Efficient Credit Risk Modeling: Enhancing Scalability and Performance

Terin Ambat
tambat@ucsd.edu

Zafeer Syed
zsyed@ucsd.edu

Rod Albuyeh
ralbuyeh@ucsd.edu

Abstract

Our study focuses on enhancing the prediction accuracy of loan default likelihood beyond the traditional credit score model. We aim to develop an advanced machine learning model using the Freddie Mac Single Family Loan-level dataset. This model will integrate credit score with other dataset attributes to create a robust method for predicting credit loan defaults.Our approach includes optimizing runtime performance, scalability, and code maintainability. We plan to surpass the predictive accuracy of existing benchmarks while building a machine learning pipeline that employs distributed computing techniques, and Python best practices for code efficiency and readability. Key steps involve training our model using credit score and other dataset features, establishing an efficient pipeline, and exploring the dataset for potential feature engineering opportunities. By adopting various model architectures and leveraging distributed libraries and frameworks, our implementation aims to parallelize data processing and model training, enhancing overall system performance. This research is significant for its potential to provide a more accurate, efficient, and scalable solution for predicting credit loan defaults, benefiting financial institutions and borrowers alike.

Code: https://github.com/prod-lab/RiskNet

1 Introduction 2
2 Methods 2
3 Results 4
4 Discussion 5
5 Conclusion 6
6 Figures 7
References 9

1 Introduction

As we develop our model to improve upon FICO's credit score for predicting loan default, we will explore distributed libraries such as Dask, Ray, and Modin to improve the performance of our model training. Additionally, we intend to further scale up our performance by running our code on UCSD's dedicated Data Science and Machine Learning Platform (DSMLP) servers. Furthermore, we expect to document the effectiveness of these changes and compare it to the results of the legacy repository by utilizing pipelines to create an efficient process of managing this project as well as making the process reproducible to others. In this project, will be focusing on enhancing an XGBoost model with Bayesian Optimization. Additionally, we plan to reorganize the codebase with a more efficient pipeline structure by utilizing efficient packages such as parquet, as well as making it a python PIP package. We will also use distributed systems instead of just pandas to improve upon the end-to-end runtime performance. The dataset we will be using for this project is the Freddie Mac Single-Family Loan-Level Dataset. Freddie Mac is a loan company that specializes in mortgage loans. This dataset contains anonymized loan information of loan borrowers along with their subsequent payments from 1999 to 2023. Some of this information includes, credit score, loan term, loan purpose, and property type. This information can be useful for developing models to use this information as feature inputs. Additionally, these loan details can be used to selectively input relevant information into our model that may be beneficial for predicting loan default and ignoring irrelevant statistics

2 Methods

For our initial steps, we began by looking into the Freddie Mac Loan Dataset and performing some exploratory data analysis to familiarize ourselves with the dataset. Then, our group members individually implemented a machine learning pipeline with an XGBoost model in order to predict whether or not a borrower will default on their loan. Because the Freddie Mac Loan dataset is quite large, we decided to initially focus on a small subset of the data from quarter 1 of 2009. We also set FICO's credit score as a baseline comparison, and aimed to beat its accuracy on predicting credit loan default (while also using the FICO credit score as a feature in our model). Because everyone in our group initially implemented the baseline XGBoost model in an individual Jupyter Notebook, we then decided to migrate and merge our works into a single repository, while also making our codebase more structured and modular (So that we would not be confined to running our code in a Jupyter Notebook environment). Before further improving on and optimizing our (so far) simple machine learning pipeline, we also decided to make our code easily reusable and reproducible, so we reformatted our code into structured modules and pushed our package to PYPI, making it easy to distribute and install via pip.

