# LAB 2 - Classification et régression à l'aide de l'apprentissage supervisé

> Auteur : Badr TAJINI - Introduction à l'IA - ESIEE-IT - 2024/2025

Dans ce lab, nous allons apprendre la classification et la régression des données en utilisant des techniques l'apprentissage supervisé. A la fin de ce laboratoire, vous aurez une meilleure compréhension de ces sujets :

- Différences entre l'apprentissage supervisé et non supervisé
- Méthodes de classification
- Méthodes de prétraitement des données
- Codage des étiquettes
- Classificateurs de régression logistique
- Le classificateur Naïve Bayes
- Matrices de confusion
- Machines à vecteurs de support et classificateurs SVM
- Régression linéaire et polynomiale
- Régresseurs linéaires à une ou plusieurs variables
- Estimation des prix du logement à l'aide de régresseurs vectoriels de soutien

## Apprentissage supervisé ou non supervisé

Il n'est pas difficile de constater, en lisant la presse populaire, que l'apprentissage automatique est l'un des domaines les plus dynamiques de l'intelligence artificielle à l'heure actuelle. l'apprentissage automatique est généralement classé en apprentissage supervisé et non supervisé. Il existe l'autres classifications, mais nous les aborderons plus tard.

Avant de donner une définition plus formelle de l'apprentissage supervisé et de l'apprentissage non supervisé, il convient de comprendre intuitivement ce qu'est l'apprentissage supervisé par rapport à l'apprentissage non supervisé. Supposons que vous disposiez l'un ensemble de portraits de personnes. Les personnes de cet ensemble constituent un groupe très diversifié l'hommes et de femmes, avec toutes sortes de nationalités, l'âges, de poids, etc. Au départ, vous soumettez l'ensemble de données à un algorithme l'apprentissage non supervisé. Dans ce cas, sans aucune connaissance a priori, l'algorithme non supervisé commencera à classer ces photographies en fonction l'une caractéristique qu'il reconnaît comme similaire. Par exemple, il pourrait commencer à reconnaître que les hommes et les femmes sont différents et à regrouper les hommes dans un groupe et les femmes dans un autre. Mais rien ne garantit qu'il trouvera ce

modèle. Il pourrait regrouper les images parce que certains portraits ont un arrière-plan sombre et l'autres un arrière-plan clair, ce qui serait probablement une déduction inutile.

Imaginons maintenant la même série de photographies, mais cette fois-ci, nous avons également une étiquette qui accompagne chaque photographie. Disons que l'étiquette est le sexe. Comme nous disposons maintenant l'une étiquette pour les données, nous pouvons soumettre ces données à un algorithme supervisé et utiliser les variables l'entrée (dans ce cas, les variables l'entrée sont les pixels de la photographie) pour calculer la variable cible (dans ce cas, le sexe). En l'autres termes, il s'agit l'un algorithme supervisé :

l'**apprentissage supervisé** fait référence au processus de construction l'un modèle l'apprentissage automatique basé sur des données de formation étiquetées. Dans l'apprentissage supervisé, chaque exemple ou rangée est un tuple composé de variables l'entrée et l'une variable cible souhaitée. Par exemple, un ensemble de données couramment utilisé dans l'apprentissage automatique est l'ensemble de données "TitaniC". Cet ensemble de données contient des caractéristiques permettant de décrire les passagers du célèbre navire RMS Titanic. Voici quelques-unes des caractéristiques l'entrée :

- Nom du passager
- Le sexe
- Classe cabine
- l'âge
- Lieu l'embarquement

Dans ce cas, la variable cible serait le fait que le passager ait survécu ou non.

l'**apprentissage non supervisé** fait référence au processus de construction l'un modèle l'apprentissage automatique sans s'appuyer sur des données de formation étiquetées. Dans un certain sens, il s'agit de l'opposé de l'apprentissage supervisé. Comme il n'y a pas l'étiquettes disponibles, vous devez extraire des informations en vous basant uniquement sur les données qui vous sont fournies. Avec l'apprentissage non supervisé, nous formons un système dans lequel des points de données distincts seront potentiellement séparés en plusieurs clusters ou groupes. Il est important de souligner que nous ne connaissons pas exactement les critères de séparation. Par conséquent, un algorithme l'apprentissage non supervisé doit séparer l'ensemble de données donné en plusieurs groupes de la meilleure façon possible.

Maintenant que nous avons décrit l'une des principales façons de classer les approches l'apprentissage automatique, voyons comment nous classons les données.

## Qu'est-ce que la classification ?

Dans cette section, nous aborderons les techniques de classification supervisée. Le processus de classification est une technique utilisée pour classer les données dans un nombre fixe de catégories afin qu'elles puissent être utilisées de manière efficace et efficiente.

Dans l'apprentissage automatique, la classification est utilisée pour identifier la catégorie à laquelle appartient un nouveau point de données. Un modèle de classification est construit sur la base de l'ensemble de données l'apprentissage contenant les points de données et les étiquettes correspondantes. Par exemple, disons que nous voulons déterminer si une image donnée contient ou non le visage l'une personne. Nous construirons un ensemble de données de formation contenant des classes correspondant à deux catégories : visage et non-visage. Un modèle serait ensuite formé sur la base des échantillons de formation disponibles. Le modèle formé peut ensuite être utilisé pour l'inférence.

Un bon système de classification facilite la recherche et l'extraction des données. La classification est largement utilisée dans la reconnaissance des visages, l'identification des spams, les moteurs de recommandation, etc. Un bon algorithme de classification des données génère automatiquement les bons critères pour séparer les données données en un nombre donné de classes.

Pour que la classification produise des résultats satisfaisants, un nombre suffisamment important l'échantillons est nécessaire afin de pouvoir généraliser ces critères. Si le nombre l'échantillons est insuffisant, l'algorithme s'adaptera trop aux données l'apprentissage. Cela signifie qu'il n'obtiendra pas de bons résultats sur des données inconnues parce qu'il a trop affiné le modèle pour s'adapter aux modèles observés dans les données l'apprentissage. Il s'agit en fait l'un problème courant dans le monde de l'apprentissage automatique. C'est une bonne idée de prendre en compte ce facteur lors de la construction de divers modèles l'apprentissage automatique.

## Prétraitement des données

Les données brutes sont le carburant des algorithmes l'apprentissage automatique. Mais tout comme nous ne pouvons pas mettre du pétrole brut dans une voiture et que nous devons utiliser de l'essence, les algorithmes l'apprentissage automatique s'attendent à ce que les données soient formatées l'une certaine manière avant que le processus de formation puisse commencer. Afin de préparer les données à être ingérées par les algorithmes l'apprentissage automatique, les données doivent être prétraitées et converties dans le bon format. Examinons quelques-uns des moyens l'y parvenir.

Pour que les exemples que nous allons analyser fonctionnent, nous devons importer quelques paquets Python :

```
import numpy as np
from sklearn import preprocessing
```

Définissons également quelques exemples de données :

```
input_data = np.array([[5.1, -2.9, 3.3],
                       [-1.2, 7.8, -6.1],
                       [3.9, 0.4, 2.1],
                       [7.3, -9.9, -4.5]])
```

Ce sont les techniques de prétraitement que nous allons analyser :

- Binarisation
- Enlèvement moyen
- Mise à l'échelle
- Normalisation

## Binarisation

La binarisation est utilisée pour convertir des valeurs numériques en valeurs booléennes. Utilisons une méthode intégrée pour binariser les données l'entrée en utilisant `2,1` comme valeur seuil.

Ajoutez les lignes suivantes au même fichier Python :

```
# Binarize data
data_binarized = preprocessing.Binarizer(threshold=2.1).transform(input_data)
print("\nBinarized data:\n", data_binarized)
```

Si vous exécutez le code, vous obtiendrez le résultat suivant :

```
Binarized data:
[[  1.  0.  1.]
[  0.  1.  0.]
[  1.  0.  0.]
[  1.  0.  0.]]
```

Comme on peut le voir ici, toutes les valeurs supérieures à `2,1` deviennent `1`. Les autres valeurs deviennent `0`.

## Enlèvement moyen

La suppression de la moyenne est une technique de prétraitement couramment utilisée dans l'apprentissage automatique. Il est généralement utile de supprimer la moyenne l'un vecteur de caractéristiques, de sorte que chaque caractéristique soit centrée sur zéro. Nous procédons ainsi pour éliminer le biais des caractéristiques de notre vecteur de caractéristiques.

Ajoutez les lignes suivantes au même fichier Python que dans la section précédente :

```
# Print mean and standard deviation
print("\nBEFORE:")
print("Mean =", input_data.mean(axis=0))
print("Std deviation =", input_data.std(axis=0))
```

La ligne précédente affiche la moyenne et l'écart-type des données l'entrée. Supprimons la moyenne :

```
# Remove mean
data_scaled = preprocessing.scale(input_data)
print("\nAFTER:")
print("Mean =", data_scaled.mean(axis=0))
print("Std deviation =", data_scaled.std(axis=0))
```

Si vous exécutez le code, vous obtiendrez le résultat suivant :

```
BEFORE:
Mean = [ 3.775 -1.15  -1.3  ]
Std deviation = [ 3.12039661  6.36651396  4.0620192 ] AFTER:
Mean = [  1.11022302e-16  0.00000000e+00  2.77555756e-17]
Std deviation = [ 1.  1.  1.]
```

Comme le montrent les valeurs obtenues, la valeur moyenne est très proche de `0` et l'écart-type est de `1`.

## Mise à l'échelle

Comme nous l'avons fait dans les sections précédentes, nous allons nous faire une idée de ce qu'est la mise à l'échelle à l'aide l'un exemple. Supposons que vous disposiez l'un ensemble de données contenant des caractéristiques liées aux habitations et que vous essayiez de prédire les prix de ces habitations. Les plages des valeurs numériques de ces caractéristiques peuvent varier considérablement. Par exemple, la superficie l'une maison est normalement de plusieurs milliers

de mètres carrés, alors que le nombre de pièces est généralement inférieur à 10. En outre, certaines de ces caractéristiques peuvent contenir des valeurs aberrantes. Par exemple, notre ensemble de données peut contenir quelques maisons de maître qui faussent le reste de l'ensemble de données.

Nous devons trouver un moyen l'échelonner ces caractéristiques de sorte que la pondération accordée à chacune l'entre elles soit à peu près la même et que les valeurs aberrantes n'aient pas une importance démesurée. Une façon l'y parvenir est de réajuster toutes les caractéristiques de manière à ce qu'elles se situent dans un petit intervalle, tel que `0` et `1`. l'algorithme MinMaxScaler est probablement la façon la plus efficace l'y parvenir. La formule de cet algorithme est la suivante

$$\frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Où *max(x)* est la plus grande valeur l'une variable, *min(x)* est la plus petite valeur et ~~xi~~ est chaque valeur individuelle.

