Apprentissage automatique supervisé

Nous allons aujourd'hui traité le cas d'apprentissage supervisé, pour cela nous nous appuyons sur l'exemple de données présent dans sklearn. La base d'image des IRIS.

Cette base contient des Iris qu'un botaniste, Ronald Fisher, a classés en 1936 à l'aide d'une clef d'identification des plantes (type de pétales, sépale, type des feuilles, forme des feuilles, ...). Puis, pour chaque fleur classée il a mesuré les longueurs et largeurs des sépales et pétales.

L'idée qui nous vient alors, consiste à demander à l'ordinateur de déterminer automatiquement l'espèce d'une nouvelle plante en fonction de la mesure des dimensions de ses sépales et pétales que nous aurions réalisée sur le terrain. Pour cela nous lui demanderons de construire sa décision à partir de la connaissance extraite des mesures réalisées par M. Fisher.

Autrement dit, nous allons donner à l'ordinateur un jeu de données déjà classées et lui demander de classer de nouvelles données à partir de celui-ci.

Une fois alimentés avec les observations connues, nos prédicteurs vont chercher à identifier des groupes parmi les plantes déjà connues et détermineront quel est le groupe duquel se rapproche le plus notre observation.

Chargement de notre base

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

Dans ce cas, la variable définie "iris" est d'un type inhabituel, découvrons-le :

```
print(iris)
```

<class 'sklearn.datasets.base.Bunch'> : ce qui correspond à une sorte dictionnaire. Et que peut-on faire avec ?

```
print(dir(iris))
iris.feature_names
```

La première instruction nous affiche une description de son contenu. La deuxième nous donne les noms des paramètres de nos donénes/enregistrements.

L'attribut *feature_names* contient le nom des différents paramètres de nos données, il s'agit des longueurs et largeurs de pétales et sépales.

```
iris.feature_names
['sepal length (cm)',
'sepal width (cm)',
'petal length (cm)',
'petal width (cm)']
```

Comme nous avons maintenant l'habitude de faire, l'affichage des 5 premiers enregistrements peut se faire comme suit :

```
iris.data[:5]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2]])
```

L'attribut *target_names* contient la liste des espèces connues, nos labels de classification :

```
iris.target_names
array(['setosa', 'versicolor', 'virginica'],
      dtype='<U10')
```

Afin d'avoir un accès plus facile à cet attribut, nous allos créer une variable pour cela :

```
target = iris.target
for i in [0,1,2]:
    print("classe : %s, nb exemplaires: %s" % (i,
    len(target[ target == i]) ) )
```

L'affichage de cette variable donnera une tableau contenant soit 0, 1 ou 2, représentant les 3 classes des iris. Qu'affiche la boucle *for* ?

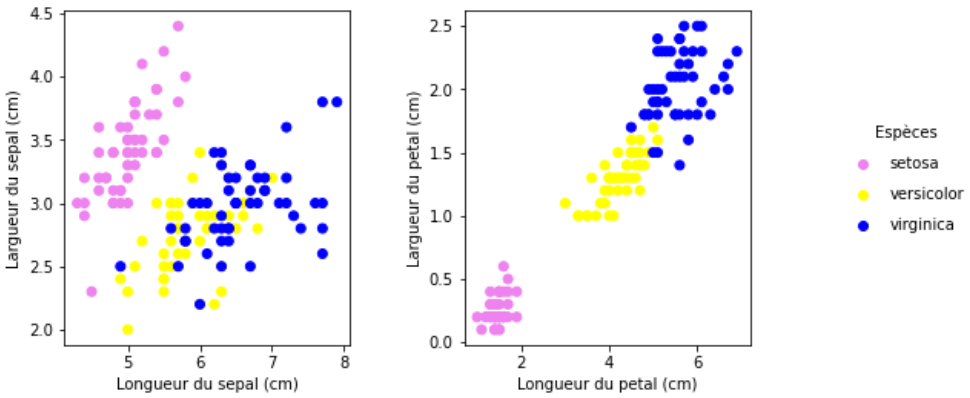
En résumé, l'échantillon de fleurs propose plusieurs informations :

