MEMOIRE METHODOLOGIQUE

Project 7 -
Parcours Data
Scientist
«Implémentez un
modèle de scoring »

1. Contexte

Ce mémoire constitue l'un des livrables du projet «Implémentez un modèle de scoring » du parcours Data Scientist d'Openclassrooms. Il présente le processus de modélisation et d'interprétabilité du modèle mis en place dans le cadre du projet.

Le projet consiste à développer pour la société « Prêt à Dépenser », une société de crédit de consommation, un modèle de scoring de la probabilité de défaut de paiement d'un client avec pas ou peu d'historique de prêt.

Les données utilisées pour ce projet sont une base de données de 307 000 clients comportants 121 features (âge, sexe, emploi, logement, revenus, informations relatives au crédit, notation externe, etc.)

2. Méthodologie d'entrainement du modèle

L'entreprise met à disposition 10 fichiers CSV contenant des données spécifiques à certains paramètres disponible a cette adresse https://www.kaggle.com/c/home-credit-default-risk/data

Nous disposons d'une base de données nommée « train », qui nous a servi à entraîner notre modèle. Cette base contient une variable « cible ».

Nous avons aussi à disposition une base de données « test ». Cette ne nous a pas été utile à la construction et à l'entraînement du modèle. Son utilisation a été extérieure au sujet de la présente note technique.

Dans cette partie, nous présentons brièvement le Kernel Kaggle que nous avons récupéré, afin de comprendre la base qui a été réalisé et sur laquelle repose notre modèle.
Kernel Kaggle :
Avant toute chose, il est bon de savoir que les travaux réalisés dans ce Kernel ont été fait uniquement sur la même base de données, à savoir « train.csv ».

- Rappel : "test.csv" est le dataset que nous utilisons pour simuler un nouveau client dans la base. Toutefois il convient que ces deux datasets aient la même structure à l'issu du feature engineering
- Traitement par imputation de la médiane
- Label encoding pour les variables à 2 catégories.
- One Hot Encoding pour les variables à plus de deux catégories.
- Alignement des datasets "train" et "test" pour conserver des structures identiques.
- Remplacement des outliers par des valeurs nulles. *Ensuite les valeurs sont imputées par la médiane (Voir partie - Preprocessing).*
- Ajout d'une "flag feature" pour identifier les lignes qui contiennent les outliers
- Création de deux hypothèses de feature engineering :
    - *"Polynomial Features"* : Amélioration de la correlation des variables EXT SOURCES avec la target
    - *"Domain Features"* : Construction de variables s'appliquant plus au domaine de la banque comme :

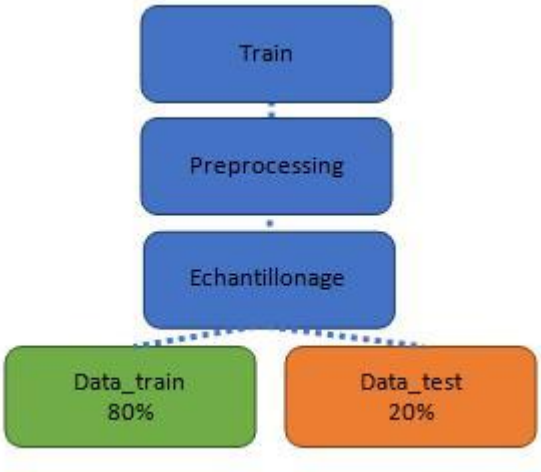
      "CREDIT_INCOME_PERCENT","ANNUITY_INCOME_PERCENT","CREDIT_TERM","DAYS_EMPLOY ED_PERCENT

Pour rappel, l'entraînement du modèle se fait uniquement sur le fichier « train.csv ». Le preprocessing est constitué de deux traitements :

- Une transformation des données comprenant :

- Une imputation des valeurs manquantes créées lors tu traitement des outliers.
- Une application d'un MinMaxScaler sur le dataset.

- Un échantillonnage du dataset. Cette partie a été réalisée avec la méthode « Train_Test_Split » de Scikit-learn. Avec un ratio de 80% pour les données d'entraînement et 20% pour les données de tests.



