



PRET A DEPENSER

Note méthodologique



TABLE DES MATIÈRES

- Introduction3
- 1. Méthodologie d’entraînement du modèle 4
- 2. Traitement du déséquilibre des classes6
- 3. Fonction coût métier, algorithme d’optimisation et métriques d’évaluation7
- 4. Tableau de synthèse des résultats9
- 5. Interprétabilité globale et locale du modèle 10
- 6. Limites et améliorations possibles11
- 7. Analyse du Data Drift.....12

Introduction

La société Prêt à Dépenser propose des crédits à la consommation pour les personnes ayant **peu ou pas du tout d'historique de prêt**. Elle souhaite mettre en œuvre un **outil de scoring crédit** pour :

- Calculer la **probabilité** qu'un client **rembourse son crédit**
- Classifie la demande en **crédit accordé ou refusé**
- En faisant preuve de **transparence** vis-à-vis des décisions d'octroi de crédit

Les données à notre disposition proviennent de **sources variées**. Il s'agit de données comportementales, de données sur les demandes de prêts précédentes ou en cours chez Prêt à Dépenser ou de prêts accordés par d'autres institutions financières. Ces données concernent un peu plus de **307 000 clients**.

Cette note méthodologique a vocation à présenter la **démarche de modélisation** de manière synthétique. Elle comprend toute la démarche d'élaboration du modèle jusqu'à l'analyse du Data Drift.

1. Méthodologie d'entraînement du modèle

L'analyse exploratoire, la préparation des données et le feature engineering nécessaires à l'élaboration du modèle de scoring sont principalement issus des kernels suivants :

- <https://www.kaggle.com/code/jsaguiar/lightgbm-with-simple-features/script>
- <https://www.kaggle.com/code/willkoehrsen/start-here-a-gentle-introduction#Exploratory-Data-Analysis>

Ils ont été adaptés et enrichis notamment sur la partie de préparation des données avec le retraitement des valeurs manquantes ou encore le regroupement de modalités.

Trois méthodes de sélection de variables ont été utilisées : la suppression de variables sans variance, l'utilisation de méthodes statistiques (Chi2 pour les variables qualitatives et Anova pour les variables quantitatives) ainsi que la sélection des variables les plus importantes via Feature Importance.

1.1. Sélection des données d'entraînement et de test

En Machine Learning il ne faut jamais valider un modèle sur les données qui ont servi à son entraînement. Le modèle doit être testé sur des données qu'il n'a jamais vues. On aura ainsi une idée de sa performance future. Le dataset sera mélangé de façon aléatoire avant d'être divisé en deux parties:

- un **train set** dont les données sont utilisées pour entraîner le modèle (80% des données)
- un **test set** réservé uniquement à l'évaluation du modèle (20% des données)

La séparation du dataset en données d'entraînement et de test va permettre de **détecter de l'overfitting** (modèle trop complexe qui apprend parfaitement les données d'entraînement mais n'arrive pas à généraliser) ou de **l'underfitting** (modèle trop simple ou mal choisi).

1.2. Pipeline, optimisation et entraînement des modèles

Nous allons dans un premier temps créer une **pipeline** pour chacun de nos modèles. Cette pipeline va nous permettre d'affecter des étapes de preprocessing à nos données, c'est-à-dire des transformations comme le traitement des données déséquilibrées, la standardisation de nos données, et de choisir le type de modèle.

