CS-107 : Mini-projet 1
Reconnaissance de l'écriture manuscrite

Vincenzo Bazzucchi, Barbara Jobstmann

-

version 1.1

Table des matières

1 Présentation

Le but de ce projet est d'implémenter et d'évaluer un programme capable de reconnaître des chiffres écrits à la main. Cette tâche de vision par ordinateur trouve de nombreuses applications pratiques dont la plus importante est sans doute la numérisation de documents. Elle peut facilement être généralisée pour reconnaître des caractères autres que des chiffres.

Pour réaliser notre objectif, nous allons utiliser un algorithme *d'apprentissage automatique* (*machine learning*) : au lieu d'expliquer à l'ordinateur comment reconnaître des chiffres au travers de règles (instructions conditionnelles `if / else`), nous lui montrerons une large collection d'images de chiffres manuscrits en lui indiquant, pour chaque image, à quel chiffre elle correspond. Ceci permettra au programme, par la suite, de traiter et d'interpréter de nouvelles images de chiffres manuscrits.

1.1 L'algorithme des K plus proches voisins

La discipline de l'apprentissage automatique se sert de connaissances mathématiques et statistiques pour exploiter les immenses bases de données que l'on désigne aujourd'hui sous le vocable de *big data*. L'idée est d'extraire des connaissances (et de « l'intelligence »[1]) non pas en enseignant aux machines des règles fixes mais en leur permettant *d'apprendre* en traitant beaucoup d'exemples.

Le traitement de bases de données si larges impose une programmation très attentive aux performances : la complexité des algorithmes et la mémoire utilisée peuvent faire toute la différence entre un programme qui s'exécute en quelques secondes ou en plusieurs heures.

Parmi les nombreuses applications de cette nouvelle discipline nous allons nous concentrer sur la **classification supervisée** : le programme recevra une image et devra lui attribuer une étiquette représentant le chiffre écrit sur l'image. L'ensemble des étiquettes est fixe et connu au départ ($\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$) et le programme apprend en recevant un ensemble d'images déjà étiquetées.

L'algorithme que nous utiliserons s'appelle **k-nearest neighbors** ou **méthode des k plus proches voisins**, abrégé KNN. Il s'agit d'un algorithme simple mais puissant qui prend en entrée une image à classifier, une valeur k et une base d'images déjà classifiées. L'algorithme commence par identifier les k images déjà classifiées qui sont les plus *proches* de l'image à classifier. Il choisira ensuite l'étiquette la plus fréquente parmi ces k images retenues[2]. Les deux éléments fondamentaux de cette méthode sont le calcul de la proximité ou *distance* entre l'image à classifier et celles déjà classifiées et l'*aggregation* permettant de choisir une étiquette dans l'ensemble des images les plus proches.

Cet algorithme permet de s'approcher au monde de l'apprentissage automatique sans avoir besoin de connaissances avancées en analyse et statistiques comme la dérivation des gradients, la théorie de l'échantillonnage, les algorithmes de dérivation automatique et l'algèbre linéaire qui caractérisent les algorithmes plus complexes comme la régression logistique ou les réseaux neuronaux (voir la section 8.2)

[1] des liens sémantiques pertinents.

[2] par exemple, si l'étiquette 2 est celle la plus fréquemment associées aux images les plus proches, alors c'est le 2 qui est choisi comme étiquette du nombre à classifier.

1.2 La base de données MNIST

L'ingrédient fondamental pour toute recette d'apprentissage automatique est l'ensemble des données qui permettent à l'algorithme d'apprendre. Dans le cadre de ce projet, nous utiliserons la base de données MNIST qui est un sous ensemble d'un « dataset » développé par le National Institute of Standards and Technology (NIST) du gouvernement des États-Unis. Il est divisé en deux parties : une partie destinée à l'apprentissage (*training*) et une partie pour tester l'efficacité des algorithmes. La séparation de la base de données en plusieurs parties est nécessaire pour évaluer les algorithmes et nous permet de mesurer comment ceux-ci se comportent face à des échantillons qu'ils n'ont pas traités en phase d'apprentissage. Il se peut en effet que l'algorithme apprenne à reconnaître très bien les images destinées à l'apprentissage mais qu'il fasse des erreurs importantes sur des images inconnues (*overfitting*).

Le sous ensemble destiné à l'apprentissage de MNIST contient 60000 images en noir et blanc ainsi que les étiquettes correspondantes alors que la partie destinée aux tests en contient 10000. Les images ont été normalisées pour avoir les mêmes dimensions et centrées. Elles sont stockées au format IDX décrit à la section 3. Nous vous fournissons quatre partitions de tailles différentes du sous ensemble destiné à l'apprentissage avec 10, 100, 1000 et 5000 images par étiquette ainsi que le dataset de test en entier avec les 10000 images.

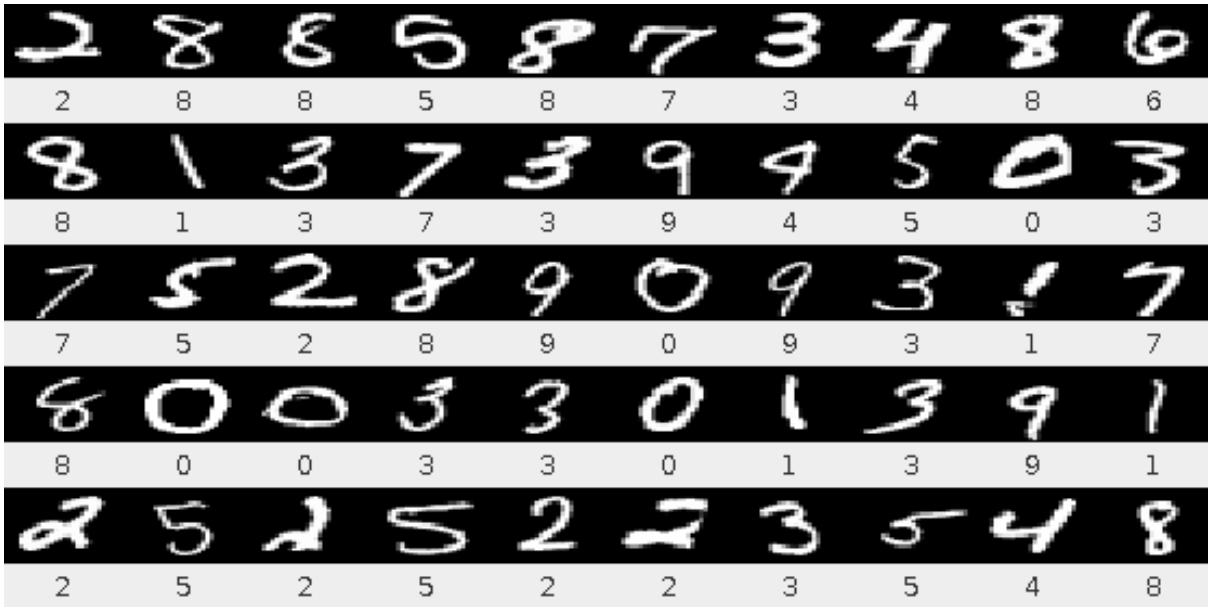

Fig. 1 : Un petit extrait de la base de données MNIST

2 Structure et code fournis

Le projet, dans sa partie obligatoire, est articulé en quatre parties :

