Voici une description du Machine Learning :

Le ML permet à l'ordinateur d'apprendre au travers de l'étude de données et de statistiques
Le ML est une étape vers l'intelligence artificielle
Le ML analyse les données entrantes pour prédire des sorties

Comment faire fonctionner le ML ?

Nous allons revenir aux mathématiques et aux statistiques d'étude, et comment calculer des nombres importants basés sur des ensembles de données.

Nous apprendrons également à utiliser divers modules Python pour obtenir les réponses dont nous avons besoin.

Et nous apprendrons comment créer des fonctions capables de prédire le résultat en fonction de ce que nous avons appris.

Ensemble de données

Dans l'esprit d'un ordinateur, un ensemble de données est n'importe quel ensemble de données, qu'il s'agisse d'un tableau ou d'une base de données complète.

Exemple de tableau :
[99,86,87,88,111,86,103,87,94,78,77,85,86]

Exemple de base de données:

Marque	Couleur	Age	Vitesse	CarPass
BMW	red	5	167	Y
Volvo	black	7	148	Y
VW	gray	8	150	N
VW	white	7	140	Y
Ford	white	2	190	Y
VW	white	17	150	Y
Tesla	red	2	160	Y
BMW	black	9	159	Y
Volvo	gray	4	152	N
Ford	white	11	137	N
Toyota	gray	12	175	N
VW	white	9	172	N
Toyota	blue	6	159	Y

En regardant le tableau, nous pouvons deviner que la vitesse moyenne est probablement autour de 140 ou 150, et nous sommes également en mesure de déterminer la valeur la plus élevée et la valeur la plus basse, mais que pouvons-nous faire d'autre?
En regardant la base de données, nous pouvons voir que la couleur la plus populaire est le blanc, et la plus vieille voiture est de 17 ans, mais si nous pouvions prédire si une voiture avait un CarPass, rien qu'en regardant les autres valeurs?

Voilà à quoi sert le Machine Learning ! Analyser les données et ensuite, prédire le résultat !

Dans le Machine Learning, il est courant de travailler avec de très grands ensembles de données. De grosses bases de données : clients EDF, personnes atteintes du COVID long,….

Types de données

Pour analyser les données, il est important de savoir quel type de données nous traitons.

Nous pouvons diviser les types de données en trois catégories principales :

- Numérique
- Catégorique
- Ordinale

1/ Les données numériques sont des nombres et peuvent être divisées en deux catégories numériques :

1.1/ Données discrètes
- les nombres qui sont limités à des nombres entiers. Exemple : Le nombre de voitures qui passent.

1.2/ Données continues
- des nombres d'une valeur infinie. Exemple : Le prix d'un article ou la taille d'un article

2/ Les données catégoriques sont des valeurs qui ne peuvent pas être mesurées les unes par rapport aux autres. Exemple : une valeur de couleur ou des valeurs oui/non.

3/ Les données ordinales sont comme des données catégoriques, mais peuvent être mesurées les unes par rapport aux autres. Exemple : notes scolaires où A est meilleur que B et ainsi de suite.

En connaissant le type de données de votre source de données, vous serez en mesure de savoir quelle technique utiliser lors de leur analyse. Il faut donc d'abord connaître le type de données

Partie 1 - Apprentissage automatique (Machine Learning) - Mode Médiane Moyenne
Moyenne, médiane et mode

Que pouvons-nous apprendre en examinant un groupe de chiffres?

Dans l'apprentissage automatique (et en mathématiques), il y a souvent trois valeurs qui nous intéressent :

Moyenne - La valeur moyenne
Médiane - La valeur médiane
Mode - La valeur la plus courante

Exemple : Nous avons enregistré la vitesse de 13 voitures :
vitesse = [99,86,87,88,111,86,103,87,94,78,77,85,86]

Quelle est la moyenne, la médiane ou la valeur de vitesse la plus courante?

Moyenne

La valeur moyenne est la valeur moyenne. Réfléchissons bien à ce que cela signifie : la moyenne est une frontière.

Pour calculer la moyenne, il faut trouver la somme de toutes les valeurs et diviser la somme par le nombre de valeurs :
(99+86+87+88+111+86+103+87+94+78+77+85+86) / 13 = 89.77
Le module NumPy a un module pour faire ce calculer

```
import numpy
vitesse = [99,86,87,88,111,86,103,87,94,78,77,85,86]
x = numpy.mean(vitesse)
print(x)
```

Médiane
Soit les valeurs suivantes : 99,86,87,88,111,86,103,87,94,78,77,85,86

La valeur médiane est la valeur au milieu, après avoir trié toutes les valeurs :
77, 78, 85, 86, 86, 86, 87, 87, 88, 94, 99, 103, 111
Il est important que les nombres soient triés avant que vous puissiez trouver la médiane.

Le module NumPy a un module pour faire ce calculer

```
import numpy
vitesse = [99,86,87,88,111,86,103,87,94,78,77,85,86]
x = numpy.median(vitesse)
print(x)
```

S'il y a deux nombres au milieu, divisez la somme de ces nombres par deux.
77, 78, 85, 86, 86, 86, 87, 87, 94, 98, 99, 103
(86 + 87) / 2 = 86.5

```
import numpy
vitesse = [99,86,87,88,86,103,87,94,78,77,85,86]
x = numpy.median(vitesse)
print(x)
```

Mode

La valeur Mode est la valeur qui apparaît le plus souvent :
99, 86, 87, 88, 111, 86, 103, 87, 94, 78, 77, 85, 86 = 86

```
from scipy import stats
vitesse = [99,86,87,88,111,86,103,87,94,78,77,85,86]
x = stats.mode(vitesse)
print(x)
```

et on obtient : ModeResult(mode=array([86]), count=array([3]))
Note : il peut y avoir des erreurs de paramétrage puisqu'il y a un changement de version de NumPy : scipy.stats.mode(*a, axis=0, nan_policy='propagate', keepdims=False*)

Dans cet exemple, nous avons utilisé la librairie Scipy

Numpy contre SciPy

Numpy :

- Numpy est écrit en C et utilisé pour les calculs mathématiques ou numériques.
- Numpy est plus rapide que les autres bibliothèques Python
- Numpy est la bibliothèque la plus utile pour la science des données pour effectuer des calculs de base.
- Numpy ne contient rien d'autre qu'un type de données tableau qui effectue les opérations les plus élémentaires comme le tri, la mise en forme, l'indexation, etc.

SciPy : *Python3 -m pip install --user numpy scipy*

I. SciPy est construit au-dessus de NumPy

- Le module SciPy en Python est une version complète de l'algèbre linéaire tandis que Numpy ne contient que quelques fonctionnalités.
- La plupart des nouvelles fonctionnalités de Data Science sont disponibles dans Scipy plutôt que dans Numpy.

Machine Learning – écart type (standard déviation)

Qu'est-ce que l'écart-type?

L'écart-type est un nombre qui décrit la répartition des valeurs autour de la valeur moyenne.

Un écart-type faible signifie que la plupart des nombres sont proches de la valeur moyenne (moyenne).

Un écart-type élevé signifie que les valeurs sont réparties sur une plage plus large.

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

N : effectif ; x_i : chaque valeur ; μ : moyenne

Exemple : Cette fois, nous avons enregistré la vitesse de 7 voitures :

vitesse = [86,87,88,86,87,85,86]

L'écart-type est ici de 0,9

Cela signifie que la plupart des valeurs se situent dans la plage de 0,9 par rapport à la valeur moyenne, qui est 86,4.

Faisons de même avec une sélection de numéros avec une gamme plus large :

vitesse = [32,111,138,28,59,77,97]

L'écart-type est ici de 37,85

Cela signifie que la plupart des valeurs se situent dans la plage de 37,85 par rapport à la valeur moyenne, qui est 77,4.

=> Comme vous pouvez le voir, un écart-type plus élevé indique que les valeurs sont réparties sur une plage plus large. Les valeurs sont , dans la plupart des cas, éloignées de la valeur moyenne

Le module NumPy dispose d'une méthode pour calculer l'écart type (en anglais, standard deviation) :

```
import numpy
vitesse = [86,87,88,86,87,85,86]
x = numpy.std(vitesse)
print(x)
```

Machine Learning – Variance

La variance est un autre nombre qui indique la répartition des valeurs.

En fait, si vous prenez la racine carrée de la variance, vous obtenez l'écart-type!

Ou l'inverse, si vous multipliez l'écart type par lui-même, vous obtenez la variance!

$$S^2 = \frac{\sum(x_i - \bar{x})^2}{n-1}$$ N : nb d'observation; x_i : chaque valeur ; μ : moyenne

Voici comment calculer la variance :

1. Trouvez la moyenne :
   (32+111+138+28+59+77+97) / 7 = 77.4
2. Pour chaque valeur : trouver la différence par rapport à la moyenne :
   32 - 77.4 = -45.4
   111 - 77.4 = 33.6
   138 - 77.4 = 60.6
   28 - 77.4 = -49.4
   59 - 77.4 = -18.4

77 - 77.4 = - 0.4
97 - 77.4 = 19.6

3. Pour chaque différence : trouver la valeur au carrée :

$(-45.4)^2 = 2061.16$

$(33.6)^2 = 1128.96$

$(60.6)^2 = 3672.36$

$(-49.4)^2 = 2440.36$

$(-18.4)^2 = 338.56$

$(-0.4)^2 = 0.16$

$(19.6)^2 = 384.16$

4. La variance est le nombre moyen de ces différences au carré :
(2061.16+1128.96+3672.36+2440.36+338.56+0.16+384.16) / 7 = 1432.2

Numpy a une méthode pour calculer la vraiance : la méthode est var()

```
import numpy
vitesse = [32,111,138,28,59,77,97]
x = numpy.var(vitesse)
print(x)
```

Écart-type

Comme nous l'avons appris, la formule pour trouver l'écart-type est la racine carrée de la variance : √1432.25 = 37.85

Ou, comme dans l'exemple précédent, utiliser le NumPy pour calculer l'écart type :

```
import numpy
vitesse = [32,111,138,28,59,77,97]
x = numpy.std(vitesse)
print(x)
```

Symboles

L'écart type est souvent représenté par le symbole Sigma : σ

La variance est souvent représentée par le symbole Sigma au carré: σ^2

Machine Learning - Percentiles

Centiles

Que sont les centiles?

Les centiles sont utilisés dans les statistiques pour vous donner un nombre qui décrit la valeur qu'un pourcentage donné des valeurs sont inférieurs à.

Exemple : Disons que nous avons un éventail d'âges de toutes les personnes qui vivent à l'IS-TY :-).

ages = [5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]

Qu'est-ce que le 75 centile? La réponse est 43, ce qui signifie que 75 % des gens ont 43 ans ou moins.

Le module NumPy a une méthode pour trouver le centile spécifié :

```
import numpy
ages = [5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]
x = numpy.percentile(ages, 75)
print(x)
```

Quel est l'âge auquel 90 % des gens sont plus jeunes?

Le module NumPy a une méthode pour trouver le centile spécifié :

```
import numpy
ages = [5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]
x = numpy.percentile(ages, 90)
print(x)
```

Distribution des données

Plus tôt dans ce TP/tutoriel, nous avons travaillé avec de très petites quantités de données dans nos exemples, juste pour comprendre les différents concepts.

Dans le monde réel, les ensembles de données sont beaucoup plus grands, mais il peut être difficile de recueillir des données réelles, au moins à un stade précoce d'un projet.

Comment pouvons-nous obtenir des ensembles de données volumineuses?

Pour créer des jeux de big data pour les tests, nous utilisons le module Python NumPy, qui est livré avec un certain nombre de méthodes pour créer des jeux de données aléatoires, de toute taille.

Exemple
1/ Créons un tableau contenant 250 valeurs flottantes entre 0 et 5

```
import numpy
x = numpy.random.uniform(0.0, 5.0, 250)
print(x)
```

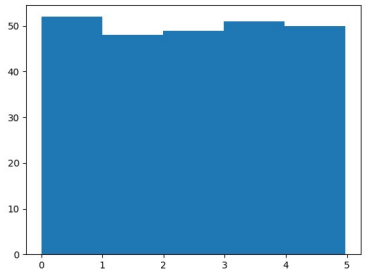
2/ Histogramme

Pour visualiser l'ensemble de données, nous pouvons dessiner un histogramme avec les données que nous avons collectées.

Nous utiliserons le module Python Matplotlib pour dessiner un histogramme.

```
import numpy
import matplotlib.pyplot as plt
y = numpy.random.uniform(0.0, 5.0, 250)
plt.hist(y, 5)
plt.show()
```

et voici le résultat :



Explication de l'histogramme

Nous utilisons le tableau de l'exemple ci-dessus pour dessiner un histogramme avec 5 barres.

La première barre représente le nombre de valeurs dans le tableau entre 0 et 1.

La deuxième barre représente le nombre de valeurs comprises entre 1 et 2. Etc.

Ce qui nous donne ce résultat:

- 52 valeurs sont comprises entre 0 et 1
- 48 valeurs sont comprises entre 1 et 2
- 49 valeurs se situent entre 2 et 3
- 51 valeurs se situent entre 3 et 4
- 50 valeurs se situent entre 4 et 5

Remarque : Les valeurs du tableau sont des nombres aléatoires et n'affichent pas exactement le même résultat sur votre ordinateur.

Distributions des mégadonnées

Un tableau contenant 250 valeurs n'est pas considéré comme très grand, mais maintenant vous savez comment créer un ensemble aléatoire de valeurs, et en changeant les paramètres, vous pouvez créer l'ensemble de données aussi grand que vous le souhaitez.

Exemple

Créons un tableau avec 100000 nombres aléatoires et les afficher à l'aide d'un histogramme avec 100 barres :

```
import numpy
import matplotlib.pyplot as plt
y = numpy.random.uniform(0.0, 5.0, 100000)
plt.hist(y, 100)
plt.show()
```

Distribution normale des données

Dans la partie précédente, nous avons appris comment créer un tableau complètement aléatoire, d'une taille donnée, et entre deux valeurs données.

Dans cette partie, nous allons apprendre comment créer un tableau où les valeurs sont concentrées autour d'une valeur donnée : c'est le cas réel, le cas normal.

