CSCI 6316 Final Applications Project

Emma Glass (tfz5vy) & Gabriel Hanson ()

DATASET PROCESSING AND SET UP

Both datasets were read into separate pandas dataframes using the pd.read_csv() function. Then, the feature columns were extracted from each of the datasets. In dataset 2, there is a feature that has string values ‘absent’ and ‘present’. These strings were changed to numerical values of 0 for ‘absent’ and 1 for ‘present’ so the classification algorithms work correctly. These feature columns were then normalized using min-max normalization to ensure that relationships amongst the original data values are preserved. Then, the label columns were extracted from the original dataframes. Then, before implementing machine learning algorithms; recall, precision, and f1 measure functions were defined to be used later in calculating these model statistics.

GENERAL APPROACH FOR ALL METHODS

For each machine learning implementation we performed a similar methodology. First, we utilized the sklearn.model_selection.train_test_split() function to split the dataset into training and testing groups (75% training, 25% testing). Next, we create the model instance and define any necessary parameters in the model instance. Then, if necessary, we define parameters to be used in the grid search cross validation step. 10 fold cross validation is then performed on the training set. Then, we determine the accuracy of the resulting trained model on the training set. We define the training error then as 1-accuracy of the trained model on training set. A confusion matrix is created to determine the number of true positives, true negatives, false positives, and false negatives. These values are then used to calculate precision, recall, F1 measure. Finally, we calculate AUC by using the metrics.roc_curve() and metrics.roc_auc() functions from the sklearn package. Next, we used the trained model on the held out test set and calculated all of the same metrics in the same manner.

GRIDSEARCH CV INTERNAL MECHANISM DESCRIPTION

GridsearchCV is a useful function for tuning hyperparameters of machine learning models in python. We define a dictionary of parameters as the input to the gridsearchCV function. The keys in the dictionary are the names of the hyperparameters to tune and the values are the possible hyperparameter values. GridsearchCV tries all combinations of the values that you

pass into the gridsearchCV function, and uses cross validation to determine the accuracy of each hyperparameter combination and selects the one with the best performance.

LOGISTIC REGRESSION IMPLEMENTATION

Here, we will implement the logistic regression classifier to classify dataset 1 and dataset 2. The logistic regression classifier is a supervised, binary classification algorithm that uses a logistic function to model the dependent variable. The dependent variable must be dichotomous in nature, that is it must only have two distinct classes. The logistic regression cost function is negative because when we train the model, we want to maximize the probability by minimizing the loss function. Decreasing cost will increase the maximum likelihood assuming samples are independently and identically distributed. Logistic regression has some notable advantages. First, logistic regression is easy to implement and efficient to train. Secondly, it preforms well when the data is linearly separable. Finally, it can interpret model coefficients as indicators of the importance of a given feature. As with all machine learning techniques there are disadvantages as well. The first notable disadvantage of logistic regression is that it constructs linear decision boundaries, so it needs independent variables that are related to the log loss linearly. Secondly, there is an assumption of linearity between independent and dependent variables. Finally, more powerful and compact algorithms can easily outperform, like a neural network for example.

Dataset 1

First, we implemented logistic regression on dataset 1 using sklearn.linear_model.LogisitcRegression() from the sklearn python package. The metrics resulting from this model implementation can be seen in Table 1. To begin, we first used the default parameters for the model and NO regularization, to determine the accuracy, precision, recall, F1 Measure, Training/Testing Error, and AUC metrics on both the training dataset and testing dataset. One important thing to note is the Recall for the trained model on the training set is 1.0, which is perfect recall. Additionally, the testing error is almost 15 times greater than the training error. Both of these observations suggest that the model is overfitting on the training dataset, and there could be a better model with better-tuned parameters that would perform better on the testing set than this default model.

Table 1. Base Model With Default Parameters

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of	0.995	0.988	1.0	0.996	0.00469	0.996

10-fold Cross Validation on Training Set						
Results on held out test set	0.930	0.855	0.959	0.904	0.0699	0.937

Next, we decided to tune the regularization parameters to create a model that better fits the testing set, i.e. better generalizes to data not within the training set. We tuned the regularization parameter using gridsearch cv. The metrics resulting from this model implementation can be seen in Table 2. The resulting best regularization parameter was L2. This slightly more complex model fit the testing set better than the default model with no regularization. The recall on the training set was no longer 1.0 (perfect recall) suggestign the model is no longer overfitting the training set. Additionally, the testing error is now only ~2 times greater than the training error further confirming that the model with regularization is no longer over-fitting the training data. This is consistent with what we learned in class and know about regularization. Regularization is meant to reduce overfitting and that is what was exhibited in this logistic regression classifier. Regularization reduces overfitting by adding a penalty term with residual sum of squares to the complex model, thereby shrinking the coefficient estimates closer to zero. Although regularization greatly increases the ability of the model to fit the held out test set, we tried to increase the complexity of the model by including more/tuned hyperparameters to see if we could fit the held out testing set even better.

Table 2. Gridsearch CV to find optimal regularization parameter

	Accuracy	**Precision**	**Recall**	**F1 Measure**	**Training/Testing Error**	**AUC**
Results of 10-fold Cross Validation on Training Set	0.974	1.0	0.933	0.965	0.0258	0.996
Results on held out test set	0.958	0.978	0.898	0.936	0.0420	0.944

Optimal Regularization parameter: L2

To attempt to create a model that performs even more optimally on the held out test set, we tuned the C parameter. A high C value indicates that there should be a large weight on the training data and a lower weight to the complexity penalty. A low value of C tells the model to give more weight to the complexity penalty at the expense of fitting to the training data. We determined the optimal value of C using gridsearchCV, and kept the regularization parameter as L2 since this was determined to be the best regularization parameter in the previous model.

GridsearchCV determined the optimal value of C to be 1.481. The resulting model metrics are shown in Table 3. The resulting model shows larger accuracy on the testing set, and lower testing error than the previous model with only the regularization parameter optimized. This suggests that tuning the hyperparameter C and using the L2 regularization parameter creates a model that is incredibly accurate on the testing set, and has similar metrics when considering the training/testing metrics. This model fits the testing data well, but we further increased the complexity of the model by further tuning other hyperparameters next to see if we could get an even better fit.

Table 3. Gridsearch to find optimal C, use the best regualrization parameter from before (l2)

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.974	0.994	0.939	0.965	0.0258	0.967
Results on held out test set	0.972	0.978	0.9358	0.958	0.0280	0.964

Best C Parameter: 1.481

Next, we decided to use gridsearchCV to tune C, penalty (regularization parameter), solver, and max_iter parameters to see if we could create a model that fit the testing set even better than the previous model. The output of the gridsearch CV with all of the above hyper parameters determined the best regularization parameter as L2, best C as 1.871, best solver as saga, and best max_iter as 300. The resulting metrics of this model are shown in table 4. The accuracy of the model on the held out test set is less than the previous model (Table 3). Additionally, the testing error is higher than the previous model. Furthermore, all metrics on the testing set of this highly tuned model are worse than in the previous model. This could suggests that the complexity of this model is too high to accurately classify the test set. Additionally, this model seems to be overfitting the training set. So, just because this model is better parameterized, it does not mean that it is a better model.

Table 4. Tune C, penalty, solver, and max_iter parameters using gridsearch CV

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.979	0.987	0.957	0.972	0.0211	0.975

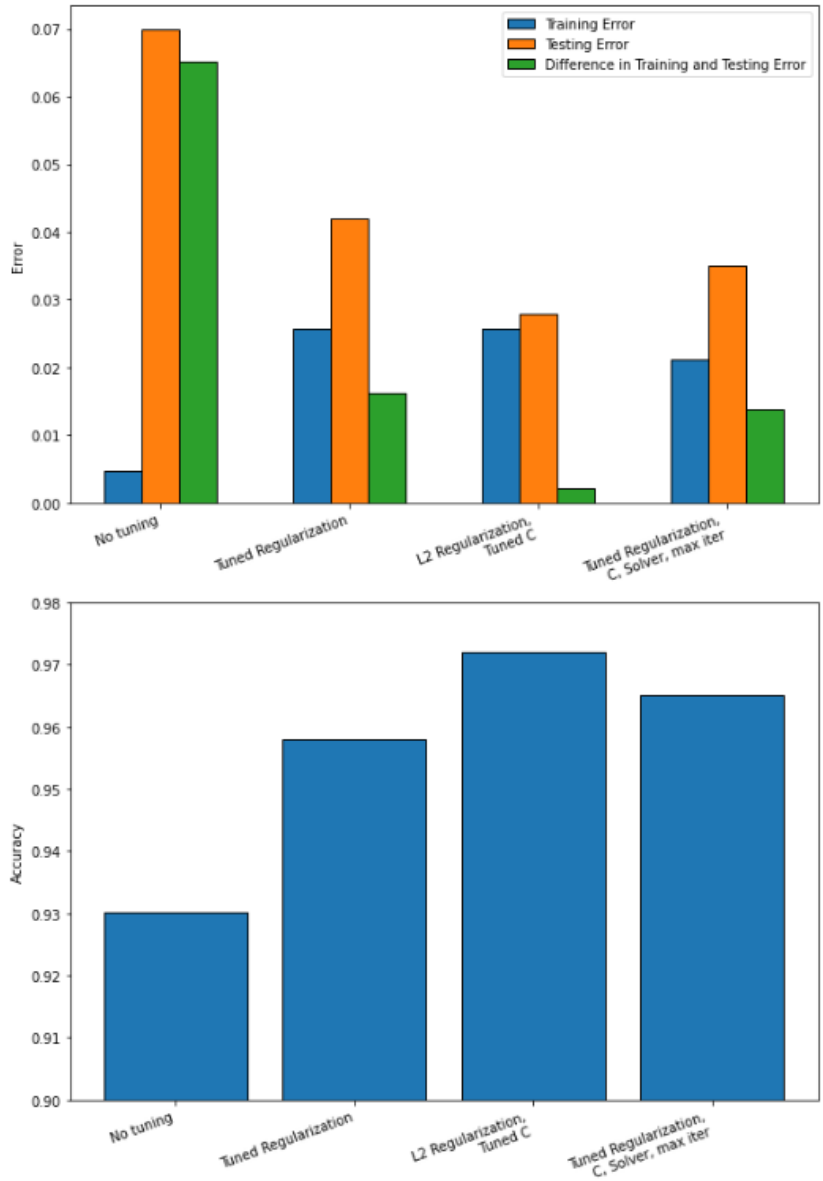
Results on held out test set	0.965	0.958	0.939	0.948	0.0350	0.959

Best regularization: L1 **Best C value:** 1.871 **Best solver:** saga **Best max_iter:** 300

Bias/Variance Tradeoff

We can use the four models generated above to discuss/analyze the bias/variance tradeoff of the logistic regression model on dataset 1. Figure 1 A) is a visualization of the training and testing error, and difference between training and testing errors for each of the 4 logistic regression model implementations for dataset 1. Testing error is incredibly high for the default model, while training error is incredibly low. This suggests overfitting, high bias, and low variance. As we tune hyperparameters, first by tuning regularization, then tuning C, testing error becomes smaller, and training error increases, while the difference between testing and training error decreases. This suggests that we are beginning to reach the optimal model complexity, therefore bias and variance are minimized. However, as we continue to add/tunem more hyperparameters, the model begins to overfit the training set once again. This is evidenced by decreased training error and increased testing error. Based on the over/underfitting analyses and discussion of training/testing error, we can conclude that the best logistic regression model for dataset 1 is the model with tuned regularization and C, resulting in the L2 regularization parameter and the 1.481 C parameter. This is further confirmed in Figure 1 B. The accuracy for the model with tuned regularization and C has the highest accuracy on the test set, over 97%..