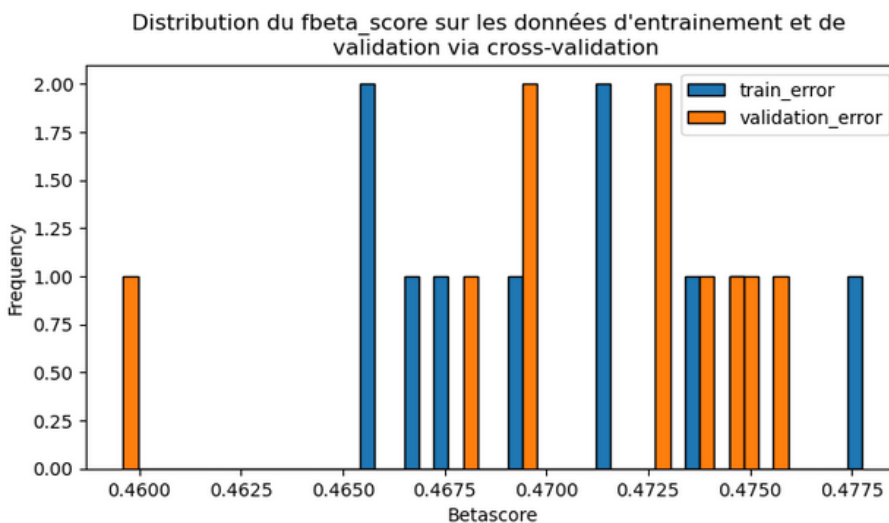
Cette pipeline sera ensuite intégrée dans une fonction d'optimisation et d'entraînement qui va utiliser la **validation croisée** pour tester la robustesse du modèle prédictif en répétant la procédure de split. Elle donnera plusieurs erreurs d'apprentissage et de test et donc une **estimation de la variabilité de la performance de généralisation du modèle**. Ici, nous avons utilisé la méthode **KFold** qui consiste à découper le train set en cinq parties, l'entraîner sur les quatre premières parties puis le valider sur la cinquième. On recommencera sur toutes les configurations possibles puis on fera la moyenne des cinq scores et on pourra donc comparer nos modèles pour être sûr de prendre celui qui a en moyenne la meilleure performance.

Le réglage des hyperparamètres s'effectuera soit à l'aide du **GridSearchCV** qui va tester toutes les combinaisons possibles d'hyperparamètres afin de trouver celles qui vont minimiser le plus l'erreur

(méthode exhaustive très **coûteuse en termes de puissance de calcul et de temps**), soit du **RandomizedSearchCV** qui va sélectionner des combinaisons aléatoires d'hyperparamètres. Cette méthode est un peu **moins précise** mais beaucoup plus **rapide**. Elle sera utilisée pour les modèles plus complexes.

Enfin, pour évaluer la **performance réelle de nos modèles**, nous calculerons les métriques sélectionnées sur les données de test.

La validation croisée peut être utilisée à la fois pour le **réglage des hyperparamètres** et pour l'estimation de la **performance de généralisation** d'un modèle. Cependant, l'utiliser à ces deux fins en même temps est problématique. Le réglage des hyperparamètres est une forme d'apprentissage automatique et, par conséquent, nous avons besoin d'une **autre boucle externe de validation croisée** pour évaluer correctement la performance de généralisation de la procédure de modélisation complète. Lorsque l'on optimise certaines parties de la pipeline d'apprentissage automatique (par exemple, les hyperparamètres, la transformation, etc.), il est nécessaire d'utiliser la **nested cross validation** pour évaluer la performance de généralisation du modèle prédictif. Sinon, les résultats obtenus sans nested cross validation sont souvent trop optimistes. C'est ce qui a été fait pour visualiser la distribution de la métrique sélectionnée sur les données d'entraînement et de validation :



Dans l'image de gauche, les performances étant aussi mauvaises sur le train que sur le test, nous faisons probablement face à de l'underfitting.

1.3. Modèles testés

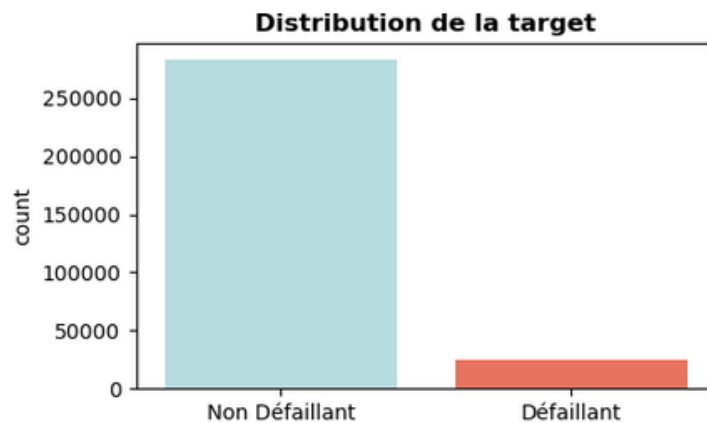
Nous cherchons à classer les demandes en **crédit accordé ou refusé**. Il s'agit donc d'un modèle de **classification binaire**. Plusieurs modèles allant du plus simple au plus complexe ont été optimisés :

