Unveiling the Mysteries of White Wines: A Machine Learning Analysis of Wine Quality Dataset

Yan Lin

Executive Summary

Embark on a journey through the picturesque vineyards of northern Portugal, where we explore the unique characteristics of white vinho verde wines. Discover the hidden secrets of white vinho verde wines with this innovative analysis report.

The aim of this report is to provide a new perspective on the quality of wine by examining the relationship between physical and chemical tests and wine quality. In Section 1, wine is of high quality when its quality score is seven or higher, and of low quality otherwise. Many machine learning techniques were employed to examine the relationship between these 11 predictors and the corresponding wine quality attributes, and the resulting dataset was utilised to develop a model which extends the way we predict the quality of white wines into the future.

The initial technique was based on a single classification tree (Section 2.1), which despite being basic and straightforward, failed to accurately represent the connection between the predictor variables and wine quality, even when accounting for alcohol content alone. This renders the generation of useful predictions impossible. The second method is the advanced tree-based model (Section 2.2), which effectively models the importance of each predictor variable and is able to make accurate predictions. In particular, one of the bagging methods is recommended for use in any future analysis or similar dataset. It is very stable and fault tolerant, and it can be applied to more than just predicting wine quality. Deep neural networks (Section 2.3) would have been the most accurate type of prediction, but the advanced tree-based model offers certain advantages for this modestly sized dataset. In Section 2.4, a comparison of modelling methodologies is offered. Deep neural networks are a vast universe, and we can experiment to our heart's content with optimising and tuning the parameters within them.

Through our analysis, we could determine which factors have the greatest impact on wine quality: alcohol, volatile acidity, and chlorides seem to have the greatest impact on wine quality. The effects of citric acidity and fixed acidity are negligible. We used to just consider alcohol, acidity, and sweetness when evaluating wine. My report challenges the conventional notions of what constitutes a good wine and offers a more nuanced perspective on wine quality, which will be of great use to wine enthusiasts. We encourage stakeholders to incorporate them into future planning and decision-making.

1 Introduction

Portugal is a top ten global exporter of wine, making the evaluation of wine quality a critical component of the country's wine production. By quantifying wine quality and distinguishing it as either good or bad, we can improve wine production for future years. In this analysis, we will examine the wine quality dataset

from the UCI database[1], focusing on the white vinho verde wines from northern Portugal collected between May 2004 and February 2007, as the red and white variants have distinct tastes, the correlations between the predictor variables vary greatly, and the white variants make up the majority of the dataset.

The point is that I will consider apply a tree-based model classifier and a deep neural network classifier for this dataset to identify the most crucial features, to investigate how performance compares when performing dichotomous and multiclassification tasks on white wine quality, and to find out which classifier is simpler to understand and interpret its predictions.

Although though this dataset is more than ten years old, analysing these challenges can still assist guide our judgement of future approaches when tackling this type of problem (with a similar dataset size). It is therefore crucial to think critically when creating and evaluating classifiers so that their predictions can be given the weight they deserve in practical applications.

1.1 Data Cleaning

The original dataset for the white variant contains 4898 observations in 12 columns (the response variable is the last column 'quality') and contains information on the numerous chemical characteristics of the fermentation system and associated mass fractions, which have been inspected for the absence of missing values. See Appendix 1 Table 1 for a summary of the variables.

I examined the data categories in both the R and Python environments, converting the first 11 columns to numeric based on the predictor variables for the physical chemistry tests. The last column 'quality' was found to be a discrete variable falling only between 3 and 9, as opposed to a continuous variable, with most of its values falling between 5, 6, 7, which already occupies 92.58% of the observed sample (Figure 1.1 in Appendix 1). Based on these sensory characteristics, I will develop a wine classifier to differentiate between wines of high and low quality. When the quality value is more than or equal to 7, it means 'good' and maps to a value of 1. When the quality value falls within the range of 3 to 6, it means 'bad' and maps to a value of 0.

1.2 Exploratory data analysis

Numerical analysis

The cleaned dataset had 4898 observations, of which 1060 were good wines and 3838 were bad wines. Hence, 78% of the observations were categorised as bad. The base R summary function (see Appendix 1 Table 2) shows clustering at lower values of the variables.

I evaluated the correlation coefficients between all the variables in the dataset (see Appendix 1 Table 3) and discovered that the linear association between them was quite weak, with the most significant correlation coefficient value being 0.39 (variable: 'alcohol') and the second most significant correlation coefficient being -0.28 (variable: 'density'). This shows that quality levels may be affected by multiple variables simultaneously, as opposed to a single one.

Graphical analysis

Given that there are 4898 cases, the level of overlap is likely to be high. The variance of the two classes

[1] UCI Machine Learning Repository: Wine Quality Data Set. (n.d.). Archive.ics.uci.edu. https://archive.ics.uci.edu/ml/datasets/wine+quality

looks significantly less for the good wines group compared to the bad wines group (see Appendix 1 Table 4). This can also be observed in the covariance matrices of the two groups (see Appendix 1 Table 5&6).

An important observation from the analysis of the scatter plot and other visualisation tools (see Appendix 1 Figure 1.2-1.4) is the wide range of values for the various pairs of variables in the bad wine group. The lower values of the variables alone are not sufficient to classify them as bad wines (corresponding to a quality value of 0). It is also evident from the fact that the variables 'density' and 'alcohol' were selected in the scatter plots of the two groups (see Appendix 1 Figure 1.5). This suggests that there is no model based on this data that can confidently distinguish between the different classes. Another observation is that some features are skewed and almost all features have outliers.

In summary, the quality of red wine may be related to the interaction between several variables, for example, a very strong positive association between density and residual sugar and a particularly strong negative association between alcohol and density. This non-linear relationship also suggests that using traditional linear regression models to predict the quality of red wines may not be accurate enough. Therefore, we need to consider more sophisticated machine learning algorithms.

2 Analysis and Modelling

Motivation

Due to the discrete nature of the response variable 'quality' for binary classification, only the tree-based model and deep neural network model are investigated in this research. Although graphical analysis has proved the existence of non-linear relationships between variables and the generalised weighted model (GAM) is well suited for dealing with non-linear interactions (classification / regression). The determining factor is that the output variable of a binary classification problem is a binary variable (not 0 or 1), whereas the output of a GAM model is typically a continuous value that must be classified by manually establishing a threshold, with different thresholds leading to different classification results. Hence, this is not the best option.

The capacity of Convolutional Neural Networks (CNNs) models to extract spatial and temporal information from the input data makes them ideally suited for modelling data types such as photos and text. CNNs build too complex models for binary classification problems and require a considerable quantity of training data. In addition, self-compilers, which are used to translate source code to machine code, are not ideal for modelling classification problems.

Tree-based models are advantageous because they permit feature selection and are simple to interpret and visualise. With vast volumes of data and parameters, deep neural networks can automatically extract and learn features and obtain good classification results. GAM, on the other hand, are less adept in feature selection and autonomous learning. Hence, only the application of these two models to this dataset is investigated in the following section.

2.1 Single Classification Tree

A classification tree separates the training data into homogenous subgroups (i.e., groups with comparable response values) and then fits a simple constant in each subgroup if the entire data set is divided into half training set and half test set. Mathematically, A classification tree terminates in a leaf node corresponding to a hyper-rectangle in feature (predictor) space, $R_j, j = \{1, \dots, j\}$, indicating that the feature space

represented by $X_1, X_2, \ldots, X_P$ is partitioned into J unique and non-overlapping regions.

There are two sorts of labels in this dataset: 0 and 1. The tree model assigns a class label to each R_j, i.e., the model predicts the output based on the class with majority representation or offers predicted probabilities depending on the proportion of each class within the subgroup. We use a top-down and greedy method, recursive binomial splitting, to generate this single classification tree model (i.e., two choices are made at each step). Yet, as it is formed in stages, it is not necessarily the best tree.

We chose the classification error rate (the proportion of non-most common classes in the training set in this region) as the evaluation criterion, whose expression is:

$$E = 1 - \max_k(\hat{p}_{mk})$$

$\hat{p}_{mk}$ represents the proportion of class k in the training set of the m^{th} region. We can also use the Gini coefficient[2] and Cross-entropy[3] as evaluation indicators, both of which are more sensitive to the purity of the nodes. However, our aim is to fit a higher prediction accuracy, so we choose the classification error rate in this paper.

Fitting a single classification tree using the training set, we found that the actual feature variables used were 'alcohol', 'volatile.acidity', 'chlorides', 'pH'. This model produced a total of 9 leaf nodes with the misclassification rate was 18.29%, i.e., 448 out of 2449 samples were misclassified. (See Appendix 2 Figure 2.1) For classification tree, the deviance is calculated using cross-entropy. Cross entropy is a frequently employed loss function for classification issues, but its mean value of 0.7822 does not provide an adequate match at this time. Performing predictive classification on the test set, we found a correct rate of 79.67%.

The above tree may be suffered from overfitting: a pruned tree with fewer terminal nodes may have better predictive performance, have a better bias-variance trade-off and be easier to interpret. [4] Cost complexity pruning introduces a nonnegative tuning parameter α, for trees minimizing

$$\sum_{j=1}^{|T|} \sum_{i:x_i \in R_j} (y_i - \hat{y}_{R_j})^2 + \alpha|T|$$

Where |T| is the number of terminal nodes of the tree T. Minimization is accomplished by removing terminal nodes from the bottom of the tree up. The larger α, the more pruning.

By analysing the graph of the classification tree after cross-validation pruning (See Appendix 2 Figure 2.2), I chose the model at the third cross-validation pruning for optimal pruning to get a new classification tree (See Appendix 2 Figure 2.3), with a correct rate of 79.37%, a decrease of 0.30%. The actual characteristic variable used is 'alcohol'. In addition, the cross-entropy increased from 0.7822 to 0.8703, indicating a performance decline. This circumstance suggests an excessive amount of pruning, which has rendered a reasonably simple tree unfit.

---

[2] $G = \sum_{k=1}^{K} \widehat{p_{mk}}(1 - \widehat{p_{mk}})$, it measures the total variance of the k categories.

[3] $D = -\sum_{k=1}^{K} \widehat{p_{mk}} log \widehat{p_{mk}}$, if the m^{th} node is purer, the value of cross entropy is smaller.

[4] James et al. 2013, p. 307.

Table 2.1 Confusion matrix for a single tree unpruned and pruned

Single tree	bad(Unpruned)	good(Unpruned)	bad(prune)	good(prune)
bad	1810	409	1828	434
good	89	141	71	116

2.2 Advanced Tree-Based Methods

Despite the fact that the model tree of a single classification tree is simple to comprehend, it frequently has a high variance, with tiny changes in the sample resulting in huge cascading changes in the fit and low predictive accuracy. Thus, it is time to implement the integrated learning techniques bagging, random forest, and boosting to enhance the model's performance.

Bagging

The aim of bagging is to improve prediction accuracy by reducing variance. Repeated sampling of this dataset produces a B distinct self-sampling training set. This model is fitted using the b^{th} self-sampling training set and the prediction $\hat{f}^{*b}(x)$ is derived. Finally averaging over all the predicted values gives:

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x)$$

For the bagging strategy in classification issues：at a given test value, the prediction classes supplied by each of the B trees are recorded, and then majority voting is used to get the overall forecast, which is the class that appears most frequently among the B predictions. When B is large, no overfitting occurs either. Bootstrap samples are random samples with replacement, which means that samples from the original dataset may appear many times in the bootstrap sample. The fact that only 2/3 of the original training data will be utilised in the bootstrap resampling and the other 1/3 will automatically provide out-of-sample estimates is equivalent to 3-fold cross-validation.

