Predicting Wine Quality Scores: A Multi-Class Classification Problem

Farris Nabulsi, Jialin Cui, Zhi Wang, Matthew Hess

***Abstract*—After comparing and analyzing five different machine learning models, we were able to correctly predict wine quality, based on 12 given features, with an accuracy of 70%. This accuracy was achieved by using the Decision Tree classification model. Although the dataset provided us with 12 features to help predict the wine quality, some of these features, such as the total sulfur dioxide, proved to be collinear to the ground truth quality labels and were better left out. After utilizing each model for our problem, we were able to understand that some models, such as the Deep Neural Network, are better for larger datasets and tend to produce poor results with a small dataset like this. These conclusions will be further explored throughout the course of this paper.**

Dataset: https://archive.ics.uci.edu/ml/datasets/wine+quality

I. Introduction

THIS paper will thoroughly analyze a multitude of approaches to solving the classification problem of predicting wine quality scores. Using five different models, we will adjust each model's hyper-parameters with the goal of getting the optimal accuracy. The models we will be analyzing are: Logistic Regression, Deep Neural Network, K Nearest Neighbors, Random Forest, and Decision Tree. When solving this wine quality problem using each model, we will look closely at our accuracy and how it changes when we change our hyper-parameters and alter our feature selection.

Our first model that we will use is one of the most common and basic models, Logistic Regression.

II. Logistic Regression

For our Logistic Regression model we used the sklearn library, which uses the coordinate descent method; however, when manually coded, we use the gradient descent method to reduce the loss of the model's predictions. This is done by taking the partial derivative of the cost function with respect to each feature's weight. We then pass this value to an activation function (sigmoid in our model) that will give us the predicted class. The formal equation of this core function is:

$$\frac{\partial F(\theta)}{\partial \theta}: \quad \theta_j = \theta_j - \frac{1}{m}\sum_{i=0}^{m}(h_\theta(x_i) - y_i)\,x_i$$

$$\{let\ h_\theta(x) = g(\theta^T x) \quad where\ g = \frac{1}{1+e^{-x}}\}$$

A. Initial Results

We used the sklearn library to apply this model to our dataset. Initially using the default values for our hyper-parameters we got an accuracy score of 0.4928 which is about 50% accuracy. Initially this shocked me because it was so low; however, after examining the data within the dataset, the features do not have a high variance. Our model was always better than chance which is 1/12 or 8.33% which was a good indicator that our model was not completely wrong. After normalizing the data using sklearn's Standard Scaler, our accuracy rose about 5% with an accuracy score of 0.5549. In order to get the optimal performance with this model, we looked into each hyper-parameter and found their optimal values.

B. C Hyper-Parameter

The C hyper-parameter is the inverse of the regularization constant; therefore, a small C correlates with strong regularization. To test which C is optimal we tested 300 different C-values ranging from 0.1 to 30, incrementing every 0.1. Below is the graph of C-values to their error:

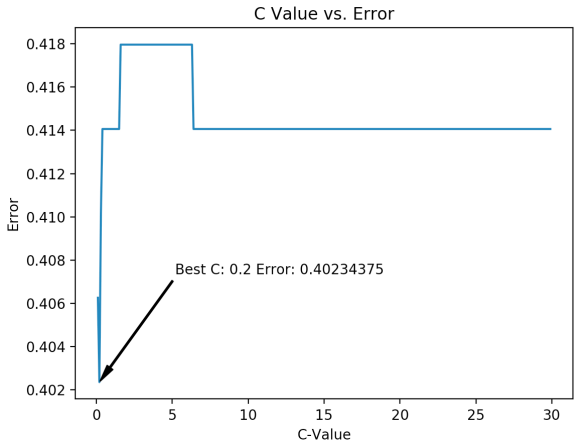

Fig. 1. This graph illustrates the change in error when the regularization parameter "C" is modified. C-values increment by 0.1. The optimal C is the one with the minimum error, which is pointed out by the arrow.

As seen in Figure 1, a C-value of 0.2 will yield the lowest error so that is the value we will use from here on out.

Looking at the graph, the error does not fluctuate too not and has a range of about 0.015. Because our dataset it not too big and we normalized our data initially, regularization will not affect our overall error; however, if this dataset were a lot bigger, our hypothesis is that this range would be a lot larger. When deciding our C-value, we used the default sklearn values for the rest of the hyper-parameters as a control. To assure that we prevent overfitting, the error in the above graph is based off of the test data. Now we will examine the best maximum iterations of training our model will endure.

