que l'une dépend de l'autre¹. Eh bien le *Naive Bayes* prendra justement l'assumption contraire, c'est-à-dire que :

$$p(\text{salaire} | \text{âge}) = p(\text{salaire})$$

Voyons maintenant comment le modèle fonctionne, sur la base de cette propriété et du théorème de Bayes.

Le modèle Naive Bayes par l'exemple

Illustrons le fonctionnement de ce modèle par un exemple. Vous travaillez pour un moteur de recherche. Supposons que vous deviez développer un algorithme de placement publicitaire en fonction des recherches des internautes. Vous devez en particulier choisir d'envoyer ou non une publicité à ceux dont la recherche contient le mot « fleur ». Leurs recherches liées à ce mot peuvent correspondre à une volonté d'achat de fleur, mais aussi à une simple recherche de renseignements sur une fleur. En règle générale, l'internaute précisera sa recherche en tapant tantôt « fleur pas cher » ou « bouquet de fleurs » par exemple dans le premier cas et tantôt « fleur rouge » ou « quand arroser fleur ? » dans le second.

1. Le lecteur sceptique face à cette évidence pourra se référer à ce document de l'INSEE pour s'en convaincre : http://www.insee.fr/fr/publications-et-services/docs_doc_travail/g2003-06.pdf. Ce document montre précisément que le salaire dépend de l'âge !

Votre moteur de recherche dispose ainsi d'une importante base de données contenant les recherches des internautes associées à « fleur ». Il dispose aussi d'une autre information importante : les actions consécutives aux recherches, comme les actes d'achat effectifs ou au moins les clics sur des sites commerçants. Ceci permet de constituer une base supervisée, afin de répondre à un problème qui intéresse beaucoup d'annonceurs : quelle recherche a le plus de probabilité d'être un signal annonciateur d'un acte d'achat ? On pourrait par exemple chercher à envoyer une publicité lorsqu'un tel signal est détecté.

Pour cela, on aimerait calculer la probabilité d'achat sachant les mots de la recherche, soit $p(\text{achat} | \text{mots de la recherche})$, étant donné la base de données illustrative suivante du tableau 7-1.

Tableau 7-1. La base de données supervisée des recherches associées au mot « fleur »

ID_internaute	Recherche contenant « pas cher »	Recherche contenant « rouge »	Achat ?
1	Oui	Non	Oui
2	Non	Oui	Non
3	Non	Oui	Non
4	Non	Oui	Non
5	Oui	Non	Non
6	Oui	Oui	Oui

Nous connaissons les états (OUI ou NON) des mots-clés « pas cher » et « rouge » (toujours associés au mot « fleur ») pour chaque internaute, ainsi que leur comportement d'achat. À partir de cette base, on peut par exemple se poser la question suivante : faut-il envoyer une publicité lorsque la recherche contient à la fois les mots-clés « pas cher » et « rouge » ?

Cela revient à évaluer $p(\text{achat} | \text{pas cher, rouge})$. Selon le théorème de Bayes, on peut écrire :

$$(0) \quad p(\text{achat} | \text{pas cher, rouge}) = \frac{p(\text{pas cher, rouge} | \text{achat}) * p(\text{achat})}{p(\text{pas cher, rouge})}$$

C'est maintenant qu'on va utiliser l'hypothèse forte du classifieur sur le terme en gras, à savoir l'indépendance des variables. Cela signifie plusieurs choses :

- (1) $p(\text{rouge} | \text{pas cher}) = p(\text{rouge})$
- (2) $p(\text{pas cher} | \text{rouge}) = p(\text{pas cher})$
- (3) $p(\text{pas cher, rouge} | \text{achat}) = p(\text{pas cher} | \text{achat}) * p(\text{rouge} | \text{achat})$

C'est la troisième expression qui nous intéresse. En l'utilisant pour remplacer le terme en gras de (0), on peut réécrire (0) de la façon suivante :

$$\begin{aligned} & p(\text{achat} | \text{pas cher, rouge}) * p(\text{pas cher, rouge}) \\ &= p(\text{pas cher} | \text{achat}) * p(\text{rouge} | \text{achat}) * p(\text{achat}) \end{aligned}$$

Le terme à droite de l'égalité va nous permettre d'évaluer la probabilité d'achat étant donnée la recherche de l'internaute. En effet, les valeurs des termes $p(\text{pas cher} | \text{achat})$, $p(\text{rouge} | \text{achat})$, $p(\text{achat})$ peuvent directement être trouvés dans la base de données ci-dessus, quelle que soit la valeur de la variable « achat » (OUI ou NON). Ainsi, la probabilité d'achat peut être évaluée en calculant le quotient A suivant :

$$A(\text{pas cher}, \text{rouge}) = \frac{p(\text{pas cher} | \text{achat} = \text{oui}) * p(\text{rouge} | \text{achat} = \text{oui}) * p(\text{achat} = \text{oui})}{p(\text{pas cher} | \text{achat} = \text{non}) * p(\text{rouge} | \text{achat} = \text{non}) * p(\text{achat} = \text{non})}$$

On décidera d'envoyer une publicité si A est supérieur à un seuil α (en général 1, c'est-à-dire qu'il y a plus d'achats que de non-achats pour cette recherche).

Notre micro-base de données va fournir les valeurs des termes suivantes :

$$p(\text{pas cher} | \text{achat} = \text{oui}) = \frac{2}{2} = 1$$

$$p(\text{rouge} | \text{achat} = \text{oui}) = \frac{1}{2} = 0.5$$

$$p(\text{achat} = \text{oui}) = \frac{2}{6} = 0.33$$

$$p(\text{pas cher} | \text{achat} = \text{non}) = \frac{1}{4} = 0.25$$

$$p(\text{rouge} | \text{achat} = \text{non}) = \frac{3}{4} = 0.75$$

$$p(\text{achat} = \text{non}) = \frac{4}{6} = 0.66$$

On peut donc calculer A pour la recherche « fleur pas cher rouge » :

$$A(\text{pas cher}, \text{rouge}) = \frac{1 * 0.5 * 0.33}{0.25 * 0.75 * 0.66} = 1.33$$

$A(\text{pas cher}, \text{rouge}) > 1$: il est donc pertinent d'envoyer une publicité.

Sur le même principe, on peut calculer A pour d'autres recherches :

$$A(\text{pas cher}) = \frac{1 * 0.33}{0.25 * 0.66} = 2$$

$$A(\text{rouge}) = \frac{0.5 * 0.33}{0.75 * 0.66} = 0.33$$

On retrouve bien évidemment un ratio élevé pour « fleur par cher », ce qui signifie que la présence du terme « pas cher » est très discriminante par rapport à notre question de recherche de signaux d'achat ($2 \gg 1$). Quant à lui, le terme « rouge » n'est clairement pas annonciateur d'une volonté d'achat, mais reflète une simple recherche d'information ($0,33 \ll 1$). Dans le premier cas présenté, « fleur pas cher rouge », on observe que la présence de « pas cher » l'« emporte » sur la présence de « rouge » et c'est pourquoi l'on proposera une publicité également. Vous pouvez finir de vous convaincre de tout cela en cherchant dans Google ces trois expressions.

Le cadre général

Plus généralement, et selon une formulation un peu plus rigoureuse, le *Naive Bayes* peut être défini comme suit. Soient N variables (X_1, X_2, \dots, X_n) prenant des valeurs x_{ij} , sur la base desquelles on cherche à prendre une décision binaire $d \in \{0,1\}$.

On calcule le ratio de probabilité A défini par :

$$A = \prod_{i=1}^n \frac{p(x_i | d=1)}{p(x_i | d=0)} * \frac{p(d=1)}{p(d=0)}$$

La prise de décision est alors :
$$d = \begin{cases} 1 & \text{si } A > \alpha \\ 0 & \text{si } A \leq \alpha \end{cases}$$

À noter que le *Naive Bayes* devient un classifieur linéaire quand il est exprimé dans un espace logarithmique $\left(\log \prod_i x_i = \sum_i \log x_i \right)$, mais nous reparlerons de la linéarité des modèles plus tard.

À RETENIR *Naive Bayes*

Apprentissage supervisé – classification

Le *Naive Bayes* est l'un des classificateurs les plus simples.

Il repose sur une hypothèse forte : l'indépendance des variables.

Références

L'article cité en introduction, décrivant la surprenante efficacité de ce modèle pourtant très simple :

- Domingos P., Pazzani M.. 1999. On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. *Machine Learning*, 29:2-3, p. 103-130.

L'excellent MOOC du Dr. Gautam Shroff d'où est reprise notre illustration florale :

- Dr. Gautam Shroff. *Web Intelligence and Big Data*. Coursera.

La régression logistique

Introduction

Nous arrivons désormais à un grand classique de la classification, la célèbre régression logistique. En effet, c'est sans doute le modèle le plus répandu chez les banquiers et les assureurs. Cet algorithme leur offre un bon compromis entre performance du modèle et pouvoir explicatif. Après la lecture de ce chapitre, vous saurez enfin pourquoi on vous a refusé votre dernier crédit !

Comme tout classifieur, son but est donc de prédire des valeurs connues, vues en apprentissage. Pour simplifier la compréhension de cet algorithme, nous nous concentrerons sur un problème de classification binaire, dont les sorties seront $\{0,1\}$. Pour cette explication, nous aurons à préciser un ensemble de concepts importants pour bien comprendre la régression logistique : fonction hypothèse, fonction sigmoïde et fonction de coût.

Nous verrons ensuite comment cette méthode peut se généraliser à des problèmes de classification multilabels, c'est-à-dire lorsque les valeurs de sortie ne sont plus binaires.

Le modèle en détail

La fonction hypothèse

Théoriquement, rien ne nous empêche d'utiliser la même fonction hypothèse que dans le cadre de la régression linéaire (chapitre 3 et chapitre 4). Il suffirait en effet, pour respecter les sorties binaires de notre problème, de post-traiter les données de la sorte :

$$\text{logistic_regression}(x) = \begin{cases} 0, & h(x) < 0.5 \\ 1, & h(x) \geq 0.5 \end{cases}$$

Mais ce traitement est loin d'être optimal pour la classification. En particulier, on préférerait que h représente une probabilité alors que, dans le cas de la régression linéaire, h représente directement des valeurs continues appartenant à \mathbb{R} . Autrement dit, la fonction h pour la classification doit respecter la condition suivante :

$$0 \leq h(x) \leq 1$$

$$h(x) = P(y=1 | x, \theta)$$

Pour modéliser cette propriété en statistique et en *machine learning*, on se réfère fréquemment à des fonctions mathématiques un peu spéciales, dites sigmoïdes, reconnaissables aisément à leur forme en « S » caractéristique.

