PROJET 7 - « IMPLEMENTEZ UN MODELE DE SCORING »
NOTE METHODOLOGIQUE

Le présent document fait partie des livrables du projet 7 intitulé « **Implémentez un modèle de scoring** » du parcours Data Scientist d'Openclassrooms. Ce projet consiste à développer pour la société « Prêt à Dépenser » un modèle de Scoring afin de prédire la probabilité de défaut de paiement d'un client avec peu ou pas d'historique de prêt. Il s'agit donc d'un problème de classification binaire supervisée dans lequel la variable cible (TARGET) à prédire vaudra 0 si le client est solvable et 1 s'il ne l'est pas.

Ce document décrit de façon détaillée la méthodologie d'entrainement du modèle ; la fonction coût métier, l'algorithme d'optimisation et la métrique d'évaluation ; l'interprétabilité globale et locale du modèle puis se termine en exposant les limites et les améliorations possibles du travail effectué

1. Problématique Business

Lorsqu'une institution financière prête de l'argent à une personne, elle prend le risque que cette dernière ne rembourse pas cet argent dans le délai convenu. Ce risque est appelé **Risque de Crédit**. Alors avant d'octroyer du crédit, les institutions financières vérifient si la personne qui demandent un prêt sera capable ou pas de le rembourser (solvabilité).
Grâce à des modèles de Machine Learning, les institutions financières peuvent modéliser la **probabilité de défaut -** PD de paiement et ainsi attribuer un score à chaque nouveau demandeur de crédit : **Crédit Scoring**. Dans ce projet, nous apprendrons à construire et évaluer un modèle de Machine Learning pour prédire si un demandeur de crédit sera en défaut de paiement ou non. Il s'agit d'un problème de **classification binaire**.
En générale le défaut est observé en fonction du taux d'endette, plus celui-ci est élevé, plus la probabilité que le client ne rembourse pas son crédit soit élevé.

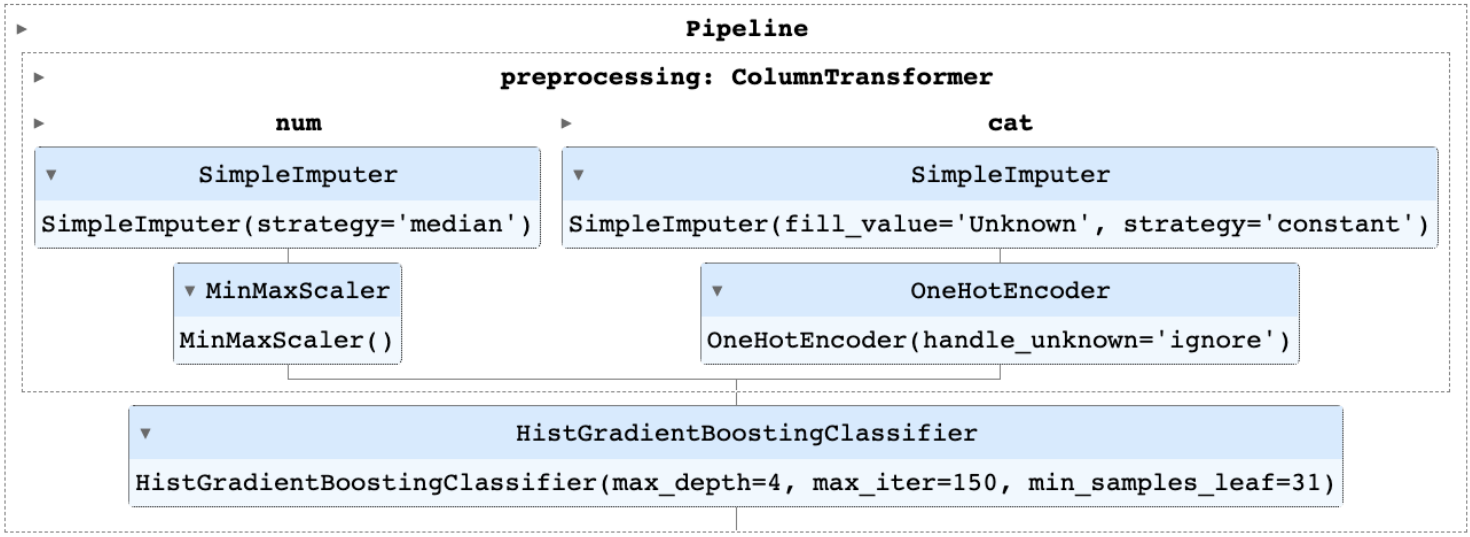
Mais dans le cas espèce, l'évènement défaut est déclenché lorsque le client est difficulté de paiement c-à-d qu'il a un retard de paiement de plus de X jours sur au moins une des Y premières échéances.