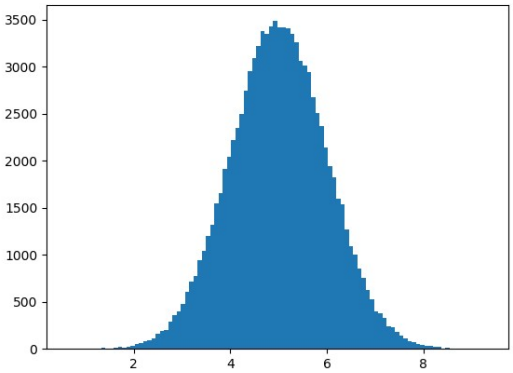
En théorie des probabilités, ce type de distribution de données est connu sous le nom de distribution de données normale, ou distribution de données gaussienne, d'après le mathématicien Carl Friedrich Gauss qui a proposé la formule de cette distribution de données.

Exemple

Une distribution de données normale typique :

```
import numpy
import matplotlib.pyplot as plt
x = numpy.random.normal(5.0, 1.0, 100000)
plt.hist(x, 100)
plt.show()
```

et voici le résultat obtenu :



Note : Un graphique de distribution normale est également connu comme la courbe de cloche en raison de sa forme caractéristique d'une cloche.

Explication de l'histogramme

Nous utilisons le tableau de la méthode numpy.random.normal(), avec 100 000 valeurs, pour dessiner un histogramme avec 100 barres.

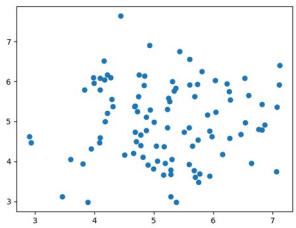
Nous spécifions que la valeur moyenne est de 5,0 et que l'écart type est de 1,0.

Cela signifie que les valeurs doivent être concentrées autour de 5,0, et rarement plus loin que 1,0 de la moyenne.

Et comme vous pouvez le voir dans l'histogramme, la plupart des valeurs se situent entre 4,0 et 6,0, avec un sommet à environ 5,0.

Diagramme de dispersion

Un nuage de points est un diagramme où chaque valeur de l'ensemble de données est représentée par un point.



Le module Matplotlib a une méthode pour dessiner des diagrammes de dispersion (scatter en anglais), il a besoin de deux tableaux de même longueur, un pour les valeurs de l'axe des x et un pour les valeurs de l'axe des y :

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]

y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

→ Le tableau x représente l'âge de chaque voiture.

→ Le tableau y représente la vitesse de chaque voiture.

Utilisez la méthode scatter() (scatter = dispersion, éparpiller, répandre en français) pour représenter un diagramme de nuage de points (diagramme de dispersion)

```
import matplotlib.pyplot as plt
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
plt.scatter(x, y)
plt.show()
```

et voici le résultat

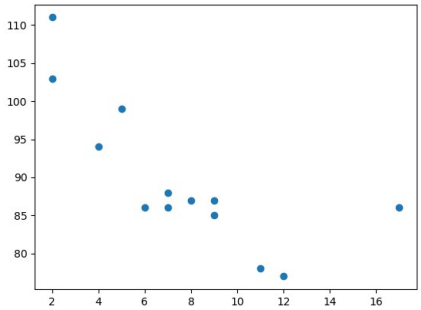


Diagramme de dispersion expliqué

L'axe des x représente les âges, et l'axe des y représente les vitesses.

Ce que nous pouvons lire dans le diagramme, c'est que les deux voitures les plus rapides avaient à la fois 2 ans et la voiture la plus lente 12 ans.

Remarque : Il semble que plus la voiture est récente, plus elle roule vite, mais cela pourrait être une coïncidence, après tout, nous n'avons enregistré que 13 voitures.

Machine Learning – Régression linéaire

Régression

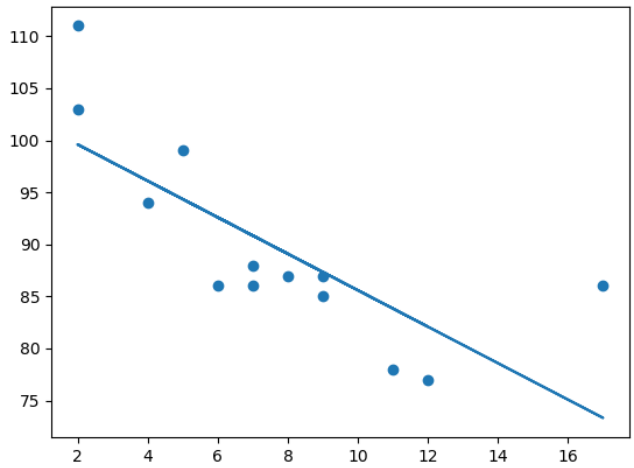
Le terme régression est utilisé lorsque vous essayez de trouver la relation entre les variables.

Dans le Machine Learning, et dans la modélisation statistique, cette relation est utilisée pour prédire le résultat des événements futurs.

Régression linéaire

La régression linéaire utilise la relation entre les points de données pour tracer une ligne droite à travers tous.

Cette ligne peut être utilisée pour prédire les valeurs futures.



Dans le Machine Learning, prédire l'avenir est très important.

Comment cela fonctionne-t-il ?

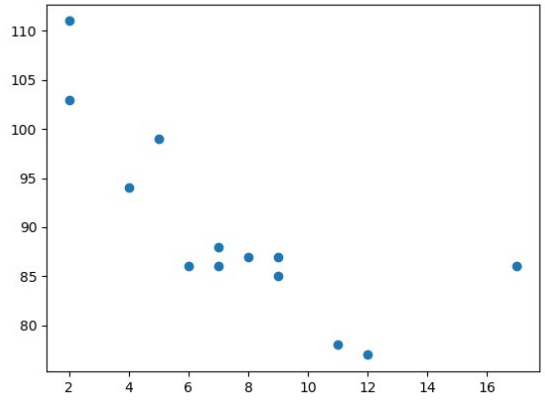
Python a des méthodes pour trouver une relation entre les points de données et pour tracer une ligne de régression linéaire. Nous allons vous montrer comment utiliser ces méthodes au lieu de passer par la formule mathématique.

Dans l'exemple ci-dessous, l'axe des x représente l'âge, et l'axe des y représente la vitesse. Nous avons enregistré l'âge et la vitesse de 13 voitures qui passaient devant un poste de péage. Voyons si les données que nous avons recueillies pourraient être utilisées dans une régression linéaire :

Exemple

Commencer par tracer un diagramme de dispersion :

```
import matplotlib.pyplot as plt
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
plt.scatter(x, y)
plt.show()
```

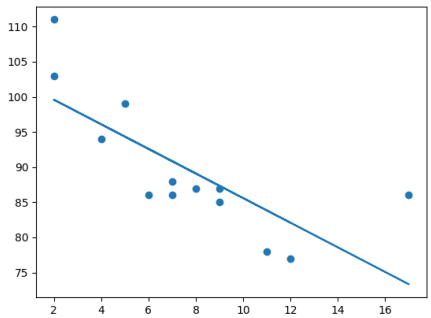


Exemple

Importez scipy et représentez la ligne de régression linéaire

```
import matplotlib.pyplot as plt
from scipy import stats

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
slope, intercept, r, p, std_err = stats.linregress(x, y)

#slope = pente; intercept = interception; rvalue est le coefficient de corrélation

def myfunc(x):
 return slope * x + intercept

mymodel = list(map(myfunc, x))

plt.scatter(x, y)
plt.plot(x, mymodel)
plt.show()
```

Voici le résultat ci-dessous



Explications de l'exemple :

Sur les 2 premières lignes, nous importons les librairies utiles

Ensuite, nous créons nos 2 tableaux de valeurs pour les axes x et y

Exécuter une méthode qui renvoie certaines valeurs clés importantes de régression linéaire : slope, intercept, r, p, std_err = stats.linregress(x, y)

Créez une fonction qui utilise les valeurs de pente (slope) et d'interception (intercept) pour retourner une nouvelle valeur. Cette nouvelle valeur représente l'endroit où la valeur x correspondante sera placée sur l'axe des y :

```
def myfunc(x):
  return slope * x + intercept
```

Exécuter chaque valeur du tableau x à travers la fonction. Cela donnera un nouveau tableau avec de nouvelles valeurs pour l'axe y :

```
mymodel = list(map(myfunc, x))
```

Tracer le diagramme de dispersion d'origine :

```
plt.scatter(x, y)
```

Tracer la ligne de la régression linéaire :

```
plt.plot(x, mymodel)
```

et afficher le diagramme :

```
plt.show()
```

Régression linéaire : Le « r » pour la relation

Il est important de savoir comment la relation entre les valeurs de l'axe des x et les valeurs de l'axe des y est, s'il n'y a pas de relation, la régression linéaire ne peut pas être utilisée pour prédire quoi que ce soit.

Cette relation - le coefficient de corrélation - est appelée r.

La valeur r du coefficient de corrélation varie de **-1 à 1**, où 0 signifie aucune relation, et 1 (comme -1) signifie 100% lié.

Python et le module Scipy calculeront cette valeur pour vous, tout ce que vous avez à faire est de l'alimenter avec les valeurs x et y.

Exemple

Dans quelle mesure mes données s'intègrent-elles dans une régression linéaire?

```
from scipy import stats
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
slope, intercept, r, p, std_err = stats.linregress(x, y)
print(r)
```

Note : Le résultat -0.76 montre qu'il y a une relation, pas parfaite, mais elle indique que nous pourrions utiliser la régression linéaire dans les prédictions futures.

Régression linéaire - Prévoir les valeurs futures

Nous pouvons maintenant utiliser les informations que nous avons recueillies pour prédire les valeurs futures.

Exemple : Essayons de prédire la vitesse d'une voiture de 10 ans.

Pour ce faire, nous avons besoin de la même fonction myfunc() de l'exemple ci-dessus :

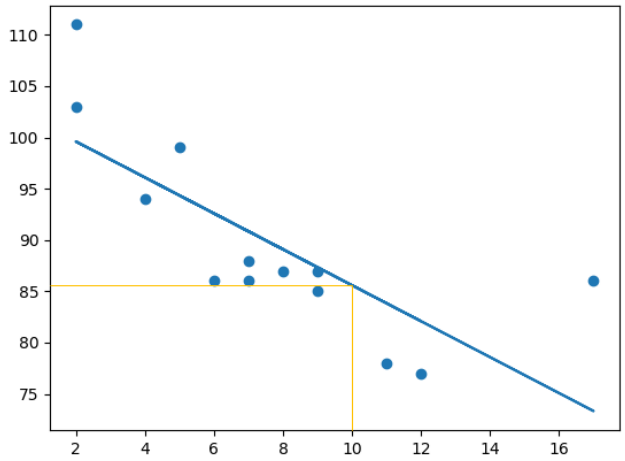
```
def myfunc(x):
  return slope * x + intercept
```

Exemple :

Essayons de prédire la vitesse d'une voiture de 10 ans d'age

```
from scipy import stats
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
  return slope * x + intercept

speed = myfunc(10)
print(speed)
```

L'exemple prévoyait une vitesse de 85,6, que nous pourrions également lire dans le diagramme :



Régression linéaire - Mauvais ajustement?

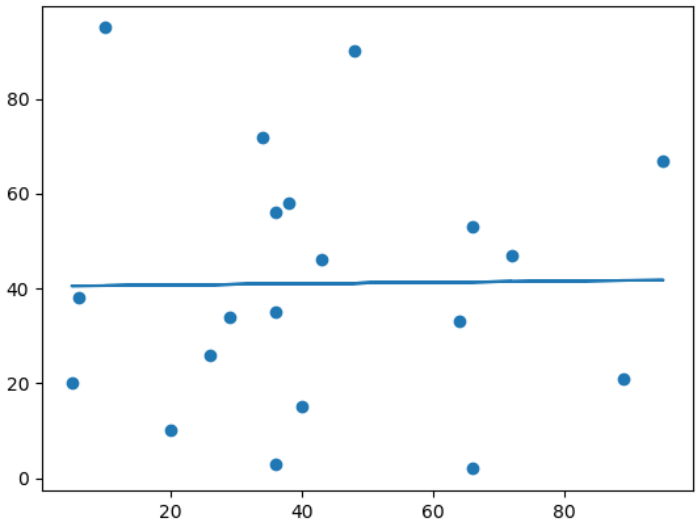
Créons un exemple où la régression linéaire ne serait pas la meilleure méthode pour prédire les valeurs futures.

Exemple

Ces valeurs pour les axes x et y devraient donner un très mauvais ajustement pour la régression linéaire :

```
import matplotlib.pyplot as plt
from scipy import stats

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
  return slope * x + intercept

mymodel = list(map(myfunc, x))

plt.scatter(x, y)
plt.plot(x, mymodel)
plt.show()
```

et on obtient :



Et le r pour la relation ?

Ici, on devrait obtenir une valeur de r très faible :

```
import numpy
from scipy import stats
```

```
x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

slope, intercept, r, p, std_err = stats.linregress(x, y)
print(r)
```

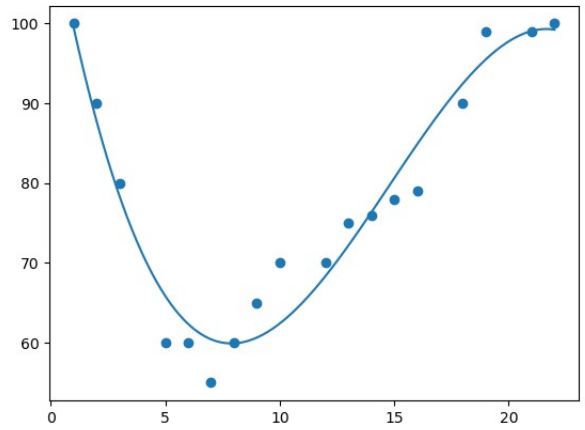
Le résultat : 0,013 indique une très mauvaise relation, et nous dit que cet ensemble de données ne convient pas à la régression linéaire.

