Wesley Pryor

Wine Classification

Final Project

M5364–Data Mining Fall 2016 Tarleton State Univ Dr. Scott Cook Assigned 2016-11-02 Due 2016-12-09

Summary of Data and Models

The Wine Quality Data set is a set of data that contains observations of wines. This data comes form the UCI Machine Learning Repository. The data is divided into two sets.The first data set contains 4898 observations of white wines. The second data set has 1599 different red wines. The data contains 12 attributes. The data has 11 numeric attributes. The numeric attributes are fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, and alcohol. The twelfth attribute is quality. This part of the data is usually what is trying to be calculated with a regression problem, or it is also determined via classification techniques. I was not sure how this attribute could be treated. I initially thought about treating it just as a normal numeric value. However, I wanted to see how to treat it as a factor.

I decided to tackle this set of data by combining the red and white rows of data. The combines red and white data had 6,497 different type of wines from Portugal. I wanted to classify the wines between red and white. For this project, I applied classification techniques that we have covered in class throughout the semester to determine the type of wine. The table below describes the techniques applied and the values of accuracy for the respective models. All original models were run with a simple 70-30 split and then cross-validation was applied to the models to determine the mean accuracy for each technique.

Table 1: Accuracy of Models Using Cross-Validation

Model	Accuracy
R Part Decision Trees	0.9804547
C Tree Decision Trees	0.9761422
K Nearest Neighbors	0.9938433
Weighted K nearest Neighbors	0.9943058
Naive Bayes	0.9752199
Support Vector Machines	0.9955366
Artificial Neural Networks	0.9723155

R Part Decision Trees

When trying to classify the model. I initially ran the decision tree with the `rpart` package. This decision tree started with a depth of 4 and and accuracy of 97.6%. Most of the data in the naive run of the tree was split from the counts of total sulfur dioxides and chlorides. The other factor involved was volatile acidity. Once this model was run, tuning was applied to the complexity parameter, minimum split, and the maximum depth of the trees.

For the complexity parameters, I decided that the best was to test small values from 0.0 to 0.2. After tuning was run, 0.001 was determined to be the most accurate parameter for the model. From our in class sources, I initially tried to test a minimum split of 5, 10, or 15. However, I was getting the same error rate for all of those values. After looking through the data more, I realized that trying to split with only 5, 10,, or 15 observations would be over-fitting the model and not very ideal. I then tried a splits of 50 125 by 5 and i saw that the errors began to change after 50 splits. 50 was determined to be the best minimum split. The maximum depth was decided from values between 1 and 10. The best tree had a depth of 6. Once complexity parameter, minimum split, and maximum depth were set to the optimal parameters, the accuracy of the rerun decision tree was 98%. The best decision tree included fixed acidity in splitting the data into the type of wines, along with the other attributes that were in the naive tree. The optimal decision tree was run using 10 fold cross-validation and the mean of those accuracies was 98%.

C Tree Decision Trees

In class this semester, our assignments led us to believe that the best decision tree was used with the `rpart` calculations instead of the `ctree` calculations. I wanted to test the method and determine which method was better for this data. The first attempt of the model split most frequently with density, alcohol, total sulfur dioxides, sulphates and chlorides. Other attributes used in the model were pH, residual sugar, volatile acidity, fixed acidity, and chlorides. With these factors, the model had 69 nodes and the model has a depth of 8. The initial accuracy was 98.1% The model was tuned for the minimum split and the maximum depth. It was tested for values of minimum split at 50, 100, and 150. The optimal value was 50. At maximum depth, values between 1 and 10 were tested. The best depth was calculated to be 10. The optimal tree was tested with these values and had an accuracy of 98.3%. The tree looked similar to the first run of a decision tree. However, the overall depth was increased by one. Once cross-validation was done with this model, the average accuracy was much lower at 97.6%. I was suprised to see this. However, I believe that this was done from splits that had more data that was had similar values in certain categories. Overall after cross-validation of both methods, the `rpart` calculations were proven to be better for this data.

K nearest neighbors

K nearest neighbors was standardized initially and was calculated with 3 nearest neighbors. This trial resulted in a 99.2% accuracy. The trial was then tuned with nearest neighbors of 2 through 20. The method was also tuned with three different types of sampling: cross-validation, fixed sampling, and bootstrapping. The best nearest neighbors for cross-validation was 5. Fixed sampling had an optimal nearest neighbor count of 11. Bootstrapping was 12. I tested the model with all the values of k that were determined in the sampling and then cross-validated. Through tuning and cross-validation, the overall average accuracy was 99.3%

Weighted K Nearest Neighbors

I wanted to test to see how the kernels of K nearest neighbors affected the values of the model. The first trial was with a triangular kernel and the accuracy was 99.2%. I tested all eight kernels available. I expected that the optimal kernel would perform better than my first trial. After testing, the inverted kernel was determined to have minimal errors with 3 nearest neighbors. The accuracy was run once for a 99.2% accuracy. cross-validation resulted in a mean 99.4% accuracy.

