Note méthodologique du Projet 7 : Implémenter un modèle de scoring

I. Contexte

En tant que Data Scientist au sein d'une société financière qui propose des crédits à la consommation pour des personnes ayant peu ou pas du tout d'historique de prêt, nous nous sommes demandés de mettre en œuvre un outil de "scoring crédit" pour calculer la probabilité qu'un client rembourse ou non son crédit.

De plus, les chargés de relation client ont fait remonter le fait que les clients sont de plus en plus demandeurs de transparence vis-à-vis des décisions d'octroi de crédit. Il est donc demandé de construire un tableau de bord interactif à destination des gestionnaires de la relation client permettant d'interpréter les prédictions faites par le modèle et d'améliorer la connaissance client des chargés de relation client.

Finalement, il est demandé d'utiliser un logiciel de version de code pour assurer l'intégration du modèle.

II. Méthodologie d'entraînement du modèle

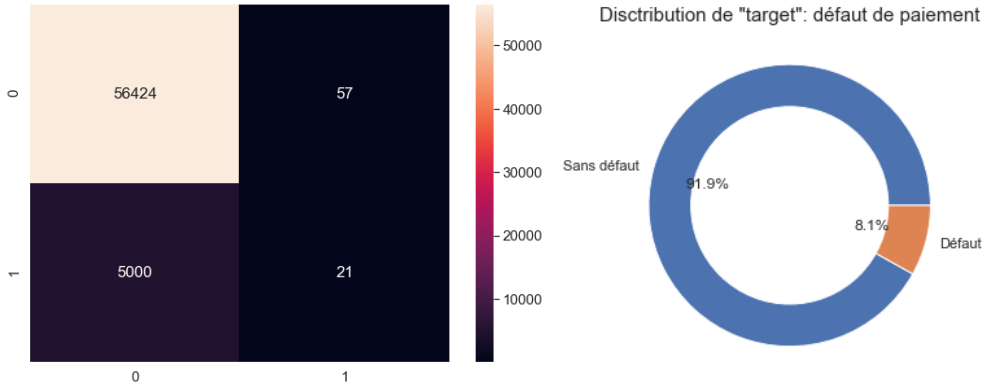
Pour commencer, nous avons besoin de connaître notre jeux de données. Selon les informations sur le site Home Credit Default Risk | Kaggle, j'ai sélectionné mon jeu de données principal : application_test qui décrit les informations basiques du client (sexe, âge, revenu total, prêt total, prix du bien, type de profession, type de contrat, durée du travail, statut familial, etc.), et le jeu de données bureau qui concerne l'historique des prêts des clients dans autres institutions financières.

Avant de commencer d'entraîner les modèles d'apprentissage automatique, il faut faire le nettoyage de données, surtout le traitement des valeurs manquantes et des valeurs aberrantes. Pour identifier les valeurs nulles, j'ai regardé le taux de taux de remplissage de chaque colonne. J'ai observé que les colonnes avec un taux de remplissage bas sont toutes les colonnes du type « objet ». J'ai donc utilisé la fonction « replace » pour remplacer les valeurs nulles de ces colonnes par « Not informed ». Pour les colonnes du type numérique, j'ai remplacé les valeurs nulles par la valeur -1. Concernant les valeurs aberrantes, j'ai utilisé la fonction « describe » afin de regarder la distribution de données pour les colonnes numériques, et j'ai découvert quelques anomalies. Par exemple, la colonne « days employed » a des valeurs supérieurs à 36 500, c'est impossible puisque personne pourrait travailler pendant plus de 100 ans. Je les ai donc retirés du jeu de données. J'ai également défini une fonction pour observer les valeurs uniques de chaque colonne et nettoyer les valeur ambiguës si besoin. Par exemple, la colonne « code_gender » a trois valeurs unique : male, female et XNA. La troisième valeur XNA nous apporte rien comme information utile, je les ai donc retirées aussi.

La prochaine étape serait de faire des traitements nécessaires pour je jeu de données de « bureau ». Dans ce jeu de données, chaque ligne représente un crédit historique pour un client. Il est donc possible que pour un client donné, ce dernier a plusieurs lignes qui le concernent. En revanche, si le client n'a pas d'historique de prêt, alors aucune ligne le concerne. Afin de centrer les informations dans ce jeu de données, j'ai utilisé la fonction « groupby » par l'identification du client et a fait « sum » et « count » comme fonction d'agrégation. Une fois que j'ai le jeu de données regroupé, j'ai fait une jointure à gauche du jeu de données principal et celui du bureau. Une jointure à gauche va sauvegarder toutes les informations du jeu de données à gauche (application_test) et utiliser le jeu de données à droite (bureau) pour compléter les informations. S'il n'y a pas d'information à mapper, alors cette valeur sera laissé à 0.

