Loan Approval Prediction Comparison

Jacob Brubaker, Trevor Klein, Nicholas Dibello-Hitta

***Abstract* - This paper investigates the performance of three machine learning models—Logistic Regression, Support Vector Machine (SVM), and Neural Networks—in predicting loan acceptance outcomes. Using a dataset from Kaggle containing age, gender, credit score, education level, loan amount, and approval outcome, the models were trained and evaluated on their ability to accurately classify loan applications as accepted or rejected. Metrics such as accuracy, precision, and recall were used to assess model performance. Logistic Regression demonstrated simplicity and interpretability but was outperformed by the more flexible SVM in terms of precision and recall. Meanwhile, the Neural Network had the best performance in capturing complex patterns in the data. This study highlights the successes and failures of each model, the testing done to optimize each model on the dataset, and a comparison of the models' accuracies.**

I. Introduction

Determining the most effective machine learning model for a given task is critical to optimizing predictive accuracy and performance. In this study, we investigate the effectiveness of three popular machine learning algorithms—Logistic Regression, Support Vector Machine (SVM), and Neural Networks—in predicting loan acceptance outcomes.

Using a dataset sourced from Kaggle containing attributes such as age, gender, credit score, education level, loan amount, and approval outcome, each model was trained and evaluated to identify the best-performing approach. The comparison is based on metrics like accuracy, precision, recall, and F1-score. This paper aims to identify the model that achieves the highest predictive performance and provide insights into its suitability for financial decision-making scenarios.

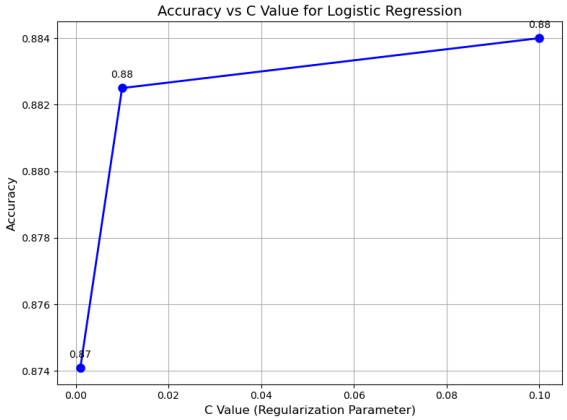
II. Logistic Regression Model

Logistic Regression models are among the simpler of the machine learning models, oftentimes limited in their usefulness to simpler datasets, with more clearly defined relationships. We wanted to use the logistic regression model as a baseline against which to compare the Support Vector Model and Neural Network model, to see how much better they really were.

The first iteration of a logistic regression model was based on the manually computed model provided in HW3.py by our professor, Dr. Xiao-Bai Liu. It was modified to use our new dataset, and utilized a train/test split of 80/20. This basic model achieved an overall accuracy score of 77.67%, and an attempt to improve that score by scaling the dataset (which helped the later models converge) decreased the accuracy to 75.10%.

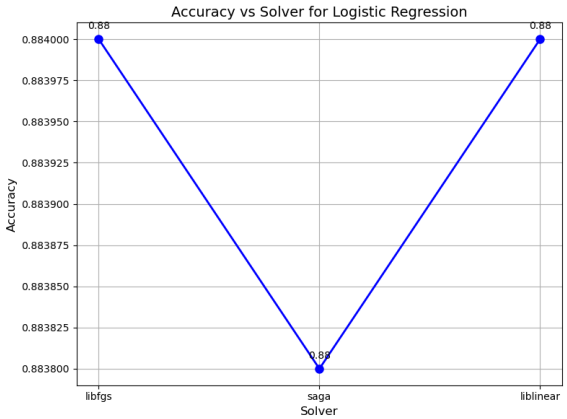
The second iteration of the logistic regression model utilized scikit-learn's LogisticRegression model, which achieved a baseline accuracy, without any hyperparameter tuning or data scaling, of 88.31%. Then, after scaling the dataset properly, the model converged with an accuracy of 88.34%.