1. La création d'un *parseur* pour le format IDX utilisé par le dataset MNIST. Le rôle du parseur est d'importer des données enregistrées dans le format IDX dans des structures de données utilisables par un développeur Java.
2. L'implémentation de deux fonctions permettant de mesurer la similarité entre une image et les images contenues dans le dataset d'apprentissage.
3. L'implémentation d'un algorithme permettant aux K images du dataset les plus proches de l'image à classifier de voter pour leur étiquette et de permettre ainsi le choix de l'étiquette ayant reçu le plus de votes.
4. L'implémentation d'outils permettant d'évaluer l'impact du choix de la valeur de K, de la taille du dataset d'apprentissage et de la fonction de distance utilisées sur la précision et les temps d'exécution du programme.

Une partie bonus facultative est également proposée pour ceux d'entre vous qui désireraient aller plus loin (voir la section 7).

Les fichiers fournis sont :

- `KNN.java` contient les méthodes à compléter pour la partie obligatoire du projet
- `KMeansClustering.java` contient les méthodes à compléter pour la partie bonus du projet
- `Helpers.java` contient des méthodes utilitaires pour la lecture des fichiers et l'affichage des données
- `SignatureChecks.java` donne l'ensemble des signatures à ne pas changer. Il sert notamment d'outil de contrôle lors des soumissions. Il permettra de faire appel à toutes les méthodes requises **sans en tester le fonctionnement**[3]. Vérifiez que ce programme compile (et uniquement qu'il compile) bien avant de soumettre. **Votre rendu sera refusé si ce fichier ne compile pas.**

Les entêtes des méthodes à implémenter sont fournies et **ne doivent pas être modifiées**.

La vérification du comportement correct de votre programme vous incombe. **Vous êtes encouragés à tester par vos propre moyens**. À cette fin, vous trouverez dans le matériel fourni, un fichier `KNNTest.java` que vous pourrez utiliser et enrichir selon vos besoins[4].

Lors de la correction de votre mini-projet, nous utiliserons des tests automatisés, qui passeront des entrées générées aléatoirement aux différentes fonctions de votre programme. Il y aura aussi des tests vérifiant comment sont gérés les cas particuliers. Ainsi, il est important que votre programme traite correctement n'importe quelle donnée d'entrée **valide**.

Pour chaque méthode à compléter, il sera spécifié si les arguments peuvent être invalides (et doivent donc de ce fait être vérifiés).

[3] Cela permet de vérifier que vos signatures sont correctes et que votre projet ne sera pas rejeté à la soumission.

[4] ceux d'entre vous qui savent ce que sont les tests unitaires peuvent en programmer, mais cela n'est évidemment pas demandé comme résultat du projet

En dehors des méthodes imposées, libre à vous de définir toute méthode supplémentaire qui vous semble pertinente.

Modularisez et tentez de produire un code propre !

La manipulation de fichiers et de fenêtres étant fastidieuse et trop avancée pour ce cours, une partie du code vous est donnée. Vos programme auront notamment à travailler avec des fichiers binaires et des données de type `byte`.

Le fichier `Helpers.java` vous simplifie l'interaction avec les fichiers binaires et l'affichage d'images :

- `byte[] readBinaryFile(String path)` ouvre le fichier se trouvant dans le chemin, `path`, fourni et retourne sont contenu en bytes.
- `void writeBinaryFile(String path, byte[] data)` ouvre le fichier se trouvant dans le chemin fourni et y écrit les bytes contenus dans `data`.
- `String byteToBinaryString(byte b)` écrit la séquence de bits constituants le byte `byte` en une chaîne de caractères.
- `byte binaryStringToByte(String bits)` crée un byte à partir de la séquence de bits reçue.
- `String interpretSigned(String bits)` retourne le byte obtenu en interprétant la séquence de bits reçue comme un nombre signé (voir 3.2).
- `String interpretUnsigned(String bits)` retourne le byte obtenu en interprétant la séquence de bits reçue comme un nombre non signé (voir 3.2).
- Plusieurs variations de la méthode `show` vous permettent de visualiser les images, les images et leurs étiquettes, les images et si les étiquettes associées sont correctes. Elles affichent une fenêtre à l'écran (dont le titre est passé en argument) et attendent que l'utilisateur la ferme pour continuer l'exécution :
  - `void show(String title, byte[][][] tensor, int rows, int columns)` affiche les images contenues dans le tenseur `tensor` dans une grille de dimension $rows \times columns$. Seules les premières $rows \times columns$ images seront lues. Le terme tenseur sera expliqué dans la section 3.3 : chaque image sera représentée utilisant un tableau bidimensionnel et un ensemble d'images sera représenté comme un tableau à trois dimensions (un tenseur). Le format de ce tenseur est expliqué en 3.4.2
  - `void show(String title, byte[][][] tensor, byte[] labels, int rows, int columns)` ajoute à la grille les étiquettes contenues dans `labels`
  - `void show(String title, byte[][][] tensor, byte[] labels, byte[] trueLabels, int rows, int columns)` produit la même fenêtre que la méthode précédente en colorant le fond de l'étiquette en vert si celle-ci est correcte (égale à celle stockée dans `trueLabels` pour la même position) ou en rouge dans le cas contraire.

Le fichier `KNNTest.java` vous en donne des exemples d'utilisation.

Les tableaux produits par les méthodes de `Helpers.java` pourront être considérés comme non nuls et correctement constitués. Il n'y a donc pas besoin de les revérifier au moment de leurs utilisations.

Nous vous invitons à utiliser les assertions Java, qu'il faut activer en lançant le programme avec l'option "-ea" [Lien cliquable ici], pour vérifier les paramètres passés à vos fonctions.

Une assertion s'écrit sous la forme

```
assert expr ;
```

avec `expr` une expression booléenne. Si l'expression est fausse, alors le programme lance une erreur et s'arrête, sinon il continue normalement.

