Implémenter un modèle de Scoring :

Note Méthodologique

Projet n°7 - Data Scientist -
OpenClassrooms – Kchaou Mariem

https://github.com/KKmaryam

Contexte et Objectifs :

La société financière *Prêt à dépenser* propose des crédits à la consommation pour des personnes ayant peu ou pas du tout d'historique de prêt.

- ***Implémentation d'un modèle de scoring :***

L'entreprise souhaite mettre en oeuvre un outil de scoring crédit qui calcule la probabilité qu'un client rembourse son crédit, puis classifie la demande en crédit accordé ou refusé. Elle souhaite donc développer un algorithme de classification en s'appuyant sur des sources de données.
Les données utilisées pour ce projet sont une base de données de 356 251 clients comportant 798 features (âge, sexe, emploi, logement, revenus, informations relatives au crédit, notation externe, etc).
Les données originales sont téléchargeables sur Kaggle à cette **adresse.**

- ***Dashboard interactif réalisé avec Streamlit :***

De plus, les chargés de relation client ont fait remonter le fait que les clients sont de plus en plus demandeurs de transparence vis-à-vis des décisions d'accorder un crédit. Cette demande de transparence des clients va tout à fait dans le sens des valeurs que l'entreprise veut représenter.
Prêt à dépenser décide donc de développer un dashboard interactif pour que les chargés de relation client puissent à la fois expliquer de façon la plus transparente possible les décisions d'accorder un crédit, mais également permettre à leurs clients de disposer de leurs informations personnelles et de les explorer facilement.
Le dashboard réalisé avec Streamlit est accessible en **cliquant ici** (Heroku).
Tous les fichiers sont accessibles sur **Github**.

- ***API réalisé avec Flask :***

API permettant d'appeler le score à partir de l'ID du client.
L'API réalisé avec Flask est accessible en **cliquant ici** (Heroku).
Tous les fichiers sont accessibles sur **Github**.

Note Méthodologique

Cette note méthodologique présente les techniques employées pour la construction du modèle de scoring ainsi que les outils mis en place pour l'interprétation.

Pour répondre à cet objectif, nous détaillons brièvement :

- La méthodologie d'entraînement du modèle.
- La fonction coût métier, l'algorithme d'optimisation et la métrique d'évaluation.
- L'interprétation globale et locale du modèle.
- Les limites et les améliorations possibles.

I. 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 Fbeta score.

II. La méthodologie d'entraînement du modèle :

Le features engineering des données a été effectué à partir du kernel suivant : **LightGBM with Simple Features**
Nous nous retrouvions désormais avec un fichier contenant 356251 clients et 798 colonnes.
Au vu du grand nombre de variables, nous avons opté pour la suppression des colonnes qu'ont plus 60% de valeurs manquantes et nous avons fait l'imputation des autres avec la moyenne.
Nous nous sommes retrouvés avec un fichier contenant 356251 lignes et 586 variables.
Nous avons donc gardé un fichier de 10000 clients en tant que fichier d'entrainement.

1. Prétraitement 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 entrainés.

- Dummy Classifier
- SVC
- Logistic Regression
- RandomForestClassifier
- DecisionTree Classifier
- KNeighborsClassifier
- MultinomialNB
- Gradient Boosting Classifier
- LGBMClassifier
- XGBClassifier

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.
Tois 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. Un algorithme de création d'observation va être testé : SMOTE.

SMOTE (Synthetic Minority Oversampling Technique) est considéré comme l'un des algorithmes d'échantillonnage de données les plus populaires et les plus influents dans la Machine Learning et l'exploration de données. Avec SMOTE, la classe minoritaire est sur-échantillonnée en créant des exemples «synthétiques» plutôt qu'en sur-échantillonnant avec remplacement. Utilisation de:
from imblearn.over_sampling (imbalanced-learn Python library SMOTE class)

```
print("Label 1, Before using SMOTE: {} ".format(sum(y_train==1)))
print("Label 0, Before using SMOTE: {} ".format(sum(y_train==0)))
```

```
Label 1, Before using SMOTE: 517
Label 0, Before using SMOTE: 6483
```

```
print("Label 1, After using SMOTE: {}".format(sum(y_train_res==1)))
print("Label 0, After using SMOTE: {}".format(sum(y_train_res==0)))
```

```
Label 1, After using SMOTE: 6483
Label 0, After using SMOTE: 6483
```

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, …)

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 le teste sur un jeu de donnée équilibré apporterai une grande différence (dans le cas de SMOTE notamment).

