Loan Default Prediction for Consumer Lending

Team Members: Novoni Banerjee (nb887) and Saul Santiago (ss3840)

Project Definition

Predict whether a loan will default for consumer lending, given a loan application, by constructing a complete data pipeline that cleans, integrates, and transforms raw financial datasets into a unified view for machine learning.

The project's ultimate goal is to address the data management challenge, enabling financial institutions to accurately predict loan default risk by leveraging large financial datasets for informed, data-driven credit decision-making. The project is focused on the construction of a structured end-to-end pipeline that ingests raw loan datasets, validates them, and integrates borrower and loan attributes to produce a unified training view for machine learning. By doing this, we address a frequent challenge in typical credit-risk workflows caused by raw financial data not being immediately suitable for data analysis and predictions. We developed consistent schemas, accurate mappings, transformation logic, and feature preparation to create a stable foundation to allow the models to make dependable default predictions. Getting a loan is a very complex process, and while there are a lot of calculators online that can provide this information, there is no guarantee that a particular calculator uses real datasets to provide realistic information. This is where our project is unique; we aim to use real datasets provided by publicly available resources such as the Home Mortgage Disclosure Act and Kaggle.

Strategically, this project aligns with lecture topics on data collection, schema design, data cleaning, integration, transformation, and analytical preparation prior to modeling. We implemented a full ETL pipeline to ingest the raw datasets, stage them, validate them, and map them into our normalized database schema before producing a training view for our analysis. Our project directly integrates and applies topics from class on design, data lineage, and reproducible workflows. As the final step in our pipeline, the machine learning component demonstrates how predictive modeling is more reliable with well-managed data. Overall, in constructing our loan default prediction pipeline, we demonstrate how data engineering decisions shape the quality and scalability of the entire analytical process.

Introduction

Importance

Our project is important because we use real data provided by financial institutions to make our model training accurate. We don't just create some basic graphs with a lot of numbers on them; we draw an actual relationship between debt-to-income ratios, loan amounts requested, credit scores, and more. This makes the data easy to interpret, and users aren't left in the dark about how loan defaults are calculated; all of the functions and methods are public for anyone to access. This is where the novelty of our project lies; there isn't a math equation used to determine a basic yes or no answer, we use various realistic financial information to output an accurate answer. Our project relies heavily on public data, since without it, gathering loan information is not a realistic goal.

Existing Issues in Current Data Management Practices

1. **Data Quality:** While the Kaggle dataset is very rich in data, it isn't fully cleaned. There are missing values and redacted information that need to be manually filtered out or filled with reasonable methods.
2. **Private Information:** Loan data naturally requires private information like bank information, income, and social security numbers, which requires careful handling to ensure no sensitive info is leaked.
3. **Scalability**: Information is not consistent across every dataset, which requires efficient data management and processing methods that scale efficiently with input data.

Prior Related Works

There are prior related works on GitHub as open-source credit risk projects. They focus mostly on ML workflows with little to no database design or data management practices. They mostly use single-table datasets without implementing robust ETL pipelines and SQL schemas. The prior related works emphasize model accuracy over data lineage, transparency, or schema normalization. The limitations in these past projects emphasize how our project addresses those gaps by integrating full ETL processing, validation layers, normalized database design, and a complete ML pipeline within one entire system.

Methodology

Data Science ----------

Data Collection: What kind of data did you use?

At first, data sourcing was a challenge. Given that loans require private identifiable information, we had to make sure that all of our sources removed such data. The other challenge was finding sufficient datasets to train our model on. Originally, we had 3 datasets we were going to source from Kaggle, Datalab, and HDMA. A quick look at the data revealed that Datalab was redundant and not impactful to our analysis. Datalab had approximately 15000 entries, while Kaggle and HDMA individually had 1mil+ entries. Acquiring the Kaggle dataset was the hardest to do because the datasets were over 1gb in size. Attempts to compress this large dataset still resulted in .zip files of over 100mb, and Github has a limit of 100mb per file uploaded, so we had to come up with some other method to acquire the Kaggle dataset reliably. Another challenge was making an automated process so that the users of our code don't have to manually input commands in their terminal. This led us to look into the Kaggle API, which requires a unique token associated with every account.

