TP 20

K PLUS PROCHES VOISINS

L'idée de ce TP est de mettre en œuvre la méthode des k plus proches voisins sur deux problèmes de classification :

- l'un, ultra-classique, est la reconnaissance de chiffres manuscrits (données *MNIST*) ;
- l'autre, déjà classique bien que plus récent, est la classification d'images de vêtements (données *Fashion-MNIST*).

FIGURE 20.1 – Extrait du dataset *MNIST*.

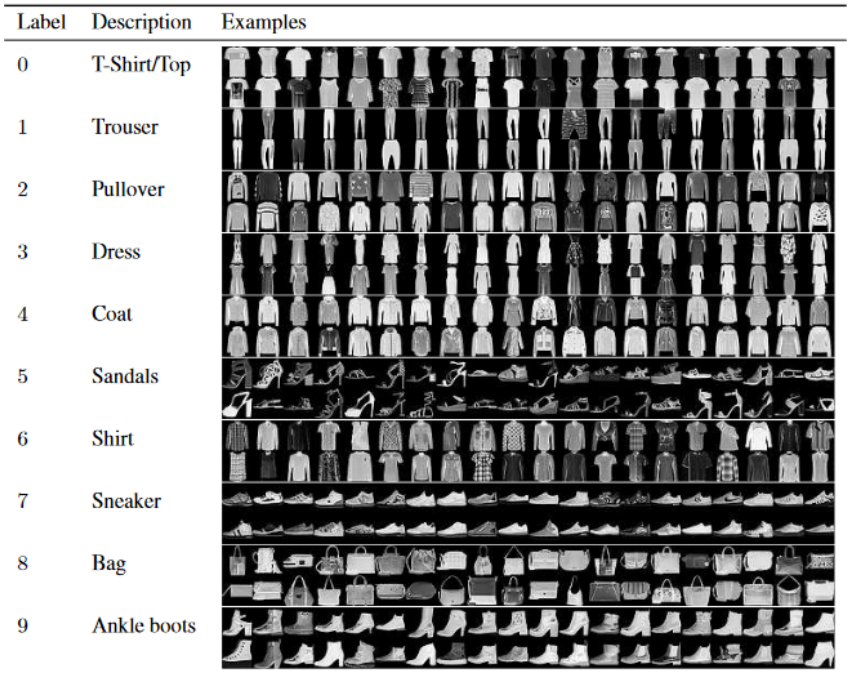

FIGURE 20.2 – Extrait du dataset *Fashion-MNIST*, avec le détail des classes.

1 Introduction

Le dossier de code qui vous est fourni est organisé comme suit.

- Le répertoire `common` contient les fichiers pour la structure du jeu de données et ses éléments, ainsi que le parser associé. Vous trouverez également des implémentations de structures déjà vues (à savoir les listes doublement chaînées et les tas-min).
  Vous n'avez pas à les modifier et vous n'avez pas besoin de les lire.
- Le répertoire `corr` contient les corrigés pour les fichiers.
  Vous n'avez pas à les modifier, ni à les lire pour le moment (il faudra le faire ensuite !).
- Dossier `data` : contient les jeux de données qui seront utilisés pour tester les programmes.
  Vous n'avez pas à les modifier.
- Le répertoire `eleves` contient les fichiers que vous avez à compléter. Il s'agit de :
  — `knn.c` pour implémenter l'algorithme des k plus proches voisins ;
  — `kdtree.c` pour implémenter les arbres d-dimensionnels.
- Le fichier `main.c` est le point d'entrée du programme, il contient la fonction `main`.
  Vous n'avez pas à le modifier ni même à le lire.
- Le fichier `test_median.c` permet de tester votre algorithme pour la médiane.
  Vous êtes encouragés à l'utiliser et, si besoin, à le modifier. Vous pouvez également vous en inspirer pour tester les autres structures.
- Le répertoire `headers` contient les fichiers d'entête pour tous les codes.

