|  |
| --- |
| Compte rendu sur la classification de données  S3.02 – Développement d’application |
| Repo : <https://gitlab.univ-lille.fr/sae302/2022/equipe-I1> |
| 28 novembre  BUT 2 Informatique 2022/2023  Équipe I1 |



**Sommaire**

I. Équipe I1 3

A. Canoen Théo 3

B. De Wancker Randy 3

C. Defossez Florine 3

D. Varlamoff Léopold 3

II. Analyse des données 3

A. Chargement des données 3

B. Types de données et normalisation 4

C. Distances 4

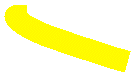
III. Implémentation de k-NN 5

IV. Robustesse des modèles 6

A. Avec fichier de test 6

B. Sans fichier de test 6

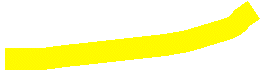
# Équipe I1



## Canoen Théo

Réalisations : classes titanic et iris, test du Dataset, une partie du rapport de classification.

## De Wancker Randy



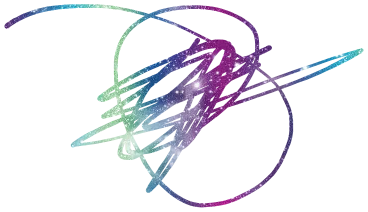
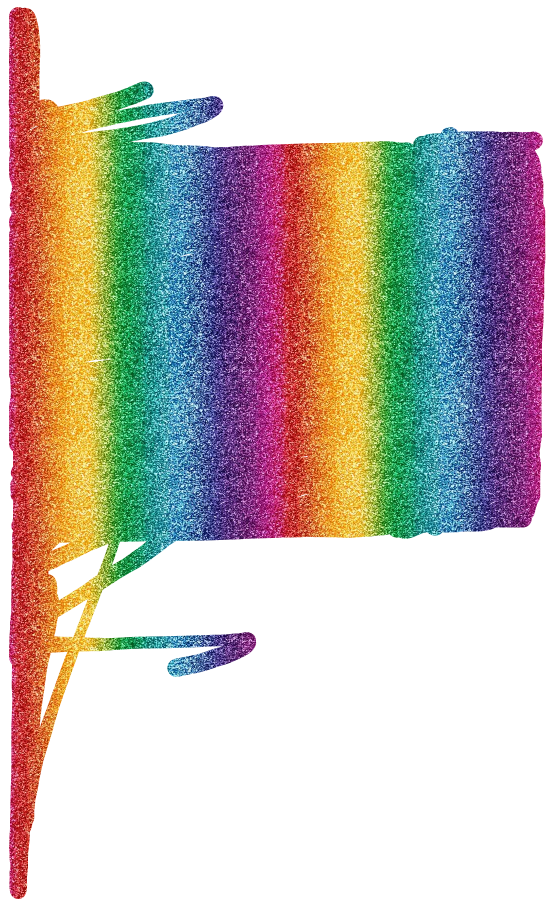
Réalisations : classe NumberNormalizer, le FXML et l’interface JavaFX.

## Defossez Florine

Réalisations : classe Titanic, autres classes de normalisation, et des tests.

## Varlamoff Léopold

Réalisations : organisations des packages et des interfaces, chargement des données, classes Dataset et des distances, le MVC, des tests, l’autre partie du rapport.



# Analyse des données

## Chargement des données

Pour charger les données, nous considérons que chaque instance de la classe ‘Dataset’ peut contenir un type de point. Nous demandons la classe des points qui seront ajoutés dans le set et chargeons les données à l’aide de la librairie ‘opencsv’. Dans le cas où il nous manquerait des données, comme c’est le cas plusieurs fois dans les données du Titanic, nous avons décidé de simplement mettre les valeurs à la valeur minimale de la colonne. Nous avons hésité à mettre les valeurs absentes à 0, mais dans le cas où aucune autre valeur n’est à 0, cela « fausse » la normalisation.