Naive Bayes

Naive Bayes has the assumption of normality. I had to first test the normality of all attributes of the data. The numerical values of the data were not normal. At this point I tried to see if I could treat the quality as a factor instead of a numerical value. All models had obvious curvature in all of the Q-Q Plots. Also, all Shapiro-Wilk tests returned the smallest possible p-values for a normal distribution. The data had to be cut to assume normality with the `mycut` function that was developed in this semester. The attributes that needed normalizing were cut into 10 factor levels. Due to time constraints, I was not able to test multiple cuts. I wish that I had tested multiple cuts to find the best cut for the model. Due to the size of the data, I tried 10 and felt like this was a good estimate. After normalization, the first attempt at the model had a 97.8% accuracy. The model then underwent 10 fold cross-validation and had a mean accuracy of 97.5%.

Support Vector Machines

Before we could start Support Vector Machines, I removed the the combined database that was adjusted for normality. Then I recommitted the combined set of data and ran the support vector machines. The trial support vector machines had an accuracy of 99.6% percent. The tuning then was tested for gamma and costs parameters. The first time that I tuned the function, the upper extremities were the best parameters. I then attempted to tune again with the upper extremities as the lower boundaries of the function. The second attempt at tuning the function resulted in the lower bounds, the original upper extremities, were the optimal gamma and cost. Therefore these values were used and the model after tuning was 99.6%.

Artificial Neural Networks

The neural networks was run using the `nnet` package. The first time it was run with 2 levels and a maximum iteration at 100. The first accuracy was at 99.8% I changed the iterations to 500 and the size of the neural network to 10 hidden layers. During the run of the neural network command. The first iterations stopped at 230 and the accuracy was 99.7%. The network was tuned with sizes of hidden layers between 1 and 15. Also the tuning function was applied with controls of 5 repeats, cross validation sampling, and 10 crosses. The tuning function ran cross validation and resulted in 97.2% accuracy. I was shocked at this low value considering the first values that i got that were extremely high. I think that my first guess of 10 layers was the best for the model and the sum from all other sizes of hidden layers caused the value to overall be lower.

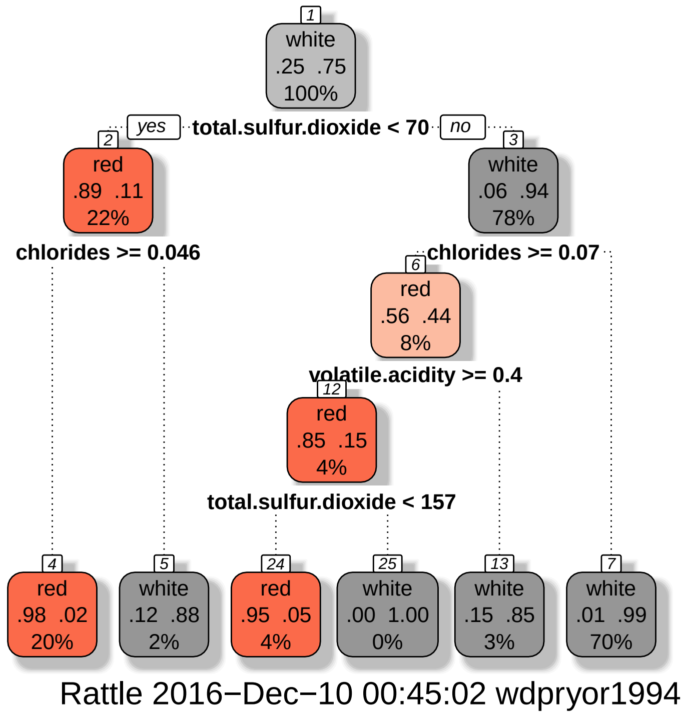
Conclusion

Overall, Support Vector Machines were the optimal model for this data. Neural Networks and the C Tree Decision Trees were the overall worst models for the data. As I was working with the data, I really did not consider the chemical make up of a red wine and the chemical makeup of a white wine. I just wanted to run the data to see how it behaved on it's own. After looking more into the data, there are certain key attributes that really determine the model. Alcohol, density, total sulfur dioxides, free sulfur dioxides, chlorides, and pH were the best attributes that could determine the type of wine. Models where these were these were used instead of the whole data could probably perform at similar accuracies to the models run with all attributes.

