TP 2 : Classification supervisée

Objectif du TP :

- Manipuler les algorithmes de classification supervisés
- Résoudre des problèmes de classification
- Maîtriser les algorithmes KNN, Arbre de décision et SVM

Exercice N°1 : Implémentation de l'algorithme KNN

❖ Présentation :

K-Nearest Neighbors (KNN) : C'est un algorithme d'apprentissage supervisé. Il sert aussi bien pour la classification que la régression
Principe : Dis-moi qui sont tes voisins, je te dirais qui es-tu !

Les prévisions sont établies pour un nouveau point de données en effectuant une recherche dans l'ensemble complet d'apprentissage pour les K instances les plus similaires (les voisins) et en résumant la variable de sortie pour ces K instances.

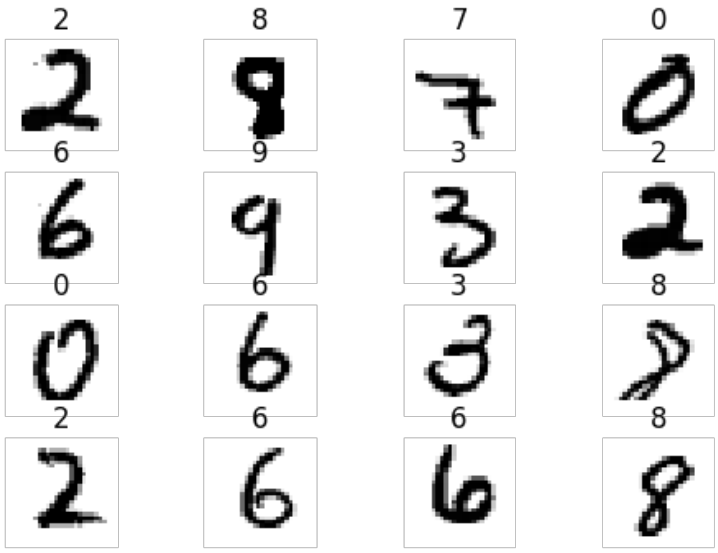
❖ Importation des bibliothèques :

Nous travaillons avec les mêmes packages à importer comme les exemples précédents. Il faut ajouter le package de KNN Classifier ainsi qu'une base de données existante sous **sklearn** pour cet exercice.

```
#Importer les packages nécessaires

from sklearn import neighbors

from sklearn import databases

from sklearn.model_selection import train_test_split
```

❖ Chargement de la base de données

Nous utilisons dans cet exemple la base de données mnist qui provient du package sklearn.datasets. Elle est constitué d'un ensemble de 70000 images 28x28 pixels en noir et blanc annotées du chiffre correspondant (entre 0 et 9). Mnist a la structure suivante :




```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1)
```

```
# On affiche la taille de la base de données

# Le dataset principal qui contient toutes les images

print (mnist.data.shape)


# Le vecteur d'annotations associé au dataset (nombre entre 0 et 9)

print (mnist.target.shape)
```

❖ Echantillonnage de la base de données

```
sample = np.random.randint(70000, size=5000)

data = mnist.data[sample]

target = mnist.target[sample]
```

```
xtrain, xtest, ytrain, ytest = train_test_split(data, target, train_size=0.8)
```

❖ Création du modèle

On peut créer un premier classifieur 3-NN, c'est-à-dire qui prend en compte les 3 plus proches voisins pour la classification. Pour cela, on va utiliser l'implémentation de l'algorithme qui existe dans la librairie scikit-learn :

```
# Création du modèle knn
```

Testons à présent l'erreur de notre classifieur.

❖ Optimisation du modèle :

Pour trouver le k optimal, on va simplement tester le modèle pour tous les k de 2 à 15, mesurer l'erreur test et afficher la performance en fonction de k :

❖ Evaluation du modèle :

À titre d'exemple, on peut afficher les prédictions du classifieur sur quelques données.

```
# On récupère le classifieur le plus performant
knn = neighbors.KNeighborsClassifier(4)
knn.fit(xtrain, ytrain)

# On récupère les prédictions sur les données test
predicted = knn.predict(xtest)
# On redimensionne les données sous forme d'images
images = xtest.reshape((-1, 28, 28))
# On selectionne un echantillon de 12 images au hasard
select = np.random.randint(images.shape[0], size=12)

# On affiche les images avec la prédiction associée
fig,ax = plt.subplots(3,4)

for index, value in enumerate(select):
    plt.subplot(3,4,index+1)
    plt.axis('off')
    plt.imshow(images[value],cmap=plt.cm.gray_r,interpolation="nearest")
    plt.title('Predicted: {}'.format( predicted[value]) )

plt.show()
```

