Introduction au Machine Learning

Application sous Python

Duvérier DJIFACK ZEBAZE

Introduction au Machine Learning

Application sous Python

Duvérier DJIFACK ZEBAZE

Table des matières

1

Initiation au Machine Learning

Sommaire

1.1 Qu'est - ce que le machine learning ?

On l'utilise tous, des centaines de fois par jour, sans même nous en rendre compte. Chaque fois que nous faisons une recherche dans google, une des raisons pour laquelle nous tombons sur des résultats plutôt pertinents c'est parce que derrière, c'est un algorithme de machine learning qui a appris comment trouver les résultats les plus pertinents parmi des milliards de résultats possibles. Quand nous postons une photo de nous même sur Facebook, et bien c'est un algorithme de machine learning qui a appris à reconnaître des visages sur des photos et qui peut ainsi nous identifier. Dernier exemple, quand nous allons sur Youtube, Netflix ou Amazon, il y a un algorithme de machine learning qui a appris à nous connaître nous, personnellement, et qui peut ainsi nous recommander le contenu que nous sommes le plus susceptible d'acheter ou bien de regarder.

Faire tout ce genre de chose, ce serait complètement impossible avec la programmation classique parce qu'il faudrait coder des milliards de cas possibles. Mais avec le machine learning, on donne à un algorithme la capacité d'apprendre et tout devient beaucoup plus facile : c'est ça le **machine learning**. Selon **Arthur Samuel**, le machine learning, *C'est donner à une machine la* ***capacité d'apprendre*** *sans la programmer de façon explicite.*

Le machine learning, ça concerne tout le monde aujourd'hui. C'est en train de révolutionner le monde entier. C'est en train de révolutionner **l'industrie des transports**, évidemment avec les voitures autonomes, **l'industrie aérospatiale, l'industrie biomécanique**, mais également le monde de **la santé** parce qu'aujourd'hui, le machine learning est capable de diagnostiquer des millions de cancer tous les ans. Le monde de **la finance, le marketing, le business, l'éducation, la sécurité, la justice**, même **l'industrie agricole** est touchée par le machine learning. Sans compter les objets connectés : **Alexa, ok google** ou bien **la vision par ordinateur**.

1.1.1 Comment apprendre ?

Le plus souvent, nous, les êtres humains, nous apprenons à partir **d'exemples**. Imaginons que l'on commence à apprendre la langue anglaise, on va sûrement acheter un bouquin dans lequel on pourra trouver des exemples de traduction français - anglais. ou bien payer un professeur particulier qui pourra superviser notre apprentissage en nous fournissant des exemples de traduction français - anglais que nous devrons mémoriser.

En machine learning, la technique d'apprentissage la plus courante s'inspire directement de ce mode de fonctionnement : c'est ce qu'on appelle **l'apprentissage supervisé.** Dans l'apprentissage supervisé, c'est nous qui jouons le rôle du *professeur*. Nous donnons à la machine des exemples qu'elle doit étudier pour en créer ce qu'on appelle **un modèle**. Ces exemples en général, on les regroupe dans ce qu'on appelle un **dataset**. c'est un tableau de données.

Imaginons qu'on ait un tableau de données dans lequel on a une colonne x et une colonne y (*cf.* Tableau 1.1).

Table 1.1 – Tableau de données

n°	x	y
1		
2		
⋮	⋮	⋮
i		
⋮	⋮	⋮
n		

et on veut que la machine apprenne la relation qui relie x à y.

$$y \approx f(x) \tag{1.1}$$

Notre tableau de données nous donnera peut être le nuage de points suivant :

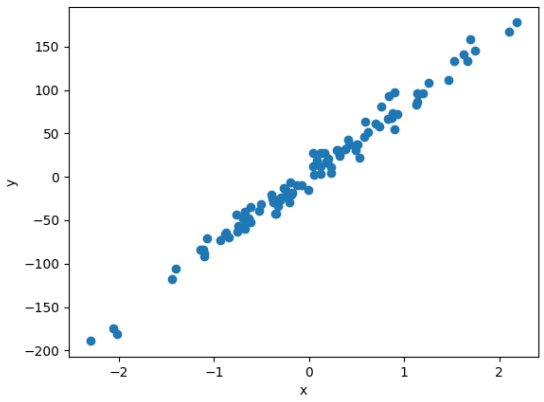


Figure 1.1 – Nuage de points

...à partir duquel la machine pourra apprendre un **modèle linéaire** (*cf.* Figure 1.2) :

...ou un **modèle polynomiale** (*cf.* Figure 1.3) :

En supposant que nous sommes un agent immobilier et que nous cherchons à développer un programme pour prédire le prix d'un appartement selon sa surface habitable, auquel cas on aura :

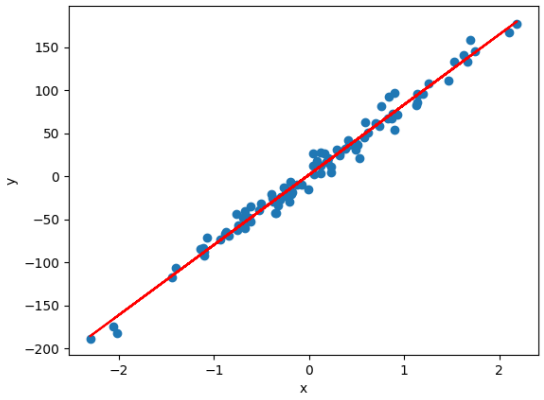


Figure 1.2 – Régression linéaire

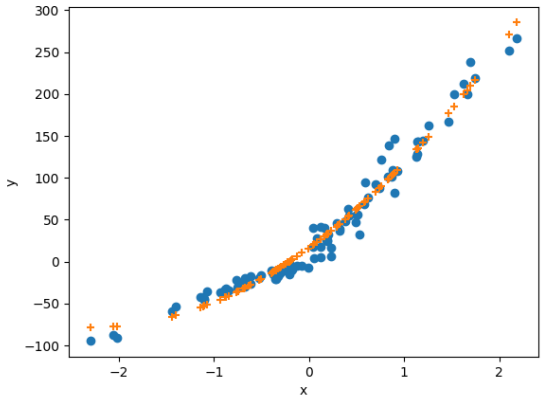


Figure 1.3 – Régression polynomiale

— y qui correspond au prix de l'appartement (en unité monétaire) ;
— x qui correspond à la surface de l'appartement en mètre carré.

et on recoltera ainsi des données sur internet par exemple ou chez des concurents. En tant qu'agent immobilier, nous pouvons ainsi nous servir de ce modèle que l'intelligence artificielle aura crée pour nous afin de vendre nos appartements à un meilleur prix que celui que nous avons acheté. Ce genre de problème est ce qu'on appelle un problème de régression.

Dans l'apprentissage supervisé, on rencontre deux types de problèmes : d'un côté, on a les problèmes de **régression** dans lesquels on cherche à prédire la valeur d'une variable continue (*cf.* Figure 1.4), c'est-à-dire une variable qui peut prendre une infinité de valeur, exemple, le prix d'un appartement

et d'un autre côté, on a les problèmes de **classification** dans lesquels on cherche à prédire la valeur d'une variable discrète (*cf.* Figure 1.5), c'est à dire une variable qui peut prendre certaines valeurs. Par exemple, une variable catégorielle telle que le type de fleur dans le jeu de données fleurs d'iris.

Dans les faits, le machine learning consiste à développer un **modèle mathématique** à partir de **données expérimentales**. Pour cela, il existe trois techniques :

— **L'apprentissage supervisé** (Supervised learning)
— **L'apprentissage non supervisé** (Unsupervised learning)

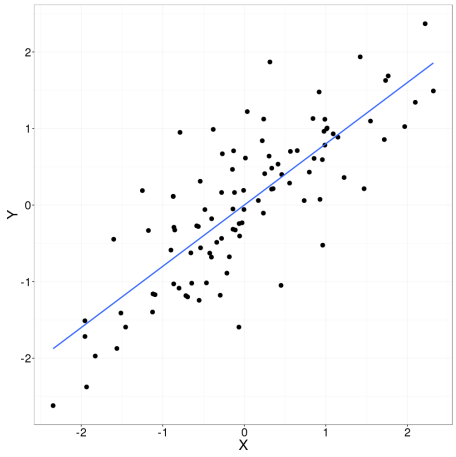


Figure 1.4 – Problème de régression

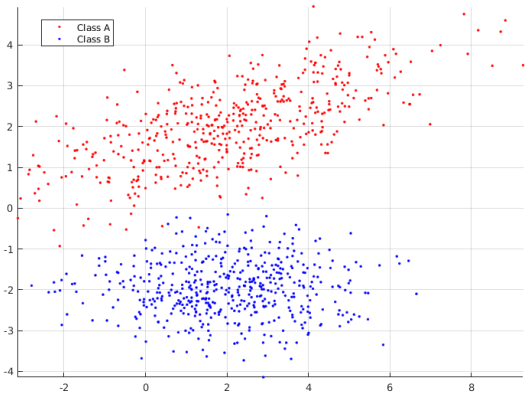


Figure 1.5 – Problème de classification

— **L'apprentissage par renforcement** (Reinforcement Learning)

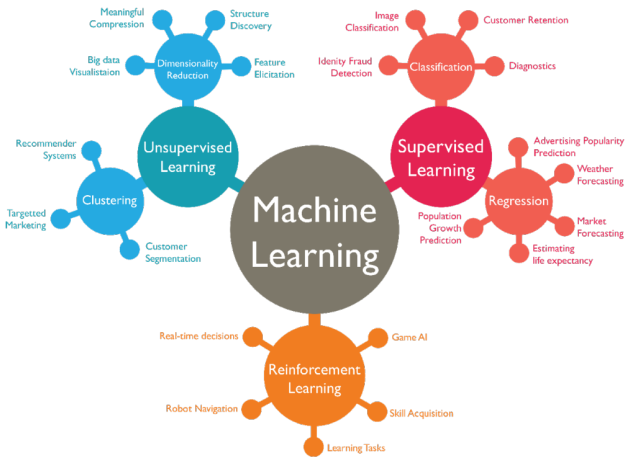


Figure 1.6 – Algorithme de Machine Learning

1.1.2 Apprentissage supervisé : les 4 notions fondamentales

1.1.2.1 Dataset

Les exemples qu'on montre à la machine, on les met dans un **dataset**. En apprentissage supervisé, le dataset contient toujours deux types de variables :

— **Les targets** : c'est l'objectif c'est-à-dire ce qu'on veut que la machine apprenne à prédire. Par exemple le prix d'un appartement, identifier si un email est spam ou non ;
— **Les features** : Les facteurs. Ceux qui viennent influencer la valeur du target

On dit que target est une fonction des features. Par convention, on note :

— **m** : le nombre d'exemple qu'on a dans le dataset correspondant au nombre de lignes.
— **n** : le nombre de features qu'on a dans le dataset correspondant au nombre de colonnes hormis la colonne y.

Pour désigner un cellule dans le dataset, on note le numéro de l'exemple au dessus de x et celui de la feature en-dessous : $x_{feature}^{(exemples)}$. Par exemple $ 93 000 = x_{3}^{(2)}$, c'est-à-dire le 3ème feature du 2ème exemple. Au final, on a un dataset(x,y) suivant :

Table 1.2 – Exemple de dataset

$y^{(1)}$	$x_1^{(1)}$	...	$x_n^{(1)}$
$\vdots$	$\vdots$	$x_j^{(i)}$	$\vdots$
$y^{(m)}$	$x_1^{(m)}$	...	$x_n^{(m)}$

Par traduction vectoriel, on a :

— le vecteur target $y \in IR^{m\times 1}$: $y = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{pmatrix}$

— la matrice features $X \in IR^{m\times n}$: $X = \begin{pmatrix} x_1^{(1)} & \cdots & x_n^{(1)} \\ \vdots & x_j^{(i)} & \cdots \\ x_1^{(m)} & \vdots & x_n^{(m)} \end{pmatrix}$

1.1.2.2 Le modèle et ses paramètres

A partir du dataset, on peut visualiser un nuage de points. On peut se dire qu'on peut créer un modèle, par exemple un modèle linéaire auquel cas, on aura un modèle

$$f(x) = ax + b$$

.

Mais aussi, on pourrait se dire qu'un modèle polynomiale est préférable :

$$f(x) = ax^2 + bx + c$$

ou

$$f(x) = ax^3 + bx^2 + cx + d$$

.

On a un **modèle** et ce modèle contient des **paramètres** (ce sont **les coefficients** du polynôme).

1.1.2.3 La fonction coût

Un modèle, lorsqu'on l'utilise par rapport à notre dataset, il nous donne des erreurs. Ces erreurs, on peut les trouver sur l'ensemble de nos points et lorsqu'on les assemble, on a **la fonction coût**. Et donc avoir un bon modèle, c'est avoir celui qui *minimise les erreurs*.

1.1.2.4 Algorithme d'apprentissage ou de minimisation

En marchine learning, on développe une stratégie qui cherche à trouver *quels sont les paramètres (a, b, c, etc...) qui minimisent la fonction coût* c'est-à-dire qui minimisent l'ensemble des erreurs. Pour minimiser cette fonction, on utilise un algorithme d'apprentissage et il en existe beaucoup. Un des plus connus, c'est **l'algorithme de la descente de gradient** ou **Gradient Descent** (en anglais). Il en existe d'autres. On peut par exemple citer : l'algorithme des k plus proches voisins, le classifieur bayésien naif, l'analyse discriminante, etc...

1.2 Machine learning avec sklearn

Pour créer un modèle, on génère un objet de la classe correspondant à ce modèle. Dans sklearn, c'est ce qu'on appelle un **estimateur**. On peut aussi préciser entre parenthèses les **hyperparamètres** de notre modèle. Par exemple dans **le gradient descent**, on peut insérer *le learning rate*, ou pour un **randomForest**, on peut préciser le nombre d'arbre. Dans le cas d'un modèle linéaire, on a la formulation suivante : **model = LinearRegression()**

Une fois qu'on a initialisé notre modèle, on va pouvoir l'entraîner, l'évaluer et l'utiliser en employant des méthodes qu'on retrouve dans toutes les classes de scikit-learn. Même si tous ces modèles ont des mécanismes différents, leur interface d'utilisation est toujours la même. Cela signifie que pour développer une régression linéaire ou pour développer un arbre de décision, on va écrire quasiment le même code.

Pour entraîner notre modèle, on utilise la méthode **.fit()** dans laquelle on fait passer les données X et y qui doivent être présentées dans 2 tableaux numpy bien distints : **model.fit(X,y)**.

Au passage, ces tableaux doivent toujours avoir 2 dimensions :

- La première dimension pour le nombre d'échantillon présent dans le dataset : n_samples
- La deuxième dimension pour le nombre de features qu'on a dans le dataset : n_features. Pour y, ce ne sera pas le nombre de features, mais le nombre de target, c'est-à-dire 1.

Une fois le modèle entraîné, on peut l'évaluer avec la méthode **.score()** : **model.score(X,y)**. Dans cette méthode, On fait passer une nouvelle fois les données X et y. Cette fois ci, la machine utilise les données X pour faire une prédiction et comparer ces prédictions aux valeurs y qui sont données dans la méthode.

Une fois satisfait par la performance du modèle, on peut l'utiliser pour faire une nouvelle prédiction en utilisant la méthode **.predict()** : **model.predict(X)**. Donc globalement, dans le cadre du modèle linéaire, on doit :

1. Sélectionner un **estimateur** et préciser ses **hyperparamètres** : model = **LinearRegression(.....)**
2. Entraîner le modele sur les données X, y (divisées en deux tableaux numpy) : model.**fit(X,y)**
3. Evaluer le modèle : model.**score(X,y)**
4. Utiliser le modèle : model.**predict(X)**

2 Régression Linéaire

Sommaire

Sous python, plusieurs librairies permettent de mettre en oeuvre un modèle de régression linéaire (sklearn, statsmodels, scipy, etc...). Dans ces librairies, l'algorithme de régression linéaire est déjà mis en oeuvre. Certaines de ces librairies utilisent la méthode du Gradient Descent (SGDRegressor) tandis que d'autres, celle des équations normales (LinearRegression). Nous verrons premièrement comment faire de la régression linéaire basée sur la méthode du Gradient descent en utilisant les librairies numpy, pandas et matplotlib. Ensuite, nous verrons comment cette méthode est implémentée dans sklearn. Enfin, nous effectuerons une régression linéaire basée sur les équations normales.

2.1 La méthode du Gradient Descent

2.1.1 Modèle de régression linéaire simple : Rappel

La régression linéaire simple est un algorithme prédictif supervisé. Il prend en entrée une variable prédictive x et va essayer de trouver une fonction de prédiction $f(x)$. Cette fonction sera une droite qui s'approchera le plus possible des données d'apprentissage. La fonction de prédiction étant une droite, elle s'écrira mathématique sous la forme :

$$f(x) = \beta_0 + \beta_1 x \tag{2.1}$$

avec β_0 et β_1 les coefficients de la droite.

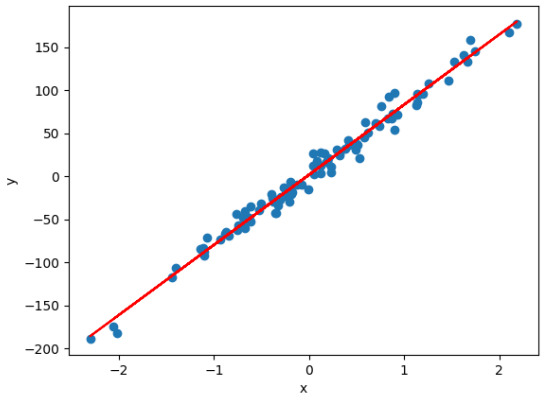


Figure 2.1 – Régression linéaire

La droite en rouge (*cf.* Figure 2.1) représente la meilleure approximation par rapport au nuage de points. Cette approximation est rendue possible parce qu'on a pu calculer les paramètres prédictifs β_0 et β_1 qui définissent notre droite.

La question qui se pose est : *Comment calcule-t-on les valeurs de* β_0 *et* β_1 ?

2.1.1.1 La fonction de coût d'erreur (Cost Function)

La figure 2.1 montre que la droite tente d'approcher le plus de points possibles (en réduisant l'écart avec ces derniers). En d'autres termes, elles minimise au maximum l'erreur globale.

Pour la régression linéaire simple, le but est de trouver un couple (β_0, β_1) optimal tel que $f(x)$ soit le plus proche possible de y (la valeur qu'on essaye de prédire), et ce, pour tous les couples (x, y) qui forment notre ensemble de données d'apprentissage. Ainsi, trouver le meilleur couple (β_0, β_1) revient à minimiser le coût global des erreurs unitaires qui se définit comme suit :

$$\sum_{i=1}^{i=m} \left(f(x_i) - y_i\right)^2 \tag{2.2}$$

La fonction de coût est définie comme suit :

$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^{i=m} \left(f(x_i) - y_i\right)^2 \tag{2.3}$$

En remplaçant le terme $f(x)$ par sa valeur, on obtient :

$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^{i=m} \left(\beta_0 + \beta_1 x_i - y_i\right)^2 \tag{2.4}$$

Cette formule représente la fonction de coût (*cost function/Error function*) pour la régression linéaire simple.

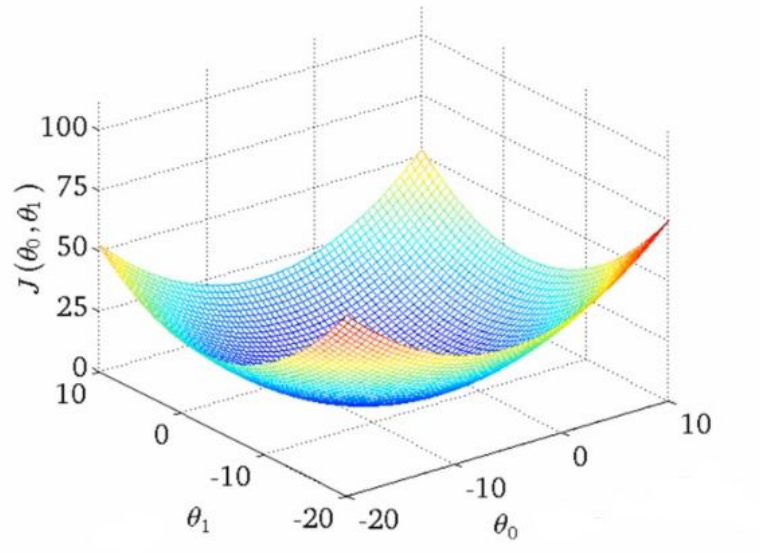


Figure 2.2 – Gradient Descent visualisation

Visuellement, on remarque que la fonction $J(\beta_0, \beta_1)$ a la forme d'un bol (*cf.* Figure 2.2). Mathématiquement, on dit la fonction est convexe. La convexité d'une fonction implique que cette dernière possède un seul minimum global. Les valeurs de β_0 et β_1 qui sont au minimum global de $J(\beta_0, \beta_1)$ seront les meilleures valeurs pour notre hypothèses $f(x)$.

2.1.1.2 Minimisation de la fonction de coût $J(\beta_0, \beta_1)$

Une façon de calculer le minimum de la fonction de coût $J(\beta_0, \beta_1)$ est d'utiliser l'algorithme de descente du gradient (*Gradient Descent*). Ce dernier est un algorithme itératif qui va changer, à chaque itération, les valeurs de β_0 et β_1 jusqu'à trouver le meilleur couple possible. L'algorithme se décrit comme suit :

Gradient Descent

— Initialiser aléatoirement les valeurs de β_0 et β_1
— répéter jusqu'à convergence au minimum global de la fonction de coût

$$\beta_j^{\text{new}} = \beta_j^{\text{old}} - \alpha \frac{\partial}{\partial \beta_j} J(\beta_0, \beta_1), \quad \text{pour} \quad j \in \mathbb{N} \quad \forall j \in \{0, 1\} \tag{2.5}$$

— retourner $\hat{\beta}_0$ et $\hat{\beta}_1$

L'algorithme peut sembler compliqué à comprendre, mais l'intuition derrière est assez simple. Imaginez que vous soyez dans une colline, et que vous regardez autour de vous pour trouver la meilleure pente pour avancer vers le bas. Une fois la pente trouvée, vous avancez d'un pas d'une grandeur α.

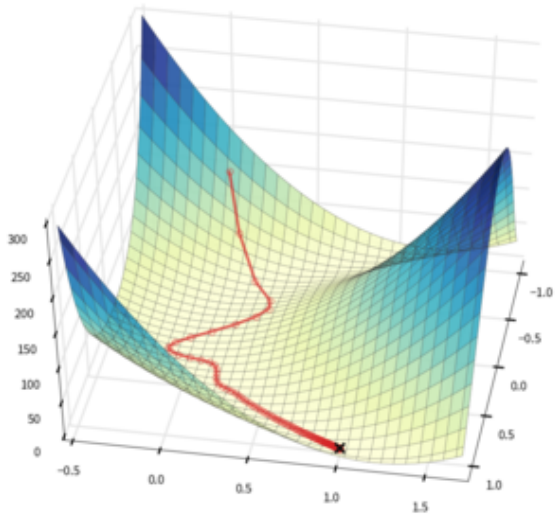


Figure 2.3 – Gradient Descent algorithm

Dans la définition de l'algorithme, on remarque ces deux termes :

— Le terme α s'appelle le *Learning Rate* : il fixe la « grandeur » du pas de chaque itération du Gradient Descent.
— $\frac{\partial}{\partial \beta_j} J(\beta_0, \beta_1)$: Ce terme est la **dérivée partielle** pour chacun des termes β_0 et β_1.

Pour les matheux, vous pouvez calculer les dérivées partielles de β_0 et β_1. Sinon, les voici :

— $\frac{\partial}{\partial \beta_0} J(\beta_0, \beta_1) = \frac{1}{m} \sum_{i=1}^{m} (f(x_i) - y_i)$
— $\frac{\partial}{\partial \beta_0} J(\beta_0, \beta_1) = \frac{1}{m} \sum_{i=1}^{m} (f(x_i) - y_i)\, x_i$

A chaque itération, l'algorithme avancera d'un pas et trouvera un nouveau couple β_0 et β_1. Et à chaque itération, le coût d'erreur global se réduira

2.1.2 Implémentation du Gradient Descent

Pour la mise en oeuvre, nous avons besoin de :

— **Dataset :** (x, y) avec m exemples (individus), n variables
— **Modèle :** $F = X\Theta$. Θ est le vecteur des paramètres a et b.
— **Fonction coût :** $J(\Theta) = \frac{1}{2m} \sum (X\Theta - y)^2$
— **Gradient :** $\frac{\partial f(\Theta)}{\partial \Theta} = \frac{1}{m} X^T (X\Theta - y)$
— **Gradient Descent :** $\Theta = \Theta - \alpha \frac{\partial}{\partial \Theta} f(\Theta)$ où α est le learning rate.

2.1.2.1 Mise en oeuvre sous Python

On aura besoin des librairies numpy pour la manipulation des matrices, matplotlib pour les représentation graphique et sklearn pour notre jeu de données. Tout d'abord,

on va commencer par importer notre dataset en générant un tableau de données (x, y) aléatoires. On va fixer le nombre de features à 1 grâce au paramètre n_features et contrôler l'aléa.

```
# Chargement du jeu de données (dataset)
import numpy as np
from sklearn.datasets import make_regression
np.random.seed(123)
x, y  = make_regression(n_samples=100,n_features=1,noise=10)
```

On peut à présent visualiser notre dataset créer.

```
# Visualisation du nuage de points
import matplotlib.pyplot as plt
fig, axe = plt.subplots(figsize=(16,6))
axe.scatter(x,y,color="black");
axe.set(xlabel="x",ylabel="y");
plt.show()
```

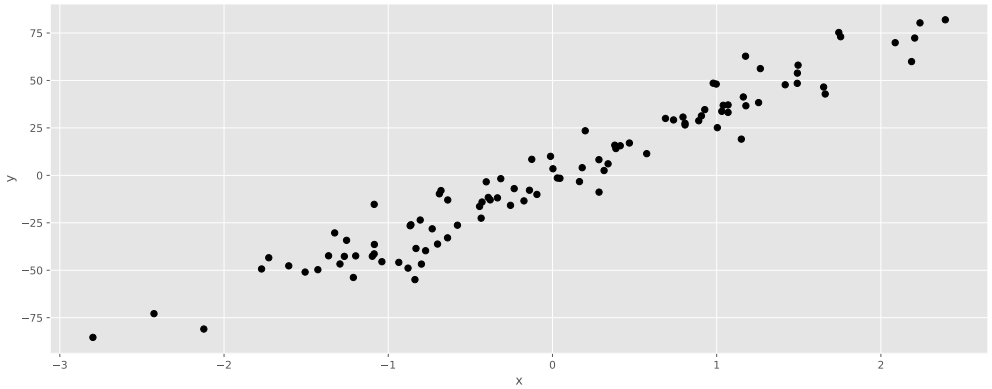


Figure 2.4 – Nuage de points

Avant de passer à la suite , il faut tout d'abord vérifier les dimensions de nos différentes matrices x et y.

```
# Vérification des dimensions de nos éléments
print(x.shape)
```

```
## (100, 1)
```

```
print(y.shape)
```

```
## (100,)
```

On se rend compte qu'il y a un petit souci avec les dimensions du vecteur y. Dans x, on a 100 lignes et 1 colonne, pourtant dans y, on 100 lignes et après rien n'est

mentionné. Ceci c'est pas un beug. En fait, à chaque fois qu'on utilise make_regression, les dimensions de y sont incomplètes. Pour réécrire les dimensions du vecteur y, on va utiliser la fonctionnalité reshape présente dans numpy.

```
# Réécriture des dimensions de y
import numpy as np
y = y.reshape(y.shape[0],1) # y.shape[0] correspond au nombre
print(y.shape)              # de ligne de y déjà présent
```

```
## (100, 1)
```

On doit avoir une matrice X qui contient une colonne remplie de 1.

```
# Création de la matrice X
X = np.hstack((x, np.ones(x.shape)))
X[:5,:]
```

```
## array([[-0.09470897,  1.        ],
##        [-1.25388067,  1.        ],
##        [ 0.00284592,  1.        ],
##        [ 1.03972709,  1.        ],
##        [-0.43435128,  1.        ]])
```

2.1.2.2 Initialisation du vecteur Θ

Le vecteur Θ, on ne le connait pas. C'est le vecteur qui caractérise les propriétés de notre modèle.

```
# Initalisation du vecteur théta = (a,b)
theta = np.random.randn(2,1)
print(theta)
```

```
## [[0.02964293]
##  [2.95862545]]
```

2.1.2.3 Modèle

Pour notre modèle, qui est un modèle linaire, on doit écrire une fonction $F(X) = X\Theta$.

```
# Modèle linéaire
def model(X, theta):
  return X.dot(theta)
```

La fonction retourne le produit matriciel de X et Θ. On teste notre fonction `model` avec notre matrice X et notre vecteur Θ initialisé.

```
# Test de la fonction model
print(model(X, theta)[:5,:])
```

```
## [[2.955818  ]
##  [2.92145676]
##  [2.95870981]
##  [2.989446  ]
##  [2.94575   ]]
```

Affichons avec matplotlib les résultats de notre modèle par rapport à notre dataset X.

```
# Nuage de points et tendance linéaire
fig, axe = plt.subplots(figsize=(16,6))
axe.scatter(X[:,0],y,color="black")
axe.plot(X[:,0], model(X, theta), color = "red");
axe.set(xlabel="Axe x",ylabel="Axe y")
plt.show()
```

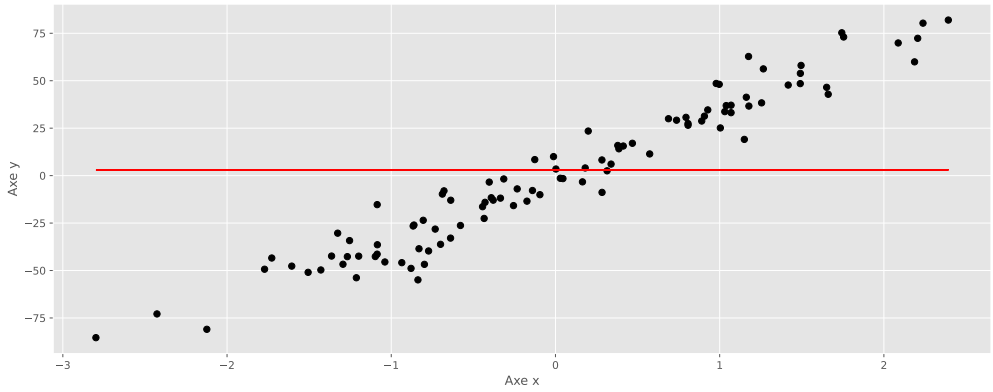


Figure 2.5 – Nuage de points avec tendance sur théta initialisé

Wow! On a un modèle qui n'entre pas tout à fait bien dans le nuage de points. Mais tout va rentrer dans l'ordre. En attendant, il faut qu'on calcule la fonction coût.