Code Appendix

```r
set.seed(1506)
# Importing Data Sets
Red_Wine <- read.csv("~/Downloads/winequality-red.csv", sep = ";")
White_Wine <- read.csv("~/Downloads/winequality-white.csv", sep = ";")

# Adding the Type of Wine to the data set.
Red <- c(rep("red", nrow(Red_Wine)))
White <- c(rep("white", nrow(White_Wine)))
Red_Wine <- cbind(Red_Wine, Red)
White_Wine <- cbind(White_Wine, White)

# Changing the type of wine columns
colnames(Red_Wine)[13] <- "wine.type"
colnames(White_Wine)[13] <- "wine.type"

# Combining Both Red and White Wines into one data set
Total_Wine <- rbind(Red_Wine, White_Wine)
Total_Wine$quality <- as.factor(Total_Wine$quality)

# Item that will be predicted with the classification.
Wine.Type <- Total_Wine$wine.type

# Splitting Data for testing purposes
Wine.Split <- splitdata(Total_Wine, 0.7)
Wine.train <- Wine.Split$traindata
Wine.test <- Wine.Split$testdata

```

```
# Decision Trees to determine the type of wine, red or white.
wine.tree ← rpart(wine.type ~ ., data = Wine.train)
plot.wine.tree ← fancyRpartPlot(wine.tree, palettes = c("Reds", "Greys"))
```

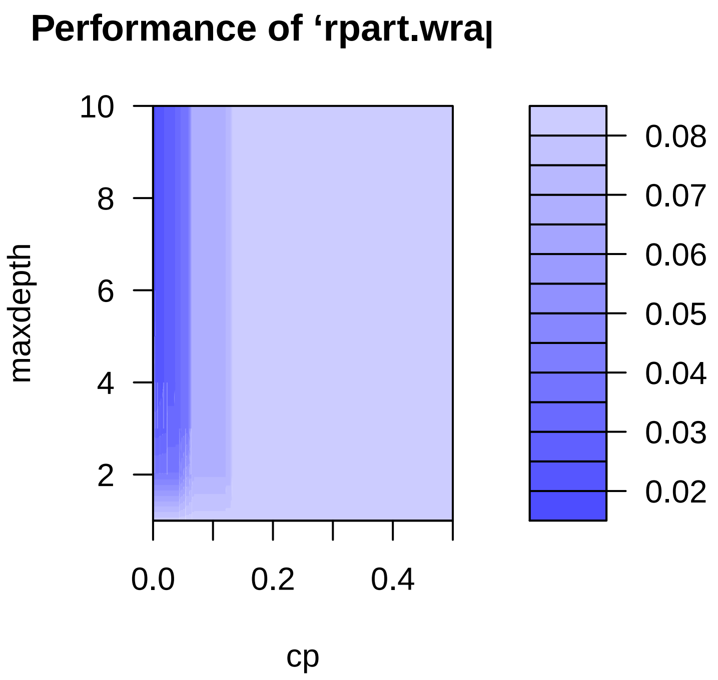


```
pred.tree ← predict(wine.tree, newdata = Wine.test, type = "class")

# Analysis of the decision tree
tree.conf ← confusion(Wine.test$wine.type, pred.tree, costs = NULL)
tree.conf
```

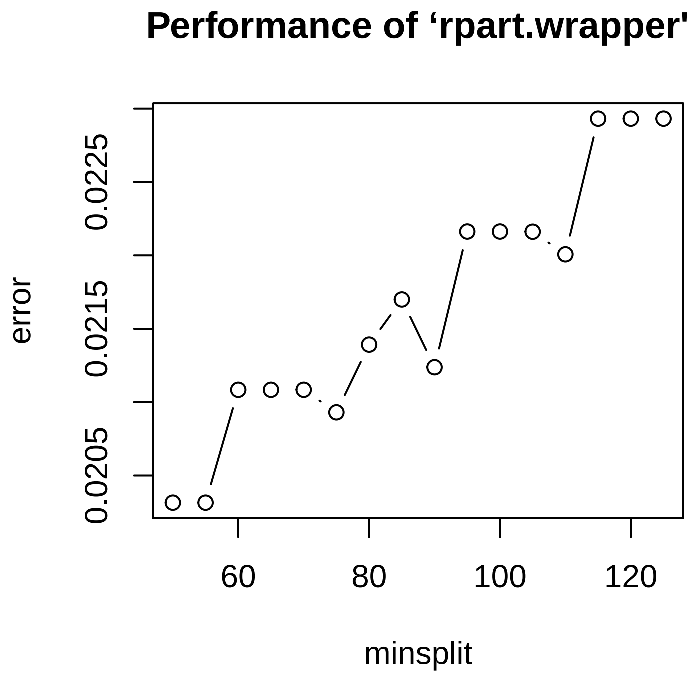
```
$counts
       pred
true     red white
  red    442    36
  white   10  1461

$acc
[1] 0.9763982

$rates
       pred
true            red       white
  red   0.924686192 0.075313808
  white 0.006798097 0.993201903

$sensitivities
      red     white
0.9246862 0.9932019

$precisions
      red     white
0.9778761 0.9759519

$F1s
```

```
      red     white
0.9505376 0.9845013

$cost
[1] 46
```

```
# Tuning decision tree
tree.tune ← tune.rpart(wine.type ~ ., data = Total_Wine, cp = seq(0, 0.5, 0.001),
    maxdepth = 1:10)
tree.tune.2 ← tune.rpart(wine.type ~ ., data = Total_Wine, minsplit = seq(50,
    125, 5))
plot(tree.tune)
```



```
plot(tree.tune.2)
```