Les fonctions sigmoïdes

Plusieurs fonctions sigmoïdes peuvent être utilisées dans le cadre d'une régression logistique. Intéressons-nous à quelques-unes des plus connues : la courbe de Gompertz, la tangente hyperbolique et les fonctions de répartition de la loi normale et de la loi logistique. Nous les détaillons ci-après.

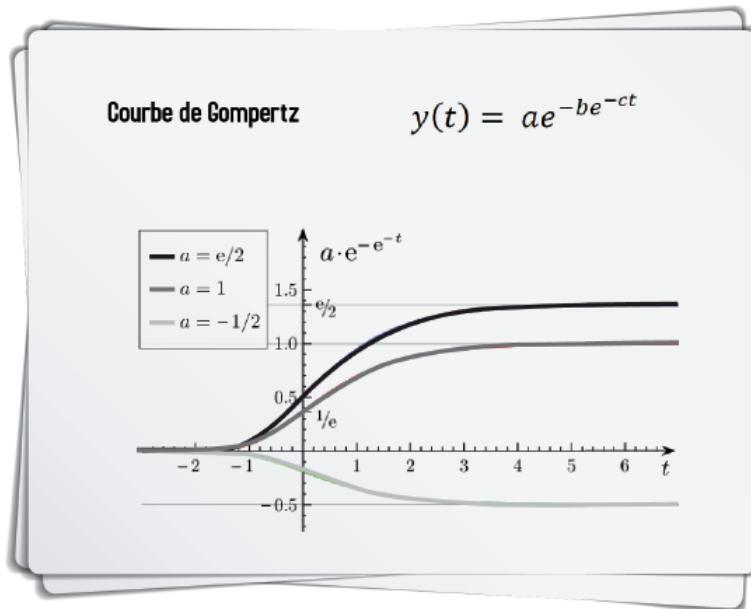


Figure 8-1 – Courbe de Gompertz

La courbe de Gompertz est utilisée pour modéliser certaines séries temporelles, dont la croissance est lente au départ et s'arrête à un moment. Cette fonction est employée pour décrire une population dans un espace confiné, dont la reproduction augmente au début puis ralenti une fois que les ressources à disposition deviennent rares¹.

1. Laird a démontré en 1964 que la courbe de Gompertz était une bonne modélisation de l'évolution des tumeurs, finalement un cas particulier de population (les cellules cancéreuses) disposant de ressources limitées.

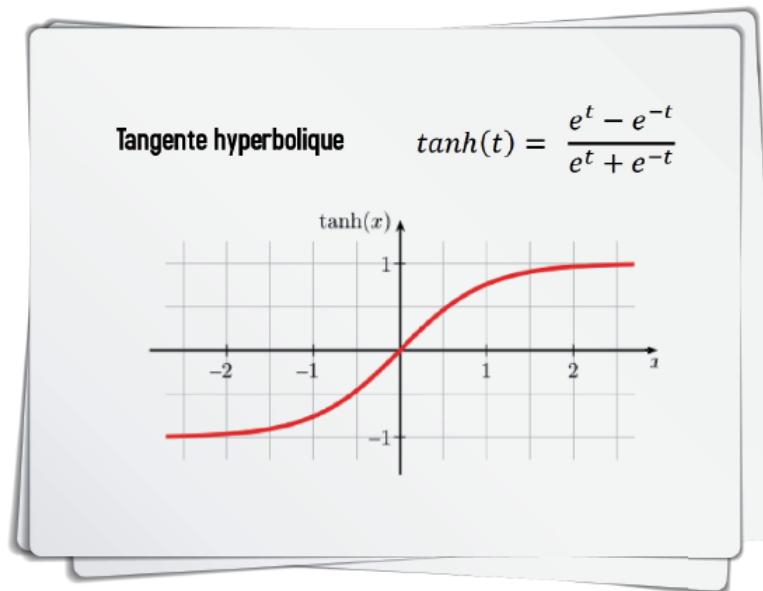


Figure 8-2 – Tangente hyperbolique

Bien que la tangente hyperbolique soit à la base une fonction complexe, on s'intéresse ici à sa restriction à \mathbb{R} , qui est une bijection strictement croissante de \mathbb{R} dans $[-1,1]$.

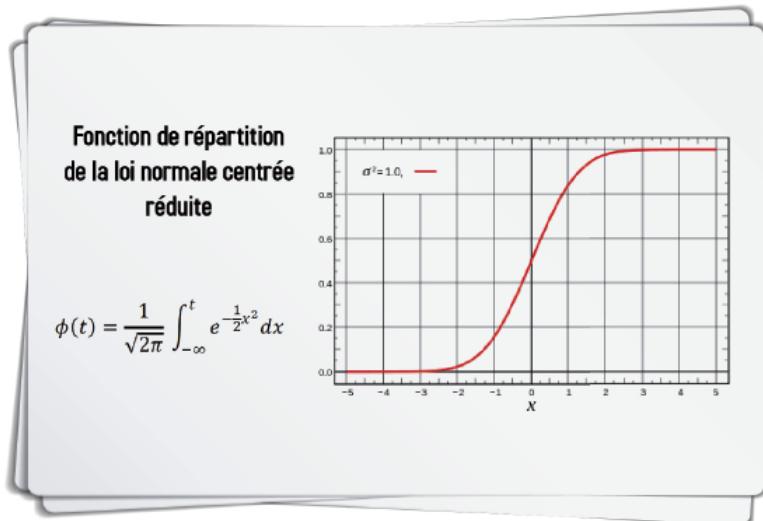


Figure 8-3 – Fonction de répartition de la loi normale centrée réduite

Les fonctions de répartition de la loi de normale centrée réduite sont de magnifiques candidates pour la fonction hypothèse que l'on recherche. Dans les faits, on préférera la fonction de répartition de la loi logistique, décrite ci-après.

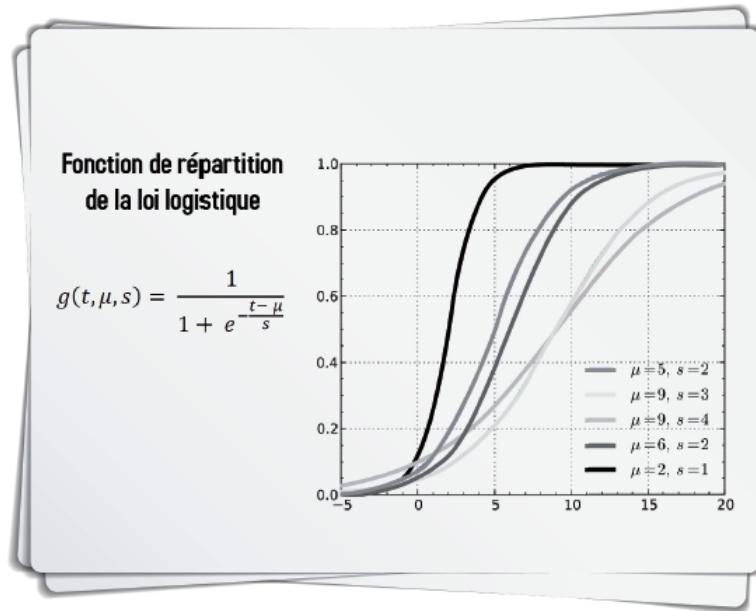


Figure 8-4 – Fonction de répartition de la loi logistique

La loi logistique ressemble beaucoup à une loi normale, mais elle présente un plus grand kurtosis (indicateur qui mesure le coefficient d'aplatissement d'une distribution).

Un cas particulier de la fonction de répartition de la loi logistique est celle de paramètre (0,1). On l'appelle fonction logistique et elle se définit comme suit (figure 8-5).

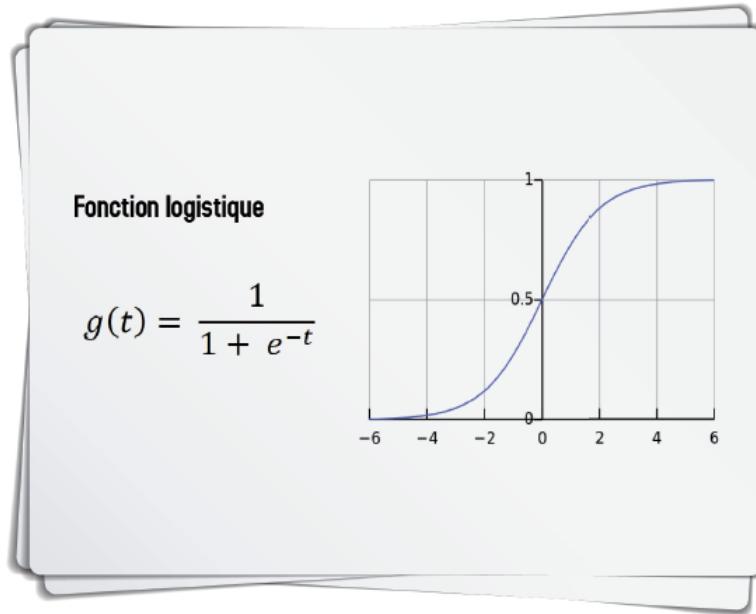


Figure 8-5 – Fonction logistique

Dans la suite de ce livre, on nommera fonction sigmoïde cette fonction logistique. Elle est effectivement, dans la littérature et la pratique, la fonction sigmoïde la plus utilisée.

