

# Algorithme K-NN - Rapport R3.02

## Groupe H-4

- [MENNECART Matias](#)
- [DEBUYSER Hugo](#)
- [ANTOINE Maxence](#)
- [DEKEISER Matisse](#)
- [DESMONS Hugo](#)

---

## Implémentation de K-NN

*Une description de votre implémentation de l'algorithme k-NN : classe implémentant l'algorithme, méthode(s) de cette classe implémentant le calcul de la distance, traitement de la normalisation, méthode(s) de cette classe implémentant la classification, méthode(s) évaluant la robustesse. N'hésitez pas à mettre en avant l'efficacité de ces méthodes (approprié pour un grand volume de données, normalisation efficace des distances).*

Nous avons implémenté l'algorithme K-NN dans une classe `MethodKNN`. Nous avons choisi d'utiliser des méthodes statiques pour faciliter l'utilisation de cet algorithme. Cette classe contient 5 méthodes permettant d'implémenter l'algorithme K-NN et ses différentes fonctionnalités.

```
public static void updateModel(List datas)
```

Cette première méthode permet de mettre à jour les données de l'algorithme. Ainsi, il faut passer en paramètre les données sur lesquelles on souhaite travailler. Cette méthode va calculer les valeurs max et min ainsi que l'amplitude de chaque attribut des données passé en paramètre. Cette méthode nécessite 2 parcours : 1 sur les données, puis un sur les valeurs min et max de chaque attribut pour calculer l'amplitude. Il est donc conseillé de n'utiliser cette méthode qu'une fois par jeu de données (le résultat serait le même). C'est pour cela que cette méthode n'est pas directement appelée dans les méthodes ci-dessous, mais qu'elle doit explicitement être appelée auparavant.

```
public static List kVoisins(List datas, LoadableData data, int k, Distance distance)
```

Cette méthode a pour objectif de récupérer les **k** voisins les plus proches d'une donnée parmi un jeu de données et selon une distance. Elle prend donc en paramètres le jeu de données, la donnée pour laquelle on souhaite obtenir les voisins, le nombre de voisins souhaités ainsi que la distance avec laquelle les calculs doivent être effectués. Cette méthode n'effectue qu'un seul parcours de boucle, et calcule pour chacune des données sa distance avec la donnée passée en paramètre. Les calculs de distance sont définis dans l'objet implémentant l'interface `Distance` passé en paramètre. S'il s'agit d'une distance normalisée, la normalisation s'effectue au moment de la recherche des voisins.

```
public static double robustesse(List datas, int k, Distance distance, double testPart)
```

Cette méthode a pour objectif d'évaluer la robustesse de l'algorithme sur un jeu de données avec un **k** et une distance donnés. Elle prend en paramètres le jeu de données sur lequel effectuer l'évaluation, le **k** à tester, la distance à utiliser ainsi qu'un pourcentage correspondant à la partie des données qui sera utilisée afin de tester les données. Exemple avec `testPart=0.2`, 80% des données serviront de données de référence et 20% seront utilisées pour tester la validité de l'algorithme. Cette méthode effectue une validation croisée. (voir `Validation croisée`)

```
public static int bestK(List datas, Distance distance)
```

Cette méthode a pour objectif de rechercher le meilleur **K** possible pour un jeu de données et une distance donnée. Elle va tester la totalité des **K** impairs compris entre 1 et racine carrée du nombre de données. Cette valeur max permet d'éviter que le **K** choisi soit trop grand et fausse les résultats. Elle teste donc la robustesse de chaque **K** et renvoie le **K** ayant la meilleure robustesse. Cette méthode parcourt le jeu de donnée  $K_{max} * (1/testPart)$  fois, où `testPart` correspond au pourcentage des données utilisées comme valeurs de test.

---

## Validation croisée

## Rappel de la méthode de validation croisée

La validation croisée est une méthode d'évaluation utilisée pour mesurer la performance d'un modèle en le testant sur des données qu'il n'a pas utilisées pour l'entraînement. Dans ce cas précis :

Les données sont divisées en plusieurs parties égales (appelées *folds* ou *partitions*). À chaque itération, une des partitions est utilisée comme jeu de test, tandis que les autres servent pour l'entraînement. Les résultats des tests sont cumulés pour calculer un score global. Cette méthode permet de minimiser le biais d'évaluation en utilisant toutes les données tour à tour pour l'entraînement et le test.

## Implémentation de la validation croisée

La méthode effectue une validation croisée en divisant les données en plusieurs partitions. À chaque itération :

Une partie des données sert de jeu de test. Le reste des données sert de jeu d'entraînement.

