Credit Default Prediction: A Comparative Analysis of Machine Learning Models

---

title: "Credit Default Prediction: A Comparative Analysis of Machine Learning Models" author: "Asia Di Girolamo" date: "January 11, 2026" geometry: margin=2.5cm output: pdf_document

1. Abstract

Financial institutions face significant risks related to borrower default. Accurately predicting whether a customer will fail to repay a loan is critical for minimizing capital losses and maintaining economic stability. This project investigates the efficacy of various Machine Learning algorithms in predicting credit default risk. By implementing a robust data processing pipeline that includes feature scaling, categorical encoding, and stratified sampling, we compared the performance of six distinct models: Logistic Regression, K-Nearest Neighbors (KNN), Decision Trees, Random Forest, XGBoost, and CatBoost.

The experimental results demonstrate that **CatBoost** outperforms the other models, achieving an Accuracy of **77.5%** and a Receiver Operating Characteristic Area Under Curve (ROC-AUC) score of **0.814**. The study highlights the superiority of Gradient Boosting techniques over traditional linear models and single decision trees for complex financial data. Furthermore, we discuss the technical challenges related to library compatibility in production environments and the importance of reproducibility in ML pipelines.

Keywords: Credit Risk, Machine Learning, CatBoost, Gradient Boosting, Financial Prediction, Python.

\newpage

Table of Contents

\newpage

2. Introduction

Background and Motivation

Credit default risk—the risk that a lender takes on in the chance that a borrower will be unable to make required payments on their debt obligations—is one of the most fundamental challenges in the banking and financial services industry. Traditional methods of credit scoring often rely on linear statistical techniques or manual rule-based systems, which may fail to capture complex, non-linear relationships between a borrower's demographic data, financial history, and their likelihood of default. In the era of Big Data, Machine Learning (ML) offers powerful tools to automate and improve the accuracy of these predictions. By analyzing historical data, ML models can identify subtle patterns that human analysts might miss, leading to more informed lending decisions.

Problem Statement

The core problem addressed in this project is a **binary classification task**: given a set of features describing a loan applicant (e.g., age, income, loan amount, repayment status), the goal is to predict the target variable Y, where $Y=1$ represents a "Default" (failure to repay) and $Y=0$ represents "No Default" (successful repayment).

Objectives and Goals

The primary objectives of this study are:

1. To develop a reproducible Machine Learning pipeline for data preprocessing and model training.
2. To implement and tune multiple classification algorithms, ranging from interpretable baselines (Decision Trees, KNN) to advanced ensemble methods (Random Forest, XGBoost, CatBoost).
3. To rigorously evaluate these models using appropriate metrics for imbalanced datasets, specifically focusing on **F1-Score** and **ROC-AUC** in addition to standard Accuracy.
4. To identify the best-performing model for deployment in a hypothetical production environment.

3. Research Question and Relevant Literature

Research question

How can we design and evaluate a credit default prediction pipeline that produces accurate *and* decision-useful probabilities under class imbalance and asymmetric costs?

This question is different from "maximize accuracy." In credit scoring, the operational objective is often to minimize expected losses or maximize expected profit subject to constraints. That requires:

1. **Ranking quality** (e.g., ROC AUC),
2. **Performance on the minority class** (e.g., Precision–Recall AUC),
3. **Probability quality** (calibration, Brier score),
4. **Decision quality** under a business cost matrix (cost-sensitive thresholding).

Relevant literature (high-level)

- **Imbalanced classification**: PR-AUC is often more informative than ROC-AUC when the positive class is rare because it focuses on precision/recall trade-offs for the minority class.
- **Probability calibration**: many strong classifiers (especially tree ensembles) produce poorly calibrated probabilities; calibration methods like **Platt scaling (sigmoid)** and **isotonic regression** can improve probabilistic interpretability.
- **Cost-sensitive learning and thresholding**: when costs differ between FP and FN, the optimal threshold generally differs from 0.5, and should be selected by minimizing expected cost on validation data.
- **SMOTE vs class weighting**: oversampling (SMOTE) and cost-sensitive weighting are two common strategies to address imbalance; which one works better is data-dependent and should be tested empirically.