Figure 1 | A) Training error, testing error, difference between training and testing error for each of the 4 logistic regression models for dataset 1. B) Accuracy of each of the 4 logistic regression models for dataset 1



Dataset 2

First, just like with dataset 1, we implemented logistic regression on dataset 1 using sklearn.linear_model.LogisitcRegression() from the sklearn python package. The metrics resulting from this model implementation can be seen in Table 5. Again, just as with dataset 1,

we first used the default parameters for the model and NO regularization (penalty = 'none'). We then used this model to determine the accuracy, precision, recall, F1 Measure, Training/Testing Error, and AUC metrics on both the training dataset and testing dataset. Overall, all metrics for this default model were significantly lower than the metrics reported for dataset 1. This could mean there is more variability in the dataset, thereby making it more difficult for the logistic regression classifier to classify the data observations. Additionally, this could mean that this is simply not the best classifier to use on this dataset. It is clear that the accuracy of the model on the training dataset is higher than on the testing set, but the discrepancy is not as obvious as with dataset 1. Additionally, the difference between training and testing metrics for all other metrics are not significant. However, it is likely that there is overfitting of the training dataset occurring because the accuracy of the training dataset is higher. We can tune model parameters to reduce the difference between training and testing accuracy to generate a better model.

Table 5. Base Model With Default Parameters

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.754	0.670	0.526	0.589	0.246	0.678
Results on held out test set	0.707	0.614	0.614	0.614	0.293	0.689

Next, as with dataset 1, we decided to tune the regularization parameters to create a model that better fits the testing set, i.e. better generalizes to data not within the training set. We tuned the regularization parameter using gridsearch cv. The metrics resulting from this model implementation can be seen in Table 6. The resulting best regularization parameter was L2. This slightly more complex model fit the testing set better than the default model with no regularization. The accuracy of the model on the testing set was higher than the default model with no regularization. Additionally, there was a smaller difference between training and testing errors, suggesting that there is no major overfitting occuring. This model is better than the previous model but just slightly. Next, we will further tune parameters to try to achieve an optimal model.

Table 6. Gridsearch CV to find optimal regularization parameter

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC

Results of 10-fold Cross Validation on Training Set	0.749	0.675	0.483	0.563	0.251	0.683
Results on held out test set	0.733	0.697	0.523	0.597	0.267	0.692

Optimal Regularization parameter: L2

To attempt to create a model that performs even more optimally on the held out test set, we tuned the C parameter like we did with dataset 1. We determined the optimal value of C using gridsearchCV, and kept the regularization parameter as L2 since this was determined to be the best regularization parameter in the previous model. GridsearchCV determined the optimal value of C to be 0.251. The resulting model metrics are shown in Table 7. The resulting model shows higher accuracy on the testing set, and lower testing error than the previous model with only the regularization parameter optimized. This suggests that tuning the hyperparameter C and using the L2 regularization parameter creates a model that is incredibly accurate on the testing set, and has similar metrics when considering the training/testing metrics. This model fits the testing data better than the other models so farl, but we further increased the complexity of the model by further tuning other hyperparameters next to see if we could achieve an even better fit.

Table 7. Gridsearch to find optimal C, use the best regualrization parameter from before (l2)

	Accuracy	**Precision**	**Recall**	**F1 Measure**	**Training/Testing Error**	**AUC**
Results of 10-fold Cross Validation on Training Set	0.737	0.692	0.388	0.497	0.263	0.650
Results on held out test set	0.741	0.75	0.477	0.583	0.259	0.690

Best C Parameter: 0.251

Next, like with dataset 1, we decided to use gridsearchCV to tune C, penalty (regularization parameter), solver, and max_iter parameters to see if we could create a model that fit the testing set even better than the previous model. The output of the gridsearch CV with all of the above hyper parameters determined the best regularization parameter as L1, best C as .331, best solver as saga, and best max_iter as 100. The resulting metrics of this model are shown in Table 8. The accuracy of the model on the held out test set is less than the previous model (Table 3). Additionally, the testing error is higher than the previous model. Furthermore, all

metrics on the testing set of this highly tuned model are worse than in the previous model. The accuracy of this model is actually the worst out of all of the previous models generated based on this dataset. This could suggest that the complexity of this model is too high to accurately classify the test set. Additionally, this model seems to be overfitting the training set. So, just because this model is better parameterized, it does not mean that it is a better model.

Table 8. Tune C, penalty, solver, and max_iter parameters using gridsearch CV

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.737	0.691	0.405	0.508	0.263	0.655
Results on held out test set	0.690	0.643	0.409	0.500	0.310	0.635

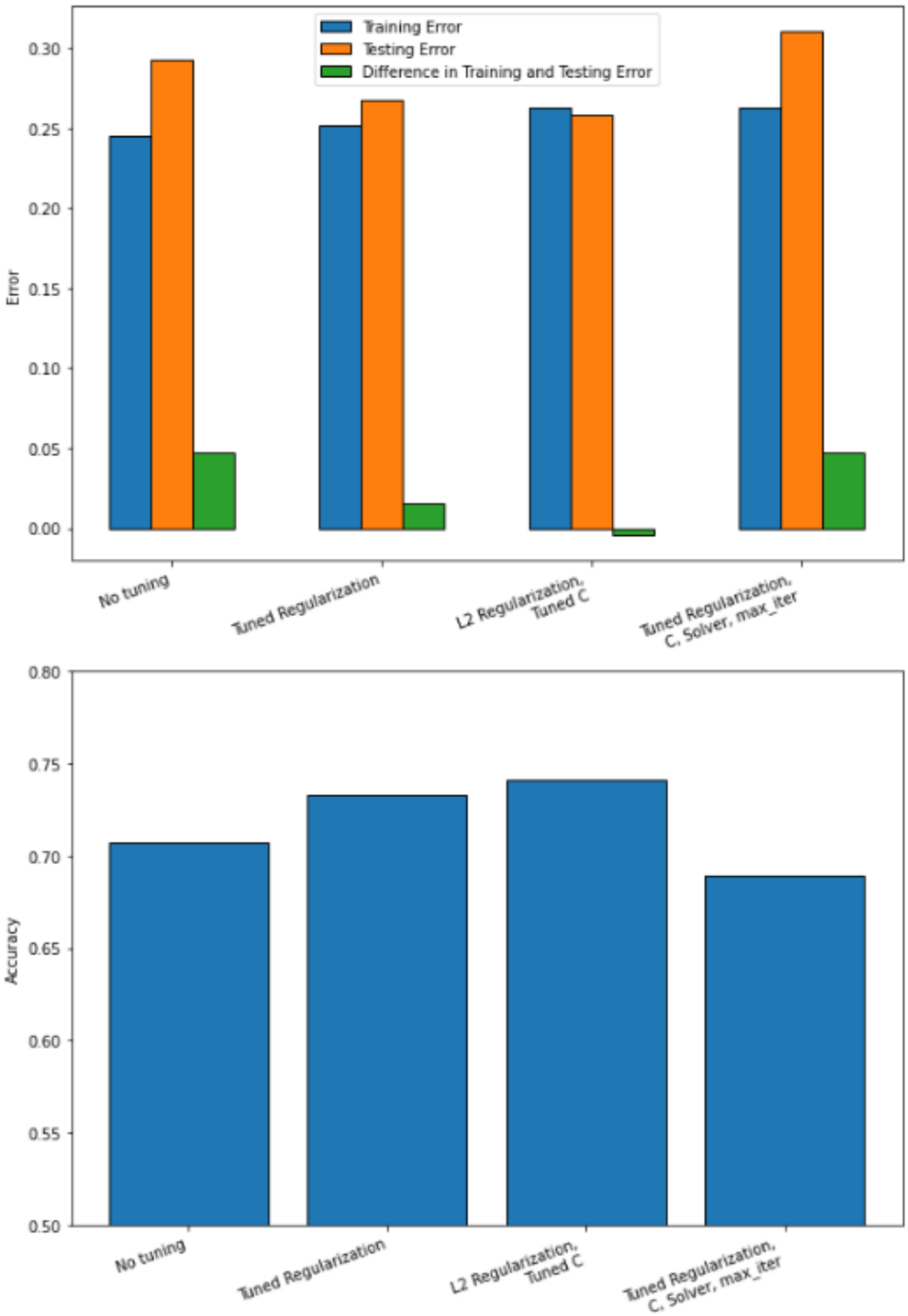
Best regularization: L1 **Best C value:** 0.331 **Best solver:** saga **Best max_iter:** 100

Bias/Variance Tradeoff

We can use the four models generated above to discuss/analyze the bias/variance tradeoff of the logistic regression model on dataset 2. Figure 2 A) is a visualization of the training and testing error, and difference between training and testing errors for each of the 4 logistic regression model implementations for dataset 2. The testing error for the model with the default parameters is quite high, but not as significantly high as the same model trained on dataset 2. This could still suggest overfitting, high bias, and low variance, although the discrepancy between training and testing error is not as large as seen with dataset 1. Nonetheless, we tuned hyperparameters to attempt to generate a model that better classifies the held out test set. As we tune hyperparameters, first by tuning regularization, then tuning C, testing error becomes smaller, and training error increases, while the difference between testing and training error decreases. This suggests that we are beginning to reach the optimal model complexity, therefore bias and variance are minimized. However, as we continue to add/tune more hyperparameters, the model begins to overfit the training set once again. This is evidenced by decreased training error and increased testing error. Based on the over/underfitting analyses and discussion of training/testing error, we can conclude that the best logistic regression model for dataset 2 is the model with tuned regularization and C, resulting in the L2 regularization parameter and the 0.251 C parameter. This is further confirmed in Figure 1 B. The accuracy for the model with tuned regularization and C has the highest accuracy on the test set. The tuned regularization/C model was also the best model choice for dataset 2. However, it is also

important to note that logistic regression might just not be the optimal classifier for this dataset since the accuracy of the classifier even on the training set is is only around 79% accurate.

Figure 2 | A) Training error, testing error, difference between training and testing error for each of the 4 logistic regression models for dataset 2. B) Accuracy of each of the 4 logistic regression models for dataset 2



DECISION TREE IMPLEMENTATION

A decision tree is a supervised learning classification algorithm represent the decision-making process through a branching structure. The decision tree algorithm splits dataset features by using a cost function. Typically, decision trees are used for classification problems like we are dealing with in this project. The base of the tree is the root node. Then, from the root there are a multiple decision nodes that depict decisions to be made. From each decision node, there are leaf nodes that represent the outcome of those decisions. Each decision node represents a split point, and leaves are the answers. There are several advantages to using a decision tree for a classification problem. Decision trees can use numerical or categorial inputs, can be used for models with multiple outputs, and generally requires less data cleaning than other techniques. However, there are also disadvantages to using a decision tree classification algorithm. First, decision trees are impacted by noise in the data, are not ideal for large datasets, decisions are limited to binary outcomes, and can disproportionately weigh attributes.

Dataset 1

Next, we implemented decision tree classification on dataset 1 using sklearn.tree.DecisionTreeClassifier() from the sklearn python package with the default parameters. The results of this model implemetation can be seen in Table 9. This model implementation with only the default parameters. No hyperparmaeter tuning/optimization was preformed. Training the model with 10 fold cross validation and then using the model on the trainin set resulted in metris that were a perfect 1.0 score. This suggests that there is a high degree of overfitting to the training data. Although the accuracy of the model on the test set was not incredibly low, we can seek to find a better model implementation that reduces overfitting on the training set while improving the accuracy of the classifier on the testing set.

Table 9. Base Model With Default Parameters

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.923	0.865	0.918	0.891	0.0769	0.922

To create a better classifier that doesn't overfit the training set and improves the accuracy of the classifier on the testing set, we can tune the hyperparameters of the model. We can use gridserchCV to find the optimal parameters for a classifier given the training set. Here, we will

use gridsearchCV to find the optimal criterion parameter. The criterion parameter of the decision tree, which determines the quality of a split in a decision tree. The output of this specific classifier are shown in Table 10. While it is still clear that there is overfitting of the training set as evidenced by metrics that are all 1.0, the accuracy of the model on the held out test set has been improved by selecting the optimal criterion parameter. Gridsearch CV determined the optimal criterion parameter to be entropy. Although tuning the criterion parameter clearly increased the classification ability of the model on the testing set, there is still room for improvement, and still room for reducing overfitting of the training set.

