

Premières classifications

Le but de ce notebook est de faire des premières classifications. Pour cela nous utilisons le jeu de données IRIS qui est très connu dans la communauté.

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 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.

In [1]:

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

Le jeu de données IRIS

Nous considérons par la suite le jeu de données des IRIS

In [2]:

```

1 import pandas as pd
2 import numpy as np
3 #https://archive.ics.uci.edu/ml/machine-learning-databases/iris/
4 url="https://archive.ics.uci.edu/ml/machine-learning-databases/i
5 names = ['SepalLengthCm', 'SepalWidthCm',
6          'PetalLengthCm', 'PetalWidthCm',
7          'Species']
8
9 df = pd.read_csv(url, names=names)
10 # 5 premières lignes du fichier
11 display(df.head())

```

	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

▼ Toute première classification

La classification supervisée considère les données sous la forme (X, Y) où X correspond aux variables prédictives et Y 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.

In [3]:

```

1 array = df.values #nécessité de convertir le dataframe en numpy
2 #X matrice - utilisation du X majuscule
3 X = array[:,0:4]
4 #y vecteur - utilisation du y minuscule
5 y = array[:,4]
6

```

Dans scikit-learn, pour apprendre un modèle, un estimateur est créé en appelant sa méthode `fit(X, y)`.

Dans l'exemple qui suit nous utilisons un classifieur naïve Bayes (https://scikit-learn.org/stable/modules/naive_bayes.html) (https://scikit-learn.org/stable/modules/naive_bayes.html)).

```
In [4]: 1 import sklearn
2 from sklearn.naive_bayes import GaussianNB
3
4 clf = GaussianNB()
5
6 clf.fit(X, y)
7 # la ligne affichée dans le Out indique les hyperparamètres
8 #du classifieur s'ils existent
```

```
Out[4]: GaussianNB(priors=None, var_smoothing=1e-09)
```

La ligne affichée dans le Out indique les hyperparamètres du classifieur s'ils existent. Il est également possible de les obtenir à l'aide de :

```
In [5]: 1 clf.get_params()
```

```
Out[5]: {'priors': None, 'var_smoothing': 1e-09}
```

Il est alors possible de prédire une valeur non lue à l'aide de la méthode `predict`. 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.

```
In [6]: 1 #Prediction du resultat
2 result = clf.predict([[ 5.0, 3.6, 1.4, 0.2]])
3 print ('La prédiction du modèle pour [ 5.0, 3.6, 1.4, 0.2] es
4         result)
5
6
```

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

Test de la prédiction sur les données d'apprentissage.

```
In [7]: 1 result = clf.predict(X)
2 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-setos
a'

'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setos
a'
'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setos
a'
'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setos
a'
'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setos
a'
'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setos
a'
'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setos
a'
'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica' 'Iris-versicol
or'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-virginica' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicol
or'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-virgini
ca'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versico
lor'
'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica' 'Iris-virginic
a'
'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).

```
In [8]: 1 from sklearn.metrics import accuracy_score  
        2  
        3  
        4 print ('accuracy: ',accuracy_score(result, y))  
        5  
        6
```

```
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 :

```

In [9]: 1 y = np.asarray(y)
2 #misclassified = np.where(y_test != clf.predict(X_test))
3 misclassified = np.where(y != clf.predict(X))
4
5
6 print('Les objets mal classés sont :')
7
8 i=0
9 ▼ for i in misclassified:
10     #print('\n', 'o', i, " ", df.iloc[i,:], " ", y[i])
11     print('\n', df.iloc[i,:]['Species'])
12     #print('\n', df.iloc[i,:], " ", y[i])
13     print ('\n')
14
15 ▼ for i in misclassified:
16     print ('\n', i, 'classé en ', clf.predict(X)[i], '\n')
17     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

```

```

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

```

▼ Jeu d'apprentissage et de test

```
In [10]: 1 import pandas as pd
2 #https://archive.ics.uci.edu/ml/machine-learning-databases/iris/
3 url="https://archive.ics.uci.edu/ml/machine-learning-databases/i
4 names = ['SepalLengthCm', 'SepalWidthCm',
5          'PetalLengthCm', 'PetalWidthCm',
6          'Species']
7
8 df = pd.read_csv(url, names=names)
9
10 array = df.values
11 X = array[:,0:4]
12 y = array[:,4]
13
```

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*.

Depuis la version 0.21 il est également nécessaire de préciser la taille du jeu de test *test_size*. De manière classique ce nombre est égal à la différence entre la taille du jeu de données - le *test_size* (1-*test_size*). Via cette fonctionnalité sickit learn permet de faire de l'échantillonnage sur le jeu d'apprentissage.

