1. Contexte

La société financière "Prêt à dépenser" propose des crédits à la consommation pour des personnes ayant peu ou pas du tout d'historique de prêt. Elle souhaite mettre en œuvre un outil de «crédit scoring» qui calcule la probabilité de remboursement d'un crédit-client et classifie la demande en crédit accordé ou refusé.

Elle souhaite donc développer un algorithme de classification qui s'appuiera sur des sources de données variées (données comportementales, données provenant d'autres institutions financières, etc...).

Ce mémoire présente donc un processus de modélisation et une interprétation de ce modèle pour proposer un modèle de Scoring complet de la probabilité de défaut de paiement d'un client.

2. Méthodologie d'entraînement

Pour l'entraînement de nos modèles un jeu de données quasi-relationnel issu de Kaggle est utilisé et analysé.

Il est décrit par :

- 8 tables dont 1 table principale descriptive des clients pour 1 total de 307511 clients et de 7 autres tables historiques de détail du crédit.
- 122 caractéristiques (features) descriptives d'un client dont :
  - 1 Cible (feature Target de valeur 1 pour indiquer un client en défaut de paiement et de valeur 0 sinon)
  - 121 autres caractéristiques clientes (de type âge, sexe, emploi, logement, revenus, informations relatives au crédit, notation externe, ...)

1) On utilisera pour l'analyse Exploratoire (EDA) préalable le Kernel Kaggle «LightGBM with Simple Features» disponible sur https://www.kaggle.com/code/mehmettt123/lightgbm-with-simple-features.

   Ce noyau permet :

   - ➔ de fournir des features déjà analysées, traitées et nettoyées.
   - ➔ d'agréger les features de détail de crédit avec 5 les opérations suivantes : moyenne, somme, variance, min et max. Il s'agit de compresser l'information globale du jeu de données.
   - ➔ de combiner certaines features pour en créer de nouvelles et réduire ainsi la taille du nombre de features et augmenter l'information utile.
   - ➔ d'encoder toutes les variables catégoriques (réprésentation des variables sous forme de vecteurs binaires) avec l'encodage One-Hot Encoder de python, exigence préalable nécessaire pour la modélisation.

2) Après analyse EDA on continuera la sélection des features pertinentes avec une réduction du nombre de features :

   - ➔ en conservant les features ayant un taux de valeurs manquantes Missing Value % <= 45%
   - ➔ en imputant les valeurs manquantes des features ayant un taux de remplissage > 55%
   - ➔ puis en filtrant les features d'information quasi-nulle c'est à dire celles de variance **proche de 0** ou de **grande variance**.

     On utilisera pour cela le coefficient de variation avec la règle de filtrage suivante :

     Coefficient de variation : CV = (Écart-type / Moyenne) * 100

     Si CV < 0.01 alors feature rejetée

     Si CV >= 0.01 et CV <= 30 alors feature acceptée

3) Pour l'entraînement du modèle le jeu de données sera scindé en 3 parties :

   - ➔ les données clientes sans Target sont isolées et ne sont pas utilisées directement pour l'entrainement des modèles mais sont utilisées par le Kernel Kaggle pour le calcul agrégatif. Elles pourront ensuite être utilisées pour la prédiction de score de ces clients.

   - ➔ les données clientes avec Target sont scindées suivant le modèle (Train,Test) avec la répartition suivante :

     - ✔ un jeu d'entrainement Train (80% d'individus) .
     - ✔ un jeu de Test (20 % d'individus) pour l'évaluation finale du modèle.

Il s'agit d'un problème de classification binaire avec une **classe minoritaire** en sous-représentation (9 % de clients en défaut contre 91 % de clients sans défaut).

Il s'agit d'un problème d' « imbalance classification » pour lequel on rétablir le déséquilibre pour pouvoir modéliser correctement.

On utilisera le modèle Naïf Bayes comme modèle de référence. Il donne une très bonne précision (accuracy) de classement de l'ordre 0.92 avec la stratégie « most frequent » et une précision de 0.5 avec une stratégie aléatoire (uniform).