The bagging approach considers the same number of predictor variables at each split point as the total number of predictor variables. For this data set, $m = p = 11$. Here is the big difference between the bagging method and the random forest method ($m \approx \sqrt{p} = 4$).

For the same training set, I fit a bagging tree classifier, create a bootstrap tree with B=10 and 11 features, and calculate feature importance. A second time change to bootstrap tree B=100 and a third time change to bootstrap tree B=1000, fitting three models, confusion matrix, out-of-bag error estimation, and prediction accuracy are shown below (see Figure 2.4 in Appendix 2 for more information).

Table 2.2 Information on three times bagging classification trees

Bagging tree	ntree=10		ntree=100		ntree=1000	
	bad	good	bad	good	bad	good
bad	1772	272	1810	263	1810	263
good	127	278	89	287	89	287
OOB estimate of error rate	18.24%		13.96%		13.6%	
success rate	0.8370764		0.8562679		0.8562679	

Bagging Tree's classification performance improved as the number of trees increased. The OOB error rate has fallen from 18.24% to 13.96% and 13.6% as the number of trees increased from 10 to 100 and 1000. A significant finding is that the success rate of the Bagging Tree exceeded 83% for all tree counts, where bootstrap trees B=1000 and B=100 have higher success rates, but the out-of-bag error estimates are smallest at B=1000. Thus, only the random forest and boost models will be explored for the bootstrap tree B=1000. Another extremely interesting discovery is that in the confusion matrix the Bagging tree performs better for the 'bad' category and worse for the 'good' category.

Examining the feature importance and Gini coefficient of the bagging model at B=1000, we find that 'alcohol' is the most important feature, with MeanDecreaseAccuracy and MeanDecreaseGini of 113.30142 and 182.22565, respectively (see Figure 2.5 & 2.6 in Appendix 2). this means that during training, alcohol has a very significant impact on the classification accuracy and Gini coefficient of the model. 'volatile. acidity', 'chlorides', and 'pH' are also more important features. In this tree model, the higher the Gini coefficient, the greater the contribution of the features to the classification model. Thus, the category variable 'bad' with a Gini coefficient of 0.08669174 and the other category variable 'good' with a Gini coefficient of 0.50017224 contribute more to the classifier. At this time, we can also inspect Figure 2.7 in Appendix 2 and compare the structural distinctions between the various decision trees underlying the Bagging classification tree model.

In the set of bagging trees, most trees will use the strongest predictor variables for the top split point, which will cause all bagging methods to look similar. As a result, the predictor variables in these bagging trees are all highly correlated. This also means that the bagging method does not result in a significant reduction in average variance compared to a single tree.

Random Forest

An improvement on the bagging method led us to the random forest method. Very similar to the bagging method, a series of decision trees need to be built for the automatically sampled training set. But for each split point in the tree considered, a random sample containing m predictor variables is selected as a candidate variable from the full set of p predictor variables. The predictor variables used for this split point can only be selected from these m variables and are resampled at each split point, usually $m \approx \sqrt{p}$. In this dataset, we take m = 4. When many predictor variables are correlated, it is often effective to take a small value of m to build a random forest. Constructing the model in this way results in a reduction in both test error and OOB error.

Random forests consider only one of the predictor variables by requiring each split point to consider only one of itself, a procedure regarded as decorrelated to the tree. Using bootstrap trees of 1000 and m = 4 to continue building the random forest model (see Figure 2.8 in Appendix 2), we evaluated the OOB error to be 13.56%, with a success rate of 85.34% on the test set, a decrease of 0.285% compared to the bagging tree. Based on the importance measures of Gini importance and accuracy (see Figure 2.9 & 2.10 in Appendix 2), decorrelation still confirmed that 'alcohol' was the most significant feature.

Table 2.3 Confusion matrix for a random forest tree

A random forest tree	bad	good
bad	1809	269
good	90	281

Table 2.4 Confusion matrix for a boosting tree

A boosting tree	bad	good
bad	1767	303
good	132	247

Boosting

Each bagging tree is constructed on top of a self-sampling dataset, independently of the other trees. Like the gradient descent approach, the boosting method generates trees sequentially, with each tree produced utilising information from the previous tree. The boosting method is different from generating a large-scale decision tree as described above. Generating a large tree implies a strict fit data hard with possible overfitting, whereas the boosting method is a soothing way to train the model.

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \widehat{f^b}(x)$$

Continuing to fit a boosted classification tree, the parameter d controls the size of the tree (number of split points per tree, we take $d = 2$), the compression parameter λ (taking a very small positive value, we take $\lambda = 0.01$) permits more trees of different structure to vary the residuals, and the total number of trees B is the number of trees used in the fit (set above $B = 1000$). Binary classification of predicted outcomes (probability values), where probability values less than or equal to 0.5 are labelled as "bad" and those greater than 0.5 are labelled as "good", resulting in a confusion matrix (Table 2.4). The most important feature was 'alcohol', with a relative importance of 21.96% (see Figure 2.11 in Appendix 2). The boosted classification tree's correct prediction rate is 82.24%, which is inferior to that of the random forest tree and the bagging tree.

2.3 Deep Neural Networks

Neural networks are based on extensions of perceptrons, whilst deep neural networks (DNNs) can be understood as neural networks with many hidden layers, i.e., multi-layer perceptron (MLP). DNNs could learn extremely complex nonlinear functions and can automatically extract high-level features from raw input, making them ideal at processing large-scale and high-dimensional data.

Typically, such models contain numerous hidden layers and one output layer. Each hidden layer has numerous 'neurons' (h) that receive the output of the preceding layer and generate new outputs that serve as input for the subsequent layer. A deep neural network's neurons analyse their inputs with a linear transformation, which often consists of a weighted sum, and an activation function, which maps the results of the linear transformation to a non-linear space. Generally, training is performed using a back-propagation technique. The back propagation algorithm modifies the neuronal weights and biases based on the error between the neural network's output and the target label.

$$h_n = f(c_n + \boldsymbol{x}_n^T \boldsymbol{w}_n \boldsymbol{h}_{n-1})$$

The predictor variables (as a matrix $\boldsymbol{x}$) interact with the intercept c ('bias') and the coefficients $\boldsymbol{w}$ ('weights') of the non-linear activation function (f), together with the entire output of the previous layer (a matrix $\boldsymbol{h}_{n-1}$) is also combined with the current layer. Non-linear activation functions can make neural networks particularly good at classification tasks, where computing probabilities is central.

The dataset was divided into a training set consisting of 80% of the data and a test set of 20%. Features and labels were distinguished, and the features of the training and test sets were normalised. I subsequently attempted to configure a few hyperparameters to construct a classification model using a deep neural network.

As there are only two potential classes for wine quality, set the number of output classes to 2. This is a binary classification problem. Setting 'batch size = 128', the batch size controls the number of data samples processed simultaneously by the model during training. Greater batch sizes can reduce training durations on modern hardware, but they necessitate additional memory and can slow convergence. A batch size of 128 is a reasonable compromise between speed and memory usage in this situation. With 'epochs = 250', a high number, I desired to investigate the model's structure in its depths. Epochs controls the model to perform 250 iterations across the full training dataset while being trained.

A sequential model was defined using Keras' Sequential class[5], which enables the creation of a neural network by adding layers to the stack sequentially, with an input shape of 11 (all features) and a flatten layer to reshape the input data into a vector that can be passed to the first fully connected layer. The subsequent four layers are dense, completely connected layers, each containing 32 neurons with a 'ReLU' activation function. The final output layer is also a fully connected layer composed of two neurons and a 'softmax' activation function that outputs the probability distribution of the two categories.

In compile, the loss function 'categorical cross entropy' is taken for this dataset due to the anticipated binary class labels. And during training, the 'Adam' optimizer is employed to update the model weights. The metric used to evaluate the model's performance during training is 'accuracy'. In addition, I transform the binary training labels into a single thermal coding vector, which is precisely what the 'categorical cross entropy' loss function requires. Furthermore, 10% of the training data should be used for validation throughout the training process. The final calculations resulted in a loss of 1.49 and an accuracy of 0.8418 for the model on the normalised test features and binary test labels. Although the accuracy was satisfactory, it was evident that the model had been over-fitted and that the training and validation sets were rather distinct.

I tried changing the activation function to 'sigmoid' while leaving all other settings same. Not only did the accuracy decrease, but it was evident after 100 iterations that it had been overfitted again. To prevent overfitting, I added L1 (Lasso), L2 (Ridge) regularization terms at layers 2, 4, 6, and 8, so now there are 7 fully connected layers. The coefficients for both L1 and L2 are 0.01. this regularizer encourages the model to learn sparse and/or small values of weights, which helps prevent overfitting. It is evident from the iterations that the overfitting has been significantly reduced, the loss in the test set has been considerably lowered, and the accuracy is still greater than 83%, indicating a relatively balanced performance of the fit.

To continue refining the model, I added some additional features to the L1, L2 regularization terms to improve its training. Callbacks for early stopping and reduced learning rate were added using the callbacks parameter. These callbacks will be executed during training to monitor validation accuracy and adjust the learning rate if necessary. 'EarlyStopping' callbacks monitor 'val_accuracy'. If the accuracy does not improve beyond 30 epochs, training is stopped. 'ReduceLROnPlateau' callbacks monitor 'val_accuracy' and reduces the learning rate by a specified factor (0.1) if the accuracy does not improve by at least 0.0001 in 10 epochs time. This helps the model to converge faster and avoid falling into local minima. The fourth modification to the model achieves a better balance between continuing to reduce the loss and maintaining nearly constant accuracy. The following two figures show the change in accuracy and loss values during this model fit.

[5] https://keras.io/zh/

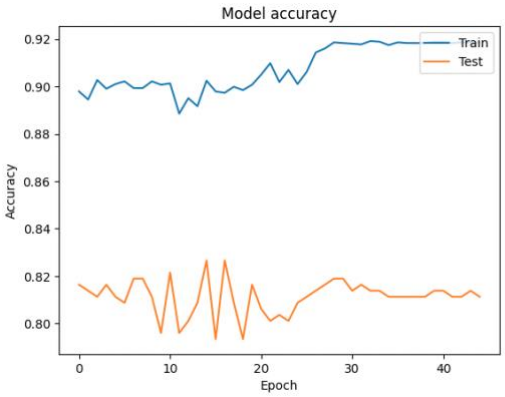


Figure 2.8 Accuracy fluctuation for model IV

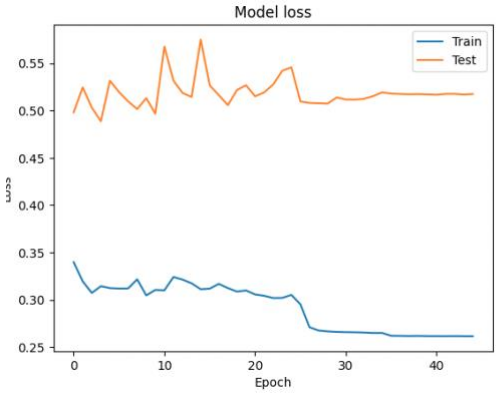


Figure 2.9 Loss fluctuation for model IV

A learning rate scheduler callback can also be included to alter the training learning rate. The optimiser is now set to the 'opt' object that was previously generated, which specifies the 'Adam' optimizer with a learning rate of 0.001. This is a great approach to prevent model loss.