C. Maximum Iterations Hyper-Parameter

When training the model, the maximum iterations is important as too many iterations can cause the model too diverge and too little can cause the model to never converge. To test the number of iterations, we tested 100 different values, from 100 to 10,000 each incrementing by 100. Below is the graph delineating the error associated with each iteration count:

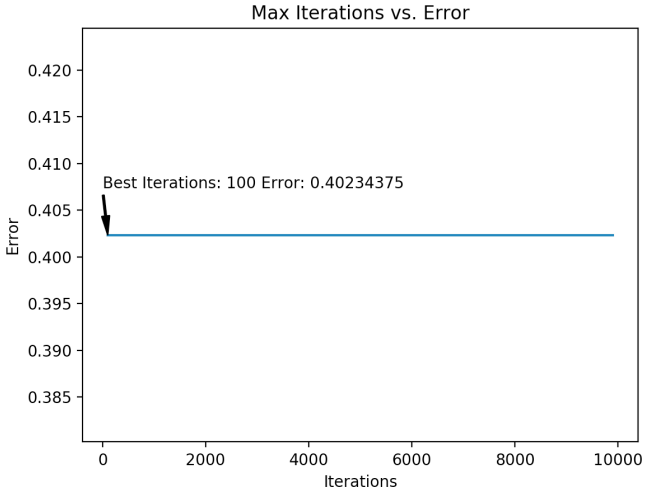

Fig. 2. This graph illustrates how the error of the Logistic Regression model is affected by modifying the maximum iterations parameter. Number of iterations increment by 100. The flat line indicates no change in the error.

Looking at Figure 2, we can see that the maximum iterations have no effect on the error of the model. After looking into the documentation of sklearn's logistic regression function, they actually do not use gradient descent and rather uses an optimization method known as Coordinate Descent. This made sense as this method of optimization does not rely on a high amount of iterations to converge, whereas gradient descent in fact highly relies on the number of iterations.

Using maximum iterations of 100 and a C value of 0.2 we get an accuracy of 0.5678 which is better than our initial accuracy; however, it is still low, considering most machine learning models tend to yield over 90% accuracy.

D. Feature Selection

Even with the most optimized hyper-parameters for this model, our accuracy was low so we decided to look into feature selection where hopefully we would find a way to bump up our accuracy even more. Below is a graph showing the error when we drop a certain feature.

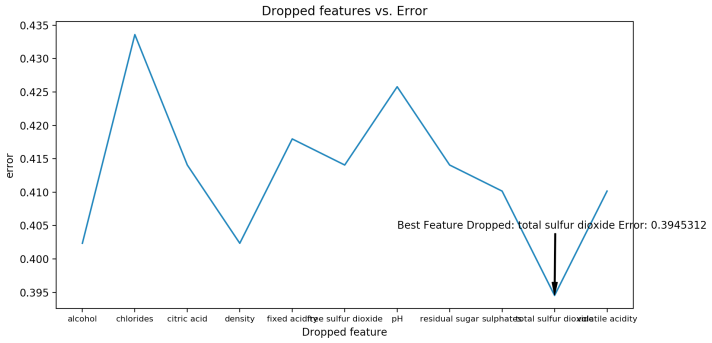

Fig. 3. This graph displays the change in the model's error in relation the dropping certain features. Each tick on the X-axis indicates the feature that was dropped. The optimally dropped feature is indicated by the arrow.

This gave us our highest fluctuation in our error and was interesting to look at. Through this graph we were able to see that some features were in fact collinear and actually worsened the model's accuracy. After removing the "Total Sulfur Dioxide" we were able to get our accuracy up to 61.5% which was a 5% improvement upon our previous best. This looked like our limit for the logistic regression so we moved on to another classification model in an attempt to raise our accuracy.

E. Confusion Matrix and More Results

With the optimal parameters that we got from the previous test, we calculated the confusion matrix to display more about which wine qualities we were predicting wrong. Below is the confusion matrix when our model is given our test data, both numerically and as a heat map:

```
[[0,  0,  0,  2,  0,  0]
 [0,  0,  7,  2,  0,  0]
 [0,  0,  102, 30, 0,  0]
 [0,  0,  46,  78, 3,  0]
 [0,  0,  2,   37, 3,  0]
 [0,  0,  1,   6,  1,  0]]
```

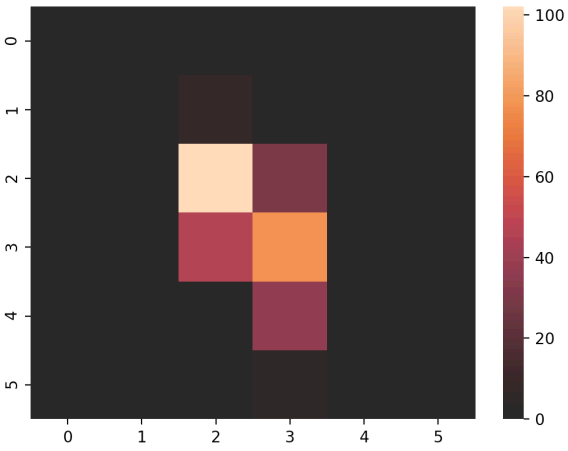

Fig. 4. This chart displays the confusion matrix of the Logistic Regression model after being run on the test data. The lighter the region, the more dense. The diagonal of the chart represents the correctly predicted values.

