Note Méthodologique

Le projet consiste en la création d'une page internet qui affichera les résultats de prédiction d'accord ou non de crédit à pour un client donné, à partir d'une base de données de test. Cette prédiction sera réalisée en utilisant un algorithme capable de traiter des problèmes de classification binaire.

La méthodologie d'entraînement du modèle est un processus qui consiste à définir les étapes nécessaires pour créer un modèle de qualité. Cette méthodologie doit prendre en compte les données utilisées, le choix de l'algorithme d'apprentissage, les hyperparamètres, la sélection de la métrique d'évaluation et la validation croisée. Pour garantir un modèle de qualité, il est également important de bien préparer les données en effectuant des étapes de nettoyage, de traitement et de transformation des données.

Partie 1 : Préparation des données :

Les données fournies se composent de 10 fichiers au format csv situés à l'adresse suivante :

https://www.kaggle.com/c/home-credit-default-risk/data

Elles seront agrégées et préparées avec le feature engineering du kernel kaggle dont le lien est le suivant :

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

Il en résulte un dataset composé de 307507 clients dont nous savons s'ils ont obtenu ou non l'emprunt demandé (qui servira pour la création du modèle) et de 48744 clients dont nous ne le savons pas (qui servirons à tester le modèle sur la page internet), avec 797 variables numériques par client.

Etape 1 : Nettoyage des données

Nettoyage des données en amont de l'entraînement :
- Les colonnes avec plus de 30% de valeurs manquantes ont été éliminées.
- Les colonnes contenant des informations redondantes, retrait par analyse de corrélations des variables dont celle-ci est supérieure à 0.5

Etape 2 : Sélection des variables

La méthode SelectKBest est utilisée pour sélectionner les 20 features les plus pertinentes pour la prédiction de la variable cible. La fonction de score utilisée est f_classif, qui mesure la corrélation linéaire entre chaque feature et la variable cible. En fixant k=20, la méthode SelectKBest ne conserve que les 20 features les plus corrélées à la variable cible. Le dataset de création du modèle est

temporairement imputé avec KNNImputer(n_neighbors=5) pour remplacer les valeurs manquantes dans le jeu de données avant d'etre utilisé avec SelectKbest.

Partie 2 : Elaboration d'un modèle d'apprentissage :

Etape 1 : Utilisation de SMOTE (pour équilibrer les données) :

Lorsque les données sont déséquilibrées, c'est-à-dire lorsque certaines classes ont beaucoup plus d'exemples que d'autres, cela peut poser un problème pour la formation du modèle. En effet, le modèle aura tendance à prédire la classe majoritaire. Pour traiter ce déséquilibre, il existe plusieurs techniques telles que le suréchantillonnage, le sous-échantillonnage, la pondération des classes et l'utilisation de techniques de coûts métiers.

Les classes "prêt accordé" et "prêt non accordé" sont déséquilibrées dans le jeu de données. Pour remédier à cela, utilisation de la technique de suréchantillonnage SMOTE (Synthetic Minority Over-sampling Technique) pour générer des données synthétiques de la classe minoritaire.
La méthode SMOTE est un moyen couramment utilisé pour traiter le déséquilibre de classe dans les ensembles de données.

Cependant, lors de l'utilisation de SMOTE, il est important de s'assurer que cela n'affecte pas le test de l'ensemble de données ; intégration de SMOTE dans un pipeline de la bibliothèque imblearn en suivant les étapes suivantes :

VARIABLES EN ENTREES [20 variables independantes]
>KNNimputer>MinMaxScaler (preprocessors)
>SMOTE
>Algorithme
=>CIBLE['TARGET']

1. Importation des bibliothèques nécessaires : imblearn.pipeline, imblearn.over_sampling, sklearn.preprocessing, sklearn.impute, et l'algorithme choisi.
2. Creation d'un pipeline en utilisant pipeline () et enchaînement des étapes de prétraitement nécessaires, telles que KNNImputer pour gérer les données manquantes et MinMaxScaler pour la normalisation.
3. Ajout de SMOTE au pipeline en utilisant SMOTE () à partir de la bibliothèque imblearn.over_sampling.
4. Utilisation de l'algorithme choisi pour entraîner le modèle.
5. La Variable dépendante cible est ['TARGET'] (0 = crédit accordé / 1 = crédit refusé)

L'intégration de SMOTE dans un pipeline permet d'éviter d'utiliser SMOTE sur l'ensemble des données, ce qui pourrait entraîner une fuite de données (data leakage) et ainsi biaiser le test. Au lieu de cela, SMOTE ne sera appliqué qu'aux données d'entraînement, avant que l'algorithme soit entraîné. Cela garantit que l'évaluation du modèle est effectuée sur des données non modifiées.