This project follows these principles by implementing both calibration and cost-based threshold selection, and by explicitly comparing imbalance-handling methods.

4. Methodology / Algorithm and Complexity

4.1 Data Description

We work with a standard credit risk tabular dataset containing both numerical and categorical variables (e.g., loan duration, credit amount, and borrower characteristics). The target is a binary default label.

Preprocessing is implemented via a scikit-learn style pipeline:

- **Missing values**: handled within the preprocessing flow (imputation where needed).
- **Categorical features**: one-hot encoding.

- **Numerical features**: scaling (StandardScaler) to support linear models and stabilize training.
- The transformation is fit on training data only to avoid leakage.

Data quality assessment revealed no significant missing values, but standardization was required for distance-based algorithms like KNN.

4.2 Models

We compare a diverse set of models:

1. **Logistic Regression**: strong linear baseline; outputs naturally probabilistic scores.
2. **Random Forest**: non-linear model; often strong ranking but can be miscalibrated.
3. **Gradient boosting family**:
    - **XGBoost**
    - **LightGBM**
    - **CatBoost**
4. **Interpretable model**: **EBM/GAM-style** model (Explainable Boosting), included to explore the accuracy–interpretability trade-off.

Where relevant, we use **class weights** to address imbalance.

4.3 Evaluation framework and metrics

We adopt:

- **Train/test split** for final reporting.
- **Stratified K-Fold cross-validation** (10 folds) for stable estimates and to reduce variance in results.
- Metrics:
    - **Accuracy** (reported, but not the main objective under imbalance),
    - **ROC AUC**,
    - **PR AUC**,
    - **Brier Score** for calibration quality,
    - **Total expected cost** from a custom cost matrix.Implementation

The project was implemented using **Python 3.11** within a managed **Conda** environment to ensure dependency isolation. The project structure follows industry best practices:

- `environment.yml`: Defines the exact versions of libraries used (pandas, scikit-learn, catboost, xgboost) to prevent "it works on my machine" issues.
- `src/`: Contains modular code.
    - `data_loader.py`: Handles data ingestion and cleaning.
    - `models.py`: Defines the model architectures.
    - `evaluation.py`: Computes metrics and generates plots.
- `main.py`: The entry point script that orchestrates the entire pipeline.

Code Snippet: Model Training Wrapper

```
def train_model(model_class, X_train, y_train, **kwargs):
    """
    Generic function to train different models ensuring
```

```
    random_state consistency.
    """
    if 'random_state' in kwargs:
        model = model_class(**kwargs)
    else:
        model = model_class(random_state=42, **kwargs)

    model.fit(X_train, y_train)
    return model
```

4.4 Cost-sensitive objective
We define asymmetric costs consistent with credit risk intuition:

- Cost(FN) = 5: predicting “good” when the borrower defaults.
- Cost(FP) = 1: predicting “bad” when the borrower would not default.

Let the confusion matrix be ordered as \[\[TN, FP\], \[FN, TP\]\].
Then the total cost is:

$$
\text{Cost} = 5 \cdot FN + 1 \cdot FP.
$$

This metric is used to rank models and to optimize the decision threshold.

4.5 Complexity (qualitative)
- Logistic Regression: roughly linear in number of samples and features per iteration.
- Random Forest: scales with number of trees × depth × samples.
- Gradient boosting: scales with number of boosting iterations × tree complexity; often more expensive but high-performing.
- Calibration adds overhead proportional to the calibration method:
  - Sigmoid calibration is light (parametric).
  - Isotonic can be heavier and more prone to overfitting on small samples.

5. Implementation Discussion (Code and Performance)

5.1 Project structure and pipeline design
Implementation is provided in a Jupyter notebook that follows a clear sequence:

1. Data loading and initial inspection,
2. Preprocessing pipeline (ColumnTransformer),
3. Baseline supervised models,
4. Unsupervised analysis (PCA + clustering),
5. Advanced evaluation (CV + multiple metrics),
6. Calibration experiments,
7. Final evaluation and model comparison,
8. Cost-sensitive threshold optimization,
9. Robustness checks: class weighting vs SMOTE.