- un modèle baseline, **le dummy classifieur** : non optimisé car il ignore les variables en entrée.
- **la régression logistique** : tentatives d'optimisations en jouant sur différents **preprocessings**, le **nombre de features**, le traitement du **déséquilibre des classes** et les **hyperparamètres** via **cross validation** et **GridSearch/RandomizedSearch**
- **le LightGBM** : tentatives d'optimisations en jouant sur différents **preprocessings**, le **nombre de features**, le traitement du **déséquilibre des classes** et les **hyperparamètres** via **cross validation** et **GridSearch/RandomizedSearch**, le choix du **scoring**, en passant par la librairie **FLAML**

La présentation des résultats des modèles ainsi que des meilleurs hyperparamètres sera faite dans la partie 4 : Tableau de synthèse des résultats.

2. Traitement du déséquilibre des classes

Lors de l'analyse exploratoire, nous avons remarqué que les données étaient très **déséquilibrées** entre les défaillants et non défaillants. Les non défaillants sont largement sur-représentés (> 91%).



La plupart des modèles de Machine Learning vont **ignorer la classe minoritaire** et donc avoir des **performances médiocres** dans cette classe alors qu'en général c'est la performance de la classe minoritaire qui est la plus importante. Nous avons testé deux méthodes :

- La **pondération de classes** : il s'agit d'ajuster la fonction de coût du modèle de manière à ce qu'une mauvaise classification d'une observation de la classe minoritaire soit plus lourdement pénalisée qu'une mauvaise classification d'une observation de la classe majoritaire. Cette approche contribue à améliorer la précision du modèle en rééquilibrant la distribution des classes. Elle est implémentée via le paramètre `class_weight` présent dans chaque algorithme de classification sklearn.
- L'utilisation de la technique **SMOTE** (Synthetic Minority Oversampling Technique): de nouveaux exemples sont synthétisés à partir des exemples existants. Il s'agit d'un type d'augmentation de données pour la classe minoritaire. Un **exemple aléatoire de la classe minoritaire** est choisi et les k plus proches voisins sont trouvés (avec $k = 5$ en général). Un voisin est choisi au hasard et un segment est tracé entre les 2 points. Il est recommandé d'utiliser d'abord un **sous-échantillonnage aléatoire** pour réduire le nombre d'exemples dans la classe minoritaire puis d'utiliser SMOTE pour sur-échantillonner la classe minoritaire afin d'équilibrer la distribution des classes. C'est une approche efficace car les nouveaux exemples synthétiques de la classe minoritaire sont plausibles (proches dans l'espace des caractéristiques des exemples existants de la classe minoritaire).

Différents pourcentages de sur-échantillonnage de la classe minoritaire et sous-échantillonnage de la classe majoritaire ont été testés sur la régression logistique ainsi que l'utilisation du paramètre **class_weight**. Ce dernier ayant donné un modèle bien plus performant, nous l'avons retenu avec la valeur « **balanced** » (c'est-à-dire que le modèle attribue automatiquement aux classes des poids inversement proportionnels à leurs fréquences respectives).

3. Fonction coût métier, algorithme d'optimisation et métriques d'évaluation

3.1. Fonction coût métier

Comme il serait extrêmement **coûteux** pour la banque d'accorder un crédit à un client défaillant qui ne le rembourserait pas ou en partie, il nous faut **minimiser le nombre de faux négatifs** c'est-à-dire un client prédit non défaillant alors qu'il est défaillant. Il faut également tâcher de **minimiser les faux positifs** c'est-à-dire prédire qu'un client est défaillant alors qu'il ne l'est pas (risque de perte de clients, de manque à gagner).

Un faux positif n'a pas le même coût qu'un faux négatif. Ce dernier est beaucoup plus coûteux pour la banque. **Nous avons supposé qu'il était 10 fois plus coûteux qu'un faux négatif.**

$$\text{Score métier à minimiser} = 10 * FN + 1 * FP$$

Le **seuil de probabilité** a été défini de manière à détecter efficacement les clients défaillants. Nous souhaitons donc maximiser le nombre de vrais positifs (client défaillant prédit défaillant) mais sans trop augmenter le nombre de faux positifs. Le seuil de **0.47** permet d'avoir le **score métier le plus faible**.