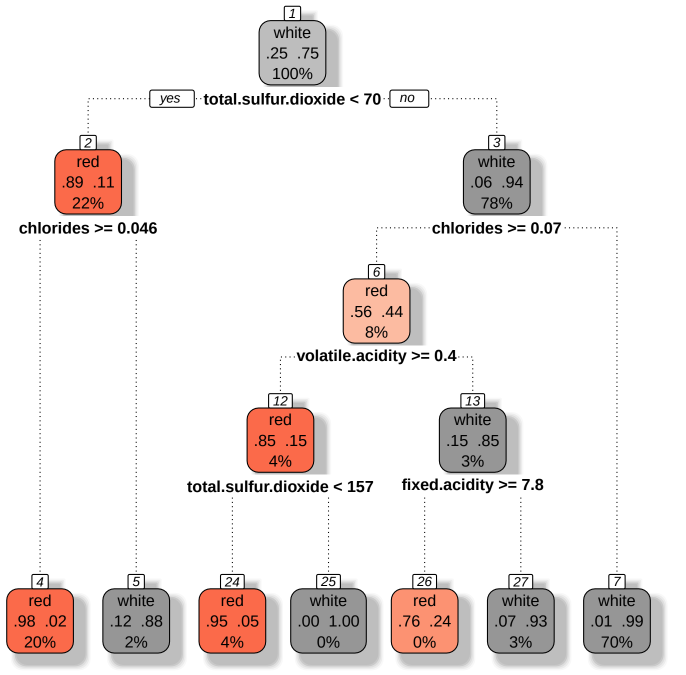
```r
# The best parameters for the rpart decision tree
tree.tune$best.parameters
```

```
       cp maxdepth
2507 0.001        6
```

```r
tree.tune.2$best.parameters
```

```
  minsplit
1       50
```

```r
# Optimal rpart tree
tree.2 ← rpart(wine.type ~ ., data = Wine.train, control = rpart.control(minsplit =
    tree.tune.2$best.parameters$minsplit,
    cp = tree.tune$best.parameters$cp, maxdepth = tree.tune$best.parameters$maxdepth))
fancyRpartPlot(tree.2, palettes = c("Reds", "Greys"))
```