Néanmoins cette précision n'est pas suffisante car elle ne tient pas compte du déséquilibre des classes et en fin de compte ce modèle classe très mal les clients à risque - ceux qui sont en défaut de paiement.

On va donc utiliser ici une technique d' **Oversampling** de type **SMOTE** pour rééquilibrer le jeu de données. SMOTE permet de créer de nouveaux individus de la classe minoritaire de façon synthétique à partir de données existantes de cette même classe minoritaire (en 3 étapes: 1/ un individu A de la classe minoritaire est choisi au hasard et ses k voisins les plus proches de la même classe sont calculés 2/ Un voisin B est choisi au hasard 3/ un nouvel exemple est créé en sommant une distance aléatoire issu de la distance des 2 points au point A).

En positionnant le resampling à «auto» dans SMOTE on obtient une répartition 50/50 (classe majoritaire / classe minoritaire) au final.

Pour entraîner le modèle on va utiliser conjointement :

- ➔ la standardisation des features en entrée : utile pour optimiser l'entraînement (algorithme de Descent Gradient utilisé pour minimiser la fonction de coût de la régression logistique).
- ➔ L'algorithme d'optimisation décrit ci-dessous (voir paragraphe 3.3)

Etant donné la taille du jeu d'entrainement il n'est pas possible d'entrainer directement tout le jeu de données car les temps d'exécution sont très longs avec la recherche des hyper-paramètres.

On va donc procéder à 1 entraînement sur 1 sous-échantillon (subsample) de faible taille de l'ordre de 0.05 %.

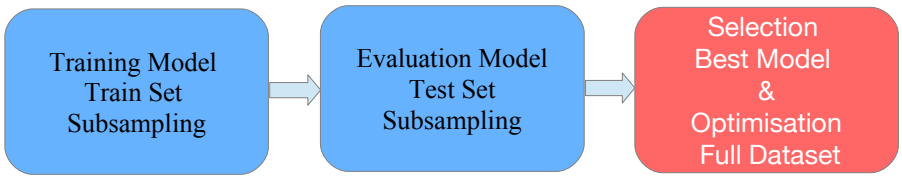
On calculera ensuite le score sur l'ensemble de Test (décrit plus haut) pour évaluer la qualité de l'apprentissage.

Les modèles utilisés sont :

- ✔ Logistic Regression
- ✔ XGBoost
- ✔ RandomForest

4) Enfin la dernière étape sera d'optimiser le meilleur des 3 modèles sur le jeu complet (ici on prendra une taille de 70% pour des raisons de temps d'exécution) : on s'appuiera sur les meilleurs hyper-paramètres obtenus sur le meilleur modèle qu'on affinera légèrement ensuite.

On peut résumer ce processus global par le schéma suivant :



3. Fonction coût, métrique d'évaluation et algorithme d'optimisation

3.1.Fonctions coût

Modèle	Fonction de Coût
Régression Logistique	$$\text{Log Loss} = \sum_{(x,y)\in D} -y\log(y') - (1-y)\log(1-y')$$ On utilise la régularisation L1 pour réduire la complexité du modèle sans laquelle la régression logistique continue à entraîner une perte jusqu'à 0 dans les dimensions élevées et limiter l'overfitting. La fonction de coût Log Loss est ainsi constituée qu'elle pénalise les classifications fausses en prenant en compte les probabilités de classification : cela signifie sur 1 problème de classification à 2 classes 0 et 1 que l'on quantifie le coût de classification d'une instance par sa probabilité sur la classe 1 ou sur sa classe 0 même si l'une des 2 est fausse.
XGBoost	$$\text{Log Loss} = \sum_{(x,y)\in D} -y\log(y') - (1-y)\log(1-y')$$ La fonction de côut Log Loss est identique à celle de la régression logistique.
RandomForest	Minimisation du coefficient d'impureté de GINI pour chaque noeud La classe majoritaire est choisie par vote majoritaire sur les N Trees de la forêt La probabilité est calculée comme le comptage de la fraction d'arbres qui votent pour une certaine classe (deux ici)

3.2. Métriques d'évaluation

On a pour notre jeu de données :

- 91 % de clients sans défaut de paiement
- 9 % de clients en défaut de paiement

L'objectif de la banque idéalement est double :

- prévoir parfaitement les client sans défaut de paiement
- et donc de prévoir parfaitement les clients en défaut de paiement

Le risque de prévoir à tort un client sans défaut est une perte financière.
Le risque de prévoir à tort un client en défaut est une perte cliente.