3.2. Algorithme d'optimisation

Parmi les différents modèles testés, le **LightGBM** a donné les meilleurs résultats. Lors de l'optimisation des hyperparamètres via cross validation, une approche par GridSearchCV / RandomizedSearchCV a été tentée sur les hyperparamètres suivants:

- **n_estimators** = nombre d'arbres de décision (il est préférable d'en avoir plusieurs)
- **max_depth** = profondeur de chaque arbre qui contrôle le degré de spécialisation de chaque arbre par rapport à l'ensemble des données d'apprentissage. Il existe deux façons principales de **contrôler la complexité des arbres** : la **profondeur maximale** des arbres et le **nombre maximal des feuilles (num_leaves)** dans l'arbre.
- **learning_rate**: importance de la contribution de chaque modèle à la prédiction de l'ensemble (taux plus faibles => plus d'arbres de décision)

Une optimisation des hyperparamètres via la librairie FLAML a ensuite été faite comme étape ultime.

3.3. Métriques d'évaluation

Le **Recall** qui mesure le taux de vrais positifs est à favoriser au détriment de la **Précision** qui est la capacité du classificateur à ne pas étiqueter comme positif un échantillon qui est négatif. Pour faire cela, nous allons nous baser sur le **F-beta score** qui est la moyenne harmonique pondérée de la précision et du rappel. Le paramètre beta détermine le poids du rappel dans le score. Lorsqu'il est supérieur à un, il favorise le rappel. Un pallier ayant été atteint à partir de $\beta \sim 9$, nous conserverons F-beta score = 9.

Nous conserverons également **l'Accuracy** et **l'AUC** comme éléments de comparaison. Le **temps d'entraînement** sera également tracké.

Prêt à Dépenser

Note méthodologique

Le choix du meilleur modèle se fera via **cross validation** sur le **F-beta score = 9** puis sur les données de test en fonction du **score métier** et du **temps d'entraînement**.