Machine Learning – Régression polynomiale

Régression polynomiale

Si vos points de données ne correspondent clairement pas à une régression linéaire (une ligne droite à travers tous les points de données), il pourrait être idéal pour la régression polynomiale.

La régression polynomiale, comme la régression linéaire, utilise la relation entre les variables x et y pour trouver la meilleure façon de tracer une ligne à travers les points de données.



Comment cela fonctionne-t-il ?

Python a des méthodes pour trouver une relation entre les points de données et pour tracer une ligne de régression polynomiale. Nous allons vous montrer comment utiliser ces méthodes au lieu de passer par la formule mathématique.

Dans l'exemple ci-dessous, nous avons enregistré 18 voitures qui passaient devant un poste de péage.

Nous avons enregistré la vitesse de la voiture, et le moment de la journée (heure) le passage s'est produit.

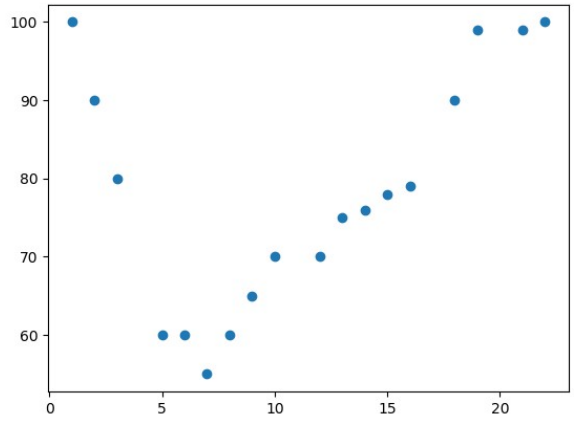
L'axe des x représente les heures de la journée et l'axe des y représente la vitesse :

Exemple

Commencer par tracer un diagramme de dispersion :

```
import matplotlib.pyplot as plt

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

plt.scatter(x, y)
plt.show()
```

Et ce programme donne le résultat suivant :

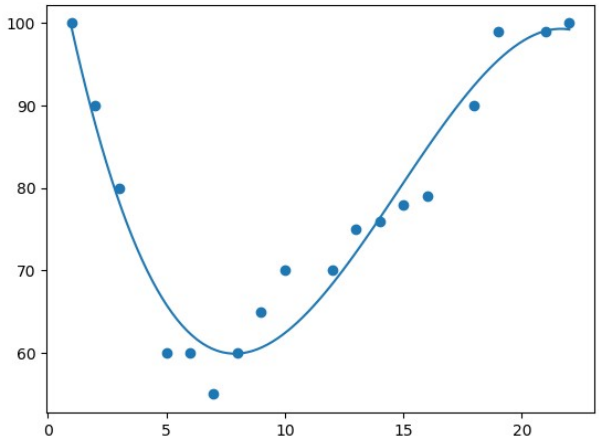


Exemple :

Importez les modules numpy et matplotlib et ensuite, représenter la ligne de la régression polynomiale :

```
import numpy
import matplotlib.pyplot as plt

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

myline = numpy.linspace(1, 22, 100)

plt.scatter(x, y)
plt.plot(myline, mymodel(myline))
plt.show()
```

Donne le résultat suivant :



Exemple expliqué :

Tout d'abord, nous importons les modules necessaires :

```
import numpy
import matplotlib.pyplot as plt
```

Ensuite, nous créons les 2 tableaux pour les axes x et y :

```
x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]
```

NumPy a une méthode qui nous permet de faire un modèle polynomial :

```
mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))
```

Préciser ensuite comment la ligne s'affichera, on commence à la position 1, et on termine à la position 22 :

```
myline = numpy.linspace(1, 22, 100)
```

Ensuite, on affiche le diagramme de dispersion d'origine :

```
plt.scatter(x, y)
```

et ensuite, on affiche la ligne de régression polynomiale :

```
plt.plot(myline, mymodel(myline))
```

et on fait réellement l'affichage de l'objet

```
plt.show()
```

Régression polynomiale – le R-carré

Il est important de savoir dans quelle mesure la relation entre les valeurs des axes x et y est bonne, s'il n'y a pas de relation, la régression polynomiale ne peut pas être utilisée pour prédire quoi que ce soit.

La relation est mesurée avec une valeur appelée le r-carré.

La valeur au carré de r varie de 0 à 1, où 0 signifie aucune relation, et 1 signifie 100% liée.

Python et le module Sklearn calculeront cette valeur pour vous, il vous suffit de l'alimenter avec les tableaux x et y :

Exemple

Dans quelle mesure mes données s'intègrent-elles dans une régression polynomiale ?

```
import numpy
from sklearn.metrics import r2_score
x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]
mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

print(r2_score(y, mymodel(x)))
```

Note : Le résultat 0.94 montre qu'il y a une très bonne relation, et nous pouvons utiliser la régression polynomiale dans les prédictions futures.

Régression polynomiale - Prévoir les valeurs futures

Nous pouvons maintenant utiliser les informations que nous avons recueillies pour prédire les valeurs futures.

Exemple : Essayons de prédire la vitesse d'une voiture qui passe le péage à environ 17:00:

Pour ce faire, nous avons besoin du même tableau mymodel de l'exemple ci-dessus :

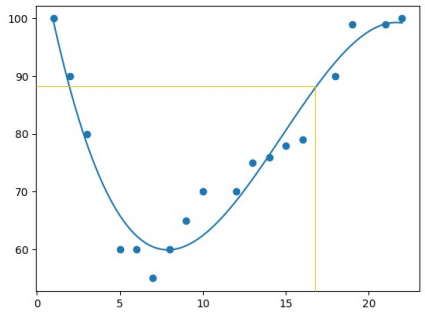
```
mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))
```

Exemple

On va prédire la vitesse d'une voiture qui passe à 17 h :

```
import numpy
from sklearn.metrics import r2_score
x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))
speed = mymodel(17)
print(speed)
```

L'exemple prévoyait une vitesse de 88,87, que nous pourrions également lire dans le diagramme :



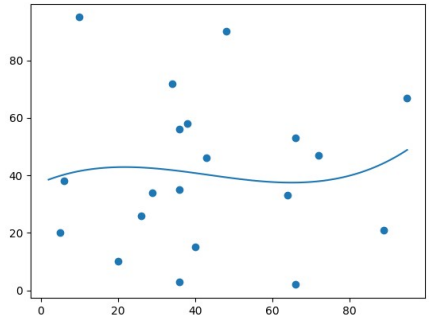
Régression polynomiale - Mauvais ajustement?

Créons un exemple où la régression polynomiale ne serait pas la meilleure méthode pour prédire les valeurs futures.

Exemple

Ces valeurs pour les axes x et y devraient entraîner un très mauvais ajustement pour la régression polynomiale :

```
import numpy
import matplotlib.pyplot as plt

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

myline = numpy.linspace(2, 95, 100)

plt.scatter(x, y)
plt.plot(myline, mymodel(myline))
plt.show()
```



Et la valeur de r au carré ?

On devrait obtenir une valeur très faible de r^2

```
import numpy
from sklearn.metrics import r2_score
```

```
x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

print(r2_score(y, mymodel(x)))
```

Le résultat : 0.00995 indique une très mauvaise relation, et nous dit que cet ensemble de données ne convient pas à la régression polynomiale.

Régression multiple

La régression multiple est comme la régression linéaire, mais avec plus d'une valeur indépendante, ce qui signifie que nous essayons de prédire une valeur basée sur deux variables ou plus.

Jetez un œil à l'ensemble de données ci-dessous, il contient des informations sur les voitures.

Car	Model	Volume	Weight	CO2
Toyota	Aygo	1000	790	99
Mitsubi-shi	Space Star	1200	1160	95
Skoda	Citigo	1000	929	95
Fiat	500	900	865	90
Mini	Cooper	1500	1140	105
VW	Up!	1000	929	105
Skoda	Fabia	1400	1109	90
Mercedes	A-Class	1500	1365	92
Ford	Fiesta	1500	1112	98
Audi	A1	1600	1150	99
Hyundai	I20	1100	980	99
Suzuki	Swift	1300	990	101
Ford	Fiesta	1000	1112	99
Honda	Civic	1600	1252	94
Hundai	I30	1600	1326	97
Opel	Astra	1600	1330	97
BMW	1	1600	1365	99
Mazda	3	2200	1280	104
Skoda	Rapid	1600	1119	104
Ford	Focus	2000	1328	105
Ford	Mondeo	1600	1584	94
Opel	Insignia	2000	1428	99
Mercedes	C-Class	2100	1365	99
Skoda	Octavia	1600	1415	99
Volvo	S60	2000	1415	99
Mercedes	CLA	1500	1465	102
Audi	A4	2000	1490	104
Audi	A6	2000	1725	114
Volvo	V70	1600	1523	109
BMW	5	2000	1705	114
Mercedes	E-Class	2100	1605	115
Volvo	XC70	2000	1746	117
Ford	B-Max	1600	1235	104
BMW	2	1600	1390	108
Opel	Zafira	1600	1405	109

Car	Model	Volume	Weight	CO2
Mercedes	SLK	2500	1395	120

Nous pouvons prédire les émissions de CO2 d'une voiture en fonction de la taille du moteur, mais avec une régression multiple, nous pouvons ajouter plus de variables, comme le poids de la voiture, pour rendre la prédiction plus précise.

Comment cela fonctionne-t-il ?

En Python, nous avons des modules qui feront le travail pour nous. Commencez par importer le module Pandas.

```
import pandas
```

Le module Pandas nous permet de lire des fichiers csv et de retourner un objet DataFrame.

Le fichier est destiné à des fins de test uniquement, vous pouvez le télécharger ici :data.csv

```
df = pandas.read_csv("data.csv")
```

Faites ensuite une liste des valeurs indépendantes et appelez cette variable X.

Mettez les valeurs dépendantes dans une variable appelée y.

```
X = df[['Weight', 'Volume']]
y = df['CO2']
```

Conseil : Il est courant de nommer la liste des valeurs indépendantes avec une majuscule X, et la liste des valeurs dépendantes avec une minuscule y.

Nous utiliserons certaines méthodes du module **sklearn**, de sorte que nous devrons également importer ce module :

```
from sklearn import linear_model
```

À partir du module sklearn, nous utiliserons la méthode LinearRegression() pour créer un objet de régression linéaire.

Cet objet a une méthode appelée fit() qui prend les valeurs indépendantes et dépendantes comme paramètres et remplit l'objet de régression avec des données qui décrivent la relation :

```
regr = linear_model.LinearRegression()
regr.fit(X, y)
```

Nous avons maintenant un objet de régression qui est prêt à prédire les valeurs de CO2 en fonction du poids et du volume d'une voiture :

```
#predict the CO2 emission of a car where the weight is 2300kg, and
the volume is 1300cm3:
predictedCO2 = regr.predict([[2300, 1300]])
```

Exemple :

Regardez l'exemple complet en fonctionnement :

```
import pandas
from sklearn import linear_model
```

```
df = pandas.read_csv("data.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

#predict the CO2 emission of a car where the weight is 2300kg, and
the volume is 1300cm3:
predictedCO2 = regr.predict([[2300, 1300]])

print(predictedCO2)
```

Ce qui nous donne : [107.2087328]

Nous avons prédit qu'une voiture avec un moteur de 1,3 litre et un poids de 2300 kg libérera environ 107 grammes de CO2 pour chaque kilomètre parcouru.

Régression multiple - Coefficient

Le coefficient est un facteur qui décrit la relation avec une variable inconnue.

Exemple : si x est une variable, alors 2x est x deux fois. x est la variable inconnue, et le nombre 2 est le coefficient.

Dans ce cas, nous pouvons demander le coefficient de poids par rapport au CO2, et le volume par rapport au CO2. Les réponses que nous obtenons nous disent ce qui se passerait si nous augmentions ou diminuions l'une des valeurs indépendantes.

Exemple

Imprimer les valeurs de coefficient de l'objet de régression :

```
import pandas
from sklearn import linear_model

df = pandas.read_csv("data.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

print(regr.coef_)
```

Nous donne comme résultat : [0.00755095 0.00780526]

Explication du résultat

Le tableau de résultats représente les valeurs de coefficient de poids et de volume.

Poids : 0.00755095

Volume : 0.00780526

Ces valeurs nous indiquent que si le poids augmente de 1kg, les émissions de CO2 augmentent de 0,00755095g.

Et si la taille du moteur (Volume) augmente de 1 cm3, les émissions de CO2 augmentent de 0,00780526 g.

Je pense que c'est une estimation juste, mais laissez-le tester!

Nous avons déjà prédit que si une voiture avec un moteur de 1300cm3 pèse 2300kg, l'émission de CO2 sera d'environ 107g.

Et si nous augmentions le poids avec 1000kg ?

Exemple

Copiez l'exemple précédent, mais changez le poids de 2300 à 3300 :

```
import pandas
from sklearn import linear_model

df = pandas.read_csv("data.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

predictedCO2 = regr.predict([[3300, 1300]])

print(predictedCO2)
```

et la prédiction nous donne : [114.75968007]

Nous avons prédit qu'une voiture avec un moteur de 1,3 litre et un poids de 3300 kg libérera environ 115 grammes de CO2 pour chaque kilomètre parcouru.

Ce qui montre que le coefficient de 0,00755095 est correct :

107,2087328 + (1000 * 0,00755095) = 114,75968

Caractéristiques de l'échelle

Lorsque vos données ont des valeurs différentes, et même des unités de mesure différentes, il peut être difficile de les comparer. Qu'est-ce que les kilogrammes comparés aux compteurs? Ou l'altitude par rapport au temps ?

La réponse à ce problème est la mise à l'échelle. Nous pouvons convertir les données en nouvelles valeurs qui sont plus faciles à comparer.

Regardez le tableau ci-dessous, c'est le même jeu de données que nous avons utilisé dans le chapitre de régression multiple, mais cette fois la colonne de volume contient des valeurs en litres au lieu de cm3 (1,0 au lieu de 1000).