Exercice N°2 : Les arbres de décision :

Principe :

- Un arbre de décision est une succession de questions dont les réponses vont nous amener à des conclusions différentes
- Chaque nœud représente une variable d'entrée unique(x) et un point de partage sur cette variable
- Les feuilles de l'arbre contiennent une variable de sortie (y) utilisée pour effectuer une prédiction

❖ Importation des bibliothèques :

Nous travaillons avec les mêmes packages à importer comme les exemples précédents. Il faut ajouter le package du classifieur de l'arbre de décision.

```
#Importer les packages nécessaires
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
from sklearn.metrics import accuracy_score
```

❖ Chargement de la base de données

Nous utilisons dans cet exemple la base de données recrutement.csv qui a la structure suivante :

```
# On charge la base de données
data = pd.read_csv('recrutement.csv')
```

```
# On affiche les 10 premières valeurs
data.head(10)
```

	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
0	10	Y	4	BS	N	N	Y
1	0	N	0	BS	Y	Y	Y
2	7	N	6	BS	N	N	N
3	2	Y	1	MS	Y	N	Y
4	20	N	2	PhD	Y	N	N
5	0	N	0	PhD	Y	Y	Y
6	5	Y	2	MS	N	Y	Y
7	3	N	1	BS	N	Y	Y
8	15	Y	5	BS	N	N	Y
9	0	N	0	BS	N	N	N

- **Question :** Afin de manipuler correctement le problème de classification, nous devons remplacer les valeurs actuelles des attributs avec le format 'Chaine de caractères' (Employed?, 'Level of Education''Top-tier school', 'Interned et 'Hired') en **des valeurs numériques**.

❖ **Création du modèle**

```
# Création du modèle


-#Préparer les données d'apprentissage et de test
X = data.iloc[ :,:-1]
y = data.iloc[ :,-1]
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size =0.2, random_state = 42)
model.fit(X_train, y_train)
y_pred=model.predict(X_test)
print('précision', metrics.accuracy_score(y_test, y_pred))
```

❖ Visualiser l'arbre de décision

Nous devons installer une nouveau package 'pydotplus'. Ouvrez une console et tapez la commande suivante :

```
pip install 'pydotplus'
```

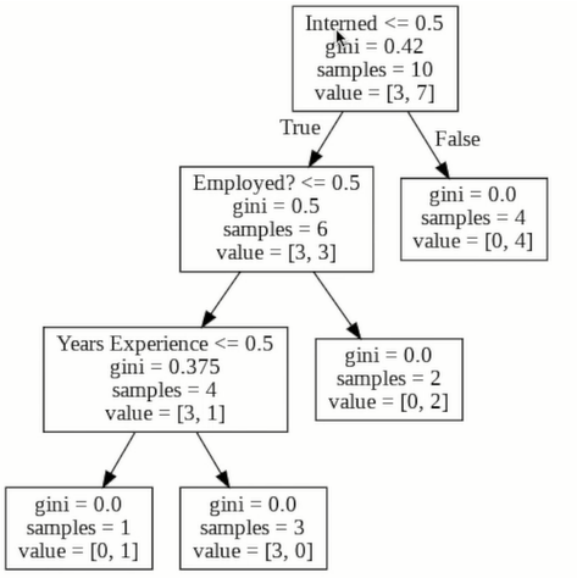
Nous souhaitons visualiser l'arbre de décision construite. On commence par l'importation des bibliothèques requises

```
from sklearn import tree
from sklearn.tree import export_graphviz
from IPython.display import Image
import pydotplus
from sklearn.externals.six import StringIO
```

