Premières classifications

Le but de ce notebook est de faire ses premiers pas en classification supervisée, i.e. lorsque les données d'apprentissage disposent de classes labelisées. Pour cela nous utiliserons un jeu de données très connu dans la communauté : IRIS.

Dans un premier temps, nous présentons une première classification pour apprendre à utiliser un classifieur et à prédire une valeur. Les jeux de données d'apprentissage et de de test sont présentés par la suite. Nous présentons ensuite différentes mesures pour évaluer un modèle. Etant donné qu'il n'est pas possible d'avoir un classifieur universel (NO FREE LUNCH THEOREM), nous verrons comment utiliser différents classifieurs et comment rechercher les meilleurs paramètres d'un classifieur. Enfin nous verrons comment sauvegarder et ré-utiliser un modèle appris.

Installation

Avant de commencer, il est nécessaire de déjà posséder dans son environnement toutes les librairies utiles. Dans la seconde cellule nous importons toutes les librairies qui seront utiles à ce notebook. Il se peut que, lorsque vous lanciez l'éxecution de cette cellule, une soit absente. Dans ce cas il est nécessaire de l'installer. Pour cela dans la cellule suivante utiliser la commande :

! pip install nom_librairie

Attention : il est fortement conseillé lorsque l'une des librairies doit être installer de relancer le kernel de votre notebook.

Remarque : même si toutes les librairies sont importées dès le début, les librairies utiles pour des fonctions présentées au cours de ce notebook sont ré-importées de manière à indiquer d'où elles viennent et ainsi faciliter la réutilisation de la fonction dans un autre projet.

```
# utiliser cette cellule pour installer les librairies manquantes
# pour cela il suffit de taper dans cette cellule : !pip install nom_librairie_m
# d'exécuter la cellule et de relancer la cellule suivante pour voir si tout se
# recommencer tant que toutes les librairies ne sont pas installées ...


#!pip install ..

# ne pas oublier de relancer le kernel du notebook
```

```
# Importation des différentes librairies utiles pour le notebook

#Sickit learn met régulièrement à jour des versions et
#indique des futurs warnings.
#ces deux lignes permettent de ne pas les afficher.
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

# librairies générales
import pickle
import pandas as pd
from scipy.stats import randint
import numpy as np
import string
import time
import base64
import sys


# librairie affichage
import matplotlib.pyplot as plt
import seaborn as sns


# librairies scikit learn
import sklearn
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import precision_recall_fscore_support as score
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
```

```
from sklearn.svm import SVC
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn import datasets
```

Pour pouvoir sauvegarder sur votre répertoire Google Drive, il est nécessaire de fournir une autorisation. Pour cela il suffit d'éxecuter la ligne suivante et de saisir le code donné par Google.

```
# pour monter son drive Google Drive local
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, c
```

Corriger éventuellement la ligne ci-dessous pour mettre le chemin vers un répertoire spécifique dans votre répertoire Google Drive :

```
my_local_drive='/content/gdrive/My Drive/Colab Notebooks/ML_FDS'
# Ajout du path pour les librairies, fonctions et données
sys.path.append(my_local_drive)
# Se positionner sur le répertoire associé
%cd $my_local_drive

%pwd
```

```
/content/gdrive/My Drive/Colab Notebooks/ML_FDS
'/content/gdrive/My Drive/Colab Notebooks/ML_FDS'
```

```
# fonctions utilities (affichage, confusion, etc.)
from MyNLPUtilities import *
```

Le jeu de données IRIS

IRIS est un jeu de données multivariées qui a été présenté la première fois par R. Fisher en 1936 (R. Fisher, "The use of multiple measurements in taxonomic problems", Annals Eugen. 7 (1936) 179-188). Introduit tout d'abord comme exemple d'application de l'analyse discriminante linéaire, il a été très largement utilisé par la communauté de l'apprentissage automatique.

Le jeu de données est constitué de 3 classes correspondant à des espèces différentes d'Iris : 'Iris Setosa', 'Iris Virginica' et 'Iris Versicolor' et possède 4 caractéristiques correspondant à : la longueur et largeur des sépales et des pétales en centimètres.

tiré de : https://www.datacamp.com/community/tutorials/machine-learning-in-r

Tout au long de ce notebook, nous illustrerons les principales notions à l'aide de ce jeu de données.

```
import pandas as pd
import numpy as np

# Le jeu de données est disponible sur le site de l'UCI
# https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data

# chargement du jeu de données à partir de l'URL
url="https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['SepalLengthCm', 'SepalWidthCm',
         'PetalLengthCm', 'PetalWidthCm',
         'Species']

# creation d'un dataframe pour récupérer les données
df_iris = pd.read_csv(url, names=names)

# affichage des 5 premières lignes du jeu de données
display(df_iris.head())

# affichage du nombre de classes et de val
print ("Nombre d'occurrences par classe : \n",df_iris['Species'].value_counts())
```

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
Nombre d'occurrences par classe :
 Iris-versicolor    50
Iris-virginica     50
Iris-setosa        50
Name: Species, dtype: int64
```

Toute première classification

La classification supervisée considère les données sous la forme (X,Y) où X correspond aux variables prédictives et Y est le résultat d'une observation, i.e. la variable à prédire. En se basant sur un jeu d'apprentissage, un algorithme de classification supervisée cherche une fonction mathématique F qui permet de transformer (au mieux) X vers Y, i.e. $F(X) \approx Y$.

Convention : les variables prédictives sont celles associées aux objets, généralement stockées sous la forme d'une matrice aussi, par convention, elles sont souvent notées en majuscule (notation d'une matrice). Les variables à prédire sont généralement stockées dans un vecteur et sont souvent notées avec une lettre majuscule (notation d'un vecteur).

Autrement il est tout à fait possible d'utiliser des noms de variables significatives comme data, target.

```
# Conversion du dataframe
array = df_iris.values #necessité de convertir le dataframe en numpy

# Récupération des différentes colonnes
#X matrice représentant les variables prédictives
X = array[:,0:4]
#y vecteur : représentant la variable à prédire
y = array[:,4]
```

Pour apprendre un modèle la première chose à faire est de lui créer un estimateur. Scikit-learn, permet de faire le faire en appelant la méthode *fit(X, y)*.

Pour une première illustration, nous utilisons un classifieur naïve Bayes (https://scikit-learn.org/stable/modules/naive_bayes.html).

```
import sklearn
from sklearn.naive_bayes import GaussianNB

clf = GaussianNB()

# création de l'estimateur
clf.fit(X, y)
```

```
GaussianNB()
```

Il est possible d'obtenir les hyperparamètres d'un classifieur à l'aide de *get_params*.

```
clf.get_params()
```

```
{'priors': None, 'var_smoothing': 1e-09}
```

Pour prédire la classe d'une valeur, il suffit d'appliquer la méthode *predict* sur des variables prédictives. Par exemple, nous savons que les valeurs du 5ième IRIS sont 5.0, 3.6, 1.4, 0.2 et qu'il appartient à la classe Iris-setosa.

```
#Prediction du résultat

result = clf.predict([[ 5.0,  3.6,  1.4,  0.2]])
print ('La prédiction du modèle pour [ 5.0,  3.6,  1.4,  0.2] est',
       result)
```

```
La prédiction du modèle pour [ 5.0,  3.6,  1.4,  0.2] est ['Iris-setosa']
```

Et si nous appliquions notre modèle sur ..... nos données d'apprentissage. Bien sûr cela n'a aucune sens ! l'objectif ici est de voir un peu comment notre modèle se comporte.

```
result = clf.predict(X)
print (result)
```

```
['Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-virginica' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-versicolor' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-versicolor' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-versicolor'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica']
```

Une première évaluation de la qualité de la prédiction peut se faire avec le calcul de l'*accuracy* (pourcentage de prédictions correctes).

```
from sklearn.metrics import accuracy_score

print ('accuracy: ',accuracy_score(result, y))
```

```
accuracy:  0.96
```

Comme nous pouvons le constater, même sur le jeu d'apprentissage il y a des erreurs dans le modèle appris. Pour connaître les objets mal classés :

```
y = np.asarray(y)
misclassified = np.where(y != clf.predict(X))

print('Les objets mal classés sont :')

i=0
for i in misclassified:
    print('\n',df_iris.iloc[i,:]['Species'])


print ("ils sont classés respectivement en :")
for i in misclassified:
    print ('\n', i,'classé en ',clf.predict(X)[i],'\n')
    print ('\n')
```

```
 Les objets mal classés sont :

  52     Iris-versicolor
 70     Iris-versicolor
 77     Iris-versicolor
 106     Iris-virginica
 119     Iris-virginica
 133     Iris-virginica
 Name: Species, dtype: object
 ils sont classés respectivement en :

  [ 52  70  77 106 119 133] classé en  ['Iris-virginica' 'Iris-virginica' 'I
  'Iris-versicolor' 'Iris-versicolor']
```

Jeu d'apprentissage et de test

En classification, il est indispensable de créer un jeu d'apprentissage sur lequel un modèle est appris et un jeu de test pour évaluer le modèle. 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.

Le paramètre *train_size* indique la taille du jeu d'apprentissage qui sera utilisé.
le paramètre *random_state* spécifie un entier germe du nombre aléatoire pour le tirage. S'il n'est pas spécifié sickit learn utilise un générateur de nombre aléatoire à partir de np.random.

Dans notre exemple nous prenons 30% du jeu de données comme jeu de test.

```
from sklearn.model_selection import train_test_split

trainsize=0.7 # 70% pour le jeu d'apprentissage, il reste 30% du jeu de données

testsize= 0.3
seed=30
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=trainsize,
                                               random_state=seed,
                                               test_size=testsize)
```

L'apprentissage du modèle se fait comme précédemment

```
clf = GaussianNB()
clf.fit(X_train, y_train)
```

```
GaussianNB()
```

De même pour la prédiction

```
from sklearn.metrics import accuracy_score

y_pred = clf.predict(X_test)
print('\n accuracy : %0.3f'%(accuracy_score(y_pred, y_test)),'\n')
```

```
 accuracy : 0.956
```

Le problème essentiel de cette approche est que le modèle est appris sur un seul jeu de données et qu'en fonction de la sélection les résultats peuvent être très différents. La bonne solution consiste à utiliser la **cross validation**. Dans notre cas, nous allons utiliser une 10-fold cross validation pour évaluer la qualité. Le jeu de données sera découpé en 10 parties, entrainé sur 9, testé sur 1 et cela sera répété pour toutes les combinaisons du découpage.

