Group: Luke Bogacz, Vishal Sinha, Jason Long

Dr. A. Jafari

DATS6203: Machine Learning II

29 April 2019

Mortgage Delinquency Prediction

Using Pytorch and a Multi-Layer Perceptron Network, our goal is to predict whether a mortgage loan for a single-family home will fall delinquent by one or more months.

Fannie Mae Single-Family Loan Performance Dataset

The Federal National Mortgage Association, known as Fannie Mae, is a United States government-sponsored enterprise whose purpose is to expand the secondary mortgage loan market. Fannie Mae was founded in 1938 and provides a large dataset of mortgage loan acquisition and performance records for research purposes. Currently, the dataset contains over 35 million mortgage loans that originated after January 1999 or were acquired between January 2000 and March 2016. The mortgage loans are fixed-rate, fully amortizing mortgage loans for single-family homes whose original term is greater than five and less than 35 years. This dataset excludes from its population several types of loans to include Home Affordable Refinance Program (HARP), interest-only, balloon amortization loans. Accessing this data requires an account with Fannie Mae which precludes a direct link, it is addressed later in this report.

Due to Fannie Mae relying on a network of sellers and services, there are errors within the dataset for both the acquisition and performance portions. These errors include field omission as well as incorrect data values. Due to privacy concerns, the full zip code is not provided. The truncated zip code precludes fine granularity for delinquency prediction but still provides enough information for

general geographic exploration. For a complete breakdown of the 55 fields within the acquisition and performance data, please refer to the Fannie Mae glossary.

Fig 1: Dataset Features

Data Element	File	Data Element	File
Adjusted Months To Maturity	P	Number of Units	A
Asset Recovery Costs	P	Occupancy Type	A
Associated Taxes for Holding Property	P	Original Combined Loan-to-Value (CLTV)	A
Borrower Credit Score at Origination	A	Original Debt to Income Ratio	A
Co-Borrower Credit Score at Origination	A	Origination Channel	A
Credit Enhancement Proceeds	P	Original Interest Rate	A
Current Actual UPB	P	Original Loan Term	A
Current Interest Rate	P	Original Loan-to- Value (LTV)	A
Current Loan Delinquency Status	P	Original UPB	A
Disposition Date	P	Origination Date	A
First Payment Date	A	Other Foreclosure Proceeds	P
First Time Home Buyer Indicator	A	Primary Mortgage Insurance Percent	A
Foreclosure Costs	P	Principal Forgiveness Amount	P
Foreclosure Date	P	Product Type	A
Foreclosure Principal Write-off Amount	P	Property Preservation and Repair Costs	P
Last Paid Installment Date	P	Property State	A
Loan Age	P	Property Type	A
Loan Identifier	A/P	Relocation Mortgage Indicator	A
Loan Purpose	A	Remaining Months to Legal Maturity	P
Maturity Date	P	Repurchase Make Whole Proceeds	P
Metropolitan Statistical Area (MS A)	P	Repurchase Make Whole Proceeds Flag	P
Miscellaneous Holding Expenses and Credits	P	Seller Name	A
Miscellaneous Holding Expenses and Credits	P	Servicing Activity Indicator	P
Miscellaneous Holding Expenses and Credits	P	Servicer Name	P
Mortgage Insurance Type	A	Zero Balance Code	P
Net Sale Proceeds	P	Zero Balance Effective Date	P

Non-Interest Bearing UPB	P	Zip Code Short	A
Number of Borrowers	A		

Our initial intention was to use ten or more years of data, which required iterative preprocessing and a database to host the data due to direct data downloads requiring an account and these project requirements include execution from GitHub. The data loader was successful, but handling such a large amount of data over SQL took a substantial amount of time. To overcome the time requirements the team opted to use only one year of data, which was randomly sampled and paired down to three-fourths of the 2017 data. Of the 877,803 records used from the acquisition table, only 33,876 were delinquent (3%). The data was split into three views in the SQL database: training, testing, and validation. Target 0 denotes compliant mortgage loans, and Target 1 denotes delinquent mortgage loans.