On peut formaliser ces deux assertions dans une matrice de confusion décrivant l'ensemble des cas possibles avec la convention préalable suivante :

Un client en défaut est un Vrai Positif (True Positif = TP)
Un client sans défaut est un Vrai Négatif (True Négatif = TN)

Prédiction / Client réel	En défaut	Sans défaut
En défaut	TP	FN
Sans défaut	FP	TN

On a donc deux types de prédictions erronées :
– soit un Faux Positif (FP) : on prédit un client qui est en défaut de paiement (Positif) alors qu'il ne l'est pas réellement. Il s'agit d'une erreur de type 1.
– soit un Faux Négatif (FN) : on prédit un client qui n'est pas en défaut de paiement (Négatif) alors qu'il l'est réellement. Il s'agit d'une erreur de type 2.

On cherchera donc à optimiser (en maximisant) soit le nombre de TP (client prédit positif et réellement positif) soit le nombre de TN (client prédit négatif et réellement négatif) ou de façon équivalente à minimiser soit le nombre de FP soit le nombre de FN car il n'est pas possible d'optimiser les deux simultanément (le compromis precision/recall impose un compromis à faire entre Recall et Precision, si l'on augmente la précision alors le recall décroît et inversement)

On peut traduire ces deux objectifs par les 2 métriques suivantes :

$$\text{Recall} = TP / (TP + FN)$$

$$\text{Precision} = TP / (TP + FP)$$

Si on prend pour hypothèse qu'une erreur de type 2 est plus coûteuse financièrement qu'une erreur de type 1 alors il est plus important de minimiser le nombre de FN que celui de FP.

On cherchera donc à maximiser le Recall plus que la Precision puisque c'est le Recall qui prend en compte le nombre de FN.

On utilisera alors une combinaison linéaire pondérée de FN et FP qui accordera plus de poids aux FN et c'est la métrique du Fβ Score suivante qui nous fournira la moyenne harmonique pondérée de la Precision et du Recall (sa valeur optimale est de 1 et sa valeur minimale de 0) :

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{\beta^2 \cdot precision + recall}$$

$$F\beta = \frac{(1+\beta^2) \times TP}{((1+\beta^2) \times TP + \beta^2 \times FN + FP)}$$

C'est le coefficient β qui détermine le poids du Recall relativement à celui de la Precision :

– si beta < 1 alors on donnera plus de poids à la Précision

– si beta > 1 alors on favorise le Recall.

Cette donnée ne nous étant pas accessible on prendra pour hypothèse une valeur β=2 avec une valeur de Recall deux plus fois importante que celle de la Précision.

On obtient donc la métrique F(β=2) suivante qu'on cherchera à maximiser :

$$F2 = \frac{5.TP}{(5.TP + 4FN + FP)}$$

3.3. Algorithme d'Optimisation

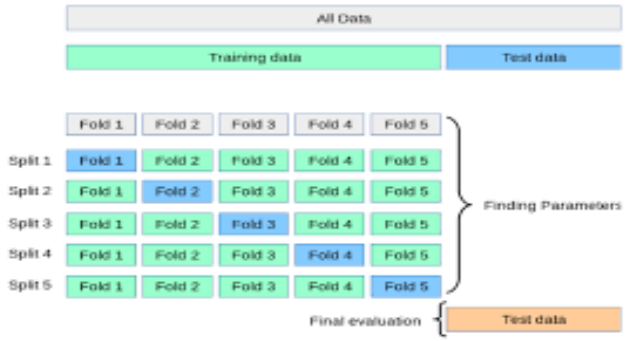
Pour optimiser chacun des 3 modèles on utilisera conjointement :

