HCDR Team 1 Bears

Zack Seliger	Keegan Moore	Rajasimha	Jagan Lakku
zseliger@gmail.com	keegmoor@iu.edu	ragallam@iu.edu	slakku@iu.edu




Project Abstract

The objective of this project is to use historical loan application data to predict whether or not an applicant will be able to repay a loan. This is a standard supervised classification project.

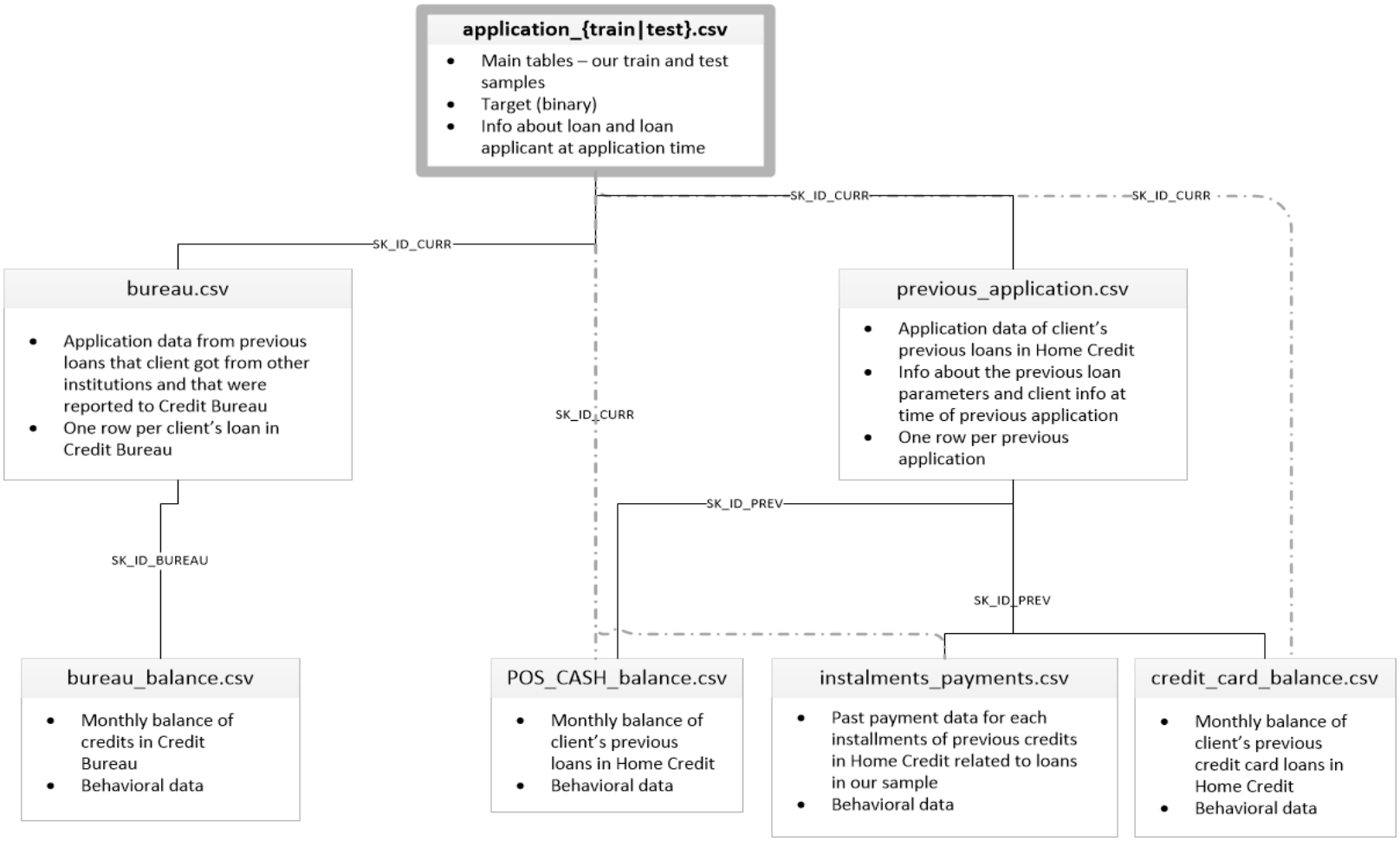
The goal of the Phase-3 is to implement the neural network model with the additional features and hyper parameters tuning to have better AUC score compared to the LGBM/XGBoost model.