Fig 2: Database Size

Database	# Target 0	# Target 1
train	638326	24713
test	79415	2987
validate	160062	6176

The following SQL query was used to split the data:

```
CREATE VIEW test AS
  SELECT *
  FROM acquisition
  WHERE ABS(MOD(loan_id, 1.1)) = 1; /* random selection */

CREATE VIEW validate
  AS
    SELECT *
    FROM
      acquisition
    WHERE loan_id NOT IN (
      SELECT loan_id
```

```
    FROM test
  )
      AND ABS(MOD(loan_id, 2.5)) = 2; /* random selection */

CREATE VIEW train AS
 SELECT *
 FROM acquisition
 WHERE loan_id NOT IN (
  SELECT loan_id FROM test
 )
 AND loan_id NOT IN (
  SELECT loan_id FROM validate
 )
```

Dataset pre-processing

The data was processed in chunks ranging from 5,000 to 15,000-row chunks from the acquisitions data, while only the target feature of, ‘current_loan_delinquency_status’, was extracted from the performance data. The delinquency feature was grouped by, ‘loan_id,’ and condensed into a binary (0/1) if the loan was delinquency was one or more months during any period.

Normalization

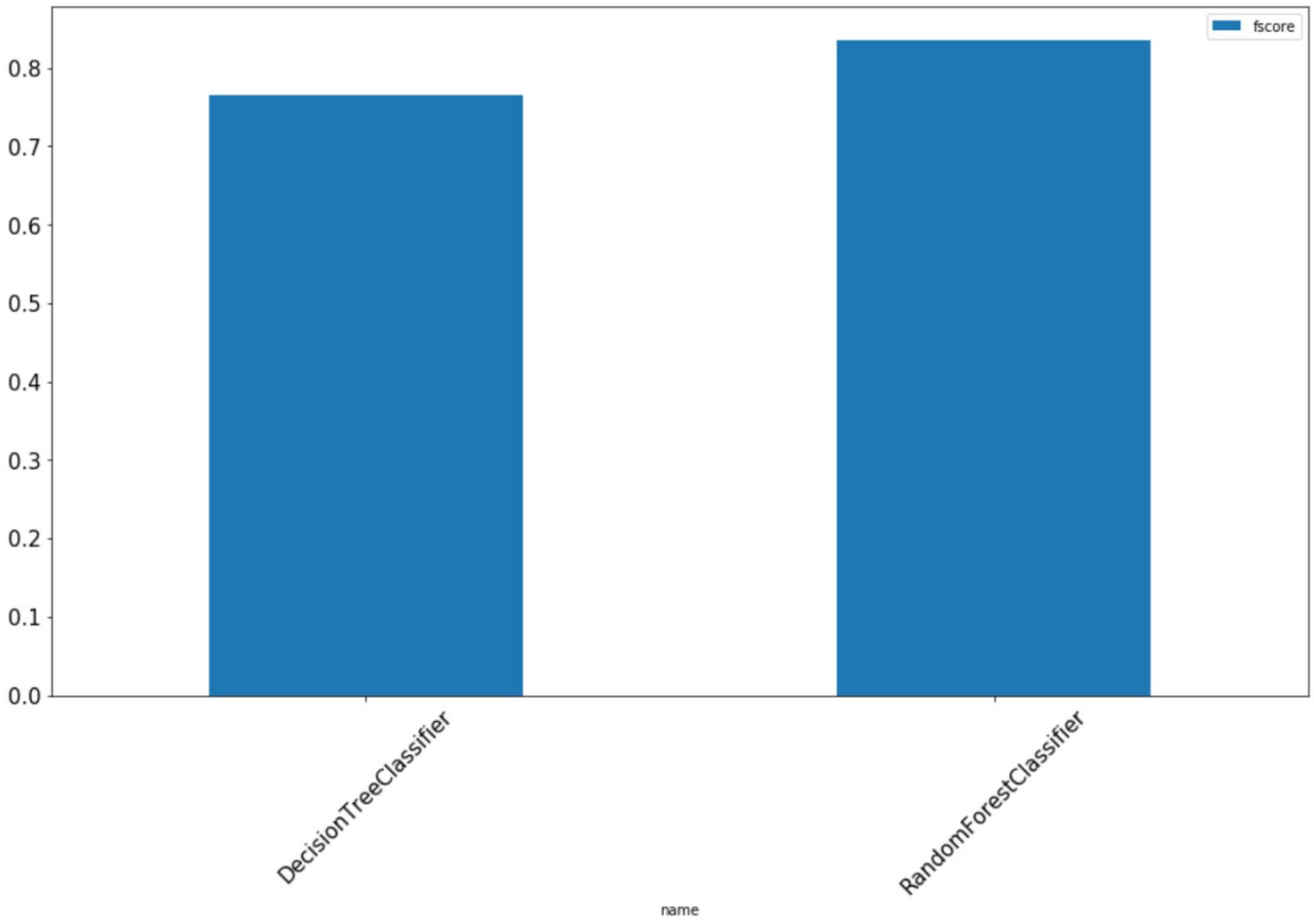
All numerical values were encoded using scikit-learns, StandardScalar, the encodings were applied and then pickled for later use (model training, testing, and validation), the production pickles are saved in the pickles directory. Columns with month data were converted to cyclical numbers before encoding to capture the cyclical nature of the months.

