PROJET 7 : Implémenter un modèle de scoring

NOTE MÉTHODOLOGIQUE

Contexte :

Prêt à dépenser est un organisme de crédit à la consommation qui cible les personnes ayant peu ou pas d'historique de crédit.

L'entreprise souhaite mettre en œuvre un outil de 'scoring crédit' afin de calculer la probabilité de remboursement d'un client. Elle souhaite également mettre en place un dashboard afin de restituer le résultat de la modélisation. L'objectif étant de disposer d'un score qui conditionnera l'accord d'un crédit et de pouvoir justifier cette décision.

Le jeu de donnée est initialement constitué de 307 000 observations et 121 variables.

Problématique :

Il s'agit d'un problème de classification avec deux classes déséquilibrées. La classe majoritaire, 0, compte 91.2% du jeu de données. La classe minoritaire, 1, compte 8.8 % du jeu de donnée.

La notion de solvabilité d'un client sera définie **selon un seuil de classification**.
L'évaluation des modèles sera basée sur les métriques suivants : **ROC_AUC** et **F_{beta} score**.

ROC_AUC

Une courbe ROC Receiver Operator Characteristic est un graphique utilisé pour montrer la capacité de prédiction d'un classifieur binaire.

Cette dernière est construite en traçant le taux de vrais positifs (TPR ou Recall) en fonction du taux de faux positifs (FPR). La mesure ROC_AUC est l'aire sous la courbe ROC, elle est comprise entre 0 et 1.

- Pour un modèle de classification dont les prédictions sont 100 % fausses, ROC_AUC = 0.
- Pour un modèle dont les prédictions sont 100 % correctes, ROC_AUC = 1.

Fbeta SCORE
Matrice de confusion (selon la convention sklearn) :

		PREDICTIONS	
	CLASS	**Clients prédits solvables (0)**	**Clients prédits en défaut de paiement (1)**
ACTUAL	**Clients solvables (0)**	Vrai Négatifs (TN)	Faux positifs (FP)
	Clients en défaut de paiement (1)	**Faux Négatifs** (FN)	Vrais Positifs (TP)

Afin de donner vie à cette fonction "coût", il faut prendre en compte la problématique métier. Nous devons prendre en compte qu'un faux positif n'a pas le même coût qu'un faux négatif. Contextualisons :

- Un faux positif est un client solvable à qui nous allons refuser une demande de crédit. Ceci entraine un manque à gagner pour l'organisme de crédit.
- Un faux négatif est un client non solvable à qui on accorde une demande de crédit. Cette fois cela entraine des pertes pour l'organisme de crédit.

Ainsi, nous comprenons que le second cas est bien plus impactant pour l'entreprise que le premier. Nous devons minimiser les **faux négatifs**. Ceci est possible avec la fonction $F_\beta score$. Cette dernière permet d'attribuer plus de poids à la minimisation des faux négatifs à travers la pondération du paramètre beta.

Résumé :

- Si $\beta > 1$, on accorde plus d'importance au recall. On minimise les faux négatifs.
- Si $\beta < 1$, on accorde plus d'importance à la précision. On minimise les faux positifs.
- Si $\beta = 1$, la précision et le recall ont la même importance. C'est le $F_1 score$.

La valeur doit normalement être déterminée par les équipes métiers. Elles seules peuvent déterminer l'impact des FP ou FN.

Notre solution est de choisir la valeur Beta qui nous rapprochera le plus du recall.

Conclusion : Nous penchons vers **Beta = 3 ou 4**.

Méthodologie d'entrainement des modèles :

Le *features engineering* des données a été effectué à partir du kernel suivant : LightGBM with Simple Features
En sortie, nous nous retrouvions désormais avec un fichier contenant **307 observations** et **801 colonnes**.
Au vu du grand nombre de variables, j'ai opté pour la suppression de donnée plutôt que l'imputation. En effet, je ne pouvais justifier de l'imputation de la moyenne pour chacune des colonnes.

Les suppressions se sont donc faites comme suit :
- Suppression des variables renseignées à moins de 60 %
- Suppression de tous les clients présentant des valeurs manquantes.

Je me suis retrouvée avec un fichier contenant 79060 lignes et 555 variables, avec une répartition similaire des classes. J'ai donc gardé ce fichier en tant que fichier d'entrainement.

1. Pré-traitement des données

Avant de pouvoir commencer l'entrainement des modèles, il est nécessaire de préparer le jeu de donnée. Dans le cadre d'un apprentissage supervisé, il est nécessaire de définir deux sets de données :

- Données d'entrainement. Elle constitue la majorité du jeu de donnée principal. Ce sont les données sur lesquelles le modèle s'entraine.
- Données de test. Elle représente un pourcentage mineur du jeu de donnée principal. Ce sont les données sur lesquelles nous allons tester le modèle entrainé.

Cette découpe peut être effectuée à l'aide de la fonction *test_train_split* du module Scikit-Learn. On obtient ainsi les paramètres suivants : X_train, y_train, X_test, y_test.

2. Choix des modèles

Afin de pouvoir prédire le défaut de paiement d'un client, plusieurs types de modèles de classification vont être entrainé.

- Régression logistique. Cette dernière servira de référence en termes de résultat.
- RandomForestClassifier
- GradientBoostingClassifier
- LGBMClassifier

3. Méthode de traitement du déséquilibre des classes

Le déséquilibre des classes détériore la qualité des prédictions des modèles. En effet, lors de l'entrainement, l'apprentissage se fera au détriment de la classe minoritaire.

