Ryan Herrin, Luke Stodgel
DS7333
04/06/2023

Binary Classification using a Dense Neural Network

Introduction

In this project we used a dense neural network model to predict a binary target while trying as much as possible to reduce incorrect predictions and money lost. Each false positive prediction cost us $35 and each false negative prediction cost us $15. Every correct prediction cost us $0.

Methods

About the Dataset

Our data set did not have very much descriptive information, so we can't say what this data is about. The data had three categorical columns and 47 continuous columns. The categorical columns had values such as names of months, days of the week, and names of continents. The continuous columns were made up of negative and positive decimal numbers.

Our dataset had 160,000 rows and 51 columns, including the binary target variable, "y".

Kfold Cross-validation

All of our model metrics are based on 160,000 out-of-fold predictions on our data set. We were able to achieve 160,000 out-of-fold predictions using a custom 5-fold CV for-loop that utilized sklearn's Kfold function. Each cross-validation iteration performed predictions on 32,000 entries and after five folds we combined the predictions from each fold and produced our model's metrics. In our Kfold function, we set shuffle=True to reduce the potential for bias from the order of the data in the data set. And, Kfold creates each fold without replacement so we can be sure that predictions were made on all 160,000 rows in the data set.

Data Preprocessing

First, we loaded our data into a dataframe using pandas' read_csv function. Next, using our dataframe, we separated out our X and y variables. Then we performed several transformations in order to get our X variables into their final form.

First, we saw that two columns, "x37" and "x32" had "$" and "%" characters mixed in with numeric data which caused issues when trying to scale them, so we removed those characters from those column rows. After that, we created a categorical data frame consisting of only the dataset's categorical columns "x24", "x29", and "x30". Then we used sklearn's SimpleImputer() to impute the mode of each categorical column and fill in any missing values in those columns. They each only had about 30 missing values. Next we used pd.get_dummies() to one hot encode these categorical columns. We set the parameter "drop_first" = true in the get_dummies() function to attempt to reduce multicollinearity in the one-hot-encoded columns and it ended up boosting our performance by about 0.003% accuracy. At this point we had our final categorical data frame.

In order to prepare our continuous features, we first created a continuous only data frame. Next we used KNNImputer(n_neighbors=3) to fill in missing values in those columns. We also tested values 1-5 for n_neighbors with our highest performing model and we did not see noticeably different results, so we stuck with n_neighbors = 3.

After that we concatenated the continuous data frame and the categorical data frame and we used StandardScaler to scale the full X dataframe. Lastly, we converted this entire data frame from float64 to float32 in order to save on time and RAM when training our models.

Addressing the Missing Data

To reiterate, we imputed our categorical features using the mode of those columns and for our continuous features we imputed missing data using KNNImputer(n_neighbors=3). There were between 20 and 50 missing values per column and every column except the target column had missing values. This strategy worked very well for us as you will see from our model metrics shown in the results section of this paper.

Building our Dense Neural Network Structure

Now we will go over how we tested different neural network structures and how we ended up with our final model.

We started our trials with a baseline dense, sequential, neural network model made of one dense input layer, size 32, activation function relu, input size 64, three dense layers, each with relu activation functions, sizes 32, 64 and 128, respectively, one dropout(0.3) layer, and an output layer that used a sigmoid activation function.

The reason we used relu activation functions in our input and central layers was because from what we've seen, relu is commonly used for this and is less computationally intensive than other options like sigmoid or tahn.

Our purpose for including a dropout layer was to try to prevent overfitting on the training data. The dropout layer randomly drops some of the neurons in a layer during training and makes the model focus on more robust and generalizable features rather than any one specific neuron.

Finally, we chose a sigmoid activation function for our output layer because our target variable was binary. We purposely did not use a softmax activation function for our output layer because softmax is better suited for multiclass classification tasks.

For our base model's compile function, we used the Adam optimizer with a learning rate of 0.001. We set our loss to binary_crossentropy and metrics to accuracy. We used binary_crossentropy because it was the most appropriate loss function given our binary target variable. Also, we did not test using loss functions mean_squared_error or mean_absolute_error because we had a binary target variable and thus they were not applicable. For the metrics variable we chose 'accuracy' just for the sake of keeping track of each model's accuracy at each epoch.

Next, we created an early-stopping callback with the monitor parameter set to 'val_loss' and patience set to 5. Our reasoning for including early-stopping was to prevent overfitting on the training data and to save on computing resources; early-stopping will stop training after the model has stopped improving. We tested using monitor='val_accuracy' instead of monitor='val_loss' and it had no effect on our model's metrics, so we just stuck with monitor='val_loss'.