All categorical values were encoded using scikit-learns, OneHotEncoder, by collecting the distinct values from the string based columns within the PostGre database. After encoding, the encoders were pickled into the same pickles directory referenced for numerical values. The zip codes are the most noteworthy category, as they nearly tripled the size of the features in the dataset to approximately 1,100.

Feature Selection

Feature selection was conducted using scikit-learns, DecisionTreeClassifier and RandomForestClassifier. No hyperparameter tuning was applied (most due to time constraints). Even without tuning the parameters, both models yielded f scores of around 80 (see Figure XX below), which is a relatively high level of accuracy. Importantly, this denotes that a Neural Network may not be necessary for the target question depending on the degree of accuracy desired.

Figure 3: f score



For the project we decided to continue using the dataset and target question given the amount of time spent on pre-processing and to fulfill the academic purpose of understanding Neural Networks. Using the ensemble classifiers, we investigated feature importance on 1,026 features. Because the data was analyzed using samples, the feature selections varied. Some features seemed more consistent in importance and are the borrower credit score at origination, original upb, original debt to income ratio, original loan to value, co-borrower credit score at origination, primary mortgage insurance percent, first payment month cos, and origination month cos.

In some cases, these features are well known in the credit industry. Specifically, ‘borrower credit score at origination’ and ‘original debt to income ratio,’ are always asked for. In other cases, the zip codes showed as important, so we opted to keep the zip codes as features in some of our model testings. The feature relationships are plotted as a pair plot in Figure XX. Here you can see that ‘borrower credit score at origination’ and ‘co-borrower credit score at origination’ have a co-linear relationship and so only on feature is needed. This helped reduce the size of the data going into the model. A heat-map correlation matrix, Figure XX, and feature importance are plotted in Figure XX.