Reprenons maintenant nos exemples d'apprentissage et notre vecteur de poids Θ pour voir leurs liens avec la fonction logistique, notre fonction hypothèse fraîchement définie. Pour l'observation x_i , la probabilité de prédire un 1 étant donnée x_i et Θ est donnée par :

$$g(x_i\Theta) = g(\theta_0 + \theta_1 x_{i1} + \theta_2 x_{i2} + \dots + \theta_n x_{in})$$

Et la décision est donc :

$$\begin{cases} 1, & g(x_i\Theta) \geq 0.5 \\ 0, & g(x_i\Theta) < 0.5 \end{cases}$$

Ce qui est parfaitement équivalent à :

$$\begin{cases} 1, & x_i\Theta \geq 0 \\ 0, & x_i\Theta < 0 \end{cases}$$

Si on se rappelle que Θ et x_i sont des vecteurs de même dimension, on remarque que la décision est basée sur le produit scalaire de ces deux vecteurs. Autrement dit, si Θ et x_i sont « dans la même direction », leur produit scalaire, positif, rendra le terme $e^{-x_i\Theta}$ inférieur à 0,5. La fonction de décision $\frac{1}{1+e^{-x_i\Theta}}$ devient alors très proche de 1, et ceci d'autant plus que Θ et x_i se rapprochent de la colinéarité. L'inverse est évidemment vrai : si le produit scalaire de Θ et x_i est négatif, la décision sera d'autant plus proche de 0 que Θ et x_i seront anti-colinéaires. Clarifions ce lien entre géométrie et prise de décision pour la régression logistique avec le schéma 8-6.

On remarque sur la figure 8-6 que x_2 sera classé à 0 alors que x_1 et x_3 seront tous deux classés à 1. Néanmoins, on peut observer que la décision sera plus franche pour x_3 que pour x_1 . En effet, la projection de x_1 sur Θ est faible, les deux vecteurs étant quasiment orthogonaux, et la fonction de décision est proche de 0,5.

Méfiez-vous des prédictions qui sont proches de la frontière de décision ! Sachez qu'en modifiant quelque peu les méta-paramètres de la régression logistique ou bien les observations utilisées en apprentissage, une probabilité de 0,49 peut facilement se transformer en 0,51. Cela peut donc changer drastiquement votre décision² (ce qui n'est pas le cas pour la régression linéaire du chapitre 3 et du chapitre 4). Ce n'est probablement pas très critique si vous souhaitez prédire les survivants du Titanic (nous parlerons de cela plus en détail dans le chapitre 17 de la partie pratique de ce livre). Mais cela va sérieusement le devenir si vous devez annoncer à un patient qu'il a un cancer ou pas ! Nous reviendrons plus tard sur la criticité des faux positifs. Pour le moment, remarquons que nous n'avons toujours pas dit comment trouver le meilleur vecteur Θ ...

2. Voilà peut-être à quoi s'est joué votre dernier refus de crédit...

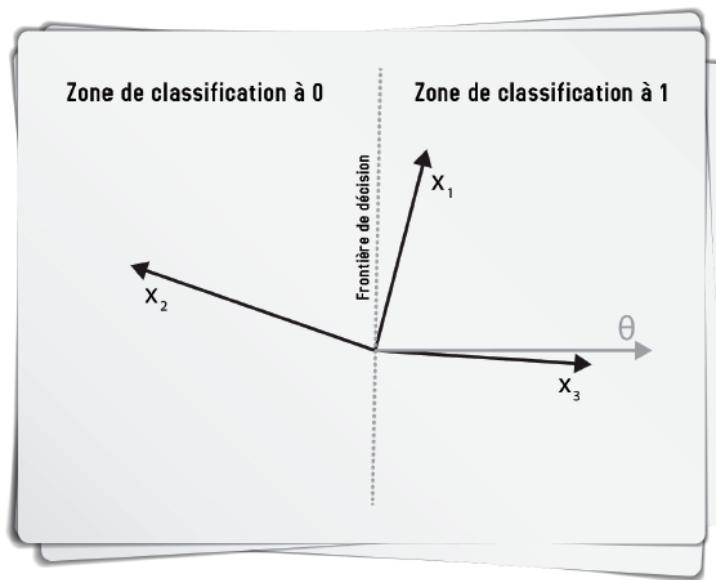


Figure 8-6 – C'est le produit scalaire de l'observation et de Θ qui détermine la décision

La fonction de coût

Comme dans le cas de la régression linéaire, c'est la minimisation d'une certaine fonction de coût qui permet de trouver le meilleur vecteur de paramètres. On souhaite construire, dans le cas d'un algorithme de classification, une fonction de coût qui pénalise très fortement les faux positifs et les faux négatifs. C'est le cas de la fonction par morceau de la figure 8-7.

Prenons le cas d'un faux négatif (la prédiction est de 0 à tort). Comme y est en réalité égal à 1, J est égal pour ce cas précis à $-\log(h(x))$. En revanche, la prédiction étant égale à 0, $h(x)$ est inférieur à 0,5 et, possiblement, proche de zéro.

On voit bien que notre fonction remplit parfaitement son rôle puisque pour cet exemple la fonction J devient quasi infinie. On laissera le soin au lecteur de vérifier que ceci est également valable pour les faux positifs.

Note pour les *Kagglers* : quand la métrique d'évaluation est le *log loss*, on fera attention à borner nos prédictions pour éviter les trop fortes pénalités sur les faux positifs/négatifs. En général, le script d'évaluation de la plateforme le fait pour vous et borne à environ 15 la pénalité par faux. Vous pouvez, par précaution, post-traiter vos probabilités avec le script Python suivant :

```
bound_limit = 10e-8
def logloss(p, y):
    p = max(min(p, 1. - bound_limit), bound_limit)
    return -log(p) if y == 1. else -log(1. - p)
```

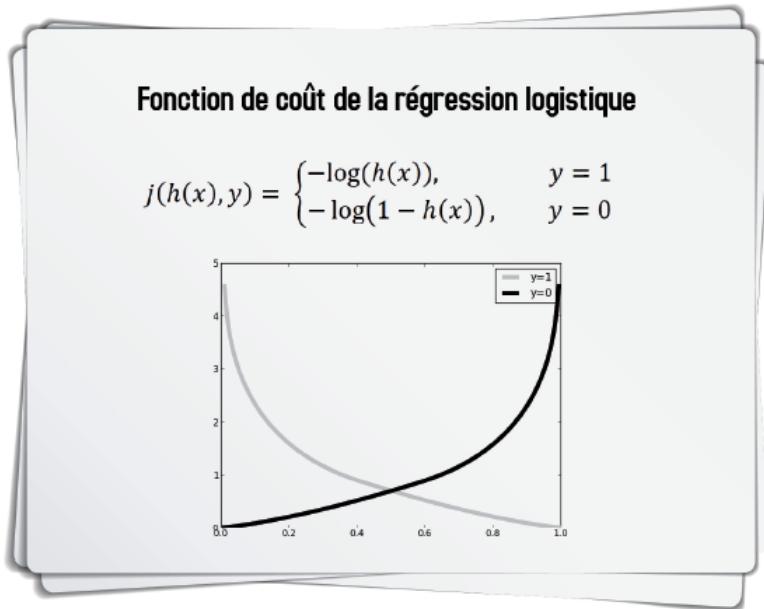


Figure 8-7 – Fonction de coût de la régression logistique

Revenons à notre fonction de coût. En observant que y est toujours égal à 0 ou 1, on peut réécrire plus simplement la fonction, pour un exemple d'apprentissage :

$$j(h(x), y) = -y \log(h(x)) - (1-y) \log(1-h(x))$$

On peut ensuite sommer sur les erreurs de tous les exemples d'apprentissage, pour obtenir la fonction de coût globale :

$$J(\theta) = \frac{1}{m} \sum_{i=1}^n j(h(x_i), y_i)$$

Minimisation de la fonction de coût

Cette fonction $J(\theta)$ a l'énorme avantage d'être convexe et on peut donc appliquer la descente de gradient, que l'on rappelle ici :

- itération 0 : initialisation d'un vecteur $(\theta_0, \theta_1, \theta_2, \dots, \theta_n)$
- itérer jusqu'à convergence :

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ pour } j = 0, \dots, n$$

Sachez, même si nous ne nous étendrons pas sur ce point, qu'il existe d'autres méthodes pour minimiser la fonction de coût. Citons par exemple la méthode BFGS, qui est un algorithme que vous retrouverez assez fréquemment implémenté dans vos outils logiciels.

Derrière la linéarité

En aparté, abordons un point qui pose problème à bon nombre de personnes. Qu'appelle-t-on, bon sang, linéarité en *machine learning* ?

Sans doute avez-vous déjà rencontré cette définition d'une application linéaire :

Soit f une application de E dans F , E et F étant deux espaces vectoriels sur un corps K . L'application f est linéaire si et seulement si :

$$\forall (x, y) \in E^2, \forall \lambda \in K, f(\lambda x + y) = \lambda f(x) + f(y)$$

Ce n'est certainement pas le cas de la fonction sigmoïde, qui est franchement non linéaire. Alors pourquoi dire que la régression logistique est encore un modèle linéaire ?

Tout simplement parce qu'en *machine learning*, la linéarité s'entend par rapport aux variables du problème. Un modèle linéaire est donc un modèle qui, quelle que soit l'observation i , appliquera toujours le même poids θ_j à la variable correspondante x_{ij} .

Les modèles linéaires présentent donc l'avantage d'être facilement explicables, mais ont l'inconvénient de moins capturer des interactions complexes entre variables une fois celles-ci définies. Par exemple, dans le fameux modèle linéaire de *scoring* de crédit de votre banquier, on accordera toujours le même poids à la variable salaire, et ceci indépendamment de l'âge du client³. On pourrait trouver intéressant de pondérer plus fortement cette variable en fonction de l'âge, pour capturer par exemple un jeune de 32 ans qui gagne 150 000 € par an. Cet individu semble en effet avoir un plus fort potentiel qu'un cadre en fin de carrière gagnant le même salaire.

Si vous êtes perdus, réfléchissons à ce cas simple. Comment séparer linéairement ces observations ?

On voit bien qu'aucune droite (plus généralement hyperplan) ne permet de séparer correctement les cas positifs (+) et les cas négatifs (O). Mais rien ne nous interdit de nous positionner dans un autre espace de variables. Un choix judicieux ici est d'élever au carré les variables X_1 et X_2 . Dans ce nouvel espace, nos observations deviendraient linéairement séparables !

Même si la transformation appliquée aux variables est non linéaire, le modèle reste linéaire en ce sens que ce sont toujours les mêmes poids de la régression logistique qui s'appliqueront sur X_1^2 et X_2^2 .