2. METHODOLOGIE D'ENTRAINEMENT DES MODELES

1.1 Les étapes avant l'entrainement des modèles

Les étapes majeures qui ont précédé l'étape d'entrainement des modèles sont les suivantes savoir : l'étape de pré-processing, feature engineering, le sampling des données, la séparation des données en jeux d'entrainement et de test et enfin le pré-traitement des données.

- **Feature engineering :** la création des features pertinents pour la modélisation a été entièrement inspirée du notebook Kaggle.
- **Sampling des données :** le jeu de données étant relativement important et déséquilibré (plus de 300000 clients), nous avons utilisé un échantillonnage des données sur la classe minoritaire (soit 100000 clients).
- **Séparation des données :** l'échantillon de données sélectionné a été séparé en jeu d'entrainement (70%) et jeu de test (70%) en appliquant une stratification sur la variable cible (TARGET) pour s'assurer que chaque côté contient une proportion raisonnable des classes 0 et 1.
- **Pré-traitement des données :** a consisté à imputer les valeurs manquantes (via un SimpleImputer de la librairie Scikit-Learn) et à normaliser les valeurs (via un Min-MaxScaler de la librairie Scikit-Learn qui ramène l'ensemble des valeurs entre 0 et 1)



1.2 Modèles candidats et métrique d'évaluation

Afin de modéliser au mieux le problème, les performances des trois algorithmes de différentes familles ont été comparées :

- **Régression Logistique** (famille des algorithmes linéaires) via la classe LogisticRegression de la librairie Scikit-Learn

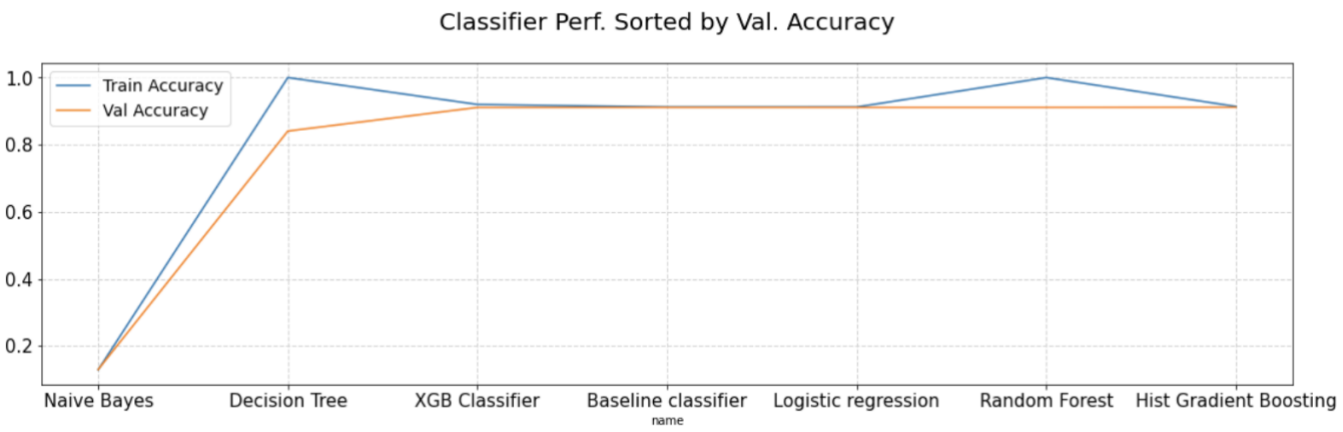
- **RandomForestClassifier** (famille des algorithmes ensemblistes) via la classe RandomForestClassifier de la librairie Scikit-Learn

- **Hist Gradient Boosting Machine** (famille des algorithmes gradient boosting) via la classe HistGradientBoostingClassifier de la librairie Scikit-learn.