2.1.3 Fonction coût

La fonction coût dans le cas de notre exemple correspond à l'erreur quadratique moyen ou *mean squared error* (en anglais). On va créer une fonction cost_function.

2.1.3.1 Visualisation de la fonction de coût

Pour comprendre la fonction de coût $J(\Theta)$, nous allons tracer le coût sur une grille à 2 dimensions de θ_0 et θ_1 valeurs.

```
# Visualisation de la fonction coût
from sklearn.metrics import mean_squared_error as mse
def predict_y(thetas, xs):
    ys = thetas[0] + thetas[1]*xs
    return ys
```

On définit des valeurs pour θ_0 et θ_1

```
th0 = np.arange(-100, 100, 0.5)
th1 = np.arange(-1, 5, 0.5)
TH0, TH1 = np.meshgrid(th0, th1)
Js = np.array([mse(y,
        predict_y([th0,th1], x)) for th0, th1 in zip(np.ravel(TH0), np.ravel(TH1))])
Js = Js.reshape(TH0.shape)
```

On peut maintenant faire une représentation de la fonction coût dans un espace à 3-dimensions en fonction de θ_0 et θ_1.

```
from matplotlib import cm
fig, ax = plt.subplots(subplot_kw={"projection": "3d"},figsize=(16,6))
ax.plot_surface(TH0, TH1, Js,cmap=cm.coolwarm)
ax.set(xlabel=r"$\theta_0$",ylabel=r"$\theta_1$",
       zlabel=r"J($\theta_0$, $\theta_1$)");
plt.show()
```

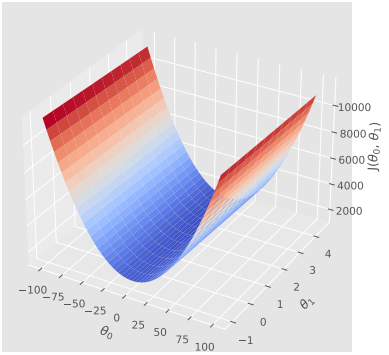


Figure 2.6 – Fonction coût

On peut également observer les valeurs de cette fonction dans un espace à 2-dimensions.

```
fig, axe = plt.subplots(figsize=(16,6))
CS = plt.contour(TH0, TH1, Js)
axe.clabel(CS, inline=1, fontsize=10);
axe.set(title='Simplest default with labels',xlabel=r"$\theta_0$",
        ylabel=r"$\theta_1$");
plt.show()
```

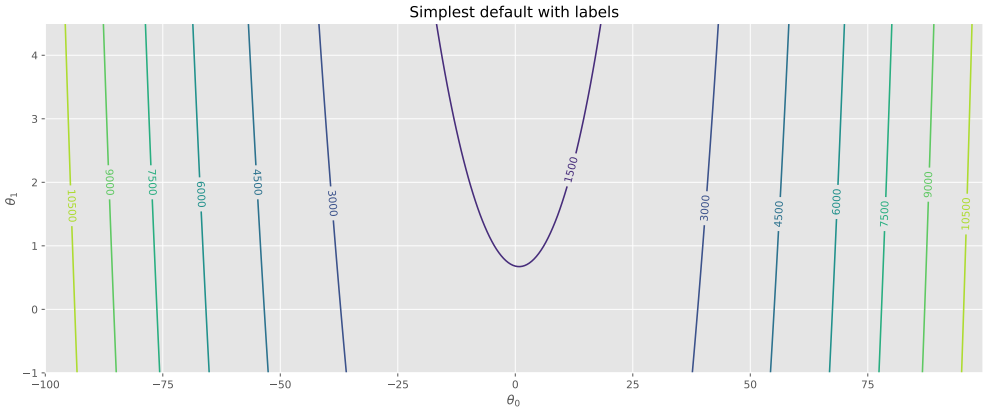


Figure 2.7 – Fonction coût

Définissons à présent la fonction coût.

```
### La fonction coût (= Erreur Quadratique Moyen)
def cost_function(X, y, theta):
  m = len(y)
  return 1/(2*m)*np.sum((model(X,theta)-y)**2)
```

On teste notre fonction coût sur le modèle qu'on a crée.

```
# test de la fonction coût
print(cost_function(X, y, theta))
```

```
## 779.7415024477377
```

Notre valeur de la fonction coût est censé être minimisé (tendre vers zéro). Mais ce n'est pas grave car Θ n'est pas le vecteur optimal.

2.1.3.2 Gradients et descente de gradient

Nous créeons notre fonction de dsecente de gradient.

```
# Gradient descent
def gradient_descent(X,y,theta,learning_rate,n_iterations):
  cost_history = np.zeros(n_iterations)
  theta_history = np.zeros((n_iterations,X.shape[1]))
  m = len(y)
  for i in range(n_iterations):
    y_pred = model(X,theta)
    residuals = y_pred - y
    gradient_vector = np.dot(X.T,residuals)
    theta -= (learning_rate/m)*gradient_vector
    cost_history[i] = np.sum((residuals**2))/(2*m)
    theta_history[i,:] = theta.T
  return theta,cost_history,theta_history
```

`learning_rate` et `n_iterations` correspondent respectivement l'hyperparamètre α et le nombre d'itération. La fonction `gradient_descent` renvoie le vecteur Θ optimal et un vecteur `cost_histoty` qui contient l'ensemble des différentes valeurs de la fonctions coût à chaque itération.

2.1.3.3 Entraînement du modèle

On a tous les éléments nécessaires pour produire notre vecteur Θ final qui sera celui qui minimise la fonction coût.

```
# Entraînement du modèle
alpha = 0.01
n_iter = 1000
theta_final,cost_history,theta_history=gradient_descent(X,y,theta,alpha,n_iter)
print(theta_final)
```

```
## [[33.965432  ]
##  [-0.08711021]]
```

Maintenant voyons si le paramètre Θ obtenu représente au mieux notre dataset. Pour cela, nous allons créer un vecteur `prédictions` qui va récupérer l'ensemble des valeurs de $X\Theta$.

```
# Valeurs ajustées
predictions = model(X, theta_final)
```

On peut maintenant visualiser le nuage de points avec la nouvelle droite tendance fournie par le vecteur Θ final ou optimal.

```
# Nuage de points et tendance linéaire optimale
fig, axe = plt.subplots(figsize=(16,6))
axe.scatter(X[:,0], y,color="black");
axe.plot(X[:,0], predictions, color = "red");
axe.set(xlabel="x",ylabel="y");
plt.show()
```

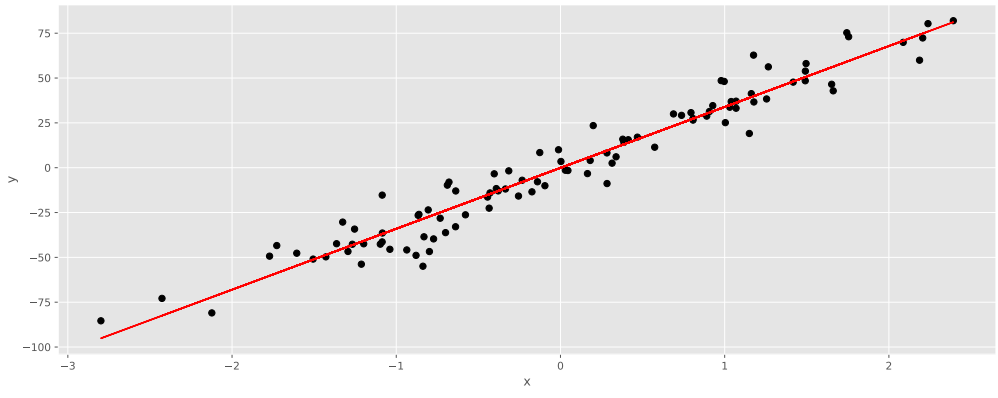


Figure 2.8 – Nuage de points avec tendance sur théta optimal

On voit effectivement que la droite de tendance obtenue est meilleure comparée à celle obtenue avec le paramètre Θ initial. Donc `theta_final` est celui qui minimise la fonction coût de notre dataset. Il est possible de jouer sur les hyperparamètres `learning_rate` ou `n_iterations` pour voir le résultat d'apprentissage voulu.

2.1.3.4 La courbe d'apprentissage

On appelle courbe d'apprentissage (*Learning curves*) les courbes qui montrent l'évolution de la Fonction Coût au fil des itérations de Gradient Descent. Si notre modèle apprend, alors sa Fonction Coût doit diminuer avec le temps.

```
# Courbe d'apprentissage
fig, axe = plt.subplots(figsize=(16,6))
axe.plot(range(n_iter), cost_history,color="black")
axe.set(xlabel="Iterations",ylabel=r"$J(\Theta)$")
plt.show()
```

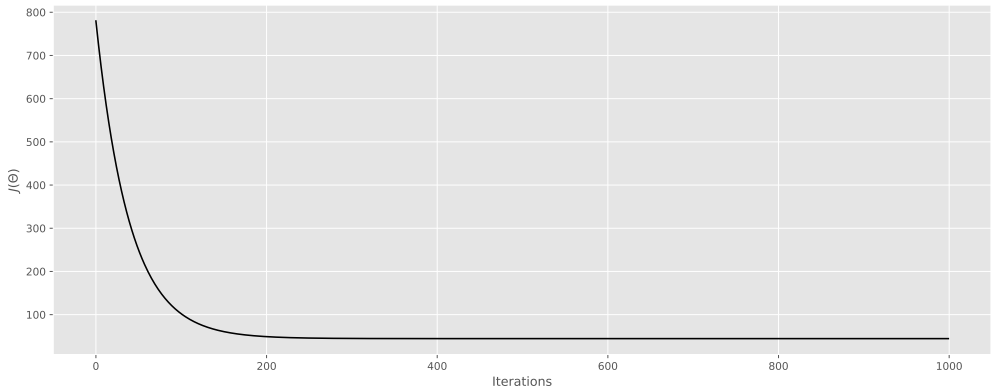


Figure 2.9 – Courbe d'apprentissage

En observant le graphique, on voit bien, à travers le temps (c'est-à-dire à travers les 1000 itérations de notre descente de gradient), comment est-ce que la machine a réussi à minimiser la fonction coût. Ce qui est intéressant avec cette courbe, c'est qu'on peut voir que, passé les 250-300 itérations, la courbe n'a plu vraiment diminué et donc on aurait pu arrêté l'algorithme de la descente du gradient aux alentours de l'itération numéro 400. Cela aurait permis à la machine d'économiser du temps et de l'énergie. A chaque itération, le modèle s'améliore pour donner la droite ci-dessous.

```
# Visualisation du modèle au cours de son apprentissage
fig,axe = plt.subplots(figsize=(16,6))
axe.scatter(x,y,color="black")
for i in range(n_iter):
    axe.plot(x, model(X, theta_history[i]), lw = 1)
plt.show()
```

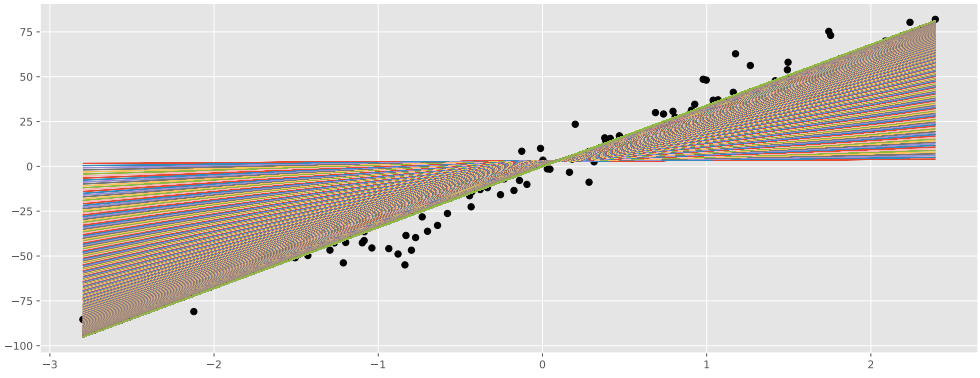


Figure 2.10 – Courbe d'apprentissage

2.1.3.5 Le coefficient de détermination (R^2)

C'est un indicateur qui permet de mesurer la qualité d'ajustement du modèle (ou de juger de la performance du modèle). Sa formule est :

$$R^2 = 1 - \frac{\sum (y - f(x))^2}{\sum (y - \overline{y})^2} \tag{2.6}$$

```
# Coefficient de déterminantion
def coef_determination(y, pred):
    u = ((y - pred)**2).sum() #SCR
    v = ((y - y.mean())**2).sum() # SCT
    return 1 - u/v
score = coef_determination(y, predictions)
print("Coeff R2 : %.2f" %(score))


## Coeff R2 : 0.94
```

On obtient un score de 94.29 % avec notre modèle.

Nous allons créer une class `LinearRegressionUsingGD` qui encapsule toutes les différentes fonctions créées :

```
# Régression Linéaire avec descente de gradient
class LinearRegressionUsingGD:
    """Linear Regression Using Gradient Descent.
    Parameters
    ----------
    eta : float
        Learning rate
    n_iterations : int
        No of passes over the training set
    Attributes
    ----------
    w_ : weights/ after fitting the model
```

```
    cost_ : total error of the model after each iteration
    """
    def __init__(self, eta=0.05, n_iterations=1000):
        self.eta = eta
        self.n_iterations = n_iterations
    def fit(self, x, y):
        """Fit the training data
        Parameters
        ----------
        x : array-like, shape = [n_samples, n_features]
            Training samples
        y : array-like, shape = [n_samples, n_target_values]
            Target values
        Returns
        -------
        self : object
        """
        self.cost_ = []
        self.w_ = np.zeros((x.shape[1], 1))
        m = x.shape[0]
        for _ in range(self.n_iterations):
            y_pred = np.dot(x, self.w_)
            residuals = y_pred - y
            gradient_vector = np.dot(x.T, residuals)
            self.w_ -= (self.eta / m) * gradient_vector
            cost = np.sum((residuals ** 2)) / (2 * m)
            self.cost_.append(cost)
        return self
    def predict(self, x):
        """ Predicts the value after the model has been trained.
        Parameters
        ----------
        x : array-like, shape = [n_samples, n_features]
            Test samples
        Returns
        -------
        Predicted value
        """
        return np.dot(x, self.w_)
```

2.1.4 Modèle de régression de type polynomiale

Cette fois-ci, on suppose que notre modèle à la forme suivante :

$$f(x) = ax^2 + bx + c \tag{2.7}$$

Les étapes de mise en oeuvre sous python sont similaires à celles présentées dans le cas d'un modèle de régression linéaire simple, mais à quelques différences près. Ici, on suppose que la variable x et son carré x^2 expliquent la variable y. Dans le cas d'espèce, la tendance n'est plus de type linéaire, mais a l'allure d'une courbe. On va travailler

avec le même dataset que précédent, mais à on va apporter une petite modification à la valeur de y afin d'avoir un nuage de points qui respecte l'allure d'une courbe.

2.1.4.1 Données

Modifions la forme de y et observons graphiquement notre nouveau jeu de données.

```
# Modification de y
y = y + abs(y/2)
# Nuage de points
fig, axe = plt.subplots(figsize=(16,6))
axe.scatter(x,y,color="black");
axe.set(xlabel="x",ylabel="y");
plt.show()
```

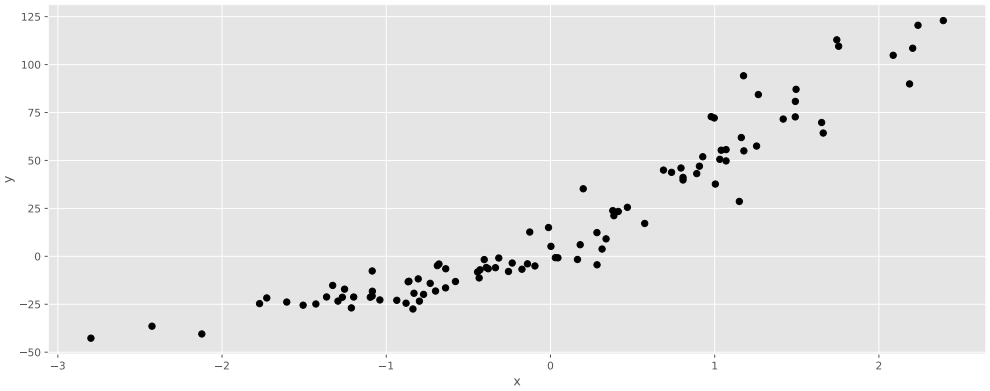


Figure 2.11 – Nuage de points

A partir du nuage de points, on voit qu'un ajustement linéaire de type $ax+b$ serait une erreur. Il faut donc une autre forme fonctionnelle afin de bien représenter l'ajustement. Nous allons redimensionner le vecteur y et créer notre matrice X.

```
# Redimension de y
y = y.reshape(y.shape[0],1)
print(y.shape)
```

```
## (100, 1)
```

```
# Matrice X
X = np.hstack((x, np.ones(x.shape)))
X = np.hstack((x**2, X))
print(X.shape)
```

```
## (100, 3)
```

2.1.4.2 Initialisation de la valeur de Θ

```
# Initialisation de Théta
np.random.seed(0)
theta = np.random.randn(3,1)
print(theta)
```

```
## [[1.76405235]
##  [0.40015721]
##  [0.97873798]]
```

On constate que le premier paramètre de *np.random.randn* est égale à 3. En effet, dans le modèle, nous avons trois paramètres à estimer : a, b et c.

```
# Visualisation
fig,axe = plt.subplots(figsize=(16,6))
axe.scatter(x, y, marker = "o",color="black");
axe.scatter(x, model(X, theta), marker = "+",color="red");
axe.set(xlabel="x",ylabel="y");
plt.show()
```

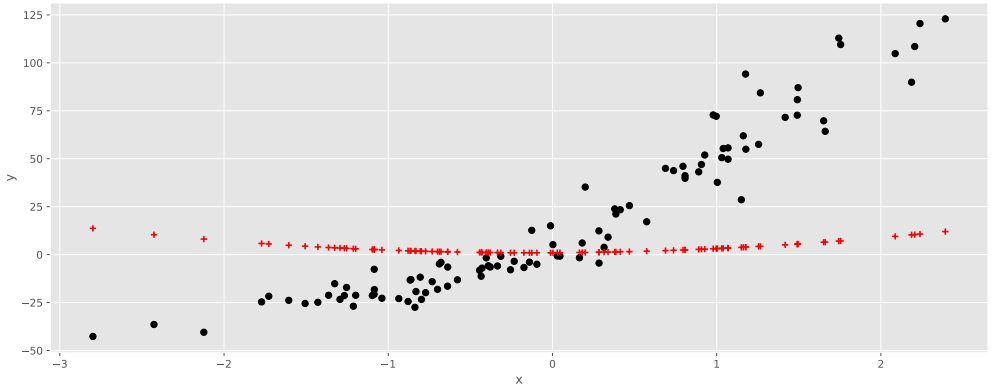


Figure 2.12 – Nuage de points

```
# Entraînement
theta_final,cost_history,theta_history = gradient_descent(X, y, theta,
                                                          learning_rate = alpha,
                                                        n_iterations = n_iter)
```

```
# Nuage de points avec ajustement final
predictions = model(X, theta_final)
fig, axe = plt.subplots(figsize=(16,6))
axe.scatter(x, y, marker = "o",color="black");
axe.scatter(x, predictions, marker = "+",color="red");
axe.set(xlabel="x",ylabel="y");
plt.show()
```

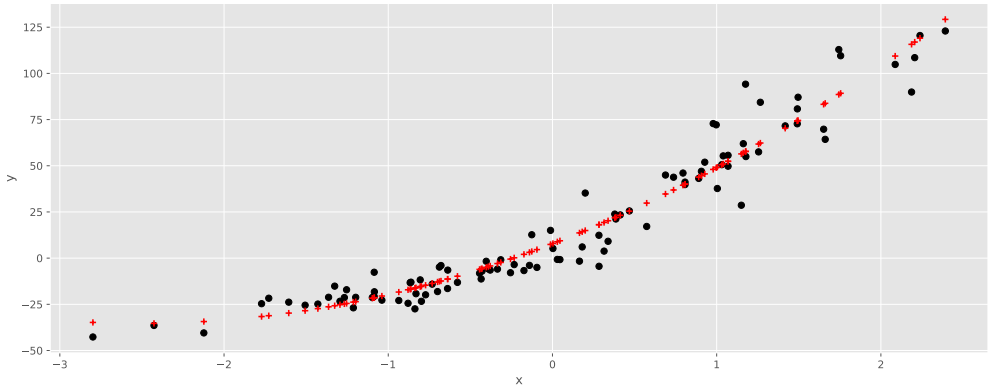


Figure 2.13 – Nuage de points

```
# Courbe d'apprentissage
fig, axe = plt.subplots(figsize=(16,6))
axe.plot(range(n_iter), cost_history);
axe.set(xlabel="Iterations",ylabel=r"$J(\Theta)$");
plt.show()
```

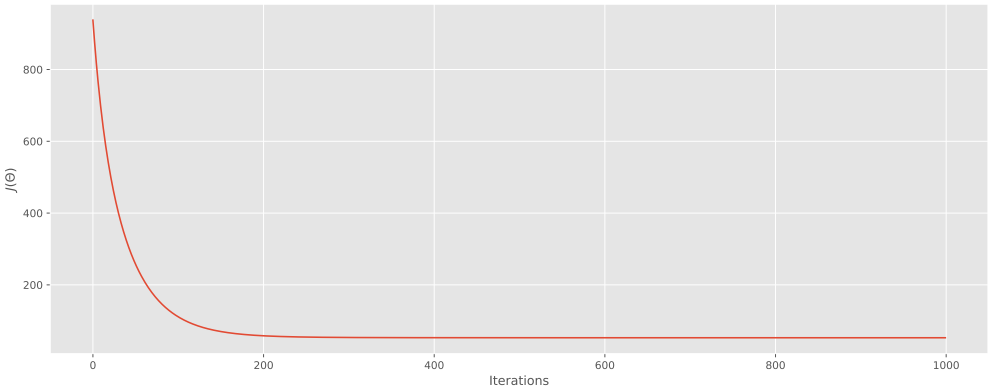


Figure 2.14 – Nuage de points

```
# Score
score = coef_determination(y, predictions)
print("Coeff R2 : %.2f" %(score))
```

```
## Coeff R2 : 0.94
```

On obtient un score de 94.04 %.

2.2 Régression linéaire avec sklearn

Dans cette section, nous allons utiliser deux générateurs de modèle linéaire : SGDRegressor qui est basé sur l'algorithme du Gradient Descent et LinearRegression basé sur les équations normales.

2.2.1 Gradient Descent sous sklearn

Dans la section précédente, on a vu que pour développer et entraîner un modèle, il a fallu beaucoup de mathématiques : entre la Fonction Coût, les dérivées, l'algorithme de Gradient Descend, etc... Dans **Sklearn**, tout cela est **déjà fait** à travers la méthode **SGDRegressor** (qui signifie Stochastic **Gradient Descent** Regressor) et qui contient le calcul de la **Fonction Coût**, dex **gradients**, de **l'algorithme** de **minimisation**, etc.... Il faut savoir que Sklearn est la bibliothèque par défaut de toutes les méthodes de machine learning en python.