Car	Model	Volume	Weight	CO2
Toyota	Aygo	1.0	790	99
Mitsubishi	Space Star	1.2	1160	95
Skoda	Citigo	1.0	929	95
Fiat	500	0.9	865	90
Mini	Cooper	1.5	1140	105
VW	Up!	1.0	929	105
Skoda	Fabia	1.4	1109	90
Mercedes	A-Class	1.5	1365	92
Ford	Fiesta	1.5	1112	98
Audi	A1	1.6	1150	99
Hyundai	I20	1.1	980	99
Suzuki	Swift	1.3	990	101
Ford	Fiesta	1.0	1112	99
Honda	Civic	1.6	1252	94
Hundai	I30	1.6	1326	97
Opel	Astra	1.6	1330	97
BMW	1	1.6	1365	99
Mazda	3	2.2	1280	104
Skoda	Rapid	1.6	1119	104
Ford	Focus	2.0	1328	105
Ford	Mondeo	1.6	1584	94
Opel	Insignia	2.0	1428	99
Mercedes	C-Class	2.1	1365	99
Skoda	Octavia	1.6	1415	99
Volvo	S60	2.0	1415	99
Mercedes	CLA	1.5	1465	102
Audi	A4	2.0	1490	104
Audi	A6	2.0	1725	114
Volvo	V70	1.6	1523	109
BMW	5	2.0	1705	114
Mercedes	E-Class	2.1	1605	115

Car	Model	Volume	Weight	CO2
Volvo	XC70	2.0	1746	117
Ford	B-Max	1.6	1235	104
BMW	2	1.6	1390	108
Opel	Zafira	1.6	1405	109
Mercedes	SLK	2.5	1395	120

Il peut être difficile de comparer le volume 1.0 avec le poids 790, mais si nous les mettons tous les deux en valeurs comparables, nous pouvons facilement voir combien une valeur est comparée à l'autre.

Il existe différentes méthodes de mise à l'échelle des données, dans ce TP/tutoriel, nous allons utiliser une méthode appelée normalisation.

La méthode de normalisation utilise cette formule : z = (x - u) / s où z est la nouvelle valeur, x est la valeur originale, u est la moyenne et s est l'écart-type.

Si vous prenez la colonne de poids de l'ensemble de données ci-dessus, la première valeur est 790, et la valeur mise à l'échelle sera : (790 - 1292.23) / 238.74 = -2.1

Si vous prenez la colonne volume de l'ensemble de données ci-dessus, la première valeur est 1,0 et la valeur mise à l'échelle sera : (1,0 - 1,61) / 0,38 = -1,59

Vous pouvez maintenant comparer -2.1 avec -1.59 au lieu de comparer 790 avec 1.0.

Vous n'avez pas à le faire manuellement, le module Python sklearn a une méthode appelée StandardScaler() qui retourne un objet Scaler avec des méthodes pour transformer des ensembles de données.

Exemple :

Mettre à l'échelle toutes les valeurs des colonnes Poids et Volume :

```
import pandas
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()

df = pandas.read_csv("data.csv")

X = df[['Weight', 'Volume']]

scaledX = scale.fit_transform(X)

print(scaledX)
```

Résultat : Notez que les deux premières valeurs sont -2.1 et -1.59, ce qui correspond à nos calculs :

```
[[-2.10389253 -1.59336644]
 [-0.55407235 -1.07190106]
```

```
[-1.52166278 -1.59336644]
[-1.78973979 -1.85409913]
[-0.63784641 -0.28970299]
[-1.52166278 -1.59336644]
[-0.76769621 -0.55043568]
[ 0.3046118  -0.28970299]
[-0.7551301  -0.28970299]
[-0.59595938 -0.0289703 ]
[-1.30803892 -1.33263375]
[-1.26615189 -0.81116837]
[-0.7551301  -1.59336644]
[-0.16871166 -0.0289703 ]
[ 0.14125238 -0.0289703 ]
[ 0.15800719 -0.0289703 ]
[ 0.3046118  -0.0289703 ]
[-0.05142797  1.53542584]
[-0.72580918 -0.0289703 ]
[ 0.14962979  1.01396046]
[ 1.2219378  -0.0289703 ]
[ 0.5685001   1.01396046]
[ 0.3046118   1.27469315]
[ 0.51404696 -0.0289703 ]
[ 0.51404696  1.01396046]
[ 0.72348212 -0.28970299]
[ 0.8281997   1.01396046]
[ 1.81254495  1.01396046]
[ 0.96642691 -0.0289703 ]
[ 1.72877089  1.01396046]
[ 1.30990057  1.27469315]
[ 1.90050772  1.01396046]
[-0.23991961 -0.0289703 ]
[ 0.40932938 -0.0289703 ]
[ 0.47215993 -0.0289703 ]
[ 0.4302729   2.31762392]]
```

Prédire les valeurs de CO2

La tâche dans le chapitre **Régression multiple** était de prédire les émissions de CO2 d'une voiture lorsque vous ne connaissiez que son poids et son volume.

Lorsque l'ensemble de données est mis à l'échelle, vous devez utiliser l'échelle lorsque vous prévoyez des valeurs :

Exemple

Prévoir les émissions de CO2 d'une voiture de 1,3 litre pesant 2300 kilogrammes :

```
import pandas
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()

df = pandas.read_csv("data.csv")
```

```
X = df[['Weight', 'Volume']]
y = df['CO2']

scaledX = scale.fit_transform(X)

regr = linear_model.LinearRegression()
regr.fit(scaledX, y)

scaled = scale.transform([[2300, 1,3]])

predictedCO2 = regr.predict([scaled[0]])
print(predictedCO2)
```

Et le résultat obtenu est : [97.07204485]

Machine Learning - Train/Test

Évaluer votre modèle

Dans l'apprentissage automatique, nous créons des modèles pour prédire le résultat de certains événements, comme dans le chapitre précédent où nous avons prédit les émissions de CO2 d'une voiture lorsque nous connaissions le poids et la taille du moteur.

Pour mesurer si le modèle est assez bon, nous pouvons utiliser une méthode appelée Train / Test.

Qu'est-ce que Train/Test

Train / Test est une méthode pour mesurer la précision de votre modèle.

Il est appelé Train / Test parce que vous divisez l'ensemble de données en deux ensembles : un ensemble de formation (entraînement) et un ensemble de test.

80% pour la formation (entraînement) et 20% pour les tests.

Vous entraînez le modèle en utilisant l'ensemble de formation.

Vous testez le modèle en utilisant le jeu de test.

Former le modèle signifie créer le modèle.

Tester le modèle signifie tester la précision du modèle.

Démarrer avec un ensemble de données

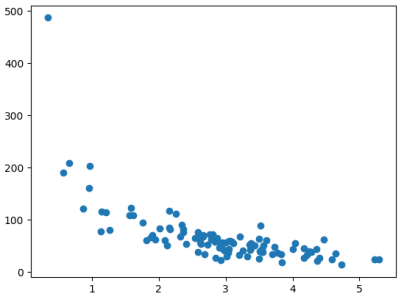
Commencez par un ensemble de données que vous souhaitez tester.

Notre ensemble de données illustre 100 clients dans un magasin et leurs habitudes d'achat.

Exemple :

```
import numpy
import matplotlib.pyplot as plt
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

plt.scatter(x, y)
plt.show()
```

Résultat : L'axe des x représente le nombre de minutes avant l'achat et l'axe des y représente le montant dépensé pour l'achat.



Machine Learning - Diviser en formation/essai

L'ensemble de formation devrait être une sélection aléatoire de 80 % des données originales.

L'ensemble de test devrait être les 20% restants.

```
train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]
```

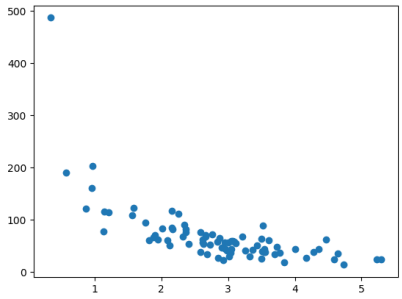
Afficher l'ensemble de formation

Afficher le même diagramme de dispersion avec l'ensemble de formation :

Exemple

```
plt.scatter(train_x, train_y)
plt.show()
```

Résultat : Cela ressemble à l'ensemble de données original, alors il semble que ce soit un choix équitable :



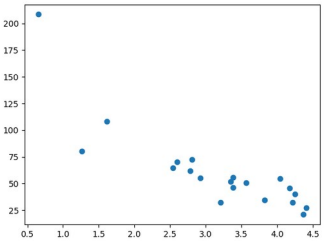
Afficher l'ensemble de tests

Pour nous assurer que l'ensemble de tests n'est pas complètement différent, nous examinerons également l'ensemble de tests.

Exemple :

```
plt.scatter(test_x, test_y)
plt.show()
```

Résultat : L'ensemble de tests ressemble également à l'ensemble de données d'origine :


200
175
150
125
100
75
50
25
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5

Arrêt du TP

Ajuster l'ensemble de données

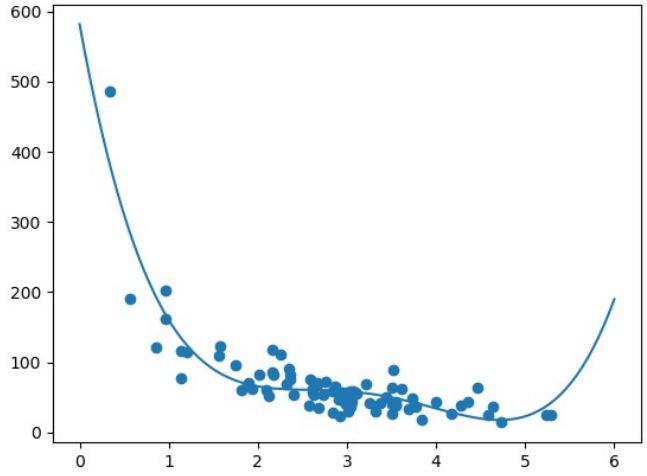
À quoi ressemble l'ensemble de données? À mon avis, je pense que le meilleur ajustement serait une régression polynomiale, alors dessinons une ligne de régression polynomiale.

Pour tracer une ligne à travers les points de données, nous utilisons la méthode plot() du module matplotlib :

Exemple : Tracer une ligne de régression polynomiale à travers les points de données :

```
import numpy
import matplotlib.pyplot as plt
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

myline = numpy.linspace(0, 6, 100)

plt.scatter(train_x, train_y)
plt.plot(myline, mymodel(myline))
plt.show()
```

et le résultat obtenu est :



Le résultat peut confirmer ma suggestion de l'ensemble de données correspondant à une régression polynomiale, même si cela nous donnerait des résultats étranges si nous essayons de prédire des valeurs en dehors de l'ensemble de données. Exemple : la ligne indique qu'un client passant 6 minutes dans le magasin ferait un achat d'une valeur de 200. C'est probablement un signe de sur-ajustement.

Mais qu'en est-il du score R-squared? Le score R-squared est un bon indicateur de la façon dont mon ensemble de données correspond au modèle.

R2

Vous vous souvenez de R2, aussi appelé R au carré ?

Il mesure la relation entre l'axe des x et l'axe des y, et la valeur varie de 0 à 1, où 0 signifie aucune relation, et 1 signifie totalement lié.

Le module sklearn a une méthode appelée r2_score() qui nous aidera à trouver cette relation.

Dans ce cas, nous aimerions mesurer la relation entre les minutes qu'un client reste dans la boutique et combien d'argent il dépense.

Exemple : Dans quelle mesure mes données d'entraînement s'intègrent-elles dans une régression polynomiale ?

```
import numpy
from sklearn.metrics import r2_score
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]
```

```
mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

r2 = r2_score(train_y, mymodel(train_x))

print(r2)
```

Remarque : Le résultat 0.799 indique qu'il existe une relation OK.

Portez l'ensemble des données dans notre modèle de tests

Maintenant, nous avons fait un modèle qui est OK, au moins quand il s'agit de données de formation.

Maintenant, nous voulons tester le modèle avec les données de test aussi, pour voir si nous donne le même résultat.

Exemple :

Trouvons le score R2 lorsque nous utilisons des données de test :

```
import numpy
from sklearn.metrics import r2_score
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

r2 = r2_score(test_y, mymodel(test_x))

print(r2)
```

Remarque : Le résultat 0.809 montre que le modèle correspond également à l'ensemble de tests, et nous sommes confiants que nous pouvons utiliser le modèle pour prédire les valeurs futures.

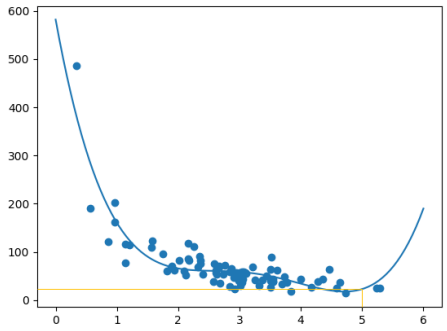
Prévoir les valeurs

Maintenant que nous avons établi que notre modèle est OK, nous pouvons commencer à prédire de nouvelles valeurs.

