MAP553

Dario Shariatian, Hugo Massonnat

18 septembre 2024

Challenge Kaggle

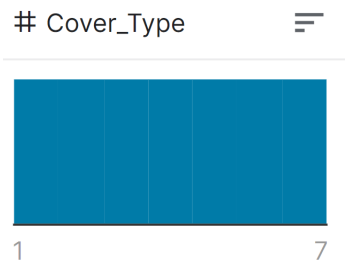
Introduction

Le problème posé lors de ce challenge Kaggle était celui de la classification non binaire (avec 7 classes différentes) à partir d'un jeu de données concernant des parcelles de forêt. Les données étaient réparties selon 55 colonnes, avec 11 variables numériques et 2 variables catégorielles (avec respectivement 4 et 40 classes). Pour aborder ce problème, nous avons déployé une stratégie décrite chronologiquement dans ce rapport. Après une phase d'exploration des données et des tentatives de réduction de dimension, nous avons essayé quelques algorithmes classiques de classification puis procédé à des optimisations lorsque c'était possible.

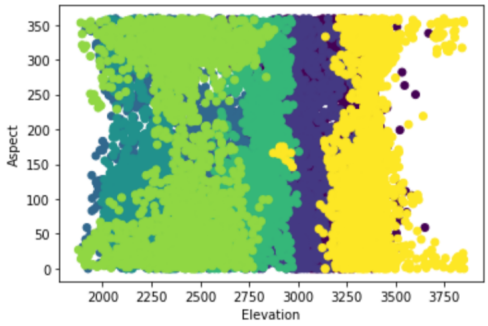
1 Exploration des données

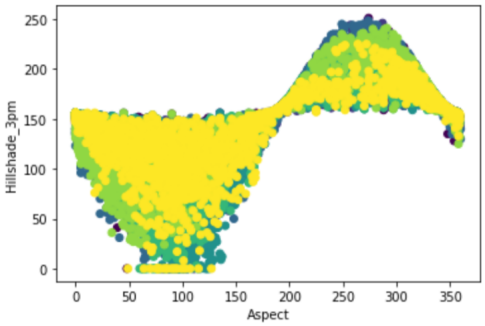
1.1 Visualisation des données

Dans un premier temps, nous avons exploré le jeu de données d'entrainement afin de se familiariser avec les différents paramètres. En étudiant leurs distributions, il apparait que les données d'entrainement sont réparties équitablement entre les 7 catégories de couverture forestière à calculer.



Nous avons également visualisé les dépendances entre les différents paramètres, ce qui peut orienter vers la création de combinaison de variables plus adaptées au problème. Cette étape nous apprend que l'altitude est une variable très importante pour différencier les différents types de couverture, et qu'il n'y a pas de corrélation importante entre les variables, sauf entre les taux d'exposition et la pente / orientation de la parcelle (la corrélation n'étant pas linéaire mais ressemblant plutôt à un sinus, ce que l'on pouvait deviner avec un modèle physique d'éclairement simple).





1.2 Variables catégorielles

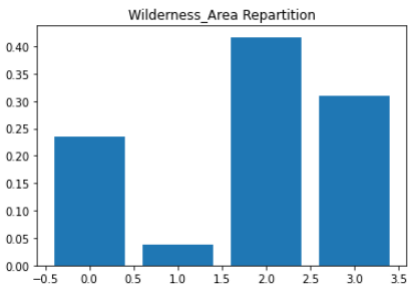
Nous n'avons pas trop touché aux variables continues, en comptant sur le fait qu'une PCA puisse indiquer les variables explicatives les plus pertinentes exprimées en tant que combinaison linéaires des features.

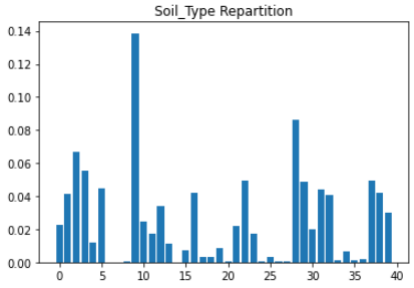
1.2.1 Dummy Coding

Tout d'abord, avec le dummy coding, afin d'éviter la colinéarité sur les colonnes, nous pouvons supprimer l'une d'entre elle parmi les variables d'une même catégorie (nous avons supprimé `Wilderness_Area_4` et `Soil_Type_40`). Les résultats restent peu changés.