2.2.1.1 Gradient Descent pour la régression linéaire

Nous allons réutiliser le dataset aléatoire crée au début du chapitre pour faire une application de cette méthode.

a) Modèle

On va à présent définir notre *model* depuis le générateur **SGDRegressor** en entrant le nombre d'itérations que le Gradient Descend doit effectuer ainsi que le Learning Rate. Et une fois le modèle défini, il faut **l'entraîner**. Pour cela, il suffit d'utiliser la fonction **fit**. Par exemple, entraînons notre modèle sur **100** itérations avec un Learning Rate de **0.0001**.

```
# Model SGDRegressor
from sklearn.linear_model import SGDRegressor
model = SGDRegressor(max_iter = 100, eta0 = 0.0001)
model.fit(x,y)
```

```
## SGDRegressor(eta0=0.0001, max_iter=100)
```

Nous pouvons maintenant observer la précision de notre modèle en utilisant la fonction **score** qui calcule le **coefficient de détermination** entre le modèle et les valeurs ***y*** de notre dataset.

```
# Score
score = model.score(x,y)
print("Coeff R2 : %.2f"%(score))
```

```
## Coeff R2 : 0.13
```

On peut aussi utiliser notre modèle pour faire de nouvelles prédictions avec la fonction **predict** et tracer ces résultats avec matplotlib.

```
# Prédiction
pred = model.predict(x)
# Visualisation
fig, axe =plt.subplots(figsize=(16,6))
axe.scatter(x,y,color="black");
```

```
axe.plot(x, pred, c= "red", lw = 3);
axe.set(xlabel="x",ylabel="y");
plt.show()
```

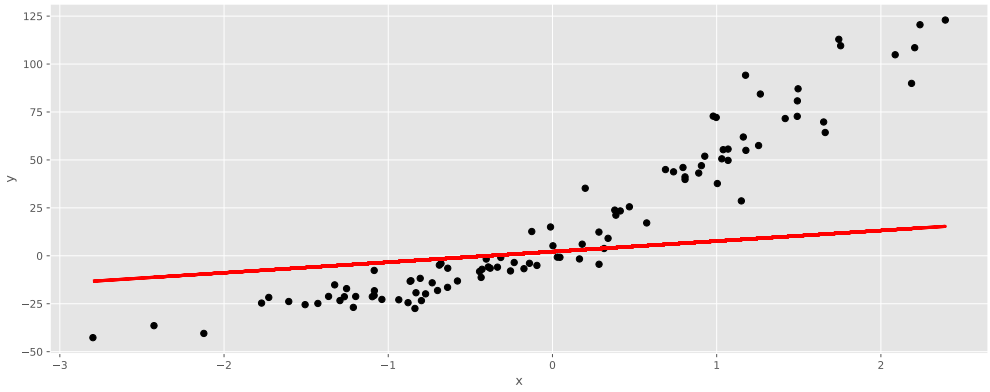


Figure 2.15 – Nuage de points

WOw ! Notre modèle semble vraiment mauvais (score = 13 %). C'est parce que nous ne l'avons pas entraîné suffisamment **longtemps** et parce que le *Learning rate* est top **faible**. Aucun problème, il est possible de le ré-entraîner avec de meilleurs hyper-paramètres.

b) Amélioration des performances

En Machine Learning, les valeurs **qui fonctionnent bien** pour la plupart des entraînements sont :

- Nombre d'itérations = **1000**
- Learning rate = **0.001**

```
# Nouveau modèle
model = SGDRegressor(max_iter = 1000, eta0 = 0.001)
model.fit(x,y)
```

```
## SGDRegressor(eta0=0.001)
```

```
# Score
score = model.score(x,y)
print("Coeff R2 : %.2f"%(score))
```

```
## Coeff R2 : 0.88
```

Le nouveau modèle fonctionne vraiment bien avec un coefficient de détermination de 88 %.

```
# Prédiction
pred = model.predict(x)
# Visualisation
fig, axe = plt.subplots(figsize=(16,6))
axe.scatter(x,y,color="black");
axe.plot(x, pred, c= "red", lw = 3);
axe.set(xlabel="x",ylabel="y");
plt.show()
```

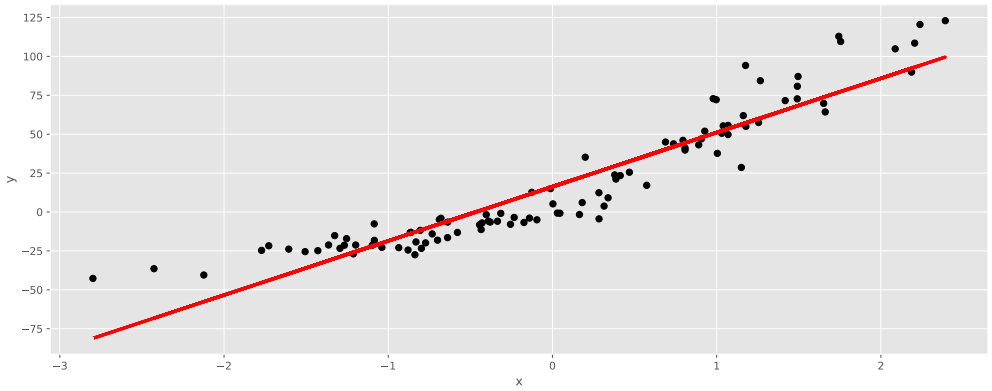


Figure 2.16 – Nuage de points

Le graphique est meilleur que le précédent et la droite d'ajustement passe par le maximum de point.

2.2.1.2 Régression polynomiale à plusieurs variables

Le code que nous avons écrit pour la régression linéaire peut être utilisé pour des problèmes bien plus complexes. Il suffit de générer des **variables polynomiales** dans notre Dataset en utilisant la fonction PolynomialFeatures présente dans Sklearn. Grâce au **calcul matriciel** (présent dans Numpy et Sklearn), la machine peut intégrer ces nouvelles variables polynomiales **sans changer** son calcul.

a) Modification du dataset

Nous allons modifier la forme de y afin qu'elle ne varie plus linéairement selon x.

```
# Modification de y
y = y**2
```

On ajoute à présent une variable polynomiale de **degré 2** dans le dataset pour forcer la machine à développer un modèle qui épousera l'allure **parabolique** de y en fonction x. Grâce à PolynomialFeatures, on va créer une variables polynomiale à partir x.

```
# Forme polynomiale
from sklearn.preprocessing import PolynomialFeatures
```

```
poly_features = PolynomialFeatures(degree = 2, include_bias = False)
x = poly_features.fit_transform(x)
```

Voyons maintenant l'allure de notre nuage de points.

```
# Visualisation
fig, axe = plt.subplots(figsize=(16,6))
axe.scatter(x[:,0],y,color="black");
axe.set(xlabel="x",ylabel="y");
plt.show()
```

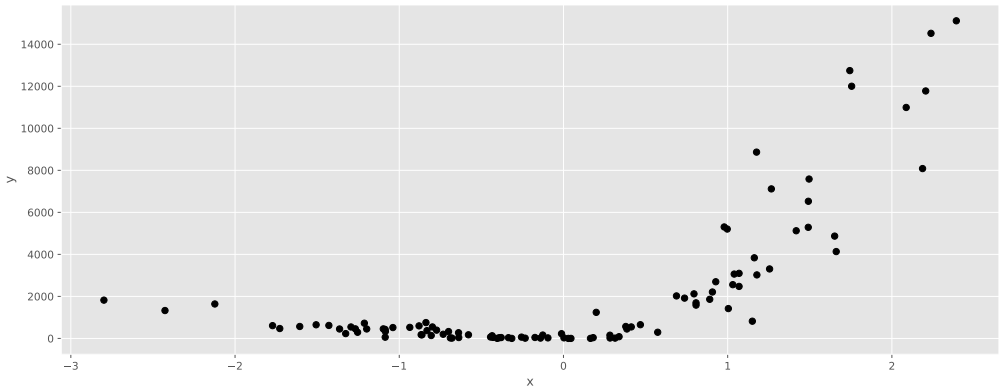


Figure 2.17 – Nuage de points

On voit que la forme du nuage a changé. Faire un ajustement linéaire avec notre jeu de donnée produira un modèle erroné.

Remarque 2.1 *Dans le cas où on dispose de plusieurs variables, le transformers PolynomialFeatures va effectuer toutes les combinaisons possibles pour créer un polynôme. Cependant, ces variables n'étant pas sur la même échelle, il faut les normaliser avant leur passage dans l'estimateur.*

b. Modèle polynomial

On peut à présent importer notre modèle, l'entraîner et évaluer sa performance.

```
# modèle par défaut
model = SGDRegressor()
model.fit(x,y)
```

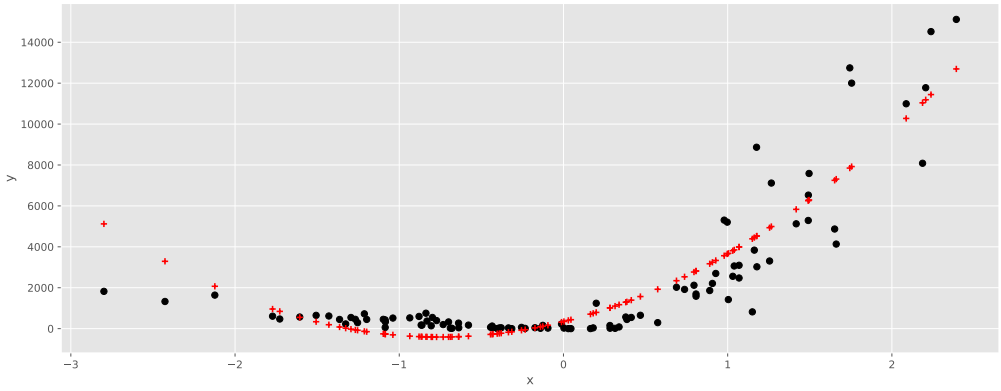
```
## SGDRegressor()
```

```
# Score
score = model.score(x,y)
print("Coeff R2 : %.2f"%(score))
```

```
## Coeff R2 : 0.83
```

Avec ce modèle, on obtient un score de 83‘.

```
# Prédiction
pred = model.predict(x)
# Visualisation
fig, axe =plt.subplots(figsize=(16,6))
axe.scatter(x[:,0],y, marker = "o",color="black");
axe.scatter(x[:,0], pred, c= "red", marker = "+");
axe.set(xlabel="x",ylabel="y");
plt.show()
```



2.2.2 Méthode des équations normales

La méthode des équations normales est celle qui repose sur l’algorithme des moindres carrés ordinaires. Sous Sklearn, on la retrouve à travers la fonction **LinearRegression**.

2.2.2.1 LinearRegression et régression linéaire simple

Nous allons utiliser un jeu de données présent dans sklearn. Il s’agit du dataset diabetes.

```
# Chargement de la base de données (dataset)
from sklearn.datasets import load_diabetes
diabetes_X, diabetes_y = load_diabetes(return_X_y = True)
```

Notre base de données dispose de 442 lignes (individus) et 10 prédicteurs (diabetes_X) et une variable à expliquer y (diabetes_y). Cependant, nous utiliserons uniquement un seul prédicteur à titre d’exemple d’application.

```
# Utilisation d'un seul prédicteur
diabetes_X = diabetes_X[ :, np.newaxis, 2]
```

a) Train set - test set

On sépare notre jeu de données en deux sous-échantillons : un échantillon d'apprentissage sur lequel on va entraîner notre modèle et un échantillon test sur lequel on va tester le modèle (évaluer la performance). L'échantillon d'apprentissage représente 70% de notre jeu de données et l'échantillon test 30%. Ensuite, on le visualise graphiquement.

```
# Echantillon d'apprentissage (X_train, y_train)
diabetes_X_train = diabetes_X[:-133]
diabetes_y_train = diabetes_y[:-133]
# Echantillons test (X_test, y_test)
diabetes_X_test = diabetes_X[-133:]
diabetes_y_test = diabetes_y[-133:]
# Représentation du nuage de points
fig, axe =plt.subplots(figsize=(16,6))
axe.scatter(diabetes_X_train, diabetes_y_train,color="black");
axe.set(xlabel="X",ylabel="y");
plt.show()
```

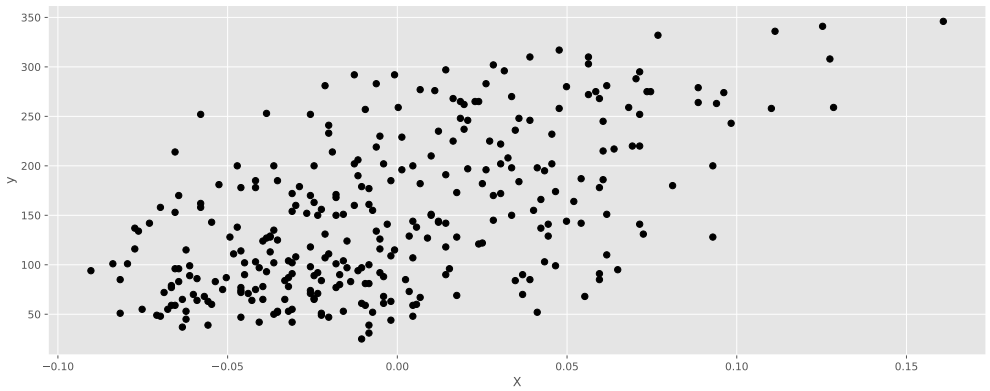


Figure 2.18 – Nuage de points

b) Mise en oeuvre de la modélisation

On construit notre estimateur. On l'entraine ensuite sur le train set.

```
# Création de l'objet de regression linéaire (estimateur)
from sklearn.linear_model import LinearRegression
regrModel = LinearRegression()
regrModel.fit(diabetes_X_train, diabetes_y_train)
```

```
## LinearRegression()
```

Grâce à la fonction **mean_squared_error** du module **metrics** de sklearn, on calcule la fonction coût (=l'erreur quadratique Moyen) du modèle.

```
# Erreur quadratique moyen (= fonction coût)
from sklearn.metrics import mean_squared_error
regrModelPredict = regrModel.predict(diabetes_X_test)
meanSquaredError = mean_squared_error(diabetes_y_test, regrModelPredict)
print("MQE : %.2f" %(meanSquaredError))
```

```
## MQE : 3673.67
```

L'erreur quadratique moyen est de 3673.67.

c) Coefficient de détermination

On évalue la performance de notre modèle de deux manières. Soit en calculant le score sur l'échantillon test représentant ainsi le R^2 sur les valeurs prédictes, soit d'utiliser la fonction r2_score du module **metrics**.

```
## Evaluation de la performance du modèle
# Méthode 1 : La fonction score
regrModelScore = regrModel.score(diabetes_X_test, diabetes_y_test)
print("Coeff R2 : %.2f "% (regrModelScore))
```

```
## Coeff R2 : 0.35
```

```
# Méthode 2 : (r2_score)
from sklearn.metrics import r2_score
regrModelPredict = regrModel.predict(diabetes_X_test)
r2Score = r2_score(diabetes_y_test, regrModelPredict)
print("Coeff R2 : %.2f " % (r2Score))
```

```
## Coeff R2 : 0.35
```

Avec ce modèle, on a un coefficient de détermination de 35 $\%$.

```
# Représentation graphique
fig, axe = plt.subplots(figsize=(16,6))
axe.scatter(diabetes_X_test,diabetes_y_test, color = "black")
axe.plot(diabetes_X_test, regrModelPredict, c = "blue", linewidth = 3)
axe.set(xlabel="X",ylabel="y");
plt.show()
```

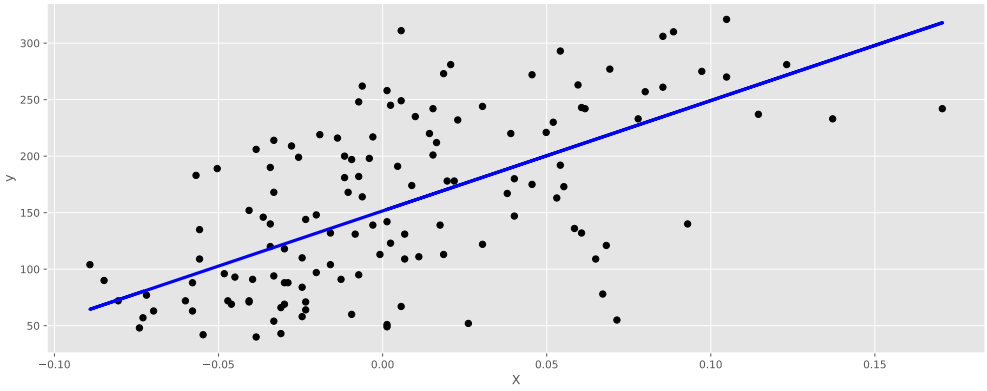


Figure 2.19 – Nuage de points avec tendance linéaire

On peut améliorer la précision de notre modèle en changeant la forme du modèle, soit en passant par un ajustement polynomiale, soit en augmentation le nombre de variables. C'est cette dernière qui sera utilisée.

2.2.2.2 LinearRegression et régression linéaire multiple

Nous allons utiliser tous nos features contenus dans notre data set.

```
# Chargement de la base de données (dataset)
diabetes_X, diabetes_y = load_diabetes(return_X_y = True)
```

On va allouer 80% de notre data set à l'échantillon d'apprentissage et 20% à l'échantillon test.

```
# Echntillon d'apprentissage - echantillon test
from sklearn.model_selection import train_test_split

diabetes_X_train,diabetes_X_test,diabetes_y_train,diabetes_y_test = train_test_split(
    diabetes_X, diabetes_y, test_size = 0.2, random_state = 5)
```

a) Estimateur

On estime le modèle sur le train set.

```
# Instanciation
model = LinearRegression()
# Entraînement
model.fit(diabetes_X_train, diabetes_y_train)


## LinearRegression()
```

b) Performance

On évalue la performance du nouveau modèle.

```
# Evaluation de la performance du modèle
# Erreur quadratique moyen
modelPredict = model.predict(diabetes_X_test)
modelMQE = mean_squared_error(diabetes_y_test, modelPredict)
print("EQM : %.2f" %(modelMQE))
```

```
## EQM : 2981.59
```

```
# coefficient de détermination
modelScore = r2_score(diabetes_y_test, modelPredict)
print("Coeff R2 : %.2f" %(modelScore))
```

```
## Coeff R2 : 0.53
```

Contrairement au cas précédent„ on obtient un coefficient de détermination de 53 % et une erreur quadratique moyen de 2981.59. Le score est supérieur et la fonction de coût est inférieure. Cependant, on peut encore améliorer notre modèle en faisant une sélection de variables (AIC, BIC, etc..).

Remarque 2.2 *Le générateur Linear Regression de Sklearn fonctionne très bien, mais il s'adapte mal aux gros datasets (quand il y a plusieurs centaines de features).*

3

K Nearest Neighbors (KNN)

Sommaire

L'algorithme KNN (K Nearest NeighBors) ou k plus proches voisins en français, est un algorithme d'apprentissage supervisé qui nous permet de faire des prédictions sur des variables qualitatives ou quantitatives à partir d'un ensemble de variables explicatives nominales, ordonnées ou quantitatives. Dans le cas où la variable à prédire est *qualitative*, on parle de **classification** ; lorsqu'elle est *quantitative*, on parle de **régression**.

Dans le KNN, pour prédire la classe (cas où la variable à prédire est qualitative) ou la valeur sur Y (cas où la variable à prédire est quantitative) d'une observation i,

— On calcule la **distance** qui sépare cette observation de toutes les observations
— On sélectionne **les k points plus proches**.

Cependant, la règle de décision diffère suivant le type de la variable à prédire :

— En **classification**, on applique la règle du **vote à la majorité**
— En **régression**, on calcule la **valeur prédicte** en faisant la moyenne des y mesurées sur ces plus proches voisins.

Sous scikit-learn, il existe une multitude d'algorithme de plus proches voisins contenu dans le module neighbors. Nous allons nous concentrer sur deux d'entre eux, notamment KNeighborsClassifier dans le cadre d'une variable à prédire de type catégorielle et KNeighborsRegressor dans le cas d'une variable à prédire quantitative.

3.1 Données

3.1.1 Tableau des données

Le jeu de données à partir duquel nous allons mettre en oeuvre l'algorithme des k plus proches voisins est le célèbre dataset iris contenu dans le module datasets .Ce dernier est une base de données regroupant les caractéristiques de trois espèces de fleurs d'Iris, à savoir *Setosa*, *Versicolour* et *Virginica*. Chaque ligne de ce jeu de données est une observation des caractéristiques d'une fleur d'Iris. Ce dataset décrit les espèces d'Iris par quatres propriétés : longueur et largeur de sépales ainsi que longueur et

largeur de pétales. La base de données comporte 150 observations.

```
# Chargement du dataset
from sklearn.datasets  import load_iris
iris = load_iris(as_frame = True)
```

Le dataset **iris** est composé de 4 features (data) ou prédicteurs et 1 target (variable à prédire) qu'on met dans deux variables différentes.

```
# data - target
iris_X = iris.data
iris_y = iris.target
```

Les features sont disponibles dans le tableau iris_X et correspondent à : sepal length (en cm), sepal width (en cm), petal length (en cm), petal width (en cm). Le target est contenu dans le tableau iris_y et correspond à une variable qualitative prenant trois modalités : 0, 1 et 2.

On peut observer les 5 permières observations des features.

Table 3.1 – 5 premières observations

sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2

Le target est sous forme de séries. Nous allons le transformer en le mettant sous forme de dataframe à l'aide module pandas et de quelques manipulations comme **reshape**.

```
# Tranformation en dataframe
import pandas as pd
n = iris_y.values.shape[0]
iris_y = iris_y.values.reshape(n,1)
iris_y = pd.DataFrame(iris_y, columns = ["class"])
```

Nous calculons le nombre d'individus par classes.

```
# Effectifs des classes
n_k = iris_y.value_counts().to_frame("Effectifs").reset_index()
```

Table 3.2 – Effectifs des classes

class	Effectifs
0	50
1	50
2	50

On constate effectivement que les classes sont d'éffectifs égaux., donc il y a équiprobabilité. C'est ce qu'on vérifie en mettant à True l'hyperparamètre *normalize*.

```
# Distribution des classes
p_k = iris_y.value_counts(normalize=True).to_frame("Proportions").reset_index()
```

Table 3.3 – Probabilités a priori

class	Proportions
0	0.3333333
1	0.3333333
2	0.3333333

Remarque 3.1 *On peut également concatener les dataframes iris_X et iris_y afin d'avoir un dataset complet sous forme de dataframe.*

3.1.2 Statistiques

3.1.2.1 Statistiques descriptives

Elles consistent à analyser les indicateurs de tendance centrale et de dispersion.

```
# Statistiques descriptives
stats = iris_X.describe(include = "all").T
```

Table 3.4 – Statistiques descriptives sur les variables

	count	mean	std	min	25%	50%	75%	max
sepal length (cm)	150	5.843333	0.8280661	4.3	5.1	5.80	6.4	7.9
sepal width (cm)	150	3.057333	0.4358663	2.0	2.8	3.00	3.3	4.4
petal length (cm)	150	3.758000	1.7652982	1.0	1.6	4.35	5.1	6.9
petal width (cm)	150	1.199333	0.7622377	0.1	0.3	1.30	1.8	2.5

3.1.2.2 Matrice de corrélation de Pearson

La matrice de corrélation indique les valeurs de corrélation, qui mesurent le degré de relation linéaire entre chaque paire de variables. Les valeurs de corrélation peuvent être comprises entre -1 et +1. Si les deux variables ont tendance à augmenter et à diminuer en même temps, la valeur de corrélation est positive. Lorsqu'une variable augmente alors que l'autre diminue, la valeur de corrélation est négative.

Utilisez la matrice de corrélation pour évaluer l'importance et la direction de la relation entre deux variables. Une valeur de corrélation positive élevée indique que les variables mesurent la même caractéristique. Si les items ne sont pas fortement corrélés, ils peuvent mesurer des caractéristiques différentes ou ne pas être clairement définis.

```
# Matrice de corrélation entre les variables
matcorr = iris_X.corr(method="pearson")
```

Une relation linéaire positive existe entre les variables `sepal length` et `petal length`, `sepal length` et `petal width` et `petal length` et `petal width`. Pour ces paires, les coefficients de Pearson sont les suivants :

Table 3.5 – Matrice des corrélations linéaires de Pearson

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
sepal length (cm)	1.00	-0.12	0.87	0.82
sepal width (cm)	-0.12	1.00	-0.43	-0.37
petal length (cm)	0.87	-0.43	1.00	0.96
petal width (cm)	0.82	-0.37	0.96	1.00

— sepal length et petal length : 0.87
— sepal length et petal width : 0.82
— petal length et petal width : 0.96

Ces valeurs indiquent une relation positive modérée entre les variables.

Une relation linéaire négative existe pour les paires suivantes, avec des coeffcients de corrélation de Pearson négatifs :

— sepal length et sepal width : -0.12
— sepal width et petal length : -0.42
— sepal width et petal width : -0.36

3.1.2.3 Histogramme

En statistique, un histogramme est une représentation graphique permettant de représenter la répartition empirique d'une variable aléatoire ou d'une série statistique en la représentant avec des colonnes correspondant chacune à une classe et dont l'aire est proportionnelle à l'effectif de la classe.

```
import seaborn as sns
fig, axs = plt.subplots(figsize=(16,6),ncols=4)
sns.set(font_scale=4)
for i, name in enumerate(iris_X.columns):
  sns.histplot(iris_X[name],color='black',ax=axs[i]);
plt.tight_layout();
plt.show()
```

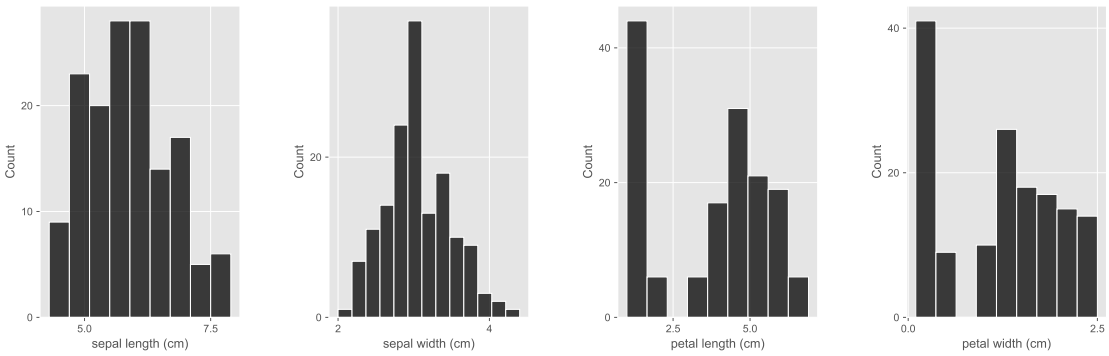


Figure 3.1 – Histogramme

3.1.3 Train set - Test set

Pour trouver le meilleur compromis **biais - variance**, il est recommandé de subdiviser notre dataset en deux parties :

— Une partie qui sera utilisée pour entraîner le modèle avec différents niveaux de complexité et différentes valeurs d'hyperparamètres qu'on appelle **échantillon d'apprentissage** ou (**train set** en anglais).
— Une partie pour mesurer l'erreur commise par la meilleure règle de prédiction et la comparer à l'erreur commise par des règles de familles différentes sur des données qui n'ont pas du tout participé à la construction des règles : C'est **l'échantillon test** ou (**test set** en anglais).

Sous scikit-learn, on utilise la fonction **train_test_split** du module model_selection qui permet de séparer le dataset en train set et en test set. Cette méthode mélange le dataset de façon aléatoire avant d'effectuer le tirage. La part des observations consacrée à test set dépend de la taille totale du dataset. Avec un test set trop grand, on peut nuire à la qualité de l'apprentissage. En général, on met 80% de notre dataset dans le train set et 20% sur le test set.

```
from sklearn.model_selection import train_test_split
# Echantillon d'apprentissage - échantillon test
iris_X_train, iris_X_test, iris_y_train, iris_y_test = train_test_split(
    iris_X,iris_y, test_size = 0.2, random_state = 5)
```

L'hyperparamètre test_size permet de définir le pourcentage de données à mettre dans l'échantillon test et random_state permet de contrôler l'aléa.

3.2 Mise en oeuvre de l'algorithme KNN

3.2.1 Algorithme KNN et entraînement

On va importer la fonction KNeighborsClassifier du module neighbors à partir duquel on définit notre modèle en fixant le nombre de voisin à 1, puis on va l'entraîner sur le train set. On rappelle que l'algorithme des k plus proches voisins est d'une complexité croissante en fonction de $\frac{1}{k}$. Donc le choix de k est central dans ce modèle. Par conséquent, en fixant le nombre de proches voisins à 1, on part d'un modèle extrêmement complexe.

```
# Algorithme KNN
from sklearn.neighbors import KNeighborsClassifier
knnModel = KNeighborsClassifier(n_neighbors = 1)
# Entrainement - Apprentissage
knnModel.fit(iris_X_train, iris_y_train)


## KNeighborsClassifier(n_neighbors=1)
```

Une fois notre modèle entraîné on peut à présent l'évaluer.

3.2.1.1 Score

On peut évaluer la performance de notre modèle de deux manières différentes : soit en utilisant l'attribut score(X,y) du modèle ; soit en important la fonction accuracy_score contenue dans le module metrics.

a) L'attribut score

On peut voir que si on évalue notre modèle sur les mêmes données qui ont servi d'apprentissage, alors on obtient un score de 100%.

```
# Evaluation du modèle sur le train set
knnModelScoreTrain = knnModel.score(iris_X_train, iris_y_train)
print("Score Train : %.4f" % (knnModelScoreTrain))
```

```
## Score Train : 1.0000
```

Est-ce que cela signifie que notre modèle réussira à 100% de ses prédictions dans le futur ? La réponse est **non**. Pour avoir une idée de la performance future de notre modèle, il faut le tester sur les données qu'il n'a jamais vu, c'est-à-dire sur le test set.

```
# Evaluation du modèle sur le test set
knnModelScoreTest1 = knnModel.score(iris_X_test, iris_y_test)
print("Score Test : %.4f" % (knnModelScoreTest1))
```

```
## Score Test : 0.9000
```

On obtient un score de 90% avec ce modèle.

La fonction accuracy_score

Cette fonction nécessite l'utilisation de la prédiction faite sur les features contenus dans le test set.

```
from sklearn.metrics import accuracy_score
# Prediction
knnModelPredictions = knnModel.predict(iris_X_test)
print("Predictions :\n", knnModelPredictions)
```

```
## Predictions :
##  [1 2 2 0 2 1 0 2 0 1 1 1 2 2 0 0 2 2 0 0 1 2 0 2 1 2 1 1 1 2]
```

```
# Evaluation du modèle sur l'échantillon test : accuracy_score
knnModelScoreTest2 = accuracy_score(iris_y_test, knnModelPredictions)
print("Score test accuracy : %.4f" % (knnModelScoreTest2))
```

```
## Score test accuracy : 0.9000
```

On obtient également un score de 90% avec cette fonction.