Finally, we fit our model with 1000 epochs, batch_size=100, we gave it validation data parameters, and set callbacks to ['early_stopping']. Additionally we stored the output from our model's fit function in a variable so that we could use the model's history to plot the model's loss, validation loss, accuracy, and validation accuracy at each epoch, which we will show in the results section of this paper.

After we ran each model, we calculated the total amount of money our model cost us by multiplying the number of false positives by 35 and the number of false negatives by 15.

Our base model returned 96.6% accuracy and $129,735 total cost when tested on all 160,000 out-of-fold predictions. After testing several different combinations of numbers and sizes of dense and dropout layers, we were never able to beat our base model's metrics.

Here is an overview of a few different models we tested before we concluded on our final model.

The different models we tested consisted of various combinations of different amounts and sizes of dense layers, different learning rates, different amounts and sizes of dropout layers and we also tried using an RMSprop optimizer instead of Adam. Assume the layer used the activation function, 'relu', unless specified otherwise. Here are a few of the different combinations of parameters that we tested:

1. Dense input layer 1 size 32, input shape size 64, Dense layer 1 size: 32, dense layer 2 size: 64, dense layer 3 size: 128, dropout 1 size: 0.3, output layer 1 activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
2. Keeping everything else constant, except: dense input layer 1 size 128 dense layer 1 size: 128, dense layer 2 size: 128, dense layer 3 size: 128.
3. Keeping everything else constant, except: dense input layer 1 size 256 dense layer 1 size: 256, dense layer 2 size: 256, dense layer 3 size: 256.
4. Keeping everything else constant, except: dense input layer 1 size 512 dense layer 1 size: 512, dense layer 2 size: 512, dense layer 3 size: 512.
5. Keeping everything else constant, except: dense input layer 1 size 512 dense layer 1 size: 512, dropout layer 1 size 0.3, dense layer 2 size: 512, dropout layer 2 size 0.3, dense layer 3 size: 512, dropout layer 3 size 0.3.

If you want the full list of the different model structure combinations we tried, please find them in the appendix of this paper.

Eventually we concluded on our highest performing model and it looked like this:

```
Sequential ([
        Dense input layer size 32, activation relu, input shape 64,
        Dense layer 1 size 32, activation relu,
        Dense layer 2 size 64, activation relu,
        Dense layer 3 size 128, activation relu,
        Dropout layer 3 size 0.3,
        Dense output layer size 1, activation sigmoid
]
Optimizer Adam(0.001), loss='binary_crossentropy', metrics=['accuracy']
Early-stopping monitor: 'val_loss', patience: 5.
Epochs: 1000, batch_size: 100.
```

There are a few reasons why we think our less-complex base model outperformed the more-complex models we tested.

- With the base model, using fewer layers and fewer neurons per layer could make it easier for the model to learn the relevant patterns in the data.
- Using a smaller input layer could make the model less prone to overfitting and more resistant to input feature noise.
- With the success we saw when using a lower dropout rate, we suspect that we were over-regularizing in the more complex models.

Using an Optimal Threshold for Predicted Probabilities vs. the Default 0.5 Threshold

Since our sigmoid function output layer outputs an array of predicted probabilities, we used that to our advantage and found an optimal prediction threshold. This consistently increased the performance of our model.

We tested using optimal thresholds based on the highest f1_score, accuracy, precision and recall, and the best threshold for accuracy ended up giving us the best results and saving us the most money.

In our view, finding the best threshold comes with trade offs. Sometimes it might just be best to use a prediction threshold of 0.5 instead of creating a custom threshold function. In the context of this specific project and dataset, a custom threshold proved to be beneficial but sometimes it can be a waste of time.

Results

Figure 1 (below) is a graph showing the prediction scores for each threshold based on our final model's 160,000 predictions. For our final model's predictions, we used the optimal prediction threshold that gave us the highest accuracy score. In this run in particular, our threshold was 0.42. As mentioned in the methods section, we tested using optimal thresholds based on the highest f1_score, precision and recall too, but the best threshold for accuracy score ended up giving us the best results and saving us the most money.

