Machine Learning Project

Nicholas Muir 1399185, Michael Vogt 1533057,
Nicholas Baard 1387558, Goolam Bangie 1828201

June 28, 2020

Abstract

This Project serves to apply multiple supervised learning algorithms to a data-set, Such as Naive Bayes, Logistic regression and Decision trees. The data set is a combination of feature of wine that correlate to the quality of the wine.

Contents

1 Data-set

1.1 Description

The Data-set is a combination of both red and white wine data to asses the quality of wines given specific data.

- fixed acidity
- volatile acidity
- citric acid
- residual sugar
- chlorides
- free sulfur dioxide
- total sulfur dioxide
- density
- pH
- sulphates
- alcohol
- quality (score between 0 and 10)

Number of Instances: red wine - 1599; white wine - 4898.
Number of Attributes: eleven + output attribute

We are using the eleven attributes to predict the quality of the wine We initially had it as the quality between 0 and 10,but this posed a classification problem as there where too many possible outcomes and data for the outlier's (0,1,2,8,9,10) as most of the data was centred around (4,5,6). The data points are structure as such :
fixed acidity;volatile acidity;citric acid;residual sugar;chlorides;free sulfur dioxide;total sulfur dioxide;density;pH;sulphates;alcohol;quality;Quality
7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5;2
7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5;2
7.8;0.76;0.04;2.3;0.092;15;54;0.997;3.26;0.65;9.8;5;2
11.2;0.28;0.56;1.9;0.075;17;60;0.998;3.16;0.58;9.8;6;2
7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5;2

1.2 Structure of Inputs and Targets

The feature variables of the dataset are all continuous and therefore, we decided to normalise the data by dividing all feature variables by the largest value of the feature. This caused every variable to take on a continuous value between 0 and 1, taking away the bias between the features.
The way we structured our data was to perform a 60%,20%,20% split into the training, testing and validation datasets respectively. We felt that we could create the most accurate models with this split, as well as effectively be able to tune the models to achieve optimal predictions.
The correlation between features is a unit-less statistical measure that identifies the relationship between two features. It is similar to the covariance between two features, however the correlation between two features is more informative, providing us with the strength of those relationships as well! We applied the idea of the correlation between features to both the white wine and red wine data-sets using the pandas correlation function to calculate all correlation values when comparing all the features with one another, as well as the seaborn library to display a heatmap.

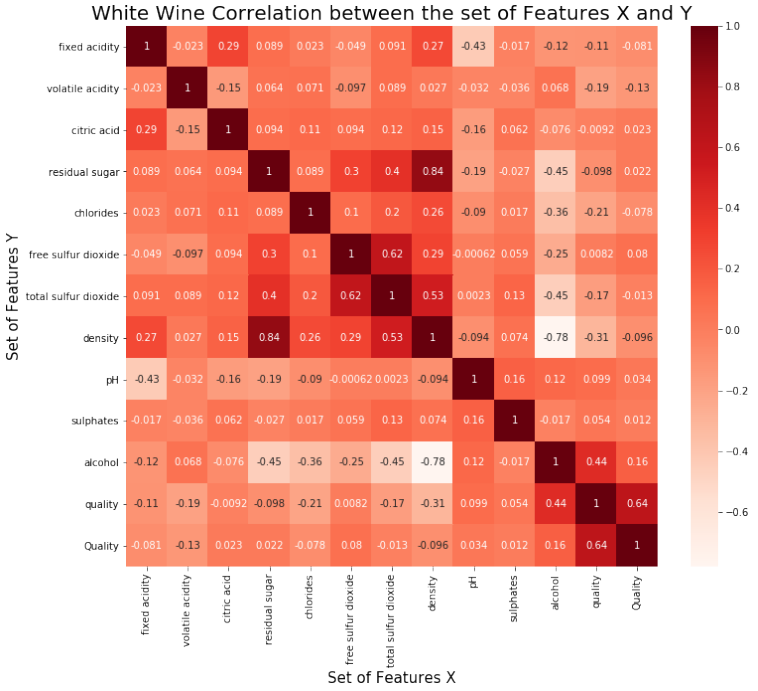

Figure 1: The Correlation Matrix for White Wine

2 Classification Algorithms

- Decision Trees
- Naive Bayes
- Normal Logistic Regression

2.1 Decision Trees

The decision tree algorithm was implemented to determine the quality of both red and white wines through the traversal of the features of the respective wines. The algorithm that was used to create the decision tree was the ID3 Algorithm, choosing the feature with the highest information gain to create a node, creating branches from the node, and traversing the training data through the branches. A leaf node of the tree predicts the most common label that has been passed to it. This process is repeated until all training data points are perfectly classified. Entropy is the measure of uncertainty in a system. This is an important factor of the ID3 Algorithm as the entropy of nodes is used in the calculation of information gain. We want to find the features with the lowest amount of entropy so that we can make more precise predictions. The formula for entropy is:

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

Figure 2: Entropy Formula

Where p is the probability of an event in a feature.
Information gain considers the largest reduction in entropy of a system. This is used to determine which feature will have the maximum gain of information when chosen. The formula is:

$$Gain(T, X) = Entropy(T) - Entropy(T, X)$$

Figure 3: Information Gain Formula

Where the gain of the feature is equal to the entropy of the target class minus the entropy of the feature. This calculation determines which feature is selected for the decision tree.
The features of our dataset each contained continuous variables and thus split points for each feature had to be determined. The split point is one of the parameters that had to be tuned specifically to both the red wine and white wine

trees for optimal results. The maximum and minimum data points of each feature were found, and the amount of increments (split points) were exploratively chosen. The algorithm then splits the data from the feature into left and right nodes and measures the entropy of both sides. The algorithm measures the information gain from that split point. This is done for every split point for the feature to identify the maximum information gain for the feature. This process is done to every feature to compare the maximum information gain and the feature with the highest information gain gets selected as the node. This is all done in the find_feature() function in the tree node class.
Two children nodes are created from the node and all data points would traverse down each one of the branches to one of the nodes. Unlike decision trees with discrete variables, the data does not traverse down a branch if it is equal to the child node's class value but rather traverses down a left branch if the data point is less than the split value and down a right branch if the data point is higher or equal to the split value. Once the feature had been traversed through by all training data points, the feature is then dropped from the list of available features and the process continues until there are no features left. This is completed in the descend_tree() function. At this point, the current node becomes leaf node and the majority class value from all data points that are in the leaf node becomes the class value for that node. Once this process has been completed then the tree has been constructed and is ready to classify testing data points.
With a constructed tree, we should be able to classify a wine into its quality class by using the information about its features. There is a function named infer() that takes a testing data point and traverses through the tree and classifies the data point. The way this function does this is by taking the feature value for the data point and compares it to the split value of the feature. The data point then traverses down a specific branch depending if the feature value is greater or less than the split value of the feature. This happens recursively until the data point reaches a leaf node and then returns the class value of the leaf node.

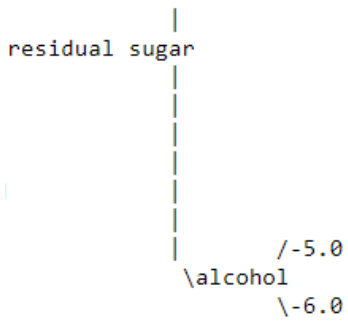

Figure 4: The 'alcohol' node is an example of a lead node

We then compare the class value assigned to the data point to the 'quality'

feature value of the data point. If both values match, then the prediction that we have made using our decision tree has been successful. After our initial creation of the decision trees, our testing accuracy was:

- 41.09% for the red wine dataset
- 51.71% for the white wine dataset

Reasons for the white wine dataset having a higher accuracy could be that there is more data for the white wine dataset (4898 data points opposed to the 1599 data points that the red wine dataset has). Having more data points is useful as the decision tree has more data to train with, and it can build a more precise model. Another reason could be that even though both datasets are types of wines, the features relating to white wines could be more decisive in determining the quality of the wine than the features relating to the red wine.

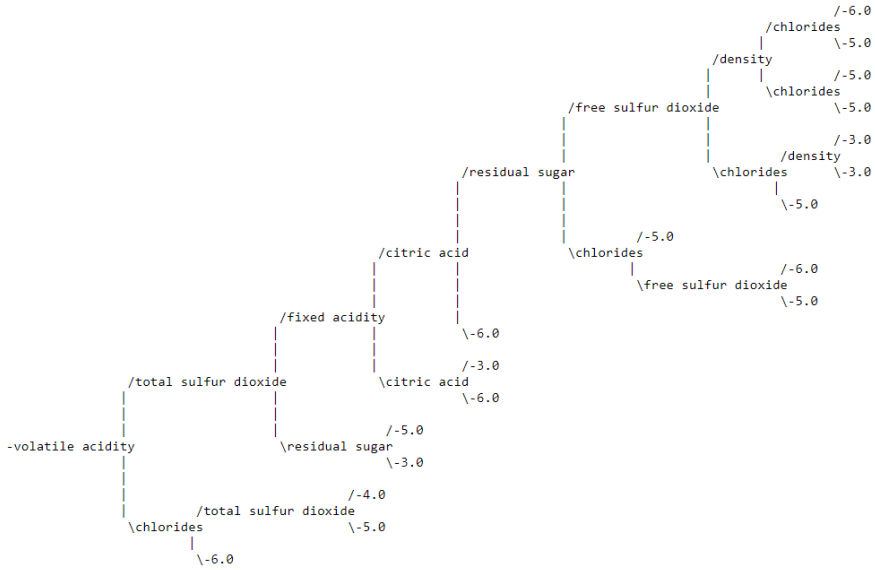

Figure 5: Decision Tree for Red Wine Dataset

2.2 Naive Bayes

The Naive Bayes Algorithm is a Machine Learning method that makes use of conditional probabilities and class independence to quantitatively classify data into certain desired classes, given our prior knowledge. It produces predictions by estimating the probability of the observation belonging to a certain class. Its strengths lie in being an easy to implement and understand algorithm with high prediction accuracy's. It further is useful as it is a widely studied algorithm

with well-understood shortcomings, providing many resources to learn, apply and improve the model.
For our Naive Bayes model, we classify the quality of white wine and red wine given 11 different features to train the model. We first looked at the data, specifically the features, and deduced that they are all numerical values - leading to our choice of using a Gaussian Distribution in our model. Since classification can be considered as a key:value mapping from class values to certain data points, we implemented dictionaries as the primary data structure for storing and manipulating the data. Since a model and data structure have been chosen, the next step was to build and implement the Naive Bayes model to our data. To achieve this a very easy to understand 3-step approach was used, namely:

- Separate the data by class value using dictionaries
- Summarize the data set by class value(Calculate summary statistics like mean and standard deviation to achieve this)
- Determine the Class Probabilities of each class using a Gaussian Distribution

From here, with a strategy in place and with the chosen data structure and distribution, we implement the model. After implementing the first step of our approach - separate data by class value - we found that the class values we were trying to classify for white wine and red wine are:

```
White Wine keys:  dict_keys([6.0, 5.0, 7.0, 8.0, 4.0, 3.0, 9.0])
Red Wine keys:  dict_keys([5.0, 6.0, 7.0, 4.0, 8.0, 3.0])
```