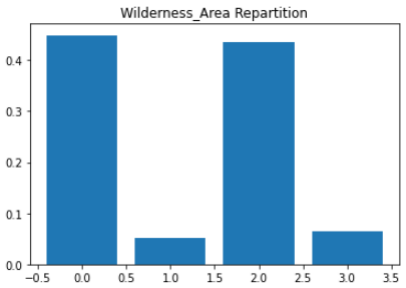
1.2.2 Variables ordinales

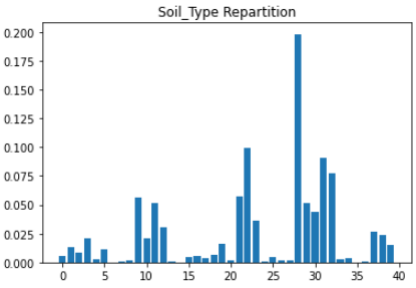
Remarquons que les 40 types de sols augmentent drastiquement la dimension des données. Une méthode pour la réduire est de décrire le type de sol avec une seule variable entière allant de 0 à 39. On peut tout de même se demander à quel point un algorithme de classification arrive à discriminer les différentes classes sur cette grille de nombreux entiers, là où peut-être l'utilisation de dimensions supplémentaires correspondrait mieux à son pouvoir discriminatif. Jetons un coup d'oeil à la répartition des types de `Soil` et de `Wilderness` sur notre ensemble `train` :





Faisons de même sur notre ensemble `test` :





On remarque une répartition relativement similaire pour `Wilderness` mais assez différente pour `Soil`. Pour ce dernier en revanche, on peut voir que de nombreux types de sols sont pratiquement inexistants (type de sol `14` inexistant dans `train`, nous l'enlèverons). On observe globalement cinq clusters de types de sols regroupant la plupart des observations. On propose alors la méthode suivante :

A partir du jeu `train` ou `test`, classer par un nombre entier les types de sols majoritaire représentant 90% de la distribution, et regrouper le reste des types en un dernier entier. On récupère ainsi globalement les mêmes types de sols majoritaires dans `train` et `test`, et on baisse de plus de moitié le nombre d'ordinaux à utiliser.

1.2.3 Autres méthodes possibles

Une autre méthode, le `frequency coding`, peut aussi être utilisé ; au lieu de spécifier le type de sol par un nombre entier, on donne la fréquence d'apparition dans tout le dataset (ce que l'on voit sur les tableaux précédents). La répartition étant relativement discriminante cela peut être une bonne idée.

Malheureusement aucune de ces méthodes n'offrent d'amélioration significative.

1.3 Normalisation

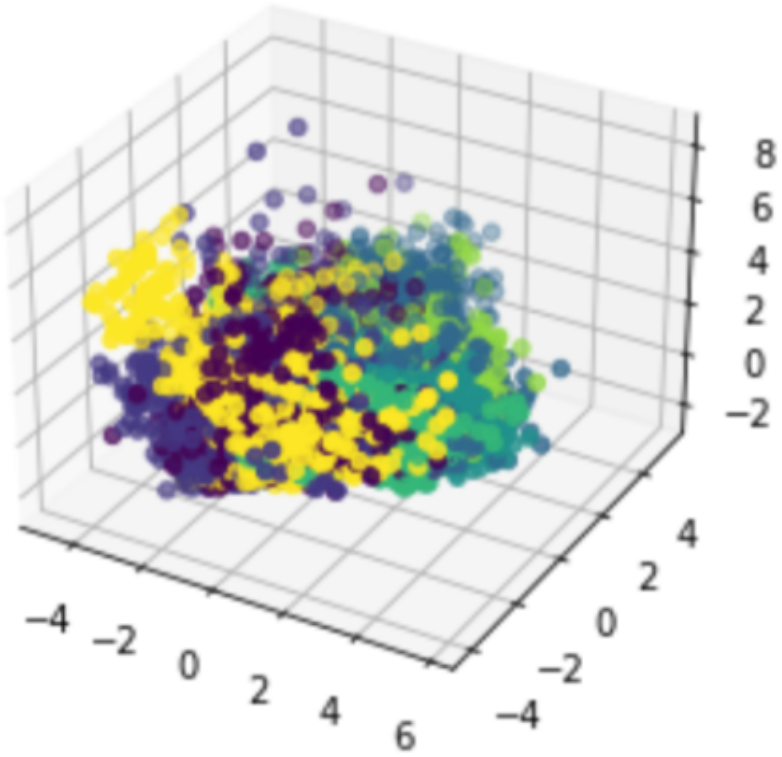
Certaines familles d'algorithmes prennent en entrée des données normalisées afin d'assurer leur convergence. Lorsque cela est nécessaire, on normalise les données colonne par colonne :

- Pour les **variables continues**, on applique la normalisation standard (retrancher la moyenne puis diviser par l'écart-type)
- Pour les **variables catégoriques**, si l'on utilise le *dummy coding* proposé dans le jeu d'entrainement, il faut diviser les colonnes associées à une même variable par le nombre de valeurs différentes que celle-ci peut prendre.

