Objectifs. Les objectifs de ce TP sont de :

- réviser la manipulation de fichiers en OCaml et les fonctions des modules `Array` et `List` ;
- implémenter l'algorithme des k plus proches voisins ;
- manipuler sur un exemple concret les concepts relatifs à la classification supervisée.

Dans ce TP, on s'intéresse au problème de classification suivante : étant donnée une image, dire quel chiffre entre 0 et 9 elle représente. Les images manipulées seront des carrés de 28 pixels de côté, chaque pixel représentant une nuance de gris, entre 0 (blanc) et 255 (noir). Une image est donc un vecteur à $28 \times 28 = 784$ coordonnées (représentant les lignes mises bout à bout) ayant une valeur comprise entre 0 et 255 : c'est à partir de ce vecteur que notre fonction de classification devra décider la classe de l'image.

Préliminaires. Un fichier `tp10.zip`, disponible sur la page web du cours, contient ces données ainsi qu'un fichier `TP10_eleve.ml` pré-rempli, ainsi qu'une implémentation d'une file de priorité-max. Une fois le fichier téléchargé et dézippé, compiler les fichiers suivants avant de démarrer le TP (si vous les compilez après avoir lancer le toplevel OCaml, le toplevel ne les trouvera pas...) :

- `ocamlc graphics.mli`
- `ocamlc tasMax.mli`
- `ocamlc tasMax.ml`

Les fichiers `tasMax.ml` et `tasMax.mli` fournissent une structure de priorité-max utile pour la question 10 :

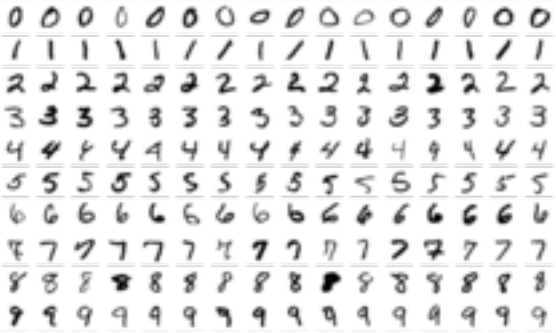
tasMax.mli

```
type ('a,'b) tasmax = { mutable nb_elem : int ; mutable tab : ('a * 'b) array }

val creer : 'a * 'b -> ('a,'b) tasmax
val taille : ('a,'b) tasmax -> int
val est_vide : ('a,'b) tasmax -> bool
val push : ('a,'b) tasmax -> 'a -> 'b -> unit
val peek : ('a,'b) tasmax -> 'a
val pop : ('a,'b) tasmax -> 'a
```

1 Lecture de données

Comme pour toute tâche de classification supervisée, il faut d'abord nous doter d'un ensemble d'apprentissage constitué d'objets pour lesquels on connaît déjà la classe. Ici, cet ensemble est fourni par la base de données **MNIST** : elle répertorie des images de chiffres manuscrits comme ceux ci-dessous pour lesquels on sait quel chiffre est censé être représenté.



Plus précisément, on travaillera dans ce TP sur :

- un jeu d'entraînement d'environ 10000 images (la base MNIST en fournit 60000) ;
- un jeu de test d'environ 1600 images (la base MNIST en fournit 10000).

Les données en question ont déjà été pré-traitées et stockées dans les fichiers suivants du dossier `data_mnist` de l'archive :

- `x_train.csv` (resp. `x_test.csv`) contient les images du jeu d'entraînement (resp. de test) au format suivant : chaque ligne correspond à une image et sur une ligne se trouvent les valeurs des pixels de l'image séparées par des virgules ;
- Le fichier `y_train.csv` (resp. `y_test.csv`) contient les classes de chaque image du jeu d'entraînement (resp. de test) au format suivant : la ligne i contient un entier entre 0 et 9 indiquant la classe de l'image en ligne i dans `x_train.csv` (resp. `x_test.csv`).

L'objectif de cette partie est de lire ces fichiers de sorte à construire des jeux d'exemples (d'entraînement ou de test) dont le type est `(image * int) array` où le type `image` est défini par :

```
type image = int array
```

Un jeu d'exemples est donc un tableau contenant en case i un couple constitué d'une image représentée par un tableau d'entiers correspondant à ses pixels et de la classe de cette image, c'est-à-dire de l'entier représenté sur l'image.
Pour répondre aux questions de cette partie, on rappelle les fonctions suivantes :

- `open_in : string -> in_channel`, prend en entrée le nom d'un fichier texte (sous forme d'une chaîne de caractères) et renvoie un **flux d'entrée** de type `in_channel` pointant vers le début du fichier ;
- `close_in : in_channel -> unit` ferme le flux d'entrée pris en argument ;
- `input_line : in_channel -> string` lit la prochaine ligne du flux d'entrée pris en argument, et la renvoie sous la forme d'une chaîne de caractère.
  Elle lève l'exception `End_of_file` si l'on a atteint la fin du fichier.