Our next aim was to optimize our pipeline's runtime performance on a single machine, with the eventual goal of moving our pipeline to a cluster so that it can run across multiple nodes. In order to do this, we began to look into various distributed libraries and frameworks to

help us achieve our goal of improving our runtime performance. We looked into Modin, Ray, and Dask as potential pandas replacements and spent time testing and assessing the benefits and drawbacks of each. While Modin was in theory the simplest, as a drop in pandas replacement and would require the least amount of change to our current codebase, we had issues with setting it up and successfully running on our codebase. We also looked into Ray and Dask, and found various potential benefits in terms of performance, ease of use, and scalability. We ultimately decided to settle on using Dask Dataframe as our pandas replacement, and planned to use it to work with larger datasets and parallelize our pandas code. Additionally, we also decided to use ray to improve performance and scalability of other aspects of our python codebase, namely using ray core api to parallelize for-loops. As of right now, all of our testing,development, and benchmarking is done in the context of a single machine. Additionally, in order to speed up potential bottlenecks in terms of reading and writing large sets of data to disk, we decided to use Parquet as our file format of choice, which was faster and more space-efficient than our initial serialization method of using python's pickle library. Also, we incorporated feature engineering into our pipeline in order to minimize the amount features the model needed to train on, while maintaining the knowledge of default likelihood. Finally, we also decided to move our development and testing environment to DSMLP, because we decided that it would provide us with a more powerful and consistent platform to develop and run our code (as opposed to everyone in our group developing personal machines of varying operating systems and architectures).

We attempted to utilize cloud platforms to scale our pipeline in order to increase runtime performance and so that we could train our model on larger amounts of data. However, we faced various logistical issues when attempting to select and use a cloud platform. We initially considered using AWS as our cloud platform of choice, but we were unable to receive a sufficient amount of credits in order to be productive on AWS. Consequently, we attempted to run our codebase distributively with DSMLP utilizing kubernetes, but experienced various issues with setting up our framework. Creating kubernetes clusters easily required various administrative tools such as kubeadm that would require elevated permissions from DSMLP to install and utilize. Additionally, due to the nature of DSMLP's container volatility, kubenetes clusters would not be able to run constantly and would have to be re-initialized at every session leading to unnecessary downtime. Finally, attempting to fix theses problems would lead to us focusing more on developer operations orchestration rather than trying to optimizing and paralleling loan prediction with our codebase.

Due to these issues, we decided to hone in on optimizing our pipeline's single machine runtime performance. As a result, we will continue to use parallel frameworks such as Ray and Dask to parallelize our pipeline as well as leverage the Ray Tune library to improve our training and tuning performance and speed. We also attempted to increase the size of our training dataset for our model. In order to do this, we leveraged a python script to automatically fetch quarterly or yearly data from Freddie Mac's website, making the process faster and more efficient.

3 Results

When running our entire data preprocessing and XGBoost model pipeline on individual computers through Jupyter Notebooks, the overall runtime was much slower and varied than compared to running it on DSMLP. Therefore, in order to simplify our results and make them more consistent, we ran all of our different model types on DSMLP. Credit score, our baseline FICO score model, took almost 2 minutes to run. Our original pipeline with no major changes took around 4 minutes to run. Our model converting the dataset to Parquet and using it as input, took around 15 minutes to run. Our model with both Parquet and feature engineering took around 33 minutes to run (Figure 2). This same model was also the pipeline, which resulted in the highest average precision of 0.38 (Figure 1).

By using the runtime of our model on DSMLP, we were able to generate a table to estimate the likely cost of running our pipeline on AWS servers (Figure 3). From the table, we can see the estimated costs for running RiskNet on various sized nodes on AWS are relatively small. However, it is important to note that the largest recorded node size is almost half the cost of the smallest node size. When we scale up our model with more worker nodes, it will be more cost effective to use the larger node sizes than the smaller nodes because the overall runtime will be faster per node.