The first hyperparameter that was tested was the regularization parameter, or “c” value. This parameter dictates the regularization of the model, which in turn affects how much the model tends to overfit or underfit the data. The three C values tested were 0.1, 0.01, and 0.001. Here are the results:



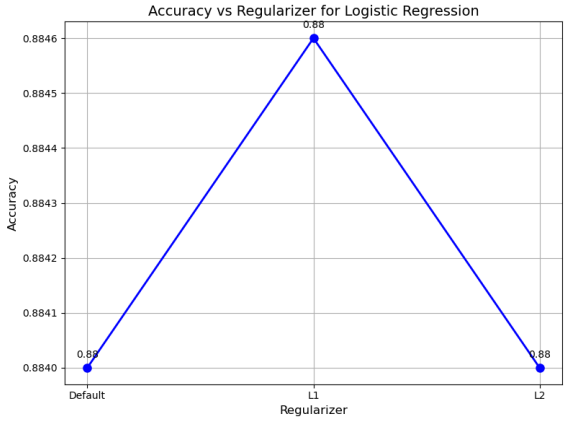
The highest accuracy was achieved with a c value of 0.1, giving us an accuracy of 88.4%. Values larger and smaller than the ones listed were also tested but produced poor results.

The next hyperparameter that was tested was the solver that the model used. The solver affects how the loss function is calculated, and is ultimately the main algorithm that determines how well the model is trained to fit the data. All three possible solvers were tested, “lbfgs”, “saga”, and “liblinear”. Here are the results:



All three solvers performed similarly well, with a maximum delta between each performance of only 0.3%. The lbfgs and liblinear solvers performed equally well, and the liblinear solver was only selected for the final model because of the interoperability with the next hyperparameter.

The final hyperparameter that was tested was the regularizer. The three possible types of regularizers were L1 (Lasso), L2 (Ridge), and Elastic Net. Regularizers work by constraining the coefficient values, further preventing overfitting than just modifying the c value alone. Here are the results:



One caveat to these results is that not all regularizers worked with the best solver algorithms. The L1 algorithm only worked with the liblinear solver, the L2 regularizer worked with both lbfgs and liblinear solvers, and there was no way to get the elastic net regularizer to work with our model and dataset, leading to it ultimately being abandoned. While both the default (no regularization) and L2 regularizer performed equally well, the best performer was the L1 (Lasso) regularizer (though only by 0.05%). Additionally, because this regularizer only worked with the liblinear solver, this solver was ultimately chosen to be used in the final testing.

After manually testing each of the three hyperparameters, GridSearchCV was then used to verify the results, and the results were consistent with previous testing, the best hyperparameters for the model were a c of 0.1, the liblinear solver, and the L1 regularizer.

Lastly, K-fold cross-validation was implemented to attempt to improve the training of the model by using all of the available data for both training and testing, though it had negligible effects on the performance of the model (accuracy decreased by 0.01%), and even the best-performing fold only had a 0.14% improvement over the regular model.

In summary, the logistic regression model's best performance had an accuracy of 88.45%, and hyperparameter tuning only showed an improvement in the model's accuracy of 0.14%. While it would be possible to continue to extract miniscule improvements in performance by testing a larger set of c values, the Logistic Regression model is ultimately limited in its abilities to accurately predict the behavior of complex datasets like ours. This leaves almost all potential for improvements in accuracy to the SVM and Neural Network models.

The L2 (Ridge) regularizer worked the best

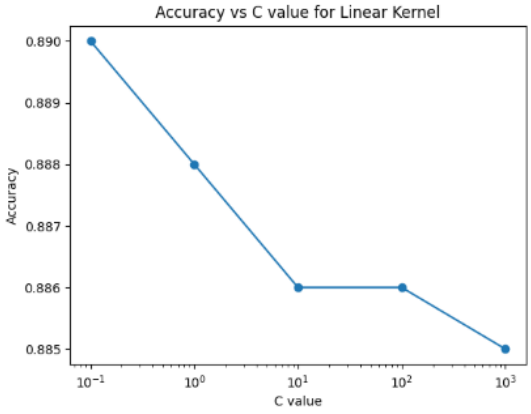
III. Support Vector Model

Support Vector Machines (SVM) are widely recognized for their effectiveness in classification problems, particularly when dealing with complex and high-dimensional datasets. For this study, an SVM model was implemented to predict loan acceptance based on the provided Kaggle dataset. Of the 45,000 data points, 20% of the points were reserved for testing the model. This section details the methodology used in constructing the SVM model, the features selected, and the results obtained through training and evaluation.

