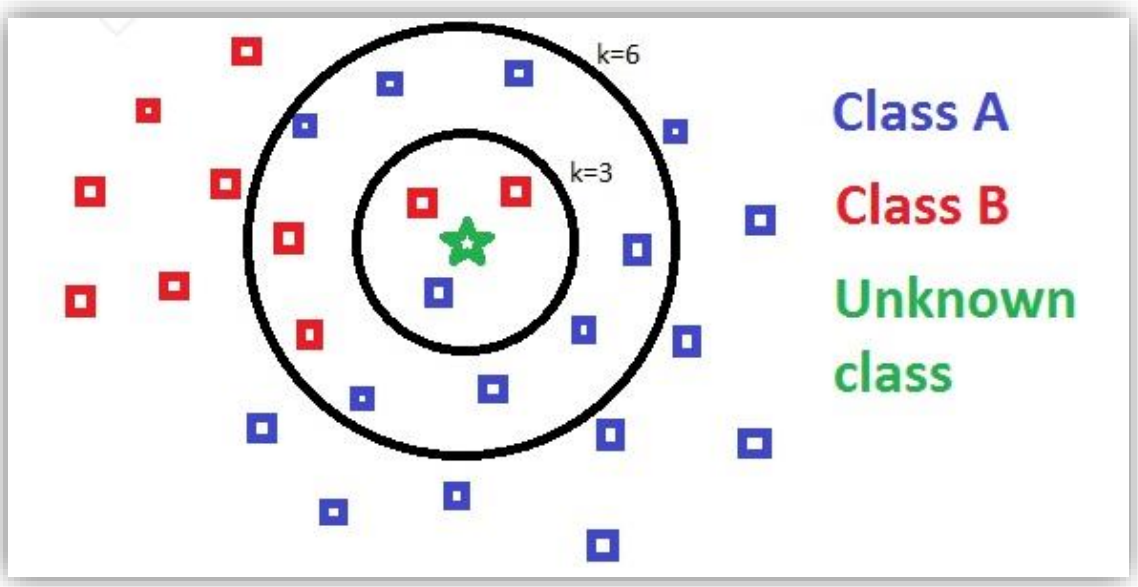
Rapport Knn



Quentin KAUFMAN

TD-N 2020-2021

ESILV
LÉONARD
DE VINCI

Introduction

Cette année en datascience et IA, nous devons réaliser plusieurs projets. L'un d'entre eux est l'algorithme du KNN(K-Nearest Neighbours). C'est un algorithme dont le concept est assez simple, mais qui à néanmoins beaucoup d'applications. Cet algorithme sert à grouper des données. On peut notamment l'utiliser pour déterminer à quelle variété appartient une fleur à partir de certains de ses paramètres (taille de la tige, des pétales, couleurs de ces derniers, etc). Pour cela on compare l'écartement des paramètres de cet individu avec ceux d'individus dont on connaît déjà la classe. Selon le nombre d'individus auxquels on compare, l'estimation peut être plus ou moins précise. On doit donc trouver le bon nombre d'individus auxquels il faut comparer. C'est le fameux « K »

Le temps de la réflexion

J'ai commencé par loader les données dans excel pour me donner une idée de leur organisation.

J'ai ensuite commencé un début de code pour les exploiter sous forme de liste.

```
import numpy as np
import matplotlib.pyplot as plt
import random as rnd
import pandas as pd

data = []

with open("data.csv","r") as file:
    for i in file:
        data.append(i.split(","))
#data = rnd.shuffle(data)


def separe(liste, pourcentageTest = 0.7):
    try:
        pourcentageTest>=0 and pourcentageTest<=1
    except :
        raise ValueError("Pourcentage de test doit être compris entre 0 et 1")
    #liste = rnd.shuffle(liste)
    nbrTest = round(len(liste)*pourcentageTest)
    test = liste[0:nbrTest]
    verif = liste[nbrTest+1:len(liste)]
    return test,verif

a = separe(data,2.01)
#print(a[0])
```