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

```
In [11]: 1 from sklearn.model_selection import train_test_split
2
3 validation_size=0.3 #30% du jeu de données pour le test
4
5 testsize= 1-validation_size
6 seed=30
7 X_train,X_test,y_train,y_test=train_test_split(X,
8                                                y,
9                                                train_size=valida
10                                                random_state=seed
11                                                test_size=testsiz
12
```

L'apprentissage du modèle se fait comme précédemment

```
In [12]: 1 clf = GaussianNB()
          2 clf.fit(X_train, y_train)
          3
```

```
Out[12]: GaussianNB(priors=None, var_smoothing=1e-09)
```

De même pour la prédiction

```
In [13]: 1 from sklearn.metrics import accuracy_score
          2
          3 result = clf.predict(X_test)
          4 print('\n accuracy :', accuracy_score(result, y_test), '\n')
          5
```

```
accuracy : 0.9428571428571428
```

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 partie, entraîné sur 9, testé sur 1 et cela sera répété pour toutes les combinaisons du découpage.

```
In [14]: 1 from sklearn.model_selection import KFold
          2 from sklearn.model_selection import cross_val_score
          3 seed=7
          4 k_fold = KFold(n_splits=10, shuffle=True, random_state=seed)
```

```
In [15]: 1 clf = GaussianNB()
          2
          3 scoring = 'accuracy'
          4 score = cross_val_score(clf, X, y, cv=k_fold, scoring=scoring)
          5
          6 ▼ print('Les différentes accuracy pour les 10 évaluations sont : \
          7     score, '\n')
          8 ▼ print ('Accuracy moyenne : ', score.mean(),
          9     ' standard deviation', score.std())
         10
```

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

```
Accuracy moyenne : 0.9533333333333334 standard deviation 0.0669991
7080747259
```


L'écart type (standard deviation) 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% mais lorsque qu'un objet de la classe intervient nous ne pouvons pas le retrouver.

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 corrélation et les différentes mesures : precision, rappel et F1-score.

```
In [16]: 1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.metrics import accuracy_score
4 url="https://archive.ics.uci.edu/ml/machine-learning-databases/i
5 names = ['SepalLengthCm', 'SepalWidthCm',
6          'PetalLengthCm', 'PetalWidthCm', 'Species']
7 df = pd.read_csv(url, names=names)
8 array = df.values
9 X = array[:,0:4]
10 y = array[:,4]
11
12 validation_size=0.3 #30% du jeu de données pour le test
13
14 testsize= 1-validation_size
15 seed=30
16 X_train,X_test,y_train,y_test=train_test_split(X, y,
17                                                  train_size=valida
18                                                  random_state=seed
19                                                  test_size=testsize)
20 clf = GaussianNB()
21 clf.fit(X_train, y_train)
22 result = clf.predict(X_test)
23
24 print('\n accuracy:',accuracy_score(result, y_test),'\n')
```

accuracy: 0.9428571428571428

La matrice de confusion permet de connaître les objets bien ou mal classés. Il suffit d'utiliser la fonction `confusion_matrix`.

```
In [17]: 1 from sklearn.metrics import confusion_matrix
2
3 conf = confusion_matrix(y_test, result)
4 print ('\n matrice de confusion \n',conf)
5
```

```
matrice de confusion
[[34  0  0]
 [ 0 33  5]
 [ 0  1 32]]
```

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

```
In [18]: 1 from sklearn.metrics import classification_report
2 conf = confusion_matrix(y_test, result)
3 print ('\n matrice de confusion \n',conf)
4 print ('\n',classification_report(y_test, result))
```

```
matrice de confusion
[[34  0  0]
 [ 0 33  5]
 [ 0  1 32]]
```

	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
micro avg	0.94	0.94	0.94	105
macro avg	0.95	0.95	0.94	105
weighted avg	0.95	0.94	0.94	105

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 ?

$$recall = \frac{\text{Nombre de SPAM correctement reconnus}}{\text{Nombre total de SPAM dans le jeu de données}} = \frac{TP}{FN + TP} = \frac{40}{40 + 5} = 0.8$$

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

$$precision = \frac{\text{Nombre de SPAM correctement reconnus}}{\text{Nombre de fois où un objet a été prédit SPAM}} = \frac{TP}{TP + FP} = \frac{40}{40 + 10} =$$

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 précision 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 :

<i>Iris – setosa</i>	34	0	0
<i>Iris – versicolor</i>	0	33	5
<i>Iris – virginica</i>	0	1	32

classification_report retourne le résultat suivant :

	<i>precision</i>	<i>recall</i>	<i>f1 – score</i>	<i>support</i>
<i>Iris – setosa</i>	1.00	1.00	1.00	34
<i>Iris – versicolor</i>	0.97	0.87	0.92	38
<i>Iris – virginica</i>	0.86	0.97	0.91	33
<i>avg/total</i>	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 :

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

Pour connaître les objets mal classés :

```
In [19]: 1 misclassified = np.where(y != clf.predict(X))
2
3 print('Les objets mal classés sont :')
4
5 i=0
6 ▼ for i in misclassified:
7     print('\n',df.iloc[i,:]['Species'])
8     print ('\n')
9
10 ▼ for i in misclassified:
11     print ('\n', i,'classé en ',clf.predict(X)[i],'\n')
12
```

Les objets mal classés sont :

```
52     Iris-versicolor
56     Iris-versicolor
70     Iris-versicolor
77     Iris-versicolor
83     Iris-versicolor
106    Iris-virginica
119    Iris-virginica
Name: Species, dtype: object
```

```
[ 52  56  70  77  83 106 119] classé en ['Iris-virginica' 'Iris-vi
rginica' 'Iris-virginica' 'Iris-virginica'
'Iris-virginica' 'Iris-versicolor' 'Iris-versicolor']
```

Pour afficher, avec seaborn, la matrice de confusion.