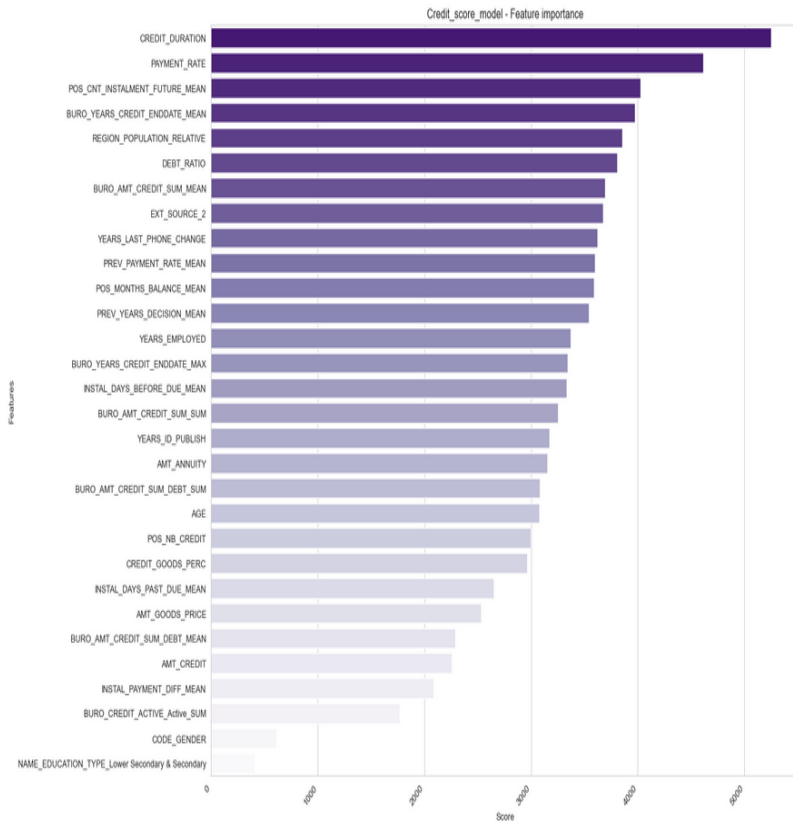
4. Tableau de synthèse des résultats

	Modèle	Best_Params	Score_métier	Betascore	Recall	Precision	Accuracy	AUC	Train_Time
0	Baseline - Dummy classifier	{'classifier__strategy': 'uniform'}	53949	0.46	0.49	0.08	0.50	0.49	28.68
1	Régression Logistique	{'preprocessor': StandardScaler(), 'classifier__solver': 'saga', 'classifier__penalty': 'l2', 'classifier__C': 10}	49388	0.01	0.01	0.47	0.92	0.51	2312.90
2	Régression Logistique - Class Weight	{'preprocessor': StandardScaler(), 'classifier__solver': 'lbfgs', 'classifier__penalty': 'l2', 'classifier__C': 0.01}	33947	0.64	0.67	0.16	0.69	0.68	4693.82
3	Régression Logistique - SMOTE 10/50	{'preprocessor': StandardScaler(), 'classifier__solver': 'lbfgs', 'classifier__penalty': 'l2', 'classifier__C': 10}	37483	0.36	0.36	0.24	0.86	0.63	2204.43
4	Régression Logistique - SMOTE 20/30	{'preprocessor': StandardScaler(), 'classifier__solver': 'saga', 'classifier__penalty': 'l2', 'classifier__C': 100}	46480	0.09	0.09	0.33	0.91	0.54	6153.89
5	Régression Logistique - SMOTE 10/60	{'preprocessor': MinMaxScaler(), 'classifier__solver': 'saga', 'classifier__penalty': 'l2', 'classifier__C': 100}	36032	0.43	0.43	0.22	0.83	0.65	2210.69
6	Régression Logistique - CW - FS	{'preprocessor': StandardScaler(), 'classifier__solver': 'lbfgs', 'classifier__penalty': 'l2', 'classifier__C': 100}	38147	0.59	0.61	0.14	0.66	0.64	95.94
7	LightGBM	{'preprocessor': StandardScaler(), 'classifier__num_leaves': 8, 'classifier__n_estimators': 5000, 'classifier__max_depth': 10, 'classifier__learning_rate': 1.0, 'classifier__boosting_type': 'gbdt'}	32546	0.65	0.67	0.17	0.71	0.69	585.05
8	LightGBM - Feature Importance *5	{'preprocessor': MinMaxScaler(), 'classifier__num_leaves': 8, 'classifier__n_estimators': 500, 'classifier__max_depth': 5, 'classifier__learning_rate': 1.0, 'classifier__boosting_type': 'gbdt'}	36203	0.61	0.64	0.15	0.68	0.66	63.78
9	LightGBM - Feature Importance *10	{'preprocessor': MinMaxScaler(), 'classifier__num_leaves': 16, 'classifier__n_estimators': 100, 'classifier__max_depth': 3, 'classifier__learning_rate': 0.1, 'classifier__boosting_type': 'dart'}	34980	0.63	0.65	0.16	0.69	0.67	79.95
10	LightGBM - Feature Importance *20	{'preprocessor': StandardScaler(), 'classifier__num_leaves': 4, 'classifier__n_estimators': 500, 'classifier__max_depth': 6, 'classifier__learning_rate': 0.001, 'classifier__boosting_type': 'dart'}	34051	0.64	0.66	0.16	0.70	0.68	124.68
11	LightGBM - Feature Importance *30	{'preprocessor': StandardScaler(), 'classifier__num_leaves': 32, 'classifier__n_estimators': 5000, 'classifier__max_depth': 10, 'classifier__learning_rate': 1.0, 'classifier__boosting_type': 'gbdt'}	33173	0.64	0.67	0.17	0.70	0.69	174.92
12	LightGBM 30 - GridSearchCV	{'classifier__num_leaves': 512, 'classifier__n_estimators': 300, 'classifier__max_depth': 2, 'classifier__learning_rate': 0.005, 'classifier__boosting_type': 'rf'}	33173	0.64	0.67	0.17	0.70	0.69	175.67
13	LightGBM 30 - Beta = 2	{'classifier__num_leaves': 3, 'classifier__n_estimators': 300, 'classifier__max_depth': 7, 'classifier__learning_rate': 0.002, 'classifier__boosting_type': 'rf'}	33173	0.64	0.67	0.17	0.70	0.69	174.59
14	LightGBM 30 - FLAML	{'n_estimators': 31204, 'num_leaves': 4, 'min_child_samples': 3, 'learning_rate': 0.009033979476164342, 'log_max_bin': 10, 'colsample_bytree': 0.5393339924944204, 'reg_alpha':	32750	0.64	0.66	0.17	0.72	0.69	277.58

Le LightGBM avec les 30 features les plus importantes optimisé via FLAML est le modèle qui associe:

- un **score métier faible**
- un **temps d'entraînement** relativement court
- un **F-beta score = 9** assez important
- un **nombre de features** relativement faible
- des **features facilement explicables** pour quelqu'un de non expert en datascience

5. Interprétabilité globale et locale du modèle

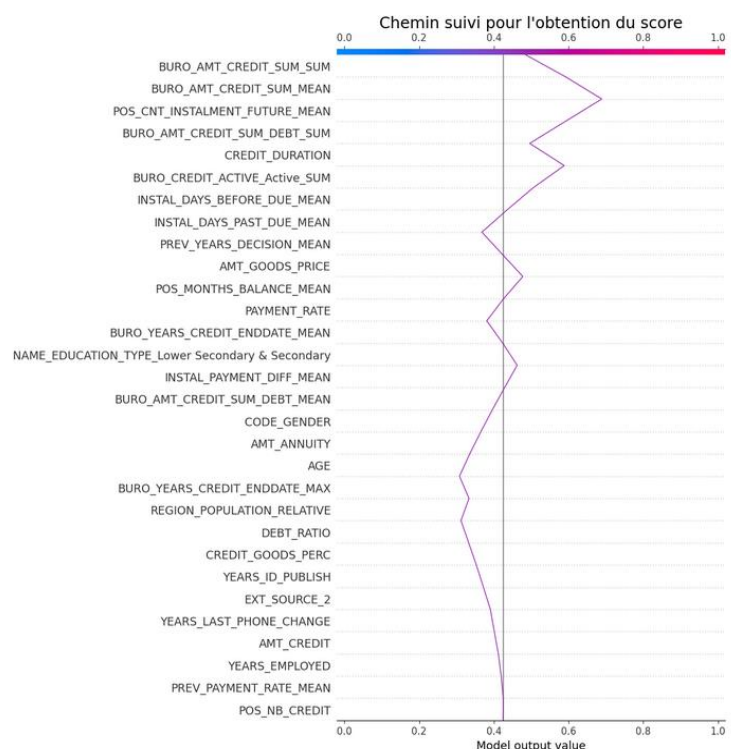


Les décisions d'octroi de crédit devant être expliquées aux clients, il est nécessaire de mettre à disposition du conseiller clientèle non expert en datascience des outils simples et facilement compréhensibles.

Tout d'abord, visualiser l'importance des features permet de mieux comprendre quelles sont les **variables en entrée** du modèle qui ont le **plus d'utilité dans la prédiction de la variable cible** (interprétabilité globale). Dans un modèle de prédiction, l'importance des features permet d'avoir une **meilleure compréhension des données et du modèle** et offre une **base pour la réduction de dimension**, permettant la sélection de caractéristiques qui peuvent améliorer notre modèle. Nous remarquons que la

durée du crédit ou encore la part de l'annuité dans le montant du crédit vont permettre au modèle mis en production de prédire le risque de défaillance alors que le genre du client est nécessaire au modèle mais dans une moindre mesure.

Ensuite, la visualisation des features importance locales est nécessaire afin de comprendre quelles sont les **caractéristiques qui contribuent à une prédiction particulière et dans quelle mesure** afin de pouvoir justifier l'accord ou non du prêt au client. Deux bibliothèques ont été testées : LIME et SHAP. **SHAP** a été retenu pour sa **facilité d'intégration dans le dashboard créé via Streamlit**. La valeur de shap représente l'impact moyen qu'une variable a pour toutes les combinaisons de variables possibles. **La prédiction pourra être décrite comme la somme des différents effets des variables (valeur de shap) ajoutée à la valeur de base (moyenne de toutes les prédictions du dataset).**



6. Limites et améliorations possibles

La **méconnaissance du milieu bancaire**, le **nombre de variables** disponibles ainsi que la **difficulté à les comprendre** a rendu les parties sur le nettoyage de données, le feature engineering et la sélection de variables relativement compliqués. Il aurait été nécessaire de pouvoir échanger avec un **expert métier** pour avoir une idée sur les variables prises en compte en général dans les décision d'octroi de crédit et qui aurait pu apporter sa contribution lors de la création de nouvelles variables par exemple.