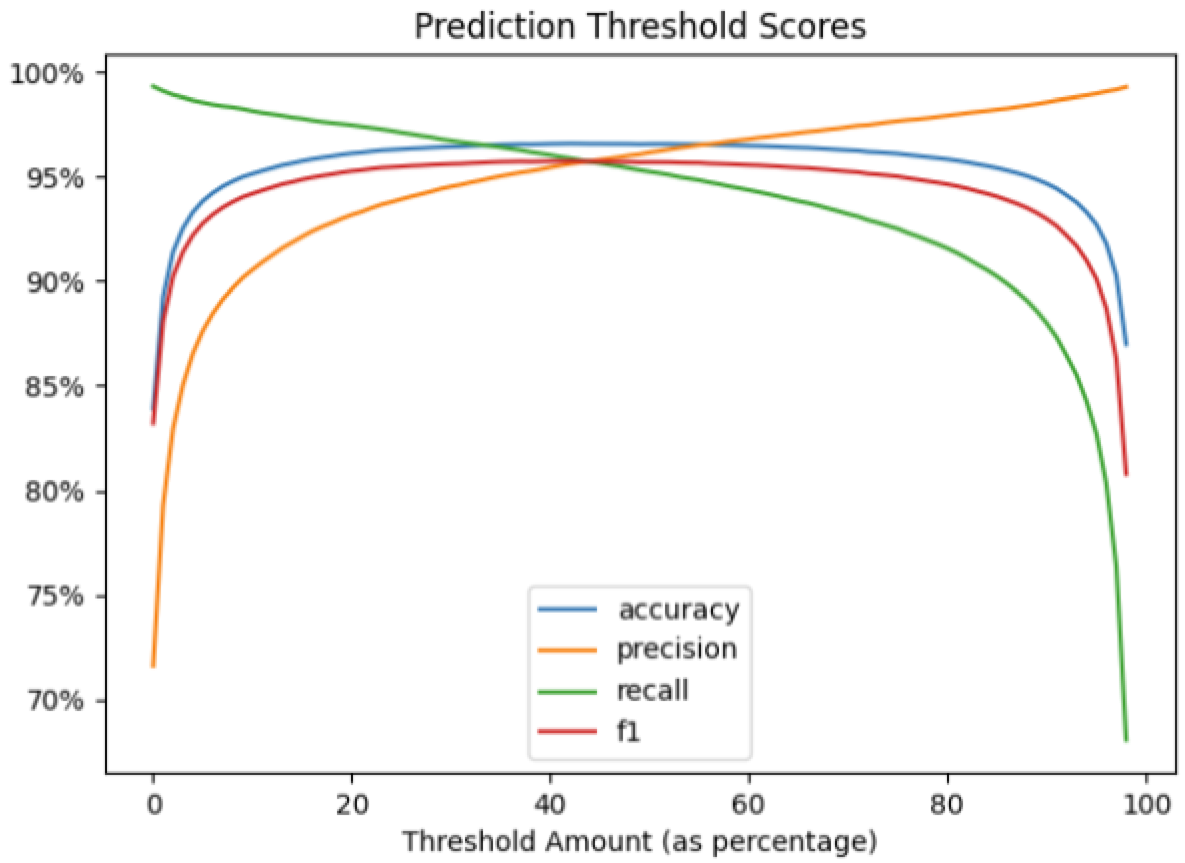

Figure 1 - Prediction Threshold Scores

Figure 2 (below) is a series of charts showing our final model's training losses and validation losses at each epoch for each of the five cross-validation folds. Looking at the training loss, for each fold, it consistently decreased. That suggests that our model was not overfitting on the training data. For each fold, our training loss ended at around 0.08. For our validation loss, the stopping point varied across each fold. The model stopped training because of early-stopping between 30 and 37 epochs depending on the fold. And, this model's final validation_loss was consistently between 0.1 and 0.12. We used validation loss for our early-stopping metric and this can be seen in this graph. More specifically, for each cross-validation fold, when the validation loss of an epoch did not decrease after five attempts in a row, the model stopped there and moved on to the next fold or finished training entirely.