Dans notre vecteur de caractéristiques, la valeur de chaque caractéristique peut varier entre de nombreuses valeurs aléatoires. Il est donc important l'échelonner ces caractéristiques afin l'obtenir des conditions égales pour l'entraînement de l'algorithme l'apprentissage automatique. Aucune caractéristique ne doit être artificiellement grande ou petite en raison de la nature des mesures.

Pour mettre cela en œuvre en Python, ajoutez les lignes suivantes dans le fichier :

```
# Min max scaling
data_scaler_minmax = preprocessing.MinMaxScaler(feature_range=(0, 1))
data_scaled_minmax = data_scaler_minmax.fit_transform(input_data)
print("\nMin max scaled data:\n", data_scaled_minmax)
```

Si vous exécutez le code, vous obtiendrez le résultat suivant :

```
Min max scaled data:
 [[ 0.74117647  0.39548023   1.         ]
 [ 0.           1.          0.         ]
 [ 0.6          0.5819209   0.87234043]
 [ 1.           0.          0.17021277]]
```

Chaque ligne est échelonnée de manière à ce que la valeur maximale soit `1` et que toutes les autres valeurs soient relatives à cette valeur.

## Normalisation

Les gens confondent souvent la mise à l'échelle et la normalisation. l'une des raisons pour lesquelles ces termes sont souvent confondus est qu'ils sont en fait très similaires. Dans les deux cas, vous transformez les données pour les rendre plus utiles. Mais alors que la mise à l'échelle modifie la *plage de* valeurs l'une variable, la normalisation modifie la *forme de la distribution des* données. Pour que les modèles l'apprentissage automatique fonctionnent mieux, il est souhaitable que les valeurs l'une caractéristique soient normalement distribuées.

Mais la réalité est désordonnée et il arrive que ce ne soit pas le cas. Par exemple, la distribution des valeurs peut être asymétrique. La normalisation permet de répartir normalement les données. Voici un graphique des données avant et après la normalisation :

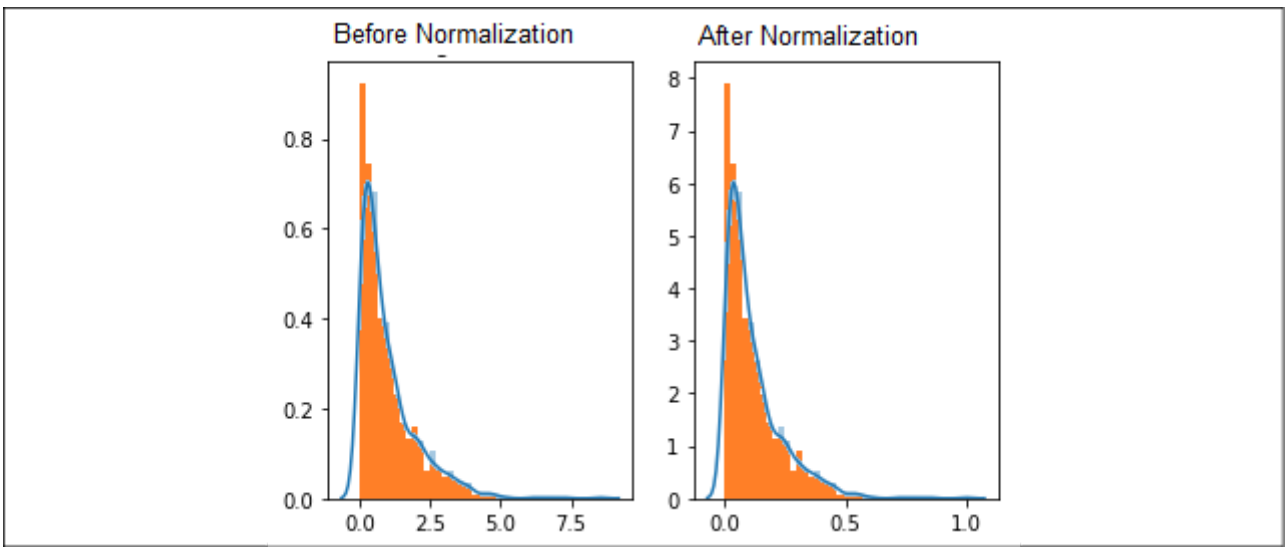


Figure 1 : Avant et après la normalisation

Nous utilisons le processus de normalisation pour modifier les valeurs du vecteur de caractéristiques afin de pouvoir les mesurer sur une échelle commune. Dans l'apprentissage automatique, nous utilisons de nombreuses formes de normalisation. Certaines des formes de normalisation les plus courantes visent à modifier les valeurs de manière à ce que leur somme soit égale à 1. La normalisation **L1**, qui fait référence aux **moindres écarts absolus**, permet de s'assurer que la somme des valeurs absolues est égale à *1* dans chaque ligne. La normalisation **L2**, qui fait référence aux moindres carrés, permet de s'assurer que la somme des carrés est égale à *1*.

En général, la technique de normalisation L1 est considérée comme plus robuste que la technique de normalisation L2. La technique de normalisation L1 est robuste parce qu'elle résiste aux valeurs aberrantes des données. Souvent, les données ont tendance à contenir des valeurs aberrantes et nous ne pouvons rien y faire. Nous voulons utiliser des techniques qui peuvent les ignorer efficacement et en toute sécurité pendant les calculs. Si nous résolvons un

problème dans lequel les valeurs aberrantes sont importantes, la normalisation L2 est peut-être un meilleur choix.

Ajoutez les lignes suivantes au même fichier Python :

```
# Normalize data
data_normalized_l1 = preprocessing.normalize(input_data, norm='l1')
data_normalized_l2 = preprocessing.normalize(input_data, norm='l2')
print("\nL1 normalized data:\n", data_normalized_l1)
print("\nL2 normalized data:\n", data_normalized_l2)
```

Si vous exécutez le code, vous obtiendrez le résultat suivant :

```
L1 normalized data:
 [[ 0.45132743  -0.25663717    0.2920354 ]
 [-0.0794702    0.51655629   -0.40397351]
 [ 0.609375     0.0625  0.328125    ]
 [ 0.33640553    -0.4562212    -0.20737327]]
L2 normalized    data:
 [[ 0.75765788    -0.43082507    0.49024922]
 [-0.12030718    0.78199664 -0.61156148]
 [ 0.87690281    0.08993875    0.47217844]
 [ 0.55734935    -0.75585734 -0.34357152]]
```

Le code de toute cette section se trouve dans le fichier `data_preprocessor.py.`

## Codage des étiquettes

Lors de la classification, nous avons généralement affaire à un grand nombre l'étiquettes. Ces étiquettes peuvent se présenter sous la forme de mots, de nombres ou l'autres éléments. De nombreux algorithmes l'apprentissage automatique ont besoin de chiffres en entrée. Par conséquent, s'il s'agit déjà de nombres, ils peuvent être directement utilisés pour l'apprentissage. Mais ce n'est pas toujours le cas.

Les étiquettes sont normalement des mots, car les mots peuvent être compris par les humains. Les données l'apprentissage sont étiquetées avec des mots afin que la correspondance puisse être suivie. Un codeur l'étiquettes peut être utilisé pour convertir les étiquettes de mots en nombres. l'encodage l'étiquettes fait référence au processus de transformation des étiquettes de mots en nombres. Cela permet aux algorithmes de traiter les données. Prenons un exemple :

Créez un nouveau fichier Python et importez les paquets suivants :

```
import numpy as np
from sklearn import preprocessing
```

Définir quelques exemples l'étiquettes :

```
# Sample input labels
input_labels = ['red', 'black', 'red', 'green', 'black', 'yellow', 'white']
```

Créer l'objet codeur l'étiquettes et l'entraîner :

```
# Create label encoder and fit the labels
encoder = preprocessing.LabelEncoder()
encoder.fit(input_labels)
```

Imprimez la correspondance entre les mots et les chiffres :

```
# Print the mapping
print("\nLabel mapping:")
for i, item in enumerate(encoder.classes_):
    print(item, '-->', i)
```

Encodons un ensemble l'étiquettes ordonnées de manière aléatoire pour voir comment il se comporte :

```
# Encode a set of labels using the encoder
test_labels = ['green', 'red', 'black']
encoded_values = encoder.transform(test_labels)
print("\nLabels =", test_labels)
print("Encoded values =", list(encoded_values))
```

Décodons un ensemble aléatoire de chiffres :

```
# Decode a set of values using the encoder
encoded_values = [3, 0, 4, 1]
decoded_list = encoder.inverse_transform(encoded_values)
print("\nEncoded values =", encoded_values)
print("Decoded labels =", list(decoded_list))
```

Si vous exécutez le code, vous obtiendrez le résultat suivant :

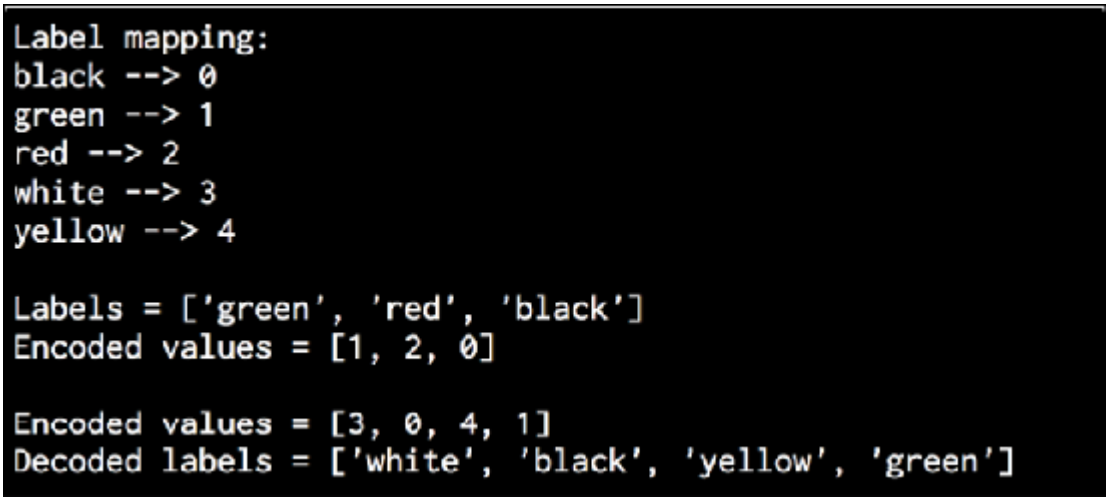


Figure 2 : Résultats du codage et du décodage

Vous pouvez vérifier la correspondance pour voir si les étapes l'encodage et de décodage sont correctes. Le code de cette section se trouve dans le fichier `label_encoder.py.`

## Classificateurs de régression logistique

La régression logistique est une technique utilisée pour expliquer la relation entre les variables l'entrée et les variables de sortie. La régression peut être utilisée pour faire des prédictions sur des valeurs continues, mais elle peut également être utile pour faire des prédictions discrètes où le résultat est *Vrai* ou *Faux*, par exemple, ou *Rouge*, *Vert* ou *Jaune* comme autre exemple.

