|  |
| --- |
| Université du Québec à Rimouski |
| Travail Pratique I |
| INF11207 - Programmation orientée objet II  Professeur – Yacine Yaddaden, Ph. D. |

|  |
| --- |
| ⭢ Boutin, Frédérik  ⭢ Létourneau, Gabriel |

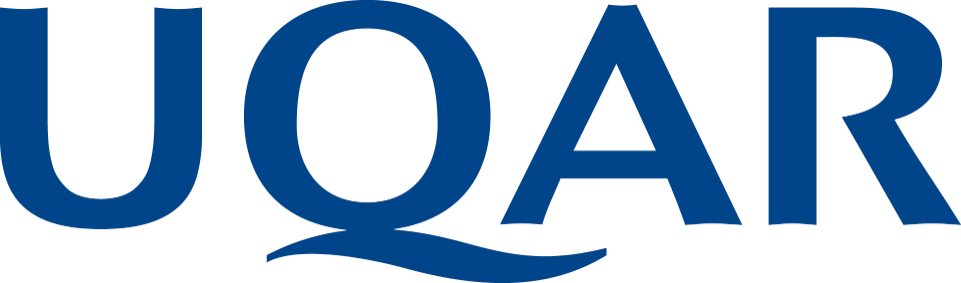


Table des matières

[Introduction 2](#_Toc97645476)

[La qualité du vin 2](#_Toc97645477)

[Algorithme du -NN et Résultats 2](#_Toc97645478)

[Utilisation du programme développé 2](#_Toc97645479)

[Problèmes et Difficultés rencontrées 3](#_Toc97645480)

[Conclusion 4](#_Toc97645481)

[Références 4](#_Toc97645482)

# Introduction

Le mandat de ce travail pratique 1 était la programmation d’un algorithme de mesure de la qualité de vins. Cet algorithme permettra de mettre en pratique les notions de programmation orientée objet, d’algorithmes de tri, d’interfaces et bien d’autres. Nous avions, à notre disposition, des fichiers ressources ainsi qu’un document de spécification. Ce dernier indiquait les notions nécessaires à la compréhension de l’algorithme. Les fichiers ressources étaient constitués, entre autres, de 30 fichiers csv (*samples)* représentant des vins individuels, du fichier *train.csv* contenant les vins de références ainsi que du fichier *test.csv*. Ce dernier était lui-même constitué de plusieurs échantillons de vins à tester avec notre algorithme pour en déterminer l’efficacité.

# La qualité du vin

Dans notre projet, il est possible de voir que nous avons bien implémenté chaque méthode des deux classes interfaces du document ressource (*IKNN* et *IWine*). Ainsi, toutes les signatures des méthodes identifiées par *main methods* ont bien été appelées à partir du *Main* et celles identifiées *utils* ont bien été appelées au sein de leur classe de référence. Notre programme obtient des résultats de qualités de vins qui sont précis à 65% du temps (pour un k = 10). Cette précision est la même que celle obtenue dans le document explicatif. Ce faible résultat vient du fait que notre algorithme compare uniquement une distance entre le vin à tester et les autres échantillons pour lesquels nous avons déjà la qualité. De plus, en fonction des paramètres de départ, l’algorithme choisi la qualité correspondant au plus proche échantillon du référentiel.

# Algorithme du -NN et Résultats

Nous avons construit notre algorithme *k-NN* avec des compteurs d’indices qualités. Ainsi, nous traitons 6 cas à l’aide de 3 compteurs de qualité, soit les compteurs 3, 6 et 9. Le premier cas est lorsque nous avons le compteur 3 qui est supérieur à ces homologues 6 et 9, le retour est alors une qualité 3. Le deuxième et troisième cas reprennent le même principe pour les compteurs 9 et 6 respectivement. Le quatrième cas indique que lorsque les compteurs 3 et 6 sont à égalité (et dont nous savons que le compteur 9 n’a pas la plus grande valeur de par la position de la condition dans la série de *if*, *if else*), le retour de qualité sera 3. De même, le cinquième cas lorsque les compteurs 3 et 9 sont à égalité, le retour est une qualité 3. Finalement, dans le sixième cas avec les compteurs 6 et 9 à égalité, c’est le résultat de qualité 6 qui est retourné. Nous avons également testé d’autres types d’arrondi de qualité, mais ce n’est qu’avec celui d’écrit ci-haut que nous avons obtenu une précision d’algorithme équivalente à celle du document explicatif.

# Utilisation du programme développé

Pour utiliser le programme, l’utilisateur devra entrer certains paramètres (-e ou -p, -t). Voici une description de tous les paramètres : **-e** pour spécifier le fichier csv contenant une liste de vins à faire évaluer par le programme, **-p** pour spécifier un fichier csv d’un seul vin à faire évaluer, **-t** pour spécifier la liste de vins qualifiée par un expert, **-k** pour indiquer le nombre de proches parents à considérer dans le *k-NN* du programme (1 par défaut), **-s** pour choisir l’algorithme de tri qui sera utilisé (tri par sélection par défaut), **-i** pour afficher visuellement la liste des vins et finalement **-h** ou **--help** pour faire afficher de l’aide.