Rattle 2016−Dec−10 01:40:35 wdpryor1994

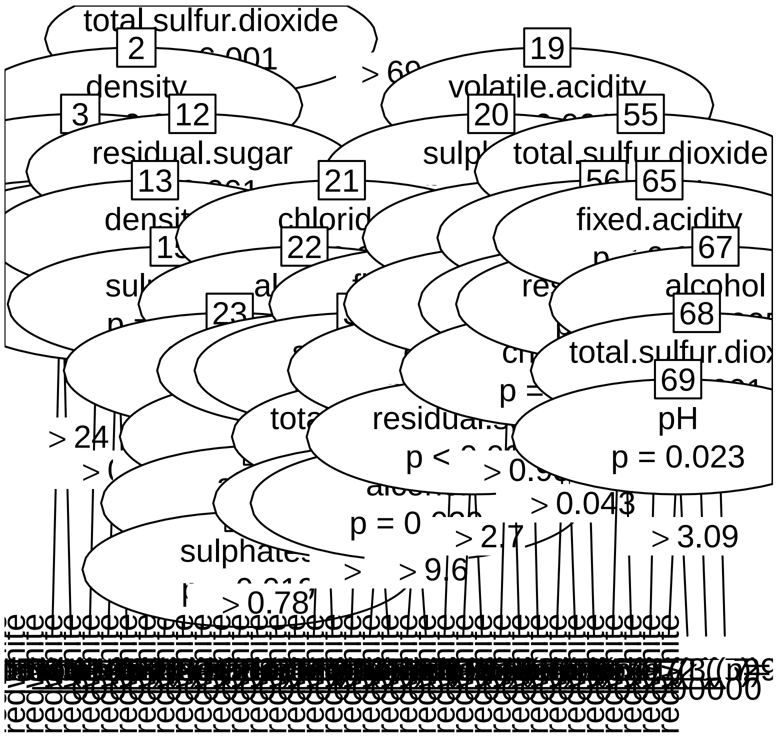
```
pred.tree.2 ← predict(tree.2 , newdata = Wine.test , type = "class")

# Analysis of optimal decision tree with cross validation
tree.conf.2 ← confusion(Wine.test$wine.type , pred.tree.2 , costs = NULL)
tree.conf.2
```

```
$counts
       pred
true      red white
  red     452    26
  white    12  1459

$acc
[1] 0.9805028

$rates
       pred
true            red       white
  red   0.945606695 0.054393305
  white 0.008157716 0.991842284

$sensitivities
      red     white
0.9456067 0.9918423

$precisions
      red     white
0.9741379 0.9824916

$F1s
      red     white
0.9596603 0.9871448

$cost
[1] 38
```

```
kflval ← function(k, data) {
    folds = createfolds(nrow(data), k)
    accvector = 1:k
    for (k in 1:k) {
        temptrain = data[folds != k, ]
        temptest = data[folds == k, ]
        temptree = rpart(wine.type ~ ., data = temptrain, control = rpart.control(minsplit =
            tree.tune.2$best.parameters$minsplit,
            cp = tree.tune$best.parameters$cp, maxdepth = tree.tune$best.parameters$maxdepth))
        temppred = predict(temptree, newdata = temptest, type = "class")
        analysis = confusion(temptest$wine.type, temppred)
        accvector[k] = analysis$acc
    }
    return(mean(accvector))
}

kflval(10, Total_Wine)
```

```
[1] 0.9789119
```

```
# Ctree
wine.ctree ← ctree(wine.type ~ ., data = Wine.train)
plot(wine.ctree)
```



```
# Prediction and confusion of ctree decision tree
pred.tree.3 ← predict(wine.ctree, newdata = Wine.test)
wine.ctree.conf ← confusion(Wine.test$wine.type, pred.tree.3, costs = NULL)
wine.ctree.conf
```

```
$counts
       pred
true      red white
```

```
  red    458    20
  white   17  1454

$acc
[1] 0.9810159

$rates
       pred
true            red       white
  red   0.95815900 0.04184100
  white 0.01155676 0.98844324

$sensitivities
      red     white
0.9581590 0.9884432

$precisions
      red     white
0.9642105 0.9864315

$F1s
      red     white
0.9611752 0.9874363

$cost
[1] 37
```

```
# Testing minsplit and maxdepth of ctree
maxdepth <- 1:10
# 50 minsplit
accvec <- NULL
for (i in 1:length(maxdepth)) {
    temp.tree <- ctree(wine.type ~ ., data = Wine.train, controls = ctree_control(minsplit = 50,
        maxdepth = maxdepth[i]))
    temp.pred <- predict(temp.tree, newdata = Wine.test)
    temp.conf <- confusion(Wine.test$wine.type, temp.pred, costs = NULL)
    accvec[i] <- temp.conf$acc
}

# 100 misplit
accvec.2 <- NULL
for (i in 1:length(maxdepth)) {
    temp.tree <- ctree(wine.type ~ ., data = Wine.train, controls = ctree_control(minsplit = 100,
        maxdepth = maxdepth[i]))
    temp.pred <- predict(temp.tree, newdata = Wine.test)
    temp.conf <- confusion(Wine.test$wine.type, temp.pred, costs = NULL)
    accvec.2[i] <- temp.conf$acc
}

# 150 minsplit
accvec.3 <- NULL
for (i in 1:length(maxdepth)) {
    temp.tree <- ctree(wine.type ~ ., data = Wine.train, controls = ctree_control(minsplit = 150,
        maxdepth = maxdepth[i]))
    temp.pred <- predict(temp.tree, newdata = Wine.test)
    temp.conf <- confusion(Wine.test$wine.type, temp.pred, costs = NULL)
    accvec.3[i] <- temp.conf$acc
}
```

