TD n° 5 : algorithme des k plus proches voisins

1 Les fonctions de base

Exercice 1. *Distance*

1. Écrire une fonction d'entête
   ```
   def distance2(pt1: (float, float), pt2: (float, float)) -> float:
   ```
   qui renvoie le carré de la distance entre les points du plan `pt1` et `p2` donnés par leurs coordonnées.
2. Plus généralement, écrire une fonction d'entête
   ```
   def distance(pt1: tuple, pt2: tuple) -> float:
   ```
   qui renvoie le carré de la distance entre `pt1` et `pt2` donnés par leurs coordonnées dans un espace de dimension quelconque. On pourra supposer que `pt1` et `pt2` sont des tuples de même longueur.

Exercice 2. *Déterminer les k plus proches voisins*

1. *Rappel de sup* : Écrire une fonction `tri` qui prend en argument une liste de nombres et qui trie cette liste en place, *i.e.* en la modifiant.
2. Adapter cette fonction afin d'obtenir une fonction d'entête
   ```
   def tri_couple(ensemble: [(tuple, float)]) -> None:
   ```
   qui trie en place la liste donnée en argument par ordre croissant de la valeur du flottant.
3. En déduire une fonction d'entête
   ```
   def kppv(ensemble: [tuple], point: tuple, k: int) -> [tuple]:
   ```
   où `ensemble` représente une liste de points (donnés par leurs coordonnées) et qui renvoie les `k` voisins les plus proches de `point` pour la fonction `distance`.

Exercice 3. *Classe majoritaire*

On dispose d'une liste `points` d'objets étiquetés, *i.e.* chaque élément de `points` est composé de :

— un tuple de flottants donnant ses coordonnées,

— une chaîne de caractères représentant sa classe.

1. Écrire une fonction `classe_majo` prenant en argument cette liste `points` et qui renvoie la classe la plus présente (on suppose provisoirement qu'il n'y a pas d'égalité).
2. Pour gérer les égalités, on va tirer au sort parmi les classes les plus représentées. Pour cela on utilise la fonction `choice` du module `random` dont voici l'aide :
   ```
   >>> help(random.choice)
       Choose a random element from a non-empty sequence.
   ```
   Adapter la fonction `classe_majo` précédente afin qu'elle renvoie la classe majoritaire s'il n'y en a qu'une, ou aléatoirement n'importe laquelle des classes majoritaires s'il y a des égalités.

Exercice 4. *Taux bonne prédiction*

Écrire une fonction `taux_bonne_prediction` qui renvoie le taux de bonnes prédictions et ayant pour arguments :

— une liste `donnees_test` d'entiers représentant la classe des points (on supposera que cet entier varie entre 0 et `nb_classes - 1`). Cette liste représente les données de test **étiquetées**.

— une liste `predictions` de même nature que la liste `donnees_test` mais pour laquelle l'entier représente la classe prédite par l'algorithme. Les points sont supposés dans le même ordre dans ces deux listes.

Exercice 5. *Matrice de confusion*

Écrire une fonction `mat_confusion` qui prend les mêmes arguments que la fonction de l'exercice 4 ainsi que le nombre de classes existantes et qui renvoie une liste de listes d'entiers représentant la matrice de confusion.

Exercice 6. *Analyses des résultats*

1. Écrire une fonction `taux_bonne_prediction_mat` qui prend en argument une matrice de confusion (sous forme d'une liste de listes d'entiers) et qui renvoie le taux de bonne prédiction de l'algorithme.
2. Écrire une fonction d'entête

```
taux_correct_classe(confusion: [[int]], classe: int) -> float:
```

où `confusion` est une liste de listes d'entiers représentant la matrice de confusion et `classe` le numéro d'une classe (le numéro de la ligne ou de la colonne correspond à ladite classe), et qui renvoie le taux de prédictions correctes pour les objets de cette classe.

3. ★ Écrire une fonction `taux_mauvais_classe` prenant les mêmes arguments que la fonction précédente mais qui cette fois renvoie le ratio $\frac{\text{mauvaises prédictions de } \texttt{classe}}{\text{objets pas de cette } \texttt{classe}}$. Par exemple, pour la matrice de confusion $\begin{pmatrix} 92 & 8 & 0 \\ 1 & 93 & 6 \\ 1 & 4 & 95 \end{pmatrix}$, `taux_mauvaise_classe(confusion, 0)` doit renvoyer $\frac{2}{200} = 0{,}01$.

2 Quelques améliorations

Exercice 7. ★ *Raffinement pour la classe majoritaire*
Lorsque l'on considère les voisins les plus proches, il pourrait être intéressant de pondérer le poids de chaque voisin selon sa distance avec le point à classer.

En effet, plaçons-nous dans le cas $k = 5$ et imaginons un point ayant deux voisins très proches d'une classe C_1 et trois voisins plus éloignés d'une classe C_2. Le choix de la classe majoritaire fait à l'exercice 3 conduirait à une prédiction de classe C_2 ce qui ne serait pas nécessairement satisfaisant : les voisins très proches sont probablement plus représentatifs de la classe à prédire.

On reprend les notations de l'exercice 3 à ceci près qu'on suppose maintenant que chaque élément de la liste `points` est composé en plus d'un flottant donnant la distance entre cet élément et celui que l'on cherche à classifier.

Modifier alors la fonction `classe_majo` définie à l'exercice 3 pour que le poids de chaque voisin soit inversement proportionnel à sa distance au point à classer. Par exemple, si les voisins de classe C_1 sont à distance 2 et 4 et les voisins de classes C_2 à distances 3, 6 et 6, ce sera C_1 qui sera prédite car $\frac{1}{2}+\frac{1}{4} > \frac{1}{3}+\frac{1}{6}+\frac{1}{6}$.

Exercice 8. ★ *Pas besoin de trier !*
Pour déterminer les k voisins les plus proches il n'est pas nécessaire de trier toute la liste des points par distance au point à classer, il suffit de connaître les k plus proches, l'ordre des autres n'importe pas.

Notons `ensemble` la liste des points étiquetés (donnés par leurs coordonnées), `point` le point (également donné par ses coordonnées) dont on veut déterminer les k plus proches voisins parmi ceux de `ensemble` et `distance` la fonction distance utilisée (voir exercice 1).

Pour déterminer la liste `voisins` des k plus proches voisins de `point`, on va utiliser l'idée suivante :

① `voisins` comprend au départ les k premiers éléments de `ensemble` ;

② on parcourt un à un les autres éléments `x` de `ensemble` et à chaque fois on regarde si sa distance à `point` est inférieure ou non à la distance maximale entre `point` et un élément actuel de `voisins` ;

③ si oui, on supprime l'élément de `voisins` le plus éloigné de `point` et on place `x` dans `voisins`. Sinon, on passe simplement à l'élément suivant dans `ensemble`.

L'avantage de cette méthode est d'être linéaire en la taille n de `ensemble` si k est très petit devant n (ce qui est le cas en général) alors que le tri de toute la liste est au mieux de complexité $O(n \log n)$.

Écrire une fonction d'entête :

```
def knn(ensemble: [tuple], point: tuple, k: int) -> [tuple]:
```

qui réalise l'algorithme décrit ci-dessus.