Et la machine apprit

INFORMATIQUE COMMUNE - TP n° 3.6 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ importer des données stockées dans un fichier de type csv
- ☞ coder l'algorithme knn pour classifier des données étiquetées
- ☞ coder l'algorithme k-means pour classifier des données non étiquetées
- ☞ utiliser le module scikit-learn pour explorer l'apprentissage automatique
- ☞ visualiser un arbre de décision

Ce TP est consacré à l'apprentissage automatique. Les quatre premières sections permettent d'appréhender les algorithmes KNN et K-means en le programmant de A à Z sur des jeux de données simples. C'est l'occasion de réviser la lecture des fichiers en Python et la bibliothèque Numpy. Les sections suivantes sont consacrées à l'usage de la bibliothèque Scikit-learn dans le but de mieux cerner les possibilités fantastiques des outils contemporains d'apprentissage automatique. Elles abordent notamment la compression d'image avec Kmeans et les arbres de décision.

Ⓡ La bilbiothèque Scikit-learn n'est pas au programme. Les arbres de décision non plus.

A Description des jeux de données

Au cours de TP, vous allez manipuler plusieurs jeux de données. En apprentissage automatique, en ce qui concerne la forme d'un jeu de données, on adopte le plus souvent les conventions suivantes :

1. pour les petites quantités de données, le format csv est privilégié et la virgule est souvent le séparateur,
2. les échantillons sont placés en ligne, les colonnes sont les paramètres de chaque échantillon,
3. la première ligne contient une description textuelle de chaque colonne (entêtes),
4. si les données sont étiquetées, la dernière colonne est dédiée à l'étiquette de l'échantillon (la classe).

■ **Exemple 1 — Extrait d'un fichier de données.** Voici un extrait du fichier diabetes.csv.

```
Pregnancies,Glucose,BloodPressure,SkinThickness,Insulin,BMI,DiabetesPedigreeFunction,Age,Outcome
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
```

0,137,40,35,168,43.1,2.288,33,1
...

Les sections suivantes feront appel à deux jeux de données qui sont disponibles en ligne. Ils ont été sélectionnés pour leur simplicité car il existe des jeux plus complets et plus complexes à analyser sur les mêmes sujets ou d'autres thèmes. Ces jeux ne possèdent que des paramètres numériques et l'étiquette est un nombre entier. Ils se prêtent donc bien à l'exploration de KNN et K-means.

1. Prédiction du diabète : diabetes.csv. L'étiquette (Outcome) 1 signifie que le patient est atteint du diabète, 0 qu'il n'est pas atteint.
2. Classification de variétés d'iris : iris.csv. Les étiquettes (variety) 0,1 et 2 correspondent aux variétés Setosa, Versicolor et Virginica.

B Préparation du jeu de données

B1. Coder une fonction de prototype `import_csv(filename)` où `filename` est une chaîne de caractère décrivant le nom d'un fichier. Cette fonction renvoie deux objets : l'entête du fichier sous la forme d'une liste de chaînes de caractères ainsi qu'un tableau Numpy contenant toutes les données. Si la donnée est un nombre, on fera attention à la convertir en type `float`. Par exemple pour le fichier `'diabetes.csv'`, cette fonction renvoie :

```
header, data = import_csv('diabetes.csv')
print(header, data)
# résutlat sur la console -->
['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI',
    'DiabetesPedigreeFunction', 'Age', 'Outcome\n']
[[  6.    148.     72.    ...   0.627  50.      1.   ]
 [  1.     85.     66.    ...   0.351  31.      0.   ]
 [  8.    183.     64.    ...   0.672  32.      1.   ]
 ...
   [  1.     93.     70.    ...   0.315  23.      0.   ]]
```

Solution :

```
def import_csv(filename):
    # conventions :
    # -- data are numbers
    # -- label is at the last column and is a number
    file = open(filename, 'r')
    all_lines = file.readlines()
    file.close()
    headers = all_lines[0].split(",")
    n_lines = len(all_lines) - 1
    n_cols = len(headers)
    data = np.zeros((n_lines,n_cols))
    for i in range(1, len(all_lines)):
        for j in range(n_cols):
            line = all_lines[i].split(",")
            data[i-1,j] = float(line[j])
    return headers, data
```