```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
seed=7
k_fold = KFold(n_splits=10, shuffle=True, random_state=seed)
```

```
clf = GaussianNB()

scoring = 'accuracy'
score = cross_val_score(clf, X, y, cv=k_fold, scoring=scoring)

print('Les différentes accuracy pour les 10 évaluations sont : \n',
      score,'\n')
print ('Accuracy moyenne : %0.3f'%(score.mean()),
       ' standard deviation %0.3f'%(score.std()))
```

```
Les différentes accuracy pour les 10 évaluations sont :
 [0.8        0.86666667 1.         1.         0.93333333 1.
 1.         1.         0.93333333 1.        ]

Accuracy moyenne : 0.953  standard deviation 0.067
```

L'écart type est très important car il montre les grandes variations qui peuvent exister par rapport aux jeux de données.

Plus loin sur l'évaluation d'un modèle

L'accuracy (nombre d'objets correctement classés) est la métrique la plus simple pour comprendre le résultat de la classification mais ne tient pas du tout compte de la distribution des données et ne permet pas d'indiquer les erreurs. Par exemple avec des classes très déséquilibrées (1 vs 99), nous pouvons avoir un modèle avec une accuracy de 99%.

Par la suite, par simplification, nous reprenons une classification réalisée sans cross validation mais le principe est évidemment le même avec cross validation. Nous introduisons la matrice de correlation et les différentes mesures : precision, rappel et F1-score.

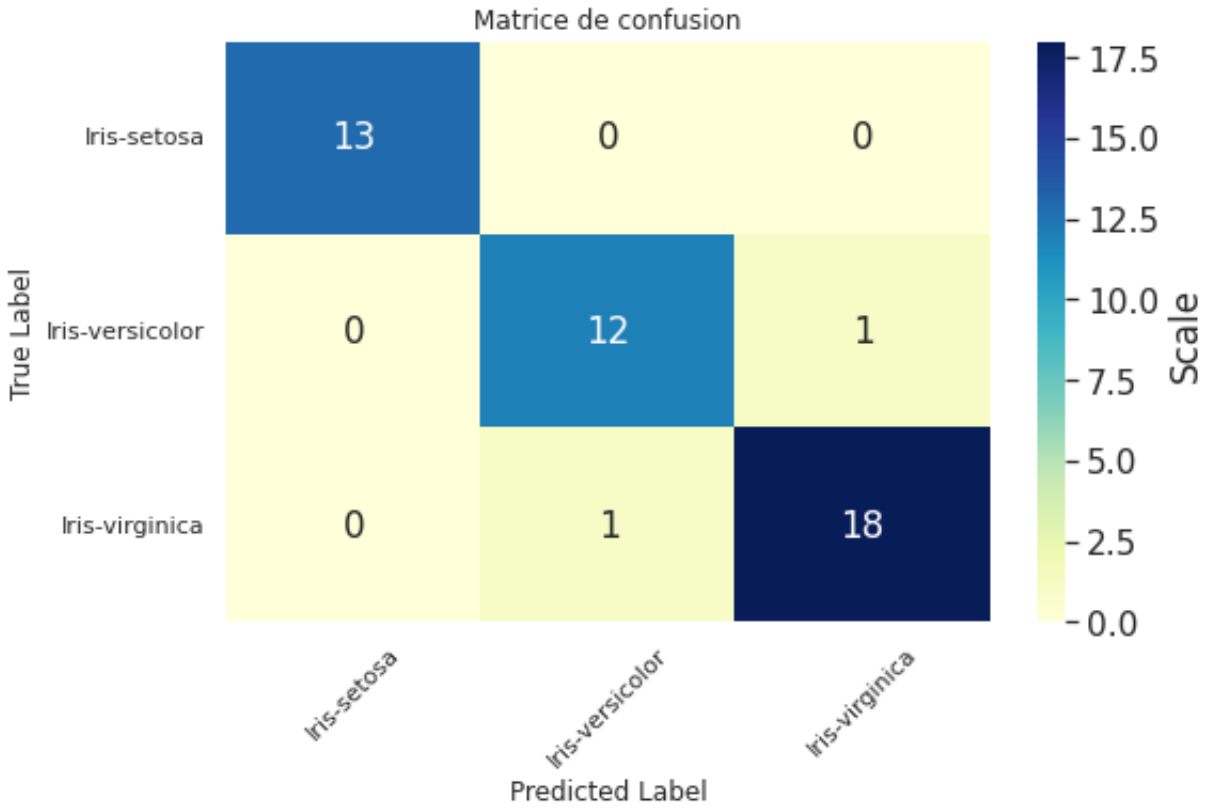
```
array = df_iris.values
X = array[:,0:4]
y = array[:,4]

trainsize=0.7 # 70% pour le jeu d'apprentissage, il reste 30% du jeu de données

testsize= 0.3
seed=30
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=trainsize,
                                               random_state=seed,
                                               test_size=testsize)
clf = GaussianNB()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print('\n accuracy : %0.3f'%(accuracy_score(y_pred, y_test)),'\n')
```

```
     accuracy : 0.956
```

La matrice de confusion permet de connaître les objets bien ou mal classés. Il suffit d'utiliser la fonction *confusion_matrix*. Nous visualisons également cette matrice.

```
from sklearn.metrics import confusion_matrix

conf = confusion_matrix(y_test, y_pred)
print ('\n matrice de confusion \n',conf)
plot_confusion_matrix(conf, ['Iris-setosa','Iris-versicolor','Iris-virginica'])
```

```
matrice de confusion
[[13  0  0]
 [ 0 12  1]
 [ 0  1 18]]
```



Il est possible d'obtenir plus d'information : *precision*, *recall* et *f1-measure* à l'aide de *classification_report*.

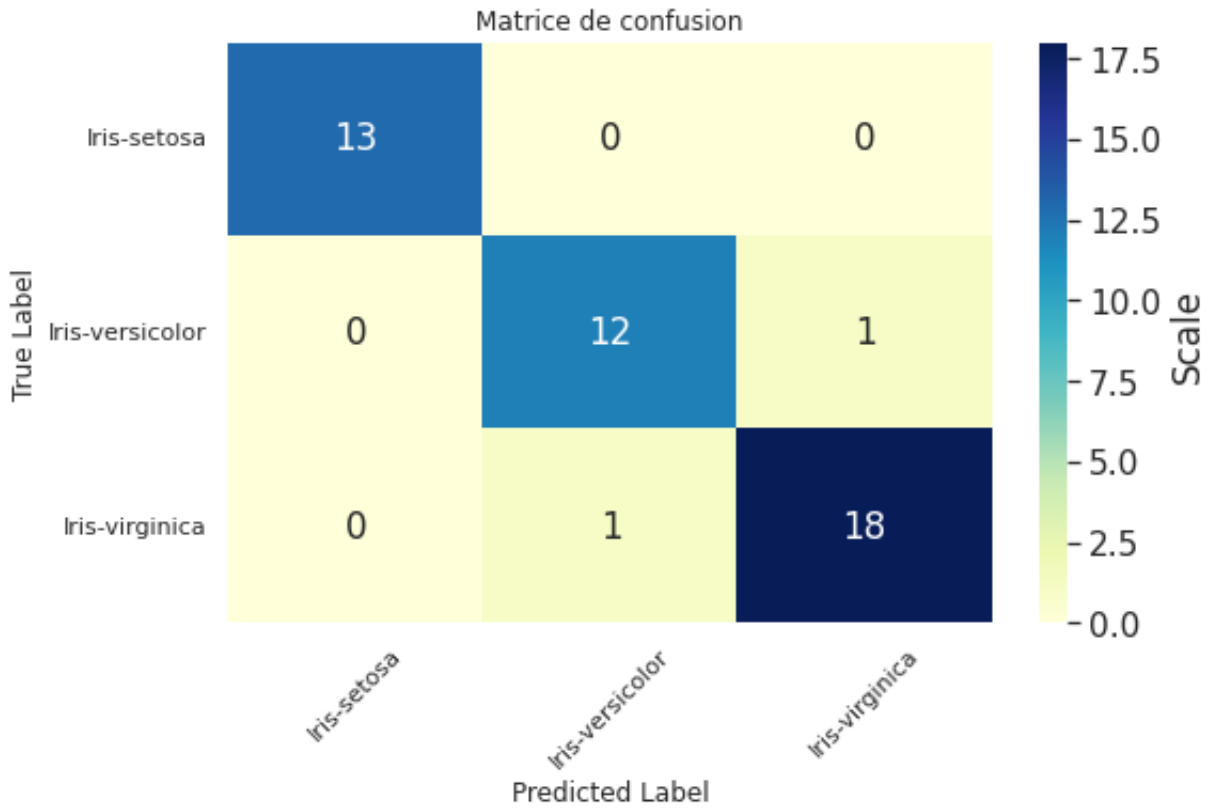
```
from sklearn.metrics import classification_report
conf = confusion_matrix(y_test, y_pred)
print ('\n matrice de confusion \n',conf)
print ('\n',classification_report(y_test, y_pred))
```

```
 matrice de confusion
 [[13  0  0]
 [ 0 12  1]
 [ 0  1 18]]

                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        13
Iris-versicolor       0.92      0.92      0.92        13
 Iris-virginica       0.95      0.95      0.95        19

       accuracy                           0.96        45
      macro avg       0.96      0.96      0.96        45
   weighted avg       0.96      0.96      0.96        45
```

Par la suite, pour afficher les résultats de classification : rapport de classification et matrice de confusion, nous utilisons la fonction suivante :

```
# fonction qui affiche le classification report et la matrice de confusion
def MyshowAllScores(y_test,y_pred):
  classes= np.unique(y_test)
  print("Accuracy : %0.3f"%(accuracy_score(y_test,y_pred)))
  print("Classification Report")
  print(classification_report(y_test,y_pred,digits=5))
  cnf_matrix = confusion_matrix(y_test,y_pred)
  plot_confusion_matrix(cnf_matrix, classes)

MyshowAllScores(y_test,y_pred)
```

```
Accuracy : 0.956
Classification Report
                 precision    recall  f1-score   support

