Note méthodologique

Table des matières

Méthodologie d'entraînement du modèle

Etape 1 : Préparation des données

La première étape consista en la récupération des données, récupérables à l'adresse suivante :

https://www.kaggle.com/competitions/home-credit-default-risk/data.

Le code servant à l'obtention d'un DataFrame « fusionné » est inspiré du code trouvable à l'URL suivante :

https://www.kaggle.com/code/jsaguiar/lightgbm-with-simple-features.

Une fois remanié et adapté à nos besoins, celui-ci se décompose de la façon suivante :

- Fusion des fichiers principaux
- Création de variables catégoriques (distinction sur le type d'observations)
- Correction d'erreurs (application de NaNs sur la variables DAYS_EMPLOYED)

Cela permet alors d'obtenir un DataFrame assez massif au niveau du nombre de colonnes (+ de 500). Une séparation initiale sera réalisée en fonction de la présence ou d'une cible identifiée par la colonne 'TARGET'. Deux jeux de données seront donc constitués : un possédant une cible (et servant directement dans la suite de l'entraînement), et l'autre, constituant de par l'absence de cible, un jeu de données 'réel' idéal pour des tests futurs. En parallèle de cela, un Feature Engineering peut être établi via une réalisation de la Feature Importance de nos colonnes par rapport à la cible. Une nouvelle fois, plusieurs code peuvent permettre de dégager une quantité réduite de variables significatives. On peut alors passer à une cinquantaine de variables, lesquelles sont encore raffinées par des tests plus poussés avec les divers modèles testés et proposés dans la suite de cette note. Cela permet alors de réduire le nombre de colonnes à moins de 20. Enfin, pour conclure cette étape, une création de nouvelles variables est menée, conduisant à 4 nouvelles variables issues de 3 variables initiales, ces dernières étant ensuite retirées du DataFrame.

Etape 2 : Création des jeux de données

Une scission en plusieurs jeux de données est réalisée par l'intermédiaire de la méthode StratifiedShuffleSplit de Scikit-Learn. Celle-ci permet de diviser un dataframe en jeux d'entraînement / test. Les folds de validations sont aléatoires et stratifiés, permettant de préserver le pourcentage d'échantillons pour chaque classe prédite. On se retrouve alors avec les jeux suivants :

- train_df : le jeu d'entraînement principal
- test_df : le jeu servant à tester en cas « réel » les modèles entraînés
- score_df : jeu pouvant servir à diverses applications pour du scoring

Le ratio retenu est 0.7 / 0.15 / 0.15 par rapport au jeu de données issu de l'étape 1.

Etape 3 : Recherche des meilleurs hyperparamètres par modèle

La recherche est confiée à un Pipeline basé autour des packages Scikit-Learn et Imbalanced-Learn. De par le caractère fortement déséquilibré de la cible (ratio estimé à 0.9/0.1 de cibles négatives/positives), un oversampling a été réalisé (voir partie dédiée pour de plus amples détails) autour de la méthode SMOTE. Celui-ci a en réalité été inclus dans un Pipeline intégré à une Random Search CV, laquelle a permis d'obtenir un compromis intéressant sur les performances (voir partie dédiée pour plus de détails).

Etape 4 : Génération de statistiques, puis comparaison entre modèle et sélection du meilleur modèle

Plusieurs statistiques (détaillées en partie « **Fonction coût métier, algorithme d'optimisation et métrique d'évaluation** ») ont servi à la décision du modèle le plus performant, lequel a résulté dans la sélection du XGBoost comme meilleur modèle.

Etape 5 : Fine Tuning du modèle retenu par recherche Bayesienne sur les hyperparamètres retenus

Une recherche Bayesienne basée sur le package Scikit-Optimize a été conduite sur la base des meilleurs hyperparamètres trouvés en étape 3. Celle-ci a été conduite dans un Pipeline similaire à l'étape 3.

Etape 6 : Création des SHAP Values et sérialisation des modèles

L'interprétabilité locale et globale du modèle retenu sera effectué au travers de SHAP (Shapley Additive exPlanations). Ceci nécessite la création d'un Explainer, objet permettant de créer les SHAP Values, détaillant pour chaque ligne l'importance des variables dans la prise de décision finale du modèle. Plusieurs types de graphes existent, mais il sera retenu deux graphes : le beeswarm, proposant une interprétation locale mais également globale ; et le bar plot, donnant une interprétabilité globale du modèle.