3. Nous caricaturons un peu. En réalité, les banquiers sont un peu plus malins que ça : ils découpent les variables continues pour en faire des catégories. L'âge sera ainsi découpé en tranches d'âge qui deviennent des variables catégorielles. Bien que fastidieuse, cette étape manuelle va créer de belles variables que le modèle linéaire va capturer. Vous pouvez voir l'apport de ce travail dans le chapitre « Prédire les survivants du Titanic ».

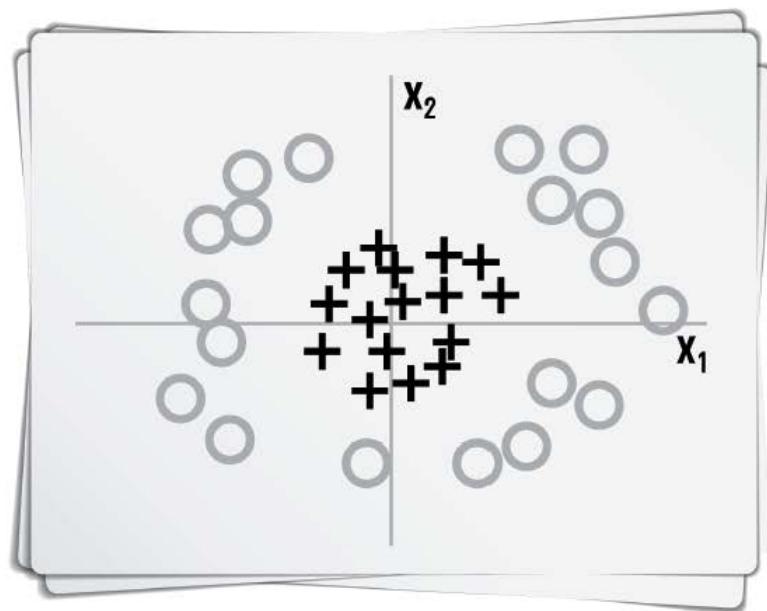


Figure 8-8 – Il n'est pas possible de séparer linéairement ces observations dans l'espace formé par les variables (X_1, X_2) .

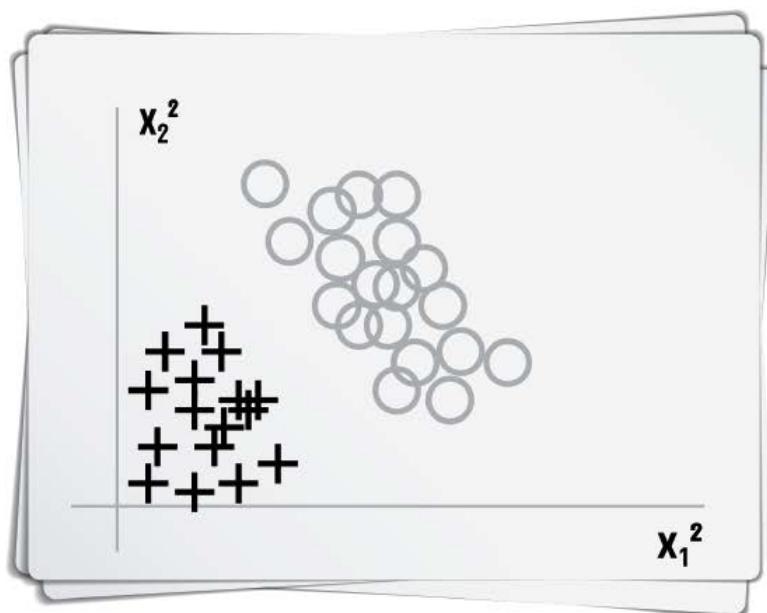


Figure 8-9 – Un choix judicieux de variables permet de séparer linéairement les observations.

Classification multiclasses

Telle que présentée jusqu'ici, la régression logistique pose une contrainte forte : la classification doit être supervisée par un vecteur contenant des valeurs binaires : OUI/NON, MALADE/SAIN, NOIR/BLANC, etc. mais le monde est souvent plus nuancé, n'est-ce pas ? Pour cela, il existe une variante de la régression logistique, dite multiclass, qui permet d'entraîner un modèle à partir d'un vecteur Y à I modalités (ou classes), avec $I > 2$ (figure 8-10).

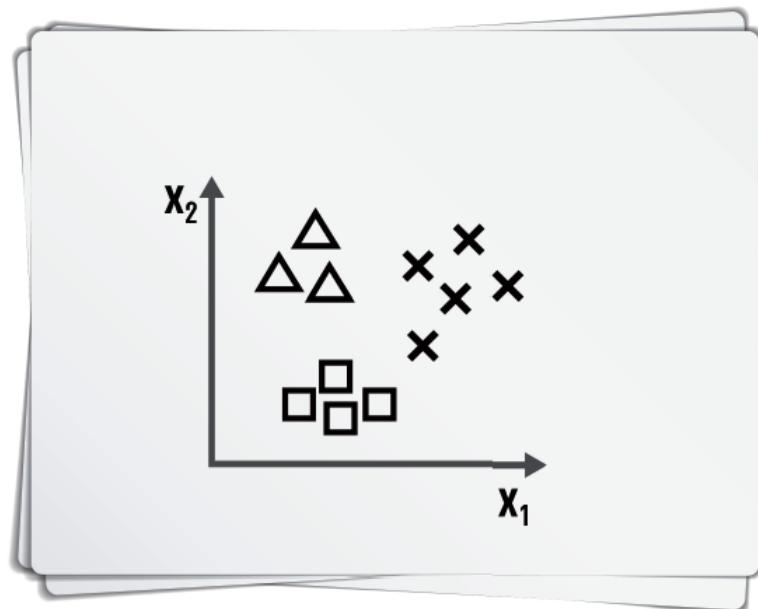


Figure 8-10 – Un problème de classification multiclasses ($I = 3$)

Pour cela, l'algorithme le plus utilisé en *machine learning*⁴ est l'algorithme *one-versus-all* (ou *one-versus-rest*). Son principe est simple : il consiste à transformer le problème multiclass en plusieurs problèmes binaires. Pour cela, il attribue un index de classe allant de 1 à I à chaque modalité pouvant être prise par Y . Prenons l'exemple de la figure ci-dessus : \triangle va correspondre à la classe 1, \square à la classe 2 et \times à la classe 3. La classification va alors être décomposée en trois étapes.

4. En statistique classique, on parlera plutôt de régression logistique polytomique et on emploiera des méthodes de modélisation un peu différentes.

- Étape 1 : on considère que les Δ représentent des cas positifs et \square et \times des cas négatifs. On entraîne un classifieur $h^{(1)}(x)$ pour cette représentation des données, qui saura reconnaître les Δ de tout ce qui n'est pas des Δ .
- Étape 2 : \square deviennent les cas positifs, Δ et \times les cas négatifs, et on entraîne un classifieur $h^{(2)}(x)$. Celui-ci sera spécialisé dans la reconnaissance des \square par rapport à tout ce qui n'est pas des \square .
- Étape 3 : même principe avec \times pour cas positifs, Δ et \square pour cas négatifs, afin d'entraîner $h^{(3)}(x)$, expert en \times .

On peut représenter cela comme dans la figure 8-11.

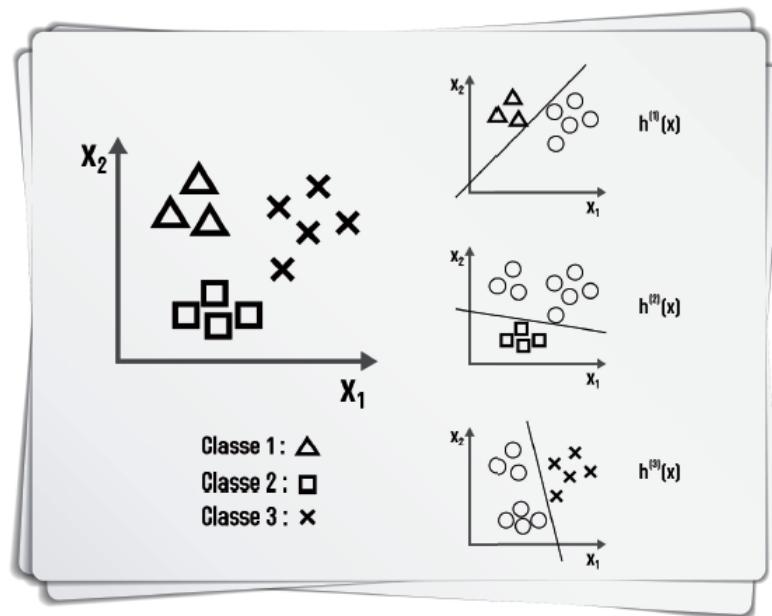


Figure 8-11 – L'algorithme one-versus-all : décomposition du problème en classifications binaires

Dans le cas général, on va donc entraîner un classifieur classique de régression logistique $h^{(i)}(x)$ pour chacune des i valeurs pouvant être prise par le vecteur Y , afin de prédire la probabilité que $y = i$. On peut écrire :

$$0 \leq h^{(i)}(x) \leq 1$$

$$h^{(i)}(x) = P(y = i | x, \theta) \quad (i = 1, \dots, I)$$

Ensuite, pour utiliser ce modèle afin d'effectuer des prédictions, on applique chacun des I classifiants aux nouvelles observations. On obtiendra ainsi i probabilités de classification. La probabilité

la plus élevée indiquera la classe à laquelle la nouvelle observation est la plus susceptible d'appartenir. Autrement dit, on cherche :

$$\max_i h^{(i)}(x)$$

Par exemple, faisons une prédiction pour une nouvelle observation du problème de notre exemple. $h^{(1)}(x)$ nous apprend que la probabilité que la nouvelle observation soit un \triangle est de 0,15 (sans avoir à distinguer si les autres observations sont des \square ou des \times) ; $h^{(2)}(x)$ que la probabilité que la nouvelle observation soit un \square est de 0,8 ; et $h^{(3)}(x)$ que la probabilité que la nouvelle observation soit un \times est de 0,05. En conséquence, on peut raisonnablement penser que la nouvelle observation est un \square . On s'assurera que les probabilités somment à 1, pour que la sortie de la régression logistique soit bien une loi de probabilité.