A noter également, l'entrainement se fera à l'aide d'un RandomizedSearchCV (cv =3). Ceci afin de tester différentes possibilités.

Paramètre :

Modèles	Paramètres
Dummy Classifier	strategy
RandomForestClassifier	random_state, n_estimators, max_depth, class_weight
SVC	random_state, C, class_weight
LogisticRegression	random_state, C, penalty, class_weight
DecisionTreeClassifier	random_state, max_depth, min_samples_split, class_weight
KNeighborsClassifier	n_neighbors
MultinomialNB	alpha
GradientBoostingClassifier	random_state, n_estimators, max_depth
LGBMClassifier	boosting_type, verbose, n_jobs, random_state, objective, n_estimators, learning_rate, colsample_bytree, reg_alpha, reg_lambda, subsample, max_depth, num_leaves, min_child_samples, min_child_weight, is_unbalance
XGBClassifier	objective, random_state, tree_method, predictor, interaction_constraints, scale_pos_weight, max_delta_step

L'optimisation des paramètres s'est fait à l'aide du module RandomizedSearchCV (cv =3)).

Ceci afin de maximiser ROC_AUC et Fbeta_score.

Le modèle choisi : **LGBMClassifier**

Paramètres à optimiser:

n_estimators
learning_rate
max_depth
num_leaves
subsample
min_child_weight
min_child_samples
reg_alpha
reg_lambda
colsample_bytre

III. La fonction coût métier, l'algorithme d'optimisation et la métrique d'évaluation:

Les modèles ont été entraînés dans la fonction de cross validation et testés suivant différentes combinaisons d'hyperparamètres.
Dans le jeu de données de base, il y a une part de 93,1 % des clients qui n'ont pas d'incident de paiement, tandis que 6,9 % des clients ont eu des incidents.
La matrice de confusion est la suivante :

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

Du point de vue d'une banque, on cherchera à éviter de mal catégoriser un client avec un fort risque de défaut (pertes financières et frais de recouvrements qu'on imagine importants). On cherche donc à minimiser le pourcentage de faux négatifs et à maximiser le pourcentage de vrais positifs.
Autrement dit, on va chercher à maximiser le Recall

$$\boldsymbol{Recall = \frac{vrais\ positis}{vrais\ positifs\ + faux\ négatifs}}$$

Par ailleurs on cherche à maximiser le nombre de clients potentiels donc à ne pas tous les classer en défaut.
On cherche donc à éviter d'avoir un trop grand nombre de faux positifs.
On cherche donc à maximiser la Precision

$$\boldsymbol{Precision = \frac{vrais\ positifs}{vrais\ positifs\ + faux\ positifs}}$$

Pour notre problématique métier, le Recall est plus important que la Precision car on préfèrera semblablement limiter un risque de perte financière plutôt qu'un risque de perte de client potentiel.
On cherche donc une fonction qui optimise les 2 critères en donnant plus d'importance au Recall.
Fonction permettant de faire cela : F Beta Score : https://en.wikipedia.org/wiki/F1_score avec Beta le coefficient d'importance relative du Recall par rapport à la Precision.

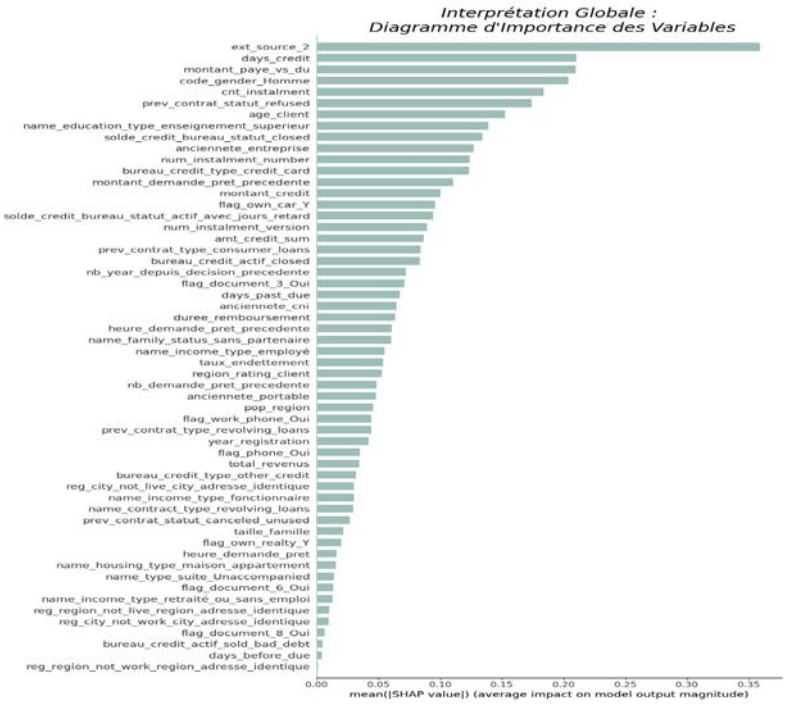
$$\boldsymbol{F_\beta = (1+\beta^2).\frac{precision\ .recall}{\beta^2\ .precision\ + racall}}$$