Through this API, we were able to obtain .gzip files of the Kaggle data and extract them into .csv files. Extracting them led to the creation of helper functions that read these .csv files into dataframes so we can properly analyze, clean, and train our ML model. While the entire extraction process takes a minute or two, everything is done without any user input. From here, our functions automatically unzip the files, read them into a pandas dataframe, and then delete these files due to the Kaggle .csv and .gzip files no longer needed after loading.

HDMA files were a much simpler process. There is no need for private auth keys since this information is publicly available. HDMA has API documentation with a function that downloads all the data in a .csv file. While this file was also massive, parquet compression produced files that were under 100mb. The parquet files are left in our project files with helper functions that do the same as our Kaggle functions (reads the files and moves them into dataframes).

Database Component ----------

Data Storage:

Keeping everything in something like a Jupyter notebook was not an option. Memory would eventually run out, and the kernel would often crash when initial Jupyter-only testing was performed. Debugging proved to be inefficient and often required letting our devices run for many minutes before we could start creating fixes. In contrast, PostgreSQL is reliable for large datasets and supports structured schemas. It also works well with SQLAlchemy and Python and efficiently creates staging and final tables. SQL Queries proved to be much faster than processing with strictly Pandas. We could also perform complex filters without loading all of the data directly onto our memory. Jupyter notebooks, while efficient for local testing, would sometimes cause glitches in our system because of the memory it would use. Any accidental cell clearing, kernel restart or ending the work session would result in having to boot up everything all over again.

Staging Tables: We stored Kaggle and HDMA datasets after initial cleaning steps that were taken in our data ingestion files. It still contains data that was unvalidated and untransformed.

Validated Tables: By understanding the financial context of each attribute, the validated tables contain logically cleaned and filtered Kaggle and HDMA attributes. These tables store what is left after removing invalid records and inconsistent data types.

Final Normalized Tables: The main tables are Borrowers, Accepted Loans, and Rejected. The Borrowers table has borrower_id as a primary key and includes related attributes that appear in the datasets. The Accepted Loans table stores all successfully issued loan records along with a reference to borrower_id. The Rejected table contains basic information about unapproved loan applications. Additionally, every table has a created_at attribute because we wanted to maintain data lineage and traceability throughout the entire pipeline.

Data Integration:

Data Integration relied heavily on our staging tables.

KAGGLE

- Kaggle CSV data is downloaded using **get_kaggle_data()** and loaded using **retrieve_training_csv()**
- **kaggle_accepted_loans_df()** and **kaggle_rejected_loans_df()** are used to perform an initial cleaning of the data and select only relevant fields to move to the SQL staging tables
- **remove_outliers()** removed initial outliers in ‘loan_amnt’ are removed using a Z-score method with a threshold of **3**

HDMA

- HDMA data is read from parquet.gzip files using **read_hdma()**
- Any columns that hold similar data to the Kaggle dataframes are renamed (co-applicant_credit_score_type → co_applicant_credit_score_type)
- **clean_data()** is used to convert any generic datafields like a range of numbers into the midpoint to prepare for analysis
- Similar to Kaggle outliers were removed using a Z-score threshold of **3**

STAGING TABLES

- The cleaned up dataframes are loaded into staging tables in PostgreSQL database using **write_to_df()**

    - staging_accepted_kaggle, staging_accepted_hdma, staging_rejected_kaggle, staging_rejected_hdma
- Unique queries were written for each staging table to perform data validation and apply logical rules before moving data to their final tables
    - Constraints were kept consistent amongst the valid_accepted_* tables to maintain consistency ('loan_amnts' > 1000, income ranges between 0 & 3,000,000, etc)
    - valid_rejected_* tables were less restrictive since people can get rejected for multiple reasons

FINAL TABLES

Borrower Table

- Given that the Kaggle datasets have a lot of information, we kept a majority of the data such as "fico_range_low, fico_range_high, home_ownership" for future data usage
- **SELECT DISTINCT** ensures no duplicate borrower profiles are inserted into our table
- HDMA data is not as rich in contrast to Kaggle, the focus relied on numeric data to insert into the table (income, debt_to_income_ratio, applicant_credit_score_type, co_applicant_credit_score_type)