Figure 6: Keys associated with White Wine and Red Wine respectively

This matches our evaluation of the data when conducting the exploratory data analysis. Hereafter, with all the data for white and red wine split according to class, as well as being split into training, validation and testing data, we proceed with the second step of our approach - summarizing the data across each of these new classification groups for our training data. This will help match a distribution to an associated class value, assisting us in the process of classifying our data.
Thus, we move to the final step in our approach - Determining the Class Probabilities of each class using a Gaussian Distribution for our training data. Due to us using real-world data, it fluctuates and can be seen as sporadic. This can make it quite difficult to calculate the probability or determine the likelihood of our given real-world data. To assist in this, the assumption is made that the data must follow some sort of distribution. This assumption is often made locally, and data manipulation might need to take place in order to remove redundant features, outliers, or anything else that might lie in the way of a visible distribution. Often times, for numerical data the best choice is Gaussian, however if your data matches a different distribution, such as Exponential, Naive

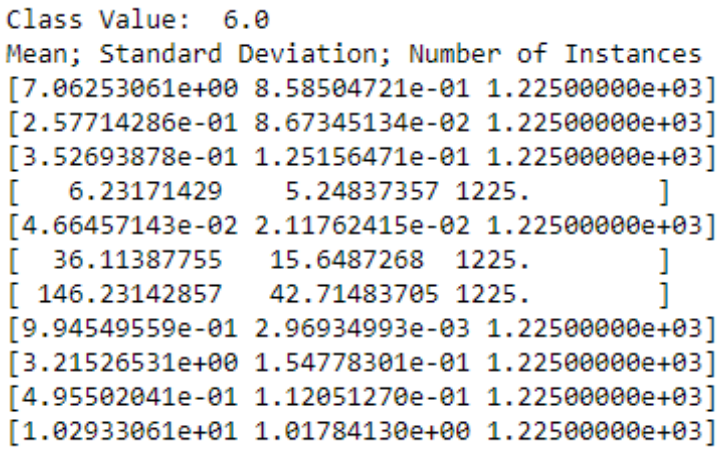

Figure 7: Ex of one class being summarized using summary statistics for White Wine with associated Key Value 6.0

Bayes may be applied to that distribution. As in the general numerical case, our data primarily matched a Gaussian Distribution. A Gaussian Distribution works very well for numerical data, and can thus be summarized using only the mean and standard deviation.
Finally, with everything in place - we can evaluate class probabilities. In this regard, prior to the final Naive Bayes method, a class-probabilities() function was implement where it again uses dictionaries as the primary data structure, mapping the class value to associated class conditional probabilities. This was implemented in the general sense of Naive Bayes, where we first assume class independence in probabilities being calculated separately for each class. Along with this assumption we then calculate the priors, and multiply that with the probabilities produced by the Gaussian Distribution. However, for simplification and implementation purposes we have removed the division for the Naive Bayes Classification as it is redundant in accordance with our end goal. Thus, the output will in fact no longer be strictly the probability of the data belonging to a class, however we still choose the largest value or Maximum a Posteriori. The reason for this implementation is due to the fact that we are more interested in prediction, rather than calculating accurate probabilities. Often, when using real-world data, the conditional probabilities of each class given a feature value may be extremely small. Thereafter, this can result in values too small for python to handle, and this is known as floating point underflow. We do not experience this in our calculations of the class conditional probabilities. This coupled with our analysis of there being no difference in accuracy when using log probabilities led us to removing the implementation of it and rather just using the simplified conditional probability calculation of:
Therefore, taking all of the above into consideration with our training data, we

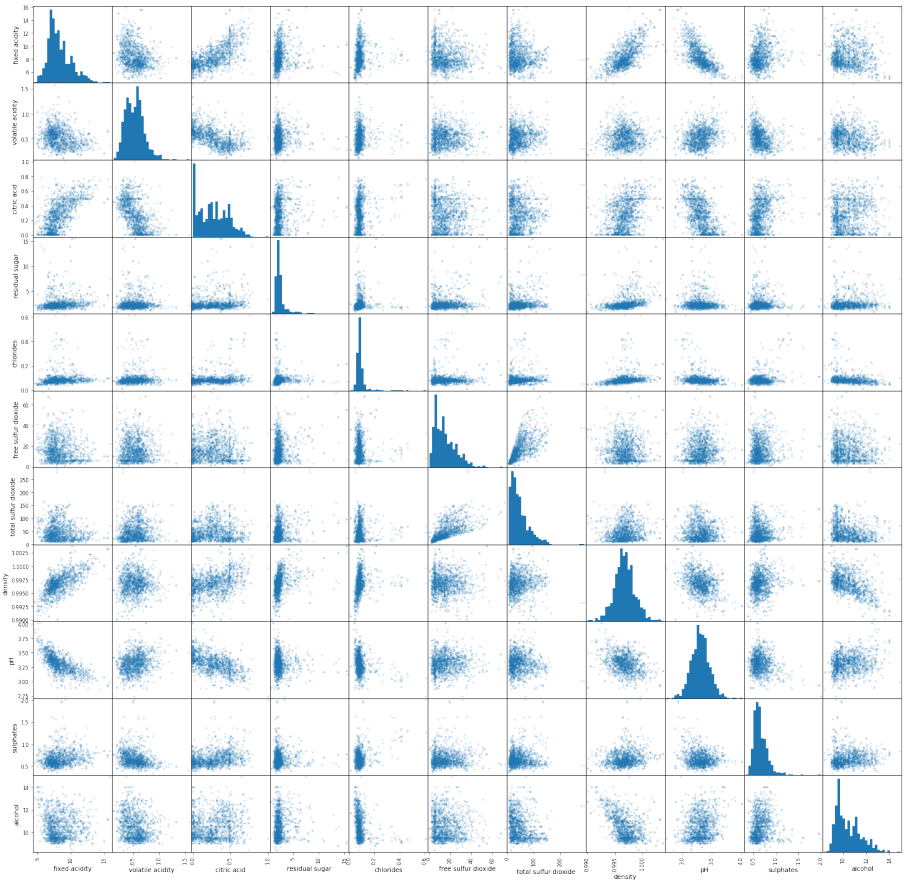

Figure 8: Scatter Plot Matrix for Red Wine

$$P(\text{class}|\text{data}) = P(X|\text{class}) * P(\text{class})$$

Figure 9: Simplified Class Conditional Model used

first use summary statistics to summarize our training data for red and white wine by class. With this and the calculated class conditional probabilities for the training data set, we have trained our Naive Bayes model using the training data.

```
Class Probabilites for White Wine: {6.0: 3.199782431593758, 5.0: 0.1315450541282772, 7.0: 4.433485035372507, 8.0: 2.20412068589
37668, 4.0: 0.02746286685368201, 3.0: 0.00781341047195483, 9.0: 0.1545820241917562}
```

Figure 10: Calculated Class Conditional Probabilities for White Wine

```
Class Probabilites for Red Wine: {5.0: 9.956502185919783e-05, 6.0: 0.0007866659853139375, 7.0: 0.0006713772440339156, 4.0: 2.98
4951338916504e-06, 8.0: 4.7910626164218e-05, 3.0: 1.3827287216920545e-06}
```

Figure 11: Calculated Class Conditional Probabilities for Red Wine

Having successfully trained the Naive Bayes model, the next step would be to use validation data to tune hyper parameters! For Naive Bayes, there are however next-to-none. Naive Bayes is so simple and with its independence assumption already achieves a desirable accuracy for predictions.

2.3 Normal Logistic Regression

Logistic Regression learns the parameters of its respective model with theta values corresponding to each column of the design matrix.(e.g.If we have 11 columns for a data point , together with the bias column we would have 12 and thus we would we have 12 theta values for the design matrix.The key factor in our model is that we had 11 different classes so our task was complicated in that we had to perform a 1 vs all algorithm 11 times to create 11 models with its own respective theta values(parameters). We learn these parameters using gradient descent(Refer to figure 9) together with the heuristic function for each data point in the data-sets. We used a learning rate of 1e-4 as this gave the best result when checked with the validation data which allowed us to learn this hyper parameter. We decided to refrain from using regularisation as our model was not over-fitting as the process of regularisation is to mitigate the effects of over-fitting and preventing it from occurring.The accuracy of our model ranges from 88-92 percent and error would be between 9-12 percent for the red wine data set.The accuracy of our models are great as we took into account not only the diagonal elements but the elements to the left and right of a particular diagonal element due to the relationship between our classes being so similar.The confusion matrices are printed at the end of our data-sets.Due to the multi-class nature of our data-sets, we use the softmax function which calculates the probability of a data point belonging to a particular class and thus if we use the 11 different parameter sets together with a data point we will gather the probability of a data point belonging to all the classes and we can choice the maximum value from there to predict the class of a data point.

- We chose normal regularisation as its a great algorithm when trying to classify data points.Gradient descent is also super efficient and also allows

us to choose how fast our model learns.It is used efficiently when our data is linearly separable.The algorithm is also very easy to implement.One last advantage is that it can be used for multi class classification as well as one vs one classification.As discussed above we didn't add regularisation as our model does not over-fit.

- Of course with logistic regression we used the sigmoid function in our heuristic function as it is the best non-linear basis function to use when trying to differentiate between two classes as if the value of a data point after applying the sigmoid function to it is a value between 0 and 1, if the value is above 0.5 we can state that the data point is associated to class and if the value is below 0.5 we can safely assume that this data point corresponds to class 0. This function allows us to draw a decision boundary of model by applying the dot product between the parameters(theta values) and the features and equating it to zero.

- Our alpha that we used was 1e-4 as we saw that with this value our algorithm runs most efficiently , we made it a parameter to our gradient descent function so it can be tested relatively easily by changing the parameter passed to the gradient descent function.If alpha exceeded this value it would sometimes overshoot the minimum and diverge.If the alpha is lower than this it would converge very slowly.