Table 10. Gridsearch CV to find optimal criterion parameter

	Accuracy	**Precision**	**Recall**	**F1 Measure**	**Training/Testing Error**	**AUC**
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.930	0.855	0.959	0.904	0.0699	0.937

Optimal Regularization parameter: entropy

Next, to further reduce overfitting and improve accuracy, we tuned both the criterion and max_depth parameters of the decision tree classifier using gridsearchCV. This resulted in the best criterion to be entropy, and the best max_depth to be 4. Max_depth refers to the maximum depth of the tree. The results of the classifier are summarized below in Table 10. The metrics of the classifier on the training set are no longer all 1.0 suggesting that the classifier is no longer overfitting the test set. However, the accuracy of the classifier on the test set was not improved. But, overall this classifier is better than the previous one, since there is no longer any overfitting occuring, and the accuracy of the model on the testing set was not reduced.

Table 10. Gridsearch to find optimal criterion and max_depth parameters

	Accuracy	**Precision**	**Recall**	**F1 Measure**	**Training/Testing Error**	**AUC**
Results of 10-fold Cross Validation on Training Set	0.979	1.0	0.944	0.972	0.0211	0.972
Results on held out test set	0.930	0.882	0.912	0.9	0.0699	0.927

Best Criterion: entropy **Best max_depth:** 4

While the previous model proved to be a successful implementation of the decision tree classifier that did not overfit the training set and accurately classified the testing set, we can seek to further improve the classifier by tuning more hyperparameters. GridsearchCV determine the best criterion parameter to be entropy, best max_depth to be 25, best splitter to be random, and best min_samples_split to be 4. The decision tree classifier we tried next used gridsearchcv to tune the cirterion, max_depth, splitter, and min_samples_split parameters. The metrics resulting from this classifier on the training and testing sets are shown in Table 11. The metrics resulting from this classifier are quite similar to the previous classifier with only criteron and max_depth parameters tuned. However, the recall ability of this tuned classifier is very slightly better than the previous classifier.

Table 11. Tune Criterion, max_depth, splitter, and min_samples_split parameters using gridsearch CV

	Accuracy	**Precision**	**Recall**	**F1 Measure**	**Training/Testing Error**	**AUC**
Results of 10-fold Cross Validation on Training Set	0.991	1.0	0.975	0.988	0.00939	0.988
Results on held out test set	0.930	0.882	0.918	0.9	0.0699	0.927

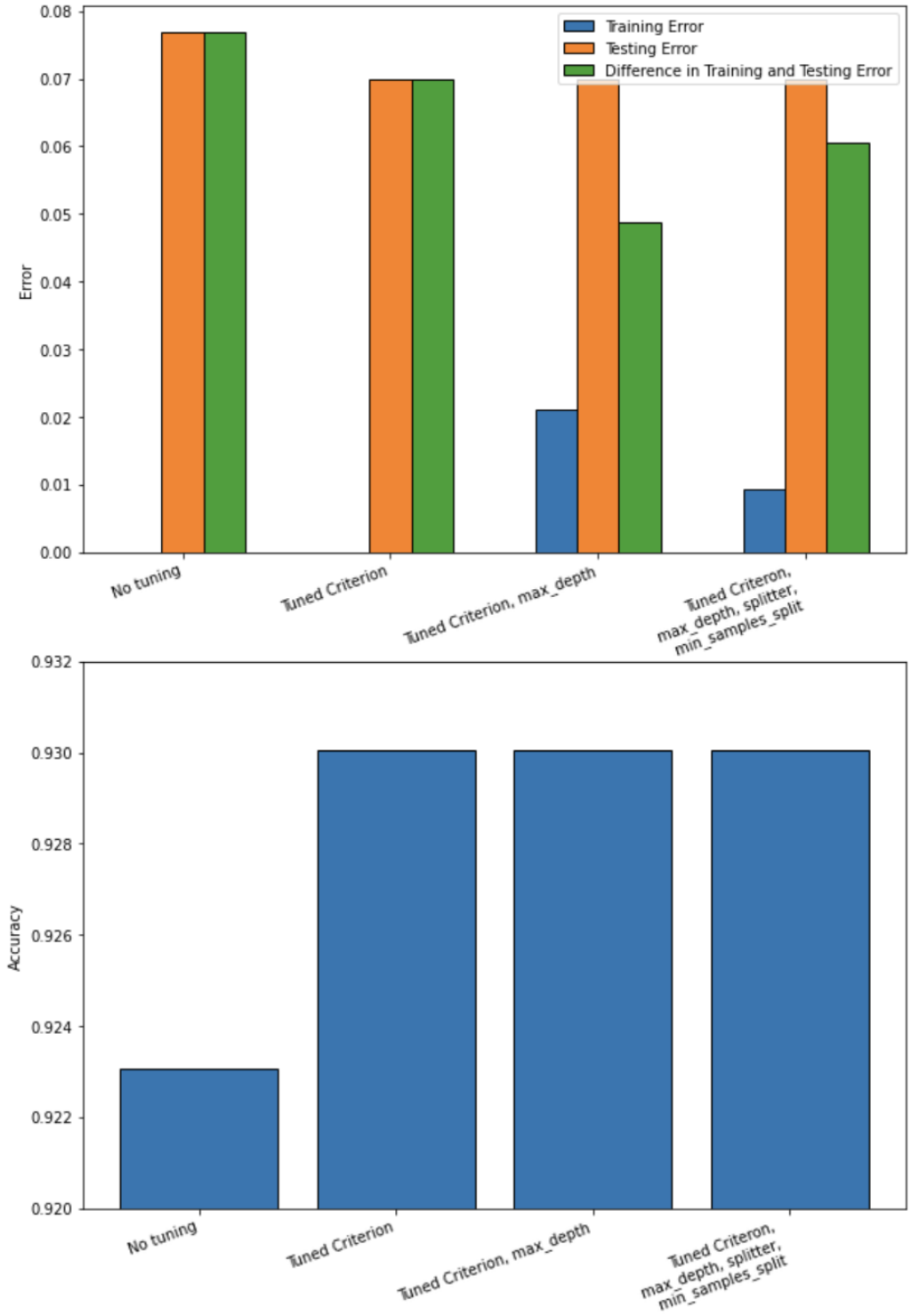
Best criteron: entropy **Best max_depth:** 25 **Best splitter:** random **Best min_samples_split:** 4

Bias/Variance Tradeoff

We can use the four decision tree models generated above to discuss/analyze the bias/variance tradeoff of the decision tree model on dataset 1. Figure 3 A) is a visualization of the training and testing error, and difference between training and testing errors for each of the 4 decision tree classifier implementations for dataset 1. The training error for both the models with the default parameters and the tuned criterion parameters is zero, but the testing error for the classifier with the tuned criterion parameter is lesst than the classifier with default parameters. This suggests that for each of these classifiers there is significant overfitting of the test set, high bias, and incredibly low variance. As we continue to tune more hyperparameters the, the training error begins to increase and the testing error decreases. This suggests less overfitting, lower bias and lower variance. The classifier with the tuned criterion and max_depth parameters and the classifier with the tuned criterion, max_depth, splitter, and min_samples_split do not differ significantly. However, the classifier with tuned criterion, max_depth, splitter, and min_samples_split has very low training error, suggesting overfitting. So, the optimal decision

tree classifier for dataset 1 should be the classifier with tuned citerion/max_depth parameters. This classifier achieves 93% accuracy on the testing set, as evidenced in Figure 3C.

Figure 3 | A) Training error, testing error, difference between training and testing error for each of the 4 decision tree classifiers for dataset 1. B) Accuracy of each of the 4 decision tree classifier models for dataset 2



Dataset 2

Next, we implemented decision tree classification on dataset 2 using sklearn.tree.DecisionTreeClassifier() from the sklearn python package with the default parameters, just like with dataset 1. The results of this model implementation can be seen in Table 12. This model implementation with only the default parameters. No hyperparmaeter tuning/optimization was preformed. Training the model with 10 fold cross validation and then using the model on the trainin set resulted in metris that were a perfect 1.0 score. This suggests that there is a high degree of overfitting to the training data. The accuracy of the model on the test set was not incredibly low and not much better than chance. We can seek to find a better model implementation that reduces overfitting on the training set while improving the accuracy of the classifier on the testing set.

Table 12. Base Model With Default Parameters

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.569	0.435	0.455	0.444	0.431	0.547

To create a better classifier that doesn't overfit the training set and improves the accuracy of the classifier on the testing set, we can tune the hyperparameters of the model. We can use gridserchCV to find the optimal parameters for a classifier given the training set. Here, we will use gridsearchCV to find the optimal criterion parameter. The criterion parameter of the decision tree, which determines the quality of a split in a decision tree. The output of this specific classifier are shown in Table 13. While it is still clear that there is overfitting of the training set as evidenced by metrics that are all 1.0, the accuracy of the model on the held out test set has been improved by selecting the optimal criterion parameter. Gridsearch CV determined the optimal criterion parameter to be gini. Tuning the criterion parameter did not change the metrics since the default criterion parameter in the decision tree classifier is gini. We can further refine other parameters to achieve a classifier that is reduces overfitting and increases accuracy on the testing set.

Table 13. Gridsearch CV to find optimal criterion parameter

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.569	0.435	0.445	0.444	0.431	0.547

Optimal Regularization parameter: gini

Next, to further reduce overfitting and improve accuracy, we tuned both the criterion and max_depth parameters of the decision tree classifier using gridsearchCV. This resulted in the best criterion to be gini, and the best max_depth to be 1. Max_depth refers to the maximum depth of the tree. The results of the classifier are summarized below in Table 14. The metrics of the classifier on the training set are no longer all 1.0 suggesting that the classifier is no longer overfitting the test set. However, the accuracy of the classifier on the test set was not significantly improved. But, overall this classifier is better than the previous one, since there is no longer any overfitting occuring, and the accuracy of the model on the testing set was improved slightly.

Table 14. Gridsearch to find optimal criterion and max_depth parameters

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.714	0.571	0.586	0.579	0.286	0.682
Results on held out test set	0.629	0.508	0.705	0.590	0.371	0.644

Best Criterion: gini
Best max_depth: 1

While the previous model proved to be a successful implementation of the decision tree classifier that did not overfit the training set, we can seek to further improve the classifier by tuning more hyperparameters to improve the accuracy of the classifier. GridsearchCV determined the best criterion parameter to be entropy, best max_depth to be 5, best splitter to be random, and best min_samples_split to be 2. The metrics resulting from this classifier on the

training and testing sets are shown in Table 15. This classifier improved the accuracy of the model on the testing set, but not significantly.