Table 2.5 Accuracy and loss after five DNN model adjustments

	I	II	III	IV	V
Model Accuracy	0.8479591608047485	0.8459183573722839	0.8346938490867615	0.8377550840377808	0.8408163189888
Model Loss	1.3344687223434448	1.2608349323272705	0.4674417674541734	0.4723587036132812	0.4792219400405884

Model I is the most precise, however its accuracy is a result of overfitting. Model III is the outcome of adding L1, L2 regularisation while maintaining a relatively good accuracy rate. It has the lowest loss. Adding the callback procedure to Model IV achieves a satisfactory balance between accuracy and loss. Consequently, I feel model IV is the most effective deep neural network model.

2.4 Model Comparison

The correct prediction rates for each of the machine learning models are summarised in Table 2.6. The relatively low classification accuracy of the Single Tree model (79%) and the Gradient Boosting model (82%) may be attributable to the low complexity of the single decision tree model, which is unable to capture the dataset's complex relationships, and the Gradient Boosting model, which is more sensitive to outliers and requires more careful processing and conditioning of this dataset.

In contrast, the classification accuracy of the Bagging (85.6%) and Random Forest model (85.3%) is higher, most likely as a result of their ability to reduce the bias and variance of individual models by integrating the predictions of multiple underlying models, thereby enhancing the generalisation and performance of the models.

Deep neural network classification accuracy (83.7%) is marginally inferior to that of Bagging model classification accuracy. In reality, Deep Neural Network model has a strong fitting and expressive power, but the complexity of the model applied to this dataset is high, and overfitting issues occur several times. The Bagging model, on the other hand, is relatively straightforward and possesses excellent robustness and generalizability.

Table 2.6 Comparison of correct rates

Model	Correct prediction rate
Single tree (unpruned)	0.7966517
Single tree (pruned)	0.7937934
Bagging tree (B=1000)	0.8562679
Random forest tree (B=1000)	0.8534096
Boosting tree (B=1000)	0.8223765
Deep Neural Network	0.8377550840377808

3 Discussion & Further Work

The best model in terms of trade-offs between appropriateness, interpretability, and forecast accuracy is the bagging classification tree. By examining the bagging classification tree (B=1000), we found that 'alcohol' was the most critical feature, followed by 'volatile acidity' and 'chlorides' which were both quite significant. Citric acidity and fixed acidity were of little importance.

This dataset has 11 features, but creating new features by combining or transforming existing ones may improve the performance of the model. We also can use various techniques such as box plots, scatter plots and Z-cores to detect outliers in the dataset. This dataset is unbalanced, with significantly more bad wine than good wine. The dataset can be balanced by using techniques such as oversampling, under sampling or Synthetic Minority Oversampling Technique to improve the model's ability to predict minority groups. Since this work only investigates the white variant of red wine quality, the model performance of the red variant may be very different from that of the white variant; nevertheless, I can either explore the red variant separately or combine the two variants and resume. Considering the tests related to wine quality are mostly covered by the 11 variables, finding the appropriate predictive model is quite important for the application.

Five times I tweaked the neural network, but none of the predictions surpassed the accuracy of the bagging classification tree. Before working on this dataset, I hypothesised that a deep neural network would be the optimal model. The possible reasons for poor prediction performance are: 1. this dataset is not large enough (4898: medium); 2. there are relatively few features selected (11: medium); 3. the hyperparameters are poorly set. I briefly adjusted the batch size, learning rate, hidden layer, regularisation, and activation function, but was unable to find the optimal hyperparameters due to my insufficient personal knowledge base of deep neural networks. This issue should be solved by attempting cross-validation and, if the process takes too long, by accelerating the process through transfer learning. Alternatively, we may automatically identify the ideal neural network structure by employing neural architecture search (NAS).

In addition, we can explore more machine learning techniques for this white variant dataset, including support vector machines (SVM), k-nearest neighbours (KNN), and plain Bayes. The SVM model is well-suited to non-linearly differentiable classification tasks and this dataset, which is not analysed here due to space limitations. I may also investigate various approaches in integrated learning, such as AdaBoost, Stacking, Voting, etc., to weightedly combine the predictions of many base models to enhance the prediction accuracy to over 90%. Currently, the best prediction rate is just 85.6%, which is not flawless and comes at the expense of some interpretability. Continuously experimenting and exploring various machine learning techniques until we find the best model for the need.

Appendices

Appendix 1- Data cleaning & Exploratory data analysis

Table 1- Variables

Serial	Name	Description	Type
1	fixed.acidity	fixed acidity	numeric
2	volatile.acidity	volatile acidity	
3	citric.acid	citric acid	
4	residual.sugar	residual sugar	
5	chlorides	chlorides	
6	free.sulfur.dioxide	free sulfur dioxide	
7	total.sulfur.dioxide	total sulfur dioxide	
8	density	density	
9	pH	pH	
10	sulphates	sulphates	
11	alcohol	alcohol	
12	quality	output variable (based on sensory data): quality (score between 0 and 10)	target

Figure 1.1- Distribution of quality values for the white variant

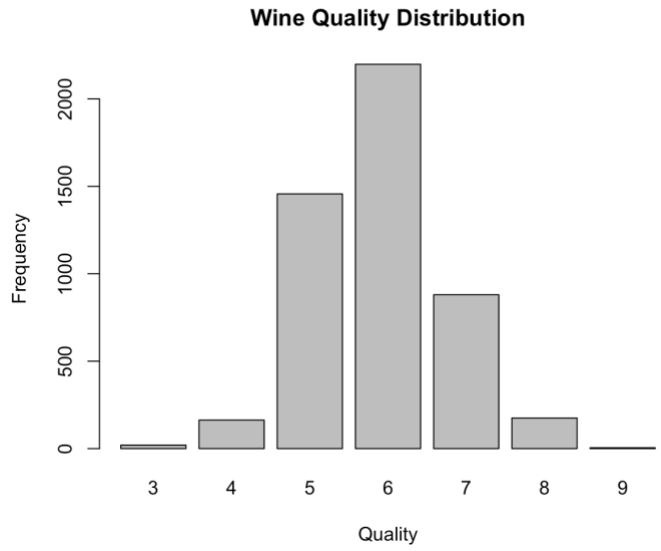


Table 2- Clean dataset summary

	count	mean	std	min	25%	50%	75%	max
fixed acidity	4898.0	6.854788	0.843868	3.80000	6.300000	6.80000	7.3000	14.20000
volatile acidity	4898.0	0.278241	0.100795	0.08000	0.210000	0.26000	0.3200	1.10000
citric acid	4898.0	0.334192	0.121020	0.00000	0.270000	0.32000	0.3900	1.66000
residual sugar	4898.0	6.391415	5.072058	0.60000	1.700000	5.20000	9.9000	65.80000
chlorides	4898.0	0.045772	0.021848	0.00900	0.036000	0.04300	0.0500	0.34600
free sulfur dioxide	4898.0	35.308085	17.007137	2.00000	23.000000	34.00000	46.0000	289.00000
total sulfur dioxide	4898.0	138.360657	42.498065	9.00000	108.000000	134.00000	167.0000	440.00000
density	4898.0	0.994027	0.002991	0.98711	0.991723	0.99374	0.9961	1.03898
pH	4898.0	3.188267	0.151001	2.72000	3.090000	3.18000	3.2800	3.82000
sulphates	4898.0	0.489847	0.114126	0.22000	0.410000	0.47000	0.5500	1.08000
alcohol	4898.0	10.514267	1.230621	8.00000	9.500000	10.40000	11.4000	14.20000
quality	4898.0	0.216415	0.411842	0.00000	0.000000	0.00000	0.0000	1.00000

Table 3- Correlation matrix

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
fixed acidity	1.000000	-0.022697	0.289181	0.089021	0.023086	-0.049396	0.091070	0.265331	-0.425858	-0.017143	-0.120881	-0.080748
volatile acidity	-0.022697	1.000000	-0.149472	0.064286	0.070512	-0.097012	0.089261	0.027114	-0.031915	-0.035728	0.067718	-0.067225
citric acid	0.289181	-0.149472	1.000000	0.094212	0.114364	0.094077	0.121131	0.149503	-0.163748	0.062331	-0.075729	-0.035330
residual sugar	0.089021	0.064286	0.094212	1.000000	0.088685	0.299098	0.401439	0.838966	-0.194133	-0.026664	-0.450631	-0.117085
chlorides	0.023086	0.070512	0.114364	0.088685	1.000000	0.101392	0.198910	0.257211	-0.090439	0.016763	-0.360189	-0.183118
free sulfur dioxide	-0.049396	-0.097012	0.094077	0.299098	0.101392	1.000000	0.615501	0.294210	-0.000618	0.059217	-0.250104	-0.023413
total sulfur dioxide	0.091070	0.089261	0.121131	0.401439	0.198910	0.615501	1.000000	0.529881	0.002321	0.134562	-0.448892	-0.162202
density	0.265331	0.027114	0.149503	0.838966	0.257211	0.294210	0.529881	1.000000	-0.093591	0.074493	-0.780138	-0.283871
pH	-0.425858	-0.031915	-0.163748	-0.194133	-0.090439	-0.000618	0.002321	-0.093591	1.000000	0.155951	0.121432	0.093510
sulphates	-0.017143	-0.035728	0.062331	-0.026664	0.016763	0.059217	0.134562	0.074493	0.155951	1.000000	-0.017433	0.047410
alcohol	-0.120881	0.067718	-0.075729	-0.450631	-0.360189	-0.250104	-0.448892	-0.780138	0.121432	-0.017433	1.000000	0.385132
quality	-0.080748	-0.067225	-0.035330	-0.117085	-0.183118	-0.023413	-0.162202	-0.283871	0.093510	0.047410	0.385132	1.000000

Table 4- Variance matrix

Group	qualtiy=1	qualtiy=0
fixed.acidity	0.590493	0.739594
volatile.acidity	0.008846	0.010461
citric.acid	0.006440	0.016885
residual.sugar	18.393985	27.294014
chlorides	0.000124	0.000554
free.sulfur.dioxide	190.181651	316.324283
total.sulfur.dioxide	1069.905402	1948.308789
density	0.000008	0.000008
pH	0.024684	0.022021
sulphates	0.017684	0.011697
alcohol	1.574064	1.210951

Table 5- Covariance matrix for quality =0

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
fixed acidity	0.739786	-0.001064	0.032992	0.207067	-0.000205	-0.943361	2.253866	0.000524	-0.051838	0.001115	-0.037553
volatile acidity	-0.001064	0.010464	-0.001873	0.041570	0.000247	-0.155567	0.513420	0.000024	-0.000707	-0.000305	-0.001738
citric acid	0.032992	-0.001873	0.016889	0.066537	0.000338	0.206014	0.680726	0.000057	-0.003441	0.001205	-0.008349
residual sugar	0.207067	0.041570	0.066537	27.301127	0.005776	30.673526	87.988364	0.012907	-0.113968	0.005439	-2.511217
chlorides	-0.000205	0.000247	0.000338	0.005776	0.000554	0.038477	0.161859	0.000013	-0.000254	0.000068	-0.007840
free sulfur dioxide	-0.943361	-0.155567	0.206014	30.673526	0.038477	316.406723	496.111385	0.016813	-0.006801	0.067361	-5.540173
total sulfur dioxide	2.253866	0.513420	0.680726	87.988364	0.161859	496.111385	1948.816558	0.064413	0.194902	0.871520	-20.693733
density	0.000524	0.000024	0.000057	0.012907	0.000013	0.016813	0.064413	0.000008	-0.000016	0.000036	-0.002341
pH	-0.051838	-0.000707	-0.003441	-0.113968	-0.000254	-0.006801	0.194902	-0.000016	0.022027	0.002171	0.013291
sulphates	0.001115	-0.000305	0.001205	0.005439	0.000068	0.067361	0.871520	0.000036	0.002171	0.011700	-0.004385
alcohol	-0.037553	-0.001738	-0.008349	-2.511217	-0.007840	-5.540173	-20.693733	-0.002341	0.013291	-0.004385	1.211267