In [20]:

```

1  import seaborn as sns
2  import matplotlib.pyplot as plt
3
4  labels = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
5  fig, ax = plt.subplots(1, 1, figsize=(10,6))
6  hm = sns.heatmap(conf,
7                  ax=ax,                # Axes où afficher
8                  xticklabels=labels, # labels sur les x
9                  yticklabels=labels, # labels sur les colonnes
10                 cmap="YlGnBu", # Couleur
11                 square=True,       # Si True, toutes les cellules
12                                    # ont le même aspect carré
13                 annot=True         # Pour afficher les valeurs
14                 )
15  fig.suptitle('GaussianNB\nAccuracy:{0:.3f}'.format(accuracy_score(y_test, y_pred)),
16              fontsize=12,
17              fontweight='bold')
18  plt.show()

```

<Figure size 1000x600 with 2 Axes>

Les métriques peuvent être appelées indépendamment. Par exemple

from sklearn.metrics import precision_score

print("Precision score: {}".format(precision_score(y_true,y_pred)))

ou à l'aide de la fonction *precision_recall_fscore_support*

In [21]:

```

1  from sklearn.metrics import precision_recall_fscore_support as s
2
3  precision, recall, fscore, support = score(y_test, result)
4
5  print('precision: {}'.format(precision))
6  print('recall: {}'.format(recall))
7  print('fscore: {}'.format(fscore))
8  print('support: {}'.format(support))

```

```

precision: [1.          0.97058824 0.86486486]
recall: [1.          0.86842105 0.96969697]
fscore: [1.          0.91666667 0.91428571]
support: [34  38  33]

```



Utiliser plusieurs classifieurs

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.

```
In [22]: 1 #preparation des données
2 import pandas as pd
3 from sklearn.model_selection import train_test_split
4 from sklearn.metrics import accuracy_score
5 url = "https://archive.ics.uci.edu/ml/machine-learning-databases
6 names = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'Peta
7 df = pd.read_csv(url, names=names)
8 array = df.values
9 X = array[:,0:4]
10 y = array[:,4]
11
12 validation_size=0.3 #30% du jeu de données pour le test
13
14 testsize= 1-validation_size
15 seed=30
16 X_train, X_test, y_train, y_test = train_test_split(X, y, train_
17
```

Dans l'exemple, nous utilisons 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.

```
In [23]: 1 from sklearn.linear_model import LogisticRegression
2 from sklearn.tree import DecisionTreeClassifier
3 from sklearn.neighbors import KNeighborsClassifier
4 from sklearn.naive_bayes import GaussianNB
5 from sklearn.svm import SVC
6
7 seed = 7
8 scoring = 'accuracy'
9 models = []
10 models.append(('LR', LogisticRegression(solver='lbfgs')))
11 models.append(('KNN', KNeighborsClassifier()))
12 models.append(('CART', DecisionTreeClassifier()))
13 models.append(('NB', GaussianNB()))
14 models.append(('SVM', SVC(gamma='auto')))
15
16
```

In [24]:

```

1  from sklearn.metrics import confusion_matrix
2  from sklearn.metrics import classification_report
3  from sklearn.model_selection import KFold
4  from sklearn.model_selection import cross_val_score
5  results = []
6  names = []
7  for name,model in models:
8      kfold = KFold(n_splits=10, random_state=seed)
9      cv_results = cross_val_score(model, X, y, cv=kfold, scoring=
10     #pour avoir les paramètres utilisés dans le modèle enlever c
11     #print (model.get_params())
12      results.append(cv_results)
13      names.append(name)
14      msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.s
15      print(msg)

```

LR: 0.900000 (0.116428)
 KNN: 0.933333 (0.084327)
 CART: 0.933333 (0.084327)
 NB: 0.946667 (0.058119)
 SVM: 0.953333 (0.052068)

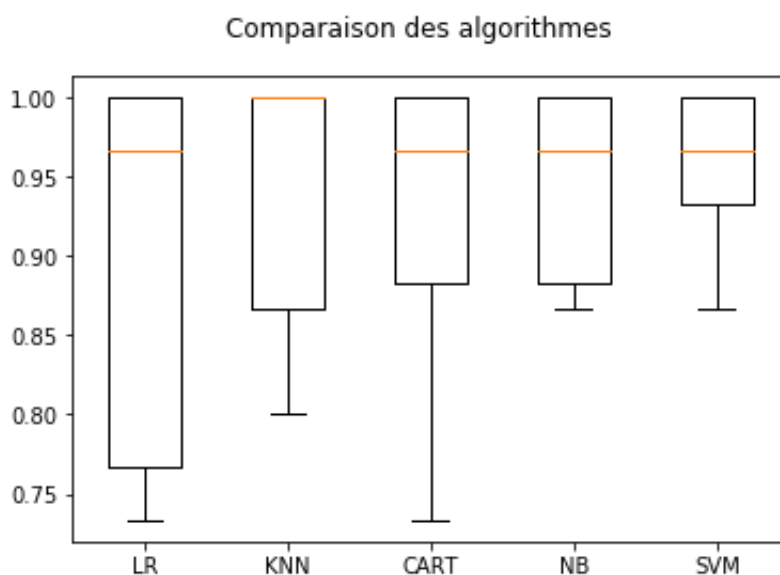
In [25]:

```

1  fig = plt.figure()
2  fig.suptitle('Comparaison des algorithmes')
3  ax = fig.add_subplot(111)
4  plt.boxplot(results)
5  ax.set_xticklabels(names)

```

Out[25]: [Text(0, 0, 'LR'),
 Text(0, 0, 'KNN'),
 Text(0, 0, 'CART'),
 Text(0, 0, 'NB'),
 Text(0, 0, 'SVM')]



