# Rendu CLASSIFICATION SAÉ S3.02

## Groupe H3

Membres :

- Ehis Okpebho
  - JavaFX
  - Gestion des données
  - Généricité

- Thomas Dujardin
  - Chargement des données
  - Rapport

- Nathan Accart
  - Normalisation des données
  - Tests
  - Robustesse (Knn)

- Augustin Beeuwsaert
  - JavaFx

## Sommaire :

# I) Contexte :

Pendant ce semestre 3 du BUT Informatique, nous devions lors de la SAE 3.02 (2e Situation d'Apprentissage et d'Évaluation), créer une application graphique de classification avec le langage de programmation Java. Pour la partie graphique, nous avons utilisé les librairies JavaFx.

En partant d'un besoin décrit de manière imprécise ou incomplète par un client, l'objectif était de clarifier, compléter, collecter et formaliser le besoin, puis de développer une application communicante intégrant la manipulation des données tout en respectant les paradigmes de qualité (ergonomie des IHM, qualité logicielle…).

Le but de ce projet est donc de développer un outil de chargement, classification, affichage, et consultation d'un ensemble de données. Nous l'avons appelé *Classificator*.

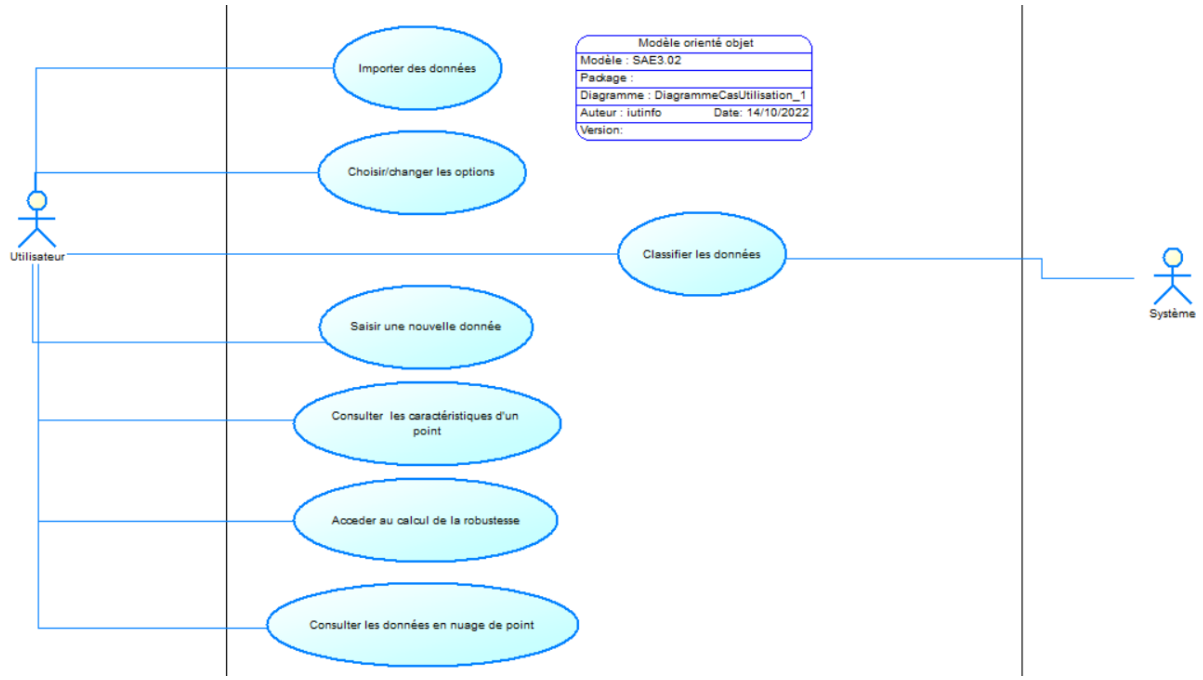
# II) Déroulement de cette SAE :

Cette SAE s'est étendue du 3 octobre au 1 décembre 2022.

Nous avons commencé par prendre en main les outils de développement et de travail en équipe. Nous avons ensuite démarré par l'analyse du sujet afin de comprendre ce qui était demandé.

Pour cela, nous avons créé le diagramme de cas d'utilisation, les fiches descriptives et le diagramme de classe.

Voici notre diagramme de cas d'utilisation :



Nous avons ensuite commencé l'implémentation des fonctionnalités, que nous allons diviser en 3 catégories : L'analyse des données, l'implémentation de k-nn et nous terminerons par la robustesse.

