CS7641: Project 1

Rory Michelen | rmichelen3@gatech.edu

I. Introduction

This project explores 5 supervised learning methods on two unique and interesting datasets. It starts with a description of the data, followed by a high-level discussion of the methodology. Next there is an in depth discussion of hyperparameter tuning for each model. Finally, an analysis of each model and comparison of their performance for each of the two datasets.

II. Datasets

Dataset 1: Credit Card Fraud

The first dataset I selected was a set of 285,000 credit card transactions from 2013. Of these transactions 0.17% of them are labeled as fraudulent. The objective for this dataset is to correctly classify transactions as fraudulent and non fraudulent using the 30 features provided.

Fraud classification is a highly practical application of supervised learning due the obvious financial loss associated with missed fraud. However, from a technical standpoint, this dataset, and the fraud detection problem in general, has interesting and unique characteristics to study.

Firstly, the dataset is highly imbalanced. Only 0.17% of the data is labeled as fraudulent whereas the remaining 99.83% is labeled as non-fraudulent. This imbalance wouldn't be much of an issue if we cared about each class equally. In fact, it would be very easy to obtain a model with high accuracy. However, the cost of mislabeling a fraudulent transaction as non-fraudulent is more costly than mislabeling a non-fraudulent transaction as fraud (better safe than sorry). Correctly classifying the minority class will pose issues for algorithms with loss functions that consider both classes equally. Additionally having a "target class", that is, a class that correctly labeling is of most importance, will allow us to explore error and evaluation metrics that are not applicable for supervised learning problems concerned with overall accuracy such as precision, recall, and F1 score.

The second characteristic of this data that is interesting to study is that the target class only has 492 data points. It is a scarce resource. We will have to think critically about how we utilize these 492 data points to extract as much value out of them as possible.

Dataset 2:Kickstarter Projects

The second dataset I selected was a set of 379,000 kickstarter projects along with their outcome. Kickstarter is a crowdfunding website where users can raise funds for their projects. This dataset is interesting from the application standpoint because predicting successful projects can inform how future project creators create their projects.

From a technical standpoint, the data is also very interesting. The outcomes have several classes- success, failure, canceled, suspended, and live. These outcomes are imbalanced which is another interesting feature. Further there is a mix of categorical and numerical features. Most importantly, this dataset is unlike the fraud dataset because all of the 13 features are intuitive to interpret (e.g Budget, Country, etc.) since they relate to data points we encounter in everyday life. This is useful for generating hypotheses and performing feature engineering. Finally, one of the features is a free text field which provides opportunity for natural language processing.

Unlike the fraud detection dataset, it is not obvious that predicting one class is more important than the others. So this dataset will enable us to stick to more traditional error and evaluation metrics such as accuracy.

III. Methodology

Feature Selection: Dataset 1

The credit card fraud dataset has 30 features. The runtime of many supervised models grows with the number of features. However, it is possible that many of these features do not provide much value. Therefore, the first experiment was to determine if any features could be removed without significantly impacting the quality of the model.

To answer this question, 10 decision tree models were fit each with a different max depth on the training set. The feature importance was then calculated and graphed in figure 1.

Since the most important features are independent (mostly) of max depth and the top 15 features contribute more than 95% of feature importance, we can safely select the 15 most important features and drop the remaining 15 in order to improve model run time. Selecting 15 features as a cutoff as opposed to say 10, or 20 is a bit arbitrary but was informed by % of importance captured (95%). However, this could be tweaked in future model iterations

Note that these 15 features will be used for all learning algorithms, not just decision trees. Note that the relative importance of features may vary for each algorithm and this puts decision trees at a slight advantage. A future improvement would be to select important features for each algorithm individually

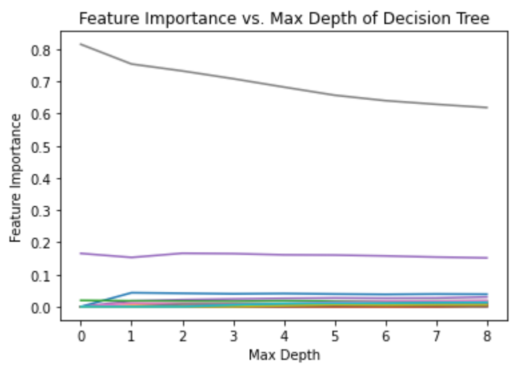


Figure 1: Feature importance vs Max Depth. This demonstrates that the relative ranking of feature importance doesn't change much with depth. This is likely due to the top down nature of the ID3 algorithm. Importance features are selected first. Therefore max depth will not have significant impact on relative importance of features

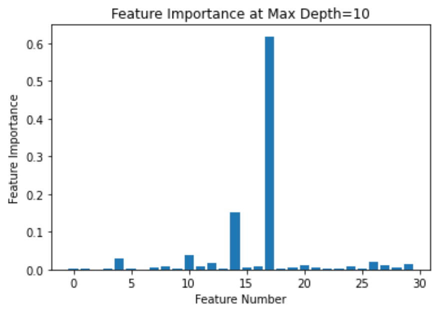


Figure 2: Feature importance at max depth =10. This shows that the top 10 highest-importance features contribute 90% of the value. Therefore it is possible to eliminate many of the features without significantly impacting the quality of the model.

Feature Selection: Dataset 2

All of the scalar features in the original dataset were used. However, there were also several text-based features. Among these were categorical variables. For some algorithms, categorical variables were encoded using one-hot encoding. One-hot encoding was used instead of ordinal encoding, since each of the categories are equally distant from one another. In the hyperparameter section, inclusion/exclusion of categorical variables for each model is discussed. Free-text variables such as "Name" were excluded from the model. While useful features could have been extracted from this free-text, that was not the focus of this project.

Training and Testing

For both datasets, 80% of the data was used for training and testing received the remaining 20%.

Cross Validation: Dataset 1

Since the fraud dataset was highly imbalanced, several oversampling/undersampling techniques were tested and will be discussed in the hyperparameters section. However, this had implications for cross-validation. Oversampling modifies the class distribution of the data. As a result, error metrics on an oversampled dataset will appear better than on the test set. Additionally, some of the algorithms took very long to run. With cross-validation, the run time is multiplied by the number of folds.

For these two reasons, instead of cross-validation, a single validation set was used. This required splitting the training data again into an 80/20 split. Note that this left the validation data with very few examples of the minority class which resulted in some highly variable results and will be considered in future improvements.

Cross Validation: Dataset 2

To better understand how a different number of folds impact the results, a decision tree was trained using different K-Fold values from 1 to 25

Intuitively, a lower value of K suggests that we are using a larger set of data to validate and should therefore increase bias and reduce variance. The opposite will be true of a high value of K. Additionally, high K values implies more folds which implies a longer training time. The same model was then fit to the entire training set and used to predict the test set. Figure 3 displays the absolute difference in accuracy between the CV error and testing set error. As expected, there is much more error (negative error) for lower values of K whereas higher values of K have a smaller amount of error. However, K=5 produces an elbow, suggesting that there are diminishing returns after K=5. All values in figure 8 are negative, suggesting that the CV error was lower than test error.