Comme SVM donne des meilleurs résultats, nous pouvons l'utiliser comme modèle de prédiction.

```
In [26]: 1 from sklearn.metrics import accuracy_score
2 from sklearn.metrics import confusion_matrix
3 from sklearn.metrics import classification_report
4 clf = SVC(gamma='auto')
5 clf.fit(X_train, y_train)
6 result = clf.predict(X_test)
7
8 print('\n accuracy: ', accuracy_score(result, y_test),'\n')
9
10 conf = confusion_matrix(y_test, result)
11 print ('\n matrice de confusion \n',conf)
12 print ('\n',classification_report(y_test, result))
```

```
accuracy: 0.9619047619047619
```

```
matrice de confusion
```

```
[[34  0  0]
 [ 0 34  4]
 [ 0  0 33]]
```

	precision	recall	f1-score	support
Iris-setosa	1.00	1.00	1.00	34
Iris-versicolor	1.00	0.89	0.94	38
Iris-virginica	0.89	1.00	0.94	33
micro avg	0.96	0.96	0.96	105
macro avg	0.96	0.96	0.96	105
weighted avg	0.97	0.96	0.96	105

▼ 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 complètement différents (choix du noyau SVM, nombre de K dans KNeighbors, etc.). Sikit learn permet de pouvoir faire une recherche exhaustive (grid search) pour trouver les paramètres les plus pertinents pour un classifieur.

In [27]:

```

1  ▼ #preparation des données
2  import pandas as pd
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import accuracy_score
5  url = "https://archive.ics.uci.edu/ml/machine-learning-databases
6  names = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'Peta
7  df = pd.read_csv(url, names=names)
8  array = df.values
9  X = array[:,0:4]
10 y = array[:,4]
11
12 validation_size=0.3 #30% du jeu de données pour le test
13
14 testsize= 1-validation_size
15 seed=30
16 ▼ X_train, X_test, y_train, y_test = train_test_split(X, y,
17                                                         train_size=v
18                                                         random_state
19                                                         test_size=te
20
21

```

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*. Elle 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.

In [28]:

```

1  from sklearn.tree import DecisionTreeClassifier
2  from sklearn.model_selection import GridSearchCV
3
4  ▼ grid_param = {
5      'max_depth': [1,2,3,4,5,6,7,8,9,10],
6      'criterion': ['gini', 'entropy'],
7      'min_samples_leaf': [1,2,3,4,5,6,7,8,9,10]
8  }
9
10
11  ▼ gd_sr = GridSearchCV(estimator=DecisionTreeClassifier(),
12                        param_grid=grid_param,
13                        scoring='accuracy',
14                        cv=5,
15                        n_jobs=-1,
16                        iid=True,
17                        return_train_score=True)
18
19  gd_sr.fit(X_train, y_train)
20
21

```

```

Out[28]: GridSearchCV(cv=5, error_score='raise-deprecating',
                      estimator=DecisionTreeClassifier(class_weight=None, criterion
='gini', max_depth=None,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort=False, random_stat
e=None,
                      splitter='best'),
                      fit_params=None, iid=True, n_jobs=-1,
                      param_grid={'max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 'cr
iterion': ['gini', 'entropy'], 'min_samples_leaf': [1, 2, 3, 4, 5, 6
, 7, 8, 9, 10]}},
                      pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
                      scoring='accuracy', verbose=0)

```

Pour connaître les meilleures conditions :

In [29]:

```

1 print ('meilleur score ',gd_sr.best_score_,'\n')
2 print ('meilleurs paramètres', gd_sr.best_params_,'\n')
3 print ('meilleur estimateur',gd_sr.best_estimator_,'\n')

```

meilleur score 0.9777777777777777

meilleurs paramètres {'criterion': 'gini', 'max_depth': 2, 'min_samples_leaf': 1}

meilleur estimateur DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=2, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=None, splitter='best')

Avec KNeighborsClassifier()

In [30]:

```

1 from sklearn.neighbors import KNeighborsClassifier
2 grid_param = {
3     'n_neighbors': list(range(1,15)),
4     'metric': ['minkowski', 'euclidean', 'manhattan']
5 }
6
7 gd_sr = GridSearchCV(estimator=KNeighborsClassifier(),
8                       param_grid=grid_param,
9                       scoring='accuracy',
10                      cv=5,
11                      n_jobs=-1,
12                      iid=True,
13                      return_train_score=True)
14
15 gd_sr.fit(X_train, y_train)
16
17 print ('meilleur score ',gd_sr.best_score_,'\n')
18 print ('meilleurs paramètres', gd_sr.best_params_,'\n')
19 print ('meilleur estimateur',gd_sr.best_estimator_,'\n')
20

```

meilleur score 1.0

meilleurs paramètres {'metric': 'minkowski', 'n_neighbors': 1}

meilleur estimateur KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski', metric_params=None, n_jobs=None, n_neighbors=1, p=2, weights='uniform')