To start off, in order to select the most effective kernel for the data, testing was conducted to identify which kernel had the best overall performance. To do this, Scikit Learn libraries were imported and three separate kernels were tested with the cleaned dataset: linear, rbf, and polynomial. All three of these models were tested on the same random testing and training data from the dataset using the default settings of c = 1 and gamma = 1. These were the results:

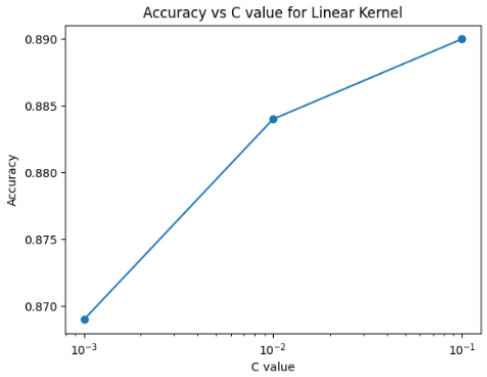
Linear kernel accuracy: 0.888
Rbf kernel accuracy: 0.896
Poly kernel accuracy: 0.895

With all of the various forms of kernels being extremely close, there needed to be more testing done for each kernel, particularly with different c values for linear, different gamma values for rbf, and different degrees for polynomials. Starting off with more testing for the linear kernel, linear kernels revolve around a hyperparameter of c, so the kernel will be tested using various c values. The first set of c values used were 0.1, 1, 10, 100, and 1000. Here were the results:

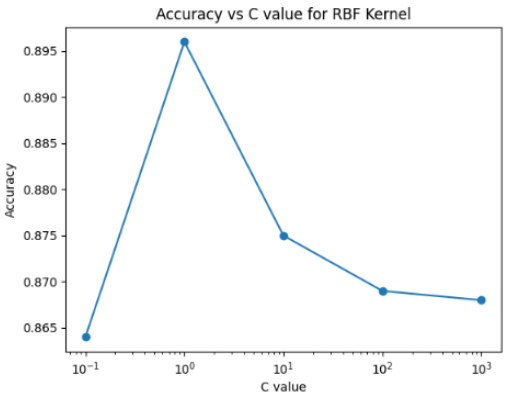


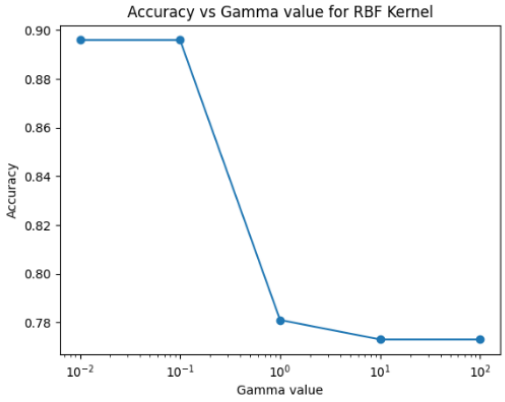
The highest achieved accuracy was 0.890 with a c value of 0.1, this slightly raised

the original accuracy of the linear kernel by 0.002. The graph reveals a downward trend in c values, however, where as c is increased, the overall accuracy of the model is decreased. After this, further testing was done to figure out if smaller c values led to a higher accuracy, so the next tests done used 0.001, 0.01, and 0.1. Here were the results:

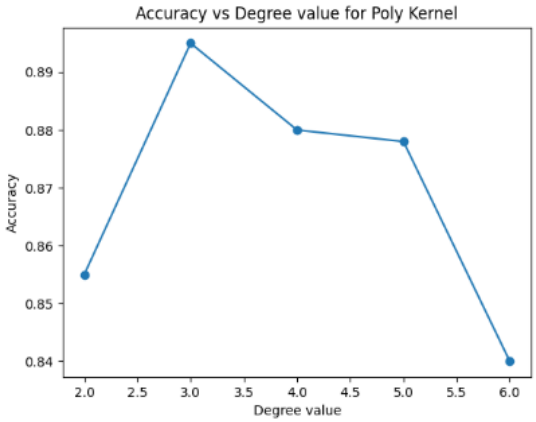


The graph reveals how the trend falls off at 0.1, where smaller C values after 0.1 show another decrease in accuracy, revealing that C = 0.1 is the most optimal value for this data set with an accuracy of 89%. The next kernel to be tested was the rbf kernel. The rbf kernel will have two separate tests, testing c values of 0.1, 1, 10, 100, and 1000 whereas gamma values tested will be 0.01, 0.1, 1, 10, and 100. Here are the results:





The most successful hyperparameters from these tests were c = 1 and gamma = 0.1, using these hyperparameters together, the model achieved an accuracy of 0.896, overall slightly more successful than the optimized linear kernel. Now to test the poly kernel with different degree values. The degree values tested were 2, 3, 4, 5, and 6. Here were the results:



The degree of 3 had the highest overall accuracy on the dataset, attaining an accuracy of 0.895, which was slightly worse than the rbf kernel but slightly better than the linear kernel. After manual testing to find the best hyperparameters, GridSearchCV from the Scikit Learn library was the next step to see if the hyperparameters were optimal. The GridSearchCV functions test out hyperparameters for kernel and after doing

testing for each of the kernels, these were the results:

Best parameters for RBF kernel: {'C': 1, 'gamma': 0.1}
Best cross-validation accuracy for RBF kernel: 0.9014999999999999
Best parameters for polynomial kernel: {'C': 1, 'degree': 3}
Best cross-validation accuracy for polynomial kernel: 0.8932499999999999
Best parameters for linear kernel: {'C': 1}
Best cross-validation accuracy for linear kernel: 0.8905000000000001
Test set accuracy for RBF kernel: 0.896
Test set accuracy for polynomial kernel: 0.895
Test set accuracy for linear kernel: 0.888

The results from manual testing and GridSearchCV indicate that the RBF kernel achieved the highest cross-validation accuracy (0.9015) among the tested kernels, followed closely by the polynomial and linear kernels. This suggests that the RBF kernel is better suited for capturing non-linear patterns in the dataset compared to the other kernels.

When evaluating test set accuracy, the RBF kernel slightly outperformed the polynomial kernel (0.896 vs. 0.895), while the linear kernel trailed behind with an accuracy of 0.888. The consistent performance of the RBF kernel across both cross-validation and test data highlights its robustness and suitability for this classification problem.

To assess whether the RBF kernel was overfitting, training and test set accuracy were compared. The training set accuracy (94.85%) is significantly higher than the test set accuracy (89.6%), suggesting potential overfitting. Additionally, the cross-validation scores ranged from 89.3% to 92.5%, with a mean accuracy of 90.15%, further suggesting the model performs variably depending on the specific validation set.

This discrepancy may indicate that the model is too tightly fitted to the training data, capturing noise or overly complex patterns. To address this, further experiments could involve reducing the model's complexity by lowering the C value, increasing the tolerance for margin violations, or exploring additional preprocessing techniques to reduce noise in the dataset.

In order to take a deeper look at the performance of the model on the testing data, here is a look at the confusion matrix that was conducted on the testing data.

Confusion Matrix	Actually Positive	Actually Negative
Predicted Positive	6696	275
Predicted Negative	501	1528

On this run, the rbf kernel achieved an accuracy of 91.4%, a precision of 96.06% and a recall of 93.04%. The high precision of the model indicates that the model is good at avoiding false positives in the data and the high recall indicates that model is good at minimizing false negatives. Overall, with a good balance between precision and recall, it is unlikely that the model has been overfitted to the training data and also means that the model generalizes well to unseen data.