## A) Analyse des données

*Dans cette partie, vous devez relater tout ce qui concerne le chargement et la préparation des données pour vos deux problèmes: présentation des types de données, plages de valeurs et/ou d'énumération, détails sur les distances utilisées et la normalisation éventuellement faite.*

Nous avons eu à disposition 2 types de fichiers de données sous la forme .csv (Comma Separated Values), chaque donnée étant séparée par une virgule.

- Des iris (avec les longueurs et hauteurs des sépales et des pétales ainsi que leurs variétés qui prend 3 types possibles).

  Voici un extrait du fichier iris.csv :

| | sepal.length | sepal.width | petal.length | petal.width | variety |
|---|---|---|---|---|---|
| 2 | 5.1 | 3.5 | 1.4 | .2 | Setosa |
| 3 | 4.9 | 3 | 1.4 | .2 | Setosa |
| 4 | 4.7 | 3.2 | 1.3 | .2 | Setosa |
| 5 | 4.6 | 3.1 | 1.5 | .2 | Setosa |

- Des passagers du Titanic (avec l'identifiant, s'il a survie, sa classe à bord, son nom, son sexe, son âge, son sibsp, son parch, son numéro de ticket et son prix, son nom de cabine (si existe) et s'il a embarqué.

  Voici un extrait du fichier titanic.csv :

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22 | 1 | 0 | A/5 21171 | 7.25 | | S |
| 3 | 2 | 1 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Thayer) | female | 38 | 1 | 0 | PC 17599 | 71.2833 | C85 | C |
| 4 | 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26 | 0 | 0 | STON/O2.3101282 | 7.925 | | S |
| 5 | 4 | 1 | 1 | Futrelle, Mrs. Jacques Heath (Lily May Peel) | female | 35 | 1 | 0 | 113803 | 53.1 | C123 | S |

La première problématique que nous nous sommes posée était l'importation des données. Nous avons utilisé la librairie *opencv* et créé une classe pour chaque fichier csv.
Un fichier d'importation est caractérisé par toutes les colonnes du fichiers avec ses Getters et ses Setters correspondant afin de créer une instance.

Pour l'importation des données, nous avons choisi de vérifier que la donnée était du bon type (un nombre est bien un nombre etc…) avant de la mettre dans l'instanciation d'un élément.

Nous avons créé une interface IPoint permettant de récupérer la valeur d'une colonne passée en paramètre et de regrouper toutes les structures de données importables.

Nous avons regroupé tous les types de valeur numérique (double, float, integer) en number pour ne pas avoir à se soucier de leur type et ne créer qu'une colonne qui les regroupe.

Pour les chaînes de caractères, nous avons regardé dans le fichier de données si une énumération pourrait apparaître (quelques valeurs différentes répétées plusieurs fois).

Nous avons créé une énumération interne dans la classe de chaque import.

Une fois cela importé, nous avons pu commencer à réfléchir à la normalisation.
Elle consiste à faire apparaître les valeurs entre 0 et 1 (pour éviter d'avoir des écarts trop importants). Chaque valeur par exemple est normalisée.

La normalisation s'effectue avec une formule différente selon le type de valeur :

- Pour les données numériques : nous prenons la valeur normalisée à laquelle nous soustrayons la valeur minimale de sa colonne et nous divisons ce résultat par la différence entre la valeur maximale et la valeur minimale de sa colonne. Nous obtenons alors un nombre entre 0 et 1.

  $$\Rightarrow \frac{X - MIN}{MAX - MIN}$$

- Pour les variables ordinales (une énumération par exemple) : nous prenons le rang de cette variable -1 à laquelle nous divisons par le nombre de variable -1.
  Par exemple, nous avons l'énumération suivante: petit, moyen, grand. La normalisation de l'attribut moyen sera : 2-1 / 3-1 = 0,5.

  $$\Rightarrow \frac{X - 1}{NBTotal - 1}$$

## B) Implémentation de k-NN

k-nn est un algorithme qui permet de savoir les k plus proches voisins. K étant le nombre de voisins recherchés à partir d'un point.

Nous avons créé une fonction getNeighbours qui retourne une liste des k plus proches voisins et qui prend en paramètre le point, le nombre de voisins à avoir, la distance et le model qui contient la liste des points.

Le calcul de distance peut se faire par 2 formules, celle d'Euclidienne ou celle de Manhattan.