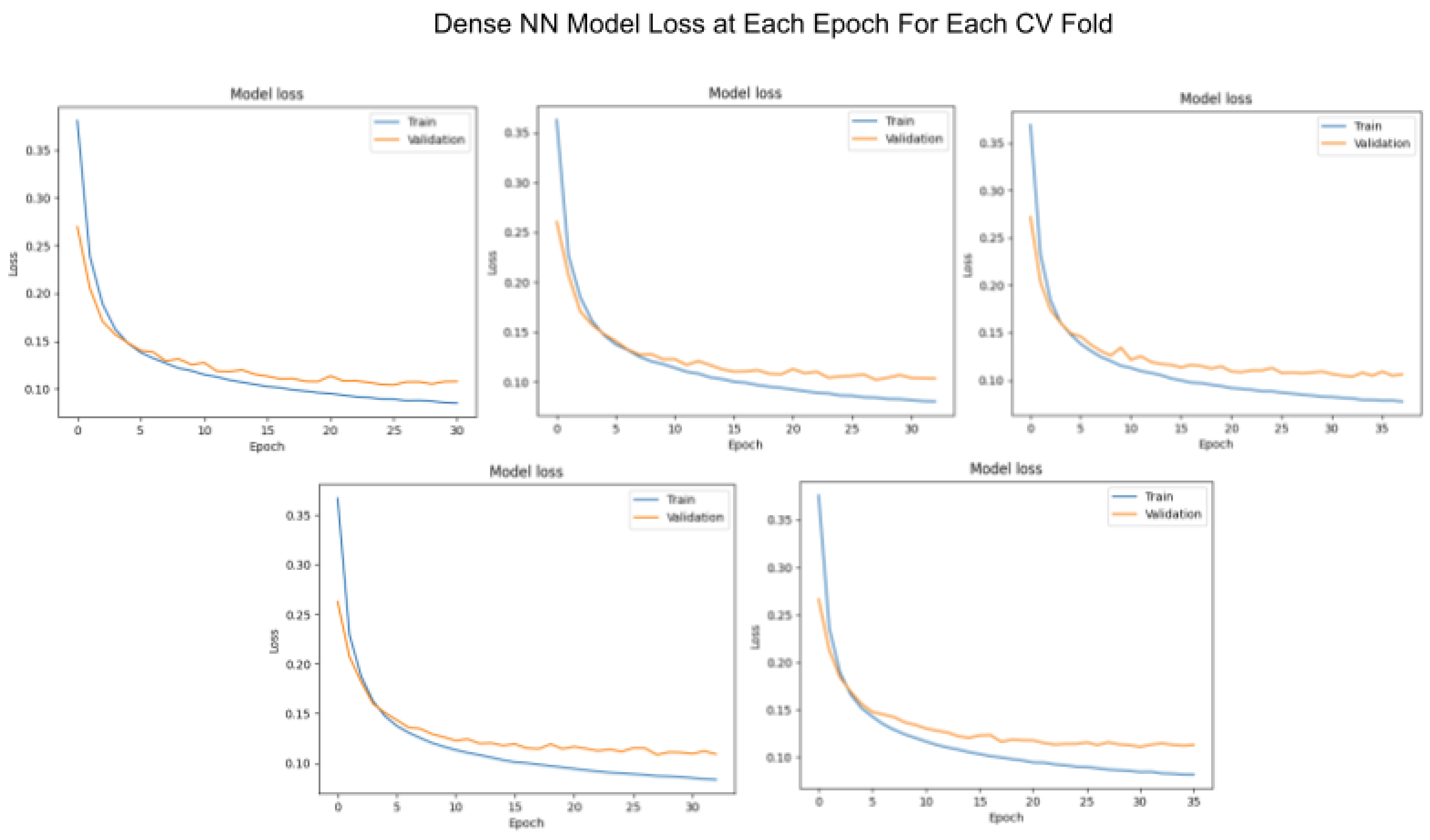

Figure 2 - Loss at Each Epoch For Each CV Fold

Figure 3 (below) is a series of charts showing our final model's training accuracies and validation accuracies at each epoch for each of the five cross-validation folds. Each fold trained for between 31 and 38 epochs and ended with about 0.97 training accuracy and between 0.96 and 0.97 validation accuracy. In these charts, we can see that our model's early-stopping mechanism worked as intended and stopped our model's training before overfitting hurt our model's performance.

