RAPPORT KNN

Malo Bardin-Enderlin & Max Baumberger
ESILV A3 TD-B

Introduction

Ce rapport a pour objectif d'expliquer le fonctionnement de notre algorithme, nos idées, les problématiques rencontrées ainsi que les solutions trouvées pour y pallier. Notre rapport va donc être construit chronologiquement avec d'abord une explication de la v1 de notre code avec les différentes améliorations jusqu'à arriver à la version finale.

Pour la première version de notre code, nous sommes partis de notre code fait pour l'exercice sur le classement des fleurs. Nous avons donc commencé par retravailler ce dernier en modifiant notamment nos classes, les paramètres de nos fonctions mais également toute la gestion du stockage des points. Nous avons décidé d'avoir un code plus épuré et dynamique afin d'éviter de fixer des valeurs en dur, ce qui pourrait nous poser problème si nous souhaitons ré-utiliser l'algorithme KNN sur un autre projet.

Une fois l'algorithme KNN basique correctement implémenté ainsi qu'une structure de données correcte, il fallait désormais vérifier si tout fonctionnait avec les données du challenge. Pour cela, il fallait donc importer toutes les données d'un fichier CSV et pouvoir également écrire le résultat. Cette étape, qui nous semblait assez aisée, était un peu plus complexe que prévu puisqu'il fallait respecter une mise en page sur le fichier Excel de sortie afin de pouvoir l'uploader sur Kaggle par la suite.

Prédiction du taux d'erreur en interne

Une fois le code fonctionnel, nous avons obtenu un score d'approximativement 97.5 en cherchant à optimiser le k à la main. Nous avons ensuite voulu éviter de faire cette étape à la main et avons donc ajouté une boucle qui calcule le score potentiel des données du test afin de connaître en amont la précision de notre algorithme. Il suffit ensuite d'appeler notre algo en mo-

difiant le k, de récupérer le pourcentage de réussite pour chaque k et de récupérer le k le plus efficient.

Cette optimisation nous a permis d'obtenir le score de 97.756. La fonction de calcul d'erreur nous a été très utile car nous avons pu tester en amont nos optimisations avant de les uploader sur Kaggle afin d'éviter de "gaspiller" nos essais. Son fonctionnement est assez simple, elle prend chaque point de test et essaie de prédire sa catégorie. Elle vérifie ensuite son estimation avec l'information du label de chaque point contenue dans sa classe. En effectuant cela sur chaque point et en récupérant le nombre de faux, nous avions notre pourcentage d'erreurs pour un k donné.

Optimisation de la distance et pondération

Nous nous sommes ensuite tourné sur une autre forme d'optimisation, le calcul de la distance. En effet, le calcul de la distance classique (euclidienne) était notre premier choix car plus classique. Après quelques essais infructueux qui donnaient un moins bon pourcentage de réussite (notamment avec la distance de Manhattan), nous avons décidé de garder le calcul par la distance Euclidienne.

Frustrés de cet échec, nous avons souhaité tout de même continuer à travailler autour de cette distance afin d'accorder plus d'importances aux points les plus proches. De ce fait, nous avons ajouté de la pondération. Désormais le choix de la classe ne fait pas sur uniquement le nombre de points dans un rayon k mais sur des points pondérés par leur distance. Chaque point valait donc 1/distance**2. Nous avons évidemment fait plusieurs tests afin de déterminer s'il faut diviser par la distance au carré, la distance au cube, la racine de la distance ou juste la distance.

Normalisation des données

Une fois les fondements de l'algorithme finis, nous avons cherché des moyens de l'optimiser afin d'avoir de meilleurs résultats. Nous avons donc cherché sur différents forums et autres sites les techniques courantes d'optimisation de l'algorithme KNN et voilà ce qui est revenu : la normalisation et la pondération par la distance. Normalisation : Nous avons essayé différents types de normalisation de nos jeux de données :

- **MinMax**, qui place toutes les données entre 0 et 1 avec la formule (x - min(x)) / (max(x) - min(x))
- **IQR**, avec la formule (x - Q1) / IQR, où Q1 est le premier quartile et IQR = Q3 - Q1

— **Z-score**, qui convertit les données en écarts par rapport à la moyenne avec la formule (x - moyenne(x)) / écart-type(x)

Résultats : résultats sur notre jeu de données train (ici avec une pondération $1/\text{distance}^2$)

— **MinMax :** 98.91%, 11 erreurs
— **IQR :** 91.89%, 82 erreurs
— **Z-score :** 98.12%, 19 erreurs

MinMax a donné des résultats très encourageants dès le début, tandis que IQR et Z-score ont été moins efficaces (en particulier IQR qui était vraiment contre-productif). Nous avons également essayé de combiner les différentes normalisations mais ça n'a rien donné de pertinent (Z-score puis MinMax donnait les mêmes résultats que MinMax seul).

Nous avons donc décidé de garder la normalisation MinMax pour la suite.

Oversampling

Après ces méthodes classiques d'optimisation du KNN, nous avions toujours un problème non résolu : dans le fichier train, certaines populations étaient beaucoup plus grandes que d'autres :

— **0 :** 814 éléments
— **1 :** 46 éléments
— **2 :** 77 éléments
— **3 :** 72 éléments

Après des recherches plus approfondies, nous sommes tombés sur l'oversampling, une technique prometteuse dans notre cas. Nous avons donc décidé d'utiliser SMOTE pour l'oversampling avec la librairie imblearn (M. Rodrigues et notre chargée de TD Mme Allam nous ont tous les deux donné le feu vert pour utiliser cette extension).