B2. Écrire une fonction dont le prototype est `describe_data(data, header, labeled=True)` où `data` et `header` sont issus de l'importation des données et `labeled` un booléen qui spécifie si les données sont étiquetées ou non. Cette fonction affiche sur la console les données statistiques des paramètres du jeu de données. Par exemple, pour le fichier `diabetes.csv`, elle affiche :

```
Pregnancies --> Average : 3.85, Std Dev : 3.37, Min : 0.00, Max : 17.00
Glucose --> Average : 120.89, Std Dev : 31.95, Min : 0.00, Max : 199.00
BloodPressure --> Average : 69.11, Std Dev : 19.34, Min : 0.00, Max : 122.00
SkinThickness --> Average : 20.54, Std Dev : 15.94, Min : 0.00, Max : 99.00
Insulin --> Average : 79.80, Std Dev : 115.17, Min : 0.00, Max : 846.00
BMI --> Average : 31.99, Std Dev : 7.88, Min : 0.00, Max : 67.10
DiabetesPedigreeFunction --> Average : 0.47, Std Dev : 0.33, Min : 0.08, Max :
    2.42
Age --> Average : 33.24, Std Dev : 11.75, Min : 21.00, Max : 81.00
```

Solution :

```python
def describe_data(data, header, labeled=True):
  stop = 0
  if labeled:
    stop = data.shape[1] - 1
  else:
    stop = data.shape[1]
  for i in range(stop):
    print(header[i] + " --> ", end="")
    print(f"Average : {np.mean(data[:, i]):3.2f}",end=", ")
    print(f"Std Dev : {np.std(data[:, i]):3.2f}", end=", ")
    print(f"Min : {np.min(data[:, i]):3.2f}", end=", ")
    print(f"Max : {np.max(data[:, i]):3.2f}")
```

Comme on peut l'observer grâce à la question précédente, les données en colonne (paramètres) ne sont pas normalisées. Cependant, avant d'être traitées par les algorithmes d'apprentissage automatique, celles-ci doivent l'être. Dans le cas contraire, la plupart du temps, un paramètre dont les valeurs sont plus compactes que les autres provoque un biais dans la prédiction. C'est pourquoi, on normalise chaque colonne de paramètres entre 0 et 1.

B3. Écrire une fonction de prototype `normalize(data, labeled=True)` où `data` est le tableau Numpy de données issues de l'importation et `labeled` un booléen qui spécifie si les données sont étiquetées ou non. Cette fonction renvoie le tableau Numpy des données normalisées pour chaque colonne, sauf les étiquettes éventuelles. On normalisera par la moyenne et l'écart type :

$$x_i^n = \frac{x_i - \mu_x}{\sigma_x} \tag{1}$$

où x_i est la ième valeur de x, μ_x est la moyenne de x et σ_x l'écart type de x.

Par exemple, pour le jeu de données `diabetes.csv`, après normalisation, on peut utiliser la fonction précédente et afficher à l'écran :

```
Pregnancies --> Average : -0.00, Std Dev : 1.00, Min : -1.14, Max : 3.91
Glucose --> Average : -0.00, Std Dev : 1.00, Min : -3.78, Max : 2.44
BloodPressure --> Average : 0.00, Std Dev : 1.00, Min : -3.57, Max : 2.73
SkinThickness --> Average : 0.00, Std Dev : 1.00, Min : -1.29, Max : 4.92
Insulin --> Average : -0.00, Std Dev : 1.00, Min : -0.69, Max : 6.65
```

```
BMI --> Average : 0.00, Std Dev : 1.00, Min : -4.06, Max : 4.46
DiabetesPedigreeFunction --> Average : 0.00, Std Dev : 1.00, Min : -1.19, Max :
    5.88
Age --> Average : 0.00, Std Dev : 1.00, Min : -1.04, Max : 4.06
```

Solution :

```python
def normalize(data, labeled=True):
    if labeled:
        # Do not normalize labels !
        m = data[:, :-1].mean(axis=0)
        sigma = data[:, :-1].std(axis=0)
        e_data = (data[:, :-1] - m)/ sigma
        labels = np.reshape(data[:, -1], (e_data.shape[0], 1))
        return np.concatenate((e_data, labels), axis=1)
    else:
        m = data[:, :-1].mean(axis=0)
        sigma = data[:, :-1].std(axis=0)
        return (data -m) / sigma
```

C KNN avec Numpy : classification supervisée

Cette section est dédiée à la programmation de l'algorithme KNN de A à Z en s'appuyant uniquement sur Numpy. On rappelle l'algorithme ci-dessous (cf. algorithme 1).