Firstly we implemented the Deep Learning model with a single hidden layer with the help of Application data and with all other data available. We used Adam optimizer, and also tried adding more layers to make the ROC better but unfortunately couldn't. We then did K-fold training to try and prevent overfitting, but we saw little difference. The output layer has a softmax activation function as we are doing classification. The Deep neural network is compiled with Adam optimizer, Binary Cross Entropy loss function and the evaluation metrics are accuracy and ROC_AUC.

Our final submission's score was 0.750, which is slightly worse than our LGBM model's score, which was 0.752.

Project Description

Our data is split up among several different CSV files and looks like this:



Application{train|test}.csv contains static data for all applications and one row represents one loan. SK_ID_CURR represents the ID for that row, which can be tied in with the other datasets for this problem. There are also more IDs, like SK_ID_BUREAU and SK_ID_PREV, that correspond to bureau and previous application data, respectively.

In Phase 1, we looked at the application dataset and datasets that we could access with just SK_ID_CURR. In Phase 2, we experimented with aggregating datasets using

SK_ID_BUREAU and SK_ID_PREV to collect more information about our loan. We also created new features by manipulating the features we had.
In phase 3, Firstly we implemented the Deep Learning model with a single hidden layer, using SGD as our optimizer. We noticed that adding more layers hurt the model's performance, so we looked at other things. First, we noticed that we were overfitting pretty quickly, so we tried adding Dropout and BatchNorm layers, but neither helped, and the Dropout layers actually hurt our model's performance. We then tried K-fold training to avoid overfitting, but that also didn't help. In the end, we used our original model with a single layer to achieve an ROC_AUC score of 0.750 on Kaggle.

Roughly, our workflow for this part was pretty simple. We were able to get quick feedback after training by seeing our train vs test AUC, so we had a good idea of the performance of our model and how fast it was overfitting. Then, we made tweaks to try and improve our performance.

Neural Networks

We created a Neural Network model for our data. When initially making it, we only used data from application_train and application_test so we could make sure it works. We used our sklearn pipeline to preprocess our data before feeding it into our model. Our architecture was pretty simple as well.

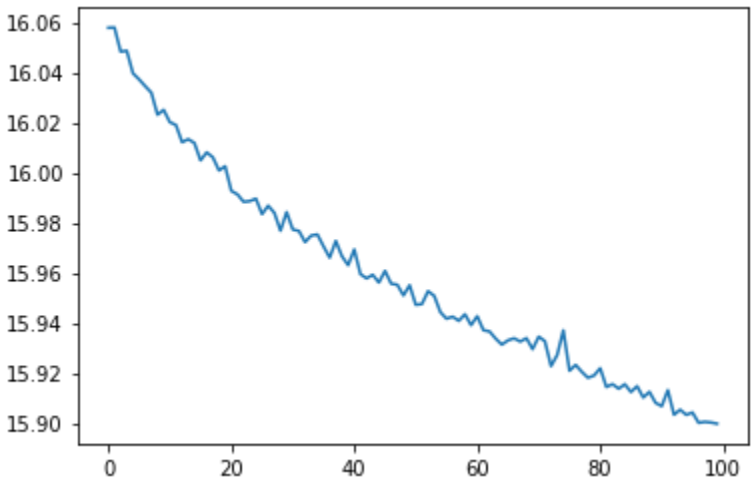
```
class CustomModel(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear = nn.Sequential(
        nn.Linear(num_in, num_layer_1),
        nn.ReLU(),
        nn.Linear(num_layer_1, num_output)
    )

  def forward(self, x):
    out = self.linear(x)
    return nn.functional.softmax(out)

model = CustomModel()
opt = optim.SGD(model.parameters(), lr=0.01)
loss_fn = nn.BCELoss()
```

We had an input of the number of features we had, which was around 78. We used a ReLU activation function before going to our hidden layer, which we arbitrarily decided to be 20 nodes large. We used softmax for our output.