Les variables l'entrée sont supposées être indépendantes et la variable de sortie est appelée variable dépendante. La variable dépendante ne peut prendre qu'un ensemble fixe de valeurs. Ces valeurs correspondent aux classes du problème de classification.

Notre objectif est l'identifier la relation entre les variables indépendantes et les variables dépendantes en estimant les probabilités à l'aide l'une fonction logistique. Dans ce cas, la fonction logistique sera une **courbe sigmoïde** utilisée pour construire la fonction avec différents paramètres. Voici quelques-unes des raisons pour lesquelles une fonction sigmoïde est utilisée dans les modèles de régression logistique :

- Il est compris entre 0 et 1
- Sa dérivée est plus facile à calculer
- Il s'agit l'une méthode simple pour introduire la non-linéarité dans le modèle

Elle est étroitement liée à l'analyse du modèle linéaire généralisé, dans laquelle nous essayons l'ajuster une ligne à un ensemble de points afin de minimiser l'erreur. Au lieu l'utiliser la régression linéaire, nous utilisons la régression logistique. La régression logistique n'est pas en soi une technique de classification, mais elle est utilisée de cette manière pour faciliter la

classification. Elle est couramment utilisée dans l'apprentissage automatique en raison de sa simplicité. Voyons comment construire un classificateur à l'aide de la régression logistique. Assurez-vous que le paquetage `Tkinter` est installé avant de continuer. Si ce n'est pas le cas, vous pouvez le trouver ici : https://docs.python.org/2/library/tkinter.html.

Créez un nouveau fichier Python et importez les paquets suivants :

```
import numpy as np
from sklearn import linear_model
import matplotlib.pyplot as plt
from utilities import visualize_classifier
```

Définir les données l'entrée de l'échantillon avec des vecteurs à deux dimensions et les étiquettes correspondantes :

```
# Define sample input data
X = np.array([[3.1, 7.2], [4, 6.7], [2.9, 8], [5.1, 4.5], [6, 5], [5.6, 5], [3.3,
0.4], [3.9, 0.9], [2.8, 1], [0.5, 3.4], [1, 4], [0.6, 4.9]])
y = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3])
```

Nous allons entraîner le classificateur à l'aide de ces données étiquetées. Créez maintenant l'objet classifieur de régression logistique :

```
# Create the logistic regression classifier
classifier = linear_model.LogisticRegression(solver='liblinear', C=1)
```

Entraînez le classificateur à l'aide des données définies précédemment :

```
# Train the classifier
classifier.fit(X, y)
```

Visualisez les performances du classificateur en observant les limites des classes :

```
# Visualize the performance of the classifier
visualize_classifier(classifier, X, y)
```

La fonction doit être définie avant de pouvoir être utilisée. Nous utiliserons cette fonction plusieurs fois dans ce laboratoire, il est donc préférable de la définir dans un fichier séparé et de l'importer. Cette fonction est donnée dans le fichier `utilities.py` fourni.

Créez un nouveau fichier Python et importez les paquets suivants :

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

Créez la définition de la fonction en prenant l'objet `classificateur`, les données l'entrée et les étiquettes comme paramètres l'entrée :

```
def visualize_classifier(classifier, X, y):
    # Define the minimum and maximum values for X and Y
    # that will be used in the mesh grid
    min_x, max_x = X[:, 0].min() - 1.0, X[:, 0].max() + 1.0
    min_y, max_y = X[:, 1].min() - 1.0, X[:, 1].max() + 1.0
```

Nous avons également défini les valeurs minimales et maximales des directions `X` et `Y` qui seront utilisées dans notre grille de maillage. Cette grille est essentiellement un ensemble de valeurs utilisées pour évaluer la fonction, de sorte que nous puissions visualiser les limites des classes. Définissez la taille du pas de la grille et créez-la en utilisant les valeurs minimales et maximales :

```
    # Define the step size to use in plotting the mesh grid     mesh_step_size = 0.01

    # Define the mesh grid of X and Y values
    x_vals, y_vals = np.meshgrid(np.arange(min_x, max_x, mesh_step_size),
np.arange(min_y, max_y, mesh_step_size))
```

Exécutez le classificateur sur tous les points de la grille :

```
    # Run the classifier on the mesh grid
    output = classifier.predict(np.c_[x_vals.ravel(), y_vals.ravel()])

    # Reshape the output array
    output = output.reshape(x_vals.shape)
```

Créez la figure, choisissez une palette de couleurs et superposez tous les points :

```
    # Create a plot
    plt.figure()
```

```
    # Choose a color scheme for the plot
    plt.pcolormesh(x_vals, y_vals, output, cmap=plt.cm.gray)

    # Overlay the training points on the plot
    plt.scatter(X[:, 0], X[:, 1], c=y, s=75, edgecolors='black', linewidth=1,
cmap=plt.cm.Paired)
```

Spécifiez les limites des tracés à l'aide des valeurs minimales et maximales, ajoutez les coches et affichez la figure :

```
    # Specify the boundaries of the plot
    plt.xlim(x_vals.min(), x_vals.max())
    plt.ylim(y_vals.min(), y_vals.max())
```

```
    # Specify the ticks on the X and Y axes
    plt.xticks((np.arange(int(X[:, 0].min() - 1), int(X[:, 0].max() + 1), 1.0)))
    plt.yticks((np.arange(int(X[:, 1].min() - 1), int(X[:, 1].max() + 1), 1.0)))
    plt.show()
```

Si le code est exécuté, vous verrez la capture l'écran suivante :

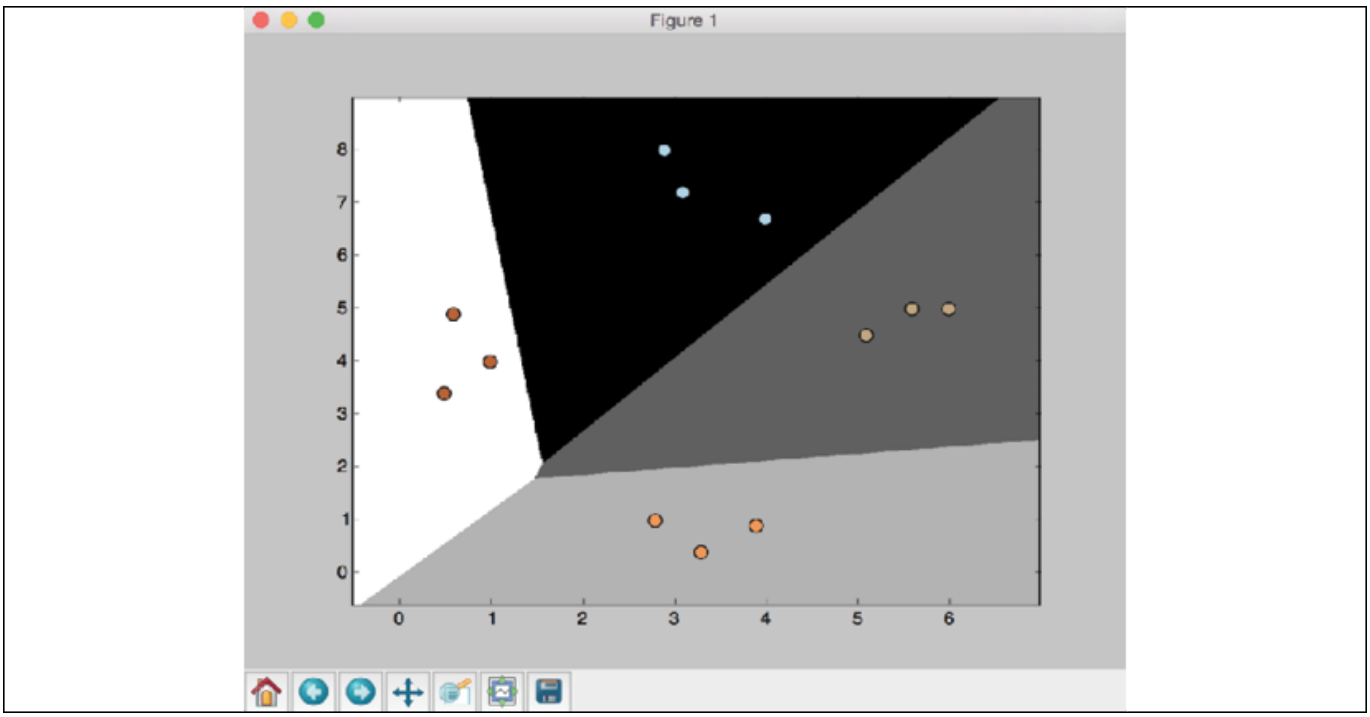


Figure 3 : La figure affichée une fois que les limites des parcelles ont été définies

Si vous changez la valeur de `C` en `100` dans la ligne suivante, vous verrez que les limites deviennent plus précises :

```
classifier = linear_model.LogisticRegression(solver='liblinear', C=100)
```

La raison en est que `C` impose une certaine pénalité en cas de mauvaise classification, de sorte que l'algorithme s'adapte davantage aux données l'apprentissage. Il convient l'être prudent avec ce paramètre, car si vous l'augmentez considérablement, l'algorithme s'adaptera trop aux données l'apprentissage et ne se généralisera pas bien.