- La distance euclidienne est la racine carrée de la différence entre les 2 points mise au carré.

  $$\Rightarrow \sqrt{(x-y)^2}$$

- La distance manhattan est la différence entre les 2 points.

  $$\Rightarrow (x-y)$$

La règle de distance est différente selon le type de donnée :

- Si c'est une chaîne de caractères et si les 2 valeurs sont égales, alors la distance est de 0 sinon si ce n'est pas égal, la distance est égale à 1.
- Si c'est une énumération : la distance est égale à la différence de rang.

- Si c'est un nombre : la distance est calculée soit avec la distance euclidienne soit avec la distance de Manhattan.

Nous calculons uniquement avec les valeurs normalisées (pour éviter d'avoir un écart trop important des valeurs).

Nous avons créé cette fonction générique afin qu'elle puisse fonctionner avec tout type de donnée.

Nous avons également créé les fonctions "removeElement" et "addElement":
- removeElement prend en paramètre une liste et 2 nombres (nombre début et nombre fin). Elle permet de retirer les éléments de la liste ayant leurs index compris entre les 2 nombres.
- addElement prend en paramètre 2 listes. Elle va prendre chaque élément d'une liste pour les ajouter dans une autre.

Elles sont appelées lors du calcul de robustesse.

La fonction "Classifier" qui est également appelée lors de la robustesse prend en paramètre une liste de points et va retourner celle qui apparaît le plus. Avec une HashMap <String, Integer>, elle va tout d'abord déterminer le nombre d'apparition de chaque catégorie. Pour chaque point de la liste, on regarde cette catégorie, si elle est déjà dans la HashMap on incrémente sa valeur associée, sinon on l'ajoute (initialisé à 0) puis on incrémente.
Elle parcourt ensuite la map pour retourner la catégorie qui est apparue le plus.

## C) Robustesse de nos modèles

Le calcul de la robustesse permet de savoir en pourcentage le taux de bonne estimation. Pour cela, on prend nos données et on regarde si les calculs donnent le bon résultat, c'est-à-dire que nous vérifions si la catégorie déterminée est bien celle du point.
Cela permet d'obtenir le taux de réussite lors d'une classification d'une nouvelle donnée par le programme.

Le calcul se base sur le nombre de réussite divisé par le nombre de tests effectués.

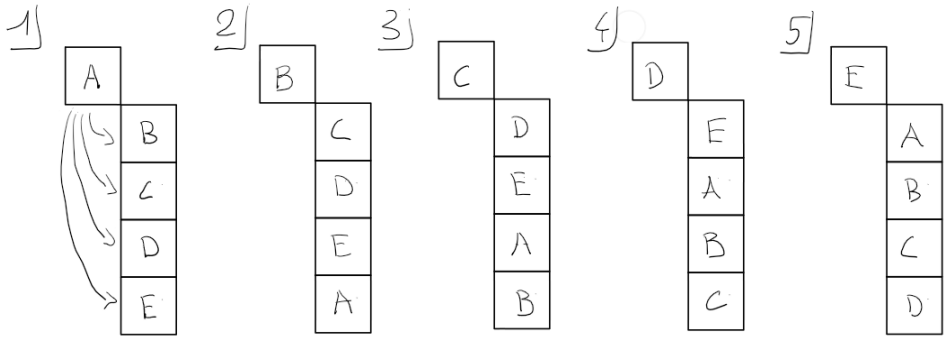
<u>Notre implémentation :</u>
Nous avons implémenté ce calcul de robustesse dans la classe KnnMethod. La fonction prend en paramètre une distance et la liste des points que l'on appellera data.

Elle crée :
- une liste vide temporaire qui sera la liste d'un paquet en cours.
- une variable numérique comptabilisant le nombre de tests effectués.
- une variable numérique comptabilisant le nombre de réussite de tests (de bonne classification).

Nous allons diviser la liste data en plusieurs paquets car nous allons diviser le calcul en plusieurs étapes. Nous avons choisi de décomposer en 5 paquets de points, soit 1 paquet contiendra ⅕ des points. Le nombre de points par paquet sera stocké dans une variable au début de la fonction appelée divisor.

Pour tester un paquet, nous allons déplacer dans la liste temporaire, du premier point au ⅕ des points de la liste data, tester tous les points du paquet actuel, puis remettre ces points dans la liste data et effacer la liste temporaire.