Leurs performances ont été comparées à celles d'une Baseline naïve, une instance de la classe **DummyClassifier** de la librairie Pandas, instanciée avec la stratégie "most_frequent", c'est-à-dire prédisant systématiquement la classe la plus fréquente.

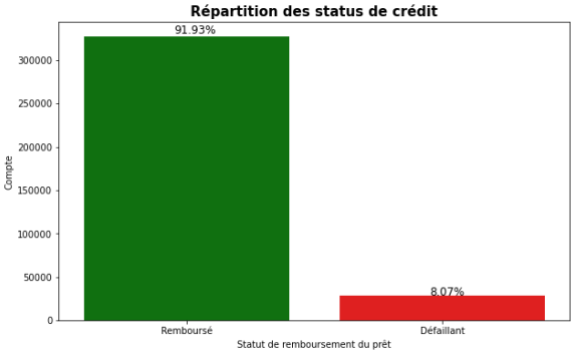
La métrique utilisée pour évaluer les performances de chacun des modèles est l'**AUC** (Area Under the ROC Curve). Plus l'AUC est élevée, plus le modèle est capable de prédire les 0 comme 0 et les 1 comme 1

```
models =[
          {"name": "Baseline classifier", "clf": DummyClassifier()},
          {"name": "Naive Bayes", "clf": GaussianNB()},
          {"name": "Logistic regression", "clf": LogisticRegression()},
          {"name": "Decision Tree", "clf": DecisionTreeClassifier()},
          {"name": "Random Forest", "clf": RandomForestClassifier()},
          {"name": "Hist Gradient Boosting", "clf": HistGradientBoostingClassifier()},
          {"name": "XGB Classifier", "clf": XGBClassifier()},
        ]
```



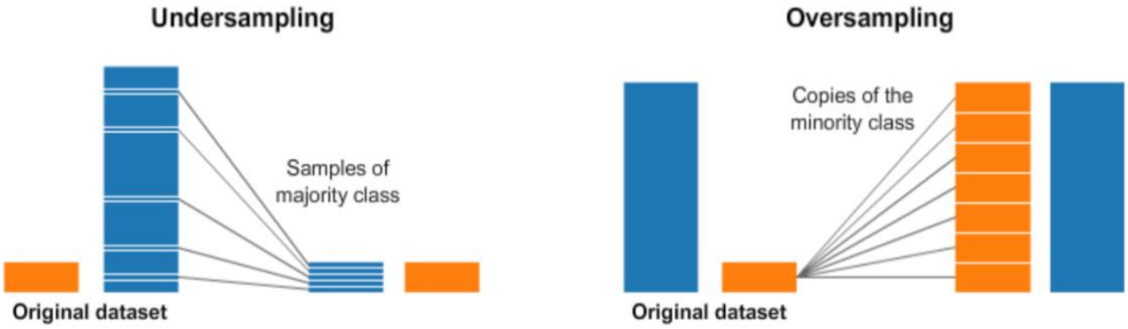
1.3 Rééquilibrage des classes

L'analyse exploratoire a permis de révéler le caractère déséquilibré de la variable cible (91,93% des prêts remboursés, contre seulement 8,07% de prêts non remboursés). Étant en face d'un problème de classification binaire déséquilibré, ce déséquilibre des classes doit être pris en compte dans l'entraînement des modèles.



Nous avons donc modifié l'ensemble de données utilisé avant d'entraîner le modèle prédictif afin d'avoir des données plus équilibrées. Cette stratégie est appelée rééchantillonnage et il existe deux méthodes principales que vous pouvez utiliser pour égaliser les classes : Le **sur-échantillonnage** (Oversampling) et le **sous-échantillonnage** (Undersampling).

- Les **méthodes d'Oversampling** fonctionnent en augmentant le nombre d'observations de la (des) classe(s) minoritaire(s) afin d'arriver à un ratio classe minoritaire/ classe majoritaire satisfaisant.
- Les **méthodes d'Undersampling** fonctionnent en diminuant le nombre d'observations de la (des) classe(s) majoritaire(s) afin d'arriver à un ratio classe minoritaire/ classe majoritaire satisfaisant.
- **Class_weight="balanced"** : argument qui indique le déséquilibre à l'algorithme afin qu'il en tienne compte directement.