Table 6- Covariance matrix for quality =1

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
fixed acidity	0.591050	-0.007206	0.015680	0.824538	0.001451	0.014381	4.764318	0.000932	-0.058659	-0.009971	-0.295082
volatile acidity	-0.007206	0.008854	-0.001780	-0.017252	-0.000301	-0.217825	-0.308155	-0.000077	0.000759	-0.000625	0.059990
citric acid	0.015680	-0.001780	0.006446	0.014592	0.000094	0.141068	0.278093	0.000029	-0.001091	-0.000278	-0.012533
residual sugar	0.824538	-0.017252	0.014592	18.411355	0.013530	7.075646	62.403926	0.009755	-0.235830	-0.076221	-2.606353
chlorides	0.001451	-0.000301	0.000094	0.013530	0.000124	0.027436	0.140047	0.000015	-0.000199	0.000047	-0.007606
free sulfur dioxide	0.014381	-0.217825	0.141068	7.075646	0.027436	190.361237	246.919148	0.006724	0.043305	0.297391	-3.259269
total sulfur dioxide	4.764318	-0.308155	0.278093	62.403926	0.140047	246.919148	1070.915700	0.051000	-0.187214	0.032694	-18.474242
density	0.000932	-0.000077	0.000029	0.009755	0.000015	0.006724	0.051000	0.000008	-0.000082	0.000007	-0.002935
pH	-0.058659	0.000759	-0.001091	-0.235830	-0.000199	0.043305	-0.187214	-0.000082	0.024707	0.004209	0.025242
sulphates	-0.009971	-0.000625	-0.000278	-0.076221	0.000047	0.297391	0.032694	0.000007	0.004209	0.017701	-0.007292
alcohol	-0.295082	0.059990	-0.012533	-2.606353	-0.007606	-3.259269	-18.474242	-0.002935	0.025242	-0.007292	1.575551

Figure 1.2- Scatterplot of all predictor variables

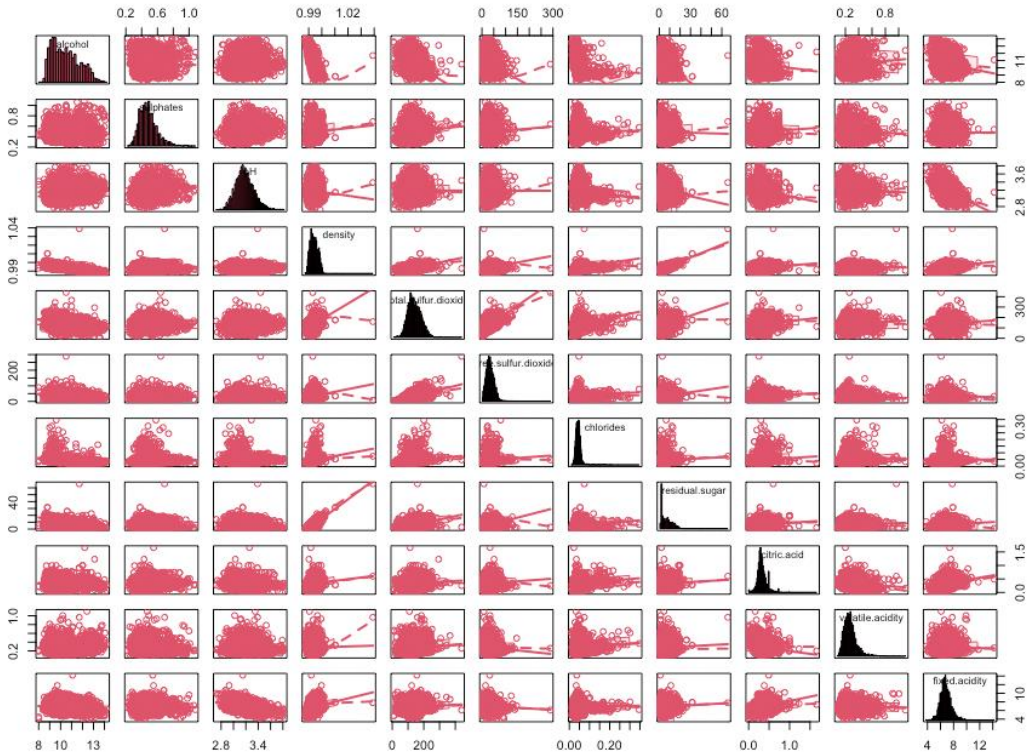


Figure 1.3- Coefficient matrix visualisation

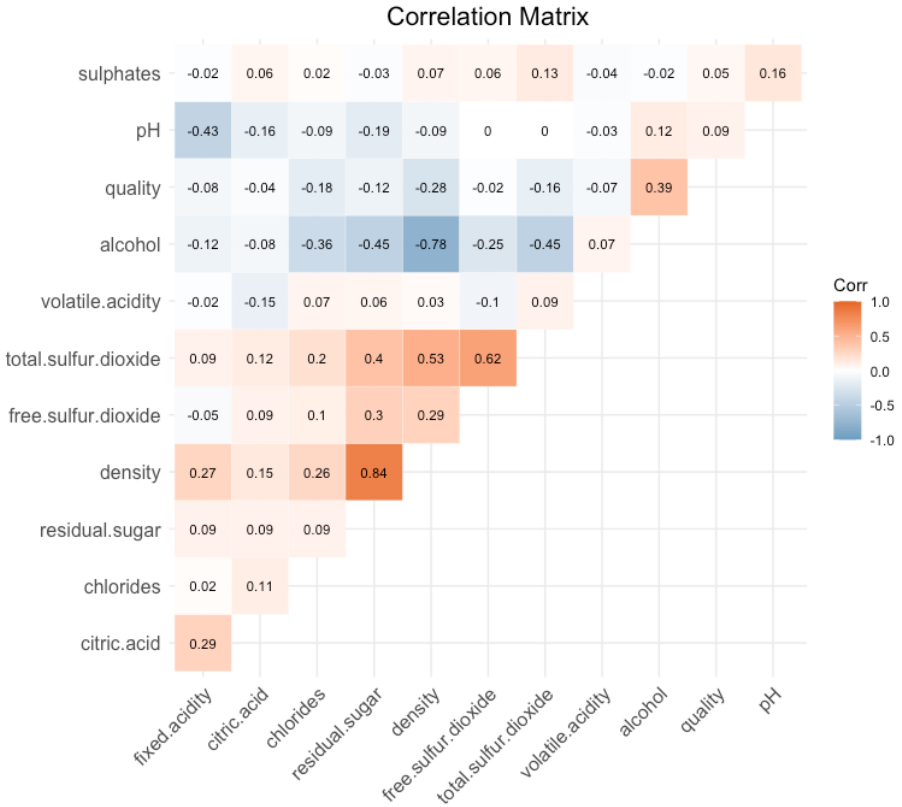


Figure 1.4- Paired plots of the distributions and correlation coefficients of the 11 variables

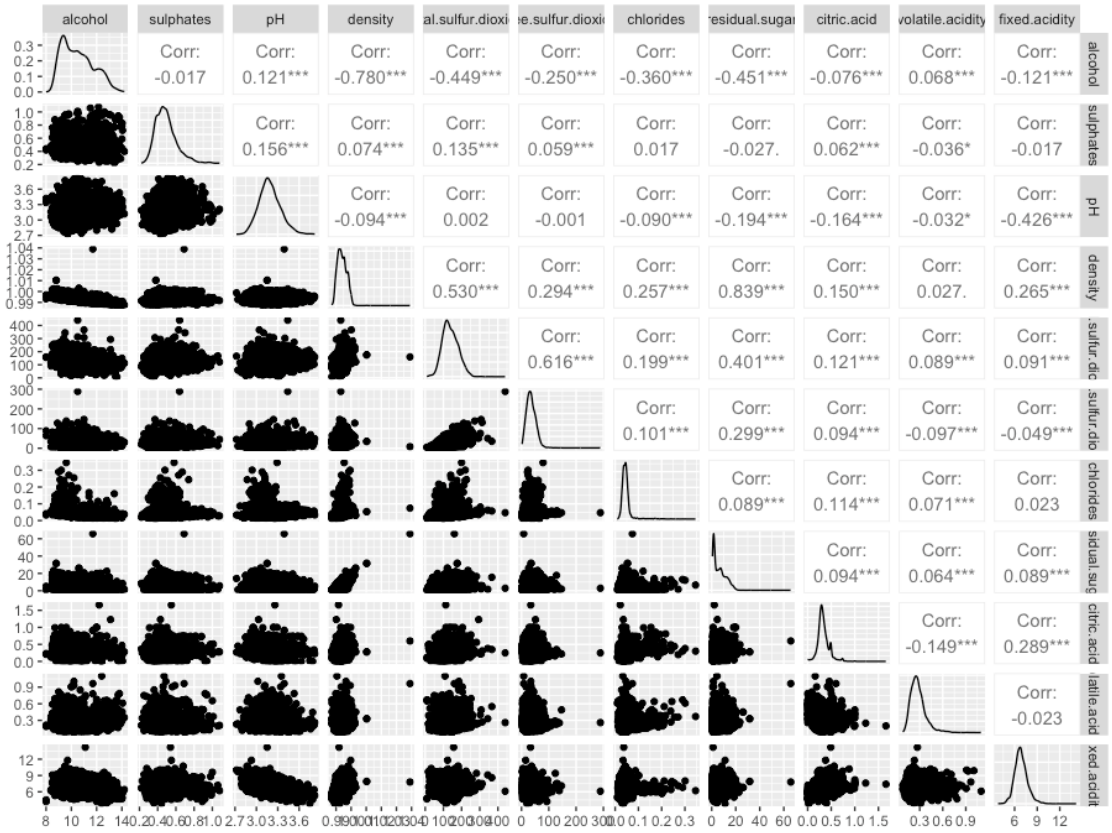
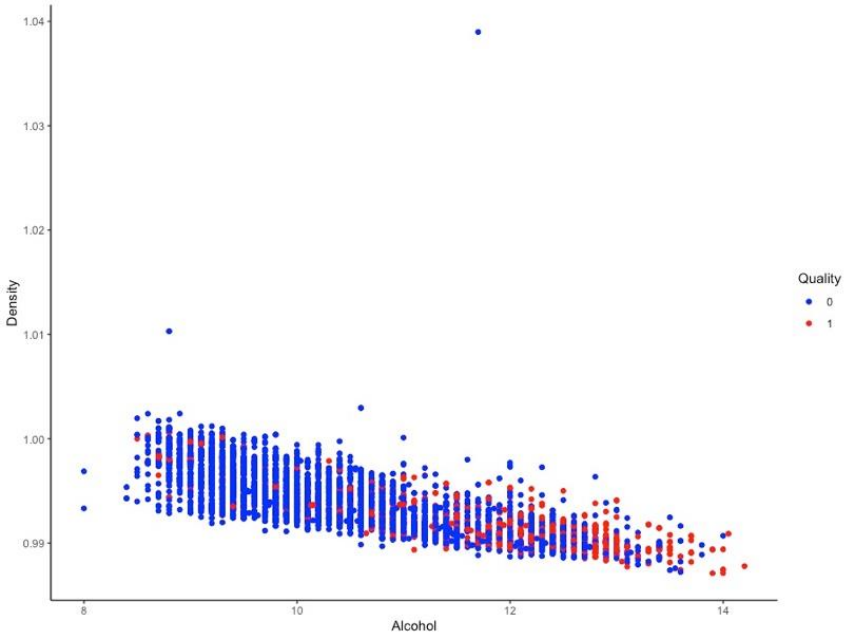