Dense NN Model Accuracy at Each Epoch For Each CV Fold

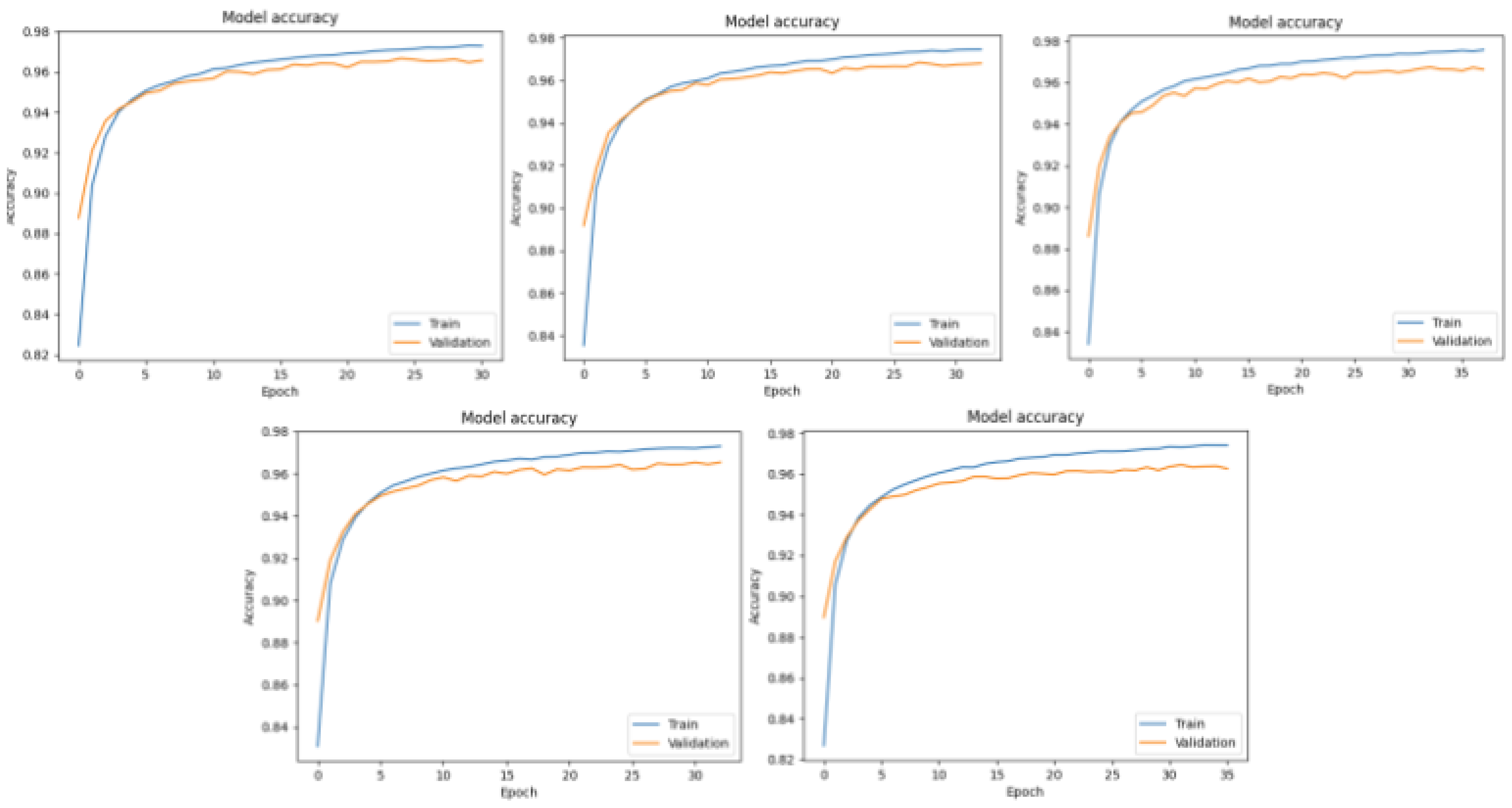

Figure 3 - Accuracy at Each Epoch For Each CV Fold

Figure 4 (below) shows our model's average validation accuracy and loss for its 160,000 predictions. Our model's average validation accuracy was 0.966 and its average validation loss was 0.108.