It would be cheating (the bad kind of cheating) to use Figure 8 to inform the number of folds used in the training set, since that would suggest that we leveraged testing set data in the training set. However I will use K=10 for all algorithms.

If I had dedicated more time to this experiment, I would have repeated the experiment using different hyperparameters and calculated the variance in the CV/Testing Accuracy diff for each value of K. That would have provided a deeper explanation of the tradeoffs between low and high K values. Additionally, I would have also recorded wall clock time.

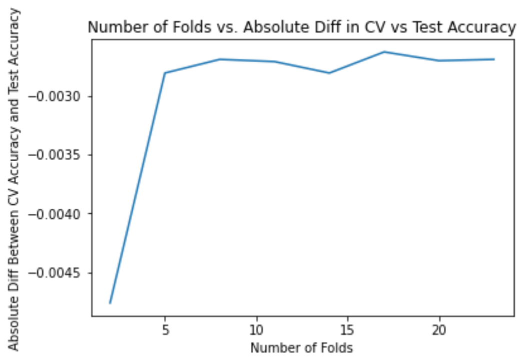


Figure 3: Difference in CV Accuracy and Test Accuracy as a function of K for K-folds cross validation.

Error and Evaluation Metrics

F1 score was used as the primary metric for the first dataset. This is because F1 distinguishes between a target or minority class and a majority class. F1 score is calculated as follows:

$$f1 = 2 * \frac{precision * recall}{precision + recall}$$

$$precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

$$recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

Note that f1 attempts to balance the tradeoff of precision and recall. However, for any given application, the associated cost of precision relative to recall may vary. For this project, we will use the traditional definition of F1 score.

For the 2nd dataset, accuracy was used as the primary evaluation metric. This is because there isn't a particular class that is most important to predict. Rather, overall prediction quality among all classes is important.

IV. Hyperparameter Tuning

Decisions Trees: Dataset 1

In order to incentivize the decision tree model to prioritize the minority class, two methods were explored- oversampling and custom class weights. Oversampling is the act of duplicating data points in the minority class so that it has a stronger representation in the data. Custom class weights increases the weight of the minority class in the entropy calculation. Both of these methods have benefits and drawbacks. Oversampling can easily lead to overfitting whereas custom weights can significantly decrease precision. However, it is hypothesized that using a combination of the two will yield better results than using a single technique.

The oversampling rate and class weights were tuned simultaneously. Since overfitting is the primary drawback of these techniques, training error cannot be used as an evaluation metric. As discussed earlier, a predetermined validation set was used to estimate test error.

The results for each pairwise combination of hyperparameters is shown below:

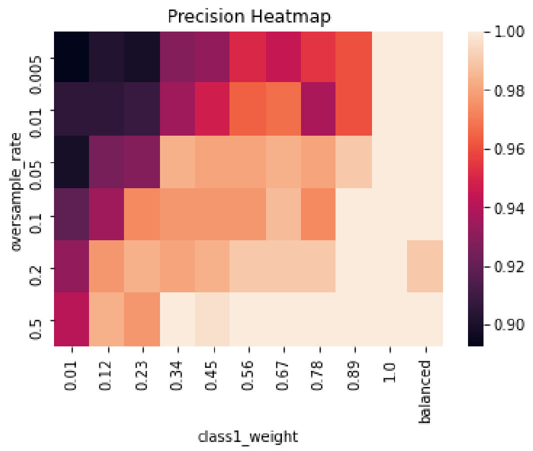


Figure 3: Model precision (training set) as a function of class weights and oversample rate. High resample rates combined with high class weights produced the highest precision.

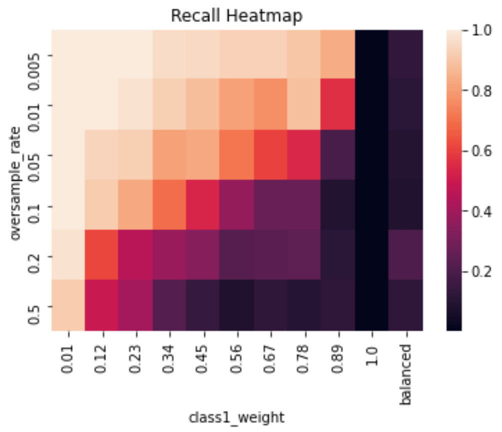
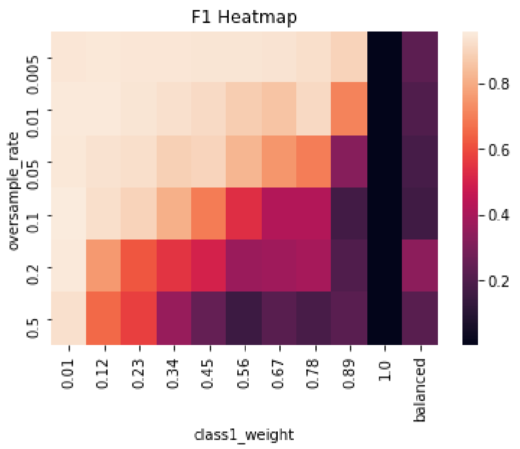


Figure 4: Model recall(training set) as a function of class weights and oversample rate. Here we see the reverse effect as precision. Low values of both parameters produce the highest recall (% of minority class correctly labeled)



·

Figure 5: Model f1(training set) as a function of class weights and oversample rate. Here we see how the competing objectives of precision and recall net out in the f1 metric.

There is a lot to unpack in these charts. As we increase both our oversampling rate and class 1 weights, we see precision go up and recall go down. In figure 5, we see how these competing objectives net out in the F1 metric. To understand why these hyperparameters have adverse effects on precision and recall, we'll have to unpack how the decision tree chooses it's splits. Since we are using information gain, the tree is attempting to find splits that yield the biggest reduction in entropy. The formula for entropy is:

$$E(S) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

Note that custom class weights modifies the above function by multiplying each class by a constant. As we increase weights towards the minority class, the decision tree increasingly wants low entropy nodes for the minority class. The result is high-precision nodes, with any false positives buried in nodes with a high number of the majority class.

The second big takeaway is that our original hypothesis is correct- a blend of oversampling and class weights produces a higher quality result than either technique used independently. As a result, the hyperparameters yielding the highest f1 score were selected.

After tuning the hyperparameters to handle class imbalance, the next hyperparameter to be tuned was the maximum depth of the tree. There are other hyperparameters that are useful in pruning. However these were explored primarily in the second dataset and will be discussed later.