Looking at Figure 4, we can see that most of the times our model is wrong is when predicting the higher wine quality

scores. This could mean that the features do not vary too much when it comes to a score of 4 and a score of 5. Hopefully one of our next classification models will be able to further differentiate these classes and raise our overall accuracy.

III. Deep Neural Network

To emulate a deep learning model, we used the MLPClassifier package in the sklearn library. Using the neural network model, we were able to get an optimal accuracy of 51%. Although this accuracy is still low, the neural network had a lot more hyper-parameters that were interesting to fiddle around with to get the highest possible accuracy.

A. Initial Results

The initial run of this model with the sklearn default values and one hidden layer of size 15 yielded one of the worst accuracies that we saw, sitting at 44.32% accuracy. Our hypothesis going into this model was that it would produce one of the highest accuracies because it is such a complex and smart model; however, this was not the case. Every time we would add hidden layers, our accuracy would peak at around 50% and then decrease due to overfitting. This performance goes back to the features in this problem having a low variance and being collinear.

B. Hidden Layers

A critical part of designing a neural network is the number of hidden layers and their size. This hyper-parameter decides the complexity of the model, so it was one of our top choices of parameters we would like to analyze. To test the optimal number of hidden layers for our model, we tested different amounts of hidden layers, each of size 50 to keep some aspect of the parameter constant. Below is a graph showing the error that each model produced on our validation data.

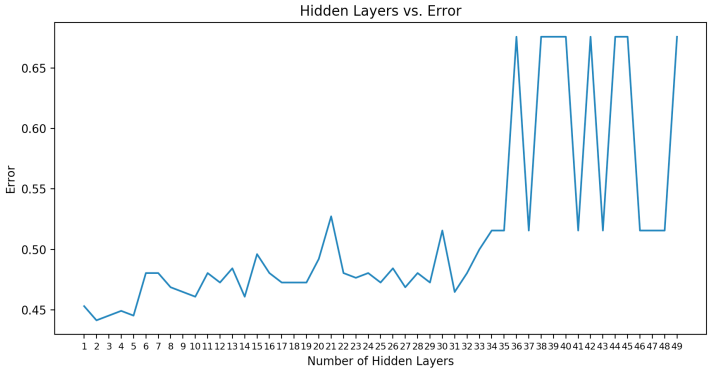

Fig. 5. This graph displays the error associated with each hidden layer size. Hidden layers of size 1 through 50 were tested and reported on this graph. The minimum error was generated by a hidden layer size of 2.

Looking at Figure 5, we can see that the least error was generated by using a neural network with two hidden layers. As we increase the number of hidden layers we can see that the error is very inconsistent and fluctuates highly. This is caused when the model is over fitted to the dataset. After about 30 hidden layers, the error is very volatile because a 30 layer neural network is overly complex for a simple classification problem.

C. Learning Rate

The neural network model adjusts its weights in a similar way that logistic regression does. The algorithm runs forward and predicts a set of features and then uses the back propagation algorithm to adjust the weights based on the error. This is done by minimizing the following cost function:

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_\theta\left(x^{(i)}\right)\right)_k +$$

$$(1 - y_k^{(i)})\log\left(1 - (h_\theta\left(x^{(i)}\right))_k\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\theta_{ji}^{(l)})^2$$

In the equation, λ represents the learning rate which, in other words, is how much we adjust our weights every iteration. If this parameter is too large then we may diverge from our target weights; moreover, if it is too small we may never converge. This parameter is important to get right so we decided to visualize its effects on our model.

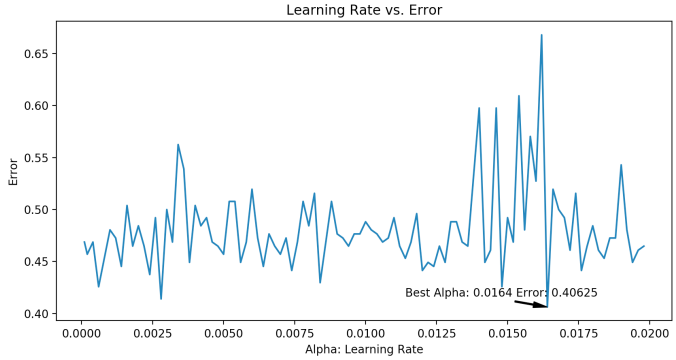

Fig. 6. This graph illustrates the error that the neural network produces when fit with the designated alpha. Alphas values are tested between 0.0001 and 0.02, each incrementing by 0.0002.

Looking at Figure 6, we can see that our optimal alpha is at 0.0164. When deciding the range of alphas to test, we did not want to go over 0.02 because anything above that was too fast of a learning rate. Now that we have our optimal hyper-parameters for our neural network, we will take a look at feature selection and how that affects our accuracy.