In conclusion, the Support Vector Machine (SVM) with the RBF kernel demonstrated the best performance in predicting loan acceptance, achieving the highest accuracy and cross-validation scores among the kernels tested. The optimized model with C = 1 and gamma = 0.1 performed consistently well, with a peak accuracy of 91.4% on the test set. While the RBF kernel exhibited slight overfitting as indicated by the disparity between training and test accuracies, its strong precision and recall suggest that it effectively captures relevant patterns in the dataset. Future improvements could focus on reducing overfitting by adjusting hyperparameters or employing further data

preprocessing techniques. Overall, the RBF kernel's robustness makes it the most suitable model for this classification task, providing valuable insights into loan acceptance prediction.

IV. Neural Network

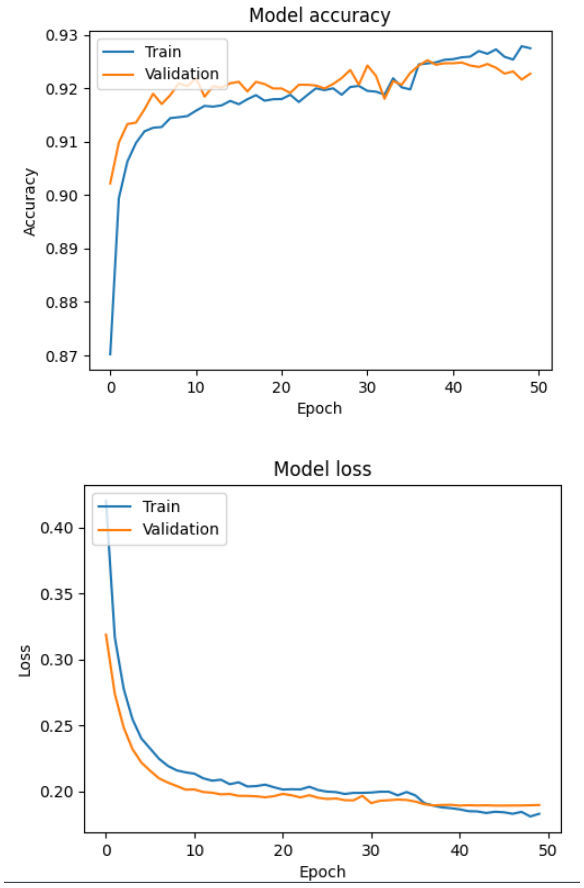
Neural Networks are a powerful tool when it comes to solving classification problems, especially when it comes to working with large and complex datasets. In this study, the neural network was utilized to predict loan acceptance based on information from our dataset. The data was split into two parts, with 80% being used for training and validation and the other 20% being used for testing purposes. When looking to optimize the network a variety of configurations was explored including differing numbers of layers, activation functions, dropout rates, and regularization techniques. Additionally other strategies such as early stopping and learning rate reduction were implemented in an attempt to prevent overfitting and improve training efficiency. Then to evaluate the models visualizations of loss, accuracy, confusion matrices, and AUC graphs were created alongside the raw output data to help effectively find the best configuration of the neural network. In this section the visualizations and data will be presented to help explain the results and findings.

