Daniel Kurmel

# Home Credit Default Risk

# Framing the Problem

## Business Objective

**Broad goal**: Home Credit wants to support financial inclusion by offering safe and fair loans to people who usually don't have access to traditional banking. To do this, they use alternative data—like telecom usage and transaction history—to understand if someone is likely to repay a loan.

**Specific objective**: Predict whether a customer will have issues repaying a loan (TARGET = 1). This is challenging because the dataset is imbalanced: only 8.1% of the customers fall into the risky group (TARGET = 1), while 91.9% do not (TARGET = 0). This imbalance means models might just predict "no issues" all the time unless we do something about it, like using class weights or resampling.

To address this, we will start by exploring the training and test data to understand how it's structured and what patterns it contains. This exploration will guide our feature selection, data cleaning, and model-building efforts. The end goal is to create a model that can accurately predict loan repayment risks—even in this imbalanced setting.

## Data Overview and Setup

For this project, we're working with several datasets provided by Home Credit. The heart of it all is the application_train.csv file, which contains the training data and the TARGET column — our key variable to predict. It's a binary classification problem, where 1 means the client had payment difficulties, and 0 means they didn't. But there's a challenge: the dataset is highly imbalanced, with only about 8.1% in the positive class. That means we need to be thoughtful in how we train models, using techniques like class weighting or resampling to avoid a model that simply predicts the majority class.

The supporting files — like bureau.csv, credit_card_balance.csv, installments_payments.csv, and others — give us a richer view of the customer's financial behavior and history. These

datasets can be merged with the main file to create features that better reflect the customer's overall financial profile.

We've been using a combination of Google Colab and VS Code throughout the process. If you're on an older machine, like a 2019 Intel MacBook, heavier tasks like data exploration and model training can be slow. That's where Colab really helps – giving you access to faster processing and more memory without any setup. On the other hand, VS Code has been great for managing Docker containers and deploying our FastAPI-based app to Google Cloud Platform, which isn't as smooth on Colab due to its limited support for Docker.

For modeling, we've primarily focused on LightGBM because it's fast, efficient, and great at handling large datasets – especially when they have a mix of numeric and categorical features. It also has solid support for handling imbalanced classes, which makes it a good match for this challenge. Still, we tested XGBoost and CatBoost too, just to see how they compared. Exploring different models helps us ensure we're not missing out on better performance.

## Exploratory Data Analysis (EDA)

This is the part where we roll up our sleeves and start understanding what's actually inside the data. Good EDA isn't just about making pretty charts—it's about discovering patterns, catching issues early, and uncovering the signals that will guide our modeling.

1. **Get a Feel for the Dataset** We'll load a sample of the data first to understand its structure: how many rows and columns, the types of variables (numerical, categorical, text), and some basic stats like mean, min, max, and missing values. This gives us the "lay of the land."

2. **Understand the** TARGET We'll look at the distribution of the target variable—how many clients had repayment issues (TARGET = 1) vs. those who didn't. Since the dataset is imbalanced, we'll want to visualize this and keep it in mind through all later steps.

3. **Missing Values and Data Quality** We'll check which features have missing values, how severe they are, and how we might deal with them later (drop, fill, or ignore). This helps prevent issues during modeling and keeps our dataset healthy.

4. **Distribution of Features** For numerical features, we'll look at histograms and box plots to understand the shape of the distributions—are they skewed? Are there outliers? For categorical features, we'll check value counts and bar plots to understand common values and possible encoding needs.

5. **Relationship with the Target** We'll compare feature distributions for TARGET = 0 and TARGET = 1. This helps us spot which variables might be good predictors. For example, if people with late payments tend to have lower income or fewer credit lines, that's useful to know.

6. **Correlation and Redundancy** We'll look at correlations between numeric features, including how strongly each one is linked to the target. This can help us detect duplicate features or highly related ones that might need attention.

7. **Early Feature Importance (Optional)** Even at this stage, it can be useful to train a quick LightGBM model and see what features it finds important. This gives us a sneak peek at which features might drive predictions.

8. **Notes and Documentation** Throughout EDA, we'll take notes—what we found, what we're assuming, and what we still need to explore. These notes will guide feature engineering, modeling, and even help with communicating results later.

