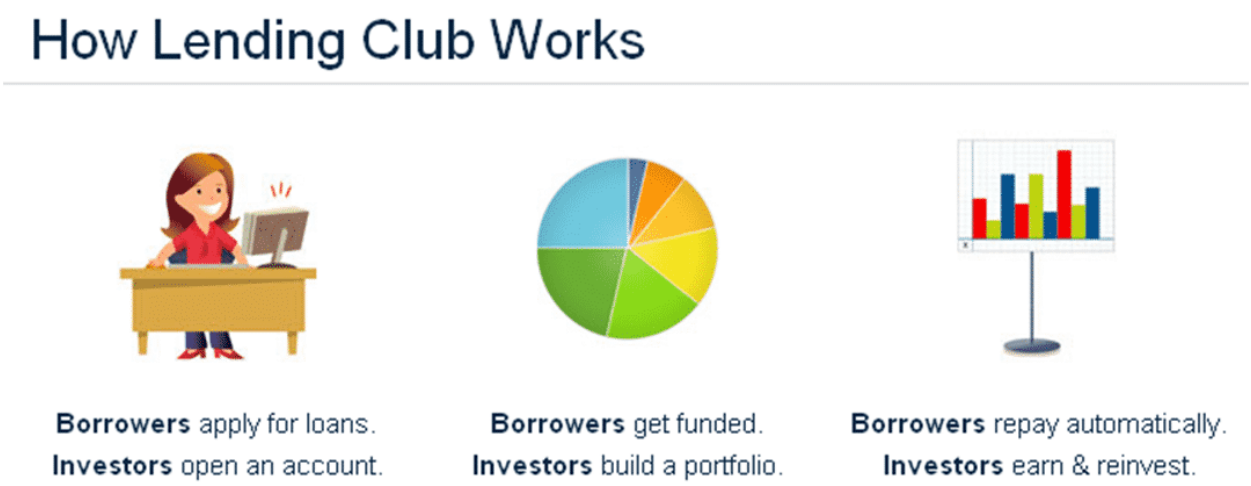
Lending Club Loan Defaulters Prediction

Parag Bhingarkar
(bhingarkar.p@husky.neu.edu)

Introduction

Lending Club is a peer to peer lending company based in the United States, in which investors provide funds for potential borrowers and investors earn a profit depending on the risk they take. Lending Club provides the "bridge" between investors and borrowers.



Objectives

The aim of this project is to build a predictive machine learning tool to classify the loan defaulters and non-defaulters based on their profile. It is important for investors and lenders to know their borrower so they could determine the chances of getting their funds back before lending the money. Lenders, investors could use this model to decide whether a borrower is a fit for sanctioning a loan or not.

Dataset Information

The dataset contains information about all loans issued through 2007-2015, including the current loan status. Loans are divided in categories like Current, Late, Fully Paid, Default etc.) along with that latest payment information is provided as well. Additional features include credit scores, number of finance inquiries, address including zip codes, and state, and collections among others. There are nearly 2 million observations and 144

independent features present in dataset. The dependent variable, loan status, is a categorical variable. The dependent variable is highly imbalanced, there are 8 categories present in loan status however most of the loans are either fully paid or ongoing state. The number of actually defaulted loans is very less compared to Non-defaulted loans.
Data Source - https://www.kaggle.com/wendykan/lending-club-loan-data

Algorithms and Methodologies:

I used Nvidia's Rapids.ai library for data manipulation and data preprocessing. The library works similarly to the Pandas library, but the difference is that the Pandas library utilizes CPU power however Rapids library utilizes GPU power. Since rapids use GPU memory for task execution the processing speed is extremely high. The difference in their execution times is significant, the computation time for rapids when compared with pandas is about 5-10 times better. In rapids, library operations and supporting in-built methods are similar to that of the Pandas library. However, there is a pre-requisite for the Rapids library. To install the library it requires at least 1 Nvidia GPU with pascal architecture, Cuda 9+ support. The rapids.ai is very new in the market so not many functionalities are present in the library however users can easily convert the Cuda DataFrame to pandas DataFrame and vice-a-versa easily.

The first glance of data shows that data cleaning is necessary since there are multiple columns present with missing values. The dataset requires balancing since the number of defaulted loans is very less compared to the non-defaulted loans. If not handled properly this may create an unnecessary bias in the model after training. There are more than 20 features in a categorical format which needs to be converted in numeric format and then to dummy variables.

The dependent feature, loan status, has a total of 8 categories. However, we cannot say whether the ongoing loan will default or the borrower will pay-off the amount so we'll drop that category. Other categories i.e. Charged Off, Late, In Grace Period, Default are combined since they mean the same thing that borrower is not in good standing and has crossed his last payment deadline. After merging multiple categories, total non-defaulted/paid loans contribute 70% of total observations and the remaining 30% are defaulted loans.