Avec SVM

In [31]:

```
1  from sklearn.svm import SVC
2  grid_param = {
3      'C': [0.001, 0.01, 0.1, 1, 10],
4      'gamma': [0.001, 0.01, 0.1, 1],
5      'kernel': ['linear', 'rbf']}
6
7  gd_sr = GridSearchCV(estimator=SVC(),
8                      param_grid=grid_param,
9                      scoring='accuracy',
10                     cv=5,
11                     n_jobs=1,
12                     iid=True,
13                     return_train_score=True)
14
15  gd_sr.fit(X_train, y_train)
16
17  print ('meilleur score ',gd_sr.best_score_,'\n')
18  print ('meilleurs paramètres', gd_sr.best_params_,'\n')
19  print ('meilleur estimateur',gd_sr.best_estimator_,'\n')
20
```

meilleur score 1.0

meilleurs paramètres {'C': 1, 'gamma': 0.001, 'kernel': 'linear'}

meilleur estimateur SVC(C=1, cache_size=200, class_weight=None, coef_0=0.0, decision_function_shape='ovr', degree=3, gamma=0.001, kernel='linear', max_iter=-1, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False)

Pour voir l'ensemble des évaluations effectuées par GridSearchCV :

In [32]:

```

1
2 # conversion en DataFrame
3 results = pd.DataFrame(gd_sr.cv_results_)
4 # Affichage des 5 premières lignes
5 display(results.head())
6

```

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_C	param_gam
0	0.000723	0.000389	0.444444	0.452069	0.001	0.
1	0.000432	0.000296	0.444444	0.452069	0.001	0.
2	0.000491	0.000369	0.444444	0.452069	0.001	0.
3	0.000488	0.000377	0.444444	0.452069	0.001	0.
4	0.000389	0.000261	0.444444	0.452069	0.001	0.

5 rows × 7 columns

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*.

In [33]:

```

1  from sklearn.tree import DecisionTreeClassifier
2  from sklearn.model_selection import RandomizedSearchCV
3  from scipy.stats import randint
4
5  ▼ rand_param = {
6      'max_depth': randint(1, 20),
7      'criterion': ['gini', 'entropy'],
8      'min_samples_leaf': randint(1, 20)
9  }
10
11
12  ▼ rand_sr = RandomizedSearchCV(estimator=DecisionTreeClassifier(),
13                                param_distributions = rand_param,
14                                random_state=1,
15                                n_iter=20,
16                                cv=3,
17                                n_jobs=-1,
18                                scoring='accuracy',
19                                iid=True,
20                                return_train_score=True)
21
22  rand_sr.fit(X_train, y_train)
23
24  print ('meilleur score ',rand_sr.best_score_,'\n')
25  print ('meilleurs paramètres', rand_sr.best_params_,'\n')
26  print ('meilleur estimateur',rand_sr.best_estimator_,'\n')
27
28  # conversion en DataFrame
29  results = pd.DataFrame(rand_sr.cv_results_)
30  # Affichage des 5 premières lignes
31  display(results.head())
32
33

```

```
meilleur score  0.9555555555555556
```

```
meilleurs paramètres {'criterion': 'entropy', 'max_depth': 16, 'min_
samples_leaf': 1}
```

```
meilleur estimateur DecisionTreeClassifier(class_weight=None, criter
```

```

ion='entropy', max_depth=16,
    max_features=None, max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    min_samples_leaf=1, min_samples_split=2,
    min_weight_fraction_leaf=0.0, presort=False, random_stat
e=None,
    splitter='best')

```

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_criterion	para
0	0.000873	0.000495	0.733333	0.733136	entropy	
1	0.000649	0.000485	0.711111	0.733136	gini	
2	0.000369	0.000250	0.955556	1.000000	entropy	
3	0.000301	0.000234	0.733333	0.733136	gini	
4	0.000272	0.000227	0.955556	1.000000	entropy	



Gridsearch et plusieurs classifieurs

Il est tout à fait possible d'utiliser Gridsearch avec plusieurs classifieurs. Il suffit pour cela d'initialiser les classifieurs dans une bibliothèque et faire de même pour les paramètres.

In [34]:

```

1
2 from sklearn.tree import DecisionTreeClassifier
3 from sklearn.neighbors import KNeighborsClassifier
4 from sklearn.svm import SVC
5
6 classifiers = {
7     'KNeighborsClassifier': KNeighborsClassifier(),
8     'DecisionTreeClassifier': DecisionTreeClassifier(),
9     'SVC': SVC()
10 }
11
12 params = {'KNeighborsClassifier': [{'n_neighbors': list(range(1
13     {'metric': ['minkowski', 'euclidean', 'manhattan'])}],
14     'DecisionTreeClassifier': [{'max_depth': [1,2,3,4,5,6
15     {'criterion': ['gini', 'entropy']},
16     {'min_samples_leaf': [1,2,3,4,5,6,7,8,9,10]}],
17     'SVC': [{'C': [0.001, 0.01, 0.1, 1, 10],
18     'gamma': [0.001, 0.01, 0.1, 1],
19     'kernel': ['linear', 'rbf']}] }
20
21

```

In [35]:

```

1 class Result:
2     def __init__(self, name, score, parameters):
3         self.name = name
4         self.score = score
5         self.parameters = parameters
6     def __repr__(self):
7         return repr((self.name, self.score, self.parameters))
8
9
10 results = []
11 for key, value in classifiers.items():
12     gd_sr = GridSearchCV(estimator=value,
13         param_grid=params[key],
14         scoring='accuracy',
15         cv=5,
16         n_jobs=1,
17         iid=True)
18     gd_sr.fit(X_train, y_train)
19     result=Result(key, gd_sr.best_score_, gd_sr.best_estimator_)
20     results.append(result)
21
22
23
24 results=sorted(results, key=lambda result: result.score, reverse
25
26 print ('Le meilleur resultat : \n')
27 print ('Classifier : ', results[0].name,
28     ' score %0.2f' % results[0].score,
29     ' avec ', results[0].parameters, '\n')
30
31 print ('Tous les résultats : \n')

