K plus proches voisins*

INFORMATIQUE COMMUNE - TP n° 3.5 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ importer des données stockées depuis un fichier de type csv
- ☞ coder l'algorithme knn pour classifier des données étiquetées
- ☞ visualiser la performance de l'algorithme de classification

A Préparation des données pour la classification

On dispose de trois jeux de données étiquetées sous la forme de fichiers au format csv :

1. `iris.csv` pour la classification de variétés d'iris : iris.csv. Les étiquettes (variety) 0,1 et 2 correspondent aux variétés Setosa, Versicolor et Virginica.
2. `diabetes.csv` pour la prédiction du diabète : l'étiquette (Outcome) 1 signifie que le patient souffre de diabète, 0 qu'il n'en souffre pas.
3. `bdiag.csv` pour la prédiction de tumeurs cancéreuses : l'étiquette (diagnosis) 0 signifie que la tumeur est bénigne, 1 maligne.

Ⓡ La classe de chaque échantillon est toujours la **dernière colonne du fichier**.

A1. Coder une fonction de prototype `import_csv(filename)` où `filename` est une chaîne de caractères décrivant le nom d'un fichier. Cette fonction renvoie deux objets :

1. la liste `E` des paramètres de chaque échantillon (ligne) sous la forme d'une liste de liste de flottants,
2. la liste `C` des classes de chaque échantillon, sous la forme d'une liste d'entiers.

Vérifier que cette fonction est opérationnelle sur les deux jeux de données.

Solution :

```
def import_csv(filename):
    with open(filename, 'r') as file:
        headers = file.readline()
        lines = file.readlines()
        # on lit tout le fichier d'un seul coup.
    E = []
    C = []
```

*from scratch!

```
        for line in lines:
            data = []
            fields = line.split(',')
            for f in fields[:-1]:  # ne pas prendre la classe
                data.append(float(f))
            # data = [float(f) for f in fields[:-1]]
            C.append(int(fields[-1]))
            E.append(data)
        return E, C
```

On souhaite mélanger les données pour que les classes n'apparaissent pas regroupées. Pour cela, on peut utiliser la fonction suivante qui réunit, modifie l'ordre de deux listes simultanément puis renvoie les deux listes (conservant ainsi la cohérence des données) :

```
def mixid(E,C):
    zipped = list(zip(E, C))
    random.shuffle(zipped)
    E, C = zip(*zipped)
    E, C = list(E), list(C)
    return E,C
```

Afin de mesurer a posteriori l'efficacité du jeu de données, on crée deux jeux de données à partir des données initiales : un jeu d'entrainement et un jeu de test.

A2. Écrire une fonction de signature `split_train_test(E,C,ratio)` qui renvoie les listes :

1. `Xtrain` le jeu d'échantillons d'entrainement,
2. `Ytrain` les classes des échantillons d'entraînement,
3. `Xtests` le jeu d'échantillons de test,
4. `Ytests` les classes des échantillons de test.

Le paramètre `ratio` est un nombre compris entre 0 et 1 qui précise le rapport entre la taille des données d'entrainement et la taille du jeu de données. Par exemple, un paramètre `ratio` à 0.7 sépare le jeu de données en 70% pour l'entrainement et 30% pour les tests. On pourra utiliser le tranchage de liste (slicing).

Solution :

```
def split_train_test(E,C,ratio):
    n = len(C)
    stop = math.floor(ratio*n)
    Xtrain = E[:stop]
    Xtests = E[stop:]
    Ytrain = C[:stop]
    Ytests = C[stop:]
    return Xtrain, Ytrain, Xtests, Ytests
```

Algorithme 1 k plus proches voisins (KNN)

Fonction KNN(k, E, x, δ)
  $n \leftarrow |E|$
  $\Delta \leftarrow \emptyset$ ▷ Distances à calculer
  pour chaque échantillon étiqueté $e \in E$ **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

B KNN pour la classification

B1. Écrire une fonction de signature `de(A : list[float],B : list[float])-> float` qui renvoie la distance euclidienne entre deux vecteurs de dimension n.

Solution :

```
import math
def de(A, B):
    assert len(A) == len(B)
    n = len(A)
    d = 0
    for i in range(n):
        d += (A[i] - B[i]) ** 2
    return math.sqrt(d)
```

B2. Écrire une fonction de signature `k_plus_proches(k : int, E : list[list[float]], X : list[float])-> list[int]:` qui renvoie la liste des k plus proches voisins de X dans E. On utilisera la distance euclidienne. On peut utiliser un tri des distances et un tri des indices des échantillons conforme au tri des distances, ou bien trier directement une liste de tuples `(dist, indice_ech)` d'après le premier élément du tuple. Plusieurs solutions sont possibles :

- tri par sélection partiel : on s'arrête aux k plus petits ($O(nk)$),
- un autre tri en s'arrêtant également aux k plus petits,
- ★ utiliser un tas (--→ HORS PROGRAMME),
- modifier le tri rapide afin qu'il sélectionne les k permiers (Quick Select, $O(n)$ en moyenne).