To start off, a base build and train was created for the neural network. This base model is what the other build and train models are based off of with each of the other four having their own slightly different modifications to differentiate it from the base model. All five of the build and train models take in five parameters in their respective function calls. These parameters are the number of neurons per layer, the dropout_rate, the regularization, the learning_rate, and epochs. Each of these parameters are called configurations in the code of which there are ten that are tested on each build and train model. Going back to the base model, it has two hidden layers, one Dense layer and one Dropout layer. The Dense layer in the base model uses the relu activation function and an L2 regularizer with the regularization rate specified in the configuration. These layers are applied to the neurons specified in the configurations. Additionally, there is one more dense layer for output that uses the sigmoid activation function to give it the classification feel of 0 or 1. Finally the model is compiled and trained with the Adam optimizer and the binary cross-entropy loss function. The four other models follow the same overall structure, each having one variation. For model two that is adding an additional hidden layer to do batch normalization. For model three that is replacing the relu activation function with tanh. For model four a keras callback, ReduceLRonPlateau is added. This callback is designed to improve the models performance by reducing the learning rate when a metric stops improving or as the name suggests, hits a plateau. Finally, model five makes use of a different callback, EarlyStopping. This callback works in a similar way to the ReduceLRonPlateau, but this one will stop the training if after an allotted amount of epochs no improvement is seen in the monitored variable. Now that the differences between the models are covered, let's look at the differing configurations that each model was run on. Like the parameters of the build and train functions these configurations are the number of neurons per layer, the dropout_rate, the regularization, the learning_rate, and epochs. For the number of neurons per layer the neurons ranged from 512 down to 32 and the number of layers from 2 to 4. The dropout rate ranged from 0.5 to 0.2 and both the regularization rate and learning rate varied from 0.01 to 0.0001. Finally the epochs ranged from 10 to 50. Having these varying configurations allowed the model to systematically explore how changing

hyperparameters would affect the overall performance of the model.

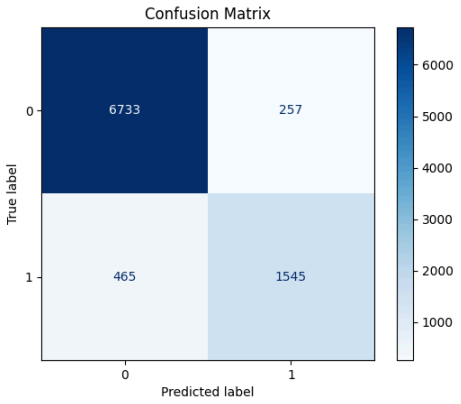
Now that the build and train along with the hyperparameters have been covered the results of the models can be explored. In the training each build and train was run with each configuration of hyperparameters over the dataset. After each model was trained the code prints the validation accuracy, the test loss, and the test accuracy. From this information two things can be determined, the best set of configurations per build and train model and the best pairing of build and train model and configurations. First, going through the best set of configurations per build and train. To determine the best configuration the validation accuracy is used as that is what is used to both monitor and tune the model during training. With the validation accuracy as our baseline for training the following results show the best configurations for each build and train models. For the base build and train model the best performance came from the following: Configuration: {'layers': [256, 128, 64, 32], 'dropout_rate': 0.3, 'regularization': 0.0005, 'learning_rate': 0.0005, 'epochs': 50}. With this configuration a validation accuracy of 92.36% was achieved. For the build and train model two the same configuration once again generated the best results this time producing a validation accuracy of 92.38%. For model three, the following set of configurations was better: Configuration: {'layers': [512, 256, 128], 'dropout_rate': 0.4, 'regularization': 0.0001, 'learning_rate': 0.001, 'epochs': 30}. With these configurations the model produced a 91.99% accuracy. For model four, the best set of configurations returned to that of models one and two, producing a validation accuracy of 92.52%. Finally model five had the same best configuration as model three, producing a validation accuracy of 92.34%.

Now that each of the models has been built and trained, the highest performance model will be applied to the test data, in this case that is model four, with the Configuration: {'layers': [256, 128, 64, 32], 'dropout_rate': 0.3, 'regularization': 0.0005, 'learning_rate': 0.0005, 'epochs': 50} and accuracy of 92.52%. When this model was applied to the test data, the following visualizations were created to help explore the results.

