Aix*Marseille université
Initiative d'excellence
Année 2018-19

Diplôme Inter-Universitaire
Enseigner l'informatique au lycée

Bloc 1 : TP 10

Méthode des k plus proches voisins

Ce TP a été imaginé par Cécile Capponi, Valentin Eymia et Rémi Eyraud.

La documentation Scikit-learn sur les k plus proches voisins se trouve ici : http ://scikit-learn.org/stable/modules/neighbors.html

Avant de commencer, vérifiez les versions des paquets Python à l'aide du code suivant :

```
import sys
import numpy as np # importation du package numérique
import matplotlib
import sklearn

print('python: {} (version 3 obligatoire)'.format(sys.version))
print('numpy: {} '.format(np.__version__))
print('matplotlib: {}
      (version conseillée mais pas obligatoire: 3.0.0 au moins)'.format(matplotlib.__version__))
print('scikit-learn: {}
      (version conseillée mais pas obligatoire: 0.19 au moins)'.format(sklearn.__version__))
```

Une version récente de Matplotlib est particulièrement souhaitable pour que l'affichage des images se fasse correctement, sans "lissage".

1 Données *digits*

Nous allons utiliser des données déjà présentes dans scikit-learn. Ces données sont très connues en apprentissage, sous le noms de MNIST. Elles sont composées d'images de chiffres manuscrits à une résolution de 8*8. En scikit-learn, elles se nomment digits :

```
from sklearn.datasets import load_digits

digitsData=load_digits() # jeu de données digits
X=digitsData.data # les exemples, un array numpy, chaque élément est aussi un array
y=digitsData.target # les classes
```

On peut regarder quelques informations :

```
print(X.dtype, X.shape)
print(y.dtype, y.shape)
```

Chaque donnée est donc une image de 8 pixels par 8 pixels, en niveau de gris (256 nuances possibles), stockée sous la forme d'un vecteur de dimension 64 comme une ligne de la matrice X (il y a 1797 images) et avec la valeur de la classe associée stockée dans un vecteur y à part. Mais on peut quand même regarder l'image initiale :

```
import matplotlib.pyplot as plt  # le package de visualisation
# ligne spéciale pour que le notebook affiche comme il faut :
%matplotlib inline
```

```
donnee = X[42,:] # on récupère une ligne, donc une donnée
classe = y[42]   # et sa classe
print("Le vecteur de l'image d'indice 42 : ", donnee)

image = np.reshape(donnee,(8,8)) # on met les 8 morceaux de taille 8 du vecteur
                                 # les uns en dessous des autres
print(image) # on affiche la matrice de pixels
plt.imshow(image) # on affiche l'image qui lui correspond
plt.title('Donnee numero 42, de la classe %i \n' % classe, fontsize = 16) # avec un titre
plt.show()
```

On peut faire des affichages plus intéressant, exemple sur les 5 premières données :

```
plt.figure(figsize=(16,4))
for index in range(5):
    image = X[index, :]
    classe = y[index]
    plt.subplot(1, 5, index + 1)
    plt.imshow(np.reshape(image, (8,8)), cmap=plt.cm.gray)
    plt.title('Classe : %i\n' % classe, fontsize = 18)
plt.show()

plt.figure(figsize=(16,4))
for index in range(5):
    image = X[42+index, :]
    classe = y[42+index]
    plt.subplot(1, 5, index + 1)
    plt.imshow(np.reshape(image, (8,8)), cmap="copper")
    plt.title('Classe : %i\n' % classe, fontsize = 18)
plt.show()
```

2 Création et entraînement d'un classifieur

Notre objectif est maintenant d'apprendre, sur la base d'un échantillon d'images "chiffres", un classifieur capable de prédire le chiffre qui correspond à une nouvelle image. Nous allons utiliser la méthode des k-plus proches voisins pour cet apprentissage. Elle est implémentée dans un package appelé *neighbors*. Examinons la série d'instructions suivante :

```
# importation du package d'algorithmes travaillant sur les points voisins
from sklearn import neighbors as nn
help(nn.KNeighborsClassifier)
```

Continuons l'exploration des k plus proches voisins. Dans la série d'instructions suivante, on indique comment un classifieur peut être appris à partir de données étiquetées, et comment réaliser la prédiction sur un nouvel exemple.