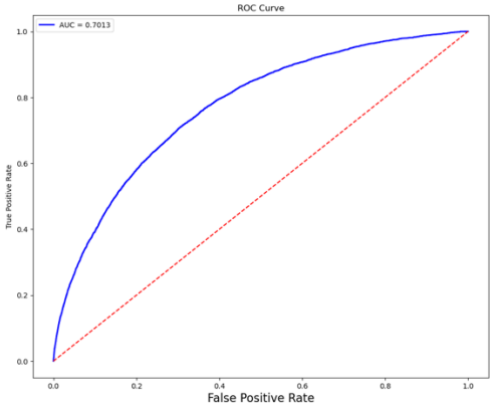


1.4 Stratégie d'entrainement des modèles

Tous les modèles ont été entrainés avec une **validation croisée** (séparation du jeu d'entrainement en **4 folds**) et leurs hyperparamètres ont été optimisés via **GridSearch**. Les algorithmes ont été testés avec chacune des méthodes de rééquilibrage des classes présentées plus haut. Toutefois, l'algorithme DummyClassifier n'acceptant pas l'argument class_weight="balanced", son comportement a été simulé avec la stratégie "stratified" au lieu de "most_frequent", qui génère des prédictions en respectant la distribution de classe du jeu de données d'entraînement.

1.5 Choix du meilleur modèle

Le meilleur modèle a été choisi sur la base de la valeur du score AUC moyen obtenu sur l'ensemble des splits de la validation croisée (GridSearchCv). Le score AUC obtenu sur le jeu de test a également été calculé. L'algorithme Hist Gradient Boosting associé à la stratégie de rééquilibrage consistant à indiquer "balanced" comme valeur pour " class_weight" donne les meilleurs résultats.



	Model	Accuracy	precision	recall	f1_score	AUC	train_time
5	Hist Gradient Boosting	0.773801	18.25	35.11	24.02	0.5989	3.259215
4	Random Forest	1.000000	11.54	74.09	19.97	0.5935	13.028679
6	XGB Classifier	0.878236	10.64	83.87	18.88	0.5757	15.459811
2	Logistic regression	0.693417	37.30	8.18	13.42	0.5342	0.719950
1	Naive Bayes	0.555864	50.00	0.02	0.03	0.5001	0.277565
0	Baseline classifier	0.500000	0.00	0.00	0.00	0.5000	0.151067
3	Decision Tree	1.000000	7.08	46.54	12.29	0.4347	2.963696

2. FONCTION COUT METIER ET ALGORITHME D'OPTIMISATION

2.1 Fonction coût métier

Définitions des terminologies de la matrice de confusion

- **FP (*Faux Positifs*) :** la classe a été prédite comme "1", mais la classe réelle est "0" => le modèle a prédit que le client ne rembourserait pas, mais il a bien remboursé son crédit.
- **FN (*Faux Négatifs*) :** la classe a été prédite comme "0", mais la classe réelle est "1" => le modèle a prédit que le client rembourserait, mais il a été en défaut.
- **TP (*Vrais Positifs*) :** la classe a été prédite comme "1", et la classe réelle est bien "1" => le modèle a prédit que le client ne rembourserait pas, et il a bien été en défaut.
- **TN (*Vrais Négatifs*) :** la classe a été prédite comme "0", et la classe réelle est bien "0"=> le modèle a prédit que le client rembourserait, et il l'a bien fait.

La problématique « métier » est de prendre en compte qu'un faux positif (bon client considéré comme mauvais = crédit non accordé à tort, donc manque à gagner de la marge pour la banque) n'a pas le même coût qu'un faux négatif (mauvais client à qui on accorde un prêt, donc perte sur le capital non remboursé). Un faux négatif est en effet 10 fois plus coûteux qu'un faux positif. Nous avons ainsi défini une fonction coût métier adaptée au projet qui permet d'attribuer plus de poids à la minimisation des FN.