Solution :

```
def k_plus_proches(k, E, X):
    assert len(E) > 0
    assert len(X) == len(E[0])
    n = len(E)
    D = [(de(X, E[i]), i) for i in range(len(E))]
    n = len(D)
    # Cas limites
    if k >= n or n == 0:
        return D
```

```
    if k <= 0:
        return []
    T = D[:]  # copie profonde
    # Tri sélection limité aux k premiers
    for i in range(k):
        i_min = i
        for j in range(i + 1, n):
            if T[j][0] < T[i_min][0]:
                i_min = j
        T[i], T[i_min] = T[i_min], T[i]
    return [i for d, i in T[:k]]  # Réduction indices des k premiers
```

Autre solution en utilisant le tri rapide modifié. On peut démontrer que complexité moyenne du QuickSelect est en $O(n)$. Intuitivement, si on fait l'hypothèse que le partitionnement est bon, on n'a qu'un seul appel récursif sur une taille de tableau plus faible. Le partitionnement est une opération linéaire en fonction de la taille du tableau. La récurrence de la complexité devient alors $T(n) = n + n/2 + n/4 + \cdots$ qui se résout en $O(n)$ car la série $1 + 1/2 + 1/4$ converge vers 2.

```
def quickselect(t: list[int], k: int) -> list[int]:
    if k <= 0:
        return []
    if len(t) <= k:
        return t

    # 1. Choix du pivot aléatoire (-> garantit la complexité moyenne en O(n)
       )
    pivot = random.choice(t)[0]
    # 2. Partitionnement en 3 nouvelles listes
    smaller = [x for x in t if x[0] < pivot]
    equal = [x for x in t if x[0] == pivot]
    larger = [x for x in t if x[0] > pivot]

    len_s = len(smaller)
    len_e = len(equal)

    # 3. Où chercher les éléments manquants ?
    if k <= len_s:
        # Cas 1 : Les k éléments sont tous dans la liste 'smaller'
        return quickselect(smaller, k)

    elif k <= len_s + len_e:
        # Cas 2 : On prend tout 'smaller' + une partie de 'equal'
        # (Pas besoin de récursion ici, on complète juste avec des valeurs
           égales au pivot)
        return smaller + equal[:k - len_s]

    else:
        # Cas 3 : On prend tout 'smaller', tout 'equal' + le reste dans '
           larger'
        # Le nouveau k devient : k - (ce qu'on a déjà pris)
        return smaller + equal + quickselect(larger, k - len_s - len_e)


def k_plus_proches(k, E, X):
    assert len(E) > 0
```

```
        assert len(X) == len(E[0])
        n = len(E)
        D = [(de(X, E[i]), i) for i in range(len(E))]
        R = quickselect(D, k)
        return [i for d, i in R]  # indices des k premiers
```

B3. Écrire une fonction de signature `classe_majoritaire(K : list[int], Ytrain : list[int], N : int)-> int` qui renvoie la classe majoritaire d'un échantillon X d'après la liste de ses k plus proches voisins `K`. `N` est le nombre de classes, `Ytrain` les classes des échantillons du jeu d'entrainement.

Solution :

```
def classe_majoritaire(K : list[int], Ytrain : list[int], N : int) -> int:
    # N est le nombre de classes
    h = [0 for _ in range(N)]
    for k in K:
        h[Ytrain[k]] += 1
    cmaj = 0
    vmax = h[0]
    for i in range(len(h)):
        if h[i] > vmax:
            vmax = h[i]
            cmaj = i
    return cmaj
```

B4. Écrire une fonction de signature `knn_test(filename, N, ratio)` où `N` est le nombre de classes, `ratio` la proportion de données d'entrainement. Cette fonction renvoie la matrice de confusion associée aux prédictions de KNN sur le jeu de tests.