Si vous exécutez le code avec `C` fixé à `100`, vous obtiendrez la capture l'écran suivante :

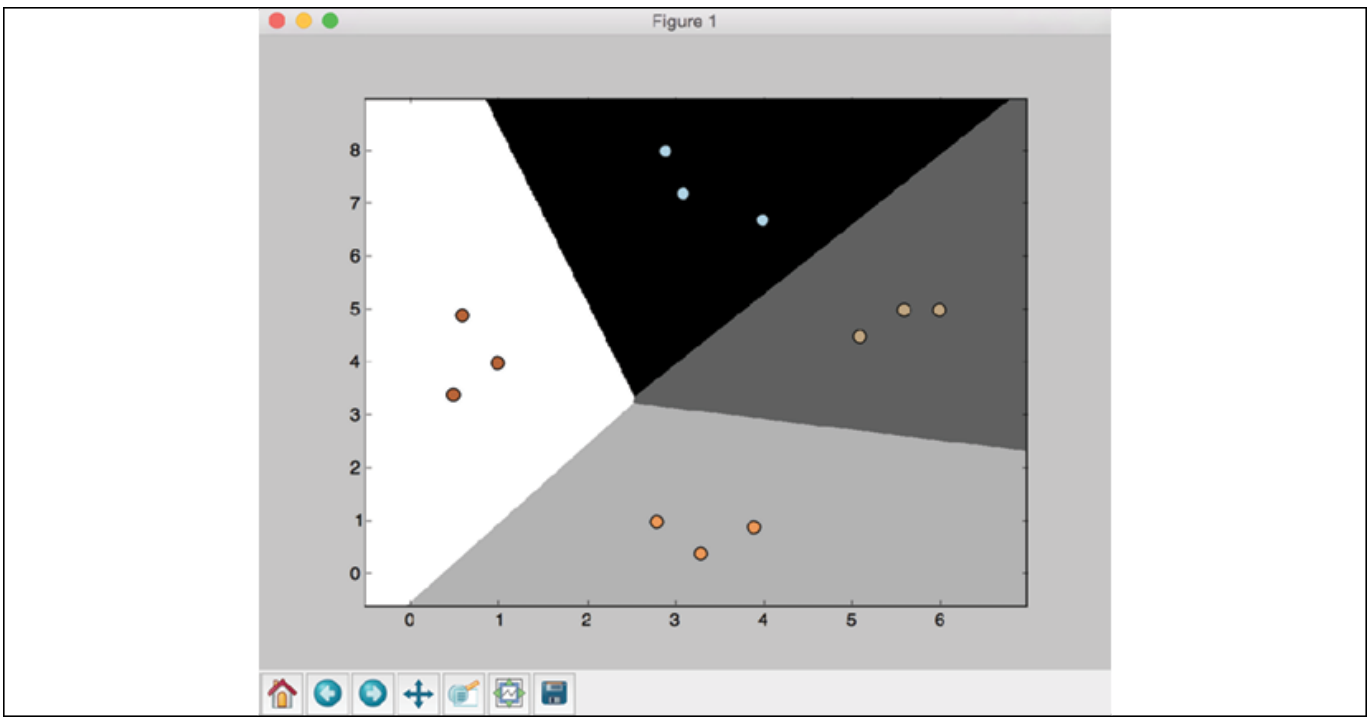


Figure 4 : Résultats de l'exécution du code avec C fixé à 100

Si vous comparez avec la figure précédente, vous verrez que les limites sont maintenant meilleures. Le code de cette section se trouve dans le fichier `logistic_regression.py.`

## Le classificateur Naïve Bayes

**Naïve Bayes** est une technique utilisée pour construire des classificateurs à l'aide du théorème de Bayes. Le théorème de Bayes décrit la probabilité qu'un événement se produise en fonction de différentes conditions liées à cet événement. Nous construisons un classificateur de Naïve Bayes en attribuant des étiquettes de classe aux instances de problèmes. Ces instances de problèmes sont représentées par des vecteurs de valeurs de caractéristiques. On suppose ici que la valeur l'une caractéristique donnée est indépendante de la valeur de toute autre caractéristique. C'est ce que l'on appelle l'hypothèse l'indépendance, qui constitue la partie naïve l'un classificateur de Naïve Bayes.

Étant donné la variable classe, nous pouvons simplement voir comment une caractéristique donnée l'affecte, indépendamment de son effet sur l'autres caractéristiques. Par exemple, un animal peut être considéré comme un guépard s'il est tacheté, s'il a quatre pattes, s'il a une queue et s'il court à une vitesse l'environ 70 MPH. Un classificateur Naïve Bayes considère que chacune de ces caractéristiques contribue indépendamment au résultat. Le résultat correspond à la probabilité que cet animal soit un guépard. Nous ne nous préoccupons pas des corrélations qui peuvent exister entre les motifs de la peau, le nombre de pattes, la présence l'une queue et la vitesse de déplacement. Voyons comment construire un classificateur Naïve Bayes.

Créez un nouveau fichier Python et importez les paquets suivants :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
```

```
from utilities import visualize_classifier
```

Nous utiliserons le fichier `data_multivar_nb.txt` comme source de données. Ce fichier contient des valeurs séparées par des virgules dans chaque ligne :

```
# Input file containing data
input_file = 'data_multivar_nb.txt'
```

Chargeons les données de ce fichier :

```
# Load data from input file
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1]
```

Créez une instance du classificateur Naïve Bayes. Nous utiliserons ici le classificateur gaussien de Naïve Bayes. Dans ce type de classificateur, nous supposons que les valeurs associées à chaque classe suivent une distribution gaussienne :

```
# Create Naïve Bayes classifier
classifier = GaussianNB()
```

Entraînez le classificateur à l'aide des données l'entraînement :

```
# Train the classifier
classifier.fit(X, y)
```

Exécutez le classificateur sur les données l'apprentissage et prédisez la sortie :

```
# Predict the values for training data
y_pred = classifier.predict(X)
```

Calculons la précision du classificateur en comparant les valeurs prédites avec les vraies étiquettes, puis visualisons les performances :

```
# Compute accuracy
accuracy = 100.0 * (y == y_pred).sum() / X.shape[0]
print("Accuracy of Naïve Bayes classifier =", round(accuracy, 2), "%")
```

```
# Visualize the performance of the classifier
visualize_classifier(classifier, X, y)
```

La méthode précédente pour calculer la précision du classificateur n'est pas robuste. Nous devons procéder à une validation croisée, afin de ne pas utiliser les mêmes données l'apprentissage lorsque nous les testons.

Divisez les données en sous-ensembles de formation et de test. Comme le précise le paramètre `test_size` de la ligne suivante, nous affecterons 80 % des données à l'apprentissage et les 20 % restants au test. Nous allons ensuite entraîner un classificateur Naïve Bayes sur ces données :

```
# Split data into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=3)
classifier_new = GaussianNB()
classifier_new.fit(X_train, y_train)
y_test_pred = classifier_new.predict(X_test)
```

Calculer la précision du classificateur et visualiser les performances :

```
# compute accuracy of the classifier
accuracy = 100.0 * (y_test == y_test_pred).sum() / X_test.shape[0]
print("Accuracy of the new classifier =", round(accuracy, 2), "%")
```

```
# Visualize the performance of the classifier
visualize_classifier(classifier_new, X_test, y_test)
```

Utilisons les fonctions intégrées pour calculer les valeurs l'exactitude, de précision et de rappel sur la base l'une validation croisée triple :

```
num_folds = 3
accuracy_values = cross_val_score(classifier,
        X, y, scoring='accuracy', cv=num_folds)
print("Accuracy: " + str(round(100*accuracy_values.mean(), 2)) + "%")

precision_values = cross_val_score(classifier,
         X, y, scoring='precision_weighted', cv=num_folds)
print("Precision: " + str(round(100*precision_values.mean(), 2)) + "%")

recall_values = cross_val_score(classifier,
        X, y, scoring='recall_weighted', cv=num_folds)
print("Recall: " + str(round(100*recall_values.mean(), 2)) + "%")

