Rapport TDM2 : Introduction aux sciences des données et à l'intelligence artificielle

Fanani Selma et Amsaf Rim (Groupe 4)

September 20, 2024

1 Introduction

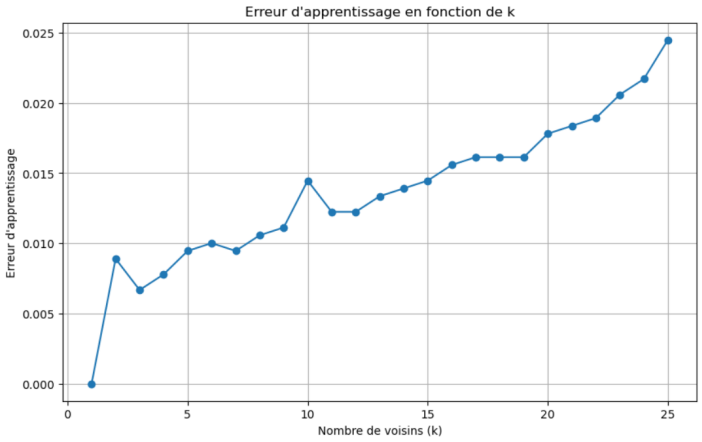
Pour ce deuxième travail dirigé, nous allons évaluer l'efficacité de deux algorithmes d'apprentissage sur deux jeux de données différents issus de `scikit-learn`.

Dans un premier temps, nous utiliserons le jeu de données `digits` pour tester l'algorithme des *k plus proches voisins* (*K-Nearest Neighbors*), en étudiant l'influence du paramètre k ainsi que la méthode de validation *hold-out*.

Dans un second temps, nous analyserons l'efficacité des arbres de décision sur le jeu de données `Iris`, en explorant l'impact des paramètres `random-state` et `max-depth`. Nous nous initierons également à deux stratégies de discrétisation des données.

2 Variation du nombre de voisins:

Dans cette partie, nous avons exploré l'influence du nombre de voisins k sur l'erreur d'apprentissage du classifieur k-plus proches voisins. Pour cela, nous avons fait varier k dans un intervalle de 1 à 25 et calculé l'erreur d'apprentissage pour chaque valeur.



En observant le graphique, nous remarquons que le classifieur atteint son meilleur taux de précision pour $k = 1$, avec une erreur d'apprentissage nulle. Cela s'explique par le fait que, pour $k = 1$, chaque point est comparé à lui-même, ce qui entraîne une classification parfaite sur les données d'entraînement. Cependant, cela peut induire un surapprentissage, où le modèle devient trop spécifique aux données qu'il a vues, ce qui peut nuire à ses performances sur de nouvelles données. À mesure que k augmente, l'erreur croît progressivement, car le classifieur devient plus général en prenant en compte plusieurs voisins pour prédire une classe. Sur ce jeu de données, des petites valeurs de k donnent des résultats précis mais spécifiques, tandis que des valeurs plus grandes aident à mieux généraliser, bien que l'erreur augmente légèrement.

3 Estimation de l'erreur réelle

Le score du classifieur sur l'ensemble d'apprentissage est souvent surévalué, car il peut mémoriser les particularités spécifiques des données d'entraînement, menant ainsi à un surapprentissage. Cela se traduit par une performance élevée sur les données d'apprentissage, mais ne reflète pas la véritable capacité du modèle à généraliser sur de nouvelles données. Pour cette raison, nous allons utiliser la méthode *hold-out* et évaluer notre modèle sur un ensemble de test distinct de celui utilisé pour l'apprentissage.

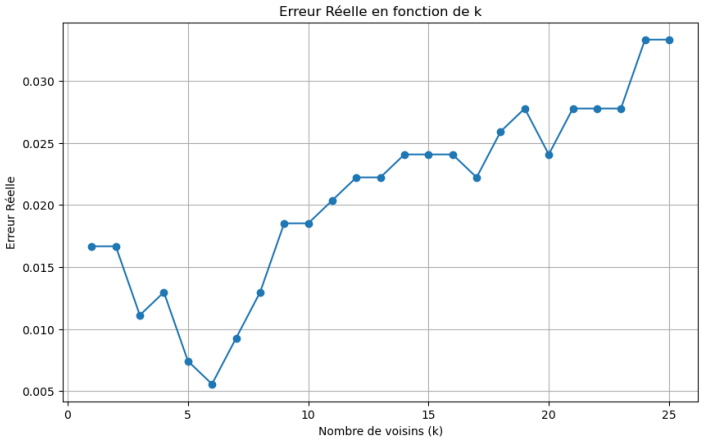
3.1 Estimation sur données hold-out

En premier temps, nous avons fait varier le paramètre `random_state` pour examiner comment la division des données en sous-ensembles d'apprentissage et de test affecte les prédictions du modèle. En analysant les premiers éléments de chaque ensemble, nous avons constaté que lors des deux premiers appels avec la même valeur de `random_state` (42), les ensembles d'entraînement et de test contenaient les mêmes données. Cela démontre que l'initialisation du générateur de nombres aléatoires permet de reproduire les mêmes résultats. En revanche, lors du troisième appel avec une valeur différente de `random_state` (99), les premiers éléments des ensembles sont différents, illustrant ainsi l'impact du changement de graine sur la répartition aléatoire des données.