## Types de données et normalisation

Pour effectuer les calculs et la classification, nous avons travaillé avec plusieurs types de données :

* Des valeurs numériques (par exemple une longueur de pétale), que nous normalisons grâce à la formule où min et max sont respectivement les valeurs minimales et maximales de la colonne en traitée.
* Des valeurs ordinales, comme une classe de passager. La normalisation consiste à trier les différentes valeurs possibles et à leur attribuer une valeur entre 0 et 1 selon leur ordre d’importance.
* Des valeurs énumératives comme un port de départ. La normalisation est similaire à celle des données ordinales, sauf que l’ordre est arbitraire : cela n’a pas d’impact tant que cet ordre est constant.
* Des valeurs booléennes, par exemple si un Pokémon est légendaire ou pas. Une valeur vraie équivaut simplement à 1 et faux à 0.
* Enfin, nous avons travaillé avec des valeurs binaires, comme le sexe d’un passager. La normalisation est la même que pour des valeurs booléennes : une valeur vaut 1 et l’autre 0.

## Distances

Pour implémenter l’algorithme k-NN, nous avons besoin de calculer la distance entre différents points. Nous avons implémenté 3 types de distances différentes :

* La distance euclidienne, de formule , est implémentée sur les données normalisée.
* La distance de Manhattan, de formule , est également implémentée sur les données normalisées.
* Une méthode de distance complétement aléatoire, qui est choisi par la méthode ‘Math.random()’, c’est-à-dire des valeurs pseudo-aléatoires comprises dans l’intervalle .

# Implémentation de k-NN

Pour implémenter l’algorithme k-NN, nous avons besoin de 3 paramètres en plus de la liste de toutes les données importées. Premièrement, le ‘k’ en question, c’est-à-dire combien de voisin l’algorithme doit-il prendre un compte pour prendre une décision. Ensuite, le point à classer. Finalement, la méthode accepte une ‘Distance’, c’est-à-dire une implémentation d’une distance à utiliser. Comme décrit précédemment, nous avons 3 types de distances différentes : distance euclidienne et de Manhattan adaptées pour des valeurs normalisées et une distance aléatoire.

Nous avons également créé une class ‘Couple’ qui permet de stocker un point avec une distance en double, ce qui nous permet de pouvoir facilement trouver les plus proches voisins d’un point.

Notre algorithme commence par remplir une liste de Couple. Ces couples sont instanciés avec chaque point du set de données et la distance à laquelle ce point est du point que nous voulons classer. Une fois la liste alimentée, nous utilisons la méthode ‘List.sort()’ avec un comparateur que nous avons créé. Celui-ci effectue simplement une comparaison sur les distances de chaque couple. Après l’invocation de cette méthode, la liste est triée dans l’ordre croissant des distances. Il ne reste plus qu’à retourner un tableau contenant les k premiers points de la liste.

Nous avons choisi cette approche car nous l’avons trouvé simple et qu’elle a l’avantage de manipuler des méthodes implémentées par le langage Java, ce qui garantit de bonnes performances même sur d’énormes sets de données. En effet, la méthode sort utilise l’algorithme merge sort (ou tri fusion en français) qui garantit une complexité temporelle de , quel que soit le nombre de point de nos fichiers CSV.

Après avoir trouvé les k plus proches voisins, nous avons décidé, dans un premier temps, de sonder les points sur leur classe. Si une classe est majoritaire, alors c’est celle qui sera attribuée au point à classer.