1. Écrire une fonction `lire_lignes_fichier : in_channel -> string list` prenant en arguments un flux d'entrée et renvoyant une liste de chaînes de caractères dont le i-ème élément est la chaîne correspondant à la i-ème ligne sur le flux d'entrée.

   Exemple
   ```
   # lire_lignes_fichier (open_in "test.txt") ;;
   - : string list = ["Vive"; "la"; "MPI !"; ":)"]
   ```

2. Écrire une fonction `parser_ligne_image : string -> int array` prenant en argument une chaîne de caractère représentant une image et renvoyant un tableau d'entiers correspondant aux valeurs des pixels de cette image.
   Remarque : on pourra exceptionnellement utiliser la fonction `Array.of_list` ainsi que la fonction `String.split_on_char : char -> string -> string list`.

   Exemple
   ```
   # parser_ligne_image "12,42,0,255" ;;
   - : int array = [|12; 42; 0; 255|]
   ```

3. Écrire une fonction `lire_images : string -> image array` prenant en arguments un nom de fichier contenant des images comme décrit dans le sujet, et renvoyant un tableau contenant dans sa case i la représentation OCaml de la i-ème image.
4. Écrire une fonction `lire_etiquettes : string -> int array` prenant en arguments un nom de fichier contenant des étiquettes comme décrit dans le sujet, et renvoyant un tableau contenant dans sa case i la valeur de la i-ème étiquette.
5. Écrire une fonction `jeu_donnees : image array -> int array -> (image * int) array` prenant en arguments un tableau d'images et le tableau d'étiquettes correspondantes et renvoyant le tableau associant chaque image à son étiquette.

Affichage des images. Le fichier `TP10_eleve.ml` contient une fonction `afficher : image -> unit` qui permet d'afficher un objet de type `image` dans une fenêtre graphique. Voici comme l'utiliser :

- compiler le fichier `graphics.mli` si ce n'est pas déjà fait pour que le plugin `merlin` de VSCode reconnaisse les fonctions du module `Graphics` :

```
ocamlc graphics.mli
```

- taper dans le toplevel OCaml (ne pas oublier le `#`) :

```
#require "graphics" ;;
```

- exécuter **une seule fois** la commande suivante, qui va ouvrir une fenêtre graphique blanche :

```
Graphics.open_graph " 280x280+500+500" ;;
```

- vous pouvez déplacer la fenêtre, mais **il ne faut pas la redimentionner ni la fermer** ! (le module `Graphics` est assez capricieux, et ça risque de faire complètement planter OCaml).
- un appel à la fonction `afficher` va effacer le contenu de la fenêtre graphique, et dessiner la nouvelle image dans cette même fenêtre.

Astuce. Si jamais votre toplevel OCaml vient à planter, vous pouvez en lancer un nouveau et taper les commandes suivantes pour importer d'un coup tout votre travail :

- `#require "graphics ;;` (le toplevel ouvre le module `Graphics`) ;
- `#use "TP10_eleve.ml" ;;` (le toplevel exécute tout votre fichier, et obtient donc d'un coup toutes vos fonctions).

À l'aide, le module Graphics ne fonctionne pas chez moi ! (oui, il est assez capricieux...)

- Si l'instruction `#require "graphics" ;;` fonctionne, mais que `merlin` vous souligne quand même les utilisations de `Graphics` dans votre fichier :
  ↪ une fois que la fenêtre graphique a été ouverte et que la fonction `afficher` a été envoyée au toplevel, commenter les lignes qui posent problème à `merlin`.
- Si l'instruction `#require "graphics" ;;` ne fonctionne pas, ou que vous n'arrivez pas à ouvrir la fenêtre graphique :
  ↪ commenter l'instruction `Graphics.open_graph` et la fonction `afficher`, et utiliser à la place la fonction `afficher_ascii` : elle va afficher l'image en "ASCII-art" dans la console...