Ensuite, nous avons divisé le jeu de données `digits` en deux parties égales afin d'évaluer la performance du classifieur des k *plus proches voisins* sur des données différentes de celles utilisées pour l'apprentissage du modèle.

Nous avons observé que le premier modèle, sans utilisation de la méthode *hold-out*, présentait une erreur d'apprentissage de 0,0067 pour $k = 3$. Contrairement au deuxième modèle, qui applique une méthode de découpage, et qui a produit une erreur réelle légèrement plus élevée. Ce qui s'explique par le fait que l'erreur réelle est calculée sur un ensemble de données que le modèle n'a pas utilisées lors de l'entraînement. Ce qui offre une meilleure évaluation de la capacité du modèle à généraliser sur de nouvelles données, contrairement à l'erreur d'apprentissage, qui peut sous-estimer cette capacité.

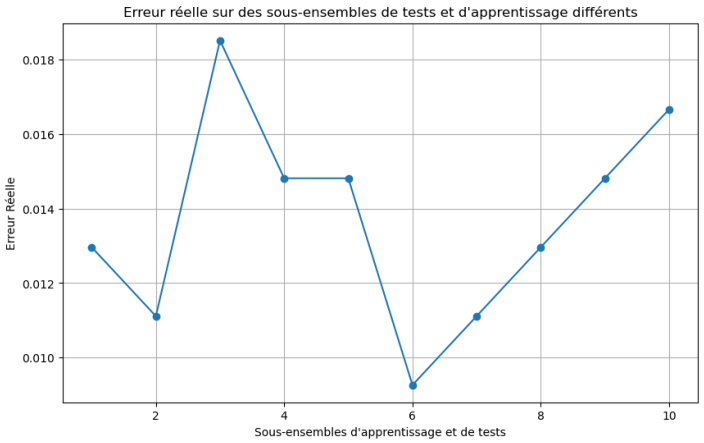
Finalement, nous avons entraîné notre modèle sur 70 % des données, et nous l'avons ensuite testé sur les 30 % des données restantes. En faisant varier la valeur de k, nous observons des variations de l'erreur réelle, comme le montre le graphique ci-dessous.



Nous observons que l'erreur réelle, obtenue en testant le modèle sur les 30 % des données restantes, est plus grande que l'erreur d'apprentissage obtenue pour une même valeur de k donnée.

L'écart entre l'erreur d'apprentissage et l'erreur réelle est souvent dû à la capacité de généralisation du modèle. Si l'erreur réelle est beaucoup plus grande que l'erreur d'apprentissage, cela peut indiquer que le modèle est en surapprentissage (il performe très bien sur les données d'entraînement mais moins bien sur des données nouvelles).

Nous allons maintenant créer 10 sous-ensembles d'entraînement et de test distincts en modifiant le paramètre random-state. Pour une valeur de k=3, nous obtenons des erreurs réelles différentes, calculées sur chaque modèle, comme le montre le graphique ci-dessous :



On observe que les erreurs réelles varient selon les sous-ensembles, car certaines divisions capturent mieux les caractéristiques clés des données, tandis que d'autres peuvent engendrer des partitions plus complexes. Lorsque la répartition des données d'entraînement est représentative de l'ensemble global, l'erreur réelle tend à être plus faible. En revanche, des répartitions moins représentatives entraînent une erreur réelle plus élevée.

4 Classification avec arbres de décision

Maintenant nous allons nous intéresser aux arbres de décision qui constituent une autre catégorie d'algorithmes de classification capables de prédire les classes de nouveaux exemples. Afin d'illustrer leur mise en œuvre dans scikit-learn, nous utiliserons le jeu de données Iris, que nous répartirons également en deux sous-ensembles : un pour l'entraînement et un pour le test.

4.1 Stabilité des arbres de décision

En premier lieu, nous allons évaluer les performance de l'arbre de décision sur le jeu de données Iris. Après la création de notre modèle nous obtenons un taux d'erreur de 0,0667 (6,67 %). Ce qui montre que l'arbre de décision fait quelques erreurs de prédictions, mais reste globalement performant. Pour améliorer la performance, il serait possible d'augmenter le paramètre max-depth qui représente la profondeur de l'arbre, tout en évitant le surapprentissage (overfitting), ce qui peut arriver si l'arbre est trop complexe.