Gradient Descent

Remember that the general form of gradient descent is:

$$\begin{aligned} &Repeat\ \{ \\ &\quad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \\ &\} \end{aligned}$$

We can work out the derivative part using calculus to get:

$$\begin{aligned} &Repeat\ \{ \\ &\quad \theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \\ &\} \end{aligned}$$

Figure 12: Gradient Decent Formulae

3 Results

3.1 Decision Trees

After our initial creation of the decision trees, our testing accuracy was:

- 41.09% for the red wine dataset
- 51.71% for the white wine dataset

To attempt to improve the model, the hyperparameters of the decision tree needed to be tuned. A hyperparameter is a parameter of the decision tree that can be controlled or changed to affect the learning process of the algorithm. These hyperparameters for the decision tree need to be explored in order to better the accuracy of the predictions. The first hyperparameter that was tested was the maximum depth of the tree. The way to test this is to remove the features from the bottom of the tree because the model might be overfitting to the data, this is also referred to as 'pruning'. Various combinations of the features were left out of the creation of the trees and in both the red wine tree as well as the white wine tree, we observed a drop in accuracy. After pruning, the accuracies were:

- Between 27-39% for the red wine dataset
- Between 46-49% for the white wine dataset

We thus decided that the original trees were to remain as they had the highest prediction accuracy.
The second hyperparameter explored was the use of gini index vs the use of entropy as the criterion for calculating information gain. They are both measures of impurity and effectively calculate the same measure of uncertainty; both values are used the same way in calculating information gain. Entropy is only slightly more computationally expensive than gini index since entropy has logarithmic functions. The fact that choosing gini index over entropy would not increase the accuracy of predictions, we chose to remain with entropy as the calculation of uncertainty for our model.

$$Gini: Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$$
$$Entropy: H(E) = -\sum_{j=1}^{c} p_j \log p_j$$

Figure 13: Gini Index and Entropy Formulae

The last hyperparameter that was investigated was the maximum features used when identifying information gain. This method states that a random percentage of features are chosen to calculate the maximum information gain for node selection. Due to the fact that at most there are ten features, it would be unhelpful to use a random small percentage of the features as we would not be

using the features with the highest information gain. We thus decided not to use this in the creation of our model. Since the hyperparameters had been explored and the accuracy of predictions was still not high enough, we re-investigated the data. We had noticed that the quality of wine had been ranked numerically between 0 and 10, and this could be one of the biggest reasons why the accuracy is low. It is a lot more difficult to distinguish the difference between a wine of quality 7 and a wine of quality 6 than it is to distinguish the difference between a 'good' and a 'bad' wine. We then decided to edit our model to classify a prediction as correct if the prediction was the same as the quality class of the data point or if it was the class above or below the quality class. For example, if a data point has a quality class value of 6, the prediction would be considered accurate if the predicted class is 5, 6, or 7. This method extended the 'radius' of classification where instead of classifying wine in a numeric value from 0 to 10, we classify the wines into larger groups where there is more distinction between them.
With this implemented, the final testing accuracy of predictions was:

- 92.7% for the white wine dataset
- 73.6% for the red wine dataset

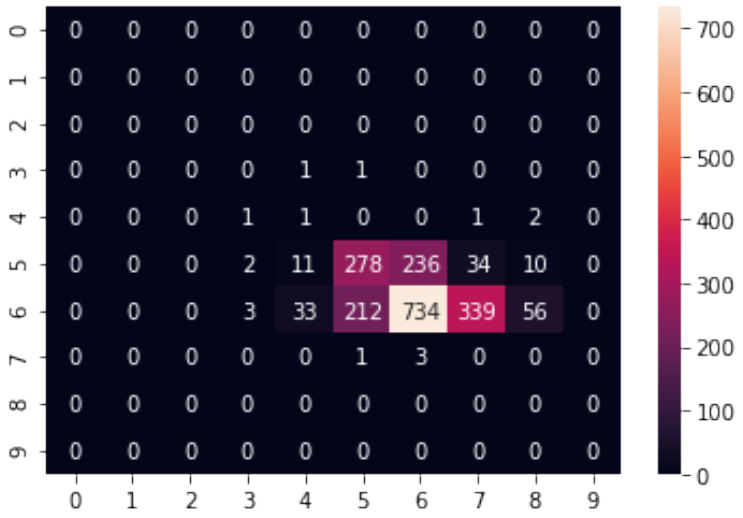

Figure 14: Confusion Matrix for the White Wine dataset

In conclusion, we were satisfied by the final accuracy of predictions. We understand that the difference between the accuracy of the white wine prediction and the red wine prediction can be explained by the white wine dataset having more data to train the model on and therefore creating a more accurate decision tree. The difference can also be due to the features of white wine being more

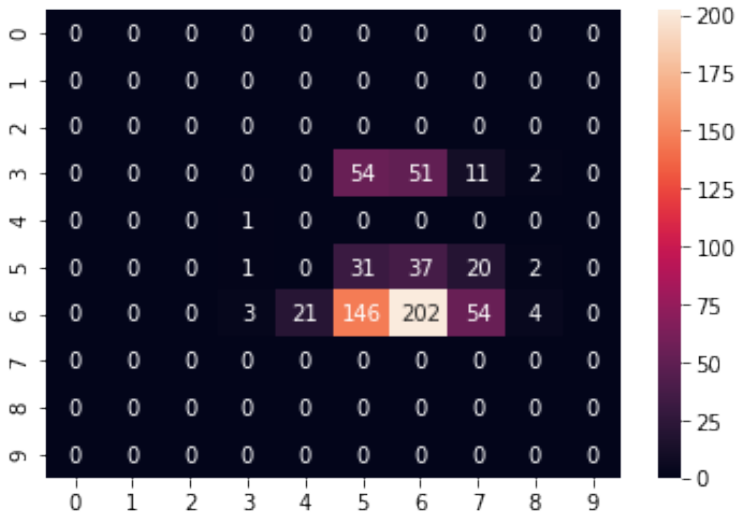

Figure 15: Confusion Matrix for the Red Wine dataset

deterministic in the quality of the wine than red wine because they are different types of wine and both have many sub-types.

3.2 Naive Bayes

With the training of our model complete, and no chosen hyper-parameters to tune, we proceeded with testing the model. So, having built the model using the training data and all available features, we ran our testing data through our model to see the types of predictions and the possible accuracies it might return. Our testing data has also been stored using dictionaries. So, for each data point that we access in our testing data, along with the summary statistics for our training data , will be passed to our function for calculating the class conditional probabilities associated with with this data point. We then iteratively look for the best class value and associated probability, while also appending the "best" chosen class values to our prediction list. So, using all the features and predicting class values in the range 0-10, based off the training of our model, we first compared these predictions to our quality label chosen for prediction from our CSV.

Initial results were as expected, achieving accuracies of 54% for red wine and 38% for white wine in data sets the size of 1599 and 4898 respectively. Thus with these initial accuracies, we went back to our data to see how we might improve them.

Looking at our correlation matrices for white and red wine, we can see that there are a number of correlated features, making them redundant and possibly affecting the accuracies. To see whether this was the case, we attempted

removing groups of features that were closely related(i.e. values very close to 1 on the correlation matrix). For red wine, we achieved the highest accuracy of 60% , when removing the correlated features 'citric acid', 'fixed acidity', and 'density'. For white wine, we achieved the highest accuracy of 47% , when removing the correlated features 'density', 'residual sugar', 'free sulfur dioxide', and 'total sulfur dioxide'.

```
Red Wine
Confusion Matrix :
[[ 0  2  3  0  0  0]
 [ 0  2  5  4  0  0]
 [ 0  3 97 38  5  0]
 [ 0  8 37 66 22  3]
 [ 0  0  3  8  9  2]
 [ 0  0  0  1  2  0]]
Accuracy Score : 0.54375
```

Figure 16: Red Wine accuracy before the removal of redundant features

```
Red Wine
Confusion Matrix :
[[ 1  1  3  0  0  0]
 [ 0  1  7  3  0  0]
 [ 0  3 95 43  2  0]
 [ 0  4 29 84 17  2]
 [ 0  0  2  8 12  0]
 [ 0  0  0  1  2  0]]
Accuracy Score : 0.603125
```

Figure 17: Red Wine accuracy after the removal of redundant features

```
White Wine
Confusion Matrix :
[[  1   0   0   0   0   0   0]
 [  0   8   8   3   5   1   0]
 [  4   6 139  82  35   0   0]
 [  1   7 102 122 277   8   0]
 [  0   0  18  15 110   7   1]
 [  0   0   8   1  10   0   1]
 [  0   0   0   0   0   0   0]]
Accuracy Score : 0.3877551020408163
```

Figure 18: White Wine accuracy before the removal of redundant features

```
White Wine
Confusion Matrix :
[[  0   0   0   1   0   0]
 [  0   5   7  11   2   0]
 [  5   4 132 101  24   0]
 [  2   5  79 224 199   8]
 [  0   2  15  36  93   5]
 [  0   0   8   1   8   3]]
Accuracy Score : 0.4663265306122449
```

Figure 19: White Wine accuracy after the removal of redundant features

Another method of achieving improved accuracies that we implemented, was to decrease the number of classes it has to predict, while still training it on the class values provided by the quality label
So, to attempt our idea, we thought about how, in today's society, many people are not wine connoisseurs and do not require the specific wine qualities ranging from 3.0-9.0. This is because, if someone sees an 8.0, they will believe it to be exceptional. The same is said for 9.0, so this difference by one unit means nothing to the ordinary person in society. For this reason, we chose to split our range of class values into three main groups - Exceptional; Average; Poor. However to implement this grading we had to choose numerical values to represent these:

- The class value of '1' was chosen to represent the grouping of our initial class values that are less than 4, relating to 'Poor'.
- The class value of '2' was chosen to represent the grouping of our initial class values in the range of 4 and 7, relating to 'Average'.
- The class value of '3' was chose to represent the grouping of our initial class values that were strictly greater than 7, relating to 'Exceptional'.

This was completed and stored in our CSV as a new class label feature, called 'Quality'. Thus, we used the same trained model, but converted the predictions to mach our new class label using simple if-statements and achieved greatly improved accuracies of 92.19% for Red Wine and 92.96% for white wine.

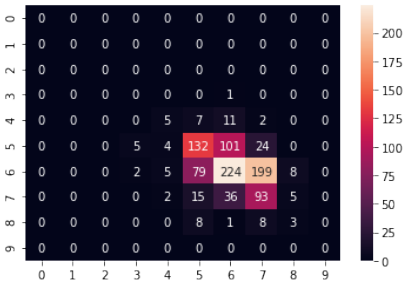

Figure 20: White Wine Heat Map for improved accuracy of 46%

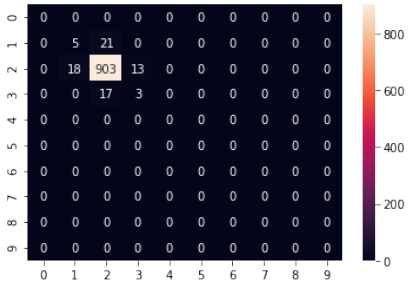

Figure 21: White Wine Heat Map for changed classes and largely improved accuracy of 92.96%

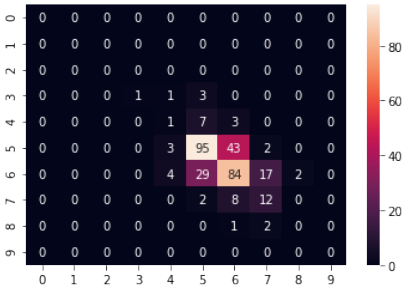

Figure 22: Red Wine Heat Map for improved accuracy of 60%

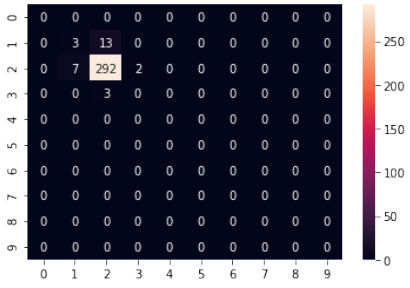

Figure 23: Red Wine Heat Map for changed classes and largely improved accuracy of 92.19%

This is very interesting! When training our model on the initial class values, and then also predicting using the same range of class values, the red wine had higher accuracies with a much smaller data set when compared to the white wine with a much larger data set. The opposite is however true, when we still

train our model on the initial class values, but decrease the prediction range of class values. We believe this is due to the the similarity between features, often causing predictions which are slightly incorrect. Thus, the more data you test on, the more you will get incorrect which was proven to be the case with white wine only achieving 46% , while red wine achieved 60% . Then with the decrease in class values, these slight errors weren't a problem any more, and now we have the case where the more data results in better predictions which is seen with white wine now achieving 92.96% , while red wine achieved 92.19% .

