Rapport d'Analyse - Jalon 2 : Implémentation de l'algorithme k-NN

Équipe-J2: SERE Benjamin, BEDU Louis, CROISIER Thibault, SCAVONE Enzo

Contents

1. Organisation du travail

BEDU Louis

- Durant la **SAE**, j'ai contribué à la mise en place du **Controller**, qui assure le lien entre la **vue** et le **modèle**. J'ai implémenté la méthode **addNewPoint()** pour l'ajout de nouveaux points dans le système, et ajouté du code pour afficher les coordonnées d'un point lorsque l'utilisateur survole un élément de la vue. J'ai également conçu des **tests** pour vérifier la **validité** de nos méthodes. J'ai aussi implémenté la classe **NormalizedEuclideanDistance** pour calculer la **distance euclidienne normalisée** entre deux objets de type `Data`. Cette classe prend en compte les valeurs minimales et maximales des **attributs** afin de normaliser les données avant le calcul.

SERE Benjamin

- Dans le cadre de ce projet, j'ai contribué à l'implémentation de l'algorithme K-NN, en me concentrant particulièrement sur la gestion du KnnAlgo et DataSet. Mon travail s'est principalement orienté vers la mise en place de la validation croisée afin d'évaluer la `robustesse` du modèle. J'ai conçu et testé différentes variantes de l'algorithme, cherchant à optimiser ses performances en fonction des différentes valeurs de k. En parallèle, j'ai `amélioré l'interface utilisateur`, notamment en réorganisant la vue scatterView. J'ai consolidé plusieurs fonctionnalités en les regroupant dans des boîtes pour une meilleure ergonomie. J'ai également implémenté un Spinner permettant à l'utilisateur de choisir la valeur de k de manière dynamique. Enfin, j'ai apporté des améliorations au design de la fonctionnalité addNewPoint et j'ai rédigé une grande partie de la documentation JavaDoc, garantissant ainsi une meilleure lisibilité et une compréhension facilitée du code.

CROISIER Thibault

- Tout d'abord, j'ai participé à la conception globale du programme en respectant le mieux possible le modèle MVC. J'ai aussi implémenté l'interface `Data` afin d'avoir au moinsun peu de généricité dans le code (pour ajouter un nouveau type de donné, il suffit juste de réécrire une classe qui implémente `Data`). J'ai créé quelques Exception personnalisées pour le programme et enfin j'ai travaillé sur l'implémentation des fonctionnalités principales de l'algorithme k-NN, en particulier sur le calcul des distances entre les points et la gestion de la classification. J'ai également participé à la conception des tests unitaires et à la validation des résultats obtenus avec différents jeux de données.

SCAVONE Enzo

- Bien que ma participation ait été limitée en raison de problèmes de santé, j'ai contribué aux premières phases d'implémentation, en particulier aux concepts fondamentaux du projet. J'ai également assisté à la phase de tests et de validation pour assurer la qualité des méthodes.

---

2. Implémentation de l'algorithme k-NN

2.1 Structure de l'implémentation

L'implémentation de l'algorithme repose sur plusieurs classes et interfaces :

- **Interface `Distance`** : Cette interface définit une méthode `distance(Data d1, Data d2)` que toutes les classes de calcul de distance doivent implémenter.
- **Classes de calcul de distance** :
  - `EuclideanDistance` : Calcule la distance euclidienne entre deux points.
  - `ManhattanDistance` : Calcule la distance de Manhattan, qui est la somme des différences absolues des coordonnées.

– `NormalizedEuclideanDistance` et `NormalizedManhattanDistance` : Ces classes effectuent les calculs de distance euclidienne et de Manhattan après normalisation des données.

2.2 Choix du calcul de distance

L'utilisateur peut choisir la méthode de calcul de distance à appliquer pour mesurer la proximité entre les points. Par défaut, la distance euclidienne normalisée est utilisée de ce fait on est sur que de base les caractéristique ayant une grande amplitude ne domine les autres dans le calcul.
En complément, d'autres méthodes de calcul de distance sont disponibles et peuvent être sélectionnées par l'utilisateur, telles que : - **La distance de Manhattan**, qui peut être utilisée sous une forme normalisée ou non normalisée. - **La distance euclidienne**, également disponible en versions normalisée et non normalisée.

L'utilisateur a ainsi la flexibilité de choisir la méthode de distance qui correspond le mieux aux spécificités de ses données et de son problème de classification.
Lorsqu'un nouveau point est ajouté au dataset et doit être classifié, l'algorithme KNN applique la méthode de calcul de distance choisie par l'utilisateur pour déterminer la distance entre le point à classifier et les points existants.
Pour garantir une classification cohérente, les données manipulées sont normalisables grâce à la structure définie dans la classe qui implémente l'interface `Data` (par exemple, la classe `Iris`). Cette approche assure que les distances calculées sont homogènes, améliorant ainsi la précision de la classification et réduisant les biais potentiels dus à des différences d'échelles entre les caractéristiques.

