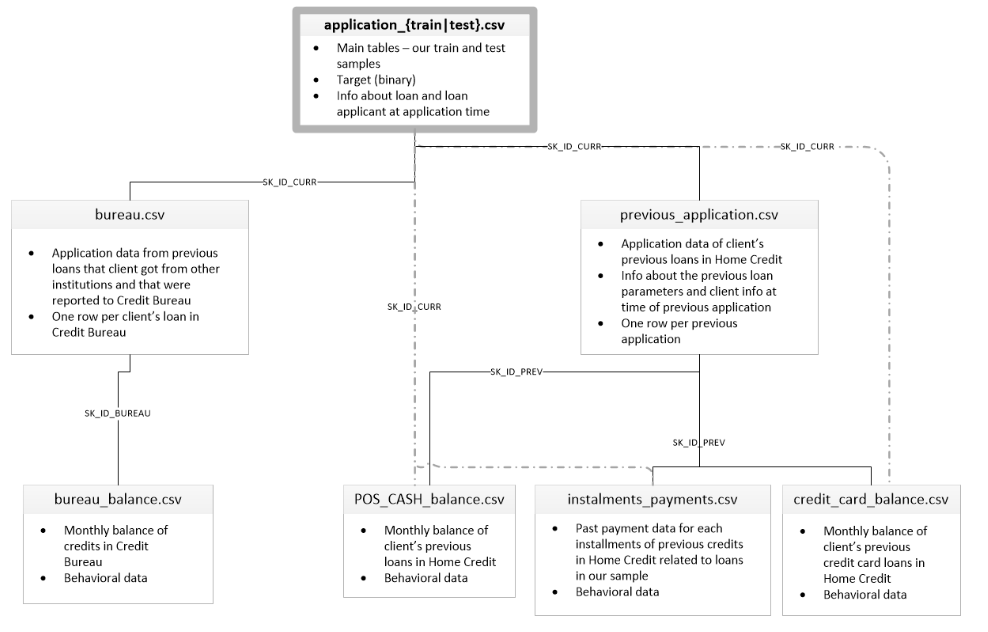
# Note Méthodologique – Prédiction de la défaillance de crédit.

## Introduction

Cette note méthodologique décrit les étapes clés suivies dans le cadre du projet de prédiction de solvabilité de clients. L'objectif de ce projet est de développer un modèle de machine learning capable de prédire la probabilité de défaillance de crédit pour des clients potentiels. D’autres étapes, non décrite ici, sont de de créer un tableau de bord interactif (frontend) alimenté par une API, le tout déployé en continu et disponible en ligne.

Pour ce faire, nous avons utilisé un ensemble de données provenant du projet Kaggle Home Credit Default Risk, qui contient 7 fichiers csv de données.

Le tableau de bord a été réalisé sur Streamlit. L’API, utilisant Flask, a été déployé sur Azure via le déploiement continu fourni par GitHub.



## Prétraitement des données

Le projet Kaggle étant une compétition datant de plusieurs années, des milliers de personnes ont eu l’occasion de travailler dessus. Par conséquent, il est possible de trouver en ligne des notebooks réalisant de bons prétraitements de nos données.

J’ai donc utilisé ce notebook <https://www.kaggle.com/code/jsaguiar/lightgbm-with-simple-features/script>, et je vais résumer ici les différentes tâches réalisées.

Voici le pipeline:

* Les fichiers csv, à savoir *application\_train.csv*, *application\_test.csv*, *bureau.csv, bureau\_balance.csv*, *previous\_application.csv*, *POS\_CASH\_balance.csv*, *installments\_payments.csv*, et *credit\_card\_balance.csv*, possèdent chacun une fonction pour les charger puis les prétraiter.
* Des données aberrantes sont identifiées et traitées, comme les valeurs 365243 dans 'DAYS\_EMPLOYED' remplacée par NaN, ou encore en supprimant les 4 clients possédant un genre inconnu (‘XNA’)
* Les caractéristiques catégorielles sont encodées en utilisant l'encodage one-hot pour les colonnes avec plus de deux catégories, et une simple factorisation (0 ou 1) pour les colonnes binaires.
* Plusieurs nouvelles caractéristiques sont créées, telles que des taux (par exemple, le taux d'annuité par rapport au revenu), des pourcentages, et des ratios.
* Les données sont agrégées à différents niveaux (par exemple, par client) en calculant des statistiques telles que la moyenne, la somme, le maximum, le minimum, la variance, etc.
* Les différentes données sont fusionnées dans un dataframe en utilisant les clés de jointure (identifiable dans le graphique plus haut), principalement la colonne 'SK\_ID\_CURR'.