f1_values = cross_val_score(classifier,
        X, y, scoring='f1_weighted', cv=num_folds)
print("F1: " + str(round(100*f1_values.mean(), 2)) + "%")
```

Si vous exécutez le code, vous verrez ceci pour le premier entraînement :

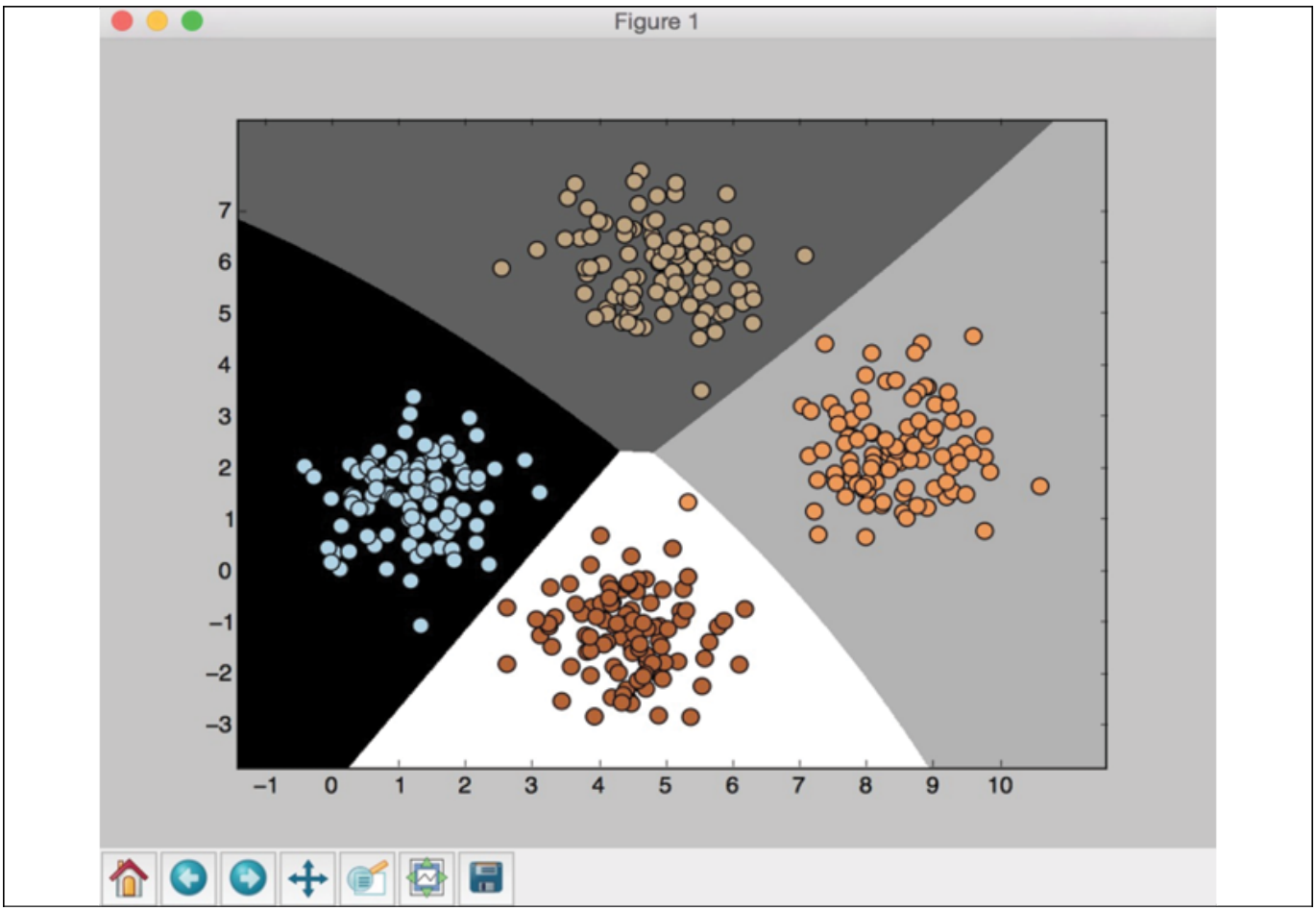


Figure 5 : Regroupement et limites après le premier cycle l'entraînement

La capture l'écran précédente montre les limites obtenues par le classificateur. Nous pouvons voir qu'elles séparent bien les quatre clusters et créent des régions avec des limites basées sur la distribution des points de données l'entrée. Vous verrez dans la capture l'écran suivante le deuxième cycle l'entraînement avec validation croisée :

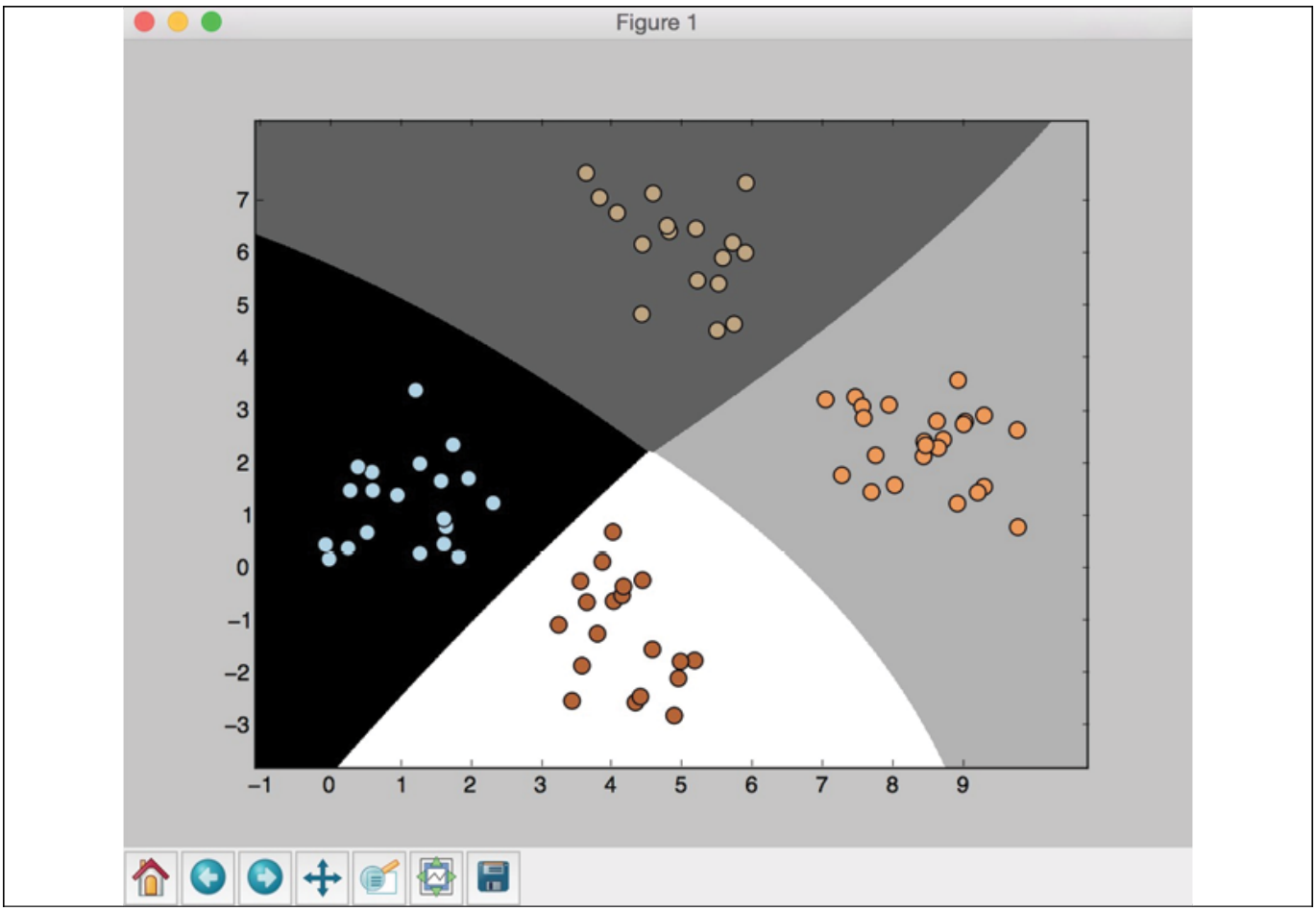


Figure 6 : Résultats de la deuxième formation, en utilisant la validation croisée

Vous obtiendrez la sortie imprimée suivante :

```
Accuracy of Naïve Bayes classifier = 99.75 %
Accuracy of the new classifier = 100.0 %
Accuracy: 99.75%
Precision: 99.76%
Recall: 99.75%
F1: 99.75%
```

Le code de cette section se trouve dans le fichier `naive_bayes.py`.

## Matrices de confusion

Une **matrice de confusion** est une figure ou un tableau utilisé pour décrire les performances l'un classificateur. Chaque ligne de la matrice représente les instances l'une classe prédite et chaque colonne représente les instances l'une classe réelle. Ce nom est utilisé parce que la matrice permet de visualiser facilement si le modèle a confondu ou mal étiqueté deux classes. Nous

comparons chaque classe avec toutes les autres classes et voyons combien l'échantillons sont classés correctement ou mal classés.

Lors de la construction de ce tableau, nous rencontrons plusieurs mesures clés qui sont importantes dans le domaine de l'apprentissage automatique. Considérons un cas de classification binaire où la sortie est soit *0*, soit *1*:

- **Vrais positifs**: Il s'agit des échantillons pour lesquels nous avons prédit *1* comme résultat et la vérité de terrain est également *1*.
- **Vrais négatifs**: Il s'agit des échantillons pour lesquels nous avons prédit *0* comme résultat et la vérité de terrain est également *0*.
- **Faux positifs**: Il s'agit des échantillons pour lesquels nous avons prédit *une valeur* de sortie de *1*, alors que la vérité de terrain est de *0*. Il s'agit également l'une *erreur de type I*.
- **Faux négatifs**: Il s'agit des échantillons pour lesquels nous avons prédit un résultat de *0*, alors que la vérité de terrain est de *1*. Il s'agit également l'une *erreur de type II*.

En fonction du problème à résoudre, nous pouvons être amenés à optimiser notre algorithme pour réduire le taux de faux positifs ou de faux négatifs. Par exemple, dans un système l'identification biométrique, il est très important l'éviter les faux positifs, car les mauvaises personnes pourraient avoir accès à des informations sensibles. Voyons comment créer une matrice de confusion.

Créez un nouveau fichier Python et importez les paquets suivants :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

Définir des étiquettes l'échantillons pour la vérité de base et la sortie prédite :

```
# Define sample labels
true_labels = [2, 0, 0, 2, 4, 4, 1, 0, 3, 3, 3]
pred_labels = [2, 1, 0, 2, 4, 3, 1, 0, 1, 3, 3]
```

Créez la matrice de confusion en utilisant les étiquettes que nous venons de définir :

```
# Create confusion matrix
confusion_mat = confusion_matrix(true_labels, pred_labels)
```

Visualisez la matrice de confusion :

```
# Visualize confusion matrix
plt.imshow(confusion_mat, interpolation='nearest', cmap=plt.cm.gray)
plt.title('Confusion matrix')
plt.colorbar()
ticks = np.arange(5)
plt.xticks(ticks, ticks)
plt.yticks(ticks, ticks)
plt.ylabel('True labels')
plt.xlabel('Predicted labels')
plt.show()
```

Dans le code de visualisation précédent, la variable `ticks` fait référence au nombre de classes distinctes. Dans notre cas, nous avons cinq étiquettes distinctes.

Imprimons le rapport de classification :