Puis j'ai voulu implémenter tous les points dans des classes pour pouvoir faire des opérations dessus. Je me suis vite rendu compte de l'inutilité de la tâche.

```python
class Ind:
    def __init__(self,var1,var2,var3,var4,var5,var6,classe):
        self.var1 = var1
        self.var2 = var2
        self.var3 = var3
        self.var4 = var4
        self.var5 = var5
        self.var6 = var6
        self.classe = classe

    def __str__(self):
        return "||{} | var1 = {:.4f} | var2 = {:.4f} | var3 = {:.4f}
```

J'ai donc commencé à utiliser la librairie pandas qui sert justement à faire des statistiques sous python.

J'ai mal importé mes datas (open with…) donc j'avais un problème de type. Mes datas étaient en string et non float. J'ai essayer de corriger ça avec les fonctions apply et to_numeric mais sans succès. Je me suis ensuite rendu compte qu'il suffisait d'utiliser read_csv de panda.

J'ai essayé d'exploiter mes données sur panda, mais j'avais beaucoup de mal à m'en servir comparé aux listes et aux classes. J'ai donc laissé tomber panda et suis revenu sur des listes. Ce fut mon choix final sur le changement d'architecture.

Pour bien commencer, j'ai posé à plat mon objectif et la méthode pour le réaliser. On veut déterminer la classe d'un individu en regardant ses similarités avec les autres individus les plus « proches ». J'ai découpé mon code en plusieurs fonctions principales :

- Sépare(data,pourcentage) elle prend un dataset et un pourcentage et le coupe en 2 dataset selon le pourcentage spécifié. Un sera utilisé pour tester nos données et l'autre pour vérifier les données.
- Distance(p1,p2) elle prend 2 points en paramètres et retourne un float correspondant à la distance entre ces points
- distanceTout(p1,data) elle prend 1 point et une liste de points et retourne une liste de liste contenant la distance entre p1 et chaque points de data ainsi que la classe des points de data (la liste est triée par distance croissante). Le premier élément est la classe de p1 (permet de stocker l'information pour après)
- maxClasse(data,k) elle prend une liste de distance/classe et un entier k et renvoie une estimation de la classe du point dont la liste découle, basé sur les k-plus proches points.
- testTable et testTables2 qui effectuent l'estimation sur tous les points du dataset de vérification. Elles retournent une liste contenant l'estimation de la classe ainsi que la classe effective
- accuracy(data) elle prend une liste classeEstimée/classeEffective et renvoie le pourcentage de précision de cette liste. Elle renvoie en gros le pourcentage de précision de de l'estimation pour un entier k.
- multiTest et multiPreTest ce sont des fonctions finales qui font exécuter toutes le fonctions précédentes en boucle en itérant la valeur de k. Elles renvoient une liste avec la valeur de k et la précision associée. On peut en déduire quel est le k pour lequel l'estimation est la plus précise.
- J'ai par la suite rajouté une fonction normalise qui prend un dataset, et renvoie un nouveau dataset normalisé.