Figure 1.5- Scatter plot of alcohol and density (blue: bad wine, red: good wine)



Appendix 2

Figure 2.1- Single tree

```
Classification tree:
tree(formula = quality ~ ., data = data_train)
Variables actually used in tree construction:
[1] "alcohol"          "volatile.acidity" "chlorides"        "pH"
Number of terminal nodes:  9
Residual mean deviance:  0.7822 = 1908 / 2440
Misclassification error rate: 0.1829 = 448 / 2449
```

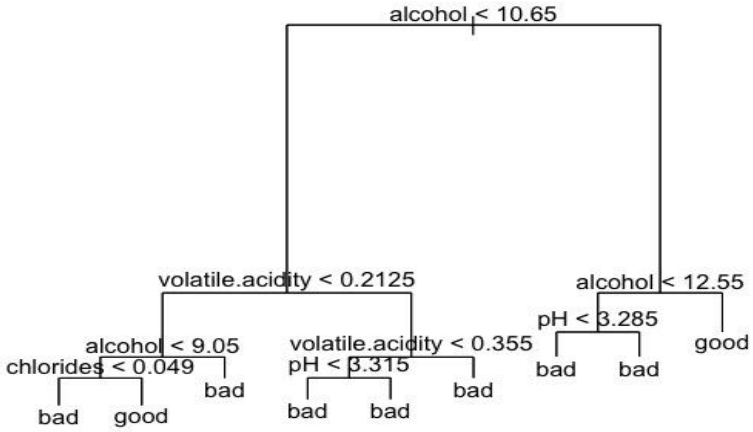


Figure 2.2- Graph of classification tree after cross-validation pruning

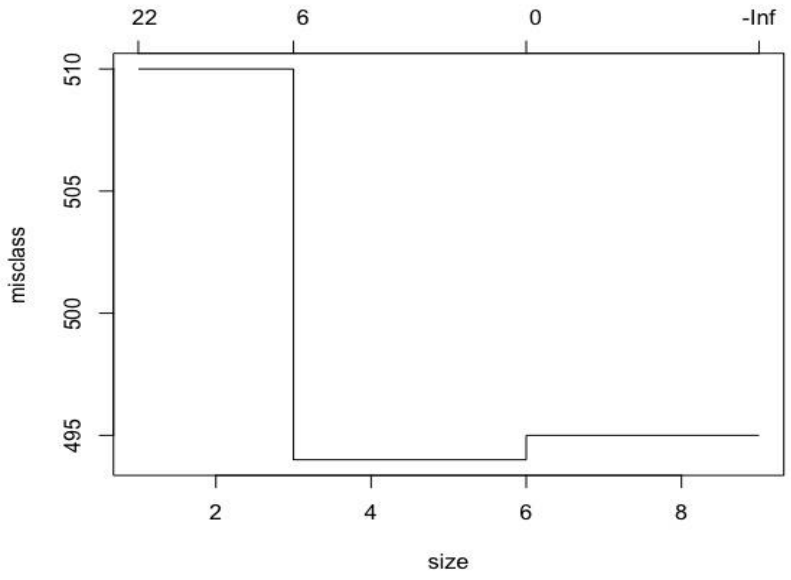


Figure 2.3- Single tree after pruning

```
Classification tree:
snip.tree(tree = tree.wine, nodes = c(6L, 2L))
Variables actually used in tree construction:
[1] "alcohol"
Number of terminal nodes:  3
Residual mean deviance:  0.8703 = 2129 / 2446
Misclassification error rate: 0.1903 = 466 / 2449
```

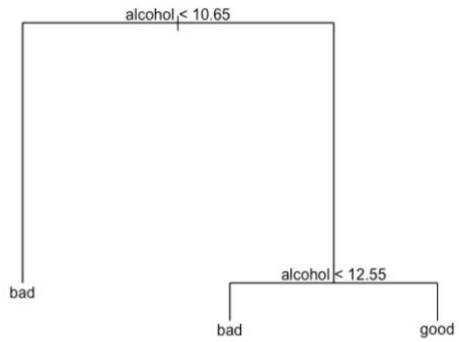


Figure 2.4- Three times bagging tree

```
Call:
 randomForest(formula = quality ~ ., data = data_train, mtry = 11,      ntree = 10, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 10
No. of variables tried at each split: 11

        OOB estimate of  error rate: 18.24%
Confusion matrix:
      bad good class.error
bad  1728  191  0.09953101
good  251  253  0.49801587
```

```
Call:
 randomForest(formula = quality ~ ., data = data_train, mtry = 11,      ntree = 100, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 100
No. of variables tried at each split: 11

        OOB estimate of  error rate: 13.96%
Confusion matrix:
      bad good class.error
bad  1842   97  0.05002579
good  245  265  0.48039216
```

```
Call:
 randomForest(formula = quality ~ ., data = data_train, mtry = 11,      ntree = 1000, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 1000
No. of variables tried at each split: 11

        OOB estimate of  error rate: 13.6%
Confusion matrix:
      bad good class.error
bad  1851   88  0.04538422
good  245  265  0.48039216
```

Figure 2.5- Check the importance of the features of the bagging tree at B=1000

	bad	good	MeanDecreaseAccuracy	MeanDecreaseGini
fixed.acidity	32.39242	34.67238	45.04222	56.54260
volatile.acidity	39.54924	85.74921	80.08097	70.99669
citric.acid	18.42357	43.00040	41.21592	54.48774
residual.sugar	41.98789	23.97258	52.15437	57.81136
chlorides	32.92944	52.43365	60.25139	65.80654
free.sulfur.dioxide	40.63080	27.17461	50.21320	62.54257
total.sulfur.dioxide	22.25062	40.23667	46.31414	60.64268
density	28.93593	33.65498	41.00443	62.34085
pH	32.92549	51.32178	57.51646	72.09893
sulphates	29.26266	37.36690	46.17562	61.59395
alcohol	49.80827	108.56536	113.30142	182.22565

Figure 2.6- A bar chart of the importance measure for each feature of the bagging tree at B=1000

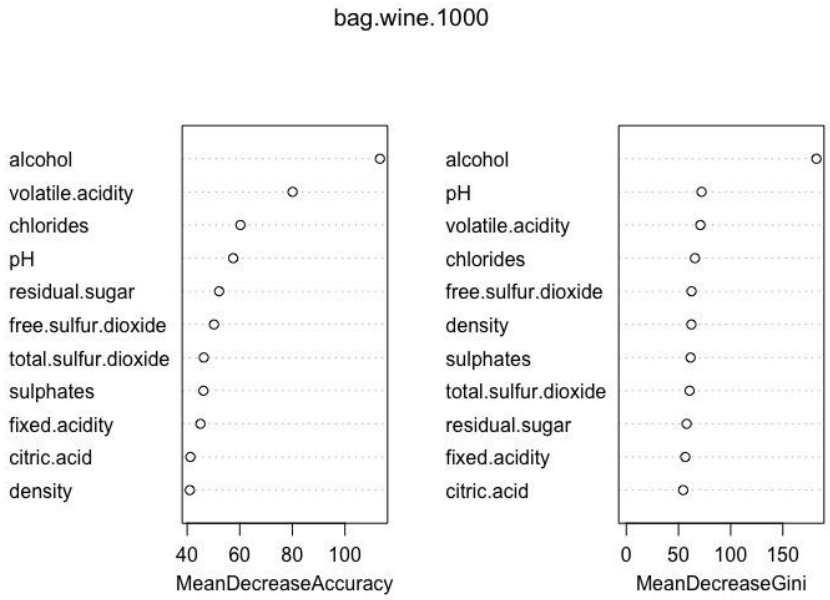


Figure 2.7- Base decision tree in bagging method

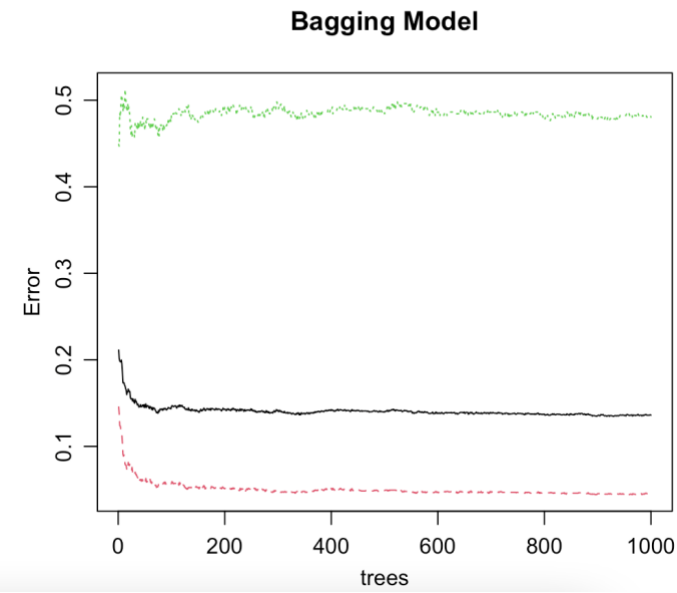


Figure 2.8- Random forest tree

```
Call:
 randomForest(formula = quality ~ ., data = data_train, mtry = 4,      ntree = 1000, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 1000
No. of variables tried at each split: 4

        OOB estimate of  error rate: 13.6%
Confusion matrix:
      bad good class.error
bad  1851   88  0.04538422
good  245  265  0.48039216
```

Figure 2.9- Check the importance of the features of the random forest tree

	bad	good	MeanDecreaseAccuracy	MeanDecreaseGini
fixed.acidity	25.76764	32.93359	40.39498	53.59947
volatile.acidity	33.89651	67.49646	63.76918	67.96048
citric.acid	22.35232	43.97946	42.97482	54.67040
residual.sugar	40.53661	24.15039	49.94224	67.23923
chlorides	31.36616	50.91386	57.90186	73.52342
free.sulfur.dioxide	40.53465	28.83104	49.41243	62.34689
total.sulfur.dioxide	24.51445	38.30657	40.33163	63.39945
density	35.62784	41.93907	50.08132	97.58938
pH	27.00968	51.47907	52.83549	75.60727
sulphates	27.46878	41.79000	47.48130	63.22725
alcohol	39.69571	73.93411	83.17372	129.00783

Figure 2.10- A bar chart of the importance measure for each feature of the random forest tree