Associated files: lib/helpers

Figure 4: Correlation Matrix

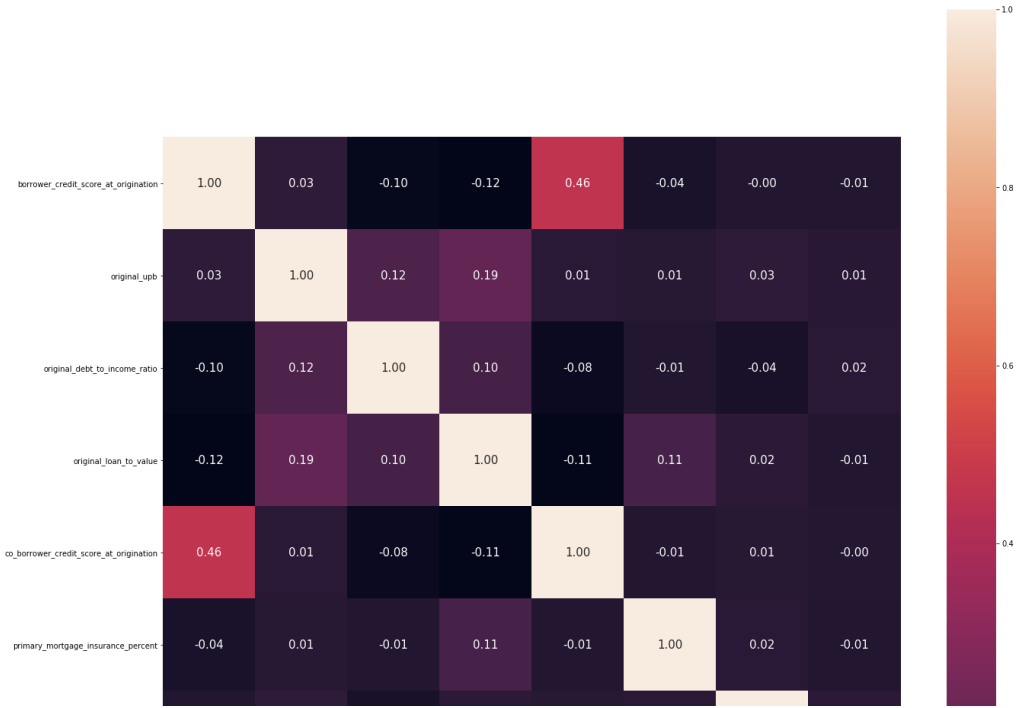