```

```

32  ▼ for result in results:
33  ▼     print ('Classifier : ',result.name,
34             ' score %0.2f' %result.score,
35             ' avec ',result.parameters,'\n')
36
37

```

Le meilleur resultat :

```

Classifier : KNeighborsClassifier score 1.00 avec KNeighborsClassifier(
algorithm='auto', leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=None, n_neighbors=1, p=2,
weights='uniform')

```

Tous les résultats :

```

Classifier : KNeighborsClassifier score 1.00 avec KNeighborsClassifier(
algorithm='auto', leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=None, n_neighbors=1, p=2,
weights='uniform')

```

```

Classifier : SVC score 1.00 avec SVC(C=1, cache_size=200, class_weight=None,
coef0=0.0,
decision_function_shape='ovr', degree=3, gamma=0.001, kernel='linear',
max_iter=-1, probability=False, random_state=None, shrinking=True,
tol=0.001, verbose=False)

```

```

Classifier : DecisionTreeClassifier score 0.98 avec DecisionTreeClassifier(
class_weight=None, criterion='gini', max_depth=2,
max_features=None, max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, presort=False, random_state=None,
splitter='best')

```

▼ 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 classifieur. Le principe consiste à d'abord mettre la chaîne de pré-traitement, d'ensuite mettre le classifieur 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.

In [36]:

```

1  from sklearn.model_selection import train_test_split
2  from sklearn.preprocessing import StandardScaler
3  from sklearn.decomposition import PCA
4  from sklearn.pipeline import Pipeline
5  from sklearn.tree import DecisionTreeClassifier
6
7  from sklearn.preprocessing import LabelEncoder
8
9  import pandas as pd
10 from sklearn.model_selection import train_test_split
11 from sklearn.metrics import accuracy_score
12 url = "https://archive.ics.uci.edu/ml/machine-learning-databases
13 names = ['SepalLengthCm', 'SepalWidthCm',
14          'PetalLengthCm', 'PetalWidthCm', 'Species']
15 df = pd.read_csv(url, names=names)
16
17 #transformation de Species en float pour StandardScaler
18 class_label_encoder = LabelEncoder()
19 df['Species']=class_label_encoder.fit_transform(df['Species']).va
20
21
22 array = df.values
23 X = array[:,0:4]
24 y = array[:,4]
25
26
27 print ('Création du pipeline \n')
28 pipeline = Pipeline([('scl', StandardScaler()),
29                      ('pca', PCA(n_components=2)),
30                      ('clf', DecisionTreeClassifier(random_state=
31
32 validation_size=0.3 #30% du jeu de données pour le test
33
34 testsize= 1-validation_size
35 seed=30
36 X_train,X_test,y_train,y_test=train_test_split(X, y,
37                                                  train_size=valida
38                                                  random_state=seed
39                                                  test_size=testsize
40
41
42 pipeline.fit(X_train, y_train)
43 result = pipeline.predict(X_test)
44
45 print('\n accuracy:',accuracy_score(result, y_test),'\n')
46

```

Création du pipeline

accuracy: 0.9142857142857143

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

```
In [37]: 1 from sklearn.model_selection import GridSearchCV
2 print ('Création du pipeline \n')
3 pipeline = Pipeline([('pca', PCA()),
4                       ('clf', DecisionTreeClassifier(random_state=
5
6 grid_param = {
7     'pca__n_components': [2,3]
8 }
9
10
11 gd_sr = GridSearchCV(pipeline,
12                      param_grid=grid_param,
13                      scoring='accuracy',
14                      cv=5,
15                      n_jobs=1,
16                      iid=True,
17                      return_train_score=True)
18
19 gd_sr.fit(X_train, y_train)
20
21 print ('meilleur score ',gd_sr.best_score_,'\n')
22 print ('meilleurs paramètres', gd_sr.best_params_,'\n')
23 print ('meilleur estimateur',gd_sr.best_estimator_,'\n')
```

Création du pipeline

meilleur score 0.9555555555555556

meilleurs paramètres {'pca__n_components': 2}

```
meilleur estimateur Pipeline(memory=None,
    steps=[('pca', PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
        svd_solver='auto', tol=0.0, whiten=False)), ('clf', DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
        max_features=None, max_leaf_nodes=None,
        min_impurity_decrease=0.0, min_impurity_split=None,
        min_samples_leaf=1, min_samples_split=2,
        min_weight_fraction_leaf=0.0, presort=False, random_state=42,
        splitter='best'))])
```

Ou bien de faire les deux en même temps.

```
In [38]: 1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.metrics import accuracy_score
```

```

4 url = https://archive.ics.uci.edu/ml/machine-learning-databases
5 names = ['SepalLengthCm', 'SepalWidthCm',
6          'PetalLengthCm', 'PetalWidthCm', 'Species']
7 df = pd.read_csv(url, names=names)
8
9 #transformation de Species en float pour StandardScaler
10 class_label_encoder = LabelEncoder()
11 df['Species']=class_label_encoder.fit_transform(df['Species'].va
12
13
14 array = df.values
15 X = array[:,0:4]
16 y = array[:,4]
17
18 pipeline = Pipeline([('pca', PCA()),
19                       ('clf', DecisionTreeClassifier())])
20
21
22
23 grid_param = [{'pca__n_components': [2,3]},
24               {'clf': [DecisionTreeClassifier()],
25                  'clf__max_depth': [1,2,3,4,5,6,7,8,9,10],
26                  'clf__criterion': ['gini', 'entropy'],
27                  'clf__min_samples_leaf': [1,2,3,4,5,6,7,8,9,10]
28               }]
29
30
31
32 gd_sr = GridSearchCV(estimator=pipeline,
33                      param_grid=grid_param,
34                      scoring='accuracy',
35                      cv=5,
36                      n_jobs=-1,
37                      iid=True,
38                      return_train_score=True)
39
40 gd_sr.fit(X_train, y_train)
41 print ('meilleur score ',gd_sr.best_score_,'\n')
42 print ('meilleurs paramètres', gd_sr.best_params_,'\n')
43 print ('meilleur estimateur',gd_sr.best_estimator_,'\n')

