

Projet de scoring de crédit avec pipeline MLOps (Home Credit Default Risk)

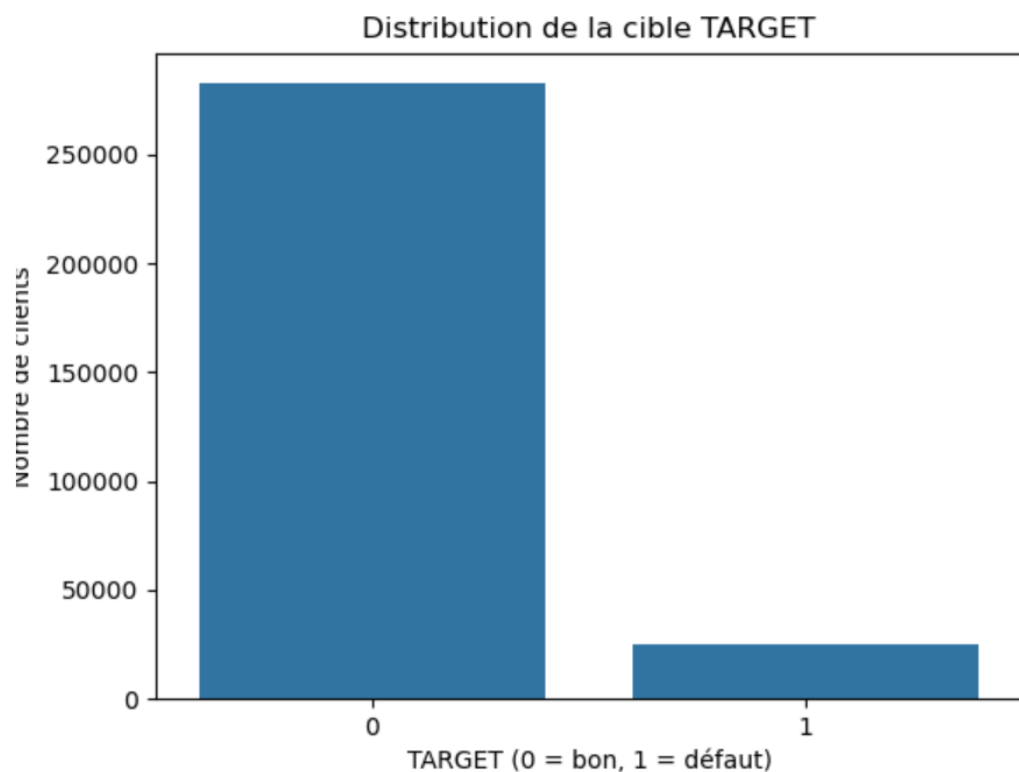
1. Contexte métier

Le modèle doit :

- estimer la **probabilité de défaut** de remboursement d'un client (TARGET = 1) ;
- transformer cette probabilité en **décision** (crédit accordé / refusé) via un seuil ;
- s'appuyer sur des données variées : informations socio-démographiques, historiques de crédits externes, comportement de remboursement, etc.

Deux contraintes métier importantes sont intégrées :

- le jeu de données est **très déséquilibré** ($\approx 8\%$ de défauts, 92% de bons payeurs) ;
- le coût d'un **faux négatif** (mauvais payeur accepté) est environ **10 fois plus élevé** que celui d'un **faux positif** (bon client refusé).



2. Données et préparation (Étape 1)

Les données proviennent du concours Kaggle **Home Credit Default Risk**. Nous utilisons principalement :

- `application_train / application_test` : informations client + cible TARGET (0 = bon payeur, 1 = défaut) pour l'entraînement ;
- `bureau` et `bureau_balance` : crédits externes antérieurs ;
- les autres tables sont préparées mais non encore intégrées dans la version minimale du modèle (elles pourront être ajoutées en extension).

La préparation est factorisée dans le module `src/data_prep.py` :

- `load_raw()` charge l'ensemble des fichiers CSV depuis le dossier `data/` ;
- `aggregate_bureau()` agrège la table `bureau` au niveau client (`SK_ID_CURR`) :
 - agrégations numériques (mean, max, min, sum) sur les montants et soldes de crédits ;
 - encodage One-Hot des variables catégorielles puis moyenne par client (proportion de chaque type de crédit, statut, etc.) ;
- `merge_all()` fusionne `application_train / application_test` avec ces agrégats sur `SK_ID_CURR` ;
- `build_preprocessor()` construit un **préprocesseur scikit-learn** :
 - imputation des valeurs manquantes numériques (médiane) ;
 - imputation des catégories (modalité la plus fréquente) ;
 - encodage One-Hot des variables catégorielles.

