NOTE METHODOLOGIQUE : Implémentation d'un modèle de scoring

Introduction

Ce document est un livrable du projet 7 « implémentez un modèle de scoring » du parcours Data Scientist d'Openclassrooms. Ce dernier explique de manière synthétique les différentes étapes de résolution de ce projet :

- Méthodologie d'entrainement de modèle et métrique d'évaluation.
- Fonction coût métier, algorithme d'optimisation.
- Interprétabilité globale et locale du modèle.
- Limites et améliorations possibles.

Contexte

La société « Prêt à dépenser » propose des crédits à la consommation pour des personnes ayant peu ou pas d'historique de prêt. Il est question dans ce projet de développer un modèle de scoring afin de prédire la probabilité de défaut de paiement d'un client.

Il s'agit d'un problème de classification binaire supervisée dans lequel la « target » à prédire vaudra **0 si le client est en mesure de rembourser son prêt et 1 si le client n'est pas en mesure de rembourser son prêt.**

1. Méthodologie d'entrainement de modèles et métrique d'évaluation

Etapes de préparations

Avant l'entrainement des modèles, Il y 4 étapes : le **feature engineering**, le **sampling des données**, la **séparation des données**, et **le pre-processing.**

Feature engineering : la création de nouvelles variables pertinentes pour la modélisation est entièrement réalisée à l'aide du notebook Kaggle suivant :

https://www.kaggle.com/jsaguiar/lightgbm-with-simple-features/

Sampling des données : le jeu de données est relativement important (plus de 300 000 clients), j'utilise un sampling des données pour garder uniquement 100 000 clients.

Séparation des données : séparation en jeu d'entrainement (80%) et jeu de test (20%) en utilisant le paramètre « stratify » de la fonction « train_test_split » de la librairie Scikit-Learn. Ce paramètre

« stratify » effectue un fractionnement de sorte que la proportion de valeurs dans l'échantillon produit soit la même que la proportion de valeurs fournie au paramètre « stratify » (ici la target).

Pre-processing :

- Imputation des valeurs manquantes en utilisant un « SimpleImputer » de la librairie Scikit-Learn.
- Normalisation des valeurs à l'aide d'un « MinMaxScaler » de la librairie Scikit-Learn qui ramène l'ensemble des valeurs entre 0 et 1.

Modélisation et métrique d'évaluation

Pour répondre au mieux au problème, j'évalue les performances de trois algorithmes :

- Un **modèle linéaire**, ici un modèle de **régression logistique** grâce à la classe « LogisticRegression » de Scikit-Learn.
- Un **modèle ensembliste**, ici un **RandomForestClassifier** via Scikit-Learn.
- Un **modèle de gradient boosting** grâce à la classe **LGBMClassifier** de la librairie lightgbm.

Les performances de ces modèles sont comparées à la performance d'une baseline, une instance de la classe **DummyClassifier** de la librairie Scikit-Learn, en utilisant la stratégie « most-frequent ».

La métrique qui permet d'évaluer les performances des modèles est l'**AUC** (Ara Under the ROC Curve). Plus l'AUC est élevée, plus le modèle est précis.

Equilibrage des classes

En réalisant l'analyse exploratoire il est possible de constater un déséquilibre de la « target » (92% des prêts remboursés, contre seulement 8% de prêt non remboursés). Un tel déséquilibre peut biaiser l'apprentissage des modèles, c'est pourquoi il doit être pris en compte. Pour gérer ce dernier, j'ai testé 3 approches :

- **Undersampling** (RandomUnderSampler de la librairie Imbalanced-Learn) : supprime les observations de la classe majoritaire afin de rééquilibrer le jeu de données.
- **Oversampling** (SMOTE de la librairie Imbalanced-Learn) : répétition des observations de la classe minoritaire afin de rééquilibrer le jeu de données.
- **Class_weight= 'balanced' :** argument directement appliqué aux algorithmes lors de l'instanciation pour indiquer le déséquilibre à ce dernier.