3.3 Normal Logistic Regression

Performance of the red wine data set produced a range of accuracy from 88-92 percent on the test data , this is the best possible performance of our algorithm. What worked best was using a learning rate of 1e-4 as this was the right value which allowed for gradient descent to converge , increasing our learning rate resulted in divergence and overshooting the minimum. Adding regularisation was also not done due to our training data not overfitting but also because using it in the algorithm caused errors in our confusion matrix as the model was just predicting one class on the test data.We tried many different approaches to our dataset, one included encoding our classes into three classes but this resulted in a decent but not great accuracy and also prevented us from finding out which classes were being confused with one another.The best performance was when we used our original data and when we produced our confusion matrices , we computed the accuracy by adding the diagonal elements of the confusion matrix as well as the elements to the left and right of a particular diagonal element,meaning if we were at the (5,5) position in our confusion matrix we would add this to our counter as well as the elements at (5,6) and (5,4) resulting in a great accuracy. If an interested party would like to use this dataset to test various algorithms , some interesting outcomes such as:

- Try to gather more data for the classes that don't appear as frequently as the other classes, such as classes for quality of 0.
- Work out the features that are the most important to classify a data point using the covariances between the features and working out eigen values and eigen vectors using PCA.Thus reducing the dimensions of our dataset as well as allowing us to understand the structure of the data.
- Use EDA to build a picture of what is going on with the dataset and "play" with the dataset by running small models on the dataset to understand the dataset better as well as maybe running a clustering algorithm on the dataset to view whether the clusters coincide with the classes of the data points in the dataset.

The accuracy of normal logistic regression with our data-set being heavily based on that of each individual quality number (being 0 to 10),The Accuracy will suffer since there are many possible outcomes.Thus for our case we will accept

a prediction within a bounds of the correct quality. Below are the Validation and Test confusion matrices for both Red and White wine.

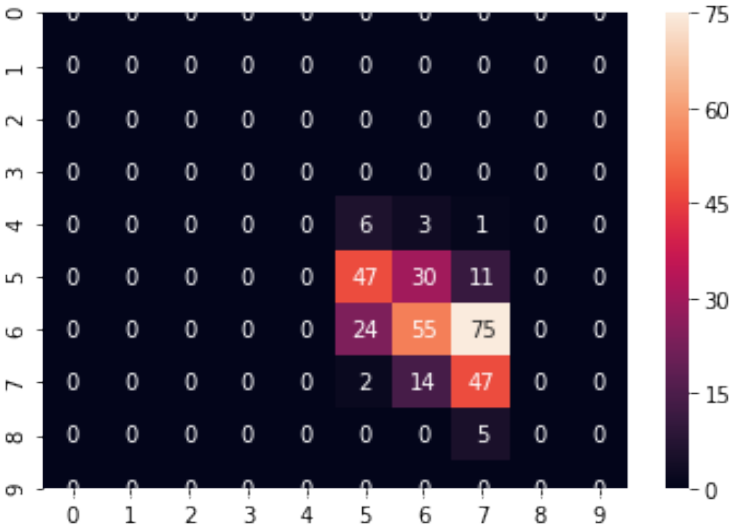

Figure 24: Validation Data for Red Wine

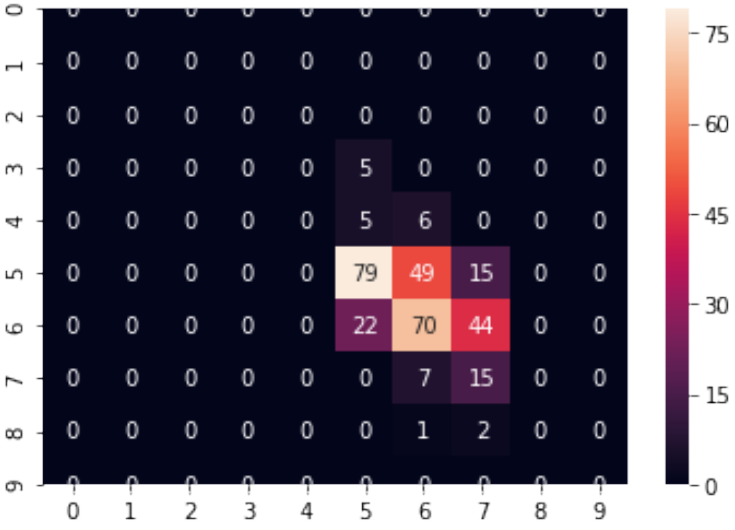

Figure 25: Test Data for Red Wine

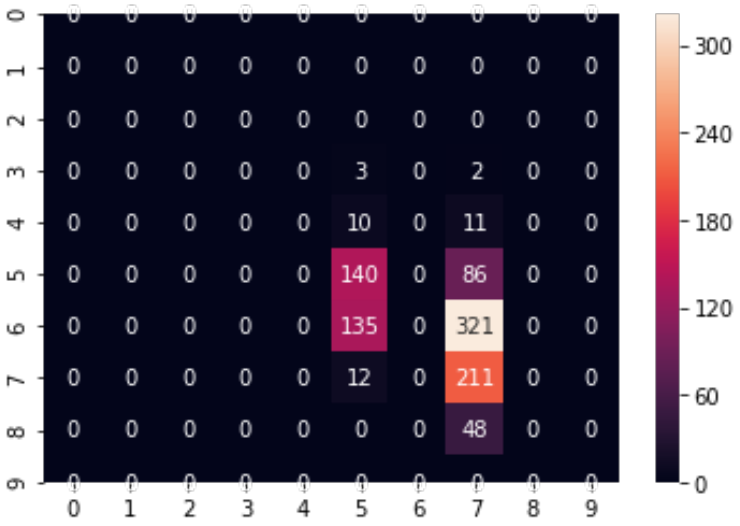

Figure 26: Validation Data for White Wine

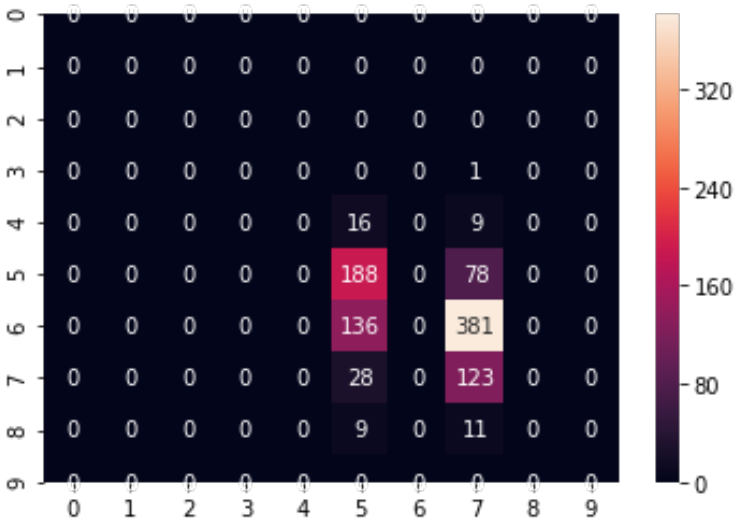

Figure 27: Test Data for White Wine

4 Citation of data

P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553. ISSN: 0167-9236.
Available at: [@Elsevier] http://dx.doi.org/10.1016/j.dss.2009.05.016 [Pre-press (pdf)] http://www3.dsi.uminho.pt/pcortez/winequality09.pdf [bib] http://www3.dsi.uminho.pt/pcortez/dss09.bib

A Code

A.1 Decision Trees

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
from ete3 import Tree, TreeStyle, TextFace, add_face_to_node
import pandas as pd
import seaborn as sn


# In[2]:


df_white = pd.read_csv('winequality-white.csv', ";")
df_red = pd.read_csv('winequality-red.csv', ";")


# In[3]:


print(np.asarray(df_white))
full_data_white = np.asarray(df_white)
full_data_red = np.asarray(df_red)
feature_names = df_white.columns
f_names = np.asarray(feature_names)
print(f_names)


# In[4]:


y_values_white = full_data_white[:,-2:]
data_white = full_data_white[:,:11]
```

```
y_values_red = full_data_red[:,-2:]
data_red = full_data_red[:,:11]
f_names = f_names[:11]

# Normalisation of the datapoints
data_norm_white = data_white
data_norm_red = data_red
for i in range(data_norm_white.shape[1]):
    data_norm_white[:,i] = data_norm_white[:,i]/np.max(data_norm_white[:,
        i])

for i in range(data_norm_red.shape[1]):
    data_norm_red[:,i] = data_norm_red[:,i]/np.max(data_norm_red[:,i])

print("Feature Names:")
print(f_names)
print('\n')
print("Target Columns (White wine):") #First column is before pre-
    proccessing
print(y_values_white)
print('\n')
print("Data Normalized (White wine):")
print(data_norm_white)
print('\n')

# Class problem refers to the column number of the target values

class_problem = 0

# Training / Test split for white wine data
no_of_datapoints_white = data_white.shape[0]
no_of_y_values_white = y_values_white.shape[0]

training_data_white = data_white[:round(no_of_datapoints_white*.6),:]
testing_data_white = data_white[-round(no_of_datapoints_white*.4):,:]

training_y_values_white = y_values_white[:round(no_of_y_values_white*.6)
    ,:]
testing_y_values_white = y_values_white[-round(no_of_y_values_white*.4)
    :,:]

no_of_test_datapoints_white = testing_data_white.shape[0]
no_of_test_y_values_white = testing_y_values_white.shape[0]

# Training / Test split for red wine data
no_of_datapoints_red = data_red.shape[0]
no_of_y_values_red = y_values_red.shape[0]

training_data_red = data_red[:round(no_of_datapoints_red*.6),:]
```

```
testing_data_red = data_red[-round(no_of_datapoints_red*.4):,:]

training_y_values_red = y_values_red[:round(no_of_y_values_red*.6),:]
testing_y_values_red = y_values_red[-round(no_of_y_values_red*.4):,:]

#print(validation_data_white)
print(.2*data_white.shape[0])


# In[5]:


def calc_entropy(data):
    unique, counts = np.unique(data, return_counts = True)
    for (u, c) in zip(unique, counts):
        continue
        #print("Node has {} elements of Class {}".format(c, u))
    count_table = np.array([unique,counts]).T
    prob = np.array(count_table[:,1]/len(data))
    data_entropy = -sum(prob*np.log2(prob))

    entropy = np.sum(
        [
            (-counts[i] / np.sum(counts)) * np.log2(counts[i] / np.sum(
                counts))
            for i in range(len(unique))
        ]
    )
    return([entropy, unique])

def info_gain(data, left, right, node_entropy, split_values):
        total = data.shape[0]
        entropy_left = calc_entropy(left)[0]
        entropy_right = calc_entropy(right)[0]
        weighted_entropy = ((int(left.shape[0]) / total) * entropy_left)
            + (
            (right.shape[0] / total) * entropy_right
        )
        info_gain = node_entropy - weighted_entropy
        print("Info gain at split {} is {} \n".format(split_values,
            info_gain))
        return info_gain