- la K-Fold Cross-Validation : elle permet de minimiser l'overfitting. Elle s'appliquera sur l'ensemble Train décrit plus haut avec un K Folds (1 Fold est un sous-échantillon) de valeur K=3 pour minimiser les temps d'entraînement (dans la K-Fold cross-validation on découpe les data avec K-1 folds pour l'entraînement et un fold restant pour évaluer le modèle et on itère K fois ce découpage en décalant à chaque fois le fold restant sur la droit comme montré dans le schéma ci-dessous)
- la recherche et l'optimisation par Grille (GridSearchCV) des hyper-paramètres qui s'appuie sur la K-Fold Cross-validation et :

  1/ prendre en entrée des données oversamplées (Fold) SMOTE (pipeline SMOTE – seul les Folds d'entraînement doivent être oversamplés et pas le Fold validation)

  2/ appliquer les modèles choisis (énoncés ci-dessus)

On peut décrire la Cross-Validation à K=5 Folds par le schéma suivant:



Optimisation des Hyper-Paramètres

Modèles	Hyper-Paramètres
Régression Logistique	**Penalty** : Add a penalty tem of type l1 or l2. L1 Penalty (lasso) adds "*absolute value of magnitude*" of coefficient as penalty term to the loss function. Penalty L2 adds "*squared magnitude*" of coefficient as penalty term to the loss function. The key difference between these techniques is that Lasso shrinks the less important feature's coefficient to zero, removing some feature together **C**: Constante Inverse of regularization strength; smaller values specify stronger regularization. Les meilleurs paramètres obtenus sont : 'penalty' : L1 'C ' : 0.003
XGBoost	**learning_rate** : **It affects how quickly the model fits the residual error using additional base learners.** A low learning rate will require more boosting rounds to achieve the same reduction in residual error as an XGBoost model with high learning rate. step size shrinkage used to prevent overfitting. Range is [0,1] A smaller learning rate will increase the risk of overfitting **max_depth**: determines how deeply each tree is allowed to grow during any boosting round. Directly control model complexity (min_child_weight and gamma too). Dépend du degré d'interaction entre les features : avec 1 degré 2 interaction de 2 variables **subsample**: percentage of samples used per tree. Low value can lead to underfitting. **colsample_bytree**: percentage of features used per tree. High value can lead to overfitting. **n_estimators**: number of trees you want to build. Les meilleurs paramètres obtenus sont : 'colsample_bytree : 0.5 'gamma' : 0

	'learning_rate' : 0.1 'max_depth' : 2 'n_estimators' : 2 'subsample' : 0.3
RandomForest	**n_estimators :** determines the number of different decision trees used for averaging. Higher number of trees give you better performance but makes your code slower. **max_depth :** represents maximum the depth of each tree in the forest. The deeper the tree, the more splits it has and it captures more information about the data. overfits for large depth values. **max_samples :** determines the number of predictor variables to be considered in each tree. There is exists an unexplained trade-off balance between the number of features chosen and the performance of the model. If the number of features increases then the model will be obviously able to learn more about the data which makes the coherence better but it will also affect the performance and complexity of the model. The inverse scenario is also possible **min_samples_split :** specifies the minimum number of samples required to split an internal node. A higher value can conduct to underfitting and a too low value can conduct to overfitting **max_features :** represents the number of features to consider when looking for the best split. max_features generally improves the performance of the model as at each node now we have a higher number of options to be considered. **min_samples_leaf :** minimum number of samples required to be at a leaf node. A higher value can conduct to underfitting and a too low value can conduct to overfitting 'max_depth' : 256 'max_features' : 0.02 'max_samples' : 1 'rmin_samples_leaf' : 5 'min_samples_split' : 10 'n_estimators' : 256

4. Interprétabilité

Le modèle est destiné à des équipes opérationnelles et il faudra donc pouvoir expliquer les décisions de l'algorithme aux clients pour les décisions d'acceptation ou de refus de crédit.

On parle alors d'interprétabilité du modèle pour définir notre capacité à pouvoir comprendre les raisons de décision du modèle.