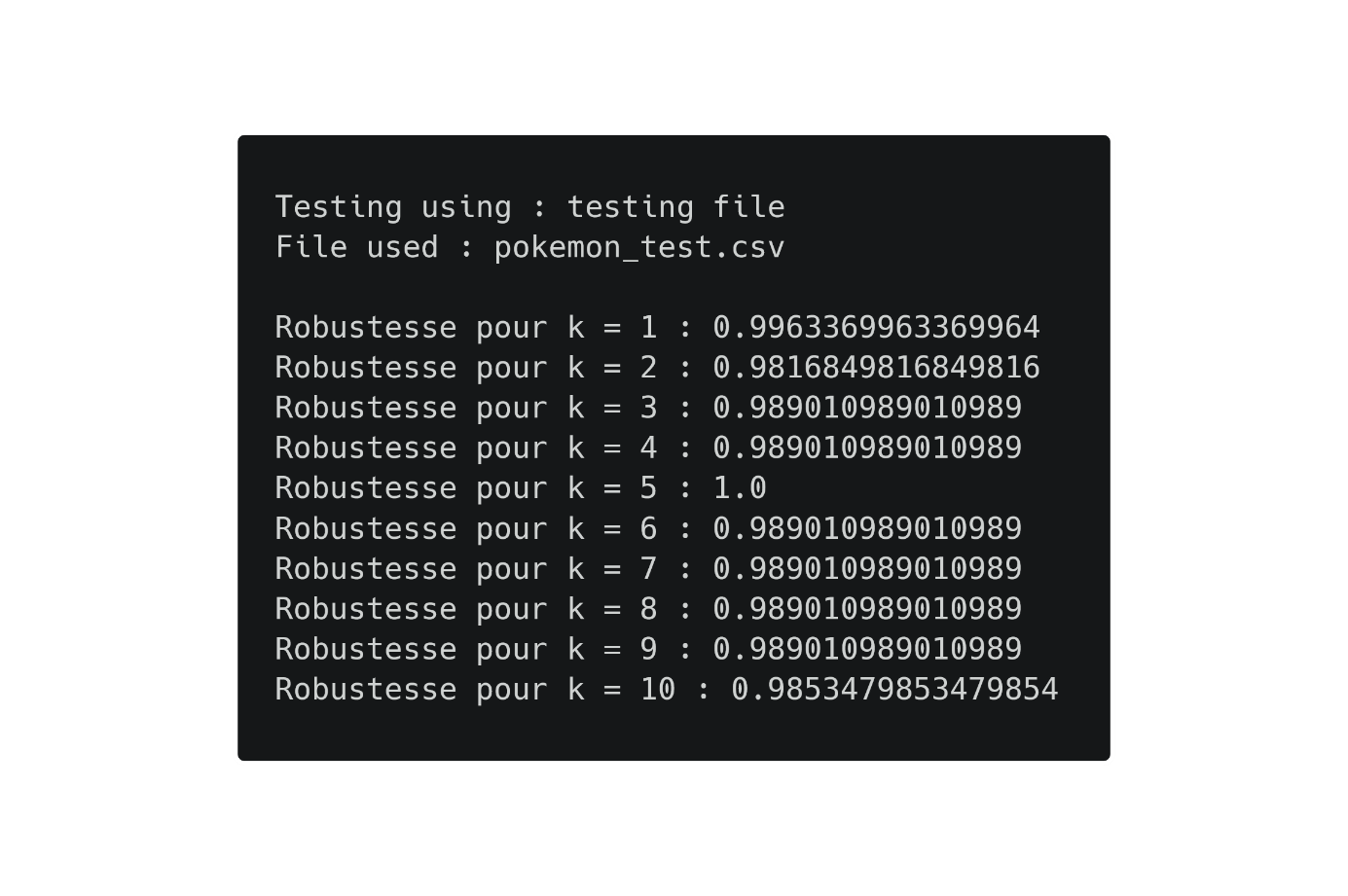
En revanche, dans le cas où plusieurs classes sont majoritaires, nous utilisons le principe du vote majoritaire : pour chaque point, nous appliquons la formule . Pour avoir le vote de chaque classe, on fait la somme des ω. La classe qui à la plus de poids sera finalement appliquée au point.

# Robustesse des modèles

## Avec fichier de test

Lorsqu’un fichier de test nous est fourni, ce qui est le cas uniquement avec les données des Pokémons, nous avons décidé de simplement faire tourner l’algorithme k-NN sur les données de test et de comparer le taux de réussite de celui-ci par rapport aux données du fichier test.

**Scénario** : robustesse du modèle des données Pokémon ‘pokemon\_train.csv’ grâce au fichier de test ‘pokemon\_test.csv’ en utilisant la distance de Manhattan normalisée.