Figure 5: Feature Pairplot

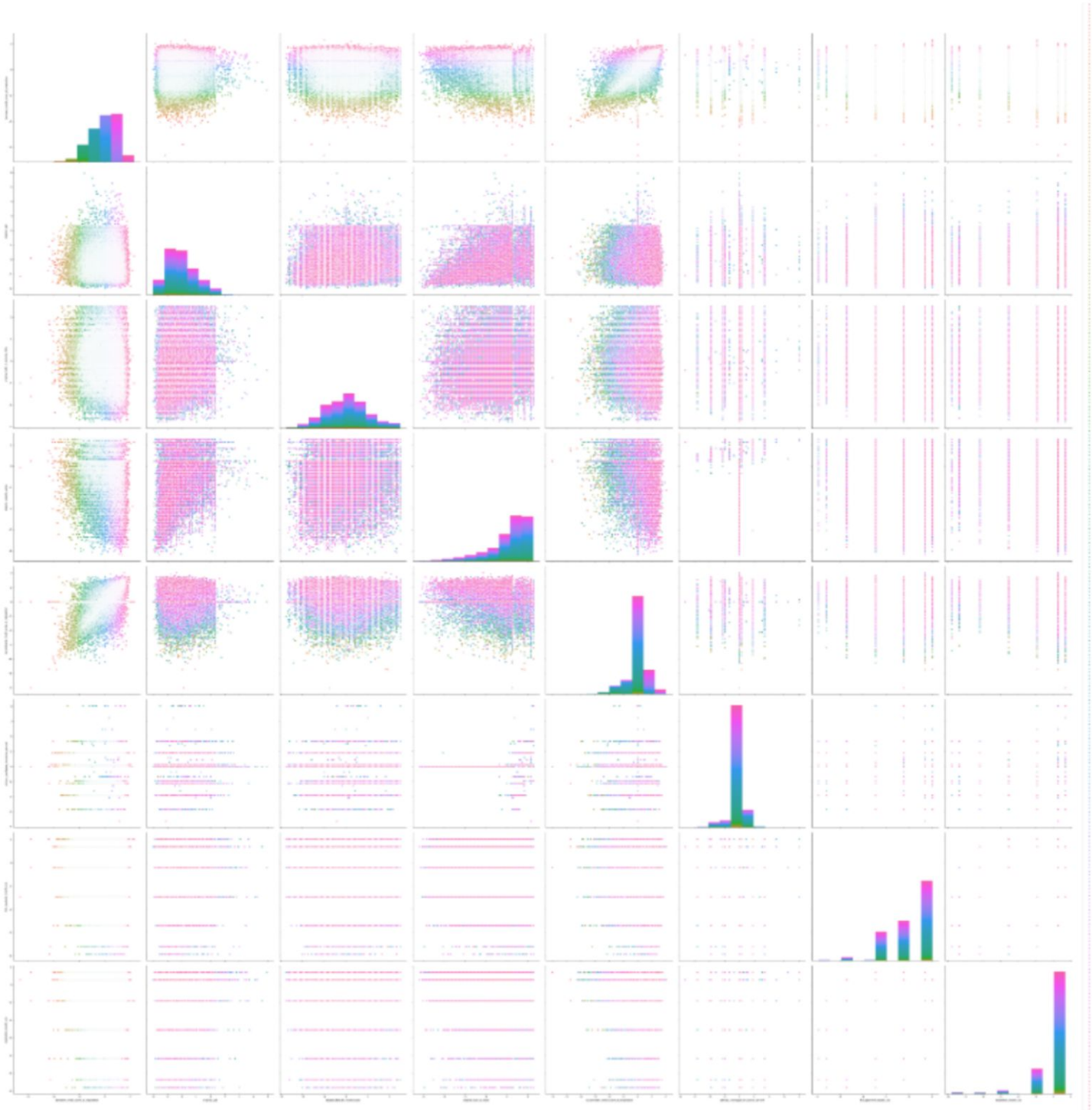
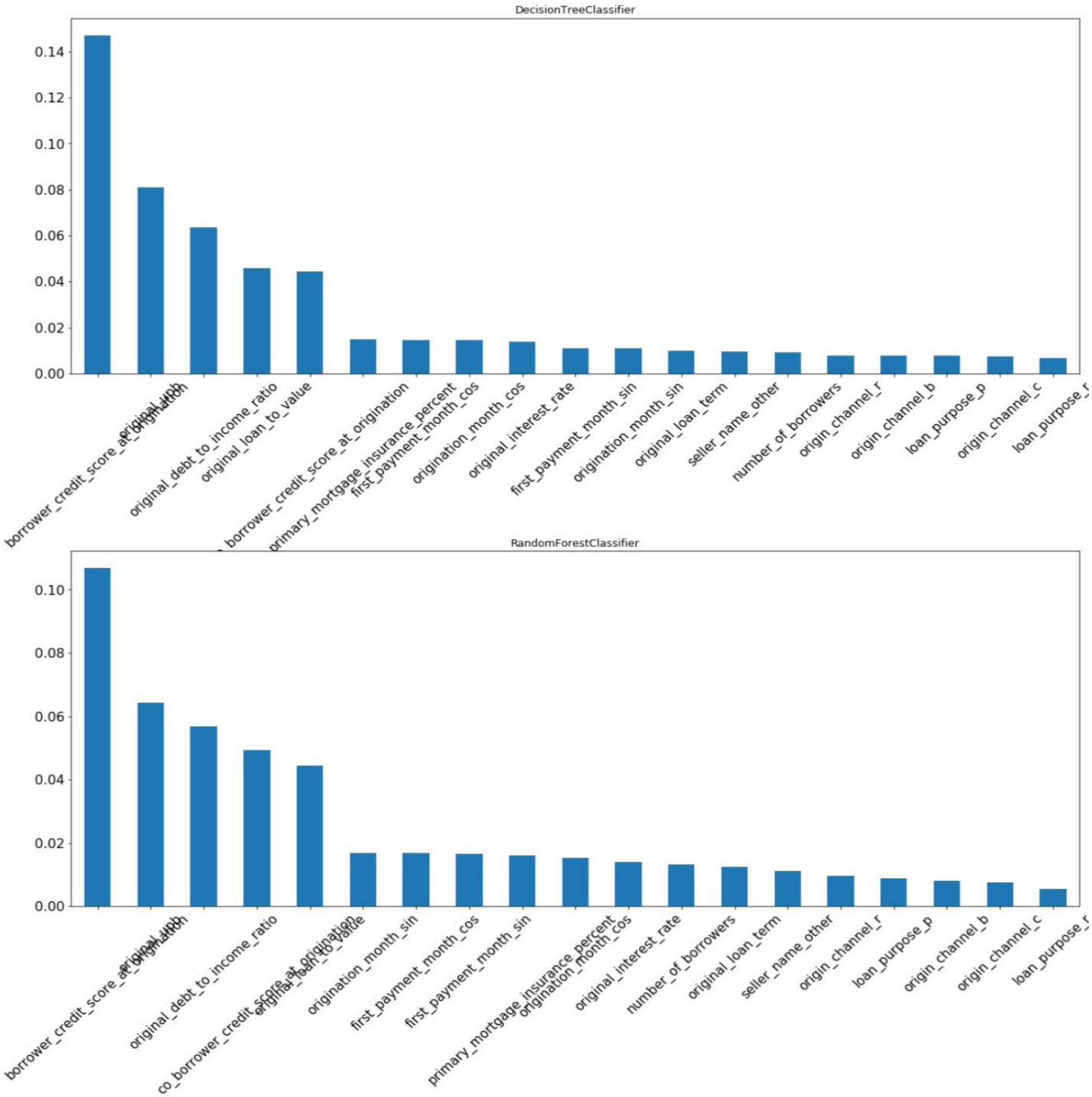