Traitement du déséquilibre des classes

Le traitement du déséquilibre des classes a été confié à la méthode SMOTE (Synthetic Minority Oversampling Technique) issu du package Imbalanced-Learn. Celui-ci génère des données synthétiques issues de la distribution de la classe minoritaire. Cela permet donc d'obtenir un ratio 50/50 entre nos classes à prédire. La limite directe de la méthode est donc l'entraînement sur des données potentiellement fausses, car issues d'une interprétation des diverses distributions des variables de notre classe minoritaire. D'autres tests ont été menés sur un tandem Oversampling / Undersampling, réalisé via la méthode SMOTENN (Synthetic Minority Oversampling Technique Edited Nearest Neighbours) d'Imbalanced-Learn. Les résultats étant plutôt mitigés par rapport à l'oversampling direct, il a pour l'instant été retenu l'emploi de SMOTE.

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

La fonction coût métier a comme but principal de limiter au maximum un des impacts financiers évaluables par la mesure du ratio Faux Positifs / Faux Négatifs. Les Faux Négatifs (mauvais clients, comptabilisés bons par le modèle) ont en effet un impact négatif bien plus important sur les revenus de l'entreprise que les Faux Positifs (bons clients prédits mauvais par le modèle). Un scoring a ainsi été développé, utilisé dans la recherche des meilleurs hyperparamètres. Celui-ci établit un score final, basé sur deux entrées : une liste comprenant les labels de la cible réelle, et une autre liste comprenant la probabilité négative / positive pour chaque client prédit par le modèle. Un mismatch sur une position donné déclenchera l'incrémentation du score. Si la fonction établit que le client prédit est un Faux Négatif, le score sera incrémenté 10x plus que pour un client prédit Faux Positif. Ce score est donc à optimiser afin d'avoir le plus faible possible. Il est ainsi créé un scorer via la méthode idoine de Scikit-Learn, incluant cette condition.

L'algorithme d'optimisation des hyperparamètres retenu est le RandomSearchCV. Celle-ci sera exploitée avec la méthode de prédiction predict_proba de chaque modèle testé afin d'inclure la possibilité de rechercher un seuil de décision idéal pour chaque ensemble d'hyperparamètres testé. L'avantage de la RandomSearchCV est de ne pas tester l'intégralité des valeurs d'hyperparamètres présentes dans l'espace de recherche, mais de sélectionner aléatoirement un ensemble parmi ceux disponibles. Le nombre d'hyperparamètres testés peut également être fixé via l'argument n_iter. La RandomSearchCV est donc computationnnellement parlant plus économe que la GridSearchCV.

Les métriques d'évaluation principales sont :

- ROC AUC : mesure de l'aire sous la courbe ROC (Receiver Operating Characteristic) pour obtention d'un indice de la performance du modèle sur la base du taux de vrais positifs en fonction du taux de faux positifs.

- Fbeta Score : mesure de la moyenne harmonique de la précision et du recall. Bêta est ajusté à 2 afin de donner plus d'importance au recall face à la précision, et donc de correspondre à la problématique métier de diminuer la quantité de faux négatifs.

- Score Métier : mesure de l'influence des faux positifs et faux négatifs. Plus la valeur du score métier est importante, plus celle-ci reflète une tendance du modèle à donner des faux positifs et surtout des faux négatifs.

- Temps d'entraînement : mesure du temps nécessaire à l'entraînement du modèle sur le jeu d'entraînement, actuellement composé de 221404 lignes pour 20 variables.