L'étape suivante est de faire le prétraitement de colonnes. Pour pouvoir entraîner les modèles de machine learning, il est souvent nécessaire de faire quelques traitements. Souvent, il nécessite deux choses : faire une échelle type pour les colonnes numériques afin de faciliter la comparaison, et transformer les colonnes d'objet en numérique pour que les modèles reconnaissent. Les deux fonctions que j'ai utilisées est la fonction « standard scaler » et la fonction « one hot encoder ».

Une fois que j'ai le jeu de données prêt à être modélisé, j'ai fait un test avec « dummy classifier », le modèle qui utilise les règles simples pour faire les prédictions. Il est souvent utilisé comme base de référence simple à comparer avec d'autres classificateurs réels. Avec ce classificateur, j'ai eu un très bon résultat de 91% de prédiction correcte. En observant la matrix de confusion (image 1) :

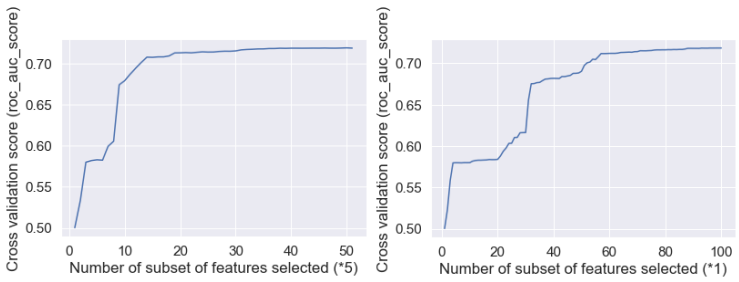


J'ai découvert que, comme 91% de nos données sont étiquetées 0 (image 2), si le modèle prédit tous en 0, alors nous avons un score de précision de 91%. Cependant, le modèle ne prédit pas la fraude correctement, alors que c'est la fraude qui nous intéresse le plus. Par conséquent, avant d'entraîner les vrais modèles, nous avons besoin de rendre notre jeu de données moins « déséquilibré ».

Trois approches pour rééquilibrer les deux classes ont été testées :

- Over-sample : augmenter le nombre d'échantillon pour la classe minoritaire
- Under-sample : diminuer le nombre d'échantillon pour la classe majoritaire
- SMOTE : créer les données synthétiques à partir des données existantes

Ainsi, nous avons un jeu de données plus équilibré. Mais nous avons un autre problème, c'est que nous avons trop de variables : plus de 300. Un nombre de variables trop important conduit souvent à un surapprentissage du modèle et de mauvaise performances dans le monde réel. Il existe beaucoup de méthodes pour réduire le nombre de variables tout en gardant les informations principale. Dans ce projet, j'ai utilisé la méthode RFE GRID. C'est une méthode qui nous permet d'observer la performance du modèle selon le nombre de variables :



Depuis la graphique, nous pouvons constater que la performance du modèle stagne à partir du 100^{e} (20*5) variables. J'ai ensuite réappliqué le RFE GRID sur les 100 variables. Le nombre de variable final s'élève à 40.

Maintenant, nous pouvons tester les différents modèles avec recherche d'hyperparamètres. J'ai entrainé les modèles suivants :

- Light GBM Classifier de SkLearn
- Light GBM Booster
- Random Forest Classifier
- Logistic Regression Classifier

Parmi ces modèles, Light GBM Booster à de meilleurs résultats. Néanmoins, dans l'optique de concevoir une API dans le web avec le modèle final, cette méthode est assez compliqué de déployer puisqu'elle nécessite des traitements spéciaux. Pour cette raison, j'ai testé le modèle Catboosting. C'est un modèle qui n'a pas besoin de prétraitement en amont et il a de très bonne performance. Avec la méthode de features_importances_, j'ai eu les 30 variables les plus importantes dans la prédiction. Avec ces 30 variables, la performance de prédiction n'a pas changé.

III. Fonction de coût, algorithme d'optimisation et métrique d'évaluation

La fonction de coût est réalisé avec une approche de l'évolution de coûts potentiels avec le changement de seuil de prédiction. Pour être plus précis, le modèle de machine learning ne prédit pas directement si notre cible est 0 ou 1 mais une probabilité entre 0 et 1. Si on décide que le seuil est à 0.5 alors toutes les prédictions inférieures à 0.5 seront considérées comme 0 (peu de risque) et l'inverse à 1 (risqué). Après quelques discussions avec les personnes du métiers, les faux négatives (le client est un mauvais payeur alors que le modèle le prédit comme un bon payeur) coûtent beaucoup plus chers à l'entreprise que les faux positives (le client est un bon payeur alors que le modèle le prédit comme un mauvais payeur). J'ai donc fait l'hypothèse que les faux négatives coûtent 5 fois plus chers à l'entreprise que les faux positives. J'ai donc un graphique pour représenter le résultat :