Since the dataset is huge and the number of features is high as well the model could overfit easily. The imbalance in dependent feature could create

high bias in the model so to deal with all these I decided to implement ensemble learning (i.e. decision tree-based) algorithms.

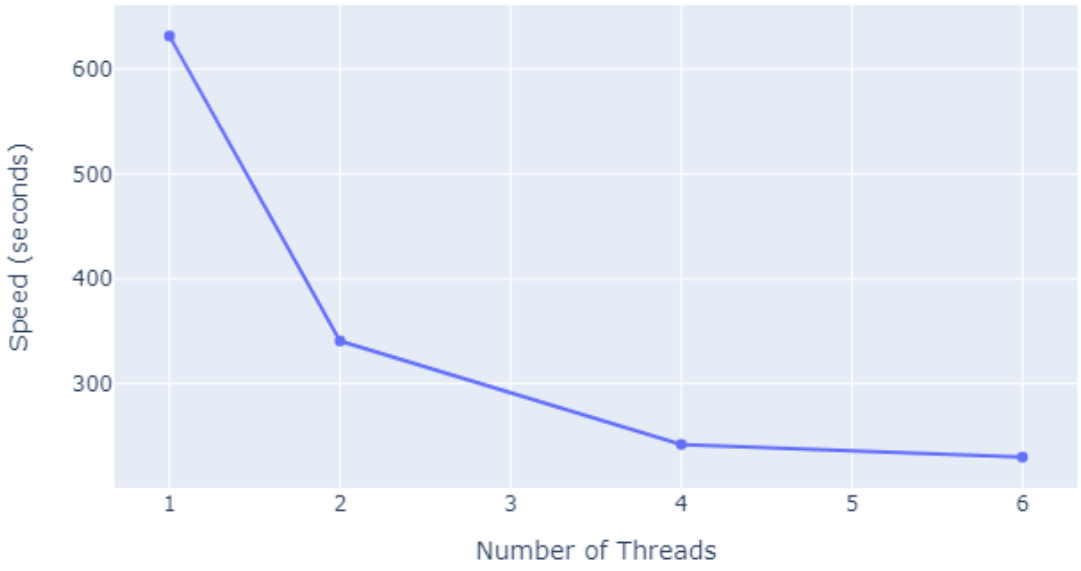
Random Forest

The random forest is a model made up of many decision trees. Rather than just simply averaging the prediction of trees (which we could call a "forest"), this model uses two key concepts that gives it the name *random*:

1. Random sampling of training data points when building trees
2. Random subsets of features considered when splitting nodes

I decided to use random forest first since the algorithm is know for its performance when size of dataset is huge and number of features is more.

The model was trained on CPU with varied number of cores to check the model training time. Following graph shows the training time required to train random forest model on complete training dataset.



The model took max time when it was trained on one core. But as the number of cores was increased the training time decreased significantly. When trained on all available cores model takes ~2 mins to train. So, distributing the work on multiple cores and parallelizing the work reduced time significantly.

The accuracy of model is 100% on training data. When model is validated with k-fold cross validation it shows same accuracy, recall and precision. So we can say that the model is not overfitting. When the out of sample data is used for prediction model gives 99.6% accuracy and 98% recall and 100% precision.

The results for random forest testing/out of sample data :

Accuracy Score: 0.99633

Classification Report:

	precision	recall	f1-score	support
default	1.00	0.98	0.99	5133
non-default	1.00	1.00	1.00	194709
accuracy			1.00	246045
macro avg	1.00	0.99	0.99	246045
weighted avg	1.00	1.00	1.00	246045

Confusion Matrix:

```
[[ 50432    904]
 [     0 194709]]
```

Considering the size of the dataset the model performed better than I expected, did not overfit or created a bias in prediction. However the we are still have 2% type 1 error. The target for the model is to predict the defaulter and using the current model we cannot predict/classify all the borrowers. Still, there are some cases where our model is classifying them as non-defaulters.

We need a technique where we need the algorithm to work on its mistakes and improve to perform better. So I decided to implement extreme gradient boosting.

Gradient Boosting and Extreme Gradient Boosting (XG Boost):

In boosting, the trees are built sequentially such that each subsequent tree aims to reduce the errors of the previous tree. Each tree learns from its predecessors and updates the residual errors. Hence, the tree that grows next in the sequence will learn from an updated version of the residuals.

The base learners in boosting are weak learners in which the bias is high, and the predictive power is just a tad better than random guessing. Each of these weak learners contributes some vital information for prediction, enabling the boosting technique to produce a strong learner by effectively combining these weak learners. The final strong learner brings down both the bias and the variance.

The XG boosting algorithm can build trees in 2 ways.

1. Using CPU
2. Using GPU

Both ways utilize parallel implementation of the XG boost algorithm. By utilizing GPU memory the computation speed can be increased significantly.

