Note Méthodologique :

Projet 7 : Implémentez un modèle de scoring

Dashboard : https://projet7openclassroom-credit.streamlit.app/
Lien vers git : https://github.com/Bright-Sheep/projet_7_openclassrooms_flask
Lien vers git : https://github.com/Bright-Sheep/projet_7_openclassrooms_dashboard

Objectif :

On veut faire un modèle pour prédire quel client est susceptible de ne pas rembourser son crédit.
On a à notre disposition 10 dataframes d'informations relatives aux clients afin d'entraîner nos modèles.
Ce modèle est destiné à être utilisé dans un dashboard pour permettre de justifier les décisions d'autorisations de crédit aux clients d'un organisme de prêt.

Démarche de modélisation :

Notebook : Préparation des données.ipynb
Pour entraîner nos modèles, on commence par regrouper toutes nos informations dans un seul DataFrame.
Pour cela, on a récupéré un notebook sur Kaggle qui s'occupe de fusionner les tableaux et de faire le One Hot Encoding des colonnes catégoriel.
Le notebook : https://www.kaggle.com/code/jsaguiar/lightgbm-with-simple-features/script

Notebook : Prédiction.ipynb
On se retrouve avec un tableau de 307 507 lignes pour 798 colonnes.
On remarque que les clients désignés comme mauvais payeur représentent 8% des clients de notre Dataframe d'entraînement.
Il va falloir prendre en compte l'imbalance entre les classes.

Tout d'abord, on commence par sélectionner les features ayant le plus d'influence sur notre Target.
Pour le moment, on choisit les 20 features ayant le plus d'influence sur la Target.
On se débarrasse ensuite des colonnes où les features ne sont pas renseignés.
On se retrouve avec un DataFrame de 9433 lignes, ce qui sera suffisant pour entraîner nos modèles.
Après vérification, on constate que l'imbalance n'a pas trop changé par rapport au DataFrame de départ. Avec 9,5% de clients mauvais payeurs, c'est le cas.
Pour s'occuper de l'imbalance, on utilisera la méthode SMOTE **(Synthetic Minority Oversampling Technique)** pour créer artificiellement des clients de la classe minoritaire.

On fait un train test split pour entraîner et évaluer nos modèles.

Comme **fonction de coût métier**, on utilisera le fbeta_score avec un beta de 10, qui pénalise les faux négatifs.

Comme **fonction de coût technique** le roc_auc_score, (l'aire sous la courbe de ROC) qui maximise le score quand les classes sont correctement prédites.

Comme **modèle**, on utilisera le Random_Forest_Classifier, la LogisticRegression, le KNN Classifier, et le Gradiant Boosting Classifier.

Comme **métrique d'évaluation**, on utilisera le fbeta_score avec un beta de 10 car on veut éviter en priorité les clients mauvais payeurs (Faux négatifs).

On met le modèle et le SMOTE dans un pipeline et on évalue le modèle avec différents paramètres grâce à un RandomizedSearchCV.
Après l'entraînement de tous les modèles, on se retrouve avec le rapport ci-dessous:

	model	score	best_params	best_score	train	test	fbeta_score_test
0	search_tree	fbeta	{'classification__n_estimators': 13, 'classifi...	0.526590	0.527499	0.528562	0.528562
0	search_tree	roc_auc	{'classification__n_estimators': 13, 'classifi...	0.613108	0.611411	0.614265	0.487458
0	search_logi	fbeta	{'classification__tol': 0.0047508101621027985,...	0.642679	0.640188	0.625189	0.625189
0	search_logi	roc_auc	{'classification__tol': 0.0001668100537200059,...	0.609928	0.616308	0.591660	0.550495
0	search_KNN	fbeta	{'classification__n_neighbors': 10}	0.416634	0.932034	0.368335	0.368335
0	search_KNN	roc_auc	{'classification__n_neighbors': 10}	0.535191	0.831119	0.515405	0.375276
0	search_gradiant	fbeta	{'classification__n_estimators': 10, 'classifi...	0.492654	0.493365	0.499841	0.499841
0	search_gradiant	au_roc	{'classification__n_estimators': 43, 'classifi...	0.618447	0.623650	0.616262	0.457393

La **LogisticRegression** nous donne les meilleurs résultats.

