Méthodologie

1. Présentation des modèles

DummyClassifier

C'est un classifieur dit "naïf" dans le sens où il ne tient pas compte des données d'entrée pour faire des prédictions. Il est souvent utilisé comme point de référence pour comparer les performances d'un modèle réel par rapport à un modèle qui prédit de manière simple.

Hyperparamètres

- *most_frequent* : prédit la target la plus fréquente dans les données d'entraînement
- *prior* : fait des prédictions en respectant la distribution de notre target
- *uniform* : chaque classe (0 ou 1) a une probabilité égale d'être prédite
- *stratified* : prédit un échantillon de manière probabiliste
- *constant* : la classe à prédire est fournie par l'utilisateur

Régression Logistique

C'est un classifieur linéaire utilisant une fonction logistique (ou fonction sigmoïde) pour serrer les prédictions dans un intervalle compris entre 0 et 1. Si cette probabilité est supérieure à un certain seuil, l'échantillon est classé dans la classe positive, sinon il est classé dans la classe négative.

Cette probabilité possède la formule suivante :

$$p = \frac{e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_k x_k}}{1+e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_k x_k}}$$

avec x = valeurs des features de l'échantillon et β = coefficients à déterminer en maximisant la vraisemblance.

Hyperparamètre

- *régularisation* : Ridge

LightGBM

C'est un classifieur non linéaire utilisant des arbres de décision. Il se sert d'une approche basée sur les feuilles plutôt que sur la profondeur pour faire croître ses arbres. Cela lui permet d'être plus rapide et de consommer moins de mémoire comparé à d'autres modèles utilisant des arbres de décision tels que *XGBoost*.

Hyperparamètres

- *learning_rate* : contrôle la contribution de chaque arbre
- *num_leaves* : nombre maximum de feuilles
- *n_estimators* : nombre d'arbres à entraîner

2. Entrainement du modèle

2.1. Imputation des données d'entraînement (ou non)

Pour le *DummyClassifier* et la *Régression Logistique*, il est nécessaire d'imputer nos données pour que la simulation puisse s'effectuer. L'imputation choisie est la stratégie "mean".
Pour le *LightGBM*, il est possible de se passer de cette imputation et de lui donner les données brutes (possédant des NaN). Ce qui a pour avantage d'entraîner notre modèle sur des données qui ne sont pas dénaturées.

2.2. Standardisation des données d'entraînement (*StandardScaler*)

Le *StandardScaler* transforme les caractéristiques de telle sorte à ce que leur moyenne soit nulle et que leur écart type soit égal à 1. Standardiser les données avant d'entraîner nos modèles donne plusieurs avantages :

- Convergence plus rapide : pour des algorithmes basés sur la descente de gradient (comme la régression logistique), elle peut aider à accélérer la convergence.
- Évite la dominance des caractéristiques : la mise à l'échelle garantit que toutes les caractéristiques ont le même poids initial.
- Importance des caractéristiques : si les caractéristiques sont mises à l'échelle, leurs poids peuvent être comparés plus facilement et directement.

2.3. Validation Croisée (*StratifiedKFold*) {Nombre de folds = 5}

Dans des problèmes de classification où il y a un important déséquilibre de classe sur la target (c'est le cas dans ce projet), il est essentiel que lorsqu'une validation croisée s'effectue, chaque fold soit représentatif du dataset complet. C'est pour ça qu'il a été décidé de réarranger les données de manière à ce que chaque fold conserve les proportions similaires d'échantillons par rapport à l'ensemble du dataset. Cette technique s'appelle la stratification et c'est pour ça que la méthode *StratifiedKFold* de scikit-learn a été utilisée.