    Iris-setosa    1.00000   1.00000   1.00000        13
Iris-versicolor    0.92308   0.92308   0.92308        13
 Iris-virginica    0.94737   0.94737   0.94737        19

       accuracy                        0.95556        45
      macro avg    0.95682   0.95682   0.95682        45
   weighted avg    0.95556   0.95556   0.95556        45
```



Rappel :

Considérons une matrice de confusion dans un cas binaire. Par exemple présence de SPAM ou non dans des mails.

$N = 115$	**PREDIT NON**	**PREDIT OUI**
REEL NON	60	10
REEL OUI	5	40

La matrice nous permet de voir qu'il y a deux classes prédites (OUI ou NON). Le classifieur fait un total de 115 prédictions. Sur ces 115 cas, le classifieur a prédit OUI 50 fois et NON 65 fois. En fait 45 documents sont des SPAMS et 70 ne le sont pas.

TP (True positive) : il s'agit des objets qui étaient prédits OUI (il s'agit de SPAM) et qui sont effectivement des SPAM.
TN (True negative) : il s'agit des objets qui étaient prédits NON (il ne s'agit pas de SPAM) et qui effectivement ne sont pas des SPAM.
FP (False positive) : il s'agit des objets qui étaient prédits comme SPAM mais qui en fait n'étaient pas des SPAM.
FN (False negative) : il s'agit des objets qui étaient prédits comme non SPAM qui en fait s'avèrent être des SPAM.
Dans la matrice ci-dessous ces éléments sont reportés :

$N = 115$	**PREDIT NON**	**PREDIT OUI**	
REEL NON	$TN = 60$	$FP = 10$	70
REEL OUI	$FN = 5$	$TP = 40$	45
	65	50	

L'**accuracy** correspond au pourcentage de prédiction correcte. Elle est définie par

$$\frac{TP + TN}{TN + FP + FN + TP} = \frac{40 + 60}{60 + 10 + 5 + 40} = 0.86.$$

Le **recall** (ou sensitivity ou True Positive Rate ou rappel) correspond au nombre d' objets pertinents retrouvés par rapport aux nombres d'objets pertinents du jeu de données. Dans notre cas, pour tous les OUI présents combien de fois le OUI a t'il été prédit ?

$$recall = \frac{Nombre\ de\ SPAM\ correctement\ reconnus}{Nombre\ total\ de\ SPAM\ dans\ le\ jeu\ de\ données} = \frac{TP}{FN+TP} = \frac{40}{40+5}$$

La **precision** correspond à la proportion d'objets pertinents parmi les objets sélectionné. Tous les objets retournés non pertinents constituent du bruit.

$$precision = \frac{Nombre\ de\ SPAM\ correctement\ reconnus}{Nombre\ de\ fois\ où\ un\ objet\ a\ été\ prédit\ SPAM} = \frac{TP}{TP+FP} = \frac{40}{40+}$$

Le **f1-score** (ou f-measure) est la moyenne harmonique du rappel et de la précision.

$$f1-score = 2 \times \frac{precision \times recall}{precision + recall} = 2 \times \frac{0.8 \times 0.88}{0.8 + 0.88}.$$

Dans le cas d'une classification multiclasse, à partir de la matrice de confusion, la precision est calculée, pour une colonne i, par :

$$precision_i = \frac{M_{ii}}{\sum_j M_{ji}}$$

et le recall par :

$$recall_i = \frac{M_{ii}}{\sum_j M_{ij}}$$

Pour la matrice de confusion suivante :

$Iris-setosa$	34	0	0
$Iris-versicolor$	0	33	5
$Iris-virginica$	0	1	32

classification_report retourne le résultat suivant :

	$precision$	$recall$	$f1-score$	$support$
$Iris-setosa$	1.00	1.00	1.00	34
$Iris-versicolor$	0.97	0.87	0.92	38
$Iris-virginica$	0.86	0.97	0.91	33
$avg/total$	0.95	0.94	0.94	105

La precision d'Iris-versicolor est obtenue par :

$$precision_i = \frac{M_{ii}}{\sum_j M_{ji}} = \frac{33}{33+1} = 0.97.$$

Le rappel d'Iris-versicolor est obtenue par :

$$recall_i = \frac{M_{ii}}{\sum_j M_{ij}} = \frac{33}{33+5} = 0.87.$$

La precision d'Iris-virginica est obtenue par :

$$precision_i = \frac{M_{ii}}{\sum_j M_{ji}} = \frac{32}{32+5} = 0.86.$$

Le rappel d'Iris-versicolor est obtenue par :

$$recall_i = \frac{M_{ii}}{\sum_j M_{ij}} = \frac{32}{32+1} = 0.96.$$

Les métriques peuvent être appelées indépendamment :

```
from sklearn.metrics import precision_recall_fscore_support as score

precision, recall, fscore, support = score(y_test, y_pred)

print('precision: {}'.format(precision))
print('recall: {}'.format(recall))
print('fscore: {}'.format(fscore))
print('support: {}'.format(support))
```

```
precision: [1.         0.92307692 0.94736842]
recall: [1.         0.92307692 0.94736842]
fscore: [1.         0.92307692 0.94736842]
support: [13 13 19]
```

Remarque : il existe, bien entendu, d'autres mesures pour évaluer un classifieur. Par exemple, la sensiblité, la spécificité, l'air sous la courbe roc (AUC), l'indice de Gini, etc.

Utiliser plusieurs classifiers

Comme l'indique le NO FREE LUNCH THEOREM il n'existe pas un classifieur universel et en fonction des données il est souvent nécessaire d'en évaluer plusieurs pour retenir le plus efficace. Le principe est similaire au précédent, il suffit de les mettre dans une structure et de boucler dessus.

Dans cette section, nous transformons très légérement nos données. Si nous regardons les différentes valeurs associées aux caractéristiques nous voyons que celles-ci sont assez grandes (e.g. 5.9 cm). Cela peut fortement impacter les différents classifiers (e.g. SVM), aussi nous allons normaliser les données de manière à ce qu'elles soient dans un intervalle de valeur. Pour cela, nous utilisons *StandardScaler* qui va transformer les données telles que la distribution aura une valeur moyenne de 0 et un écart type de 1.

Comme vous pouvez le constater dans la cellule suivante, le principe pour cette transformation est assez similaire à ce que nous avons vu précédemment : application d'un estimateur (ici le changement de distribution) et transformation des données.

```
from sklearn.preprocessing import StandardScaler



# Certains algorithmes, notamment SVM qui résout simplement un problème d'optimi
# sont très sensibles et ne peuvent pas converger si les valeurs ne sont pas nor
# Normalisation en utilisant StandardScaler qui transforme les caractéristiques
# en valeurs entre [-1 .. 1].
# Cette plage de valeurs peut être changée via les parametres feature_range=(min

# creation d'un objet de la classe StandardScaler
standardscaler = StandardScaler()

# application du changement de distribution aux variables descriptives

X_standardscale = standardscaler.fit_transform(X)


X=X_standardscale

trainsize=0.7 # 70% pour le jeu d'apprentissage, il reste 30% du jeu de données

testsize= 0.3
seed=30
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=trainsize,
                                               random_state=seed,
                                               test_size=testsize)
```

Dans la suite de la section, nous utilisons différents types de classifier : 'LogisticRegression', 'DecisionTree','KNeighbors', 'GaussianNB' et 'SVM'.
Les paramètres utilisés pour chacune des approches sont ceux par défaut. Pour chaque approche nous faisons une cross validation de 10.

```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC

seed = 7
scoring = 'accuracy'
models = []
models.append(('LR', LogisticRegression(solver='lbfgs')))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto')))
```

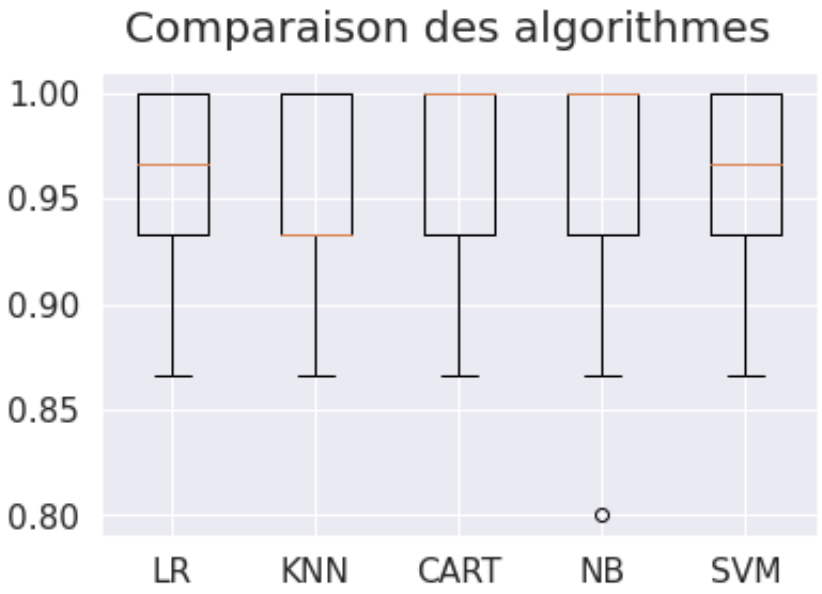
Une fois les modèles définis, il suffit de boucler en faisant une cross validation

```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
import time
results = []
names = []
for name,model in models:
    kfold = KFold(n_splits=10, random_state=seed,shuffle=True)
    start_time = time.time()
    cv_results = cross_val_score(model, X, y, cv=kfold, scoring=scoring)
    #pour avoir les paramètres utilisés dans le modèle enlever commentaire ligne
    #print (model.get_params())
    print ("Time pour",name," %0.5f"%(time.time() - start_time),'s')
    results.append(cv_results)
    names.append(name)
    msg = "%s: %0.3f (%0.3f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

```
Time pour LR  0.13586 s
LR: 0.953 (0.052)
Time pour KNN  0.03219 s
KNN: 0.953 (0.043)
Time pour CART  0.01810 s
CART: 0.960 (0.053)
Time pour NB  0.01944 s
NB: 0.953 (0.067)
Time pour SVM  0.02266 s
SVM: 0.960 (0.044)
```

```
fig = plt.figure()
fig.suptitle('Comparaison des algorithmes')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
```

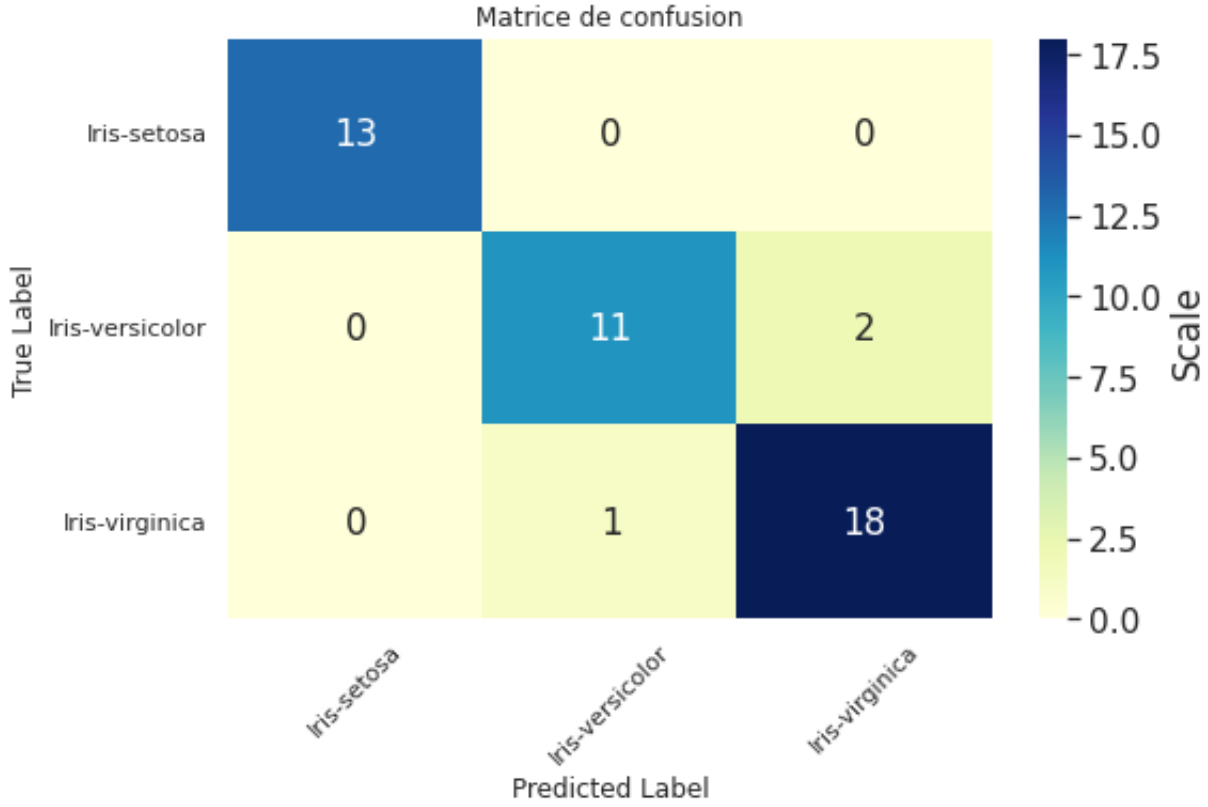
```
[Text(0, 0, 'LR'),
 Text(0, 0, 'KNN'),
 Text(0, 0, 'CART'),
 Text(0, 0, 'NB'),
 Text(0, 0, 'SVM')]
```



