Notes méthodologiques

# Méthodologie d'entraînement du modèle

Le but est de construire un modèle qui peut prédire si un client sera en mesure de rembourser un crédit ou non. Le développement de l'algorithme de classification est basé sur une approche de machine learning supervisée de type XGBoost avec la librairie *lightgbm*. Ce choix a été motivé par sa rapidité et sa capacité à gérer de grands ensembles de données et pour ses fortes performances. Le protocole est le suivant :

* Les données ont été divisées en un ensemble d'entraînement et un ensemble de validation afin de pouvoir mesurer les performances du modèle en dehors des données d'entraînement.
* L’optimisation des hyperparamètres à été faite par une approche bayésienne couplée à une validation croisée en *k-fold* pour évaluer les performances des modèles
* Les paramètres utilisés dans la recherche comprennent
  + la profondeur maximale de l'arbre,
  + le taux d'apprentissage et le nombre d'estimateurs.
  + Le nombre d’arbres
* Les probabilités du modèle ont été calibrées

# Traitement du déséquilibre des classes

Le jeu de données initial présente un déséquilibre entre les classes, avec beaucoup plus de bons clients que de mauvais clients. Pour remédier à cela, j’ai choisi d'utiliser les arguments ‘class\_weight’ de XGBoost pour traiter le déséquilibre de classes. Cette méthode permet de donner un poids différent à chaque classe, en accordant plus d'importance à la classe minoritaire, de sorte que l'algorithme prenne en compte ce déséquilibre dans le processus d'apprentissage.

Par ailleurs, j’ai utilisé ‘sample\_weight’ pour la calibration des probabilités, ce qui permet d'attribuer un poids différent à chaque observation en fonction de la classe à laquelle elle appartient. Cela permet de corriger les biais dans les probabilités de prédiction du modèle, qui peuvent être influencées par la distribution déséquilibrée des classes dans les données d'entraînement.

Cette approche de « poids » des échantillons a l’avantage de ne pas réduire la taille d’échantillon, contrairement à une approche de sous-échantillonnage aléatoire des données.

# Fonction coût métier, algorithme d'optimisation et métrique d'évaluation

## Score personnalisé

Le score d'évaluation utilisé pour évaluer les performances du modèle lors des *k-fold* est la cross-entropie. Il s’agit d’une fonction de perte couramment utilisée pour évaluer la qualité des prédictions de classification. Elle mesure la différence entre la probabilité prédite pour une observation et sa valeur réelle.

J’ai choisi de personnaliser la cross-entropie en pénalisant les faux positifs plus fortement que les faux négatifs, en utilisant un coefficient beta pour tenir compte des coûts des faux positifs et des faux négatifs. La cross-entropie personnalisée peut être définie comme suit :

où :

* est la valeur réelle de la classe pour l'observation (0 ou 1).
* est la probabilité prédite pour l'observation.
* est le coefficient de pénalité pour les faux positifs, qui est fixé à 0.4 dans ce cas.

Cette fonction de perte personnalisée permet de tenir compte du déséquilibre de classes dans les données et de donner plus de poids aux faux positifs, qui ont un impact plus important sur le coût métier de la classification. En outre, en fixant $\beta$ à une valeur plus élevée, on peut augmenter la pénalité pour les faux positifs, ce qui peut améliorer les performances du modèle en termes de classification des bonnes classes. L'AUC-ROC à tout de même été conservé en parallèle afin de s’assurer de la fiabilité de cette nouvelle métrique. Les coûts des faux négatifs sont supposés être 10 fois plus élevés que ceux des faux positifs (i.e. beta = 0.4).