On note $\mathcal{E} = \{(e_i, c_i), i \in [\![1, n]\!], e_i \in \mathbb{R}^d, c_i \in \mathcal{C}\}$, l'ensemble des données étiquetées dont on dispose. On cherche à trouver la classe de $e \in \mathbb{R}^d$. On dispose d'une distance δ sur $\mathbb{R}^d$.

Algorithme 1 k plus proches voisins (KNN)

Fonction KNN(D, x, k, δ)
  $n \leftarrow |D|$
  $\Delta \leftarrow \emptyset$ ▷ Distances à calculer
  pour chaque échantillon étiqueté $e \in \mathcal{E}$ **répéter**
    Ajouter $\delta(x, e)$ à Δ
  Sélectionner les k voisins les plus proches de x en utilisant Δ
  Compter le nombre d'occurrences de chaque classe des k voisins de x
  renvoyer la classe c la plus représentée parmi les k plus proches voisins

Cette section n'a pas pour but d'implémenter l'algorithme d'une manière très performante. On cherche avant tout à comprendre le fonctionnement de l'algorithme.

C1. Écrire une fonction de prototype `dist(p1, p2)` où `p1` et `p2` sont des vecteurs de données numériques de dimension quelconque. Cette fonction renvoie la distance euclidienne entre les deux points.

Solution :

```python
def dist(p1, p2):
  return np.linalg.norm(p1 - p2)
```

C2. Écrire une fonction de prototype `knn(data, x, k, d)` où `data` est le tableau Numpy de données, `x` l'élément qu'on cherche à classifier, `k` l'entier k de l'algorithme KNN et d une fonction calculant la distance entre deux points. Cette fonction implémente l'algorithme KNN et renvoie la classe de x.

Solution :

```python
def knn(data, x, k, d):
  n = data.shape[0]
  assert k < n
  D = []
  for e in data:
    D.append((dist(e[:-1], x), int(e[-1])))
  D = sorted(D)[:k]
  class_nb = np.unique(data[:, -1]).shape[0]
  C = np.zeros((class_nb,))
  for _, c in D:
    C[c] += 1
  return np.argmax(C)
```

C3. Tester la fonction précédente sur les deux jeux de données en calculant la matrice de confusion. On prendra 30% du jeu de données pour les tests. On choisira k tel que $k = \left\lfloor \sqrt{\frac{n}{c}} \right\rfloor$ où n est le nombre d'échantillons et c le nombre de classes.

Solution :

```python
def create_confusion_matrix(data, ratio_train=0.7):
  test_part = int(ratio_train * data.shape[0])
  class_nb = np.unique(data[:, -1]).shape[0]
  k = int(np.sqrt(data.shape[0] / class_nb))  # heuristic for choosing k
  confusion_matrix = np.zeros((class_nb, class_nb))
  for x in data[test_part:, :]:
    c = int(x[-1])
    predicted = knn(data[:test_part], x[:-1], k, dist)
    confusion_matrix[c, predicted] += 1
  # optional normalization
  confusion_matrix = confusion_matrix / np.sum(confusion_matrix)
  return confusion_matrix
```

C4. Tracer la matrice de confusion à l'aide de la bibliothèque matplotlib. On pourra utiliser la fonction `matshow`.