Nous avons donc fait des essais avec SMOTE combiné avec les liens Tomek, ou avec ENN (Edited Nearest Neighbours) pour modifier le fichier train.csv afin d'obtenir un meilleur fichier de référence pour entraîner notre modèle KNN.

Fonctionnement de SMOTE, des liens Tomek et d'ENN

SMOTE (Synthetic Minority Oversampling TEchnique)

SMOTE fait de l'oversampling : il génère de nouveaux éléments qui ressemblent aux précédents dans les classes minoritaires, jusqu'à ce que toutes les populations soient égales.

Il fonctionne en 5 étapes :

1. Il sélectionne aléatoirement un point d'une des populations minoritaires.
2. Il sélectionne aléatoirement l'un de ses k-plus proches voisins (nous avons testé différents k, $k = 5$ nous donnait un score de **0.99012** sur Kaggle avec les liens Tomek et $1/\text{distance}^2$, et pareil pour $k = 3$).
3. Il génère aléatoirement un coefficient α entre 0 et 1 (non inclus).
4. Il crée un nouvel individu entre les deux points choisis, avec α la distance entre l'ancien et le nouveau :
   — $\alpha = 0.5 \Rightarrow$ point à mi-chemin entre les deux.
   — $\alpha = 0.2 \Rightarrow$ point 5 fois plus proche du premier que du deuxième point.
5. Il répète cette boucle autant de fois que nécessaire (automatique ou défini par l'utilisateur ; nous avons laissé ça sur automatique).

Liens Tomek

Les liens Tomek font de l'undersampling et servent à enlever des données de train les éléments ambigus :

— Ils se collent à la fin de l'algorithme SMOTE (donc se répètent à chaque fois que l'algorithme SMOTE boucle).
— Ils prennent un point au hasard (*) et le point le plus proche de lui : ces deux points deviennent un lien TOMEK.
— Si ces deux points sont de classes différentes, alors l'algorithme supprime le lien (donc supprime les deux individus qui formaient le lien).

(*) : En fonction des paramètres donnés dans la librairie, les points éligibles à être sélectionnés sont différents (source : documentation imblearn) :

— `'majority'` : resample only the majority class ;
— `'not minority'` : resample all classes but the minority class ;
— `'not majority'` : resample all classes but the majority class ;

— `'all'` : resample all classes ;
— `'auto'` : equivalent to `'not minority'`.

Nous avons fait des tests en `auto` et en `"not majority"` :

— Sur Kaggle, `auto` (`not minority`) a donné un score de **0.99024**.
— `Not majority` a donné un score de **0.99121**.

ENN (Edited Nearest Neighbors)

ENN (Edited Nearest Neighbors) fait aussi de l'undersampling : il sert également à enlever des éléments ambigus des données de train. Nous avons essayé ENN, mais les résultats étaient moins bons.

ENN fonctionne comme suit :
— Il prend un point au hasard (*) et les k points les plus proches de lui.
— Un vote est réalisé : si la majorité des points voisins appartiennent à une autre classe, alors le point est supprimé.

(*) : En fonction des paramètres donnés dans la librairie, les points éligibles à être sélectionnés sont différents (source : documentation imblearn) :

— `'majority'` : resample only the majority class ;
— `'not minority'` : resample all classes but the minority class ;
— `'not majority'` : resample all classes but the majority class ;
— `'all'` : resample all classes ;
— `'auto'` : equivalent to `'not minority'`.

Les résultats étant vraiment bien moins bons qu'avec Tomek en `auto`, nous n'avons pas approfondi cette méthode et nous nous sommes concentrés sur l'optimisation des liens Tomek.

Méthodes combinées

Nous avons essayé de combiner les liens Tomek et ENN grâce à une pipeline. En nous entraînant sur le jeu de données d'entraînement, nous avons obtenu 100% de réussite, une première, c'était très encourageant, donc nous avons uploadé ces résultats sur Kaggle.

Malheureusement, nous avons obtenu un score de **0.84780** (oui oui), nous avons peut-être fait surapprendre notre modèle... Grosse déception (on s'attendait à largement battre notre record de 0.99121), mais cela fait partie du processus.

(update : erreur de notre part, nous avions oublié de mettre le SMOTE dans la pipeline...)

Nous avons donc tenté un dernier essai avec SMOTE + Tomek + ENN grâce à une pipeline. Sur le jeu de données d'entraînement, nous avons obtenu 99.08% de réussite (vraiment peu par rapport à d'autres essais). Sur Kaggle, cet essai a obtenu un score de **0.97073**.

Conclusion

L'essai que nous gardons est donc celui de **0.99121** avec les caractéristiques suivantes :

- Utilisation de SMOTE et des liens Tomek
- k (de KNN) = 3
- Pondération de la distance : $1/\text{distance}^2$
- Normalisation : MinMax
- k (de SMOTE) = 3
- Méthode d'échantillonnage (de Tomek) : `'not majority'`

Références

[1] Article sur SMOTE, son fonctionnement et comment l'utiliser correctement : `https://kobia.fr/imbalanced-data-smote/`

[2] Article sur SMOTE et les liens TOMEK : `https://towardsdatascience.com/imbalanced-classification-in-python-smote-tom`

[3] Documentation d'imblearn sur les liens TOMEK : `https://imbalanced-learn.org/dev/references/generated/imblearn.under_sampling.TomekLinks.html`

[4] Documentation d'imblearn sur Edited Nearest Neighbors : `https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.EditedNearestNeighbours.html`

[5] Pondération d'un algorithme KNN : `https://www.geeksforgeeks.org/weighted-k-nn/`

[6] Article sur les différentes techniques d'undersampling : `https://machinelearningmastery.com/undersampling-algorithms-for-imbalanced-classification/`