Utilizing distributed libraries to replace Pandas had varying levels of success. While Modin was in theory the simplest, as a drop in Pandas replacement and would require the least amount of change to our current codebase, we had issues with setting it up and successfully running on our codebase. We used Dask Dataframe as our Pandas replacement, and used it to work with larger datasets and parallelize our Pandas code by converting the data preprocessing steps in our pipeline to Dask. Additionally, we were able to use Ray to improve performance and scalability of other aspects of our python codebase, namely using the ray core api to parallelize for-loops.

Despite using a script to fetch a larger volume of data for our model from Freddie Mac's website, our model did not see any meaningful improvements in terms of performance. However, this may potentially change if we further increase the size of our training dataset for our model. However, we are unable to further test this hypothesis due to memory constraints on the DSMLP environment.

As mentioned, we used Ray throughout our pipeline to parallelize operations like for loops, and replaced them with calls to the Ray Remote api. While the runtime performance on one quarter of data was relatively similar to our original runtime, as we add more and more training data to our model, the runtime increase will likely be more evident. While we also attempted to incorporate Ray Tune into our pipeline, in order to further speed up our runtime with distributed hyperparameter tuning. We ran into various issues with getting set up, many of which were related to the DSMLP environment. As a result, we would like to further continue to attempt to implement distributed hyperparameter tuning in teh future, as we believe that our issues may be resovled on an environment like AWS.

4 Discussion

Developing a consistent pipeline using Jupyter Notebooks was very difficult. It was impossible to ensure consistent versioning between computers, hard to run each section of the pipeline individually, and tough to work in collaboration with other group members. Regardless of the downsides, we were able to generate a model that beat the baseline FICO score without much tuning. Now our goal was to optimize this model to predict default even better. Part of this process would be to convert our code into a python package. This process initially seemed perplexing to setup, but ended up being quick to deploy and allowed our group to be able to collaboratively develop changes to the codebase.

With the codebase refactored in a package, we were able to explore various distributed packages, such as Modin, Ray, and Dask. We were not able to get Modin to work at all. This may have been due to our lack of experience with this library as it is relatively new compared to the others we explored. Additionally, the backend engine Modin uses runs on Dask or Ray (depending on user preference), therefore, any bugs that may occur with Modin could be due to errors in the engine processing that may be difficult to solve. It would be much easier to write our code using Dask and Ray directly to solve these potential problems. This is another reason why we decided to not use Modin in our pipeline. Ray, on the other hand, was able to be easily added to our codebase with the Ray Core API onto our for-loops with little configuration. Finally, Dask was a little tricky to get working and broke some of our pipeline, but our experience in developing Dask programs allowed us to handle these bugs with relative ease.

Getting our codebase running on DSMLP was a bit difficult as the setup process was unique and proprietary. However, once the proper server space was able to be allocated (8 CPU cores and 16GB of RAM), the codebase installation was simple as it was already in a PIP package. When we ran our codebase on DSMLP, we were surprised at how much faster it was compared to running it on our own computers. This may be due to the fact that on our own computers we were also running various tabs and programs that could have taken up CPU and Memory space instead of being used on running the pipeline as well as the inefficiency that might come up when running code through Jupyter Notebooks. Additionally it is possible that the individual cores on DSMLP are much more powerful than the standard desktop CPU cores that we used on our own computers. Nevertheless, this deployment on DSMLP showed how effective our codebase was on a consistent and repeatable system.

Accordingly, from Figure 1, the feature engineering model ended up doing the best with almost 4 times as much average precision as the baseline model. This clearly shows that the effort developed into adding feature engineering in the pipeline led to a significant improvement in model performance as the model actually ended up using only the most important features to train on.

5 Conclusion

In this study, we were able to develop a machine learning model that is able to predict loan default likelihood better than traditional credit models such as the FICO score. We made an end-to-end pipeline with data preprocessing, model training, and data visualization. Additionally, we developed a reproducible and installable PIP python package with understandable documentation, so that other individuals could easily start developing and improving upon our codebase. Finally, we explored multiple distributed libraries and created an effective distributed infrastructure for preprocessing data that allowed us to run our pipeline more efficiently. For our next steps, we would like to analyze our codebase to develop more distributed optimizations with Ray. Also, we would like to see how our pipeline would run on distributed cloud clusters of Amazon Web Services with a larger dataset size as input.