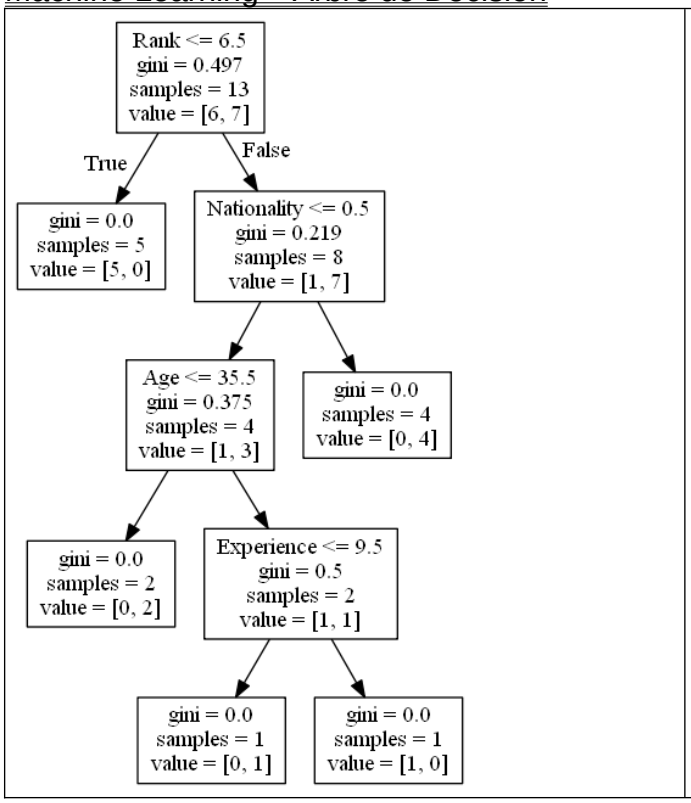
Exemple : Combien d'argent un client acheteur dépensera-t-il s'il reste dans le magasin pendant 5 minutes ?

```
print(mymodel(5))
```

L'exemple prévoyait que le client dépenserait 22,88 dollars, ce qui semble correspondre au diagramme :



Machine Learning – Arbre de Décision



Rank : rang
gini : c'est le nom d'une méthode
samples : échantillon
value : valeur

Arbre décisionnel

Dans ce chapitre, nous allons vous montrer comment faire un "arbre de décision". Un arbre de décision est un organigramme et peut vous aider à prendre des décisions en fonction de votre expérience antérieure.

Dans l'exemple, une personne essaiera de décider si elle / elle doit aller à un spectacle de comédie ou non.

Heureusement, notre exemple de personne s'est inscrit chaque fois qu'il y a eu un spectacle de comédie en ville, et a enregistré des informations sur le comédien, et aussi enregistré s'il / elle est allé ou non.

Age	Experience	Rank	Nationality	Go
36	10	9	UK	NO
42	12	4	USA	NO
23	4	6	N	NO
52	4	4	USA	NO
43	21	8	USA	YES
44	14	5	UK	NO
66	3	7	N	YES
35	14	9	UK	YES
52	13	7	N	YES
35	5	9	N	YES
24	3	5	USA	NO
18	3	7	UK	YES
45	9	9	UK	YES

Maintenant, sur la base de cet ensemble de données, Python peut créer un arbre de décision qui peut être utilisé pour décider si de nouvelles émissions valent la peine d'être suivies.

Comment cela fonctionne ?

En 1er lieu, il faut lire et afficher les données

```
import pandas

df = pandas.read_csv("data.csv")

print(df)
```

Pour créer un arbre de décision, toutes les données doivent être numériques.

Nous devons convertir les colonnes non numériques 'Nationalité' et 'Go' en valeurs numériques.

Pandas a une méthode **map()** qui prend un dictionnaire avec des informations sur la façon de convertir les valeurs.

{'UK' : 0, 'USA' : 1, 'N' : 2} signifie convertir les valeurs **'UK' à 0, 'USA' à 1, et 'N' à 2.**

Exemple : Changer les valeurs de chaîne en valeurs numériques :

```
d = {'UK': 0, 'USA': 1, 'N': 2}
df['Nationality'] = df['Nationality'].map(d)
d = {'YES': 1, 'NO': 0}
df['Go'] = df['Go'].map(d)

print(df)
```

Ensuite, nous devons séparer les colonnes d'entités de la colonne cible.

Les colonnes d'entités sont les colonnes que nous essayons de prédire, et la colonne cible est la colonne avec les valeurs que nous essayons de prédire.

Exemple : X représente les colonnes d'entités, y représente la colonne cible :

```
features = ['Age', 'Experience', 'Rank', 'Nationality']

X = df[features]
y = df['Go']

print(X)
print(y)
```

Nous pouvons maintenant créer l'arbre de décision, l'ajuster à nos détails. Commencez par importer les modules dont nous avons besoin :

Maintenant, nous pouvons créer et afficher notre arbre de décision :

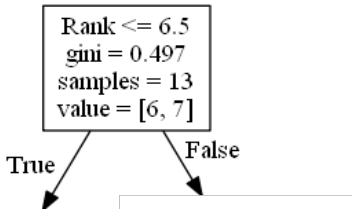
```
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt



df = pandas.read_csv("data.csv")
d = {'UK': 0, 'USA': 1, 'N': 2}
df['Nationality'] = df['Nationality'].map(d)
d = {'YES': 1, 'NO': 0}
df['Go'] = df['Go'].map(d)

#print(df)

features = ['Age', 'Experience', 'Rank', 'Nationality']

X = df[features]
y = df['Go']
'''
print ("\n Colonnes d'entité : ")
print(X)
print ("\n Colonne cible qui sera à prédire :")
print(y)
'''
dtree = DecisionTreeClassifier()
dtree = dtree.fit(X, y)

# Vos données de prédiction
data_to_predict = pd.DataFrame([[40, 10, 7, 1]], columns=features)

# Faire la prédiction
prediction = dtree.predict(data_to_predict)

print(prediction)
```

```
tree.plot_tree(dtree, feature_names=features)
plt.show()
#plt.savefig('tree.png')
```

Explication du résultat

L'arbre de décision utilise vos décisions antérieures pour calculer la probabilité que vous souhaitiez voir un comedien ou non.

Lisons les différents aspects de l'arbre de décision :



Rang

Rang <= 6.5 signifie que chaque comédien avec un rang de 6.5 ou moins suivra la flèche Vrai (à gauche), et le reste suivra la flèche Faux (à droite).

gini = 0,497 fait référence à la qualité de la division, et est toujours un nombre compris entre 0,0 et 0,5, où 0,0 signifierait que tous les échantillons ont obtenu le même résultat, et 0,5 signifierait que la division est effectuée exactement au milieu.

échantillons = 13 signifie qu'il reste 13 comédiens à ce stade de la décision, ce qui est le cas de tous puisque c'est la première étape.

valeur = [6, 7] signifie que de ces 13 comédiens, 6 auront un "NON", et 7 auront un "GO".

Gini

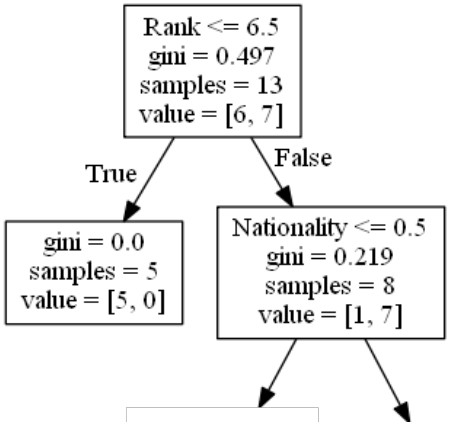
Il existe de nombreuses façons de diviser les échantillons, nous utilisons la méthode GINI dans ce tutoriel.

La méthode Gini utilise cette formule :

Gini = $1 - (x/n)^2 - (y/n)^2$

Où x est le nombre de réponses positives (« GO »), n est le nombre d'échantillons et y est le nombre de réponses négatives (« NO »), ce qui nous donne ce calcul :

$1 - (7 / 13)^2 - (6 / 13)^2 = 0,497$



L'étape suivante contient deux boîtes, une boîte pour les comédiens avec un 'rang' de 6,5 ou moins, et une boîte avec le reste.

Vrai - 5 comédiens Fin ici :

gini = 0,0 signifie que tous les échantillons ont obtenu le même résultat.

échantillons = 5 signifie qu'il reste 5 comédiens dans cette branche (5 comédiens avec un rang de 6,5 ou moins).

valeur = [5, 0] signifie que 5 obtiendra un "NON" et 0 obtiendra un "GO".

Faux - 8 comédiens Continuer :

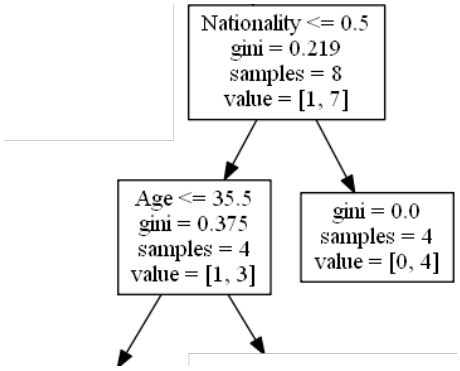
Nationalité

Nationalité <= 0,5 signifie que les comédiens avec une valeur de nationalité inférieure à 0,5 suivront la flèche vers la gauche (ce qui signifie tout le monde du Royaume-Uni,) et le reste suivra la flèche vers la droite.

gini = 0,219 signifie qu'environ 22 % des échantillons iraient dans une direction.

échantillons = 8 signifie qu'il reste 8 comédiens dans cette branche (8 comédiens avec un rang supérieur à 6,5).

valeur = [1, 7] signifie que de ces 8 comédiens, 1 aura un "NON" et 7 aura un "GO".



Vrai - 4 comédiens Continuer :

Âge

L'âge <= 35,5 signifie que les comédiens de 35,5 ans ou moins suivront la flèche vers la gauche, et le reste suivra la flèche vers la droite.

gini = 0,375 signifie qu'environ 37,5 % des échantillons iraient dans une direction.

échantillons = 4 signifie qu'il reste 4 comédiens dans cette branche (4 comédiens du Royaume-Uni).

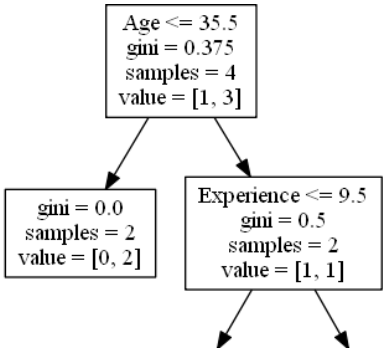
valeur = [1, 3] signifie que de ces 4 comédiens, 1 aura un "NON" et 3 aura un "GO".

Faux - 4 comédiens Fin ici :

gini = 0,0 signifie que tous les échantillons ont obtenu le même résultat.

échantillons = 4 signifie qu'il reste 4 comédiens dans cette branche (4 comédiens non du Royaume-Uni).

valeur = [0, 4] signifie que de ces 4 comédiens, 0 aura un "NON" et 4 aura un "GO".



Vrai - 2 comédiens Fin ici :

gini = 0,0 signifie que tous les échantillons ont obtenu le même résultat.

échantillons = 2 signifie qu'il reste 2 comédiens dans cette branche (2 comédiens à 35,5 ans ou moins).

valeur = [0, 2] signifie que de ces 2 comédiens, 0 aura un "NON" et 2 aura un "GO".

Faux - 2 comédiens Continuer :

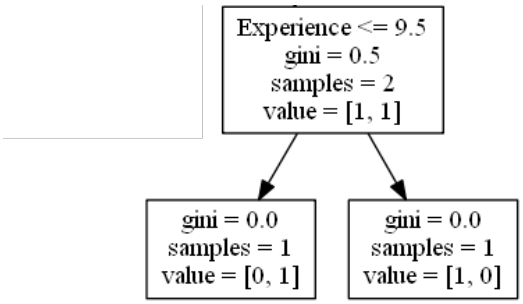
Expérience

Expérience <= 9,5 signifie que les comédiens avec 9,5 ans d'expérience, ou moins, suivront la flèche vers la gauche, et le reste suivra la flèche vers la droite.

gini = 0,5 signifie que 50 % des échantillons iraient dans une direction.

échantillons = 2 signifie qu'il reste 2 comédiens dans cette branche (2 comédiens de plus de 35,5 ans).

valeur = [1, 1] signifie que de ces 2 comédiens, 1 aura un "NON" et 1 aura un "GO".



Vrai - 1 comédien se termine ici :

gini = 0,0 signifie que tous les échantillons ont obtenu le même résultat.

échantillons = 1 signifie qu'il reste 1 comédien dans cette branche (1 comédien avec 9,5 ans d'expérience ou moins).

valeur = [0, 1] signifie que 0 obtiendra un "NON" et 1 obtiendra un "GO".

Faux - 1 comédien s'arrête ici :

gini = 0,0 signifie que tous les échantillons ont obtenu le même résultat.

échantillons = 1 signifie qu'il reste 1 comédien dans cette branche (1 comédien avec plus de 9,5 ans d'expérience).

valeur = [1, 0] signifie que 1 obtiendra un "NON" et 0 obtiendra un "GO".

Prédire les valeurs

Nous pouvons utiliser l'arbre de décision pour prédire de nouvelles valeurs.

Exemple : Dois-je aller voir un spectacle mettant en vedette un humoriste américain de 40 ans, avec 10 ans d'expérience, et un classement comique de 7 ?

Exemple : Utiliser la méthode predict() pour prédire de nouvelles valeurs :

```
print(dtree.predict([[40, 10, 7, 1]]))
```

Exemple : Quelle serait la réponse si le rang de comédie était 6 ?

```
print(dtree.predict([[40, 10, 6, 1]]))
```

Résultats différents

Vous verrez que l'arbre de décision vous donne des résultats différents si vous l'exécutez suffisamment de fois, même si vous l'alimentez avec les mêmes données.

C'est parce que l'arbre décisionnel ne nous donne pas une réponse sûre à 100 %. Il est fondé sur la probabilité d'un résultat, et la réponse variera.

→ Testez ce programme plusieurs fois et vous obtiendrez des résultats différents

Machine Learning – Matrice de Confusion

Qu'est-ce qu'une matrice de confusion ?

Il s'agit d'un tableau qui est utilisé dans les problèmes de classification pour évaluer les erreurs dans le modèle.

Les lignes représentent les classes réelles, les résultats auraient dû être. Tandis que les colonnes représentent les prédictions que nous avons faites. En utilisant ce tableau, il est facile de voir quelles prédictions sont fausses.

Créer une matrice de confusion

Les matrices de confusion peuvent être créées par des prédictions faites à partir d'une régression logistique.

Pour l'instant, nous allons générer des valeurs réelles et prévues en utilisant NumPy :

```
import numpy
```

Ensuite, nous devrons générer les nombres pour les valeurs "réelles" et "prévues".

```
actual = numpy.random.binomial(1, 0.9, size = 1000)
predicted = numpy.random.binomial(1, 0.9, size = 1000)
```

Afin de créer la matrice de confusion, nous devons importer des métriques à partir du module sklearn.

```
from sklearn import metrics
```

Une fois les métriques importées, nous pouvons utiliser la fonction de matrice de confusion sur nos valeurs réelles et prévues.

```
confusion_matrix = metrics.confusion_matrix(actual, predicted)
```

Pour créer un affichage visuel plus interprétable, nous devons convertir le tableau en un affichage matriciel de confusion.

```
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix =
confusion_matrix, display_labels = [False, True])
```

Pour visualiser l'affichage, nous devons importer pyplot à partir de matplotlib.