Etape 2 : Création d'une fonction de cout métier ainsi que définition des métriques pour comparer les modèles :

La fonction coût métier est une mesure qui permet d'évaluer la performance du modèle en prenant en compte les coûts associés aux erreurs de classification. Cette fonction peut être utilisée dans l'algorithme d'optimisation pour guider le modèle vers une meilleure performance. La métrique d'évaluation est une mesure qui permet de quantifier la performance du modèle.

Le déséquilibre entre le nombre de bons et de moins bons clients doit donc être pris en compte pour élaborer un modèle pertinent.
Le déséquilibre du coût métier entre un faux négatif (FN - mauvais client prédit bon client : donc crédit accordé et perte en capital) et un faux positif (FP - bon client prédit mauvais : donc refus crédit et manque à gagner en marge).

On supposera que le coût d'un FN est dix fois supérieur au coût d'un FP, parce qu'un faux négatif est un client à qui on accordera un crédit alors que celui-ci ne sera pas en mesure de le payer (perte) alors qu'un faux positif est un client qui aurai dû obtenir le crédit mais qui ne l'a pas eu (manque à gagner).

La fonction de coût métier est définie de la manière suivante :

> ***10*FN + FP (où FN = nombre de Faux Négative dans la matrice de confusion, FP = nombre de Faux Positif), puis intégration de cette fonction de cout a make_scorer afin de prévenir l'algorithme d'entrainement que le but est de la minimiser (greater_is_better = False).***

Les métriques retenues pour comparer les modèles sont :

<u>Score-metier, l'accuracy , AUC_test,</u> ainsi que la precision, recall, et f1_score.

Bien que le SMOTE puisse aider à équilibrer les classes en augmentant le nombre d'exemples de la classe minoritaire, il ne prend pas en compte les coûts associés aux erreurs de classification. C'est pourquoi SMOTE a été combiné avec une approche basée sur le coût métier pour obtenir de meilleurs résultats de classification.

Etape 3 : Comparaison des modèles :

Création d'une fonction qui va permettre de publier sur l'interface MLFLOW les résultats obtenus avec 5 modèles avec les algorithmes de classification binaire suivants :

- ***DummyClassifier***
- ***KNeighborsClassifier***
- ***LogisticRegression***
- ***RandomForestClassifier***
- ***LGBMClassifier***

La stratégie d'entraînement et de sélection du modèle est effectuée par cross validation en utilisant le cout métier créé dans l'étape précédente via ***GridSearchCV.***

Deux à trois hyperparamètres pour chaque algorithme avec quelques valeurs pour afin de réduire le temps de traitement et réduire le poids des ressources de calculs.

Pour cette étape il est réalisé un ***pipeline d'entraînement de modèles reproductibles*** et mener des « expériences » sur les différents algorithmes.

Afin de réduire les temps de traitement le dataset d'entrainement représentera 10% du dataset soit (30750 ,20), et 5% pour les données de test soit (15376, 20).

Les ***résultats sont enregistrés dans MLFLOW***, et des résultats comparatifs des meilleurs résultats obtenus avec chacun des algorithmes sont affichés sur un tableau dans le notebook.

	X_shape	accuracy	precision	recall	F1_score	auc_test	best_sc	score_test	score_train	best_pa	model_name
0	(30750, 20)	0.919225	NaN	0.000000	NaN	0.500000	-4964.0	12420	24820	{'DummyClassifier__strategy': 'most_frequent'}	DummyClassifier
1	(30750, 20)	0.680151	0.128837	0.513688	0.206006	0.641000	-4309.0	10354	7824	{'KNeighborsClassifier__metric': 'manhattan', 'KNeighborsClassifier__n_neighbors': 11}	KNeighborsClassifier
2	(30750, 20)	0.684834	0.156893	0.663446	0.253773	0.740893	-3409.8	8608	17044	{'LogisticRegression__C': 10, 'LogisticRegression__penalty': 'l2'}	LogisticRegression
3	(30750, 20)	0.706686	0.156867	0.601449	0.248834	0.711281	-3728.6	8965	17597	{'RandomForestClassifier__max_depth': 6, 'RandomForestClassifier__min_samples_split': 7, 'RandomForestClassifier__n_estimators': 150}	RandomForestClassifier
4	(30750, 20)	0.776795	0.169982	0.454106	0.247368	0.702589	-3844.2	9534	17701	{'LGBMClassifier__learning_rate': 0.05, 'LGBMClassifier__n_estimators': 50}	LGBMClassifier

Le rapport montre que le modèle LGBMClassifier a le meilleur accuracy et le deuxième meilleur score AUC (aire sous la courbe ROC) parmi tous les modèles testés, avec un accuracy de 0,776 et un score AUC de 0,70. Cela indique que le modèle LGBMClassifier est capable de fournir de bonnes prédictions équilibrées pour les deux classes.