To deepen our understanding, we also brought in secondary datasets like bureau.csv and bureau_balance.csv. These datasets don't contain date stamps, but we made a practical assumption that higher SK_ID_BUREAU values likely indicate more recent records. Using this assumption, we computed time-difference-like features by comparing records across IDs. This approach treats the data as "continuous" or snapshot-like rather than strictly temporal.

The Medium article you followed "Mastering EDA" helped structure the primary steps in your analysis. It provided a solid approach to inspecting distributions, correlation structures, missing values, and feature relationships—especially helpful for high-dimensional data like this.

Even though patterns haven't jumped out clearly yet, this EDA lays the groundwork for the next steps: cleaning, feature engineering, and model training.

## Data Preparation and feature Engineering

1. **Data Cleaning** We started with basic cleaning of the main training dataset. That means handling missing values, fixing outliers (if necessary), and making sure the data types are correct.

Instead of dropping rows with missing values (which can be risky in imbalanced problems), we likely opted to fill them — using the median and most frequent entries for categorical variables. This helps keep the training data intact while still being model-friendly.

2. **Feature Reduction and Filtering** We reduced the feature space by removing columns with low variance, excessive missing data, or no predictive power. Importantly, we also identified and dropped highly correlated features that had low importance scores based on an initial LightGBM model. This helped us reduce redundancy and potential overfitting while keeping the model efficient.

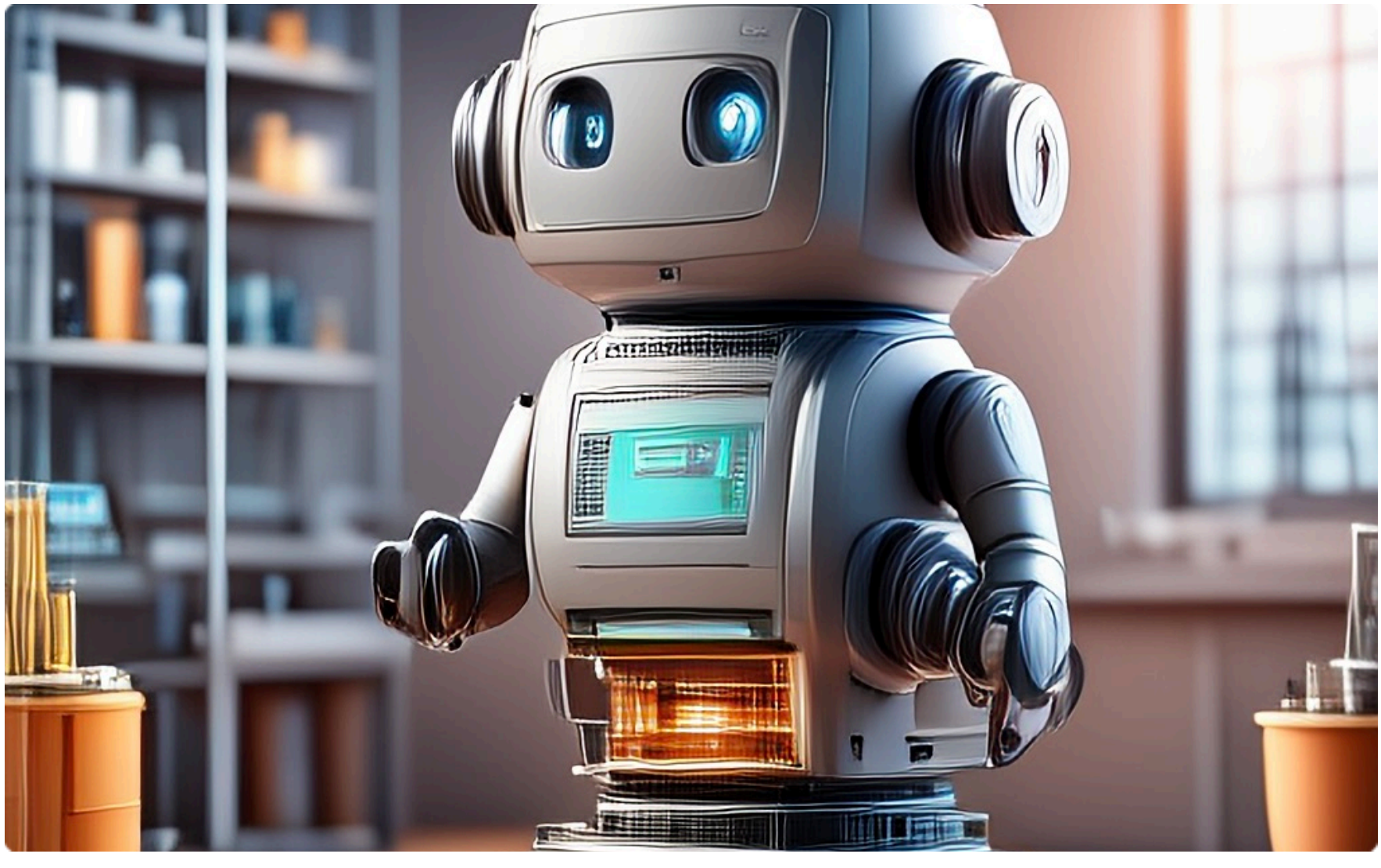
3. **Feature Engineering from Additional Datasets** We used the assumption that higher IDs may represent newer records (since timestamps are missing), and calculated meaningful differences or aggregates. For example:

- Average or max of loan amounts from bureau.csv
- Count of past loans or active credits per customer
- Aggregates of days past due, loan status, etc.

These aggregations were grouped by SK_ID_CURR, allowing us to join them back to the main dataset as additional features.

5. **Feature Scaling** Since LightGBM doesn't require scaled data, we didn't have to standardize or normalize features. But if we ever use models like logistic regression or neural nets, we'll apply scaling where needed.

6. **Save Transformed Data** Once cleaned and engineered, the datasets were saved in a serialized format (e.g., CSV or joblib) to streamline the next steps — making it easy to retrain or deploy models without repeating data prep.



## Model Building and Selection

We focused first on LightGBM, a gradient boosting framework that's known for being fast, memory-efficient, and highly accurate with tabular data. It's especially handy for handling missing values and categorical variables directly, and we leveraged parameters like scale_pos_weight to deal with the class imbalance.

**Other Models for Comparison:**
To cover our bases, we also tested XGBoost and CatBoost. Each of them has its own strengths — CatBoost handles categorical features particularly well, and XGBoost is a widely respected benchmark. Still, LightGBM stood out as the top performer across multiple evaluation runs.

**Evaluation Metrics and Strategy:**
We used stratified cross-validation to evaluate performance with metrics that matter for imbalance: F1-score, Recall, and AUC-ROC. Since our focus is catching class 1 cases, we paid close attention to Recall to avoid false negatives — even if it meant trading off a bit of precision.

**Feature Engineering Experiments:**
We explored custom features through reusable scikit-learn transformers:

- days_transformer converted day-based features into year units.
- social_ratios created ratios like DEF/OBS for social circle features.
- credit_ratios added features like Hour/Day or Quarter/Year inquiry ratios from credit bureau data.

These additions were well-structured and designed to capture more complex behaviors. However, during testing, we found that these engineered features actually hurt model performance — particularly for class 1. Recall and F1-score dropped noticeably. Rather than forcing complexity into the model, we made the decision to remove these features and keep the pipeline clean and interpretable.

## Fine-Tuning and Ensembling

Once we had a strong base model in place with LightGBM and a streamlined feature set, we shifted focus to hyperparameter tuning and pipeline optimization to squeeze out the best performance — especially for the minority class (TARGET = 1).

**Structured Tuning with Optuna**

Instead of relying on manual tweaks or traditional grid/random search, we integrated Optuna — a modern, efficient framework for hyperparameter optimization. We paid close attention to parameters like:

- n_estimators
- learning_rate
- max_depth
- num_leaves
- min_child_samples
- subsample
- colsample_bytree
- reg_alpha
- reg_lambda
- random_state

We used a custom function optimize_model() to streamline the tuning process. It leverages:

- optuna.create_study() to set up the optimization task, targeting maximum AUC.
- A TPE sampler (Tree-structured Parzen Estimator), which efficiently searches the parameter space.
- A structured way to evaluate and retrieve the best pipeline using build_pipeline() with the best parameters.