```
# Classification report
targets = ['Class-0', 'Class-1', 'Class-2', 'Class-3', 'Class-4'] print('\n',
classification_report(true_labels, pred_labels, target_names=targets))
```

Le rapport de classification imprime les performances de chaque classe. Si vous exécutez le code, vous verrez la capture l'écran suivante :

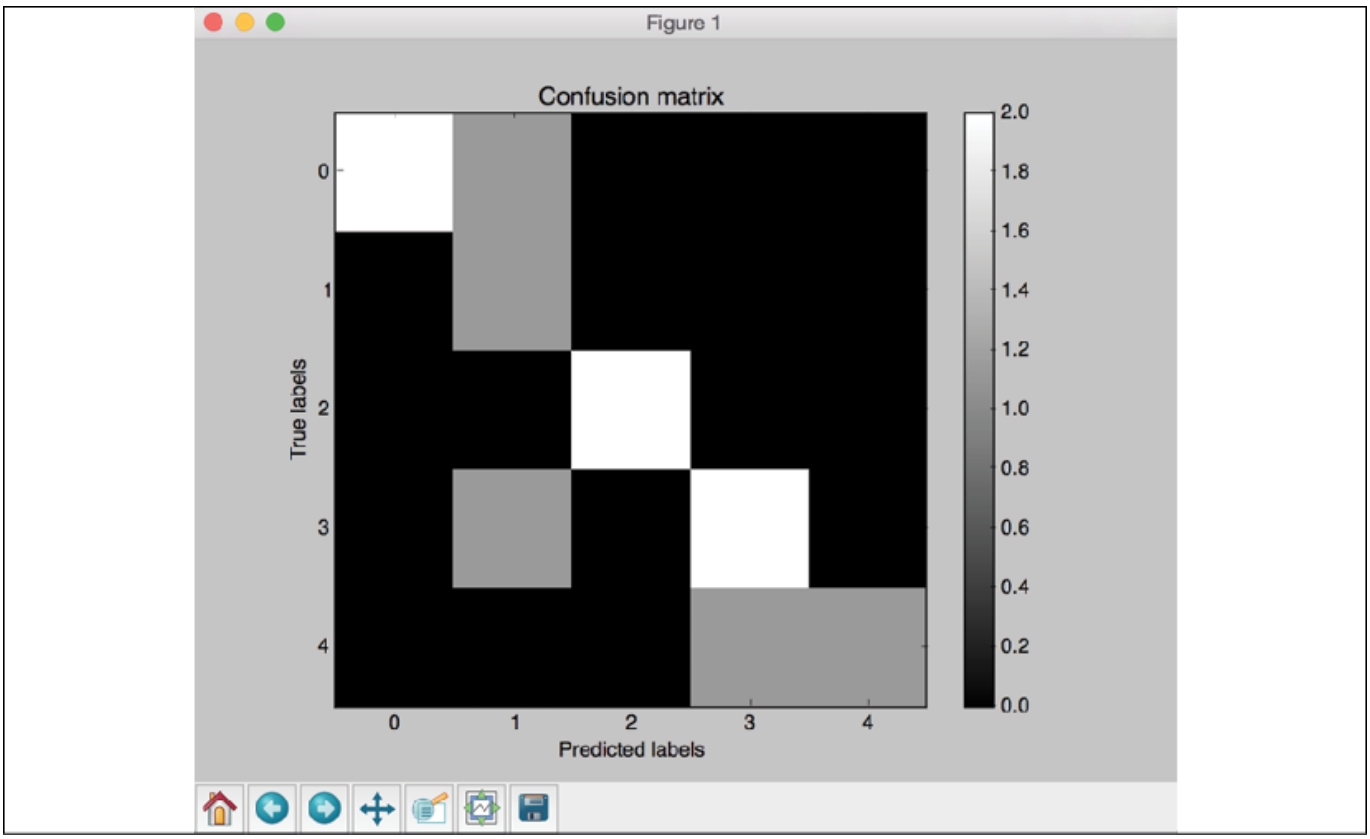


Figure 7 : Performance de chaque classe dans le rapport de classification

Le blanc indique les valeurs les plus élevées, tandis que le noir indique les valeurs les plus basses, comme le montre la clé de la carte des couleurs à droite de l'image. Dans un scénario idéal, les carrés diagonaux seront entièrement blancs et tout le reste sera noir. Cela indique une précision de 100 %.

Le résultat devrait être le suivant :

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Class-0 | 1.00 | 0.67 | 0.80 | 3 |
| Class-1 | 0.33 | 1.00 | 0.50 | 1 |
| Class-2 | 1.00 | 1.00 | 1.00 | 2 |
| Class-3 | 0.67 | 0.67 | 0.67 | 3 |
| Class-4 | 1.00 | 0.50 | 0.67 | 2 |
| avg / total | 0.85 | 0.73 | 0.75 | 11 |

Figure 8 : Valeurs du rapport de classification

Comme nous pouvons le voir, la `précision` moyenne était de 85% et le `rappel` moyen de 73%. Le `score f1` est quant à lui de 75 %. En fonction du domaine sur lequel nous travaillons, ces résultats peuvent être excellents ou médiocres. Si le domaine en question consistait à déterminer si un patient est atteint l'un cancer et que nous ne parvenions à obtenir qu'une précision de 85 %, 15 % de la population aurait été mal classée et serait assez mécontente. Si le domaine que nous analysons consiste à déterminer si une personne va acheter un produit et que notre précision donne les mêmes résultats, nous pourrions considérer qu'il s'agit l'un succès et que nos dépenses de marketing s'en trouveraient considérablement réduites.

Le code de cette section se trouve dans le fichier `confusion_matrix.py.`

## Machines à vecteurs de support

Une **machine à vecteurs de support(SVM)** est un classificateur défini à l'aide l'un hyperplan de séparation entre les classes. Cet **hyperplan** est la version à N dimensions l'une ligne. Étant donné des données l'apprentissage étiquetées et un problème de classification binaire, le SVM trouve l'hyperplan optimal qui sépare les données l'apprentissage en deux classes. Cette méthode peut facilement être étendue à un problème à `N` classes.

Considérons un cas bidimensionnel avec deux classes de points. Étant donné qu'il s'agit l'un cas en deux dimensions, nous ne devons traiter que des points et des lignes sur un plan en deux dimensions. C'est plus facile à visualiser que des vecteurs et des hyperplans dans un espace à haute dimension. Bien sûr, il s'agit l'une version simplifiée du problème du SVM, mais il est

important de le comprendre et de le visualiser avant de pouvoir l'appliquer à des données de haute dimension.

Considérons la figure suivante :

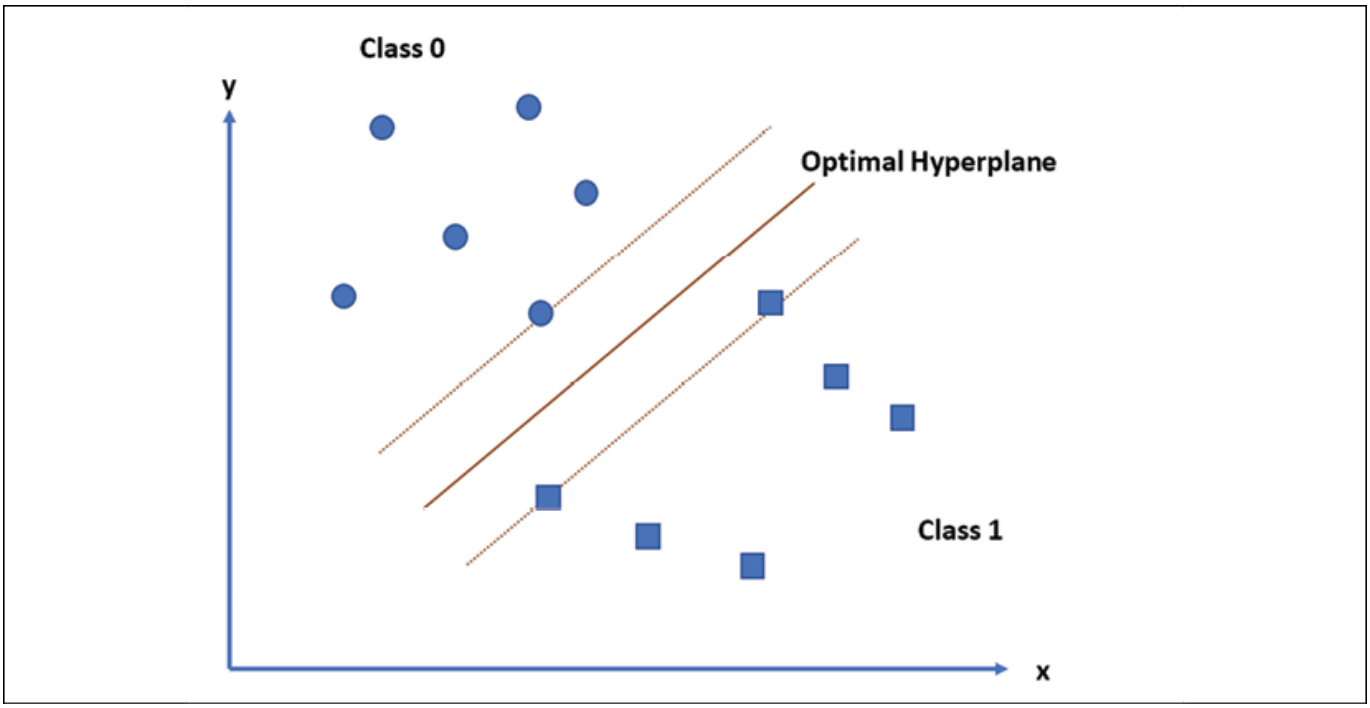


Figure 9 : Séparation de deux classes à l'aide l'un hyperplan

Il y a deux classes de points et nous voulons trouver l'hyperplan optimal pour séparer les deux classes. Mais comment définir optimal ? Dans cette image, la ligne continue représente le meilleur hyperplan. Vous pouvez tracer de nombreuses lignes différentes pour séparer les deux classes de points, mais cette ligne est le meilleur séparateur, car elle maximise la distance de chaque point par rapport à la ligne de séparation. Les points situés sur les lignes pointillées sont appelés vecteurs de support. La distance perpendiculaire entre les deux lignes en pointillés est appelée marge maximale. Vous pouvez considérer la marge maximale comme la bordure la plus épaisse pouvant être tracée pour un ensemble de données donné.

## Classification des données sur les revenus à l'aide de machines à vecteurs de support

Nous allons construire un classificateur de type Support Vector Machine pour prédire la tranche de revenus l'une personne donnée sur la base de 14 attributs. Notre objectif est de déterminer si le revenu est supérieur ou inférieur à 50 000 dollars par an. Il s'agit donc l'un problème de classification binaire. Nous utiliserons l'ensemble des données de recensement sur les revenus disponibles à l'adresse https://archive.ics.uci.edu/ml/datasets/Census+Income. Un point à noter dans cet ensemble de données est que chaque point de données est un mélange de mots et de

chiffres. Nous ne pouvons pas utiliser les données dans leur format brut, car les algorithmes ne savent pas comment traiter les mots. Nous ne pouvons pas tout convertir à l'aide l'un codeur l'étiquettes, car les données numériques sont précieuses. Nous devons donc utiliser une combinaison l'encodeurs l'étiquettes et de données numériques brutes pour construire un classificateur efficace.

Créez un nouveau fichier Python et importez les paquets suivants :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.svm import LinearSVC
from sklearn.multiclass import OneVsOneClassifier
from sklearn.model_selection import train_test_split
```

Nous utiliserons le fichier `income_data.txt` pour charger les données. Ce fichier contient les détails du revenu :

```
# Input file containing
data input_file = 'income_data.txt'
```

Afin de charger les données à partir du fichier, nous devons les prétraiter pour les préparer à la classification. Nous utiliserons au maximum 25 000 points de données pour chaque classe :

```
# Read the data X = []
y = []
count_class1 = 0
count_class2 = 0
max_datapoints = 25000
```

Ouvrez le fichier et commencez à lire les lignes :

```
with open(input_file, 'r') as f:
    for line in f.readlines():
        if count_class1 >= max_datapoints and count_class2 >= max_datapoints:
            break
        if '?' in line:
            continue
```