3.2.1.2 Taux d'erreur

Le taux d'erreur peut être calculé de deux manières différentes : soit en faisant la somme des individus mal classés que l'on divise par la taille de l'échantillon, soit en utilisant l'évènement contraire du score. On aboutit au même résultat. Dans la première approche, on utilise la fonction sum de la librairie numpy pour sommer l'ensemble des individus mal classés.

```
# Approche 2
knnModelErrorRate = 1 - knnModelScoreTest1
print("Error rate 2 : %.4f" % (knnModelErrorRate))
```

```
## Error rate 2 : 0.1000
```

On a un taux d'erreur de 10% avec ce modèle.

Cependant, on doit améliorer les performances de notre modèle de manière à obtenir un score de 91%, 92%, voire 99%. Pour cela, il va valoir régler les **hyperparamètres** de notre modèle. Par exemple, si on change le nombre de voisin en le fixant à 3.

```
# Algorithme KNN avec 3 voisins
knnModel = KNeighborsClassifier(n_neighbors = 3)
knnModel.fit(iris_X_train, iris_y_train)
```

```
## KNeighborsClassifier(n_neighbors=3)
```

```
knnModelScoreTest = knnModel.score(iris_X_test, iris_y_test)
print("Score Test (3 voisins) : %.4f" % (knnModelScoreTest))
```

```
## Score Test (3 voisins) : 0.9333
```

on obtient un score 93.33% sur les données de l'échantillon test. Il y a eu amélioration. Ce n'est pas mal. Disons qu'on teste un nombre de voisin égal à 6.

```
# Algorithme KNN avec 6 voisins
knnModel = KNeighborsClassifier(n_neighbors = 6)
knnModel.fit(iris_X_train, iris_y_train)
```

```
## KNeighborsClassifier(n_neighbors=6)
```

```
knnModelScoreTest = knnModel.score(iris_X_test, iris_y_test)
print("Score Test (6 voisins) : %.4f" % (knnModelScoreTest))
```

```
## Score Test (6 voisins) : 0.9667
```

On obtient un score de 96.67%.

ATTENTION !!!

Si on règle notre modèle en optimisant sa performance sur l'échantillon test, comme ce qu'on est en train de faire, alors on ne pourra plus utiliser ces données pour faire l'évaluation finale de notre modèle. **Pourquoi ?**. En fait, pour évaluer un modèle, il faut le soumettre à des données qu'il n'a jamais vu. Or si on règle notre modèle sur les données de l'échantillon test, alors il aura indirectement vu ces données puisqu'il est réglé dessus. Pour cette raison, on découpe une troisième section dans notre dataset : on l'appelle **l'échantillon de validation**.

3.2.2 Échantillon de validation

L'échantillon de validation permet de mesurer l'erreur commise par les prédicteurs et le choix de la règle dont le niveau de complexité et les valeurs des hyperparamètres qui minimisent le risque empirique. Cette section nous permet de chercher les réglages du modèle qui donne les meilleures performances tout en gardant de côté les données de l'échantillon test pour évaluer la machine sur des données qu'elle n'aura jamais vu.

Quand on veut comparer deux modèles de machine learning, par exemple un KneighborsClassifier avec 3 voisins et un autre avec 6 voisins, on va commencer par entraîner ces deux modèles sur l'échantillon d'apprentissage, puis on sélectionne celui qui aura la meilleure performance sur l'échantillon de validation. Ensuite, on pourra évaluer ce modèle sur l'échantillon test afin d'avoir une idée de sa performance dans la vraie vie.

Qu'est - ce qui nous garantie que la façon dont on découpe notre dataset est la bonne ?

Si ça se trouve, en entraînant puis en validant nos deux modèles (A = 92% et B = 90%) sur une autre portion de données, alors on découvrira que c'est le modèle B (A = 88% et B = 93%) qui est le meilleur.

Que faire ?

Face à cette situation, il existe une solution : c'est **la cross validation** ou **validation croisée**.

3.2.2.1 La validation croisée

La validation croisée consiste à entraîner, puis valider notre modèle sur plusieurs découpes possibles de l'échantillon d'apprentissage. Par exemple, en découpant l'échantillon d'apprentissage en 5 parties, on peut entraîner notre modèle sur les 4 premières parties puis le valider sur la 5^e partie. Ensuite, on refait l'algorithme pour toutes les configurations possibles. Au final, on fait la moyenne des 5 scores que l'on obtient. Ainsi, lorsqu'on voudra comparer deux modèles, alors on est sûr de prendre celui qui a en moyenne eu les meilleures performances. Il existe plusieurs façons de découper l'échantillon d'apprentissage avec la technique de cross-validation que vous pouvez consulter ici. On utilisera la méthode KFold avec 5 blocs. Pour cela nous allons importer la fonction cross_val_score du module **model_selection**.

```
# Validation croisée
from sklearn.model_selection import cross_val_score
knnModelCrossValidation  = cross_val_score(KNeighborsClassifier(),iris_X_train,
                                iris_y_train, cv = 5, scoring = "accuracy")
print(knnModelCrossValidation)
```

```
## [1.         1.         1.         0.95833333 0.95833333]
```

L'hyperparamètre **cv** correspond au nombre de bloc à définir pour la cross validation. On obtient 5 scores correspondant à nos 5 splits de validation. knnModelCrossValidation renvoit un array de type numpy contenant le score du modèle à chaque itération. On fait la moyenne pour obtenir le score final.

```
# Moyenne
knnModelCrossValidationMean = knnModelCrossValidation.mean()
print("Score moyen : %.4f" % (knnModelCrossValidationMean))
```

```
## Score moyen : 0.9833
```

On obtient un score de 98.33%. Désormais, on peut évaluer différents modèles pour retenir celui qui a la meilleure performance. Par exemple, quand `n_neighbors = 1`, on a :

```
knnModelCrossValidationMean = cross_val_score(KNeighborsClassifier(1),iris_X_train,
                    iris_y_train, cv = 5, scoring = "accuracy").mean()
print("Score cross (1 voisin) : %.4f" % (knnModelCrossValidationMean))
```

```
## Score cross (1 voisin) : 0.9750
```

On obtient un score de 97.5%. Quand `n_neighbors = 2`, on a :

```
knnModelCrossValidationMean = cross_val_score(KNeighborsClassifier(2),iris_X_train,
                    iris_y_train, cv = 5, scoring = "accuracy").mean()
print("Score cross (2 voisins) : %.4f" % (knnModelCrossValidationMean))
```

```
## Score cross (2 voisins) : 0.9667
```

On obtient un score de 96.67%. Quand `n_neighbors = 3`, on a :

```
knnModelCrossValidationMean = cross_val_score(KNeighborsClassifier(3),iris_X_train,
                    iris_y_train, cv = 5, scoring = "accuracy").mean()
print("Score cross (3 voisins) : %.4f" % (knnModelCrossValidationMean))
```

```
## Score cross (3 voisins) : 0.9750
```

On obtient 97.5%.

Pour aller plus vite, on peut tester dans une boucle for en enregistrant chaque score qu'on obtient dans une liste **knnModelCrossValidationScore** :

```
# Pour un nombre de voisin allant de 1 à 50
knnModelCrossValidationScore = []
for k in range(1,51):
    score = cross_val_score(KNeighborsClassifier(n_neighbors = k),iris_X_train,
                            iris_y_train,cv = 5).mean()
    knnModelCrossValidationScore.append(score)
```

En affichant ce résultat avec matplotlib, on obtient le résultat suivant.

```
# Représentation graphique
fig, axe = plt.subplots(figsize=(16,6))
axe.plot(knnModelCrossValidationScore,color="blue");
axe.set(xlabel="Nombre de voisins",ylabel="Score moyen");
plt.show()
```

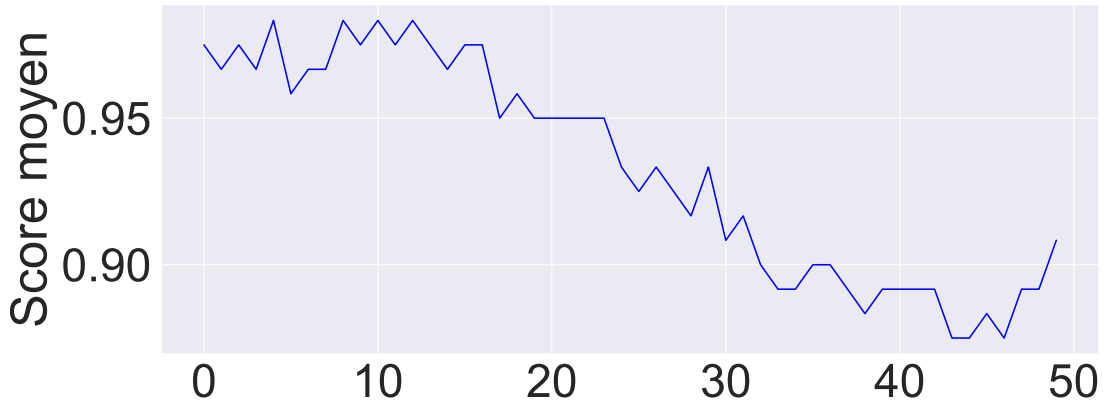


Figure 3.2 – Score moyen par validation croisée

On peut voir qu'on obtiendra les meilleures performances lorsqu'on aura des nombres de voisin qui seront autour de 10.

Cependant, toute cette boucle for, nous n'avons pas besoin de l'écrire car il existe dans scikit-learn une fonction qui permet de créer ce genre de graphique : c'est la fonction validation_curve du module model_selection.

3.2.2.2 La fonction validation_curve

La fonction validation_curve teste toutes les valeurs pour un hyperparamètre donné. Elle calcule tous les scores sur l'échantillon d'apprentissage et tous les scores sur l'échantillon de validation grâce à la cross validation.

```
# La fonction validation_curve
from sklearn.model_selection import validation_curve
irisTrainScore, irisValScore = validation_curve(KNeighborsClassifier(),
                                                iris_X_train,iris_y_train,
                                                param_name="n_neighbors",
                                    param_range = [k for k in range(1,51)],cv = 5)
```

Sur irisValScore, on obtient un tableau numpy dont le nombre de ligne correspond au nombre de voisins et le nombre de colonnes au nombre de bloc (cv).

```
# Dimension
print("Shape :",irisValScore.shape)
```

```
## Shape : (50, 5)
```

On obtient $(50, 5)$. Pour avoir le score moyen par validation, prend la moyenne de chaque ligne c'est-à-dire la moyenne suivant l'axe 1.

```
# Moyenne de chaque ligne
irisValScoreMean = irisValScore.mean(axis = 1)
print("Score moyen pour chaque cross validation : \n",irisValScoreMean)
```

```
## Score moyen pour chaque cross validation :
##  [0.975      0.96666667 0.975      0.96666667 0.98333333 0.95833333
##  0.96666667 0.96666667 0.98333333 0.975      0.98333333 0.975
##  0.98333333 0.975      0.96666667 0.975      0.975      0.95
##  0.95833333 0.95       0.95       0.95       0.95       0.95
##  0.93333333 0.925      0.93333333 0.925      0.91666667 0.93333333
##  0.90833333 0.91666667 0.9        0.89166667 0.89166667 0.9
##  0.9        0.89166667 0.88333333 0.89166667 0.89166667 0.89166667
##  0.89166667 0.875      0.875      0.88333333 0.875      0.89166667
##  0.89166667 0.90833333]
```

On obtient un tableau qui contient 50 éléments. On affiche ce tableau à l'aide de matplotlib.

```
# Graphique
k = range(1,51)
fig, axe = plt.subplots(figsize=(16,6))
axe.plot(k, irisValScoreMean, label = "validation");
axe.set(ylabel="score",xlabel="Nombre de voisins");
plt.legend()
plt.show()
```

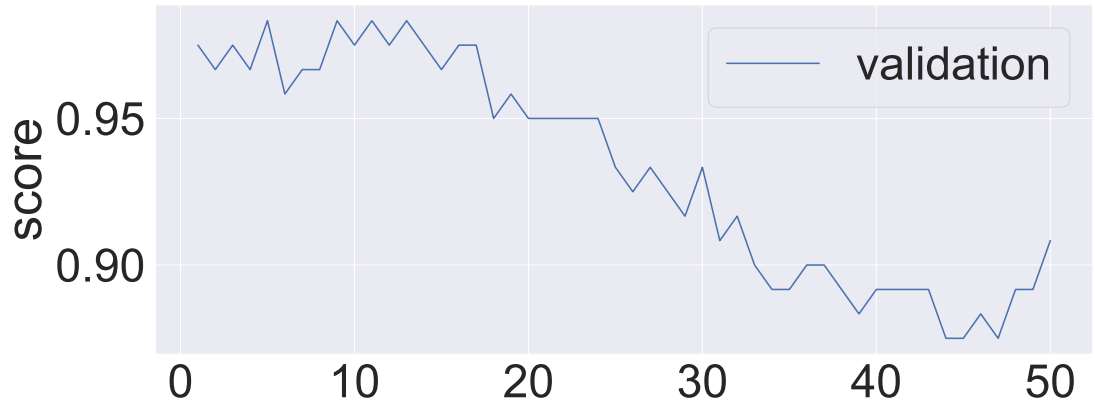


Figure 3.3 – Score moyen par validation croisée (validation_curve)

On voit qu'on obtient exactement le même graphique que le précédent. On peut aussi voir le score sur l'échantillon d'apprentissage.

```
# Graphique avec score sur le train set
fig, axe = plt.subplots(figsize=(16,6))
axe.plot(k, irisValScoreMean, label = "validation");
axe.plot(k, irisTrainScore.mean(axis =1), label = "Apprentissage");
axe.set(ylabel="score",xlabel="Nombre de voisins");
plt.legend()
plt.show()
```

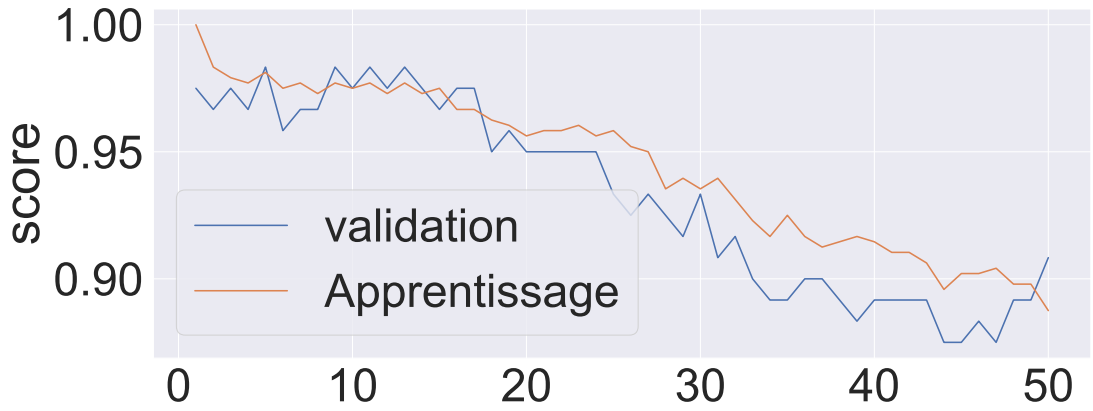


Figure 3.4 – Score moyen par validation croisée (validation_curve)

Ceci est très utile pour détecter les cas d'overfitting ou de **sur-apprentissage** c'est-à-dire le modèle s'est trop perfectionné sur l'échantillon d'apprentissage et a perdu tout sens de généralisation. On peut donc repérer ce problème lorsqu'on a un très bon score sur l'échantillon d'apprentissage mais un moins bon score sur l'échantillon de validation. Le modèle commence à **overfitter**, les erreurs sur le train set augmentent tandis que celles sur le test set augmentent.

En occurence dans les algorithmes de nearest neighbors, on est très souvent en cas d'overfitting lorsqu'on a un nombre de voisin égale à 1.

Avec le graphique précédent, on sait qu'on peut atteindre 98% de performance en choisissant la bonne valeur pour l'hyperparamètre `n_neighbors`. Dans l'algorithme nearest neighbors, il existe d'autres hyperparamètres que le `n_neighbors` comme par exemple le type de `distance` (distance de Manhattan, distance euclidienne). On peut aussi choisir d'accorder ou non les coefficients sur nos distances. Du coup, en reglant ces autres hyperparamètres, on peut peut-être avoir encore une meilleure performance. Pour tester toutes ses combinaisons, le mieux serait d'utiliser la fonction GridSearchCV du module model_selection.

3.2.2.3 GridSearchCv

GridSearchCV permet de trouver le modèle avec les meilleures hyperparamètres en comprarant les différentes performances de chaque combinaison grâce à la technique de cross_validation. On crée un dictionnaire qui contient les différents hyperparamètres à régler ainsi que chaque valeur à tester pour ces hyperparamètres.

```
from sklearn.model_selection import GridSearchCV
import numpy as np
# Distionnaire des différents hyperparamètres à regler
param_grid = {"n_neighbors" : np.arange(1,51),
              "metric": ["euclidean", "manhattan", "minkowski"]}
# Modèle
grid = GridSearchCV(KNeighborsClassifier(), param_grid, cv = 5)
```

On a un grid qui contient plusieurs estimateurs. On va donc entraîner ce grid avec la méthode fit(), comme s'il s'agit d'un estimateur, en passant les données de l'échantillon d'apprentissage.

```
# Entraînement
grid.fit(iris_X_train, iris_y_train);
```

Une fois l'entraînement terminé, on peut voir le modèle qui a obtenu le meilleure score.

```
# Modèle avec meilleure score
best_score = grid.best_score_
print("Best score : %.4f" % (best_score))
```

```
## Best score : 0.9833
```

On obtient 98.33%. Ce qui n'est pas mal. On peut également voir les meilleurs paramètres de ce modèle.

```
# Meilleur paramère
bestParams = grid.best_params_
print("Best params :", bestParams)
```

```
## Best params : {'metric': 'euclidean', 'n_neighbors': 5}
```

On voit donc que le meilleur modèle utilise la distance `"euclidean"` avec un nombre de voisin égal à 5. Et pour finir, on peut sauvegarder ce modèle en écrivant :

```
# Sauvegarde du modele
bestmodel = grid.best_estimator_
```

Pour finir, testons ce modèle sur les données d'échantillon test afin d'avoir un apercu de sa performance dans la vraie vie.

```
# Tester le modèle
score = bestmodel.score(iris_X_test, iris_y_test)
print("Score : %.4f" % (score))
```

```
## Score : 0.9333
```

On aurait 93.33% avec un tel modèle.

A présent on pourrait se demander si notre modèle pourrait encore avoir de meilleures performances si on lui fournissait plus de données. Pour répondre à cette question très importante, il faut tracer ce qu'on appelle **les courbes d'apprentissage**.

3.2.3 Les courbes d'apprentissage

Les courbes d'apprentissage montrent l'évolution des performances du modèle en fonction de la quantité de données qu'on lui fournit. Typiquement, plus la machine dispose de données pour s'entraîner, meilleure sera sa performance. Cependant, la performance finit toujours par atteindre un plafond et lorsque ceci arrive, il est inutile d'avoir plus de données. Pour cela, on va utiliser la fonction learning_curve du module model_selection.

```
# Les courbes d'apprentissage
from sklearn.model_selection import learning_curve
N, train_score, val_score = learning_curve(bestmodel, iris_X_train, iris_y_train,
                                           train_sizes = np.linspace(0.1, 1.0, 10),
                                           cv = 5)
```

np.linspace(start, stop, num=50) représente les quantités de données à entrer pour l'entraînement. La plupart du temps, cela est en pourcentage. `np.linspace(0.1, 1.0, 10)` signifie que si notre dataset comptient 100 points, on aura : 10 points dans le premier lot, 20 points dans le second lot, ainsi de suite jusquà 100 points au dernier lot. Ces différentes quantités seront retournées dans la fonction **N**.

```
# Graphique
fig,axe = plt.subplots(figsize=(16,6))
axe.plot(N, train_score.mean(axis = 1), label = "train");
axe.plot(N, val_score.mean(axis = 1), label = "Validation");
plt.legend()
plt.show()
```

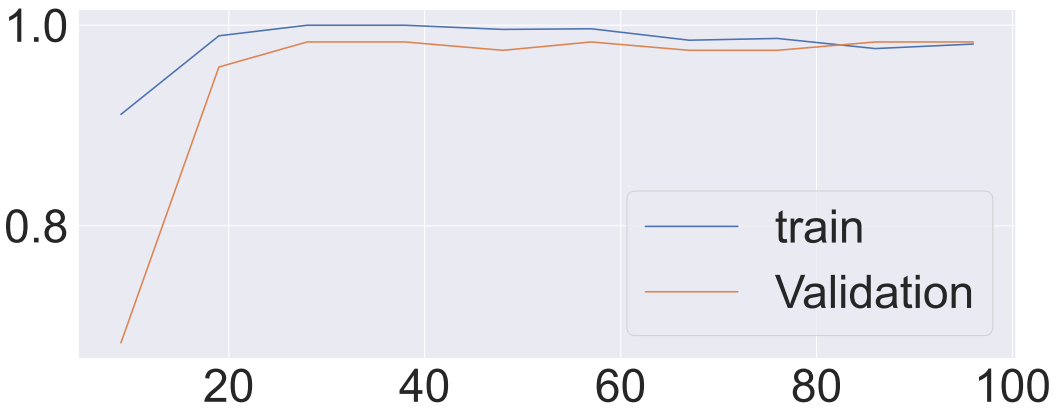


Figure 3.5 – Courbe d'apprentissage

On peut voir que la performance n'évolue presque plus à partir du moment où on a 80 points dans notre dataset.

4 Naive Bayes Classifier

Sommaire

Le classifieur naif bayésien repose sur l'hypothèse forte que les variables explicatives X_j sont indépendantes conditionnellement à la classe de Y. Malgré cette hypothèse simpliste d'indépendance entre les regresseurs conditionnellement aux valeurs prises par Y, le classifieur bayésien naif s'avère souvent plus performant que des modèles plus sophistiqués.

Si le nombre de régresseurs est grand, cette technique est particulière appropriée. En effet, en grande dimension, l'estimation des fonctions de densité devient très compliquée. Avec le classifieur bayésien naif, chaque loi de probabilité des X_j conditionnellement aux valeurs de Y peut être estimée indépendamment en tant que loi de probabilité à une dimension.

La bibliothèque scikit-learn de Python destinée à l'apprentissage automatique fourni le module sklearn.naive_bayes qui contient plusieurs classificateurs Bayes Naifs, dont chacun performe mieux sur un certain type de donnée. Nous allons nous intéresser au classifieur gaussien GaussianNB.

4.1 Données

4.1.1 Tableau de données

Nous allons mettre en oeuvre l'algorithme GaussianNB avec le fameux dataset titanic contenu dans la bibliothèque seaborn. Certains régresseurs sont quantitatifs et d'autres catégorielles. La variable à prédire est « `survived` ».

```
# Chargement du dataset
import seaborn as sns
titanic = sns.load_dataset("titanic")
# Dimension
titanic.shape
```

```
## (891, 15)
```

Le dataset contient 891 observations et 14 features. Voici les premières lignes de notre dataset.

Table 4.1 – Premières lignes - Données TITANIC

survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	3	male	22	1	0	7.2500	S	Third	man	TRUE	NA	Southampton	no	FALSE
1	1	female	38	1	0	71.2833	C	First	woman	FALSE	C	Cherbourg	yes	FALSE
1	3	female	26	0	0	7.9250	S	Third	woman	FALSE	NA	Southampton	yes	TRUE
1	1	female	35	1	0	53.1000	S	First	woman	FALSE	C	Southampton	yes	FALSE
0	3	male	35	0	0	8.0500	S	Third	man	TRUE	NA	Southampton	no	TRUE
0	3	male	NaN	0	0	8.4583	Q	Third	man	TRUE	NA	Queenstown	no	TRUE

Nous allons extraire les features dont nous aurons besoin pour l'implémentation de l'algorithme. Nous aurons besoin de pclass, `sex`, `age` et `fare`.

```
# Données à utiliser
titanic = titanic[["survived", "pclass", "sex", "age", "fare"]]
```

Avec cette réduction de dimension, nous affichons les informations sur le type des variables

```
# Information sur les variables
print(titanic.info())
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 891 entries, 0 to 890
## Data columns (total 5 columns):
##  #   Column    Non-Null Count  Dtype
## ---  ------    --------------  -----
##  0   survived  891 non-null    int64
##  1   pclass    891 non-null    int64
##  2   sex       891 non-null    object
##  3   age       714 non-null    float64
##  4   fare      891 non-null    float64
## dtypes: float64(2), int64(2), object(1)
## memory usage: 34.9+ KB
## None
```

On peut a présent séparer notre dataset en target et features.

```
# Separation du jeu de donnée
target = titanic.survived
data = titanic.drop("survived", axis = "columns")
```

4.1.2 Préprocessing

On va effectuer certains nettoyages et transformations de nos variables (gestion de données manquantes : suppression, imputation, repondération). La variable sex sera séparée en deux dummies variables, ceci grâce à pandas.

```
# Dummy variables
import pandas as pd
dummies = pd.get_dummies(data.sex)
```

On fusionne notre dataset avec cette nouvelle base

```
# Concaténation
data = pd.concat([data, dummies], axis = "columns")
```

On supprime la variable age.

```
# Suppression de la variable sex
data.drop("sex", axis = "columns", inplace = True)
```

On vérifie s'il n'y a pas de valeur manquante dans notre dataset.

```
# Vérification des valeurs manquantes
data.columns[data.isna().any()]
```

```
## Index(['age'], dtype='object')
```

On constate que la variable age possède des variables manquantes. On va les remplacer par la moyenne de la variable.

```
# Remplacer NAN par la valeur moyenne
data.age = data.age.fillna(data.age.mean())
```

4.1.3 Train set - Test set

On subdivise notre dataset en 80-20 : 80% pour le train set et 20% pour le test set.

```
# Train set - test set
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(data, target,test_size = 0.2,
                                                random_state = 5)
```

4.2 Implémentation de l'algorithme

4.2.1 Instanciation et entrainement

Nous faisons appel à la fonction GaussianNB du module naive_bayes de sklearn.

```
# Appel du model
from sklearn.naive_bayes import GaussianNB
```

...puis nous instancions et entrainon notre model sur le train set.

```
# Instanciation  - Entrainement
model = GaussianNB().fit(Xtrain, ytrain)
```

On peut à présent faire des prédictions et évaluer les performances de notre modèle sur le test set.

```
# Prédictions
ypred = model.predict(Xtest)
# Evaluation
from sklearn.metrics import accuracy_score
ac = accuracy_score(ytest, ypred)
print("score : %.4f" %(ac))
```

```
## score : 0.7989
```

Notre modèle réussi à 79.89% sur ces prédictions.

4.2.2 Validation croisée

On peut améliorer les performances de notre modèle en effectuant une validation croisée à 10 blocs.

```
# Cross_val_score
from sklearn.model_selection import cross_val_score
crossval = cross_val_score(model, Xtrain, ytrain,cv = 10,
                           scoring = "accuracy")
# Moyenne
crossvalmean = crossval.mean()
print("Score par validation croisée : %.4f" % (crossvalmean))
```

```
## Score par validation croisée : 0.7752
```

Celle-ci nous fournit un score de 77.52%. Ce qui n'est pas mal.

4.2.3 GaussianNB avec probabilités à priori égales

Le modèle précédent, lors de son instanciation, à utiliser les probabilités a priori telles que définies pour chaque classe de la target. Cependant, les effectifs des 2 groupes sont déséquilibrés. Ceux de la classe 0 sont sur-représentés et la règle de décision va privilégier leur classement. On choisira une équi-pondération a priori qui ne favorise aucun groupe.

```
# Modèle avec probabilités à priori égales
model2 = GaussianNB(priors = [[0.5],[0.5]]).fit(Xtrain, ytrain)
# Prédiction
ypred2 = model2.predict(Xtest)
```

On évalue la performance de ce nouveau modèle.

```
# Evaluation
ac2 = accuracy_score(ytest, ypred2)
print("score : %.4f" %(ac2))
```

```
## score : 0.8045
```

Le score obtenu est de 80.45%. il diffère de celui obtenu précédemment. Le rééquilibrage des classes accroît la performance du modèle.

```
# Cross_val_score
crossval = cross_val_score(model2, Xtrain, ytrain,cv = 10,
                           scoring = "accuracy")
# Moyenne
crossvalmean = crossval.mean()
print("Score par validation croisée : %.4f" % (crossvalmean))
```

```
## Score par validation croisée : 0.7780
```

En validation croisée, le score de 77.8%. L'augmentation n'est pas très significative.