Grids Search was performed on maximum-depth values of 2 through 15 using the same predetermined validation set

Figure 6 shows the Cross validated Precision, Recall, and F1. Note that at max depth of 3, the highest possible recall is achieved However as we increase the depth of the tree precision and consequently F1 increase. While we are using cross validation, I suspected that this was the result of overfitting, so I peaked at the evaluation metrics for the test data, seen in figure 7.

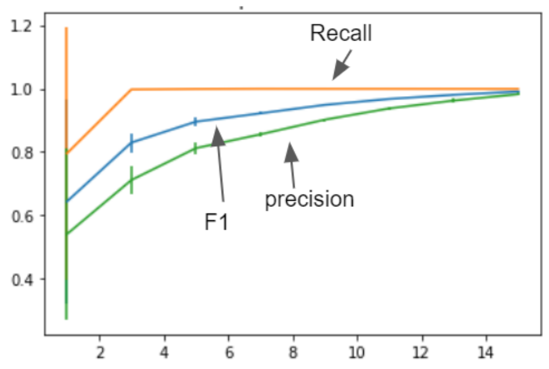


Figure 6: Cross-validated precision, recall and f1 as a function of maximum depth.

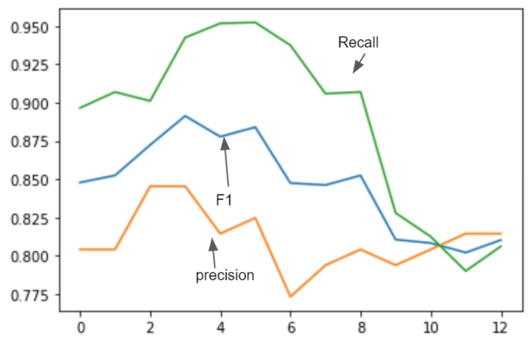


Figure 7: Precision, recall and f1 as a function of maximum depth.for testing set.

Figure 7 demonstrates that the monotonically increasing f1 score and precision seen in figure 6 is indeed due to overfitting. While it is cheating (the bad kind of cheating) to use evaluation metrics from the test set to pick model parameters, this was indeed a good learning experience. However, I will pretend I didn't look at the test set and use a max depth of 5 in the final model since that is where we start to see diminishing returns on F1 in the cross validated metrics (aka the elbow).

(Note it was later discovered that due to a bug, the predetermined validation set was also used in training. This is what led to the overfitting).

Decisions Trees: Dataset 2

4 Hyperparameters were tested for the second dataset. However, exploring all possible combinations of 4 hyperparameters would have required fitting 100,000+ models. Therefore it was determined to experiment with two parameters, select values and then experiment with the remaining two. While this methodology might result in local optima, it reduces the number of required fits from 100,000 to 2,000.

The first pair of hyperparameters tested were the max depth and max features. Max depth determines how many levels deep the tree can search whereas number of features determines how many features are evaluated at each split. Figure 9 shows the cross validation accuracy as a function of these two parameters. There is a "sweet spot" for max depth. Two shallow of a tree underfits whereas too deep of a tree overfits. Based on this analysis the sweet spot appears to be max depth=10. This is conveniently the number of features considered in the model. The max features parameter unsurprisingly monotonically increases as more features are considered at each split. However, the run time significantly increased as max features increased. Nevertheless, runtime was only 12 seconds for max features = 10 and thus this value was selected.

The second set of hyperparameters evaluated were the Min Leaf Split- which determines the minimum number of data points required to justify a new node and Min Impurity Decrease- which determines the tradeoff between tree complexity and impurity of the nodes. As seen in figure 10, the minimum impurity decrease parameter had a much bigger impact on the accuracy then the min leaf split parameter. This is likely due to the space of values explored- a wider range for the Min Samples Split parameter would have yielded a wider range in accuracy. Nevertheless, this second experiment suggests that setting both parameter values to allow for more complicated tree will improve accuracy with little or no overfitting. Therefore min impurity decrease was set to 0 and Min Leaf Split was set to 2.

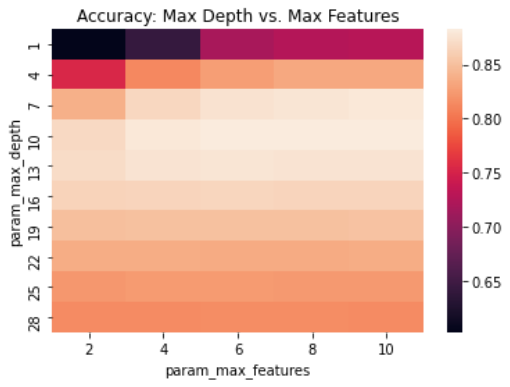


Figure 9: CV accuracy as a function of Max Depth and Max Features parameters.

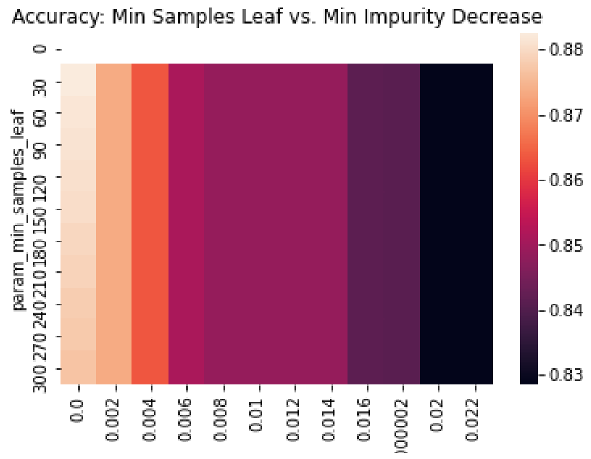


Figure 10: CV error as a function of Min Samples Leaf and Max Impurity Decrease parameters.

A. Neural Networks Dataset 1

Similar to decision trees, neural networks do not perform well with imbalanced datasets. This is because the loss function- cross-entropy- does not consider the relative importances of the two classes. Therefore, it can achieve a very low cross-entropy by focusing on the majority class. This will be reflected by poor recall metrics.

Unfortunately, Scikit-Learn does not have a means of incorporating class-weights into the loss function of a multilayer-perceptron classification network. However, we can rely on oversampling to communicate to the model that the minority class is of greater importance. Similar to the decision trees methodology, the 20% of the training set was held out for validation. This was to ensure that evaluation metrics for different oversampling rates were being evaluated using a dataset with a distribution similar to the original data. Figure 11 demonstrates the results of this experiment.

Oversampling Rate	F1	Recall	Precision
0.005	0.862129	0.928571	0.804560
0.010	0.893617	0.898026	0.889251
0.050	0.834473	0.719340	0.993485
0.100	0.657267	0.492683	0.986971
0.200	0.637591	0.467988	1.000000
0.500	0.704937	0.544326	1.000000

Figure 11: Evaluation metrics for validation data for different oversampling rates.