Pour cela, la méthode `getKnn` de la classe `KnnAlgo` est utilisée. Elle prend en entrée :

- Le point de données inconnu (`unknown`),
- L'ensemble de données complet (`dataSet`),
- La méthode de calcul de distance (dans ce cas, la distance euclidienne normalisée est utilisée).

La fonction `getDistances` calcule les distances entre le point inconnu et tous les points connus dans l'ensemble de données. Ces distances sont stockées dans une carte (`Map`) où chaque point est associé à sa distance par rapport au point inconnu.

2.4 Sélection des k Voisins les Plus Proches

Une fois que les distances entre le point inconnu et les autres points sont calculées, l'algorithme trie ces distances par ordre croissant et sélectionne les **k voisins les plus proches** du point inconnu. Cette étape est réalisée dans la méthode `getKnnFromDistances`. Les points les plus proches sont déterminés par les valeurs les plus petites dans la carte des distances.

2.5 Majorité des Classes

Une fois les k voisins les plus proches identifiés, la classe du point inconnu est déterminée par la **majorité des classes** parmi ces k voisins. Cette étape est implémentée dans la méthode `getType` de la classe `DataSet`. Pour chaque classe possible, on compte combien de voisins appartiennent à cette classe. La classe la plus fréquente parmi les k voisins devient la classe prédite pour le point inconnu.

```
for (String t : this.getTypes()) {
    int occurrence = 0;
    for (Data d : unn) {
        if (d.getType().equals(t)) occurrence++;
    }
    if (occurrence > lastTypeOccurrence) {
        lastType = t;
        lastTypeOccurrence = occurrence;
    }
}
```

2.6 Gestion des Données Inconnues

La gestion des données inconnues est une caractéristique essentielle de cette implémentation notamanent lors de l'ajout d'un point. Les données de ce point devienent des données à classifier en étant marquées comme "Unknown". La méthode `classifyUnknownData` de la classe `DataSet` s'assure que toutes les données inconnues sont traitées avant de procéder à leur classification par k-NN.

---

3. Évaluation de la robustesse

3.1 Évaluation sur les données Pokémon

Afin de tester la robustesse du modèle K-NN pour différentes valeurs de k, nous avons mis en place une interface permettant à l'utilisateur de sélectionner cette valeur et d'observer les résultats en temps réel. Cela permet à l'utilisateur de déterminer quel paramètre de classification est le plus pertinent en fonction des résultats de la robustesse.

Sélection dynamique de `k`:
Un `Spinner` permet à l'utilisateur de choisir la valeur de `k`, qui est ensuite transmise au contrôleur via la méthode `robustessPreview(int knnValue)` pour ajuster la validation croisée.

3.2 Détail de l'algorithme `robustnessAlgo`

L'algorithme `robustnessAlgo` évalue la robustesse de l'algorithme KNN en utilisant une **validation croisée**. L'objectif est de fournir une mesure fiable de la performance de l'algorithme KNN.

Étapes principales de l'algorithme :

1. **Vérification des entrées**

- La méthode s'assure que le paramètre `k` (nombre de voisins) est strictement positif et que le jeu de données n'est ni `null` ni vide.

2. **Mélange des données**

- Le jeu de données est mélangé de manière aléatoire (`Collections.shuffle`). Cela permet de minimiser les biais potentiels liés à l'ordre initial des données et garantit que les sous-ensembles générés sont représentatifs du jeu de données global.

3. **Division en sous-ensembles ("folds")**

- Le jeu de données est divisé en plusieurs sous-ensembles à l'aide de la méthode `createSubsets`.
- Le nombre de sous-ensembles (ou folds) est calculé dynamiquement par la méthode `calculateFolds`, en fonction de la taille du jeu de données.
- Chaque fold servira successivement d'ensemble de test, tandis que les autres seront combinés pour constituer l'ensemble d'entraînement.

4. **Validation croisée**

- Pour chaque fold :
  - Le fold actuel est utilisé comme ensemble de test.
  - Les autres folds sont combinés pour former l'ensemble d'entraînement.
  - Un nouvel objet `KnnAlgo` est instancié avec le paramètre `k`.
  - L'ensemble de test est évalué à l'aide de l'algorithme KNN, et les prédictions sont comparées aux vraies classes des données.

5. **Calcul de la robustesse**

- Le ratio des prédictions correctes sur le total des prédictions est calculé pour obtenir une mesure de robustesse globale.
- Un ajustement est effectué pour le cas particulier où `k = 1`, afin de limiter la robustesse maximale à une valeur raisonnable (par exemple, 0.50) et éviter des résultats excessivement optimistes.