Quatre méthodes de traitements ont été appliquées :

- **Pondération** : Un poids est affecté à chaque classe. La classe minoritaire aura le poids le plus élevé. Le poids peut être déterminé grâce à la fonction *compute_class_weight* de **scikit-learn**.
  Cette pondération est prise en compte par les algorithmes à l'aide de l'option class_weight='balanced'. A noter que l'algorithme Gradient Boosting ne dispose pas de cette option. Il le fait nativement.

- **Undersampling** ou *sous-échantillonnage* : La classe majoritaire est réduite à l'effectif de la classe minoritaire. La réduction se fait à l'aide d'un échantillonnage aléatoire via la fonction RandomUnderSampler, par exemple.

- **Oversampling** ou *sur-échantillonnage* : La classe minoritaire est augmentée au niveau de l'effectif de la classe majoritaire. De nouvelles observations sont créées. Plusieurs algorithmes de création d'observations vont être testés : SMOTE et Adasyn.

- En dernier, nous allons tester une combinaison d'Oversampling (SMOTE) et d'undersampling (RandomUnderSampler)

SMOTE – **S**ynthetic **M**inority **O**versampling **TE**chnique – Au lieu d'effectuer un simple clonage des individus minoritaires, cet algorithme se base sur un principe simple : **générer de nouveaux individus minoritaires qui ressemblent aux autres, sans être absolument identiques**.
On part d'une observation minoritaire connu. On choisit un proche voisin. Puis on crée un nouvel individu entre ces deux observations, conditionné par un coefficient α.

ADASYN – **A**daptive **S**ynthetic **S**ampling **A**pproach – Cette technique peut être considérée comme une version améliorée de SMOTE. Ce qu'elle apporte en supplément : Elle va classer les observations de la classe minoritaire en fonction de leur niveau de difficulté d'apprentissage. Et elle va générer davantage de nouvelles données pour les cas difficile à apprendre. Les nouvelles observations sont moins corrélées aux observations originales qu'avec SMOTE.
SMOTE and ADASYN (Handling Imbalanced Data Set) | by Indresh Bhattacharyya | Coinmonks | Medium

4. Réalisation des entrainements

Chaque modèle sera entraîné avec chacune des méthodes de traitement de déséquilibre de données.

Afin d'éviter le leakage (surtout avec le sur-échantillonnage), ces méthodes seront appliquées uniquement lors de l'entrainement.
Pour ce faire, nous utiliserons un pipeline composé de 3 étapes :

1. Méthode de traitement du déséquilibre des données (Undersampling, Smote, Adasyn…)
2. Méthode de recalibrage des données (StandardScaler)
3. Modèle à entrainer

Ainsi seules les données d'entrainement seront transformées. Le modèle ainsi entrainé sera évalué via les données de test, qui elles seront déséquilibrées.

Néanmoins, nous vérifierons si tester sur un jeu de donnée équilibré apporte une grande différence (dans le cas de SMOTE notamment).

A noter également, l'entrainement se fera à l'aide d'un GridSearchcv (cv = 5), sur un seul paramètre néanmoins. Ceci afin de tester différentes possibilités.

Paramètres :
- Logistic Regression : C
- Random Forest Classifier : n_estimators
- Gradient Boosting Classifier : n_estimators
- LGBMClassifier : n_estimators

L'optimisation des paramètres s'est faite à l'aide des modules GridSearchcv (cv=5) et hyperopt. Ceci afin de maximiser ROC_AUC et Fbeta_score.

Le modèle choisi : **LGBMClassifier**
Paramètres à optimiser :
- **max_depth**
- **num_leaves**
- **learning_rate**
- **n_estimators**

Interprétabilité du modèle :

Le modèle est la réponse à un besoin métier. Il est destiné à des équipes fonctionnelles (sans notion technique) qui devront justifier leur décision à des clients finaux.

En d'autres termes, l'interprétabilité désigne l'évaluation du processus de prise de décision. Elle vise à représenter l'importance de chaque variable en jeu.

Il existe plusieurs méthodes. La première est inhérente au modèle, il s'agit de *.features_importances_* .Elle permet de mettre en avant les variables qui ont le plus de poids sur les décisions du modèle.
La seconde méthode, Shap (*Shapley Additive exPlanations*), a la particularité d'être indépendante du modèle. Qui plus est, en plus de pouvoir relever les variables les plus influentes du modèle, elle permet également d'identifier localement les variables ayant le plus poids dans la décision pour un client donné.

Un dashboard a été développé afin de restituer le résultat de la décision et l'expliquer.

Limites et améliorations :

Les résultats obtenus sont basés sur une métrique dont les paramètres n'ont pas été validés par les équipes métiers. Le coefficient β nécessite d'être validé par les métiers.

Ceci vaut également pour le seuil de solvabilité. Ce dernier a été défini de manière arbitraire.

L'étape du features engineering est basé sur un kernel existant, qui a donné lieu à un très grand nombre de variables. Il aurait été intéressant de le faire soi-même afin de maîtriser le jeu de données. Néanmoins, il semble nécessaire d'avoir des connaissances métiers en crédit.

Enfin, il serait nécessaire de travailler sur l'optimisation de mon tableau de bord. L'application telle qu'elle est aujourd'hui, est trop lourde pour Heroku ce qui la rend peu performante.