$$\boldsymbol{F_\beta = (1+\beta^2).\frac{true\ positive}{(1+\beta^2).true\ positive\ + \beta^2\ .false\ negative + false\ positive}}$$

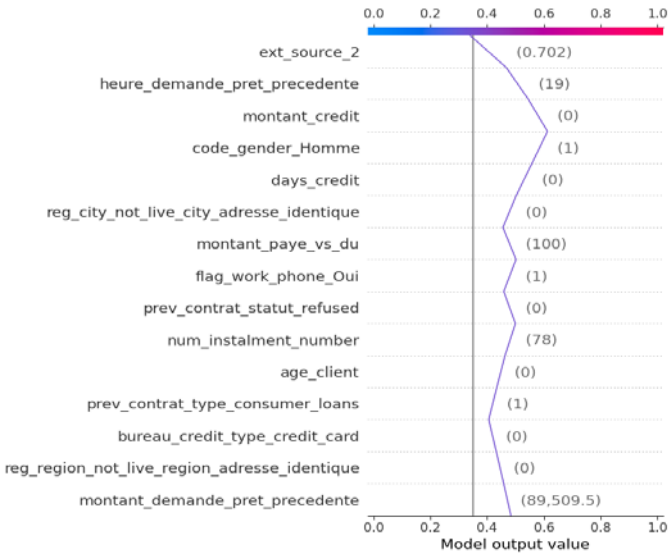
On cherchera à **maximiser** cette métrique.
Résumé :
• Si $\beta > 1$, on accorde plus d'importance au Recall. On minimise les faux négatifs.
• Si $\beta < 1$, on accorde plus d'importance à la Precision. On minimise les faux positifs.
• Si $\beta = 1$, la précision et le Recall ont la même importance. C'est le $F1score$.
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 = 2.
La meilleure combinaison d'hyperparamètres a été retenue pour chaque algorithme. Un modèle stacking a été réalisé en combinant tous les modèles pour chaque approche (Sample Weights et Smote). Le modèle ayant le meilleur score en cross validation sur le jeu de training a été retenu : il s'agit du modèle LGBMClassifier. Ce modèle dispose d'une bonne performance en combinant des modèles de machine learning différents.

IV. L’interprétabilité globale et locale du modèle:

Nous avons retenu des interprétations qui permettent une lecture aisée pour les chargés de relation clientèle. La technique des permutations calcule les variations de score d'une feature quand toutes les valeurs de cette feature sont mélangées. Plus cette variation est importante, plus la feature est importante. Pour notre modèle, les features les plus importantes sont regroupées dans le graphique suivant



La technique SHAP locale explique la prédiction pour chaque client en calculant la contribution de chaque feature à la prédiction. Par exemple pour le client 158201, les contributions des 15 features qui impactent le plus sa probabilité de ce client sont les suivantes :



V. Les limites et les améliorations possibles:

La modélisation effectuée dans le cadre du projet a été effectuée sur la base d'une hypothèse forte: la définition d'une métrique d'évaluation : le F Beta Score avec Beta fixé suivant certaines hypothèses non confirmées par le métier. L'axe principal d'amélioration serait de définir plus finement la métrique d'évaluation en collaboration avec les équipes métier.

Par ailleurs, la partie de traitement préalable du jeu de données a été abordée de façon superficielle en réutilisant un notebook issu de Kaggle qui se base uniquement sur une table du jeu de données. Il y a très probablement l'opportunité d'améliorer la modélisation en utilisant d'autres features des données fournies, ainsi qu'en créant de nouvelles features en collaboration avec les équipes métier.