SVM et NB donnent sensiblement les mêmes résultats. Par la suite, nous utilisons SVM comme modèle de prédiction.

```
clf = SVC(gamma='auto')
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
MyshowAllScores(y_test,y_pred)
```

```
Accuracy : 0.933
Classification Report
                 precision    recall  f1-score   support

    Iris-setosa    1.00000   1.00000   1.00000        13
Iris-versicolor    0.91667   0.84615   0.88000        13
 Iris-virginica    0.90000   0.94737   0.92308        19

       accuracy                        0.93333        45
      macro avg    0.93889   0.93117   0.93436        45
   weighted avg    0.93370   0.93333   0.93285        45
```



Les hyperparamètres

Dans l'approche précédente nous avons pris les valeurs par défaut pour les différents classifieurs. Cependant en fonction des paramètres du classifieur les résultats peuvent être complétement différents (choix du noyeau SVM, nombre de K dans KNeighbors, etc.). Scikit learn permet de pouvoir faire une recherche exhaustive (grid search) pour trouver les paramètres les plus pertinents pour un classifieur.

```
array = df_iris.values
X = array[:,0:4]
y = array[:,4]

standardscaler = StandardScaler()
X_standardscale = standardscaler.fit_transform(X)
X=X_standardscale

trainsize=0.7 # 70% pour le jeu d'apprentissage, il reste 30% du jeu de données

testsize= 0.3
seed=30
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=trainsize,
                                               random_state=seed,
                                               test_size=testsize)
```

Considérons un arbre de décision. Les principaux paramètres sont le critère pour découper (gini ou entropy), la profondeur maximale de l'arbre, et le nombre d'échantillons par feuille. Il faut, dans un premier temps, initialiser les variables à tester dans un dictionnaire.
Le test de toutes les valeurs se fait à l'aide de la fonction *GridSearchCV*. Ele prend comme paramètre le classifieur, le dictionnaire des paramètres, le type de scoring, le nombre de crossvalidation.

Quelques paramètres souvent utilisés :

- *n_jobs* : (par défaut 1) nombre de coeurs à utiliser pour effectuer les calculs, dépend du cpu. Si la machine possède plusieurs coeurs, il est possible d'indiquer de tous les utiliser en mettant *n_jobs=-1*
- *verbose* : affichage du déroulement des calculs, 0 = silent.
- *random_state* : si le classifieur utilisé utilise de l'aléatoire, random_state permet de fixer le générateur pour reproduire les résultats.

Un grid search est long à obtenir dans la mesure où il faut essayer l'ensemble des cas. La possibilité de répartir sur plusieurs processeur permet de faire gagner beaucoup de temps.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

grid_param = {
    'max_depth': [1,2,3,4,5,6,7,8,9,10],
    'criterion': ['gini', 'entropy'],
    'min_samples_leaf': [1,2,3,4,5,6,7,8,9,10]
}


gd_sr = GridSearchCV(estimator=DecisionTreeClassifier(),
                     param_grid=grid_param,
                     scoring='accuracy',
                     cv=5,
                     n_jobs=-1,
                    return_train_score=True)

gd_sr.fit(X_train, y_train)
```

```
GridSearchCV(cv=5, estimator=DecisionTreeClassifier(), n_jobs=-1,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                         'min_samples_leaf': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10
             return_train_score=True, scoring='accuracy')
```

Pour connaître les meilleures conditions :

```
print ('meilleur score %0.3f'%(gd_sr.best_score_),'\n')
print ('meilleurs paramètres', gd_sr.best_params_,'\n')
print ('meilleur estimateur',gd_sr.best_estimator_,'\n')
```

```
meilleur score 0.971

meilleurs paramètres {'criterion': 'gini', 'max_depth': 2, 'min_samples_lea

meilleur estimateur DecisionTreeClassifier(max_depth=2)
```

Avec KNeighborsClassifier

```
from sklearn.neighbors import KNeighborsClassifier
grid_param = {
    'n_neighbors': list(range(1,15)),
    'metric': ['minkowski','euclidean','manhattan']
}

gd_sr = GridSearchCV(estimator=KNeighborsClassifier(),
                     param_grid=grid_param,
                     scoring='accuracy',
                     cv=5,
                     n_jobs=-1,
                    return_train_score=True)

gd_sr.fit(X_train, y_train)

print ('meilleur score %0.3f'%(gd_sr.best_score_),'\n')
print ('meilleurs paramètres', gd_sr.best_params_,'\n')
print ('meilleur estimateur',gd_sr.best_estimator_,'\n')
```

```
meilleur score 0.952

meilleurs paramètres {'metric': 'minkowski', 'n_neighbors': 5}

meilleur estimateur KNeighborsClassifier()
```

Avec SVM

```
from sklearn.svm import SVC
grid_param = {
    'C': [0.001, 0.01, 0.1, 1, 10],
    'gamma' : [0.001, 0.01, 0.1, 1],
    'kernel': ['linear','rbf']}

gd_sr = GridSearchCV(estimator=SVC(),
                     param_grid=grid_param,
                     scoring='accuracy',
                     cv=5,
                     n_jobs=1,
                    return_train_score=True)

gd_sr.fit(X_train, y_train)

print ('meilleur score %0.3f'%(gd_sr.best_score_),'\n')
print ('meilleurs paramètres', gd_sr.best_params_,'\n')
print ('meilleur estimateur',gd_sr.best_estimator_,'\n')
```

```
meilleur score 0.990

meilleurs paramètres {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}

meilleur estimateur SVC(C=10, gamma=0.1)
```

Pour voir l'ensemble des évaluations effectuées par GridSearchCV :

```
# convertion en DataFrame
results = pd.DataFrame(gd_sr.cv_results_)
# Affichage des 5 premières lignes
display(results.head())
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	pa
0	0.001432	0.000589	0.000573	0.000167	0.001	
1	0.001066	0.000014	0.000475	0.000009	0.001	
2	0.000837	0.000023	0.000403	0.000016	0.001	
3	0.001029	0.000006	0.000470	0.000011	0.001	
4	0.000900	0.000105	0.000409	0.000011	0.001	

L'avantage de GridSearchCV est qu'il va parcourir toutes les conditions et retourner celles qui sont les meilleures pour la ou les mesures de scoring recherchée (dans notre cas nous avons privilégié l'accuracy). Cela est très pratique mais est malheureusement impossible dans certains cas car beaucoup trop long à mettre en place. Une solution possible est d'utiliser *RandomizedSearchCV* qui parcourt de manière aléatoire l'espace de recherche. Il suffit dans ce cas de spécifier des tirages aléatoires pour les valeurs possibles des paramètres et de préciser le nombre d'itérations voulues. Le second usage de *RandomizedSearchCV* est, lorsque l'on n'a pas une très bonne idée de ce que cela peut donner ou des paramètres à utiliser de faire appel à lui pour avoir des valeurs qui peuvent être significatives et de faire suivre à partir de ces valeurs une recherche via *GridSearchCV*.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

rand_param = {
    'max_depth': randint(1, 20),
    'criterion': ['gini', 'entropy'],
    'min_samples_leaf': randint(1, 20)
}


rand_sr = RandomizedSearchCV(estimator=DecisionTreeClassifier(),
                             param_distributions = rand_param,
                             random_state=1,
                             n_iter=20,
                             cv=3,
                             n_jobs=-1,
                             scoring='accuracy',
                             return_train_score=True)

rand_sr.fit(X_train, y_train)


print ('meilleur score %0.3f'%(rand_sr.best_score_),'\n')
print ('meilleurs paramètres', rand_sr.best_params_,'\n')
print ('meilleur estimateur',rand_sr.best_estimator_,'\n')

# convertion en DataFrame
results = pd.DataFrame(rand_sr.cv_results_)
# Affichage des 5 premières lignes
display(results.head())
```

```
meilleur score 0.962
meilleurs paramètres {'criterion': 'entropy', 'max_depth': 12, 'min_samples
meilleur estimateur DecisionTreeClassifier(criterion='entropy', max_depth=1
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_crite
0	0.001110	0.000175	0.000560	0.000024	en
1	0.000977	0.000077	0.000523	0.000023	
2	0.001309	0.000117	0.000583	0.000007	en
3	0.000978	0.000170	0.000782	0.000497	
4	0.001030	0.000154	0.000545	0.000192	en