2.4. Intégration de l'hyperparamètre Seuil (*threshold*)

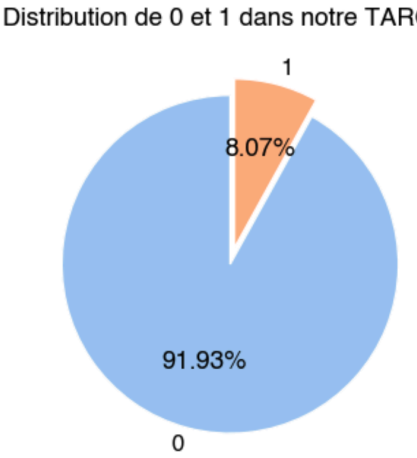
Les modèles de classification donnent en sortie une probabilité permettant de conclure si oui ou non, l'accord d'un crédit à un client peut s'effectuer. En général, la valeur par défaut du seuil est de 0,5. Cependant, il peut arriver que cette valeur puisse ne pas être optimale. En décidant d'ajuster ce seuil, il est possible d'optimiser le nombre de faux positifs et de faux négatifs.

2.5. Enregistrement des résultats avec MLflow

MLflow est une plateforme de tracking qui gère le cycle de nos modèles. Cela permet de suivre et de comparer les différents essais et modèles, facilitant la reproductibilité et le déploiement. Dans ce projet, MLflow enregistre les hyperparamètres, les sorties ainsi que les modèles sous forme de format pickle.

3. Traitement du déséquilibre des classes

La target (0 = Aucun problème de payement | 1 = Soucis de payement) est très déséquilibrée.




```
Nombre de 0: 282682
Nombre de 1: 24825
```

Un tel déséquilibre soulève de nouvelles problématiques :

❖ Performance trompeuse : un modèle qui prédit toujours la classe majoritaire obtiendra une précision (accuracy) élevée malgré le fait qu'il ne soit pas vraiment efficace.

❖ Biais du modèle : car il est davantage "récompensé" pour avoir correctement prédit cette classe étant donnée sa prédominance.

Pour résoudre ces problématiques, nous pouvons utiliser la méthode *class_weights* incluse dans les modèles de Scikit-learn.

```python
nb_0 = len(df_classification[df_classification["TARGET"] == 0])
nb_1 = len(df_classification[df_classification["TARGET"] == 1])
class_weights = {0: 1, 1: nb_0 / nb_1}
```

Ce paramètre permet de donner un poids différent à différentes classes. En faisant cela, il est possible d'augmenter le coût des erreurs de classification sur la classe minoritaire, ce qui rend le modèle plus attentif à cette classe lors de l'apprentissage. Ici, la classe 1 (minoritaire) a un poids qui est égal au ratio du nombre d'échantillons ayant la classe 0 par rapport à la classe 1.
Cette méthode offre une manière simple d'aborder le problème des classes déséquilibrées sans avoir besoin de techniques de sur-échantillonnage ou de sous-échantillonnage.

4. Fonction coût métier, Algorithme d'optimisation et Métrique d'évaluation

4.1. Fonction coût métier

L'objectif premier d'une banque lorsqu'elle octroie des crédits à ses clients est d'optimiser ses bénéfices. Cependant, deux scénarios peuvent entraîner des pertes financières :

- Faux positifs (FP) : cas où la banque estime qu'un client ne sera pas capable de rembourser son prêt, alors qu'en réalité, il le pourrait.
- Faux négatifs (FN) : cas où la banque estime que le client remboursera le crédit, mais ce dernier finit par défaillir.