Par exemple, pour vérifier qu'un paramètre de méthode `tensor` n'est pas `null`, vous pouvez écrire `assert tensor != null ;` au début de la méthode.

Un exemple d'utilisation d'assertion est donné dans la coquille de la méthode `parseIDXimages` dans le fichier `KNN.java`.

Il vous incombe donc de **bien vérifier à chaque étape que vous produisez bel et bien des données correctes avant de passer à l'étape suivante qui va utiliser ces données**.

Les données que nous fournissons de notre côté (images/tableaux) sont correctes.

Enfin, notez que les descriptions des étapes sont volontairement courtes et concises.

La mise en oeuvre du mini-projet implique de connaître certains concepts de base : les types primitifs de Java et une représentation binaire des données. Une partie de ces éléments vous aura été expliquée en cours. Certaines notions sont également décrites dans les compléments en fin de document.

Commencez par lire ces compléments dans les grandes lignes pour appréhender les points importants impliqués.

3 Tâche 1 : Traitement du format IDX

Dans cette première partie, vous extrairez des images et des étiquettes à partir de fichiers binaires utilisant le format IDX.

Il convient d'abord de comprendre les objectifs que visés. Vous aurez à comparer une image à classifier avec une ensemble d'images déjà classifiées (que vous extrairez depuis des fichiers binaires). L'un des problèmes principaux qui se pose alors est de comment stocker l'ensemble des images déjà classifiées. Il convient d'être très attentif au choix de cette structure de données appelée à être extrêmement volumineuse. En effet, pour pouvoir traiter des bases de données de taille importantes, l'écriture du code impose une attention particulière aux performances, en particulier à la complexité des algorithmes et à l'utilisation de la mémoire.

L'idée que vous allez mettre en oeuvre est de représenter l'ensemble des images classifiées comme un tenseur à 3 dimensions (voir le cas le plus à droite de la figure 2).

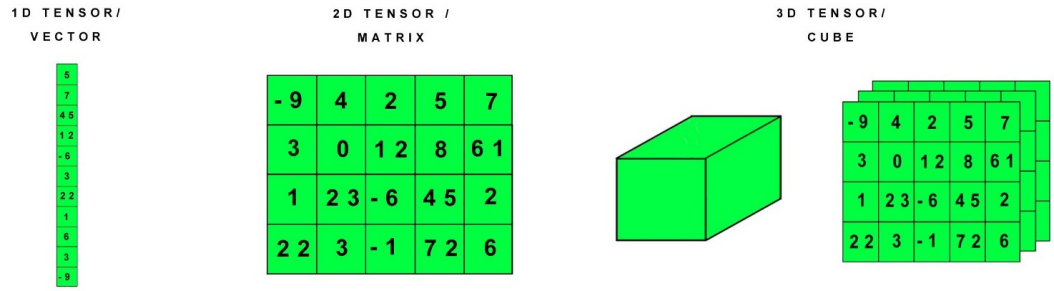

Fig. 2 : Un tenseur à trois dimensions n'est rien d'autre qu'un tableau contenant des tableaux à deux dimensions

Concrètement, il s'agira donc d'une pile d'images, telle que schématisée par la figure 3.

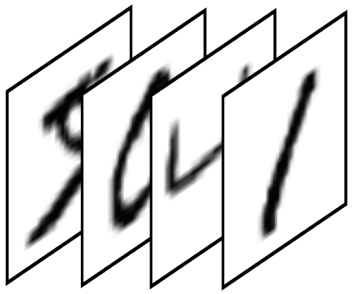

Fig. 3 : Le format utilisé pour stocker les images dans votre programme ressemble donc à une pile d'images

Une image est quant à elle simplement une matrice (tableau à deux dimensions) de pixels. Enfin, un pixel est modélisé au moyen de sa couleur.

Vous travaillerez avec des images en niveau de gris où un pixel (plus précisément sa couleur) peut être modélisé par un entier entre 0 et 255.

Comme un des objectifs est de minimiser la taille des tenseurs, il convient de choisir le type le plus économe pour modéliser la couleur d'un pixel et vous travaillerons pour cela avec le type `byte`.

Dans cette partie du projet, vous allez commencer par réviser les type primitifs de Java, dont les subtilités du type `byte`, pour ensuite définir précisément la mise en oeuvre des tenseurs (et également des ensembles d'étiquettes). Vous implémenterez ensuite le « parsing » de fichiers IDX qui vous permettra de construire les tenseurs en important des images de la base de données MNIST. Vous exploiterez aussi cette bases de données pour extraire les ensembles d'étiquettes.

3.1 Les types primitifs de Java

Dans le cadre de ce projet vous travaillerez avec les types primitifs de Java (*primitive data types*) en particulier avec les tableaux de tailles fixes et les types numériques. Les types numériques que vous utiliserez sont :

1. `byte` définit un nombre entier codé avec 8 bits dans l'intervalle $[-128, 127]$
2. `int` définit un nombre entier codé avec 32 bits dans l'intervalle $[-2^{31}, 2^{31} - 1]$
3. `float` définit un nombre à virgule flottante codé avec 32 bits selon le standard IEEE 754 [Lien cliquable ici].

Comme vous le remarquerez sur les intervalles des valeurs possibles, tous ces types sont **signés** : ils peuvent avoir des valeurs positives et négatives.

3.2 Représentation des nombres entiers en binaire

Les nombres entiers peuvent être signés et non signés.

Les nombres non signés : peuvent avoir des valeurs **uniquement positives**. Ainsi avec 8 bits, le nombre non signé u appartient à l'intervalle $u \in [0, 2^8 - 1] \Leftrightarrow u \in [0, 255]$. On définit la représentation binaire de u comme une séquence U de 8 bits en utilisant la convention *big endian* c'est-à-dire où le bit de poids le plus fort se trouve au début. Si on appelle U_i le i-ème bit de U (avec $i \in [1, 8]$) on a que

$$u = \sum_{i=1}^{8} 2^{8-i} U_i$$

Par exemple $U = 00000101 \Rightarrow u = 2^2 \cdot 1 + 2^1 \cdot 0 + 2^0 \cdot 1 = 5$ et $U = 11111010 \Rightarrow u = 250$

Les nombres signés : peuvent avoir des valeurs **positives et négatives**. Ainsi, toujours 8 bits, le nombre signé s appartient à l'intervalle $s \in [-2^7, 2^7 - 1] \Leftrightarrow s \in [-128, 127]$. En ce qui concerne les valeurs positives, leur représentation marche comme pour les valeurs non signé **codées sur 7 bits**. Normalement, pour calculer l'opposé d'un nombre signé s on calcule le ***complément à deux*** :

$$-s = \bar{s} + 1$$

où $\bar{s}$ est obtenu en inversant tous les bits de s.