6 Figures

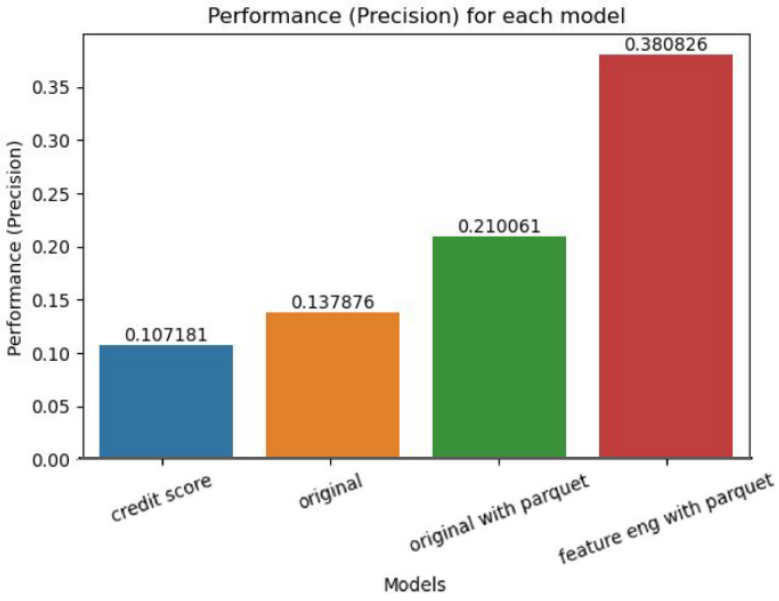

Figure 1: Average Precision for Each Model Type on DSMLP

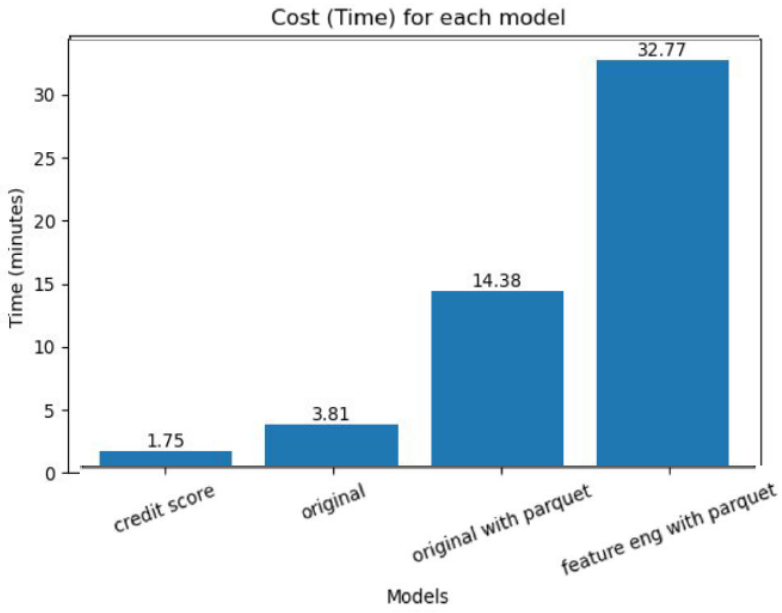

Figure 2: Runtimes for Each Model Type on DSMLP

CPU Cores	Memory	Runtime	Cost on EC2 Instance
4	16GB	56.73 minutes	$0.60
4	32GB	52.56 minutes	$0.56
8	16GB	36.63 minutes	$0.40
8	32GB	32.77 minutes	$0.36

Figure 3: Estimated AWS Costs per node sizes

References

Single Family Loan-Level Dataset. Freddie Mac. (n.d.). https://www.freddiemac.com/research/datasets/loanlevel-dataset