Il est intéressant de noter qu'un faux négatif est nettement plus coûteux pour la banque qu'un faux positif. Il peut être supposé, à titre illustratif, que le coût associé à un FN soit dix fois supérieur à celui d'un FP.
Ainsi, l'enjeu est de minimiser une fonction dite de **coût métier** formulée ci-dessous :

$$\textit{Coût métier} = \textit{Somme}(10*FN+FP)$$

4.2. Algorithme d'optimisation

En apprentissage automatique, l'optimisation des hyperparamètres est cruciale pour obtenir les meilleures performances d'un modèle. L'une des méthodes pour réaliser cette optimisation est celle de l'arbre de Parzen. Ce dernier utilise une approche bayésienne, cela veut dire qu'il évalue la probabilité que des hyperparamètres particuliers produisent de bons résultats en fonction des essais précédents. Pour guider cette optimisation, nous définissons une fonction objective que l'on souhaite soit minimiser soit maximiser.
Au fur et à mesure des essais, l'arbre de Parzen apprend et adapte ses suggestions, se concentrant davantage sur les régions de l'espace des hyperparamètres où les performances seraient probablement meilleures. Le package *Optuna* fut utilisé pour appliquer cette méthode d'optimisation.

4.3. Métrique d'évaluation

Comme précisé dans la partie **4.2**, l'arbre de Parzen souhaite optimiser une fonction objective pour qu'il puisse trouver le meilleur jeu d'hyperparamètres. Ici, cette fonction objective sera le **coût métier**.

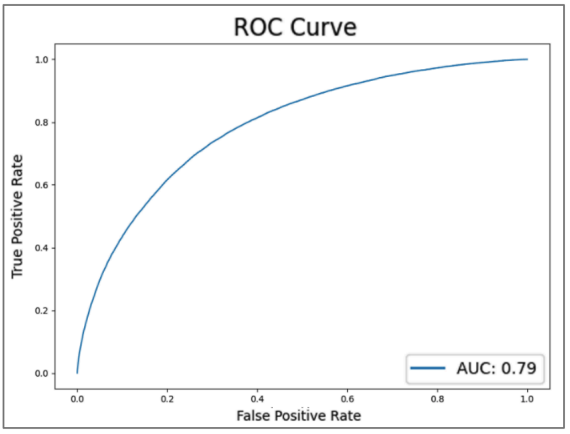
5. Synthèse des résultats

Pour le *LightGBM* et la *Régression Logistique*, 50 essais à *Optuna* ont été accordés, lui permettant ainsi d'identifier de manière optimale les bons jeux d'hyperparamètres.

	Model	Strategy	C	Learning Rate	Num Leaves	Threshold	AUC	Accuracy	Business Score
0	LightGBM	NaN	nan	0.014864	47.000000	0.480000	0.786902	0.727163	30064
1	LogisticRegression	NaN	115.866475	nan	nan	0.510000	0.772723	0.719424	31386
2	DummyClassifier	uniform	nan	nan	nan	nan	0.500000	0.919271	49650

Il est souhaitable d'optimiser en priorité le **coût métier** (Business Score) mais il est toujours intéressant de regarder d'autres métriques pour évaluer de manière plus globale les performances de nos modèles :

- *Accuracy* : C'est le ratio du nombre total de prédictions correctes sur le nombre total de prédictions. Cette valeur peut être trompeuse surtout si les classes sont fortement déséquilibrées (comme c'est le cas ici).
- *AUC (Area Under the Curve)* : Elle mesure l'aire sous la courbe ROC (Receiver Operating Characteristic). La courbe ROC trace le taux de vrais positifs par rapport au taux de faux positifs à différents seuils de classification. Une AUC de 1 indique une classification parfaite, tandis qu'une AUC de 0,5 indique une performance équivalente à une prédiction aléatoire.



LightGBM

Les résultats du modèle suggèrent qu'utiliser un seuil de 0,5 n'est pas forcément idéal. En effet, la *Régression Logistique* optimise le Business Score pour un seuil de 0,51 tandis qu'il est à 0,48 pour le *LightGBM*.