Note that as the oversampling rate increases, precision increases whereas recall decreases. This is the same phenomenon seen with decision tree oversampling. However this increase isn't monotonic. Recall actually increases between 0;2 and 0.5 oversample rates. One hypothesis for this phenomenon is that lower resample rates (0.01-0.2) allow the model to achieve low entropy by hiding the minority class in leaf nodes that are primarily composed of the majority class. If this strategy is effective for the model, then it will result in high-precision minority class nodes with low recall due to many minority-class examples hidden in nodes with overwhelmingly more data from majority class . However, for higher resample rates (0.5), the minority class has just as much data as the majority. Therefore, the model focuses on accurately predicting both classes. These results could also be the result of overfitting since very few data points were duplicated hundreds of times to achieve balanced classes. Nevertheless, the highest F1 score was achieved at a resample rate of 0.1 and was therefore selected for the final training.

The next set of hyperparameters explored were the number of nodes and layers in the network. Several node-layer combinations were tested on the oversampled dataset. The results for F1 score, precision, recall and wall clock time are provided in figure, 12.

There are several takeaways from these results. Firstly, the number of hidden layers is more influential on the fit time than the nodes/layer of the model as seen by the bottom right chart. Secondly, 2 of the node-layer (2x10,5x10) pairs produced very low F1 scores in a non-intuitive way. It is possible that these models did not receive enough of the minority class- either in training or cross validation. The third takeaway is that precision begins to decrease for very large networks. This is most exemplified by the rightmost bars in the top right plot in figure 12. A final takeaway is that the F1 score for all of the tested networks was lower than the maximum F1 score in figure 11. However, this is likely due to the fact that the evaluation metrics in figure 12 were calculated on the oversampled dataset whereas figure 11 used a dataset with a distribution similar to the original data.

Ultimately, a network of 10 layers and 5 nodes per layer was selected since it produced the highest F1 score.

One additional hyperparameter tweak was the batch size. Batch size determines the number of samples evaluated prior to back propagating error through the model. The default value of this parameter is 200. However, this small sample has a high probability of containing zero samples from the minority class. Therefore, the batch size was increased to 2,400.

While there are many hyperparameters that were not able to be explored due to time constraints such as learning rate, batch size, number of epochs, activation-types, and alternative loss functions, these will be explored on the second dataset. However, future work for this dataset could include experimenting with various epoch/batch-size combinations.

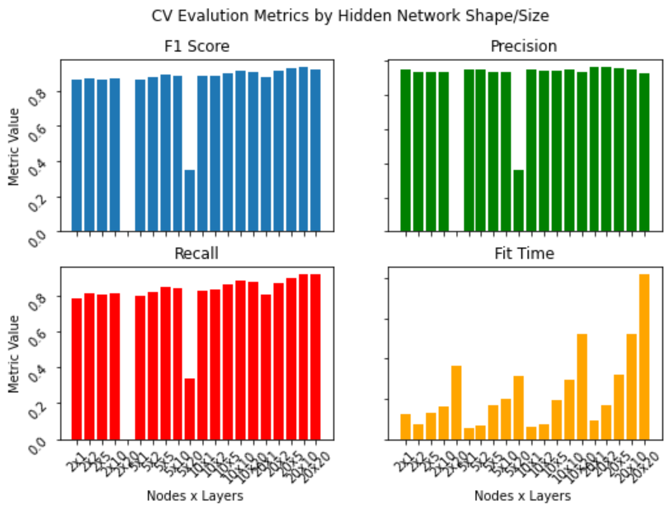


Figure 12: Cross-Validated F1-score, precision, recall, and wall-clock time as a function of number of hidden layers and nodes/layer.

B. Neural Networks Dataset 2

The first hyperparameter experiment was to test various network sizes. Figure 13 demonstrates the cross-validated accuracy as a function of the number of hidden layers and nodes per layer. The results demonstrate a significant increase in accuracy as soon as 5 or more layers are used, suggesting that two hidden layers is not enough to capture the complexity of operations. However, there is no increase in accuracy as 6 or more layers are used and run time becomes unnecessarily long. Based on these results a network size of 10 layers and 5 nodes per layer was selected.

Note that all of the hidden networks tested are rectangular. That is, the number of nodes per layer is constant. Future improvements could explore various hidden network shapes such as a triangle (shallow then wide or vice versa), diamond (shallow, wide, shallow), or other more complex shapes.

The next set of experiments was to explore various learning rates and iterations. Note that it is hypothesized that these two hyperparameters are highly interconnected- a slower learning rate will require more iterations to converge whereas a faster one will need fewer. Therefore, I performed tuning on various pairs of these hyperparameters instead of a single parameter followed by the other.

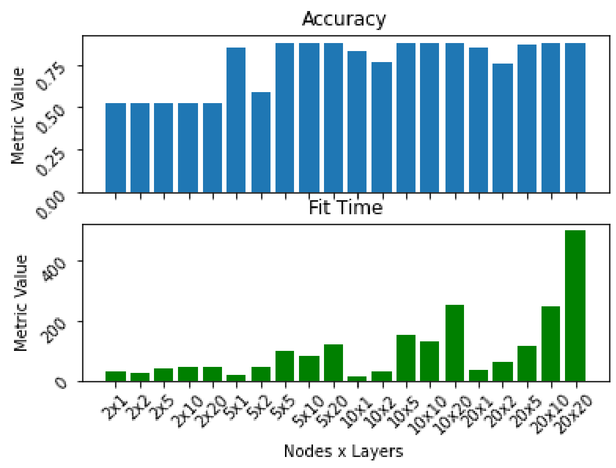


Figure 13: Cross-Validated accuracy and wall-clock time as a function of number of hidden layers and nodes/layer.

Figure 14 shows the somewhat unexciting results of testing various learning rates and max iterations. Reducing the learning rate slightly from the default value demonstrated an improvement in overall accuracy. However, too fast or slow learning rates resulted in much worse performance. In the case of too high learning rates, the model is not able to reach an equilibrium since it is incorporating too much noise into its learning. However a too small learning rate does not learn fast enough to absorb all of the available information.

The maximum iterations had minimal impact on the results and the model converged surprisingly quickly. However, with very small learning rates, having too few iterations did reduce accuracy. This makes sense since the learning is slower, it will require more iterations to converge.

