# Note méthodologique

## La méthodologie d'entraînement du modèle

### La préparation des données nécessaires à l’élaboration du modèle

Le pré-traitement et la création des variables ont été élaborés à partir des kernels publiés par Will Koehrsen

* [Start Here: A Gentle Introduction | Kaggle](https://www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction)
* [Introduction to Manual Feature Engineering | Kaggle](https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering)
* [Introduction to Manual Feature Engineering P2 | Kaggle](https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering-p2)
* [Introduction to Feature Selection | Kaggle](https://www.kaggle.com/willkoehrsen/introduction-to-feature-selection)

Notre choix a été guidé par l’utilisation de l’exhaustivité des données et par la cohabitation de l’approche métier et d’approches automatisées de Feature engineering.

La richesse des explications de l’auteur a facilité l’adoption.

Ce choix a permis, par ailleurs, d’enrichir notre bibliothèque.

Les adaptations ont consisté à agréger ces quatre kernels et à élaguer les redondances ce qui a permis de répondre aux besoins de la mission. En revanche, nous avons créé un jeu de données supplémentaire en utilisant une réduction de dimensions différente de la méthode initialement développée.

Les caractéristiques des principaux traitements :

* L’encodage des variables catégorielles utilise le « Label encoding » jusqu'à 2 modalités et le « One-Hot Encoding » au-delà. A noter, la nécessité d’aligner les variables des dataframes de test et de train à la suite de l’encodage.
* Le Feature engineering enrichit la table principale (238 variables initialement) de 4, 35 puis 1200 variables résultantes des approches métier, polynomiale et par agrégat, respectivement.
* Le traitement des valeurs manquantes consiste à supprimer les variables ayant plus de 90% de valeurs manquantes et à remplacer les autres valeurs manquantes par la médiane
* Le centrage-réduction est basiquement réalisé au travers d’un « Standard Scaler ».
* La réduction de dimension initialement prévue utilise l’importance des variables d’un modèle de classification « Gradient Boosting ».

La fonction prévoit l’utilisation de plusieurs folders pour éviter le sur-apprentissage.

Comme nous avons souhaité rendre nos résultats déterministes, nous avons défini la graine sur le générateur aléatoire utilisée lors des séparations « train-test » utilisées pour créer les différents folders.

Nous conservons les 11 variables les plus importantes. Le jeu de données résultant de cette réduction, dite ‘LGBM’, présente donc 11 variables.

* La réduction de dimension alternative développée est réalisée au travers :
  + D’une « Analyse en Composantes Principales » avec 60% de variance expliquée qui aboutit à 70 variables et
  + D’une analyse des corrélations à l’issue de laquelle nous conservons les 15 variables les moins corrélées à la variable cible.

Le jeu de données résultant de cette réduction, dite ‘PCA’, présente donc 15 variables.

* L’obtention d’un jeu d’apprentissage équilibré (par rapport à la variable cible) et cohérent avec la puissance de calcul dont nous disposons (Processeur i5, RAM 32 Go) est réalisé comme suit :
  + Isoler 20% du jeu d’entraînement initial pour constituer le jeu de validation.
  + Echantillonner aléatoirement le jeu d’entraînement résultant, ce qui permet de passer de 246 000 à 5 000 individus.
  + Equilibrer avec la méthode SMOTE « Synthetic Minority Over-sampling Technique »

Avec « sampling\_strategy » = 1.0 pour obtenir autant d’individus dans la classe minoritaire que d’individus dans la classe majoritaire initiale et considérer un voisinage à 5 plus proches voisins.

Ainsi, nous passons de 8% à 50% d’individus en défaut (4 599 individus de chaque classe).

L’entrainement du modèle

### Nous avons choisi d’utiliser Pycaret pour sa fonctionnalité de comparaison de modèles côte-à-côte en 2 lignes de code (setup, compare\_models) et pour la qualité de ses visualisations (plot\_model).

## La fonction coût métier, l'algorithme d'optimisation et la métrique d'évaluation

« Prêt à dépenser » cherche à garantir que les clients capables de rembourser ne sont pas rejetés dans la perspective d'élargir l'inclusion financière de la population.

La fonction coût métier, proportionnelle au montant de la demande, pénalise donc 2 fois plus les refus sur les bonnes demandes que les accords sur les mauvaises demandes.

Une image contenant table

Description générée automatiquement

### **L'algorithme d'optimisation** commence par la déclaration d’une fonction de calcul des coûts de chaque demande avec, en argument, le vecteur des classes cibles, le vecteur des classes prédites et un dictionnaire de vecteurs des paramètres supplémentaires kwargs. Ici, le dictionnaire n’aura qu’une clé ‘amt’ car nous n’utilisons que le montant de la demande.

### Ensuite, nous déclarons une fonction de coût total comme étant la somme les coûts des demandes.

### Puis, avec la fonction « add\_metric » de Pycaret, nous ajoutons la fonction de coût total comme métrique d’évaluation tout en transmettant le dictionnaire des paramètres additionnels.

### Tout est prêt pour lancer la fonction « compare\_model » de Pycaret avec sort='Cout' afin d’identifier le meilleur type de modèle dont nous règlons les hyper-paramètres avec la fonction « tune\_model » de Pycaret avec optimize='Cout'.