Le dataset final (`train_df`) contient donc TARGET + l'ensemble des variables d'application et des agrégats bureau.

Un premier diagnostic confirme :

- le **déséquilibre** de la cible ($\approx 8\%$ de défauts) ;
- la présence de nombreuses colonnes avec un **taux élevé de valeurs manquantes**, conservées puis imputées afin de ne pas perdre d'information potentiellement utile.

3. Modélisation et métriques (Étape 3)

La modélisation est réalisée dans `02_model_training.ipynb` à l'aide de deux modèles :

3.1. Baseline : régression logistique

- Modèle linéaire, utilisé comme **référence simple**.
- Paramètres principaux :
 - `max_iter = 1000`,
 - `class_weight = "balanced"` pour compenser le déséquilibre.
- Le modèle est encapsulé dans un pipeline : `préprocesseur + LogisticRegression`.
- Évaluation par **validation croisée stratifiée** (3 folds) sur un sous-échantillon pour réduire le temps de calcul.
- Métrique principale : **AUC-ROC**.

Résultat typique :

AUC moyenne $\approx 0,63 \rightarrow$ baseline raisonnable, mais insuffisante pour un scoring métier exigeant.

3.2. Modèle avancé : LightGBM

Pour améliorer les performances, nous utilisons **LightGBM**, un modèle de gradient boosting sur arbres, particulièrement adapté aux données tabulaires avec interactions complexes :

- Hyperparamètres principaux :
 - `n_estimators = 300`,
 - `learning_rate = 0.05`,
 - `num_leaves = 31`,
 - `subsample = 0.8`,
 - `colsample_bytree = 0.8`,
 - `objective = "binary"`,
 - `scale_pos_weight = ratio négatifs / positifs ($\approx 11,4$)`, pour mieux prendre en compte la rareté des défauts.
- Le modèle est lui aussi encapsulé dans un pipeline : `préprocesseur + LGBMClassifier`.
- Validation croisée (3 folds) sur un échantillon plus large que pour la baseline.

Résultat :

AUC moyenne \approx **0,75**, avec un écart-type faible \rightarrow gain clair par rapport à la régression logistique.

Des métriques complémentaires (précision, rappel, F1) sont calculées via `src/metrics.py` pour analyser le comportement du modèle sur la classe minoritaire.

4. Coût métier et choix du seuil de décision (Étape 4)

Pour refléter les enjeux économiques, nous avons défini une **fonction de coût métier** dans `src/metrics.py` :

- coût d'un **faux négatif (FN)** = `cost_fn = 10` ;
- coût d'un **faux positif (FP)** = `cost_fp = 1`.

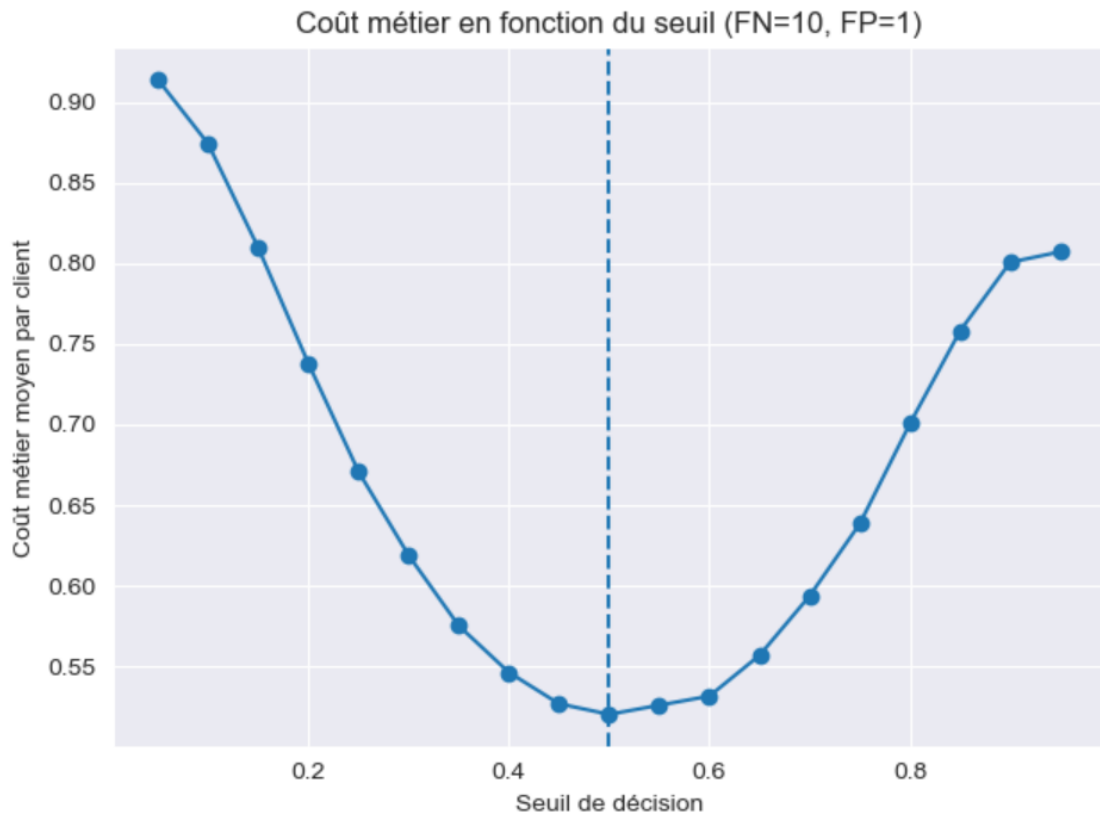
La fonction `business_cost(y_true, y_proba, threshold, cost_fn, cost_fp)` calcule :