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

Enfin, pour afficher le tracé, nous pouvons utiliser les fonctions plot() et show() de pyplot.

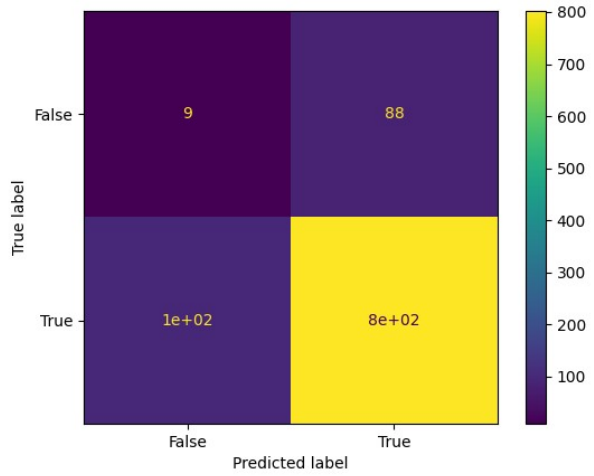
```
cm_display.plot()
plt.show()
```

Regardons le programme en entier en fonctionnement :

```
import matplotlib.pyplot as plt
import numpy
from sklearn import metrics

actual = numpy.random.binomial(1,.9,size = 1000)
predicted = numpy.random.binomial(1,.9,size = 1000)
```

```
confusion_matrix = metrics.confusion_matrix(actual, predicted)

cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix =
confusion_matrix, display_labels = [False, True])

cm_display.plot()
plt.show()
```

et ce programme donne le résultat suivant :



Résultats expliqués

La matrice de confusion créée comporte quatre quadrants différents :

Faux négatif (quadrant supérieur gauche)

Faux positif (quadrant supérieur droit)

Vrai négatif (quadrant inférieur gauche)

Vrai positif (quadrant inférieur droit)

Vrai signifie que les valeurs ont été prédites avec précision, Faux signifie qu'il y avait une erreur ou une mauvaise prédiction.

Maintenant que nous avons créé une matrice de confusion, nous pouvons calculer différentes mesures pour quantifier la qualité du modèle.

Mesures créées

La matrice nous fournit de nombreuses mesures utiles qui nous aident à évaluer le modèle de classification.

Les différentes mesures comprennent : la précision, la précision, la sensibilité (rappel), la spécificité et le score F, expliqué ci-dessous.

Précision

La précision mesure la fréquence à laquelle le modèle est correct.

Comment calculer

(Vrai positif + Vrai négatif) / Prédictions totales

Exemple

Exactitude = métriques.accuracy_score(réel, prévu)

Précision :

Parmi les résultats positifs prévus, quel pourcentage est vraiment positif?

Comment calculer

Vrai positif / (Vrai positif + Faux positif)

Precision n'évalue pas correctement les cas négatifs prévus :

Exemple : Précision = métriques.precision_score(réel, prévu)

Sensibilité (rappel)

De tous les cas positifs, quel pourcentage est prévu positif?

La sensibilité (parfois appelée rappel) mesure la capacité du modèle à prédire les résultats positifs.

Cela signifie qu'il examine les vrais positifs et les faux négatifs (qui sont des positifs qui ont été prédits à tort comme négatifs).

Comment calculer :

Vrai positif / (Vrai positif + Faux négatif)

La sensibilité est bonne pour comprendre dans quelle mesure le modèle prédit que quelque chose est positif :

Exemple : Sensitivity_recall = metrics.recall_score(actual, predicted)

Spécificité

Dans quelle mesure le modèle est-il capable de prédire des résultats négatifs?

La spécificité est similaire à la sensibilité, mais l'examine à partir de la persistance de résultats négatifs.

Comment calculer

Vrai négatif / (Vrai négatif + Faux positif)

Comme il s'agit du contraire de Recall, nous utilisons la fonction recall_score, en prenant l'étiquette de position opposée :

Exemple :

```
Spécificité = metrics.recall_score(actual, predicted, pos_label=0)
```

F-score :

F-score est la "moyenne harmonique" de précision et de sensibilité.

Il considère à la fois les faux positifs et les faux négatifs et est bon pour les ensembles de données déséquilibrés.

Comment calculer

2 * ((Précision * Sensibilité) / (Précision + Sensibilité))

Ce score ne tient pas compte des valeurs True Negative :

Exemple : F1_score = metrics.f1_score(réel, prévu)

Tous les calculs en un seul calcul :

```
#metrics
print({"Accuracy":Accuracy,"Precision":Precision,"Sensitivity_recall":Sensitivity_recall,"Specificity":Specificity,"F1_score":F1_score})
```

Voici le programme complet :

```
import matplotlib.pyplot as plt

import numpy

from sklearn import metrics


actual = numpy.random.binomial(1,.9,size = 1000)

predicted = numpy.random.binomial(1,.9,size = 1000)


confusion_matrix = metrics.confusion_matrix(actual, predicted)


cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix,
display_labels = [False, True])
```

```
cm_display.plot()
plt.show()
Spécificité = metrics.recall_score(actual, predicted, pos_label=0)
print(Spécificité)

Accuracy = metrics.accuracy_score(actual, predicted)
Precision = metrics.precision_score(actual, predicted)
Sensitivity_recall = metrics.recall_score(actual, predicted)
Specificity = metrics.recall_score(actual, predicted, pos_label=0)
F1_score = metrics.f1_score(actual, predicted)

#metrics
print({"Accuracy":Accuracy,"Precision":Precision,"Sensitivity_recall":Sensitivity_recall,"Specificit
y":Specificity,"F1_score":F1_score})
```

Regroupement hiérarchique

Le regroupement hiérarchique est une méthode d'apprentissage non supervisée pour regrouper les points de données. L'algorithme construit des clusters en mesurant les différences entre les données. L'apprentissage non supervisé signifie qu'un modèle n'a pas besoin d'être formé et que nous n'avons pas besoin d'une variable "cible". Cette méthode peut être utilisée sur toutes les données pour visualiser et interpréter la relation entre les points de données individuels.

Ici, nous utiliserons le clustering hiérarchique pour regrouper les points de données et visualiser les clusters à l'aide d'un dendrogramme et d'un nuage de points.

Comment ça se passe ?

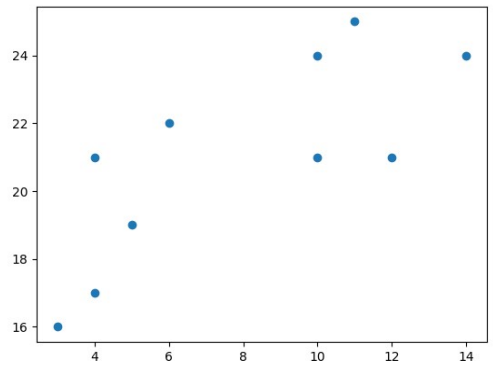
Nous utiliserons Agglomerative Clustering, un type de regroupement hiérarchique qui suit une approche ascendante. Nous commençons par traiter chaque point de données comme son propre cluster. Ensuite, nous joignons des clusters qui ont la distance la plus courte entre eux pour créer des clusters plus grands. Cette étape est répétée jusqu'à ce qu'un grand cluster soit formé contenant tous les points de données.

La classification hiérarchique nous oblige à décider à la fois d'une méthode de distance et de liaison. Nous utiliserons la distance euclidienne et la méthode de liaison de Ward, qui tente de minimiser la variance entre les grappes.

Exemple : Commencez par visualiser certains points de données :

```
import numpy as np
import matplotlib.pyplot as plt

x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

plt.scatter(x, y)
plt.show()
```

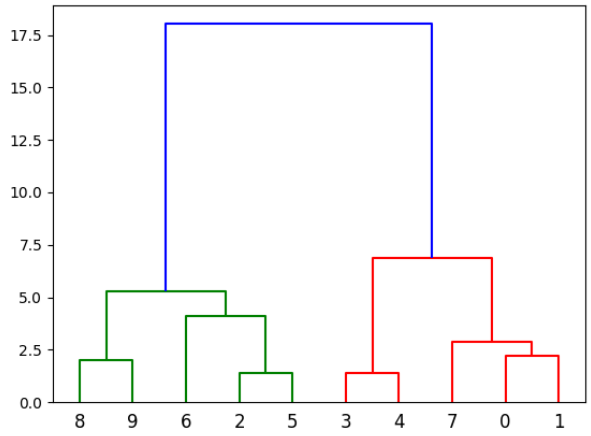
et, comme le savez, le résultat donne celui-ci :



Maintenant, nous allons utiliser la méthode « ward » utilisant la distance euclidéenne et en la visualisant en utilisant le Dendrogramme

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage

x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

data = list(zip(x, y))

linkage_data = linkage(data, method='ward', metric='euclidean')
dendrogram(linkage_data)

plt.show()
```

et le résultat est :

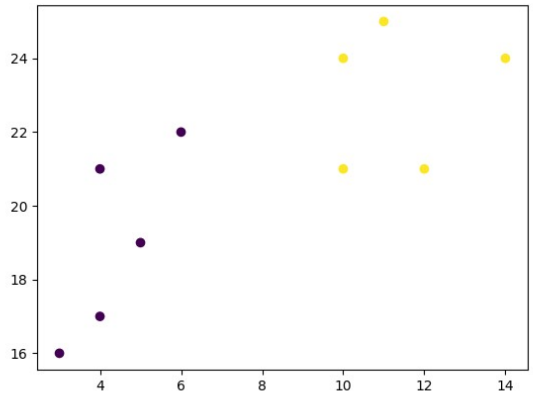


Ici, nous faisons la même chose avec la bibliothèque scikit-learn de Python. Ensuite, visualisez sur un tracé bidimensionnel :

Exemple :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering

x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

data = list(zip(x, y))

hierarchical_cluster = AgglomerativeClustering(n_clusters=2,
affinity='euclidean', linkage='ward')
labels = hierarchical_cluster.fit_predict(data)

plt.scatter(x, y, c=labels)
plt.show()
```

et le résultat est :



Exemple expliqué :

Importons les modules nécessaires :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering
```

scikit-learn est une bibliothèque populaire pour l'apprentissage automatique.

Créez des tableaux qui ressemblent à deux variables dans un jeu de données. Notez que même si nous n'utilisons que deux variables ici, cette méthode fonctionnera avec n'importe quel nombre de variables :

```
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
```

Transformer les données en un ensemble de points :

```
data = list(zip(x, y))
print(data)
```

et le résultat est :

[(4, 21), (5, 19), (10, 24), (4, 17), (3, 16), (11, 25), (14, 24), (6, 22), (10, 21), (12, 21)]

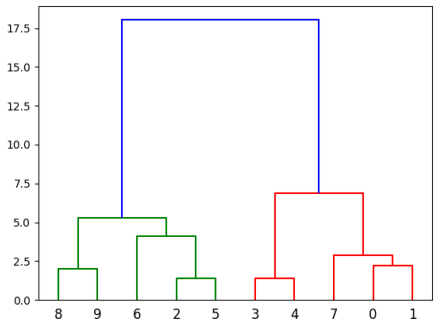
Calculez le lien entre tous les différents points. Nous utilisons ici une simple mesure de distance euclidienne et la liaison de Ward, qui cherche à minimiser la variance entre les clusters.

```
linkage_data = linkage(data, method='ward', metric='euclidean')
```

Tout d'abord, tracez les résultats dans un dendrogramme. Ce graphique nous montrera la hiérarchie des clusters du bas (points individuels) au haut (un seul cluster composé de tous les points de données).

plt.show() nous permet de visualiser le dendrogramme au lieu de seulement les données de liaison brutes.

Et le résultat est alors :



La bibliothèque scikit-learn nous permet d'utiliser le clustering hiérarchique d'une manière différente. Tout d'abord, nous initialisons la classe AgglomerativeClustering avec 2 clusters, en utilisant la même distance euclidienne et la liaison de Ward.