Solution :

```python
def draw_confusion_matrix(cm, labels):
```

```
  fig = plt.figure()
  ax = fig.add_subplot(111)
  cax = ax.matshow(cm)
  plt.title('Confusion matrix of the classifier')
  fig.colorbar(cax)
  ax.set_xticklabels([''] + labels)
  ax.set_yticklabels([''] + labels)
  plt.xlabel('Predicted')
  plt.ylabel('True')
  plt.show()
```

D K-means : classification non supervisée

On ne dispose pas toujours d'un jeu de données étiquetées, c'est à dire des échantillons dont on connaît la classe. C'est pourquoi l'algorithme K-means peut être intéressant.

On considère à nouveau un problème de **classification**. On suppose qu'on dispose d'un ensemble d'échantillons $\mathcal{E} = \{e_i, i \in [\![1,n]\!], e_i \in \mathbb{R}^d\}$. Il s'agit de créer une partition de $\mathcal{E}$ selon k classes.

Algorithme 2 k moyennes (k-means)

Fonction KMEANS($\mathcal{E}$, k, δ)
  $\mathcal{P} \leftarrow \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k\}$ une partition quelconque de $\mathcal{E}$ en k classes
  tant que des échantillons changent de partition **répéter**
    $(b_1, b_2, \ldots, b_k) \leftarrow$ BARYCENTRE$(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k)$
    pour chaque échantillon e de $\mathcal{E}$ **répéter**
      Trouver la partition $\mathcal{P}_i$ la plus proche de e, $\|e - b_i\|^2$ est minimale sur $\mathcal{P}$
      Ajouter e à la partition $\mathcal{P}_i$ trouvée
  renvoyer $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k)$

Cette section n'a pas pour but d'implémenter l'algorithme d'une manière très performante. On cherche avant tout à comprendre le fonctionnement de l'algorithme.

D1. Écrire une fonction `create_partitions(data)` qui génère une liste de k partitions aléatoires non vides du jeu de données. Une partition contient uniquement les indices des échantillons.

Solution :

```
import random as rd
import numpy as np

def create_partitions(n, k):
  assert n >= k
  L = list(range(n))
  rd.shuffle(L)
  # P = [[L[i]] for i in range(k)]
  P = []
  for i in range(k):
    P.append([])
    P[i] = [L[i]]
```

```
  for i in range(k, n):
    P[rd.randrange(0, k)].append(L[i])
  return P
```

D2. Écrire une fonction `barycentres(P,data)` qui calcul les k barycentres des partitions `P` du jeu de données. Le résultat de cette fonction est un tableau Numpy de dimension (k, m) où m est le nombre de paramètres d'un échantillon [1].

Solution :

```
def barycentres(P, data):
  b = np.zeros((len(P), data[0, :].shape[0]))
  for i in range(len(P)):
    for j in P[i]:
      b[i] += data[j, :]
    b[i] /= len(P[i])
  return b
```

D3. Écrire une fonction `nearest_partition(e,B)` qui calcule l'indice de la partition dont le barycentre est le plus proche de l'échantillon `e`. `B` est la liste des barycentres obtenue à la question précédente.

Solution :

```
def nearest_partition(e, B):
  index = 0
  d = dist(e, B[0])
  for j in range(len(B)):
    if dist(e, B[j]) < d:
      index, d = j, dist(e, B[j])
  return index
```

D4. Programmer l'algorithme K-means.

Solution :

```
def kmeans(data, k):
  n = data.shape[0]
  assert n >= k
  P = create_partitions(n, k)
  while True:
    B = barycentres(P, data)
    new_P = [[] for _ in range(k)]
    for i in range(n):
      index = nearest_partition(data[i, :], B)
      new_P[index].append(i)
      if P == new_P: # No more changes -> exit !
```

1. le nombre de colonnes d'un échantillon

```
12          return P
13        else:
14          P = new_P
```

D5. Afin d'analyser les résultats, écrire une fonction `get_class(i, P)` qui permet de récupérer la classe de l'échantillon numéro `i`. Il s'agit de l'indice de la partition dont il fait partie. Si l'indice de l'échantillon n'est pas trouvé, la fonction renvoie `None`.

Solution :

```
1 def get_class(i, P):
2   for k, L in enumerate(P):
3     if i in L:
4       return k
5   return None
```

D6. Tester cet algorithme sur les deux jeux de données `diabetes.csv` et `iris.csv`.

