KNN vs. les Sup

On utilisera pour ce TP `Spyder`. Les données à exploiter dans ce TP sont disponible dans l'archive donnees_knn.zip.

On y trouvera 3 fichiers.

1. `data.csv` contient les données vectorisées de tous les projets musiques depuis 2023.
2. `data_training.csv` contient une sélection de 80% des données bien reparties selon les étiquettes.
3. `data_test.csv` contient les 20% restant.

On cherche ici à attribuer une note automatiquement à un projet musique. On choisi un approche par apprentissage supervisée. Les données sont étiquetées par la note sur 20 arrondi à l'entier le plus proche. Ce qui fait un total de 21 étiquettes.

Les projets on été préalablement transformé en *vecteurs* de $\mathbb{R}^{154}$ ou, pour chacune des 14 questions du projet, on a compté le nombre d'utilisations de 11 éléments structurels du code (**`if`**, **`for`**, **`while`**, listes, tuple, accès, opérations, comparaisons, return, appels, liste en compréhension). On a concaténer les 11 valeurs entières de chacune des 14 questions, puis on a normalisé le vecteur.

Ainsi les fichiers de données contiennent une ligne par projet et sur chaque ligne, séparés par des virgules :

1. Un identifiant unique,
2. L'étiquette de la donnée (note arrondie),
3. 154 floattant correspondant au projet vectorisé.

I Fonctions auxiliaires

Il s'agit d'abord d'importer les données dans le programme sous la forme de deux listes :

1. La liste des projets vectorisés (tuple de 154 flottants).
2. La liste des étiquettes de chaque projet.

De telle manière à ce que, pour le projet d'identifiant `i`, le projet vectorisé soit à l'indice `i` de `data` et son étiquette à l'indice `i` de `labels`.

On notera `vec` le type tuple de 154 flottants.

Exercice 1. *Chargement des données.*

Écrire une fonction `chargement(chemin : str) -> tuple[list, list]` qui prend en entrée le chemin d'un fichier `csv` contenant des données et qui renvoie les deux listes décritent ci-dessus.

On utilisera **`float`**`(s)` et **`int`**`(s)` pour convertir en flottant une chaîne de caractère représentant un flottant ou un entier. On utilisera `s.split(',')` pour séparer une chaîne de caractère selon les virgules et obtenir la liste des chaînes entre chaque virgule.

N.B. : cliquer ici.

Exercice 2. *Distances*

On aura besoin de la distance entre deux vecteurs de $\mathbb{R}^{154}$.

Implémenter une fonction `distance(v1 : vec, v2 : vec) -> float` qui prend deux tuples et renvoie la distance euclidienne de $\mathbb{R}^{154}$ entre ces deux vecteurs.

II KNN

L'algorithme KNN décrit ici décrit comme ceci :

```
def knn(k, strategie, donnees, etiquettes, vecteur):
    kpp = plus_proches(k, donnees, vecteur)
    return strategie(kpp, etiquettes)
```

L'algorithme prend en entrée :

1. `k`, un entier qui détermine le nombre de plus proche voisins à considérer ;
2. `strategie`, une fonction qui prend la liste des indices des `k` plus proches voisins et de leur distance ainsi que la liste des étiquettes et renvoie une étiquette ;
3. `donnees`, les projets vectorisés ;
4. `etiquettes`, les étiquettes des données ;
5. `vecteur`, le vecteur à étiqueter.

On va implémenter, la fonction `plus_proche` qui renvoie les `k` plus proches voisins de `vecteur` et trois stratégies pour choisir l'étiquette.

Exercice 3. *Plus proche*

Implémenter une fonction `plus_proche(k: int, donnees: list, vecteur : vec) -> list)` qui prend en entrée un entier `k`, une liste de vecteurs et un vecteur de l'espace et renvoie la liste des couples `[(i1,d1), ⋯, (ik,dk)]` avec `i1, ⋯, ik` les *indices* des `k` plus proches voisins, par ordre de distance et `d1, ⋯, dk` les distances de `vec` à chacun de ses voisins.

On pourra utiliser `float("inf")` pour représenter l'infini.

Exercice 4. *Stratégie majoritaire*

Implémenter une fonction `majo(kpp: list], etiquettes: list) -> int` qui renvoie l'étiquette majoritaire par rapport aux étiquettes des éléments de `kpp`. On utilisera un simple décompte des occurrences de chaque étiquettes et en cas d'égalité on prendra l'étiquette la plus grande.

Exercice 5. *Stratégie moyenne*

Implémenter une fonction `moyenne(kpp: list], etiquettes: list) -> int` qui renvoie la moyenne des étiquettes de `kpp` arrondie à l'entier le plus proche. On utilisera `round(f)` pour arrondir le flottant `f` à l'entier le plus proche.

Exercice 6. *Stratégie moyenne pondérée*

Implémenter une fonction `moyenne_ponderee(kpp: list], etiquettes: list) -> int` qui renvoie la moyenne, pondérée par la distance des voisins, des étiquettes de `kpp` arrondie à l'entier le plus proche. On utilisera `round(f)` pour arrondir le flottant `f` à l'entier le plus proche.