```
hierarchical_cluster = AgglomerativeClustering(n_clusters=2,
affinity='euclidean', linkage='ward')
```

La méthode .fit_predict peut être appelée sur nos données pour calculer les clusters en utilisant les paramètres définis à travers notre nombre de clusters choisis

```
labels = hierarchical_cluster.fit_predict(data)
print(labels)
```

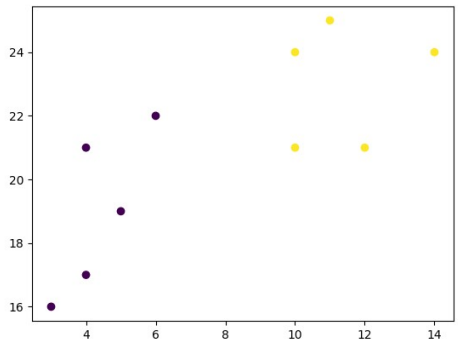
et le résultat est :

[0 0 1 0 0 1 1 0 1 1]

Enfin, si nous traçons les mêmes données et colorions les points en utilisant les étiquettes attribuées à chaque index par la méthode de classification hiérarchique, nous pouvons voir le cluster auquel chaque point a été affecté :

```
plt.scatter(x, y, c=labels)
plt.show()
```

et le résultat est :



Voici l'explication détaillée du code :

Dans le code ci-dessus, la variable `labels` joue un rôle clé dans le processus de clustering hiérarchique agglomératif utilisant la bibliothèque `scikit-learn`.

Voici une explication étape par étape :

1. **Importation des bibliothèques nécessaires** : Le code commence par importer `numpy`, `matplotlib.pyplot`, et la classe `AgglomerativeClustering` de `scikit-learn`. Ces bibliothèques sont utilisées pour le traitement des données, la visualisation, et l'exécution du clustering hiérarchique, respectivement.
2. **Préparation des données** : Les listes `x` et `y` contiennent des coordonnées de points dans un espace bidimensionnel. Ces points sont combinés en une liste de tuples appelée `data`, où chaque tuple représente les coordonnées d'un point.
3. **Configuration du modèle de clustering** : Un objet `AgglomerativeClustering` est créé avec des paramètres spécifiques (`n_clusters=2`, `affinity='euclidean'`, `linkage='ward'`). Ces paramètres définissent le nombre de clusters à former (plus on défini de clusters, plus on obtient des regroupements représentatifs), la méthode utilisée pour calculer la distance entre les points (euclidienne dans ce cas), et le critère de liaison (Ward) pour déterminer comment les clusters sont formés.
4. **Fitting du modèle et prédiction** : La méthode `fit_predict` est appelée sur l'objet `hierarchical_cluster` avec les données `data` comme argument. Cette méthode effectue le clustering sur les données et renvoie un tableau de labels. Chaque élément dans le tableau `labels` correspond à un point de données dans `data`, indiquant le cluster auquel ce point a été attribué. Puisque `n_clusters=2`, il y aura deux clusters distincts, souvent représentés par les valeurs 0 et 1 dans le tableau `labels`.
5. **Visualisation des clusters** : Enfin, le code utilise `matplotlib.pyplot` pour tracer les points de données. La couleur de chaque point (`c=labels`) est déterminée par le label de cluster qui lui a été attribué, permettant une visualisation claire de la manière dont les points ont été groupés dans l'espace bidimensionnel.

En résumé, la variable `labels` contient les résultats du clustering, indiquant à quel cluster chaque point appartient. Cette information est utilisée pour colorier les points lors de la visualisation, illustrant ainsi les groupes formés par le processus de clustering hiérarchique

Machine Learning – Régression Logistique
Régression logistique

La régression logistique vise à résoudre les problèmes de classification. Il le fait en prédisant des résultats catégoriques, contrairement à la régression linéaire qui prédit un résultat continu.

Dans le cas le plus simple il y a deux résultats, qui est appelé binomial, dont un exemple est de prédire si une tumeur est maligne ou bénigne. D'autres cas ont plus de deux résultats à classer, dans ce cas, il est appelé multinomial. Un exemple courant de régression logistique multinomiale serait de prédire la classe d'une fleur d'iris entre 3 espèces différentes.

Ici, nous utiliserons la régression logistique de base pour prédire une variable binomiale. Cela signifie qu'elle n'a que deux résultats possibles.

Comment ça se passe ?

En Python, nous avons des modules qui feront le travail pour nous. Commencez par importer le module NumPy.

```
import numpy
```

Stockez les variables indépendantes dans X.

Stockez la variable dépendante dans y.

Voici un exemple d'ensemble de données :

```
#X represents the size of a tumor in centimeters.
X = numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37, 4.96, 4.52, 3.69, 5.88]).reshape(-1,1)

#Note: X has to be reshaped into a column from a row for the LogisticRegression() function to work.
#y represents whether or not the tumor is cancerous (0 for "No", 1 for "Yes").
y = numpy.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])
```

Nous allons utiliser une méthode du module sklearn, donc nous devrons également importer ce module :

```
from sklearn import linear_model
```

À partir du module sklearn, nous utiliserons la méthode LogisticRegression() pour créer un objet de régression logistique.

Cet objet a une méthode appelée fit() qui prend les valeurs indépendantes et dépendantes comme paramètres et remplit l'objet de régression avec des données qui décrivent la relation :

```
logr = linear_model.LogisticRegression()
logr.fit(X,y)
```

Nous avons maintenant un objet de régression logistique qui est prêt à déterminer si une tumeur est cancéreuse en fonction de la taille de la tumeur :

```
#predict if tumor is cancerous where the size is 3.46mm:
predicted = logr.predict(numpy.array([3.46]).reshape(-1,1))
```

Exemple : Regardez le programme complet :

```
import numpy
from sklearn import linear_model

#Reshaped for Logistic function.
X = numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37,
4.96, 4.52, 3.69, 5.88]).reshape(-1,1)
y = numpy.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

logr = linear_model.LogisticRegression()
logr.fit(X,y)

#predict if tumor is cancerous where the size is 3.46mm:
predicted = logr.predict(numpy.array([3.46]).reshape(-1,1))
print(predicted)
```

et le résultat est 0

Nous avons prédit qu'une tumeur de 3,46 mm ne sera pas cancéreuse.

Coefficient

Dans la régression logistique, le coefficient est le changement attendu des probabilités logarithmiques d'avoir le résultat par changement d'unité dans X.

Cela n'a pas la compréhension la plus intuitive, donc nous allons l'utiliser pour créer quelque chose qui a plus de sens, les chances.

Exemple : Voir l'exemple complet en action :

```
import numpy
from sklearn import linear_model

#Reshaped for Logistic function.
X = numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37,
4.96, 4.52, 3.69, 5.88]).reshape(-1,1)
y = numpy.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

logr = linear_model.LogisticRegression()
logr.fit(X,y)
```

```
log_odds = logr.coef_
odds = numpy.exp(log_odds)

print(odds)
```

et le résultat obtenu est : [4.03541657]

Cela nous indique que lorsque la taille d'une tumeur augmente de 1 mm, les chances qu'elle soit cancéreuse augmentent de 4 fois.

coef_ndarray of shape (1, n_features) or (n_classes, n_features)
Coefficient of the features in the decision function.

`coef_` is of shape (1, n_features) when the given problem is binary.

Probabilité

Les valeurs de coefficient et d'interception peuvent être utilisées pour trouver la probabilité que chaque tumeur soit cancéreuse.

Créez une fonction qui utilise le coefficient du modèle et les valeurs d'interception pour retourner une nouvelle valeur. Cette nouvelle valeur représente la probabilité que l'observation donnée soit une tumeur :

```
def logit2prob(logr,x):
  log_odds = logr.coef_ * x + logr.intercept_
  odds = numpy.exp(log_odds)
  probability = odds / (1 + odds)
  return(probability)
```

Fonction expliquée

Pour trouver les « log_odds » pour chaque observation, nous devons d'abord créer une formule qui ressemble à celle de la régression linéaire, en extrayant le coefficient et l'interception.

log_odds = logr.coef_ * x + logr.intercept_

Pour ensuite convertir les log-odds en cotes, nous devons les exposer.

odds = numpy.exp(log_odds)

Maintenant que nous avons la cote, nous pouvons la convertir en probabilité en la divisant par 1 plus la cote.

probability = odds / (1 + odds)

Utilisons maintenant la fonction avec ce que nous avons appris pour découvrir la probabilité que chaque tumeur soit cancéreuse.

Exemple : Voir l'exemple complet en action :

```
import numpy
from sklearn import linear_model
```

```
X = numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37, 4.96,
4.52, 3.69, 5.88]).reshape(-1,1)
y = numpy.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

logr = linear_model.LogisticRegression()
logr.fit(X,y)

def logit2prob(logr, X):
 log_odds = logr.coef_ * X + logr.intercept_
 odds = numpy.exp(log_odds)
 probability = odds / (1 + odds)
 return(probability)

print(logit2prob(logr, X))
```

et le résultat obtenu est :

```
[[0.60749955]
  [0.19268876]
  [0.12775886]
  [0.00955221]
  [0.08038616]
  [0.07345637]
  [0.88362743]
  [0.77901378]
  [0.88924409]
  [0.81293497]
  [0.57719129]
  [0.96664243]]
```

Résultats expliqués

3,78 : 0,61 La probabilité qu'une tumeur de la taille 3,78 cm soit cancéreuse est de 61%.

2,44 : 0,19 La probabilité qu'une tumeur de la taille 2,44 cm soit cancéreuse est de 19%.

2.09 : 0.13 La probabilité qu'une tumeur de taille 2.09cm soit cancéreuse est de 13%.

….

Ne pas faire

Machine Learning – Quadrillage (Grid Search)
Recherche dans la grille

La majorité des modèles d'apprentissage automatique contiennent des paramètres qui peuvent être ajustés pour modifier la façon dont le modèle apprend. Par exemple, le modèle de régression logistique, à partir de sklearn, a un paramètre C qui contrôle la régularisation, ce qui affecte la complexité du modèle.

Comment choisir la meilleure valeur pour C? La meilleure valeur dépend des données utilisées pour former le modèle.

Comment ça se passe ?

Une méthode consiste à essayer différentes valeurs, puis à choisir la valeur qui donne le meilleur score. Cette technique est connue sous le nom de recherche par grille. Si nous devions sélectionner les valeurs pour deux paramètres ou plus, nous évaluerions toutes les combinaisons des ensembles de valeurs formant ainsi une grille de valeurs.

Avant d'entrer dans l'exemple, il est bon de savoir ce que fait le paramètre que nous changeons. Des valeurs plus élevées de C indiquent le modèle, les données d'entraînement ressemblent à des informations du monde réel, accordent plus de poids aux données d'entraînement. Alors que les valeurs inférieures de C font le contraire.

Utilisation des paramètres par défaut

Tout d'abord, voyons quel type de résultats nous pouvons générer sans recherche de grille en utilisant uniquement les paramètres de base.

Pour commencer, nous devons d'abord charger l'ensemble de données avec lequel nous travaillerons.

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

Ensuite, pour créer le modèle, nous devons avoir un ensemble de variables indépendantes X et une variable dépendante y.

```
X = iris['data']
y = iris['target']
```

Nous allons maintenant charger le modèle logistique pour classer les fleurs d'iris.

```
from sklearn.linear_model import LogisticRegression
```

Création du modèle, réglage de max_iter sur une valeur plus élevée pour s'assurer que le modèle trouve un résultat.

Gardez à l'esprit que la valeur par défaut pour C dans un modèle de régression logistique est 1, nous comparerons cela plus tard.

Dans l'exemple ci-dessous, nous examinons l'ensemble de données de l'iris et essayons de former un modèle avec des valeurs variables pour C dans la régression logistique.

```
logit = LogisticRegression(max_iter = 10000)
```

Après avoir créé le modèle, nous devons l'adapter aux données.

```
print(logit.fit(X,y))
```

Pour évaluer le modèle, nous utilisons la méthode de score.

```
print(logit.score(X,y))
```

Exemple :

```
from sklearn import datasets
from sklearn.linear_model import LogisticRegression

iris = datasets.load_iris()

X = iris['data']
y = iris['target']

logit = LogisticRegression(max_iter = 10000)

print(logit.fit(X,y))

print(logit.score(X,y))
```

Avec le réglage par défaut de C = 1, nous avons obtenu un score de 0,973.

Voyons si nous pouvons faire mieux en implémentant une recherche par grille avec des valeurs de différence de 0,973.

Mise en œuvre de la recherche par grille

Nous suivrons les mêmes étapes que précédemment, sauf que cette fois, nous allons définir une plage de valeurs pour C.

Savoir quelles valeurs définir pour les paramètres recherchés prendra une combinaison de connaissance du domaine et de la pratique.

Puisque la valeur par défaut pour C est 1, nous allons définir une plage de valeurs qui l'entoure.

```
C = [0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]
```

Ensuite, nous allons créer une boucle for pour changer les valeurs de C et évaluer le modèle à chaque changement.

Nous allons d'abord créer une liste vide pour stocker le score.

scores = []

Pour modifier les valeurs de C, nous devons boucler la plage de valeurs et mettre à jour le paramètre à chaque fois.

```
for choice in C:
 logit.set_params(C=choice)
 logit.fit(X, y)
 scores.append(logit.score(X, y))
```

Avec les scores stockés dans une liste, nous pouvons évaluer quel est le meilleur choix de C.

```
print(scores)
```

Exemple :

```
from sklearn import datasets
from sklearn.linear_model import LogisticRegression

iris = datasets.load_iris()

X = iris['data']
y = iris['target']

logit = LogisticRegression(max_iter = 10000)

C = [0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]

scores = []

for choice in C:
 logit.set_params(C=choice)
 logit.fit(X, y)
 scores.append(logit.score(X, y))

print(scores)
```

Résultats expliqués

Nous pouvons voir que les valeurs inférieures de C ont été pires que le paramètre de base de 1. Cependant, lorsque nous avons augmenté la valeur de C à 1,75, le modèle a connu une précision accrue.

Il semble que l'augmentation de C au-delà de ce montant n'aide pas à augmenter la précision du modèle.

Note sur les bonnes pratiques

Nous avons évalué notre modèle de régression logistique en utilisant les mêmes données qui ont été utilisées pour l'entraîner. Si le modèle correspond trop étroitement à ces données, il n'est peut-être pas très efficace pour prédire des données invisibles. Cette erreur statistique est connue sous le nom de sur-ajustement.

Pour éviter d'être induit en erreur par les scores sur les données de formation, nous pouvons mettre de côté une partie de nos données et les utiliser spécifiquement pour tester le modèle. Se reporter à l'exposé sur la séparation des trains et des essais pour éviter d'être induit en erreur et d'être trop ajusté.

Ne pas faire

Machine Learning - Prétraitement - Données catégoriques

Données catégoriques

Lorsque vos données ont des catégories représentées par des chaînes, il sera difficile de les utiliser pour former des modèles d'apprentissage automatique qui n'acceptent souvent que des données numériques.

Au lieu d'ignorer les données catégoriques et d'exclure les informations de notre modèle, vous pouvez transformer les données afin qu'elles puissent être utilisées dans vos modèles.

Regardez le tableau ci-dessous, c'est le même jeu de données que nous avons utilisé dans le chapitre sur la régression multiple.

Exemple :

```
import pandas as pd

cars = pd.read_csv('data.csv')
print(cars.to_string())
```

et le résultat obtenu est :