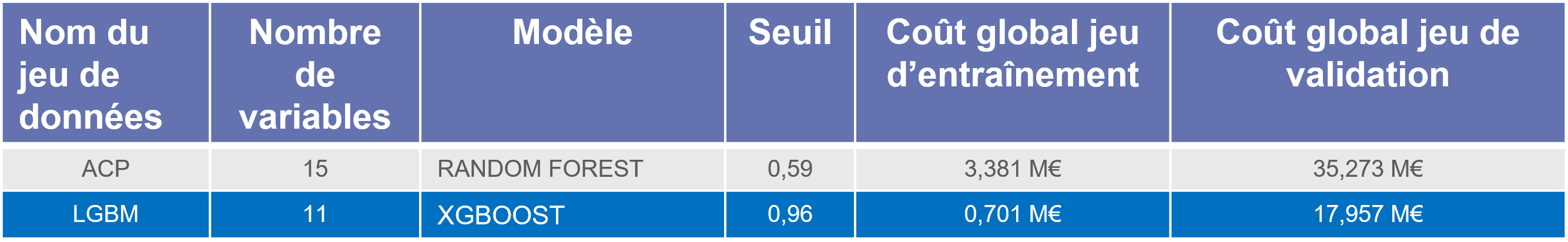
### Nous pouvons ainsi calculer la prédiction, la probabilité de défaut et le coût de chaque demande.

### Enfin, nous pouvons évaluer les résultats à plusieurs niveaux :

### Identifier si nous sommes en situation de sur-apprentissage lors de la phase d’entraînement

* Observer les résultats en coût en distinguant les combinaisons (matrice de confusion) et globalement
* Evaluer le caractère significatif de l’évolution du coût autour du seuil de probabilité minimum en traçant la fonction de coût. Pour cela, nous traçons le coût en fonction du seuil des probabilités de défaut.
* Evaluer la qualité de la généralisation toujours sur la base du coût mais cette fois sur les données de validation que nous avions isolées plus tôt
* Evaluer l’importance globale des variables

Comparons les résultats pour chacun de nos 2 jeux de données (‘LGBM’ et ‘PCA’) :



## L’interprétabilité globale et locale du modèle

### L’interprétabilité globale traite de l’importance des variables issue du modèle optimisé.

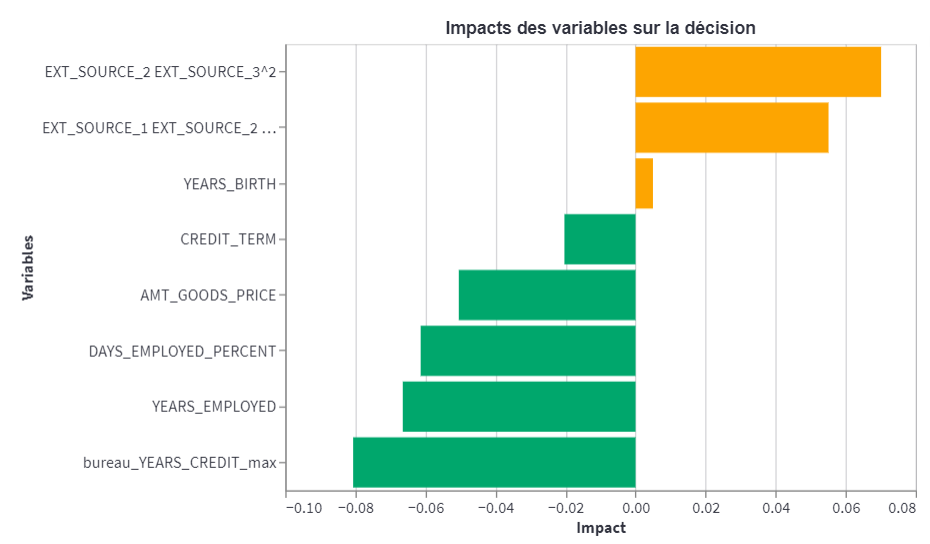
### Sa représentation graphique est immédiate avec la fonction « plot\_model » Pycaret et le paramètre plot='feature'.

### 

### En revanche, l’interprétabilité locale a vocation à expliquer la prédiction pour chaque client.

### La contribution de chaque variable à la prédiction du client est la valeur normalisée de la variable multipliée par l’importance globale de la variable.

Un impact positif, en orange, signifie que la variable fait accroître la probabilité de défaut et un impact négatif, en vert, signifie que la variable fait décroître la probabilité de défaut.



## Les limites et les améliorations possibles

Nous avons fortement limité le nombre d’individus pour des raisons de temps de calcul lors de la phase d’entrainement.

Ce choix a pour conséquence de ne pas avoir assez appris et en conséquence d’observer une mauvaise généralisation de notre modèle. Une amélioration serait d’augmenter progressivement le nombre d’individus jusqu’à obtention d’un apprentissage satisfaisant.

Nous devrions protéger notre application avec un module de demande/autorisation de souscription. Nous n’avons d’ailleurs pas mis en place de gestion d’utilisateurs multiples pour conserver les demandes consultées précédemment.

Nous nous sommes orientés vers une solution de génération d’échantillons afin de mettre en pratique ce genre de méthode. Il serait plus logique de rééquilibrer simplement en sous-échantillonnant la classe majoritaire.