A key design choice is to keep preprocessing and modeling inside consistent pipelines to avoid leakage and ensure repeatability.
5.2 Unsupervised analysis (diagnostic step)

```
Before finalizing supervised learning, we use PCA and clustering as a diagnostic tool to check whether there is visible structure in the feature space and whether default/non-default observations appear separable. This does not replace supervised evaluation, but provides intuition about the dataset and potential non-linearities.

### 5.3 Calibration experiments
We explicitly test:

- **Sigmoid (Platt scaling)** vs **Isotonic regression** calibration.
- **Binning effect**: how the number of bins in calibration plots changes perceived “jaggedness.”
- **Regularization** as an indirect calibration improvement (more conservative models can output less extreme, more stable probabilities).

In practice, sigmoid calibration is often safer when the calibration set is not large because isotonic can overfit and produce unstable curves.

### 5.4 Performance considerations
Training time is manageable on a laptop-scale environment. For ensemble models, runtime is driven by number of estimators and depth. Where supported by libraries, multi-core execution can be enabled to speed up training and cross-validation.

## 6. Codebase Maintenance and Updating (Git, Testing, Reproducibility)

This section documents how the project is maintained and how results can be reproduced and extended.

### 6.1 Version control (Git)
- The repository is tracked with Git.
- Development is organized through incremental commits that reflect meaningful changes (pipeline fixes, report updates, evaluation additions).
- This makes the work auditable: it is possible to trace when a model, metric, or preprocessing choice changed and why.

### 6.2 Reproducibility
- **Random seeds** are fixed where possible (e.g., `random_state=42`) for train/test splits, CV, and stochastic models.
- Dependencies are managed through a Python environment (conda/venv). A `requirements.txt` or environment file should be kept aligned with the notebook imports.
- The notebook is written to run top-to-bottom without hidden state.

### 6.3 Testing and safe refactoring (recommended practice)
While notebooks are often exploratory, production-quality maintenance benefits from lightweight testing:
- Utility functions (e.g., cost computation, threshold search) can be moved into a `.py` module and tested with simple unit tests (expected confusion matrix → expected cost).
- Continuous checks (even minimal) reduce the risk of silent evaluation bugs when iterating on the pipeline.

### 6.4 How to update/extend the project
Typical extensions that fit cleanly into this codebase:
```

- Add new models (e.g., calibrated linear SVM, neural nets) by plugging into the same evaluation loop.
- Change the cost matrix to reflect a different business environment and recompute optimal thresholds.
- Add explainability (feature importance, SHAP) while preserving the same preprocessing pipeline.

7. Results

7.1 Main comparison (calibrated + regularized setting)
Models are evaluated on multiple criteria: discrimination (ROC/PR), calibration (Brier), and decision cost.

A representative summary (sorted by cost) is:

- **CatBoost**: best total cost (≈167) with strong ROC AUC (≈0.817), PR AUC (≈0.683), and good Brier score (≈0.150).
- **EBM/GAM**: competitive cost (≈170) with interpretability benefits.
- **Logistic Regression**: solid baseline, competitive ROC and PR, slightly worse cost than the best model.
- **Random Forest / XGBoost / LightGBM**: strong ranking performance but not always best under the cost function.

Interpretation:
Even when several models have similar ROC AUC, the **expected cost** can differ because cost depends on FP/FN trade-offs and the chosen threshold. Therefore, model selection based on cost is more aligned with credit risk decisions than selecting purely by AUC.

7.2 Calibration insights
Calibration plots show that:
- Uncalibrated ensemble models can produce probabilities that deviate from the diagonal (“perfect calibration”).
- Sigmoid calibration tends to produce smoother, more stable improvements.
- Isotonic calibration can be flexible but may become jagged, especially when the effective calibration sample is limited or when probabilities are concentrated in narrow ranges.

This justifies using calibration not as a cosmetic step, but as a way to make probability outputs decision-relevant.