## Data Drift

Voici un article expliquant en détail le phénomène de Data Drift : <https://datascientest.com/definition-data-drift>.  
En résumé, le data drift existe lorsqu’un modèle d’apprentissage automatique s’entraine sur des données qui sont ou ne seront plus d’actualités. Par exemple, si nous entrainons un modèle sur des données périodiques, mais que nous ne disposons pas de la période entière, alors du data drift va arriver. Autrement dit, notre modèle ne sait pas généraliser, et va avoir de mauvais résultats sur les périodes qui lui sont inconnues. Ce cas peut arriver si nous voulons prédire des ventes, sans prendre en compte les périodes de soldes.

Il est donc important de prendre en compte ce risque dans la mesure du possible. Dans notre cas, nous possédons un jeu de donner ‘train’, et un deuxième ‘test’. Nous allons nous servir de notre jeu de test, l’identifiant comme des données futures, afin d’évaluer un potentiel data drift sur notre jeu d’entrainement.

La librairie Evidently permet de faire cette analyse, en quelques lignes de codes, et de nous indiquer si nos données sont ‘drifted’, et plus précisément quelles valeurs le sont.

Voici la synthèse par fichier :

* Application\_train : 9 colonnes identifiées sur 121
* Previous\_application : 1 colonne identifiée sur 37
* POS\_CASH\_balance : aucun drift sur 8 colonnes
* Instalments\_payments : aucun drift sur 8 colonnes
* Bureau : 1 colonne identifiée sur 17
* Credit\_card\_balance : aucun drift sur 23 colonnes

Globalement, nos données sont utilisables, mais les colonnes concernées par le drift sont un axe d’amélioration.

## Fonction de coût métier

Dans le cadre de l'évaluation de notre modèle, nous utilisons plusieurs mesures couramment reconnues, telles que le score F1, l'aire sous la courbe ROC (AUC) et bien entendu, la matrice de confusion. Cependant, notre contexte professionnel impose la nécessité d'introduire une mesure spécifique, tenant compte de nos contraintes métier. La principale de ces contraintes est que les faux positifs, c'est-à-dire les clients que nous prédisons comme solvables mais qui ne le sont pas en réalité, engendrent des coûts plus élevés que les faux négatifs, c'est-à-dire les clients que nous pourrions considérer comme solvables mais que nous définissons comme non solvables. En effet, dans le premier cas, il s'agit d'une perte financière nette, tandis que dans le deuxième cas, il s'agit d’un manque à gagner.

Afin de prendre en compte cette contrainte particulière, nous avons décidé de modifier le score F1 en réduisant le poids des faux négatifs (bien que nous aurions également pu envisager d'augmenter le poids des faux positifs).

Cela donne :

Ce score métier nous servira dans l’entrainement de notre modèle et l’analyse de nos résultats.

## Modélisation

Pour la modélisation, la première étape est de choisir un modèle.

Dans le notebook Kaggle que j’ai repris, un modèle LGBM est utilisé. Cependant, ce dernier a été choisi et optimisé pour avoir la meilleure AUC. Or, ce que nous voulons c’est optimiser notre F10Score.

J’ai donc testé un Dummy Regressor, une régression logistique, ainsi que le LGBM, sans hyperparamètre, seulement avec les données du fichier ‘application\_train.csv’.

Les scores sont globalement mauvais (disponible en synthèse) mais le LGBM est très légèrement au-dessus.

J’ai ensuite ajouté les données des différents fichiers, un à un, en observant l’évolution du score.

A ce stade-là, nous n’avons rien d’exceptionnel, mais le score augmente à chaque ajout de fichier.

## Optimisations : hyperparamètres, seuil et déséquilibre des classes

Nous avons choisi notre modèle. A présent il faut optimiser les hyperparamètres. Pour cela, j’ai utilisé la librairie Optuna, qui permet de choisir pour chaque hyperparamètre une rangée de valeur puis cherche parmi ces valeurs en optimisant les recherches.

La librairie à aussi le gros avantage d’enregistrer les données à chaque essai, dans un fichier type sqlite. Ceci permet d’arrêter et de reprendre à tout moment notre optimisation, contrairement au GridSearch, qui est long, n’assure pas le résultat, et n’est pas sauvegardé automatiquement.

En complément, j’ai combiné la recherche d’hyperparamètres avec une validation croisée et après 250 essais, notre score s’améliore légèrement.