III Analyse des résultats

On peut dors et déjà tester notre notation automatique sur diverses stratégies, en utilisant par exemple un projet des données de tests.

```
data_tr, labels_tr = chargement('data_training.csv')
data_test, labels_test = chargement('data_test.csv')
k = 5
itest = 3
res_ia = knn(k, majo, data_tr, labels_tr, data_test[itest])
res_reel = labels_test[itest]
```

On peut ainsi commencer à comparer les résultats suivants les différentes valeurs de `k` et les différentes stratégies.

Pour choisir les meilleurs valeurs, on va construire les matrices de confusions pour chaque `0 < k < 30` et les différentes stratégies, calculer les taux d'erreurs et l'erreur quadratique moyenne.

Exercice 7. *Matrice de confusion*

Implémenter une fonction `confusion(k, strategie, d_tr, l_tr, d_test, l_test)` qui prend en entrée, une entier `k`, les données d'entraînements `d_tr`, les étiquettes des données entraînements `l_tr`, les données de test `d_test` et les étiquettes des données de test `l_test` ; et qui renvoie la matrice de confusion 21×21 de l'algorithme.

Exercice 8. *Taux d'erreur empirique*

Implémenter une fonction `taux_erreur_empirique(matrice)` qui prend en entrée une matrice de confusion et renvoie le taux d'erreur.

Exercice 9. *Erreur quadratique moyenne*

Implémenter une fonction `erreur_quad_moyenne(matrice)` qui prend en entrée une matrice de confusion et renvoie l'erreur quadratique moyenne calculée sur la matrice de confusion.

On utilisera la formule suivante adaptée au cadre de la classification mais avec des étiquettes quantitative :

$$\frac{1}{m}\sum_{i=0}^{n}\sum_{j=0}^{n} m_{i,j}(i-j)^2$$

Avec n la dimension de la matrice (nombre d'étiquettes, ici 21), et m le nombre de données d'entraînement qu'on peut obtenir directement depuis la matrice en sommant ses coefficients.

Exercice 10. *Statistiques*

Calculez les taux d'erreurs empiriques et erreurs quadratique moyenne pour toutes les valeurs de `k` entre 1 et 30 exclus, et cela pour les trois stratégies. Quelle est le meilleur choix de `k` et la meilleur stratégie ?

IV Sélection préalables des données

Dans cette section on s'intéresse à l'extraction d'un échantillon d'entraînement et d'un échantillon de test depuis l'ensemble des données initiales. On fait le choix de ne pas sélectionner 80% des données au hasard mais de d'abord réunir les données par étiquettes, puis de sélectionner 80% des données dans chaque groupe. Ainsi les données de test et d'entraînement contiennent bien assez de données de chaque étiquettes différentes.

Exercice 11. *Tri à bulles*

La sélection des 80% de données sera aléatoire. Les indices choisis seront donnés dans un ordre, lui aussi, aléatoire. Pour répartir efficacement les données une fois la sélection faîte, il faudra trier les indices choisis. On peut se contenter ici d'un tri à bulle en place vu la quantité de données par étiquettes.

Le tri à bulle en place consiste à parcourir les éléments de la liste jusqu'à l'avant dernier et d'échanger l'élément visité avec le suivant s'il est plus grand. Puis à réitérer ce parcours jusqu'à l'élément d'indice n-3, puis n-4, et ainsi de suite jusqu'à n-(n-1) (plus d'information ici).

Implémenter une fonction `tri_bulles(l : list) -> None` qui prend en entrée une liste trie sont contenu par effet de bord, sans rien renvoyer.

Exercice 12. *Sélection aléatoire*

Implémenter une fonction `sélection(l: list, p: int) -> tuple[list, list]` qui prend en entrée une liste d'éléments, `l`, et un entier, $0 \leqslant p \leqslant 100$ et qui renvoie deux listes, `select` et `nonselect` tel que `select` contient `p%` des éléments de `l` (le nombre exact d'éléments de `select` sera $\lfloor \frac{p \cdot \text{len}(l)}{100} \rfloor$), et `nonselect` contient ceux qui ne sont pas dans `select`.

Pour ce faire on pourra générer aléatoire le bon nombre d'indices de `l` aléatoire à l'aide de la fonction `sample` du module `random` (documentation). Puis, après avoir trié cette liste d'indices aléatoire avec `tri_bulles`, on peut répartir les éléments de `l` dans les deux listes en un parcours.

Exercice 13. *GROUP BY label*

Implémenter une fonction `groupby(donnees: list, etiquettes: list) -> dict[int, list]` qui prend en entrée les données et les étiquettes à exploiter et renvoie un dictionnaire dont les clés sont les étiquettes et les valeurs sont les données étiquetées par cette étiquette.

Exercice 14. *Échantillons*

On peut finir par implémenter une fonction `echantillons(donnees: list, etiquettes: list)` qui prend en entrée les données et les étiquettes à exploiter et renvoie un tuple de quatre listes, `d_tr`, `l_tr`, `d_test`, `l_test` qui contiennent respectivement : les données de l'échantillon d'entraînement, les étiquettes correspondantes, les données de l'échantillon de test et les étiquettes correspondantes.

Pour générer ces échantillons on :

1. groupe les données par étiquettes ;
2. on sélectionne 80% des données pour chaque étiquettes ;
3. on réassemble les données d'entraînements d'un coté et celles de test de l'autre.

Exercice 15. *Sauvegarde*

Sauvegarder les échantillons produits dans des fichiers pour être réutilisé plus tard. On peut ainsi essayer plusieurs échantillons pour essayer de garder le meilleur échantillon d'entraînement.