Table 15. Tune Criterion, max_depth, splitter, and min_samples_split parameters using gridsearch CV

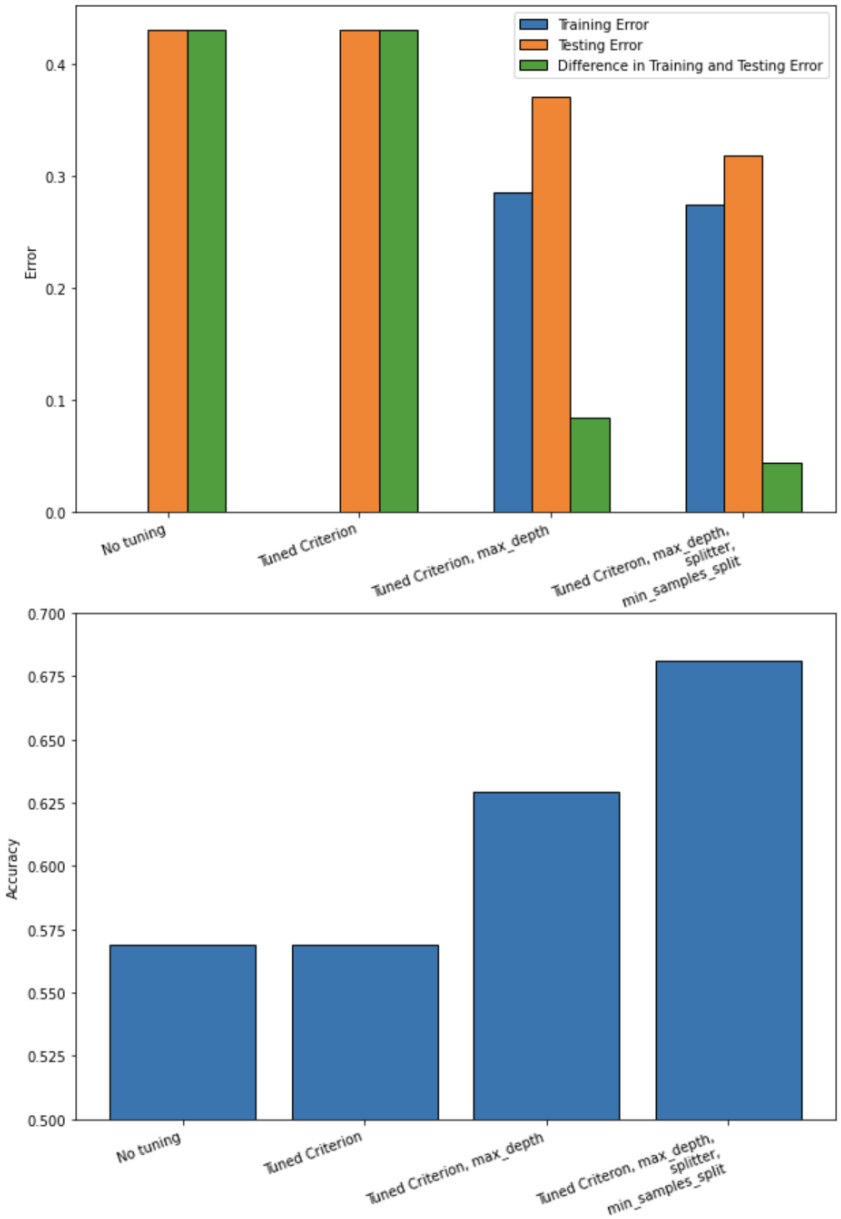
	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.725	0.590	0.595	0.592	0.274	0.693
Results on held out test set	0.681	0.600	0.477	0.532	0.319	0.641

Best criterion: entropy **Best max_depth:** 5 **Best splitter:** random **Best min_samples_split:** 2

Bias/Variance Tradeoff

We can use the four decision tree models generated above to discuss/analyze the bias/variance tradeoff of the decision tree model on dataset 2. Figure 4 A) is a visualization of the training and testing error, and difference between training and testing errors for each of the 4 decision tree classifier implementations for dataset 2. The training error for both the models with the default parameters and the tuned criterion parameters is zero, but the testing error for the classifier with the tuned criterion parameter is lesst than the classifier with default parameters. This suggests that for each of these classifiers there is significant overfitting of the test set, high bias, and incredibly low variance. As we continue to tune more hyperparameters, the training error begins to increase and the testing error decreases. This suggests less overfitting, lower bias and lower variance. The classifier with the tuned criterion, max_depth, splitter, and min_samples_split has the highest accuracy (~68%), and the lowest testing error. So, the optimal decision tree classifier for dataset 2 should be the classifier with tuned criterion, max_depth, splitter, and min_samples_split. However, this classification does not have high accuracy, suggesting that a decision tree may not be the best choice for a machine learning algorithm to classify this dataset.

Figure 4 | A) Training error, testing error, difference between training and testing error for each of the 4 decision tree classifiers for dataset 2. B) Accuracy of each of the 4 decision tree classifier models for dataset 2



RANDOM FOREST IMPLEMENTATION

Here we will implement the random forest ensemble supervised learning classification algorithm. A random forest is an ensemble of decision tree classifiers that are merged together for a more

accurate prediction. When using a random fores classifier, each tree in the forest is given a 'vote'. The forest then chooses the classifier with the majority of the votes. It is important to note that there is low correlation between the individual models in the forest. So while individual trees may produce errors, the majority of the group will be correct, therefore moving the prediction in the right direction. The advantage of using a random forest classification algorithm is that it is easy to use, is efficient, has a high degree of accuracy because it is an ensemble method. Some key disadvantages are that tree algorithms often suffer from overfitting, and can require a large amount of memory.

Dataset 1

We implemented random forest classification (an ensemble method) on dataset 1 using sklearn.ensemble.RandomForestClassifier() from the sklearn python package with the default parameters. The results of this model implementation can be seen in Table 16. This model implementation with only the default parameters. No hyperparameter tuning/optimization was performed. Training the model with 10 fold cross validation and then using the model on the training set resulted in metrics that were a perfect 1.0 score. This suggests that there is potential overfitting of the training data. Although the accuracy of the model on the test set was not incredibly low, we can seek to find a better model implementation that reduces overfitting on the training set while improving the accuracy of the classifier on the testing set.

Table 16. Base Model With Default Parameters

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.951	0.938	0.918	0.928	0.0490	0.943

Next we decided to use the parameters determined by the gridserachCV for the decision tree algorithm in the previous section, in this ensemble learning method to see if the new classifier would provide a better fit for the testing set. The results of this classifier with the hyperparameters from the decision tree classifier are shown in Table 17. With this new set of hyperparameters, the accuracy of the classifier was actually decreased from the default set of parameters. Additionally, the testing error was higher than in the default set of parameters. Next, we will turn toward gridserachCV to tune the random forest hyperparameters appropriately in hopes of better classifying the testing set.

Table 17. Base model but with optimal parameters from decision tree classifier above (criteron=entropy, max_depth = 25, min_samples_split = 4)

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.944	0.918	0.918	0.918	0.0559	0.938

In attempt to find a classifier that better classifies the testing set, we will use gridsearchCV to find the optimal n_estimators parameter. This parameter determines the number of decision trees in the random forest. GridsearchCV determined the optimal number of estimators to be 100 which is the default parameter in random forest in sklearn. So, the resulting metrics are the same as the default metrics in Table 16. The metrics for the classifier where the number of estimators was determined with gridsearchCV are shown below in Table 18.

Table 18. Gridsearch to find optimal number of estimators

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.951	0.938	0.918	0.928	0.0490	0.943

Best n_estimators: 100

To further refine the random forest classifier, we used gridsearchCV on multiple parameters including n_estimators, criterion, max_depth, bootstrap, and min_samples_split. The output of this classifier is shown in Table 19. The accuracy of this classifier was the same as the previous one, F1 measure was slightly higher. This suggests that this classifier with many tuned parameters is slightly better than the previous classifier, but overall their performance is very comparable. Although parameters were tuned, the accuracy of the classifier on the training set is still perfect (1.0) which could still suggest overfitting. However 100% accuracy on the training set is typically seen with random forest implementations.

Table 19. Tune n_estimators, gini, bootstrap, min_samples_split parameters using gridsearch CV

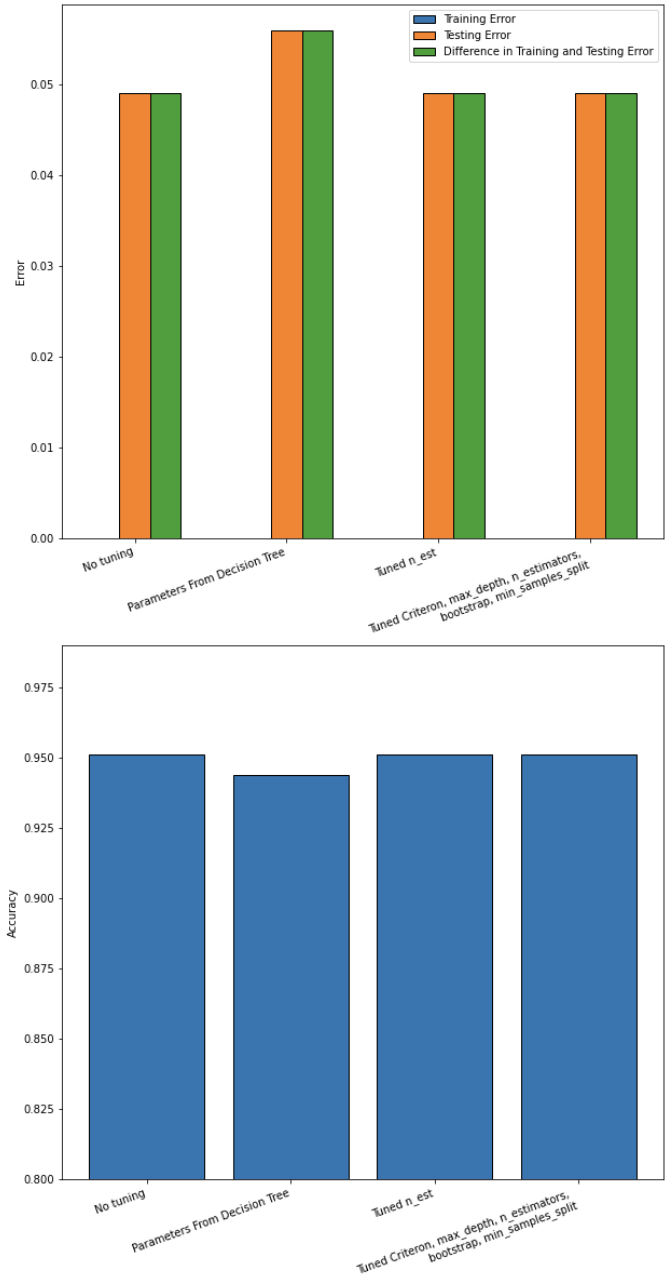
	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.951	0.920	0.939	0.929	0.0490	0.948

Best n_estimators: 50 **Best criterion:** gini **Best max_depth:** 10 **Best bootstrap:** False
Best min_samples_split: 5

Bias/Variance Tradeoff

Here, it is difficult to speak to the bias/variance tradeoff of this random forest classifier, since each of the models result in similar classification accuracy of the testing set. Additionally, for each of the random forest classifiers, the training error is always zero. Theoretically, this could suggest overfitting, high bias and low variance. But, the accuracy of the classifier on the test set for each of the random forest implementations was near 95%, which is indicative of good classification ability. However, the classifier that performed the worst was the classifier that used the optimal parameters from the decision tree implementation in the previous section. So, the optimal random forest classifier for dataset 1 would be either the implementation with default parameters, implementation with tuned n_estimator parameter, or tuned criterion, max_depth, n_estimators, bootstrap, min_samples_split, since each of these classifiers performed with 95% accuracy on the testing set.

Figure 5 | A) Training error, testing error, difference between training and testing error for each of the 4 random forest classifiers for dataset 1. B) Accuracy of each of the 4 random forest classifier models for dataset 1



Dataset 2

We implemented random forest classification (an ensemble method) on dataset 1 using sklearn.ensemble.RandomForestClassifier() from the sklearn python package with the default parameters. The results of this model implementation can be seen in Table 20. This model implementation with only the default parameters. No hyperparameter tuning/optimization was performed. Training the model with 10 fold cross validation and then using the model on the training set resulted in metrics that were a perfect 1.0 score. This suggests that there is potential overfitting of the training data. The accuracy of this implementation on the training set was very low, not much greater than chance, further confirming possible overfitting of the training dataset. We can tune the hyperparameters of this classifier to try to achieve a better accuracy on the testing set.

Table 20. Base Model With Default Parameters

	Accuracy	**Precision**	**Recall**	**F1 Measure**	**Training/Testing Error**	**AUC**
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.664	0.576	0.432	0.494	0.336	0.619

Next, to try to achieve a better testing accuracy, we used the parameters we determined to be optimal from the decision tree implementation in the previous section. The metrics of this resulting classifier are shown in Table 21. Using these parameters both increased the accuracy of the testing set, and changed the metric of the training set so they are no longer all 1.0. This suggests that the classifier is no longer overfitting the training data. Additionally, the testing error has decreased, while the training error has increased. Further suggesting that overfitting is no longer occurring. While this classifier is better than the previous one, there is still room for improvement.