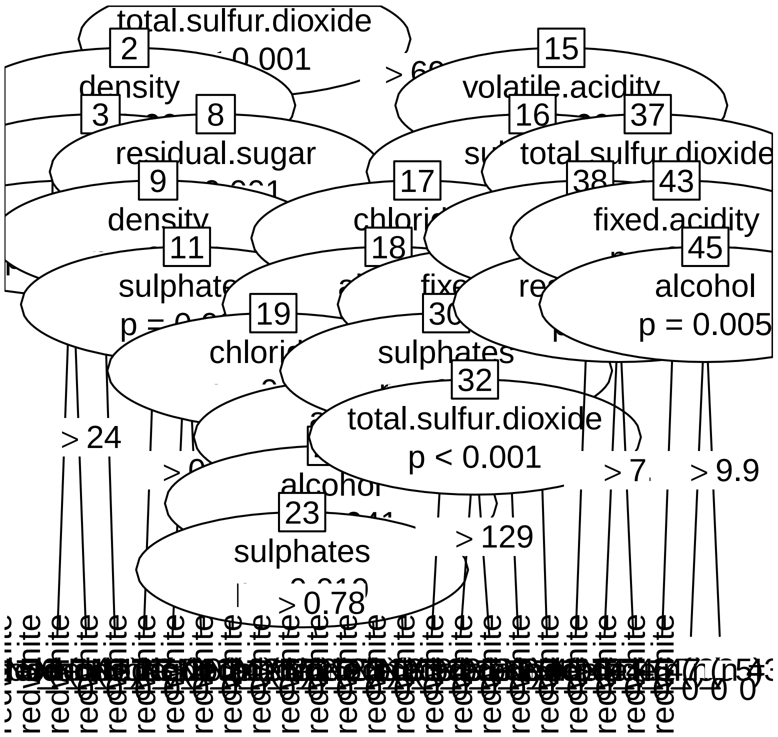
```
# Creating information matrix
max.depth ← as.vector(c(rep(list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), 3)))
min.split ← as.vector(c(rep(list(50, 100, 150), 10)))
accuracy ← as.vector(rbind(accvec, accvec.2, accvec.3))
acc.matrix ← cbind(max.depth, min.split, accuracy)
colnames(acc.matrix)[1] ← "maxdepth"
colnames(acc.matrix)[2] ← "misplit"
colnames(acc.matrix)[3] ← "accuracy"
which.max(accuracy)
```

```
[1] 19
```

```
acc.matrix[which.max(accuracy), ]
```

```
$maxdepth
[1] 9

$misplit
[1] 50

$accuracy
[1] 0.9712673
```

```
# Testing Optimal Ctree
opt.ctree ← ctree(wine.type ~ ., data = Wine.train, controls = ctree_control(minsplit = 50,
    maxdepth = 10))
plot(opt.ctree)
```



```
opt.pred ← predict(opt.ctree, newdata = Wine.test)
opt.conf ← confusion(Wine.test$wine.type, opt.pred, costs = NULL)
opt.conf
```

```
$counts
       pred
```

```
true      red white
  red     451    27
  white    29  1442

$acc
[1] 0.9712673

$rates
       pred
true            red      white
  red   0.94351464 0.05648536
  white 0.01971448 0.98028552

$sensitivities
      red     white
0.9435146 0.9802855

$precisions
      red     white
0.9395833 0.9816201

$F1s
      red     white
0.9415449 0.9809524

$cost
[1] 56
```

```
# Cross Validation with C Tree
ctree.kflval ← function(k, data) {
    folds = createfolds(nrow(data), k)
    accvector = 1:k
    for (k in 1:k) {
        temptrain = data[folds != k, ]
        temptest = data[folds == k, ]
        temptree = ctree(wine.type ~ ., data = temptrain, controls = ctree_control(minsplit = 50,
            maxdepth = 10))
        temppred = predict(temptree, newdata = temptest)
        analysis = confusion(temptest$wine.type, temppred)
        accvector[k] = analysis$acc
    }
    return(mean(accvector))
}

ctree.kflval(10, Total_Wine)
```