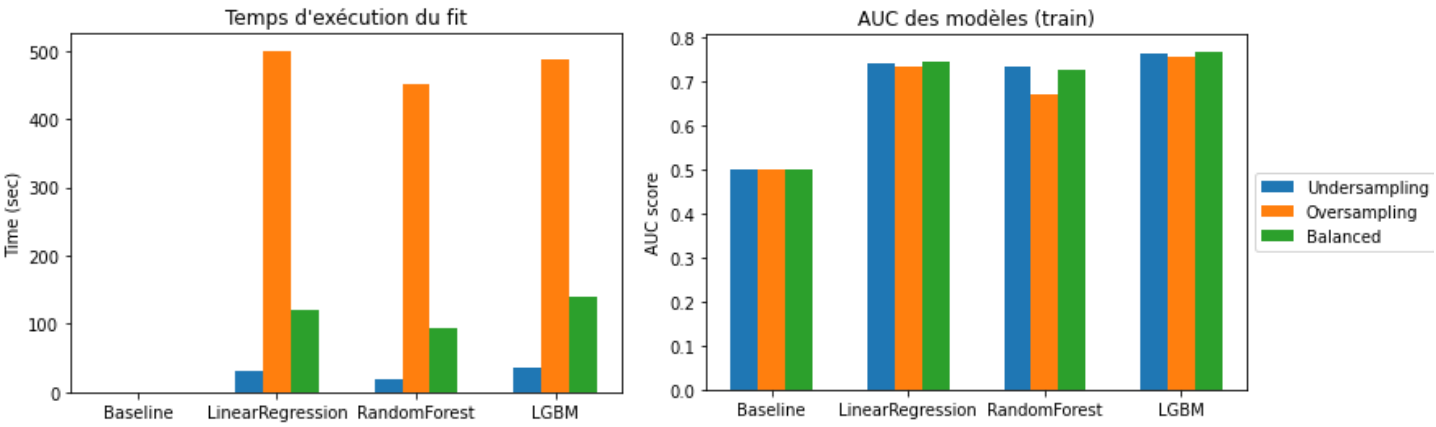
Stratégie d'entrainement des modèles

Les modèles sont entrainés avec une **validation croisée** (séparation du jeu d'entrainement en 5 folds) et leurs hyperparamètres sont optimisés à l'aide d'un **GridSearch.**

Les algorithmes sont testés avec chacune des méthodes d'équilibrage de classe cité ci-dessus. Cependant, le DummyClassifier ne permet pas d'utiliser l'argument class_weight='balanced', il est alors entrainé en utilisant la stratégie « stratified » pour respecter la distribution de classe.

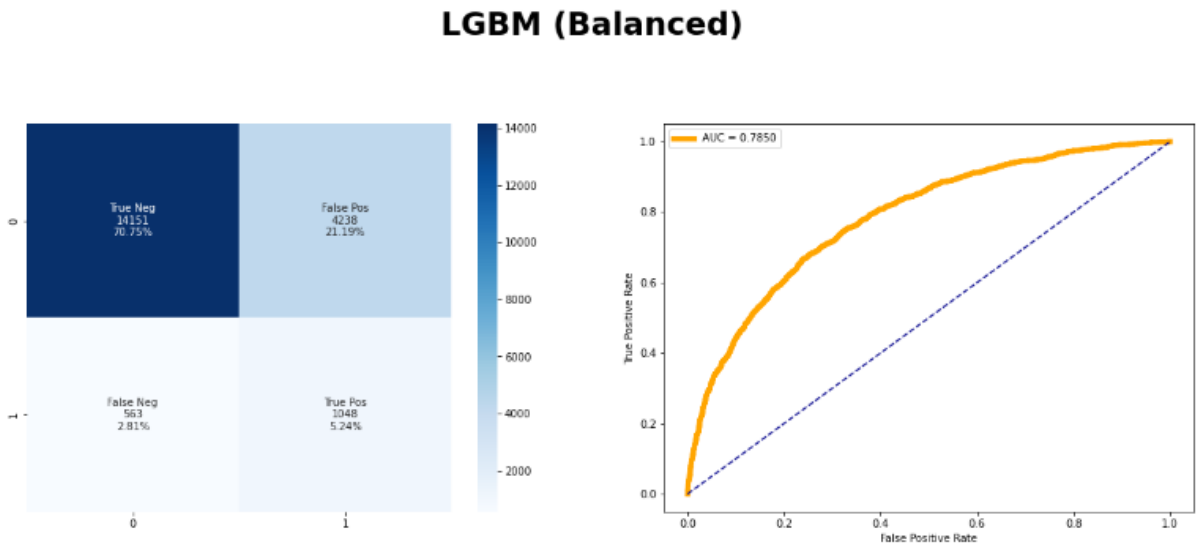
Sélection du meilleur modèle

Le meilleur modèle est sélectionné pour le score AUC le plus élevé, et le temps d'entrainement minimum. Ici, le LGBM « Balanced » donne le meilleur score AUC pour un temps d'entrainement raisonnable.