Accepted Loans Table

- Unique borrower profiles are extracted from both **valid_accepted_*** tables and inserted
- IDs like those from Kaggle are transferred over to the final table as primary keys
- Any fields that are too broad and exclusive to each datasource are explicitly set to **NULL::TYPE**
- Any duplicate entries are handled with **DO NOTHING** to safely return the script

Rejected Table

- Profiles are extracted from both **valid_rejected_*** tables and inserted
- Table entries were built around **valid_rejected_kaggle** due to the large amount of information it provides
- Any HDMA specific columns that don't overlap with Kaggle are set to **NULL::TYPE**

<u>Data Quality:</u>

While the datasets for both Kaggle and HDMA were rich in entries, they still had null entries, outliers, and broad value entries. Each dataset was separated into source_raw_accepted_loans and source_raw_rejected_loans for ease of data manipulation. The accepted vs raw datasets have different formats that prevent us from easily recycling cleaning methods. Each data set required custom cleaning methods to prepare the dataframes for database entry. For example Kaggle and HDMA had debt to income ratios sometimes listed with a general range ("33-39%"). Entries like this had to be cleaned up as a string and remove any special characters like '-' or '%', temporarily turn any ranges into two integer values, and for ease of calculation we used the midpoints of the ranges provided. Examples of other issues amongst the data were missing incomes, negative incomes (in the case of rejected loans), and unneeded columns. All of these issues were addressed in the custom functions.

While the datasets for both Kaggle and HDMA were rich in entries, they still had null entries, outliers, and broad value entries. Each dataset was separated into source_raw_accepted_loans and source_raw_rejected_loans for ease of data manipulation. The accepted vs raw datasets have different formats that prevent us from easily recycling cleaning methods. Each data set required custom cleaning methods to prepare the dataframes for database entry. For example, Kaggle and HDMA had debt-to-income ratios sometimes listed with a general range ("33-39%"). Entries like this had to be

cleaned up as a string and remove any special characters like '-' or '%', temporarily turn any ranges into two integer values, and, for ease of calculation, we used the midpoints of the ranges provided. Examples of other issues amongst the data were missing incomes, negative incomes (in the case of rejected loans), and unneeded columns. All of these issues were addressed in the custom functions.

We created a data validation layer to ensure that all attributes from the staging tables satisfied the financial context of each attribute. We applied filter rules and removed malformed entries, emphasized valid numeric ranges, standardized formats, and dropped records with missing information. Each validated table was created with a valid_ prefix. This step made the pipeline easier to debug and more resilient to inconsistencies among HDMA and Kaggle datasets.

Data Transformation:

After the validation step, we mapped fields from the Kaggle and HDMA fields in the validated tables into the final normalized tables. Here, we enforced correct data types, standardized attributes, and aligned categorical attributes. We also mapped HDMA fields into Kaggle fields, where possible, to reduce repeated attributes. In transformation, we also had to generate surrogate borrower_id and application_id fields since our datasets didn't provide them. In addition, we normalized numeric fields (e.g.: debt-to-income ratio, interest rate) into consistent formats. In general, in the transformation step, we ensured the database was clean and optimized before creating the ML training view.

ETL Pipeline:

The ETL Pipeline is structured to convert raw HDMA and Kaggle datasets into validated and fully integrated SQL tables, to minimize preprocessing for machine learning. Each stage of extraction, validation, and transformation is handled by their own python module to ensure each stage works independently, to aid with maintenance and debugging. The pipeline automates the process of ingesting raw data, cleaning, staging, validating, and mapping all attributes into a normalized relational schema. ETL pipeline is implemented by 4 ''.py files.

run_etl.py — Master script

- The file that executes the entire ETL process. Automates the creation of SQL database using postgreSQL and psycopg2
- Calls all helper functions needed to go from no data to final tables ready for ML training
- Ensures the pipeline runs from start to finish, getting raw data all the way to finalized tables to be used for ML training

staging_loader.py:

- Calls helper functions from our ETL/Ingestion directory that loads all the raw Kaggle and HDMA data into usable dataframes
    - Calls Kaggle API and HDMA API
- Helper methods here were made to initialize SQL tables
- Loads the dataframes into SQL staging tables to prepare for further data validation

validation_loader.py:

- Contains the bulk of all the filtering performed on our SQL tables
- Runs filtering queries on each of the staging tables
- Creates unique tables with the prefix 'valid' so that we can refer to these cleaned tables for further database development
- Each cleaning function is dedicated to one particular table to narrow any errors that may arise

- Separate validation steps makes debugging and maintenance significantly easier

transf_loader.py:

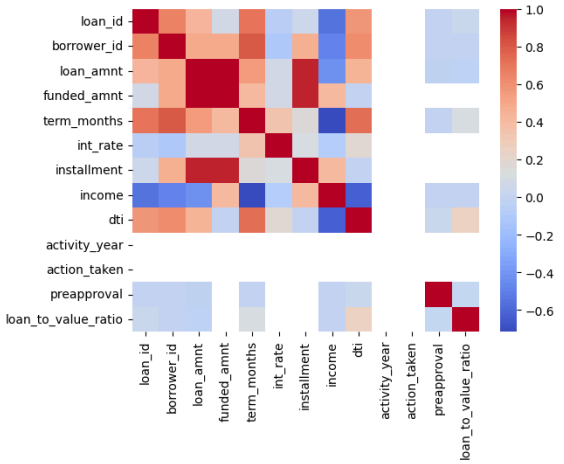
- Contains the final helper function calls that creates the finalized data tables called before creating our model
- Calls functions from files that are found in ETL/load/mapping/ directory
    - Each mapping function is dedicated to a specific table (accepted_loans, rejected_loans, borrowers)
    - These files help narrow down errors to specific queries instead of having everything being done in one massive file
- Final normalization, key mapping, and schema alignment are performed

Preliminary Data Analysis/Visualization:

Hypothesis

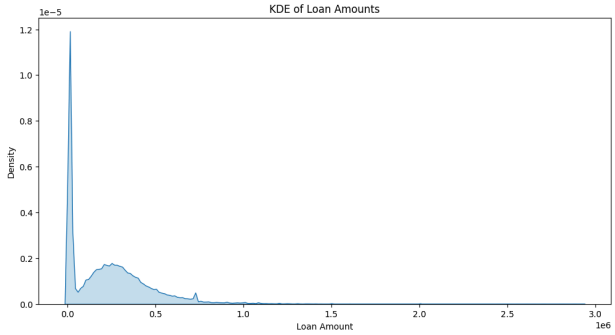
Determine the influence of specific values on predicting loan defaults. Initial hypothesis expects term months, interest rates, income and debt-to-income ratios to play a strong factor in determining loan default, as well as strong connection between loan amount requested and loan amount funded.

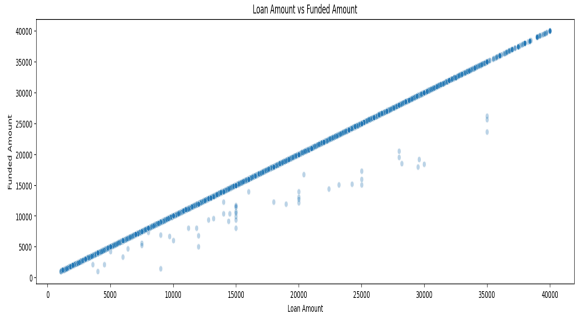
An initial heat map of the final **accepted loans table** was made using a heatmap (red means a positive correlation, blue means negative). While it doesn't explicitly tell us the importance of every column, this heat map is still helpful to guide the selection process for the datafields to be used in our ML model according to the relationships between 2 fields.