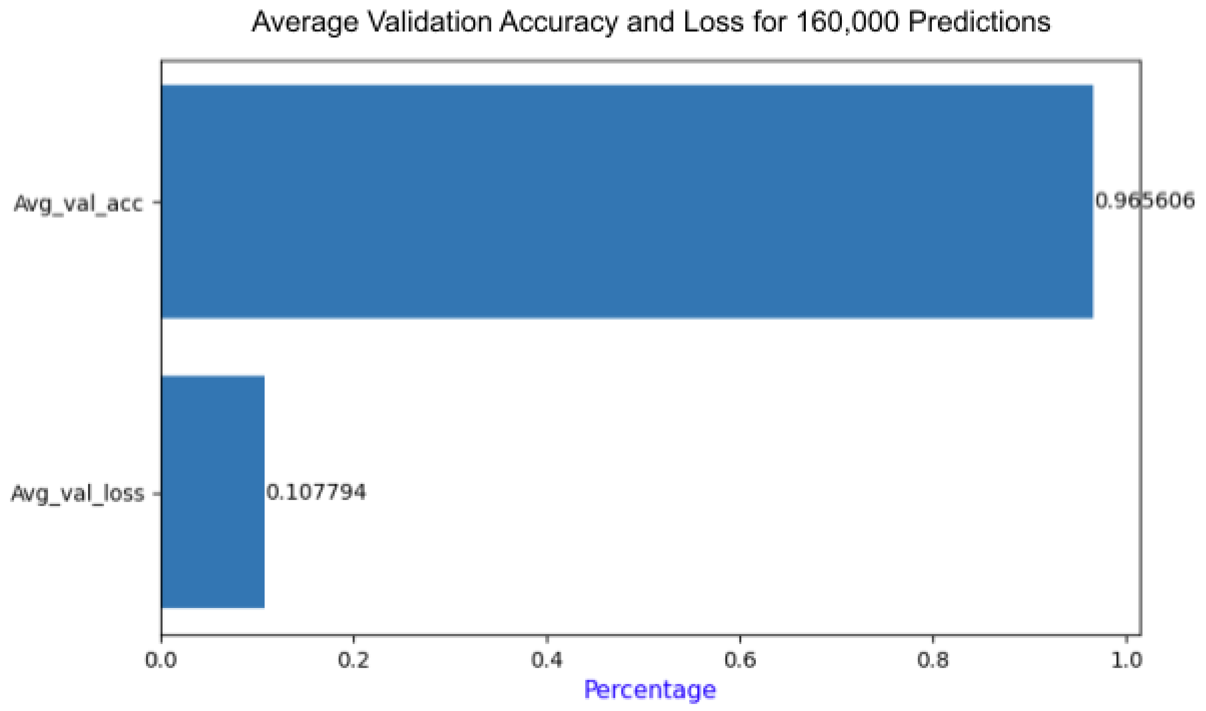

Figure 4 - Average Validation Accuracy and Loss for 160,000 Predictions

Figure 5 (below) shows the metrics for our final model. These metrics are based on our model's 160,000 out-of-fold predictions. Our model returned an accuracy of 0.966, precision of 0.955, recall of 0.959, and f1_score of 0.957.

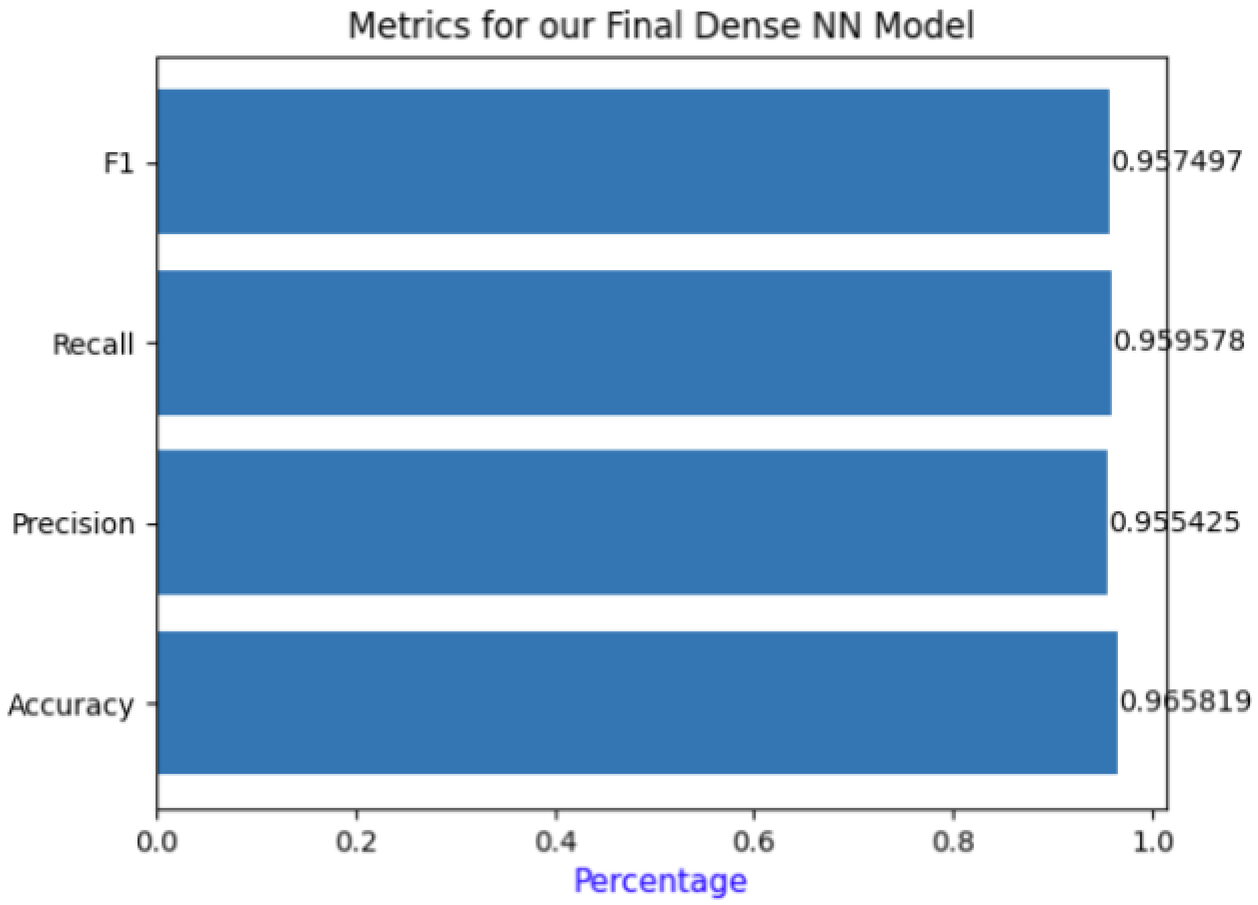

Figure 5 - Final Model Metrics

Figure 6 (below) is a confusion matrix from our final model. Our model accurately predicted label 0 (92,929 / 95,803) times and accurately predicted label 1 (61,602 / 64,197) times. We had 2,874 false positive classifications of label 0 compared to the 2,595 false negative classifications of label 1. Our model's total misclassification rate was 0.034 or about 3.4% and we are very happy with that. Our model did extremely well classifying both the 0 and 1 target class labels.