Régularisation

Tout comme la régression linéaire multiple, la régression logistique peut engendrer des solutions instables, causées par des termes polynomiaux trop élevés ou un nombre trop important de variables. Le traitement de ce problème par les méthodes de pénalisation a déjà été largement discuté dans les chapitres précédents, c'est pourquoi nous n'en dirons pas plus ici. En effet, le principe est exactement le même que pour la régression linéaire : il suffit d'ajouter un terme de pénalisation (norme l_1 , l_2 ou autre) à la fonction de coût de la régression logistique.

À RETENIR Régression logistique

Apprentissage supervisé – classification

La régression logistique est un algorithme de classification très répandu, offrant un pouvoir explicatif très fort du fait de sa linéarité.

La fonction de coût de la régression logistique est basée sur le *log loss*, métrique très importante qui pénalise les faux positifs et les faux négatifs. Par construction, la régression logistique minimise directement le *log loss*.

Références

À propos des fonctions sigmoïdes :

- http://en.wikipedia.org/wiki/Sigmoid_function
- Laird AK. 1964. Dynamics of tumor growth. *British Journal of Cancer*, 18:3, p.490-502.

Sur l'optimisation des fonctions convexes :

- Cohen G. 2000. *Convexité et optimisation*. École d'ingénieur, École Nationale des Ponts et Chaussées.

Quelques éléments théoriques sur les méthodes d'optimisation de Newton et la méthode BFGS :

- Skajaa A. 2010. *Limited Memory BFGS for Nonsmooth Optimization*. Master's thesis, Institute of Mathematical Science New York University.

Le clustering

Introduction

Dans certains cas, on peut être amené à analyser un grand nombre d'objets ou d'individus qu'on cherche à répartir en catégories de façon non supervisée. Grâce aux méthodes de *clustering*¹, on peut mettre en évidence des familles d'individus homogènes selon un critère donné. Les possibilités d'applications sont nombreuses. Par exemple, on peut s'en servir dans l'analyse de bases de données marketing, pour trouver des groupes de clients aux comportements similaires et auxquels on pourra adresser des campagnes commerciales personnalisées. Autre exemple : une de nos missions nous a mené chez un grand industriel, où nous réalisons un *clustering* des pièces mécaniques produites en fonction de leurs caractéristiques de production, de façon à identifier des sources de variation et de défaillance dans le processus de production.

Au sens où nous l'entendons dans ce livre, le *clustering* est typiquement un cas d'apprentissage non supervisé : l'appartenance des observations aux familles de sortie n'est pas connue a priori. Pour votre culture générale, sachez que lorsque l'on parle de *clustering* au sens large avec des statisticiens français (donc de classification... voir note de bas de page), on peut faire référence à des approches supervisées où l'on affecte des objets à des classes identifiées au préalable : on parle alors plutôt de techniques de classement. Néanmoins, ces approches spécifiques sont très connotées « stateux » et ne seront pas développées dans ce livre. Le *clustering* est une branche extrêmement prolixe et diversifiée de l'analyse de données, qui n'a eu de cesse de se développer,

1. Attention : dans la terminologie statistique française, on parle de classification ou de classification automatique pour ce type de méthodes. Nous n'emploierons pas ce terme ici, pour éviter toute confusion avec la terminologie usuelle en *machine learning*, dans laquelle la classification correspond à un apprentissage de type supervisé avec un nombre fini de valeurs de sortie.

de se perfectionner... et de se complexifier depuis les années 1970. Aussi, nous n'en détaillerons pas toutes les subtilités, mais en introduirons les grands principes.

Le problème de base est classique : partant d'une matrice X de données, comment en extraire de façon automatique des groupes nommés *clusters* (ou classes par les statisticiens) ? On distingue deux principales approches.

- Le *clustering* hiérarchique : on va ici construire une suite de *clusters* emboîtés les uns dans les autres. Ces relations entre *clusters* de différents niveaux peuvent ainsi être représentées dans le cadre d'une hiérarchie arborescente. Plus on se situe bas dans l'arbre, plus les observations se ressemblent. Cette méthode peut ainsi être appliquée lorsqu'on ne connaît pas à l'avance le nombre k de groupes dans lesquels seront réparties les observations.
- Le *clustering* non hiérarchique (aussi nommé partitionnement par les statisticiens français) : les individus sont répartis en k classes, pour une valeur de k fixée. Chaque individu n'appartient qu'à un seul groupe.

Dans les deux cas, un bon *clustering* devra permettre de produire des groupes avec :

- une forte similarité intraclasse (ou faible variabilité intraclasse), ce qui signifie que les individus d'un groupe donné doivent se ressembler ;
- et une faible similarité interclasses (ou forte variabilité interclasses), ce qui signifie que les individus de groupes distincts ne doivent pas se ressembler.

Une fois les *clusters* constitués, on peut ensuite les interpréter en utilisant par exemple des outils statistiques usuels tels que moyennes, écarts-types, min/max, etc. Simple, non ?

Voyons désormais comment on procède pour créer ces *clusters*. Pour cela, nous présentons successivement le *clustering* hiérarchique et le *clustering* non hiérarchique. Nous montrerons aussi que ces approches peuvent être utilisées conjointement.

Le clustering hiérarchique

Principe

La construction d'un *clustering* hiérarchique passe par l'emploi d'algorithmes qu'on peut distinguer selon deux grandes familles :

- les algorithmes ascendants (agglomératifs) construisent les classes par aggrégations successives des objets deux à deux ;
- les algorithmes descendants (divisifs) réalisent des dichotomies progressives de l'ensemble des objets.

Dans les deux cas, les algorithmes aboutissent à la constitution d'un arbre appelé dendrogramme, qui rassemble des individus de plus en plus dissemblables au fur et à mesure qu'on s'approche de la racine de l'arbre. La figure 9-1 illustre le mode de construction du dendrogramme selon les deux méthodes.

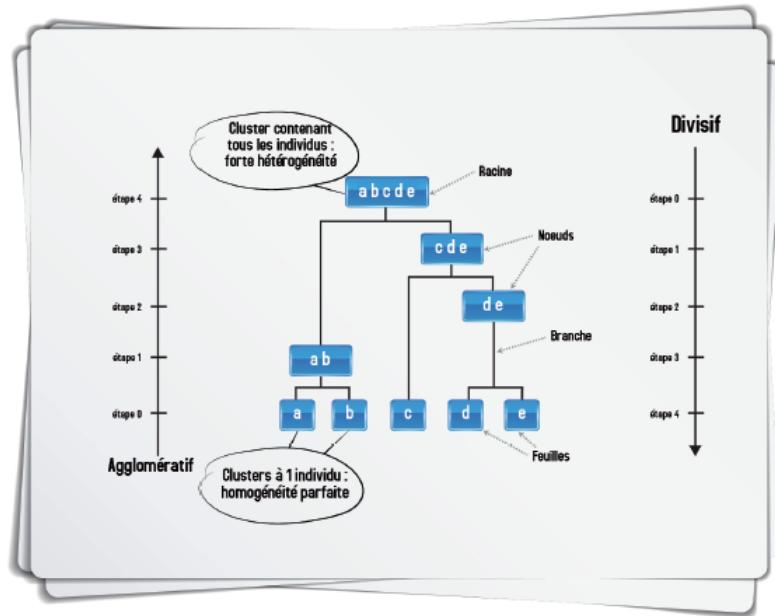


Figure 9-1 – Deux modes de constructions d'un dendrogramme : agglomératif et divisif

La méthode agglomérative est la plus populaire² et nous ne rétablirons pas l'injustice aujourd'hui : présentons les grands principes du *clustering* hiérarchique ascendant. On peut résumer les grandes étapes de l'algorithme comme suit (tableau 9-1, selon Lebart *et al.*, 2006) :

 Tableau 9-1. Algorithme de *clustering* hiérarchique

Étape 1	Il y a m observations à classer.
Étape 2	On construit une matrice des distances (ou, plus généralement, des dissimilarités – nous préciserons ce terme plus loin –) entre les m observations et l'on cherche les deux plus proches, que l'on agrège en un nouvel élément. On obtient un premier <i>clustering</i> à $m-1$ classes.
Étape 3	On construit une nouvelle matrice des distances qui résultent de l'agrégation précédente, en calculant les distances entre le nouvel élément et les observations restantes (les autres distances restent inchangées). On se trouve dans les mêmes conditions qu'à l'étape 1, mais avec $m-1$ observations à classer et en ayant choisi un critère d'agrégation (nous préciserons ce terme plus loin). On cherche de nouveau les deux observations (ce peut aussi être l'un des clusters constitués) les plus proches, que l'on agrège. On obtient un deuxième <i>clustering</i> avec $m-2$ classes, qui englobe le premier <i>clustering</i> .
Étape m	On calcule les nouvelles distances, et l'on réitère le processus jusqu'à n'avoir plus qu'un seul élément regroupant toutes les observations : c'est le <i>clustering</i> final.

2. Pour diverses raisons que nous ne commenterons pas, variables selon les auteurs : grossièreté des résultats de l'approche descendante pour les uns, coûts de calcul pour les autres, etc.

Ces regroupements sont représentés par le dendrogramme. Celui-ci est une hiérarchie indicée : chaque niveau a un indice (ou niveau d'agrégation) qui représente de manière générale la distance entre ses éléments³.

L'application de cet algorithme pose deux questions : (1) comment calcule-t-on les distances et (2) qu'est-ce qu'un critère d'agrégation ? Par ailleurs, une fois l'arborescence réalisée, on peut aussi se demander combien de classes doivent être conservées.

Les distances

Cette problématique doit être abordée en amont de l'application de l'algorithme. Elle revient à se demander ce qu'on entend par ressemblance entre individus. Cette interrogation préalable explicite est une grande force, puisqu'elle permet de s'adapter à des particularités liées aux données.

La distance la plus intuitive, communément utilisée, est la distance euclidienne standard. Soit une matrice X à n variables quantitatives. Dans l'espace vectoriel \mathbb{R}^n , la distance euclidienne d entre deux observations x_1 et x_2 est donnée par :

$$d(x_1, x_2) = \sqrt{\sum_{j=1}^n (x_{1j} - x_{2j})^2}$$

(les données étant centrées réduites).