3.Le traitement du déséquilibre des classes

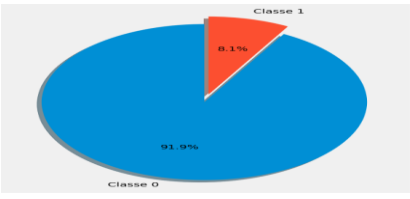
Le principal problème dans notre cas de figure se situe dans le déséquilibre des *targets*. Ici, nous avons à faire à une classification binaire dans laquelle la classe 0 représente les personnes qui ont payé leur crédit, et la classe 1, les personnes ayant rencontré des problèmes pour rembourser leur prêt.
Afin d'éviter de construire un modèle ne prédisant que la classe majoritaire, il convient d'entraîner ce modèle sur un jeu de données équilibré. Pour cela nous avons utilisé une librairie Python nommée *Imblearn [4]*.
Technique utilisée dans la librairie *Imblearn* : **Random Under Sampler** Il existe plusieurs techniques de sampling de données. Parmi les plus efficaces, on trouve le Tomek links pour l'under-sampling et le SMOTE pour l'over-sampling. Dans notre cas, dans un souci d'optimisation de temps de calculs, nous avons utilisé le Random Under Sampler qui s'avère le plus rapide.
Sur la figure gauche ci-dessous, on constate le déséquilibre initial entre les deux classes. Tandis que sur la figure droite, nous avons rééquilibré parfaitement les 2 classe

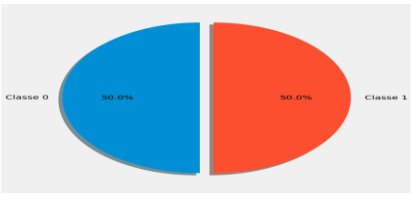
	Classe 0	**Classe 1**
Avant Rééquilibrage	226038	19970
Après Rééquilibrage	24825	24825

3. Fonction coût, algorithme d'optimisation et métrique d'évaluation

Dans le contexte bancaire, ce qui est important c'est de minimiser le risque financier lié à l'insolvabilité des clients en d'autres termes c'est de réduire le taux des faux négatifs des clients à défaut qui sont prédits comme des bons clients .

	Clients prédits en défaut	Clients prédits sans défaut
Clients réellement en défaut	Vrais positifs	**Faux négatifs**
Clients sans défaut	Faux positifs	Vrais négatifs





La courbe ROC représente le taux de vrais positifs (TPR) par rapport au taux de faux positifs (FPR). La métrique ROC_AUC correspond à l'aire sous la courbe ROC, elle est comprise entre 0 et 1.

Contrairement à La métrique ROC_AUC qui ne tient pas en compte le seuil de classifcation. Une autre métrique sera utilisée dans ce sens qui est le fbeta score . Cette métrique permet de défnir le poids qu'on souhaite attribuer au recall ou à la précision. Pour rappel les formules de recall et précision sont les suivantes:

$$recall = \frac{Vrai\ Positif}{Vrai\ Positif + Faux\ Négatif} \qquad precision = \frac{Vrai\ Positif}{Vrai\ Positif + Faux\ Positif}$$

- La Précision : Ce coefficient détermine que, quand le classifieur déclare que la prédiction est un 1, il a raison à X%.
- Le Recall : Ce coefficient détermine le pourcentage de détection des 1 du classifieur.

Comme mentionné déjà , on préfère limiter un risque fnancier plutôt que le risque de perdre un client donc le recall est plus important que la précision pour exprimer

4 Modèle testé et choix final

Dans le cadre de la présente problématique ,différents modèles de classifcation ont été testés LogisticRegression, RandomForestClassifer et XGBClassifer.

Parmi ces modèles, les meilleurs résultats sont obtenus avec XGBClassifer qui est un algorithme de classifcation binaire.