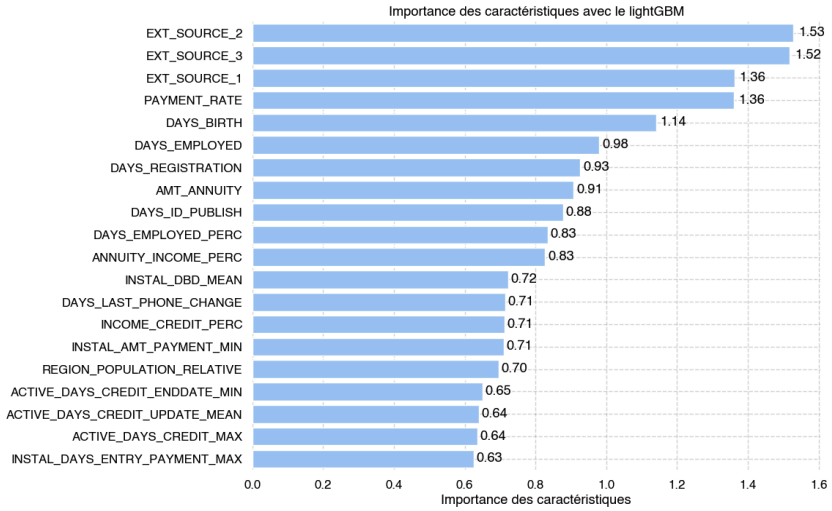
En conclusion, compte tenu de ces résultats, le modèle à utiliser en production est le suivant :

LightGBM (Num Leaves = 55 | Learning rate = 0,017 | Threshold = 0,48)

6. Interprétabilité globale et locale du modèle

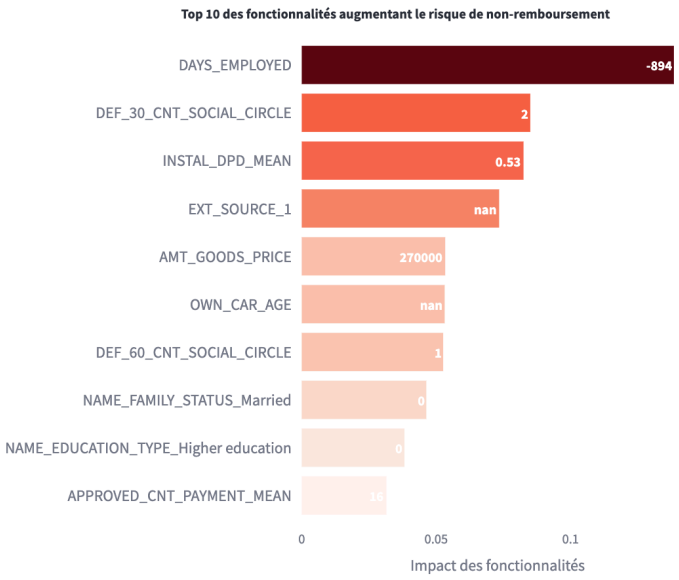
6.1. Interprétabilité globale

L'interprétabilité globale d'un modèle se réfère à notre capacité à comprendre comment le modèle prend ses décisions pour l'ensemble d'un jeu de données. Pour répondre à ces questions, l'importance des features a été utilisée. Cette métrique est intéressante pour détecter du data leakage (fuite de données). Le data leakage se produit lorsque des informations provenant de la variable cible s'infiltrent dans les caractéristiques utilisées pour former le modèle, ce dernier pourrait par conséquent afficher une performance artificiellement élevée.



Pour le LightGBM, il est constaté qu'aucune Feature est anormalement élevée relativement aux autres. Cela permet de conclure qu'il n'y a pas de Data Leakage.

6.2. Interprétabilité locale



Exemple d'utilisation de SHAP

Contrairement à l'interprétabilité globale, qui cherche à comprendre comment un modèle fonctionne sur l'ensemble de ses données, l'interprétabilité locale cherche à comprendre pourquoi un modèle a fait une prédiction particulière pour un seul échantillon.

Par exemple, la méthode *SHAP* (*SHapley Additive exPlanations*) sera utilisée pour expliquer la probabilité donnée par le modèle. Cette méthode attribue une valeur a chaque fonctionnalité. Si cette valeur est positive, alors elle a tendance à augmenter la probabilité finale, et inversement.