Figure 6: Top Feature Importances


DecisionTreeClassifier
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
original_debt_to_income_ratio
original_loan_to_value
first_payment_month_cos
origination_month_cos
original_interest_rate
first_payment_month_sin
origination_month_sin
original_loan_term
seller_name_other
number_of_borrowers
origin_channel_b
loan_purpose_p
origin_channel_c
RandomForestClassifier
0.10
0.08
0.06
0.04
0.02
0.00
original_debt_to_income_ratio
origination_month_sin
first_payment_month_cos
first_payment_month_sin
origination_month_cos
original_interest_rate
number_of_borrowers
original_loan_term
seller_name_other
origin_channel_r
loan_purpose_p
origin_channel_b
origin_channel_c

Data Loader

Due to the data being too large to store in memory, a Pytorch dataset was created. The Pytorch dataset allowed the iteration of the data specified within ‘chunk_size.’ A ‘torch.DataLoader’, was then wrapped around the dataset to allow further batching and shuffling of the data. However, a batch size of one was used, allowing batch sizes to be handled using the ‘chunk’ variable. (ref, https://pytorch.org/docs/stable/torchvision/datasets.html)

Associated files: lib/data

Main.py

Our team decided to proceed with Pytorch as the machine learning library for this project. The Pytorch library is an optimized tensor library for deep learning using graphic processing units and central processing units. Our design was focused on a multilayer perceptron neural network design. (ref)

The main.py file is responsible for the model training, testing, and validation processes. This file leverages the data-loader previously discussed, but also a custom built class, ModelRunner, which is responsible for running epochs, and handling other data collection and mode internals. Training the model was fairly erratic, this is likely due to to the limited amount of positive targets (3%) available throughout the training set. In some cases all targets are likely 0, causing over-fit for the batch. In one test case we used softmax, however, results were suboptimal at 75%, possibly because the CrossEntropyLoss criterion was not used. We experimented with various neuron sizes, layer sizes, dropout and activation function changes. Though results differ, this is more likely due to variation in the training data than adjustments. However, on average, a larger neuron count in layers

one and two seemed to yield more accurate results. It is likely that as the network was trained on a larger portion of the dataset, the loss would be minimized further.

We tried five layers model configurations using the Adam optimizer: 2 ReLu, 1 ReLu and 1 Sigmoid, 3 ReLu, 1 ReLu and 1 Softmax, 2 ReLu with 20% drop out. They are noted below in Tables M0-M4 and include a confusion matrix.