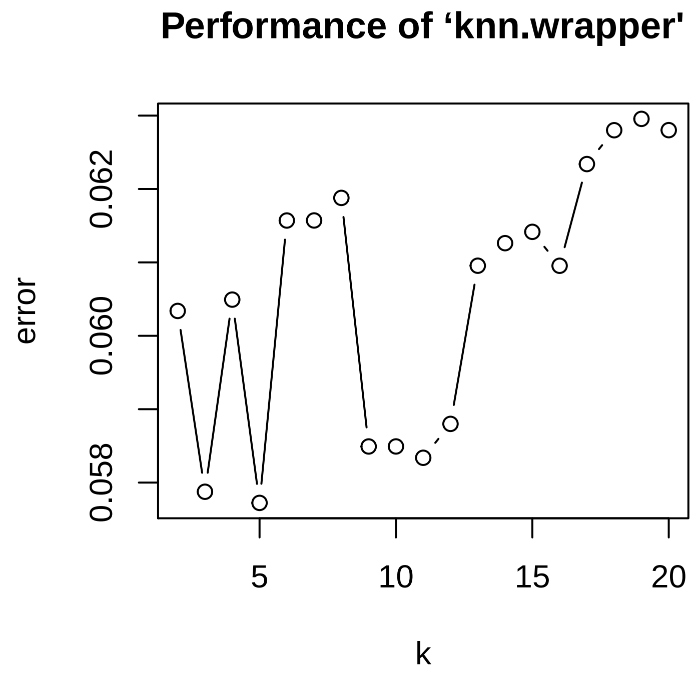
```
[1] 0.9738326
```

```
# K nearest neighbors standardize the data
x = Total_Wine[, 1:11]
xbar = apply(x, 2, mean)
xbarMat = cbind(rep(1, nrow(Total_Wine))) %*% xbar
s = apply(x, 2, sd)
sMat = cbind(rep(1, nrow(Total_Wine))) %*% s
z = (x - xbarMat)/sMat

# K nearest neighbors sampling the data
z.split ← sample(nrow(z), round(nrow(z) * 0.7, 0))
z.train ← z[z.split, ]
```

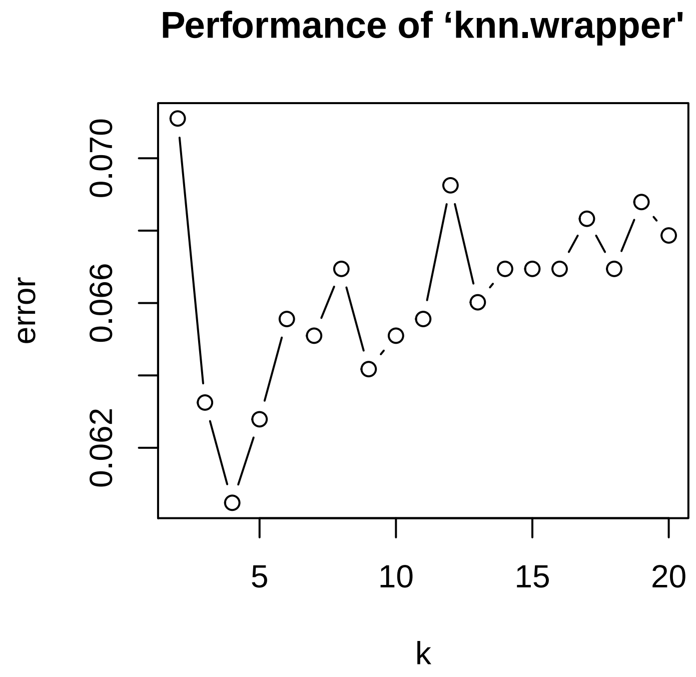
```r
z.test <- z[-z.split, ]

# K nearest neighbors Test
wine.knn <- knn(train = z.train, test = z.test, k = 3, cl = Wine.Type[z.split])
confusion(Wine.Type[-z.split], wine.knn, costs = NULL)
```

```
$counts
       pred
true      red white
  red     478     7
  white     6  1458

$acc
[1] 0.9933299

$rates
       pred
true              red       white
  red   0.985567010 0.014432990
  white 0.004098361 0.995901639

$sensitivities
      red     white
0.9855670 0.9959016

$precisions
      red     white
0.9876033 0.9952218

$F1s
      red     white
0.9865841 0.9955616

$cost
[1] 13
```