Ainsi, on rédige le code suivant pour l'affichage de l'arbre de décision

```
dot_data=StringIO()
export_graphviz(model,out_file=dot_data,feature_names=X.columns)
graph=pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

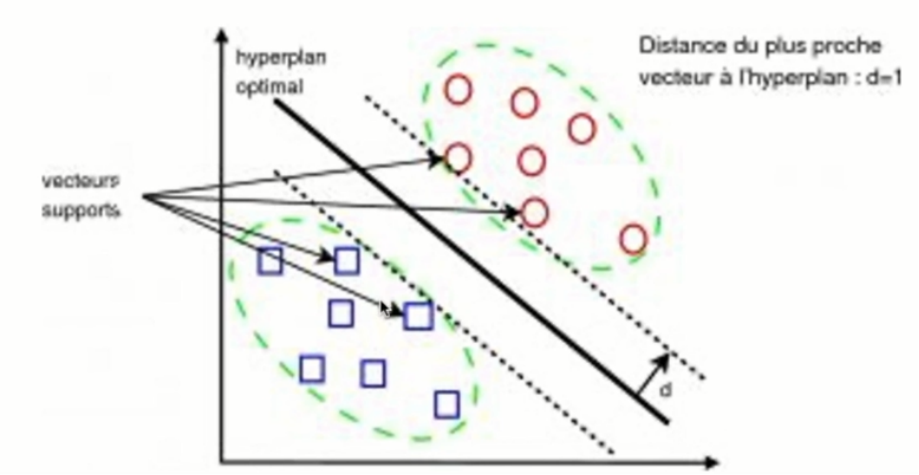
Le résultat doit être ainsi :



Exercice N°3 : Algorithme SVM :

Principe :

- SVM permet de séparer les deux groupes en gardant les plus de **marge** possible
- Lorsque l'on a plus de 2 classes, SVM (on parle de **SVM Multi Classes**) fonctionne très bien
- Souvent, les données ne sont pas séparables, c'est-à-dire qu'il y a toujours des exceptions (dits **slack variables**) qui se retrouvent au milieu d'une autre classe. La technique est de construire la droite qui sépare au mieux avec une erreur la plus faible possible
- On veut séparer les objets encombrants de ceux normaux, dans l'optique d'appliquer une taxe supplémentaire
- On ne garde que l'information « Volume du colis » en fonction du « Poids du colis »
    - ⇨ La droite retenue (hyperplan) est celle dont la distance d (qui la sépare de tous les points au minimum) est maximale.



❖ Importation des bibliothèques :

Nous travaillons avec les mêmes packages à importer comme les exemples précédents. Il faut ajouter le package du classifieur de l'arbre de décision.

```
#Importer les packages nécessaires
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
#Importer les packages nécessaires
from sklearn import datasets, metrics
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import svm
```

❖ Charger les données :

Nous allons utiliser la base de données de la détection du cancer mammaire (breast_cancer).

```
# On charge la base de données
```

```
data.keys()
```

Le résultat affiché doit être comme suit :
dict_keys([‘data’, ‘target’, ‘target_names’, ‘DESCR’, ‘features_names’, ‘filename])
Ce sont les clefs des attributs de notre sélection.

```
print(data.target_names)
```

[‘malignant’, ‘begnin’]

```
# Afficher la description de la base de données
print(data.DESCR)
```

Question : Analyser la description de la base données.

❖ Standardisation des données :

Les valeurs de la base de données nécessitent une standardisation afin que le traitement soit plus facile.

```
X = data.data
Y = data.target
```

```
# Standariser les données
```

```
# Partitionner les données
```

❖ **Application et évaluation du modèle :**

```
# Création du modèle
model = svm.SVC(kernel='linear')
#Apprentissage du modèle
model.fit(X_train, y_train)
#Faire des prévisions
y_pred=model.predict(X_test)
```

```
# Comparer les prévisions avec les valeurs réelles
comp=pd.DataFrame(y_pred, y_test)
comp
```

```
# Evaluation du modèle
print("Précision du modèle", metrics.accuracy_score(y_test, y_pred))
# Affichage de la matrice de confusion
print(metrics.confusion_matrix(y_test, y_pred))
```

❖ SVM avec un noyau polynomial :

```
# Création du modèle
model = svm.SVC(kernel='poly', degree=1, gamma=100)
#Apprentissage du modèle
model.fit(X_train, y_train)
#Faire des prévisions
y_pred=model.predict(X_test)
```

Question : Analyser la précision du modèle avec l'ordre polynomial et comparer-le avec le modèle linéaire ?

❖ SVM avec un noyau Sigmoid :

```
# Création du modèle
model = svm.SVC(kernel='sigmoid', degree=200, C=0.2)
#Apprentissage du modèle
model.fit(X_train, y_train)
#Faire des prévisions
y_pred=model.predict(X_test)
```

Question : Analyser la précision du modèle avec le noyau Sigmod ?