2. Fonction coût métier et algorithme d'optimisation

On peut observer la matrice de confusion, et la courbe de l'AUC du LGBM « Balanced » :



Analyse de la matrice de confusion :

- FP (faux positifs) : le modèle prédit que le client ne rembourserait pas, mais il a bien remboursé son crédit.
- FN (faux négatifs) : le modèle prédit que le client rembourserait, mais il a été en défaut.

- TP (vrais positifs) : le modèle prédit que le client ne rembourserait pas, et il a bien été en défaut.
- TN (vrais négatifs) : le modèle prédit que le client rembourserait, et il l'a bien fait.

La problématique « métier » est de considérer qu'un faux positif (FP) n'a pas le même coût qu'un faux négatif (FN).

En effet, un FP est un bon client mais considéré comme mauvais où le crédit n'est pas accordé, à tort, c'est donc un manque à gagner pour l'entreprise.

Un FN est un mauvais client à qui on accorde un prêt et qui ne rembourse pas, c'est donc une perte sur le capital non remboursé. C'est la pire situation. Il est ainsi réalisé une fonction d'optimisation pour réduire le nombre de FN.

Fonction d'optimisation

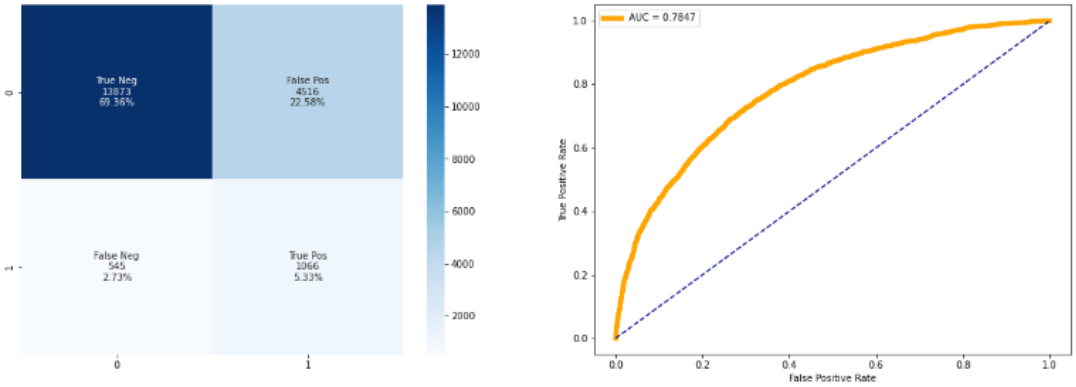
Une nouvelle recherche d'hyperparamètre est effectué en utilisant **HyperOpt,** un algorithme qui utilise l'optimisation bayésienne. Cette fois ci on utilise une fonction d'optimisation (image-ci-dessous) qui retourne un score custom spécifique au besoin de l'entreprise, qui permet de réduire le nombre de FN.

```python
def custom_score(y_true, y_pred):

    TN, FP, FN, TP = confusion_matrix(y_true, y_pred).ravel()
    N = TN + FP  #total negatives
    P = TP + FN  #total positives

    #weights
    w_FN = -200000 #worst case
    w_TN = 10000
    w_TP = 0
    w_FP = 0

    #total gains
    gain = TP*w_TP + TN*w_TN + FP*w_FP + FN*w_FN

    #maximum gain : all corrects predictions
    max_score = N*w_TN

    #minimum gain : all wrong predictions
    min_score = P*w_FN

    # normalize to get score between 0 and 1
    score = (gain - min_score) / (max_score - min_score)

    return score
```

Les poids sont fixés arbitrairement pour diminuer le nombre de FN.

En utilisant donc cette fonction comme métrique d'évaluation lors du nouvel entrainement de notre modèle on obtient le résultat suivant :

HyperOpt optimized LGBM

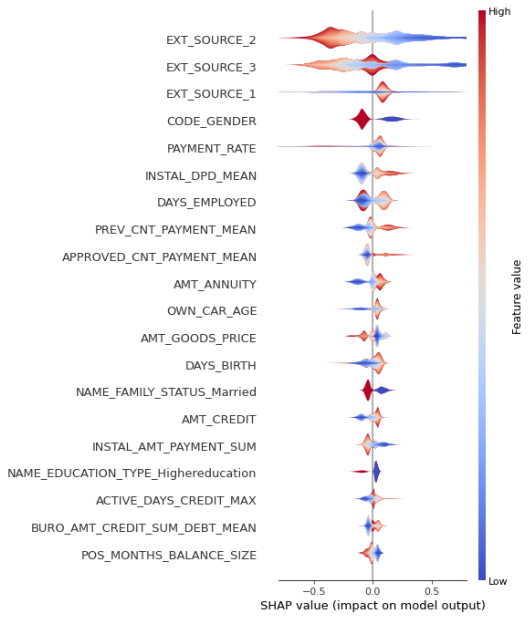


On peut voir qu'on a réduit le nombre de « False Neg », passant ainsi de 563 clients à 545 clients.