Solution :

```
def knn_test(filename, N, ratio):
    E, C = import_csv(filename)
    E, C = mixid(E, C)
    Xtrain, Ytrain, Xtests, Ytests = split_train_test(E, C, ratio)
    n = len(Xtrain)
    k = math.floor(math.sqrt(n / N))
    confusion_matrix = [[0 for _ in range(N)] for _ in range(N)]
    for i in range(len(Xtests)):
        X = Xtests[i]
        K = k_plus_proches(k, Xtrain, X)
        cmaj = classe_majoritaire(K, Ytrain, N)
        confusion_matrix[Ytests[i]][cmaj] += 1
    return confusion_matrix
```

B5. Tester l'algorithme sur les différents jeux de données et tracer les matrices de confusion associées à l'aide du code suivant :

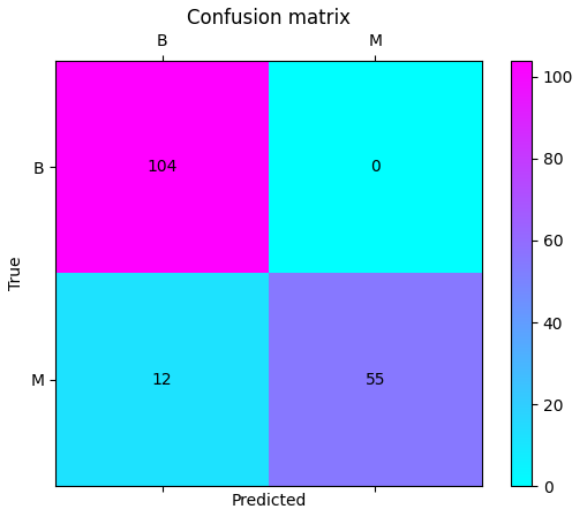

FIGURE 1 – Matrice de confusion pour la détection de tumeurs cancéreuses

```
from matplotlib import pyplot as plt

def draw_confusion_matrix(cm, classes_labels):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(cm, cmap='cool')
    plt.title('Confusion matrix')
    fig.colorbar(cax)
    ax.xaxis.set_ticks([i  for i in range(len(classes_labels))])
    ax.set_xticklabels(classes_labels)
    ax.yaxis.set_ticks([i  for i in range(len(classes_labels))])
    ax.set_yticklabels(classes_labels)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    for i in range(len(cm)):
        for j in range(len(cm[0])):
            ax.text(j, i, '{:d}'.format(cm[i][j]), ha='center', va='center')
    plt.show()
```

Vous devez obtenir des résultats proches de ceux inscrits sur la figure 1.

Solution :

```
cm = knn_test('bdiag.csv', 2, 0.7)
print(cm)
draw_confusion_matrix(cm, ['B', 'M'])

# [[98, 3], [7, 63]]
```

C KNN pour la régression

On dispose d'un jeu de données `Short_Student_Performance.csv` permettant de connaître un indice de performance (nombre flottant) d'un étudiant en fonction de paramètres tels que le nombre d'heures de travail ou le nombre d'heures de sommeil. L'objectif de cette section est de prédire la performance d'un étudiant, c'est-à-dire d'effectuer une régression.

C1. Modifier le code d'importation du fichier pour prendre en compte l'indice de performance correctement (car c'est un flottant).
Pour effectuer une régression, on ne calcule plus la classe majoritaire. À la place, on cherche à calculer la moyenne des indices de performance des k plus proches voisins.

C2. Modifier le code pour effectuer une régression sur l'indice de performance.

C3. Analyser les résultats en faisant varier la valeur de k et en calculant la racine carrée de l'erreur au carré (RMSE). Quelle valeur de k faudrait-il choisir ?

Solution :

```
def prediction(K: list[int], Ytrain: list[int]) -> int:
    m = 0
    for i in K:
        m += Ytrain[i]
    return m / len(K)

def regression(filename, k, ratio):
    E, P = import_stud(filename)
    Xtrain, Ytrain, Xtests, Ytests = split_train_test(E, P, ratio)
    n = len(Xtrain)
    Ypred = []
    for i in range(len(Xtests)):
        X = Xtests[i]
        K = k_plus_proches(k, Xtrain, X)
        p = prediction(K, Ytrain)
        #print(Ytests[i], p)
        Ypred.append(p)
    return Xtests, Ytests, Ypred

def import_stud(filename):
    file = open(filename, 'r')
    headers = file.readline()
    lines = file.readlines()
    E = []
    P = []
    for line in lines:
        current_data = []
        fields = line.split(',')
        for f in fields[:-1]:  # ne pas prendre la perf
            current_data.append(float(f))
        P.append(float(fields[-1]))
        E.append(current_data)
    return E, P

def prediction_analysis(Ytests, Ypred):
    yt = np.array(Ytests)
    yp = np.array(Ypred)
```

```
        ea = np.sqrt((yt-yp)**2)
        return np.mean(ea), np.median(ea), np.std(ea), np.min(ea), np.max(ea)
```