En conséquence, le choix du modèle LGBMClassifier est justifié car il fournit globalement de bons résultats en termes de d'accuracy et d'équilibre pour les deux classes, ce qui est essentiel pour la tâche de classification d'éligibilité a un emprunt.

En outre, il est également important de noter que le modèle LGBMClassifier a un temps d'entraînement relativement court par rapport à d'autres modèles, ce qui est un avantage important lorsqu'il est utilisé dans des applications en temps réel.

Partie 3 : Choix du modèle, optimisation et calcul du seuil optimal :

Etape 1 : Optimisation du modèle :

Après avoir sélectionné le modèle ayant les meilleurs résultats, l'étape suivante est d'optimiser le modèle. Pour cela, le package Python "Optuna" est utilisé, ce qui permet une optimisation plus rapide et flexible que l'optimisation par GridSearchCV.
Optuna est paramétré avec plus d'hyperparamètres et de valeurs, avec la même métrique de coût métier "my_scorer" (10*FN+FP) et comme paramètre "direction = minimize".

Les résultats de l'optimisation sont enregistrés dans MLFLOW pour une meilleure traçabilité.

Etape 2 : Calcul du seuil optimal du model final :

Une fonction est créée afin de calculer le seuil optimal de probabilité positive qui permet d'avoir le cout le plus bas avec le model final :

La fonction calcule le coût de la prédiction pour chaque seuil de probabilité possible, en utilisant la fonction "my_scorer" (10*FN+FP) qui prend en entrée les valeurs cibles réelles et les prédictions et renvoie le coût total de la prédiction en fonction des faux positifs et des faux négatifs. Le seuil de probabilité qui donne le coût le plus faible est sélectionné comme seuil optimal pour la classification.

Une fois le meilleur model avec ses hyperparamètres obtenus, une sauvegarde du model va etre effectue avec la librairie pickle.

Sauvegarde du notebook et du model final sauvegarde dans un GitHub privé (Pimouss75/Projet_7) avec GitHub desktop.

Partie 4 : Interprétabilité du modèle :

SHAP (SHapley Additive exPlanations) est une méthode qui permet d'expliquer les prédictions de modèles de machine learning.

Pour chaque observation du jeu de données, SHAP va calculer la contribution de chaque feature à la prédiction du modèle.

Cette contribution est appelée valeur SHAP (ou SHAP value) et représente l'impact de la feature sur la prédiction en comparant sa valeur pour cette observation à une valeur de référence (par exemple, la valeur moyenne de cette feature dans l'ensemble du jeu de données).

La somme des valeurs SHAP de toutes les features pour une observation donnée correspond à la différence entre la prédiction du modèle pour cette observation et la valeur moyenne des prédictions du modèle pour l'ensemble du jeu de données.

SHAP permet ainsi de visualiser l'importance de chaque feature dans la prédiction d'une observation donnée, mais également de calculer une importance globale des features à l'échelle de tout le jeu de données.

Le meilleur model avec ses hyperparamètres obtenus, il est important de comprendre comment il prend ses décisions. Pour cela, le package SHAP est utilisé pour obtenir une interprétabilité locale et globale.

SHAP permet de visualiser l'importance de chaque caractéristique dans les prédictions du modèle.

Cela peut être fait à l'échelle de l'ensemble des données, ou pour une observation individuelle, afin de mieux comprendre comment le modèle a pris une décision spécifique pour un client donné.

Deux types de graphiques sont créés :

- ***Un graphique permettant de représenter l'importance de chaque feature dans les prédictions à l'échelle de tout le jeu de données (interprétabilité globale).***

- ***Un graphique permettant de représenter l'importance de chaque feature dans la prédiction effectuée pour une observation donnée (interprétabilité locale, un client).***

Partie 5 : Performance du modèle.

Afin de vérifier que notre modèle reste performant lorsqu'il est utilisé sur le dataset de test, une analyse de Datadrift entre le dataset train et le dataset test est effectuée afin de vérifier que les variables n'ont pas trop 'dérivées' entre les deux datasets à travers un rapport d'analyse au format HTML crée par la librairie Evidently.

Le rapport décrit les résultats de l'analyse de dérive de données pour un ensemble de données contenant 20 caractéristiques. La dérive de données se produit lorsque la distribution des données change au fil du temps, ce qui peut affecter les performances du modèle. **Le rapport montre que la dérive de données a été détectée pour une seule des caractéristiques (DAYS_LAST_PHONE_CHANGE), ce qui représente 5% de l'ensemble de données.**
Pour les autres caractéristiques, la dérive de données n'a pas été détectée.

Le rapport fournit également des informations sur les mesures de distance utilisées pour détecter la dérive de données, telles que la distance de Wasserstein et la distance de Jensen-Shannon. Ces mesures sont des moyens de quantifier à quel point les distributions de deux ensembles de données sont différentes.

