Présentation des modèles

DummyClassifier

Classificateur 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.

5 hyperparamètres furent considérés :

1) most_frequent : Prédit la target la plus fréquente dans les données d'entraînement.
2) prior : Fait des prédictions en respectant la distribution de notre target
3) uniform : Chaque classe (0 ou 1) a une probabilité égale d'être prédite
4) stratified : Prédit un échantillon de manière probabiliste (probabilité des classes)
5) constant: La classe à prédire est fournie par l'utilisateur.

Logistic Regression (Régression logistique)

Classificateur 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 classée dans la classe positive, sinon elle est classée dans la classe négative.

1) Cette probabilité possède la formule suivante

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

Figure 1: Probabilité de la Régression logistique

Avec x = Valeurs des features de l'échantillon

Avec β = Coefficients à déterminer

2) Ces coefficients β s'estiment en maximisant la vraisemblance
3) L'hyperparamètre considéré est La régularisation L2 (Ridge)

LightGBM (Light Gradient Boosting Machine)

Classificateur 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. Cette technique lui permet d'être plus rapide et de consommer moins de mémoire comparé à d'autres modèles utilisant des arbres de décision tel que XGBoost par exemple.

3 hyperparamètres furent considérés :

1) **learning_rate :** Contrôle la contribution de chaque arbre lors de l'ajout au modèle final.
2) **num_leaves :** Nombre maximum de feuilles
3) **n_estimators :** Nombre d'arbres à entrainer

Méthodologie d'entraînement du modèle

1. Imputation des données d'entrainement (ou non)

Pour le DummyClassifier et la Logistic Regression, nous sommes obligés d'imputer nos données pour que la simulation puisse s'effectuer. L'imputation choisie est la stratégie « mean » (moyenne).

Pour LightGBM, nous pouvons nous passer de cette imputation et lui donner les données bruts (possédant des NaN) ce qui a pour avantage d'entrainer notre modèle sur des données qui ne sont pas dénaturés.

2. Standardisation des données d'entrainement (StandardScaler)

Le StandardScaler transforme les caractéristiques de tel sorte à ce que leur moyenne soit nulle et que leur écart type soit égal à 1.

Standardiser les données avant d'entrainer nos modèles donne plusieurs avantages :

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

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'on effectue une validation croisée, chaque fold soit représentatif du dataset complet. C'est pour cela que nous décidons de réarranger les données de manière à ce chaque fold possède le même pourcentage d'échantillons de chaque classe cible que l'ensemble de données. Cette technique s'appelle la stratification et c'est pour cela que nous utilisons la méthode « StratifiedKFold » de scikit-learn.

4. Intégration de l'hyperparamètre seuil (threshold)

Dans notre cas, les modèles nous donnent en sortie une probabilité nous permettant de conclure si oui ou non nous devons accorder un crédit à un client. En général, la valeur par défaut du seuil est de 0,5. Cependant, il peut arriver que cette valeur de 0,5 peut ne pas être optimal. En décidant d'ajuster ce seuil, nous pouvons contrôler le compromis entre les faux positifs et les faux négatifs.

5. Enregistrement des résultats avec MLflow

MLflow est une plateforme 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.

Le traitement du déséquilibre des classes

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

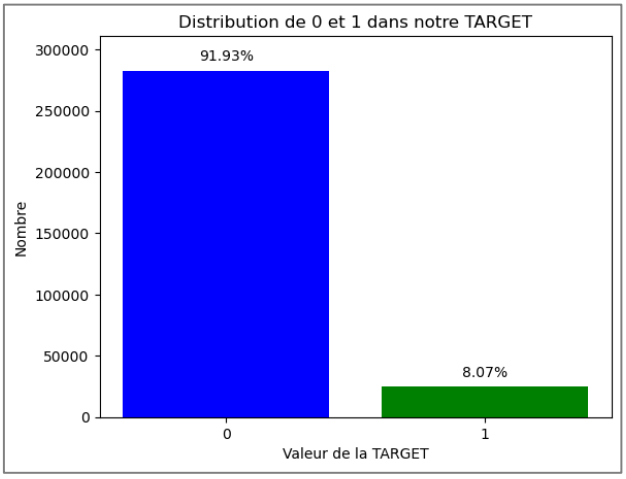


Figure 2: Déséquilibre de la distribution TARGET

Un tel déséquilibre soulève de nouvelles problématiques :

1) **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.
2) **Biais du modèle :** Car il est "récompensé" davantage pour avoir correctement prédit cette classe étant donné sa prédominance.

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

```
nb_0 = (df_classification_imputed['TARGET'] == 0).sum()
nb_1 = (df_classification_imputed['TARGET'] == 1).sum()
class_weights = {0: 1, 1: nb_0 / nb_1}
```

Figure 3: Code python pour la méthode "class_weights"

Ce paramètre class_weights permet de donner un poids différent à différentes classes. En faisant cela, vous pouvons 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 égale 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.