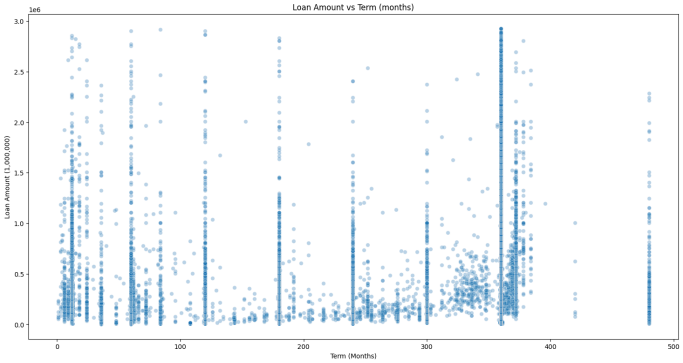
The first entry we explored was the loan amounts. Given that there was a variety of loans approved, we wanted to narrow down the range of loan amounts. The simplest graph to show this was a Density plot measuring where the majority of approved amounts lied in.

This plot was used as an initial filter for our model data. We can see here that the majority of our data lies around the 50,000/60,000 **loan amounts,** where the biggest spike can be seen. The second largest spike is roughly around 250,000, where it then trails off. Further loan amounts are extreme outliers and were removed from the dataset.

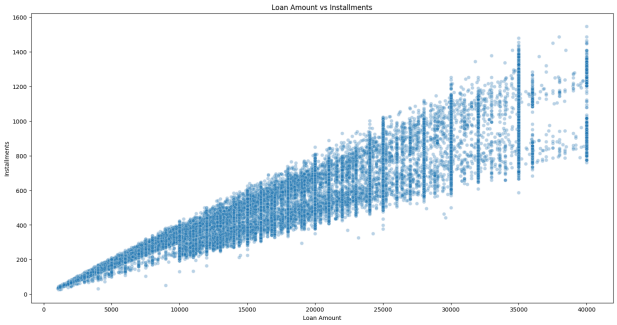




This graph represented the relationship between the **Loan amount and Funded Amount**. This shows an almost linear relationship with very few data points falling below the overall trend. Since the Loan and Funded amounts were almost the exact same, we settled on only using the Loan amount to avoid repeat data.

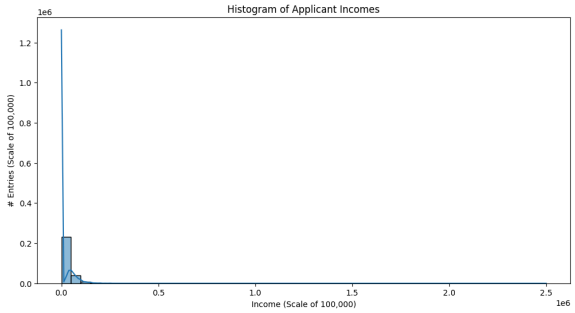


This scatter plot showed where the majority of **terms in months** lay (between 300 and 400 months). Very few loans were approved past the 400 months mark, so our model needed to learn the specific circumstances that allowed for longer terms to be approved.



Installments in relation to loan amounts showed a strong positive correlation (as expected). Installments by themselves are not very informative; however, pairing them up with other data like term in months, income, debt-to-income, etc. highlights the difference between successful and defaulting borrowers.

This histogram helped us further narrow down what the average borrower looks like. Most of our incomes fall in the left hand side below 250,000. The income outliers pushed our graph almost to 3,000,000 and we removed any income that fell outside the 99th percentile (220,000).



Machine Learning Component ----------

Data Analysis: ML Training View for Accepted Loans

Our ML component addresses the goal of predicting whether a loan application will default or not, given a loan application. I identified loan_status fields that are bad/defaulted (0), good/not defaulted (1), and neither for my ML-model. Bad fields included Charged Off, Default, and Late (31-120 Days). Good fields included Fully Paid and Current. Fields that were unclear and between the two options were not used. I then created an SQL view that selected numeric and categorical attributes that were the most

concise and efficient at influencing loan default prediction for consumer lending. This view served as an analysis-ready dataset to allow the ML models in the next steps to be better at learning patterns associated with both default and non-default outcomes.

What models/techniques/algorithms did you use or develop?