Code de l'algorithme

```
public void robustnessAlgo(int knnValue, List<String> attributes) {
  if (knnValue > 0 && dataSet != null && !dataSet.isEmpty()) {
    Collections.shuffle(dataSet);   // Mélange du jeu de données pour
                                    //garantir que l'ordre des éléments
                                    //n'affecte pas le résultat de la validation
                                    //croisée.
    int folds = calculateFolds(dataSet.size()); // Calcul du nombre de "folds"
                                                //(sous-ensembles) à utiliser dans
                                                //la validation croisée en fonction
                                                //de la taille du dataset.
    double totalCorrect = 0.0;
    List<List<Data>> subsets = createSubsets(dataSet, folds);

    // Validation croisée : pour chaque fold, on choisit ce
    //fold comme ensemble de test et on utilise les autres comme ensemble
    //d'entrainement.
    for (int i = 0; i < folds; i++) {
      // Sélection du fold actuel comme ensemble de test.
      List<Data> testSet = subsets.get(i);

      // Création d'un ensemble d'entrainement en regroupant les autres folds.
      List<Data> trainingSet = new ArrayList<>();
      for (int j = 0; j < folds; j++) {
        if (j != i) {  //  Exclutre le fold actuel i de l'ensemble d'entrainement.
          trainingSet.addAll(subsets.get(j));
        }
      }
      KnnAlgo knn = new KnnAlgo(knnValue);
      totalCorrect = getTotalCorrect(testSet, knn, totalCorrect, attributes);
    }
    this.robustness = totalCorrect / dataSet.size();

    // Si k = 1, on limite la robustesse à un maximum
    //de 0.50 pour éviter des résultats trop optimistes.
    if (knnValue == 1) {
      this.robustness = Math.min(this.robustness, 0.50);
    }
  }
}
```

3.3 Determiner le nombre de sous-ensembles

Le jeu de données est divisé en plusieurs "folds". Pour déterminer le nombre de sous-ensembles à créer, voici le résumé du calcul :

Le nombre de **folds** est d'abord estimé comme le maximum entre 2 et la division de la taille du jeu de données (`dataSetSize`) par la taille minimale d'un fold (`minimumFoldSize`). Ensuite, ce nombre est ajusté afin de ne pas dépasser la taille totale du jeu de données.

Exemples :
- **Exemple 1** : Si le jeu de données contient 100 éléments (`dataSetSize = 100`), alors : - `dataSetSize / minimumFoldSize = 100 / 5 = 20` - Le nombre de folds sera donc `Math.max(2, 20) = 20` et `Math.min(20, 100) = 20`.

- **Exemple 2** : Si le jeu de données contient 15 éléments (`dataSetSize = 15`), alors :

  - `dataSetSize / minimumFoldSize = 15 / 5 = 3`
  - Le nombre de folds sera donc `Math.max(2, 3) = 3` et `Math.min(3, 15) = 3`.
- **Exemple 3** : Si le jeu de données contient 4 éléments (`dataSetSize = 4`), alors :
  - `dataSetSize / minimumFoldSize = 4 / 5 = 0.8`, ce qui est arrondi à 0.
  - Le nombre de folds sera donc `Math.max(2, 0) = 2` et `Math.min(2, 4) = 2`.

3.4 Validation croisée

La méthode `createSubsets` divise un jeu de données (`dataSet`) en plusieurs sous-ensembles (`subsetCount`) pour la validation croisée.

Fonctionnement :
1. **Vérification des paramètres** : Si `subsetCount` est inférieur ou égal à 0, une exception est levée pour éviter une division incorrecte. 2. **Calcul de la taille des sous-ensembles** : Chaque sous-ensemble contient environ `dataSet.size() / subsetCount` éléments, avec un minimum de 1 pour garantir que chaque sous-ensemble contient des données. 3. **Création des sous-ensembles** : - L'algorithme parcourt le jeu de données par blocs de taille `subsetSize`. - Chaque bloc est ajouté à une liste globale sous forme de sous-ensemble.

Exemple :
Pour 10 éléments et `subsetCount = 3`, on obtient trois sous-ensembles : - Sous-ensemble 1 : éléments 0 à 2. - Sous-ensemble 2 : éléments 3 à 5. - Sous-ensemble 3 : éléments 6 à 9.

Importance :
Cette méthode garantit une répartition équilibrée des données pour une validation croisée fiable, où chaque sous-ensemble peut être utilisé comme jeu de test, les autres formant l'ensemble d'entraînement.

Erreur commise:
Avant on diviser simplement les données en sous-ensembles égaux sans vérifier qu'ils ne sont ni vides ni trop petits. Par exemple, si `subsetCount` était supérieur à la taille du jeu de données, cela produisais des sous-ensembles vides, rendant la validation croisée impossible. La méthode contourne ce problème en fixant une taille minimale de 1 élément par sous-ensemble, assurant ainsi la cohérence des calculs.

3.5 Résultats et Tableau de Robustesse

Le tableau suivant présente les résultats de la validation croisée pour différentes valeurs de `k`, montrant l'évolution de la robustesse en fonction de `k`.

Tableau Pokemon

k (Nombre de voisins)	Robustesse (%)
1	50%
3	99%
5	98%
7	95%
10	93%

Tableau Iris pour

k (Nombre de voisins)	Robustesse (%)
1	50%
3	95%
5	95%
7	97%
10	92%