5 Arbres de décision

Sommaire

Un arbre de décision est un outil d'aide à la décision représentant un ensemble de choix sous la forme graphique d'un arbre. Les différentes décisions possibles sont situées aux extrémités des branches (les « feuilles » de l'arbre), et sont atteintes en fonction de décisions prises à chaque étape. L'arbre de décision est un outil utilisé dans des domaines variés tels que la sécurité, la fouille de données, la médecine, etc. Il a l'avantage d'être lisible et rapide à exécuter. Il s'agit de plus d'une représentation calculable automatiquement par des algorithmes d'apprentissage supervisé.

5.1 Discrimination par arbre

5.1.1 Présentation

Les arbres de décision sont utilisés dans des domaines d'aide à la décision (par exemple l'informatique décisionnelle) ou l'exploration de données. Ils décrivent comment répartir une population d'individus (clients d'une entreprise, utilisateurs d'un réseau social,...) en groupes homogènes selon un ensemble de variables discriminantes (âge, temps passé sur un site Web, catégorie socio-professionnelle, ...) et en fonction d'un objectif fixé (aussi appelé « variable d'intérêt » ou « variable de sortie » ; par exemple : chiffre d'affaires, probabilité de cliquer sur une publicité,...). Par exemple, l'arbre de décision ci-dessous (*cf.* Figure 5.1) (tiré de l'ouvrage de Quinlan (1993)) illustre le cas où l'on cherche à prédire le comportement de sportifs (la variable à prédire « `Jouer` » prenant l'une des deux valeurs « oui » ou « non ») en fonction de données météorologiques (Ensoleillement, Température, Humidité ou Vent), appelées variables prédictives.

Chaque nœud de l'arbre décrit la distribution de la variable Jouer à prédire. Dans le cas du premier nœud, la racine de l'arbre, nous constatons qu'il y a 14 observations dans notre fichier : 9 cas où une partie a eu lieu (`Jouer = oui`) et 5 où aucune partie n'a eu lieu (`Jouer = non`). Ce premier nœud a plusieurs fils construits en utilisant la variable `Ensoleillement` : le plus à gauche (`Ensoleillement = Soleil`) comporte 5 observations, le suivant (`Ensoleillement = couvert`) en comporte 4, et ainsi de suite.

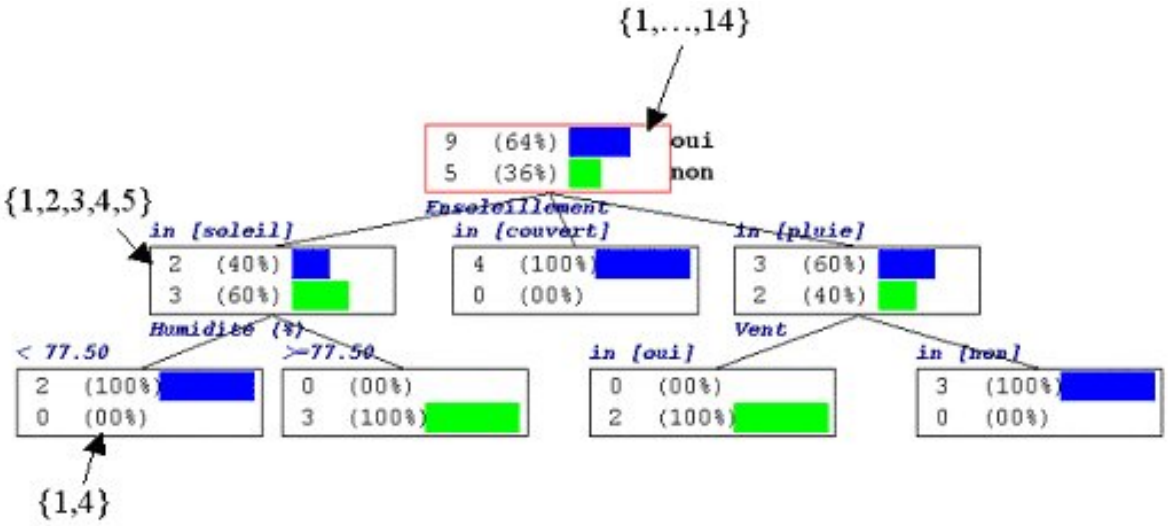


Figure 5.1 – Exemple d'arbre de décision

La suite de décisions continue jusqu'à ce que, dans l'idéal, les observations dans un nœud soient toutes « oui » ou toutes « non ». On dit alors que le nœud est `homogène`.

Le processus de décision s'arrête aux feuilles de l'arbre. Dans l'arbre ci-dessus (*cf.* Figure 5.1), toutes les feuilles sont homogènes, c'est-à-dire que les variables prédictives utilisées permettent de prédire complètement (sur ce fichier de données) si une partie va avoir lieu ou non. (Notons qu'il serait possible de construire l'arbre selon un ordre différent des variables de météo, par exemple en considérant l'humidité plutôt que l'ensoleillement à la première décision). L'arbre se lit intuitivement de haut en bas, ce qui se traduit en termes de règles logiques sans perte d'informations : par exemple, la feuille la plus à gauche se lit : « si `ensoleillement = soleil` et `humidité` $< 77.5\%$ alors `jouer = oui` ».

5.1.2 Principe de la segmentation

- Construire un **arbre** à l'aide de divisions successives d'un ensemble d'individus appartenant à un échantillon.
- Chaque **division** (ou **scission**) conduit à deux (ou plus) **noeuds** (ou **segments**) :
  - Le noeud divisé est appelé **noeud-parent**
  - Les noeuds générés par la division s'appellent **noeuds-enfants**.
- **Les noeuds contiennent des groupes d'individus** les plus **homogènes** possibles **par rapport à une variable à expliquer Y** nominale, ordinale ou quantitative.
- Les divisions s'opèrent à partir de **variables explicatives** (ou prédicteurs) $X_1, \cdots, X_j, \cdots, X_p$ qui peuvent être nominales, ordinales ou quantitatives.
- **Résultats obtenus sous la forme d'un arbre inversé**
  - La **racine** (en haut de l'arbre) représente l'échantillon total à segmenter
  - Les autres noeuds sont :
    - Soit des **noeuds intermédiaires** (encore divisibles)
    - Soit des **noeuds terminaux**

Remarque 5.1 — *L'ensemble des noeuds terminaux constitue une* ***partition*** *de*

l'échantillon initial en groupes.
- *Si la variable à expliquer Y est **qualitative**, on parle de **d'arbre de classement** ou **de discrimination par arbre**.*
- *Si la variable à expliquer Y est **quantitative**, on parle de **d'arbre de régression** ou **régression par arbre***
- *On parle d'**arbre binaire** si toutes les divisions conduisent à 2 noeuds (**divisions binaires**)*

5.2 Construction et représentation d'un arbre avec scikit-learn

5.2.1 Donéees

5.2.1.1 Importation et expertise des données

Nous traitons un exemple très simple dans un premier temps. Nous utilisons la base ultra - connue Breast Cancer Wisconsin. Nous cherchons à expliquer la variable « `classe` » décrivant la nature maligne (`malignant`) ou non (`begnin`) de cellules à partir de leurs caractéristiques (clump, ucellsize, ..., mitoses ; 9 variables numériques.)

```
# Chargement de la base
import pandas as pd
D = pd.read_excel("./donnee/breast.xlsx",sheet_name=0)
# Dimension du data frame
print(D.shape)
```

```
## (699, 10)
```

Nous disposons de 699 observations et 10 variables. Voici les premières lignes de notre dataset :

Table 5.1 – Premières lignes - Données Breast Cancer Wisconsin

clump	ucellsize	ucellshape	mgadhesion	sepics	bnuclei	bchromatin	normnucl	mitoses	classe
4	2	2	1	2	1	2	1	1	begnin
1	1	1	1	2	1	2	1	1	begnin
2	1	1	1	2	1	2	1	1	begnin
10	6	6	2	4	10	9	7	1	malignant
4	1	1	1	2	1	2	1	1	begnin
1	1	1	1	2	1	1	1	1	begnin

Nous affichons les informations sur le type des variables.

```
# Informations sur les variables
print(D.info())
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 699 entries, 0 to 698
## Data columns (total 10 columns):
##  #   Column      Non-Null Count  Dtype
```

```
## ---  ------      --------------  -----
##  0   clump       699 non-null    int64
##  1   ucellsize   699 non-null    int64
##  2   ucellshape  699 non-null    int64
##  3   mgadhesion  699 non-null    int64
##  4   sepics      699 non-null    int64
##  5   bnuclei     699 non-null    int64
##  6   bchromatin  699 non-null    int64
##  7   normnucl    699 non-null    int64
##  8   mitoses     699 non-null    int64
##  9   classe      699 non-null    object
## dtypes: int64(9), object(1)
## memory usage: 54.7+ KB
## None
```

La variable cible « `classe` » est la seule non - numérique, le type « object » lui est associé.

5.2.1.2 Distribution absolue et relative des classes

Nous affichons la fréquence absolue des classes...

```
# Vérifier la distribution abolue des classe
n_k= (D.classe.value_counts(normalize=False).to_frame("Eff.").reset_index())
```

Table 5.2 – Distribution absolue des classes

classe	effectifs
begnin	458
malignant	241

... puis relative.

```
# Distribution relative
p_k= (D.classe.value_counts(normalize=True).to_frame("prop.").reset_index())
```

Table 5.3 – Distribution relative des classes

classe	proportion
begnin	0.6552217
malignant	0.3447783

Ces informations sont importantes lorsque nous aurons à inpecter les résultats.

5.2.1.3 Partition en train set et test set

Nous souhaitons réserver 299 des observations pour le train set et 300 pour le test set, avec un échantillonnage stratifié (`stratify`) c'est-à-dire respectant les proportions des classes dans les deux sous-ensembles. Nous fixons (`randon_state =1`) pour que l'expérimentation soit reproductible.

```
# Subdiviser les données en Train et Test set
from sklearn.model_selection import train_test_split
DTrain,DTest=train_test_split(D,test_size=300,random_state=1,stratify=D.classe)
```

Nous vérifions les dimensions des données

```
# Vérification des dimensions
print(DTrain.shape)
```

```
## (399, 10)
```

```
print(DTest.shape)
```

```
## (300, 10)
```

Nous affichons les distributions relatives des classes en apprentissage.

```
# Vérification des distributions en apprentissage
pk_train = (DTrain.classe.value_counts(normalize=True)
                          .to_frame("proportion").reset_index())
```

Table 5.4 – Distribution relative sur le train set

classe	proportion
begnin	0.6541353
malignant	0.3458647

… et en test

```
# Vérification des distributions en test
pk_test = (DTest.classe.value_counts(normalize=True)
                        .to_frame("proportion").reset_index())
```

Table 5.5 – Distribution relative sur le test set

classe	proportion
begnin	0.6566667
malignant	0.3433333

Les proportions sont respectées.

5.2.2 Instanciation et modélisation

Nous instancions un arbre de décision DecisionTreeClassifier de la librairie Scikit-Learn avec deux paramètres : un sommet n'est pas segmenté s'il est composé de moins de 30 individus (`min_samples_split=30`) ; une segmentation est validée si et seulement si les feuilles générées comportement tous au moins 10 observations (`min_samples_leaf=10`).

```
# Instanciation de l'arbre
from sklearn.tree import DecisionTreeClassifier
model1 = DecisionTreeClassifier(min_samples_split=30,min_samples_leaf=10)
```

Nous lançons le processus de modélisation sur les données d'apprentissage en spécifiant la matrice (X) des variables prédictives et le vecteur (y) de la variable cible.

```
# Construction de l'arbre
model1.fit(X=DTrain.iloc[:,:-1], y=DTrain.classe)


## DecisionTreeClassifier(min_samples_leaf=10, min_samples_split=30)
```

La console affiche l'ensemble des paramètres utilisés lors de la modélisation.

5.2.2.1 Affichage graphique de l'arbre

Nous disposons d'une fonction dédiée à la génération de la représentation graphique directement dans la console. Cette fonction prend en paramètre l'arbre généré par le test set, la liste des noms des variables prédictives (`feature_names`), les sommets peuvent être coloriées selon la classe majoritaire (`filled = True`).

```
# Arbfre de décision
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree
fig, axe = plt.subplots(figsize = (16,6))
plot_tree(model1,feature_names = list(D.columns[:-1]),
          class_names=list(D.columns[-1]),filled=True)
plt.show()
```

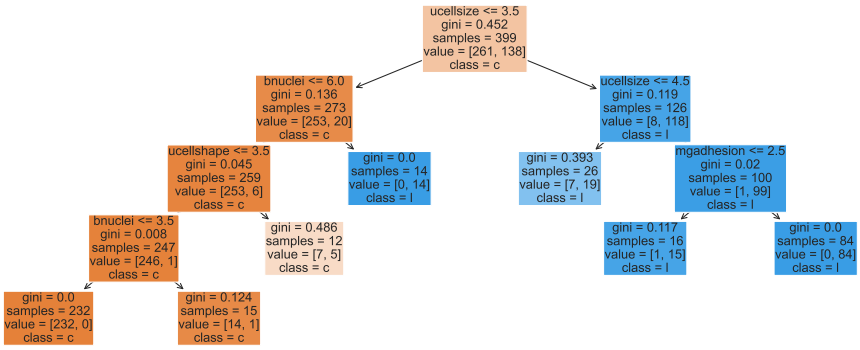


Figure 5.2 – Premier arbre de décision sur les données « breast »

Que lisons-nous ?

— L'arbre est composée de 7 feuilles (gini = 0.0). Il produit donc 7 règles prédictives matérialisées par les chemins partant de la racine aux feuilles.

— Nous observons l'effectif du train set sur la racine de l'arbre ($n_{racine} = n = samples = 399$) avec 251 « `begnin` » et 238 « `virginica` (dans l'odre alphabétique).
— Les sommets sont teintés (c'est le rôle de l'option `filled = True`) selon la classe majoritaire qu'ils portent, avec plus ou moins d'intensité selon la concentration des effectifs. Ici, visiblement, le bleu pour « `malignant` », l'orange à « `begnin`.
— La concentation des classes est calculée à l'aide de l'indice de Gini (on parle de mesure d'impureté (*impurity*) ou mesure de diversité). Pour la racine, nous avons ($K = 3$, nombre de modalités de la variable « `species` »), on aura :

$$G(Racine) = \sum_{k=1}^{K=2} \frac{n_{k,Racine}}{n_{Racine}} \left(1 - \frac{n_{k,Racine}}{n_{Racine}}\right) = \frac{261}{399}\left(1 - \frac{261}{399}\right) = 0.452 \quad (5.1)$$

— « `ucellsize` » est la variable de segmentation sur la racine, avec la condition qu'elle soit inférieur ou égal à 3.5 : ucellsize ≤ 3.5..
— La branche gauche de la racine correspond à la proposition vraie de la condition c'est - à - dire ucellsize ≤ 3.5. Nous lisons sur le sommet enfant qu'elle correspond (`samples = 273`) observations, avec 253 « `begnin` » et 20 « `malignant` ». Nous avons G(sommet) $= 0.136$. Plus la valeur de l'indice de Gini est faible, plus les classes sont concentrées sur un sommet.
— La branche droite de la racine correspond à la négation de la proposition c'est-à-dire ucellsize > 3.5, elle concerne 126 observations avec 8 « `begnin` » et 118 « `malignant` ».
— Lorsque les variables de segmentations sont quantitatives, il n'est pas rare qu'elles interviennent plusieurs fois dans l'arbre, mais avec des seuils de découpage différents. C'est la cas de variables « `ucellsize` » et `bnuclei` » ici.
— Nous pouvons simplifier l'arbre en retirant les feuilles issues du même père qui portent des conclusions identiques. En procédant ainsi de bas en haut (*bottom-up*), nous effectuons un processus (simplifié) de post - élagage qui permet de réduire la taille de l'arbre sans modifier en aucune manière ses propriétés prédictives. Suivant cette idée, nous devrions aboutir à un arbre avec 3 feuilles dans notre exemple, avec exactement le même comportement en classement.

5.2.2.2 Affichage sous forme de règles imbriquées de l'arbre

L'affichage graphique est sympathique mais devient peu lisible dès lors que la taille de l'arbre augmente. Scikit-learn propose une sortie alternance textuelle, sous la forme de règles imbriquées, à la manière de la surcharge de la fonction print() de rpart sous R.

Les paramètres de la fonctions sont identiques à celle de l'affichage graphique, mais il n'est plus question de colorier les sommets bien évidemment. Nous demandons à ce que les effectifs sur les feuilles soient spécifiées (`show_weights = True`).

```
# Affichage sous forme de règles
from sklearn.tree import export_text
tree_rules = export_text(model1,feature_names = list(D.columns[:-1]),
                         show_weights = True)
print(tree_rules)
```

```
## |--- ucellsize <= 3.50
## |   |--- bnuclei <= 6.00
```

```
## |   |   |--- ucellshape <= 3.50
## |   |   |   |--- bnuclei <= 3.50
## |   |   |   |   |--- weights: [232.00, 0.00] class: begnin
## |   |   |   |--- bnuclei >  3.50
## |   |   |   |   |--- weights: [14.00, 1.00] class: begnin
## |   |   |--- ucellshape >  3.50
## |   |   |   |--- weights: [7.00, 5.00] class: begnin
## |   |--- bnuclei >  6.00
## |   |   |--- weights: [0.00, 14.00] class: malignant
## |--- ucellsize >  3.50
## |   |--- ucellsize <= 4.50
## |   |   |--- weights: [7.00, 19.00] class: malignant
## |   |--- ucellsize >  4.50
## |   |   |--- mgadhesion <= 2.50
## |   |   |   |--- weights: [1.00, 15.00] class: malignant
## |   |   |--- mgadhesion >  2.50
## |   |   |   |--- weights: [0.00, 84.00] class: malignant
```

En faisant le parallèle avec l'arbre graphique, on distingue bien les successions de segmentations. Les effectifs sur les feuilles correspondent bien évidemment.

5.2.3 Importance des variables

Scikit-learn peut afficher l'importance des variables, en s'en tenant exclusivement à celles qui apparissent explicitement dans l'arbre. On récupère le champ feature_importance_ de l'arbre qu'on place dans un dataframe Pandas pour pouvoir afficher le scontributions des variables associées à leurs noms, et triées de manière décroissante.

```
# Importance
impvarModel = (pd.DataFrame({"variable" : D.columns[:-1],
                             "Importance" : model1.feature_importances_})
               .sort_values(by="Importance",ascending = False))
```

Table 5.6 – Importance des variables

	variable	Importance
1	ucellsize	0.8166268
5	bnuclei	0.1584766
2	ucellshape	0.0242434
3	mgadhesion	0.0006532
0	clump	0.0000000
4	sepics	0.0000000
6	bchromatin	0.0000000
7	normnucl	0.0000000
8	mitoses	0.0000000

Sans surprise, seules les variables qui apparaissent dans l'arbre présentent une valeur non - nulle « `ucellsize` » est la plus importante, puis viennent `bnuclei`, `ucellshape` et très marginalement `mgadhesion`. Comment sont calculées ces valeurs ?

5.2.3.1 Qualité globale de l'arbre

La qualité globale de l'arbre peut être quantifiée par la différence l'indice de Gini de la racine n°1, et la moyenne pondérée des Gini des L feuilles, chacune avec un effectif n_l :

$$\Delta_{\text{arbre}} = \text{G(Racine)} - \sum_{l=1}^{l=L} \frac{n_l}{n} G(l) \tag{5.2}$$

Dans notre arbre,

$$\Delta_{\text{arbre}} = 0.452 - \left(\frac{232}{399} \times 0.0 + \frac{15}{399} \times 0.124 + \cdots + \frac{16}{399} \times 0.117 + \frac{84}{399} \times 0.0 \right) = 0.40284608$$

Cette quantité peut être également exprimée sous la forme d'une sommepondérée des contributions locales de chaque opération de segmentation.

5.2.3.2 Contribution d'une variable apparaissant une fois

Pour mesurer la contribution d'une segmentation dans l'arbre, nous effectuons la différence entre l'indice de Gini du sommet à segmenter et la moyenne pondérée des Gini de ses feuilles. Cette différence étant pondérée par le poids du mommet traité.

Pour le sommet n°4 où la variable « `ucellshape` » intervient, et qui a généré les sommets n°8 et 9, nous avons :

$$\Delta_{\text{segmentation}} = \frac{259}{399} \times \left[0.045 - \left(\frac{247}{259} \times 0.008 + \frac{12}{259} \times 0.486 \right) \right] = 0.00976634$$

Elle correspond également à la contribution de la variable dans l'arbre si elle n'apparaît qu'une seule fois.

Cette valeur est ensuite ramenée à la qualité globale de l'arbre pour que la somme des contributions fasse 1. Dans notre cas :

$$\text{CTR}_{\text{ucellshape}} = \frac{0.00976634}{0.40284608} = 0.024243$$

Et c'est bien la valeur associée à « `ucellshape` » dans le tableau 5.6 proposé par Scikit-Learn.

5.2.3.3 Contribution d'une variable apparaissant plusieurs fois

Lorsqu'une variable apparaît plusieurs fois, nous additionnons les contributions des segmentations dans lesquelles elle intervient. Pour « `ucellsize` » par exemple, qui opère lors des partitions des sommets n°1 et 3, nous avons :

$$\text{CTR}_{\text{ucellsize}} = \frac{0.3289749 + 0.00695077}{0.40284608} = 0.816627$$

…comme nous l'indique le tableau 5.6 fourni par Scikit-Learn.

5.2.4 Evaluation en test

5.2.4.1 Prédiction en test

Pour évaluer les performances prédictives de l'arbre, on applique sur le test set composé de 300 observations. Nous avons une prédiction :

```
# Prédiction sur l'échantillon test
ypred1 = model1.predict(X = DTest.iloc[:,:-1])
#distribution des prédictions
import numpy as np
pred_dist = np.unique(ypred1, return_counts=True)
pred1 = pd.DataFrame({"classe" : pred_dist[0],"Eff." : pred_dist[1]})
```

Table 5.7 – Prédiction en test

classe	Eff.
begnin	199
malignant	101

La classe « `begnin` » a été assignée à 199 observations, 101 pour « `malignant` ».

5.2.4.2 score

Nous allons calculer le taux de reconnaissance (=score) de notre modèle

```
# Performance
score1 = model1.score(X = DTest.iloc[:,:-1],y=DTest.classe)
print("Score : %.4f" % (score1))
```

```
## Score : 0.9400
```

Avec ce modèle, on atteint un score de 94%. Et donc un taux d'erreur de 6%.

5.2.5 Modification des paramètres d'apprentissage

Notre arbre (*cf.* Figure 5.2) paraît surdimensionné. Nous avions remarqué notamment que plusieurs feuilles issues du même sommet père portaient des conclusions identiques.

Dans cette sous - section, nous introduisons un nouveau paramètre pour réduire la taille de l'arbre. Nous spécifions (`max_leaf_nodes=3`) c'est - à - dire nous souhaitons obtenir un arbre qui produit 3 règles au maximum [1].

```
# Modifier les paramètres d'apprentissage
model2 = DecisionTreeClassifier(min_samples_split=30,min_samples_leaf=10,
                                max_leaf_nodes=3)
# Construction de l'arbre
model2.fit(X=DTrain.iloc[:,:-1], y=DTrain.classe)
```

1. Dixit la documentation, l'outil effectue en priorité les segmentations qui maximisent les contributions.

```
## DecisionTreeClassifier(max_leaf_nodes=3, min_samples_leaf=10,
##                        min_samples_split=30)
```

Nous obtenons une nouvelle version de l'arbre de décision (*cf.* Figure 5.3) :

```
# Affichage graphique de l'arbre
fig, axe = plt.subplots(figsize=(16,6))
plot_tree(model2,feature_names = list(D.columns[:-1]),
          class_names=list(D.columns[-1]),filled=True)
plt.show()
```

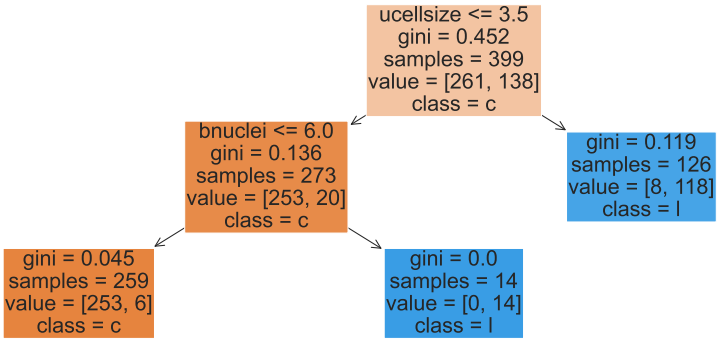


Figure 5.3 – Secodn rabre de décision sur les données « breast »

L'arbre est fortement simplifié, tout en maintenant ses qualités prédictives puisque nous obtenons le taux de reconnaissance :

```
# Performance
score2 = model2.score(X = DTest.iloc[:,:-1],y=DTest.classe)
print("Score : %.4f" % (score2))
```

```
## Score : 0.9400
```

6 Comparaison de méthodes

Sommaire

Le but de ce chapitre est de présenter des outils permettant de confronter des méthodes entre elles pour en déterminer celle qui est la meilleure face à un problème donné. Il s'agit notamment de : la matrice de confusion, la courbe ROC et la courbe LIFT. Ces outils sont utiles notamment lorsque la variable à prédire est de type catégorielle. Pour leur mise oeuvre, nous allons utiliser le dataset make_classification sur un échantillon de 1000 observations et deux classes pour la variable à prédire. On utilisera LogisticRegression pour la méthode à utiliser. Cependant, nous utiliserons également SGDClassifier pour la méthode de gradient descent pour la régression logistique.

6.1 Régression logistique avec LogisticRegression

En réalité, plusieurs outils permettent de lancer quelque chose qui ressemble à la régression logistique avec scikit-learn. Nous choisissons la classe LogisticRegression dont la désignation est la plus évidente. Autant statsmodels est d'obédience statistique, autant scikit-learn est imprégnée de la culture machine learning, tournée essentiellement vers l'efficacité des calculs et la performance prédictive. L'inférence statistique est ignorée ostensiblement. On ne dispose pas des indicateurs usuels de la régression logistique (tests de significativité, écarts-types des coefficients, etc.).