Chaque ligne est séparée par des virgules, nous devons donc la diviser en conséquence. Le dernier élément de chaque ligne représente l'étiquette. En fonction de cette étiquette, nous l'assignerons à une classe :

```
        data = line[:-1].split(', ')

        if data[-1] == '<=50K' and count_class1 < max_datapoints:
            X.append(data)
            count_class1 += 1

        if data[-1] == '>50K' and count_class2 < max_datapoints:
            X.append(data)
            count_class2 += 1
```

Convertir la liste en un tableau `numpy` afin qu'elle puisse être utilisée comme entrée de la fonction `sklearn` :

```
# Convert to numpy array
X = np.array(X)
```

Si un attribut est une chaîne de caractères, il doit être codé. s'il s'agit l'un nombre, il peut être conservé tel quel. Notez que nous allons nous retrouver avec plusieurs encodeurs l'étiquettes et que nous devons garder une trace de chacun l'entre eux :

```
# Convert string data to numerical data
label_encoder = []
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    if item.isdigit():
        X_encoded[:, i] = X[:, i]
    else:
        label_encoder.append(preprocessing.LabelEncoder())
        X_encoded[:, i] = label_encoder[-1].fit_transform(X[:, i])
X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

Créez le classificateur `SVM` avec un noyau linéaire :

```
# Create SVM classifier
classifier = OneVsOneClassifier(LinearSVC(random_state=0))
```

Entraîner le classificateur :

```
# Train the classifier
classifier.fit(X, y)
```

Effectuez une validation croisée en utilisant une répartition 80/20 pour la formation et le test, puis prédisez la sortie pour les données de formation :

```
# Cross validation
X_train, X_test, y_train, y_test = train_test_split.train_test_split(X, y,
test_size=0.2, random_state=5)
classifier = OneVsOneClassifier(LinearSVC(random_state=0)) classifier.fit(X_train,
y_train)
y_test_pred = classifier.predict(X_test)
```

Calculer le score F1 du classificateur :

```
# Compute the F1 score of the SVM classifier
f1 = train_test_split.cross_val_score(classifier, X, y, scoring='f1_weighted', cv=3)
print("F1 score: " + str(round(100*f1.mean(), 2)) + "%")
```

Maintenant que le classificateur est prêt, voyons comment prendre un point de données aléatoire en entrée et prédire la sortie. Définissons un tel point de données :

```
# Predict output for a test datapoint
input_data = ['37', 'Private', '215646', 'HS-grad', '9', 'Never-married', 'Handlers-
cleaners', 'Not-in-family', 'White', 'Male', '0', '0', '40', 'United-States']
```

Avant de pouvoir effectuer des prédictions, le point de données doit être codé à l'aide des codeurs l'étiquettes créés précédemment :

```
# Encode test datapoint
input_data_encoded = [-1] * len(input_data)
count = 0
for i, item in enumerate(input_data):
    if item.isdigit():
        input_data_encoded[i] = int(input_data[i])
    else:
        input_data_encoded[i] = int(label_encoder[count].transform(input_data[i]))
        count += 1
```

```
input_data_encoded = np.array(input_data_encoded)
```

Nous sommes maintenant prêts à prédire la sortie à l'aide du classificateur :

```
# Run classifier on encoded datapoint and print output
predicted_class = classifier.predict(input_data_encoded)
print(label_encoder[-1].inverse_transform(predicted_class)[0])
```

Si vous exécutez le code, il faudra quelques secondes pour entraîner le classificateur. Une fois l'entraînement terminé, vous obtiendrez le résultat suivant :

```
F1 score: 66.82%
```

Vous verrez également la sortie pour le point de données test :

```
<=50K
```

Si vous vérifiez les valeurs de ce point de données, vous verrez qu'il correspond étroitement aux points de données de la classe moins de 50 000. Vous pouvez modifier les performances du classificateur (score F1, précision ou rappel) en utilisant différents noyaux et en essayant plusieurs combinaisons de paramètres.

Le code de cette section se trouve dans le fichier `income_classifier.py.`

## Qu'est-ce que la régression ?

La **régression** est le processus l'estimation de la relation entre les variables l'entrée et de sortie. Il convient de noter que les variables de sortie sont des nombres réels à valeur continue. Il existe donc un nombre infini de possibilités. Ceci contraste avec la classification, où le nombre de classes de sortie est fixe. Les classes appartiennent à un ensemble fini de possibilités.

Dans la régression, on suppose que les variables de sortie dépendent des variables l'entrée, et nous voulons donc voir comment elles sont liées. Par conséquent, les variables l'entrée sont appelées *variables indépendantes*, également connues sous le nom de *prédicteurs*, et les variables de sortie sont appelées *variables dépendantes*, également connues sous le nom de *variables de critère*. Il n'est pas nécessaire que les variables l'entrée soient indépendantes les unes des autres ; en effet, il existe de nombreuses situations où il y a des corrélations entre les variables l'entrée.

l'analyse de régression nous aide à comprendre comment la valeur de la variable de sortie change lorsque nous faisons varier certaines variables l'entrée tout en gardant l'autres variables l'entrée fixes. Dans la régression linéaire, nous supposons que la relation entre l'entrée et la

sortie est linéaire. Cela impose une contrainte à notre procédure de modélisation, mais C'est une méthode rapide et efficace.

Parfois, la régression linéaire n'est pas suffisante pour expliquer la relation entre les entrées et les sorties. C'est pourquoi nous utilisons la régression polynomiale, qui consiste à utiliser un polynôme pour expliquer la relation entre l'entrée et la sortie. Cette méthode est plus complexe sur le plan informatique, mais elle offre une plus grande précision. En fonction du problème posé, nous utilisons différentes formes de régression pour extraire la relation. La régression est fréquemment utilisée pour prédire les prix, l'économie, les variations, etc.

## Construction l'un régresseur à une seule variable

Voyons comment construire un modèle de régression à une seule variable. Créez un nouveau fichier Python et importez les paquets suivants :

```
import pickle
import numpy as np
from sklearn import linear_model
import sklearn.metrics as sm
import matplotlib.pyplot as plt
```

Nous utiliserons le fichier `data_singlevar_regr.txt` qui vous a été fourni. Il s'agit de notre source de données :

```
# Input file containing data
input_file = 'data_singlevar_regr.txt'
```

Il s'agit l'un fichier séparé par des virgules, ce qui nous permet de le charger facilement à l'aide l'un appel de fonction en une seule ligne :

```
# Read data
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1]
```

Divisez-le en deux parties : la formation et le test :

```
# Train and test split
num_training = int(0.8 * len(X))
num_test = len(X) - num_training
```

```
# Training data
X_train, y_train = X[:num_training], y[:num_training]
```

```
# Test data
X_test, y_test = X[num_training:], y[num_training:]
```

Créez un objet régresseur linéaire et entraînez-le à l'aide des données l'entraînement :

```
# Create linear regressor object
regressor = linear_model.LinearRegression()
```

```
# Train the model using the training sets
regressor.fit(X_train, y_train)
```

Prédire la sortie pour l'ensemble de données de test à l'aide du modèle l'apprentissage :

```
# Predict the output
y_test_pred = regressor.predict(X_test)
```

Tracer la sortie :

```
# Plot outputs
plt.scatter(X_test, y_test, color='green')
plt.plot(X_test, y_test_pred, color='black', linewidth=4)
plt.xticks(())
plt.yticks(())
plt.show()
```

Calculer les mesures de performance pour le régresseur en comparant la vérité de terrain, qui se réfère aux sorties réelles, avec les sorties prédites :

```
# Compute performance metrics
print("Linear regressor performance:")
print("Mean absolute error =", round(sm.mean_absolute_error(y_test, y_test_pred), 2))
print("Mean squared error =", round(sm.mean_squared_error(y_test, y_test_pred), 2))
print("Median absolute error =", round(sm.median_absolute_error(y_test, y_test_pred),
2))
print("Explain variance score =", round(sm.explained_variance_score(y_test,
y_test_pred), 2))
print("R2 score =", round(sm.r2_score(y_test, y_test_pred), 2))
```

Une fois le modèle créé, nous pouvons l'enregistrer dans un fichier afin de pouvoir l'utiliser ultérieurement. Python fournit un module intéressant appelé `pickle` qui nous permet de faire cela :

```
# Model persistence
output_model_file = 'model.pkl'
```

```
# Save the model
with open(output_model_file, 'wb') as f:
    pickle.dump(regressor, f)
```

Chargeons le modèle à partir du fichier sur le disque et effectuons la prédiction :

```
# Load the model
with open(output_model_file, 'rb') as f:
    regressor_model = pickle.load(f)
```

```
# Perform prediction on test data
y_test_pred_new = regressor_model.predict(X_test)
print("\nNew mean absolute error =", round(sm.mean_absolute_error(y_test,
y_test_pred_new), 2))
```

Si vous exécutez le code, vous verrez l'écran suivant :

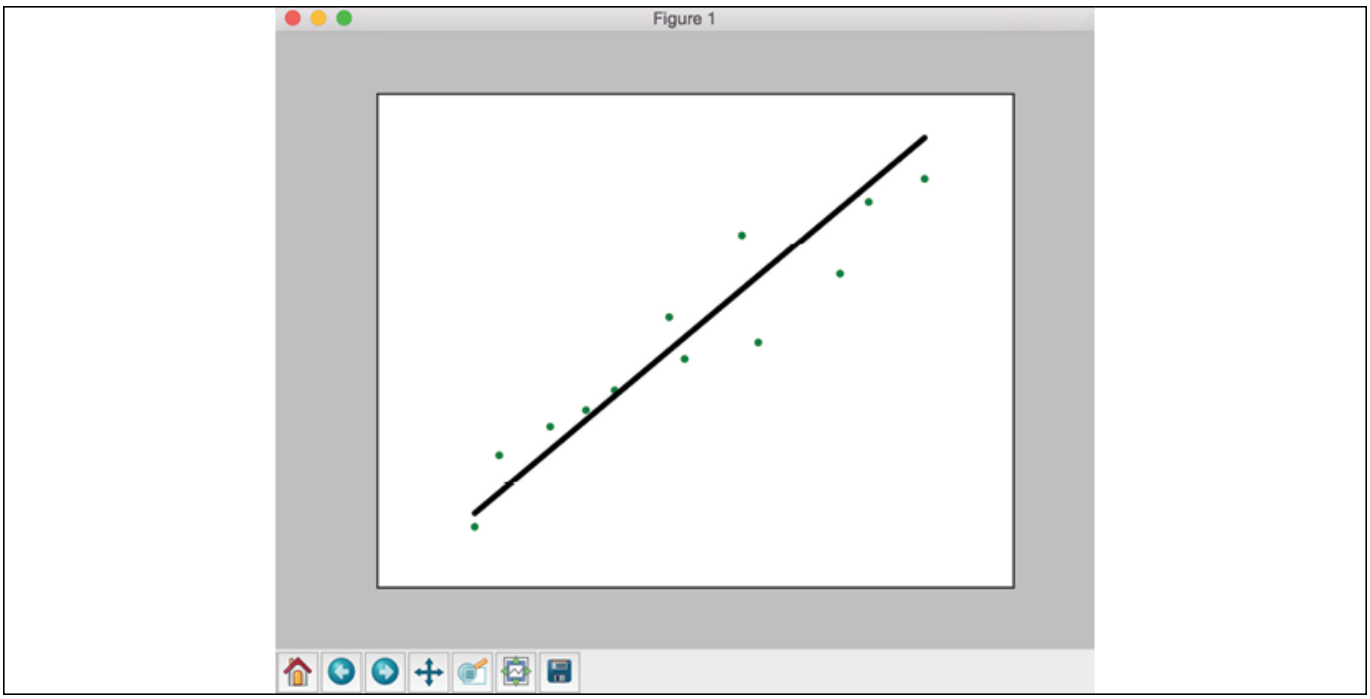


Figure 10 : Résultat après prédiction

Vous devriez obtenir le résultat suivant :