Les données possèdent aussi un déséquilibre entre les clients solvables et les clients non solvables : on a en très grande majorité des clients solvables. Il faut donc pouvoir gérer ce déséquilibre.   
Pour cela, j’ai choisi le paramètre ‘class\_weight’ avec la valeur ‘balanced’, implémenté par la librairie LGBM. Cet argument permet d’indiquer au modèle qu’il y a un déséquilibre, et l’algorithme se charge de mettre des poids à nos 2 classes. Ainsi, il pénalise plus les clients non solvables.

Notre score s’améliore grandement ! On identifie beaucoup mieux les clients non solvables. Par contre, beaucoup de faux positifs sont désormais générés.

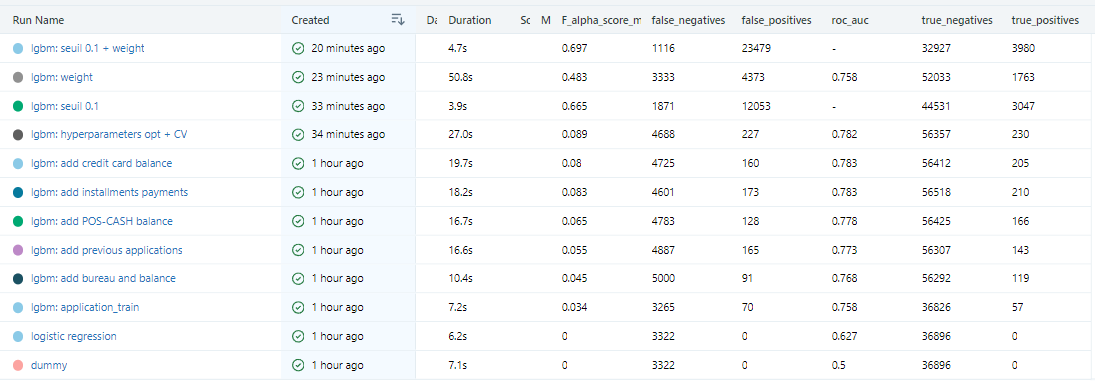
D’un autre côté, j’ai essayé d’explorer la piste du seuil de probabilité. Par défaut, LGBM à comme seuil 50% : si on est au-dessus, il prédit 1, sinon 0.

J’ai donc cherché à modifier ce seuil de 50%, en maximisant notre f10score. J’obtiens un score inférieur à notre précédente expérience, mais tout de même assez bon. Notre nouveau seuil est à 10% : si notre client à plus de 10% de risque d’être insolvable, alors il l’est.

Finalement, j’ai combiné ces 3 méthodes et j’obtiens mon dernier et meilleur score.

## Synthèses des résultats

Durant tout le projet, chaque essai était sauvegardé grâce à la librairie MLFLOW : à chaque lancement de notre modèle, ce dernier est enregistré et évalué avec de multiples mesures, incluant notre f10score. Voici le tableau récapitulatif :



Premier lancement :   
 Le Dummy Classifier et notre Régression Logistique ont un f10score de 0, ils ne détectent aucun client non solvable.  
 Le LGBM quant à lui, arrive à en détecter quelques-uns, mais à quand même un nombre de faux négatifs important

Lancement 2 à 7 : l’ajout des données  
 Le score augmente doucement. Le modèle identifie de plus en plus de clients non solvables.

Lancement 8 : seuil à 0.1  
 On identifie 60% des clients non solvables. Par contre, un grand nombre de faux positif son identifié

Lancement 9 : poids sur les classes  
 35% des défauts sont identifiés. Et on a relativement peu de faux positif.

Lancement 10 : combinaison de tous les essais   
 En combinant toutes les méthodes explorées précédemment, nous avons obtenu notre meilleur score.

## Limites et améliorations

Notre meilleur score métier, de 0.697 génère énormément de faux positif.

Pour améliorer le modèle, de nouvelles données serait évidemment appréciable, d’autant plus que nous avons identifié quelques data drifts.

Un axe d’amélioration est le pré-traitement : la gestion des valeurs nulles et l’ajout de nouvelle feature pourrait améliorer notre score.

D’autres modèles pourraient aussi mieux convenir.

Il faut aussi noter que nos données sont très probablement biaisées. En effet, nos données sont des clients pour lesquels les crédits ont déjà été accordés. Or, nous n’avons pas tous ceux qui ont préalablement été refusés par la banque.

De ce point de vue, on pourrait penser que notre LGBM le plus faible fait mieux que la réalité, si le préfiltre reste présent.