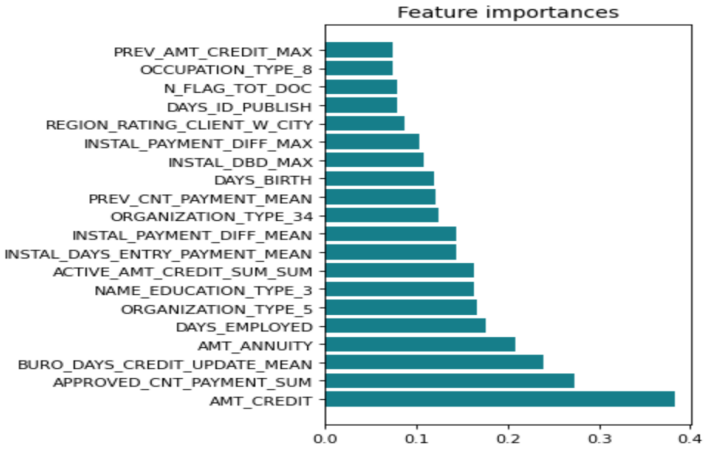
Il existe deux types d'interprétabilité :

- ➔ Interprétabilité globale
- ➔ Interprétabilité locale

4.1 Interprétabilité Globale

Il s'agit d'un niveau d'interprétation basé sur une vue holistique de toutes les features et sur chacun des paramètres appris tels le poids des features notamment. Quelles features sont importantes et quelles types d'interactions entre elles prennent place ? L'interprétabilité du modèle global doit nous permettre de comprendre la distribution des résultats à un niveau global grâce aux features apprises. C'est un niveau difficile à atteindre et c'est pourquoi l'importance des features constitue un premier niveau d'explication avec la pondération des features.

On peut donner pour le meilleur modèle de la Régression Logistique la «feature importance» suivante établie sur la base des coefficients du modèle :



4.2 Interprétabilité Locale

Avec l' interprétabilité locale on ne cherche plus à expliquer le modèle globalement mais à un niveau local pour lequel on cherche l'explication de chaque prédiction individuellement.

Cette explication pour un individu ou un petit groupe d'individus pouvant être mieux comprises par ses features que pour un modèle global où ces explications pouvant être « noyées, absorbées» par la globalité du modèle.

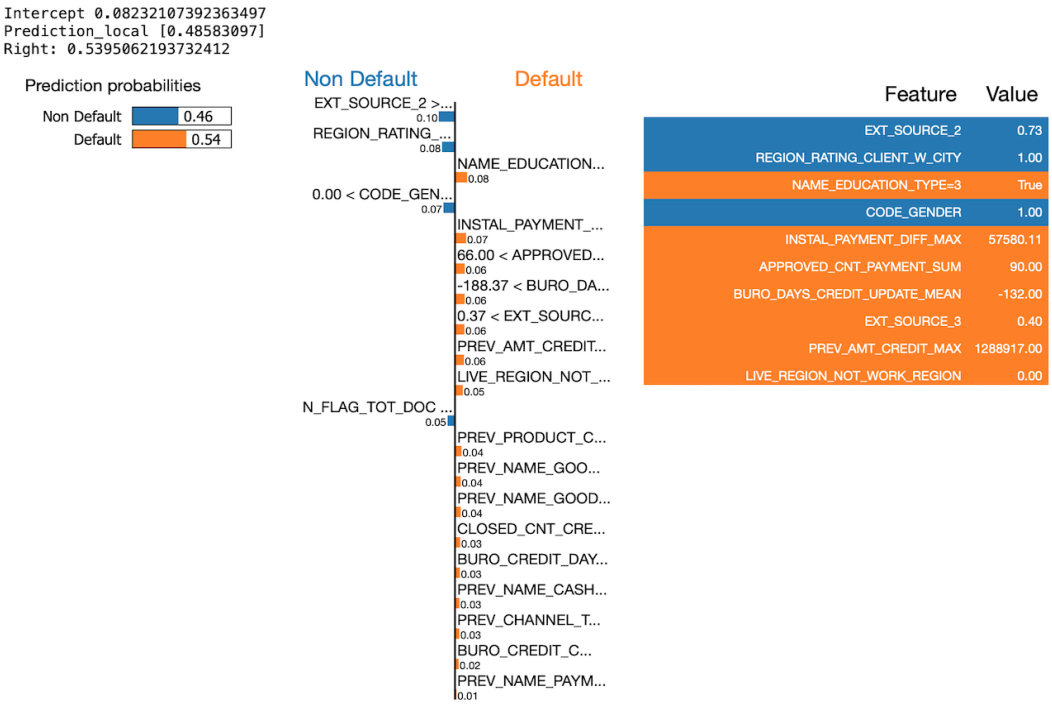
Ici on utilisera LIME acronyme de 'Local Interpretable Model-Agnostic Explanations' comme modèle d'explication locale.