Tableau de synthèse des résultats

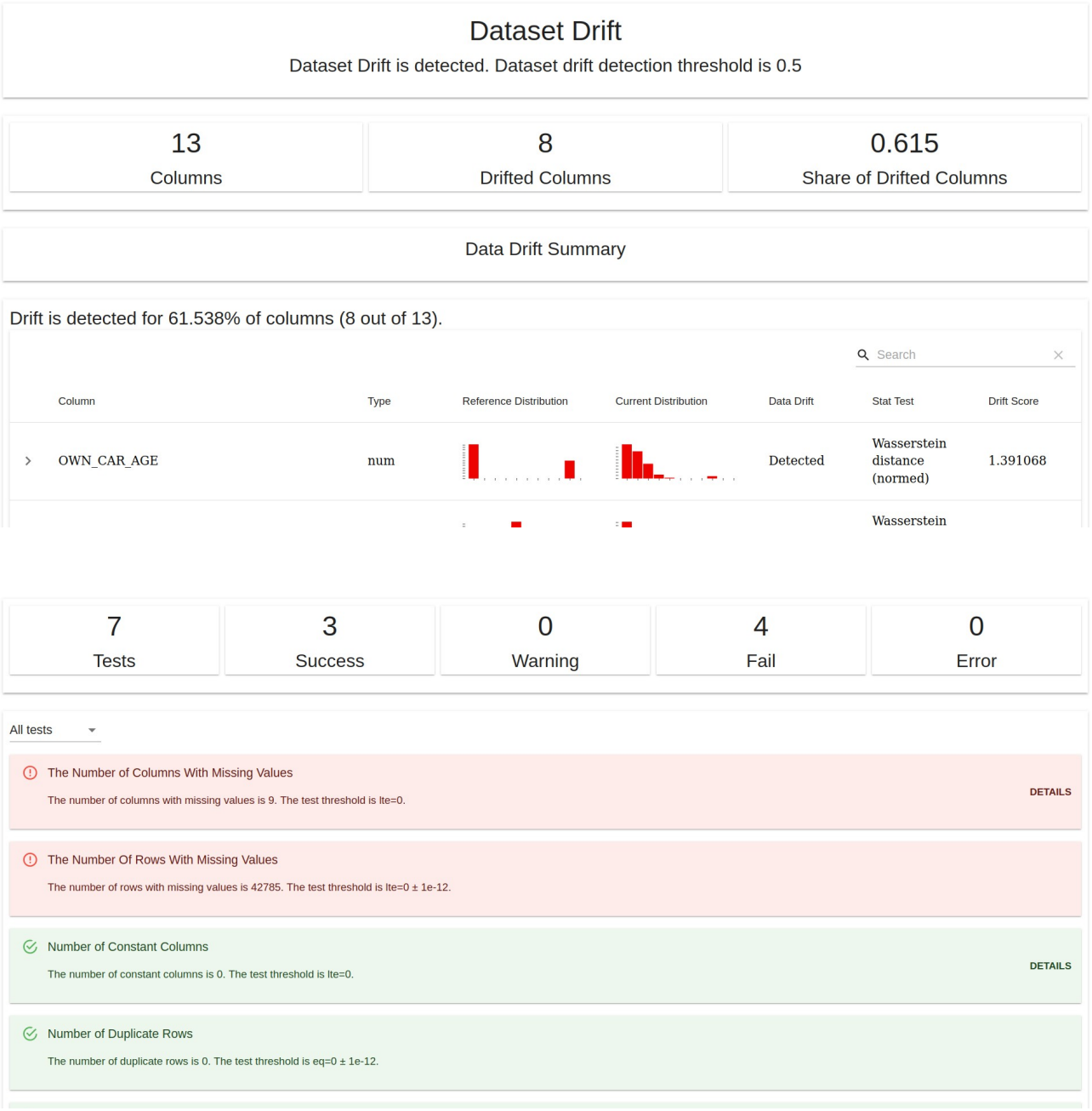
	ROC AUC	Fbeta Score	Score Métier	Accuracy	Temps d'entraînement
Dummy (Baseline)	0,5	0,31	508840	0,08	2 s (CPU)
Logistic Regression	0,52	0,31	457881	0,17	6,3 s (CPU)
XGBoost	0,65	0,38	183409	0,64	1,4 s (GPU)
LightGBM	0,68	0,42	149111	0,71	6,2 s (CPU)
Fine-Tuned XGBoost	0,68	0,42	150017	0,7	1,5 s (GPU)

Limites et améliorations possibles

Plusieurs limites sont actuellement présentes dans le travail réalisé. Ainsi, le modèle exploité est perfectible. L'optimisation des hyperparamètres peut encore être poussée sur le reste des hyperparamètres disponibles. De plus, le souci d'instabilité de LightGBM peut certainement être résolu, conférant ainsi un modèle très légèrement meilleur que le modèle actuel, à savoir XGBoost. D'autres approches peuvent également permettre d'améliorer les performances. Un meilleur Feature Engineering pourrait permettre de trouver des corrélations plus subtiles. La méthode d'échantillonnage est actuellement perfectible : un mélange d'oversampling et d'undersampling pourrait permettre de meilleures performances (empiriquement constaté sur la régression logistique), mais cela nécessite de meilleurs ajustements de la méthode, notamment pour les modèles les plus complexes. Côté application, l'interactivité peut être davantage poussée sur certains graphes, actuellement réalisés sous Matplotlib, en les passant sous Plotly. Un refactoring global du code amènerait une meilleure simplicité et stabilité. Enfin, l'API sera à stabiliser.

Analyse du Data Drift

L'analyse est réalisée par l'intermédiaire du package Evidently. Celui génère plusieurs types de rapports : cela peut aller d'un rapport assez graphique à des tests statistiques plus poussés. Afin d'obtenir une bonne exhaustivité, le choix a été réalisé de tester le Data Drift sur tous les jeux de données face au jeu d'entraînement. Cela a résulté en la génération de 3 séries de 2 fichiers :



Exemple type de génération de rapports avec Evidently

Les rapport générés mettent aisément en évidence la présence / absence de Data Drift.