Voici une illustration de ce que la fonction réalise :




Nous allons répéter ces opérations jusqu'à ce que tous les points aient été testés soit 5 fois car nous décomposons les points en 5 paquets :

- On remplit la liste temporaire des x premiers points tel que x est le nombre de points par paquet (variable divisor).
- Une fois le paquet rempli d'⅕ de points de data, on enlève à la liste data les premiers points jusqu'à la taille du paquet. Nous avons donc déplacé x points de la liste data vers la liste temporaire.
- Ensuite, pour chaque point de l'échantillon, on va comparer si sa catégorie renseignée est la même que la catégorie calculée (trouvée via les algorithmes).

  Pour trouver la catégorie d'un point:
  - On récupère la liste des k voisins les plus proches du point tel que k est le nombre de points par paquet, (getNeighbours)
  - On récupère la catégorie majoritaire parmi les k plus proches voisins.(classifier)

  Si la catégorie du point est la même que celle calculée, alors on incrémente la variable comptant le nombre de bonnes classifications d'un point.

- Une fois que tous les points du paquet en cours ont été testés, on les replace à la fin de la liste data, ainsi on vide la liste temporaire.

Une fois que tous les points de la liste data ont été testés, on retourne le pourcentage de réussite (le nombre de réussite divisé par le nombre d'échantillons).

Voici le code de la robustesse:

```
public double getRobustesse(IDistance<T> distance, List<T> data) {
    List<T> tmp = new ArrayList<>();
    int divisor = data.size() / 5;
    double wellClassified = 0;
    double totalClassified = 0;
    for(int i = 0;i < 5; i++) {
        tmp.clear();
        for (T p : data) {
            if(tmp.size() < divisor) {
                tmp.add(p);
            }
        }removeElement(data,0,divisor);
        for(T p : tmp) {
            if(p.getCategory().equals(classifier(getNeighbours(p, divisor, distance, data)))) {
                wellClassified++;
            }totalClassified ++;
        }
        addElement(data, tmp);
    }return Math.round((wellClassified/totalClassified)*100.0*100.0)/100.0;
}
```

Nous avons créé un test de robustesse avec 15 iris.

Pour la recherche des voisins, le nombre de k voisins est variable selon le nombre de données.

```
"sepal.length","sepal.width","petal.length","petal.width","variety"
5.8, 4, 1.2, 0.2, "Setosa"
6.9,3.1,5.1,2.3,"Virginica"
6.6, 2.9, 4.6, 1.3, "Versicolor"
4.9, 3, 1.4, 0.2, "Setosa"
5.6,2.7,4.2,1.3,"Versicolor"
6,3,4.8,1.8,"Virginica"
4.7, 3.2, 1.3, 0.2, "Setosa"
5.2, 2.7, 3.9, 1.4, "Versicolor"
6.7,3.1,5.6,2.4,"Virginica"
5.1, 3.5, 1.4, 0.2, "Setosa"
5.8,2.6,4,1.2,"Versicolor"
5.2,3.4,1.4,.2,"Setosa"
6.9,3.1,5.4,2.1,"Virginica"
5,2.3,3.3,1,"Versicolor"
6.4,3.1,5.5,1.8,"Virginica"
```

```
@BeforeEach
void setUp(){
    this.model = new CSVModel<>(Iris.class, "Iris");
    this.knn = new KnnMethod<>();
    model.loadFromFile("src/main/resources/irisTest.csv");
    this.distance = new ManhattanDistance<>(model.getColumns());
    allPoints = model.getPoints();
}
```

On a calculé que la robustesse trouvée était correcte

```
@Test
void test_getRobustesse() {
    assertEquals(73.33, knn.getRobustesse(distance,allPoints),0.001); // ici 11 bon sur les 15 donc 11/15 = 73,33
}
```

Pour un résultat optimal, le mieux sera de mélanger la liste de points avant l'appel de la fonction.

## III) Conclusion :

Nous avons réalisé une application de classification graphique sous javafx.

Nous pouvons :

- Sélectionner un fichier de données csv,
- Choisir l'axe des abscisses et des ordonnées avec une colonne différente,
- Afficher le graphique du nuage des points,
- Ajouter un point, une donnée pour classifier cette valeur,
- Afficher la robustesse,
- Choisir k, le nombre de voisins.

Nous avons fait en sorte de répondre au mieux à la problématique demandée et avons essayé de respecter notre diagramme de cas d'utilisation. Nous avons tous collaboré et nous nous sommes chacun aidé mutuellement.