LIME utilise un modèle de substitution au modèle que l'on cherche à expliquer pour expliquer la prédiction (modèle linéaire, arbre de décision …) à un niveau local.

On part d'un point que l'on cherche à expliquer et on calcule un voisinage de ce point (on parle de fake-points car ces points sont le résultat d'une perturbation du point d'origine) qu'on va utiliser comme entrée de notre «black box model» pour générer de nouvelles prédictions en sortie.

Ces nouveaux points reçoivent une pondération qui est fonction de leur proximité au point d'origine et un modèle linéaire (par exemple) est ensuite entraîné sur ces points pondérés pour expliquer le point d'origine.

On obtient donc un modèle linéaire d'interprétabilité locale qui permet de déterminer quelles features ont le plus d'impact sur la prédiction par leur poids (coefficients) respectifs.



5. Limites et améliorations possibles

Feature Engineering

- Le nombre de 440 Features est élevé et devrait être raffiné par le métier par la suite pour éliminer celles totalement inutiles ou rajouter celles nécessaires (et qui ont été éliminés statistiquement).
- Le seuil de 30 pris pour le Coefficient de variation pourrait être réajusté à la hausse car il peut éliminer certaines Features qui pourrait être interessantes d'un point de vue métier.
- D'autres méthodes de sélection de variables pourraient être utilisées comme 'Select Kbest' (qui permet de sélectionner des variables dont le score de dépendance est plus élevé par rapport à la target en utilisant un test de corrélation statistique) ou 'SelectFromModel' à partir d'un autre classifier par ex RandomForest Classifier pour évaluer l'importance des Features
- La synthétisation de nouvelles variables dans le noyau a pu amener une perte d'information.

Optimisation du Modéle

- Le modèle avec un F2 Score assez moyen de 0.41 mais un Recall de 0.65 est donc plutôt bon et il faudrait optimiser le Recall.
- Le choix de beta=2 pour le F2 Score pourrait être ajusté pour tenir compte de la vraie pondération relative entre la perte financière et la perte cliente.
- L'optimisation des modèles faite sur une plage restreinte d'hyper-paramètres pourrait être améliorée avec de meilleures capacités de traitements.
- Algorithme SMOTE :
    - Le nombre de voisins pour l'Oversampling à été fixé à 5 sans aucune optimisation : si la taille du voisinage minoritaire < 5 alors l'individu synthétique est créé dans une zone non représentative, si la taille du voisinage minoritaire > 5 on peut peut être améliorer l'individu synthétique avec plus d'informations à disposition.
    - Il faudrait retraiter les variables discrètes (ex nombre d'enfants) après SMOTE à l'arrondi supérieur ou inférieur car elles peuvent générer une forte variance entre le modele de training et celui de test (écart de performance entre les données Train et Validation/Test)
    - Enfin il faudrait utiliser SMOTE-NC, pour SMOTE-Nominal Continuous pour traiter les variables continues mais aussi les variables catégoriques.

DashBoard

- Le Dashboard permet d'offrir aux chargés de clientèle un outil pour accéder visuellement et simplement aux résultats du modèle avec un modèle explicatif simple fourni par Lime pour une explication locale d'un client donné ou avec la Feature Importance Globale pour un modèle global qu'on devra juxtaposer avec une connaissance métier pour en apprécier la pertinence.
- On peut améliorer la partie Information Cliente en fournissant des graphes bi-variables pour améliorer la compréhension de certaines variables.