Table 21. Base model but with optimal parameters from decision tree classifier above (criteron=entropy, max_depth = 5, min_samples_split = 2)

	Accuracy	**Precision**	**Recall**	**F1 Measure**	**Training/Testing Error**	**AUC**
Results of 10-fold Cross Validation on Training Set	0.870	0.938	0.655	0.772	0.130	0.817
Results on held out test set	0.716	0.690	0.455	0.548	0.284	0.665

Next, we used gridsearch CV to determine the optimal number of estimators, that is, the optimal number of trees in the random forest. GridsearchCV determined the optimal number of estimators to be 250. The metrics that resulted from this classifier are shown in Table 22. It is important to note that the metrics on the training set were all 1.0 suggesting overfitting of the training set. Additionally, the accuracy of the classifier on the testing set was less than the previous classifier that used parameters from the decision tree implementation. We can continue to tune and refine the parameter set to try to find an optimal classifier for this dataset.

Table 22. Gridsearch to find optimal number of estimators

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.672	0.593	0.413	0.5	0.328	0.626

Best n_estimators: 250

To achieve a better fit on the testing set, we tuned the criterion, max_cepth, splitter, and min_samples split parameters using gridsearch CV. Gridsearch CV determine the optimal parameters to be n_estimators = 150, criterion = gini, max_depth = 5, and bootstrap = True. The results of the classifier are shown below in Table 23. Tuning these parameters resulted in metrics that are indicative of the classifier not overfitting the training data, which is good. Additionally, the accuracy of the classifier on the testing set was higher than in the previous implementation.

Table 23. une Criterion, max_depth, splitter, and min_samples_split parameters using gridsearch CV

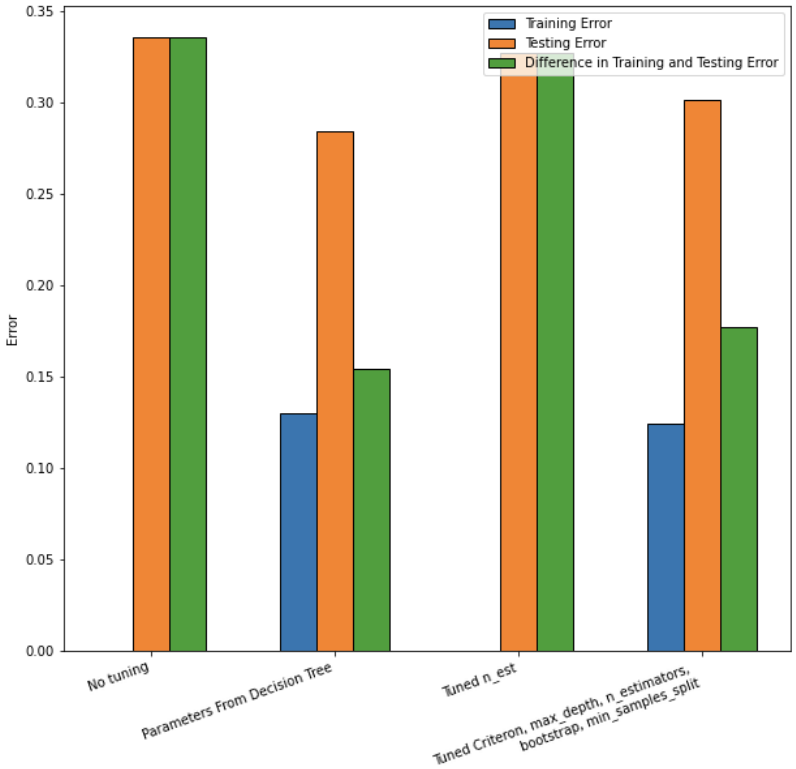
	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.876	0.929	0.681	0.786	0.124	0.827
Results on held out test set	0.698	0.655	0.432	0.520	0.302	0.646

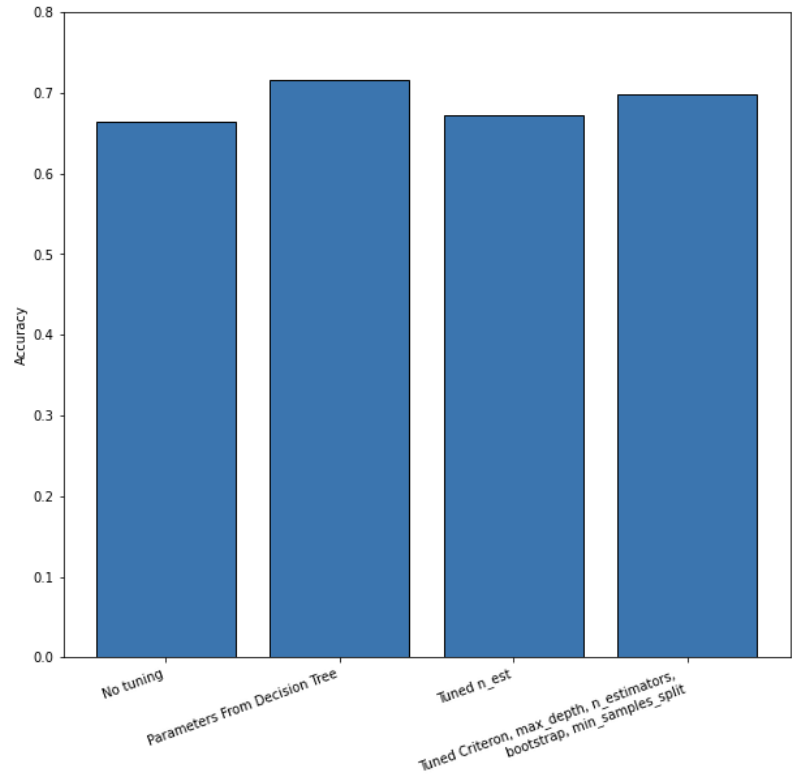
Best n_estimators: 150 **Best criterion:** gini **Best max_depth:** 5 **Best bootstrap:** True
Best min_samples_split: 3

Bias/Variance Tradeoff

We can use the four random forest models generated above to discuss/analyze the bias/variance tradeoff of the random forest model on dataset 2. Figure 4 5) is a visualization of the training and testing error, and difference between training and testing errors for each of the 4 random forest classifier implementations for dataset 2. The default implementation and the classifier with the tuned n_est parameters had the highest testing error and lowes accuracy on the testing set. This suggests overfitting of the training set, high bias and little variance. The more complex classifier implementations had lower testing error, and higher accuracy on the testing set. This suggests that these implementations are no longer overfitting the testing data and therefore have lower bias and greater variance. The optimal random forest classifier is the one that used the parameters previously defined from the decision tree implementation. This classifier had an accuracy of ~72% on the testing set.

Figure 6 | A) Training error, testing error, difference between training and testing error for each of the 4 random forest classifiers for dataset 2. B) Accuracy of each of the 4 random forest classifier models for dataset 2


Training Error
Testing Error
Difference in Training and Testing Error
Error
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
No tuning
Parameters From Decision Tree
Tuned n_est
Tuned Criteron, max_depth, n_estimators,
bootstrap, min_samples_split


Accuracy
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
No tuning
Parameters From Decision Tree
Tuned n_est
Tuned Criteron, max_depth, n_estimators,
bootstrap, min_samples_split

BOOSTING IMPLEMENTATION

Boosting is an ensemble technique that attempts to create strong classifiers from a number of weak classifiers. Boosting seeks to improve the prediction power by training a sequence of weak models, each compensating for the weakness of the one before it. Adaboost is a boosting algorithm that follows a decision tree model with a depth equal to one, making a forest of stumps rather than a forest of trees. Adaboost puts more weight on instances that are more difficult to classify and less on instances that are already handled well. Here we will implement adaboost using sklearn.ensemble.AdaboostClassifier() method in sklearn. This method works by first fitting a classifier on the original dataset and then proceeds to fit additional copies of the classifier on the same dataset. But, the weights of the incorrectly classified instances are adjusted so subsequent classifiers will focus specifically on the more difficult cases to classify.

Dataset 1

First, we implemented adaboosting with the default parameters. The metrics that resulted from this implementation are shown in Table 24. It is important to note that the metrics for the classifier on the testing set were all 1.0 with zero training error. This is indicative of overfitting of the classifier to the training dataset. However, despite the possibility of overfitting to the training dataset, the accuracy of the classifier on the held out test setting is also high. To try to increase the accuracy of the classifier on the testing set and reduce overfitting on the training set, we can tune the hyperparameters of the classifier using gridsearchCV.

Table 24. Default Adaboosting Parameters

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.958	0.906	0.980	0.941	0.0420	0.963

Next, we tuned the n_estimators parameter using gridsearchCV. The n_estimators parameter determines the number of weak learners to use in the boosting algorithm. GridsearchCV determined that the optimal number of estimators was 150. The outputs of this classifier are shown in Table 25. The metrics from this implementation are the same as the previous classifier. This is because gridsearch CV chose the optimal value of n_estimators to be 150, which is the same as the default value for n_estimators. We can continue to tune the hyperparameters of the

boosting algorithm to decrease overfitting and increase accuracy of the classifier on the testing set.

Table 25. Tuned n_estimators

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.958	0.922	0.959	0.940	0.0420	0.958

Best n_estimators: 150

Here, we will tune both n_estimators and learning rate to attempt to generate a classifier that has higher testing accuracy than previous implementations. We used gridsearch CV to tune these parameters, and ultimately determined that the best n_estimators parameters was 300 and the best learning rate was 1.1. The metrics produced by the resulting classifier are shown in Table 26. Again, we see overfitting of the training set as evidenced by the 1.0 metrics for the classifier on the training set, and 0 training error. The accuracy of the classifier on the test set was improved with this implementation. We can continue to tune more hyperparameters to make a classifier that will have high accuracy on the testing set while reducing training overfitting.

Table 26. Tuned n_estimators and learning rate

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.972	0.925	1.0	0.961	0.0280	0.979

Best n_estimators: 300 **Best learning_rate:** 1.1

Finally, we used gridsearchCV to tune the n_estimators, learning_rate, and algorithm parameters. GridsearchCV determined the best n_estimators to be 150, the best learning rate to be 1.2, and the best algorithm to be SAMME. The metrics for this classifier are shown in Table

27. There is still possible overfitting of the classifier on the training set, this is evidenced by the 1.0 metrics and the 0.0 training error. However, the accuracy of the classifier on the testing set is the highest of any of the above implementations.

Table 27. Tuned n_estimators, learning rate, algorithm

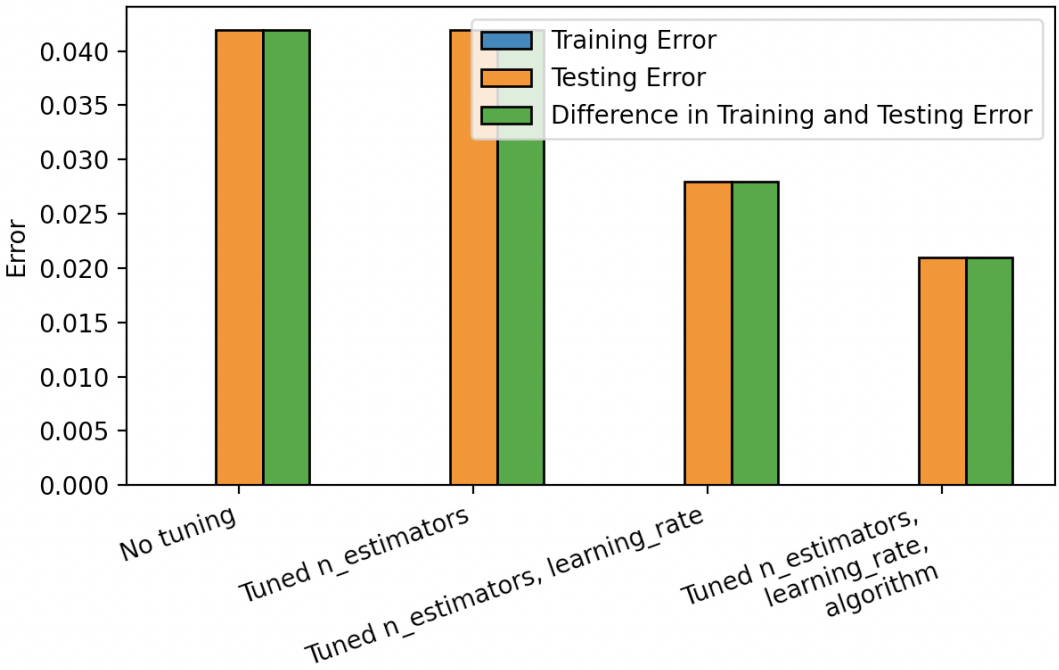
	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	1.0	1.0	1.0	1.0	0.0	1.0
Results on held out test set	0.979	0.96	0.980	0.970	0.0210	0.979