Afin de mieux illustrer notre programme, deux scénarios serons présentés. Le premier scénario sera pour un choix de -p, donc lorsque le programme devra comparer un seul échantillon de vin avec des paramètres par défaut (k=1, tri par sélection). Le deuxième scénario sera pour un choix de -e avec une liste de vins csv, un -k = 4 et pour un -s = tri *Shell*.

Scénario 1 : Une fois les paramètres sélectionnés, le programme va localiser le bon échantillon de vin, puis un objet vin1 va appeler la méthode *ImportOneSample* avec le fichier csv du vin à tester. Cette méthode va transposer les informations csv du fichier dans un objet vin. Puis, la méthode *Predict* sera appelée avec l’objet vin, elle va créer les listes de distances euclidiennes et de résultats qualités, pour ensuite les utiliser dans l’appel de la méthode *EuclideanDistance* ainsi qu’avec le vin à tester. Le retour de la méthode *EuclideanDistance* sera une liste complète de chaque distance entre l’échantillon du vin par rapport à chaque vin de référence, ces deux listes seront envoyées, à leur tour, à la méthode *SelectionSort.* C’est cette méthode qui va trier la liste des distances euclidiennes de la plus petite à la plus grande et faire suivre les positions dans la liste des qualités (par rapport à la liste des distances). De retour à la méthode *Predict* avec les listes en ordre, la dernière méthode appelée sera *Vote* avec la liste des qualités et le k=1. Cette dernière va choisir le résultat de la qualité en fonction du compteur de valeur maximale. Puisque qu’ici k=1, il n’y aura qu’un seul des 3 compteurs qui sera incrémenté et qui retournera le résultat. La méthode *Predict* peut maintenant retourner, à son tour, le résultat de qualité à l’utilisateur.

Scénario 2 : Afin d’éviter les répétitions, nous allons uniquement mentionner les particularités qui se distinguent par rapport au scénario 1 et non toutes les étapes du scénario 2. Une fois les paramètres entrés et le fichier csv localisé par le logiciel, les vins seront affichés dans la console avec la méthode *PrintInfo* si le -i est spécifié dans les arguments, puis la méthode *Evaluate* sera appelée avec cette liste de vins. L’avantage de la méthode *Evaluate* vient du fait qu’elle va s’adapter à la particularité que c’est une liste de vins qui est comparée plutôt qu’un seul échantillon. En effet, des méthodes équivalentes au scénario 1 seront appelées, mais qui travaillent sur des listes. Ainsi, *Evaluate* va appeler *ImportAllSample* qui va retourner une liste d’objets Wine. *Evaluate* fera ensuite appel à *Predict*, pour comparer chaque vin de la liste avec ceux de la liste des comparatifs et les étapes du scénario 1 pourront se répéter. Cependant, dans notre cas, avec le tri Shell et pour 4 proches voisins. La liste des résultats qualités sera ensuite comparée en appelant la méthode *ConfusionMatrix.* Cette méthode va comparer les résultats qualités de l’algorithme avec ceux de l’expert et afficher la matrice résultante ainsi que le pourcentage d’efficacité du logiciel.

# Problèmes et Difficultés rencontrées

* La première difficulté a été la compréhension du problème et comment implémenter l’algorithme. Entre autres, à savoir que le fichier test permettait de tester la précision de l’algorithme, que le fichier train servait de valeurs références fournies par l’expert, que nous devions, à la fois, comparer notre train avec des échantillons de vins individuels et avec une liste de vins.
* Dans la première version du TP1, il y a eu une incompréhension par rapport à certaines signatures de méthodes retournant un objet de type interface (impossible d’instancier un objet d’une interface). Par la suite, avec la nouvelle version du TP1, il y a eu des changements dans les signatures de certaines méthodes et nous avons dû nous adapter à cette nouvelle disposition.
* Le code pour la construction de la matrice de confusion était, au départ, très chargé et ça simplification a nécessité plusieurs étapes. Bien que nous comprenions le principe de fonctionnement de la matrice de confusion, la transcription en code de manière épurée et dynamique nous a posé certaines difficultés.

# Conclusion

En conclusion, malgré les trois difficultés présentées ci-haut, nous avons été en mesure de programmer un algorithme répondant aux conditions du document explicatif. En effet, nous respectons les conditions d’appels des différentes méthodes pour les classes *INN* et *IWine*, tant par les paramètres d’entrées/sorties que par celui qui appel chaque méthode. De plus, le résultat de précision de l’algorithme ainsi que la matrice de confusion nous ont permis de confirmer que notre algorithme arrive aux mêmes résultats (à quelques différences près) que ceux présentés dans le document explicatif.

Voici le lien du vidéo explicatif du programme : <https://www.youtube.com/watch?v=uMORr2YBE10&ab_channel=wgabylouwgab>

# Références

Josh Close et al. (2021). *CsvHelper* (27.2.1) [logiciel]. NET library. <https://joshclose.github.io/CsvHelper/>

Yacine Yaddaden. (2022). Programmation orientée objet II [notes de cours]. Département de mathématiques, informatique et génie. Université du Québec à Rimouski. <https://portail.uqar.ca/course/view.php?id=21020>