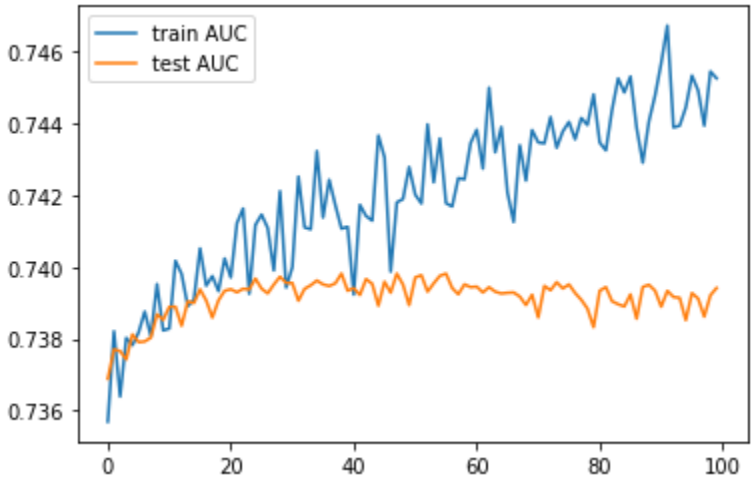
We chose to use Stochastic Gradient Descent as our optimizer and Binary Cross-Entropy as our loss function, which can be defined as:

$$l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right]$$

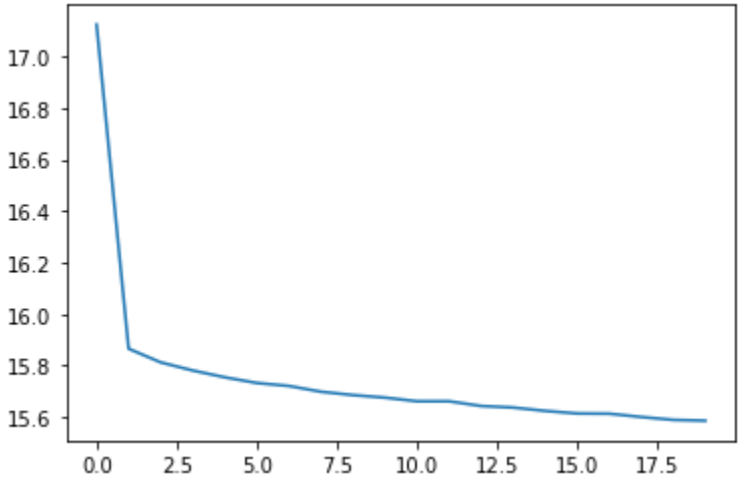
We trained this model for 100 epochs and a batch size of 64. Here is the loss over time:

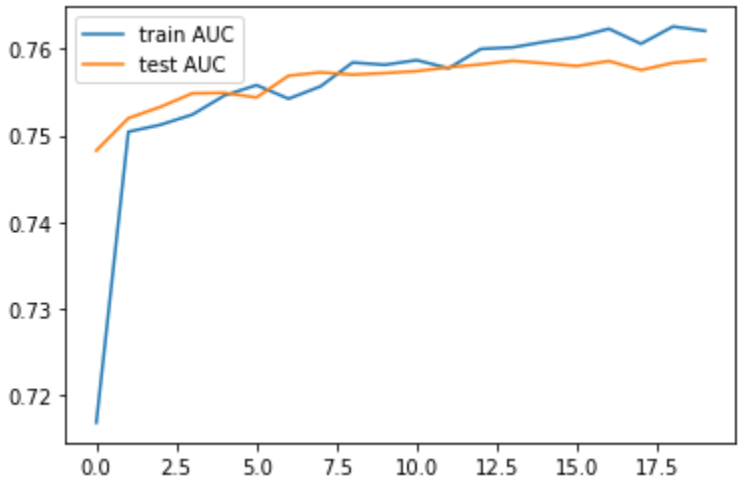


And here is the AUC scores over time:



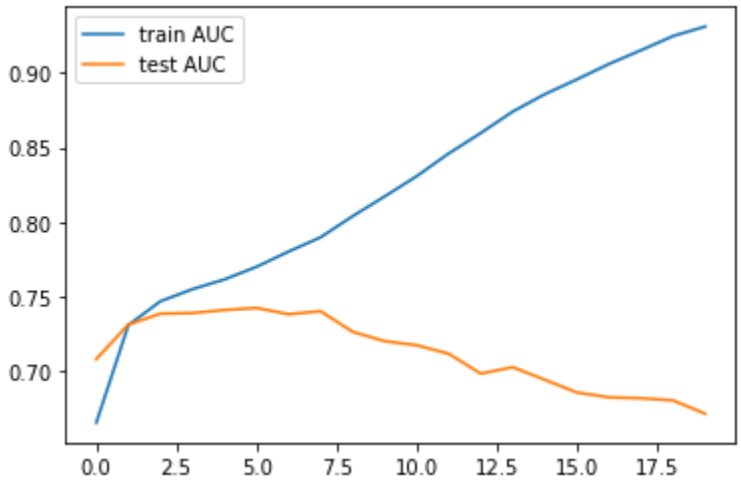
We achieved a final ROC_AUC of 0.739, which is actually better than our baseline model for the same data, which was 0.734 for our test dataset. We then added the rest of the features and switched the optimizer to Adam, since it's usually the better choice. For all other models, we have 173 input features. We chose to run it for 20 epochs after some experimenting. Here was our loss and auc scores:



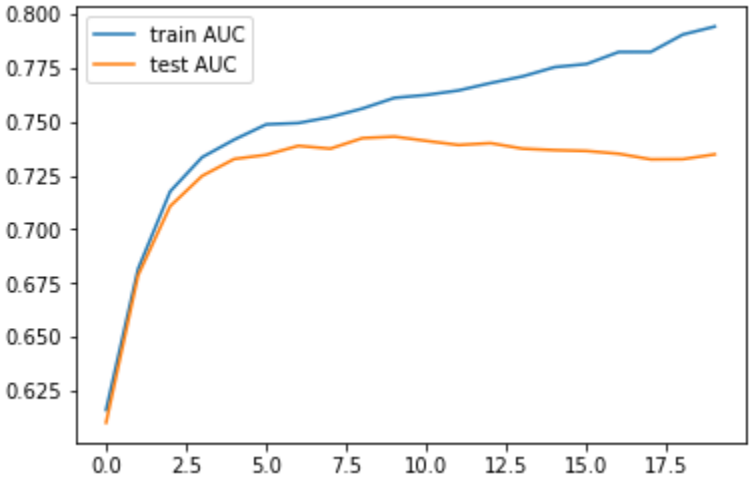


Generally, we found that after 20 epochs, the model overfits. We determined whether a model was overfit or not based on the difference between test and train AUC. This

model had a test ROC of 0.758 and after submitting to Kaggle we had an ROC score of 0.750, which was our best of all neural networks we considered in this phase.
We thought that increasing the size of our network would help. We tried several layers with different parameters, but the AUC always look roughly like this:



Clearly, it overfits too quickly. We tried adding Dropout layers and BatchNorm layers, changed around the learning rates and so on, but we couldn't do better than 0.75 for the test AUC. As a last attempt, we added in K-Fold training, which would hopefully reduce overfitting.



Although it seemed to work, our test AUC still never crossed the 0.75 barrier. This particular model has a lot of Dropout layers. In the end, we decided to go back to our initial model, which performed better than all of our other models.

The architecture for our largest model looked like this:

```
[96] class CustomModel(nn.Module):
       def __init__(self):
         super().__init__()
         self.linear = nn.Sequential(
             nn.Linear(num_in, 256),
             nn.ReLU(),
             nn.Dropout(0.2),
             nn.BatchNorm1d(256),
             nn.Linear(256, 512),
             nn.ReLU(),
             nn.BatchNorm1d(512),
             nn.Linear(512, 256),
             nn.ReLU(),
             nn.BatchNorm1d(256),
             nn.Linear(256, 128),
             nn.ReLU(),
             nn.Dropout(0.2),
             nn.BatchNorm1d(128),
             nn.Linear(128, 64),
             nn.ReLU(),
             nn.BatchNorm1d(64),
             nn.Linear(64, 32),
             nn.ReLU(),
             nn.BatchNorm1d(32),
             nn.Dropout(0.2),
             nn.Linear(32, num_output)
         )

       def forward(self, x):
         out = self.linear(x)
         return nn.functional.softmax(out)
```

After determining that a single hidden layer gave us the best results, we tried different layer sizes. A layer size of 64 performed worse, however, so we tried even smaller, to a

layer size of 10, but it seemed identical to our base of 20 neurons. In the end, we stuck with our initial model of 1 hidden layer with 20 neurons.

In summary, our experiments looked like this:

	ExpID	ROC AUC Score	Test Accuracy	Train Time(s)	Test Time(s)	Experiment description
0	Deep Learning	0.739418	0.917529	1075.331878	0.037791	Deep Learning w/ Application Data
1	Deep Learning	0.758407	0.917631	1400.716561	0.048049	Deep Learning w/ all other data
2	Deep Learning	0.758704	0.917315	238.759920	0.047533	Adam optimizer
3	Deep Learning	0.671086	0.906519	995.408674	0.873410	More layers
4	Deep Learning	0.732227	0.918854	510.319242	1.011763	K-Fold training
5	Deep Learning	0.750459	0.917424	264.229233	0.056706	Modifying Layer Sizes

After we submitted the “Adam optimizer” model to Kaggle:

Name	Submitted	Wait time	Execution time	Score
submission.csv	just now	1 seconds	0 seconds	0.75030

Complete

This score is slightly worse than our LGBM model.