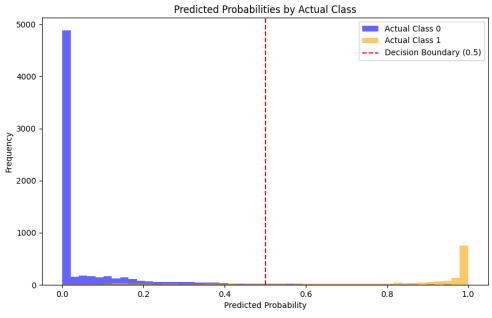


The image above helps visualize the model's improvement over time both through accuracy and loss. For the accuracy graph, the accuracy can be seen to improve over epochs and approaches a place of convergence. Similarly, the loss function also improves over epochs reaching a more defined convergence point towards the end. Another visualization produced

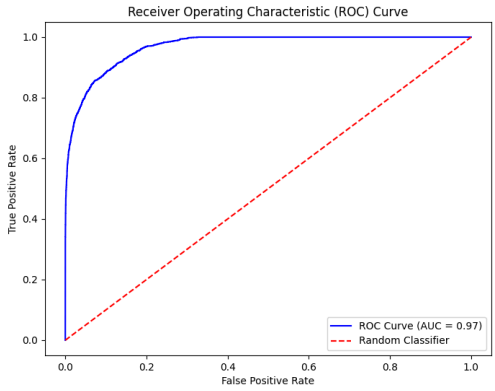
is the following confusion matrix:



The confusion matrix can be used to assess the performance of the model. The rows show the actual or true label for the model and the column shows the predicted label. The squares are then shaded on a blue gradient where the darker the color, the better the performance. At just a glance it is obvious that the true negatives and true positives are greater than the false positives and false negatives. It also accurately reflects the dataset having a class imbalance towards denials as there are more true negatives than true positives. Another image that shows the predictive performance of the model is the following histogram:



Similarly to the confusion matrix, the histogram shows the prediction performance of the model, this time using the 0.5 decision boundary, illustrated by the dotted red line. To the right of the line a cluster of orange bars can be seen and to the left a cluster of blue. These clusters are towards either 0 or 1, showing that the model had high confidence in many of its predictions. There is also minimal overlap of the colors across the decision boundary. This demonstrates that the model performs well in assigning predicted probabilities. Like the confusion matrix the class imbalance is once again highlighted with the actual class 0 having many more data points than the actual class 1. The final visualization created for analyzing the performance of the model was the following ROC curve:



This curve tells a lot about the performance of the model. First, the red dotted line represents the accuracy of a random prediction classification or a 50% confidence interval. The blue line represents the performance of the model, which is far above the 50% mark. This leads the viewer to believe that the model performs very well in its classification. The other piece to look at is the step rise of the curve towards the top-left corner of the graph. This steep rise illustrates the models ability to accurately predict the correct results quickly, maximizing the amount of true positives while minimizing the amount of false positives.

In conclusion, the performance of the model both through the math results and the data visualizations reflect the neural network's high performance in classifying whether or not a loan will be approved or not. The following proved to be the best configuration: {'layers': [256, 128,

64, 32], 'dropout_rate': 0.3, 'regularization': 0.0005, 'learning_rate': 0.0005, 'epochs': 50} and accuracy of 92.52%, with the build and train model four. This setup performed slightly better than the other combinations tested. Further work done by expanding the dataset to remove the class imbalance between acceptances and denials. Other types of layers, configurations, and combinations of both can also be tested to try and improve both the accuracy of the model and the loss of the model.

V. Conclusion

The Logistic Regression model was very limited in its ability to fit the data, and hyperparameter tuning had minimal effects on its performance. By comparison, both the SVM and neural network models responded well to the hyperparameter tuning, seeing a roughly 3% increase in performance after manual testing. While both the SVM and neural network models showed the best improvement due to hyperparameter tuning, the neural network showed the best results overall for the dataset.

Potential future improvements for these models would include addressing the class imbalances in the dataset, by either adding more data overall, or regularizing the dataset better. For the logistic regression and SVM models, further improvements from hyperparameter tuning seem unlikely, while the neural network model may benefit from additional testing with different numbers of layers, or neurons in each layer.

VI. Credits

Jacob Brubaker was responsible for writing the abstract and introduction, the python script for cleaning the data, and the development and writing of the SVM Model. Trevor Klein was responsible for creating the Neural Network code and writing portion. Nicholas Dibello-Hitta was responsible for the development and writing of the Logistic Regression model.