6.1.1 Le module LogisticRegression

Nous allons mettre en oeuvre une régression logistique sans pénalité, c'est-à-dire ni Ridge, ni Lasso. Nous lançons l'estimation des paramètres avec la fonction fit() qui prend en entrée les données en apprentissage avec la matrice des régresseurs et le vecteur des classes observées.

```
# Chargement du dataset
from sklearn.datasets import make_classification
```

```
# generate 2 class dataset
data_X, data_y = make_classification(n_samples=1000,n_classes=2,random_state=1)
```

On subdivise notre dataset en échantillon d'apprentissage et en échantillage test. On alloue 80% pour l'échnatillon d'apprentissage et 20% pour l'échantillon test tout en contrôlant l'aléa grâce à l'hyperparamètre random_state.

```
# Train set - Test set
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(data_X, data_y,test_size = 0.2,
                                                random_state = 5)
# Logistic Regression
from sklearn.linear_model import LogisticRegression
# Instanciation  - Entrainement
lrModel = LogisticRegression(penalty = "none").fit(Xtrain, ytrain)
```

On peut afficher les coefficients estimées, ceux des variables

```
# Coefficients
print(lrModel.coef_)
```

```
## [[ 0.60587462  0.03586102 -0.04161249  0.7023254  -0.14612392  0.04243093
##    0.11945616  0.02865749 -0.06483583  0.17307372  0.11312209 -0.11836518
##    1.59565671  0.10434099 -0.29328615 -0.01548132  0.01350762 -0.11341426
##    0.14727842 -0.0836404 ]]
```

et la constante.

```
# Intercept
print(lrModel.intercept_)
```

```
## [-0.00390534]
```

Remarque 6.1 *La régression logistique de scikit-learn s'appuie sur un algorithme différent de celui des logiciels de statistique. Les coefficients sont du même ordre mais différents. Ça ne veut pas dire que le modèle est moins performant en prédiction. On ne dispose pas des indicateurs usuels de la régression logistique (tests de significativité, écarts-types des coefficients, etc.)*

On peut à présent chercher la prédiction de notre test set et évaluer la performance de notre modèle.

```
# Prédiction sur le test set
lrModelPredict = lrModel.predict(Xtest)
```

6.1.2 Quelques indicateurs

6.1.2.1 Matrice de confusion

la matrice de confusion permet de confronter la vraie valeur avec la prédiction. Pour l'afficher, nous avons besoin de la fonction ConfusionMatrixDisplay qui se trouve dans la classe metrics de scikit-learn.

```
# Confusion Matrix
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm = confusion_matrix(ytest,lrModelPredict, labels=lrModel.classes_)
fig,axe = plt.subplots(figsize=(16,6))
ConfusionMatrixDisplay.from_estimator(lrModel,Xtest,ytest,ax=axe);
plt.show()
```

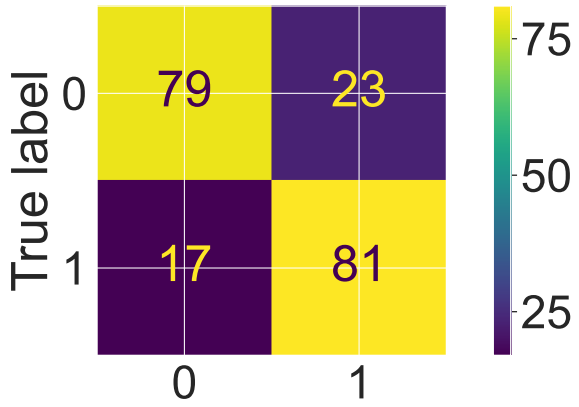


Figure 6.1 – Matrice de confusion avec LogisticRegresssion

On peut voir que pour les 102 observations de la classe 0 que l'on retrouve dans les données du test set, 79 ont été bien classés et 23 ont été rangés dans la classe 1. Egalement, pour les 98 observations de la classe 1, seulement 81 d'entre eux ont été classés dans la classe 1 et 17 ont été rangés dans la classe numéro 0.

6.1.2.2 Taux de succès - Taux d'erreur

On utilise la fonction **accuracy_score** de la classe **metrics**. Elle est similaire à l'attribut **.score** du modèle logistique.

```
# Taux de succès
from sklearn.metrics import accuracy_score
acc = accuracy_score(ytest,lrModelPredict)
print("Taux de succès : %.4f" % (acc))
```

```
## Taux de succès : 0.8000
```

```
# Taux d'erreur
err = 1.0 - acc
print("Taux d'erreur : %.4f" % (err))
```

```
## Taux d'erreur : 0.2000
```

On obtient un score de 80% sur l'échantillon test. Soit un taux d'erreur de 20%.

6.1.3 Validation croisée

A l'aide la fonction cross_val_score, on va tester s'il est possible d'améliorer les performances de notre modèle en utilisant de la validation croisée 10 blocs.

```
from sklearn.model_selection import cross_val_score
lrModelValidationScore = cross_val_score(lrModel, Xtrain, ytrain
                                         ,cv = 10, scoring = "accuracy")
# Moyenne
lrModelValidationScoreMean = lrModelValidationScore.mean()
print("Score par validation croisée : %.4f" % (lrModelValidationScoreMean))
```

```
## Score par validation croisée : 0.8425
```

La validation croisée à 10 blocs nous donne un score de 84%. Ce qui n'est pas mal. On peut encore améliore ce score en utilisant la fonction GridSearchCV.

Remarque 6.2 *Il existe une dépendance des algorithmes d'apprentissage aux paramètres. En effet, de nombreux algorithmes de machine learning repose sur des paramètres qui ne sont pas toujours évidents à déterminer pour obtenir les meilleurs performances sur un jeu de données à traiter. Exemple : Support Vector Machine.*

6.1.3.1 Support Vector Classification (SVC)

Les SVMs sont une famille d'algorithmes d'apprentissage automatique qui permettent de résoudre des problèmes tant de classification que de régression ou de détection d'anomalie. Ils sont connus pour leurs solides garanties théoriques, leur grande flexibilité ainsi que leur simplicité d'utilisation même sans grande connaissance de data mining.

```
#Support Vector Classification
from sklearn.svm import SVC
# Instanciation et entraînement
svcModel = SVC().fit(Xtrain, ytrain)
# Prediction sur l'échantillon test
svcModelPredict = svcModel.predict(Xtest)
# Matrice de confusion
cm = confusion_matrix(ytest,svcModelPredict,labels=svcModel.classes_)
fig,axe = plt.subplots(figsize=(16,6))
ConfusionMatrixDisplay.from_estimator(svcModel,Xtest,ytest,ax=axe);
plt.show()
```

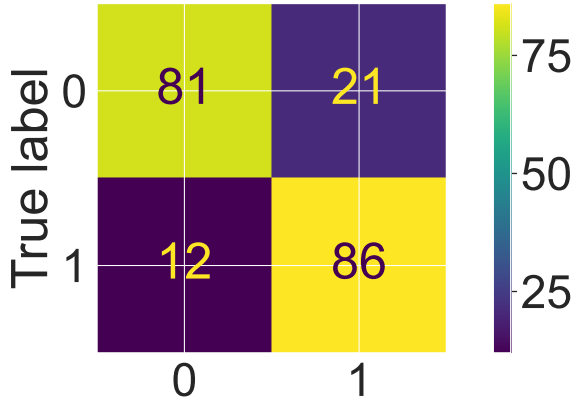


Figure 6.2 – Matrice de confusion avec SVC

On peut voir que pour les 102 observations de la classe 0 que l'on retrouve dans les données du test set81 ont été bien classés et 21 ont été rangés dans la classe 1. Egalement, pour les 98 observations de la classe 1, seulement 86 d'entre eux ont été classés dans la classe 1 et 12 ont été rangés dans la classe numéro 0.

```
# Score
svcModelScore = accuracy_score(ytest, svcModelPredict)
print("Score par svc : %.4f" % (svcModelScore))
```

```
## Score par svc : 0.8350
```

Avec SVC, on obtient un score de 84%. Ce score est moins que celui obtenu avec la validation croisée à 10 blocs, mais est nettement supérieur à celui du classifieur. Elle fait mieux que le classifieur par défaut.

6.1.3.2 La fonction gridsearchcv

Tout comme dans les chapitres précédents, on va créer un dictionnaire de paramètre à tester.

```
# Dictionnaire d'hyperparamètre
param_grid = {"C" : [0.1, 1.0, 10],
              "solver" : ["newton-cg","lbfgs", "liblinear",
                          "sag", "saga"]}
```

On peut à présent écrire le modèle et le lancer. accuracy sera le critère à utiliser pour sélectionner la meilleure configuration.

```
# Amélioration de la performance
from sklearn.model_selection import GridSearchCV
grid = GridSearchCV(LogisticRegression(), param_grid , cv = 10,
                    scoring = "accuracy")
# Lancement de la recherche
grid.fit(Xtrain, ytrain);
```

On affiche le meilleure paramétrage

```
# Meilleur parametrage
bestParams = grid.best_params_
print("Best params :", bestParams)
```

```
## Best params : {'C': 0.1, 'solver': 'newton-cg'}
```

... et le meilleur score obtenu

```
# Meilleure performance
best_score = grid.best_score_
print("Best score : %.4f" % (best_score))
```

```
## Best score : 0.8475
```

Le meilleur score est de 85%. On peut à présent sauvegarder le modèle et l'utiliser pour faire de la prédiction

```
# Sauvegarde du modele
gridModel = grid.best_estimator_
# Prédiction avec le model
gridModelPredict = gridModel.predict(Xtest)
```

On évalue sa performance sur le test set.

```
# Taux de succès en test
gridModelScore = accuracy_score(ytest, gridModelPredict)
print("Score sur gridsearchcv : %.4f" % (gridModelScore))
```

```
## Score sur gridsearchcv : 0.8050
```

On a un score de 80.5%. Pas très différent de celui obtenu avec le modèle de base.

```
# Matrice de confusion
fig,axe = plt.subplots(figsize=(16,6))
ConfusionMatrixDisplay.from_estimator(gridModel,Xtest,ytest,ax=axe);
plt.show()
```

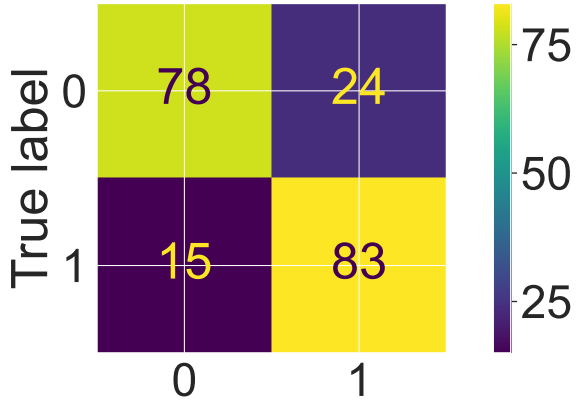


Figure 6.3 – Matrice de confusion avec gridsearchcv

6.1.4 Autres indicateurs

6.1.4.1 Sensibilité

Elle correspond au **taux de vrai positif** c'est-à-dire la probabilité que la valeur rédicte soit 1 sachant que la vraie valeur est 1 : $P(\hat{Y} = 1|Y = 1)$. Elle est calculée grâce à la fonction recall_score de la classe metrics.

```
# Sensibilité
from sklearn.metrics import recall_score
sen = recall_score(ytest, gridModelPredict, pos_label = 1)
print("Sensibilité : %.4f" % (sen))
```

```
## Sensibilité : 0.8469
```

La sensibilité est de 84.69%.

6.1.4.2 Spécificité

Elle estime la probabilité que la valeur prédicte soit 0 sachant que la vraie valeur est 0 : $P(\widehat{Y} = 0|Y = 0)$. On peut construire sa propre fonction d'évaluation et l'utiliser le cas échéant.

```
def specificity(y, ypred):
  # Matrice de confusion
  from sklearn.metrics import confusion_matrix
  mc = confusion_matrix(y,ypred)
  # "negative" est l'indice 0 dans la matrice
  import numpy as np
  res = mc[0,0]/np.sum(mc[0,:])
  # Retour
  return res
```

Il faut la rendre utilisable en utilisant make_scorer.

```
# La rendre utilisable
from sklearn.metrics import make_scorer
specificite = make_scorer(specificity, greater_is_better = True)
```

Une fois cela fait, on peut l'utiliser sur notre jeu de données.

```
# calcul de la spécificité
sp = specificite(gridModel, Xtest, ytest)
print("Spécificité : %.4f" % (sp))
```

```
## Spécificité : 0.7647
```

On obtient une spécificité de 76.47%. Soit un taux de faux positif de 23.53%.

6.1.4.3 Precision_recall_curve

Cet indicateur combine à la fois la précision et la sensibilité.

```
# calculate precision-recall curve
from sklearn.metrics import precision_recall_curve,PrecisionRecallDisplay
# Probabilités prédictes
lr_probs = gridModel.predict_proba(Xtest)
fig,axe = plt.subplots(figsize=(16,6))
prd = PrecisionRecallDisplay.from_estimator(gridModel,Xtest,ytest,ax=axe)
plt.show()
```

Figure 6.4 – Courbe de Rappel - Precision

6.1.4.4 f1_score

Le score F1 est la moyenne harmonique de la précision et du rappel. Il s'agit d'une bonne métrique équilibrée de faux positifs et de faux négatifs. Toutefois, il ne prend pas en compte les vrais négatifs.

```
# F- Measures
from sklearn.metrics import f1_score
lr_f1= f1_score(ytest, gridModelPredict)
print("Logistic: f1=%.3f" % (lr_f1))
```

```
## Logistic: f1=0.810
```

On obtient une **F-Measure** de 0.8098.

6.1.4.5 Rapport sur la qualité de prédiction

Les différents indicateurs calculés précédemment (sensibilité, f1_score, etc..) sont directement calculable grâce à la fonction classification_report de sklearn.

```
# Rapport sur la qualité de prédiction
from sklearn.metrics import classification_report
target_names = ['class 0', 'class 1']
print(classification_report(ytest,gridModelPredict,target_names=target_names))
```

```
##               precision    recall  f1-score   support
##
##      class 0       0.84      0.76      0.80       102
##      class 1       0.78      0.85      0.81        98
##
##     accuracy                           0.81       200
##    macro avg       0.81      0.81      0.80       200
## weighted avg       0.81      0.81      0.80       200
```

6.2 Régression logistique avec SGDClassifier

L'algorithme de gradient descent s'applique exactement de la même manière que pour la régression linéaire vue au chapitre 1. En plus, la dérivée de la **Fonction Coût** est la même aussi. On a :

- ***Modèle*** : $\sigma(X.\theta) = \dfrac{1}{1+e^{-X.\theta}}$
- ***Fonction Coût*** : $J(\theta) = \dfrac{-1}{m}\sum y \times \log\left(\sigma(X.\theta)\right) + (1-y) \times \log\left(1-\sigma(X.\theta)\right)$
- ***Gradient*** : $\dfrac{\partial J(\theta)}{\partial \theta} = \dfrac{1}{m}\sum\left(\sigma(X.\theta) - y\right).X = \dfrac{1}{m}X^T\left(\sigma(X.\theta) - y\right)$
- ***Gradient Descent*** : $\theta = \theta - \alpha \times \dfrac{\partial J(\theta)}{\partial \theta}$

6.2.1 Le classifieur SGDClassifier

6.2.1.1 Mise en oeuvre avec SGDClassifier

Nous allons commencer par générer des données aléatoires qui seront différentes de celles utilisées précédemment. On aura moins de données (100) et deux features.

```
# Chargement du dataset
from sklearn.datasets import make_classification
# Génération de données aléatoires: 100 exemples, 2 classes, 2 features x1 et x2
data_X, data_y = make_classification(n_samples=100, n_features =2, n_redundant = 0,
                                     n_informative = 1, n_clusters_per_class = 1,
                                     random_state=1)
```

On peut à présent visualiser notre jeu de données.

```
# Visualisation du dataset
fig, axe =plt.subplots(figsize=(16,6))
axe.scatter(data_X[:,0],data_X[:,1],marker="o",c=data_y,edgecolors="k");
axe.set(xlabel="$X_1$",ylabel="$X_2$");
plt.show()
```

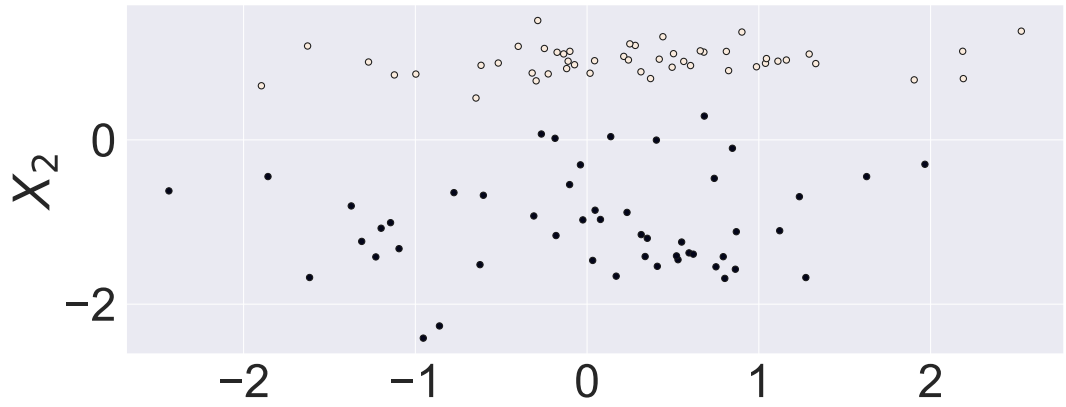


Figure 6.5 – Nuage de points des individus colorés par la classe d'appartenance

On voit effectivement que notre jeu de données n'est pas mélangé et que notre modèle pourrait avoir un score de 100% s'il est bien entraîné avec les bons hyperparamètres.

6.2.1.2 Modèle

Nous pouvons à présent créer un modèle en utilisant SGDClassifier avec les bons hyperparamètres.

```
# Modèle avec SGDClassifier
from sklearn.linear_model import SGDClassifier
model = SGDClassifier(max_iter = 1000, eta0 = 0.001, loss = "log")
model.fit(data_X, data_y)
```

```
## SGDClassifier(eta0=0.001, loss='log')
```

On évalue la performance de notre modèle.

```
# Score
modelscore = model.score(data_X, data_y)
print("Score : %.2f" % (modelscore))
```

```
## Score : 1.00
```

On obtient un score de 100%. Ce score était prévisible au regard de la structure de notre dataset fournie par le graphique.

6.2.1.3 Frontière de décision

Une fois le modèle entraîné, on peut afficher sa frontière de décision.

```
# Visualisation
import numpy as np
h = 0.02
colors = "bry"
x1_min, x1_max = data_X[:,0].min() - 1, data_X[:,0].max() + 1
x2_min, x2_max = data_X[:,1].min() - 1, data_X[:,1].max() + 1
x1x1, x2x2 = np.meshgrid(np.arange(x1_min, x1_max, h),
                         np.arange(x2_min, x2_max, h))
pred = model.predict(np.c_[x1x1.ravel(), x2x2.ravel()])
pred = pred.reshape(x1x1.shape)
# Fontière de décision
fig, axe = plt.subplots(figsize=(16,6))
cs = axe.contourf(x1x1, x2x2, pred, cmap = plt.cm.Paired)
for i, color in zip(model.classes_, colors):
    idx = np.where(data_y == i)
    axe.scatter(data_X[idx,0], data_X[idx,1], c = color,
                cmap = plt.cm.Paired, edgecolor = "black", s = 20)
axe.set(xlabel="$X_1$",ylabel="$X_2$");
plt.show()
```

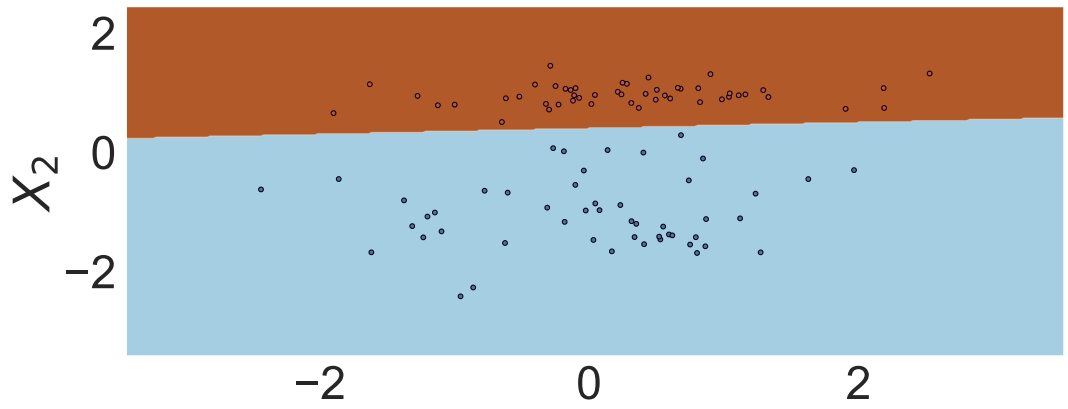


Figure 6.6 – Frontière de décision

6.2.2 Courbe ROC

La courbe ROC est un outil d'évaluation et de comparaison des modèles, dans le cas où la variable d'intérêt Y est qualitative binaire. Cette courbe permet de savoir si un modèle M1 sera toujours meilleur qu'un modèle M2 quelle que soit la matrice de coût. La courbe ROC est opérationnelle même dans le cas des distributions très déséquilibrées. C'est un outil graphique qui permet de visualiser les performances. Un seul coup d'oeil doit permettre de voir le(s) modèle(s) susceptible(s) de nous intéresser. Un indicateur synthétique associé = **aire sous la courbe ROC**.

6.2.2.1 Principe de la courbe ROC

$P(Y = 1|X = x) \geq P(Y = 0|X = x)$ équivaut à une règle d'affectation $P(Y = 1|X = x) \geq 0.5$ (seuil = 0.5). Cette règle d'affectation fournit une matrice de confusion $MC_{0.5}$, et donc 2 indicateurs $TVP_{0.5}$ et $TFP_{0.5}$. Si nous choisissons un autre seuil (0.6 par exemple), nous obtiendrons une matrice de confusion $MC_{0.6}$ et donc $TVP_{0.6}$ et $TFP_{0.6}$...etc.

L'idée de la courbe ROC est de faire varier le seuil de 1 à 0 et, pour chaque cas, calculer le TVP et le TFP que l'on reporte dans un graphique : en abscisse le TFP, en ordonnée le TVP.

Modèle au hasard = diagonale Modèle parfait = passe par les points (0,0), (0,1) et (1,1) car les scores des individus avec $Y = 1$ sont tous supérieurs aux scores des individus avec $Y = 0$.

Dans de nombreuses applications, la courbe ROC fournit des informations plus intéressantes sur la qualité de l'apprentissage que le simple taux d'erreur. C'est surtout vrai lorsque les classes sont très déséquilibrées, et lorsque le coût de mauvaise affectation est susceptible de modifications. Il faut néanmoins que l'on ait une classe cible (par exemple Y = 1) clairement identifiée et que la méthode d'apprentissage puisse fournir un SCORE proportionnel à $P(Y = 1|X)$.

Il existe plusieurs façons de faire la courbe ROC sous python. On peut utiliser la fonction RocCurveDisplay qui se trouve dans le module metrics de scikit-learn ou faire des manipulations avec la librairie matplotlib.

6.2.2.2 La fonction RocCurveDisplay

La fonction RocCurveDisplay permet la visualisation de la courbe ROC sous sklearn. Elle remplace la fonction plot_roc_curve obsolète.

```
# Courbe ROC
from sklearn.metrics import RocCurveDisplay
fig, axe = plt.subplots(figsize=(16,6))
RocCurveDisplay.from_estimator(gridModel, Xtest, ytest,ax=axe);
axe.plot([0, 1], [0, 1],'r--');
plt.show()
```

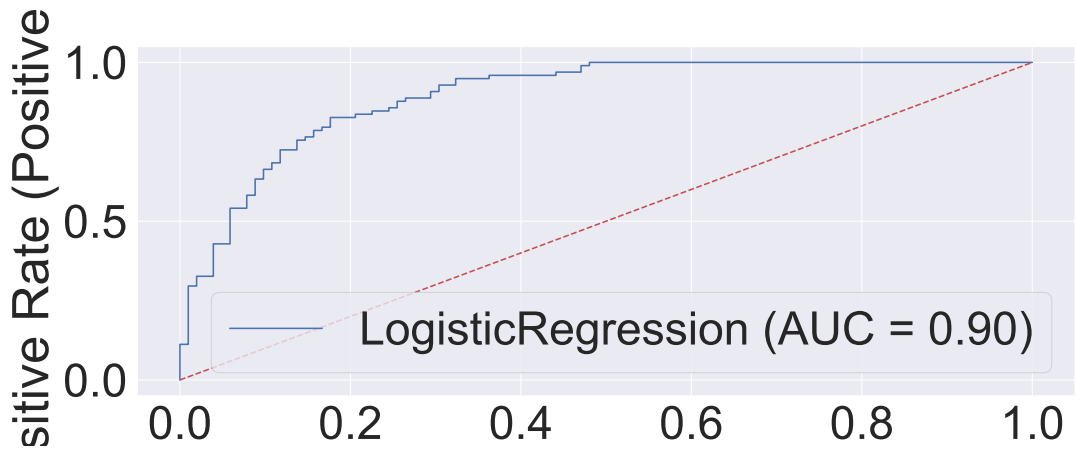


Figure 6.7 – Courbe ROC avec sklearn

6.2.2.3 Courbe ROC avec matplotlib

Pour tracer la courbe ROC avec matplotlib, on a besoin des **taux de faux positif** et des **taux de vrai positif**. Pour cela, on utilise la fonction roc_curve du module metrics de scikit-learn. Cette fonction renvoit 3 éléments : le taux de faux positif `fpr`, le taux de vrai positif `tpr` et le seuil `threshold`.

```
# Faux positifs - vrais positifs
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(ytest, lr_probs[:,1])
# ROC Curve - Matplotlib
fig, axe = plt.subplots(figsize=(16,6))
axe.plot(fpr, tpr, 'b', label = "LogisticRegression");
axe.plot([0, 1], [0, 1],'r--');
axe.set(xlabel='False Positive Rate',ylabel='True Positive Rate',
        xlim=[-0.05, 1.05],ylim=[-0.05, 1.05]);
axe.legend(loc = 'lower right');
axe.grid(visible=True);
plt.show()
```

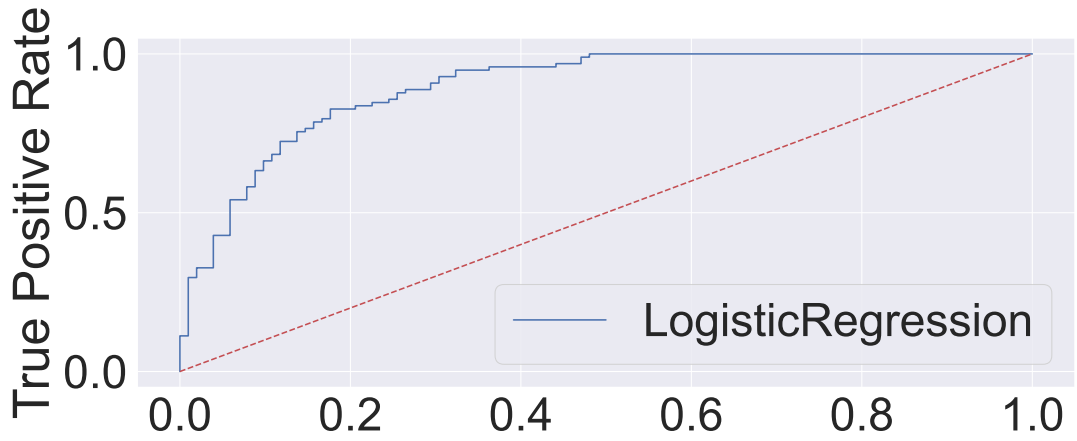


Figure 6.8 – Courbe ROC avec matplotlib

6.2.2.4 Aire sous la courbe ROC

Elle indique la probabilité pour que la fonction SCORE place un positif devant un négatif (dans le meilleur des cas AUC = 1 pour le modèle parfait). Si SCORE classe au hasard les individus (c'est-à-dire le modèle de prédiction ne sert à rien), AUC = 0.5 symbolisé par la diagonale principale dans le graphique.

Dans scikit-learn, on peut utiliser la fonction auc du module metrics et y introduire le taux de faux positif et le taux de vrai positif.

```
# Aire sous la courbe
from sklearn.metrics import auc
roc_auc1 = auc(fpr, tpr)
print("ROC AUC score : %.2f" % (roc_auc1))
```

```
## ROC AUC score : 0.90
```

On obtient 0.9.

On peut aussi utiliser la fonction roc_auc_score du même module qui est plutôt appliquée sur le target de l'échantillon test et les probabilités prédictes sur les features de cet échantillon test.

```
# Aire sous la courbe
from sklearn.metrics import roc_auc_score
roc_auc2 = roc_auc_score(ytest, lr_probs[:, 1])
print("ROC AUC score : %.2f" % (roc_auc2))
```

```
## ROC AUC score : 0.90
```

On obtient la même valeur, soit 0.9.

6.2.3 Courbe LIFT

Cette courbe représente la proportion de vrais positifs en fonction des individus sélectionnés, lorsqu'on fait varier le seuil. Sa forme dépend du taux de positif a priori. Elle a même ordonnée que la courbe ROC, mais une abscisse généralement plus grande. La courbe LIFT est généralement sous la courbe ROC.

On peut, soit utiliser la fonction plot_lift_curve de la librairie scikitplot de python afin de pouvoir avoir une une représentation graphique de la courbe LIFT, soit construire la courbe à l'aide des librairies numpy, pandas et matplotlib.