On rappelle la forme de la commande pour compiler :

```
gcc -Wall -Wextra -fsanitize=address,undefined -Iheaders [test|main].c [corr|eleve]/*.c common/*.c -lm
```

Une fois tous les *bugs* éliminés, vous pourrez éventuellement remplacer le `-fsanitize=address,undefined` par un `-O2` pour obtenir un exécutable plus rapide (le temps de calcul n'est pas négligeable dans ce TP).

On pourra lancer l'exécutable produit avec l'option `-h` pour lire la notice d'utilisation.

Pour tester ces algorithmes, deux ensembles de données vous sont fournis, avec à chaque fois :

- `..._train-images-...` : fichier avec les images d'entraînement ;
- `..._train-labels-...` : fichier avec les catégories d'entraînement (une par image) ;
- `..._t10k-images-...` : fichier avec les images de test ;
- `..._t10k-labels-...` : fichier avec les catégories de test.

Ceux commençant par *fashion* correspondent au dataset *Fashion-MNIST*, et ceux commençant par *og*, au dataset *MNIST*.
Dans les deux cas, il y a 60 000 données d'entraînement et 10 000 données de test. Chaque donnée d'entraînement consiste en une image de 28×28 pixels en nuances de gris sur 8 bits. Il y a 10 classes différentes ; pour *MNIST*, chacune correspond au chiffre associé ; pour *Fashion-MNIST*, elles sont décrites ici.

L'option `-d Num` du programme permet d'afficher `Num` données d'entraînement choisies au hasard sur le terminal.

2 L'algorithme k-NN

L'algorithme des k plus proches voisins est un algorithme de classification supervisé se basant sur le principe suivant : les objets d'une même classe sont groupés ensemble. Il y a essentiellement trois paramètres dans cet algorithme :

- la définition de la distance ;
- le nombre de voisins à considérer ;
- la pondération des contributions des k voisins.

Pour ce qui est de la distance, nous choisirons la distance euclidienne au carré (en considérant chaque image comme un grand vecteur de taille longueur fois largeur). En effet, puisque l'on considère les *plus proches voisins*, seul le comportement relatif de la distance est important.

▶ **Question 1** Implémenter dans `eleve/knn.c` la fonction `euclid_distance2` qui, étant donnés deux vecteurs d'entiers de même dimension (donnée en argument également), renvoie le carré de la distance euclidienne qui les sépare (sous forme d'entier également).

▶ **Question 2** Y a-t-il ici un risque de dépassement de capacité ?

Le nombre de voisins restera une variable dynamique que l'on donnera aux fonctions. Il faudra la faire varier pour essayer de trouver une petite valeur efficace.

▶ **Question 3** Implémenter dans `eleve/knn.c` la fonction `knn_kclosest` qui, étant donné un jeu de données, un vecteur et un nombre de voisins k, renvoie les références vers les k plus proches voisins de la valeur dans le jeu de données. On utilisera une méthode de recherche naïve (trier le jeu de donnée par distance au vecteur d'entrée, ou maintenir une liste des k plus proches et la mettre à jour au fur et à mesure de la progression dans le jeu de données).

Il reste à déterminer la méthode de pondération. On en propose deux : pondération uniforme (tous les voisins ont le même poids) et pondération par distance inverse (les voisins les plus proches ont plus de poids).

▶ **Question 4** Implémenter dans `eleve/knn.c` les fonctions `knn_majority` et `knn_weighted_majority`. La première classe le point en entrée en utilisant le même poids pour chacun des k voisins. La seconde pondère chaque voisin par l'inverse de sa distance au carré au point (plus un, pour éviter le cas pathologique de division par zéro).