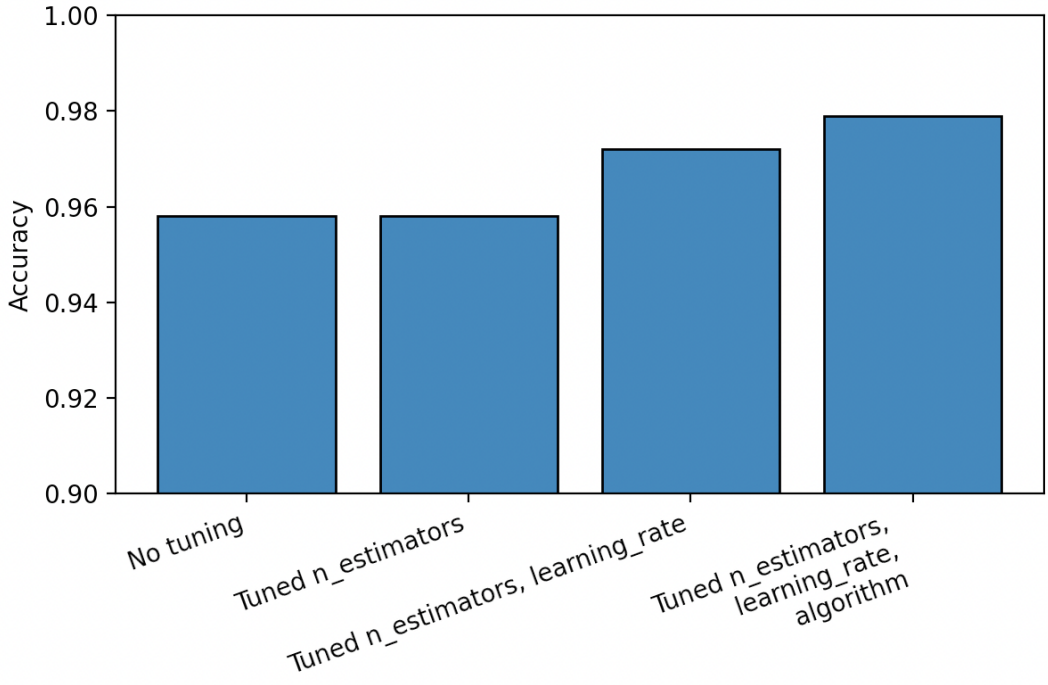
Best n_estimators:150 **Beest learning_rate:**1.2 **Best algorithm:** SAMME

Bias/Variance Tradeoff

Here, we can use each of the above boosting implementations to discuss the bias/variance tradeoff of the adaboost. The classifiers with the default adaboost parameters and the classifier that tuned the n_estimators parameter resulted in higher testing error than the other implementations. Additionally, these classifiers had zero training error, suggesting possible overfitting, high bias, and low variance. However, the classification accuracy on the testing set for both of these implementations was greater than 95%, suggesting that perhaps the classifier is not actually overfitting the training set. Nonetheless, we created two other adaboost classification implementations with different tuning of hyperparameters to try to reduce possible overfitting and increase accuracy on the testing set. Although both of these other implementations also had zero training error, the accuracy of the implementation with tuned n_estimators, learning_rate, and algorithm parameters proved to be most effective, with an accuracy of ~98%. The results of each of the boosting algorithms are shown in Figure 7.

Figure 7 | A) Training error, testing error, difference between training and testing error for each of the 4 boosting classifiers for dataset 1. B) Accuracy of each of the 4 adaboost classifier models for dataset 2





Dataset 2

First we implemented adaboost with the default parameters in sklearn.ensemble.adaboost() to classify dataset 2. The metrics of the resulting classifier on the training and testing dataset is shown below in Table 28. The accuracy of the classifier on the training and testing datasets was

not very high. Specifically, the accuracy on the testing dataset was little better than chance. We can tune hyperparameters using gridsearchCV to create a better classifier that more accurately classifies the testing dataset.

Table 28. Default Adaboosting Parameters

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.838	0.788	0.707	0.745	0.162	0.806
Results on held out test set	0.603	0.477	0.477	0.397	0.370	0.579

Next, we used gridsearch CV to tune the n_estimators parameter. Gridsearch CV determined the optimal n_estimators parameter for this dataset to be 25. The resulting classifier metrics are shown in table 29. The accuracy of this classifier on the testing set is slightly higher than the previous classifier, but there is still room for significant improvement through tuning of other parameters.

Table 29. Tuned n_estimators

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.801	0.747	0.612	0.673	0.199	0.754
Results on held out test set	0.672	0.568	0.568	0.568	0.328	0.652

Best n_estimators: 25

Here, we tuned both the n_estimators and learning rate parameters with GridsearchCV. GridsearchCV determined that the optimal n_estimators parameter was 25 and the best learning_rate parameter was 1.0. The metrics that resulted from this classifier are the same as the previous classifier, since 1.0 is the default learning rate for adaboost. We can continue tuning hyperparameters to develop a better classifier for the test set.

Table 30. Tuned n_estimators and learning rate

	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.801	0.747	0.612	0.673	0.199	0.754
Results on held out test set	0.672	0.568	0.568	0.568	0.328	0.652

Best n_estimators: 25 **Best learning_rate:** 1.0

Finally, we used gridsearchCV to tune the n_estimators, learning rate, and algorithm parameters. GridsearchCV determine the optimal n_estimators parameter to be 8, the optimal learning rate to be 1.5, and the optimal algorithm to be SAMME. The metrics of the resulting classifier are shown below in Table 31. The accuracy of the classifier on the testing set is the best of all the previous implementations shown here. Additionally, there is not evidence of overfitting with this classifier as evidenced by the metrics of the classifier on the training set.

Table 31. Tuned n_estimators, learning rate, algorithm

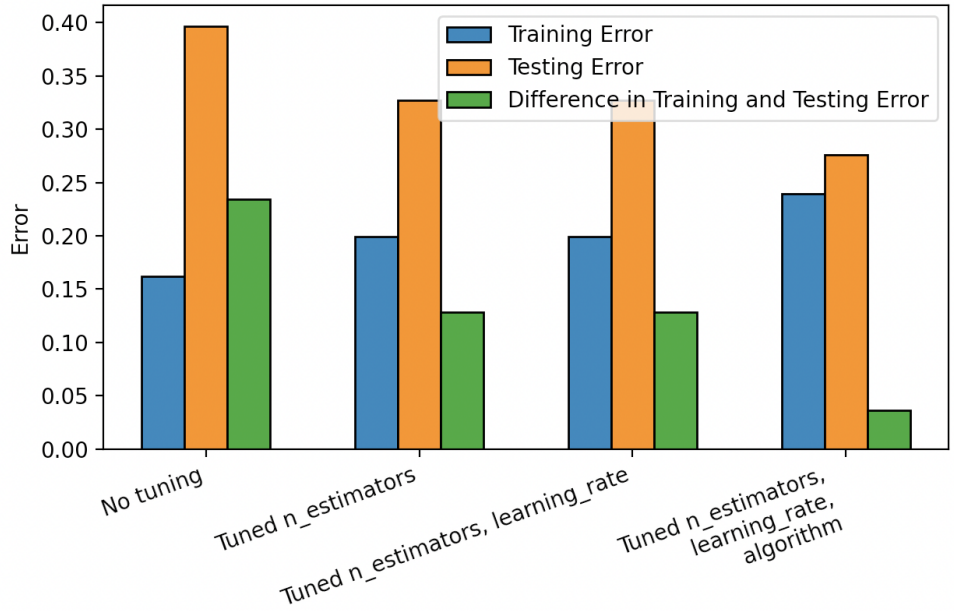
	Accuracy	Precision	Recall	F1 Measure	Training/Testing Error	AUC
Results of 10-fold Cross Validation on Training Set	0.760	0.746	0.431	0.546	0.240	0.679
Results on held out test set	0.724	0.731	0.432	0.543	0.276	0.667

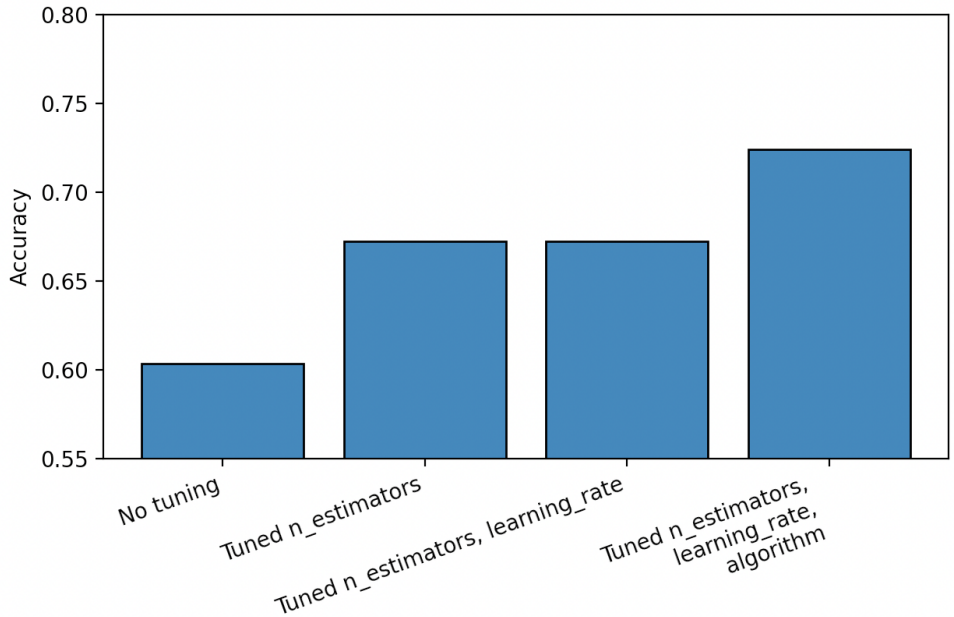
Best n_estimators:8 **Beest learning_rate:**1.5 **Best algorithm:** SAMME

Bias/Variance Tradeoff

We can use the above 4 implementations of adaboost to discuss the bias/variance of adaboost. The first implementation of adaboost with the default parameters has low training error, and significantly higher testing error. This suggests overfitting of the training set, high bias, and low variance. We can tune the hyperparameters of adaboost to reduce overfitting, reduce bias and increase variance so the classifier fits better to the testing set. After parameter tuning, the best classifier we developed tuned the n_estimators parameter to 8, the learning rate to 1.5, and the algorithm to SAMME. This developed a classifier with ~72.5% accuracy, no overfitting, low bias, and high variance.