2 Réduction de dimension

2.1 PCA

Enfin, les données possèdent 55 paramètres si l'on utilise un *dummy coding* pour les variables catégoriques. Pour améliorer l'efficacité des algorithmes déployés par la suite, il peut être intéressant de déployer une technique de réduction de dimension telle que la PCA. En appliquant cet algorithme aux données normalisées, on obtient des résultats qui semblent prometteurs : les 10 premières composantes principales suffisent à expliquer plus de 99% de la variance totale. Cependant, nous avons observé par la suite que fournir ces composantes principales aux algorithmes de catégorisation menait à des baisses de performance. Ceci est peut-être dû au fait que la projection appliquée par la PCA mélange les données, comme cela semble être le cas au niveau des trois premières composantes principales.



Représentation 3D des trois premières composantes principales

Prenons un exemple clair où une PCA n'est pas adaptée. Considérons un problème de classification à deux classes sur $\mathbb{R}^2$ tel que :

$$(X = (X_0, X_1)|Y = 0) \sim \mathcal{U}(\{0\} \times [-1, 1])$$
$$(X = (X_0, X_1)|Y = 1) \sim \mathcal{U}(\{1\} \times [-1, 1])$$

Alors la variance des données est maximalement expliquée par la projection sur l'axe y, ce qui mélange totalement les données, alors que l'on avait très simplement $P(Y = 1|X_0) = X_0$...

Une projection par LDA peut être plus adapté dans le cas précédent. Ici par exemple, la projection sur la droite engendrée par la normale de l'hyperplan séparateur trouvé par LDA discriminera parfaitement les données.

2.2 LDA

Nous nous sommes proposé l'utilisation de LDA avant et après réduction initiale de dimension par feature engineering. Nous avons tenté une réduction de dimension avec une LDA multiclasse. Nous avons aussi tenté de travailler avec (C_i) un ensemble de 7 classifieurs tels que pour chaque i, C_i considère seulement la classe i contre le reste (OnevsRest Classification). Ni la réduction de dimension ni le classifieur n'ont donné de bons résultats.

Nous avons aussi essayé un QDA (donc avec plus de liberté sur les matrices de covariance des distributions des données selon les différentes classes). Les limites du travail précédent présageaient bien que l'hypothèse de distribution gaussienne n'était pas assez pertinente pour ce jeu donné, et effectivement les résultats étaient parmi les plus mauvais.

3 Cross-Validation et Ajustement des Hyper-Paramètres

Pour les points suivants, nous avons utilisé principalement deux méthodes :

3.1 Sélection de méthode par division Train/Validation/Test

Pour choisir les hyperparamètres de nos modèles et ainsi améliorer la précision et limiter l'*overfitting*, nous avons déployé une stratégie évoquée en cours :