The XG boost library provides multiple hyper-parameters to tune the algorithm and improve the model performance. I specified the model to use the GPU for tree building.

The gradient boosting model is implemented in both ways using GPU and CPU. It took ~6 mins to train the model on CPU. While the GPU took hardly 2 mins to train the model. The model performance was similar in both cases. So the only difference was in execution time.

The CPU model performance for the gradient boosting model –

Accuracy Score: 0.99713

Classification Report:

	precision	recall	f1-score	support
defaulter	1.00	0.99	0.99	51335
non-defaulter	1.00	1.00	1.00	194710
accuracy			1.00	246045
macro avg	1.00	0.99	1.00	246045
weighted avg	1.00	1.00	1.00	246045

Confusion Matrix:
[[50628 707]
[0 194710]]

The GPU model performance for the gradient boosting model –
Accuracy Score: 0.99720

Classification Report:

	precision	recall	f1-score	support
defaulter	1.00	0.99	0.99	51335
non-defaulter	1.00	1.00	1.00	194710
accuracy			1.00	246045
macro avg	1.00	0.99	1.00	246045
weighted avg	1.00	1.00	1.00	246045

Confusion Matrix:
[[50652 683]
[5 194705]]

XG Boosting model would improve the model performance and reduce the number of misclassifications. There is a matrix designed for XGB algorithm specially called Dmatrix, which improves the performance of XGB algorithm After utilizing all the functionalities and hyper-parameters. The model was trained in just 10 seconds.
The model performance was also better than random forest model since the algorithm is known for it. It works on its weak predictors and tries to optimize them.
The model performance for XG boosting is as follows

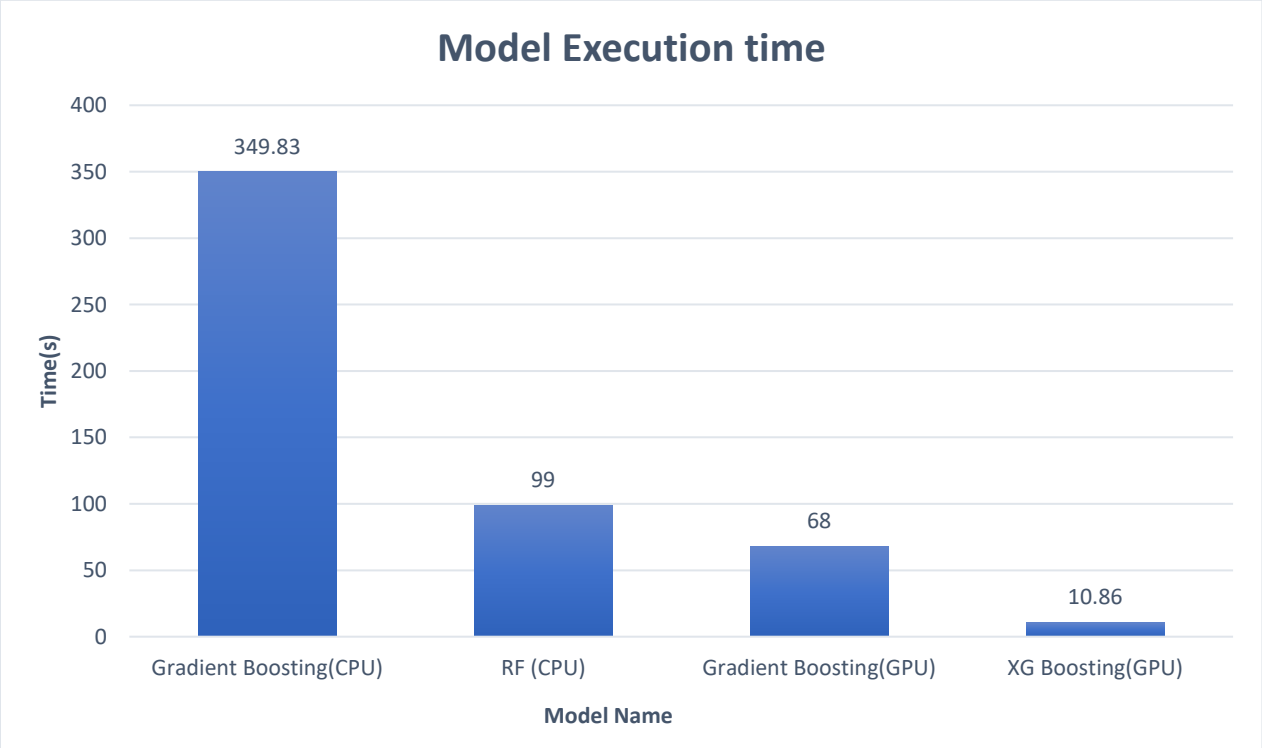
The Confusion Matrix is:
[[51211 125]
[0 194709]]
The Accuracy on Test Set is: 0.99949
Classification Report:

	precision	recall	f1-score	support
Defaulted	1.00	1.00	1.00	51336
Non-Defaulted	1.00	1.00	1.00	194709
accuracy			1.00	246045
macro avg	1.00	1.00	1.00	246045
weighted avg	1.00	1.00	1.00	246045