6.2.3.1 La fonction plot_lift_curve

L'avantage avec cette fonction est qu'elle fournit directement sur le même graphique les courbes LIFT des différentes classes contenues dans le target.

```
# LIFT curve
from scikitplot.metrics import plot_lift_curve
fig, axe = plt.subplots(figsize=(16,6))
plot_lift_curve(ytest, lr_probs,ax=axe);
plt.show()
```

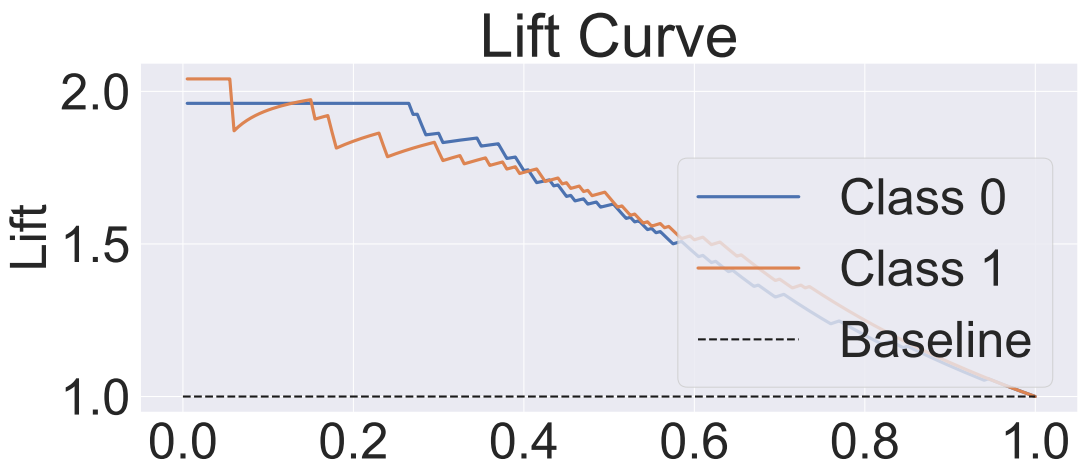


Figure 6.9 – Courbe LIFT avec plot_lift_curve

6.2.3.2 Aire sous la courbe LIFT

Il existe un lien entre les aires sous les courbes LIFT et ROC. L'aire sous la courbe LIFT, notée AUL, s'exprime simplement à priori par :

$$AUL = \frac{p}{2} + (1-p)AUC \tag{6.1}$$

où p = Proba(G1) = probabilité a prioride l'évèvement Y = 1 dans la population.

```
# Aire sous la courbe LIFT
#Positif dans la population
npos = np.sum(data_y)
# Taille de l'échantillon total
npop = ytrain.shape[0]
# Proportion
p = npos/npop
# Aire AUL
lift_aul = (p/2) + (1-p)*roc_auc1
print("LIFT AUL score : %.2f" % (lift_aul))


## LIFT AUL score : 0.87
```

On obtient une aire de 0.87.

7 Analyse en Composantes Principales

Sommaire

Introduite par Harold Hotelling Hotelling (1933) suivant des idées avancées par Karl Pearson dès 1901, l'analyse en composantes principales, en abrégé ACP (PCA, *Principal Component Analysis*) est une méthode d'analyse exploratoire des données qui s'applique à des tableaux croisant des individus et des variables quantitatives, appelés de façon concise tableaux Individus $\times$ Variables quantitatives. C'est une technique qui permet d'obtenir une carte des unités d'observations (individus, ménages, entreprises, etc...) en fonction de leur proximité et une carte des variables en fonction de leur corrélation au sein d'un tableau de mesures quantitatives. Elle cherche à repérer une structuration de la population par rapport à un ensemble de variables et les interactions entre ces variables.

L'objectif de l'analyse en composantes principales est de : repérer les groupes d'individus « semblables » vis-à-vis de l'ensemble des variables ; relever des différences entre individus ou groupes d'individus relativement à l'ensemble des variables ; mettre en évidence les individus dont les comportements sont « atypiques » vis-à-vis de l'ensemble des variables ; rechercher si l'information contenue dans le tableau brut ne pourrait pas être obtenue avec un nombre petit de variables, ces dernières pouvant être différentes des variables d'origine

Nous pouvons distinguer trois variantes de l'analyse en composantes principales :

1. **L'ACP générale**

   La recherche des directions de variance maximale est faite sans transformer les données. On travaille donc directement sur le tableau des données brutes. Cette variante est utilisée très rarement, essentiellement pour tenir compte du zéro naturel de certaines variables.
2. **L'ACP centrée**

   Les variables sont d'abord « centrées », c'est-à-dire la moyenne de chaque variable est soustraite pour chaque observation de la valeur de cette variable. Cette variante est utilisée lorsque les variables initiales sont directement comparables (de même nature, intervalles de variation comparables)
3. **L'ACP normée**

Les variables sont à la fois « centrées » et « réduites ». La réduction (après centrage) consiste à diviser pour chaque observation la valeur de cette variable par son écart-type. Chaque variable possède une moyenne nulle et un écart-type unitaire. Cette variante, la plus fréquemment rencontrée, est employée lorsque les variables quantitatives sont de nature différente (par exemple, distances, poids, durées, etc...) ou présentent des intervalles de variation très différentes.

7.1 Données et problématique de l'étude

7.1.1 Description des données

L'analyse en composantes principales s'intéresse à des tableaux de données rectangulaires de mesures X avec en lignes des individus et en colonnes des variables qui sont de nature quantitative :

Table 7.1 – Tableau de données en ACP

Individu \ Variable	1	k	K
i		x_{ik}	

Ce tableau de mesures décrit I individus à l'aide de K variables. x_{ik} est la valeur prise par l'individu i pour la variable k. On suppose généralement que $I > K$. Chaque individu étant décrit dans le tableau par une ligne de K chiffres, il peut donc être représenté par un point dans un espace à K dimensions (espace direct). De même, chaque variable étant traduite dans le tableau par une colonne de I chiffres, elle peut donc être représentée par un point dans un espace à I dimensions (espace dual).

Pour une variable k, on a : $x_{\bullet k} = (x_{1k}, ..., x_{Ik})$ un vecteur de dimension I, donc $x_{\bullet k} \in \mathbb{R}^I$. Pour chaque individu i, on a : $x_{i\bullet} = (x_{i1}, ..., x_{iK})$, un vecteur de dimension K, donc $x_{i\bullet} \in \mathbb{R}^K$.

Pour une variable k, on note

$$\overline{x}_k = \frac{1}{I} \sum_{i=1}^{i=I} x_{ik} \tag{7.1}$$

sa moyenne et

$$\sigma_k = \sqrt{\frac{1}{I} \sum_{i=1}^{i=I} (x_{ik} - \overline{x}_k)^2} \tag{7.2}$$

son écart - type.

Des tableaux de données de ce type avec des individus en lignes et des variables en colonnes, on en trouve dans de très nombreux domaines d'application tels que l'économie, la biologie, le marketing, etc...

7.1.2 Mise en œuvre de l'ACP avec scientisttools

Pour illustrer ce chapitre, nous allons prendre un exemple de données pprovenant de l'ouvrage de Saporta (2006) qui fait référence en analyse des données. Il s'agit de résumer l'information contenue dans un fichier décrivant $(n = 10)$ véhicules à l'aide de $(p = 6)$ variables.

```
# Chargement
import pandas as pd
X = pd.read_excel("./donnee/autos_acp_pour_python.xlsx",sheet_name=0,header=0,
                  index_col=0)
```

Table 7.2 – Tableau des données acrtives

	CYL	PUISS	LONG	LARG	POIDS	V_MAX
Alfasud TI	1350	79	393	161	870	165
Audi 100	1588	85	468	177	1110	160
Simca 1300	1294	68	424	168	1050	152
Citroen GS Club	1222	59	412	161	930	151
Fiat 132	1585	98	439	164	1105	165
Lancia Beta	1297	82	429	169	1080	160
Peugeot 504	1796	79	449	169	1160	154
Renault 16 TL	1565	55	424	163	1010	140
Renault 30	2664	128	452	173	1320	180
Toyota Corolla	1166	55	399	157	815	140
Alfetta-1.66	1570	109	428	162	1060	175
Princess-1800	1798	82	445	172	1160	158
Datsun-200L	1998	115	469	169	1370	160
Taunus-2000	1993	98	438	170	1080	167
Rancho	1442	80	431	166	1129	144
Mazda-9295	1769	83	440	165	1095	165
Opel-Rekord	1979	100	459	173	1120	173
Lada-1300	1294	68	404	161	955	140

Nous affichons la dimension de la matrice.

```
# Dimension
print(X.shape)
```

```
## (18, 6)
```

Nous allons mettre en œuvre une analyse en composantes principales en utilisant la librairie scientisttools développée par Duvérier DJIFACK ZEBAZE.

7.1.2.1 Instanciation et lancement des calculs

Il faut instancier l'objet PCA dans un premier temps :

```
# Instanciation
from scientisttools.decomposition import PCA
pca =PCA(normalize=True,row_labels=X.index,col_labels=X.columns,parallelize=False)
pca.fit(X)
```

```
## PCA(col_labels=Index(['CYL', 'PUISS', 'LONG', 'LARG', 'POIDS', 'V_MAX'], dtype='object'),
##     row_labels=Index(['Alfasud TI', 'Audi 100', 'Simca 1300', 'Citroen GS Club', 'Fiat 13
##        'Lancia Beta', 'Peugeot 504', 'Renault 16 TL', 'Renault 30',
##        'Toyota Corolla', 'Alfetta-1.66', 'Princess-1800', 'Datsun-200L',
##        'Taunus-2000', 'Rancho', 'Mazda-9295', 'Opel-Rekord', 'Lada-1300'],
##       dtype='object', name='Modele'))
```

```
# Résumé des informations
from scientisttools.extractfactor import summaryPCA
summaryPCA(pca)
```

```
##                      Principal Component Analysis - Results
##
## Importance of components
##                            Dim.1   Dim.2   Dim.3   Dim.4   Dim.5    Dim.6
## Variance                   4.421   0.856   0.373   0.214   0.093    0.043
## Difference                 3.565   0.483   0.159   0.121   0.050      NaN
## % of var.                 73.681  14.268   6.218   3.565   1.547    0.722
## Cumulative of % of var.   73.681  87.949  94.166  97.732  99.278  100.000
##
## Individuals (the 10 first)
##
##                  d(i,G)   p(i)  I(i,G)  Dim.1  ...   cos2  Dim.3     ctr   cos2
## Modele                                          ...
## Alfasud TI        2.868  0.056   0.457 -2.139  ...  0.388 -0.572   4.870  0.040
## Audi 100          2.583  0.056   0.371  1.561  ...  0.349 -1.315  25.762  0.259
## Simca 1300        1.469  0.056   0.120 -1.119  ...  0.211 -0.457   3.104  0.097
## Citroen GS Club   2.604  0.056   0.377 -2.574  ...  0.002 -0.149   0.329  0.003
## Fiat 132          1.081  0.056   0.065  0.428  ...  0.414  0.193   0.556  0.032
## Lancia Beta       1.065  0.056   0.063 -0.304  ...  0.034 -0.676   6.801  0.402
## Peugeot 504       1.230  0.056   0.084  0.684  ...  0.575  0.257   0.982  0.044
## Renault 16 TL     2.374  0.056   0.313 -1.948  ...  0.171  0.620   5.716  0.068
## Renault 30        4.668  0.056   1.211  4.410  ...  0.052  0.594   5.246  0.016
## Toyota Corolla    4.036  0.056   0.905 -3.986  ...  0.003  0.303   1.368  0.006
##
## [10 rows x 12 columns]
##
## Continues variables
##
##         Dim.1     ctr   cos2  Dim.2     ctr   cos2  Dim.3     ctr   cos2
## CYL     0.893  18.057  0.798 -0.115   1.542  0.013  0.216  12.504  0.047
## PUISS   0.887  17.791  0.787 -0.385  17.287  0.148  0.113   3.420  0.013
## LONG    0.886  17.763  0.785  0.381  16.959  0.145 -0.041   0.457  0.002
## LARG    0.814  14.971  0.662  0.413  19.899  0.170 -0.369  36.587  0.136
## POIDS   0.905  18.534  0.819  0.225   5.889  0.050  0.296  23.464  0.088
## V_MAX   0.755  12.884  0.570 -0.574  38.423  0.329 -0.297  23.568  0.088
```

7.1.2.2 Valeurs propres et scree plot

L'objet « pca » retroune les valeurs propres grâce à l'attribut `eig_`. Nous pouvons également utiliser la fonction get_eig pour avori un data frame.

```
# Valeurs propres
from scientisttools.extractfactor import get_eig
eig = get_eig(pca)
```

Table 7.3 – Valeurs propres

	eigenvalue	difference	proportion	cumulative
Dim.1	4.4209	3.5648	73.6810	73.6810
Dim.2	0.8561	0.4830	14.2677	87.9487
Dim.3	0.3731	0.1591	6.2178	94.1664
Dim.4	0.2139	0.1211	3.5654	97.7318
Dim.5	0.0928	0.0495	1.5467	99.2785
Dim.6	0.0433	NaN	0.7215	100.0000

La première composantes accapare 73.68% de l'information disponible. Il y a un for « effet taille » dans nos données. Nous disposons de 87.95% avec les deux premiers facteurs.

Scientisttools dispose d'outils permettant de construire le graphique « `Scree plot` » (éboulis des valeurs propres) (*cf.* Figure 7.1)

```
# Scree plot
from plotnine import *
from scientisttools.ggplot import fviz_screeplot
p = fviz_screeplot(pca,choice="eigenvalue")
print(p)
```

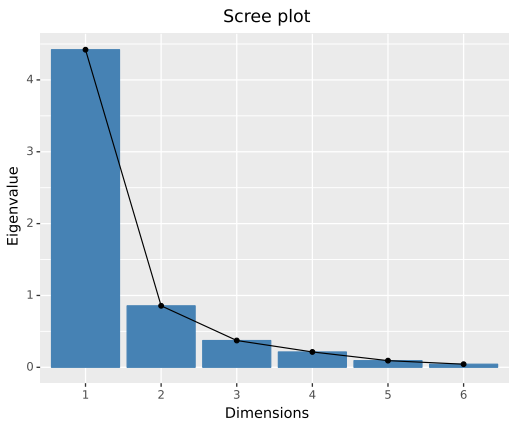


Figure 7.1 – Scree plot

Le graphique du cumul de variance restituée selon lenombre de facteurs peut être intéressant également (*cf.* Figure 7.2)

```
# Cumul de variance expliquée
import numpy as np
fig, axe = plt.subplots(figsize=(16,5))
p = X.shape[1]
axe.plot(np.arange(1,p+1),pca.eig_[3],color="blue");
axe.set(xlabel="Dimension",ylabel="Cumsm explained variance ratio");
plt.show()
```

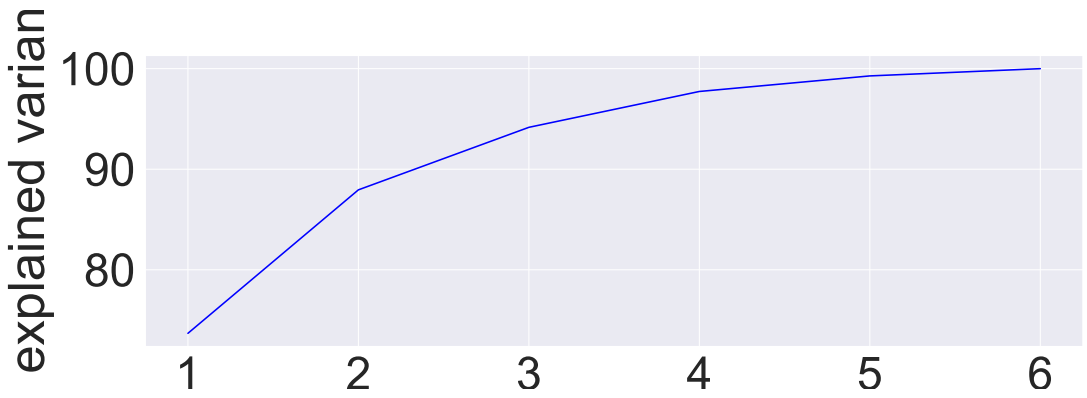


Figure 7.2 – Explained variance vs. Dimensions

7.1.2.3 Détermination du nombre de facteur à retenir

Les cassures dans les graphiques ci - dessu (Figure 7.1, Figure 7.2) sont souvent éviquées (règle du coude) pour identifier le nombre de facteurs $\widehat{K}$ à retenir.

```
# Représentation graphique - valeurs propres (Kaiser - Guttman)
p = fviz_screeplot(pca,choice="eigenvalue",add_kaiser=True)
print(p)
```

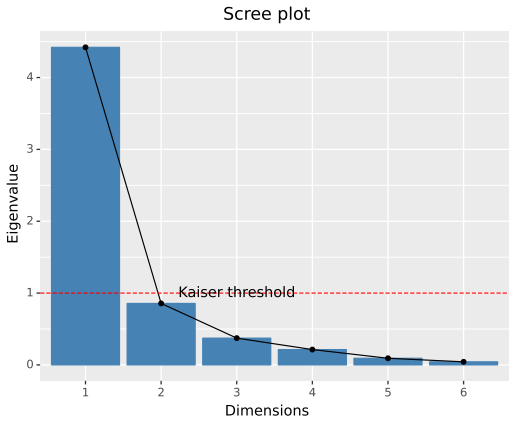


Figure 7.3 – Scree plot avec règle de Kaiser

La solution $\widehat{K} = 2$ semble s'imposer ici.

D'autres pistes existent pour répondre à cette question toujourss délicate qui conditionne l'interprétation de l'ACP, notammentle « test des bâtons braisés » de Frontier (1976) et Legendre and Legendre (1984). Les seuils sont défins par :

$$b_\alpha = \sum_{k=\alpha}^{k=K} \frac{1}{k}, \quad \forall \alpha = 1, \dots, K \tag{7.3}$$

Le facteur k est validé si $\lambda_k > b_k$, où λ_k est la valeur propre associée à l'axe k.

Scientisttools retourne l'attribut `broken_stick_threshold_` pour les seuils b_k.

```
# Seuil du test des bâtons brisés
bk = pd.DataFrame({"Dimension" : pca.dim_index_,
                   "Val. Propre" : pca.eig_[0],
                   "seuil" : pca.broken_stick_threshold_})
```

Table 7.4 – Seuil du test

Dimension	Val. Propre	seuil
Dim.1	4.4208581	2.4500000
Dim.2	0.8560623	1.4500000
Dim.3	0.3730661	0.9500000
Dim.4	0.2139221	0.6166667
Dim.5	0.0928012	0.3666667
Dim.6	0.0432903	0.1666667

Nous pouvons visualiser cela

```
# Représentation graphique - valeurs propres (tests des bâtons brisés)
# Scree plot
import matplotlib.pyplot as plt
from scientisttools.pyplot import plot_eigenvalues
fig, axe  =plt.subplots(figsize=(16,5))
plot_eigenvalues(pca,choice="eigenvalue",add_broken_stick=True,ax=axe)
plt.show()
```

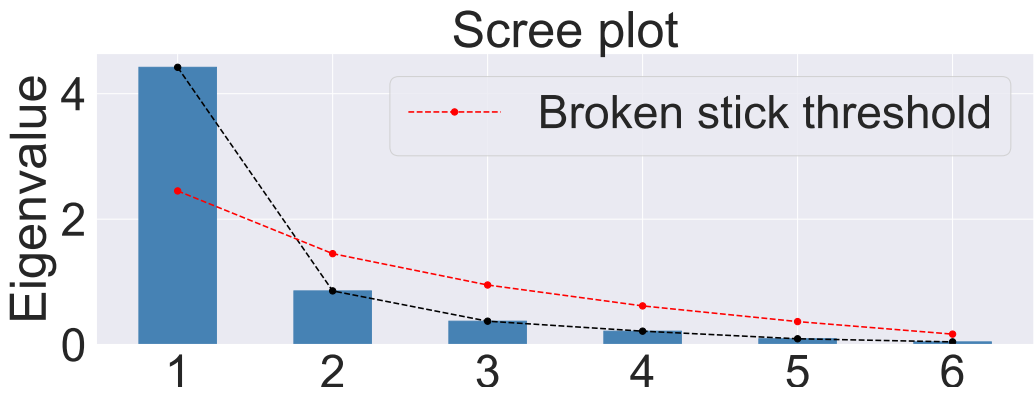


Figure 7.4 – Scree plot avec règle des brâtons brisés

Avec cette procédure, seul le premier facteur est valide. Le cercle des coorélations que nous construirons par la suite (*cf.* Figure 7.12) semble aller dans le même sens.

7.2 Outils pour l'interprétation

7.2.1 Représentation des individus

La fonction get_pca de scientisttools permet d'extraire des informations (ccordonnées,cosinus carré, etc...) aussi bien sur les individus que les variances. Nous fixons `choice = "row"`.

```
# Informations sur les individus
from scientisttools.extractfactor import get_pca
row = get_pca(pca,choice="row")
print(row.keys())
```

```
## dict_keys(['coord', 'cos2', 'contrib', 'infos'])
```

```
# Informations sur les individus - Distance, poids et inertie
row_infos = row["infos"]
print(row_infos)
```

```
##                    d(i,G)      p(i)    I(i,G)
## Modele
## Alfasud TI       2.867957  0.055556  0.456954
## Audi 100         2.583361  0.055556  0.370764
## Simca 1300       1.469465  0.055556  0.119963
## Citroen GS Club  2.603871  0.055556  0.376675
## Fiat 132         1.081261  0.055556  0.064951
## Lancia Beta      1.065340  0.055556  0.063053
## Peugeot 504      1.229956  0.055556  0.084044
## Renault 16 TL    2.374200  0.055556  0.313157
## Renault 30       4.667939  0.055556  1.210537
## Toyota Corolla   4.036105  0.055556  0.905008
## Alfetta-1.66     2.111106  0.055556  0.247598
## Princess-1800    1.397323  0.055556  0.108473
## Datsun-200L      3.333560  0.055556  0.617368
## Taunus-2000      1.566201  0.055556  0.136277
## Rancho           1.401204  0.055556  0.109076
## Mazda-9295       0.827358  0.055556  0.038029
## Opel-Rekord      2.466398  0.055556  0.337951
## Lada-1300        2.814640  0.055556  0.440122
```

7.2.1.1 Coordonnées factorielles

les coordonnées factorielles permettent d'avoir les positions des observations dans le plan factoriel.

```
# Coordonnées factorielles des individus
row_coord = row["coord"]
print(row_coord.iloc[:,:2])
```

```
##                     Dim.1     Dim.2
## Modele
## Alfasud TI      -2.138924 -1.785681
## Audi 100         1.561459  1.527040
## Simca 1300      -1.119385  0.674505
## Citroen GS Club -2.573742 -0.112884
## Fiat 132         0.427855 -0.695567
## Lancia Beta     -0.304238  0.196149
```

```
## Peugeot 504       0.683928  0.933057
## Renault 16 TL    -1.948493  0.980448
## Renault 30        4.409735 -1.063633
## Toyota Corolla   -3.985782 -0.236240
## Alfetta-1.66      0.437658 -1.912448
## Princess-1800     1.018175  0.841712
## Datsun-200L       2.941080  0.559175
## Taunus-2000       1.314880 -0.486522
## Rancho           -0.691111  0.897721
## Mazda-9295        0.385709 -0.356185
## Opel-Rekord       2.289768 -0.104345
## Lada-1300        -2.708574  0.143699
```

Nous les positionnnons dans le premier plan factoriel avec leurs labels pour situer et comprendre les proximités entre les véhicules.

```
# Positionnement des individus
from scientisttools.ggplot import fviz_pca_ind
p = fviz_pca_ind(pca,repel=True,text_type="text")+xlim(-6,6)
print(p)
```

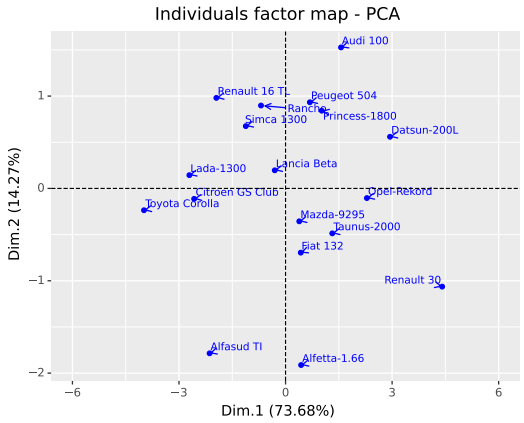


Figure 7.5 – Représentation des individus dans le premier plan factoriel

7.2.1.2 Qualité de représentation - les cosinus carré

Les cosinus carrés représentent la qualité de la représentation.

```
# Coordonnées factorielles des individus
row_cos2 = row["cos2"]
print(row_cos2.iloc[:,:2])
```

```
##                    Dim.1     Dim.2
## Modele
## Alfasud TI      0.556218  0.387670
## Audi 100        0.365334  0.349406
## Simca 1300      0.580284  0.210694
```

```
## Citroen GS Club  0.976992  0.001879
## Fiat 132         0.156579  0.413826
## Lancia Beta      0.081555  0.033900
## Peugeot 504      0.309202  0.575488
## Renault 16 TL    0.673539  0.170535
## Renault 30       0.892431  0.051920
## Toyota Corolla   0.975219  0.003426
## Alfetta-1.66     0.042978  0.820652
## Princess-1800    0.530947  0.362855
## Datsun-200L      0.778390  0.028137
## Taunus-2000      0.704819  0.096496
## Rancho           0.243273  0.410469
## Mazda-9295       0.217336  0.185337
## Opel-Rekord      0.861900  0.001790
## Lada-1300        0.926052  0.002607
```

Nous pouvons coloriés les individus par rapport au cosinus carré

```
# Représentation des individus - colorés par le cosinus carré
p = fviz_pca_ind(pca,color="cos2",repel=True,text_type="text",
                 gradient_cols = ("#00AFBB", "#E7B800", "#FC4E07"))+xlim(-6,6)
p = p + theme(legend_position=(0.2,0.73),legend_direction="vertical")
print(p)
```

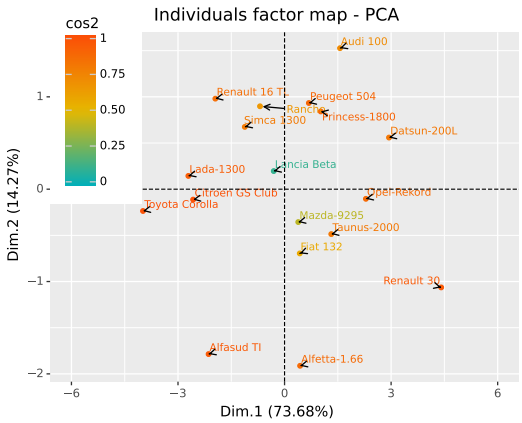


Figure 7.6 – Représentation des individus dans le premier plan factoriel coloré du cosinus carré

Nous pouvons également visualiser les individus les plus représentatifs sur l'axe 1

```
# Qualité de représentation axe 1
from scientisttools.ggplot import fviz_cosines
p = fviz_cosines(pca,choice="ind")
print(p)
```

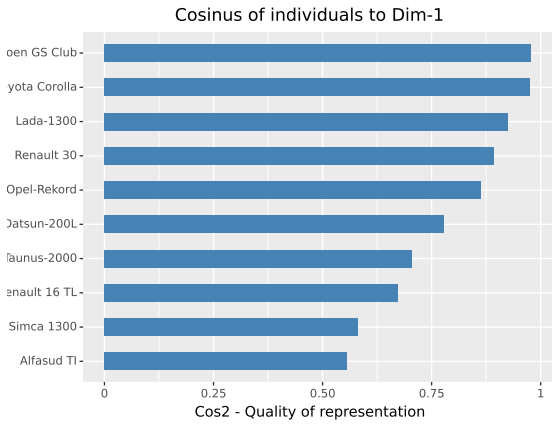


Figure 7.7 – Qualité de représentation des individus sur l'axe 1

... et sur l'axe 2

```
# Qualité de représentation axe 2
p = fviz_cosines(pca,choice="ind",axis=1)
print(p)
```

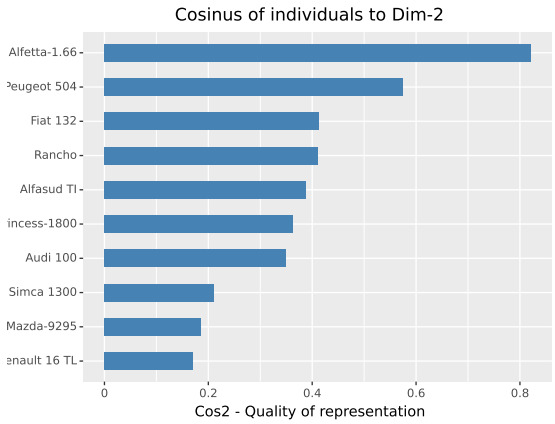


Figure 7.8 – Qualité de représentation des individus sur l'axe 2

Par défaut, la fonction retourne les 10 observations les plus représentées.