- la matrice de confusion (TN, FP, FN, TP) pour un seuil donné ;
- le coût moyen par client :
$$\text{coût} = (10 \times \text{nombre_FN} + 1 \times \text{nombre_FP}) / N.$$

Dans `02_model_training.ipynb` :

1. Le jeu de données est découpé en **train / validation** (20 % de validation, stratifié).
2. Le pipeline LightGBM est entraîné sur la partie train.
3. Nous calculons la **courbe coût vs seuil** avec `cost_curve()` pour des seuils entre 0,05 et 0,95.

4. Nous identifions le **seuil minimisant le coût métier** et sauvegardons la courbe dans `reports/figures/courbe_cout_vs_seuil.png`.



Dans la configuration actuelle, le minimum de coût est obtenu pour un seuil autour de **0,5**. Ce résultat est lié à la combinaison choisie (hyperparamètres LightGBM + pondération `scale_pos_weight`) et montre que :

- même si le seuil « classique » 0,5 n'est pas toujours optimal théoriquement en cas de déséquilibre,
- dans nos essais de base, il reste proche du seuil minimisant le coût métier.

En pratique, une optimisation plus poussée (grid search sur les hyperparamètres + différents réglages de `scale_pos_weight`) pourrait conduire à un seuil plus bas (par exemple 0,2–0,3), ce qui augmenterait encore le rappel sur les défauts.

5. Composante MLOps avec MLflow

La partie MLOps est assurée avec **MLflow** :

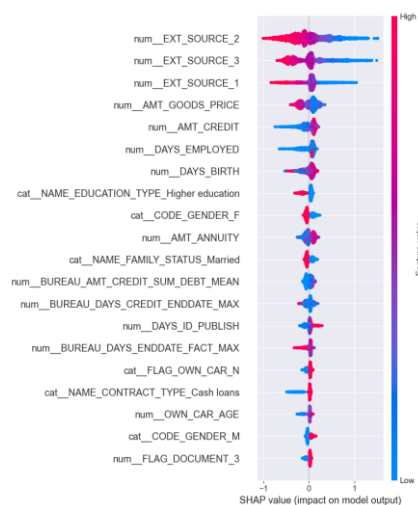
- **Tracking des expériences**
 - L'URI de tracking pointe vers un dossier dédié (`mlruns_V_F`).
 - Chaque run enregistre :
 - les **hyperparamètres** principaux (modèle, seuil métier, pondération de classes) via `mlflow.log_param` ;

- les **métriques** (AUC moyenne, écart-type, coût métier) via `mlflow.log_metric` ;
- les **artefacts** importants, comme la courbe coût vs seuil, via `mlflow.log_artifact`.
- **Registry de modèles**
 - Le pipeline final (préprocesseur + LightGBM) est loggé avec `mlflow.sklearn.log_model`, sous le nom `credit_scoring_model` dans le model registry.
 - Le même pipeline est exporté dans le dossier `model/` pour être servi.
- **Serving du modèle**
 - Le modèle est servi via `mlflow models serve` (en local ou via Docker), en exposant une API REST /invocations.
 - Le notebook `04_mlflow_serving_test.ipynb` envoie un petit échantillon de clients à cette API (format `dataframe_split`) et vérifie :
 - que l'API répond correctement (HTTP 200) ;
 - que la taille du vecteur de prédictions correspond bien à celle de l'échantillon ;
 - que le coût métier calculé à partir des réponses de l'API reste cohérent.