```

meilleur score 0.9555555555555556

meilleurs paramètres {'pca__n_components': 2}

meilleur estimateur Pipeline(memory=None,
 steps=[('pca', PCA(copy=True, iterated_power='auto', n_componen
 ts=2, random_state=None,
 svd_solver='auto', tol=0.0, whiten=False)), ('clf', DecisionTreeCl
 assifier(class_weight=None, criterion='gini', max_depth=None,
 max_features=None, max_leaf_nodes=None,
 min_impurity_decrease=0.0, min_impurity_split=None,
 min_samples_leaf=1, min_samples_split=2,
 min weight fraction leaf=0.0, presort=False, random stat

```
e=None,
        splitter='best'))))
```

Sauvegarder le modèle appris

Une fois un modèle appris il est possible de le sauvegarder pour pouvoir lui appliquer d'autres données à prédire. Deux librairies existent : **pickle** et **joblib**.

Pickle est la librairie Python standard pour sérialiser-désérialiser des objets. standard Python tool for object (de)serialization. Joblib propose également de sérialiser-désérialiser des objets lorsque ceux-ci sont très volumineux.

Le choix des deux dépend des usages.

```
In [39]: 1  #preparation des données
2  import pandas as pd
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import accuracy_score
5  from sklearn.naive_bayes import GaussianNB
6  url = "https://archive.ics.uci.edu/ml/machine-learning-databases
7  names = ['SepalLengthCm', 'SepalWidthCm',
8           'PetalLengthCm', 'PetalWidthCm', 'Species']
9  df = pd.read_csv(url, names=names)
10 array = df.values
11 X = array[:,0:4]
12 y = array[:,4]
13
14 validation_size=0.3 #30% du jeu de données pour le test
15
16 testsize= 1-validation_size
17 seed=30
18 X_train,X_test,y_train,y_test=train_test_split(X,
19                                                y,
20                                                train_size=valida
21                                                random_state=seed
22                                                test_size=testsiz
23
24 clf = GaussianNB()
25 clf.fit(X_train, y_train)
26
```

```
Out[39]: GaussianNB(priors=None, var_smoothing=1e-09)
```

pickle

Pour sérialiser et sauvegarder le modèle appris :

```
In [40]: 1 import pickle
2 filename = 'pkl_modelNB.sav'
3 pickle.dump(clf, open(filename, 'wb'))
```

Pour utiliser le modèle sauvegardé :

```
In [41]: 1 from sklearn.metrics import accuracy_score
2 from sklearn.metrics import confusion_matrix
3 from sklearn.metrics import classification_report
4
5 clf_loaded = pickle.load(open(filename, 'rb'))
6 print ('Modèle chargé',clf_loaded,'\n')
7 result = clf_loaded.predict(X_test)
8
9 print('\n accuracy:\n')
10 print (accuracy_score(result, y_test),'\n')
11
12 conf = confusion_matrix(y_test, result)
13 print ('\n matrice de confusion \n',conf)
14 print ('\n',classification_report(y_test, result))
15
16 result = clf_loaded.predict([[ 5.0, 3.6, 1.4, 0.2]])
17 print ('\nLa prédiction du modèle pour [ 5.0, 3.6, 1.4, 0.2]
18 result)
```

Modèle chargé GaussianNB(priors=None, var_smoothing=1e-09)

accuracy:

0.9428571428571428

matrice de confusion

```
[[34  0  0]
 [ 0 33  5]
 [ 0  1 32]]
```

	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
micro avg	0.94	0.94	0.94	105
macro avg	0.95	0.95	0.94	105
weighted avg	0.95	0.94	0.94	105

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

joblib

Pour sérialiser et sauvegarder le modèle appris :

```
In [42]: 1 from sklearn.externals import joblib
2 filename = 'job_modelNB.sav'
3 joblib.dump(clf, filename)
```

```
Out[42]: ['job_modelNB.sav']
```

Pour utiliser le modèle sauvegardé :

```
In [43]: 1 from sklearn.metrics import accuracy_score
2 from sklearn.metrics import confusion_matrix
3 from sklearn.metrics import classification_report
4
5 clf_loaded = joblib.load(filename)
6 print (clf_loaded)
7 #result = clf_loaded.score(X_test, y_test)
8 #print(result)
9
10 result = clf_loaded.predict(X_test)
11
12 print('\n accuracy:\n')
13 print (accuracy_score(result, y_test),'\n')
14
15
```

```
GaussianNB(priors=None, var_smoothing=1e-09)
```

```
accuracy:
```

```
0.9428571428571428
```