Des problèmes

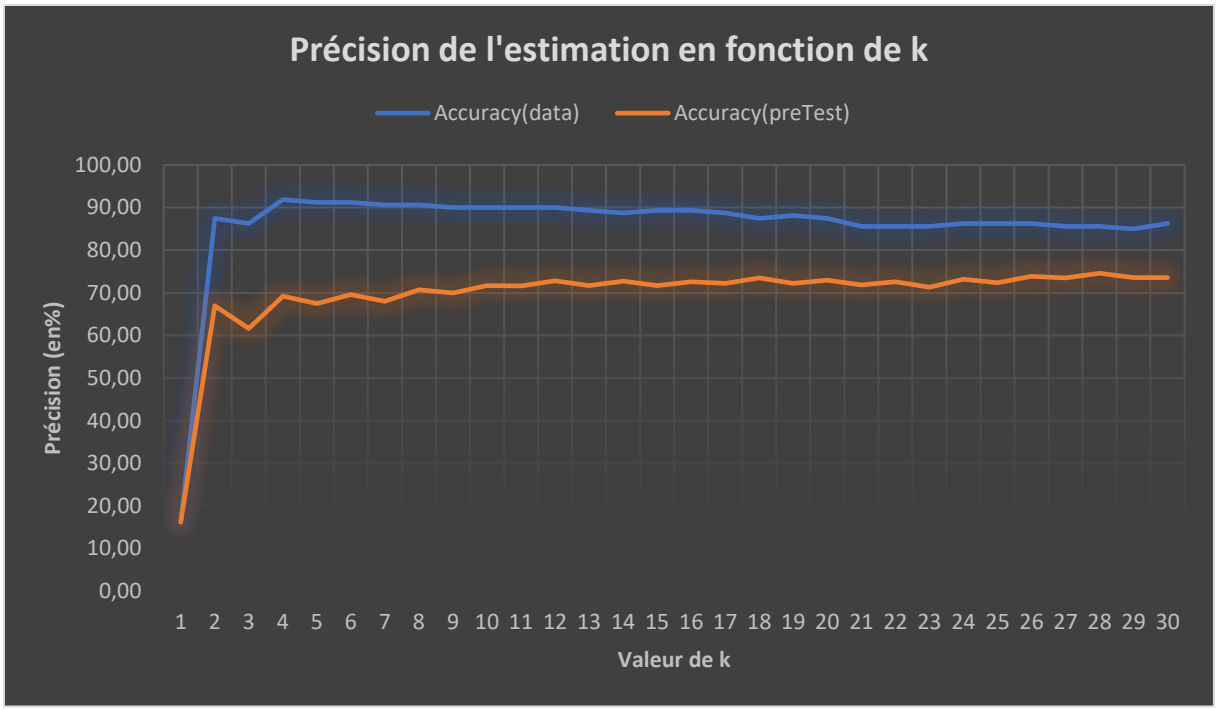
Comme dans tout projet, rien ne se passe comme prévu, j'ai rencontré quelques problèmes.

A vrai dire, je n'en ai rencontré qu'un seul vraiment important. Il était du à 2 erreurs combinées.

A la fin en utilisant multiPreTest, je me retrouvais avec la même valeur de précision pour plusieurs k différents. Cela était du au fait que dans multiPreTest, l'itération se faisait mal. Je n'itérais pas, je faisais directement un test pour k = 10 mais 10 fois d'affilée. Cela me donnait donc 10 fois le même résultat. Mais dans ce cas, comment ce fait-ce que je n'ai pas repéré cela plus tôt ?! Cela était du à une erreur dans la fonction sépare. Cette dernière avait une fonction shuffle (permet de mélanger les éléments de la liste) qui faisait doublon avec celle qui était présente juste après l'ouverture du fichier. A chaque fois que je testais avec un k différent, elle remélangeait l'échantillon. Cela avait pour effet de remélanger les valeurs et donc de changer la précision. Je pensais donc (à tort) que la précision différente était due au k alors qu'elle était due au mélange. Pour régler cela, j'ai donc enlevé le shuffle en trop et j'ai corrigé l'itérateur de ma fonction multiPreTest. J'ai eu relativement peu de problèmes, c'était du au test systématique de mes fonctions ainsi qu'à la « documentation ». En effet j'ai expliqué mes fonctions après les avoir créées, ce qui me permettait de savoir clairement ce qu'elles faisaient, en cas d'oubli il me suffisait de relire la documentation des fonctions.