2 Algorithme des k plus proches voisins

On rappelle le principe de la méthode des k plus proches voisins : pour déterminer la classe d'un objet inconnu x, calculer l'ensemble E des k objets du jeu d'entraînement les plus proches de x (selon une distance à choisir) et attribuer à cet élément la classe majoritaire dans E. Dans une tâche de classification supervisée, il y a classiquement deux étapes importantes à ne pas confondre :

- la construction d'un modèle à partir du jeu d'entraînement ;
- l'utilisation de ce modèle pour classifier un élément inconnu.

Cette deuxième étape permet d'estimer l'erreur commise par le modèle en évaluant l'erreur moyenne sur le jeu de test.

Ce qui est un peu perturbant dans la méthode des k plus proches voisins est que la construction d'un modèle selon cette méthode est triviale ; en effet elle consiste juste à stocker la valeur de k, la distance utilisée et le jeu d'entraînement. Ainsi, un modèle construit avec cette méthode aura le type suivant :

```
type modele = { distance : image -> image -> float ;
                k : int ;
                donnees : (image * int) array }
```

Pour évaluer la qualité du modèle obtenu, on observe pour chaque élément du jeu de test la classe prédite selon le principe de début de partie. L'algorithme permettant de calculer cette classe est appelé algorithme des k plus proches voisins. On se propose de l'implémenter de manière extrêmement naïve. On choisit d'utiliser la norme 1 comme distance permettant de savoir si deux images sont proches ou non.

6. Écrire une fonction `norme_1 : image -> image -> float` prenant en arguments deux images et renvoyant la distance qui les sépare selon la norme $||\ ||_1$ sous forme d'un flottant.
7. Écrire une fonction `nb_elements : int list -> int array` prenant en argument une liste ℓ d'éléments de $[\![0, 9]\!]$ et renvoyant un tableau de 10 cases contenant en case i le nombre d'éléments valant i dans ℓ.
8. Écrire une fonction `classe_plus_frequente : int list -> int` renvoyant l'un de entiers le plus fréquent dans une liste d'éléments de $[\![0, 9]\!]$.
9. Écrire une fonction :
   `construire_modele : (image -> image -> float) -> int -> (image * int) array -> modele`
   prenant en arguments une distance, un entier k et un jeu d'entraînement et qui renvoie le modèle correspondant selon la méthode des k plus proches voisins.
10. Écrire une fonction `determiner_classe : modele -> image -> int` prenant en arguments un modèle et une image, et renvoyant la classe de l'image selon le modèle.
    Remarque : puisqu'on manipule beaucoup de données, on implémentera la version avec file de priorité-max discutée en cours (qui est plus efficace si k est petit).
11. Construire le modèle pour lequel $k = 2$ et la distance est celle donnée par la norme $||\ ||_1$.
    Déterminer la classe prédite sur quelques exemples de l'ensemble de test.
    La classe prédite est-elle toujours celle attendue ?

3 Évaluation de modèles

Afin de déterminer la qualité du modèle obtenu avec $k = 2$, on va le tester sur l'entièreté de l'ensemble de test (plutôt que sur quelques exemples particuliers comme en question 11).

12. Écrire une fonction `pourcentage_erreur : modele -> (image * int) array -> float` prenant en entrée un modèle m et un jeu de test E et déterminant le pourcentage d'éléments de E qui ont été mal classifiés par m.
    Remarque : ce calcul va prendre un moment, vous pouvez utiliser des `print` pour vérifier l'avancement de votre programme.

Afin de savoir plus précisément sur quel type d'entrée notre modèle commet des erreurs, on calcule plutôt la **matrice de confusion** du modèle : c'est une matrice de taille 10×10 contenant en case (i, j) le nombre d'exemples correspondant au chiffre i mais qui sont considérés par le modèle comme faisant partie de la classe j.

13. Écrire une fonction `matrice_confusion : modele -> (image * int) array -> int array array` calculant la matrice de confusion d'un modèle sur un jeu de test.
14. Tester la fonction précédente sur le modèle construit à la question 11 et commenter les résultats obtenus.
    Remarque : ce calcul va prendre un moment, vous pouvez utiliser des `print` pour vérifier l'avancement de votre programme.
15. Calculer le pourcentage d'erreur de différents modèles construits avec la méthode des k plus proches voisins en faisant varier la distance utilisée (ou pourra essayer avec la distance euclidienne par exemple) ou le nombre k de voisins considérés pour déterminer la classe d'une image. On pourra examiner les images pour lesquelles des erreurs ont été produites et les afficher pour voir si une erreur était prévisible.