GridsearchCV et plusieurs classifieurs

Il est tout à fait possible d'utiliser GridsearchCV avec plusieurs classifieurs. Il suffit pour cela d'initaliser les classifieurs dans un dictionnaire et faire de même pour les paramètres.

```
classifiers = {
    'KNeighborsClassifier': KNeighborsClassifier(),
    'DecisionTreeClassifier': DecisionTreeClassifier(),
    'SVC': SVC()
}

params = {'KNeighborsClassifier' : [{'n_neighbors': list(range(1,15))},
    {'metric': ['minkowski','euclidean','manhattan']}],
           'DecisionTreeClassifier': [{'max_depth': [1,2,3,4,5,6,7,8,9,10]},
    {'criterion': ['gini', 'entropy']},
    {'min_samples_leaf': [1,2,3,4,5,6,7,8,9,10]}],
       'SVC':[{'C': [0.001, 0.01, 0.1, 1, 10],
    'gamma' : [0.001, 0.01, 0.1, 1],
    'kernel': ['linear','rbf']}]  }
```

```
class Result_Parameters:
     def __init__(self,name, score, parameters):
         self.name = name
         self.score = score
         self.parameters = parameters
     def __repr__(self):
         return repr((self.name, self.score, self.parameters))


results = []
for key,value in classifiers.items():
    gd_sr = GridSearchCV(estimator=value,
                     param_grid=params[key],
                     scoring='accuracy',
                     cv=5,
                     n_jobs=1)
    gd_sr.fit(X_train, y_train)
    result=Result_Parameters(key,gd_sr.best_score_,gd_sr.best_estimator_)
    results.append(result)



results=sorted(results, key=lambda result: result.score, reverse=True)

print ('Le meilleur resultat : \n')
print ('Classifier : ',results[0].name,
       ' score %0.3f' %results[0].score,
       ' avec ',results[0].parameters,'\n')

print ('Tous les résultats : \n')
for result in results:
    print ('Classifier : ',result.name,
```

```
          ' score %0.3f' %result.score,
          ' avec ',result.parameters,'\n')
```

```
Le meilleur resultat :

Classifier :  SVC  score 0.990  avec  SVC(C=10, gamma=0.1)

Tous les résultats :

Classifier :  SVC  score 0.990  avec  SVC(C=10, gamma=0.1)

Classifier :  DecisionTreeClassifier  score 0.971  avec  DecisionTreeClassi

Classifier :  KNeighborsClassifier  score 0.952  avec  KNeighborsClassifier
```

Les pipelines

Il peut arriver que différentes combinaisons de pré-traitements puissent être utilisées. Par exemple il est possible d'utiliser du changement d'échelle, du PCA (projection sur un nombre différent de dimensions), de faire du remplacement de valeurs manquantes ...

L'objectif du pipeline est de pouvoir regrouper l'ensemble de ces prétraitements et de pouvoir les faire suivre par le classifier. Le principe consiste à d'abord mettre la chaîne de pré-traitement, d'ensuite mettre le classifier et d'utiliser directement le pipeline.

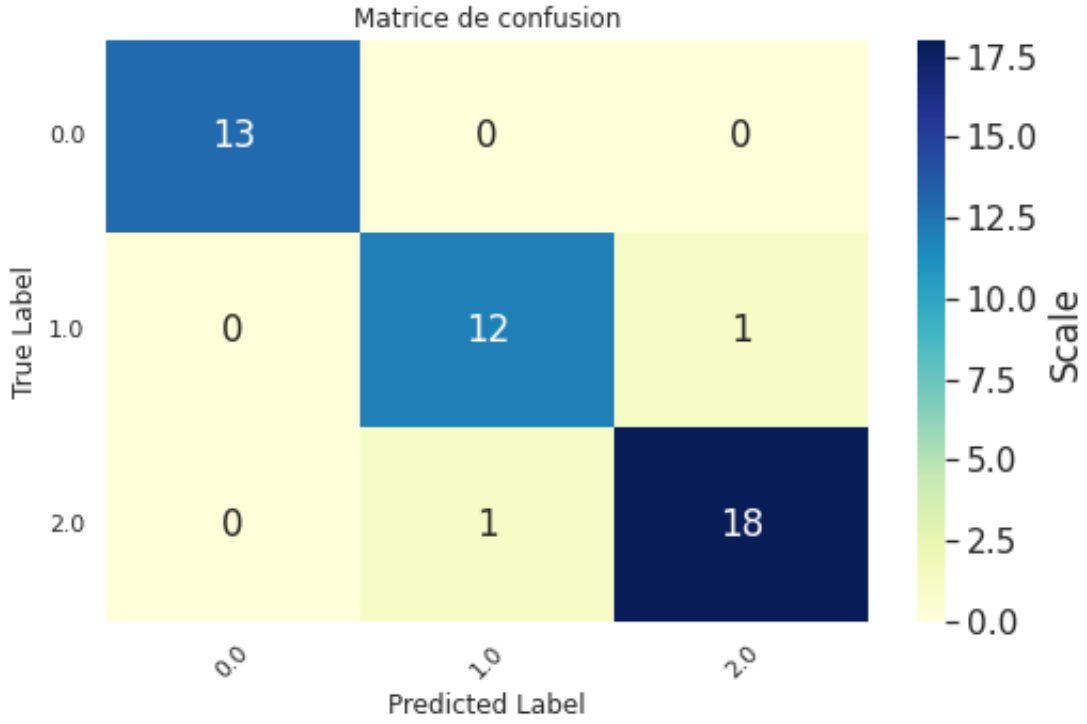
Attention : les pipelines sont très importants lorsque l'on sauvegarde un modèle. En effet comme ils prennent en compte les pré-traitements tout est sauvegardé. Cela veut dire que dans le cas de nouvelles données à évaluer avec un modèle lors de la prédiction les données seront automatiquement transformées. (Voir partie utiliser de nouvelles données plus bas).

L'exemple suivant illustre un pipeline où un standard scaling est réalisé puis un PCA et enfin un DecisionTree est appliqué.

```
from sklearn.preprocessing import LabelEncoder


url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
names = ['SepalLengthCm', 'SepalWidthCm',
         'PetalLengthCm', 'PetalWidthCm', 'Species']
df_iris = pd.read_csv(url, names=names)
```

```
#transformation de Species via LabelEncoder
class_label_encoder = LabelEncoder()
df_iris['Species']=class_label_encoder.fit_transform(df_iris['Species'].values)



array = df_iris.values
X = array[:,0:4]
y = array[:,4]



print ('Création du pipeline \n')
pipeline = Pipeline([('scl', StandardScaler()),
                     #('pca', PCA(n_components=2)),
                     ('clf', DecisionTreeClassifier(random_state=42))])

trainsize=0.7 # 70% pour le jeu d'apprentissage, il reste 30% du jeu de données

testsize= 0.3
seed=30
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=trainsize,
                                               random_state=seed,
                                               test_size=testsize)

pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
MyshowAllScores(y_test,y_pred)
```

```
Création du pipeline

Accuracy : 0.956
Classification Report
              precision    recall  f1-score   support

         0.0    1.00000   1.00000   1.00000        13
         1.0    0.92308   0.92308   0.92308        13
         2.0    0.94737   0.94737   0.94737        19

    accuracy                        0.95556        45
   macro avg    0.95682   0.95682   0.95682        45
weighted avg    0.95556   0.95556   0.95556        45
```



Il est possible d'utiliser GridSearchCV pour chercher les meilleures valeurs dans un pré-traitement.

```
from sklearn.model_selection import GridSearchCV
print ('Création du pipeline \n')
pipeline = Pipeline([('pca', PCA()),
                     ('clf', DecisionTreeClassifier(random_state=42))])

grid_param = {
    'pca__n_components': [2,3]
}


gd_sr = GridSearchCV(pipeline,
                      param_grid=grid_param,
                      scoring='accuracy',
                      cv=5,
                      n_jobs=1,
                     return_train_score=True)

gd_sr.fit(X_train, y_train)

print ('meilleur score %0.3f'%(gd_sr.best_score_),'\n')
print ('meilleurs paramètres', gd_sr.best_params_,'\n')
print ('meilleur estimateur',gd_sr.best_estimator_,'\n')
```

```
Création du pipeline

meilleur score 0.933

meilleurs paramètres {'pca__n_components': 3}

meilleur estimateur Pipeline(steps=[('pca', PCA(n_components=3)),
                 ('clf', DecisionTreeClassifier(random_state=42))])
```

Ou bien de faire les deux en même temps.

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['SepalLengthCm', 'SepalWidthCm',
         'PetalLengthCm', 'PetalWidthCm', 'Species']
df_iris = pd.read_csv(url, names=names)

#transformation de Species en float pour StantardScaler
class_label_encoder = LabelEncoder()
df_iris['Species']=class_label_encoder.fit_transform(df_iris['Species'].values)



array = df_iris.values
X = array[:,0:4]
```

```
y = array[:,4]

pipeline = Pipeline([('pca', PCA()),
                     ('clf', DecisionTreeClassifier())])



grid_param = [{'pca__n_components': [2,3]},
                {'clf': [DecisionTreeClassifier()],
                 'clf__max_depth': [1,2,3,4,5,6,7,8,9,10],
                 'clf__criterion': ['gini', 'entropy'],
                 'clf__min_samples_leaf': [1,2,3,4,5,6,7,8,9,10]
                }]



gd_sr = GridSearchCV(estimator=pipeline,
                     param_grid=grid_param,
                     scoring='accuracy',
                     cv=5,
                     n_jobs=-1,
                    return_train_score=True)

gd_sr.fit(X_train, y_train)
print ('meilleur score %0.3f'%(gd_sr.best_score_),'\n')
print ('meilleurs paramètres', gd_sr.best_params_,'\n')
print ('meilleur estimateur',gd_sr.best_estimator_,'\n')
```

```
meilleur score 0.952

meilleurs paramètres {'clf': DecisionTreeClassifier(criterion='entropy', ma

meilleur estimateur Pipeline(steps=[('pca', PCA()),
                 ('clf',
                  DecisionTreeClassifier(criterion='entropy', max_depth=4,
                                         min_samples_leaf=2))])
```

Sauvegarder le modèle appris

Une fois un modèle appris il est possible de le sauvegarder pour pouvoir lui appliquer d'autres données à prédire. Il existe différentes librairies comme pickle ou joblib.
Dans ce notebook nous utilisons pickle qui est la librairie Python standard pour sérialiser-désérialiser des objets.