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

La 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) :** Il s'agit de 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) :** Ce sont les situations 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. À titre illustratif, nous pouvons supposer 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 comme suit :

Coût métier = Somme(10*FN+FP)

L'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 (Tree-structured Parzen Estimator).

Le TPE 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, le TPE apprend et adapte ses suggestions, se concentrant davantage sur les régions de l'espace d'hyperparamètres où les performances seraient probablement meilleures. Le package Optuna fut utilisée pour utiliser cette méthode d'optimisation.

La métrique d'évaluation

Comme précisé dans la partie 2 (L'algorithme d'optimisation), l'arbre de Parzen souhaite optimiser une fonction « objective » pour qu'il puisse trouver le meilleur jeux d'hyperparamètres. Ici, cette fonction objective sera le coût métier.

Un tableau de synthèse des résultats

Pour LightGBM et la Régression Logistique, nous avons accordé 50 essais à Optuna, lui permettant ainsi d'identifier de manière optimale les bons jeux d'hyperparamètres.

	Strategy	C	Learning Rate	Num Leaves	Threshold	AUC	Accuracy	Business Score
LightGBM	nan	nan	0.017	55	0.48	0.787	0.736	149513
LogisticRegression	nan	71.438	nan	nan	0.52	0.772	0.729	156247
DummyClassifier	most_frequent	nan	nan	nan	nan	0.5	0.919	248250

Figure 4: Résultats des 3 modèles

Nous souhaitons en priorité optimiser 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 trompeur surtout si les classes sont fortement déséquilibrés (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.
-

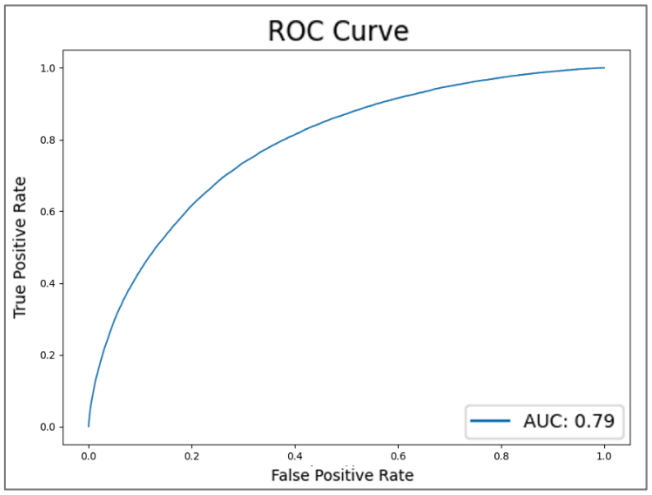


Figure 5: Courbe ROC du modèle LightGBM

La figure 4 nous fait remarquer qu'en effet, utiliser un seuil de 0.5 n'est pas forcément idéal car pour la Régression Logistique, utiliser un seuil de 0.52 optimise le Business score, le raisonnement est le même pour le LightGBM (0.48)

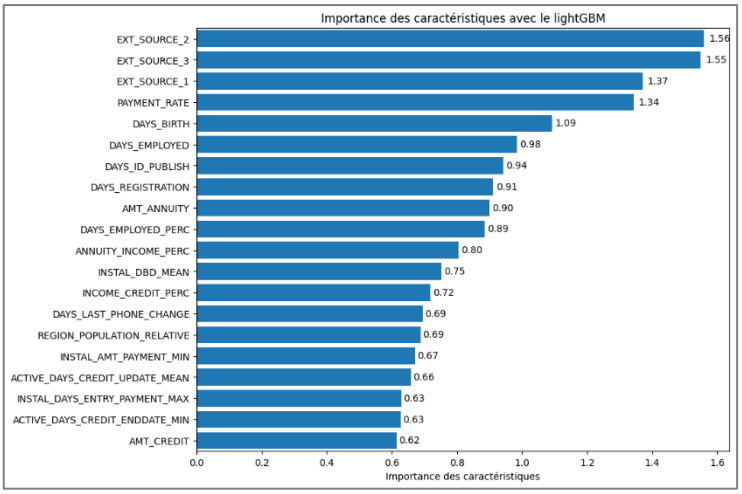
Au final, compte tenu de tout ces résultats, nous pouvons en tirer la conclusion que le modèle à utiliser en production est le suivant :

LightGBM (Num Leaves = 55 | Learning rate = 0.017 | Threshold = 0.48)

L’interprétabilité globale et locale du modèle

Interprétabilité globale

L'interprétabilité globale d'un modèle se réfère à notre capacité à comprendre comment le modèle prend ses décisions pour l'ensemble du jeu de données. Pour répondre à ces questions nos utilisons ce que nous appelons l’importance des features. 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, nous remarquons qu’aucune Feature est anormalement élevé relativement aux autres. Cela nous permet de conclure que nous n’avons pas de Data Leakage

Figure 6: Feature importances du LightGBM

Interprétabilité locale

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.