```
Linear regressor performance:
Mean absolute error = 0.59
Mean squared error = 0.49
Median absolute error = 0.51
Explain variance score = 0.86
R2 score = 0.86
New mean absolute error = 0.59
```

l'**erreur absolue moyenne(MAE)** est la moyenne des erreurs absolues :

$$|e_i| = |y_i - x_i|$$

Où ~~yi~~ est la prédiction et ~~xi~~ la valeur réelle.

l'**erreur quadratique moyenne(EQM)** est la moyenne des carrés des erreurs, C'est-à-dire la différence quadratique moyenne entre la valeur prédite et la valeur réelle. l'EQM est presque toujours strictement positive (et non nulle) en raison du caractère aléatoire. l'EQM est une mesure de la qualité l'un estimateur. Elle est toujours non négative, et plus la valeur est proche de zéro, mieux C'est.

La variation expliquée mesure la proportion dans laquelle un modèle rend compte de la variation l'un ensemble de données. Souvent, la variation est quantifiée en tant que variance ; le

terme plus spécifique de variance expliquée peut également être utilisé. Le reste de la variation totale est la variation inexpliquée ou résiduelle.

Le coefficient de détermination ou R2 est utilisé pour analyser comment les différences l'une variable peuvent être expliquées par une différence dans une deuxième variable. Par exemple, si une femme tombe enceinte, il y a une forte corrélation avec le fait qu'elle ait un enfant.

Le code de cette section se trouve dans le fichier `regressor_singlevar.py.`

## Construction l'un régresseur multivariable

Dans la section précédente, nous avons vu comment construire un modèle de régression pour une seule variable. Dans cette section, nous allons traiter des données multidimensionnelles. Créez un nouveau fichier Python et importez les paquets suivants :

```
import numpy as np
from sklearn import linear_model
import sklearn.metrics as sm
from sklearn.preprocessing import PolynomialFeatures
```

Nous utiliserons le fichier `data_multivar_regr.txt` qui vous a été fourni.

```
# Input file containing data
input_file = 'data_multivar_regr.txt'
```

Il s'agit l'un fichier séparé par des virgules, ce qui nous permet de le charger facilement à l'aide l'un appel de fonction en une seule ligne :

```
# Load the data from the input file
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1]
```

Divisez les données en deux parties, l'une pour la formation et l'autre pour le test :

```
# Split data into training and testing
num_training = int(0.8 * len(X))
num_test = len(X) - num_training
```

```
# Training data
X_train, y_train = X[:num_training], y[:num_training]
```

```
# Test data
X_test, y_test = X[num_training:], y[num_training:]
```

Créer et entraîner le modèle de régression linéaire :

```
# Create the linear regressor model
linear_regressor = linear_model.LinearRegression()
```

```
# Train the model using the training sets
linear_regressor.fit(X_train, y_train)#
```

Prédire la sortie pour l'ensemble de données de test :

```
# Predict the output
y_test_pred = linear_regressor.predict(X_test)
```

Imprimer les mesures de performance :

```
# Measure performance
print("Linear Regressor performance:")
print("Mean absolute error =", round(sm.mean_absolute_error(y_test, y_test_pred), 2))
print("Mean squared error =", round(sm.mean_squared_error(y_test, y_test_pred), 2))
print("Median absolute error =", round(sm.median_absolute_error(y_test, y_test_pred),
2))
print("Explained variance score =", round(sm.explained_variance_score(y_test,
y_test_pred), 2))
print("R2 score =", round(sm.r2_score(y_test, y_test_pred), 2))
```

Créer un régresseur polynomial de degré 10. Entraînez le régresseur sur l'ensemble de données l'entraînement. Prenons un échantillon de données et voyons comment effectuer la prédiction. La première étape consiste à le transformer en polynôme :

```
# Polynomial regression
polynomial = PolynomialFeatures(degree=10)
X_train_transformed = polynomial.fit_transform(X_train)
datapoint = [[7.75, 6.35, 5.56]]
poly_datapoint = polynomial.fit_transform(datapoint)
```

Si vous regardez de près, ce point de données est très proche du point de données de la ligne 11 de notre fichier de données, qui est *[7,66, 6,29, 5,66]*. Un bon régresseur devrait donc prédire une sortie proche de *41,35*. Créez un objet régresseur linéaire et effectuez l'ajustement polynomial. Effectuez la prédiction en utilisant à la fois des régresseurs linéaires et polynomiaux pour voir la différence :

```
poly_linear_model = linear_model.LinearRegression()
poly_linear_model.fit(X_train_transformed, y_train)
print("\nLinear regression:\n", linear_regressor.predict(datapoint))
print("\nPolynomial regression:\n", poly_linear_model.predict(poly_datapoint))
```

Si vous exécutez le code, la sortie devrait être la suivante :

```
Linear Regressor performance:
Mean absolute error = 3.58
Mean squared error = 20.31
Median absolute error = 2.99
Explained variance score = 0.86
R2 score = 0.86
```

Vous verrez également ce qui suit :

```
Linear regression:
 [ 36.05286276]
Polynomial regression:
 [ 41.46961676]
```

Comme vous pouvez le constater, la régression linéaire est de `36,05` ; le régresseur polynomial est plus proche de 41,35, de sorte que le modèle de régression polynomiale a été en mesure de faire une meilleure prédiction.

Le code de cette section se trouve dans le fichier `regressor_multivar.py.`

## Estimation des prix du logement à l'aide l'un régresseur vectoriel de soutien

Voyons comment utiliser le concept de SVM pour construire un régresseur afin l'estimer les prix des logements. Nous utiliserons l'ensemble de données disponible dans `sklearn` où chaque point de données est défini par 13 attributs.

Notre objectif est l'estimer les prix des logements en fonction de ces attributs. Créez un nouveau fichier Python et importez les paquets suivants :

```
import numpy as np
from sklearn import datasets
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, explained_variance_score
from sklearn.utils import shuffle
```

Charger l'ensemble de données sur le logement :

```
# Load housing data
data = datasets.load_boston()
```

Mélangeons les données afin de ne pas fausser notre analyse :

```
# Shuffle the data
X, y = shuffle(data.data, data.target, random_state=7)
```

Divisez l'ensemble de données en deux parties, l'une pour la formation et l'autre pour le test, selon un ratio de 80/20 :

```
# Split the data into training and testing datasets
num_training = int(0.8 * len(X))
X_train, y_train = X[:num_training], y[:num_training]
X_test, y_test = X[num_training:], y[num_training:]
```

Créez et entraînez le régresseur à vecteur de support à l'aide l'un noyau linéaire. Le paramètre `C` représente la pénalité pour l'erreur l'apprentissage. Si vous augmentez la valeur de `C`, le modèle s'ajustera davantage aux données l'apprentissage. Mais cela peut conduire à un surajustement et lui faire perdre sa généralité. Le paramètre epsilon spécifie un seuil ; il n'y a pas de pénalité pour l'erreur l'apprentissage si la valeur prédite se trouve à cette distance de la valeur réelle :

```
# Create Support Vector Regression model
sv_regressor = SVR(kernel='linear', C=1.0, epsilon=0.1)
```

```
# Train Support Vector Regressor
sv_regressor.fit(X_train, y_train)
```

Évaluez la performance du régresseur et imprimez les métriques :

```
# Evaluate performance of Support Vector Regressor
y_test_pred = sv_regressor.predict(X_test)
mse = mean_squared_error(y_test, y_test_pred)
evs = explained_variance_score(y_test, y_test_pred)
print("\n#### Performance ####")
print("Mean squared error =", round(mse, 2))
print("Explained variance score =", round(evs, 2))
```

Prenons un point de données test et effectuons une prédiction :

```
# Test the regressor on test datapoint
test_data = [3.7, 0, 18.4, 1, 0.87, 5.95, 91, 2.5052, 26, 666, 20.2,
351.34, 15.27]
print("\nPredicted price:", sv_regressor.predict([test_data])[0])
```

Si vous exécutez le code, vous devriez obtenir le résultat suivant :

```
#### Performance ####
Mean squared error = 15.41
Explained variance score = 0.82
Predicted price: 18.5217801073
```

Le code de cette section se trouve dans le fichier `house_prices.py.` Examinez la première ligne du fichier et voyez à quel point la prédiction de `18,52` est proche de la variable cible réelle.

## Résumé

Dans ce lab, nous avons appris la différence entre l'apprentissage supervisé et non supervisé. Nous avons discuté du problème de la classification des données et de la manière de le résoudre. Nous avons compris comment prétraiter les données à l'aide de différentes méthodes. Nous avons également appris à coder les étiquettes et à construire un codeur l'étiquettes. Nous avons discuté de la régression logistique et construit un classificateur de régression logistique.

Nous avons compris ce qu'est un classificateur de Naïve Bayes et avons appris à en construire un. Nous avons également appris à construire une matrice de confusion.

Nous avons discuté des machines à vecteurs de support et compris comment construire un classificateur sur cette base. Nous avons appris ce qu'est la régression et avons compris comment utiliser la régression linéaire et polynomiale pour des données à une ou plusieurs variables. Nous avons ensuite utilisé un régresseur à vecteur de support pour estimer les prix des logements à l'aide l'attributs l'entrée.