D. Feature Selection

Compared to logistic regression, dropping features was a little different for our neural network model. No matter how many times we ran the graph for our logistic regression model, sulfur dioxide was always the best feature to be dropped. When running the graph for this model, the results were inconsistent and would yield a different result every time. Here is one of the outputs:

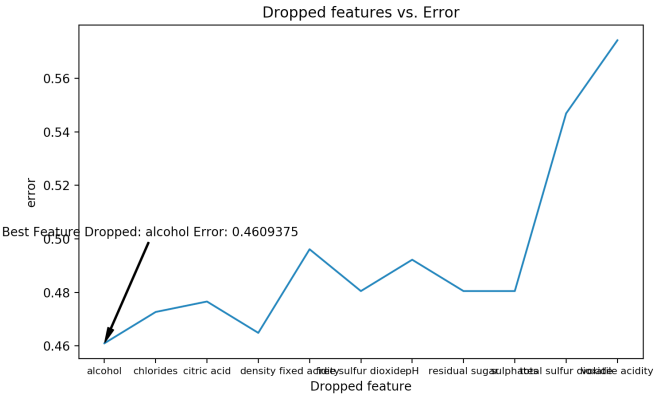

Fig. 7. This graph displays the error associated with the model when a certain feature is dropped. The dropped feature is denoted by the X-axis. The minimum error generated during this trial was by dropped alcohol.

One of the reasons the feature selection results are consistent is because of how complex a neural network is. It all depends on the assigned weights of each input and those vary every training run.

E. Confusion Matrix and More Results

After combining these optimal parameters and dropping a feature, we were able to get up to 51% accuracy. Here is the confusion matrix of our results:

```
[[0,  0,  0,  2,  0,  0]
 [0,  0,  2,  7,  0,  0]
 [0,  0,  77, 55, 0,  0]
 [0,  0,  42, 85, 0,  0]
 [0,  0,  11, 31, 0,  0]
 [0,  0,  2,  6,  0,  0]]
```

Again we can see that most of the errors are produced towards the higher wine quality scores. Now that we have explored two of the main models used for classification, we will move on to some more complex models that will hopefully yield higher accuracy scores for our dataset.

IV. K Nearest Neighbors

To emulate the K Nearest Neighbors model, we used the sklearn.neighbors library. This model gave us an initial accuracy of 51.88% and an optimal accuracy 54.68%. Again, this model produced a low accuracy; however, was able to improve given the right hyper-parameters.

A. N Neighbors

One of the parameters that the K Neighbors model takes in is the number of neighbors that will be used to classify the data. This means that an object is classified by the majority vote of its K neighbors; therefore, the higher the K, the more objects that have a say in classifying an object. By changing the hyper-parameter n_neighbors, which is the number of neighbors used by the k-neighbors queries, we got the k-value vs error plot shown below:

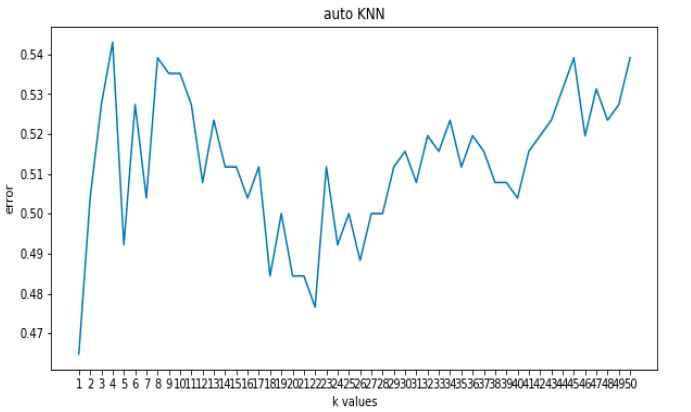

Fig. 8. This graph displays the error associated with each trial given a certain n_neighbors. We can see through the graph that too little and too many neighbors both yield a high error.

Looking at Figure 8, it is easy to see that the error grows as the k-value grows. Here, the k-value grows from 1 to 51. To ensure that error won't have another local minimum as k grows even larger, we tested k-values from 1 to 300 with step size 5. This can be seen by the graph below:

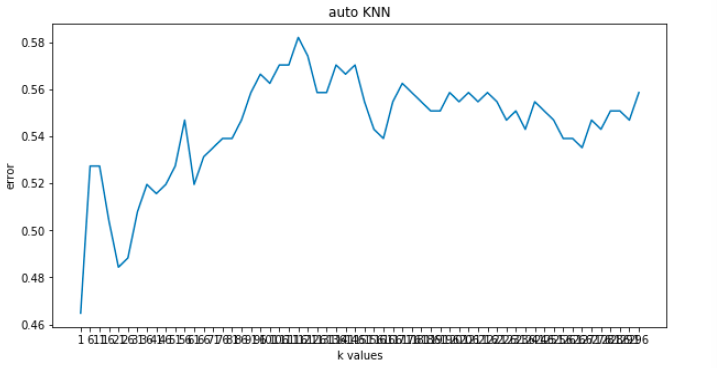

Fig. 9. This graph displays the error associated with each trial given a certain n_neighbors. We can confirm through this graph that the error does not drop even for n_neighbors of size 300.

