Challenge KNN :

Nous avions pour objectif de coder l'algorithme KNN (k-nearest neighbor). Cet algorithme permet d'attribuer une catégorie/espèce à une donnée quelconque en comparant ses caractéristiques à celles de ses voisins les plus proches, dont on connaît déjà la catégorie/espèce. Il repose sur la densité de répartition de différentes variables caractérisant un individu et peut être généralisé à plusieurs dimensions et plusieurs classes. Les résultats de l'algorithme dépendent fortement du paramètre k, qui correspond au nombre de voisins les plus proches considérés.

Le but de ce défi est donc de déterminer la catégorie des individus du fichier `finalTest.txt`, à partir d'une base de données fournie (`preTest.txt` + `data.txt`) et des caractéristiques d'individus de catégorie inconnue (finalTest.txt), puis de stocker les résultats dans un fichier texte séparé.

Nous avons choisi de coder en Python, car c'est un langage relativement simple et de haut niveau, qui dispose de plusieurs bibliothèques de traitement de données utiles pour la réalisation de l'algorithme. Cela facilitera le prétraitement des données et l'implémentation de l'algorithme, et c'est un langage rapide que nous maîtrisons bien.

Explication du choix utilisé dans le code des k plus proches voisins :

Nous avons fait le choix d'utiliser une classe "Individu", externe à la fonction knn afin de mieux traiter les données des fichiers textes, elle permet notamment de stocker la catégorie des individus connus ainsi que leur paramètre caractéristiques. Cette classe utilise la distance euclidienne pour mesurer les similarités entre les individus et donc la distance qui sépare 2 individus, cela permettra ensuite de pouvoir trier les individus en fonction de leur éloignement par rapport à l'individu que l'on souhaite identifier.

Ensuite, nous avons implémenté la fonction "`knn`" qui est chargée d'implémenter l'algorithme KNN en trouvant les k-plus proches voisins de l'individu inconnu, en comptant les catégories des voisins et en prédisant la catégorie majoritaire parmi les 4 plus proches voisins. Pour cela, nous avons importé la bibliothèque "collections" pour compter les nombres de chaque catégorie et alléger l'algorithme.

Nous avons également créé la fonction "`rechercher_k`" pour chercher la meilleure valeur de k à utiliser pour l'algorithme KNN en fonction d'un pourcentage de données à tester (choisies aléatoirement grâce à la bibliothèque "`random`"). Nous avons utilisé une matrice de confusion pour déterminer la valeur de k qui nous permet d'avoir le résultat le plus proche des valeurs réelles et ainsi minimiser les erreurs de notre algorithme.

En ce qui concerne la lecture des données, celles-ci sont stockées dans un fichier texte contenant une ligne pour chaque individu. Les données sont lues à partir du fichier et stockées dans une liste d'objets 'Individus' avant d'être traitées.

Enfin, pour le "`finalTest.txt`", dont nous ne connaissons pas les catégories des individus de ce fichier, nous avons fait le choix d'utiliser comme base de données les fichiers "`preTest.txt`" et "data.txt" afin d'avoir une grande base de données, sur laquelle s'appuyer pour déterminer les catégories de nos individus, puis une fois la meilleur valeur de k trouvé, on exécute l'algorithme KNN pour chacunes des valeurs en enregistrant les résultats dans un fichier texte.

Explication des tests utilisées dans l'implémentation de l'algorithme des k plus proches voisins :

Lors de l'implémentation du code afin de diminuer le nombre d'erreurs, on a régulièrement testé notre code en utilisant des prints pour déboguer et voir ce qui n'allait pas.
On a également comparé nos résultats au fichier réel afin de voir si ce qu'on obtenait était cohérent.
On a également testé la matrice de confusion en l'affichant dans la console et en regardant si la valeur de k affiché était effectivement celle que nous cherchions à avoir, c'est-à-dire celle qui minimiserait l'erreur.

Explication des idées utilisée pour l'implémentation de l'algorithme des k plus proches voisins :

On décide d'implémenter l'algorithme KNN pour la classification d'individus en fonction de leurs caractéristiques. La classe Individu est utilisée pour stocker les coordonnées de chaque individu dans un espace à 7 dimensions, ainsi que sa catégorie.
La fonction knn() prend un individu de ce test, un nombre k et une liste d'individus en entrée et renvoie sa catégorie qui est la plus fréquente parmi les k-plus proches de l'individu de test.

On utilise la fonction rechercher_k qui utilise la fonction knn pour trouver la meilleure valeur de k en fonction de k, en comparant la valeur ressortie par KNN à celle déjà connue. La fonction utilise un pourcentage pour diviser les données en un ensemble d'apprentissages et un ensemble de tests, puis calcule donc la précision de l'algorithme pour différentes valeurs de k. La valeur de k qui minimise l'erreur est choisie comme meilleure valeur de k.

Conclusion :

Le code est-il fonctionnel ?

Le code que nous avons réalisé est bel et bien fonctionnel comme en témoigne notre rendu du fichier txt sur DVO, ainsi que le retour successful du test checklabel et de la simple exécution de notre code.

Explication des choix qui nous ont amenés à la version déposée :

Dans le but d'améliorer nos résultat, plutôt que de prendre seulement la meilleur valeur de k pour déterminer la catégorie de chaque élément dans notre code knn, nous avons fait le choix d'exécuter le code pour chaque valeur de k comprise entre 1 et 10 et de stocker les résultat sous forme de liste, puis nous avons comparé les résultats pour les différentes valeurs de k, si les résultats était identique pour au moins 6 valeurs de k, alors nous avons fait le choix de choisir ce résultat, ainsi, pour chaque individu testé, on a choisi la catégorie moyenne qui lui était attribué pour k variant de 1 à 10.

Critique de notre algorithme des k plus proches voisins :

Le problème qui se pose est que chaque changement de fichier implique une adaptation de notre code. Il est ainsi nécessaire d'adapter la matrice de confusion à chaque fois, ce qui peut sembler fastidieux, ainsi que de redéfinir notre classe si le nouveau fichier à traiter à

plus ou moins de dimension, de caractéristique qui le définisse. Il faut également bien entendu modifier et adapter la lecture des données du nouveau fichier.
Enfin, le temps d'éxécution de l'algorithme peut vite de devenir long puisque notre fonction rechercher_k appelle la fonction knn qui sont 2 algorithme d'assez grosse complexité pouvant rapidement engendrer un gros temps de calcul sur une grosse base de données.