Notebook : Fine_Tuning_Model.ipynb
On continue d'optimiser notre **LogisticRegression.**
Cette fois, on entraînera nos données sur un échantillon du X_train pour gagner du temps, avant d'entraîner le modèle finale sur l'ensemble du X_train.

On entraînera le **LogisticRegression** avec les deux métriques utilisées auparavant.

	model	score	best_params	best_score	train	test	fbeta_score_test
0	search_logi	fbeta	{'classification__tol': 0.0022570197196339213,...	0.626346	0.654305	0.662170	0.662170
0	search_logi	roc	{'classification__tol': 0.0016297508346206436,...	0.655095	0.664117	0.667156	0.630892

On entraîne ensuite sur l'ensemble de notre X_train :

	model	score	best_params	best_score	train	test	fbeta_score_test
0	search_logi	fbeta	{'classification__tol': 0.0022570197196339213,...	0.637733	0.648908	0.63121	0.63121

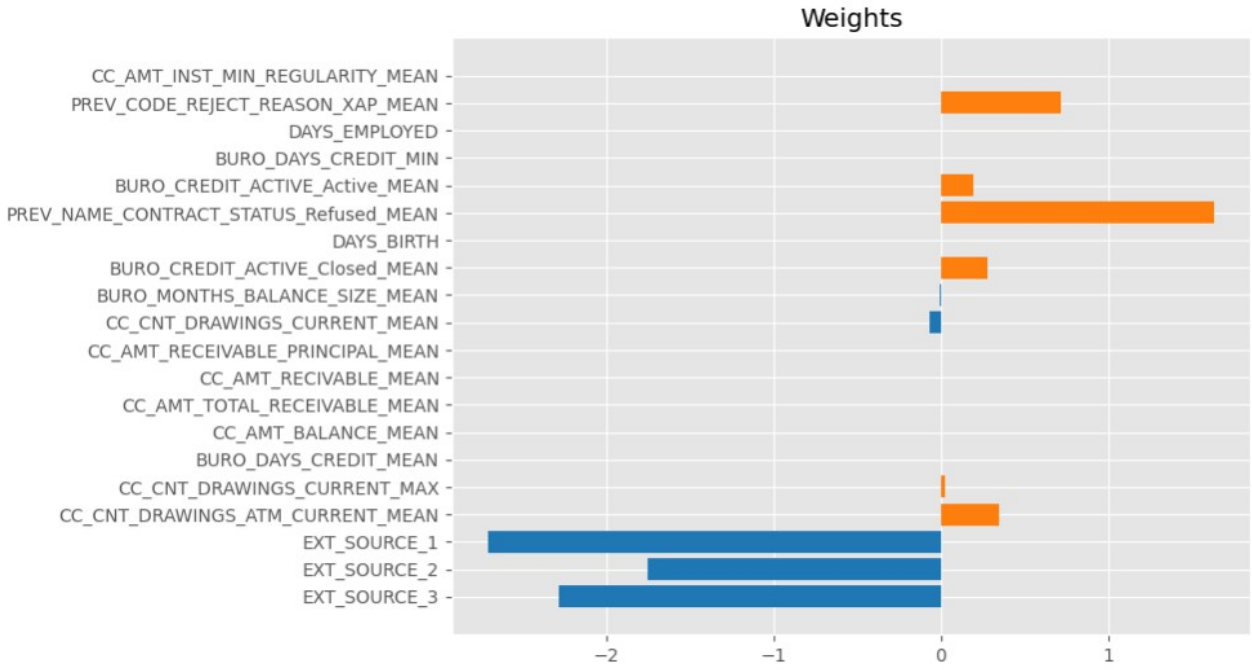
	model	score	best_params	best_score	train	test	fbeta_score_test
0	search_logi	roc	{'classification__tol': 0.0016297508346206436,...	0.669061	0.670158	0.672751	0.637546

Au final, on choisit le modèle de **LogisticRegression** entraîné avec **Roc Auc** car il overfit moins et donne un meilleur **fbeta_score**.