As shown in the graph, we can conclude that the global minimum is k-value = 1. Because this problem is not too complex, considering more than 1 or 2 neighbors only leads to over complicating the problem and gives us extremely high error.

B. KNN Algorithm

The K-Nearest Neighbors model utilizes a multitude of optimization algorithms which are useful for different types of dataset. To look into this parameter's effect on our dataset, we tried three of these algorithms.

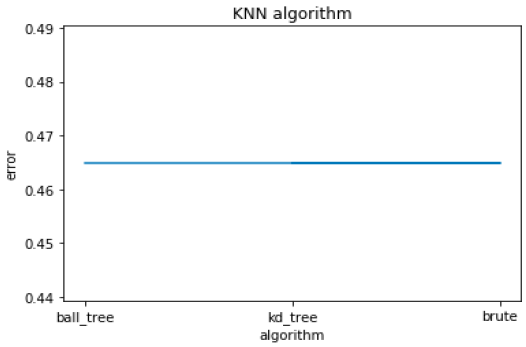

Fig. 10. This graph displays the error associated with each optimization algorithm used for our model. We can see that the error does not change when we use different algorithms.

Looking at Figure 10, we can see that the optimization algorithm had no effect on the error. We tried different k-value combined with different algorithms; however, the error stayed the same.

By picking k-value = 1 and auto picking the algorithm, the best model only gave a 54.68% accuracy, but this is definitely and improvement compared to the default model, which only gives 51.875% accuracy. We concluded that the K-nearest neighbor classier is not that appropriate for the red wine classification problem.

V. Decision Tree

The decision tree model can be used for both classification and regression through essentially creating a series of decision nodes causing a branch. This is useful for a classification problem like ours because a series of yes/no decisions can lead to an accurate prediction. Like all other machine learning models, the decision tree has many parameters that need to be adjusted correctly in order to produce valid output. To emulate this model, we used the DecisionTreeClassifier function in sklearn's tree library. We were able to get 62.1% accuracy with his model, our highest thus far.

A. Maximum Depth

When constructing the decision tree, one of the parameters that determines the complexity of the model is the tree's depth. Like the number of hidden layers in a neural network, adding more layers to a tree (increasing the depth) allows your model to be more complex. By changing the max_depth parameter, which is the maximum depth of the tree, the following effect is shown below:

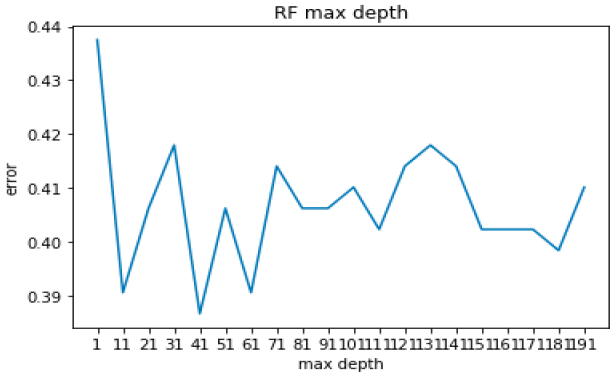

Fig. 11. This graph displays the error associated with each maximum depth of our decision tree model. We can see that the minimum error is produced by using a maximum depth of 41.

As seen in Figure 11, the max depth varies a lot. In the graph, the max depth is tested with values from 1 to 200 with step size 10. The best depth in this case is 41 which is more towards the smaller depth values. This is likely because of the lack of complexity our model has.

B. Minimum Samples Split

Another important parameter of the decision tree model is min_samples_split which is the minimum number of samples required to split an internal node. Unlike the previous parameter, the smaller this hyper-parameter is, the more complex the model becomes. When less samples are required to split a node, the smallest variances of features can lead to the creation of many more decision nodes. The effect this parameter has on our model's prediction error is shown in the graph below:

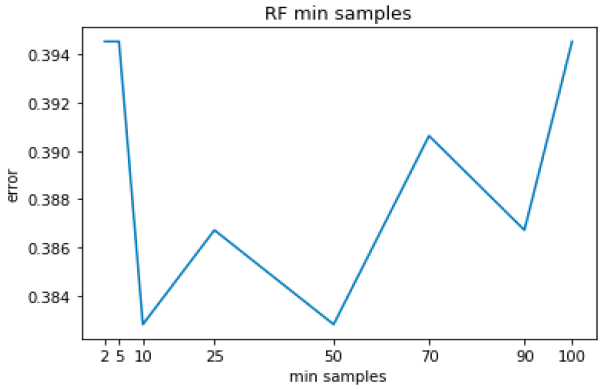

Fig. 12. This graph displays the error associated with each value of the parameter min_samples_split. We can see that the highest error is created at the two ends of the graph, with the minimum error at a value of 10.

We tried some typical minimum values here: 2, 5, 10, 25, 50, 70, 90, 100. Since the numbers that are really large do not give reliable results, they are useless to consider when choosing the lowest error; therefore, the best value is 10. Increasing the number in the beginning consistently helps in dropping the error; however, when the number becomes too large, the effect is not too obvious.