Les fonctions *predict* et *fit* existent **pour tous les classifieurs** disponibles dans scikit-learn.

On note ici la syntaxe de la fonction predict : on lui passe en réalité un tableau d'exemples (ici, un tableau avec un seul exemple constitué de 64 attributs), et elle renvoie un tableau contenant la classe prédite pour chaque exemple du tableau en paramètre. Evidemment, dans les tableaux en entrée et en sortie, les indices des classes prédites correspondent aux indices des exemples en entrée !

Ainsi, lorsque l'on sait que l'on n'applique *predict* qu'à un seul exemple, une sélection finale *[0]* comme ci-après renvoie la première (et la seule) composante du tableau de résultat.

```
# on fixe le nombre de voisins, à partir de 2 et au max le nombre d'exemples dans les données
nb_voisins = 15
# création d'un classifieur: la variable clf est un "objet" classifieur, vide pour l'instant
clf = nn.KNeighborsClassifier(nb_voisins)
print(clf) # le classifieur est vide pour l'instant, il n'a pas été entraîné sur des données

print("Apprentissage")
clf.fit(X, y) # entraînement du classifieur clf sur les données étiquetées

print("Prédiction à l'aide du modèle appris")
nouvel_ex = X[50, :]  # On extrait la 50e image
# prédiction du modèle appris sur la description d'une image aléatoire
print('prediction pour le nouvel exemple: ',
      clf.predict(nouvel_ex.reshape(1,-1)))
print('prediction pour le nouvel exemple: ', clf.predict(nouvel_ex.reshape(1,-1))[0])
```

Pour certains types de classifieurs, on peut même récupérer la probabilité que le classifieur attribue à l'appartenance de l'exemple à chaque classe possible. La fonction *predict_proba* fonctionne comme la fonction *predict*, sauf que le tableau en sortie contient, pour chaque exemple du tableau en entrée, un tableau de probabilité de la même taille que le nombre de classes.

```
autre_ex = X[123, :] # on génère un autre exemple en prenant une autre image
# probabilité d'appartenance à chaque classe pour ce chiffre
print(clf.predict_proba(nouvel_ex.reshape(1,-1))[0])
print(clf.predict_proba(autre_ex.reshape(1,-1))[0]) # idem pour un autre exemple
```

Question 1. A votre avis, quelle classe sera attribuée au deuxième exemple, et pourquoi ? Indiquez ci-après l'instruction à exécuter pour vérifier.

Une première façon d'évaluer la qualité d'un classifieur est de le tester sur les exemples qui ont servi à l'apprendre. On utilise du coup la même fonction *predict*, appliquée au tableau des exemples d'apprentissage.

```
# vecteur des classes prédites pour chaque exemple de l'ensemble d'apprentissage
Z = clf.predict(X)
print(X[Z!=y]) # le tableau d'exemples pour lesquels la prédiction a été mauvaise
```

Question 2. Pour vous rendre compte de l'origine possible des erreurs de prédiction, faites une boucle sur toutes les images pour lesquelles la prédiction est erronée en affichant à chaque fois l'image 8x8 avec dans le titre l'indice de l'image, la classe originale et la classe prédite (pour cela, on peut par exemple utiliser la fonction *numpy.argwhere*).

Chaque classifieur possède une fonction score, qui permet de comparer les prédictions d'un ensemble d'exemples X pour lesquelles on connaît les étiquettes y : la fonction calcule le taux de bonne classifications.

```
# taux de bonne classification du modèle sur l'ensemble d'apprentissage: fonction score
print('taux de bonne classification', clf.score(X,y))
```

Question 3. En déduire le taux d'erreur (vous devez obtenir 0.01446855...).

3 Variation du nombre de voisins

L'algorithme des k-plus proches voisins fonctionne avec plusieurs hyper-paramètres (paramètres de l'agorithme, pas du modèle appris) : la valeur de k est un de ces paramètres.