M1: 2 Layer ReLu Model, 92% Accuracy

Predicted	False	True	All
Actual			
False	55992	506	56498
True	3268	34	3302
All	59260	540	59800

```
## ReLu model 2 layer
 layers = [
    nn.Linear(INPUT_SIZE, NUERONS_l1),
    nn.ReLU(),
    nn.Linear(NUERONS_l1, CHUNK_SIZE),
    nn.ReLU(),
 ]
```

M2: Sigmoid Model, 97% Accuracy

Predicted	0.0	1.0	All
Actual			

0.0	57872	0	57872
1.0	1928	0	1928
All	59800	0	59800

```
## Sigmoid model
 layers2 = [
    nn.Linear(INPUT_SIZE, NUERONS_l1),
    nn.ReLU(),
    nn.Linear(NUERONS_l1, CHUNK_SIZE),
    nn.Sigmoid()
 ]
```

M3: 3 Layer ReLu Model, 92% Accuracy

Predicted	False	True	All
Actual			
False	55794	50	55844
True	3956	0	3956
All	59750	50	59800

```
## Relu model 3 layer
 layers3 = [
    nn.Linear(INPUT_SIZE, NUERONS_l1),
    nn.ReLU(),
    nn.Linear(NUERONS_l1, NUERONS_l2),
    nn.ReLU(),
    nn.Linear(NUERONS_l2, CHUNK_SIZE),
    nn.ReLU()
 ]
```

M4: Softmax Model, 88% Accuracy

Predicted	0.0	1.0	All
Actual			
0.0	56483	0	56483
1.0	3317	0	3317
All	59800	0	59800

```
## Softmax model
 layers4 = [
    nn.Linear(INPUT_SIZE, NUERONS_l1),
    nn.ReLU(),
    nn.Linear(NUERONS_l1, CHUNK_SIZE),
    nn.Softmax(dim=1)
 ]
```

M5: Dropout Model, 99% Accuracy

Predicted	False	True	All
Actual			
False	59511	165	59676
True	123	1	124
All	59634	166	59800

```
## dropout model
 layers5 = [
    nn.Linear(INPUT_SIZE, NUERONS_l1),
    nn.ReLU(),
    nn.Linear(NUERONS_l1, CHUNK_SIZE),
    nn.ReLU(),
 ]
```

The Adaptive Moment Estimation (Adam) is a method that computes adaptive learning rates. Adam keeps exponentially decaying average of past gradients which is similar to momentum. We used Adam as our principle optimizer as our previous testing (earlier in the course) and iterative testing during this project, provided a high accuracy.

During our experimentation and testing we used the following activation functions: Rectified Linear Unit (ReLu) defined as y=max(0, x); Sigmoid or Logistic function exists between 0 and 1, and functions best for predicting probability as an output; Softmax is more generalized and is better used for multiclass classification and calculates probabilities of also 0 and 1, but the sum of all probabilities will equal 1.

We used a 20% dropout layer and achieved our highest accuracy of 99%. The use of a dropout layer helps reduce overfitting. By reducing overfitting, the model is more generalized and is more accurate against the test set. The use of a dropout layer or layers would be required to further increase the accuracy of the model above 99%.

Summary and conclusions

Our results showed a high performance neural network model (~ 95%) over the decision tree and random forest classifiers (f score of ~80). Not being investment bankers, the impact of an approximately 95% model are unclear; however, would likely need to be even higher to guard against asset loss. Training over a larger dataset could assist in increasing accuracy.

Our framework basically produces a three-layer architecture for model engineering with first layer as data loader which controls fetching, massaging and feature selection of data, second layer being network design, and third layer is error and validation logic. This gives us great flexibility in fine tuning of various components of model design which includes feature selection, and adjusting error validation logic. We have tested various model configurations, however, 2 layer ReLu model with 20% dropouts gave us 99% accuracy. Since our target is binary in nature with less than 3% essentially positive; it results in highly unbalanced data. By tuning the data-loader to randomly shuffle the dataset and using the same to train various model configurations, we can further improve model performance.