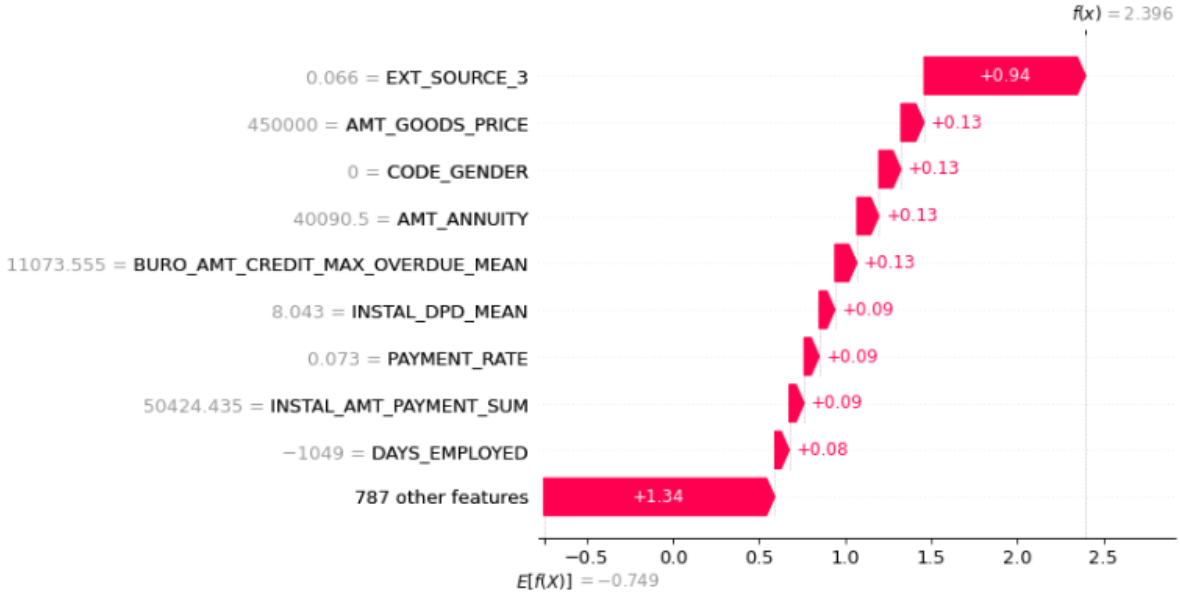
3. Interprétabilité globale et locale du modèle

L'une des problématiques du projet est de pouvoir communiquer sur les résultats de la prédiction et de comprendre pourquoi un client est jugé comme bon ou mauvais client. Il s'agit alors de connaitre les features qui expliquent le résultat de la prédiction. Il est donc nécessaire de connaitre d'une manière générale les principales features qui contribuent à la décision du modèle, et aussi de manière spécifique pour un client en particulier, et de déterminer l'influence de chaque feature dans la prédiction. Il est utilisé une librairie spécialisée, **Shap**, qui permet de calculer directement la feature importance.