4. Fonction de perte personnalisée

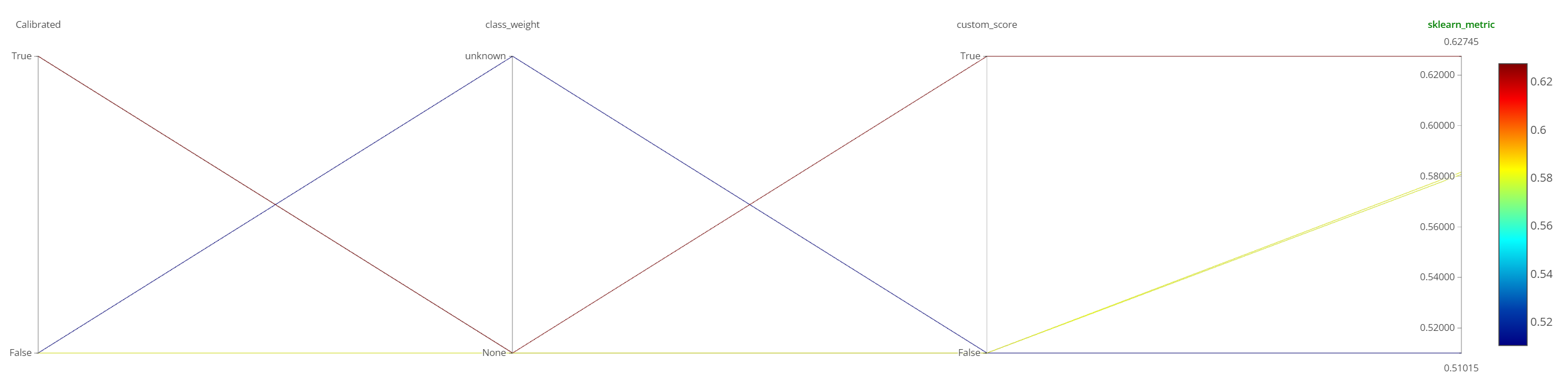
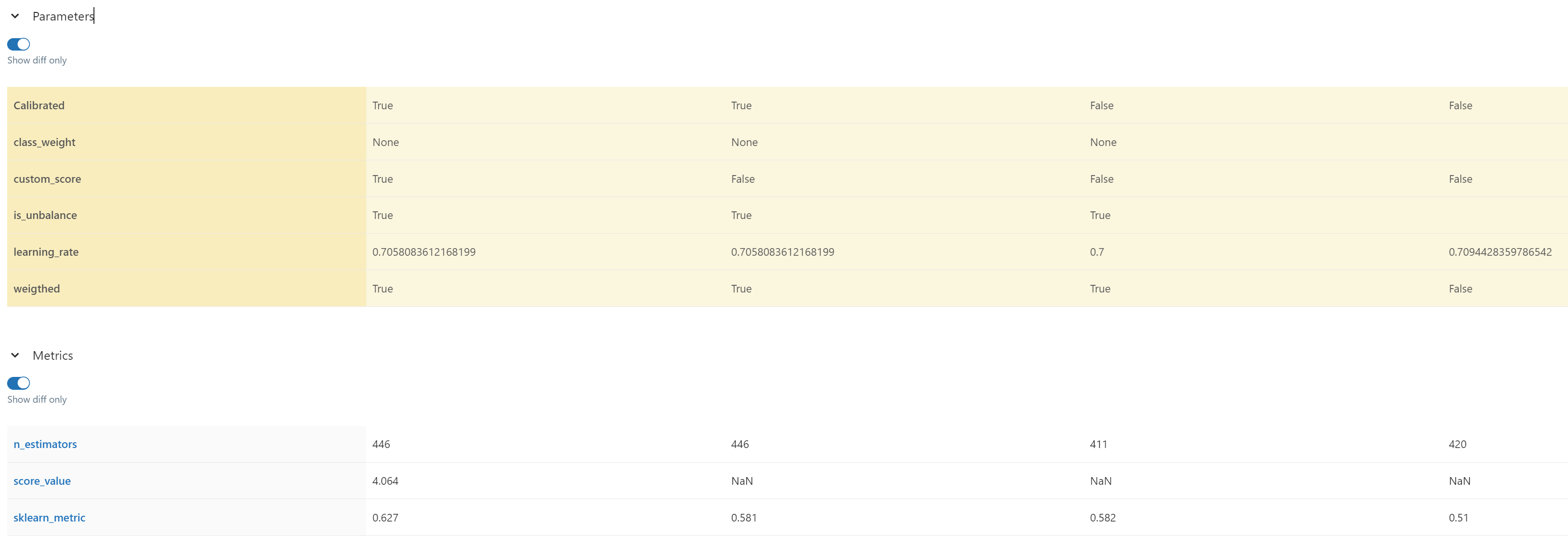
Un score permet de sélectionner, parmi un ensemble de modèles dont les hyperparamètres varient, celui qui fait le moins d’erreurs. Cependant il est possible d’aller plus loin dans l’optimisation en ajustant la descente de gradient stochastique. Cette méthode consiste à ajuster les paramètres du modèle (les poids des arbres de décision) en suivant le gradient de la fonction de perte par rapport à ces paramètres. Le gradient de la fonction de perte est calculé en utilisant les dérivées partielles de la fonction de perte par rapport à chaque paramètre du modèle. Ce gradient est utilisé pour mettre à jour les poids du modèle à chaque itération. Pour calculer le gradient de la fonction de perte, XGBoost utilise la technique de différenciation automatique pour calculer la dérivée partielle de la fonction de perte par rapport à chaque paramètre. En plus du gradient, XGBoost utilise également une approximation de la matrice Hessienne de la fonction de perte. Cette matrice mesure la courbure de la fonction de perte par rapport à chaque paire de paramètres. L'approximation de la matrice Hessienne est obtenue en calculant la dérivée seconde de la fonction de perte par rapport à chaque paire de paramètres. L'utilisation de la matrice Hessienne permet d'améliorer la convergence de l'algorithme de descente de gradient stochastique en ajustant la taille des pas d'apprentissage en fonction de la courbure de la fonction de perte. Cela permet d'éviter les oscillations et les problèmes de convergence associés à un pas d'apprentissage trop grand ou trop petit. Les dérivés du score personnalisé d’entropie et sa matrice Hessienne sont détaillés en Annexe 1.