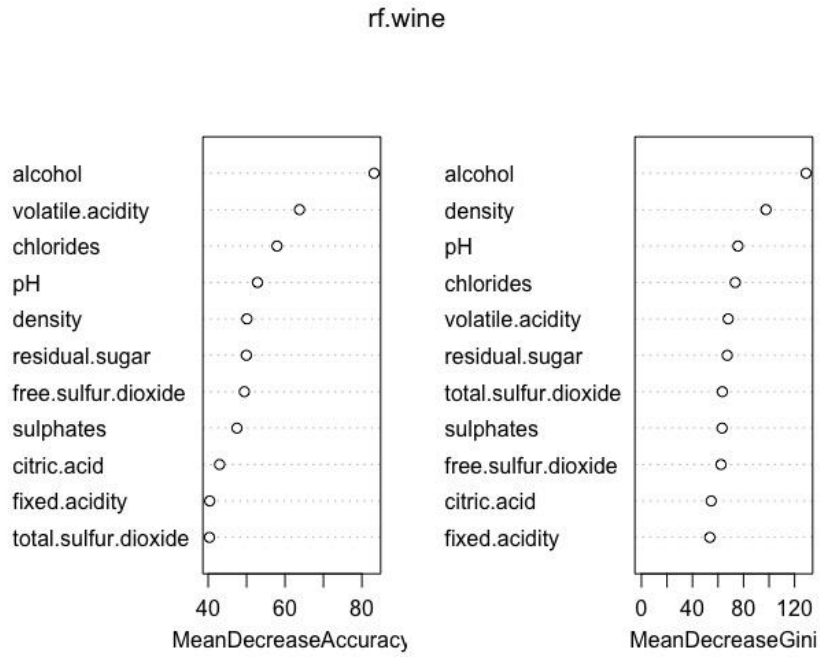
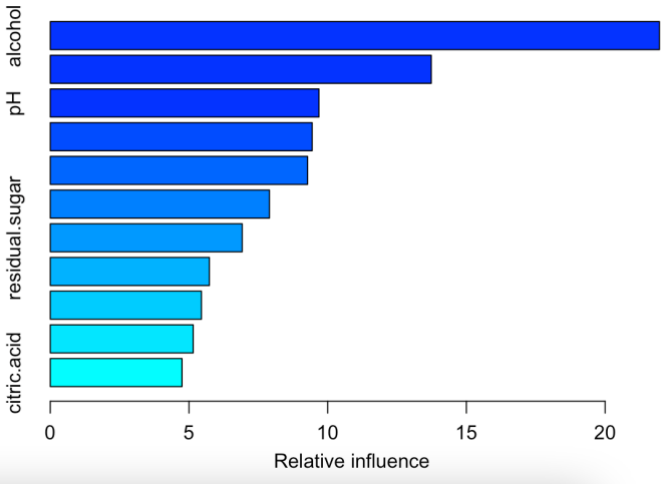


Figure 2.11- Boosting tree

	var	rel.inf
alcohol	alcohol	21.960467
density	density	13.734989
pH	pH	9.684977
chlorides	chlorides	9.440852
volatile.acidity	volatile.acidity	9.274792
sulphates	sulphates	7.901031
residual.sugar	residual.sugar	6.918866
free.sulfur.dioxide	free.sulfur.dioxide	5.734695
total.sulfur.dioxide	total.sulfur.dioxide	5.449027
fixed.acidity	fixed.acidity	5.151630
citric.acid	citric.acid	4.748674



Appendix 3 R Code

```
library(skimr)
library(dplyr)
library(ggcorrplot)
library(ggtext)

setwd("~/Desktop")
wine <- read.csv("winequality-red.csv")
d1<- read.table( "winequality-red.csv" , sep = ";" , header = TRUE )
d2<- read.table("winequality-white.csv", sep = ";" , header = TRUE)

#the red and white variants
#the correlations between the predictor variables vary greatly
cor(d1)
cor(d2)

#d1$colour="red"
#d2$colour="white"
#d4=rbind(d1,d2)
#d4$colour=c(rep("red",dim(d1)[1]),rep("white",dim(d2)[1]))
#d4$colour <- as.factor(d4$colour)
skim(d2)
colSums(is.na(d2)) #no NA value
sapply(d2, length)
str(d2)

table(d2$quality)
(1457+2198+880)/(20+163+1457+2198+880+175+5)
barplot(table(d2$quality), main = "Wine Quality Distribution", xlab = "Quality", ylab = "Frequency")

d2$quality <- as.numeric(ifelse(d2$quality >= 7, 1, 0))
table(d2$quality)

cor_matrix <- cor(mutate(d2))
cor_matrix

#d2_0 <- d2[d2$quality == 0]
#d2_1 <- d2[d2$quality == 1]
#var_matrix_0 <- var(d2_0)
#cov_matrix_0 <- cov(d2_0)
#var_matrix_1 <- var(d2_1)
#cov_matrix_1 <- cov(d2_1)
```

```
library(car)
scatterplotMatrix(d2[,11:1],
                  col=d2[,12]+2,
                  diagonal=list(method ="histogram"))

library(corrplot)
corrplot(cor(d2),method= "number",diag = FALSE)

ggcorrplot(cor_matrix, hc.order = TRUE, type = "upper",
           outline.color = "white", colors = c("#6D9EC1", "white", "#E46726"),
           lab = TRUE, lab_size = 3, lab_col = "black") +
  ggtitle("Correlation Matrix") +
  theme(plot.title = element_text(size = 16, hjust = 0.5),
        axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1))

library(GGally)
ggpairs(d2[,11:1])

library(ggplot2)
ggplot(data = d2, aes(x = alcohol, y = density, color = as.factor(quality))) +
  geom_point() +
  labs(x = "Alcohol", y = "Density", color = "Quality") +
  scale_color_manual(values = c("blue", "red")) +
  theme_classic()

#Classification trees
par(mfrow=c(1,1))
library(tree)
set.seed(180)
d2=na.omit(d2)
d2$quality <- as.factor(ifelse(d2$quality >= 7, 'good', 'bad'))
table(d2$quality)
str(d2$quality)
#good not good
#1060        3838

train=sample(1:nrow(d2),floor(nrow(d2)/2))
data_train=d2[train,]
data_test=d2[-train,]

tree.wine=tree(quality~.,data_train)
summary(tree.wine)
plot(tree.wine);text(tree.wine,pretty=0)
```

```
tree.pred=predict(tree.wine,data_test,type='class')
class_table=table(tree.pred,data_test$quality)
success_rate=(class_table[1,1]+class_table[2,2])/sum(class_table)
success_rate #0.7966517

cv.wine=cv.tree(tree.wine,FUN=prune.misclass)
cv.wine
plot(cv.wine)

prune.wine=prune.misclass(tree.wine, best=3)
prune.wine
summary(prune.wine)
plot(prune.wine);text(prune.wine,pretty=0)

tree.pred=predict(prune.wine,data_test,type='class')
class_table=table(tree.pred,data_test$quality)
success_rate=(class_table[1,1]+class_table[2,2])/sum(class_table)
success_rate    #0.7937934


library(randomForest)
set.seed(472)
bag.wine=randomForest(quality~.,data=data_train,mtry=11,ntree=10,importance=TRUE)
bag.wine
summary(bag.wine)
pred.bag=predict(bag.wine,newdata=data_test,type='class')
class_table=table(pred.bag,data_test$quality)
success_rate_bagging=(class_table[1,1]+class_table[2,2])/sum(class_table)
success_rate_bagging #0.8370764

bag.wine.100=randomForest(quality~.,data=data_train,mtry=11,ntree=100,importance=TRUE)
bag.wine.100
pred.bag.100=predict(bag.wine.100,newdata=data_test,type='class')
class_table=table(pred.bag.100,data_test$quality)
success_rate_bagging=(class_table[1,1]+class_table[2,2])/sum(class_table)
success_rate_bagging #0.8562679

bag.wine.1000=randomForest(quality~.,data=data_train,mtry=11,ntree=1000,importance=TRUE)
bag.wine.1000
pred.bag.1000=predict(bag.wine.1000,newdata=data_test,type='class')
class_table=table(pred.bag.1000,data_test$quality)
success_rate_bagging=(class_table[1,1]+class_table[2,2])/sum(class_table)
success_rate_bagging #0.8562679
```

```
# check the importance feature
importance(bag.wine.1000)
varImpPlot(bag.wine.1000)

# check the Gini coefficient
var.gini <- function(x) {
   1 - sum((x/sum(x))^2)
}
bag.gini <- apply(bag.wine.1000$confusion, 1, var.gini)
names(bag.gini) <- rownames(bag.wine.1000$confusion)
bag.gini

randomForest::getTree(bag.wine.1000, k = 1, labelVar = TRUE)
plot(bag.wine.1000, main = "Bagging Model")

rf.wine=randomForest(quality~.,data=data_train,mtry=4,ntree=1000,importance=TRUE)
rf.wine #OOB estimate of    error rate: 13.56%
pred.rf=predict(rf.wine,newdata=data_test,type='class')
class_table=table(pred.rf,data_test$quality)
success_rate_rf=(class_table[1,1]+class_table[2,2])/sum(class_table)
success_rate_rf #0.8534096

par(mfrow=c(1,1))
importance(rf.wine)
varImpPlot(rf.wine)

randomForest::getTree(rf.wine, k = 1, labelVar = TRUE)
plot(rf.wine, main = "Random Forest Model")

library(gbm)
boost.wine=gbm(unclass(quality)-1~.,data=data_train,distribution = 'bernoulli',n.trees=1000,interaction.depth =2)
boost.wine
summary(boost.wine)

pred.boost=predict(boost.wine,newdata=data_test,n.trees=1000,type='response')
quality_pred=ifelse(pred.boost<=0.5,'bad','good')

class_table=table(quality_pred,data_test$quality)
success_rate_boost=(class_table[1,1]+class_table[2,2])/sum(class_table)
success_rate_boost #0.8223765

gbm::plot.gbm(boost.wine, i.trees = 1, main = "Boosting Model")
```

Appendix 4 Python Code

In [45]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Make NumPy printouts easier to read.
np.set_printoptions(precision=3, suppress=True)
```

In [46]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

print(tf.__version__)
```

```
2.11.0
```

In [47]:
```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv'
column_names = ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides',
                'free sulfur dioxide', 'total sulfur dioxide', 'density','pH','sulphates','alcohol','quality']

raw_dataset = pd.read_csv(url, na_values='?', comment='\t',sep=';', skipinitialspace=True)
```

In [48]:
```
dataset = raw_dataset.copy()
dataset.tail()
```

Out[48]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
4893	6.2	0.21	0.29	1.6	0.039	24.0	92.0	0.99114	3.27	0.50	11.2	6
4894	6.6	0.32	0.36	8.0	0.047	57.0	168.0	0.99490	3.15	0.46	9.6	5
4895	6.5	0.24	0.19	1.2	0.041	30.0	111.0	0.99254	2.99	0.46	9.4	6
4896	5.5	0.29	0.30	1.1	0.022	20.0	110.0	0.98869	3.34	0.38	12.8	7
4897	6.0	0.21	0.38	0.8	0.020	22.0	98.0	0.98941	3.26	0.32	11.8	6

Clean the data

In [49]:
```
dataset.isna().sum()
```

Out[49]:
```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

In [50]:
```
dataset = dataset.dropna()
```

In [51]:
```
dataset = dataset.apply(pd.to_numeric, errors='coerce')
print(dataset.dtypes)
```