MODELE XGBOOST

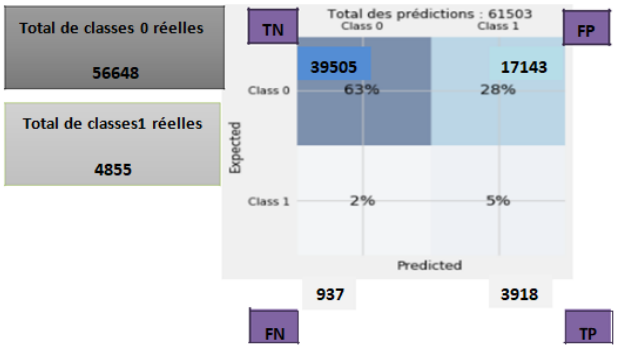
Après étude de plusieurs modèles de classification, notre choix s'est porté sur le XGBoost. Voici les premiers résultats obtenus pour la première tentative de prédiction.

	Accuracy	Precision	Recall	F1_score
Logistic regression	0.69	0.16	0.68	0.26
Random Forest	0.68	0.17	0.76	0.28
XGBoost	0.71	0.19	0.81	0.3

ANALYSE : Les 3 modèles présentent des résultats presque similaires, et ceux-ci ne sont pas bons. On pourrait traduire ces résultats en disant que notre modèle arrive à trouver +80% des classes 1, mais que, quand il prédit une classe 1, il n'a raison que dans +16% des cas.

La matrice de confusion :

PRINCIPE : La matrice de confusion consiste à compter le nombre de fois où des observations de la classe 0 ont été rangées dans la classe 1. Par exemple, si nous voulons connaître le nombre de fois où le classifieur à bien réussi à classer une classe 1, on examinera la cellule à l'intersection de la ligne 1 et de la colonne 1. Celui qui nous intérèsse est le false négative plus il est bas mieux est le résultat



Le modèle XGBoost possède trois familles d'hyperparamètres :

1. General Parameters : Pour régler les préférences du modèle
2. Booster Parameters : Pour régler le modèle à chaque étape de prédiction (Arbre / Régression)
3. Learning TaskParameters : Pour régler le type de prédiction que l'on souhaite

Il existe un grand nombre d'hyperparamètres pour ce modèle, et, à ce stade, nous n'en avons utilisé que quelques-uns : utilisation de la fonction RandomizedSearchCV pour trouver la meilleur itération pour notre modèle.

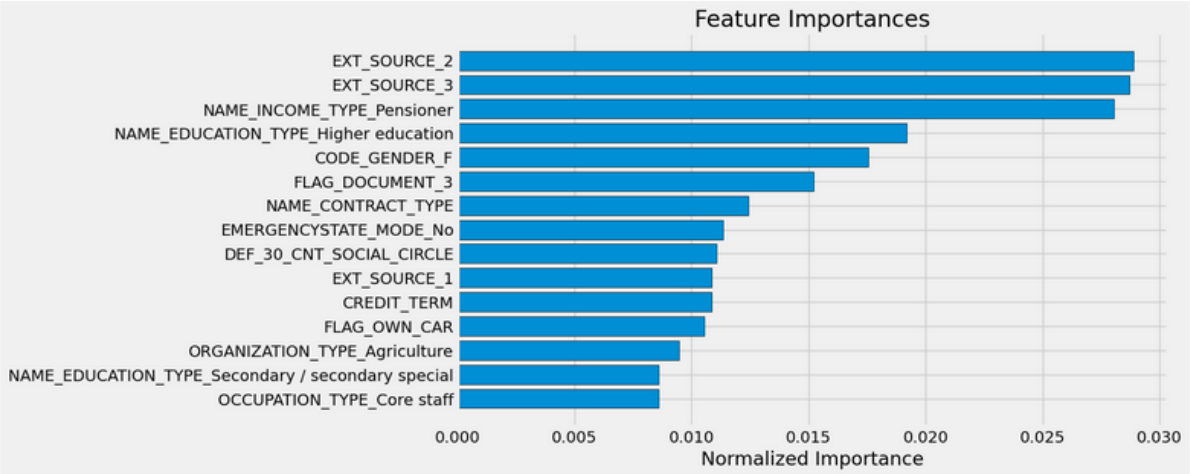
```
clf_xgb_opt = xgb.XGBClassifier(objective='binary:logistic',
                                eval_metric= "auc")

param_dist = {'n_estimators': stats.randint(150, 1000),
              'learning_rate': stats.uniform(0.01, 0.6),
              'subsample': stats.uniform(0.3, 0.9),
              'max_depth': [3, 4, 5, 6, 7, 8, 9],
              'colsample_bytree': stats.uniform(0.5, 0.9),
              'min_child_weight': [1, 2, 3, 4],
              }
xgb_opt = RandomizedSearchCV(clf_xgb_opt,param_distributions=param_dist,
                             n_iter=10,scoring='roc_auc', n_jobs=-1,cv=5,verbose=3)
```