D'autres distances peuvent être utilisées selon les spécificités des données et l'objectif de la classification.

Par exemple, si l'on cherche à comparer des proportions, on pourra utiliser la célèbre distance du χ^2 (les statisticiens l'adorent !). Pour chacune des n variables, on considère alors une proportion f_{in} par individu et une proportion moyenne pour la variable f_n . La distance du χ^2 va alors mesurer un écart entre des valeurs observées et des valeurs théoriques, en pondérant la distance euclidienne par l'inverse des proportions moyennes :

$$d(x_1, x_2) = \sqrt{\sum_{j=1}^n \frac{1}{f_n} (f_{1j} - f_{2j})^2}$$

Dans certains cas, on peut avoir recours à des distances qui ne sont plus euclidiennes. C'est le cas de la distance de Manhattan (ou distance city-block ou taxi-distance⁴) :

$$d(x_1, x_2) = \sum_{j=1}^n |x_{1j} - x_{2j}|$$

Par rapport à la distance euclidienne usuelle, qui utilise le carré des écarts, on utilisera notamment cette mesure pour minimiser l'influence des grands écarts.

3. Il peut représenter directement une distance ou un gain d'inertie intraclasse, selon le mode de calcul de la distance employé.

4. En référence à la distance entre deux points parcourue par un taxi se déplaçant dans une ville américaine, où les rues sont soit parallèles, soit orthogonales (voir son illustration au Chapitre 6).

Certaines circonstances impliquent l'utilisation de mesures qui ne sont pas des distances au sens mathématique du terme, car elles ne respectent pas ce que l'on appelle l'inégalité triangulaire⁵. On parle alors de dissimilarité plutôt que de distance. Ces mesures sont particulièrement utiles pour la classification des tableaux de présence/absence, en étudiant, pour parler comme un statisticien, la dissimilarité entre des objets constitués d'attributs binaires. Pour cela, on peut utiliser l'indice de similarité de Jaccard. Désormais, nos individus x_1 et x_2 sont définis par n variables de valeur 0 ou 1. Définissons les quantités suivantes :

- M_{11} : nombre de variables qui valent 1 chez x_1 et x_2
- M_{01} : nombre de variables qui valent 0 chez x_1 et 1 chez x_2
- M_{10} : nombre de variables qui valent 1 chez x_1 et 0 chez x_2
- M_{00} : nombre de variables qui valent 0 chez x_1 et x_2

avec $M_{11} + M_{01} + M_{10} + M_{00} = n$ (chaque paire d'attributs appartient nécessairement à l'une des quatre quantités)⁶

L'indice de Jaccard est alors⁷ :

$$J = \frac{M_{11}}{M_{01} + M_{10} + M_{11}} = \frac{M_{11}}{n - M_{00}}$$

D'où l'on calcule la distance de Jaccard :

$$d(x_1, x_2) = 1 - J$$

Ce type de distance est par exemple utilisé en botanique⁸ (en phytosociologie, plus précisément), pour mettre en évidence des associations d'espèces végétales dans des milieux similaires. Il est aussi utilisé en génomique, pour mesurer la ressemblance entre les séquences des quatre bases constituant les gènes.

Le critère d'agrégation

Que représente cette notion ? C'est simple : imaginez-vous à la première étape de l'algorithme de *clustering*, pour une distance choisie. Calculons la première matrice des distances entre tous les individus : l'application de la formule $d(x_p, x_q)$ s'applique directement et c'est parfait ainsi. Mais à partir des itérations suivantes, nous aurons à considérer des groupes d'individus et la question suivante se posera : comment calculer la distance en présence de groupes ? Différentes possibilités existent.

5. $d(x_1, x_2)$ est une distance mathématique si elle respecte les conditions suivantes :

$d(x_1, x_2) = 0 \Leftrightarrow x_1 = x_2$,

$d(x_1, x_2) = d(x_2, x_1)$,

$d(x_1, x_2) \leq d(x_1, x_3) + d(x_3, x_2)$ (inégalité triangulaire)

6. D'autres métriques peuvent être calculées sur la base de ces quantités : Dice, Ochiaï, Russe et Rao, etc.

7. La deuxième écriture, faisant intervenir n , permet de généraliser l'analyse de similarité sur plusieurs individus, en ne calculant que le nombre de variables qui valent 1 chez tous les individus ou qui valent 0 chez tous les individus.

8. C'est Paul Jaccard, un botaniste suisse, qui a proposé l'indice portant son nom en 1901, dans le Bulletin de la Société Vaudoise des Sciences Naturelles.

Tout d'abord, trois critères qui fonctionnent quelle que soit la mesure de distance ou de dissimilarité (parmi beaucoup d'autres !) :

- le critère du lien minimum (ou lien simple ou *single linkage*), qui va se baser sur les deux éléments les plus proches ;
- le critère du lien maximum (diamètre, *complete linkage*), qui va se baser sur les deux éléments les plus éloignés ;
- et le critère du lien moyen (*average linkage*), qui va utiliser la moyenne des distances entre les éléments de chaque classe pour effectuer les regroupements.

La figure 9-2 illustre ces différents critères.

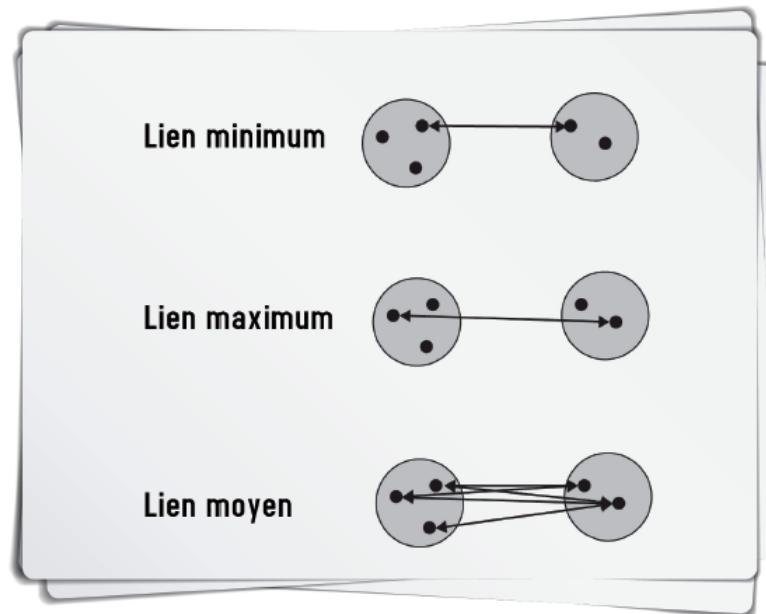


Figure 9-2 – Quelques critères d'agrégation

Dans le cadre de distances euclidiennes, d'autres critères peuvent être employés. Le plus usité est le critère de Ward, qui se base sur ce que l'on appelle l'augmentation de l'inertie. L'inertie totale de l'ensemble des individus est décomposée selon :

- l'inertie intraclasse, qui représente l'écart entre chaque point et le centre de gravité de la classe à laquelle il appartient ;
- l'inertie interclasses, qui représente l'écart entre chaque centre de gravité d'une classe et le centre de gravité général.

Sans entrer dans les détails, utiliser le critère de Ward va revenir à agréger deux classes de façon à ce que l'augmentation de l'inertie intraclasse soit la plus petite possible, pour que les classes restent les plus homogènes possible.

Selon le critère choisi, on peut aboutir à des arbres ayant des formes très différentes. Le saut minimum tend à construire des arbres aplatis, avec des accrochages successifs d'individus un à un (des « chaînes »), alors que le lien maximum tend à former des groupes isolés et très compacts. Le critère de Ward tend quant à lui à produire des classes d'effectifs similaires. Au final, il faut retenir qu'il n'existe pas une unique bonne méthode, tout comme il n'existe pas un unique *clustering*. Il est recommandé de tester plusieurs approches pour un même jeu de données. Les hiérarchies complètes peuvent beaucoup changer, mais, normalement, le « haut » de l'arbre ne doit pas beaucoup varier (les partitions à peu de groupes doivent rester assez stables).

La notion de troncature

Une fois l'algorithme appliqué et les distances calculées, nous pouvons présenter les résultats sous la forme du dendrogramme déjà évoqué. Il représente tous les *clusterings* possibles, du *clustering* le plus fin (un *cluster* par individu) au plus grossier (un *cluster* unique pour tous les individus).

La question suivante peut alors se poser : à quel niveau d'agrégation doit-on « couper » l'arbre, c'est-à-dire comment définir le nombre de *clusters* le plus pertinent ? Des réponses quantitatives peuvent être apportées, basées sur le gain relatif d'inertie engendré par le passage d'une partition à une autre, mais retenez que l'examen visuel du dendrogramme donne souvent de très bonnes réponses. On peut alors se baser sur des critères tels que :

- l'allure générale de l'arbre : elle laisse souvent apparaître un niveau de coupe « logique » indiqué par des sauts importants dans les valeurs des indices de niveau. Si ces sauts concernent les k derniers nœuds de l'arbre, alors un découpage en $(k+1)$ classes sera pertinent ;
- le nombre de *clusters* : éviter un nombre trop grand, auquel cas le *clustering* perd de son intérêt ;
- la capacité à interpréter les *clusters* : inutile de chercher à retenir des *clusters* pour lesquels on n'arrive pas à donner de sens métier ; privilégier les *clusterings* qui ont du sens.

Voilà pour ce tour d'horizon du *clustering* hiérarchique. Vous en trouverez un exemple d'application dans le cas particulier des séries temporelles au chapitre 22.

Nous avons abordé un certain nombre de notions nouvelles, qui vont nous permettre d'aborder plus rapidement le paragraphe concernant le *clustering* non hiérarchique.

Le clustering non hiérarchique

Principe

L'objectif du *clustering* non hiérarchique est le même que pour le *clustering* hiérarchique, mais cette fois-ci, on sait à l'avance le nombre de *clusters* à constituer. Pour une mesure de distance des individus donnée et pour un nombre de classes k connues, on imagine aisément une solution de classification simple et optimale : énumérer toutes les possibilités de regroupement imaginables et conserver la meilleure. Toutefois, cette solution n'est pas applicable en pratique, car le nombre de combinaisons de regroupements possibles devient très rapidement énorme. Mais des solutions approchées peuvent être obtenues grâce à des heuristiques et plusieurs algorithmes de ce genre existent. Leur principe fondamental est contenu dans la méthode des centres mobiles.