Par exemple si $s = 5 \Leftrightarrow s = 00000101$ alors

$$-s = \bar{s} + 1 = \overline{00000101} + 1 = 11111010 + 1 = 11111011 = -5$$

Cela implique que la même séquence de bits peut donner deux nombres différent selon l'interprétation signée ou non signée. Le tableau 1 montre quelques séquences de 8 bits et les valeurs correspondantes en tant que nombre signé et si non signé.

Séquence de 8 bits	**Nombre non signé**	**Nombre signé**
00 00 00 00	0	0
00 00 00 01	1	1
01 11 11 11	127	127
10 00 00 00	128	-128
10 00 00 01	129	-127
11 11 11 11	255	-1

TAB. 1 : Séquences de 8 bits et leurs valeurs correspondantes selon interprétation signée (complément à deux) ou non signée.

En Java il n'existe pas de nombre non signés. De plus la manipulation du type **byte** présente quelques difficultés :

- Il n'est pas possible de créer un `byte` à partir d'une expression littérale. Par exemple `0b101` retourne un `int`
- Les opérations bit-à-bit entre `byte`s retournent des `int`s
- Les opérations entre `byte`s peuvent causer des débordements (*overflow*) et ne sont donc pas acceptées par le compilateur sans faire recours à un transtypage :

Overflow lors de la somme de deux bytes

```
byte a = 125;
byte b = 123;
byte sumClassic = a + b; // Erreur de compilation
byte sum = (byte) (a + b); // Compile
System.out.println(sum); // Affiche -8
```

Afin de contourner la dernière difficulté présentée (qui se posera dans les sections 4.1, 4.2, 7), vous pouvez utiliser un type intermédiaire capable de représenter des intervalles plus larges. Étant donné que les calculs dans ces sections vise à approximer des distances ou des moyennes, le type `float` est conseillé.

Pour pouvoir tester votre manipulation de bytes signés/non signés nous vous fournissons 4 méthodes :

```
(1) public static String byteToBinaryString(byte b)
(2) public static byte binaryStringToByte(String bits)
(3) public static String interpretUnsigned(String bits)
(4) public static String interpretSigned(String bits)
```

Un exemple d'utilisation de certaines de ces méthodes, à des fins de tests, est donné dans le programme principal fourni dans `KNN.java`.

3.3 Le format de destination

Le dataset que vous allez exploiter est composé de paires de fichiers binaires contenant respectivement les étiquettes et les images (et dont le format précis est décrit ci-dessous, voir 3.4).

La correspondance entre images et étiquettes se fait par indice : le chiffre écrit sur l'image à la position i est donné par l'étiquette enregistrée à la position i.

Votre tâche consistera à parser ces fichiers pour construire des tableaux d'étiquettes et et des tenseurs d'images exploitables dans le cadre du projet.

Les étiquettes représentant les chiffres auront une valeur entière comprise entre 0 et 9. Vous utiliserez naturellement aussi le type `byte` pour modéliser ces entiers de petite taille. Vous stockerez donc les valeurs des étiquettes dans des tableaux de tailles fixes `byte[]`.

Vos tenseurs d'images seront comme suggéré plus haut, des tableaux de taille fixe à trois dimensions `byte[][][]` stockant des (couches de) pixels.

Dans le format IDX, les valeurs des pixels des images sont des valeurs entières codées sur 8 bits entre 0 et 255, il s'agit donc de *unsigned byte*. Or comme expliqué dans 3.1, Java n'utilise que des nombres signés, ceci va donc nécessiter quelques précautions lors de la constructions des tenseurs.

3.4 Le format IDX

Cette section décrit le format des fichiers binaires contenant les étiquettes et les images.

3.4.1 Le format des fichiers contenant les étiquettes

Les 4 premiers bytes d'un fichier forment un entier appelé nombre magique (*magic number*) qui, pour les fichiers d'étiquettes, est égal à 2049. Les 4 bytes suivants forment un autre entier déterminant le nombre d'étiquettes dans le fichier. Chaque byte **non signé** qui suit est alors une étiquette.

3.4.2 Le format des fichiers contenant les images

Les 4 premiers bytes d'un fichier forment aussi le nombre magique qui, pour les fichiers d'images, est égal à 2051. Les 4 bytes suivants forment un entier déterminant le nombre d'images. Ce nombre est suivi par 4 bytes qui forment un entier indiquant la hauteur des images suivis par 4 bytes donnant la largeur des images. Après ces 4 entiers, chaque byte **non signé** est la valeur du niveau de gris d'un pixel de l'image. 0 indique un pixel noir alors que 255 indique un pixel blanc.

Les valeurs des pixels sont enregistrées consécutivement, une ligne après l'autre : la première valeur correspond au pixel haut-gauche et la dernière au pixel bas-droite.

Pour continuer à utiliser le format `byte` afin d'encombrer la moindre quantité de mémoire, il est nécessaire de convertir les bytes non signés venant des fichiers des images IDX en bytes signés **tout en préservant la signification des valeurs**. En effet, en codifiant des couleurs au moyens de bytes signés, on va utiliser l'intervalle de valeurs $[-128, 127]$, avec -128 correspondant au noir et 127 au blanc. Si l'on utilise les valeurs binaires issues du format non signé telles qu'elles en passant à un format signé, la correspondance entre l'intervalle $[-128, 127]$ et l'intervalle $[0, 255]$ pour le représentation des niveaux de gris est perdue.

Le tableau suivant illustre le problème :

Valeurs en binaire	00000000	01100100	10010110	11001000	11111111
Valeur si non signé	0	100	150	200	255
Valeur si signé	0	100	-106	-56	-1

ce qui se traduit en

La deuxième ligne, qui interprète simplement la séquence binaire d'origine comme un byte signé sans se préoccuper de préserver la signification des valeurs, ne va pas du tout reproduire le gradient de la première ligne. De plus la dernière et et première case, qui devraient être différentes, ont presque la même couleur.

Pour éviter ce problème, les valeurs des pixels seront décalées : si `p` est la valeur d'un pixel `p`$\in [0, 255]$, celle-ci sera transformée en variable de type `int` en utilisant un masque binaire, 128 sera ensuite soustrait à cette variable avant de repasser au format `byte` au moyen d'un transtypage :

Décalage de la valeur d'un pixel

```
// p = 0b10100000 0xA0, Étape par étape :
int unsigned = p & 0xFF;            // unsigned = 0xA0 = 160
int shifted = unsigned - 128;       // shifted = 32
byte pixelValue = (byte) shifted    // pixelValue = 32
// Ou en une ligne :
```

```
byte pixelValue = (byte) ((p & 0xFF) - 128);
```

Nous attirons en particulier votre attention sur la conversion de byte signé en non signé lors de la première ligne du code ci-dessus. Le `byte p = -1` serait transformé par ceci en `int unsigned = 255`.