C. Minimum Samples Leaf

Our last hyper-parameter we can adjust for our model is min_samples_leaf, which is the minimum number of samples required to declare a node as a leaf. The smaller this value is, the more leafs the decision tree has. Going into this test, we were not too sure how this would affect our model's accuracy. Because our dataset has been hard to predict, we hypothesized that a smaller "minimum samples" would yield a higher accuracy because this would allow the tree to mold around the intricacies of our dataset. Below is the graph displaying our test:

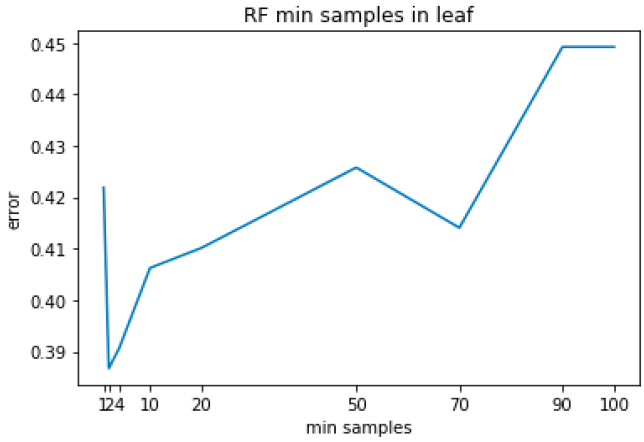

Fig. 13. This graph displays the error associated with each value of the parameter min_samples_leaf. The optimal value is seen in the beginning of the graph as the higher values yield high error.

For this parameter, we picked values: 1, 2, 4, 10, 20, 50, 70, 90, 100. The graph shows that the minimum error value is reached pretty early, validating our initial hypothesis. The exact minimum error is created with a min_samples_leaf of 2. When we go any higher than this value, the error increases strictly and does not ever drop again.

D. More Results

We built three models with respect to the best depth, min_sample_split, and min_sample_leaf. The original default model gave us an accuracy of 57.5%; however, when using our optimal parameter values, we were able to increase this number.

```
0.575
0.584375
0.58125
0.575
```

default → 57.5%
min_sample_leaf = 2 → 58.44%
min_sample_split = 10 → 58.125%
best depth = 41 →

These results show that, we do get some performance improvements from changing these three parameters. For depth, it shows an improvement during validation, but no improvement during test. In our other trails we got the same problem. Sometimes some hyper-parameters can improve the validation performance; however, not necessarily the test performance. Although this was an obstacle in increasing our test performance, at least 2 out of 3 of the modified models did in fact show a better performance on the test data.

E. Feature Selection

Because we were so close to hitting 60% accuracy for this model, we decided to attempt to reach it through another round of feature selection. Below is the feature importance of this model:

```
fixed acidity 0.0825265177344
volatile acidity 0.109803005197
citric acid 0.0995746436533
residual sugar 0.0507289561843
chlorides 0.0985816017378
free sulfur dioxide 0.0550985290622
total sulfur dioxide 0.104740151884
density 0.0867586164844
pH 0.0445837566314
sulphates 0.0971725454056
alcohol 0.170431676026
```

Fig. 14. This is the feature importance of the 11 features from the wine quality dataset, with respect to the decision tree model. The larger the number, the more important the feature is to the model.

We then dropped each feature and plotted our model's error respectively. It was interesting to see that although some features had high importance, dropping them actually increased the model's accuracy (lowered the error). One key example is volatile acidity. This can be visualized below:

```
alcohol
Accuracy Score: 0.555

chlorides
Accuracy Score: 0.617

citric acid
Accuracy Score: 0.598

density
Accuracy Score: 0.605

fixed acidity
Accuracy Score: 0.590

free sulfur dioxide
Accuracy Score: 0.582

pH
Accuracy Score: 0.586

residual sugar
Accuracy Score: 0.602

sulphates
Accuracy Score: 0.570

total sulfur dioxide
Accuracy Score: 0.539

volatile acidity
Accuracy Score: 0.621
```

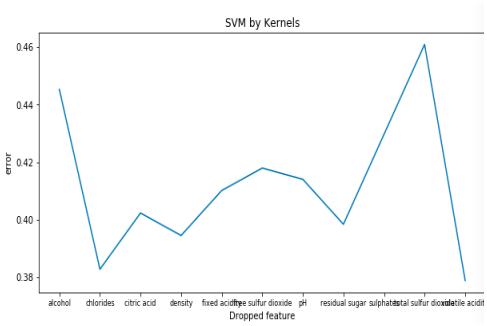

Fig. 15. To the left is the accuracy scores as result of dropping each feature respectively. To the right is that data visualized as a graph. We can see that the two minimum errors are achieved through dropping either chlorides or volatile acidity.

Clearly, after dropping some of the features, we achieved our highest accuracy so far. Through the correct feature selection, we were able to finally break 60% accuracy. By dropping volatile acidity our accuracy hit 62.1%.