Meilleur paramètre trouvés :

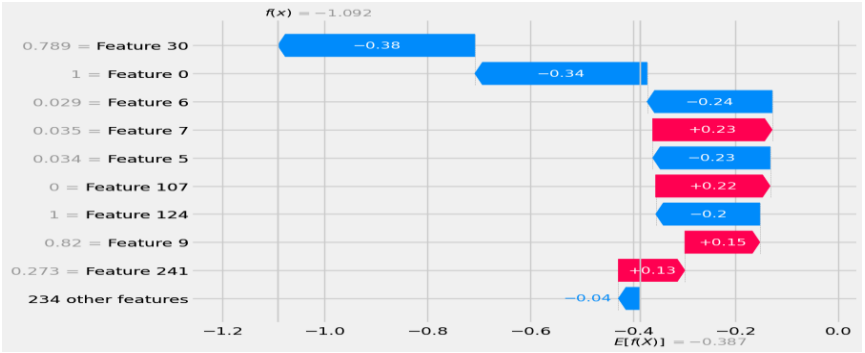
```
The best estimator across ALL searched params:
XGBClassifier(base_score=0.5, booster='gbtree', callbacks=None,
              colsample_bylevel=1, colsample_bynode=1,
              colsample_bytree=0.5949628587717157, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='auc', gamma=0, gpu_id=-1,
              grow_policy='depthwise', importance_type=None,
              interaction_constraints='', learning_rate=0.06911353948307965,
              max_bin=256, max_cat_to_onehot=4, max_delta_step=0, max_depth=4,
              max_leaves=0, min_child_weight=4, missing=nan,
              monotone_constraints='()', n_estimators=896, n_jobs=0,
              num_parallel_tree=1, predictor='auto', random_state=0,
              reg_alpha=0, reg_lambda=1, ...)

The best score across ALL searched params:
0.7587763853520464

The best parameters across ALL searched params:
{'colsample_bytree': 0.5949628587717157, 'learning_rate': 0.06911353948307965, 'max_depth': 4, 'min_child_weight': 4, 'n_
estimators': 896, 'subsample': 0.3740770236722674}
```

L'interprétabilité globale et local du modèle :

Pour l'interprétabilité du modèle qui sera destinée soit à l'équipe métier soit aux clients , on a utilisé deux méthodes :
-Une méthode d'interprétabilité globale qui tiendra en compte l'ensemble des données et qui dépendra du modèle choisi dans notre cas le XGBClassifier. notre modèle nous donne l'importance des variables suivantes :



-Une méthode d'interprétabilité locale c'est à dire pour chaque client choisit on cherche à savoir les variable qui ont influencé le plus sur le score final n a utilisé shap ci-joint un exemple d'interprétabilité local pour une seule observation (un seul client):



Axes d'améliorations a apporter

Nous avons pu constater tout au long de ce document que les performances du modèle ne sont pas bonnes. Pour résumé : Au mieux, notre modèle peut trouver 40% des classes 1, et lorsqu'il en prédit une, il a raison à 36%.
Nous pouvons l'expliquer par le features engineering qui est à améliorer. En effet, le Kernel choisi est plutôt pauvre sur le traitement des données. Il ne se focalise que sur une seule table et ne crée pas beaucoup de variable qui peuvent être utiles à un modèle de classification comme des moyennes, des médianes, des écarts-types, et ça pour plusieurs features. Il existe peut-être un Kernel plus abouti qui permettra une meilleure performance prédictive au modèle. Sinon, prendre le temps de réaliser nous même notre feature engineering, ce qui nous permettra de bien comprendre nos données et ainsi construire un feature engineering adapté à notre besoin.