Data Preprocessing: Before the modeling step, I checked my ML training view for missing values to ensure consistency, and did not blindly trust my ETL. I confirmed that my training view did not require additional cleaning because the ETL already completely handled data validation, normalization, mapping, and dropping bad records. Thus, I focused on standardizing and scaling the numerical features and applying one-hot-encoding to categorical variables.

Feature Selection: I selected features based on consistency and relevance. I dropped the HDMA only rows because they would not have values for Kaggle rows. I focused on keeping features that directly had an effect on predicting defaults. The features I chose were loan amount, installment, debt-to-income ratio, income, and term in months. Together, these features represent the loan structure, the borrower's ability to repay, and the financial context.

Logistic Regression: I used LR as a baseline model because it is simple, fast, and often well-suited for binary classification problems like default prediction. It established clear decision boundaries and explainable coefficients, to help set the foundation to compare with more complex models down the line. It also helped highlight the strengths and weaknesses of the linear assumptions in the imbalanced datasets.

Random Forest Classifier: I used it as the second model after LR as a non-linear ensemble method to find complicated interaction patterns. By combining several decision trees, it reduces variance and improves robustness.

Gradient Boosting: (HistGradientBoostingClassifier) This was the final model used, and it was the most flexible model. It is capable of learning complex patterns through sequential tree boosting.

What experiments did you design?

I designed class-weight adjustment experiments and threshold tuning experiments to improve my model performance. For the Gradient Boosting model, I experimented with threshold tuning to try to see if I could adjust the threshold to increase the F1 score to improve the model's ability to detect default risk. I found that 0.5 was the threshold that gave the best F1 score. I finetuned the class weights to emphasize the minority class, which significantly improved the recall percentage. Finally, I consolidated my results and metrics across the three models into a comparison table to easily identify which model is best at maximizing loan default detection.

Results

Table 1: Comparison Between Logistic Regression, Random Forest, and Gradient Boosting Models

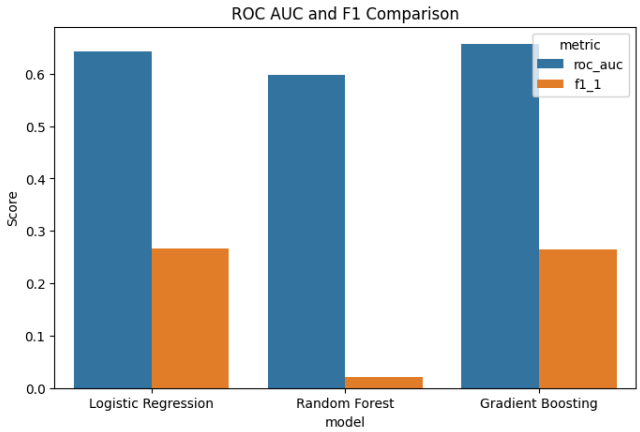
	model	accuracy	precision_1	recall_1	f1_1	roc_auc
0	Logistic Regression	0.6581	0.1819	0.5027	0.2671	0.6419
1	Random Forest	0.8733	0.2472	0.0109	0.0209	0.5979
2	Gradient Boosting	0.4593	0.1596	0.7885	0.2655	0.6568

Among the models we compared, the Gradient Boosting Model is the best model at identifying default loans with a 79% chance of catching all defaults. This is the most important metric for our project because our focus is on predicting default risk. Additionally, the ROC-AUC = 0.6568, which is not the

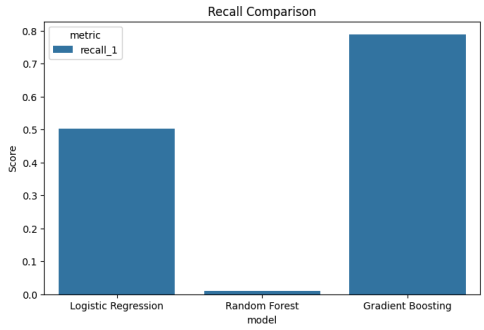
best, but it is still the highest for all of the models. Some weaknesses for this model include the low accuracy, low precision, and F1 = 0.265, which is similar to the Logistic Regression model. In the case of predicting credit risk, high recall is much more important than high accuracy because the cost of a false negative is more than a false positive. For our model, accuracy metrics are misleading in imbalanced datasets that have significantly less defaults than non-defaults. Thus, the Gradient Boosting Model is the most ideal among the three for catching as many potential defaulters as possible.