Figure 8 | A) Training error, testing error, difference between training and testing error for each of the 4 boosting classifiers for dataset 2. B) Accuracy of each of the 4 adaboost classifiers





K Nearest Neighbors

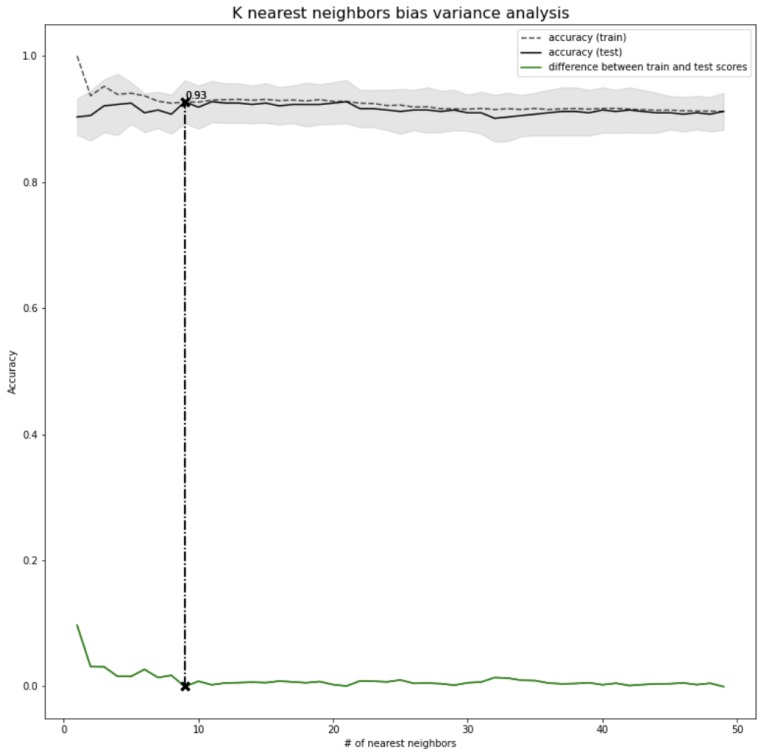
K nearest neighbors (KNN) is a supervised learning algorithm that is used to classify points based on the classification of the ‘k’ nearest points around it. KNN is a model that is simple to understand and is a nonlinear model that can predict complicated patterns in data. In order to tune the model, we need to determine the optimal value of K, the number of nearest neighbors that influences the classification of our point of interest.

Here we will implement KNN using sklearn.neighbors.NeighborsClassifier method in sklearn. GridsearchCV was used to choose the optimal value of K.

Dataset1

For dataset1, values of K from 1 to 50 were set and models were trained with a 10fold cross validation. Accuracies were reported for both the training data and the test data.
Bias and variance analysis was performed to check to see if the K parameter was creating a model that was neither under nor overfit. The results of this can be seen in Figure 9.

Figure 9: KNN bias-variance analysis



Plotted here are the training accuracy (from 10fold cross validation) with the standard deviation, the test accuracy, and the difference between training and testing error. The optimal K value was chosen as the value of K that maximized the testing accuracy. At low values of K, only the nearest point influences the classification and therefore the model has high variance and low bias. At high values of K, the classification of a point is heavily influenced by many points. As a result the model tends towards a high bias and low variance.

For dataset1 the optimal value that had the best balance between bias and variance was found to be K=9. After hyperparameter tuning, a new model was trained and tested. The optimal balance Metrics for this optimal model can be found in Table 31.

Table 31: Metrics for best KNN model on dataset1

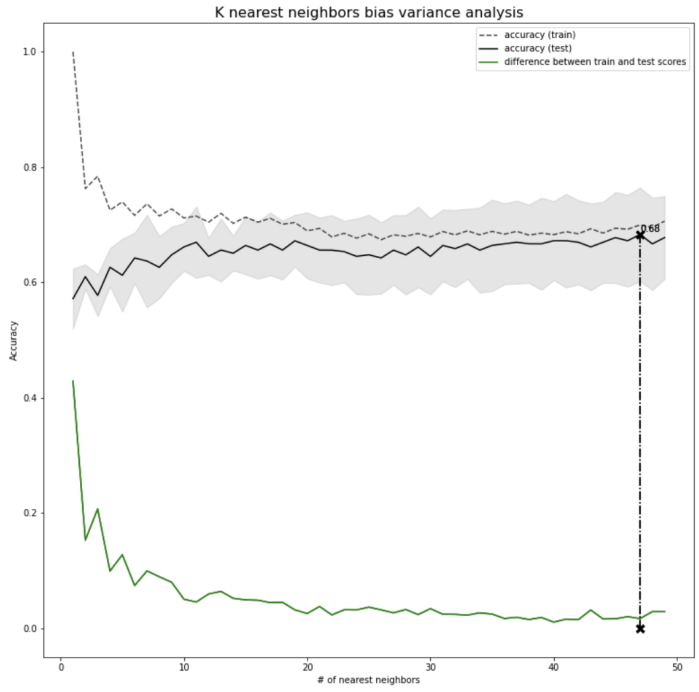
	Accuracy	Precision	Recall	F1	AUC
Mean score of 10 fold CV on training set	0.92744	0.945313	0.854779	0.894091	0.967267
Test Scores	0.947368	1	0.869565	0.930233	0.934783

You can see here that the mean accuracy for the 10 fold cross validation was 92.7%, while as for the test set it was 94.7%. You can see the other metrics in the table.

Dataset 2

Dataset2 contained factors, so before any modeling was done, the factors were converted from strings to 1 if the value was ‘present’ and 0 if it was ‘absent’. After this, gridsearchCV was used to find the optimal value of K. Overall, KNN performed much worse on dataset2 than on dataset1. You can see the results of this hyperparameter search in figure 10.

Figure 10: bias variance analysis for KNN on dataset2



For this dataset, the optimal value of K that maximized the testing accuracy while minimizing the difference between test and train accuracy was K =47. This corresponds to a model that is heavily normalized or has a higher bias and less variance. This is useful when a dataset has lots of variability and neighboring points are not always similarly classified.

A new model was trained with K = 47 and the results of this can be seen in table 32. You can see that the best model had only a 69% accuracy.

Table 32: Metrics for KNN model on dataset2

	Accuracy	Precision	Recall	F1	AUC
Mean score of 10 fold CV on training set	0.682883	0.64246	0.315385	0.410354	0.728511
Test Scores	0.688172	0.411765	0.269231	0.325581	0.559989

Support Vector Machines

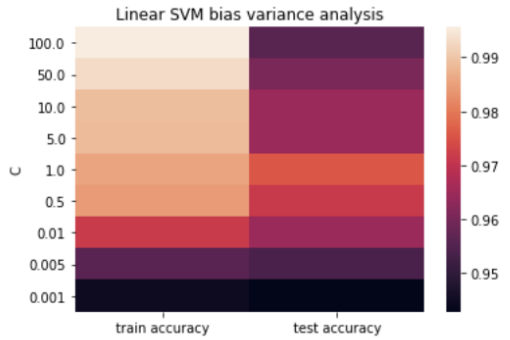
Support vector machines is a machine learning classification developed in Bell labs in the 1990s that quickly became widely used as an effective classifier due to its flexibility. The basic idea behind this method is to find the support vectors that define a region that can have misclassification. This is useful in the case where you have a dataset that has 2 fairly well separable and distinct classifications, but also have a noisey region at the intersection of these classifications where a normal hard classification would cause many mislabelings.

There are many hyperparameters that can be tuned in SVM. Here, I focused on tuning the kernel (linear, polynomial, or rbf), C (inversely proportional to the size of the margin), for polynomial kernel the degree of the polynomial, and for rbf the value of gamma (inversely proportional to the range of points to include when making a classification on a given point).

For this model, I used the sklearn standard scaler preprocessing tool to scale the data to have a zero mean and a standard deviation of 1 for each feature to help the models perform better.

First, I created a model to classify dataset 1. The first type of model I checked was a linear SVM. with different values of C. The results of this can be seen in figure 11.

Figure 11: Linear SVM bias variance analysis

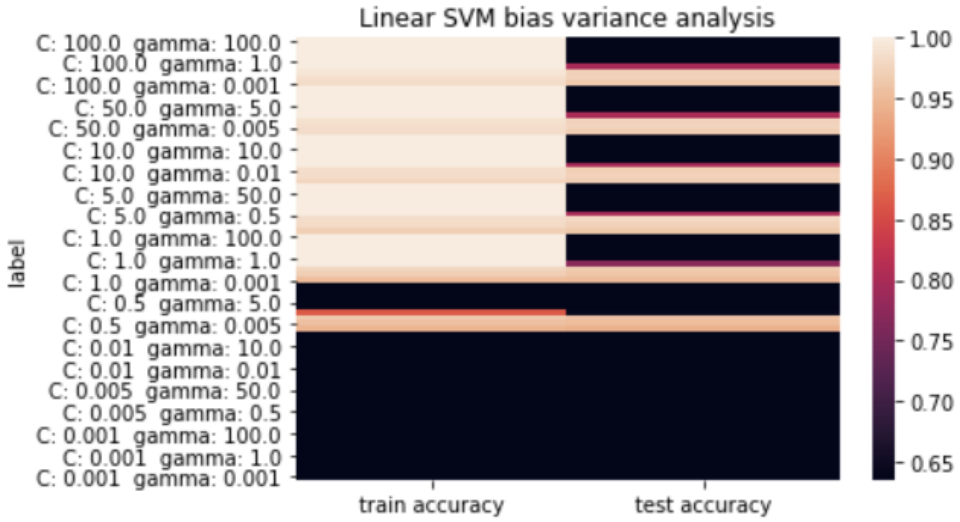


C is a parameter that sets how wide we want the margin to be. We can see that small values of C have lower accuracy (and lower error rates), whereas larger values of C have smaller margins but lower

training error. There is a sweet spot in testing error at the optimal value of C which here is C = 1.0, where the testing error is minimized.

I then tested a polynomial kernel and an rbf kernel with different degrees of polynomial, different values of C and different values of gamma. The results of the rbf kernel is shown in a heatmap in figure 12.

Figure 12: RBF SVM bias variance analysis



The rbf kernel performed better than the linear kernel or the polynomial kernel. The optimal parameter values here were C = 5 and gamma = 0.01. I then retrained an SVM with an rbf kernel with those optimal parameters as my final model. The metrics for this model can be seen in table 33. This model performed very well and was able to classify the data with a high degree of accuracy as well as high values across all metrics of interest.

Table 33: dataset1 SVM metrics

	Accuracy	Precision	Recall	F1	AUC
Mean score of 10 fold CV on training set	0.980145	0.988194	0.957353	0.971457	0.993645
Test Scores	0.973684	0.977778	0.956522	0.967033	0.970908

Dataset2

For dataset2, after scaling as described for dataset1, I fitted an SVM model with a linear kernel, polynomial kernel, and rbf kernel in the same way as described for data set 1. You can see an analysis of the bias and variance in figures 13 and 14. The best model was an RBF kernel model with a gamma value of 0.01 and a C value of 20. This means that the margin is small, but many points influence the classification of other points (from the gamma value). Once again, the model performed much worse for dataset 2 than for dataset 1.

Figure 13: Linear SVM bias variance analysis

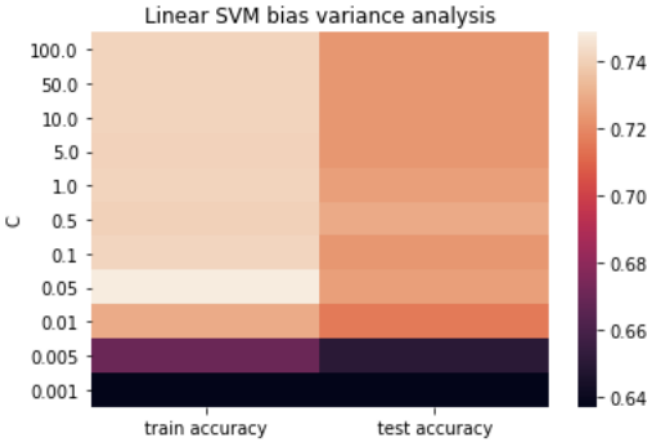
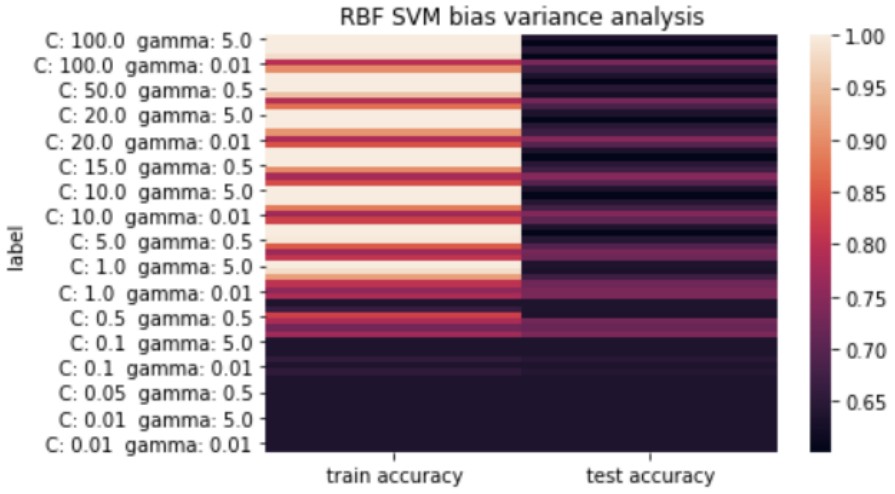


Figure 14: RBF SVM bias variance analysis



After this, I created a new rbf SVM with those opimtal parameters and retrained and tested the model. Results of this can be seen in table 34. We can see that the training accuracy is very high compared to the testing accuracy for larger values of C, indicating overfitting to the training data. Our optimal set of parameters was an attempt to minimize this overfitting while still getting high accuracy.