```
fixed acidity           float64
volatile acidity        float64
citric acid             float64
residual sugar          float64
chlorides               float64
free sulfur dioxide     float64
total sulfur dioxide    float64
density                 float64
pH                      float64
sulphates               float64
alcohol                 float64
quality                   int64
dtype: object
```

In [52]:
```
# Convert 'quality' variable to binary class
dataset['quality'] = dataset['quality'].apply(lambda x: 1 if x >= 7 else 0)
```

In [53]:
```
# Calculating the correlation coefficient matrix
cor_matrix = dataset.corr()
cor_matrix
```

Out[53]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
fixed acidity	1.000000	-0.022697	0.289181	0.089021	0.023086	-0.049396	0.091070	0.265331	-0.425858	-0.017143	-0.120881	-0.080748
volatile acidity	-0.022697	1.000000	-0.149472	0.064286	0.070512	-0.097012	0.089261	0.027114	-0.031915	-0.035728	0.067718	-0.067225
citric acid	0.289181	-0.149472	1.000000	0.094212	0.114364	0.094077	0.121131	0.149503	-0.163748	0.062331	-0.075729	-0.035330
residual sugar	0.089021	0.064286	0.094212	1.000000	0.088685	0.299098	0.401439	0.838966	-0.194133	-0.026664	-0.450631	-0.117085
chlorides	0.023086	0.070512	0.114364	0.088685	1.000000	0.101392	0.198910	0.257211	-0.090439	0.016763	-0.360189	-0.183118
free sulfur dioxide	-0.049396	-0.097012	0.094077	0.299098	0.101392	1.000000	0.615501	0.294210	-0.000618	0.059217	-0.250104	-0.023413
total sulfur dioxide	0.091070	0.089261	0.121131	0.401439	0.198910	0.615501	1.000000	0.529881	0.002321	0.134562	-0.448892	-0.162202
density	0.265331	0.027114	0.149503	0.838966	0.257211	0.294210	0.529881	1.000000	-0.093591	0.074493	-0.780138	-0.283871
pH	-0.425858	-0.031915	-0.163748	-0.194133	-0.090439	-0.000618	0.002321	-0.093591	1.000000	0.155951	0.121432	0.093510
sulphates	-0.017143	-0.035728	0.062331	-0.026664	0.016763	0.059217	0.134562	0.074493	0.155951	1.000000	-0.017433	0.047410
alcohol	-0.120881	0.067718	-0.075729	-0.450631	-0.360189	-0.250104	-0.448892	-0.780138	0.121432	-0.017433	1.000000	0.385132
quality	-0.080748	-0.067225	-0.035330	-0.117085	-0.183118	-0.023413	-0.162202	-0.283871	0.093510	0.047410	0.385132	1.000000

In [54]:
```
# Splitting the data set into two subsets
d2_0 = dataset[dataset.quality == 0].iloc[:, :-1]
d2_1 = dataset[dataset.quality == 1].iloc[:, :-1]
```

In [55]:
```
# Calculate the variance matrix and covariance matrix for each subset
var_matrix_0 = np.var(d2_0, axis=0)
var_matrix_1 = np.var(d2_1, axis=0)

cov_matrix_0 = np.cov(d2_0.T)
cov_matrix_1 = np.cov(d2_1.T)
```

In [56]:
```
var_matrix_0,var_matrix_1
```

Out[56]:
```
(fixed acidity              0.739594
 volatile acidity           0.010461
 citric acid                0.016885
 residual sugar            27.294014
 chlorides                  0.000554
 free sulfur dioxide      316.324283
 total sulfur dioxide    1948.308789
 density                    0.000008
 pH                         0.022021
 sulphates                  0.011697
 alcohol                    1.210951
 dtype: float64,
 fixed acidity              0.590493
 volatile acidity           0.008846
 citric acid                0.006440
 residual sugar            18.393985
 chlorides                  0.000124
 free sulfur dioxide      190.181651
 total sulfur dioxide    1069.905402
 density                    0.000008
 pH                         0.024684
 sulphates                  0.017684
 alcohol                    1.574064
 dtype: float64)
```

In [57]:
```
cov_df_0=pd.DataFrame(cov_matrix_0, columns=d2_0.columns, index=d2_0.columns)
cov_df_0
```

Out[57]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
fixed acidity	0.739786	-0.001064	0.032992	0.207067	-0.000205	-0.943361	2.253866	0.000524	-0.051838	0.001115	-0.037553
volatile acidity	-0.001064	0.010464	-0.001873	0.041570	0.000247	-0.155567	0.513420	0.000024	-0.000707	-0.000305	-0.001738
citric acid	0.032992	-0.001873	0.016889	0.066537	0.000338	0.206014	0.680726	0.000057	-0.003441	0.001205	-0.008349
residual sugar	0.207067	0.041570	0.066537	27.301127	0.005776	30.673526	87.988364	0.012907	-0.113968	0.005439	-2.511217
chlorides	-0.000205	0.000247	0.000338	0.005776	0.000554	0.038477	0.161859	0.000013	-0.000254	0.000068	-0.007840
free sulfur dioxide	-0.943361	-0.155567	0.206014	30.673526	0.038477	316.406723	496.111385	0.016813	-0.006801	0.067361	-5.540173
total sulfur dioxide	2.253866	0.513420	0.680726	87.988364	0.161859	496.111385	1948.816558	0.064413	0.194902	0.871520	-20.693733
density	0.000524	0.000024	0.000057	0.012907	0.000013	0.016813	0.064413	0.000008	-0.000016	0.000036	-0.002341
pH	-0.051838	-0.000707	-0.003441	-0.113968	-0.000254	-0.006801	0.194902	-0.000016	0.022027	0.002171	0.013291
sulphates	0.001115	-0.000305	0.001205	0.005439	0.000068	0.067361	0.871520	0.000036	0.002171	0.011700	-0.004385
alcohol	-0.037553	-0.001738	-0.008349	-2.511217	-0.007840	-5.540173	-20.693733	-0.002341	0.013291	-0.004385	1.211267

In [58]:
```
cov_df_1=pd.DataFrame(cov_matrix_1, columns=d2_0.columns, index=d2_0.columns)
cov_df_1
```

Out[58]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
fixed acidity	0.591050	-0.007206	0.015680	0.824538	0.001451	0.014381	4.764318	0.000932	-0.058659	-0.009971	-0.295082
volatile acidity	-0.007206	0.008854	-0.001780	-0.017252	-0.000301	-0.217825	-0.308155	-0.000077	0.000759	-0.000625	0.059990
citric acid	0.015680	-0.001780	0.006446	0.014592	0.000094	0.141068	0.278093	0.000029	-0.001091	-0.000278	-0.012533
residual sugar	0.824538	-0.017252	0.014592	18.411355	0.013530	7.075646	62.403926	0.009755	-0.235830	-0.076221	-2.606353
chlorides	0.001451	-0.000301	0.000094	0.013530	0.000124	0.027436	0.140047	0.000015	-0.000199	0.000047	-0.007606
free sulfur dioxide	0.014381	-0.217825	0.141068	7.075646	0.027436	190.361237	246.919148	0.006724	0.043305	0.297391	-3.259269
total sulfur dioxide	4.764318	-0.308155	0.278093	62.403926	0.140047	246.919148	1070.915700	0.051000	-0.187214	0.032694	-18.474242
density	0.000932	-0.000077	0.000029	0.009755	0.000015	0.006724	0.051000	0.000008	-0.000082	0.000007	-0.002935
pH	-0.058659	0.000759	-0.001091	-0.235830	-0.000199	0.043305	-0.187214	-0.000082	0.024707	0.004209	0.025242
sulphates	-0.009971	-0.000625	-0.000278	-0.076221	0.000047	0.297391	0.032694	0.000007	0.004209	0.017701	-0.007292
alcohol	-0.295082	0.059990	-0.012533	-2.606353	-0.007606	-3.259269	-18.474242	-0.002935	0.025242	-0.007292	1.575551

In [59]:
```
# Dividing the training and test sets
train_dataset = dataset.sample(frac=0.8, random_state=0)
test_dataset = dataset.drop(train_dataset.index)
```

In [60]:
```
# Distinguishing features and labels
train_features = train_dataset.drop('quality', axis=1)
test_features = test_dataset.drop('quality', axis=1)

train_labels = train_dataset['quality']
test_labels = test_dataset['quality']
```

Normalization

In [61]:
```
dataset.describe().transpose()
```

Out[61]:

	count	mean	std	min	25%	50%	75%	max
fixed acidity	4898.0	6.854788	0.843868	3.80000	6.300000	6.80000	7.3000	14.20000
volatile acidity	4898.0	0.278241	0.100795	0.08000	0.210000	0.26000	0.3200	1.10000
citric acid	4898.0	0.334192	0.121020	0.00000	0.270000	0.32000	0.3900	1.66000
residual sugar	4898.0	6.391415	5.072058	0.60000	1.700000	5.20000	9.9000	65.80000
chlorides	4898.0	0.045772	0.021848	0.00900	0.036000	0.04300	0.0500	0.34600
free sulfur dioxide	4898.0	35.308085	17.007137	2.00000	23.000000	34.00000	46.0000	289.00000
total sulfur dioxide	4898.0	138.360657	42.498065	9.00000	108.000000	134.00000	167.0000	440.00000
density	4898.0	0.994027	0.002991	0.98711	0.991723	0.99374	0.9961	1.03898
pH	4898.0	3.188267	0.151001	2.72000	3.090000	3.18000	3.2800	3.82000
sulphates	4898.0	0.489847	0.114126	0.22000	0.410000	0.47000	0.5500	1.08000
alcohol	4898.0	10.514267	1.230621	8.00000	9.500000	10.40000	11.4000	14.20000
quality	4898.0	0.216415	0.411842	0.00000	0.000000	0.00000	0.0000	1.00000

In [62]:
```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4898 entries, 0 to 4897
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         4898 non-null   float64
 1   volatile acidity      4898 non-null   float64
 2   citric acid           4898 non-null   float64
 3   residual sugar        4898 non-null   float64
 4   chlorides             4898 non-null   float64
 5   free sulfur dioxide   4898 non-null   float64
 6   total sulfur dioxide  4898 non-null   float64
 7   density               4898 non-null   float64
 8   pH                    4898 non-null   float64
 9   sulphates             4898 non-null   float64
 10  alcohol               4898 non-null   float64
 11  quality               4898 non-null   int64  
dtypes: float64(11), int64(1)
memory usage: 459.3 KB
```

In [63]:
```
datavalues=train_features.values
data_train_features_norm=np.copy(datavalues)
data_train_features_norm[:,0:11]=(datavalues[:,0:11]-np.mean(datavalues[:,0:11],axis=0))/np.std(datavalues[:,0:11],axis=0)
datavalues=test_features.values
data_test_features_norm=np.copy(datavalues)
data_test_features_norm[:,0:11]=(datavalues[:,0:11]-np.mean(datavalues[:,0:11],axis=0))/np.std(datavalues[:,0:11],axis=0)
```

In [64]:
```
data_train_features_norm
```

Out[64]:
```
array([[ 0.516,  0.417,  0.125, ...,  0.348, -0.605,  0.149],
       [ 0.165,  0.317, -0.612, ..., -0.38 , -0.255, -0.583],
       [ 0.867, -1.381,  3.318, ..., -0.777, -0.781,  0.23 ],
       ...,
       [-0.069, -0.682, -0.53 , ...,  0.745, -1.131, -1.477],
       [ 0.399,  1.215, -0.775, ...,  0.083,  0.271, -1.07 ],
       [-0.419,  1.615,  1.271, ..., -0.512, -1.043, -1.477]])
```

In [65]:
```
alcohol_train_norm = data_train_features_norm[:,10]
alcohol_test_norm = data_test_features_norm[:,10]
train_label_values=train_labels.values
test_label_values=test_labels.values
```

In [66]:
```
#Classification problems with dnn models
num_classes=2
batch_size = 128
epochs = 250
```

In [67]:
```
#model1:four hidden layer+softmax
model_DNN_1 = keras.Sequential(
    [
        keras.Input(shape=11),
        layers.Flatten(),
        layers.Dense(32, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model_DNN_1.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_4 (Flatten)         (None, 11)                0

 dense_30 (Dense)            (None, 32)                384

 dense_31 (Dense)            (None, 32)                1056

 dense_32 (Dense)            (None, 32)                1056

 dense_33 (Dense)            (None, 32)                1056

 dense_34 (Dense)            (None, 2)                 66

=================================================================
Total params: 3,618
Trainable params: 3,618
Non-trainable params: 0
_________________________________________________________________
```

In [68]:
```
model_DNN_1.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

model_DNN_1.fit(data_train_features_norm, keras.utils.to_categorical(train_label_values,num_classes),
            batch_size=batch_size, epochs=epochs, validation_split=0.1)
#Overfitting, the training and validation sets are very different
```

In [69]:
```
# model_DNN.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
# model_DNN.fit(data_train_features_norm, train_label_values, batch_size=batch_size, epochs=epochs, validation_split=0.1)
```

In [70]:
```
score_1 = model_DNN_1.evaluate(data_test_features_norm, keras.utils.to_categorical(test_label_values, num_classes), verbose=0)
print("Test loss:", score_1[0])
print("Test accuracy:", score_1[1])
```