4.2 Variation des hyper-paramètres:

En deuxième lieu, nous allons examiner l'impact des paramètres `random_state` et `max_depth` sur le taux d'erreur du modèle. Nos observations montrent qu'avec une profondeur fixée à 5 et un `random_state` de 42, le taux d'erreur est notablement plus élevé, ce qui suggère un phénomène de surapprentissage (overfitting). En revanche, lorsque nous utilisons une profondeur de 2, nous obtenons un meilleur score avec un `random_state` de 42 par rapport à un `random_state` de 2.

Cela indique que la profondeur plus faible permet au modèle de mieux généraliser sur les données de test, alors qu'un `random_state` différent peut affecter la manière dont les données sont divisées et, par conséquent, la performance du modèle. Ces résultats soulignent l'importance de l'ajustement des hyperparamètres pour optimiser la capacité de généralisation du modèle.

Il est également intéressant de noter que même en augmentant la valeur de `max_depth` à 10, nous obtenons le même arbre qu'avec `max_depth` égal à 5. Cela s'explique par le fait que l'arbre de décision a déjà atteint sa complexité maximale avec les données d'entraînement disponibles. En d'autres termes, les divisions effectuées à une profondeur de 5 sont suffisantes pour capturer les caractéristiques

essentielles des données, et des niveaux de profondeur supplémentaires ne contribuent pas à améliorer la structure de l'arbre. Cette observation souligne l'importance de choisir judicieusement les hyper-paramètres pour éviter un surajout de complexité inutile au modèle.

4.3 Discrétisation des attributs:

Pour augmenter les performances de notre modèle, nous avons testé deux stratégies de discrétisation des données : `uniform` et `quantile`. Avec la stratégie `uniform`, les bins sont répartis de manière égale en termes de plage de valeurs. Cela peut entraîner une répartition inégale des observations dans chaque bin, certaines se retrouvant isolées et d'autres concentrées. Par exemple, en appliquant cette méthode au vecteur

```
np.array([0.2, 0.3, 0.35, 0.36, 0.5, 0.6, 0.65, 9.1, 9.2]),
```

avec 4 bins, les valeurs faibles se retrouvent groupées dans le premier bin, tandis que les valeurs élevées; 9.1 et 9.2, sont isolées dans le dernier.

Avec la stratégie `quantile`, les bins sont ajustés pour contenir un nombre similaire d'observations, ce qui est plus adapté lorsque les données sont inégalement réparties. Ainsi, les premières valeurs du vecteur, comme 0.2 et 0.3, sont regroupées dans un même bin, puis 0.35 et 0.36 dans un autre, 0.5 et 0.65 dans le troisième bin, tandis que les trois dernières valeurs, 9.1 et 9.2, sont répartis dans le dernier bin, garantissant une meilleure répartition des observations dans chaque bin.

Nous allons maintenant discrétiser les données de notre jeu de données Iris en suivant la stratégie `uniform`. Les quatre attributs (longueur et largeur des sépales, longueur et largeur des pétales) seront transformés en trois catégories discrètes : 0 pour petite, 1 pour moyenne et 2 pour grande. Cette discrétisation sera réalisée en créant des bins de taille égale pour chaque attribut.

Une fois les données discrétisées, nous allons les diviser à nouveau en deux sous-ensembles de taille identique pour l'apprentissage et le test. Ensuite, nous apprendrons un arbre de décision sur ces données discrétisées, et nous visualiserons cet arbre.

Après avoir évalué l'arbre de décision sur l'ensemble de test, nous constatons que l'arbre appris sur les données discrétisées est plus simple, car les catégories discrètes réduisent la complexité des décisions. Les seuils continus des attributs sont remplacés par un nombre limité de catégories, ce qui rend l'arbre plus facile à interpréter.

Cependant, cette simplification entraîne une légère perte de précision. Sur les données d'origine, nous avions un taux d'erreur de 9,33%, tandis que l'arbre formé sur les données discrétisées montre un taux d'erreur de 9,70%. Bien que l'écart soit faible, cela illustre que la discrétisation peut parfois altérer la capacité du modèle à capturer les subtilités des données continues.

Sur les données discrétisées, nous obtenons le meilleur taux d'erreur avec un arbre de décision ayant une profondeur maximale de 2 et un *random-state* fixé à 42, avec un taux d'erreur de 0 %. Ce résultat est particulièrement optimal, indiquant que le modèle a parfaitement classifié l'ensemble de test. Toutefois, un taux d'erreur nul peut également être un signe de surapprentissage, surtout si l'ensemble de test est trop petit ou non représentatif de la diversité des données réelles. Il serait donc pertinent de vérifier la robustesse du modèle en le validant sur d'autres jeux de données ou via une méthode de validation croisée.

References

[1] Pandas Documentation. *Pandas User Guide*. https://pandas.pydata.org/docs/user_guide/index.html

[2] Scikit-learn Documentation. *Scikit-learn User Guide*. https://scikit-learn.org/stable/

[3] NumPy Documentation. *NumPy Linear Algebra Reference*. https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html

[4] Vous retrouverez le code directement en suivant ce lien. https://github.com/SelmaFanani/TDM2