VI. Random Forest

Our last model, random forest, is one of the most simple and common machine learning models that is known to produce great results. Like the decision tree, the random forest model can be used for both classification and regression. The random forest algorithm essentially builds many random decision trees and combines them to make a prediction. This model gave us our highest accuracy for this dataset, 70%, and was in fact the best model for solving the red wine quality problem. To emulate this model, we used the RandomForestClassifier function from sklearn's ensemble library.

A. Number of Estimators

One of the parameters of the random forest model is n_estimators, which is the number of trees in the forest. The larger this parameter, the more decision trees the model takes as input. For complex models, it is often useful to have a large n_estimator. Below is a graph showing our error as we alter this parameter:

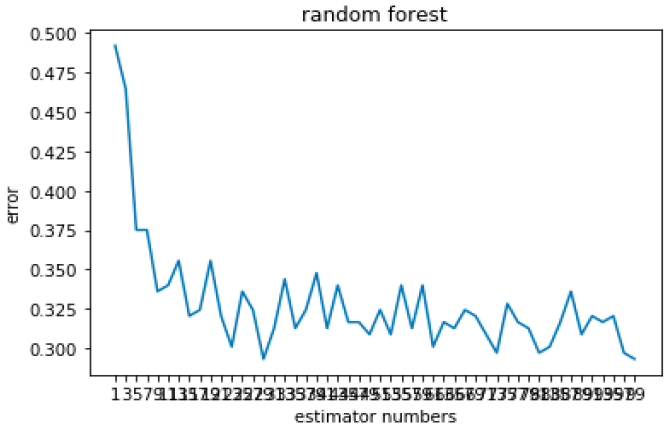

Fig. 16. This graph displays how our model's prediction error changes as we add more trees to our forest. Through analyzing the graph, one can conclude that with a larger forest comes smaller error.

As the number of estimators increase, the prediction error decreases in general, but the best estimator number is not guaranteed to be the largest one. We tested forest sizes from 1 through 100 and a size of 29 gave us the best result. Looking at the predictions of our last few models, it is easy to see why having more decision trees is favorable. Our dataset has a very low variance; thus, making classifying qualities fairly difficult. With more trees in the forest, our model can adjust to this and have more input when making a decision.

B. Maximum Features

When constructing the trees, similarly to the min_samples_split in the decision tree model, the random forest has a parameter, max_features, which dictates how many features to consider when looking for the best split. Below is three of the methods in doing this and there associated errors:

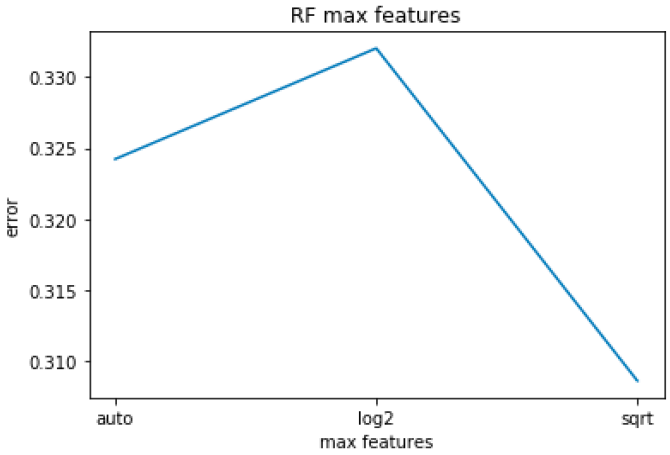

Fig. 17. This graph displays the three different ways of choosing the number of features to consider when looking for the best split. It is obvious to see that sqrt yields the best result.

The best number of features to be considered varies from dataset to dataset. Here, as shown in Figure 17, the square root of the total number of features give the best result. We will use this as our method from here on out.

C. Maximum Depth

Exactly like the max_depth parameter in the decision tree, this parameter defines the depth of each tree in the forest. This is extremely important in deciding the complexity of the model. Below is error associated with each depth:

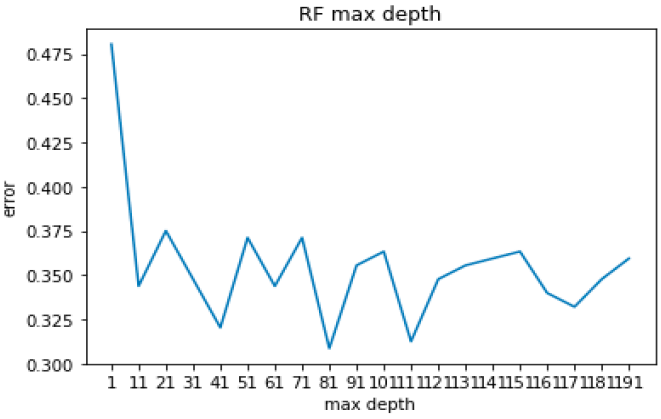

Fig. 18. This graph displays how our model's prediction error changes as we increase the depth of the trees in our forest. Through analyzing the graph, one can conclude that with a larger depth comes a smaller error/higher accuracy.