`pixelValue` aura ainsi une valeur `pixelValue`$\in [-128, 127]$ qui peut être enregistrée dans un `byte`. -128 correspond au noir alors que 127 correspond au blanc.

Ces clarifications étant faites, nous pouvons résumer la structure de données pour stocker des ensembles d'images comme suit. Chaque image sera enregistrée dans un tableau de taille fixe à deux dimensions `byte[][]` dont la première dimension représente la hauteur et la deuxième dimension la largeur. Si on appelle `image` ce tableau, alors `image[0][0]` correspond au pixel haut-gauche de l'image.

Les images (qui auront toutes les mêmes dimensions) seront empilées dans un *tenseur* de type `byte[][][]`. Ainsi, pour une variable d'exemple `byte[][][] tensor` on aura que

- `tensor.length` est le nombre d'images
- `tensor[0].length` est la hauteur des images (le nombre de lignes)
- `tensor[0][0].length` est la largeur des images (le nombre de colonnes)

3.5 Parsing de fichiers binaires au format IDX

Votre première tâche est donc de compléter les deux méthodes suivantes qui transforment un tableau de bytes respectivement en un tableau d'étiquettes et en un tenseur d'images. Commencez par implémenter la méthode utilitaire

```
public static int extractInt(byte b31ToB24, byte b23ToB16, byte
   b15ToB8, byte b7ToB0)
```

qui assemble 4 bytes en un `int` puis les deux méthodes

```
(1) public static byte[][][] parseIDXimages(byte[] data)
(2) public static byte[] parseIDXlabels(byte[] data)
```

qui réalisent respectivement :

- l'extraction d'un tenseur d'images à partir d'un fichier binaire `data` dont le format correspond à celui décrit dans la section 3.4.2
- l'extraction d'un ensemble d'étiquettes à partir d'un fichier binaire `data` dont le format correspond à celui décrit dans la section 3.4.1

Si le nombre magique n'est pas correct, vos méthodes retourneront `null`. Vous vous aiderez de la méthode `extractInt` pour coder les traitements requis.

Le programme principal fourni dans le fichier `KNN.java` vous montre comment tester vos méthodes au fur et à mesure (par exemple comment tester la méthode `extractInt`).

Exemple de code qui lit le dataset IDX

```
// Charge les étiquettes depuis le disque
byte[] labelsRaw =
   Helpers.readBinaryFile("10_per_digit_labels_train");
// Parse les étiquettes
byte[] labelsTrain = parseIDXlabels(labelsRaw);
// Affiche le nombre de labels
System.out.println(labels.length);
// Affiche le premier label
System.out.println(labels[0]);

// Charge les images depuis le disque
byte[] imagesRaw =
   Helpers.readBinaryFile("10-per-digit_images_train");
// Parse les images
byte[][][] imagesTrain = parseIDXimages(imagesRaw);
// Affiche les dimensions des images
System.out.println("Number of images : " + images.length);
System.out.println("height : " + images[0].length);
System.out.println("width : " + images[0][0].length);

// Affiche les 30 premières images et leurs étiquettes
Helpers.show("Test", imagesTrain, labelsTrain, 2, 15);
```

4 Tâche 2 : Distance entre images

Il s'agit maintenant d'implémenter deux fonctions permettant de mesurer la distance entre deux images :

- une méthode simple permettant de ne parcourir qu'une seule fois les deux images à comparer ;
- et une seconde, plus complexe, qui nécessitera deux passes sur les deux images.

Vous étudierez, dans la dernière partie du projet, l'impact du choix entre les deux sur le temps d'exécution et sur la précision du programme.

Notez que pour enregistrer les distances nous utiliserons le type `float`. La précision de ce type est suffisante car nous ne sommes pas intéressés par la valeur exacte des distances. Les distances sont ici calculées uniquement pour être comparées, dans le but de sélectionner les k images les plus similaires. En outre les dimensions réduites de la base de données MNIST ne provoquent pas de débordement (*overflow*) lors des calculs.

4.1 Distance Euclidienne

La première méthode calcule la *distance euclidienne* : on note A la première image, B la deuxième image, H leur hauteur et W leur largeur. La distance euclidienne entre A et B, notée $E(A, B)$ est donnée par

$$E(A,B) = \sqrt{\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}[A(i,j) - B(i,j)]^2}$$

où $I(i, j)$ indique la valeur du pixel à la ligne i et à la colonne j de l'image I.

Dans le cadre de ce projet les distances sont calculées dans le but d'être comparées entre elles. Ainsi nous pouvons éviter le calcul de la racine carrée :

$$\hat{E} = E(A,B)^2 = \sum_{i=0}^{H-1}\sum_{j=0}^{W-1}[A(i,j) - B(i,j)]^2$$

Implémentez le calcul de cette valeur dans le corps de la méthode

```
public static float squaredEuclideanDistance(byte[][] a, byte[][] b)
```

Attention : N'oubliez pas la remarque sur le possible débordement lors de la somme des bytes 3.2.

Les images les plus proches d'une image donnée seront donc celles *minimisant* la distance à cette image.

4.2 Similarité inversée

La deuxième mesure de distance est obtenue à partir de la corrélation entre deux échantillons. Cette grandeur est souvent utilisée dans le monde de la vision par ordinateur. Il existe dans ce cadre une formule permettant de calculer la corrélation empirique, $p(A, B)$, entre deux images A et B. Cette formule produit des valeurs normalisées dans l'intervalle $[-1, 1]$ et c'est la valeur maximale (1) qui indique une plus grande similarité. Pour pouvoir continuer à *minimiser* les valeurs permettant de comparer deux images, vous allez plutôt travailler avec la notion de *similarité inversée*, définie comme étant $SI(A, B) = 1 - p(A, B)$ où p indique la corrélation empirique normalisée évoquée plus haut :

$$SI(A,B) = 1 - \frac{\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}(A(i,j)-\bar{A})(B(i,j)-\bar{B})}{\sqrt{\left(\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}[A(i,j)-\bar{A}]^2\right)\left(\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}[B(i,j)-\bar{B}]^2\right)}}$$

où $\bar{I}$ indique la moyenne des valeurs des pixels de l'image I : $\bar{I} = (H \times W)^{-1} \sum_{i=0}^{H-1}\sum_{j=0}^{W-1} I(i,j)$.