Confusion Matrix for our Final Dense NN Model

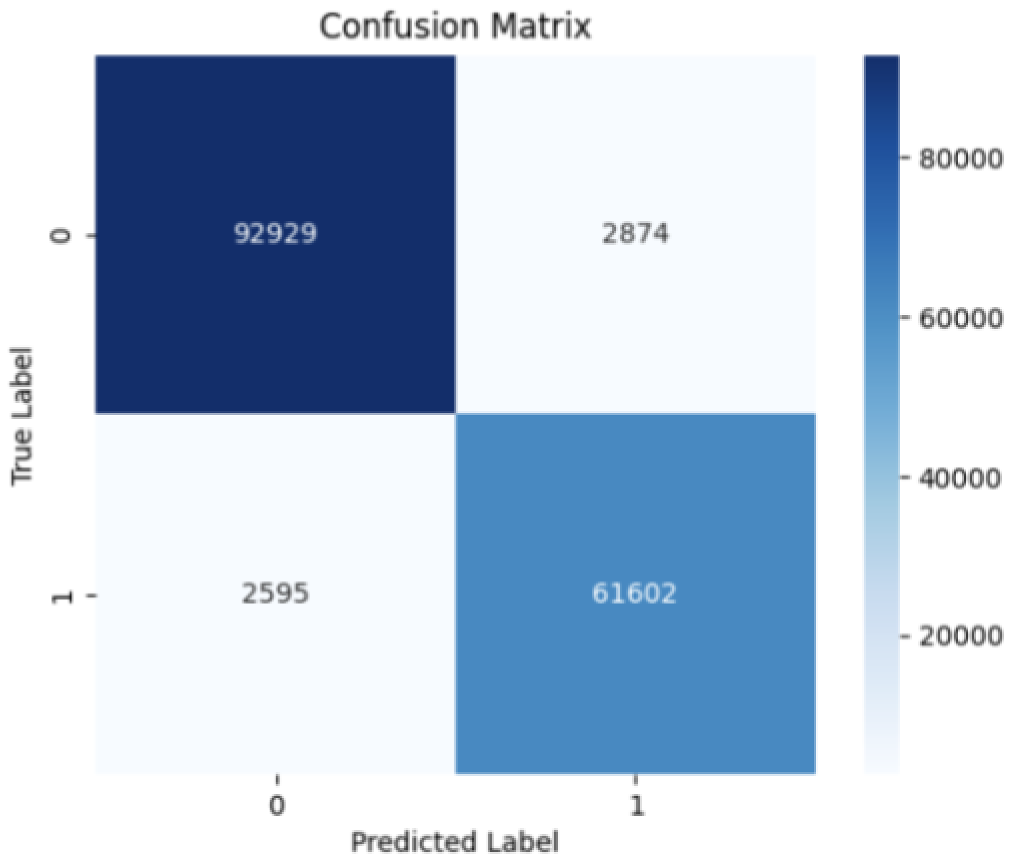

Figure 6 - Final Model Confusion Matrix

Conclusion

In conclusion, we created a dense neural network to perform binary classifications on 160,000 out-of-fold predictions. Our highest performing model looked like this:

Sequential ([
    Dense input layer size 32, activation relu, input shape 64,
    Dense layer 1 size 32, activation relu,
    Dense layer 2 size 64, activation relu,
    Dense layer 3 size 128, activation relu,
    Dropout layer 3 size 0.3,
    Dense output layer size 1, activation sigmoid
]
Optimizer Adam(0.001), loss='binary_crossentropy', metrics=['accuracy']
Early-stopping monitor: 'val_loss', patience: 5.
Epochs: 1000, batch_size: 100.

Our model returned an accuracy of 96.6%, precision of 95.5%, recall of 95.9%, and an f1_score of 95.7%.

Finally, our total cost, based on false-positives*$35 and false_negatives*15, was $129,735.

Appendix A

Link to main script:
https://github.com/Abillelatus/QTW-Case-Studies/blob/main/CaseStudy_7/MSDS-7333-CaseStudy-7.py

Link to full CaseStudy_7 full contents:
https://github.com/Abillelatus/QTW-Case-Studies/tree/main/CaseStudy_7

Continuation of model combinations tested:

6. **Dense input layer 1 size 16, input shape 64**, **Dense layer 1 size: 16**, dense layer 2 size: 64, dense layer 3 size: 128, dropout 1 size: 0.3, output layer 1 activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
7. Dense input layer 1 size 16, input shape 64, **Dense layer 1 size: 32**, dense layer 2 size: 64, dense layer 3 size: 128, dropout 1 size: 0.3, output layer 1 activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
8. Dense input layer 1 size 16, input shape 64, **Dense layer 1 size: 16**, **dense layer 2 size: 32**, **dense layer 3 size: 64**, dropout 1 size: 0.3, output layer 1 activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.