```
Test loss: 1.3344687223434448
Test accuracy: 0.8479591608047485
```

In [71]:
```
#Four hidden layers + sigmoid
model_DNN_2 = keras.Sequential(
    [
        keras.Input(shape=11),
        layers.Flatten(),
        layers.Dense(32, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(num_classes, activation="sigmoid"),
    ]
)

model_DNN_2.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_5 (Flatten)         (None, 11)                0

 dense_35 (Dense)            (None, 32)                384

 dense_36 (Dense)            (None, 32)                1056

 dense_37 (Dense)            (None, 32)                1056

 dense_38 (Dense)            (None, 32)                1056

 dense_39 (Dense)            (None, 2)                 66

=================================================================
Total params: 3,618
Trainable params: 3,618
Non-trainable params: 0
_________________________________________________________________
```

In [72]:
```
model_DNN_2.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

model_DNN_2.fit(data_train_features_norm, keras.utils.to_categorical(train_label_values,num_classes),
            batch_size=batch_size, epochs=epochs, validation_split=0.1)
#Overfitting, the training and validation sets are very different
```

In [73]:
```
score_2 = model_DNN_2.evaluate(data_test_features_norm, keras.utils.to_categorical(test_label_values, num_classes), verbose=0)
print("Test loss:", score_2[0])
print("Test accuracy:", score_2[1])
```

```
Test loss: 1.2608349323272705
Test accuracy: 0.8459183573722839
```

In [74]:
```
#Four hidden layers + softmax + canonical
from tensorflow.keras.regularizers import l1_l2
model_DNN_reg = keras.Sequential(
    [
        keras.Input(shape=11),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(64, activation='relu', kernel_regularizer=l1_l2(l1=0.01, l2=0.01)),
        layers.Dense(64, activation='relu'),
        layers.Dense(64, activation='relu', kernel_regularizer=l1_l2(l1=0.01, l2=0.01)),
        layers.Dense(64, activation='relu'),
        layers.Dense(64, activation='relu', kernel_regularizer=l1_l2(l1=0.01, l2=0.01)),
        layers.Dense(64, activation='relu'),
        layers.Dense(64, activation='relu', kernel_regularizer=l1_l2(l1=0.01, l2=0.01)),
        layers.Dense(64, activation='relu'),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model_DNN_reg.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_6 (Flatten)         (None, 11)                0

 dense_40 (Dense)            (None, 64)                768

 dense_41 (Dense)            (None, 64)                4160

 dense_42 (Dense)            (None, 64)                4160

 dense_43 (Dense)            (None, 64)                4160

 dense_44 (Dense)            (None, 64)                4160

 dense_45 (Dense)            (None, 64)                4160

 dense_46 (Dense)            (None, 64)                4160

 dense_47 (Dense)            (None, 64)                4160

 dense_48 (Dense)            (None, 64)                4160

 dense_49 (Dense)            (None, 2)                 130

=================================================================
Total params: 34,178
Trainable params: 34,178
Non-trainable params: 0
_________________________________________________________________
```

In [75]:
```
model_DNN_reg.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
model_DNN_reg.fit(data_train_features_norm, keras.utils.to_categorical(train_label_values,num_classes),
              batch_size=batch_size, epochs=epochs,
              validation_split=0.1)
```

In [76]:
```
 score_reg = model_DNN_reg.evaluate(data_test_features_norm, keras.utils.to_categorical(test_label_values, num_classes), verbose=0)
print("Test loss:", score_reg[0])
print("Test accuracy:", score_reg[1])
```

```
Test loss: 0.46744176745414734
Test accuracy: 0.8346938490867615
```

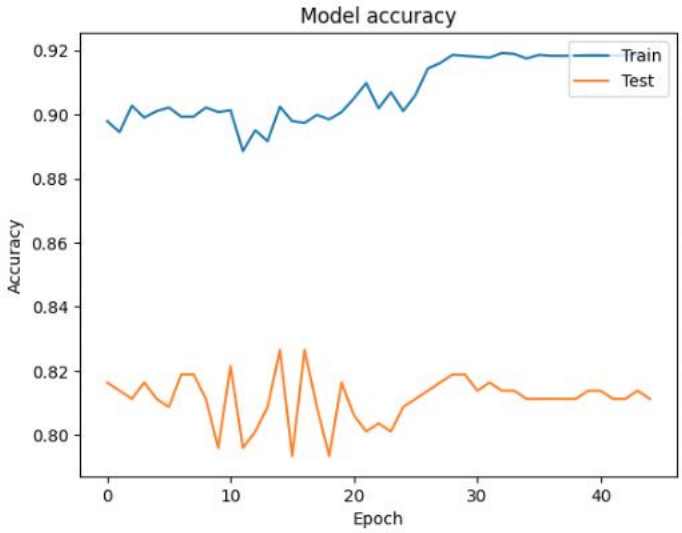
In [80]:
```
#Add callback and lr
model_DNN_reg.compile(loss="categorical_crossentropy", optimizer="Adam", metrics=["accuracy"])
early = keras.callbacks.EarlyStopping(monitor='val_accuracy', min_delta=0, patience=30,
                                      verbose=0, mode='auto', baseline=None, restore_best_weights=False)
lr = keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy', factor=0.1, patience=10,
                                       verbose=0, mode='auto', min_delta=0.0001, cooldown=0, min_lr=0)
history=model_DNN_reg.fit(data_train_features_norm.reshape(3918,11,1),
              keras.utils.to_categorical(train_label_values,num_classes),
              batch_size=batch_size,
              epochs=epochs,
              validation_split=0.1,
              callbacks=[early,lr])
```

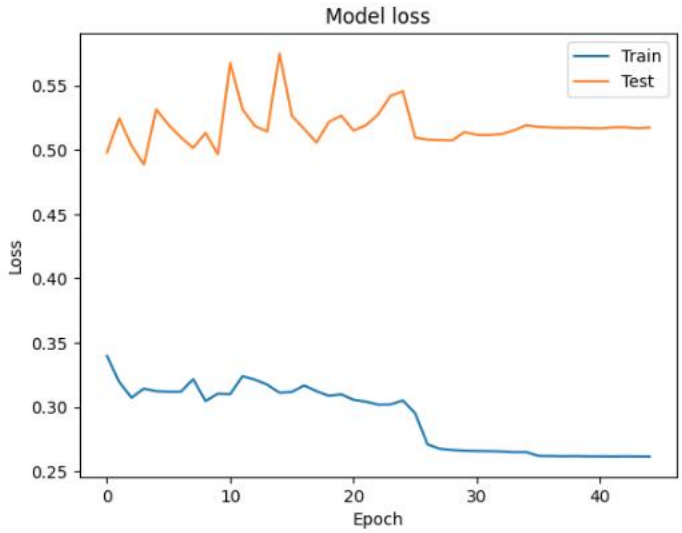
In [81]:
```
score_reg_new = model_DNN_reg.evaluate(data_test_features_norm, keras.utils.to_categorical(test_label_values, num_classes), verbose=0)
print("Test loss:", score_reg_new[0])
print("Test accuracy:", score_reg_new[1])
```

```
Test loss: 0.47235870361328125
Test accuracy: 0.8377550840377808
```

In [82]:
```
# Plotting accuracy values for training & validation
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper right')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper right')
plt.show()
```





In [83]:
```
#Changing learning rates
opt = keras.optimizers.Adam(learning_rate=0.001)
model_DNN_reg.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
```

In [90]:
```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy', factor=0.1, patience=10)
history_new=model_DNN_reg.fit(data_train_features_norm, keras.utils.to_categorical(train_label_values,num_classes),
                  batch_size=batch_size, epochs=epochs,
                  validation_split=0.1,
                  callbacks=[lr_scheduler])
```

In [91]:
```
score = model_DNN_reg.evaluate(data_test_features_norm, keras.utils.to_categorical(test_label_values, num_classes), verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 0.4792219400405884
Test accuracy: 0.8408163189888
```

In [86]:
```
#ConvolutionalNetwork
model_CNN = keras.Sequential(
    [
        keras.Input(shape=(11,1)),
        layers.Conv1D(16, kernel_size=3, activation="relu"), # 16 different 3x3 kernels
        layers.MaxPooling1D(2),
        layers.Conv1D(32, kernel_size=3, activation="relu"), # 32 different 3x3 kernels
        layers.Flatten(),               # reshape multi-dimensional array into a vector.
        layers.Dropout(0.5),            # randomly select 50% the neurons to dropout
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model_CNN.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 9, 16)             64

 max_pooling1d (MaxPooling1D  (None, 4, 16)            0
 )

 conv1d_1 (Conv1D)           (None, 2, 32)             1568

 flatten_7 (Flatten)         (None, 64)                0

 dropout (Dropout)           (None, 64)                0

 dense_50 (Dense)            (None, 2)                 130

=================================================================
Total params: 1,762
Trainable params: 1,762
Non-trainable params: 0
_________________________________________________________________
```

In [87]:
```
batch_size = 64
epochs = 300

model_CNN.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
early = keras.callbacks.EarlyStopping(monitor='val_accuracy', min_delta=0, patience=30, verbose=0, mode='auto', baseline=None, restore_best_weights=False)
lr = keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy', factor=0.1, patience=10, verbose=0, mode='auto', min_delta=0.0001, cooldown=0, min_lr=0)
model_CNN.fit(data_train_features_norm.reshape(3918,11,1), keras.utils.to_categorical(train_label_values,num_classes), batch_size=batch_size, epochs=epochs, val
```

In [88]:
```
data_test_features_norm.shape
```

Out[88]:
```
(980, 11)
```

In [89]:
```
score = model_CNN.evaluate(data_test_features_norm.reshape(980,11,1), keras.utils.to_categorical(test_label_values,num_classes))
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
31/31 [==============================] - 0s 555us/step - loss: 0.3505 - accuracy: 0.8418
Test loss: 0.35045021772384644
Test accuracy: 0.8418367505073547
```