On remarque que $SI(A, B) \in [0, 2]$ où 0 indique que A et B sont égales et 2 indique que les deux images sont complètement différentes.

Implémentez le calcul de la similarité inversée dans le corps de la méthode

```
public static float invertedSimilarity(byte[][] a, byte[][] b)
```

Pour cela vous aurez besoin d'une méthode pour calculer la moyenne d'une image. Pour accélérer l'exécution du programme considérez la possibilité de calculer la moyenne des deux images en une seule passe avec une méthode qui itère sur les deux images et retourne les deux similarités inversées dans un tableau de taille fixe de longueur 2.

Attention : La méthode retournera 2 si le dénominateur est nul.
Attention : N'oubliez pas la remarque sur le possible débordement lors de la somme des bytes 3.2.

5 Tâche 3 : La méthode des K plus proches voisins

5.1 Aggregations des distances et choix de l'étiquette

Les deux méthodes implémentées lors de l'étape précédente permettent de calculer la distance entre l'image à classifier et toutes les images de la base d'images déjà classifiées.

Pour choisir les k images les plus proches de l'image à classifier, nous allons **trier les indices des images (et donc des étiquettes) selon la valeur croissante de la distance**. Ensuite nous pourrons choisir l'étiquette la plus répandue parmi celles des k voisins de l'image à classifier

5.1.1 Quicksort

Pour trier les indexes des images nous avons choisi un algorithme très utilisé appelé *quicksort* ou tri rapide.

L'algorithme reçoit le tableau à trier ainsi que deux indices `low` et `high` qui indiquent l'intervalle du tableau à trier. Par exemple, pour trier le tableau entier, $low = 0$ et $high =$`tableau.length`. Un élément du tableau est choisi comme *pivot* : tous les éléments plus petits que le pivot sont déplacés à sa gauche alors que les éléments plus grand à sa droite. Finalement l'algorithme s'invoque lui même (appels *récursifs*) deux fois : une fois sur la partie à gauche et une fois sur la partie à droite du pivot (donc avec le même tableau mais en modifiant les paramètres `low` et `high`)

Algorithm 1 Quicksort

Require : $array$, low, $high$
  $l \leftarrow low$
  $h \leftarrow high$
  $pivot \leftarrow$ élément du tableau à la position low
  while $l \leq h$ **do**
    if $array[l] < pivot$ **then**
      incrémente l
    else if $array[h] > pivot$ **then**
      décrémente h
    else
      Échange les éléments à la position l et h de $array$
      Incrémente l
      Décrémente h
    end if
  end while
  if $low < h$ **then**
    Appelle `quicksort`$(array, low, h)$
  end if
  if $high > l$ **then**
    Appelle `quicksort`$(array, l, high)$
  end if

Étant donné que nous sommes intéressés au tri **des indices** des images, vous apporterez les

modifications suivantes à l'algorithme :

- Il recevra un argument en plus : un tableau d'entiers contenant les indices des images, c'est-à-dire qu'au premier appel de la méthode il contiendra $0, 1, 2, 3, \ldots$
- Au moment d'échanger les éléments l, h de *array*, il échangera aussi les éléments l, h du tableau d'indices introduit au point précédent.

Implémentez la méthode utilitaire

```
public static void swap(int i, int j, float[] values, int[] indices)
```

qui échange les éléments aux positions i, j des deux tableaux et qui vous permettra d'implémenter l'algorithme décrit ci-dessus dans le corps de la méthode

```
public static void quicksortIndices(float[] values, int[] indices,
   int low, int high)
```

Complétez enfin la méthode de même nom (*surcharge*) qui reçoit uniquement le tableau de distances afin qu'elle crée le tableau d'indices et appelle l'algorithme décrit ci dessus.

```
public static int[] quicksortIndices(float[] values)
```

Le choix de trier les tableaux *in-place* c'est-à-dire sans retourner une copie triée nous évite d'allouer davantage de mémoire. Quicksort a une complexité en moyenne de $\mathcal{O}(n \log n)$.

5.1.2 Choix de l'étiquette

Après avoir trié les indices des images selon la distance de l'image à classifier, les k images les plus proches doivent voter pour leur étiquette et l'étiquette avec le plus de votes sera choisie pour l'image à classifier. Dans un premier temps vous réaliserez le décompte des votes des k images les plus proches, puis vous rechercherez l'étiquette la plus populaire.

Implémentez la recherche de l'indice de l'élément maximal d'un tableau dans le corps de la méthode utilitaire

```
public static int indexOfMax(int[] array)
```

pour ensuite implémenter la votation dans le corps de la méthode

```
public static byte electLabel(int[] sortedIndices, byte[] labels,
   int k)
```

Pour ce faire, construisez un tableau de taille 10 (une case par étiquette) et itérez sur les étiquettes des k images les plus proches de l'image à classifier pour accumuler, dans la case i du tableau, les votes pour le chiffre i. Par exemple, à la position 3 vous enregistrerez le nombre de votes pour le chiffre 3. Faites attention au type utilisé pour ce tableau. Suivant cette structure, l'étiquette la plus populaire sera donnée par l'indice de l'élément le plus grand du tableau ainsi construit.

5.2 Construction du classificateur

Toutes les méthodes nécessaires pour construire le classificateur de chiffres manuscrits sont désormais en place. Complétez donc le corps de la méthode `knnClassify` qui construit le tableau des distances entre l'image à classifier et les images déjà classifiées à l'aide d'une des deux méthodes de distance implémentées lors de l'étape 4. Ensuite, utilisez le tableau d'indices triés généré au moyen de `quicksortIndices` afin d'extraire l'étiquette retenue pour le chiffre à classifier par le biais de la méthode `electLabel`.

Vous pouvez aussi compléter la méthode `main` pour

1. Importer les images et les étiquettes d'un des datasets destinés à l'apprentissage
2. Importer les images et les étiquettes du datasets de test
3. Collectionner un certain nombre d'étiquettes générées par `knnClassify` sur les images de test dans un tableau.
4. Utiliser la méthode `Helpers.show` pour visualiser si les prédictions sont correctes.

Par exemple :

Exemple de `main`

```
int TESTS = 700;
int K = 5;
byte[][][] trainImages =
   parseIDXimages(Helpers.readBinaryFile(TRAIN_IMAGES_PATH));
byte[] trainLabels =
   parseIDXlabels(Helpers.readBinaryFile(TRAIN_LABELS_PATH));

byte[][][] testImages =
   parseIDXimages(Helpers.readBinaryFile(TEST_IMAGES_PATH));
byte[] testLabels =
   parseIDXlabels(Helpers.readBinaryFile(TEST_LABELS_PATH));

byte[] predictions = new byte[TESTS];
for (int i = 0; i < TESTS; i++) {
    predictions[i] = knnClassify(testImages[i], trainImages,
       trainLabels, K);
}

Helpers.show("Test", testImages, predictions, testLabels, 20,
   35);
```

Modifiez les valeurs de `TESTS`, celle de `K`, la taille du dataset d'apprentissage ou encore la fonction de distance et observez les résultats.