9. Dense input layer 1 size 16, input shape 64, **Dense layer 1 size: 32**, **dense layer 2 size: 64**, dense layer 3 size: 64, dropout 1 size: 0.3, output layer 1 activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
10. ***This was our second highest performing model**
    Dense input layer 1 size 32, input shape 64, **Dense layer 1 size: 32**, **dense layer 2 size: 64**, dense layer 3 size: 64, dropout 1 size: 0.3, output layer 1 activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
11. **Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.1**, dense layer 2 size: 512, **dropout 2 size: 0.2**, **dense layer 3 size: 512**, **dropout 3 size: 0.3**, **dense layer 4 size: 512, dropout 4 size: 0.2**, output dense layer 1 size: 1, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
12. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, **dropout 4 size: 0.3**, output dense layer 1 size: 1, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
13. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, **dropout 4 size: 0.1**, output dense layer 1 size: 1, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
14. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, **dropout 4 size: 0.1**, output dense layer 1 size: 1, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
15. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, **dropout 4 size: 0.2**, output dense layer 1 size: 1, activation function: sigmoid, optimizer: Adam, **learning rate: .0001**, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
16. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, dropout 4 size: 0.2, output dense layer 1 size: 1, activation function: sigmoid, optimizer: Adam, **learning rate: .01**, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
17. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, dropout 4 size: 0.2, output dense layer 1 size: 1, activation function: sigmoid, **optimizer: RMSprop**, learning rate: .01, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.

18. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, dropout 4 size: 0.2, output dense layer 1 size: 1, activation function: sigmoid, optimizer: RMSprop, **learning rate: .001**, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
19. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, dropout 4 size: 0.2, output dense layer 1 size: 1, activation function: sigmoid, optimizer: RMSprop, **learning rate: .0001**, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
20. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, **dropout 4 size: 0.3**, **Dense layer 5 size: 512, dropout 5 size: 0.3, Dense layer 6 size: 512, dropout 6 size: 0.2**, output dense layer 1 size: 1, activation function: sigmoid, optimizer: RMSprop, **learning rate: .001**, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
21. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, **dropout 4 size: 0.3**, **Dense layer 5 size: 512, Dense layer 6 size: 512,** output dense layer 1 size: 1, activation function: sigmoid, optimizer: RMSprop, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
22. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, dropout 4 size: 0.3, Dense layer 5 size: 512, Dense layer 6 size: 512, output dense layer 1 size: 1, activation function: sigmoid, **optimizer: Adam**, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
23. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, dropout 4 size: 0.3, Dense layer 5 size: 512, **dropout 5 size: 0.3**, Dense layer 6 size: 512, output dense layer 1 size: 1, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
24. Dense input layer size 512, input shape 64, dense layer 1 size: 512, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, dropout 4 size: 0.3, Dense layer 5 size: 512, dropout 5 size: 0.3, Dense layer 6 size: 512, **dropout 6 size: 0.2**, output dense layer 1 size: 1, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
25. Dense input layer size 512, input shape 64, dense layer 1 size: 32, dropout 1 size: 0.3, dense layer 2 size: 512, dropout 2 size: 0.2, dense layer 3 size: 512, dropout 3 size: 0.1, dense layer 4 size: 512, dropout 4 size: 0.3, Dense layer 5 size: 512, dropout 5 size: 0.3, Dense layer 6 size: 512, **dropout 6 size: 0.2**, output dense layer 1 size: 1, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.

26. **Dense input layer 1 size 32, Dense layer 1 size: 32, dropout layer 1 size: 0.3, dense layer 2 size: 64, dropout layer 2 size: 0.2, dense layer 3 size: 128, dropout 3 size: 0.1, dense layer 4 size: 256, dropout 4 size: 0.1**, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
27. Dense input layer 1 size 32, Dense layer 1 size: 32, dense layer 2 size: 64, dense layer 3 size: 128, **dropout 1 size: 0.1**, dense layer 4 size: 256, **dropout 2 size: 0.3**, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.
28. Dense input layer 1 size 32, Dense layer 1 size: 32, dense layer 2 size: 64, dense layer 3 size: 128, dense layer 4 size: 256, **dropout 1 size: 0.3**, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_crossentropy, early-stopping monitor: 'val_loss', patience: 5.