3 Matrice de confusion

La matrice de confusion est un outil utile pour caractériser un algorithme de classification, en particulier ses erreurs. En notant ℓ le nombre de classes différentes (et en les numérotant de 0 à $\ell - 1$), on définit la matrice de confusion $M \in \mathcal{M}_{\ell\times\ell}(\mathcal{N})$ par :

$$M_{i,j} = \#\{\text{éléments de classe } i \text{ classés } j\}$$

On *normalise* cette matrice en divisant chaque ligne i par le nombre d'éléments du jeu de test ayant i pour classe. On obtient ainsi des coefficients dans $[0, 1]$ et la somme des valeurs d'une ligne vaut 1. Un algorithme de classification efficace aura des valeurs proches de 1 sur sa diagonale et presque nulles ailleurs, alors qu'un mauvais algorithme aura des valeurs similaires pour chaque coefficient de la matrice. L'intérêt principal de cette représentation est qu'elle permet de visualiser pour chaque classe si elle est confondue avec une autre.

▶ **Question 5** Implémenter dans `eleve/knn.c` la fonction `knn_confusion_matrix` qui, étant donné un algorithme de classification type k-NN, un jeu de données, un jeu de tests et une valeur pour k, renvoie la matrice de confusion normalisée mise à plat par lignes (c'est-à-dire que le coefficient $M_{i,j}$ se trouve à l'indice $i * \ell + j$ du tableau produit).
Conseil : afficher à intervalle régulier le nombre de cas tests qui ont été traités, l'exécution peut prendre du temps...

On peut maintenant compiler et lancer le programme pour évaluer l'efficacité de l'algorithme de classification !

▶ **Question 6** Compiler et lancer le programme. Comparer les deux variantes de k-NN. Essayer plusieurs valeurs de k.

4 k-d tree

Il est possible d'améliorer la vitesse de recherche des plus courts voisins en présentant les points du jeu de données d'entraînement de façon plus appropriée. Une structure couramment utilisée pour cela est celle d'arbre k-d (pour les vecteurs à k dimensions).

Le principe de construction est assez simple : tant qu'il reste des points, on sépare cet ensemble en deux selon une dimension i et un pivot x ; d'un côté vont tous les points p tels que $p_i \leqslant x_i$, de l'autres les p avec $p_i > x_i$. Pour s'assurer de l'équilibre de l'arbre, on choisit x tel que x_i soit la valeur médiane de tous les p_i. Et pour améliorer la répartition des points, on change la dimension utilisée pour séparer les points à chaque itération.

► **Question 7** Implémenter dans `eleve/kdtree.c` la fonction `find_median` qui, étant donné un ensemble de points, la taille de cet ensemble et une dimension, renvoie une référence vers le point médian pour cette dimension.
Cette fonction étant rappelée régulièrement lors de la création de l'arbre, il est important qu'elle soit efficace. Le choix de l'algorithme vous est laissé. On recommande cependant Quickselect, très similaire à Quicksort, linéaire en moyenne mais quadratique dans le pire des cas. Les plus courageux pourront implémenter Median Of Medians, dont le pire cas est linéaire.

► **Question 8** Lire `test_median.c` puis compiler et lancer l'exécutable produit, afin de tester votre implémentation de la médiane. On testera en particulier avec `testsize` 1,2 et 3 pour s'assurer du bon fonctionnement sur les extrêmes.

► **Question 9** Implémenter dans `eleve/kdtree.c` la fonction `kdtree_build` qui construit un arbre k-d du jeu de données fourni en argument.
Il est recommandé d'écrire une fonction auxiliaire récursive.

► **Question 10** Implémenter dans `eleve/kdtree.c` la fonction `kdtree_delete` qui détruit l'arbre k-d fourni en argument.
Attention ! Cette fonction détruit l'arbre, mais ne libère pas le jeu de donnée qu'il contient !

► **Question 11** Implémenter dans `eleve/kdtree.c` la fonction `kdtree_knn` qui, étant donné un arbre k-d, un point `p` et un nombre de voisins `t`, trouve les `t` plus proches voisins de `p` dans l'arbre.