Dans ce TP, on souhaite mettre en place des algorithmes d'apprentissage automatique dans l'objectif de faire de la reconnaissance de chiffres manuscrits. Pour ce faire, on utilisera la *MNIST handwritten digit database* (*Modified National Institute of Standards and Technology*).

Cette base de données contient des images de chiffres manuscrits étiquetés (de 0 à 9). On trouve sur le dépôt une archive `data.zip`, à télécharger et extraire dans un dossier de travail, qui contient 4 fichiers :

- un fichier contenant 60000 images de chiffres d'entraînement ;
- un fichier contenant les étiquettes de chacun des chiffres précédents ;
- la même chose pour 10000 images et étiquettes, correspondant à des données de test.

La figure 1 montre un échantillon des images disponibles dans les fichiers.

FIGURE 1 – Échantillon d'images de la base MNIST

Par ailleurs, le fichier `TP12_outils.ml` contient le code nécessaire à la lecture des fichiers précédents. On rappelle qu'on peut le charger dans un autre fichier OCaml par la commande (en indiquant le nom du chemin si nécessaire) :

```
#use "TP12_outils.ml";;
```

On peut également faire de la compilation des deux fichiers (fichier d'outils et fichier de travail), de la forme : `ocamlc TP12_outils.ml TP12.ml`.

La première ligne du fichier est à modifier éventuellement pour indiquer le chemin du dossier de travail contenant les fichiers précédents.

On représente une image par le type suivant :

```
type image = int array array
```

Ainsi, si `im` est de type `image`, alors `im` est une matrice de taille $28 \times 28$ tel que `im.(i).(j)` est la valeur du niveau de gris (entre 0, noir, et 255, blanc) du pixel ligne $i$ et colonne $j$ de l'image associée.

Un jeu de données est alors un couple formé d'un tableau d'images et d'un tableau d'étiquettes, supposés de même tailles. Dans l'ensemble du sujet, on supposera que les étiquettes sont comprises entre 0 et 9.

```
type data = image array * int array
```

La fonction `ouvrir` permet de créer un jeu de données. Dans le fichier `TP12_outils.ml`, la fonction `ouvrir` est appelée sur les deux jeux de données décrits précédemment. On obtient alors deux jeux de données :

- `train` correspondant aux images et étiquettes du jeu d'entraînement ;
- `test` correspondant aux images et étiquettes du jeu de test.

On trouve également dans ce fichier `.ml` deux fonctions d'affichage et de lecture d'image (qui seront facultatives pour travailler sur le TP, notamment s'il y a des problèmes d'installation). Des explications se trouvent dans les commentaires.

**Exercice 1**

1. Écrire une fonction `delta : image -> image -> int` qui calcule le carré de la distance euclidienne entre deux images. On considèrera une image comme un vecteur de $\mathbb{R}^{28\times 28} = \mathbb{R}^{784}$.

Le fichier `TP12-outils.ml` contient une implémentation de file de priorité de type `('a, 'b) tas` disposant des primitives suivantes :

- si `tas` est un élément de type `('a, 'b) tas`, alors `tas.taille` est le nombre d'éléments présents dans le tas ;
- `creer_tas : unit -> ('a, 'b) tas` crée un tas vide ;
- `inserer : 'a -> 'b -> ('a, 'b) tas -> unit` prend en argument une priorité $p$, un élément $x$ et un tas et insère l'élément $x$ avec priorité $p$ dans le tas ;
- `maximum : ('a, 'b) tas -> 'a * 'b` renvoie le couple $(p, x)$ tel que $x$ est l'élément de priorité $p$ maximale parmi les éléments du tas (mais sans modifier le tas) ;
- `extraire_max : ('a, 'b) tas -> 'a * 'b` extrait le couple $(p, x)$ de priorité maximale du tas et le renvoie ;
- `tas_vers_tab : ('a, 'b) tas -> ('a * 'b) array` renvoie un tableau contenant les couples (priorité, valeur) des éléments du tas, triés par priorité croissante.

2. Écrire une fonction `plus_proches_voisins : int -> data -> image -> int array` qui prend en arguments un entier `k`, un jeu de données et une image `im` et renvoie un tableau de taille $k$ contenant les étiquettes des $k$ images du jeu de données qui sont les plus proches de `im` (au sens de la distance euclidienne). On supposera que $k$ sera inférieur à la taille du jeu de données. On attend une complexité temporelle en $\mathcal{O}(N \times (d + \log k))$, où $N$ est la taille du jeu de données, $d$ la dimension des données (ici 784).
3. Écrire une fonction `majoritaire : int array -> int` qui prend en argument un tableau d'étiquettes `inds` et renvoie l'étiquette majoritaire.
4. En déduire une fonction `kppv : int -> data -> image -> int` qui prend en arguments un entier `k`, un jeu de données et une image et applique l'algorithme des $k$ plus proches voisins pour déterminer l'étiquette prédite pour l'image.
5. Écrire une fonction `matrice_confusion : int -> data -> data -> int array array` qui prend en argument un entier $k$, un jeu de données d'entraînement et jeu de données de test et crée et renvoie la matrice de confusion des données de test correspondant à l'algorithme KPPV avec les données d'entraînement.
6. Écrire de même une fonction `taux_erreur : int array array -> float` prenant en argument une matrice de confusion et renvoyant un flottant correspondant au taux d'erreurs.
7. Tester les deux fonctions précédentes avec les données fournies et une valeur de $k = 3$. Étant donnée la taille de la base de données, on utilisera des sous-tableaux (avec `Array.sub`) de taille 2000 et 400 (ou moins) respectivement pour les données d'entraînement et de test.
8. Comparer les résultats obtenus selon la valeur de $k$. Comparer aux résultats obtenus en utilisant la distance Manhattan.