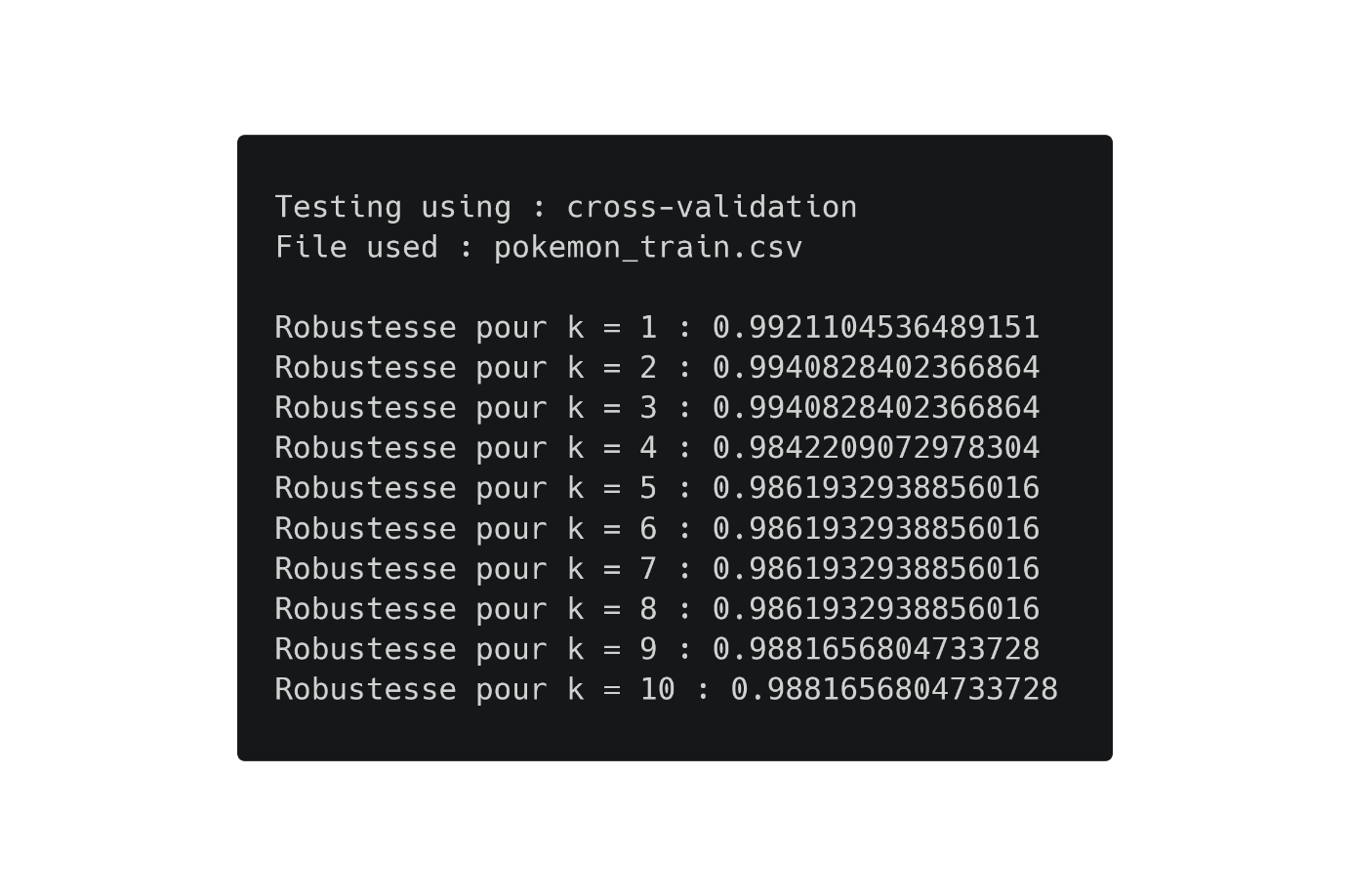
## Sans fichier de test

Dans les autres cas où nous n’avions pas de fichier de test, comme c’est le cas avec les données des iris, nous avons utilisé la technique de cross-validation (ou validation croisée), notamment la validation croisée d'un contre tous. Celle-ci se base sur le principe de tester chaque point du set individuellement grâce à tous les autres.

Pour réaliser cette validation, nous itérons sur notre set de données. A chaque itération, nous divisons le set en deux parties distinctes : la première est constituée uniquement du point sur lequel l’itérateur pointe et sera le set de test, et la seconde est constituée de tous les autres. On applique ensuite l’algorithme k-NN sur ce point et on compare les résultats avec les données de départ.

Nous avons choisi cette technique de validation car elle a l’avantage d’effectuer un nombre de tests égal au nombre de données du set, ce qui nous permet d’avoir une robustesse plus fiable, sans avoir un temps de calcul trop important (il existe d’autres techniques de validation croisée se basant sur le même principe mais avec un échantillon de test supérieur à 1, ce qui démultiplie les calculs).

**Scénario** : robustesse du modèle des données de Pokémon ‘pokemon\_test.csv’ en validation croisée, en utilisant la distance de Manhattan normalisée



On observe bien que pour les deux cas de test, on obtient des valeurs similaires mais tout de même différentes.