In general, the model error decreases as the maximum depth increases; however, the largest is not necessary the best. Here, a maximum depth of 81 gives us the best result.

D. Minimum Samples Split

Many of the features of the random forest model are shared with the decision tree model because the forest is essentially a cluster of these decision trees. Another feature that we tested is the minimum samples to split an internal node. By changing the hyper-parameter min_samples_split, the effect is shown below:

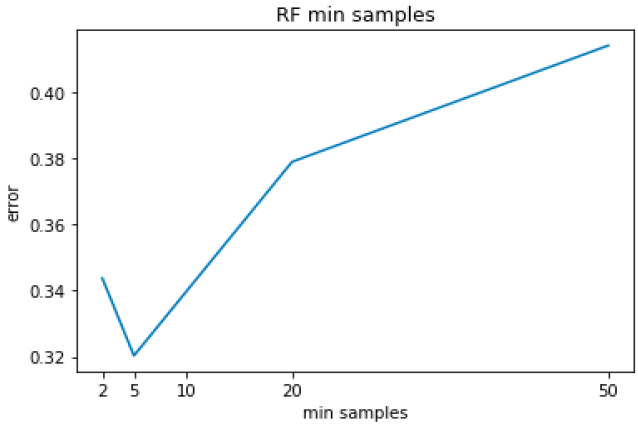

Fig. 19. This graph illustrates how the model's prediction error changes when we increase the minimum number of samples to split an internal node. The minimum error is created with a min samples value of 5.

We only tested typical numbers in this trail. The model error quickly increases when min_samples_split goes ridiculously high; therefore, 5 is the best choice.

E. Minimum Samples Leaf

By changing the hyper-parameter min_samples_leaf, which is the minimum number of samples required to be at a leaf node, the following effect is created:

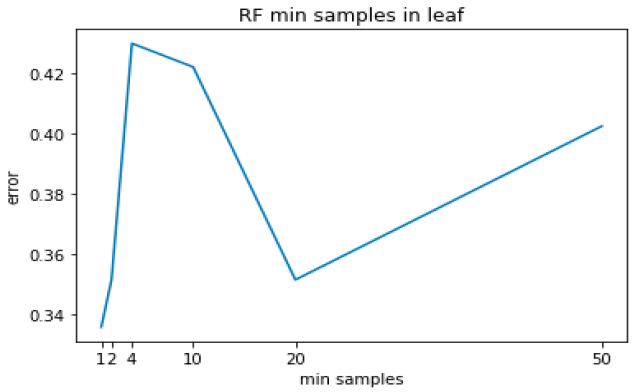

Fig. 20. This graph illustrates how the model's prediction error changes when we increase the minimum number of samples considered when determining whether a node is a leaf node. The minimum error is created with a min samples value of 1.

Figure 20 does not show an obvious trend, but 1 gives us the best result.

F. More Results

We built 7 models each with different hyper-parameters. When changing each hyper-parameter to its optimal value, we saw noticeable increases in the accuracy of the model.

```
0.640625
0.678125
0.678125
0.6875
0.6375
0.65625
```

default → 64%
best n_estimators = 29 → 67.81%
best max_features = sqrt → 67.81%
best max_depth = 81→ 68.75%
best min_sample_split = 1 → 63.75%
best min_sample_leaf = 5 → 65.63%

When adjusting each parameter, it is clear to see that most of the model accuracies jump up about a percent or two. When combining each optimal parameter and running the model once more we achieved an accuracy of 71.31% our highest out of all models and the only time we got remotely close to breaking 70% accuracy.

```
0.703125
```

Two things are worth noting here. First, an optimal parameter on the validation data does not guarantee that the parameter will be optimal for the test data. Second, the combination of all the best parameters does not guarantee the best model. Each feature and its value have trade-offs and it is important to analyze each of them so that they best fit the model.

VII. Conclusion

Through the use of these five different machine learning models, we were able to see the importance of testing parameters. For each model, simply displaying the error associated with each value of a parameter led us to increase out performance by up to 10% which is substantial. This dataset also taught us that every dataset is unique and requires its own models and parameters. Although some models may be the most popular in the industry because they can solve main stream problems efficiently, they may not be the best for a smaller problem like ours. Delving into less common models truly validated the uniqueness of our dataset. Although we were only able to get an accuracy of 70%, which is very low, the improvement from our original accuracy was about 25%.

All in all, the random forest was the best model for solving our classification problem of predicting wine quality scores. Compared to logistic regression, this model improved the overall prediction accuracy by 25% giving us the highest accuracy out of all five models. Through adjusting the hyper-parameters of this model, we were able to improve on our original accuracy of 64% and raise it up 6%.

Appendix

Future Plans

Our future plans for this project are to look into different models to help us get up to a realistic accuracy and apply what we have learned from this project to other personal projects. Each of us have an idea in mind for a project we would like to pursue in the future and going through these five models has placed down a solid base for us to work off of. This project will also be great on our resumes when applying to jobs in the future.