6. Explicabilité (SHAP)

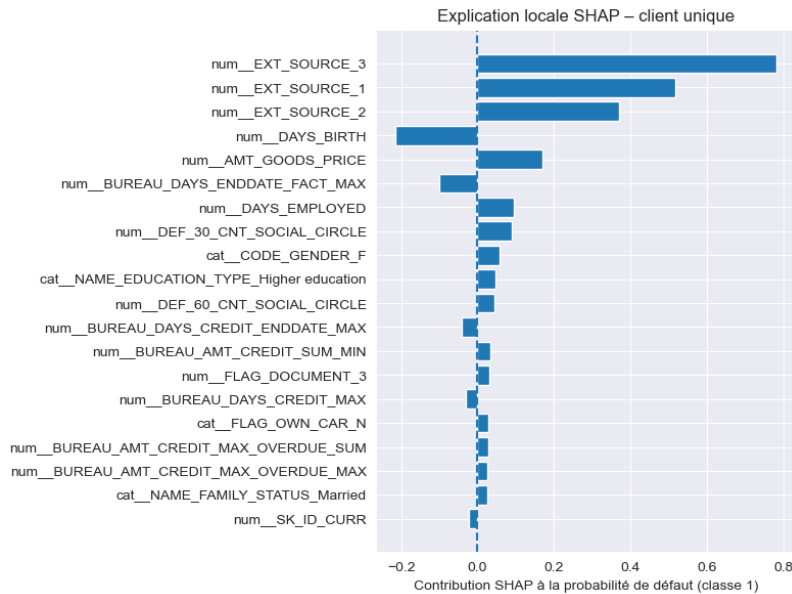
L'explicabilité du modèle est prise en charge dans `03_explainability.ipynb` via le module `src/explainability.py` :

- **SHAP global** (`shap_global_lightgbm`) :
 - utilisation de `shap.TreeExplainer` sur le modèle LightGBM ;
 - génération d'un summary plot (figure `shap_global.png`) montrant :
 - quelles variables sont les plus importantes (importance globale) ;
 - dans quel sens elles influencent la probabilité de défaut.



- **SHAP local** (`shap_local_lightgbm`) :
 - explication détaillée pour un client (figure `shap_local.png`) ;

- les features sont triées par contribution absolue, ce qui permet de justifier la décision au niveau individuel.



Ces explications répondent à la demande de **transparence** du scoring, en aidant un analyste crédit à comprendre pourquoi un client est jugé risqué ou non.

7. Limites et perspectives

Limites actuelles

- Les hyperparamètres de LightGBM sont fixés à la main (configuration raisonnable mais non optimisée).
Une **recherche systématique** (GridSearchCV / Optuna) pourrait encore améliorer AUC et coût métier.
- Seule la table `bureau` est agrégée dans la version minimale.
Les autres sources (`previous_application`, `POS_CASH_balance`, `credit_card_balance`, `installments_payments`) contiennent des signaux importants qui ne sont pas encore exploités.
- Le backend MLflow est un **FileStore local** (`mlruns_V_F`) adapté pour un projet étudiant, mais pas encore une solution de production (base de données partagée, tracking multi-utilisateurs, etc.).
- La calibration des probabilités (fiabilité absolue des scores entre 0 et 1) n'a pas encore été étudiée en détail.

Perspectives d'amélioration

- **Optimisation avancée des hyperparamètres** (nombre d'arbres, profondeur, régularisation, `scale_pos_weight`) pour réduire encore le coût métier.
- **Enrichissement des features** à partir :
 - des précédentes demandes de crédit,

- des flux POS,
 - des remboursements (installments),
 - des cartes de crédit.
- **Calibration des probabilités** (Platt scaling, isotonic regression) afin de rendre le score directement interprétable comme probabilité de défaut.
- Mise en place d'un **backend MLflow plus robuste** (SQLite ou PostgreSQL), et d'un pipeline de déploiement plus complet (Docker + CI/CD).
- Approfondissement de l'**explicabilité locale** (dashboard interactif pour les chargés d'études : affichage des SHAP locaux, variables les plus contributives, etc.).