# In[6]:


class tree_node:
    def __init__(self, data, labels, diagram_node, available_features,
```

```
     splits):
    self.data = data
    self.labels = labels
    self.diagram_node = diagram_node
    self.available_features = available_features
    self.feature_index = None
    self.feature_name = None
    self.node_values = None
    self.is_leaf = False
    self.class_value = None
    self.children = None
    self.split_val = None
    self.node_entropy = calc_entropy(labels[:,class_problem])[0]
    self.split_array = np.zeros(self.data.shape[1])
    self.splits = splits

    if not (self.node_entropy == 0.0 or self.node_entropy == -0.0 or
         self.data.shape[1] == 0):
        self.children = np.array([])
        self.split_val = None
        self.node_values = None
        self.find_feature()
        self.descend_tree()
    elif self.data.shape[1] == 0:
        self.is_leaf = True
        unique, counts = np.unique(labels[:,class_problem],
            return_counts = True)
        majority_class = np.argmax(counts)
        self.class_value = unique[majority_class]
        #print('labels: {}'.format(self.labels[:,:]))
        #print("CLASS VALUE {}".format(self.class_value))
        self.feature_name = str(self.class_value)
        self.diagram_node.name = self.feature_name
    else:
        self.is_leaf = True
        #print('labels: {}'.format(self.labels))
        self.class_value = labels[0,class_problem]
        self.feature_name = str(labels[0,class_problem])
        self.diagram_node.name = self.feature_name



# To pick the best feature with highest information gain
def find_feature(self):
    print("Finding feature for new node")
    feature_entropies = np.zeros(self.data.shape[1])
    max_split_array = ([])
    feature_info_array = ([])
    for i in range(self.data.shape[1]): # For each feature
        print("Working on feature: ", i)
```

```
    d = self.labels[:,class_problem]
    feature_entropies[i], uniqueValues = calc_entropy(d)
    feature_sub_classes = np.unique(self.data)
    #print('feature subclasses: {}'.format(feature_sub_classes))

    # choosing size of increment changes amount of split points
    increment = (np.max(self.data[:,i])-np.min(self.data[:,i]))/
        self.splits
    print("increment : {}".format(increment))

    #print(self.data[:,:])
    print('max: {}'.format(np.max(self.data[:,i])))
    print('min: {}'.format(np.min(self.data[:,i])))
    #print('increment: {}'.format(increment))
    if increment == 0:
        split_values = ([])
    else:
        split_values = np.arange(np.min(self.data[:,i]+increment),
            np.max(self.data[:,i]-increment),increment)
    #print('split values array: {}'.format(split_values))
    info_gains_array = ([])

    for j in split_values:
        left_node_data = np.where(self.data[:,i]<j)
        right_node_data = np.where(self.data[:,i]>=j)
        left_node_data = self.labels[left_node_data,class_problem]
        right_node_data = self.labels[right_node_data,
            class_problem]

        info_gains_array.append(info_gain(self.data,left_node_data
            ,right_node_data,feature_entropies[i],j))
        #print(info_gains_array)

    if increment == 0:
        feature_split_value = 0
        feature_info_array = ([])
    else:
        feature_split_value = split_values[np.argmax(
            info_gains_array)]
        feature_info_array.append(np.argmax(info_gains_array))
    max_split_array.append(feature_split_value)
    print('feature_split_value')
    print(feature_split_value)



#print(feature_info_array)
print('index of feature with max info gain:')
#print(max_split_array[0][:])
max_index = np.argmax(feature_info_array)
```

```
    print(max_index)
    #print(f_names)
    #print(f_names[max_index])
    #print('val of split_value max info gain:')
    #print(feature_info_array[max_index])
    self.split_val = max_split_array[max_index]
    #self.split_array = ([])
    chosen_feature = max_index
    self.feature_index = chosen_feature
    self.feature_name = self.available_features[self.feature_index]
    self.diagram_node.name = self.feature_name
    print("Found feature: ", self.feature_name)
    print("Found feature index: ", self.feature_index)
    print('split val for feature = ', self.split_val)
    #split_array[max_index] = self.split_val
    self.split_array[max_index] = self.split_val
    print('split array', self.split_array )

# This function is used to add nodes to the tree once the best
    feature to split the data is found
def descend_tree(self):
    print("Descending tree with node entropy value: ", self.
        node_entropy)
    data_for_feature_value_greater = np.where(self.data[:,self.
        feature_index] >= self.split_val)
    data_for_feature_value_lower = np.where(self.data[:,self.
        feature_index] < self.split_val)
    #print('data_for_feature_value_greater')
    #print(data_for_feature_value_greater)
    #print('data_for_feature_value_lower')
    #print(data_for_feature_value_lower)
    remaining_features = np.arange(self.data.shape[1])!=self.
        feature_index
    new_child_diagram_node = self.diagram_node.add_child(name="Temp")
    print('self.split_val {}'.format(self.split_val))

    self.node_values = self.split_val
    print("self.node_values {}".format(self.node_values))

    self.children = np.append(self.children,
        tree_node(self.data[data_for_feature_value_lower][:,
            remaining_features],self.labels[
            data_for_feature_value_lower],
        new_child_diagram_node, self.available_features[
            remaining_features],self.splits))

    remaining_features = np.arange(self.data.shape[1])!=self.
        feature_index
    new_child_diagram_node = self.diagram_node.add_child(name="Temp")
```

```
        self.children = np.append(self.children,
            tree_node(self.data[data_for_feature_value_greater][:,
                remaining_features],self.labels[
                data_for_feature_value_greater],
            new_child_diagram_node, self.available_features[
                remaining_features],self.splits))
        #print("Children:")
        #print(self.children)

    # This function infers (predicts) the class of a new/unseen data
        point. We call this on the test data points
    def infer(self, data_point):
        sv = self.split_val
        if not self.is_leaf:
            print(data_point)
            print(self.feature_index)
            print(self.feature_name)
            print(data_point[self.feature_index])
            if sv > data_point[self.feature_index]:
                print('data point less than split value')
                print('sv: ',sv)
                child = self.children[0]
                sv = child.split_val
                print('child sv less: ',sv)
                #print(type(self.children[0]))
                allocated_class = self.children[0].infer(data_point[np.
                    arange(data_point.shape[0])!=self.feature_index])
                #print('allocated class1:',allocated_class)
                return(allocated_class)
            else:
                print('data point more than split value')
                print('sv: ',sv)
                child = self.children[1]
                sv = child.split_val
                print('child sv more: ',sv)
                allocated_class = self.children[1].infer(data_point[np.
                    arange(data_point.shape[0])!=self.feature_index])
                #print('allocated class2:',allocated_class)
                return(allocated_class)
            print("Error found new value, can't classify")
        else:
            print("Classified data point as: ", self.class_value)
            return(self.class_value)



# In[7]:


# Testing hyperparameters - max tree depth
```

```
t = Tree()
splits = 6
tree_white = tree_node(training_data_white[:,[0,1,2,3,4,5,6,7]],
    training_y_values_white, t, f_names[[0,1,2,3,4,5,6,7]], splits)
print(t.get_ascii(show_internal=True))

ts = TreeStyle()
ts.show_leaf_name = False
def my_layout(node):
    F = TextFace(node.name, tight_text=True)
    F.rotable = True
    F.border.width = 0
    F.margin_top = 2
    F.margin_bottom = 2
    F.margin_left = 2
    F.margin_right = 2
    add_face_to_node(F, node, column=0, position="branch-right")
ts.layout_fn = my_layout
ts.mode = 'r'
ts.arc_start = 270
ts.arc_span = 185
ts.draw_guiding_lines = True
ts.scale = 100
ts.branch_vertical_margin = 100
ts.min_leaf_separation = 100
ts.show_scale = True


# In[8]:


count_correct = 0
conf_matrix_white = np.zeros(100).reshape((10,10))

for i in range(testing_data_white.shape[0]):
    out = tree_white.infer(testing_data_white[i])
    x = int(out)
    y = int(testing_y_values_white[i,class_problem])
    conf_matrix_white[x,y] = conf_matrix_white[x,y]+1
    is_correct = out == testing_y_values_white[i,class_problem]
    print('Actual data class: {}'.format(testing_y_values_white[i,
        class_problem]))
    if is_correct:
        print('CLASSIFIED CORRECTLY')
        count_correct = count_correct + 1
    else:
        print('CLASSIFIED INCORRECTLY')

print('Amount of guesses correct: {}'.format(count_correct))
```

```
print('Amount of total guesses: {}'.format(testing_y_values_white.shape
    [0]))
print("Final Testing Accuracy: ", count_correct/testing_y_values_white.
    shape[0])


# In[9]:


#Initial confusion matrix for white wine

ax = sn.heatmap(conf_matrix_white, annot=True, fmt = 'g')
print("Final Testing Accuracy: ", count_correct/testing_y_values_white.
    shape[0])


# In[11]:


# adjusted target classes

count_correct = 0
conf_matrix_white_new = np.zeros(100).reshape((10,10))
#print(conf_matrix_white)

for i in range(testing_data_white.shape[0]):
    out = tree_white.infer(testing_data_white[i])
    x = int(out)
    y = int(testing_y_values_white[i,class_problem])
    conf_matrix_white_new[x,y] = conf_matrix_white_new[x,y]+1
    is_correct = out == testing_y_values_white[i,class_problem]
    is_correct1 = out == testing_y_values_white[i,class_problem]+1
    is_correct2 = out == testing_y_values_white[i,class_problem]-1
    print('Actual data class: {}'.format(testing_y_values_white[i,
        class_problem]))
    if is_correct:
        #print('CLASSIFIED CORRECTLY')
        count_correct = count_correct + 1
    elif is_correct1:
        #print('CLASSIFIED CORRECTLY')
        count_correct = count_correct + 1
    elif is_correct2:
        #print('CLASSIFIED CORRECTLY')
        count_correct = count_correct + 1
    else:
        print('CLASSIFIED INCORRECTLY')

print('Amount of guesses correct: {}'.format(count_correct))
print('Amount of total guesses: {}'.format(testing_y_values_white.shape
    [0]))
```

```
print("Final Testing Accuracy: ", count_correct/testing_y_values_white.
    shape[0])

#print(conf_matrix)


# In[12]:


# Final Confusion Matrix for white wine predictions

ax = sn.heatmap(conf_matrix_white_new, annot=True, fmt = 'g')
print("Final Testing Accuracy: ", count_correct/testing_y_values_white.
    shape[0])


# In[13]:


t2 = Tree()
splits_red = 10
tree_red = tree_node(training_data_red, training_y_values_red, t2,
    f_names, splits_red)
print(t.get_ascii(show_internal=True))

ts = TreeStyle()
ts.show_leaf_name = False
def my_layout(node):
   F = TextFace(node.name, tight_text=True)
   F.rotable = True
   F.border.width = 0
   F.margin_top = 2
   F.margin_bottom = 2
   F.margin_left = 2
   F.margin_right = 2
   add_face_to_node(F, node, column=0, position="branch-right")
ts.layout_fn = my_layout
ts.mode = 'r'
ts.arc_start = 270
ts.arc_span = 185
ts.draw_guiding_lines = True
ts.scale = 100
ts.branch_vertical_margin = 100
ts.min_leaf_separation = 100
ts.show_scale = True


