

## I) Préliminaires

### Exercice 1 (Fonctions auxiliaires : distance)

Nous allons coder plusieurs fonctions qui serviront pour l'algorithme kNN, mais qui ne sont pas l'algorithme kNN.

- 1) Écrire une fonction `d_carre(x, y)` qui prend en arguments deux  $d$ -uplets  $x$  et  $y$  représentant des points de  $\mathbb{R}^d$  et retourne la distance entre  $x$  et  $y$  au carré.  
*On veut minorer la distance, pour ça il est inutile de calculer une racine carrée. Ici aussi, la norme euclidienne se manipule... au carré!*
- 2) Écrire une fonction `liste_dc(x, Ly)` qui prends en arguments un  $d$ -uplet  $x$ , et une liste de  $d$ -uplets  $Ly$ . Elle retourne la liste des distances au carré : si on note  $L$  cette liste,  $L[i]$  contient  $\|x - Ly[i]\|_2^2$ .  $Ly$  est une liste de points  $y$  de  $\mathbb{R}^d$ .

### Exercice 2 (Fonctions auxiliaires : minima)

- 1) Écrire une fonction qui retourne l'indice du minimum dans une liste  $L$ .
- 2) On se donne une liste  $L$  et on considère la liste :

$$M = [(i, L[i]) \text{ for } i \text{ in range(len(L))}]$$

Écrire une fonction `tri_partiel(M, k)` qui trie  $M$  définie précédemment, par ordre croissant sur la seconde composante, en s'arrêtant après avoir trié  $k \in \mathbb{N}^*$  éléments : on se contente de placer correctement les  $k$  plus petits éléments de  $M$  (pour l'ordre sur la seconde composante).

La méthode de tri utilisé sera un tri par sélection. On pourra utiliser une version modifiée de la fonction codée à la question 1.

Si vous êtes bloqués à la dernière question, contournez la temporairement – pour la suite du TP – en utilisant l'instruction suivante pour trier  $M$  selon la seconde composante : `sorted(M, key=lambda m : m[1])`.

Ce qui consiste à trier (`sorted`) en appliquant avant la fonction (`lambda`) « prendre l'élément d'indice 1 » à chaque élément de  $M$ .

### Exercice 3 (Données d'apprentissage)

Nous allons importer les données  $\mathcal{E}$  d'apprentissage.

Dans cet exercice, ce sont des points choisis aléatoirement par python dans le carré  $[-1, 1]^2 \subset \mathbb{R}^2$ . On étiquette en rouge (noté  $r = 1$ ) les points à l'intérieur du disque unité, et en bleu (noté  $r = 0$ ) ceux à l'extérieur.

- 1) À l'aide de l'instruction « `from donnees_app_TP1 import *` », importez les fonctions contenues dans `donnees_app_TP1.py` que vous avez reçues par mail. (Vous pouvez lire les fonctions pour comprendre ce qu'elles font).
- 2) Testez les deux fonctions :
  - `Ly, labels = obtenir_disque()`
  - `affiche_donnees_disque(Ly, labels)`

## II) Algorithme kNN

### Exercice 4 (Recherche des $k$ plus proches voisins)

On utilisera les fonctions écrites dans les trois précédents exercices.

- 1) Écrire une fonction `indice_voisins(x, Ly, k)` qui prend en entrée un point  $x$ , une liste de points  $Ly$  et un entier  $k$ , et renvoie la liste des indices des  $k$  voisins de  $x$  dans  $Ly$  les plus proches. On s'aidera des fonctions `liste_dc(x, Ly)` et `tri_partiel(M, k)`.
- 2) Écrire une fonction `labels_voisins(Lindices, labels)` qui retourne la valeur la plus fréquente dans la liste des `labels[i]` pour  $i$  parcourant `Lindices`.  
On pourra utiliser un dictionnaire.

**Exercice 5** (Algorithme kNN)

Écrire une fonction `kNN(x, Ly, labels, k)` qui renvoie la valeur du label associée à  $x$  via l'algorithme des  $k$ -plus proches voisins.

**Exercice 6** (Affichage du résultat)

En balayant  $[-1, 1]$  à l'aide de `np.linspace(-1, 1, n)`, créer deux listes `Ly_essai` et `labels_essai` contenant respectivement des points de  $[-1, 1]^2$  et leur label déterminée par kNN.

Afficher le résultat (les arguments de `affiche_donnees_disque` doivent être des tableaux NumPy).

**III) Amélioration :  $\mathcal{E}_A$  et  $\mathcal{E}_T$ , tests de différents  $k$** 

Dans cette section, on cherche à tester notre algorithme sur une partie des données d'apprentissage  $\mathcal{E}$ . Il faut donc séparer ces données en deux ensemble  $\mathcal{E}_A$ , qui servira à l'apprentissage, et  $\mathcal{E}_T$ , qui servira aux tests.

**Exercice 7** (Séparation des données)

Dans l'idéal, il faut faire la liste des points pour chaque label, et prendre la même proportion de points pour  $\mathcal{E}_T$  dans chaque label.

Nous allons faire plus simple.

- 1) À l'aide de `np.random.shuffle`, mélanger la liste `list(range(len(Ly)))`, où `Ly` contient l'ensemble  $\mathcal{E}$ . C'est une fonction *en place*.
- 2) Construire une fonction `partage_donnees(Ly, labels, ratio)` qui retourne un quadruplet `(Ly_test, labels_test, Ly_app, labels_app)`, où une proportion `ratio` des données est dans `(Ly_test, labels_test)` et le reste dans `(Ly_app, labels_app)`.

**Exercice 8** (test)

Nous allons tester la qualité de l'algorithme : d'abord un pourcentage de données test bien étiquetées, puis la matrice de confusion.

- 1) Écrire une fonction `prediction(Ly_app, labels_app, Ly_test, k)`, qui retourne une liste `labels_kNN` des labels calculées par `kNN(x, Ly_app, labels_app, k)`.
- 2) Écrire une fonction `confusion(labels_test, labels_kNN)` qui calcule la matrice de confusion.
- 3) Écrire une fonction `pourcentage_reussite(labels_test, labels_kNN)` qui prend en entrée les labels connues `labels_test`, ainsi que les labels `labels_kNN` prédies par l'algorithme kNN, et qui renvoie le pourcentage de bons labels.
- 4) Modifier la fonction `obtenir_disque` du fichier `donnees_app_TP1.py` :

```
labels = [int(np.linalg.norm(x) <= 1) * (np.random.random_sample() * 10 > 1) for
x in Ly]
```

Cette instruction va rajouter du « bruit », des points bleus dans le disque rouge avec une probabilité de  $1/10$ . En détail : `np.random.random_sample()` retourne un nombre au hasard entre 0 et 1, selon une loi uniforme. Donc `(np.random.random_sample() * 10 > 1)` vaut 0 avec probabilité de  $1/10$ , et vaut 1 sinon.

Faire un graphique qui affiche le pourcentage de bons labels pour les données test en fonction de la valeur  $k$  pour l'algorithme kNN.