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.

```
In [44]: 1 from sklearn import svm
2 from sklearn import datasets
3 from sklearn import preprocessing
4 from sklearn.metrics import accuracy_score
5 from sklearn.model_selection import train_test_split
6 from sklearn.metrics import accuracy_score
7 import pickle
8 from sklearn.linear_model import SGDClassifier
9 from sklearn.pipeline import Pipeline
10 from sklearn.feature_extraction.text import TfidfTransformer
11 from sklearn.metrics import accuracy_score, confusion_matrix
12 from time import time
13 from sklearn.metrics import classification_report
14
15
```

Lecture de la base iris et utilisation de SVM comme classifieur

```
In [45]: 1 clf = svm.SVC(gamma='scale')
2 iris = datasets.load_iris()
3
```

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. Nous avons vu (Ingénierie des données) que SVM était très sensible à la standardisation.

```
In [46]: 1 for i in range (len(iris.data)):
2     for j in range (0,2):
3         val = iris.data[i][j]
4         value = val*1000
5         iris.data[i][j]=value
```

Définition de X et de y

```
In [47]: 1 X = iris.data
2 y = iris.target
```

```
In [48]: 1 validation_size=0.3 #30% du jeu de données pour le test
2
3 testsize= 1-validation_size
4 seed=30
5 X_train,X_test,y_train,y_test=train_test_split(X,
6                                             y,
7                                             train_size=valida
8                                             random_state=seed
9                                             test_size=testsize)
```

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

```
In [49]: 1 def cpt_mal_classes(y_test,result):
2         nb=0
3         for i in range(len(y_test)):
4             if y_test[i] != result [i]:
5                 nb=nb+1
6         return nb
7
8 def print_nb_classes (taille,nb):
9     print ("Taille des données",
10           taille,
11           " mal classés",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.

```
In [50]: 1 clf.fit(X_train, y_train)
2
3 from sklearn.metrics import accuracy_score
4
5 result = clf.predict(X_test)
6 nb=cpt_mal_classes(y_test,result)
7 taille=len(y_test)
8 print_nb_classes (len(y_test),nb)
9 print('\n accuracy :', accuracy_score(result, y_test),'\n')
10
```

Taille des données 105 mal classés 46

accuracy : 0.5619047619047619

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.

```
In [51]: 1 validation_size=0.3 #30% du jeu de données pour le test
2
3 testsize= 1-validation_size
4 seed=30
5 X_train,X_test,y_train,y_test=train_test_split(X,
6                                                y,
7                                                train_size=valida
8                                                random_state=seed
9                                                test_size=testsize)
```

Standardisation des données et sauvegarde du jeu de test avant le passage par la standardisation. 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.

```
In [52]: 1 scaler = preprocessing.MinMaxScaler()
2 X_train = scaler.fit_transform(X_train)
3 X_save=X_test.copy()
4 X_test = scaler.fit_transform(X_test)
```

```
In [53]: 1 clf.fit(X_train, y_train)
2
3 result = clf.predict(X_test)
4 nb=cpt_mal_classes(y_test,result)
5 taille=len(y_test)
6 print_nb_classes (len(y_test),nb)
7 print('\n accuracy :', accuracy_score(result, y_test),'\n')
```

Taille des données 105 mal classés 5

accuracy : 0.9523809523809523

Sauvegarde du modèle appris

```
In [54]: 1 print("\nSauvegarde du modèle")
2 filename = 'firstmodel.pkl'
3 pickle.dump(clf, open(filename, 'wb'))
4
5
```

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.

```
In [55]: 1 print ("Chargement du modèle \n")
2 filename = 'firstmodel.pkl'
3 clf_loaded = pickle.load(open(filename, 'rb'))
4
5 result=clf_loaded.predict(X_save)
6
7 nb=cpt_mal_classes(y_test,result)
8 taille=len(y_test)
9 print_nb_classes (len(y_test),nb)
```

Chargement du modèle

Taille des données 105 mal classés 72

Nous pouvons constater qu'il y a 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).

In [56]:

```
1  pipeline = Pipeline([('vect', preprocessing.MinMaxScaler()),
2                        ('clf', svm.SVC(gamma='scale')),
3                        ])
4
5
6
7
8  X=iris.data
9  y=iris.target
10
11
12  X_train,X_test,y_train,y_test=train_test_split(X,
13                                                  y,
14                                                  train_size=valida
15                                                  random_state=seed
16                                                  test_size=testsize)
17
18  X_save=X_test.copy()
19
20  pipeline.fit(X_train, y_train)
21
22  result = pipeline.predict(X_test)
23
24  nb=cpt_maj_classes(y_test,result)
25  taille=len(y_test)
26  print_nb_classes (len(y_test),nb)
27  print('\n accuracy:',accuracy_score(result, y_test),'\n')
28
29
30  print("\nSauvegarde du pipeline ")
31  filename = 'avecscaler.pkl'
32  pickle.dump(pipeline, open(filename, 'wb'))
```

Taille des données 105 mal classés 5

accuracy: 0.9523809523809523

Sauvegarde du pipeline

```
In [57]: 1 print ("Chargement du modèle \n")
2 filename = 'avecscaler.pkl'
3 clf_loaded = pickle.load(open(filename, 'rb'))
4
5 result=clf_loaded.predict(X_save)
6 nb=cpt_mal_classes(y_test,result)
7 taille=len(y_test)
8 print_nb_classes (len(y_test),nb)
9
10
```

Chargement du modèle

Taille des données 105 mal classés 5