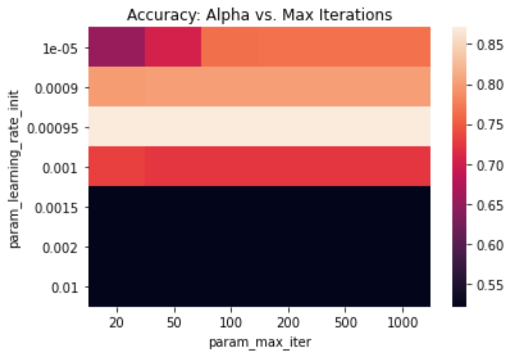


Figure 14: Cross-Validated accuracy as a function of learning rate and maximum iterations

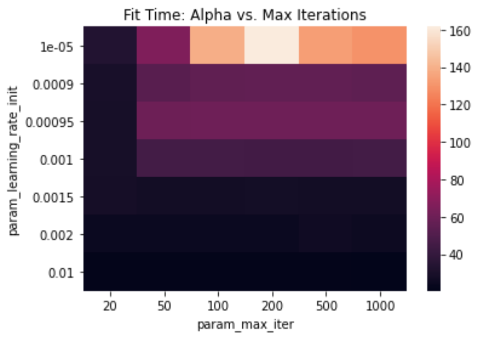


Figure 14a: Cross-Validated fit time as a function of learning rate and maximum iterations

While additional solvers were tested, the default, Adam, significantly outperformed the others and was therefore used in the final model. Note that given more time, there are a wealth of other parameters to explore. Firstly, alpha values could have been explored to combat overfitting. Secondly, the variety of momentum parameters available for the Stochastic Gradient Descent algorithm could have been tweaked and potentially outperformed the Adam Solver.

C. K Nearest Neighbors Dataset 1

Like the previous learning algorithms discussed, K-Nearest Neighbors is impacted by imbalanced datasets. In other words, if there are 1,000 non-fraudulent credit card transactions for every fraudulent one, it's more likely that most of the neighbors will be non-fraudulent.

Ideally, I would have liked to solve this problem by giving the fraudulent class more votes than the non-fraudulent class during prediction. However, I could not find a way to do this in Python after hours of research. However, we can simulate this effect by either undersampling or oversampling. Initially I attempted oversampling the minority class. However, this resulted in poor validation results. This is because KNN will easily be overfit with oversampling. If one datapoint is duplicated 10 times, then at K=10, that single datapoint could potentially be the top 10 neighbors for a new prediction. For this reason. Undersampling the minority class was used. This had the added benefit of reducing prediction time.

The performance of the KNN model at various undersampling rates was observed to be heavily dependent on the number of neighbors used. For example, high oversample rates performed better at high number of neighbors. Therefore, a hyperparameter search was conducted for a wide range of undersample rates and number of neighbor combinations.

As demonstrated in figure 16, the precision increased for high undersample rates and low values of K. While the values tested demonstrate a monotonically increasing relationship, it is expected that increasing the sample rate or decreasing K further would result in overfitting and precision would decrease. Note that the relationship between hyperparameters and recall was inverse to that of precision. Low undersample rates combined with high K values resulted in the highest recall. This makes sense since this results in the least chance of overfitting and will therefore allow the model to correctly predict a large number of the minority class samples. Selecting the highest f1-score option resulted in a K=50 and undersample rate=0.5 (balanced classes). Note that a precision of 70% is low compared to other models. Additionally, since the validation set oversampled the minority class, it is expected that precision on the test set will be even worse, suggesting that KNN is not a high-precision model for this dataset. However, test results and comparison of models will be discussed in the analysis of results section.

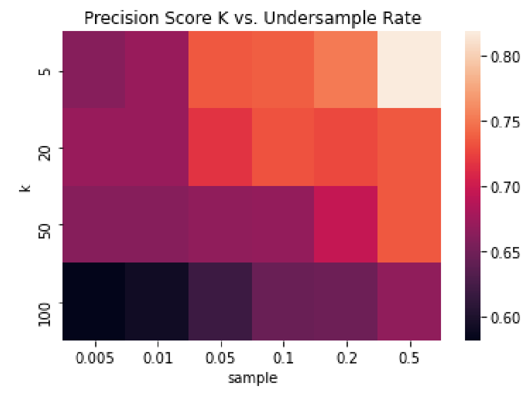


Figure 16: Validation precision as a function of number of neighbors and resample rate for the first dataset.

Feature Weighting

The previous experiments were implemented on a KNN model that treated all 15 features in the model equally. That is, we scaled the variables such that each feature was equally contributing to distance. However, figure 2 demonstrated that not all features are equally important. Additionally, the distance function used has not been explored. The next set of experiments was to determine whether adding feature weights or using Manhattan/Euclidean distance functions will impact the quality of the model. Values of K and oversampling rates found in the previous set of experiments were used.

The space of feature weights to explore is infinitely large. To select good candidate values, initial weights were set equal to feature importances in figure 2. These weights were then varied by setting to the power of n where n is between 0 and 10. Large values of n will make important features even more important whereas values close to 0 will be closer to equally scaled features. Due to time constraints, only Manhattan distances were tested however alternative distance functions were explored with the 2nd dataset.

Figure 17 shows the results of different feature weights. Note that n=0 (all features weighted equally) produced the best results. This suggests that no feature weights should be added. However, this could be due to the fact that all feature weights tested were based on feature importances derived from a decision tree model. While this experiment explored different ways of scaling these feature importances, it did not explore the starting weights.

n	precision	recall	f1	predict_time
0.00	0.765333	0.957229	0.850135	868.121349
0.10	0.765333	0.957229	0.850135	868.121349
0.25	0.765333	0.941930	0.843877	868.121349
0.50	0.762667	0.944296	0.843223	868.121349
1.00	0.709333	0.946643	0.810680	868.121349
2.00	0.688000	0.931874	0.790961	868.121349
5.00	0.682667	0.904629	0.777589	868.121349

Figure 17: Validation results for different feature weights as a function of n. n is a parameter that determines the magnitude of relative feature importance (n=0 yields equal weighting of features, n=5 yields large differences in feature importances).

Given the results, no feature weights were added to the model. However, with more time, additional starting weights would be tested along with various weight functions.

D. K Nearest Neighbors Dataset 2

The second dataset of Kickstarter projects resulted in several hurdles for KNN. Recall that the second dataset had 4 categorical features. In the previous models, these categorical features were treated as ordinal data. However, this probably resulted in decreased model performance since there is not an ordinal structure to any of these columns. In other words, the distance between any two categories is the same. However, ordinal feature encoding will assign an integer to each category. When it comes time to compute distance, a label encoded as 1 will be calculated as having a higher distance from the label encoded as 4 than it will to a label encoded as 2.

To resolve this, I attempted to use One Hot encoding. However, one of the columns had 159 categories and would have resulted in 159 additional columns. This resulted in memory errors. Despite the above mentioned issues, I opted to keep the categorical data encoded as ordinal and test the impact using feature weights.

Feature Weighting

The second dataset followed a similar methodology as the first. Again, data size was large, so instead of cross validation, 5 predefined splits were used for validation and a majority of data was kept into the training fold. I again wanted to test the impact of feature weights with this dataset. However, I attempted a different methodology since the feature importance method did not work on the first dataset.