- Les noms des données disponibles : *feature_names*
- Les mesures réalisées sur l'échantillon de fleurs connues et déjà classées : *data* Il s'agit de nos informations, des paramètres de nos vecteurs pour chaque fleur
- Le nom de chaque espèce : *target_names*

- Le classement de chaque enregistrement *data* dans son espèce: *target* Il s'agit de la classe de chaque fleur/vecteur

Observation des données

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
data = iris.data
fig = plt.figure(figsize=(8, 4))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
ax1 = plt.subplot(1,2,1)

clist = ['violet', 'yellow', 'blue']
colors = [clist[c] for c in iris.target]

ax1.scatter(data[:, 0], data[:, 1], c=colors)
plt.xlabel('Longueur du sepal (cm)')
plt.ylabel('Largueur du sepal (cm)')

ax2 = plt.subplot(1,2,2)

ax2.scatter(data[:, 2], data[:, 3], color=colors)

plt.xlabel('Longueur du petal (cm)')
plt.ylabel('Largueur du petal (cm)')

# Légende
for ind, s in enumerate(iris.target_names):
    # on dessine de faux points, car la légende n'affiche que les points ayant un label
    plt.scatter([], [], label=s, color=clist[ind])

plt.legend(scatterpoints=1, frameon=False, labelspacing=1
           , bbox_to_anchor=(1.8, .5) , loc="center right", title='Espèces')
plt.plot();
```



Avant de commencer à classer ses données il est toujours bon de visualiser à quoi elles

ressemblent et si d'éventuelles relations se dessinent. Pour les visualiser, l'algorithme suivant va parcourire notre *data*, créer 2 figures, une pour l'affichage selon la longueur et largeur de pétal et la deuxième celles de sépal.

Observer avec Seaborn

C'est assez vite écrit et déjà fort parlant: la séparation des groupes entre les longueurs et largeurs de pétales semble très nette et déterminante !

Nous pourrions aussi le faire entre les longueurs de pétales et largeurs de sépales et inversement même si cela semble moins naturel.

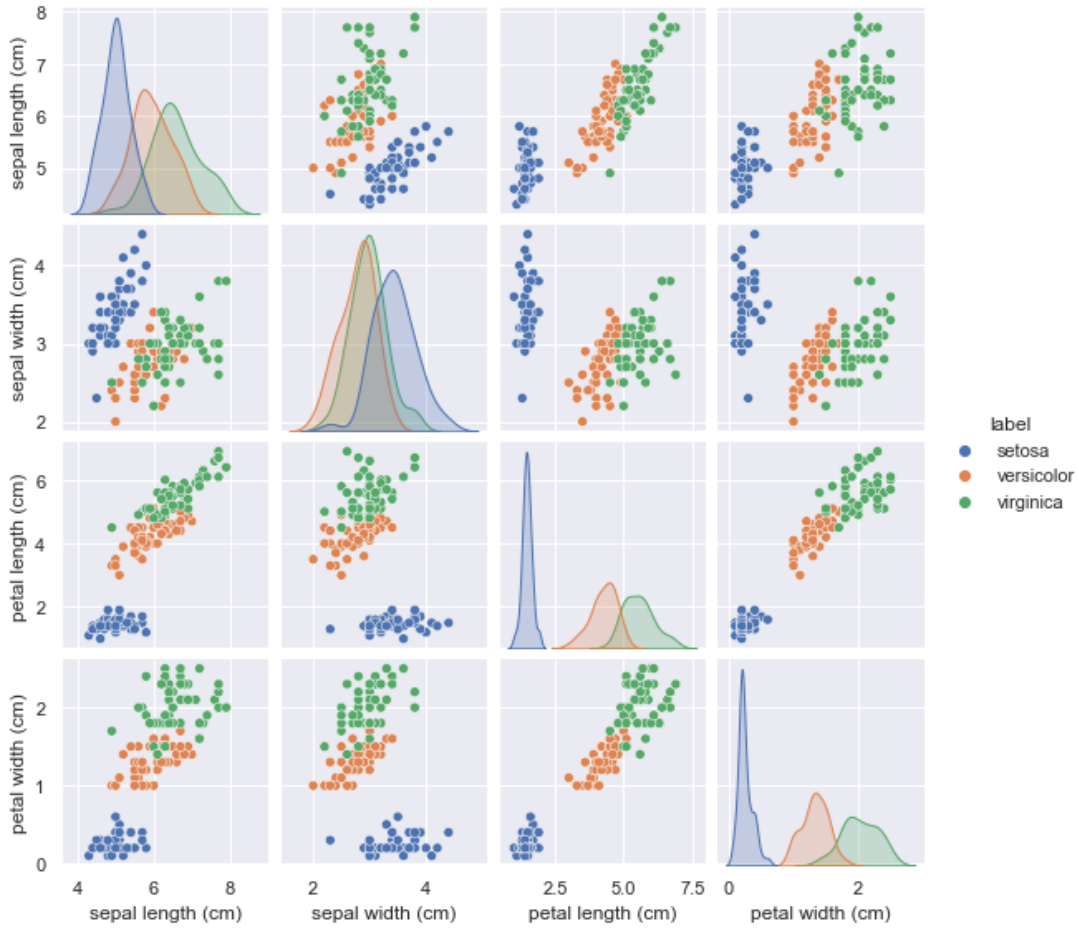
La librairie Seaborn propose une matrice prête à l'emploi via le graphique Scatterplot Matrix pour réaliser ce type de graphique:

```
import seaborn as sns
import pandas as pd
sns.set()
df = pd.DataFrame(data, columns=iris['feature_names'] )
df['target'] = target
df['label'] = df.apply(lambda x: iris['target_names']
    [int(x.target)], axis=1)
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	label
0	5.1	3.5	1.4	0.2	0	setosa
1	4.9	3.0	1.4	0.2	0	setosa
2	4.7	3.2	1.3	0.2	0	setosa
3	4.6	3.1	1.5	0.2	0	setosa
4	5.0	3.6	1.4	0.2	0	setosa

Il ne reste plus qu'à dessiner le graphique avec Seaborn...

```
sns.pairplot(df, hue='label', vars=iris['feature_names'], size=2);
```


sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
label
setosa
versicolor
virginica

Apprentissage : Classifieur Naive Bayes

Le Naîve Bayes Classifier (classifieur bayésien naïf ou classifieur naïf de Bayes) est un modèle d'apprentissage automatique de type supervisé permettant la classification en se basant sur le théorème de Bayes avec une forte indépendance des hypothèses (c'est-à-dire pas de corrélation entre les caractéristiques d'un ensemble de données). Cette classification appartient à la catégorie des classifieurs linéaires.

Tout d'abord, le théorème de Bayes décrit la probabilité d'une caractéristique en se basant au préalable sur des situations liées à celle-ci. Par exemple, si la probabilité qu'une personne soit diabétique est liée à son âge, alors en utilisant ce théorème, on peut utiliser l'âge pour prédire avec plus de précision la probabilité de souffrir de cette maladie chronique.

Sklearn contient plusieurs types de classificateurs Naive Bayes :

- Gaussian Naîve Bayes ;
- Bernoulli Naîve Bayes ;
- ....

Nous proposons de commencer par la classification Naive Bayes qui suppose que chaque classe est construite à partir d'une distribution Gaussiènne.

```
from sklearn.naive_bayes import GaussianNB
#creer notre classifieur
clf = GaussianNB()
#Apprentissage
clf.fit(data, target)
```

Exécutons la prédiction sur les données d'apprentissage elles-mêmes :

```
result = clf.predict(data)
result
array([0, 0, 0, 0, 0,..., 2, 2, 2, 2])
```

Il est bien évident que l'exemple ci-dessus, on n'a pas diviser nos données en données train et données test. C'est juste pour aller vite. On verra la qualité de notre prédiction dans la suite et fera une division de nos données pour un apprentissage plus correct.

Qualité de la prédiction

Avec la commande *result - target* : on observe la qualité de la prédiction. Là ou la prédiction est juste, la différence de result - target doit être égale à 0. Si la prédiction est parfaite nous aurons des zéros dans tout le tableau. Ce qui loin d'être notre cas, affichez et observez le

resultat de la commande précedante.

Que constatez-vous ? Affichez le nombre d'erreur ainsi que le pourcentage de la prédiction.