# In[14]:
```

```
#initial testing for red wine

count_correct = 0
conf_matrix_red = np.zeros(100).reshape((10,10))
for i in range(testing_data_red.shape[0]):
    out = tree_red.infer(testing_data_red[i])
    x = int(out)
    y = int(testing_y_values_red[i,class_problem])
    conf_matrix_red[x,y] = conf_matrix_red[x,y]+1
    is_correct = out == testing_y_values_red[i,class_problem]
    print('Actual data class: {}'.format(testing_y_values_red[i,
        class_problem]))
    if is_correct:
        print('CLASSIFIED CORRECTLY')
        count_correct = count_correct + 1
    else:
        print('CLASSIFIED INCORRECTLY')

print('Amount of guesses correct: {}'.format(count_correct))
print('Amount of total guesses: {}'.format(testing_y_values_red.shape[0])
    )
print("Final Testing Accuracy: ", count_correct/testing_y_values_red.
    shape[0])


# In[16]:


# adjusted target classes

count_correct = 0
conf_matrix_red = np.zeros(100).reshape((10,10))
#print(conf_matrix_white)

for i in range(testing_data_red.shape[0]):
    out = tree_red.infer(testing_data_red[i])
    x = int(out)
    y = int(testing_y_values_red[i,class_problem])
    conf_matrix_red[x,y] = conf_matrix_red[x,y]+1
    is_correct = out == testing_y_values_red[i,class_problem]
    is_correct1 = out == testing_y_values_red[i,class_problem]+1
    is_correct2 = out == testing_y_values_red[i,class_problem]-1
    print('Actual data class: {}'.format(testing_y_values_red[i,
        class_problem]))
    if is_correct:
        #print('CLASSIFIED CORRECTLY')
        count_correct = count_correct + 1
    elif is_correct1:
        #print('CLASSIFIED CORRECTLY')
        count_correct = count_correct + 1
```

```
    elif is_correct2:
        #print('CLASSIFIED CORRECTLY')
        count_correct = count_correct + 1
    else:
        print('CLASSIFIED INCORRECTLY')

print('Amount of guesses correct: {}'.format(count_correct))
print('Amount of total guesses: {}'.format(testing_y_values_red.shape[0])
    )
print("Final Testing Accuracy: ", count_correct/testing_y_values_red.
    shape[0])