Nous pouvons utiliser par exemple la méthode SHAP (SHapley Additive exPlanations) 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 sinon.

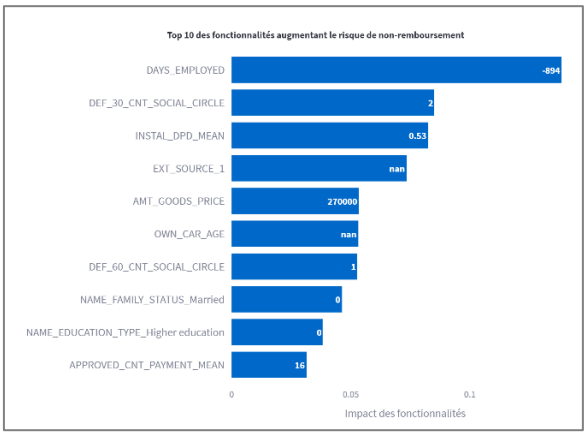


Figure 7: Exemple d'utilisation de SHAP

Les limites et les améliorations possibles

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), mais 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.

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 présentait des spécificités complexes lié 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.

L’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.

Nous allons prendre comme hypothèse 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, nous allons utiliser le test statistique de Kolmogorov-Smirnov, et pour les colonnes catégorielles, nous utilisons l’indicateur PSI (Population Stabilité Index). Un seuil de 0.2 fut défini pour chaque test statistique :

- Si le p-value du test K-S est inférieur 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 nous permet d’effectuer cette analyse de Data Drift et de créer une page html nous permettant l’analyse de ces résultats. Il est considéré que si plus de la moitié des fonctionnalités possèdent du data drift, alors nous avons bien un data drift sur l’ensemble des données.

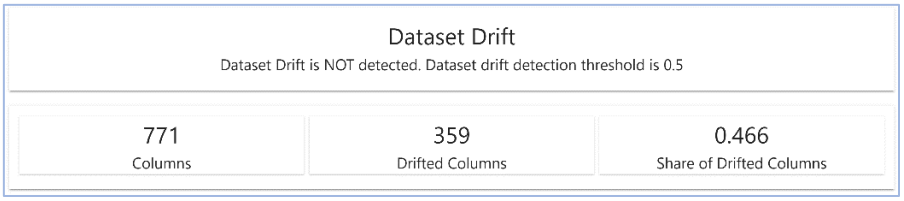


Figure 8: Résultats de l'analyse du Data Drift

Nous remarquons que 46.6% des fonctionnalités possèdent du Data Drift, nous pouvons en conclure que le data drift n’a pas encore lieu entre application_train et application_test.

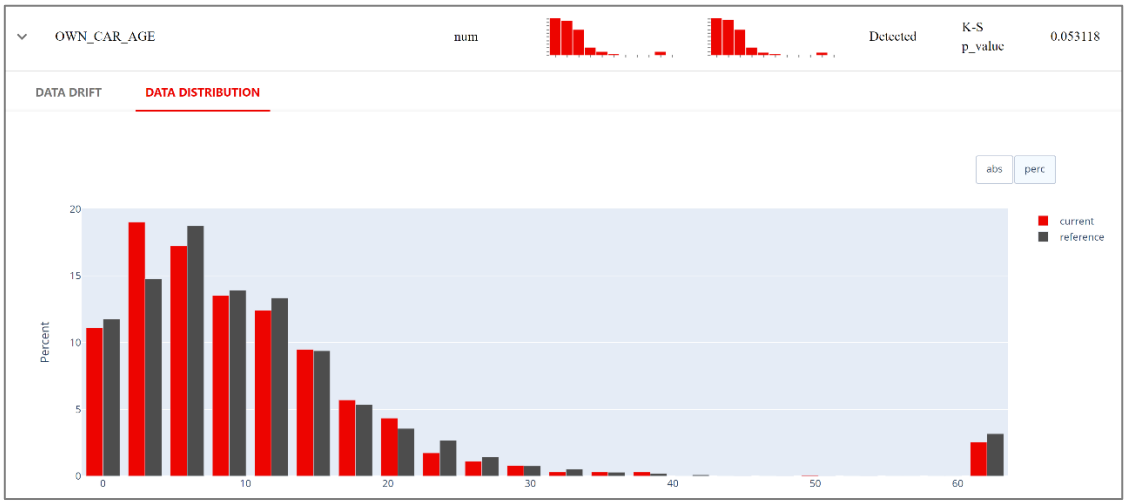


Figure 9: Exemple de data drift pour la feature OWN_CAR_AGE