On aurait pu penser que les tableaux seraient parfaitement identiques, mais l'algorithme utilisé estime vraiment le label final en fonction des règles de probabilité qu'il a établies. Ces règles ne sont pas rigoureusement identiques à la réalité.

Cela prouve aussi que l'algorithme essaye de trouver un classement intelligent et ne se contente pas de comparer les valeurs d'origines aux valeurs entrantes.

Notre solution pour mesurer la qualité de la prédiction est très rudimentaire, Scikit-Learn propose des solutions plus abouties :

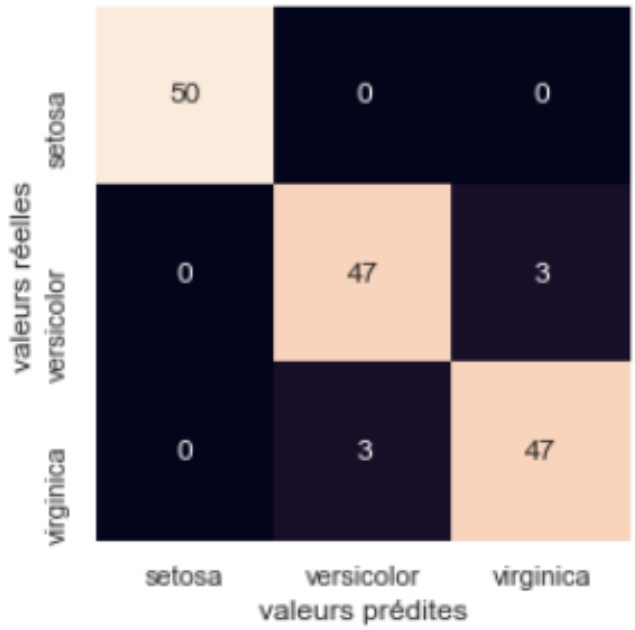
```
from sklearn.metrics import accuracy_score
accuracy_score(result, target) # 96% de reussite
```

Scikit-Learn permet aussi de calculer la matrice de confusion:

```
from sklearn.metrics import confusion_matrix
conf = confusion_matrix(target, result)
conf
array([[50,  0,  0],
       [ 0, 47,  3],
       [ 0,  3, 47]])
```

Et Seaborn permet de la représenter avec le Heatmap :

```
sns.heatmap(conf, square=True, annot=True, cbar=False
            , xticklabels=list(iris.target_names)
            , yticklabels=list(iris.target_names))
plt.xlabel('valeurs prédites')
plt.ylabel('valeurs réelles');
```



On observe ici :

- L'espèce Setosa a été parfaitement identifiée
- 3 Virginica ont été confondues avec des Versicolor et inversemment

Si vous n'avez pas seaborn, la fonction matshow de matplotlib peut vous aider : plt.matshow(conf, cmap='rainbow')

Séparation du jeu de tests et d'apprentissage

Nous ne disposons que d'un seul jeu de données connues. Généralement l'on teste l'algorithme sur de nouvelles données, sinon les résultats sont forcément toujours très bons.

Le module model_selection de Scikit-Learn propose des fonctions pour séparer le jeu de données du jeu de tests qui sont attentives à ce type de petits problèmes:

```
from sklearn.model_selection import train_test_split
# split the data with 50% in each set
data_test = train_test_split(data, target
                                , random_state=0
                                , train_size=0.5)
data_train, data_test, target_train, target_test = data_test
```