# In[17]:


# Final testing for red wine

ax_red = sn.heatmap(conf_matrix_red, annot=True, fmt = 'g')
print("Final Testing Accuracy: ", count_correct/testing_y_values_red.
    shape[0])
```

A.2 Naive Bayes

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
from math import *
import pandas as pd
from csv import reader
from csv import writer
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
import seaborn as sns


# In[2]:


df_white = pd.read_csv('winequality-white1.csv', ';')
actual_quality_w = np.asarray(df_white.quality)
actual_quality_w_test = actual_quality_w[round(len(actual_quality_w)*0.8)
    :]
```

```
actual_Quality_w = np.asarray(df_white.Quality)
actual_Quality_w_test = actual_Quality_w[round(len(actual_Quality_w)*0.8)
    :]
full_data_w = np.asarray(df_white)
df_white = df_white.drop(labels='quality', axis=1)
df_white = df_white.drop(labels='Quality', axis=1)
#print(full_data_w)



df_red = pd.read_csv('winequality-red1.csv', ';')
actual_quality_r = np.asarray(df_red.quality)
actual_quality_r_test = actual_quality_r[round(len(actual_quality_r)*0.8)
    :]
actual_Quality_r = np.asarray(df_red.Quality)
actual_Quality_r_test = actual_Quality_r[round(len(actual_Quality_r)*0.8)
    :]
full_data_r = np.asarray(df_red)
df_red = df_red.drop(labels='quality', axis=1)
df_red = df_red.drop(labels='Quality', axis=1)
#print(full_data_r)


# In[3]:


#Scatter Plot Matrices
#spm_white = pd.plotting.scatter_matrix(df_white, alpha=0.2, figsize=(25,
     25), hist_kwds={'bins':30})
#spm_red = pd.plotting.scatter_matrix(df_red, alpha=0.2, figsize=(25, 25)
    , hist_kwds={'bins':30})


# In[4]:


# Correlation between features for White Wine
plt.figure(figsize=(12,10))
correlation = df_white.corr()
sns.heatmap(correlation , annot=True, cmap=plt.cm.Reds)
plt.title('White Wine Correlation between the set of Features X and Y',
    fontsize = 20) # title with fontsize 20
plt.xlabel('Set of Features X', fontsize = 15) # x-axis label with
    fontsize 15
plt.ylabel('Set of Features Y', fontsize = 15) # y-axis label with
    fontsize 15
plt.show()
```

```
# In[5]:


# Correlation between features for Red Wine
plt.figure(figsize=(12,10))
correlation_r = df_red.corr()
sns.heatmap(correlation_r , annot=True, cmap=plt.cm.Reds)
plt.title('Red Wine Correlation between the set of Features X and Y',
    fontsize = 20) # title with fontsize 20
plt.xlabel('Set of Features X', fontsize = 15) # x-axis label with
    fontsize 15
plt.ylabel('Set of Features Y', fontsize = 15) # y-axis label with
    fontsize 15
plt.show()


# In[6]:


# Split data

# red wine
r_train = full_data_r[:round(len(full_data_r)*0.6)]
r_valid = full_data_r[round(len(full_data_r)*0.6):round(len(full_data_r)
    *0.8)]
r_test = full_data_r[round(len(full_data_r)*0.8):]

# white wine
w_train = full_data_w[:round(len(full_data_w)*0.6)]
w_valid = full_data_w[round(len(full_data_w)*0.6):round(len(full_data_w)
    *0.8)]
w_test = full_data_w[round(len(full_data_w)*0.8):]


# In[7]:


# method to split the dataset by class values, returns a dictionary
def separate(data):
    separated = dict()
    for i in range(len(data)):
        row = data[i]
        class_value = row[-2]
        if (class_value not in separated):
            separated[class_value] = list()
        separated[class_value].append(row)
    return separated


# In[8]:
```

```
# 1. seperate the training data by class
separated_r_train = separate(r_train)
separated_w_train = separate(w_train)

# print white keys
#print('White Wine keys: ', separated_w_train.keys())

# print red keys
#print('Red Wine keys: ', separated_r_train.keys())


# In[9]:


# Calculate the mean of a list of numbers
mean = lambda x: sum(x)/float(len(x))

# Calculate the standard deviation of a list of numbers
def stdev(features):
    avg = np.mean(features)
    variance = sum([(x-avg)**2 for x in features]) / float(len(features)
        -1)
    return sqrt(variance)


# Method for calculating the mean, stdev and count for each column in a
    dataset
# zip() function maps the similar index of multiple containers, so that
    they can be used as a single entity
# * operator unpacks dataset i.e. transforms list of lists --> separate
    lists for each row
# we delete the last column, as this is the column we are trying to
    predict(do not need summary statistics for it)
def calc_summary_stats(data):
    summaries = list()
    for col in zip(*data):
        summaries.append([mean(col), stdev(col), len(col)])
    del(summaries[-1])
    del(summaries[-1])
    return np.asarray(summaries)


# In[10]:


# 2. Summary of whole training data , printed : mean, standard deviation,
     count
# has 11 columns output as 11 rows repr. the input features
```

```
# White Wine
print('Whole data summary White Wine: ')
print('Mean; Standard Deviation; Number of Instances')
summaries_wTrain = calc_summary_stats(w_train)
for col in summaries_wTrain:
    print(col)

print('\n')

# Red Wine
print('Whole data summary Red Wine: ')
print('Mean; Standard Deviation; Number of Instances')
summaries_rTrain = calc_summary_stats(r_train)
for col in summaries_rTrain:
    print(col)


# In[11]:


# 3. Summarize data using according to class value

# Method for summarizing the dataset, and then organising these summaries
     by class values
# Utilises calc_summary_stats() and separate() functions
# Split dataset by class then calculate statistics for each row
def summary_stats_by_class(data):
    separated = separate(data)
    summaries = dict()
    for class_value, rows in separated.items():      # separated.items()
        returns key:value pairs from dictionary
        summaries[class_value] = calc_summary_stats(rows)
    return summaries


# In[12]:


# Summary of white wine training data according to class values
summary_wTrain = summary_stats_by_class(w_train)
for class_value in summary_wTrain:
    print('\n')
    print('Class Value: ', class_value)
    print('Mean; Standard Deviation; Number of Instances')
    for row in summary_wTrain[class_value]:
        print(row)
```

```
# In[13]:


# Summary of red wine training data according to class values
summary_rTrain = summary_stats_by_class(r_train)

for class_value in summary_rTrain:
    print('\n')
    print('Class Value: ', class_value)
    print('Mean; Standard Deviation; Number of Instances')
    for row in summary_rTrain[class_value]:
        print(row)


# In[14]:


# 5. Determining Class Probabilities
# will use statistics calculated from our training data to calculate
    probabilities for new data
def confuz(arr1,arr2):
    mx = np.zeros((10,10))
    for k in range(len(arr1)):
        mx[int(arr1[k])][int(arr2[k])] = mx[int(arr1[k])][int(arr2[k])]+1
    return(mx)


# In[15]:


# Method to calculate the Gaussian Probability Distribution Function
def calc_Gaussian_pdf(x, mean, std_dev):
    exponent = exp(-((x-mean)**2 / (2 * std_dev**2 )))
    pdf = (1 / (sqrt(2 * pi) * std_dev)) * exponent
    return pdf

# Calculate the probabilities of predicting each class for a given row
# summaries will come in as [mean, stdev, len]
def class_probabilities(summaries, row):
    total_rows = sum([summaries[label][0][2] for label in summaries])
    probabilities = dict()
    for class_value, class_summaries in summaries.items():
        probabilities[class_value] = summaries[class_value][0][2]/float(
            total_rows)
        for i in range(len(class_summaries)):
            mean, stdev, _ = class_summaries[i]
            probabilities[class_value] *= calc_Gaussian_pdf(row[i], mean,
                stdev)
    return probabilities
```

```
# In[16]:


# Naive Bayes on red wine

#some summary statistics for red wine on the training data
total_rows_red = len(r_train)
print('red wine summary statistics for r_train')
print('\n')
print('Number of rows in r_train: ', total_rows_red)

for class_value, class_summaries in summary_rTrain.items():
    print('\n')
    print('class_value: ', class_value)
    print('Number of different instances corresponding to class_value: ',
          summary_rTrain[class_value][0][2])

summary_r = summary_stats_by_class(r_train)
predictions_red_test = list()
for datapoint in r_test:
    prob_red_train = class_probabilities(summary_r, datapoint)
    best_classValue_r = None
    best_prob_r = -1
    for classValue, prob in prob_red_train.items():
        if best_classValue_r is None or prob > best_prob_r:
            best_prob_r = prob
            best_classValue_r = classValue
    predictions_red_test.append(best_classValue_r)

print('\n')
print('Class Probabilites for Red Wine:', prob_red_train)
print('\n')
print(predictions_red_test)
print('\n')

# convert predictions_red_test to match 'Quality' column
final_predictions_red = list()
for value in predictions_red_test:
    if value <= 4:
        final_predictions_red.append(1)
    elif 4 < value <= 7:
        final_predictions_red.append(2)
    else:
        final_predictions_red.append(3)

print(final_predictions_red)
```

```
# In[17]:


conf_mat_red = confusion_matrix(actual_quality_r_test,
    predictions_red_test)
print('\n')
print('Red Wine')
print ('Confusion Matrix :')
print(conf_mat_red)
print ('Accuracy Score :',accuracy_score(actual_quality_r_test,
    predictions_red_test))
print ('Report : ')
print (classification_report(actual_quality_r_test, predictions_red_test)
    )

conf_mat_red = confusion_matrix(actual_Quality_r_test,
    final_predictions_red)
print('\n')
print('Red Wine')
print ('Confusion Matrix :')
print(conf_mat_red)
print ('Accuracy Score :',accuracy_score(actual_Quality_r_test,
    final_predictions_red))
print ('Report : ')
print (classification_report(actual_Quality_r_test, final_predictions_red
    ))


# In[18]:


def confu1(arr1,arr2):
    mx = np.zeros((4,4))
    for k in range(len(arr1)):
        mx[int(arr1[k])][int(arr2[k])] = mx[int(arr1[k])][int(arr2[k])]+1
    return(mx)

confuzmc_red_quality = confuz(actual_quality_r_test, predictions_red_test
    )

df_cm_rq = pd.DataFrame(confuzmc_red_quality, range(10), range(10))
sns.heatmap(df_cm_rq, annot=True)
plt.show()


confuzmc_red_Quality = confuz(actual_Quality_r_test,
    final_predictions_red)

df_cm_rQ = pd.DataFrame(confuzmc_red_Quality, range(10), range(10))
sns.heatmap(df_cm_rQ, annot=True, fmt='g')
```

```
plt.show()


# In[19]:


# Naive Bayes on White Wine

#some summary statistics for white wine on the training data
total_rows_white = len(w_train)
print('\n')
print('white wine summary statistics for w_train')
print('Number of rows in w_train: ', total_rows_white)

for class_value, class_summaries in summary_wTrain.items():
    print('\n')
    print('class_value: ', class_value)
    print('Number of different instances corresponding to class_value: ',
          summary_wTrain[class_value][0][2])

summary_w = summary_stats_by_class(w_train)
predictions_white_test = list()
for datapoint in w_test:
    prob_white_train = class_probabilities(summary_w, datapoint)
    best_classValue_w = None
    best_prob_w = -1
    for classValue, prob in prob_white_train.items():
        if best_classValue_w is None or prob > best_prob_w:
            best_prob_w = prob
            best_classValue_w = classValue
    predictions_white_test.append(best_classValue_w)

print('\n')
print('Class Probabilites for White Wine:', prob_white_train)
print('\n')
print(predictions_white_test)
print('\n')

# convert predictions_red_test to match 'Quality' column
final_predictions_white = list()
for value in predictions_white_test:
    if value <= 4:
        final_predictions_white.append(1)
    elif 4 < value <= 7:
        final_predictions_white.append(2)
    else:
        final_predictions_white.append(3)

print(final_predictions_white)
```

```
# In[20]:


conf_mat_white = confusion_matrix(actual_quality_w_test,
    predictions_white_test)
print('\n')
print('White Wine')
print ('Confusion Matrix :')
print(conf_mat_white)
print ('Accuracy Score :',accuracy_score(actual_quality_w_test,
    predictions_white_test))
print ('Report : ')
print (classification_report(actual_quality_w_test,
    predictions_white_test))

conf_mat_white = confusion_matrix(actual_Quality_w_test,
    final_predictions_white)
print('\n')
print('White Wine')
print ('Confusion Matrix :')
print(conf_mat_white)
print ('Accuracy Score :',accuracy_score(actual_Quality_w_test,
    final_predictions_white))
print ('Report : ')
print (classification_report(actual_Quality_w_test,
    final_predictions_white))


# In[21]:


confuzmc_white_quality = confuz(actual_quality_w_test,
    predictions_white_test)

df_cm_wq = pd.DataFrame(confuzmc_white_quality, range(10), range(10))
sns.heatmap(df_cm_wq, annot=True, fmt='g')
plt.show()


confuzmc_white_Quality = confuz(actual_Quality_w_test,
    final_predictions_white)

df_cm_wQ = pd.DataFrame(confuzmc_white_Quality, range(10), range(10))
sns.heatmap(df_cm_wQ, annot=True, fmt='g')
plt.show()


# In[22]:
```

```
# drop correlated features for red wine to see improved accuracy and
    repeat Naive Bayes
df_red = pd.read_csv('winequality-red1.csv', ';')
actual_quality_r = np.asarray(df_red.quality)
actual_quality_r_test = actual_quality_r[round(len(actual_quality_r)*0.8)
    :]
actual_Quality_r = np.asarray(df_red.Quality)
actual_Quality_r_test = actual_Quality_r[round(len(actual_Quality_r)*0.8)
    :]
df_red = df_red.drop(labels='citric acid', axis=1)
df_red = df_red.drop(labels='fixed acidity', axis=1)
df_red = df_red.drop(labels='density', axis=1)
#df_red = df_red.drop(labels='total sulfur dioxide', axis=1)
#df_red = df_red.drop(labels='free sulfur dioxide', axis=1)
full_data_r = np.asarray(df_red)
#df_red = df_red.drop(labels='quality', axis=1)
df_red = df_red.drop(labels='Quality', axis=1)


# In[23]:


df_red.columns


# In[24]:


r_train = full_data_r[:round(len(full_data_r)*0.6)]
r_valid = full_data_r[round(len(full_data_r)*0.6):round(len(full_data_r)
    *0.8)]
r_test = full_data_r[round(len(full_data_r)*0.8):]

separated_r_train = separate(r_train)
summaries_rTrain = calc_summary_stats(r_train)
summary_rTrain = summary_stats_by_class(r_train)


# In[25]:


total_rows_red = len(r_train)
summary_r = summary_stats_by_class(r_train)
predictions_red_test = list()
for datapoint in r_test:
    prob_red_train = class_probabilities(summary_r, datapoint)
    best_classValue_r = None
    best_prob_r = -1
    for classValue, prob in prob_red_train.items():
```

```
        if best_classValue_r is None or prob > best_prob_r:
            best_prob_r = prob
            best_classValue_r = classValue
    predictions_red_test.append(best_classValue_r)

print('\n')
print('Class Probabilites for Red Wine:', prob_red_train)
print('\n')
print(predictions_red_test)
print('\n')

# convert predictions_red_test to match 'Quality' column
final_predictions_red = list()
for value in predictions_red_test:
    if value <= 4:
        final_predictions_red.append(1)
    elif 4 < value <= 7:
        final_predictions_red.append(2)
    else:
        final_predictions_red.append(3)

print(final_predictions_red)


# In[26]:


conf_mat_red = confusion_matrix(actual_quality_r_test,
    predictions_red_test)
print('\n')
print('Red Wine')
print ('Confusion Matrix :')
print(conf_mat_red)
print ('Accuracy Score :',accuracy_score(actual_quality_r_test,
    predictions_red_test))
print ('Report : ')
print (classification_report(actual_quality_r_test, predictions_red_test)
    )

conf_mat_red = confusion_matrix(actual_Quality_r_test,
    final_predictions_red)
print('\n')
print('Red Wine')
print ('Confusion Matrix :')
print(conf_mat_red)
print ('Accuracy Score :',accuracy_score(actual_Quality_r_test,
    final_predictions_red))
print ('Report : ')
print (classification_report(actual_Quality_r_test, final_predictions_red
    ))
```

```
# In[27]:


confuzmc_red_quality = confuz(actual_quality_r_test, predictions_red_test
    )

df_cm_rq = pd.DataFrame(confuzmc_red_quality, range(10), range(10))
sns.heatmap(df_cm_rq, annot=True)
plt.show()


confuzmc_red_Quality = confuz(actual_Quality_r_test,
    final_predictions_red)

df_cm_rQ = pd.DataFrame(confuzmc_red_Quality, range(10), range(10))
sns.heatmap(df_cm_rQ, annot=True, fmt='g')
plt.show()


# In[28]:


# drop correlated features for white wine to see improved accuracy and
    repeat Naive Bayes
df_white = pd.read_csv('winequality-white1.csv', ';')
actual_quality_w = np.asarray(df_white.quality)
actual_quality_w_test = actual_quality_w[round(len(actual_quality_w)*0.8)
    :]
actual_Quality_w = np.asarray(df_white.Quality)
actual_Quality_w_test = actual_Quality_w[round(len(actual_Quality_w)*0.8)
    :]
df_white = df_white.drop(labels='density', axis=1)
df_white = df_white.drop(labels='residual sugar', axis=1)
df_white = df_white.drop(labels='free sulfur dioxide', axis=1)
df_white = df_white.drop(labels='total sulfur dioxide', axis=1)
full_data_w = np.asarray(df_white)
df_white = df_white.drop(labels='quality', axis=1)
df_white = df_white.drop(labels='Quality', axis=1)


# In[29]:


w_train = full_data_w[:round(len(full_data_w)*0.6)]
w_valid = full_data_w[round(len(full_data_w)*0.6):round(len(full_data_w)
    *0.8)]
w_test = full_data_w[round(len(full_data_w)*0.8):]
```

```
separated_w_train = separate(w_train)
summaries_wTrain = calc_summary_stats(w_train)
summary_wTrain = summary_stats_by_class(w_train)


# In[30]:


total_rows_white = len(w_train)
summary_w = summary_stats_by_class(w_train)
predictions_white_test = list()
for datapoint in w_test:
    prob_white_train = class_probabilities(summary_w, datapoint)
    best_classValue_w = None
    best_prob_w = -1
    for classValue, prob in prob_white_train.items():
        if best_classValue_w is None or prob > best_prob_w:
            best_prob_w = prob
            best_classValue_w = classValue
    predictions_white_test.append(best_classValue_w)

print('\n')
print('Class Probabilites for White Wine:', prob_white_train)
print('\n')
print(predictions_white_test)
print('\n')

# convert predictions_red_test to match 'Quality' column
final_predictions_white = list()
for value in predictions_white_test:
    if value <= 4:
        final_predictions_white.append(1)
    elif 4 < value <= 7:
        final_predictions_white.append(2)
    else:
        final_predictions_white.append(3)

print(final_predictions_white)


# In[31]:


conf_mat_white = confusion_matrix(actual_quality_w_test,
    predictions_white_test)
print('\n')
print('White Wine')
print ('Confusion Matrix :')
print(conf_mat_white)
print ('Accuracy Score :',accuracy_score(actual_quality_w_test,
```

```
    predictions_white_test))
print ('Report : ')
print (classification_report(actual_quality_w_test,
    predictions_white_test))

conf_mat_white = confusion_matrix(actual_Quality_w_test,
    final_predictions_white)
print('\n')
print('White Wine')
print ('Confusion Matrix :')
print(conf_mat_white)
print ('Accuracy Score :',accuracy_score(actual_Quality_w_test,
    final_predictions_white))
print ('Report : ')
print (classification_report(actual_Quality_w_test,
    final_predictions_white))


# In[32]:


confuzmc_white_quality = confuz(actual_quality_w_test,
    predictions_white_test)

df_cm_wq = pd.DataFrame(confuzmc_white_quality, range(10), range(10))
sns.heatmap(df_cm_wq, annot=True, fmt='g')
plt.show()


confuzmc_white_Quality = confuz(actual_Quality_w_test,
    final_predictions_white)

df_cm_wQ = pd.DataFrame(confuzmc_white_Quality, range(10), range(10))
sns.heatmap(df_cm_wQ, annot=True, fmt='g')
plt.show()
```