Des coefficients permettant de pondérer les pénalités associées à chaque cas de figure ont été attribués : TP_value : 0 ; FN_value : -10 ; TN_value : 1 ; FP_value : -1

Ces valeurs de coefficients signifient que les Faux Négatifs engendrent des pertes 10 fois plus importantes que les pertes engendrées par les Faux Positifs. Les Vrais positifs n'entrainement aucun gain ni perte pour l'entreprise, tandis que les Vrais Négatifs permettent à l'entreprise de gagner en prêtant à de bons prospects.

Ces poids ont été fixé arbitrairement et il est tout à fait envisageable de les modifier à la convenance de l'optique métier.

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

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

    # Setting the bank's gain and loss for each case
    FN_value = -10   # The loan is granted but the customer defaults : the bank loses money (Type-II Error)
    TN_value = 1     # The loan is reimbursed : the bank makes money
    TP_value = 0     # The loan is (rightly) refused : the bank neither wins nor loses money
    FP_value = -1    # Loan is refused by mistake : the bank loses money it could have made,
                     # but does not actually lose any money (Type-I Error)

    # calculate total gains
    gain = TP*TP_value + TN*TN_value + FP*FP_value + FN*FN_value

    # best score : all observations are correctly predicted
    best = N*TN_value + P*TP_value

    # baseline : all observations are predicted = 0
    baseline = N*TN_value + P*FN_value

    # normalize to get score between 0 (baseline) and 1
    score = (gain - baseline) / (best - baseline)

    return score
```

2.2 Algorithme d'optimisation

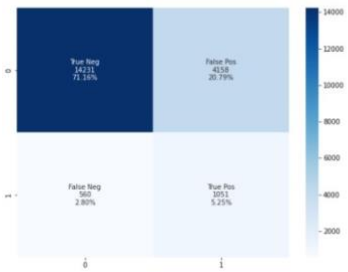
Une fois le modèle choisi (LGBM) et ses hyperparamètres optimisés d'un point de vue technique (via l'AUC avec GridSearch), nous avons à nouveau effectué une nouvelle recherche des hyperparamètres via **HyperOpt** (algorithme qui utilise l'optimisation bayésienne) en se basant sur la fonction coût métier proposée. De cette façon, les hyperparamètres optimaux du modèle sont choisis de sorte à minimiser la perte pour l'entreprise. **HyperOpt** nécessite 4 paramètres pour une implémentation de base à savoir : la fonction objectif (à minimiser **loss = 1-score**), l'espace de recherche (plages pour les hyperparamètres), l'algorithme d'optimisation et le nombre d'itérations.

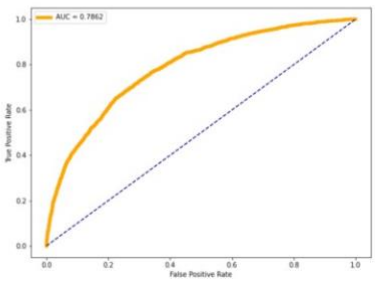
```python
#Parameter space
space = {
    'n_estimators': hp.quniform('n_estimators', 100, 600, 100),
    'learning_rate': hp.uniform('learning_rate', 0.001, 0.03),
    'max_depth': hp.quniform('max_depth', 3, 7, 1),
    'subsample': hp.uniform('subsample', 0.60, 0.95),
    'colsample_bytree': hp.uniform('colsample_bytree', 0.60, 0.95),
    'reg_lambda': hp.uniform('reg_lambda', 1, 20)
}
```

```
%%time
best = fmin(fn=objective, space=space, max_evals=10, algo=tpe.suggest)
```