**Exercice 2**

On souhaite accélérer le temps de calcul précédent. Pour ce faire, on remarque qu'une image contient une majorité de pixel noirs. Lors d'un calcul de distance, beaucoup de temps est donc passé à calculer des termes $(0-0)^2$. De plus, les chiffres étant écrits en blanc sur fond noir, on peut choisir de ne conserver que les pixels les plus lumineux. Pour améliorer le calcul, on peut déterminer, pour chaque image, la liste de ses pixels significatifs (contenant de l'information).

Pour une image $A$, on définit $S(A)$ l'ensemble de ses pixels significatifs :

$$S(A) = \{(i, j) \in [\![0, 27]\!]^2 \mid A_{i,j} > 10\}$$

On introduit un nouveau type d'image :

```
type image2 = {im : image; sign : (int * int) list}
```

tel que si `a` est un objet de type `image2`, alors `a.im` est la matrice décrivant l'image comme précédemment, et `a.sign` est la liste des éléments de $S(A)$.

Un jeu de données sera alors représenté par le type suivant, de manière similaire au type `data`.

```
type data2 = image2 array * int array
```

1. Écrire une fonction `convertir : image -> image2` qui prend en argument une image du premier type et la convertit avec le nouveau type.

Pour éviter de recalculer la liste des pixels significatifs (c'est-à-dire de forte luminosité), on créera des tableaux de listes correspondant aux listes de pixels significatifs des données d'entraînement et de test.

2. Quelle est la proportion de pixels significatifs moyenne sur les images des données d'entraînement ?

On définit une nouvelle distance entre deux images, dont le carré est donné par :

$$d_2(A,B)^2 = \sum_{(i,j)\in S(A)\cup S(B)} (A_{i,j} - B_{i,j})^2$$

3. Écrire une fonction `delta2 : image2 -> image2 -> int` qui prend en argument deux images $A$ et $B$ du nouveau type et calcule $d_2(A,B)^2$. Cette fonction devra avoir une complexité en $\mathcal{O}(|S(A)| + |S(B)|)$. En particulier, on évitera les tests d'appartenance aux listes.
4. Réécrire les fonctions de l'exercice précédent en modifiant ce qui est nécessaire pour prendre en compte ces pixels significatifs.
5. Vérifier le taux d'erreur sur le sous-jeu de données créé précédemment (au préalable converti avec le nouveau type).

On rappelle que `Sys.time ()` renvoie un flottant égal au temps en secondes depuis la création de la console/le début de l'exécution.

6. Comparer le temps de calcul du taux d'erreur selon la première et la deuxième méthode.

On souhaite à nouveau améliorer le temps de calcul. Pour cela, on remarque que pendant le calcul des plus proches voisins, si la valeur de la distance de l'image au voisin courant dépasse la distance de l'image au plus proche voisin, il n'est pas nécessaire de terminer le calcul de la distance.

7. Si on suppose que les données d'entraînements sont triées par étiquette croissante (ce qui n'est pas le cas, elles sont mélangées), quel problème pourrait se poser en appliquant la méthode décrite ci-dessus ?
8. Écrire une fonction `delta3 : image2 -> image2 -> int -> int` qui prend en argument deux images $A$ et $B$ et un seuil $s$ et calcule $d_2(A,B)^2$. Cette fonction devra arrêter les calculs dès que la distance courante dépasse $s$, et renvoyer la valeur calculée jusqu'ici.
9. Réécrire les fonctions de l'exercice précédent en modifiant ce qui est nécessaire pour prendre en compte ce seuil.
10. Comparer à nouveau le temps de calcul en incluant cette méthode.

**Remarque**

Pour l'ensemble des données d'entraînement et de test, on obtient un taux d'erreurs d'environ 3%, pour un temps de calcul d'environ 30 minutes.

Pour la suite, on souhaite créer un arbre de décision pour déterminer le chiffre associé à une image. On souhaite appliquer l'algorithme ID3.

**Exercice 3**

L'utilisation de l'algorithme ID3 n'est pas adapté à la représentation des images sous forme de vecteurs de taille 784. Pour pouvoir obtenir des résultats plus satisfaisants, il faut réduire la dimension des données. De nombreux algorithmes existent pour faire cela, mais ils font en général intervenir de l'algèbre linéaire assez complexe. On propose ici une approche plus simple ; on décrit une image selon 5 critères :

- le nombre de composantes connexes sombres (par exemple de luminosité $< 128$) ;
- la somme des luminosités dans chacun des quatre quadrants de l'image.

1. Écrire des fonctions pour calculer les valeurs de ces critères.
2. Écrire des fonctions de calcul d'entropie et de gain pour un ensemble de données et un attribut.
3. Écrire une fonction de construction d'arbre de décision selon ID3 et une fonction de recherche d'étiquette.
4. Analyser les résultats.
5. Proposer d'autres critères pour améliorer l'analyse.

**Remarque**

Avec uniquement ces 5 critères, on obtient un taux d'erreurs d'environ 40%, ce qui n'est pas très satisfaisant.