This function performs multiple trials (default 100) and returns the best pipeline with the optimal parameters injected — ready for training or evaluation. The hyperparameters tuned

included tree complexity, sampling ratios, regularization terms, and learning rate – all critical for LightGBM performance.

**Ensembling Experiments**

While LightGBM was our primary model, we explored ensembling to potentially improve performance:

- Simple voting or averaging of LightGBM + XGBoost + CatBoost.
- Weighted blending based on cross-validated scores.
- Consideration for stacking in the future (i.e., meta-models on top of base learners).

All individual models—including LightGBM, XGBoost, and CatBoost—were tuned using Optuna to ensure a fair and optimized comparison. However, given the model complexity vs. performance trade-off, and considering the class imbalance challenge, we decided to keep the pipeline clean and focused on Optuna-tuned LightGBM for now. Simpler often generalizes better – and it's easier to maintain in production.

**Model Testing on Full Test Set**

Once tuning was complete, we merged the aggregated auxiliary datasets (like bureau, installments, etc.) with the test dataset, using the same feature engineering and cleaning steps. This ensured consistency between training and test data.

We then ran the Optuna-tuned LightGBM model on the test dataset. The results were remarkably consistent:

- AUC and default rate predictions closely matched those seen in the validation set.

- F1-score dropped slightly, but the model still effectively identified risky clients, even in the face of class imbalance.

The model merged with external data and tuned with Optuna is more aggressive in catching risky clients, but it comes with a higher cost in false alarms.

## Presenting the Solution

Once the model was built, tuned, and tested, we focused on making it easy to use in real-life situations. We deployed the final LightGBM model using FastAPI, so it can be accessed through a simple web API. It's designed to be fast, flexible, and forgiving:

No extra data needed: The model only uses features from the original training and test datasets — no external data or complex joins required.

Missing features? That's okay: You don't need to send every feature. If some values are missing, the model can still make a prediction using built-in data cleaning and handling steps.

Send a simple JSON: Just send the features you have in a JSON format to the API, and you'll get a prediction in return. It's quick and easy to test, even with partial data.

The API is live and ready. You can test it by sending your own data – whether it's complete or not – and see how the model performs. It's designed to be smooth, simple, and ready for real-world use.

Give It a try: https://home-credit-optuna-model-982383075824.us-central1.run.app/docs


**Using the Model with FastAPI (Swagger UI)**

You can test the model easily in your browser – no coding needed. Just follow these steps:

1. Click on predict (This opens the model's prediction endpoint.)

2. Click Try it out (This lets you enter your input data.)

3. Paste your JSON and hit Execute (The model will process the input and return a prediction.)

4. See the result (You'll get the prediction and the probability showing the risk of default.)

## Monitoring and Maintenance

Building and deploying the model is just the beginning. To keep it performing well in the real world, we need to plan for what happens after deployment.

Why Monitoring Matters? Over time, user behavior, economic conditions, and data sources can change. This is known as data drift. Even the best model can slowly become outdated if it's not monitored and retrained regularly.

How We Monitor the Model?

- Track live predictions: We log inputs and model outputs to see how it behaves in real time.
- Check prediction quality over time: We compare predictions to actual outcomes (if available later) to measure things like accuracy, AUC, and recall as they evolve.
- Watch for unusual input patterns: If the data coming in starts to look very different from the training data, that's a sign we might need to retrain or adjust.
- Set alerts for performance drops: Automated alerts can warn us if performance metrics fall below a set threshold.

Retraining the Model

We plan to retrain the model regularly using the latest data:

- New customer records can be added to the training dataset.
- The same pipeline can be reused thanks to its modular design.
- Retraining can be automated to fit business schedules (e.g., monthly or quarterly).

## In Summary

This model isn't just accurate — it's maintainable. We've set it up to be monitored, retrained, and improved over time, ensuring it stays useful and reliable as the world changes.