```
#preparation des données
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['SepalLengthCm', 'SepalWidthCm',
         'PetalLengthCm', 'PetalWidthCm', 'Species']
df_iris = pd.read_csv(url, names=names)
array = df_iris.values
X = array[:,0:4]
y = array[:,4]

trainsize=0.7 # 70% pour le jeu d'apprentissage, il reste 30% du jeu de données

testsize= 0.3
seed=30
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=trainsize,
                                               random_state=seed,
                                               test_size=testsize)

clf = GaussianNB()
clf.fit(X_train, y_train)
```

```
GaussianNB()
```

pickle

Pour sérialiser et sauvegarder le modèle appris :

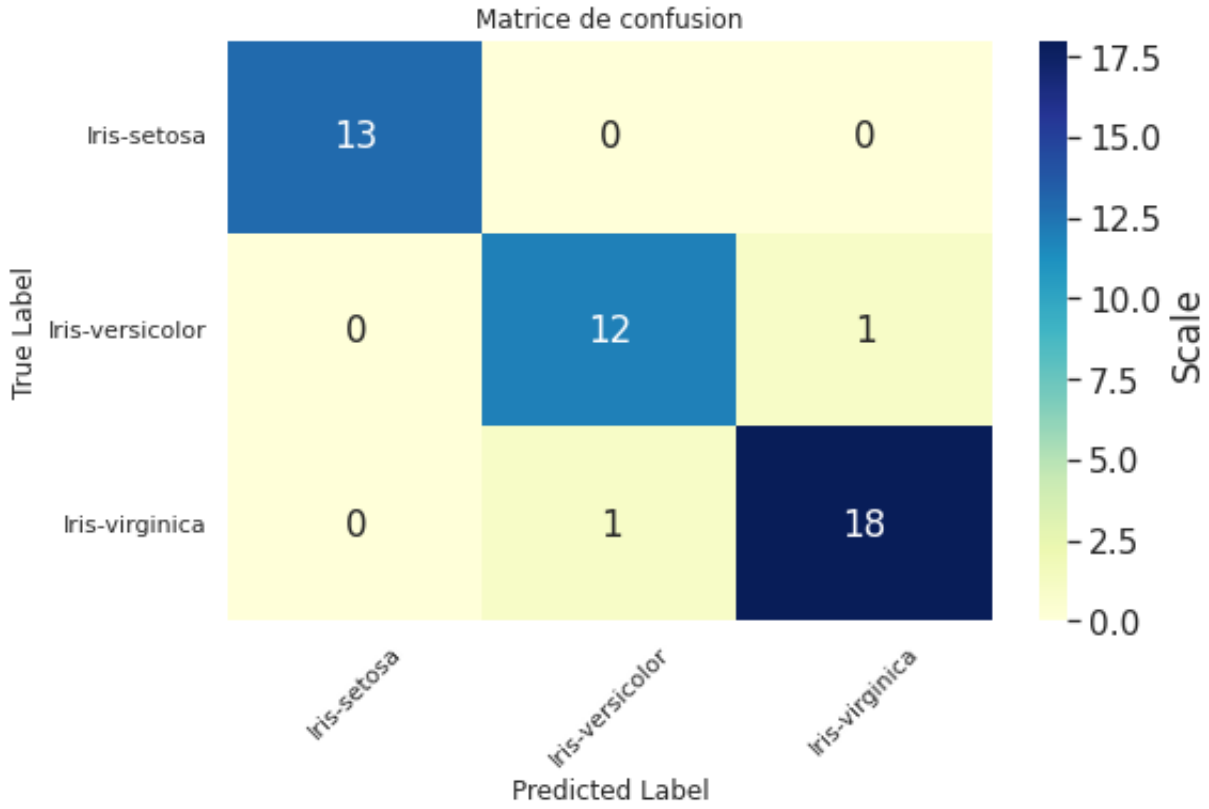
```
import pickle
filename = 'pkl_modelNB.sav'
pickle.dump(clf, open(filename, 'wb'))
```

Pour utiliser le modèle sauvegardé :

```
filename = 'pkl_modelNB.sav'
clf_loaded = pickle.load(open(filename, 'rb'))
print ('Modèle chargé',clf_loaded,'\n')

# application sur les données de test obtenues précédemment
y_pred = clf_loaded.predict(X_test)
MyshowAllScores(y_test,y_pred)
```

```
result = clf_loaded.predict([[ 5.0,  3.6,  1.4,  0.2]])
print ('\nLa prédiction du modèle pour [ 5.0,  3.6,  1.4,  0.2] est',
       result)
```

```
Modèle chargé GaussianNB()

Accuracy : 0.956
Classification Report
                 precision    recall  f1-score   support

    Iris-setosa    1.00000   1.00000   1.00000        13
Iris-versicolor    0.92308   0.92308   0.92308        13
 Iris-virginica    0.94737   0.94737   0.94737        19

       accuracy                        0.95556        45
      macro avg    0.95682   0.95682   0.95682        45
   weighted avg    0.95556   0.95556   0.95556        45
```




```
La prédiction du modèle pour [ 5.0,  3.6,  1.4,  0.2] est ['Iris-setosa']
```

Utiliser de nouvelles données

A partir d'un modèle sauvegardé, il est donc possible d'appliquer la fonction predict pour connaître la prédiction du modèle.

Dans le cas de nouvelles données il faut faire attention car des pré-traitements ont sans doute été effectués avec les données initiales (changement d'échelle, normalisation, etc) et une matrice a été obtenue pour apprendre un modèle.

Il est impératif que les nouvelles données suivent le même traitement. Nous présentons par la suite un exemple d'utilisation à l'aide des données IRIS. Cette fois-ci nous utilisons iris qui est disponible directement dans scikit learn.

Lecture de la base iris et utilisation de SVM comme classifieur

```
from sklearn import datasets
clf = SVC(gamma='scale')
iris = datasets.load_iris()
```

Dans un premier temps nous ajoutons du bruit dans la base iris en mettant pour trois colonnes des valeurs supérieures à 1000. L'objectif ici est de montrer que les valeurs sont trop différentes pour obtenir de bons résultats de classification. SVM est très sensible à la standardisation.

```
for i in range (len(iris.data)):
    for j in range (0,2):
        val = iris.data[i][j]
        value = val*1000
        iris.data[i][j]=value
```

Définition de X et de y

```
X = iris.data
y = iris.target
```

```
trainsize=0.7 # 70% pour le jeu d'apprentissage, il reste 30% du jeu de données

testsize= 0.3
seed=30
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=trainsize,
                                               random_state=seed,
                                               test_size=testsize)
```

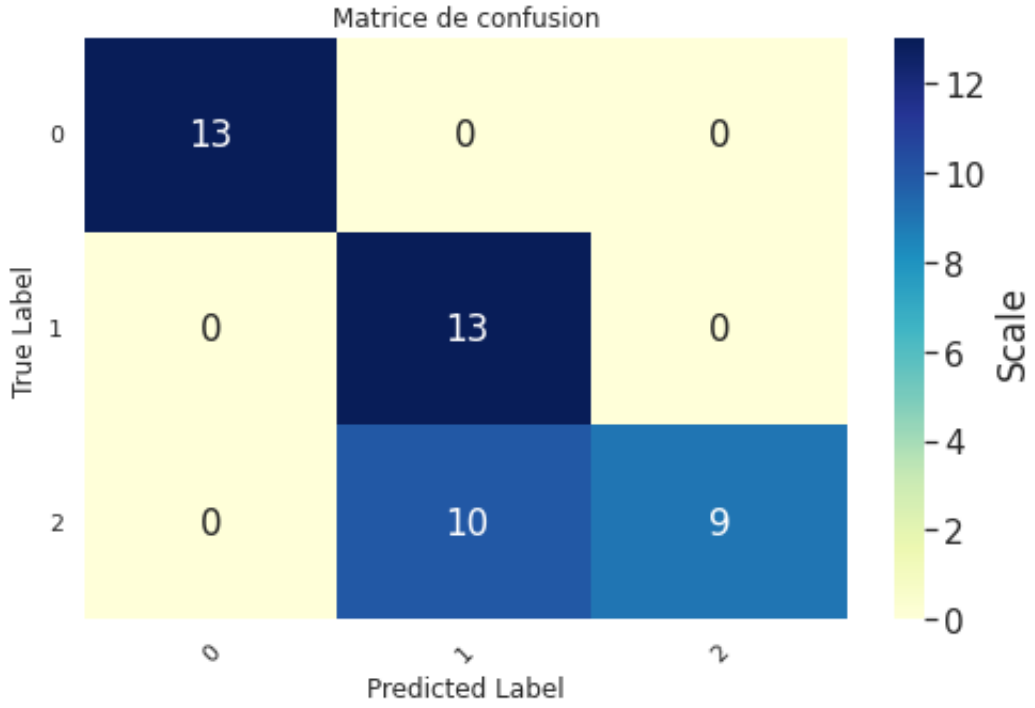
Fonction de comptage pour voir combien d'instances sont mal classés après la classification.

```
def cpt_mal_classes(y_test,result):
    nb=0
    for i in range(len(y_test)):
        if y_test[i] != result [i]:
            nb=nb+1
    return nb

def print_nb_classes (taille,nb):
    print ("Taille des données à tester",
       taille,
       " - mal classées : ",nb)
```

Première classification avec SVM. L'objectif ici est de montrer que SVM est très sensible à la standardisation. Il suffit de regarder l'accuracy pour s'en convaincre.

```
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
nb=cpt_mal_classes(y_test,y_pred)
taille=len(y_test)
print_nb_classes (len(y_test),nb)
MyshowAllScores(y_test,y_pred)
```

```
Taille des données à tester 45  - mal classées :  10
Accuracy : 0.778
Classification Report
              precision    recall  f1-score   support

           0    1.00000   1.00000   1.00000        13
           1    0.56522   1.00000   0.72222        13
           2    1.00000   0.47368   0.64286        19

    accuracy                        0.77778        45
   macro avg    0.85507   0.82456   0.78836        45
weighted avg    0.87440   0.77778   0.76896        45
```



Par la suite nous allons donc utiliser MinMaxScaler () pour standardiser les données.

Nous sauvegardons également le jeu de test (X_save=X_test.copy()). L'objectif est de sauvegarder le modèle pour évaluer en le rechargeant si le nombre d'instances bien classées est le même que celui qui a été prédit lors de l'apprentissage.

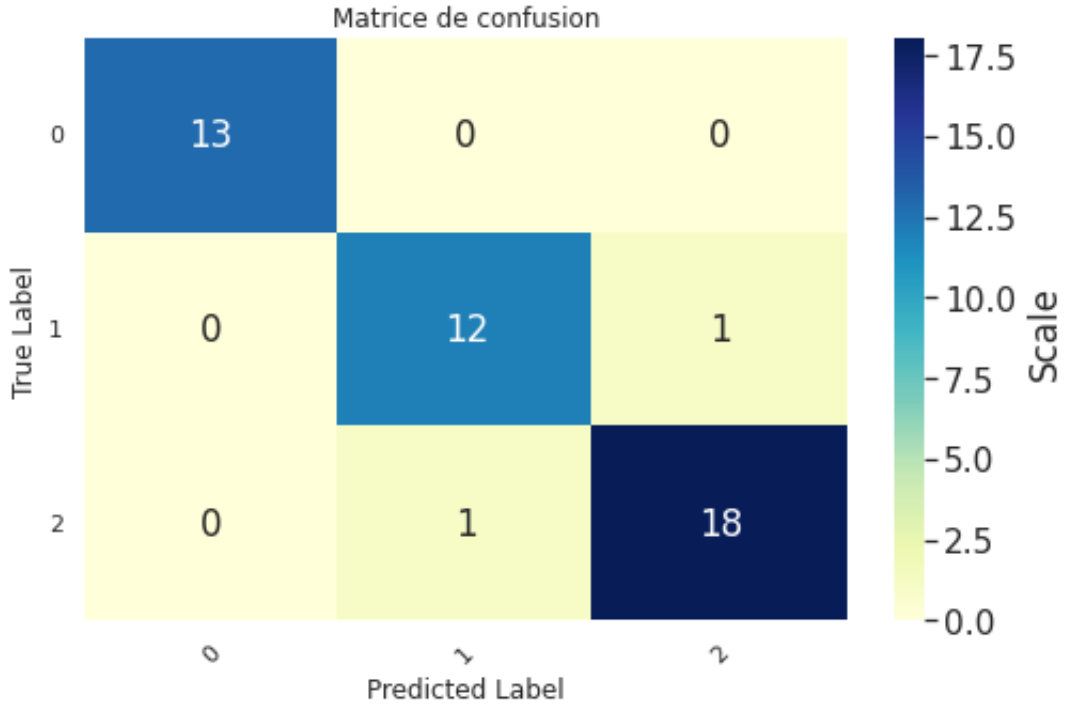
```
trainsize=0.7 # 70% pour le jeu d'apprentissage, il reste 30% du jeu de données