6 Tâche 4 : Évaluation

Il est impossible d'évaluer les performances du classificateur implémenté jusqu'ici en observant uniquement la fenêtre produite par `show`, et vous vous poserez sans doute des questions telles que : *Quelle combinaison de K et de taille de dataset d'apprentissage donnent la meilleure précision ? L'exécution la plus rapide ?*

Dans cette partie vous implémenterez quelques moyens simples d'étudier les performances de votre modèle d'apprentissage automatique en considérant deux critères : la **precision** et le **temps d'exécution**.

6.1 Précision

Une simple mesure de la performance peut être obtenue en calculant le rapport entre le nombre d'étiquettes correctement prédites et le nombre total de prédictions. Ce calcul produit une valeur entre 0 et 1 qui est souvent multipliée par 100 et considérée comme un pourcentage. On définit donc la precision (*accuracy*) :

$$a = n^{-1} \sum_{i}^{n} \mathbb{1}\{p_i = e_i\}$$

où p_i est le i-ème élément du vecteur d'étiquettes prédites, e_i est le i-ème élément du vecteur des vrais étiquettes et n est le nombre de prédictions. $\mathbb{1}$ est appelé fonction *indicatrice* : $\mathbb{1}\{p_i = e_i\}$ est égal à 1 si $p_i = e_i$ et 0 dans le cas contraire.

Implémentez ce calcul dans le corps de la méthode

```
public static double accuracy(byte[] predictedLabels, byte[]
    trueLabels)
```

6.2 Temps d'exécution

Pour estimer le temps d'exécution nous allons suivre une approche naïve : regarder et enregistrer l'heure avant et après l'exécution. La durée sera alors données par la différence des deux valeurs mémorisées. Java nous donne une méthode pour avoir l'heure actuelle en millisecondes : `System.currentTimeMillis()`

Modifiez le `main` du la section 6.2 pour calculer et afficher ces deux indicateurs des performances du programme :

Exemple de `main` avec mesure des performances

```
    // ... Import et parsing d'images et étiquettes

    byte[] predictions = new byte[TESTS];
    long start = System.currentTimeMillis();
    for (int i = 0; i < TESTS; i++) {
```

```
            predictions[i] = knnClassify(testImages[i], trainImages,
                trainLabels, K);
        }
        long end = System.currentTimeMillis();
        double time = (end - start) / 1000d;
        System.out.println("Accuracy = " + accuracy(predictions,
            Arrays.copyOfRange(testLabels, 0, TESTS)) + " %");
        System.out.println("Time = " + time + " seconds");
        System.out.println("Time per test image = " + (time / TESTS));
```

Modifiez les valeurs de `TESTS`, celle de `K`, la taille du dataset d'apprentissage ou encore la fonction de distance et comparez le temps d'exécution et la précision ainsi obtenus.

La figure 4 vous donne un exemple d'évaluation que nous avons obtenue en utilisant les deux critères ci-dessus (à noter que les outils pour dessiner les courbes ne sont pas fournis et il ne vous est pas demandé de générer de tels graphiques pour le rendu). Au vu du temps imparti, seule l'implémentation des mesures de précision et de temps vous est demandée.

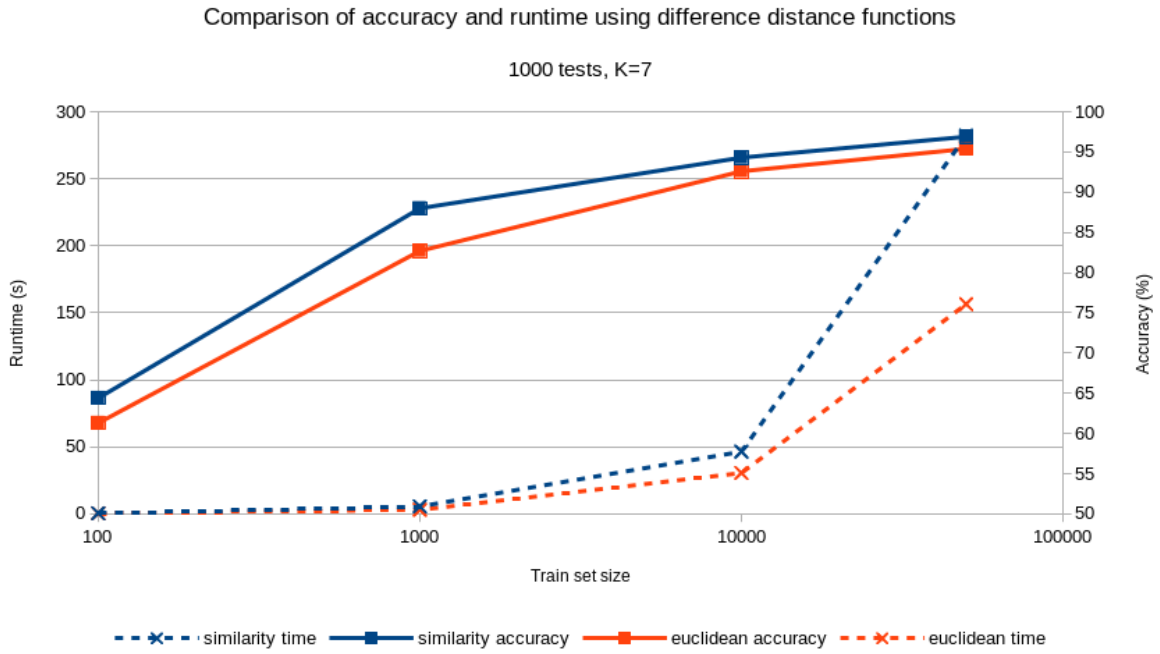

FIG. 4 : Évaluation du classificateur avec 1000 tests : les lignes continues montrent l'évolution de la précision lorsque la taille du dataset d'apprentissage varie en utilisant la distance euclidienne (ligne orange) ou la similarité inversée (ligne bleue). Les lignes pointillées montrent l'évolution du temps nécessaire pour exécuter les tests en fonction de la taille du dataset d'apprentissage en suivant le même code de couleurs. La valeur de K est fixée à 7.

7 Bonus : Réduction du dataset d'apprentissage

Vos tests devraient montrer que la meilleur précision est obtenue en utilisant la base d'apprentissage la plus grande (50000 images). Cependant cela implique un temps d'exécution plus longs.