Leakage

The main places where leakage is usually when fitting data. Specifically, when we fit our pipelines, we don’t want it to fit the test data. Then, when training data, we want to make sure we don’t train on the test dataset.

One, more subtle leakage, would be changing the distributions of classes to match. We didn’t do this, since we used sklearn’s *train_test_split* to split the datasets for us.

In our Kaggle competition, we don’t have access to the actual labels for the real test dataset, so it’s almost impossible to leak here, but we had to make sure we also didn’t leak anything from the test dataset that we split off from our train dataset. Since we were careful about our code, and our results were pretty realistic (train ROC > test > Kaggle), we are sure that we didn’t make some kind of mistake with our data.

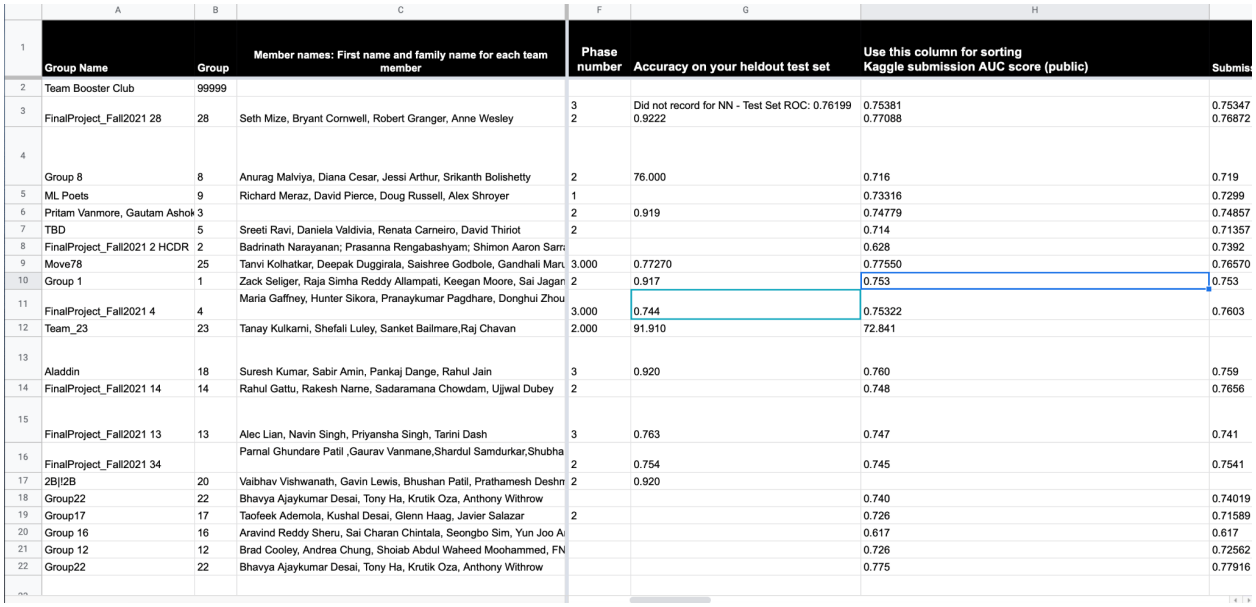
Results

The results for Part 3 were pretty disappointing to us. We expected to beat our LGBM scores after tuning the architecture of our model, but instead we found that each change didn't affect the model or made it worse. The biggest change that we made was one of the first, which was when we changed our optimizer to Adam from SGD. Our results stayed the same, but we had a speedup of 5x.

After that change, we added layers. This corresponds to the "More Layers" experiment. As you can see, we did significantly worse. However, we tried a lot of things. After increasing the network size, it seemed to overfit much quicker, as shown in the train/test AUC graphs. We think that a larger network just wasn't the answer. We tried to reduce overfitting by introducing Dropout or BatchNorm layers, but they didn't improve performance above 0.74. Since our initial model had a score of 0.75, it was clear to us that our initial model was just better.