I started by testing scenarios where only 1 of the 10 features was given a weight that was twice the other 9 features. Then I repeated this but with a weight of half the other features. This produced 20 different models. Two additional scenarios were tested: (1) all variables have equal weighted, (2) all variables have equal weights except the 4 categorical variables which were weighted at zero, effectively removing them from the model.

The weighting that removed the categorical variables produced an accuracy of 84% whereas the other 21 scenarios ranged from 77%-79%. By removing the 4 variables we experienced significant improvements in the model. The next best 2 scenarios had applied weights of 2 two the 7 other features- USD Goal and USD Pledge. Intuitively, these two features make sense as having the most importance as to whether a Kickstarter campaign is going to succeed.

Based on the results, a second round of feature weights were tested. During this round, all categorical variables were removed and equal weights were placed on all remaining variables excluding USD goal and USD pledge. These two variables were then weighted 2,5, or 10 times higher than the other features. Selecting one of these three values for the two parameters produced an additional 9 scenarios. The results indicated that weighting both features as 10 times higher than the other features produced the highest accuracy. Therefore, this weighting scheme was used in the final model.

Note that only a fraction of the feature weight was explored. Higher accuracy could be achieved by a more sophisticated search of the hyperparameter space.

Additional Tuning

Aftering finalizing the feature weights, the number of neighbors and the weight type were evaluated. K between 1and 500 were tested with both uniform weights and inverse scaling. Inverse scaling enables closer neighbors to have more weight in the final vote.

Figure 18 shows the results of this experiment. At all values of k, the uniform weighting outperformed distance-based weighting. While not investigated, I would expect the % difference in performance to decrease as K increases. This is because with a large number of neighbors, some votes are coming from very far away and therefore, distanced-based begins to make more sense. Figure 18 demonstrates the tradeoff of too many vs too few neighbors. When k was too low, the model was overfit. However as K increased, the prediction was approaching the average of the training set and was underfit. For this dataset, a value of k=25 with equal weighting produced the highest validation accuracy at 86%

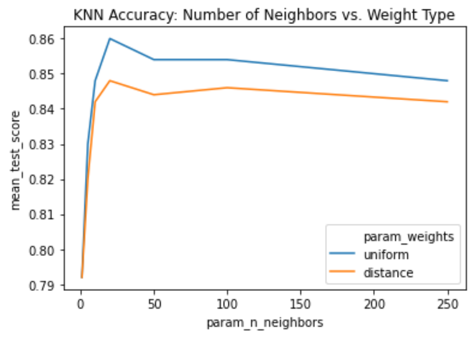


Figure 18: Validation results for different number of neighbors and weighting types.

With more time to tune, I would have dedicated more time to feature weighting and distance functions. Here only the manhattan distance was used, but the euclidean distance as well as more sophisticated weight metrics could have further reduced error. Additionally, the categorical variables were ignored since they reduced accuracy. However, if encoding was switched from ordinal to one-hot encoding, they could have contributed to higher accuracy.

E. Boosting: Dataset 1

The XGBoost algorithm was selected as the primary algorithm for all boosting-based learning due to it’s fast runtime. However, alternating boosting algorithms should be explored.

Four hyperparameters were tuned in the building of this model, oversample rate, class weights, max depth, and eta. This generated over 200 combinations of hyperparameters, which were evaluated using a predefined validation set. Figure 19 shows the validation f1 score for all 200 hyperparameter options. Note that as F1 score decreases, we see significant decreases in precision whereas model recall increases.

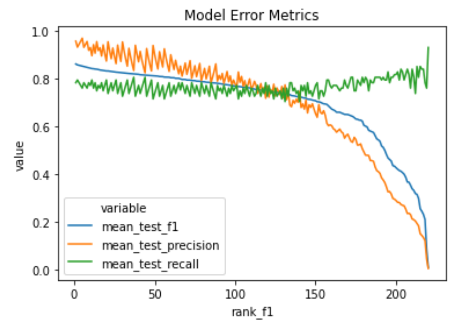


Figure 19: Precision, recall, and F1 for 200+ combinations of hyperparameters

This becomes clearer as we explore figure 20. Figure 20 shows how the values of hyperparameters change as the recall score goes from best to worse.

The first observation from figure 20 is that oversample rate and class weights are inverse to one another. This makes sense, since they both accomplish the same objective- increase the importance of the minority class. Nevertheless, the general trend is that as recall decreases, less weight- either through oversampling or class weights- is placed on the minority class. Max depth and Eta parameters also demonstrate some correlation. The highest recall solutions have low max depth. However, as recall decreases, both max depth and eta increase. Since increasing either parameter may result in overfitting, increasing both will understandably overfit the model, resulting in lower F1 score.

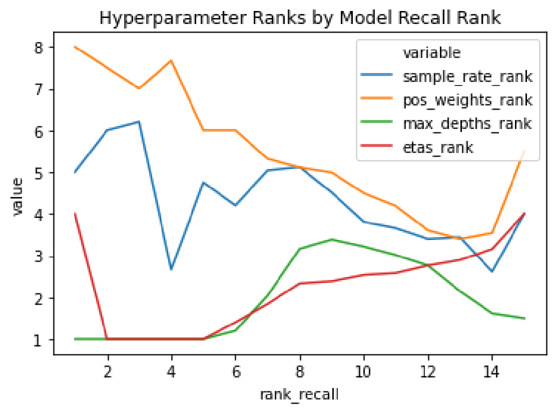


Figure 20: Hyperparameter ranks as a function of model recall. For each hyperparameter, the rank represents the relative ascending rank in value. For example, 8 options for the hyperparameter "pos_weights" were explored. The largest value for this parameter is labeled as 8 in the graph. This ranking scheme was used so that all hyperparameters could be shown on the same graph.

Note that despite the fact that boosting increases the weight of incorrect predictions with each iteration, some level of oversampling was still needed. This is because without oversampling, the missed predictions of the majority class outweighed those of the minority.

F. Boosting: Dataset 2

A similar methodology for hyperparameter search was used on the second dataset. However, since class imbalance was not an issue, oversample rate and class weights were excluded whereas subsample rate was explored. Figure 21 shares the results of the hyperparameter search. However, the results do not demonstrate many clear trends. However, the one clear trend is that as max depth increases, model accuracy decreases due to overfitting.