Est-il possible d'obtenir des meilleurs résultats plus rapidement ?

Dans cette section **la partie optionnelle** du projet est décrite. L'implémentation de la partie bonus vous permet de compenser des points éventuellement perdus dans la partie obligatoire, mais la note du projet reste plafonnée à 6. La mise en oeuvre de la partie bonus implique que **vous fassiez preuve de plus d'indépendance : les algorithmes ne seront décrits que très brièvement et les assistants vous aideront beaucoup moins**. Comme lors des étapes précédentes les méthodes sont testées individuellement : vous pouvez choisir celles que vous voulez implémenter.

L'algorithme de partionnement en k-moyennes, *K-Means algorithm* en anglais, est un algorithme très simple permettant de détecter des groupes d'objets dans un espace vectoriel. Nous utiliserons le terme anglais *cluster* pour ces groupes. En outre l'algorithme calcule, pour chaque cluster son centre *centroid* qui peut être considéré comme son représentant. Voyez l'animation :

https://en.wikipedia.org/wiki/K-means_clustering#/media/File:K-means_convergence.gif

pour une visualisation du fonctionnement de l'algorithme.

L'idée de cette partie bonus est d'exécuter l'algorithme K-Means sur le dataset d'apprentissage original pour générer K représentants qui seront étiquetés et enregistrés au format IDX pour être ensuite utilisés par le code écrit lors des étapes précédentes.

7.1 L'algorithme de partitionnement en k-moyennes

L'algorithme (après initialisation) consiste en deux étapes :

1. Chaque image est assignée au cluster le plus proche. La distance entre une image et un cluster est définie comme la distance euclidienne entre l'image et le représentant du cluster.
2. Le représentant de chaque cluster est recalculé : le nouveau représentant d'un cluster est obtenu en calculant la moyenne des images qui se trouvent dans son cluster. En particulier, si un cluster contient n images I_i on a que le pixel (x, y) du nouveau représentant est donné par :
$$R(x, y) = n^{-1} \sum_{i}^{n} I_i(x, y)$$

Ces deux étapes sont exécutées jusqu'à convergence de l'algorithme c'est-à-dire jusqu'à ce que les itérations ne modifient plus ni les clusters ni leur représentant. Pour simplifier, nous exécuterons un nombre fixe d'itérations (≤ 20)

7.2 Implémentation

Les méthodes `KMeansReduce` et `initialize` sont fournies. Vous pouvez lire le corps de la méthode `KMeansReduce` : elle initialise le nécessaire pour l'algorithme et puis appelle n fois les deux étapes de l'algorithme.

Vous pouvez implémenter :

- `encodeInt` qui écrit les 4 bytes qui forment un entier dans un tableau de byte à partir d'une certaine position
- `encodeIDXimages` qui transforme un `byte[][][] tensor` en une séquence de bytes qui peuvent être enregistrés dans un fichier IDX à l'aide de la méthode `Helpers.writeBinaryFile`. N'oubliez pas d'inverser le décalage décrit lors de l'étape 3 !
- `encodeIDXlabels` qui transforme un `byte[] labels` en une séquence de bytes qui peuvent être enregistrés dans un fichier IDX à l'aide de la méthode `Helpers.writeBinaryFile`.
- `recomputeAssignments` qui implémente la première partie de l'algorithme décrit ci-dessus. Cette méthode modifie uniquement le tableau `assignment` : le nombre à la position i donne l'identifiant du cluster auquel la i-ème image du tenseur appartient. En autres termes, l'image à la position i du tenseur appartient au cluster dont le représentant se trouve à la position i du tenseur `centroids`. Cette méthode est conceptuellement simple mais elle constitue l'étape la plus coûteuse de l'algorithme, faites donc attention à votre implémentation.
- `recomputeCentroids` qui implémente la deuxième partie de l'algorithme décrite ci-dessus. Vous modifierez uniquement le tableau `centroids`. Il s'agit de la méthode la plus délicate à implémenter : vous pouvez utiliser des tableaux dynamiques `ArrayList` pour séparer les images assignées à différents clusters puis implémenter une méthode qui calcule la moyenne des images assignées à chaque cluster. Il est théoriquement possible que des clusters restent vides, c'est-à-dire qu'il existe un représentant à la position i de `centroids` tel que $i \notin$`assignments`. L'algorithme KMeans ne décrit pas une stratégie à suivre dans ces situations. Pour simplifier, si un cluster reste vide, vous ne modifierez pas son représentant.
- `main` qui utilise les méthode définies jusqu'ici pour
  1. importer les images et les étiquettes d'un dataset de test
  2. exécuter l'algorithme KMeans sur les images
  3. générer des étiquettes pour les représentants calculés par `KMeansReduce` en utilisant `knnClassify` avec la même base que celle importée lors du premier point. Les représentants sont obtenus à partir des images utilisées pour les classifier : cela nous donne une approche simple pour étiqueter les nouveaux éléments du dataset d'apprentissage sans devoir gérer les étiquettes lors de chaque étape de chaque itération de l'algorithme.
  4. enregistrer les images et et les étiquettes ainsi obtenues.

Vous pouvez tester la partie bonus en réduisant le dataset contenant 10000 images en un nouveau dataset contenant 1000 images et en utilisant `KMeans` pendant 20 itérations.

Dans nos tests, utiliser KNN avec $K = 7$ et le dataset d'apprentissage ainsi généré donne une précision de 91.5% sur 1000 tests en 5.2 secondes alors que sans réduction, pour arriver à la même précision nous avions besoin de 30 secondes !

8 Complément théorique

8.1 Représentation binaire des entiers

Référez-vous à la section 4.2 de votre livre d'ICC et/ou à la vidéo « Représentation des entiers en binaire » de Ronan Boulic (https://youtu.be/a5gLSc0tbjI), ou à votre moteur de recherche préféré, pour plus d'information sur la représentation d'un entier en binaire (complément à deux).

8.2 Références

- Algorithmes des K plus proches voisins : https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
- Tri rapide : https://fr.wikipedia.org/wiki/Tri_rapide
- Similarité (Pearson correlation coefficient) : https://en.wikipedia.org/wiki/Pearson_correlation_coefficient
- Différentiation automatique : https://en.wikipedia.org/wiki/Automatic_differentiation
- Régression logistique : https://fr.wikipedia.org/wiki/R%C3%A9gression_logistique
- Réseaux neuronaux artificiels https://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_artificiels
- Algorithme de partitionnement en k moyennes : https://en.wikipedia.org/wiki/K-means_clustering