```
100%|██████████| 10/10 [27:48<00:00, 166.89s/trial, best loss: 0.9745731139397703]
Wall time: 27min 48s
```

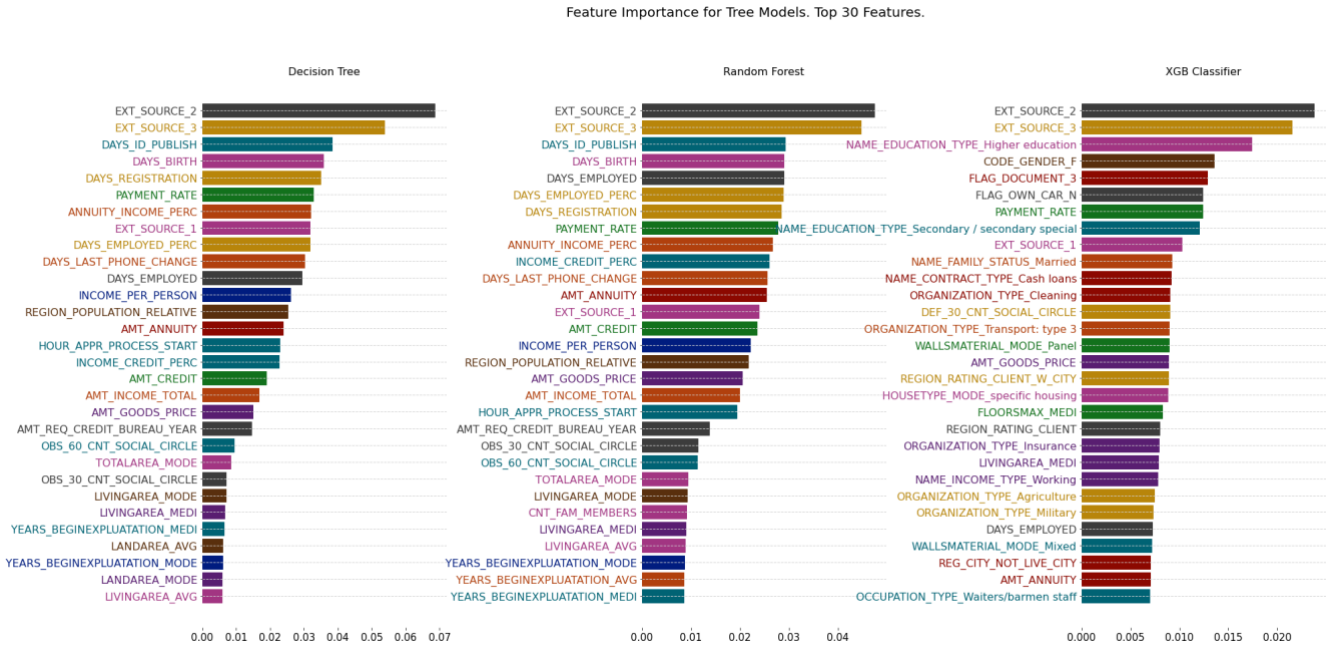
HyperOpt optimized LGBM





3. INTERPRETABILITE GLOBALE ET LOCALE DU MODELE

L'objectif métier est de communiquer des informations pertinentes d'analyse au chargé d'étude, afin qu'il comprenne pourquoi un client donné est considéré comme bon ou mauvais prospect et quelles sont ses données/features qui expliquent son évaluation. Il est donc nécessaire à la fois, de connaître d'une manière générale les principales features qui contribuent à l'élaboration du modèle, et de manière spécifique pour le client qu'elle est l'influence de chaque feature dans le calcul de son propre score (feature importance locale).



4. LIMITES ET LES AMELIORATIONS POSSIBLES

- Après le feature engineering inspiré du Kaggle mentionné plus haut, les données contiennent 795 features au total. Pour la modélisation, nous aurions pu ajouter une étape de **sélection des features** (en utilisant par exemple la technique d'**élimination récursive des features avec validation croisée (RFECV**)) afin de ne retenir que les features les plus importants et réduire ce nombre.
- A cause des soucis d'espace mémoire, nous avons dû travailler avec un échantillon de 100000 clients et pas la totalité du dataset. Appliquer la modélisation sur l'intégralité du jeu de données sur une machine avec plus de RAM serait certainement intéressant à explorer
- Chacun des modèles entrainés a été optimisé en testant plusieurs valeurs des 2 ou 3 principaux hyperparamètres (car l'exécution de GridSearch prend beaucoup de temps). Elargir le nombre d'hyperparamètres pourrait peut-être permettre une amélioration des performances des différents modèles.
- La fonction coût métier définie utilise des poids de pénalités arbitrairement fixés. Il est donc certainement possible d'affiner les prédictions du modèle en fixant ces poids de façon plus adéquate par des experts métier.