7. Limites et améliorations possibles

7.1. Utilisation d'un kernel Kaggle pour la création du dataset final

Pour accélérer ce processus, l'utilisation d'un kernel Kaggle fut choisie, disponible à l'adresse suivante : (https://www.kaggle.com/code/jsaguiar/lightgbm-with-simple-features/script)

Ce choix est intéressant car ce kernel en particulier donne des scores tout à fait satisfaisants (Public Score = 0.79070). Cependant, il serait toujours judicieux de vouloir améliorer ce kernel en ajoutant nos propres approches concernant l'analyse exploratoire, la préparation des données et le Feature engineering.

7.2. Le manque d'expertise

Bien que l'expertise en science des données soit axée sur la manipulation, l'analyse et la modélisation des données, les critères d'octroi de crédit présentaient des spécificités complexes liées au domaine de la banque. La compréhension de chaque Feature était un défi de taille, par conséquent, la collaboration avec un expert du domaine bancaire aurait apporté une valeur ajoutée significative. Cet expert aurait pu aider à clarifier la signification et la pertinence de chaque variable et à s'assurer de l'adéquation des méthodes de traitement des données.

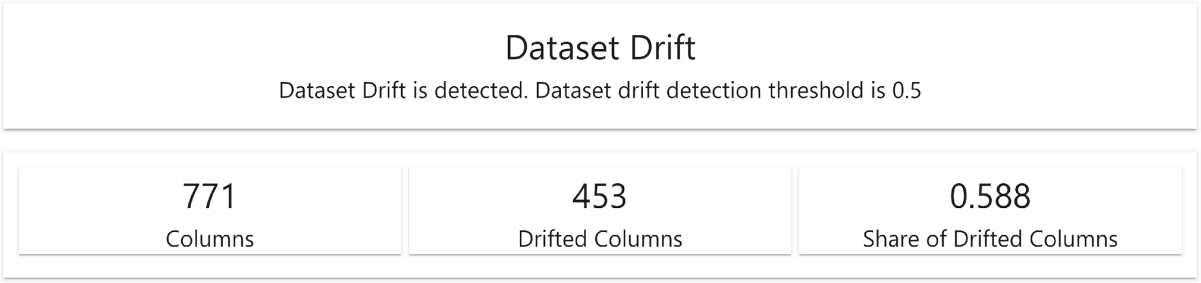
8. Analyse du Data Drift

Le Data Drift se réfère à une modification des distributions de données au fil du temps par rapport aux données initiales sur lesquelles le modèle a été formé. Cela peut avoir un impact sur les performances du modèle, car si les données d'entrée évoluent de manière significative par rapport aux données d'entraînement, les prédictions du modèle peuvent ne plus être précises ou pertinentes. L'hypothèse qui fut choisie est que le dataset application_train représente les données pour la modélisation et le dataset application_test représente les nouveaux clients une fois le modèle en production.

Pour les colonnes numériques, le test statistique de Kolmogorov-Smirnov fut utilisé, et pour les colonnes catégorielles, l'indicateur PSI (Population Stability Index) fut choisi. Un seuil de 0,2 fut défini pour chaque test statistique :

- Si la p-value du test K-S est inférieure à 0,2, cela suggère que les deux échantillons ne proviennent pas de la même distribution à un niveau de confiance de 80%.
- Si le PSI est supérieur à 0,2, cela suggère aussi une indication de Data Drift.

Le package *evidently* permet d'effectuer cette analyse de Data Drift et de créer une page html permettant l'analyse de ces résultats. Il est considéré que si plus de la moitié des fonctionnalités possèdent du Data Drift, alors le Data Drift sur l'ensemble des données est détecté.



Il est noté que 58.8% des fonctionnalités possèdent du Data Drift. En conclusion, le Data Drift a eu lieu entre application_train et application_test.