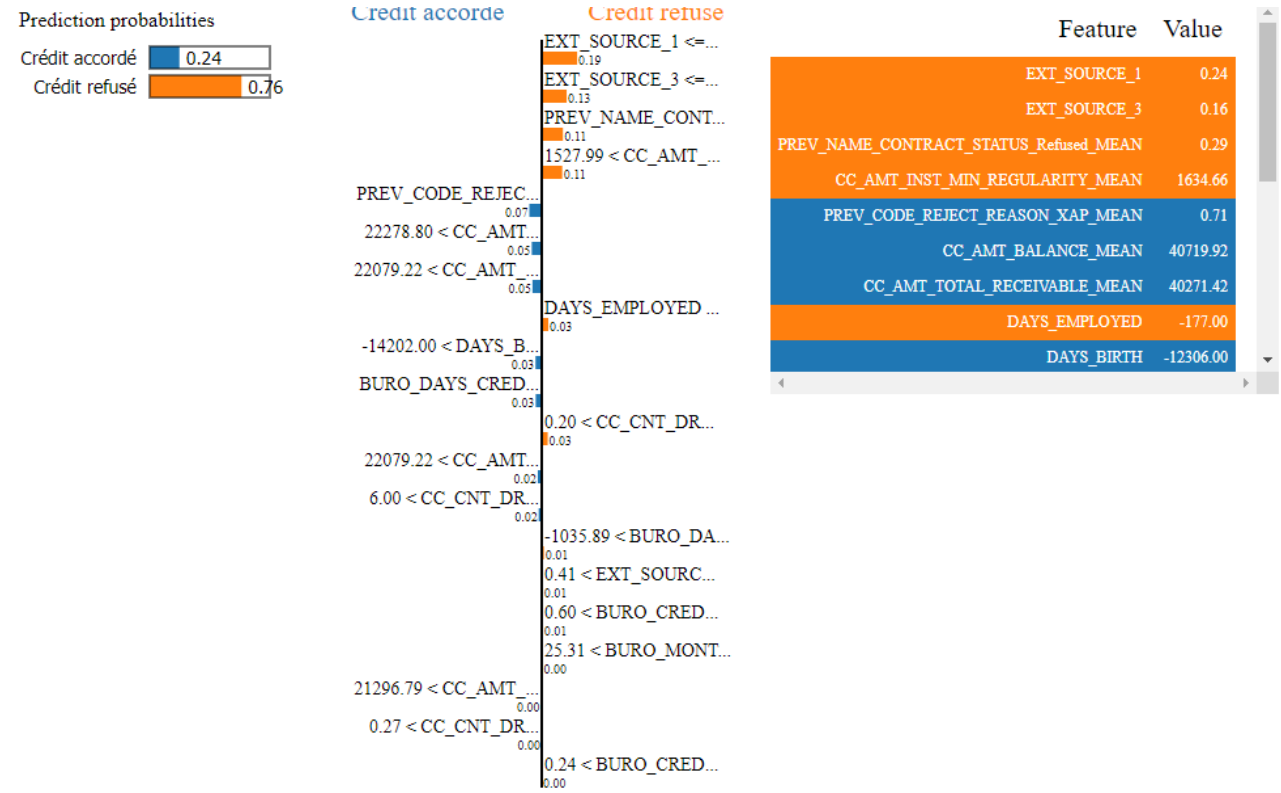
Interprétabilité locale et globale du modèle :

Notebook : Meilleur modèle.ipynb

On évalue le poids global des features :



On évalue le poids local des features grâce à la librairie lime_tabular qui nous donne l'importance locale des features pour un client, et permet de préciser le nombre de features que l'on veut voir afficher :



On peut voir qu'une feature qui a peut d'importance globalement (ici CC_AMT_INST_MIN_REGULARITY_MEAN) peut avoir beaucoup d'importance localement.

<u>Limite et améliorations :</u>

On fait des prédictions correctes, mais pour que le client puisse les interpréter, il faudrait expliciter les noms des features.

On pourrait tester plus de modèles pour trouver le meilleur score possible.

Il faut trouver comment combler les features non remplis pour pouvoir faire fonctionner le modèle sur l'ensemble des clients, pas seulement ceux dont on connaît toutes les features.
On pourrait ainsi entraîner les modèles sur un plus grand nombre de données.

Il faudrait faire des tests pour savoir à quelle fréquence il faut mettre à jour le modèle.

Des tests ont été effectué pour fine tuner le threshold du modèle finale. Il pourrait être intéressante de l'intégrer au dashboard final :