Question 4. Réalisez un programme qui fait varier cet hyper-paramètre dans un intervalle comprenant des valeurs entre 1 et 15, et stocker l'évolution de l'erreur d'apprentissage (celle calculée sur l'échantillon d'apprentissage), puis en réaliser une courbe avec en abscisse les valeurs de k, et en ordonnées les erreurs. On peut utiliser pour ce faire la fonction de construction d'un tableau *numpy.arange* (cf documentation). Pour la courbe, on utilisera simplement *plot(abs, ord)* du package *pyplot* de *matplotlib*.

Question 5. Qu'observez-vous ? A quelle valeur de k atteint-on un meilleur classifieur ? Quelle est globalement, sur ce jeu de données, l'influence de k ? Que se passe-t-il exactement pour $k = 1$?

4 Estimation de l'erreur réelle du classifieur appris

Lorsque le score du classifieur appris est évalué sur l'ensemble d'apprentissage, il est en général sur-estimé (pourquoi ?) et donc, très peu fiable. La meilleure méthode pour évaluer un classifieur consiste à calculer son score sur un échantillon test, indépendant de l'échantillon d'apprentissage mais généré dans les mêmes conditions. Lorsqu'on dispose d'un seul ensemble d'exemples (comme c'est le cas de *digits*), il faut donc répartir les données en un sous-ensemble d'apprentissage et un sous-ensemble test, entrainer un classifieur sur l'ensemble d'apprentissage, puis évaluer ce classifieur sur l'ensemble test (on a ici une évaluation de l'erreur réelle, qui reste instable puisque dépend du découpage effectué).

Scikit-learn vient avec toute une panoplie d'outils pour évaluer cette erreur. Nous n'utiliserons que la fonction qui permet de diviser un échantillon en deux parties (attributs et classes) : c'est la fonction *train_test_split* du package *model_selection*, que nous appliquons ci-après sur un autre jeu de données très connu, Iris (on n'imprime que les trois premiers exemples de chaque sous-échantillon, avec leurs étiquettes) :

```
from sklearn.model_selection import train_test_split
# production de deux sous-échantillon
Xtrain, Xtest, ytrain, ytest = train_test_split(X,y,test_size=0.25)
print(Xtrain[:3,:], ytrain[:3])
print(Xtest[:3,:], ytest[:3])
```

Ici, nous produisons un découpage dans lequel l'ensemble d'apprentissage représente 75% de l'échantillon initial, et l'échantillon de test représente 25% des données initiales.

Question 6. En vous inspirant de ce mode de découpage, écrire une séquence d'instructions permettant de séparer *digits* en deux parties égales, d'apprendre un 3-plus proches voisins sur le premier sous-échantillon, et de le tester sur le second : vous obtenez une **estimation** de l'erreur réelle. Obtenez-vous la même erreur que celle d'apprentissage mesurée précédemment ?

Question 7. Faites maintenant à nouveau varier k, et pour chaque valeur, indiquez l'erreur réelle estimée sur la base d'un train_test_split de 70%, 30% ; tracer la courbe. Observez les différences de valeurs des erreurs d'apprentissage et réelle : pourquoi sont-elles différentes ? Que constatez-vous ?

5 Variation autour de la métrique

Au-delà du nombre de voisins, un autre hyper-paramètre est la métrique utilisée pour calculer la distance entre les exemples. Par défaut, la distance de Minkowski est utilisée, avec le paramètre $p = 2$ qui indique que nous considérons la distance euclidienne. Avec $p = 1$, nous aurions la distance de manhattan, et de façon générale, avec $p > 0$, la distance utilisée est l_p définie par $l_p(x, x') = (\sum_{i=1}^{d} |x_i - x'_i|^p)^{\frac{1}{p}}$.

Question 8. Ecrire un programme permettant de faire varier la distance utilisée pour évaluer son impact sur les performances, en faisant aussi varier k. Tracez les 3 courbes sur un même plot, une pour chaque valeur de p parmi $\{1, 2, 5\}$.