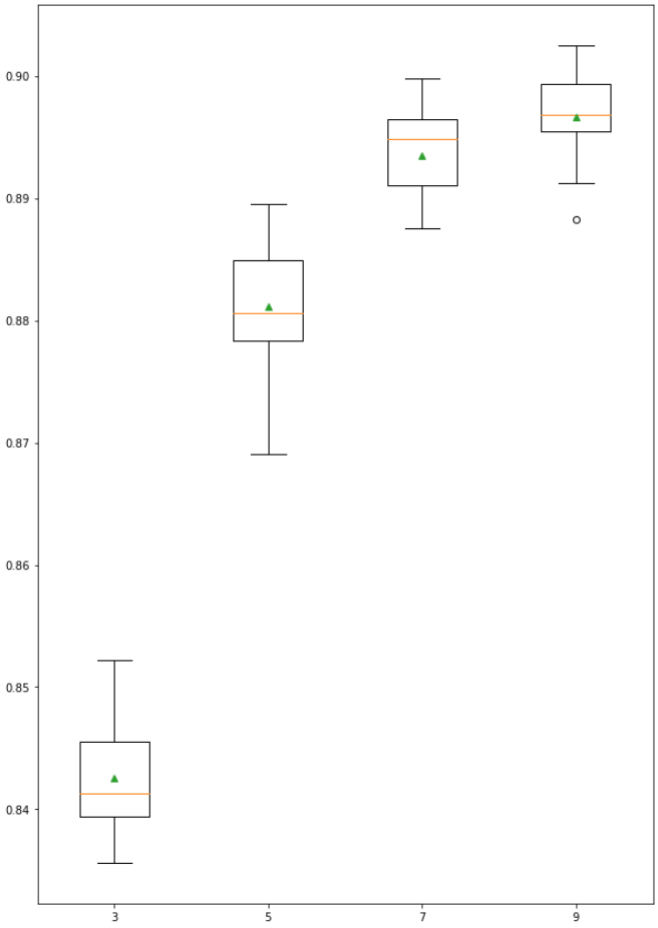
- Etude de l'influence de chaque hyperparamètre individuellement afin d'identifier des valeurs potentiellement intéressantes et limiter la taille de la *grid search* ;
- Séparation du jeu de données labellisées `train` en un jeu d'entraînement et un jeu de test grâce à la méthode `train_test_split` de `sklearn` ;
- Evaluation de la méthode pour chaque ensemble de paramètres en utilisant une *V-fold cross validation*. Cette étape a été recodée à la main pour pouvoir utiliser le paramètre `early_stopping_rounds` pendant l'ajustement du modèle et limiter les risques d'*overfitting*. On explore les possibilités avec une *grid search* qui devient rapidement très coûteuse, d'où le besoin d'avoir limité la zone de balayage des paramètres ;
- Sélection de la méthode : minimisation de l'erreur empirique moyenne obtenue après par *cross validation* (avec $V = 5$) menée sur 85% du jeu d'entraînement complet ;
- Evaluation des performances sur les 15% restants de données labellisées ;
- Entraînement du meilleur modèle obtenu sur toutes les données puis prédiction des données non labellisées.

Cette étape a permis une amélioration de la précision des modèles qui reste marginale, ce qui était attendu au vu de la difficulté de mener une *grid search* sur de nombreux paramètres. Nous avons dû nous limiter à deux ou trois valeurs différentes par paramètre, qui étaient nécessairement choisies de manière arbitraire.

3.2 Repeated (Stratified) K-Fold

Avec une simple *cross validation* telle que celle que nous avons codé, le résultat peut toutefois beaucoup dépendre de la manière dont sont divisées les données (i.e. le choix des ensembles de validation). Il est donc intéressant de répéter cette étape avec des divisions différentes puis de faire une moyenne des erreurs obtenues. Cela garantit de se rapprocher davantage de la véritable performance de la méthode sur le jeu d'entrainement. Lors de chaque *cross validation*, on peut également choisir de construire des couches stratifiées, qui ont la propriété de préserver la proportion des différentes classes. Cela est cohérent avec notre jeu de données `train` dont l'ensemble des éléments respecte cette équirépartition.

Peu importe la valeur de K (2,3,5,10), cette méthode ne prévenait que très peu l'overfitting. Par exemple voici le graphe de la précision d'un modèle de classifieur random forest comme approximé par cette méthode, avec comme variable la profondeur d'arbre maximale d :



Précision en fonction de la profondeur d'arbre maximale ($K = 10$)

Pourtant les performances réelles sur le jeu de données final était meilleures avec $d = 7$ qu'avec $d = 9$.

4 Forêts Aléatoires

Le premier algorithme de classification que nous avons implémenté est celui des forêts aléatoires, car il semblait adapté au problème et à notre connaissance de celui-ci. En effet, cette méthode fonctionne bien sur les données de haute dimension et peut manipuler des variables catégorielles et numériques sans besoin de normalisation. De plus, l'algorithme a tendance a faire de la *feature selection*, procurant ainsi des informations sur l'importance relative des variables.

Après un premier essai avec les paramètres par défaut du modèle (essentiellement ceux recommandés par le cours), nous avons utilisé les techniques de sélection de modèle détaillées précédemment pour essayer d'optimiser les hyper-paramètres. Nous avons mené une *grid search* à partir de quelques valeurs pour les paramètres qui semblaient les plus importants. Nous avons tenté d'améliorer l'entraînement du classifieur avec un bootstrap et en considérant l'out of bag error. Cela n'a toutefois pas permis d'améliorer la précision de manière significative.