En résumé, le rapport indique que la dérive de données n'est pas un problème majeur pour cet ensemble de données et que le modèle peut être utilisé avec confiance.

Le rapport généré est enregistré dans le GitHub (privé)

Partie 6 : Création de l'API et du Dashboard

Un Dashboard est créé, demande à l'utilisateur de fournir le numero de client a 6 chiffres. Le Dashboard génère une réponse à une demande de crédit avec une réponse négative ou positive, des informations relatives au client, son graphique SHAP ainsi que celui de tous les clients de la manière suivante :

- Lorsque la case ***« Prediction »*** aura été cliquée, la réponse à la demande de crédit s'affiche ainsi que l'explication et la valeur de chaque variable dont la description est disponible.

- Lorsque la case ***« Interpretation du client »*** est cliquée, la réponse à la demande de crédit s'affiche ainsi que la ***feature importance locale*** sous forme de graphique est générée en utilisant SHAP.

- Lorsque la case ***« Comparaison avec les autres clients »*** est cliquée, la ***réponse à la demande de crédit*** s'affiche ainsi que la ***feature importance locale*** sous forme de graphique est générée en utilisant SHAP ainsi que l'***importances des variables pour tous les clients de test (globale)***, soit la globalité des clients.

L'affichage du Dashboard va etre construit avec 2 outils :

- Une application **Dashboard *(app.py) réalisée avec Streamlit*** qui fait appelle l'API via une url pour récupérer la réponse à la demande de crédit, récupérer les variables et leurs valeurs respectives client afin de les décrire sous forme d'un ***tableau*** et afficher le ***graphique des features importances du client avec SHAP (locale)*** ou récupérer les variables de tous les clients de test afin d'en afficher le ***graphique d'importance des features sur tous les clients (globale)***.

- Une **API *(main.py) réalisée avec Fastapi***, qui renverra les informations décrites ci-dessus sur demande de Streamlit (***calcul de la prédiction du client, variables du clients et variables de tous les clients***).

L'interface a été simplifiée, le titre est de grande taille, les cases à cliquer sont grandes et ont un fond noir avec des caractères blancs, les graphiques sont de grande taille, tout cela afin de faciliter leur lecture par les malvoyants.

Le tout est enregistré dans le GitHub(privé) apres avoir été teste en local avec pycharm.

Partie 7 : Déploiement du Dashboard

Dans un premier temps :

Les applications (Dashboard et l'API) sont hébergées sur un serveur cloud AWS EC2 :
Le déploiement s'effectue par clonage du projet sauvegardé sur GitHub sur le serveur AWS.
Les requirements ainsi que les packages à installer le sont sur le serveur.
Cela permet de rendre l'application accessible depuis n'importe quel navigateur web et de la rendre disponible pour les utilisateurs

Puis :

Mise en place avec GitHub Actions d'un déploiement automatisé (ou mise à jour) déclenchée par une mise à jour de la branche dev du projet sur GitHub, déploiement effectué à la condition que le test de fonctionnement du model soit un succès :
Un script de pipeline CI/CD (intégration continue et livraison continue) est créé, elle est déclenchée à chaque push sur la branche "dev". Ce pipeline effectue les actions suivantes :

1. Récupération du code source depuis la branche "dev".
2. Installation de Python et des dépendances du projet.
3. Installation de pytest et exécution des tests.
4. Si les tests passent avec succès, le pipeline continue, sinon elle s'arrête et envoie une notification de défaillance.
5. Si les tests passent avec succès, le pipeline met à jour la branche "main" en fusionnant les modifications apportées à la branche "dev".

6. Si les tests passent avec succès et que la fusion des modifications avec la branche "main" est réussie, le pipeline configure et déploie une instance EC2 sur AWS en utilisant les clés SSH pour se connecter à l'hôte distant.
7. Si les tests échouent, le pipeline envoie une notification indiquant que les tests ont échoué.

Partie 8 : Limites améliorations possibles :

En conclusion, **le modèle créé est satisfaisant et a été déployé** sous forme d'une application qui peut etre utilisée par les agents bancaires.
Il existe **plusieurs améliorations possibles** qui peuvent être apportée pour en augmenter la précision et la capacité de généralisation, ***parmi lesquels*** :

-L'***ajout de variables supplémentaires*** et pertinentes.

-L'***utilisation de techniques plus avancées d'apprentissage automatique***, telles que les réseaux de neurones profonds ou les algorithmes de renforcement, pourraient permettre une meilleure prise en compte des interactions complexes entre les variables et améliorer la précision des prévisions. L'utilisation de ces techniques peut nécessiter une expertise spécialisée et des ressources informatiques supplémentaires.