We have learned that real world data is messy, and when dealing with millions of records, can become time prohibitive to properly clean and model with. Then once you finally have obtained a workable dataset, the next hurdle is database management, access limitations (responsiveness) for model iterations, overcome with the use of an early stop, are required. Hardware limitations to chunk size were noted when testing with different graphic processor configurations..

In the future, there are several areas that could be improved. One component would be a clearer understanding of the accuracy requirements that relevant industries require. This would directly drive the resources that could help achieve those goals, if possible.

References

0. Wikipedia, https://en.wikipedia.org/wiki/Fannie_Mae

1. Fannie Mae, http://www.fanniemae.com/portal/funding-the-market/data/loan-performance-data.html

2. Fannie Mae, https://loanperformancedata.fanniemae.com/lppub/index.html

3. Fannie Mae, https://loanperformancedata.fanniemae.com/lppub-docs/FNMA_SF_Loan_Performance_FAQs.pdf

4. Fannie Mae, https://loanperformancedata.fanniemae.com/lppub-docs/FNMA_SF_Loan_Performance_Glossary.pdf

Code References

Dr. A. Jafari, Deep Learning, *https://github.com/amir-jafari/Deep-Learning*, Pytorch_ resources

Facebook, Pytorch, *https://pytorch.org*

Waleed, HiddenLayer, *https://github.com/waleedka/hiddenlayer*

Code Appendix

Per guidance, commented code in the project GitHub repository has not been included in this section. ***Please note any code that you would like in this section***.

X. Below is the sample code for creating various network configuration use for testing

```
f __name__ == '__main__':
    layers = [
        nn.Linear(INPUT_SIZE, NUERONS_l1),
        nn.ReLU(),
        nn.Linear(NUERONS_l1, CHUNK_SIZE),
        nn.ReLU(),
    ]

## Sigmoid model
if __name__ == '__main__':
    layers2 = [
        nn.Linear(INPUT_SIZE, NUERONS_l1),
        nn.ReLU(),
        nn.Linear(NUERONS_l1, CHUNK_SIZE),
        nn.Sigmoid()
    ]


## Relu model 3 layer
if __name__ == '__main__':
    layers3 = [
        nn.Linear(INPUT_SIZE, NUERONS_l1),
        nn.ReLU(),
        nn.Linear(NUERONS_l1, NUERONS_l2),
        nn.ReLU(),
        nn.Linear(NUERONS_l2, CHUNK_SIZE),
        nn.ReLU()
    ]


## Softmax model
if __name__ == '__main__':
    layers4 = [
        nn.Linear(INPUT_SIZE, NUERONS_l1),
        nn.ReLU(),
        nn.Linear(NUERONS_l1, CHUNK_SIZE),
        nn.Softmax(dim=1)
    ]

## dropout model

if __name__ == '__main__':
    layers5 = [
```

```
    nn.Linear(INPUT_SIZE, NUERONS_l1),
    nn.ReLU(),
    nn.Linear(NUERONS_l1, CHUNK_SIZE),
    nn.ReLU(),
]
```

X. Sample code for the Error validation is below:

```
for features, labels in test_loader:
    if self.stop_early_at and (counter + 1) % self.stop_early_at == 0:
        break
    #print("feature ",features)
    #print("label ",labels)
    features, labels = self._to_device(features, labels)
    features = features if not self.is_image else features.view(-1, self.dimensions)
    outputs = self.model(features)
    #print("PreOutputs ", outputs)
    if(self.adjustment == 1 ):
        #print("ADJUSTING")
        outputs = torch.ceil(outputs*self.adjustment)
        outputs = torch.clamp(outputs, min=0, max=1)
    else:
        if(self.adjustment!=0):
            outputs = torch.clamp(outputs, min=0, max=self.adjustment)
            scalar_multi=1/self.adjustment
            outputs = torch.round(outputs*scalar_multi)
        else:
            outputs = torch.round(outputs)

    #print("Outputs ",outputs)
    #_, predicted = torch.max(outputs.data, 1)
    total += labels.numel()
    #correct += (predicted == labels.type(torch.long)).sum()
```

```
predicted = (outputs == labels).sum()

correct += predicted
```