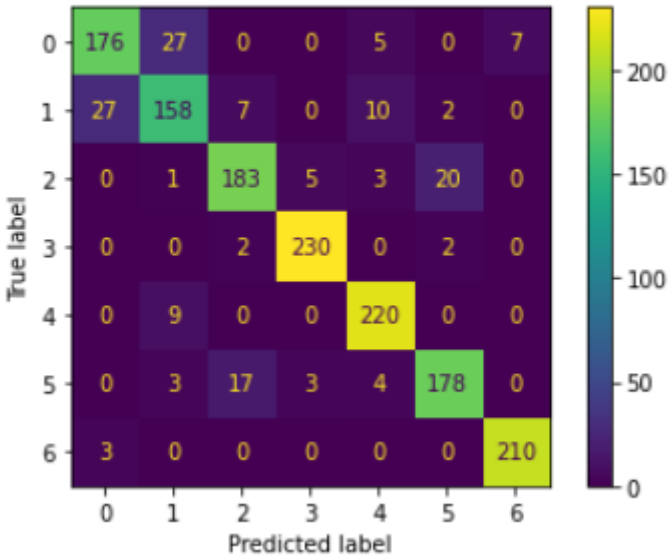
Malgré les différentes optimisations et des phases d'entraînement et de validation encourageantes, cette méthode n'a pas permis de dépasser une précision de 0.75 sur le jeu de test complet. Il semble que le modèle soit sujet à l'*overfitting*, mais il s'agit d'un problème difficile à corriger au vu de sa complexité théorique et de son caractère de "boîte noire". Par conséquent, nous avons décidé d'explorer une deuxième méthode avec d'autres atouts : XGBoost (ou Extreme Gradient Boosting).

5 XGBoost

5.1 Première paramétrisation

Nous nous sommes finalement tournés vers cet algorithme de gradient boosting connu pour être très efficace. Plusieurs paramètres intéressants facilement utilisables via `sklearn` permettent d'éviter l'overfitting comme la pénalisation ou la possibilité d'évaluer à chaque étape la perte sur un sous-ensemble `test` (que l'on aura récupéré par `train-test split` comme expliqué plus haut). Par ailleurs, le paramètre `early_stopping_rounds` permet d'arrêter l'entrainement du classifieur dès que la `log-loss` sur le sous-ensemble de `test` ne diminue plus après 5 rounds consécutifs. On se base également sur la précision (rapport entre les vrais positifs et le nombre total de prédictions) donnée à cette étape pour évaluer les hyper-paramètres avec une méthode de cross-validation plus générale (train validation test).

```
Logloss: 0.287779
Accuracy : 0.8961640211640212
```



Matrice de confusion du classifieur XGBoost optimal

5.2 Dernières optimisations

Un premier entraînement nous donne un bon modèle qui nous donne, selon kaggle, une précision de 80%. Cela est meilleur que tous les autres algorithmes. Nous nous proposons une petite méthode pour améliorer légèrement les résultats. Nous estimons la proportions d'occurences des différentes classes avec la prédiction précédemment obtenue. Ensuite, on réentraîne le modèle en assignant à chacune des données un poids correspondant à la fréquence d'occurence de son label. Ainsi, on met plus de poids à la mauvaise classification des échantillons des classes les plus représentées et on peut espérer ainsi augmenter au maximum la performance du classifieur à partir de notre petit jeu de données.

Effectivement cela fonctionne et c'est ainsi que nous obtenons les meilleures performances : environ 81.9% de prédictions correctes selon Kaggle.

Nous avons aussi essayé d'associer ce classifieur avec le random forest précédent. Les ayant d'abord fitté correctement, nous déterminons la classe associée à une observation en additionnant les probabilités assignées à chaque classe par les deux classifieurs et en considérant l'indice du résultat maximal. Cela n'a pas donné de meilleurs résultats.

Conclusion

Des méthodes de deep learning nous auraient peut-être permis d'aller plus loin en terme de performance face à ce challenge. Bien qu'étant une alternative intéressante, nous n'avions pas assez de temps, et nous manquions de connaissances préalables au sujet des frameworks Python utilisables à ce sujet.

Nous avons pu utiliser plusieurs méthodes plus aisément implémentable (principalement avec `scikit-learn`, `mlxtend` en python). Nous sommes passés par du *feature engineering*, de la réduction de dimension avec PCA et LDA, et des classifieurs utilisant des méthodes d'ensembles avec Random Forest et XGBoost. Plusieurs soumissions sur Kaggle et les tests menés sur nos machines ont montré que **XGBoost (81.9%)** donnait les meilleurs résultats, avec quelques optimisations et un *feature design* très limité. Toutes les autres méthodes ou mélanges de méthodes étaient moins efficaces.