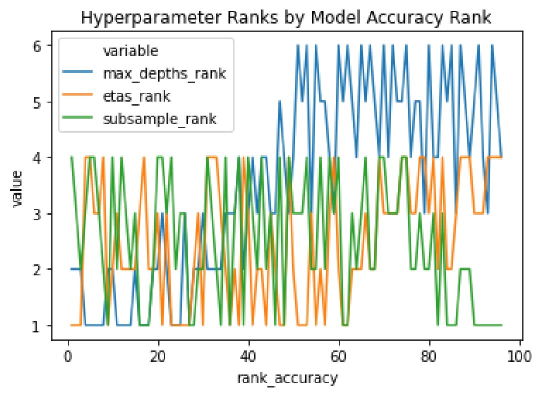


Figure 21: Hyperparameter ranks as a function of model accuracy

Ultimately, the highest-accuracy hyperparameter values were chosen. However, the heavy fluctuation may suggest there is somehow overfitting, or that the validation set was too small to adequately evaluate the model.

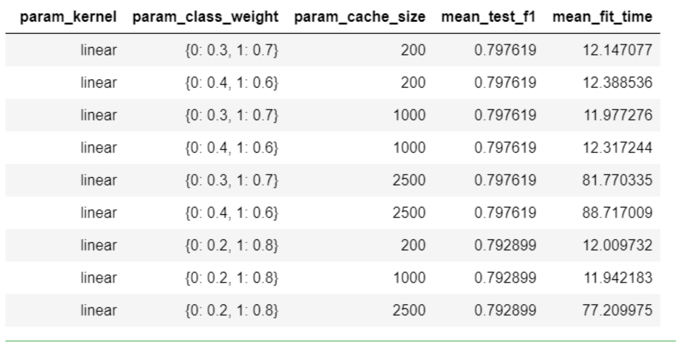
G. Support Vector Machines: Dataset 1

Several hyperparameter values were explored for the SVM model. Due to the long run time of SVM, the only way to get the model run in a practical amount of time was to undersample at a high rate. Note that this not only accomplished the goal of reducing runtime, but also reduced the class imbalance and weighted minority class errors more heavily.

Figure 22 shares the hyperparameter values for the top 10 highest f1-score options. The first observation is that the linear kernel provided the best fit on the validation set despite being the simplest kernel. This could be due to the fact that it isn't overfitting. The next observation is that among the 6 options explored for class weights, the lowest error options weighted the minority class much heavier than the majority. This makes sense since even with undersampling there was a remaining class imbalance. As expected, cache size did not impact model performance but did increase the fit time when too large.

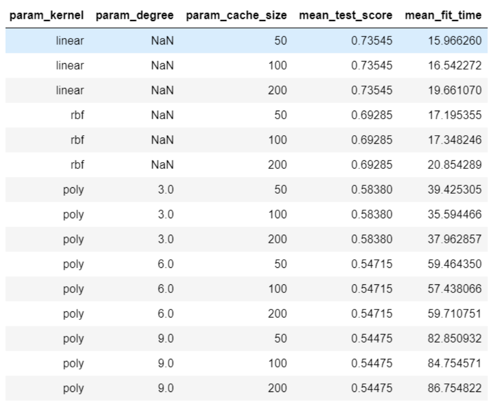
H. Support Vector Machines: Dataset 2

Similar to the first dataset, various kernels and cache sizes were tested and cross-validated. Again, the simplest kernel, linear, produced the best results, followed by Radial Basis and Polynomial kernels. Among polynomial kernels, the simpler, lower degree polynomials outperformed more complex ones both in terms of accuracy and run time. The lowest performing kernel was sigmoid kernel.



param_kernel	param_class_weight	param_cache_size	mean_test_f1	mean_fit_time
linear	{0: 0.3, 1: 0.7}	200	0.797619	12.147077
linear	{0: 0.4, 1: 0.6}	200	0.797619	12.388536
linear	{0: 0.3, 1: 0.7}	1000	0.797619	11.977276
linear	{0: 0.4, 1: 0.6}	1000	0.797619	12.317244
linear	{0: 0.3, 1: 0.7}	2500	0.797619	81.770335
linear	{0: 0.4, 1: 0.6}	2500	0.797619	88.717009
linear	{0: 0.2, 1: 0.8}	200	0.792899	12.009732
linear	{0: 0.2, 1: 0.8}	1000	0.792899	11.942183
linear	{0: 0.2, 1: 0.8}	2500	0.792899	77.209975

Figure 22: Hyperparameter values for top 10 highest f1-score options



param_kernel	param_degree	param_cache_size	mean_test_score	mean_fit_time
linear	NaN	50	0.73545	15.966260
linear	NaN	100	0.73545	16.542272
linear	NaN	200	0.73545	19.661070
rbf	NaN	50	0.69285	17.195355
rbf	NaN	100	0.69285	17.348246
rbf	NaN	200	0.69285	20.854289
poly	3.0	50	0.58380	39.425305
poly	3.0	100	0.58380	35.594466
poly	3.0	200	0.58380	37.962857
poly	6.0	50	0.54715	59.464350
poly	6.0	100	0.54715	57.438066
poly	6.0	200	0.54715	59.710751
poly	9.0	50	0.54475	82.850932
poly	9.0	100	0.54475	84.754571
poly	9.0	200	0.54475	86.754822

Figure 21: Hyperparameter values for top 10 highest accuracy option

IV. Analysis of Results

Dataset 1

Figure 22 shares the learning curves and fit times for the optimally tuned models on the primary dataset. Note that each algorithm demonstrates a different elbow, or point where the initial f1 scores near zero and then quickly increases. KNN-which leveraged undersampling and was thus provided much less data than the other models, had zero accuracy and required a minimum of 700 samples to improve F1. The neural net needed far more data than any other algorithm to yield any learning. However, this could be due to the fact that a small learning rate was selected. The decision tree provided relatively fair predictions at all tested training sizes.

Note that training times varied significantly with neural nets taking the longest. However this could be due to a high epoch parameter selection. While KNN appears to be the fastest, this is only because it is a lazy learner and in fact, the prediction time for the KNN model was impractically slow. While the SVM model was one of the fastest, the graph demonstrates an exponentially increasing fit time. Therefore this algorithm may not scale for larger datasets.