Sometimes, simpler is better, and I think our experiments this phase shows that. No matter what we tried, a single hidden layer of around 20 neurons gave the best result. There may have been more room for experimentation here, but we think that we exhausted almost every option to improve performance. In terms of what we left on the table, there were probably more ways to improve model performance in feature engineering than tuning our model.

As of writing this, we are 8th in our class's leaderboard for HCDR on Kaggle submission ROC AUC.



	A	B	C	F	G	H	
1	Group Name	Group	Member names: First name and family name for each team member	Phase number	Accuracy on your heldout test set	Use this column for sorting Kaggle submission AUC score (public)	Submis
2	Team Booster Club	99999					
3	FinalProject_Fall2021 28	28	Seth Mize, Bryant Cornwell, Robert Granger, Anne Wesley	3 2	Did not record for NN - Test Set ROC: 0.76199 0.9222	0.75381 0.77088	0.75347 0.76872
4	Group 8	8	Anurag Malviya, Diana Cesar, Jessi Arthur, Srikanth Bolishetty	2	76.000	0.716	0.719
5	ML Poets	9	Richard Meraz, David Pierce, Doug Russell, Alex Shroyer	1		0.73316	0.7299
6	Pritam Vanmore, Gautam Ashok	3		2	0.919	0.74779	0.74857
7	TBD	5	Sreeti Ravi, Daniela Valdivia, Renata Carneiro, David Thiriot	2		0.714	0.71357
8	FinalProject_Fall2021 2 HCDR	2	Badrinath Narayanan; Prasanna Rengabashyam; Shimon Aaron Sarr			0.628	0.7392
9	Move78	25	Tanvi Kolhatkar, Deepak Duggirala, Saishree Godbole, Gandhali Mar	3.000	0.77270	0.77550	0.76570
10	Group 1	1	Zack Seliger, Raja Simha Reddy Allampati, Keegan Moore, Sai Jagar	2	0.917	0.753	0.753
11	FinalProject_Fall2021 4	4	Maria Gaffney, Hunter Sikora, Pranaykumar Pagdhare, Donghui Zhou	3.000	0.744	0.75322	0.7603
12	Team_23	23	Tanay Kulkarni, Shefali Luley, Sanket Bailmare,Raj Chavan	2.000	91.910	72.841	
13	Aladdin	18	Suresh Kumar, Sabir Amin, Pankaj Dange, Rahul Jain	3	0.920	0.760	0.759
14	FinalProject_Fall2021 14	14	Rahul Gattu, Rakesh Narne, Sadaramana Chowdam, Ujjwal Dubey	2		0.748	0.7656
15	FinalProject_Fall2021 13	13	Alec Lian, Navin Singh, Priyansha Singh, Tarini Dash	3	0.763	0.747	0.741
16	FinalProject_Fall2021 34		Parnal Ghundare Patil ,Gaurav Vanmane,Shardul Samdurkar,Shubha	2	0.754	0.745	0.7541
17	2B!2B	20	Vaibhav Vishwanath, Gavin Lewis, Bhushan Patil, Prathamesh Deshn	2	0.920		
18	Group22	22	Bhavya Ajaykumar Desai, Tony Ha, Krutik Oza, Anthony Withrow			0.740	0.74019
19	Group17	17	Taofeek Ademola, Kushal Desai, Glenn Haag, Javier Salazar	2		0.726	0.71589
20	Group 16	16	Aravind Reddy Sheru, Sai Charan Chintala, Seongbo Sim, Yun Joo A			0.617	0.617
21	Group 12	12	Brad Cooley, Andrea Chung, Shoiab Abdul Waheed Moohammed, FN			0.726	0.72562
22	Group22	22	Bhavya Ajaykumar Desai, Tony Ha, Krutik Oza, Anthony Withrow			0.775	0.77916

Conclusion

The accuracy of our models is important, as it can directly affect the lives of those looking for loans. We believe that well-made machine learning models can predict the repayment ability of clients with great accuracy. In this phase of our project, we tried to build on the results of our previous phases by building a neural network. We tried to beat our best model so far, but we fell short. After all of our efforts in experimenting with more or less layers, Dropout layers, BatchNorm layers, and K-Fold training, we found that our initial model performed the best, with a single hidden layer of 20 neurons. We achieved an ROC_AUC score of 0.750, falling short of our LGBM model, which had a score of 0.752.

We almost exhausted all of the options we could think of to improve our model, and if we had more time, we think that the improvement we would get by working on the model itself would be small compared to the gains we could achieve by continuing feature engineering.