testsize= 0.3
seed=30
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=trainsize,
                                               random_state=seed,
                                               test_size=testsize)
```

La standardisation a été faite car les valeurs du jeu de données ne permettait pas de pouvoir utiliser le classifieur directement. En sauvegardant le jeu avant la standardisation nous simulons le fait que nous arrivons avec un nouveau jeu de données d'iris.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_save=X_test.copy()
X_test = scaler.fit_transform(X_test)
```

```
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
nb=cpt_mal_classes(y_test,y_pred)
taille=len(y_test)
print_nb_classes (len(y_test),nb)
MyshowAllScores(y_test,y_pred)
```

```
Taille des données à tester 45  - mal classées :  2
Accuracy : 0.956
Classification Report
              precision    recall  f1-score   support

           0    1.00000   1.00000   1.00000        13
           1    0.92308   0.92308   0.92308        13
           2    0.94737   0.94737   0.94737        19

    accuracy                        0.95556        45
   macro avg    0.95682   0.95682   0.95682        45
weighted avg    0.95556   0.95556   0.95556        45
```



Sauvegarde du modèle appris

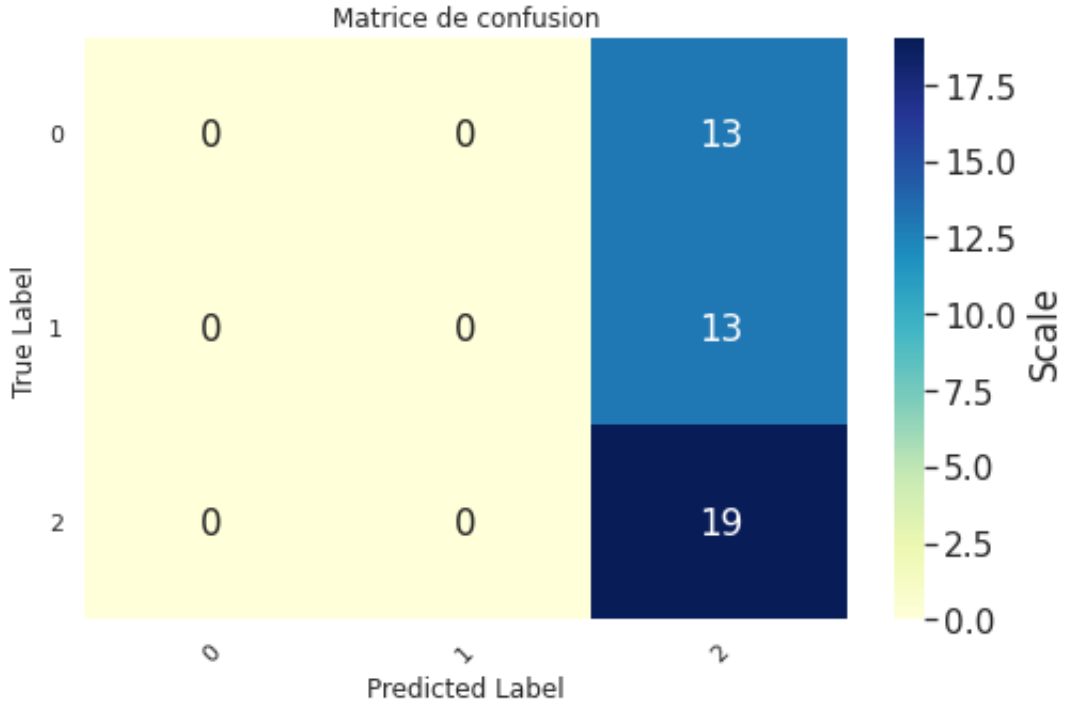
```
print("\nSauvegarde du modèle")
filename = 'firstmodel.pkl'
pickle.dump(clf, open(filename, 'wb'))
```

```
Sauvegarde du modèle
```

Ouverture du modèle pour le tester. Ici nous reprenons le jeu de test qui n'a pas eu l'étape de standardisation comme nouvelles données, i.e. nous avons de nouveaux IRIS. Si le modèle est bien appris le nombre d'objets mal classés devrait être le même.

```
print ("Chargement du modèle \n")
filename = 'firstmodel.pkl'
clf_loaded = pickle.load(open(filename, 'rb'))

y_pred=clf_loaded.predict(X_save)

nb=cpt_mal_classes(y_test,y_pred)
taille=len(y_test)
print_nb_classes (len(y_test),nb)
MyshowAllScores(y_test,y_pred)
```

```
Chargement du modèle

Taille des données à tester 45  - mal classées :  26
Accuracy : 0.422
Classification Report
              precision    recall  f1-score   support

           0    0.00000   0.00000   0.00000        13
           1    0.00000   0.00000   0.00000        13
           2    0.42222   1.00000   0.59375        19

    accuracy                        0.42222        45
   macro avg    0.14074   0.33333   0.19792        45
weighted avg    0.17827   0.42222   0.25069        45

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1
  _warn_prf(average, modifier, msg_start, len(result))
```

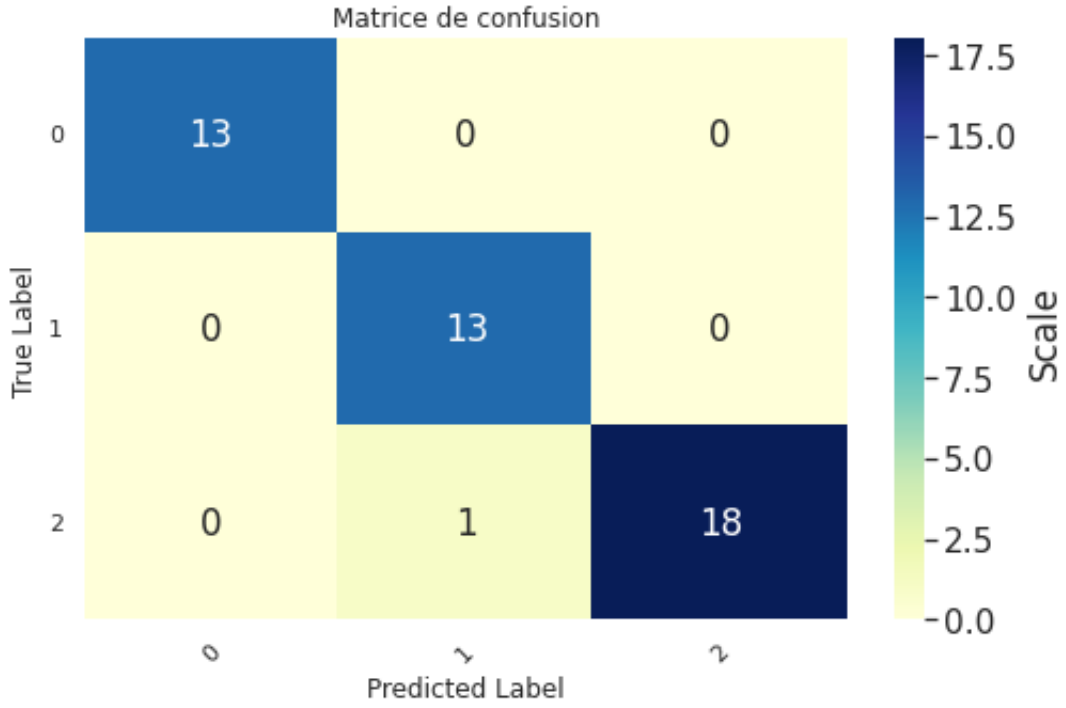


Nous pouvons constater qu'il y a presque plus d'objets mal classés. Comme nous avons fait une standardisation dans les étapes précédentes celle là n'a pas pu être faite pour les nouvelles données. La standardisation doit donc être faite pour les nouvelles données mais elle nécessite de pouvoir récupérer les anciennes valeurs pour tout standardiser.

Les pipelines sont donc utiles pour pouvoir tout sauvegarder (l'étape de standardisation et l'application du modèle).

```
pipeline = Pipeline([('vect', MinMaxScaler()),
                ('clf', SVC(gamma='scale')),
               ])




X=iris.data
y=iris.target

trainsize=0.7 # 70% pour le jeu d'apprentissage, il reste 30% du jeu de données

testsize= 0.3
seed=30
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=trainsize,
                                               random_state=seed,
                                               test_size=testsize)

X_save=X_test.copy()

pipeline.fit(X_train, y_train)

y_pred = pipeline.predict(X_test)

nb=cpt_mal_classes(y_test,y_pred)
taille=len(y_test)
print_nb_classes (len(y_test),nb)
MyshowAllScores(y_test,y_pred)



print("\nSauvegarde du pipeline ")
filename = 'avecscaler.pkl'
pickle.dump(pipeline, open(filename, 'wb'))
```

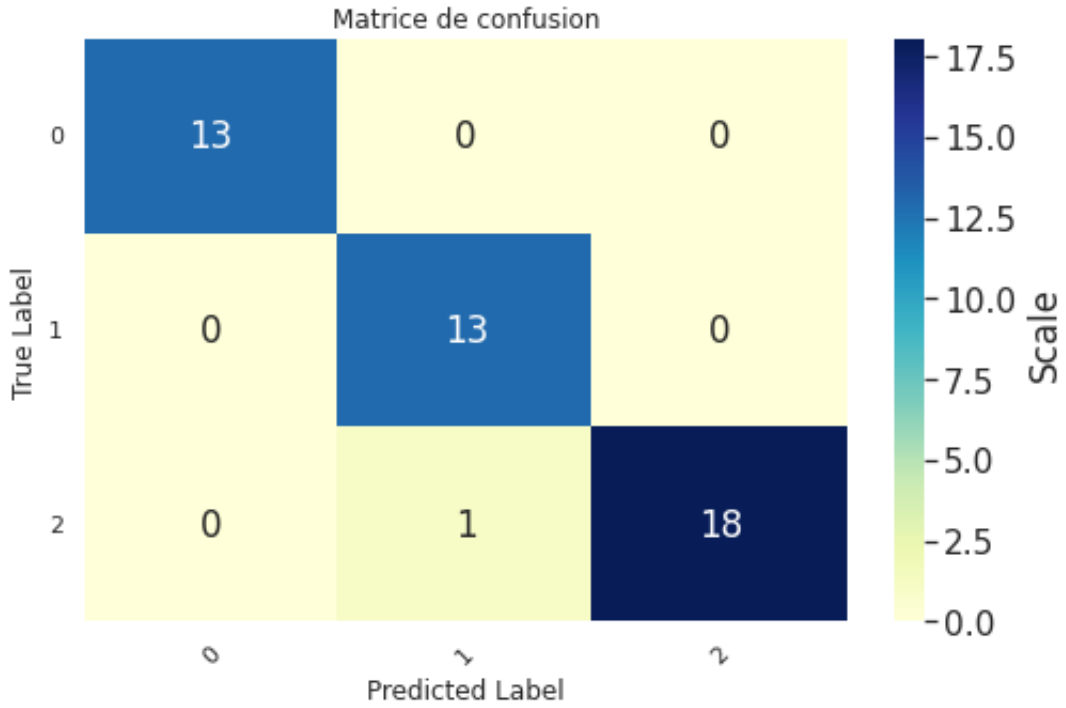
```
Taille des données à tester 45  - mal classées :  1
Accuracy : 0.978
Classification Report
              precision    recall  f1-score   support

           0    1.00000   1.00000   1.00000        13
           1    0.92857   1.00000   0.96296        13
           2    1.00000   0.94737   0.97297        19

    accuracy                        0.97778        45
   macro avg    0.97619   0.98246   0.97865        45
weighted avg    0.97937   0.97778   0.97789        45
```