```
          Car      Model  Volume  Weight  CO2
0      Toyoty       Aygo    1000     790   99
1   Mitsubishi  Space Star    1200    1160   95
2       Skoda     Citigo    1000     929   95
3        Fiat        500     900     865   90
4        Mini     Cooper    1500    1140  105
5          VW        Up!    1000     929  105
6       Skoda      Fabia    1400    1109   90
7    Mercedes    A-Class    1500    1365   92
8        Ford     Fiesta    1500    1112   98
9        Audi         A1    1600    1150   99
10    Hyundai        I20    1100     980   99
11     Suzuki      Swift    1300     990  101
12       Ford     Fiesta    1000    1112   99
13      Honda      Civic    1600    1252   94
14     Hundai        I30    1600    1326   97
15       Opel      Astra    1600    1330   97
16        BMW          1    1600    1365   99
17      Mazda          3    2200    1280  104
18      Skoda      Rapid    1600    1119  104
19       Ford      Focus    2000    1328  105
20       Ford     Mondeo    1600    1584   94
21       Opel    Insignia    2000    1428   99
22   Mercedes    C-Class    2100    1365   99
23      Skoda    Octavia    1600    1415   99
24      Volvo        S60    2000    1415   99
25   Mercedes         CLA    1500    1465  102
26       Audi         A4    2000    1490  104
27       Audi         A6    2000    1725  114
28      Volvo        V70    1600    1523  109
29        BMW          5    2000    1705  114
30   Mercedes    E-Class    2100    1605  115
31      Volvo       XC70    2000    1746  117
32       Ford      B-Max    1600    1235  104
```

```
33        BMW        216    1600    1390  108
34       Opel      Zafira    1600    1405  109
35    Mercedes        SLK    2500    1395  120
```

Dans le chapitre sur la régression multiple, nous avons essayé de prédire le CO2 émis en fonction du volume du moteur et du poids de la voiture, mais nous avons exclu les informations sur la marque et le modèle de la voiture.

Les informations sur la marque ou le modèle de voiture pourraient nous aider à mieux prédire les émissions de CO2.

Un codage à chaud

Nous ne pouvons pas utiliser la colonne Voiture ou Modèle dans nos données car elles ne sont pas numériques. Une relation linéaire entre une variable catégorique, Voiture ou Modèle, et une variable numérique, CO2, ne peut pas être déterminée.

Pour résoudre ce problème, nous devons avoir une représentation numérique de la variable catégorique. Une façon de le faire est d'avoir une colonne représentant chaque groupe dans la catégorie.

Pour chaque colonne, les valeurs seront 1 ou 0 où 1 représente l'inclusion du groupe et 0 représente l'exclusion. Cette transformation est appelée un encodage à chaud.

Vous n'avez pas à le faire manuellement, le module Python Pandas a une fonction appelée get_dummies() qui fait un encodage à chaud.

Découvrez le module Pandas dans notre tutoriel Pandas.

Exemple : Un code à chaud pour la colonne de voiture :

```
import pandas as pd

cars = pd.read_csv('data.csv')
ohe_cars = pd.get_dummies(cars[['Car']])

print(ohe_cars.to_string())
```

et le résultat est :

```
    Car_Audi  Car_BMW  Car_Fiat  Car_Ford  Car_Honda  Car_Hundai  Car_Hyundai  Car_Mazda  Car_Mercedes  Car_Mini  Car_Mitsubishi  Car_Opel  Car_Skoda  Car_Suzuki  Car_Toyoty  Car_VW  Car_Volvo
0         0        0         0         0          0           0            0          0             0         1               0         0          0           0       0         0
1         0        0         0         0          0           0            0          1             0         0               0         0          0           0       0         0
2         0        0         0         0          0           0            0          0             1         0               0         0          0           0       0         0
3         0        0         1         0          0           0            0          0             0         0               0         0          0           0       0         0
4         0        0         0         0          0           0            1          0             0         0               0         0          0           0       0         0
5         0        0         0         0          0           0            0          0             0         0               1         0          0           0       0         0
6         0        0         0         0          0           0            0          0             1         0               0         0          0           0       0         0
7         0        0         0         0          0           1            0          0             0         0               0         0          0           0       0         0
8         0        0         0         1          0           0            0          0             0         0               0         0          0           0       0         0
9         1        0         0         0          0           0            0          0             0         0               0         0          0           0       0         0
10        0        0         0         0          0           1            0          0             0         0               0         0          0           0       0         0
11        0        0         0         0          0           0            0          0             0         1               0         0          0           0       0         0
12        0        0         0         1          0           0            0          0             0         0               0         0          0           0       0         0
13        0        0         0         0          1           0            0          0             0         0               0         0          0           0       0         0
```

14	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
16	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
19	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
25	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
26	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
29	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
32	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
35	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

Résultats

Une colonne a été créée pour chaque marque de voiture dans la colonne Voiture.

Prédire le CO2

Nous pouvons utiliser ces informations supplémentaires avec le volume et le poids pour prédire le CO2

Pour combiner les informations, nous pouvons utiliser la fonction concat() de pandas.

Nous allons d'abord devoir importer quelques modules.

Nous commencerons par importer les pandas.

```
import pandas
```

Le module pandas permet de lire des fichiers csv et de manipuler des objets DataFrame :

```
cars = pandas.read_csv("data.csv")
```

Il permet également de créer les variables fictives (dummy) :

```
ohe_cars = pandas.get_dummies(cars[['Car']])
```

Ensuite, nous devons sélectionner les variables indépendantes (X) et ajouter les variables factices en colonne.

Stockez également la variable dépendante dans y.

```
X = pandas.concat([cars[['Volume', 'Weight']], ohe_cars], axis=1)
y = cars['CO2']
```

Nous devons également importer une méthode de sklearn pour créer un modèle linéaire

```
from sklearn import linear_model
```

Nous pouvons maintenant ajuster les données à une régression linéaire :

```
regr = linear_model.LinearRegression()
regr.fit(X,y)
```

Enfin, nous pouvons prédire les émissions de CO2 en fonction du poids, du volume et du fabricant de la voiture.

##predict the CO2 emission of a Volvo where the weight is 2300kg, and the volume is 1300cm3:
predictedCO2 = regr.predict([[2300, 1300,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]])

Exemple :

```
import pandas
from sklearn import linear_model

cars = pandas.read_csv("data.csv")
ohe_cars = pandas.get_dummies(cars[['Car']])

X = pandas.concat([cars[['Volume', 'Weight']], ohe_cars], axis=1)
y = cars['CO2']

regr = linear_model.LinearRegression()
regr.fit(X,y)

##predict the CO2 emission of a Volvo where the weight is 2300kg,
and the volume is 1300cm3:
predictedCO2 = regr.predict([[2300,
1300,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]])

print(predictedCO2)
```

et le résultat est : [122.45153299]

Nous avons maintenant un coefficient pour le volume, le poids et chaque marque de voiture dans l'ensemble de données

Il n'est pas nécessaire de créer une colonne pour chaque groupe de votre catégorie. L'information peut être conservée en utilisant 1 colonne de moins que le nombre de groupes que vous avez.

Par exemple, vous avez une colonne représentant les couleurs et dans cette colonne, vous avez deux couleurs, rouge et bleu.

Exemple

```
import pandas as pd

colors = pd.DataFrame({'color': ['blue', 'red']})

print(colors)
```

et le résultat est :

	color
0	blue
1	red

Vous pouvez créer 1 colonne appelée rouge où 1 représente rouge et 0 représente non rouge, ce qui signifie qu'elle est bleue.

Pour ce faire, nous pouvons utiliser la même fonction que celle utilisée pour un encodage à chaud, get_dummies, puis supprimer une des colonnes. Il y a un argument, drop_first, qui nous permet d'exclure la première colonne de la table résultante.

Exemple

```
import pandas as pd

colors = pd.DataFrame({'color': ['blue', 'red']})
dummies = pd.get_dummies(colors, drop_first=True)

print(dummies)
```

et le résultat est :

	color_red
0	0
1	1

Que se passe-t-il si vous avez plus de deux groupes? Comment les groupes multiples peuvent-ils être représentés par une colonne de moins?

Disons que nous avons trois couleurs cette fois, le rouge, le bleu et le vert. Lorsque nous get_dummies en laissant tomber la première colonne, nous obtenons le tableau suivant.

Exemple :

```
import pandas as pd

colors = pd.DataFrame({'color': ['blue', 'red', 'green']})
dummies = pd.get_dummies(colors, drop_first=True)
dummies['color'] = colors['color']

print(dummies)
```

et le résultat est le suivant :

	color_green	color_red	color
0	0	0	blue
1	0	1	red
2	1	0	green

à faire (partie K-means)

Machine Learning - K-means

K-means

K-means est une méthode d'apprentissage non supervisée pour regrouper les points de données. L'algorithme divise itérativement les points de données en clusters K en minimisant la variance dans chaque cluster.

Ici, nous allons vous montrer comment estimer la meilleure valeur pour K en utilisant la méthode du coude, puis utiliser K-means clustering pour regrouper les points de données en clusters.

Comment ça se passe ?

Tout d'abord, chaque point de données est attribué aléatoirement à l'un des clusters K. Ensuite, nous calculons le centroïde (fonctionnellement le centre) de chaque cluster, et réattribuons chaque point de données au cluster avec le centroïde le plus proche. Nous répétons ce processus jusqu'à ce que les affectations de cluster pour chaque point de données ne changent plus.

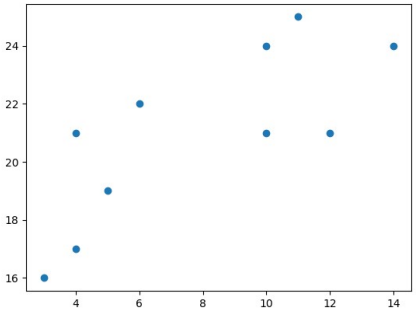
K-means clustering nous oblige à sélectionner K, le nombre de clusters dans lesquels nous voulons regrouper les données. La méthode du coude nous permet de tracer l'inertie (une métrique basée sur la distance) et de visualiser le point auquel elle commence à diminuer linéairement. Ce point est appelé "eblow" et est une bonne estimation de la meilleure valeur pour K basée sur nos données.

Exemple

Commencez par visualiser certains points de données :

```
import matplotlib.pyplot as plt

x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

plt.scatter(x, y)
plt.show()
```

et le résultat est :

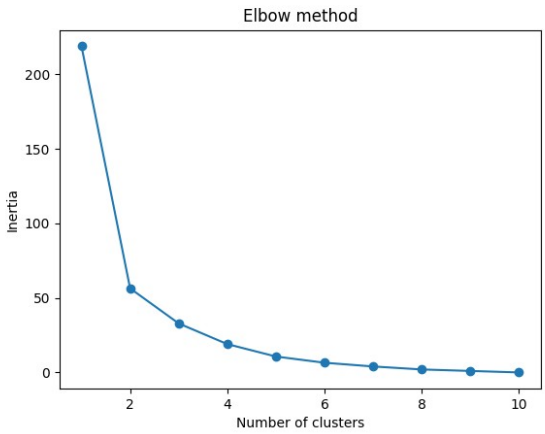


Nous utilisons maintenant la méthode du coude pour visualiser l'intertie pour différentes valeurs de K :

Exemple :

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
data = list(zip(x, y))
inertias = []
for i in range(1,11):
  kmeans = KMeans(n_clusters=i)
  kmeans.fit(data)
  inertias.append(kmeans.inertia_)
plt.plot(range(1,11), inertias, marker='o')
plt.title('Elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```

et le résultat est :

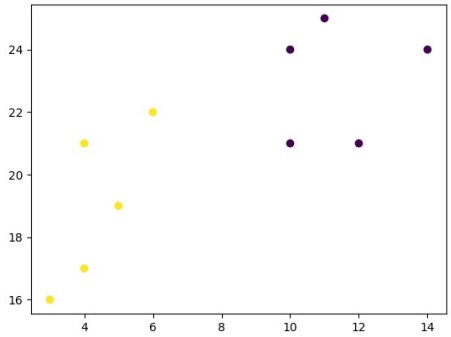


La méthode du coude montre que 2 est une bonne valeur pour K, donc nous retrainons et visualisons le résultat :

Exemple

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(data)

plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```

et le résultat est :



Exemple expliqué

Importez les modules dont vous avez besoin.

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
```

scikit-learn est une bibliothèque populaire pour l'apprentissage automatique.

Créez des tableaux qui ressemblent à deux variables dans un jeu de données. Notez que même si nous n'utilisons que deux variables ici, cette méthode fonctionnera avec n'importe quel nombre de variables :

```
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
```

Transformer les données en un ensemble de points :

```
data = list(zip(x, y))
print(data)
```

et le résultat est :

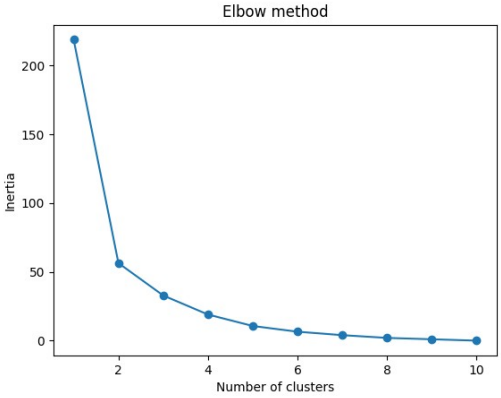
```
[(4, 21), (5, 19), (10, 24), (4, 17), (3, 16), (11, 25), (14, 24), (6, 22), (10, 21), (12, 21)]
```

Afin de trouver la meilleure valeur pour K, nous devons exécuter K-means à travers nos données pour une gamme de valeurs possibles. Nous n'avons que 10 points de données, donc le nombre maximum de clusters est de 10. Donc, pour chaque valeur K dans la plage(1,11), nous formons un modèle K-moyennes et traçons l'intertie à ce nombre de clusters :

```
inertias = []

for i in range(1,11):
  kmeans = KMeans(n_clusters=i)
  kmeans.fit(data)
  inertias.append(kmeans.inertia_)

plt.plot(range(1,11), inertias, marker='o')
plt.title('Elbow method')
plt.xlabel('Number of clusters')
```

```
plt.ylabel('Inertia')
plt.show()
```

et voici le résultat à obtenir :



Nous pouvons voir que le "coude" sur le graphique ci-dessus (où l'intérieur devient plus linéaire) est à K = 2. Nous pouvons alors ajuster notre algorithme K-means une fois de plus et tracer les différents clusters affectés aux données :

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(data)

plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```

avec pour résultat :