# Tableau de synthèse des résultats

Les différents modèles testés comprennent les paramétrisation suivantes :

* Pas de calibration de déséquilibre de classes, pas de calibration des probabilités, pas de score personnalisé, pas de fonction de perte personnalisée
* Calibration de déséquilibre de classes, pas de calibration des probabilités, pas de score personnalisé, pas de fonction de perte personnalisée
* Calibration de déséquilibre de classes, calibration des probabilités, pas de score personnalisé, pas de fonction de perte personnalisée
* Calibration de déséquilibre de classes, calibration des probabilités, score personnalisé, pas de fonction de perte personnalisée
* Calibration de déséquilibre de classes, calibration des probabilités, score personnalisé, fonction de perte personnalisée

*\*sklearn\_metric est l’auroc et \*scroe\_value est le score de la cross-entropie personnalisé*



Le MLOPS de cette approche a été effectué avec le package *mlflow* pour assurer un suivi et un archivage complet:

L’API permettant de récupérer les informations des clients est à l’adresse suivante : <https://mlflowbank.drsosa.repl.co/get/10> où 10 est l’id du client à qui calculer la probabilité et avec le code source à l’adresse suivante : <https://replit.com/@DrSosa/MLFLOWbank>

Le Dashboard permettant d’utiliser le meilleur modèle sélectionné et son résultat est accessible à l’adresse suivante : <https://sebastiansosa-streamlit-app-nm6256.streamlit.app/> avec le code source à l’adresse suivante : <https://github.com/SebastianSosa/StreamLit>

# Annexe 1

Si on considère le score d’entropie croisé personnalisé :

Où est le vrai label, est une constante, p est la probabilité de obtenue par :

En dérivant le premier terme nous avons :

En dérivant le deuxième terme on obtient :

Puisque

Combiner les trois fonctions et simplifier :

Hessian :