Les centres mobiles

Pour k classes définies à l'avance, l'algorithme procède de façon itérative comme le montre le tableau 9-2 (Lebart *et al.*, 2006).

Tableau 9-2. L'algorithme des centres mobiles

Étape 0	On détermine k centres provisoires de <i>clusters</i> (par exemple, par tirage aléatoire de k individus). Ces k centres $\{C_1^0, \dots, C_k^0\}$ induisent un premier <i>clustering</i> P^0 de l'ensemble des individus en k groupes $\{I_1^0, \dots, I_k^0\}$. Ainsi, l'individu i appartient par exemple au groupe I_i^0 s'il est plus proche de C_i^0 que de tous les autres centres.
Étape 1	On détermine k nouveaux centres $\{C_1^1, \dots, C_k^1\}$ en prenant les centres de gravité des <i>clusters</i> qui viennent d'être obtenus ($\{I_1^0, \dots, I_k^0\}$). Ces nouveaux centres induisent un nouveau <i>clustering</i> P^1 , construit selon la même règle que pour P^0 . La partition P^1 est formée de classes notées $\{I_1^1, \dots, I_k^1\}$.
Étape e	On détermine k nouveaux centres $\{C_1^e, \dots, C_k^e\}$ en prenant les centres de gravité des <i>clusters</i> qui viennent d'être obtenus ($\{I_1^{e-1}, \dots, I_k^{e-1}\}$). Ces nouveaux centres induisent un nouveau regroupement P^e , formé des <i>clusters</i> $\{I_1^e, \dots, I_k^e\}$.

Le processus se stabilise nécessairement (en général, rapidement), pour des raisons mathématiques que nous ne détaillerons pas ici. L'algorithme s'arrête soit si deux itérations successives aboutissent au même *clustering*, soit si un critère de contrôle choisi se stabilise (variance intraclasse par exemple), soit si on atteint un nombre d'itérations fixé.

L'algorithme n'aboutit pas nécessairement à un optimum global : le résultat final dépend des centres initiaux. Dans la pratique, on l'exécute souvent plusieurs fois (avec des centres initiaux différents, bien sûr). Ensuite, soit on conserve la meilleure solution, soit on cherche des regroupements stables pour identifier les individus qui appartiennent aux mêmes partitions. Ces ensembles d'individus systématiquement rattachés à une même classe sont appelés « forme forte » ou « groupements stables ».

Parlons désormais de quelques unes des variantes les plus connues des centres mobiles, destinées à en améliorer certains aspects.

Quelques variantes

Elles sont nombreuses, et nous nous limiterons à en évoquer quatre :

- la méthode des *k-means* : elle fonctionne exactement comme les centres mobiles, à une différence près, qui est le calcul des centres. Un recentrage est effectué dès qu'un individu change de *cluster*. On n'attend plus que tous les individus soient affectés à un *cluster* pour en calculer les centres de gravité, ces derniers sont modifiés au fur et à mesure des réaffectations ;
- la méthode des nuées dynamiques : elle favorise la recherche de groupements stables. C'est une généralisation des centres mobiles dont l'idée est d'associer à chaque *cluster* un représentant différent de son centre de gravité. Dans la majorité des cas, on remplace le centre de gravité par un ensemble d'individus, qu'on appelle des « étalons » et qui constituent un « noyau ». Ce noyau est censé avoir un meilleur pouvoir descriptif que des centres ponctuels. À noter que d'autres représentants plus exotiques peuvent être utilisés : une droite, une loi de probabilité, etc. ;
- la méthode *Isodata* : le principe des centres mobiles est conservé, mais des contraintes vont permettre de contrôler l'élaboration du *clustering*. Ces contraintes servent à empêcher la formation de groupes à effectifs trop faibles ou de diamètre trop grand ;

- la méthode des *k-medoids* : elle est similaire à la méthode des *k-means*, à une différence près. En effet, elle ne va plus définir une classe par une valeur moyenne (le centre de gravité, ou *centroid*), mais par son représentant le plus central (le *medoid*). C'est donc un individu du *cluster* qui va représenter ce dernier. Cette méthode a l'avantage d'être plus robuste aux valeurs aberrantes ;
- la méthode des cartes auto-organisées (*Self-Organising Maps*, ou cartes de Kohonen) : cet algorithme diffère des centres mobiles par la mise à jour des *clusters* voisins, où les *clusters* deviennent des neurones activables. Cette méthode non linéaire permet in fine de conserver toute la topologie des données, en partant le plus souvent d'une grille rectangulaire de neurones qui va se déformer.

Arrêtons là l'énumération de méthodes aux noms de plus en plus étranges ! Dans la pratique, nous vous recommandons de sélectionner une méthode « de base », centre mobile ou nuée dynamique par exemple, puis d'aller vers des méthodes plus élaborées si les résultats ne sont pas satisfaisants.

Les approches mixtes

Dans ce chapitre, nous avons présenté les approches hiérarchiques et non hiérarchiques séparément. Mais en pratique, il peut être utile de les utiliser conjointement. Terminons ce chapitre en donnant quelques indications méthodologiques à cet effet.

En mixant les approches, on peut tirer parti des principaux avantages des différentes méthodes, à savoir :

- la capacité à analyser un grand nombre d'individus, point fort des méthodes non hiérarchiques (pour un nombre d'observations supérieur à 10^3 , il devient difficile d'appliquer directement des méthodes hiérarchiques) ;
- le choix d'un nombre de classes optimal, rendu possible par la classification hiérarchique.

Un algorithme mixte global peut être résumé par le tableau 9-3 (Lebart et al., 2006).

Tableau 9-3. L'algorithme mixte de clustering

Étape 1 – Clustering non hiérarchique initial	L'objectif est d'obtenir rapidement une partition de m individus (m grand) en k classes, avec k supérieur au nombre de classes finales s souhaité, grâce à une méthode de <i>clustering</i> non hiérarchique. On prend par exemple $k = 100$, tel que $k \ll m$. Ce <i>clustering</i> n'est pas utilisable directement : les groupes sont nombreux et proches les uns des autres. Toutefois, ils ont l'avantage d'être très homogènes et de contenir des individus qui n'ont pas à être séparés. On peut répéter l'étape pour rechercher des formes fortes.
Étape 2 – Clustering hiérarchique des groupes obtenus	On peut désormais utiliser un algorithme de <i>clustering</i> hiérarchique pour regrouper les k groupes de l'étape 1 (chaque groupe a pour poids la somme des poids qu'il représente). L'étape d'agrégation va permettre de reconstituer des <i>clusters</i> qui ont été inutilement fragmentés lors de l'étape précédente, en sélectionnant un nombre de classes optimal (s). Il convient d'utiliser le critère de Ward pour la constitution de cet arbre, afin de tenir compte des masses des classes.
Étape 3 – Clustering non hiérarchique final	Enfin, un <i>clustering</i> non hiérarchique final est réalisé, pour s groupes définis lors de l'étape 2. Comme pour l'étape 1, on peut répéter l'étape pour rechercher des formes fortes.

En fonction des besoins, tout ou partie de l'algorithme mixte peut être appliqué. Si l'objectif est uniquement de réaliser un *clustering* hiérarchique sur un grand nombre d'individus, on se contentera d'appliquer les étapes 1 et 2 ; si au contraire on a peu d'individus et qu'on veut réaliser un *clustering* non hiérarchique sans définir le nombre de groupes au hasard, on omettra l'étape 1.

À RETENIR Clustering

Apprentissage non supervisé

Les méthodes de *clustering* sont des approches non supervisées visant à regrouper automatiquement les observations les plus homogènes.

Il existe deux grandes familles de *clustering* :

- le *clustering* hiérarchique ;
- le *clustering* non hiérarchique.

Il existe également des algorithmes mixtes ayant recours aux deux familles.

Le choix d'une bonne métrique de distance/similarité est un critère important pour la réalisation d'un bon *clustering*.

Références

Au fil du chapitre, nous avons exclusivement renvoyé à Lebart *et al.* (2006). Complet, rigoureux et accessible, c'est l'un des livres de référence en matière de fouille de données :

- Lebart L., Piron M., Morineau A. 2006. *Statistique exploratoire et multidimensionnelle – Visualisation et inférence en fouilles de données*. 4^e édition, Dunod. Une version en ligne d'une édition plus ancienne est disponible ici : http://horizon.documentation.ird.fr/exl-doc/pleins_textes/divers11-10/010007837.pdf.

Sans y faire référence directement, nous nous sommes également appuyés sur les références suivantes :

- Bouroche JM., Saporta G. 2005. *L'analyse des données*. PUF, Que sais-je ?

La version miniature de Lebart *et al.*, à mettre dans toutes les poches !

- Husson F., Sébastien L., Pagès J. 2009. *Analyse de données avec R*. Presses Universitaires de Rennes.

Une très bonne synthèse opérationnelle des méthodes de *clustering*, avec le code qui permet de les mettre en œuvre sous R en utilisant l'excellente librairie FactoMineR.

- Mirkes EM. 2011. University of Leicester met à disposition un ensemble d'applets, disponibles sur la page de Gorban AN. <http://www.math.le.ac.uk/people/ag153/homepage>.

Elles permettent de se faire une idée intuitive de certaines des méthodes évoquées.

- Roux M. 1997. *Algorithmes de classification*. (<http://www.imep-cnrs.com/docu/mroux/algoclas.pdf>).

Ce livre est extrêmement complet et pédagogique. Il est hélas épuisé, mais disponible en ligne.

- Singh SS., Chauchan NC. 2011. K-means v/s K-medoids: A comparative study. In *National Conference on Recent Trends in Engineering & Technology*. (13-14 May 2011)

Dans cette petite étude comparative intéressante, les algorithmes *k-means* et *k-medoids* évoqués sont comparés.

Introduction aux arbres de décision

Introduction

Ce chapitre introductif passera très rapidement sur les grands principes des arbres de décision utilisés en statistique classique. Sous leur forme simple, ils sont très rarement utilisés en *machine learning*. Par contre, la compréhension de leurs principes et de leurs modes de construction est un prérequis pour l'étude des *random forest*, premier chapitre de la partie présentant les outils d'« artillerie lourde » qui sont à la disposition du *data scientist*.