7.3 Cost-sensitive threshold optimization
Using the defined cost matrix (FN cost much larger than FP cost), the default threshold of 0.5 is not necessarily optimal. We scan thresholds in $[0,1]$ and compute total cost to find:

$$
t^* = \arg\min_t \{ 5\cdot FN(t) + 1\cdot FP(t)\}.
$$

Key insight:
A lower threshold than 0.5 is often optimal when missing defaulters is expensive, because the model should flag more borderline cases as risky to reduce FN counts.

```
### 7.4 Robustness: class weighting vs SMOTE
We compare imbalance handling strategies on a representative model (Random
Forest):

- **Class weighting** produced a higher cost (≈184).
- **SMOTE** reduced the cost (≈174).

**Interpretation:**
For this dataset and model, generating synthetic minority samples helped the
classifier reduce costly false negatives more than weighting alone. However, SMOTE
can also introduce artifacts, so the conclusion is empirical and context-specific.
## 8. Conclusion

This project demonstrates that strong credit default prediction is not only about
training a high-AUC model, but about producing **decision-useful probabilities**
under realistic constraints.

Main takeaways:
1. **Cost-sensitive evaluation changes the ranking of models** compared to
accuracy or ROC AUC alone.
2. **Probability calibration matters**: calibrated probabilities are more
trustworthy for thresholding and risk interpretation.
3. **Threshold optimization is essential** when costs are asymmetric; 0.5 is
rarely optimal in credit risk.
4. Among tested models, **CatBoost** achieved the lowest expected cost while
preserving strong discrimination and calibration performance.
5. Robustness checks suggest that **SMOTE** can outperform simple class weighting
in terms of cost for certain models.
This project implemented a complete, reproducible pipeline for credit default
prediction with a strong focus on decision-relevant evaluation. The key results
are:

CatBoost is the best overall model in our setting, achieving the lowest financial
cost (167) and the strongest ROC/PR ranking performance.

Calibration matters: post-hoc calibration (especially sigmoid/Platt scaling)
improves the interpretability and usefulness of predicted probabilities for risk
decisions.

Cost-sensitive evaluation changes model selection and threshold choice: a model
with slightly lower AUC may still be preferable if it reduces expensive false
negatives under asymmetric costs.

Robustness checks are important: SMOTE can outperform class weighting for certain
models (as shown for Random Forest), highlighting the need to validate imbalance-
handling decisions rather than assuming one method is universally best.

Limitations and future work:
- Extend evaluation with confidence intervals (e.g., bootstrap) for metrics and
cost.
- Explore more systematic hyperparameter tuning (Bayesian optimization) under the
cost objective.
- Add interpretability layers (global + local explanations) and fairness
diagnostics if demographic variables exist.
```

- Validate calibration and threshold stability on a true out-of-time split to mimic real credit portfolio drift.

References

1. Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2018). CatBoost: unbiased boosting with categorical features. Advances in neural information processing systems, 31.
2. Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining.
3. Pedregosa, F., et al. (2011). Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825-2830.
4.Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.
5. Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science. (German Credit Data).
6.Zadrozny, B., & Elkan, C. (2002). Transforming classifier scores into accurate multiclass probability estimates. KDD.
7.Chawla, N. V., et al. (2002). SMOTE: Synthetic Minority Over-sampling Technique. JAIR.

Appendices

Appendix A: Project Structure

The full source code is available in the submitted repository with the following structure:

```
MLproject/
├── environment.yml      # Conda environment configuration
├── main.py              # Entry point for the application
├── README.md            # Setup and usage instructions
├── project_report.pdf   # This document
├── src/
│   ├── data_loader.py   # Data preprocessing logic
│   ├── models.py        # Model definitions
│   └── evaluation.py    # Metrics and plotting
├── data/
│   └── raw/             # Original dataset
└── results/             # Saved plots and metrics
```

Appendix B: Installation Instructions

To reproduce the results presented in this report:

1. Create the environment:

```
conda env create -f environment.yml
```

2. Activate the environment:

```
conda activate mlproject
```

3. Run the analysis:

```
python main.py
```

Appendix C: AI Assistance Statement

This project utilized AI tools (Antigravity, Gemini 3.0, ChatGPT) primarily for debugging and code optimization purposes. A detailed declaration of usage is available in the `AI_LOG.md` file included in the GitHub repository.