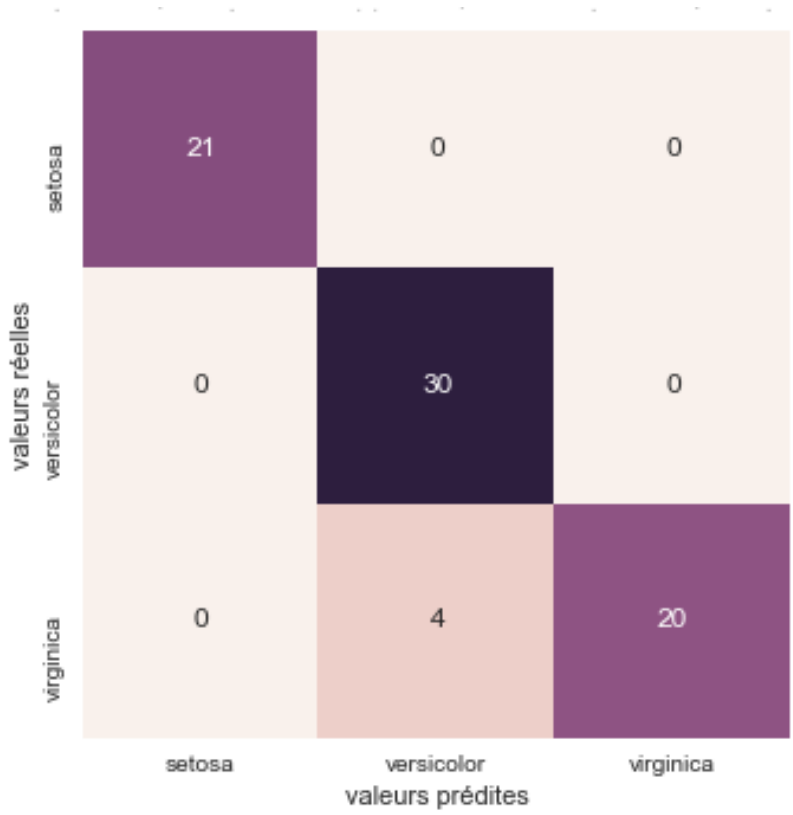
La fonction train_test_split permet de décomposer le jeu de données en 2 groupes: les données pour l'apprentissage et les données pour les tests. Il ne reste plus qu'à relancer la classification:

```
clf = GaussianNB()
clf.fit(data_train, target_train)
result = clf.predict(data_test)
#Puis de calculer de nouveau la qualite de la prediction:
accuracy_score(result, target_test)
.9466
#Cela reste toujours bon !
```

Affichez la matrice de confusion comme suit :

```
array([[21,  0,  0],
       [ 0, 30,  0],
       [ 0,  4, 20]])
```

Affichez aussi le Heatmap avec Seaborn (comme suit) :



Sur les 75 plantes prédites, 4 Virginica ont été confondues avec des Versicolor.

Apprentissage : Classifieur KNN (plus proches voisin)

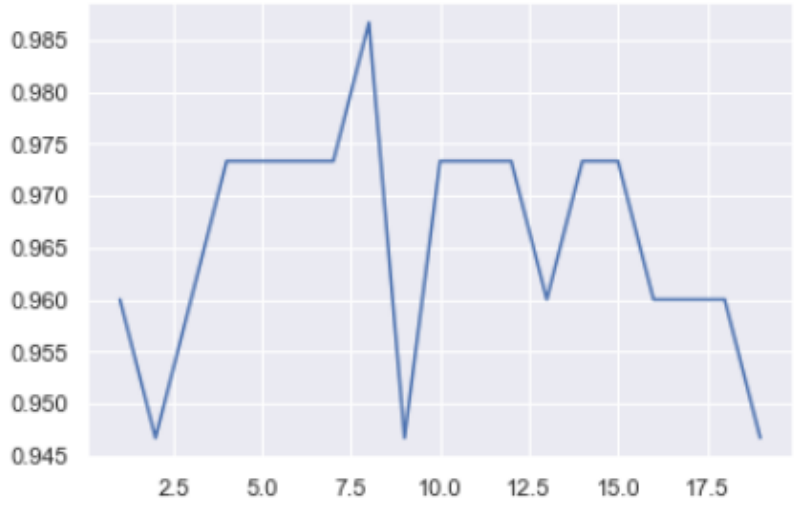
Essayons le même traitement en remplaçant GaussianNB par KN.
La classe à utiliser est

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

Calcul de la précision de la prédiction en fonction de N

```python
from sklearn import neighbors
clf = neighbors.KNeighborsClassifier()
data_test, target_test = iris.data[::2], iris.target[::2]
data_train, target_train = iris.data[1::2], iris.target[1::2]
result = []
n_values = range(1,20)
for n in n_values:
    clf = neighbors.KNeighborsClassifier(n_neighbors=n)
    clf.fit(data_train, target_train)
    Z = clf.predict(data_test)
    score = accuracy_score(Z, target_test)
    result.append(score)

plt.plot(list(n_values), result)
```

```
[<matplotlib.lines.Line2D at 0x19d2e12b280>]
```



Que montre ce beau graphique ?