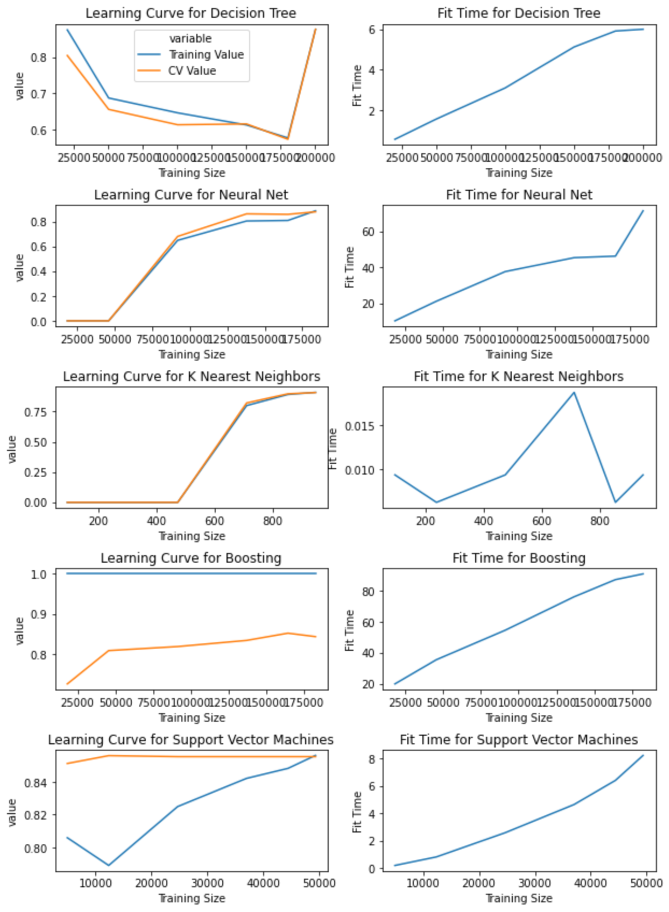
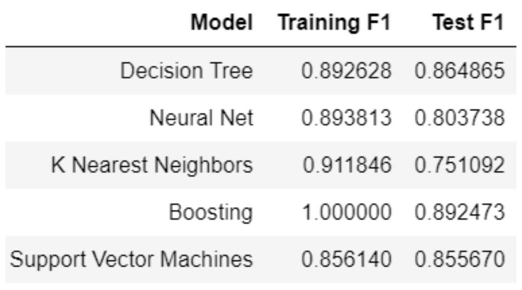


Figure 22: Learning curves and fit times for data set 1. Left chart shows F1 score. Right column shows fit time.



Model	Training F1	Test F1
Decision Tree	0.892628	0.864865
Neural Net	0.893813	0.803738
K Nearest Neighbors	0.911846	0.751092
Boosting	1.000000	0.892473
Support Vector Machines	0.856140	0.855670

Figure 23: Learning curves and fit times for data set 1.

Also of note is the difference between training and cross validation error for each model. The decision tree model demonstrated the expected trend of training error starting higher than cross validation error but converging. Note that as the decision tree added data, F1 score decreased until all the data was used. This was an unexpected trend but is perhaps due to the fact that an oversampled dataset was used. Perhaps this dataset was not properly shuffled so the class distribution varied for each of the training sizes. The boosting model was significantly overfit. Achieving 100% F1 with a much lower cross validation. Therefore, additional regularization parameters should be explored. The SVM model surprisingly had a higher validation error than training error. An undersampled dataset was used for this model and it's possible that this was again due to a shuffling error.

Note that the boosting model provided the highest F1 score on the test data despite the overfit model. Figure 23 compares training and test F1 scores for the first dataset. For many of the models, the test and training error were close. But for Boosting, KNN, and Neural Nets, there was a significant difference between training and test errors. KNN provided the lowest F1 score for the model. It is very possible that a nearest neighbors model is not appropriate for a highly imbalanced dataset. This could be due to the fact that the majority class has more votes and will lead to poor precision. Undersampling attempted to mitigate this effect but resulted in a lot of unused data. The decision tree model also performed well. This does not come as a surprise since it is a precursor to Boosting, which had the highest F1 score.

Dataset 2

Figure 24 shares the learning curves for the second dataset. We see many similar trends as the first dataset. The neural net model appears to have a shuffling issue. However, training and testing errors are very close, suggesting no overfitting. The boosting model demonstrated a much higher training accuracy, suggesting overfitting. We also see similar trends in runtime. However, the SVM model ran much slower on dataset 2 than dataset one due to the larger datasize. This speaks to the scalability of the model.

Figure 25 shares the test and training errors of the final model. Just as with the first dataset, boosting performed the best whereas K nearest neighbors performed the worst. This debunks the theory that KNN's performance was due solely to class imbalance. However, due to the significant differences in training and test error, it's likely that the KNN model was overfit. This comes as a surprise since K-Fold cross validation was used. Perhaps the test set was too small and thus highly variable to be predicted accurately using cross validation.

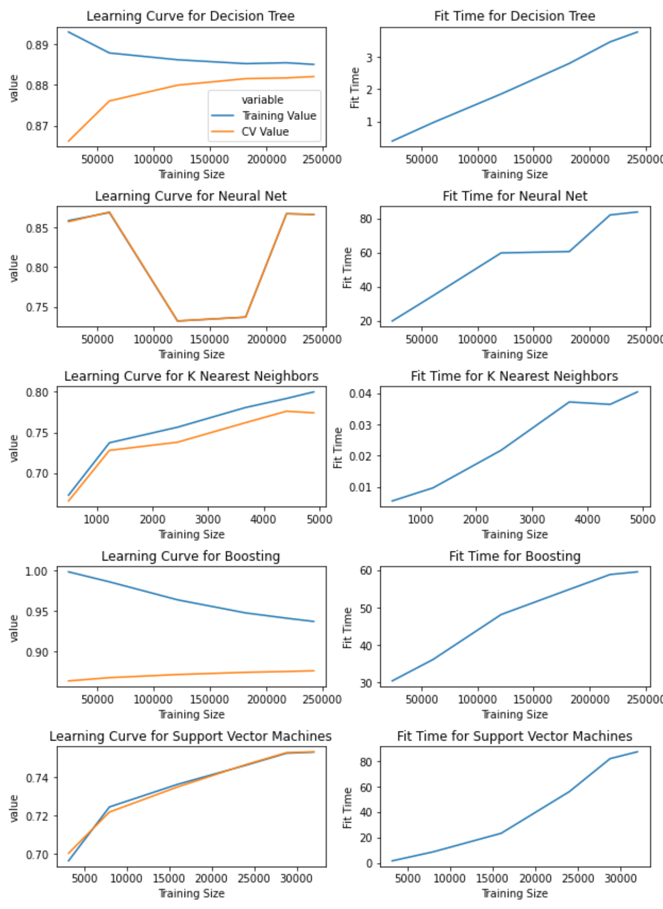
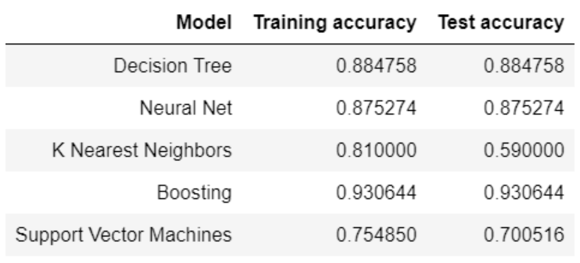


Figure 24: Learning curves and fit times for data set 2.



Model	Training accuracy	Test accuracy
Decision Tree	0.884758	0.884758
Neural Net	0.875274	0.875274
K Nearest Neighbors	0.810000	0.590000
Boosting	0.930644	0.930644
Support Vector Machines	0.754850	0.700516

Figure 25: Model performance on test data for data set 2.