Voici les étapes principales :

- Calcul du nombre d'itérations : Le nombre d'itérations est déterminé par  $1/\text{testPart}$ . Par exemple, si  $\text{testPart} = 0.1$ , la méthode effectue 10 itérations, si  $\text{testPart} = 0.2$ , la méthode effectue 5 itérations, etc.
- Division des données : Pour chaque itération  $i$ , une sous-liste correspondant au pourcentage  $\text{testPart}$  (par exemple, 10% des données) est extraite et utilisée comme jeu de test ( $\text{testData}$ ). Le reste des données sert de jeu d'entraînement ( $\text{trainingData}$ ).
- Estimation des classes : Chaque élément du jeu de test est classé à l'aide de la fonction `MethodKNN.estimateClass(trainingData, l, k, distance)`, où  $\text{trainingData}$  correspond au reste des données. Si la classe prédite correspond à la classe réelle (retournée par `l.getClassification()`), un compteur ( $\text{totalFind}$ ) est incrémenté.
- Calcul du taux de réussite : Après chaque itération, le taux de réussite est calculé ( $\text{totalFind}/\text{totalTry}$ ) et ajouté à la variable  $\text{taux}$ .
- Renvoi du taux total moyen de réussite : enfin, on divise la variable  $\text{taux}$  par le nombre d'itérations afin d'obtenir un taux de réussite moyen.

---

## Choix du meilleur K

Pour obtenir le meilleur **K**, on appelle la méthode `bestK(List<LoadableData> datas, Distance distance)` décrite plus haut. On obtient un **K** optimal, puis on appelle la méthode `robustesse(...)` avec le  $k$  trouvé plus tôt comme paramètre.

En appliquant cette méthode, voici les résultats que nous avons obtenus avec :

### Iris

Distance \ k	1	3	5	7	9	11	k choisi
Distance euclidienne	0.96	0.966	0.96	0.98	0.98	0.98	5
Distance euclidienne (normalisée)	0.946	0.946	0.96	0.96	0.96	0.96	7
Distance Manhattan	0.953	0.946	0.946	0.96	0.96	0.953	7
Distance Manhattan (normalisée)	0.946	0.96	0.946	0.953	0.953	0.953	3

On obtient donc un taux de réussite plutôt élevé. À chaque fois, l'algorithme choisit le **K** avec le plus haut taux de réussite. En cas d'égalité, il choisit le plus petit **K** parmi les égalités.

### Pokémon

### Classification selon le type

Distance \ k	1	3	5	7	9	11	13	15	17	19	21	k choisi
Distance euclidienne	0.243	0.229	0.239	0.235	0.247	0.251	0.237	0.225	0.215	0.205	0.2	11
Distance euclidienne (normalisée)	0.211	0.229	0.251	0.245	0.245	0.239	0.245	0.243	0.237	0.239	0.225	5
Distance Manhattan	0.231	0.235	0.239	0.239	0.241	0.239	0.237	0.235	0.233	0.207	0.201	9
Distance Manhattan (normalisée)	0.178	0.188	0.2	0.215	0.205	0.203	0.194	0.190	0.184	0.180	0.190	7

Le taux de réussite est ici plus bas, cela s'explique notamment par le nombre d'attributs différents et la complexité à identifier le type d'un Pokémon. Cependant, le taux de réussite reste satisfaisant et stable.

### Classification (légendaire ou non légendaire)

Distance \ K	1	3	5	7	9	11	13	15	17	19	21	K choisit
Distance euclidienne	0.986	0.978	0.984	0.980	0.984	0.984	0.984	0.984	0.984	0.984	0.984	1
Distance euclidienne (normalisée)	1.0	0.998	0.998	0.996	0.996	0.998	0.998	0.998	0.998	0.998	0.998	1
Distance Manhattan	0.978	0.972	0.984	0.980	0.984	0.984	0.984	0.984	0.984	0.984	0.984	5
Distance Manhattan (normalisée)	0.980	0.984	0.988	0.984	0.984	0.986	0.986	0.986	0.986	0.986	0.984	5

On a ici des résultats bien meilleurs. En effet, estimer si un Pokémon est légendaire ou non est bien plus simple qu'estimer son type, les attributs des Pokémon légendaires sont bien différents de ceux non-légendaires, contrairement aux types, où, selon les types, les valeurs ne fluctuent pas autant.

## Efficacité

Comme expliqué pour chaque méthode dans la partie `Implémentation de l'algorithme`, nous avons cherché à minimiser le nombre de parcours du fichier de données, et plus généralement le nombre de boucles. L'algorithme nécessite une `List` qui sera donnée en paramètre, il est donc libre à la personne qui l'utilise de fournir l'implémentation de `List` qu'il souhaite. De plus, pour le calcul des paramètres du jeu de données (`amplitude`, `valeur minimale`, `valeur maximale`), nous avons utilisé un tableau de `double` afin de limiter les performances.