De plus, de nombreuses hypothèses ont été prises dans la construction du modèle sans avoir l'avis éclairé de Prêt à Dépenser. Il s'agit par exemple du **choix du seuil de prédiction** ou encore des **métriques d'évaluation**. Un compromis doit être fait entre le nombre de faux négatifs et de faux positifs mais il nous manque la **connaissance métier pour décider ce qui est acceptable ou non pour la banque**. Il aurait également pu être intéressant de connaître le **coût des faux négatifs et faux positifs**.

Concernant le **dashboard**, d'autres fonctionnalités pourront être ajoutées comme **simuler une demande de crédit sur un prospect dont l'ID n'est pas présent** dans la base « Application ». Les variables en entrée du modèle seraient beaucoup plus simples et standards, basées sur du déclaratif. Elles permettraient uniquement au prospect de se faire une idée, sans engager le conseiller bancaire.

L'utilisation du paramètre Streamlit **st.session_state** pourrait permettre, lorsque le conseiller clique sur le profil du demandeur, de garder en mémoire l'ID sélectionné ainsi que la prédiction sur la page Scoring client.

Les **variables** pourraient également être **renommées** pour être plus facilement interprétables notamment dans les graphiques d'interprétabilité globale et locale.















Un **clustering** aurait également pu être fait afin de situer le client parmi les profils similaires (et non pas uniquement les défaillants et non défaillants).

7. Analyse du Data Drift

La librairie **evidently** a été utilisée pour détecter la présence de data drift en production dans le futur. L'hypothèse prise est que le dataset `application_train` représente les données pour la modélisation et le dataset `application_test`, les données de nouveaux clients une fois le modèle en production. Le rapport evidently sur le data drift **compare les distributions de chaque variable dans les deux datasets**. Le **test statistique** (ou la métrique appropriée) est automatiquement choisi en fonction du **type des données** et de leur **volume** et retourne les p-values ou distances et trace les distributions. Les tests statistiques ou palliers peuvent être ajustés manuellement.

Sur les 30 features de notre modèle de scoring, **7 features** montrent du data drift (soit **23.33%** du nombre total de features):

- PAYMENT_RATE
- CREDIT_DURATION
- AMT_GOODS_PRICE
- AMT_CREDIT
- AMT_ANNUITY
- YEARS_LAST_PHONE_CHANGE
- PREV_PAYMENT_RATE_MEAN

Column	Type	Reference Distribution	Current Distribution	Data Drift	Stat Test	Drift Score
PAYMENT_RATE	num			Detected	Wasserstein distance (normed)	0.582838
CREDIT_DURATION	num			Detected	Wasserstein distance (normed)	0.582854
AMT_GOODS_PRICE	num			Detected	Wasserstein distance (normed)	0.236377
AMT_CREDIT	num			Detected	Wasserstein distance (normed)	0.234577
AMT_ANNUITY	num			Detected	Wasserstein distance (normed)	0.149195
YEARS_LAST_PHONE_CHANGE	num			Detected	Wasserstein distance (normed)	0.129542
PREV_PAYMENT_RATE_MEAN	num			Detected	Wasserstein distance (normed)	0.103929

Cela signifie que la **distribution des données en entrée du modèle, vs celle des données actuelles a changé au fil du temps**. Comme les données catégorielles ont été encodées à l'aide de la méthode OneHot, toutes les features du dataset sont désormais **numériques**. Le test utilisé est donc la **distance de Wasserstein (normalisée)** (WD). Il s'agit de la méthode par défaut pour les données tabulaires numériques supérieures à 1000 lignes.

WD mesure l'**effort nécessaire pour transformer une distribution en une autre**. WD normalisée indique le **nombre d'écarts types en moyenne qu'il faudrait déplacer pour chaque ID du groupe actuel pour qu'il corresponde au groupe de référence**. Le seuil de détection du data drift de 0.1 signifie donc qu'un **changement dans la taille des écarts-types de 0.1 est significatif**.

Pistes à envisager pour gérer le data drift du modèle en production :

- 1) Mettre en place des **contrôles sur la qualité et intégrité** des données
- 2) S'assurer que les **relations** entre les features ou entre les features et la target n'ont pas changé
- 3) **Réentraîner** le modèle lorsque de nouveaux labels seront disponibles
- 4) **Recalibrer** ou **reconstruire** le modèle, créer un **modèle spécifique** pour les segments qui échouent
- 5) Modifier le traitement des **valeurs aberrantes**
- 6) Si le modèle est jugé peu performant, utiliser plutôt un **ensemble de règles de décision** moins précises mais plus robustes