Table 34: results of SVM model on dataset 2

	Accuracy	Precision	Recall	F1	AUC
Mean score of 10 fold CV on training set	0.74527	0.692114	0.537912	0.600444	0.760844
Test Scores	0.731183	0.518519	0.538462	0.528302	0.672216

Best Classifiers:

Overall the best classifier for dataset1 was the **boosting algorithm** with a testing accuracy of 97.9%

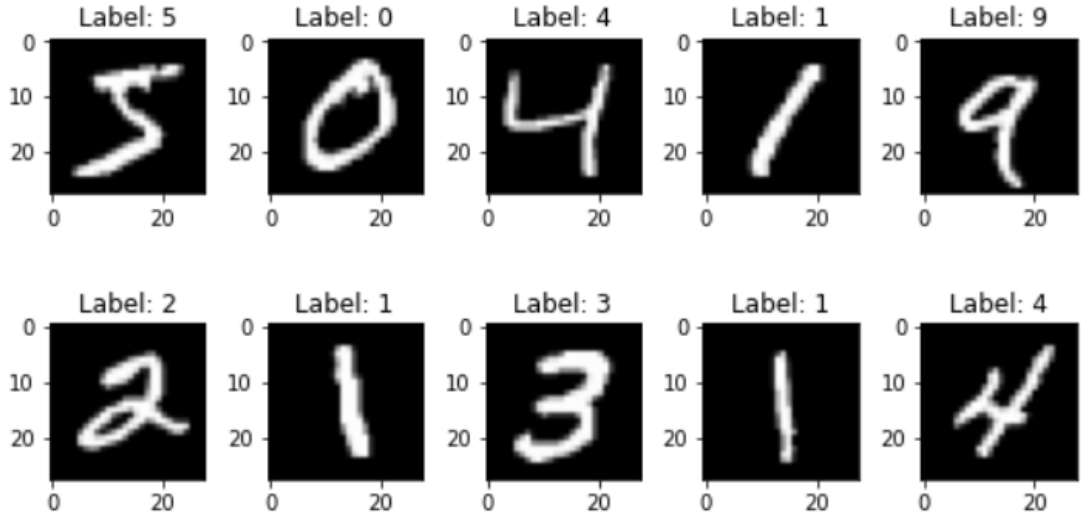
The best classifier for dataset2 was the **rbf kernel SVM algorithm** with a testing accuracy of 73.11%

Deep Learning and MNIST dataset

Deep learning and Neural networks are relatively new machine learning architectures that are able to incorporate a high degree of complexity and nonlinearity in their predictions due to the complex architecture employed. Neural networks are the general name for machine learning methods based off of the information exchange between biological neurons. In these networks nodes are connected between layers. These layers are trained to have different weights which is correlated to the importance of a node in influencing another node it is connected to. Deep learning is a subset of neural networks, where there are multiple hidden layers, or layers of neurons whose input is other neurons who have been trained on the input images. These hidden layers never actually see the original images, but rather 'learn' important features based on the important features other neurons learned. In the end, we are left with a stacked system that allows for abstract and complex patterns to be discerned.

In this part of the project, our goal was to implement a Deep learning model that classified images from the MNIST (Modified National Institute of Standards and Technology) database, images of handwritten digits from 0 to 9, see figure 15. This dataset is often used as a benchmark for developing machine learning algorithms. These images are a square of 28x28 pixels each with a value from 0 to 1 corresponding to the shade of white. These images are first flattened into a 748x1 vector before being fed into the network.

Figure 15: example images from MNIST dataset

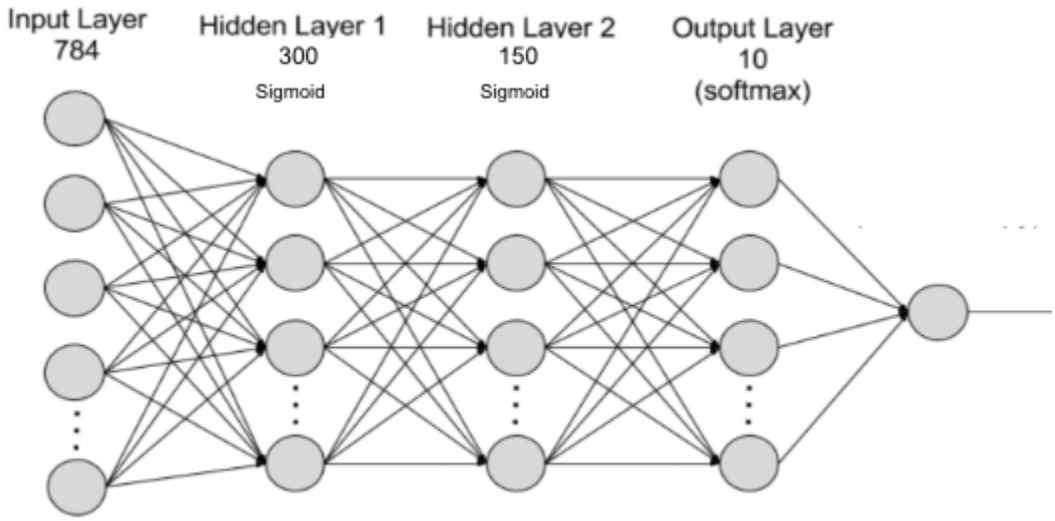


For this model, I built a Deep learning model with 2 hidden layers using the keras API in the tensorflow library in python. A figure of the architecture of the model is seen in figure 16. For the loss function, I used a sparse categorical cross entropy loss and used an 'adam' optimizer. I use accuracy as my main metric

in training. Finally, all models were trained through 2 epochs and while various batch size values were checked, a batch size of 10 was most frequently used.

For this model, I did a parameter search of various different hyperparameters. I tested various numbers of hidden units in each hidden layer. The number of hidden units tested was (2,2), (10,10), (70,70), (150,150), (300,300), (300,150), and (150,300).

Figure 16: architecture of final Deep learning model used in this project



I also tested different learning rates to see how these affect performance. The learning rate values I tested were: 1E-1, 1E-2, 5E-3, 1E-3, 5E-4, 1E-4, 1E-5, 1E-6, 5E-7. Finally I used different initialization weights to see how these affected the model performance. The methods I used to initialize weights were: Random uniform, random normal, glorot uniform, glorot normal, truncated normal, ones, zeros.

Changing the number of hidden units played a large role in determining model performance. Accuracy was much better with larger numbers of hidden units, and the model with hidden layers having (300,150) hidden units was the best, see the code Deeplearning.ipynb for more details. For learning rate, this parameter also affected model performance, though I had to train for more epochs to truly see the effects. See figure 18 for an analysis of how the different learning rates affected model performance.
Finally, in checking different initialization weights, these had a large impact on how the model performed. This surprised me, as I thought that a few epochs would be able to get all the different initializations to converge but this was not the case. The best initialization method ended up being the glorot uniform method.
The python summary of the final model can be seen in figure 17. The metrics for how it performed can be seen in table 35. After 2 epochs, the model was able to classify the images with almost 97% accuracy.

Figure 17: Final model summary

final model

```
initializer = tf.keras.initializers.GlorotUniform(seed = 42)

model = keras.Sequential()

model.add(Input(shape=(784,))) # Input tensor
model.add(Dense(units=300, kernel_initializer=initializer, activation = 'sigmoid', name="hidden_layer_1"))# hidden layer 1
model.add(Dense(units=150, kernel_initializer=initializer, activation="sigmoid", name="hidden_layer_2"))#hidden layer 2
model.add(Dense(units=10, kernel_initializer=initializer, activation="softmax", name = 'predictions'))#output layer

model.summary()

loss_fn = keras.losses.SparseCategoricalCrossentropy()
opt =  tf.keras.optimizers.Adam(learning_rate = 0.001)
model.compile(loss= tf.losses.categorical_crossentropy, optimizer='adam', metrics=['accuracy',
    tf.keras.metrics.MeanSquaredError(),
    tf.keras.metrics.AUC()])
layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

```
Model: "sequential_354"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

Figure 18: learning rates and accuracy

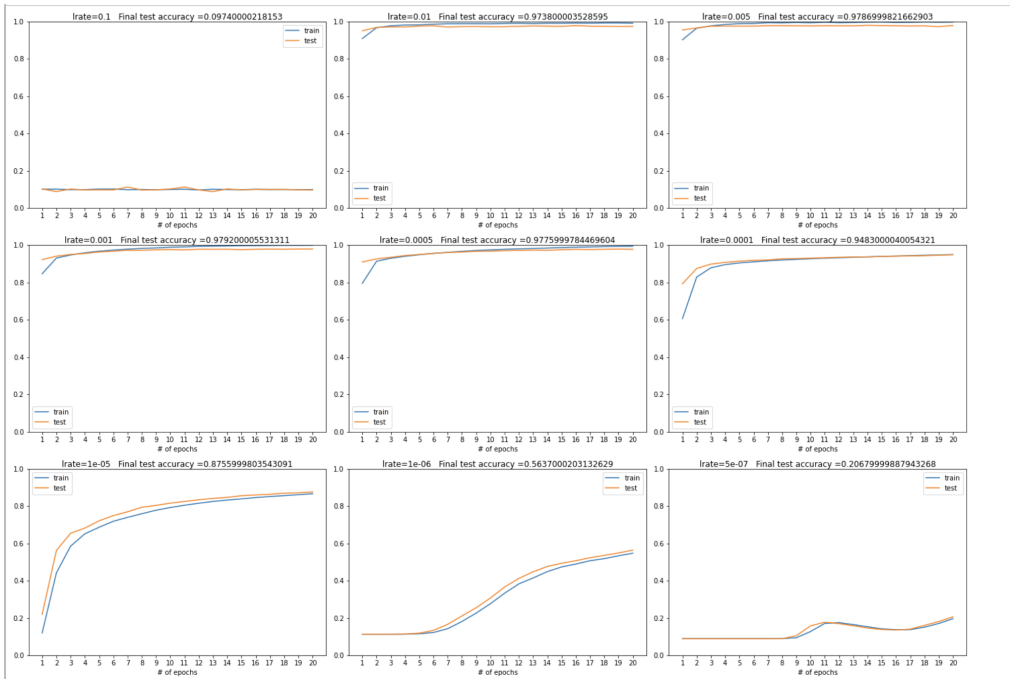


Table 35: Metrics for Deep learning Model

```
Fit model on training data
Epoch 1/2
5000/5000 [==============================] - 12s 2ms/step - loss: 0.3135 - accuracy: 0.9064 - mean_squared_error: 0.0140 - auc_348: 0.9931 - val_loss: 0.1480 - val_accuracy: 0.9573 - val_mean_squared_error: 0.0067 - val_auc_348: 0.9974
Epoch 2/2
5000/5000 [==============================] - 11s 2ms/step - loss: 0.1223 - accuracy: 0.9627 - mean_squared_error: 0.0057 - auc_348: 0.9981 - val_loss: 0.1021 - val_accuracy: 0.9679 - val_mean_squared_error: 0.0049 - val_auc_348: 0.9986
```