D7. Comparer les résultats avec ceux de l'algorithme KNN en construisant les matrices de confusion ainsi que des diagrammes 2D de paramètres.

Solution : On retrouve quasiment les mêmes résultats que précédemment.

E Scikit-learn en soutien

E1. À l'aide de ce tutotiel Scikit Learn et Seaborn, analyser les jeux de données `'iris.csv'` en utilisant la fonction KNeighborsClassifier.

(a) Quel est l'intérêt du paramètre `weight` de la fonction KNeighborsClassifier ?

(b) À quoi sert le paramètre `metric` ?

(c) Quel est l'intérêt du paramètre `algorithm`. On pourra consulter ce lien !

E2. Même question avec la fonction Kmeans.

F Compresser une image avec K-means

L'algorithme des k-moyennes peut être utilisé pour compresser des images. Le principe est le suivant : plutôt que d'enregistrer toutes les nuances de couleurs dans le fichier, on va limiter le nombre de couleurs possibles et attribuer à chaque pixel une couleur proche. Cette couleur proche est obtenue en regroupant des pixels selon leur proximité de couleur grâce à l'algorithme des k-moyennes. Par exemple, on peut décider de ne coder que 64 couleurs de l'image, ces 64 couleurs étant déterminées par l'algorithme. L'image pourra être compressée davantage à cause des répétitions de couleur qu'on fera apparaître dans le fichier.

F1. À l'aide de la bibliothèque skimage et de la fonction `imread` du module `skimage.io` (cf. ici), charger l'image `'plage.jpg'` dans un tableau Numpy.

F2. À l'aide de la fonction Numpy `reshape`, transformer les données de l'image rectangulaire en un vecteur de pixels. Ce vecteur aura donc autant d'éléments qu'il y a de pixels sur l'image. C'est un vecteur de pixels, c'est à dire un vecteur de trois entiers compris entre 0 et 255.

F3. Appliquer l'algorithme des k-moyennes au vecteur de la question précédente. On choisira de créer par exemple 64 ou 32 catégories.

F4. Les valeurs des barycentres des couleurs des catégories sont accessibles par `classifier.cluster_centers_[classifier.labels_]` si `classifier` est l'objet Python généré par la fonction `KMeans`. À l'aide de ces éléments et de la fonction `reshape`, construire la nouvelle image, l'afficher à l'écran et la sauvegarder au format png.

F5. À partir de combien de couleurs votre œil perçoit-il la différence ? Observer la différence de taille des images.

Solution :

```
1
2
3   from skimage import io
4   from sklearn.cluster import KMeans
5   import numpy as np
6
7   # LOAD IMAGE
8   image = io.imread('plage.jpg')
9   io.imshow(image)
10  io.show()
11
12  # DIMENSIONS
13  rows = image.shape[0]
14  cols = image.shape[1]
15
16  # FLATTEN IMAGE
17  image = image.reshape(rows*cols, 3)
18
19  # MACHINE LEARNING
20  classifier = KMeans(n_clusters=32)
21  classifier.fit(image)
22
23  # CREATE NEW IMAGE
24  compressed_image = classifier.cluster_centers_[classifier.labels_]
25  compressed_image = np.clip(compressed_image.astype('uint8'), 0, 255)
26  compressed_image = compressed_image.reshape(rows, cols, 3)
27
28  # SAVE AND SHOW IMAGE
29  io.imsave('plage_32.png', compressed_image)
30  io.imshow(compressed_image)
31  io.show()
```

G Des manchots et des arbres

Les arbres de décision sont des algorithmes simples et puissant en apprentissage automatique. CART est un des meilleurs algorithme d'arbre de décision. Il peut être utiliser pour classer des éléments et il

est capable de gérer des paramètres d'entrées numériques continues ou discrets. On se propose donc d'utiliser cet algorithme avec Scikit sur un jeu de données qui concerne les manchots.

On peut charger ce jeu de données ainsi :

```
import seaborn as sns
df = sns.load_dataset('penguins')
```

G1. Faire afficher la description du jeu de données. Contient-il uniquement des paramètres numériques continus ?