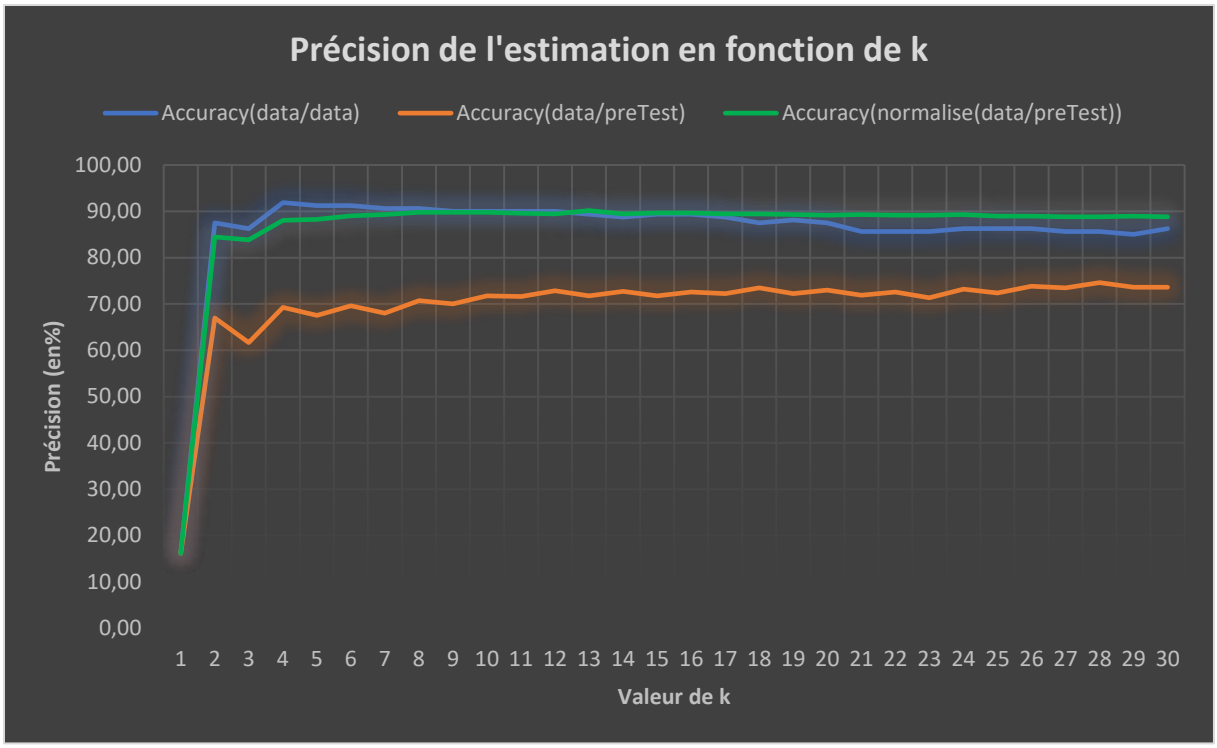
L'exploitation des résultats

Nous arrivons à la partie la plus intéressante ! L'exploitation des résultats. Mais d'abord un peu de théorie. Nous voulons identifier la classe d'un individu. Cela se fait en utilisant des données déjà connues. A savoir les classes d'individus déjà connus. Faisons une analogie avec les saumons. Nous voulons déterminer la race d'un saumon en se basant sur les saumons qui lui ressemblent le plus (taille, poids, masse graisseuse, couleur etc…). Jusque-là tout est logique. Disons que notre échantillon de saumon est constitué de saumons d'élevage. Maintenant prenons un saumon d'élevage et comparons-le à d'autres saumons d'élevage. On trouve une précision dans l'estimation de sa race assez élevée, car un saumon d'élevage ressemble à un autre saumon d'élevage. Cependant essayons de retrouver la race d'un saumon sauvage en le comparant aux saumons d'élevages. On se rend compte que l'estimation est beaucoup moins précise. C'est normal car pour une même race, le saumon sauvage sera certainement plus petit et avec beaucoup moins de masse graisseuse (car mal nourri et constamment en train de fuir les prédateurs). Il est donc plus difficile d'estimer la race du saumon. C'est même normal car on compare 2 échantillons ou datasets différents. Les données ont donc un biais d'un dataset à l'autre. C'est exactement ce que l'on retrouve dans le programme du knn. A la différence qu'ici data.csv correspond aux saumons d'élevage et preTest.csv correspond aux saumons sauvages.