Data Visualization (ML Model):

This helps visualize the difference in performance amongst our models. Logistic Regression and Gradient Boosting have close ROC-AUC scores. Both models have the same ranking quality; they distinguish positives from negatives almost equally well.



Random Forest has the lowest F1 score, making it the model with the worst performance when identifying positive cases.



This helps visualize why our Gradient Boosting model was the best choice. Random Forest has the worst default prediction rates at 1.09%. Gradient Boosting is about 57% more successful than the Logistic Regression model. Predicting a true default is more important than raising a false positive flag.

Hypothesis Evaluation:

Our hypothesis expected term months, interest rates, income, and debt-to-income ratios to play a strong factor in determining loan default, along with a strong connection between the loan amount requested and the loan amount funded. The results partially verified this hypothesis, with debt-to-income ratio, income, and installment amount being critical in predicting defaults. Loan terms contributed less than we expected. Our methods were evaluated using accuracy, precision, recall, F1-score, and ROC-AUC metrics. This allowed us to compare the Logistic Regression, Random Forest, and Gradient Boosting Model objectively by measuring their ability to detect default risk. We also included experiments of adjusting weight classes and probability threshold tuning to further better the models. In general, the experiments and metrics helped us create a well-rounded assessment of the model's performance.

Advantages and Limitations:

The biggest advantage of our approach was our robust and reproducible ETL pipeline, which guaranteed validated and normalized data, reducing preprocessing time for any modeling that occurs. The

modular architecture with separated staging, validation, transformation, and mapping steps also made debugging significantly easier. Additionally, our ML training view isolated only the most meaningful attributes to reduce overfitting and simplify the modeling process.

Some limitations include our ML model only relying on accepted loan attributes due to the HDMA and Kaggle datasets being incompatible and unable to be fully integrated at the borrower level. This reduced the diversity of the features we had planned to include. Also, the dataset is imbalanced, with very few defaults in comparison to non-defaults, which increased false positives in our ML models.

Changes After Proposal:

Our final project had several adjustments that needed to be made from the initial proposal. We initially planned to use Datalab and Kaggle for data collection, but instead, we used HDMA and Kaggle because it was more than sufficient. Also, in our final ML model, we had to remove all HDMA-only attributes to get rid of attributes with excessive missing values. Also, we didn't implement real-time scoring APIs and batch-scoring pipelines due to time constraints. We also felt it was more important to spend that time developing a comprehensive ETL pipeline instead. Additionally, instead of training multiple models with engineered payment-history features, we simplified the ML stage to focus on three core classifiers and a training view. We also didn't implement feature engineering with multiple tables due to bottlenecks of schema mismatches and missing cross-dataset keys, which required us to redesign our pipeline around validated and transformed accepted loan records instead.

Contributions:

Novoni Banerjee (nb887) worked on creating the normalized final database setup, staging table setup, loading the staging table with the initially cleaned data from the data ingestion files, mapping each individual table from the validation table into the final database, creating the ML training view, and training the ML models.

Saul Santiago (ss3840) worked on integrating API's to obtain the raw data, cleaning the raw datasets, writing the ingestion python files, creating the validation layer to contextually clean the data, creating data visualization plots for prediction analysis, assisting in feature selection, and creating final data visualization plots for the final ML model.

Both worked on writing the project proposal, interim report, final project report, and recording the demo video. Both debugged and reviewed one another's additions to the project periodically.

References:

Github Repo: https://github.com/novoni01/credit-risk-assessment

Code Walkthrough Video:
https://drive.google.com/file/d/1ootiHriOYCQZ2gjCTfHE6H9mcJWuPID0/view?usp=sharing

Other:

- https://www.kaggle.com/datasets/wordsforthewise/lending-club
- https://www.datacamp.com/datalab/datasets/dataset-python-loans
- https://ffiec.cfpb.gov/data-publication/snapshot-national-loan-level-dataset/2023