Pour que l'algorithme puisse travailler, il est nécessaire que toutes les données soient numériques. On choisit de convertir les données textuelles en entiers à l'aide d'un encodeur ordinal comme suit :

```
    categorical_columns_selector = selector(dtype_include=object)
    categorical_columns = categorical_columns_selector(df)
    categorical_columns.pop(0) # remove species
    print(categorical_columns)
    data_categorical = df[categorical_columns]
    encoder = OrdinalEncoder()
    data_encoded = encoder.fit_transform(data_categorical)
    print(data_encoded)
    df[categorical_columns] = data_encoded
    print(df.describe())
    df=df.dropna() # remove NaN values
```

G2. Faire afficher sur un graphique la dispersion des paramètres de ce jeu de données.

G3. À l'aide de la fonction `DecisionTreeClassifier` de Scikit, entraîner un arbre de décision sur le jeu de donner. On limitera la profondeur de l'arbre à 5 niveaux.

G4. Tester l'arbre obtenu et afficher la matrice de confusion.

Il est possible de visualiser l'arbre obtenu en procédant comme suit :

```
plt.figure()
_, ax = plt.subplots(figsize=(8, 6))
_ = plot_tree(classifier,
       feature_names=['island', 'bill_length_mm', 'bill_depth_mm', '
           flipper_length_mm', 'body_mass_g', 'sex'],
       class_names=classifier.classes_,
       impurity=False, ax=ax)
plt.show()
```

Cette visualisation montre qu'un arbre de décision présente un immense avantage par rapport aux autres algorithmes : la procédure de classification est intelligible par l'être humain.

Solution :

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import OrdinalEncoder
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.compose import make_column_selector as selector

```

```
# IMPORT DATASET
from sklearn.tree import DecisionTreeClassifier, plot_tree

df = sns.load_dataset('penguins')
print(df)
print(df.describe())
df.info()

# CATEGORICAL DATA TO ORDINALS

print(df.dtypes)
categorical_columns_selector = selector(dtype_include=object)
categorical_columns = categorical_columns_selector(df)
categorical_columns.pop(0)
print(categorical_columns)
data_categorical = df[categorical_columns]
encoder = OrdinalEncoder()
data_encoded = encoder.fit_transform(data_categorical)
data_encoded[:5]
print(data_encoded)
df[categorical_columns] = data_encoded
print(df.describe())
df=df.dropna()


# Matrice de dispersion des paramètres / SCATTER MATRIX
sns.pairplot(data=df, hue='species', palette='mako')
plt.savefig('penguins_pscatter.svg')
plt.show()

# CREATE DATASETS
X = df[['island', 'bill_length_mm', 'bill_depth_mm', 'flipper_length_mm',
    'body_mass_g', 'sex']].values
y = df['species'].values
print('X -> ', X)
print('Y -> ', y)

# SPLIT DATASETS -> TRAIN/TEST
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=0)
print('Train set shape -> ', X_train.shape, y_train.shape)
print('Test set shape -> ', X_test.shape, y_test.shape)

# MACHINE LEARNING
classifier = DecisionTreeClassifier(max_depth=5)
classifier.fit(X_train, y_train)
print(classifier)

# TEST CLASSIFIER / COMPARE TRUE AND PREDICTION
predicted = classifier.predict(X_test)
print('Prediction Accuracy Score (%) :', round(accuracy_score(y_test,
    predicted) * 100, 2))

# CONFUSION MATRIX
cm = confusion_matrix(y_test, predicted)
```

```
print(cm)
labels = np.unique(df['species'].values)
ax = sns.heatmap(cm / np.sum(cm), annot=True, cmap='mako', xticklabels=
    labels, yticklabels=labels)
plt.show()

# PLOT TREE
_, ax = plt.subplots(figsize=(8, 6))
_ = plot_tree(classifier,
feature_names=['island', 'bill_length_mm', 'bill_depth_mm', '
    flipper_length_mm', 'body_mass_g', 'sex'],
class_names=classifier.classes_,
impurity=False, ax=ax)
plt.show()
```

```
# Results
Prediction Accuracy Score (%) : 98.0
[[48  0  0]
[ 1 15  0]
[ 1  0 35]]
```