7.2.1.3 Contribution des individus aux axes

Elles permettent de déterliner les individus qui pèsent le plus dans la définition de chaque facteur.

```
# Contribution des individus
row_contrib = row["contrib"]
print(row_contrib.iloc[:,:2])
```

```
##                       Dim.1      Dim.2
## Modele
```

```
## Alfasud TI          5.749254  20.693307
## Audi 100            3.063951  15.132933
## Simca 1300          1.574636   2.952519
## Citroen GS Club     8.324360   0.082697
## Fiat 132            0.230046   3.139789
## Lancia Beta         0.116318   0.249686
## Peugeot 504         0.587817   5.649867
## Renault 16 TL       4.771099   6.238372
## Renault 30         24.436884   7.341856
## Toyota Corolla     19.964025   0.362185
## Alfetta-1.66        0.240708  23.735669
## Princess-1800       1.302765   4.597791
## Datsun-200L        10.870129   2.029163
## Taunus-2000         2.172668   1.536130
## Rancho              0.600229   5.230043
## Mazda-9295          0.186956   0.823327
## Opel-Rekord         6.588764   0.070658
## Lada-1300           9.219391   0.134007
```

Nous pouvons coloriés les individus par rapport au cosinus carré

```
# Représentation des individus - colorés par les contributions
p = fviz_pca_ind(pca, color = "contrib", text_type="text",repel = True,
                 gradient_cols = ("#00AFBB", "#E7B800", "#FC4E07"))+xlim(-6,6)
p = p + theme(legend_position=(0.2,0.73),legend_direction="vertical")
print(p)
```

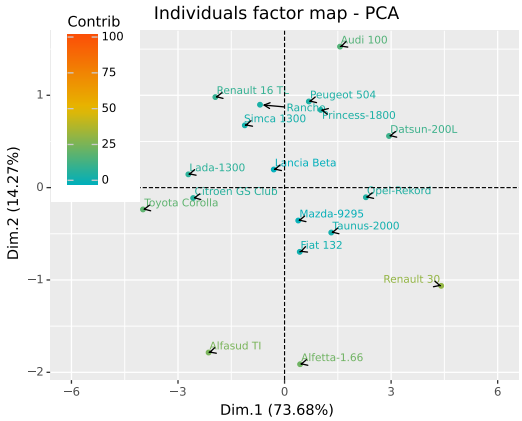


Figure 7.9 – Représentation des individus dans le premier plan factoriel coloré des contributions

Tout comme pour le cosinus carré, nous pouvons également visualiser les individus les plus contributives sur l'axe 1

```
# Contributions axe 1
from scientisttools.ggplot import fviz_contrib
p = fviz_contrib(pca,choice="ind",axis=0)
print(p)
```

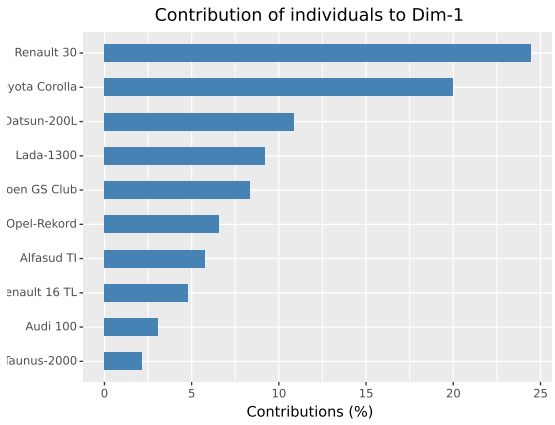


Figure 7.10 – Contribution des individus sur l'axe 1

... et sur l'axe 2

```
# Contributions axe 2
p = fviz_contrib(pca,choice="ind",axis=1)
print(p)
```

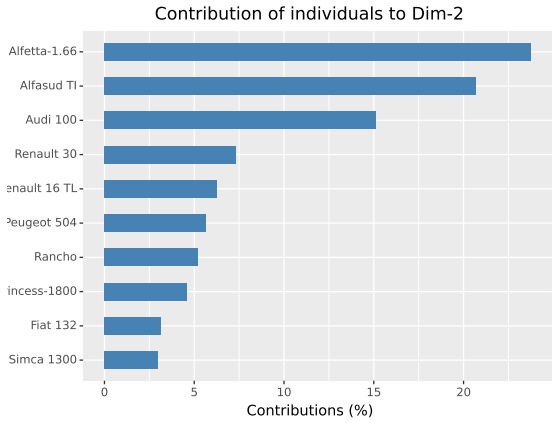


Figure 7.11 – Contribution des individus sur l'axe 2

7.2.2 Représentation des variables

La fonction get_pca de scientisttools permet d'extraire des informations (ccordonnées,cosinus carré, etc...) aussi bien sur les individus que les variances. Nous fixons `choice = "var"`.

```
# Informations sur les variables
col = get_pca(pca,choice="var")
print(col.keys())
```

```
## dict_keys(['corr', 'pcorr', 'coord', 'cos2', 'contrib', 'ftest', 'cor'])
```

7.2.2.1 Coordonnées des variables

Nous pouvons extraire les coordonnées à l'intérieur du dictionnaire en utilisant la clé « `coord` ».

```
# Coordonnées (=corrélations) des variables
print(col["coord"].iloc[:,:2])
```

```
##            Dim.1     Dim.2
## CYL     0.893464 -0.114906
## PUISS   0.886858 -0.384689
## LONG    0.886155  0.381029
## LARG    0.813536  0.412736
## POIDS   0.905187  0.224532
## V_MAX   0.754710 -0.573519
```

Nous pouvons dessiner maintenant le circle des corrélations (*cf.* Figure **??**)

```
# Cercle des corrélations
from scientisttools.ggplot import fviz_pca_var
p = fviz_pca_var(pca,text_type="text")
print(p)
```

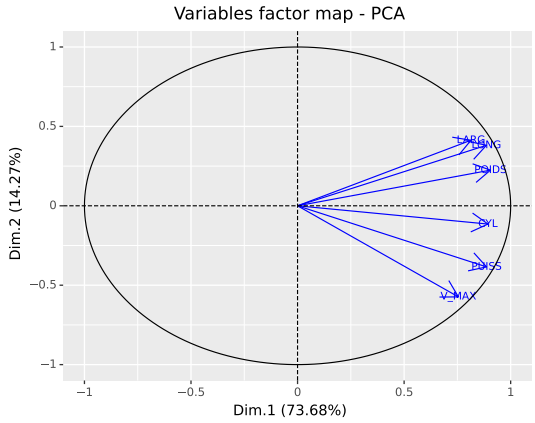


Figure 7.12 – Cercle des corrélations

On perçoit clairement l'effet taille sur le premier axe : les voitures puissantes et rapides sont aussi les plus lourdes et imposantes, la relation globale entre les variables est en réalité déterminée par la cylindrée (CYL).

7.2.2.2 Cosinus carré

On peut calculer la qualité de représentation des variables en montant la corrélation au carré.

```
# Cosinus carré des variables
print(col["cos2"].iloc[:,:2])
```

```
##            Dim.1     Dim.2
## CYL    0.798277  0.013203
## PUISS  0.786517  0.147986
## LONG   0.785270  0.145183
## LARG   0.661841  0.170351
## POIDS  0.819364  0.050415
## V_MAX  0.569588  0.328925
```

Tout comme les individus, nous pouvons colorer les variables en fonction de la qualité de représentation.

```
# Cercle des corrélations coloré du cos2
p = fviz_pca_var(pca,color="cos2",text_type="text",
                 gradient_cols = ("#00AFBB", "#E7B800", "#FC4E07"))
p = p + theme(legend_position=(0.2,0.73),legend_direction="vertical")
print(p)
```

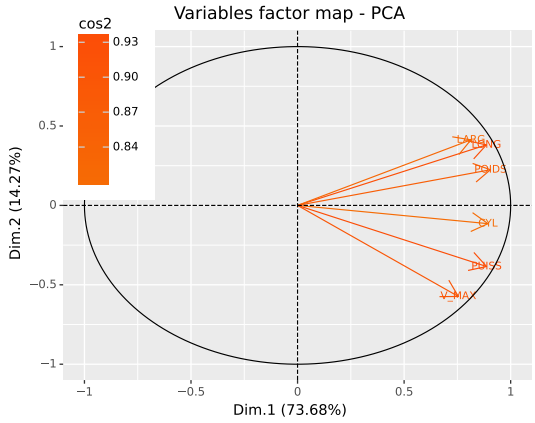


Figure 7.13 – Cercle des corrélations colorés par le cosinus carré

Il est également possible de visualiser les variables les plus représentées sur chaque axe.

7.2.2.3 Contributions des variables

La contribution est également basée sur le carré de la corrélation, mais relativisée par l'importance de l'axe.

```
# Contribution des variables
print(col["contrib"].iloc[:,:2])
```

```
##            Dim.1      Dim.2
## CYL    18.057062   1.542342
```

```
## PUISS  17.791052  17.286793
## LONG   17.762847  16.959384
## LARG   14.970882  19.899361
## POIDS  18.534057   5.889155
## V_MAX  12.884099  38.422964
```

Tout comme les individus, nous pouvons colorer les variables en fonction de la qualité de représentation.

```
# Cercle des corrélations coloré du cos2
p = fviz_pca_var(pca,color="contrib",text_type="text",
                 gradient_cols = ("#00AFBB", "#E7B800", "#FC4E07"))
p = p + theme(legend_position=(0.2,0.3),legend_direction="vertical")
print(p)
```

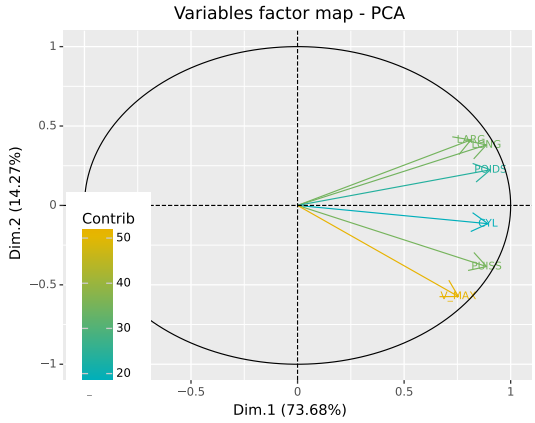


Figure 7.14 – Cercle des corrélations colorés par les contributions

7.2.3 Traitement des individus et variables illustratifs

7.2.3.1 Individus supplémentaires

Nous souhaitons positionner deux véhicules supplémentaires, des Peugeot, par rapport aux existantes.

```
# Chargement des individus supplémentaires
indSupp = pd.read_excel("./donnee/autos_acp_pour_python.xlsx",sheet_name=1,header=0,
                        index_col=0)
```

Table 7.5 – Individus supplémentaires

	CYL	PUISS	LONG	LARG	POIDS	V_MAX
Peugeot 604	2664	136	472	177	1410	180
Peugeot 304 S	1288	74	414	157	915	160

Nous faisosn appel à l afonction transform de scientisttools pour calculer leurs coordonnées.

```
# Projection dans l'espace factoriel
coordSupp = pca.transform(indSupp)
print(coordSupp)
```

```
## [[ 5.56329226 -0.33860928  0.46428878 -0.40214608  0.38981076  0.08102064]
##  [-2.21224139 -1.25777905  0.09304388  0.35370189 -0.648528   -0.12473042]]
```

Et à les représenter dans le premier plan factoriel parmi les observations actives.

```
p = fviz_pca_ind(pca,repel=True,text_type="text")+xlim(-7,7)
p = p + annotate("text", x = coordSupp[:,0], y = coordSupp[:,1],
                 label = list(indSupp.index))
print(p)
```

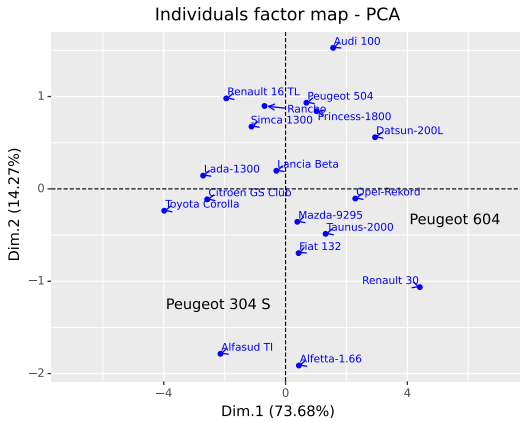


Figure 7.15 – Positionnement des individus supplémentaires dans le premiers plan factoriel

La Peugeot 604 se rapproche plutôt de la Renault 30, la Peugeot 304 de l'Alfasud TI.

7.2.3.2 Variables supplémentaires

Les variables illustratives sont situées dans la troisième feuille du classeur Excel (`sheet_name = 2`). Il faut avoir exactement les mêmes observations que les données actives bien évidemment, ce qui est le cas ici. Nous les importons :

```
#  Importation des variables supplémentaires
varSupp = pd.read_excel("./donnee/autos_acp_pour_python.xlsx",sheet_name=2,header=0,
                        index_col=0)
varSupp.index = X.index
```

Table 7.6 – Variables supplémentaires

	CYL	PUISS	LONG	LARG	POIDS	V_MAX
Peugeot 604	2664	136	472	177	1410	180
Peugeot 304 S	1288	74	414	157	915	160

a) Variables illustratives quantitatives

Nous récupérons les variables quantitatives dans une structure à part.

```
# Variables supplémentaires quanti
vsQuanti = varSupp.iloc[:,:2]
```

Nous faisons appel à la fonction _compute_quanti_sup_stats pour le calcul des corrélations de ces variables avec les axes factroriels exprimés par les coordonnées des observations.

```
# Corrélation avec les axes factoriels
col_sup= pca._compute_quanti_sup_stats(vsQuanti)
print(col_sup.keys())
```

```
## dict_keys(['corr', 'coord', 'cos2', 'ftest'])
```

```
# Corrélations
corrSupp = col_sup["coord"]
print(corrSupp)
```

```
##                   Dim.1     Dim.2     Dim.3     Dim.4     Dim.5     Dim.6
## PRIX           0.772475 -0.086708  0.133893  0.225829  0.159450  0.102549
## R_POIDS_PUIS -0.589039  0.672545  0.150176 -0.213657 -0.101628 -0.289997
```

Avec ces nouvelles coordonnées, nous pouvons placer les variables dans le cercle des corrélations.

```
# Cercle des corrélations avec les var. supp.
p = fviz_pca_var(pca,text_type="text")
p = p + annotate("text", x = corrSupp.iloc[:,0], y = corrSupp.iloc[:,1],
                 label = list(corrSupp.index))
print(p)
```

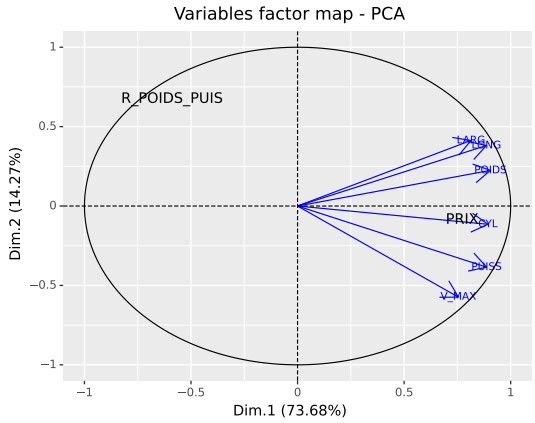


Figure 7.16 – Cercle des corrélations incluant les variables illustratives

On note dans ce cercle (*cf.* Figure 7.16) :

— La variable « PRIX » est globalement corrélée avec l'ensemble des variables, emportée par la première composante.
— « R_POIDS_PUIS » (rapport poids - puissance) est quasi - orthogonale au premier facteur. A bien y regarder, on remarque surtout qu'elle est à l'opposé de « V_MAX »

b) Variables illustrative qualitative

Nous isolons la variable illustrative qualitative dans une structure spécifique.

```
# Traitement de la var. quali. supplémentaire
vsQuali = varSupp.iloc[:,2]
```

Nous récupérons la liste des modalités

```
# Modalités de la variable qualitative
n_k= (varSupp.FINITION.value_counts(normalize=False).to_frame("Eff.")
                      .reset_index())
```

Table 7.7 – Distribution absolue de FINITION

FINITION	Effectifs
2_B	7
3_TB	6
1_M	5

Nous représentons les individus dans le plan factoriel, coloriées selon la modlaité associée de la variable illustrative.

```
#
df = pd.concat([row["coord"],varSupp["FINITION"]],axis=1)
p = (ggplot(df,aes(x="Dim.1",y="Dim.2",label=X.index))+
      geom_point(aes(color="FINITION",linetype="FINITION"))+
      geom_text(aes(color="FINITION"))+xlim(-6,6)+
      theme(legend_position=(0.2,0.3),legend_direction="vertical"))
print(p)
```

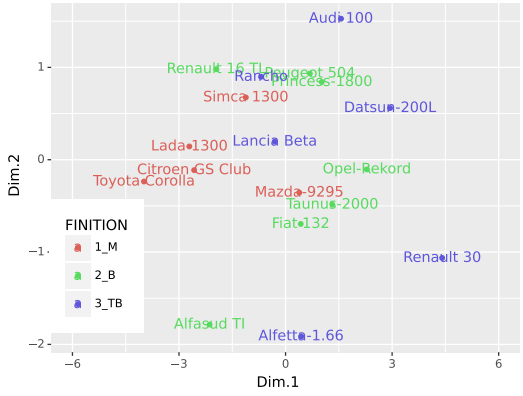


Figure 7.17 – Individus selon le type de finition

Puis nous calculons les positions des barycentres conditionnels dans le plan factoriel. Pour cela, nous faisons appel à la fonction _compute_quali_sup_stats de scientisttools.

```
# Moyennes conditionnelles
vsQuali_sup= pca._compute_quali_sup_stats(vsQuali)
print(vsQuali_sup.keys())
```

```
## dict_keys(['stats', 'coord', 'cos2', 'dist', 'eta2', 'vtest'])
```

On extrait les barycentres des modalités.

```
# Barycentres
mod_coord = vsQuali_sup["coord"]
print(mod_coord)
```

```
##                    Dim.1     Dim.2     Dim.3     Dim.4     Dim.5     Dim.6
## FINITION_1_M  -2.000355  0.022579  0.069577 -0.055847 -0.055043 -0.052354
## FINITION_2_B   0.235313 -0.045271 -0.113971 -0.218518 -0.078688 -0.007775
## FINITION_3_TB  1.392430  0.034001  0.074984  0.301477  0.137672  0.052699
```

Nous les plaçons dans le repère factoriel.

```
# Représentation graphique
p = (fviz_pca_ind(pca,repel=True,text_type="text")+xlim(-6,6)+
       annotate("text", x = mod_coord.iloc[:,0], y =mod_coord.iloc[:,1],
                label = list(mod_coord.index)))
print(p)
```

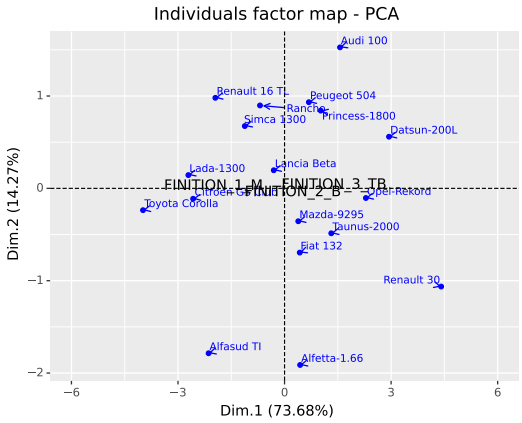


Figure 7.18 – Individus avec les modalités

Remarque 7.1 *Toutes ces étapes peuvent être exécutées sous scientisttools après instanciation de l'objet « pca ». Pour cela, toutes les variables (actives et supplémentaires) ainsi que les individus (actifs et supplémentaires) doivent être contenus dans un seul dataset.*

```
# Jointure des dataset
D = pd.concat([pd.concat([X,varSupp],axis=1),indSupp],axis=0)
```

Nous instancions l'algorithme en tenant compte de toute l'information dont on dispose :

```
# ACP
acp = PCA(normalize=True,
          n_components = None,
          row_labels=X.index,
          col_labels=X.columns,
          row_sup_labels=indSupp.index,
          quanti_sup_labels=vsQuanti.columns,
          quali_sup_labels=["FINITION"],
          parallelize=False)
acp.fit(D)
```

```
## PCA(col_labels=Index(['CYL', 'PUISS', 'LONG', 'LARG', 'POIDS', 'V_MAX'], dtype='object'),
##     quali_sup_labels=['FINITION'],
##     quanti_sup_labels=Index(['PRIX', 'R_POIDS_PUIS'], dtype='object'),
##     row_labels=Index(['Alfasud TI', 'Audi 100', 'Simca 1300', 'Citroen GS Club', 'Fiat 13
##        'Lancia Beta', 'Peugeot 504', 'Renault 16 TL', 'Renault 30',
##        'Toyota Corolla', 'Alfetta-1.66', 'Princess-1800', 'Datsun-200L',
##        'Taunus-2000', 'Rancho', 'Mazda-9295', 'Opel-Rekord', 'Lada-1300'],
##       dtype='object', name='Modele'),
##     row_sup_labels=Index(['Peugeot 604', 'Peugeot 304 S'], dtype='object', name='Modele')
```

```
# Résumé des informations
summaryPCA(acp)
```

```
##                      Principal Component Analysis - Results
##
## Importance of components
##                            Dim.1   Dim.2   Dim.3   Dim.4   Dim.5    Dim.6
## Variance                   4.421   0.856   0.373   0.214   0.093    0.043
## Difference                 3.565   0.483   0.159   0.121   0.050      NaN
## % of var.                 73.681  14.268   6.218   3.565   1.547    0.722
## Cumulative of % of var.   73.681  87.949  94.166  97.732  99.278  100.000
##
## Individuals (the 10 first)
##
##                  d(i,G)   p(i)  I(i,G)  Dim.1  ...   cos2  Dim.3     ctr   cos2
## Modele                                          ...
## Alfasud TI        2.868  0.056   0.457 -2.139  ...  0.388 -0.572   4.870  0.040
## Audi 100          2.583  0.056   0.371  1.561  ...  0.349 -1.315  25.762  0.259
## Simca 1300        1.469  0.056   0.120 -1.119  ...  0.211 -0.457   3.104  0.097
## Citroen GS Club   2.604  0.056   0.377 -2.574  ...  0.002 -0.149   0.329  0.003
## Fiat 132          1.081  0.056   0.065  0.428  ...  0.414  0.193   0.556  0.032
## Lancia Beta       1.065  0.056   0.063 -0.304  ...  0.034 -0.676   6.801  0.402
## Peugeot 504       1.230  0.056   0.084  0.684  ...  0.575  0.257   0.982  0.044
```

```
## Renault 16 TL     2.374  0.056   0.313 -1.948  ...  0.171  0.620   5.716  0.068
## Renault 30        4.668  0.056   1.211  4.410  ...  0.052  0.594   5.246  0.016
## Toyota Corolla    4.036  0.056   0.905 -3.986  ...  0.003  0.303   1.368  0.006
##
## [10 rows x 12 columns]
##
## Supplementary Individuals
##
##                Dim.1   cos2  Dim.2   cos2  Dim.3   cos2
## Modele
## Peugeot 604    5.563  0.979 -0.339  0.004  0.464  0.007
## Peugeot 304 S -2.212  0.695 -1.258  0.225  0.093  0.001
##
## Continues variables
##
##        Dim.1     ctr   cos2  Dim.2     ctr   cos2  Dim.3     ctr   cos2
## CYL    0.893  18.057  0.798 -0.115   1.542  0.013  0.216  12.504  0.047
## PUISS  0.887  17.791  0.787 -0.385  17.287  0.148  0.113   3.420  0.013
## LONG   0.886  17.763  0.785  0.381  16.959  0.145 -0.041   0.457  0.002
## LARG   0.814  14.971  0.662  0.413  19.899  0.170 -0.369  36.587  0.136
## POIDS  0.905  18.534  0.819  0.225   5.889  0.050  0.296  23.464  0.088
## V_MAX  0.755  12.884  0.570 -0.574  38.423  0.329 -0.297  23.568  0.088
##
## Supplementary continuous variable
##
##               Dim.1   cos2  Dim.2   cos2  Dim.3   cos2
## PRIX          0.772  0.597 -0.087  0.008  0.134  0.018
## R_POIDS_PUIS -0.589  0.347  0.673  0.452  0.150  0.023
##
## Supplementary categories
##
##                 dist  Dim.1   cos2  v.test  ...  v.test  Dim.3   cos2  v.test
## FINITION_1_M   2.004 -2.000  0.996  -2.433  ...   0.062  0.070  0.001   0.291
## FINITION_2_B   0.353  0.235  0.445   0.368  ...  -0.161 -0.114  0.104  -0.614
## FINITION_3_TB  1.435  1.392  0.942   1.931  ...   0.107  0.075  0.003   0.358
##
## [3 rows x 10 columns]
##
## Supplementatry categorical variable
##
##           Dim.1  Dim.2  Dim.3
## FINITION  0.402  0.002  0.022
```

Bibliographie

Buitinck, Lars, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, et al. 2013. “API Design for Machine Learning Software : Experiences from the Scikit-Learn Project.” In *ECML PKDD Workshop : Languages for Data Mining and Machine Learning*, 108–22.

Frontier, Serge. 1976. “Étude de La décroissance Des Valeurs Propres Dans Une Analyse En Composantes Principales : Comparaison Avec Le Moddle Du bâton Brisé.” *Journal of Experimental Marine Biology and Ecology* 25 (1) : 67–75.

Hotelling, Harold. 1933. “Analysis of a Complex of Statistical Variables into Principal Components.” *Journal of Educational Psychology* 24 (6) : 417.

Legendre, Louis, and Pierre Legendre. 1984. *La Structure Des Données écologiques*. Masson.

Quinlan, J Ross. 1993. “Program for Machine Learning.” *C4. 5*.

Rakotomalala, Ricco. 2005. “Arbres de décision.” *Revue Modulad* 33 : 163–87.

Saporta, Gilbert. 2006. *Probabilités, Analyse Des Données Et Statistique*. Editions technip.

Introduction au Machine Learning :

Application sous Python

L'apprentissage automatique est considéré comme un sous – ensemble de l'intelligence artificielle qui s'intéresse principalement au développement d'algorithmes permettant à un ordinateur d'apprendre par lui – même à partir de données et d'expériences passées.

Cet ouvrage présente des techniques liées au Machine Learning. Il met l'accent sur les méthodes basiques indispensables à tout étudiant ou tout praticien désireux d'appliquer les méthodes du Méthode Learning. Il s'adresse aux étudiants de Licence et Master de mathématiques appliquées, d'économie et d'économétrie, ainsi qu'aux élèves des écoles d'ingénieurs.

Duvérier DJIFACK ZEBAZE est ingénieur en Data Science et gestion des risques de l'école nationale de la statistique et de l'analyse de l'information (ENSAI) de Rennes en France et Ingénieur Statisticien Economiste (ISE) de l'école nationale de la statistique et de l'analyse économique (ENSAE) de Dakar au Sénégal. Par ailleurs, il est titulaire d'un master recherche en économie quantitative obtenu à la faculté des sciences économiques et de gestion appliquée de l'Université de Douala au Cameroun. Sa carrière débute en tant qu'ingénieur quantitatif au sein de l'équipe de modélisation de la Business Unit GLBA (Global Banking and Advisory) du groupe Société Générale.