Sauvegarde du pipeline

```
print ("Chargement du modèle \n")
filename = 'avecscaler.pkl'
clf_loaded = pickle.load(open(filename, 'rb'))

y_pred=clf_loaded.predict(X_save)
nb=cpt_mal_classes(y_test,y_pred)
taille=len(y_test)
print_nb_classes (len(y_test),nb)

MyshowAllScores(y_test,y_pred)
```

```
Chargement du modèle

Taille des données à tester 45  - mal classées :  1
Accuracy : 0.978
Classification Report
              precision    recall  f1-score   support

           0    1.00000   1.00000   1.00000        13
           1    0.92857   1.00000   0.96296        13
           2    1.00000   0.94737   0.97297        19

    accuracy                        0.97778        45
   macro avg    0.97619   0.98246   0.97865        45
weighted avg    0.97937   0.97778   0.97789        45
```



Attention : si vous testez ce modèle sauvegardé sous ce notebook il fonctionnera. Si, par exemple, vous ouvrez un fichier à part et lancer le code précédent cela fonctionnera, sans doute, encore* mais .....
Imaginer que l'on réalise un traitement particulier, via une fonction f () dans nos données, qui soit appelée dans le pipeline :
pipeline = Pipeline([('vect', f()),
Si maintenant vous utilisez dans votre autre notebook ou dans un programme extérieur votre modèle. Alors qu'il fonctionnait bien dans votre notebook précédent, vous verrez afficher un message d'erreur du type : *Can't get attribute 'f'*
En fait, pickle sauvegarde une référence à la fonction f. Dans le premier cas, votre notebook initial, cette fonction a été définie et donc pickle va pouvoir exécuter le code de cette fonction. Dans l'autre notebook, il va rechercher le code à exécuter et donc ... il ne le trouvera pas.

Il est possible d'utiliser *Drill* qui se comporte exactement de la même manière que *pickle* mais qui lui sauvegarde aussi la fonction. Alors la solution est-elle d'utiliser *Drill* ? C'est tout à fait déconseillé. Bien sûr pour votre modèle avec votre fonction il n'y aura pas de problèmes mais souvent on va utiliser des modèles définis par quelqu'un d'autre .... et pour des raisons de sécurités évidentes (on ne sait pas ce que fait la fonction cachée !!!) ... on évite cette approche. La solution consiste tout simplement à sauvegarder le modèle et les fonctions nécessaires pour son fonctionnement et de faire un import de celles-ci là où le modèle est utilisé.

Une petite mise en pratique

Jusqu'à présent vous n'avez que vu (ou revu) les principes généraux de la classification. Avant de s'attaquer aux données textuelles, voici un petit exercice pour mettre en oeuvre les concepts introduits dans ce notebook. Les IRIS est connu, très connu ... alors changeons de domaine pour ... les pingouins

De nombreux jeux de données ont été proposés en alternative à Iris qui est très (trop ?) utilisé. Le jeu de données que nous allons utiliser possède des caractéristiques assez similaires à IRIS mais concerne les espèces de pingouins. Il est disponible ici : https://allisonhorst.github.io/palmerpenguins/

Il contient à l'origine 17 caractéristiques différentes ('studyName', 'Sample Number', 'Species', 'Region', 'Island', 'Stage','Individual ID', 'Clutch Completion', 'Date Egg', 'Culmen Length (mm)','Culmen Depth (mm)', 'Flipper Length (mm)', 'Body Mass (g)', 'Sex','Delta 15 N (o/oo)', 'Delta 13 C (o/oo)', 'Comments') et concerne 3 espèces de pingouins différentes ('Adelie Penguin (Pygoscelis adeliae)', 'Gentoo penguin (Pygoscelis papua)', 'Chinstrap penguin (Pygoscelis antarctica)'). L'un des objectifs est à partir des différentes caractéristiques de prédire l'espèce de pingouins.
Le jeu de données d'origine contient de nombreuses valeurs manquantes, des données à la fois catégorielles, numériques, etc. Aussi nous en proposons une version nettoyée que vous pouvez récupérer :

```
!wget https://www.lirmm.fr/~poncelet/Ressources/penguins.csv
```

```
--2022-01-12 16:11:12--  https://www.lirmm.fr/~poncelet/Ressources/penguins
Resolving www.lirmm.fr (www.lirmm.fr)... 193.49.104.251
Connecting to www.lirmm.fr (www.lirmm.fr)|193.49.104.251|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 27042 (26K) [text/csv]
Saving to: ‘penguins.csv.1’

penguins.csv.1      100%[===================>]  26.41K  --.-KB/s    in 0.1s

2022-01-12 16:11:13 (271 KB/s) - ‘penguins.csv.1’ saved [27042/27042]
```

Exercice : la cellule suivante permet de charger le dossier et d'avoir quelques informations sur les données manipulées. En outre, il propose d'appliquer un standardScaler sur les données prédictives. L'objectif à présent est de proposer un classifier qui soit capable de prédire le mieux possible l'espèce de pingouin.

Indication : SVM se comporte bien sur ce jeu de données.

```
# lecture du fichier
df_penguins=pd.read_csv("penguins.csv", encoding='utf8')

# les premières lignes
print (df_penguins.head())

# les différentes colonnes
print (df_penguins.columns)

# le nombre d'enregistrements
print (df_penguins.shape)

# la répartition du nombre de pingouins
print ("répartition des espèces de pingouins :\n",df_penguins['Species'].value_c

# selection des variables prédictives et à prédire
array = df_penguins.values
X = array[:,0:6]
y = array[:,6]

# attention étant donné la différence entre les valeurs il est nécessaire de nor
standardscaler = StandardScaler()
X_standardscale = standardscaler.fit_transform(X)
X=X_standardscale
```

```
   Culmen Length (mm)  ...                                Species
0                39.5  ...  Adelie Penguin (Pygoscelis adeliae)
1                40.3  ...  Adelie Penguin (Pygoscelis adeliae)
2                36.7  ...  Adelie Penguin (Pygoscelis adeliae)
3                39.3  ...  Adelie Penguin (Pygoscelis adeliae)
4                38.9  ...  Adelie Penguin (Pygoscelis adeliae)

[5 rows x 7 columns]
Index(['Culmen Length (mm)', 'Culmen Depth (mm)', 'Flipper Length (mm)',
       'Body Mass (g)', 'Delta 15 N (o/oo)', 'Delta 13 C (o/oo)', 'Species'
      dtype='object')
(330, 7)
répartition des espèces de pingouins :
 Adelie Penguin (Pygoscelis adeliae)          141
Gentoo penguin (Pygoscelis papua)            122
Chinstrap penguin (Pygoscelis antarctica)     67
Name: Species, dtype: int64
```

Solution :

```
# il suffit d'appliquer les approches précédentes
```

```
# Il suffit d'appliquer les approches precedentes.
# D'abord on peut comparer des algorithmes de classification
# puis après sélectionner le meilleur et rechercher ses hyperparamètres.


seed = 7
scoring = 'accuracy'
models = []
models.append(('LR', LogisticRegression(solver='lbfgs')))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto')))

results = []
names = []
for name,model in models:
    kfold = KFold(n_splits=10, random_state=seed, shuffle=True)
    start_time = time.time()
    cv_results = cross_val_score(model, X, y, cv=kfold, scoring=scoring)
    #pour avoir les paramètres utilisés dans le modèle enlever commentaire ligne
    #print (model.get_params())
    print ("Time pour",name," %0.5f"%(time.time() - start_time),'s')
    results.append(cv_results)
    names.append(name)
    msg = "%s: %0.3f (%0.3f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)


# SVC a le meilleur score
classifiers = {
    'SVC': SVC()
}

params = {'SVC':[{'C': [0.001, 0.01, 0.1, 1, 10],
    'gamma' : [0.001, 0.01, 0.1, 1],
    'kernel': ['linear','rbf']}]  }




results = []
for key,value in classifiers.items():
    gd_sr = GridSearchCV(estimator=value,
                         param_grid=params[key],
                         scoring='accuracy',
                         cv=5,
                         n_jobs=1)
```

```
    gd_sr.fit(X, y)
    result=Result_Parameters(key,gd_sr.best_score_,gd_sr.best_estimator_)
    results.append(result)



results=sorted(results, key=lambda result: result.score, reverse=True)

print ('Le meilleur resultat :  ')
print ('Classifier : ',results[0].name,
       ' score %0.2f' %results[0].score,
       ' avec ',results[0].parameters,' ')

print ('Tous les résultats : ')
for result in results:
    print ('Classifier : ',result.name,
           ' score %0.2f' %result.score,
           ' avec ',result.parameters,' ')
```

```
Time pour LR  0.10782 s
LR: 0.997 (0.009)
Time pour KNN  0.04040 s
KNN: 1.000 (0.000)
Time pour CART  0.02252 s
CART: 0.961 (0.036)
Time pour NB  0.02559 s
NB: 0.988 (0.015)
Time pour SVM  0.03341 s
SVM: 0.994 (0.012)
Le meilleur resultat :
Classifier :  SVC  score 1.00  avec  SVC(C=1, gamma=0.001, kernel='linear')
Tous les résultats :
Classifier :  SVC  score 1.00  avec  SVC(C=1, gamma=0.001, kernel='linear')
```

Maintenant que nous avons vu comment mettre en place un classifier et l'évaluer, nous abordons les données textuelles .... dans le notebook "**Ingénierie des données textuelles**"

✓ 1 s terminée à 17:11