Decision Trees

Les arbres de décision (Decisions trees) sont des outils d'aide à la prise de décision et d'exploitation de données permettant de présenter un ensemble de données sous la forme graphique d'un arbre.

Cette méthode d'apprentissage automatique est de type supervisé permettant de résoudre à la fois des problèmes de classification et de régression.

DecisionTreeClassifier prend plusieurs paramètres optionnels : max_depth (profondeur max de l'arbre), min_samples_leaf (nmobre min d'échantillons requis pour se retrouver àun noeud feuille, par défaut = 1), etc. —–> model = DecisionTreeClassifier(max_depth=3, min_samples_leaf=2)

Je donne ici un simple exemple (sans partition de données), l'exemple porte uniquement sur la classification avec DecisionTreeClassifier. à vous de jouer pour tester comme régresseur (DecisionTreeRegressor).

```
from sklearn.tree import DecisionTreeClassifier

# faut pas oublier de fractionner votre dataset
#instanciation
model=DecisionTreeClassifier()


 #apprentissage
model.fit(data_train,target_train)

#calcul de precision
print(model.score(data_test,target_test))

#prediction
longueur=2
largeur= 0,7
model.predict([[longueur,largeur]])
#affichage de la prediction selon la classe, a vous de jouer
```

SVM

Les machines à vecteurs de support (ou Support Vector Machine, SVM) sont une famille d'algorithmes d'apprentissage automatique de type supervisé et qui peuvent être utilisées pour des problèmes de discrimination (à quelle classe appartient un échantillon), de régression et de détection d'anomalies.

Je donne ici un simple exemple (sans partition de données), l'exemple porte uniquement sur la classification avec SVC. à vous de jouer pour tester les autres classifieurs (NuSVC, LinearSVC), régresseurs (SVR, NuSVR, LinearSVR).

```
from sklearn.svm import SVC

# faut pas oublier de fractionner votre dataset
#instanciation
modelSVC=SVC(kernel='linear',gamma='scale',shrinking=False)

#apprentissage
modelSVC.fit(data_train,target_train)

#calcul de precision
print(modelSVC.score(data_test,target_test))

#prediction
longueur=2
largeur= 0,7
modelSVC.predict([[longueur,largeur]])
#affichage de la prediction selon la classe, a vous de jouer
```

LA RÉGRESSION LINÉAIRE

La régression linéaire est l'un des principaux modèles à aborder en Machine Learning. À vrai dire, il n'existe pas qu'un seul type de régression. On distingue la régression de type linéaire mais aussi la régression logistique. Lors de ce tutoriel, nous n'aborderons que la régression linéaire en utilisant la bibliothèque d'apprentissage automatique Scikit-learn.

Une régression a pour objectif d'expliquer une variable Y par le moyen d'une autre variable X. Par exemple, le salaire d'une personne peut être expliqué à travers son niveau universitaire ; c'est à dire le nombre d'années passées à l'université.

La régression linéaire est alors modélisée par l'équation linéaire suivante qui met en relation X et Y :

$$Y = \beta_0 + \beta_1 X + \epsilon$$

Ou:

- β_0 et β_1 sont les paramètres du modèle ;
- X est la variable explicative ;
- Y est la variable expliquée ;
- ϵ est l'erreur de l'estimation.

Puisqu'on a une seule variable explicative, alors on est dans le cas de la régression linéaire simple. Par conséquent, la régression linéaire multiple est lorsque vous avez au moins deux variables explicatives.

```
from sklearn.linear_model import LinearRegression

#donnees
x = np.array([6, 8, 10, 14, 18]).reshape((-1, 1))
y = np.array([7, 9, 13, 17, 18])

#instancier modele
model_linReg = LinearRegression()

#entrainement du modele
model_linReg.fit(x, y)

#precision du modele
precision = model_linReg.score(x, y)
print(precision*100)

#prediction
prediction = model_linReg.predict(x)
print(prediction)
```

Pour calculer les paramètres β_0 et β_1 nous avons les deux attributs de notre modèle : intercept_ et coef_, respectivement.

```
model_linReg.intercept_
model_linReg.coef_
```