A.3 Normal Logistic Regression

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import pandas as pd
import seaborn as sn
import matplotlib.pyplot as plt
import math
from sklearn.metrics import confusion_matrix
```

```
# In[2]:


full_data_w = [] #white wine data
full_data_r = [] #red whine data

def Pulldata(file):           #Pulling white wine data
    data_file = open(file, 'r')
    lines = data_file.readlines()
    temparr = []
    for k in lines:
        stuff = np.array(k.split(';'))
        temparr.append(stuff)
    data_file.close()
    return(temparr)

full_data_w = Pulldata('winequality-white.csv')
full_data_r = Pulldata('winequality-red.csv')

#print(full_data_r)
#print(full_data_w)


# In[3]:


#splitting data into train,validate and test data (60,20,20)%
WLabels = full_data_w[0]
del full_data_w[0]
RLabels = full_data_r[0]
del full_data_r[0]

WTrain = full_data_w[:round(len(full_data_w)*0.6)]
WValid = full_data_w[round(len(full_data_w)*0.6):round(len(full_data_w)
    *0.8)]
WTest  = full_data_w[round(len(full_data_w)*0.8):]

RTrain = full_data_r[:round(len(full_data_r)*0.6)]
RValid = full_data_r[round(len(full_data_r)*0.6):round(len(full_data_r)
    *0.8)]
RTest  = full_data_r[round(len(full_data_r)*0.8):]


print(len(WTrain))
print(len(WValid))
print(len(WTest))
print()
print(len(RTrain))
```

```
print(len(RValid))
print(len(RTest))


# In[4]:


#print(print(WTrain))


# In[5]:


#Creating Design matrix : (DM) From Train Data


DMW = np.ones((len(WTrain),len(WLabels)+1))
DMR = np.ones((len(RTrain),len(RLabels)+1))


print(DMW.shape)
print(DMR.shape)

def CreateDM(DMtemp,DM): #Function to create Design Matrix
    indexk = 0

    for k in DMtemp:
        indexj = 0
        for j in k:
            DM[indexk][indexj+1] = DMtemp[indexk][indexj]
            indexj=indexj+1
        indexk=indexk+1
    return(DM)

DMW = CreateDM(WTrain , DMW) #Realocation of each designmatrix from ones
    to data

DMR = CreateDM(RTrain , DMR) #Realocation of each designmatrix from ones
    to data

DMW = np.delete(DMW, 13, 1)
DMR = np.delete(DMR, 13, 1)

#Creating Design Matrix for validation Data
DMWValid = np.ones((len(WValid),len(WLabels)+1))
DMRValid = np.ones((len(RValid),len(RLabels)+1))

DMWValid = CreateDM(WValid,DMWValid) #Realocation of each designmatrix
    from ones to data
DMRValid = CreateDM(RValid,DMRValid) #Realocation of each designmatrix
```

```
    from ones to data

DMWValid = np.delete(DMWValid, 13, 1)
DMRValid = np.delete(DMRValid, 13, 1)


#Create design matrix for Testing Data
DMWTest = np.ones((len(WTest),len(WLabels)+1))
DMRTest = np.ones((len(RTest),len(RLabels)+1))

DMWTest = CreateDM(WTest,DMWTest) #Realocation of each designmatrix from
    ones to data
DMRTest = CreateDM(RTest,DMRTest) #Realocation of each designmatrix from
    ones to data

DMWTest = np.delete(DMWTest, 13, 1)
DMRTest = np.delete(DMRTest, 13, 1)



# In[6]:


print(DMR)


# In[7]:


def hFunction(DM,theta_values):
    return 1/(1+np.exp(-np.dot(DM,theta_values)))#H(x,Theta) Calculation

def GradientDescent(theta_values,theta_valuesOld,DM,alpha,lambduh):#
    Gradient Descent Function
    while np.sqrt(np.sum(np.power(theta_values - theta_valuesOld, 2))) >
        0.0005: # while euclidean norm > 0.0005 (so epsilon = 0.0005)
        theta_valuesOld = theta_values # set old parameter values to
            parameter values before they are updated
        for i in range(DM.shape[0]):
            copy = np.copy(theta_values)
            copy[0] = 0
            theta_values = theta_values - alpha*( (hFunction(DM[i,:12],
                theta_values) - DM[i][12]) * DM[i,:12] ) # update the
                parameters using the update rule



    return theta_values #return our theta values for this model

def convertMatrix(value,DM): #Function to convert our design matrix for
```

```
    oneVsrest multiclass classification
    convDM = DM.copy()
    index = 0
    for k in DM:
        if k[12] == float(value):
            convDM[index][12] = 1
        else:
            convDM[index][12] = 0
        index = index+1
    return convDM


#soft max function
def softMax(ThetaList,TestItem):
    probabilities=[] #initialises probabilities array that holds
        probability for each class
    value=0
    for i in ThetaList: #loops through ThetaList
        value=value+np.exp(np.dot(i,TestItem)) #calculates denominator of
            softmax function
    for j in ThetaList:
        probability=(np.exp(np.dot(j,TestItem)))/value #calculates
            probability of each class and appends it to the probabilties
            array
        probabilities.append(probability)
    return probabilities
def confuz(arr1,arr2):
    mx = np.zeros((10,10))
    for k in range(len(arr1)):
        mx[int(arr1[k])][int(arr2[k])] = mx[int(arr1[k])][int(arr2[k])]+1
    return(mx)


# In[8]:


convertedDMR=convertMatrix(1,DMR)#create a converted design matrix where
    target variable only has two values 1 and 0 (1 represents the value
    you use as a parameter and 0 represents every other value besides
    the parameter value)
print(convertedDMR)


# In[9]:


#loop for each possible quility /10
ThetaList = []
alpha=1e-4#learning rate
valuelist = []
```

```
lamduh=0.5 #regularisation

for k in range(11):
    convertedDMR=convertMatrix(k,DMR)
    theta_values=np.random.uniform(size=12)#initialize our initial thetas
    theta_valuesOld=np.zeros(12)
    theta_values=GradientDescent(theta_values,theta_valuesOld,
        convertedDMR,alpha,lamduh)
    ThetaList.append(theta_values)


# In[10]:


### After running this we have:
#ThetaList from the learning phase

DMRValidAns = DMRValid[:,12]
DMRtempValid = np.delete(DMRValid, 12, 1)
#print(DMRAns)
GuessV = []
for k in range(len(DMRtempValid)):
    GuessV.append(np.argmax(softMax(ThetaList,DMRtempValid[k])))

#print(Guess)



confuzmc = confuz(DMRValidAns,GuessV)

index = 0
correct = 0
incorrect = 0
for k in DMRValidAns:
    if k == GuessV[index] or k == GuessV[index]+1 or k == GuessV[index
        ]-1:
        correct=correct+1
    else:
        incorrect = incorrect+1
    index = index+1



df_cm = pd.DataFrame(confuzmc, range(10), range(10))
sn.heatmap(df_cm, annot=True)
plt.show()
print("Accuracy of validation data is:",(correct/len(GuessV)*100),"%")
print("Error of validation data is:",100-(correct/len(GuessV)*100),"%")
```

```
# In[11]:


plt.clf()
DMRAns = DMRTest[:,12]
DMRtempTest = np.delete(DMRTest, 12, 1)
#print(DMRAns)
Guess = []
for k in range(len(DMRtempTest)):
    Guess.append(np.argmax(softMax(ThetaList,DMRtempTest[k])))

#print(Guess)



#print(confuz(DMRAns,Guess))


# In[12]:


index = 0
correct = 0
incorrect = 0
for k in DMRAns:
    if k == Guess[index] or k == Guess[index]+1 or k == Guess[index]-1:
        correct=correct+1
    else:
        incorrect = incorrect+1
    index = index+1
print("Accuracy of Test Data is:",(correct/len(Guess)*100),"%")
print("Error of Test Data is:",100-(correct/len(Guess)*100),"%")

confizmc1 =confuz(DMRAns,Guess)
df_cm = pd.DataFrame(confizmc1, range(10), range(10))
sn.heatmap(df_cm, annot=True)
plt.show()


# In[13]:


#For white wine
#loop for each possible quility /10

ThetaListW = []
alpha=1e-4#learning rate
lamduh=0.5 #regularisation

for k in range(11):
```

```
    convertedDMW=convertMatrix(k,DMW)
    theta_values=np.random.uniform(size=12)#initialize our initial thetas
    theta_valuesOld=np.zeros(12)
    theta_values=GradientDescent(theta_values,theta_valuesOld,
        convertedDMR,alpha,lamduh)
    ThetaListW.append(theta_values)


# In[14]:


DMWValidAns = DMWValid[:,12]
DMWtempValid = np.delete(DMWValid, 12, 1)
GuessV = []
for k in range(len(DMWtempValid)):
    GuessV.append(np.argmax(softMax(ThetaList,DMWtempValid[k])))

index = 0
correct = 0
incorrect = 0
for k in DMWValidAns:
    if k == GuessV[index] or k == GuessV[index]+1 or k == GuessV[index
        ]-1:
        correct=correct+1
    else:
        incorrect = incorrect+1
    index = index+1

confizmc=confuz(DMWValidAns,GuessV)
df_cm = pd.DataFrame(confizmc, range(10), range(10))
sn.heatmap(df_cm, annot=True)
plt.show()
print("Accuracy of validation data is:",(correct/len(GuessV)*100),"%")
print("Error of validation data is:",100-(correct/len(GuessV)*100),"%")


# In[15]:


plt.clf()
DMWAns = DMWTest[:,12]
DMWtempTest = np.delete(DMWTest, 12, 1)

Guess = []
for k in range(len(DMWtempTest)):
    Guess.append(np.argmax(softMax(ThetaList,DMWtempTest[k])))
index = 0
correct = 0
incorrect = 0
for k in DMWAns:
```

```
    if k == Guess[index] or k == Guess[index]+1 or k == Guess[index]-1:
        correct=correct+1
    else:
        incorrect = incorrect+1
    index = index+1
print("Accuracy of Test Data is:",(correct/len(Guess)*100),"%")
print("Error of Test Data is:",100-(correct/len(Guess)*100),"%")


confizmc=confuz(DMWAns,Guess)
df_cm = pd.DataFrame(confizmc, range(10), range(10))
sn.heatmap(df_cm, annot=True)
plt.show()


# In[ ]:




# #
```