On observe ici la précision de l'estimation en fonction du nombre de voisins pris en compte. Dans les 2 cas, on compare nos données à data.csv. On remarque que comparer data.csv à lui-même donne une précision allant jusqu'à 91.88% pour k = 4. En revanche comparer preTest.csv à data.csv revient à comparer des saumons sauvages et d'élevage. On a donc une précision maximale de 74.6 pour k = 28 (et seulement 69.24 pour k = 4). On pourrait se dire ici : « Il suffit de prendre k = 28 pour faire nos estimations ! ». Sauf que cela n'est pas possible, nous ne sommes pas sensés connaitre la classe de preTest.csv. On n'est donc normalement pas capable de dresser la courbe orange. Passer le k de 4 à 28 reviendrait à adapter le programme au dataset des saumons sauvages. C'est ce que l'on appelle du sur-apprentissage ! On ne peut donc pas fixer k = 28 car on n'est pas censés savoir que pour k = 28 cela nous donnerait plus de précision. Quand bien même on le savait, on fait du sur-apprentissage qui revient à adapter le programme aux résultats. Ce qui fait que si on change de dataset. On aura très probablement une précision encore plus faible. On est donc « obligés » de garder la valeur k = 4 même si cela revient à baisser la précision sur ce dataset précis.

On peut cependant aller bien plus loin ! Déjà on peut améliorer notre précision en agrandissant le dataset. Donc pourquoi ne pas utiliser lors du test final un dataset d'entraînement contenant data.csv et preTest.csv. On ne peut pas savoir si c'est bon, mais on peut se dire implicitement que plus de données donnerint de meilleurs résultats. On peut aussi normaliser nos données ! Cette étape est un préprocess des données et permet de les préparer à se faire traiter. On diminue la dispersion. En effectuant ce préprocessing. On passe d'une précision de 70% avec data.csv en entraînement et preTest en validation. A une précision de 90.16% pour k = 13 en effectuant le même test avec les même dataset normalisés. En actualisant le graphique précédent on obtient cette nouvelle courbe verte.



Pour me donner une idée de la répartition de chaque classe, je l'ai calculée. Ces valeurs correspondent aux proportions des différentes races de saumons si on reprend cette analogie.

- 13.43 % de la classA |403
- 18.53 % de la classB |556
- 30.83 % de la classC |925
- 30.30 % de la classD |909
- 6.90 % de la classE |207

Pour la solution finale il me suffit de modifier la fonction testTable à laquelle j'applique des dataset normalisés ainsi que le k optimal choisi (13 en l'occurrence). J'écris ensuite les valeurs dans un fichier .txt que je passe dans le vérificateur de M.Rodrigues pour m'assurer qu'elles ont le bon format.

Améliorations

Il existe beaucoup de méthodes pour améliorer la précision de l'estimation sans « tricher ».

La première qui est la plus intuitive est de trouver le K le plus performant, c'est ce que j'ai fait ici.

On peut aussi normaliser son dataset pour réduire la dispersion (je l'ai aussi fait). On remplace tous les x par : (x-xMin)/(xMax-xMin), on effectue cette opération sur les variables et non les individus. On pourrait aussi centrer et réduire les valeurs, mais la normalisation donne plus rapidement de meilleurs résultats.

Il serait aussi intéressant de donner un poids aux individus. Un individu plus lourd serait plus pris en compte qu'un plus léger.

On pourrait aussi penser à prendre en compte la distance. Exemple : 3 individus collés à celui dont on cherche la classe et 4 individus très éloignés. Dans ce cas, si on cherche avec 7 voisins, on prendra en compte la mauvaise classe juste parce qu'elle est plsu nombreuse malgré le fait qu'elle soit très éloignée.

Conclusion

C'était un projet très intéressant qui nous permet d'aborder les concepts de la datascience de façon plutôt ludique. Je n'ai aucune idée de la véracité de mes résultats, mais je peux me fier à la performance de mon programme sur les dataset précédents. Je ressors satisfait de ce projet.