Principe

Le but général d'un arbre de décision est d'expliquer une valeur à partir d'une série de variables discrètes ou continues. On est donc dans un cas très classique de matrice X avec m observations et n variables, associée à un vecteur Y à expliquer. Les valeurs de Y peuvent être de deux sortes :

- continues : on parle alors d'arbre de régression ;
- ou qualitatives : on parlera d'arbre de classification.

Ces méthodes inductives présentent de nombreux atouts : elles sont assez performantes, non paramétriques¹ et non linéaires. Dans le principe, elles vont partitionner les individus en produisant des groupes d'individus les plus homogènes possible du point de vue de la variable à prédire, en tenant compte d'une hiérarchie de la capacité prédictive des variables considérées. Cette

1. C'est-à-dire que le modèle n'est pas décrit par un nombre fini de paramètres.

hiérarchie permet de visualiser les résultats dans un arbre et de constituer des règles explicatives explicites, orientées métier.

Les grands principes de définition des règles sont les suivants (inspirés de Tufféry, 2011). Plusieurs itérations sont nécessaires : à chacune d'elle, on divise les individus en k classes (généralement $k = 2$), pour expliquer la variable de sortie. La première division est obtenue en choisissant la variable explicative qui fournira la meilleure séparation des individus. Cette division définit des sous-populations, représentées par les « nœuds » de l'arbre. À chaque nœud est associée une mesure de proportion, qui permet d'expliquer l'appartenance à une classe ou la signification d'une variable de sortie. L'opération est répétée pour chaque sous-population, jusqu'à ce que plus aucune séparation ne soit possible. On obtient alors des nœuds terminaux, appelés « feuilles » de l'arbre. Chaque feuille est caractérisée par un chemin spécifique à travers l'arbre qu'on appelle une règle. L'ensemble des règles pour toutes les feuilles constitue le modèle.

L'interprétation d'une règle est aisée si l'on obtient des feuilles pures (par exemple, 100 % de variables à expliquer sont VRAI ou FAUX pour une règle donnée). Sinon, il faut se baser sur la distribution empirique de la variable à prédire sur chaque nœud de l'arbre.

La figure 10-1 illustre deux arbres obtenus par l'analyse de l'activité d'un web *Advertiser*, client d'OCTO. L'objectif est d'expliquer le taux de clics sur une bannière publicitaire (exemple donné à titre illustratif sur un échantillon non représentatif de l'activité réelle du client).

À partir des mêmes données, les deux arbres apportent des informations différentes :

- l'arbre de gauche est un arbre de classification. Il indique si le taux de clics est supérieur ou non à une valeur seuil, pour une règle donnée. Par exemple, le chemin le plus à gauche indique que pour les publicités de taille 728×90 et pour les affichages de type BELOW ou UNKNOWN, ce taux n'est supérieur à la valeur seuil que dans moins de 20 % des cas. A contrario, il est supérieur au seuil dans plus de 80 % des cas pour les tailles 300×250 ;
- l'arbre de droite est un arbre de régression. Chaque feuille utilise une boîte à moustache pour représenter la distribution empirique du taux de clics pour une règle donnée.

Attention, un arbre de décision peut très vite mener à du sur-apprentissage : il peut décrire parfaitement un jeu de données, avec un individu par feuille dans le cas extrême. Dans ce cas, les règles ne sont absolument pas extrapolables et cette situation doit être évitée. Pour cela, il est nécessaire de s'arrêter à un nombre de feuilles adéquat lors de la réalisation de l'arbre : on parle d'« élaguer » l'arbre.

Construction d'un arbre de décision

Pour construire l'arbre attendu, trois principales questions doivent être résolues :

- Comment choisir la variable de division ? On doit en effet pouvoir définir l'arborescence de l'arbre en sélectionnant les variables, de la plus discriminante à la moins discriminante.
- Comment traiter les variables continues ? Pour définir des nœuds à partir d'une variable continue, l'on doit savoir « couper » la variable continue, pour que ses valeurs inférieures et supérieures à cette coupe puissent caractériser des nœuds distincts.
- Comment définir la taille de l'arbre ? L'objectif est de situer le niveau de nœuds optimal, pour trouver le juste équilibre entre sur-apprentissage et arbre trivial.

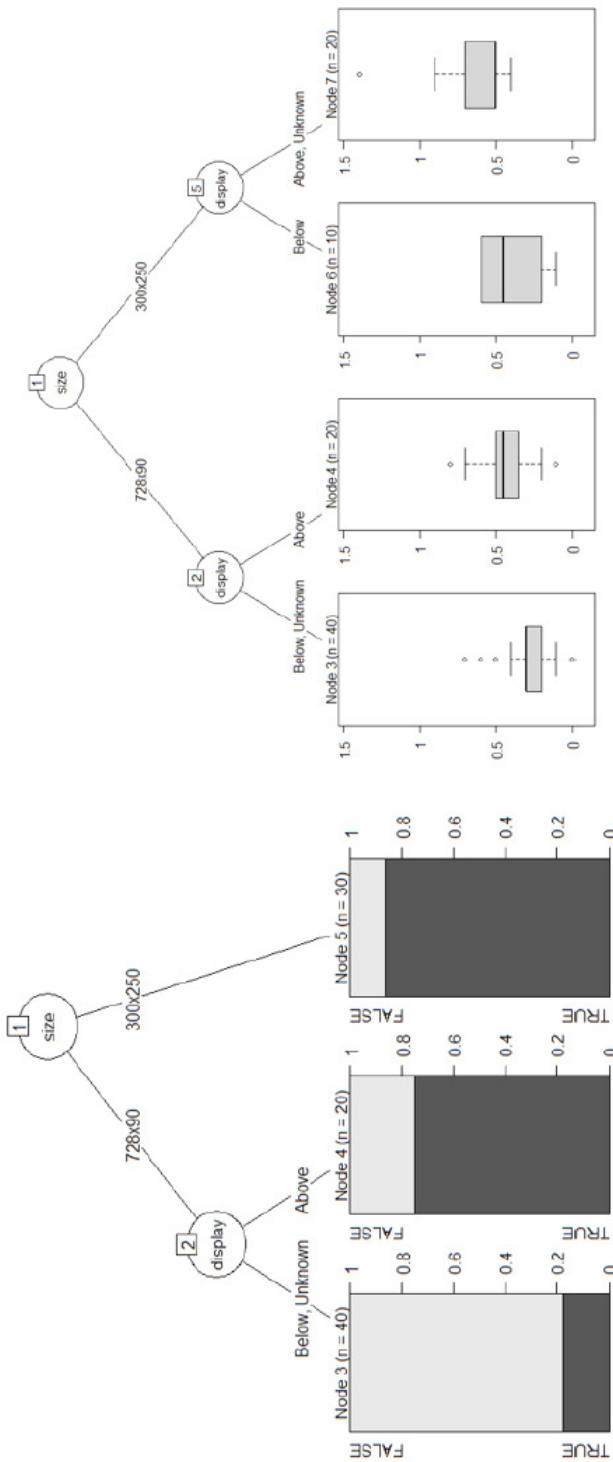


Figure 10-1 - Analyse des taux de clic sur une bannière publicitaire par arbres de décision

La réponse à ces questions dépend de l'algorithme utilisé pour constituer l'arbre.

De nombreux algorithmes existent. Par exemple, pour choisir la variable de décision et traiter les variables continues, l'algorithme CHAID, bien-aimé des statisticiens, utilise la mesure du χ^2 d'écart à l'indépendance et le t de Tschuprow. Les algorithmes CART et C4.5, quant à eux, se basent respectivement sur l'index (on parle aussi de critère de *split*) de Gini (indice de concentration) et sur la notion d'entropie (théorie de l'information). En ce qui concerne l'ajustement de la taille de l'arbre, on procède par post-élagage pour CART et C4.5 : on fait l'arbre le plus pur avec toute la segmentation, puis on utilise un critère pour comparer des arbres de tailles différentes. Pour CHAID, on procède par pré-élagage : on fixe une règle d'arrêt pour stopper la construction. Ces indications sont données à titre informatif : ces trois algorithmes sont les plus connus, mais il en existe bien d'autres et, pour un algorithme donné, on peut rencontrer diverses petites variantes.

Nous n'en dirons pas plus ici, car cette introduction aux arbres de décision n'est qu'une mise en bouche pour parler des excitantes méthodes ensemblistes, très utilisées aujourd'hui. Ces méthodes mettent en concurrence plusieurs arbres lors du classement d'un nouvel individu. Elles permettent donc d'améliorer drastiquement les performances des arbres et de les rendre moins sensibles aux idiosyncrasies des données. Mais en contrepartie, elles donnent lieu à des règles moins lisibles d'un point de vue métier.

À RETENIR Arbres de décision

Apprentissage supervisé – régression ou classification

Un arbre de décision permet de construire des règles à partir des données qui permettent de les ordonner.

Dans la pratique, on les utilise de moins en moins du fait de leur très forte propension à l'*overfitting*. Ils sont plutôt utilisés en tant que classificateurs faibles à la base de méthodes ensemblistes.

Références

On trouvera une bonne vue d'ensemble des arbres de classification, notamment par une comparaison des trois principaux algorithmes CART, C4.5 et CHAID, dans :

- Tufféry S. 2011. *Data mining and statistics for decision making*. Wiley.

Pour une analyse détaillée des différences entre les algorithmes CART et C4.5., voir :

- Kohavi R., Quinlan JR. 2002. Data mining tasks and methods: classification: decision-tree discovery. in *Handbook of data mining and knowledge discovery*, Oxford University Press, p. 267-276.

Enfin, pour le lecteur qui souhaite prendre du recul par rapport aux arbres de décision, la lecture de la thèse de Ricco Rakotomalala est recommandée. Elle date déjà de 1997, mais elle donne une vue unifiée des différentes variantes de construction d'un arbre de décision :

- Rakotomalala R. 1997. *Graphes d'induction*. Thèse présentée à l'Université Claude Bernard – Lyon 1 pour l'obtention du diplôme de Doctorat.