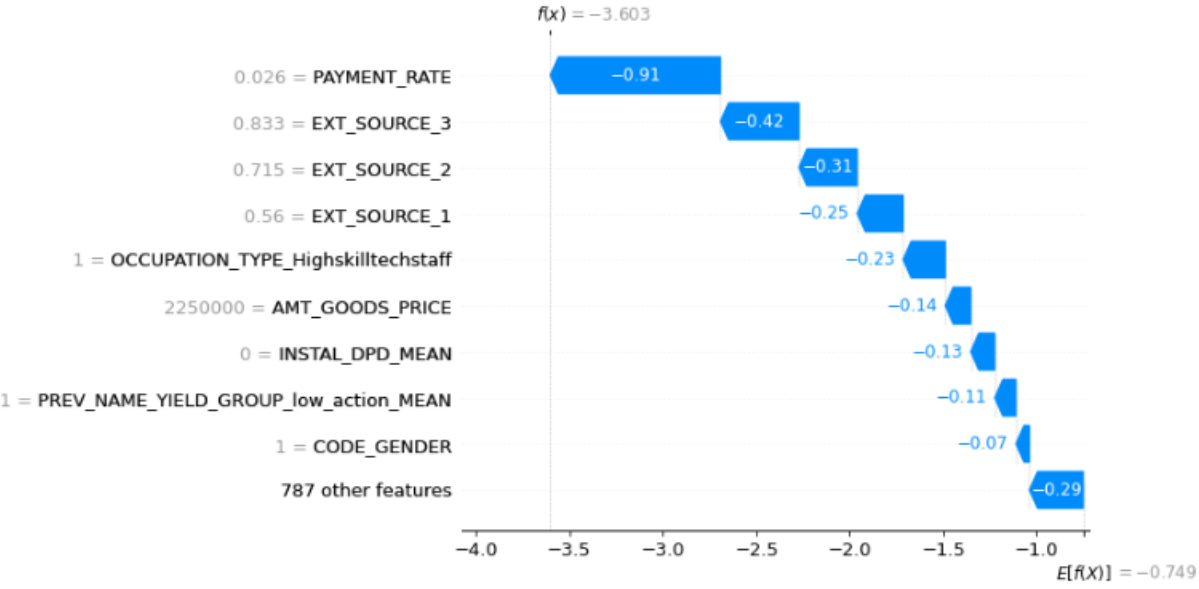
Ci-dessous une interprétation globale du modèle **LGBM** sur les 1000 premiers clients pour la classe 1(prêt non remboursé) : type **layered violin plot**



Ci-dessous les features importances locales du client le plus à risque du jeu de test.



Ci-dessous les features importances locales du client le moins à risque du jeu de test.

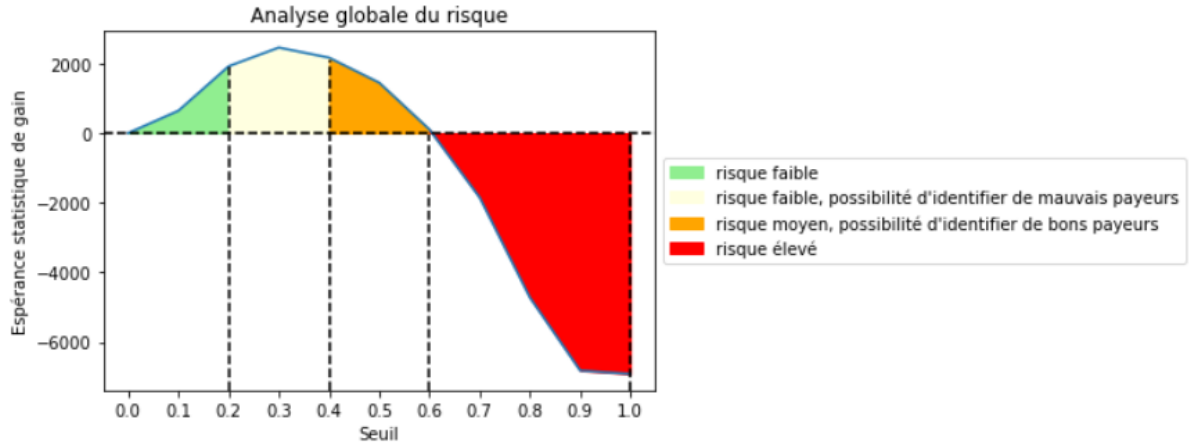


En rouge, les variables qui ont un impact positif (contribuent à ce que la probabilité de défaut soit plus élevée que la valeur de base) et, en bleu, celles ayant un impact négatif (contribuent à ce que la probabilité de défaut soit plus basse que la valeur de base). On peut y voir l'importance de la variable EXT_SOURCE_3 pour la décision du modèle. Les valeurs sont exprimées après application de la fonction logit.

4. Limites et améliorations possibles

La fonction coût métier utilise des poids de pénalités fixés de manière arbitraire. Il est possible d'affiner les prédictions du modèle en fixant ces poids de manière plus précise par des experts métier.

On peut tracer l'évolution du gain en fonction de la probabilité de défaut d'un client (seuil) à partir de la fonction custom score évoquée plus haut



On peut voir qu'on ne peut pas perdre d'argent si on refuse tous les clients au-dessus d'une probabilité de 0.6. Cependant, le manque à gagner peut-être élevé car on pourrait refuser des bon clients, jugés mauvais par le modèle. Une connaissance plus précise du métier et de l'importance de certaines features par le banquier peut permettre d'identifier de mauvais clients jugés bon par le modèle et de bon clients jugés mauvais par le modèle.