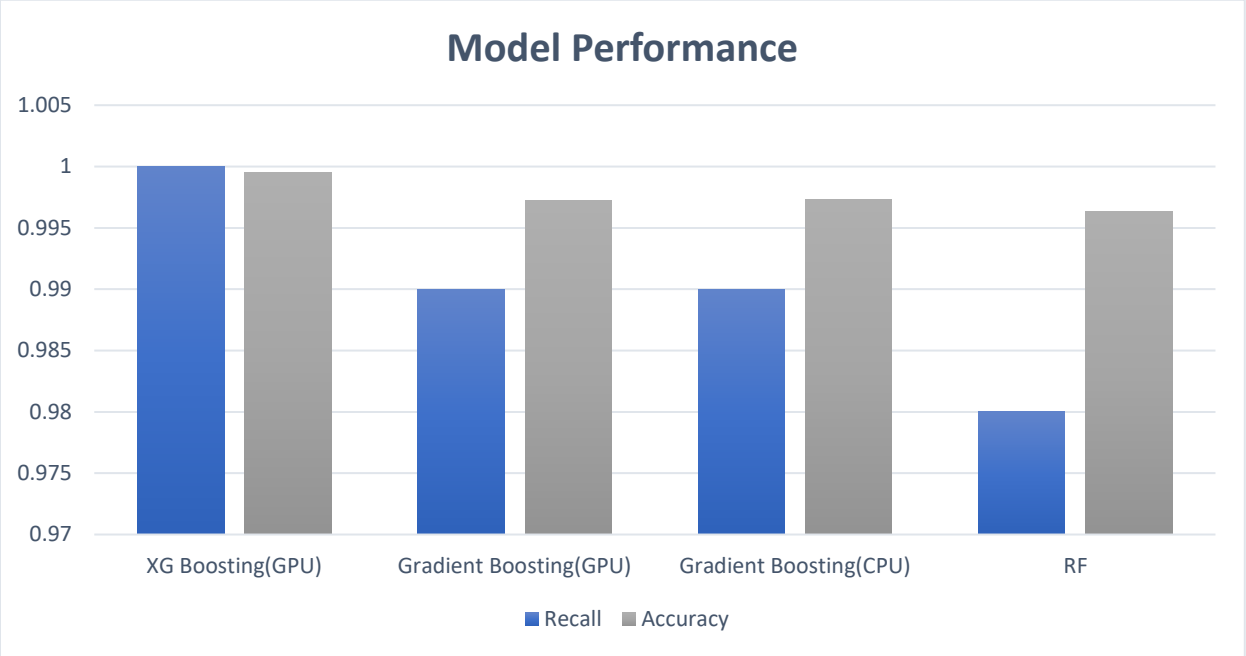
The accuracy for out of sample dataset is almost 99.99%. The f1-score, precision, and recall are better than other models. The error percentage is very less as well.

Model Selection

The dataset was huge, number of features was high regular CPU based algorithms took about 6 to 10 mins to train the dataset. But the GPU based models were trained 2 to 3 times faster and performed better as well.



We can see the XG boosting did great job at predicting with 99.9% accuracy. The precision and recall for model is 1.
The random forest model was slow compared to the XG boosting. RF model predicted the out-of-sample data with 99.6% accuracy.
However, the recall and f1-score is 98% and 99% respectively.
When run time is compared the XGB trained model in just 10 seconds but gr adient boostingRF was slower, it took less than 2 mins.
After considering all the results and computation time I can say that the XG boost model would be the best model for future use.



Conclusion

To conclude this, I would say we designed multiple models and classified the defaulters with 99.99% accuracy (with best XGB model). However, we need to decide what is more important while model development, speed or cost. The CPU computation will be slower compared to GPUs, but they will be cost-effective than a GPU. The GPUs, on the other hand, will give you speed computation power, better accuracies and overall performance will be better than CPU. However, when it comes to cost, it gets extremely expensive to own a powerful GPU. We need to make decisions according to our priorities, if money is not the issue, speed and performance are then GPUs will be a

better option. Otherwise, if time is not the concern and cost is then with we can get the job done with CPUs as well.

References:

Nvidia Rapids.ai https://rapids.ai/
Lending Club - https://www.lendingclub.com/
Towards Data Science – Medium blog
Catboost (Gradient Boost library)- https://catboost.ai/
Kaggle Kernel by Janio Martinez- https://www.kaggle.com/janiobachmann/lending-club-risk-analysis-and-metrics