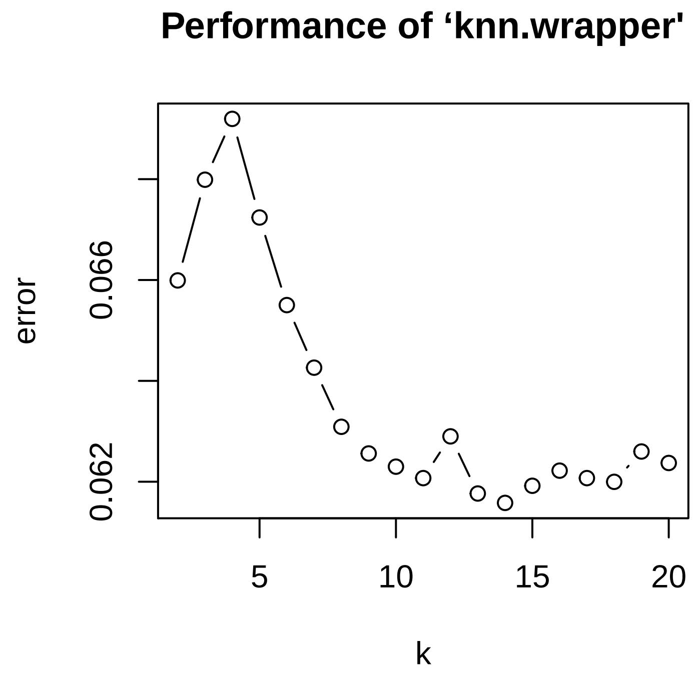
```r
# Finding the optimal k for K nearest neighbors
x <- Total_Wine[, -13]
y <- Total_Wine[, 13]
knn.tune <- tune.knn(x, y, k = seq(2, 20, 1), tunecontrol = tune.control(sampling = "cross"))
knn.tune.2 <- tune.knn(x, y, k = seq(2, 20, 1), tunecontrol = tune.control(sampling = "fix"))
knn.tune.3 <- tune.knn(x, y, k = seq(2, 20, 1), tunecontrol = tune.control(sampling = "boot"))
plot(knn.tune)
```



```
plot(knn.tune.2)
```



```
plot(knn.tune.3)
```



```
knn.tune$best.parameters$k
```

```
[1] 5
```

```
knn.tune.2$best.parameters$k
```

```
[1] 4
```

```
knn.tune.3$best.parameters$k
```

```
[1] 14
```

```
wine.knn.2 <- knn(train = z.train , test = z.test , k = knn.tune$best.parameters$k,
    cl = Wine.Type[z.split])
wine.knn.3 <- knn(train = z.train , test = z.test , k = knn.tune.2$best.parameters$k,
    cl = Wine.Type[z.split])
wine.knn.4 <- knn(train = z.train , test = z.test , k = knn.tune.3$best.parameters$k,
    cl = Wine.Type[z.split])
wine.knn.2.conf <- confusion(Wine.Type[-z.split], wine.knn.2 , costs = NULL)
wine.knn.3.conf <- confusion(Wine.Type[-z.split], wine.knn.3 , costs = NULL)
wine.knn.4.conf <- confusion(Wine.Type[-z.split], wine.knn.4 , costs = NULL)

# Cross Validation for K nearest neighbors
cross.knn <- knn.cv(train = z, cl = y, k = knn.tune$best.parameters$k)
cross.knn.conf <- confusion(Wine.Type, cross.knn)

# Weighted K nearest neighbors
z <- cbind(z, Wine.Type)
z.train <- z[z.split , ]
z.test <- z[-z.split , ]
weight.knn.wine <- kknn(Wine.Type ~ ., train = z.train , test = z.test , k = knn.tune$best.parameters$k,
    kernel = "triangular")
pred.weight <- predict(weight.knn.wine , newdata = z.test)
confusion(z.test$Wine.Type, pred.weight)
```

```
$counts
       pred
true      red white
  red     479     6
  white     4  1460

$acc
[1] 0.9948692

$rates
       pred
true            red      white
  red   0.98762887 0.01237113
  white 0.00273224 0.99726776

$sensitivities
      red     white
0.9876289 0.9972678

$precisions
      red     white
0.9917184 0.9959072

$F1s
      red     white
0.9896694 0.9965870

$cost
[1] 10
```

```
# Finding the best kernels
tune.kknn <- function(K, kernels) {
    acc <- NULL
    for (i in 1:length(K)) {
        temp.model <- kknn(Wine.Type ~ ., train = z.train, test = z.test, k = i,
            kernel = kernels, distance = 2)
        pred.temp <- predict(temp.model, newdata = z.test)
        temp.conf <- confusion(z.test$Wine.Type, pred.temp)
        acc[i] <- temp.conf$acc
    }
    opt.k <- which.max(acc)
    opt.acc <- acc[opt.k]
    return(list(k = opt.k, accuracy = opt.acc))
}
K <- 2:30
rect <- tune.kknn(K, "rectangular")
triangle <- tune.kknn(K, "triangular")
ep <- tune.kknn(K, "epanechnikov")
biw <- tune.kknn(K, "biweight")
triw <- tune.kknn(K, "triweight")
cosine <- tune.kknn(K, "cos")
invert <- tune.kknn(K, "inv")
gauss <- tune.kknn(K, "gaussian")
rank.weight <- tune.kknn(K, "rank")
optimal <- tune.kknn(K, "optimal")
weighted.accuracy <- list(rect$accuracy, triangle$accuracy, ep$accuracy, biw$accuracy,
    triw$accuracy, cosine$accuracy, invert$accuracy, gauss$accuracy, rank.weight$accuracy,
    optimal$accuracy)
```

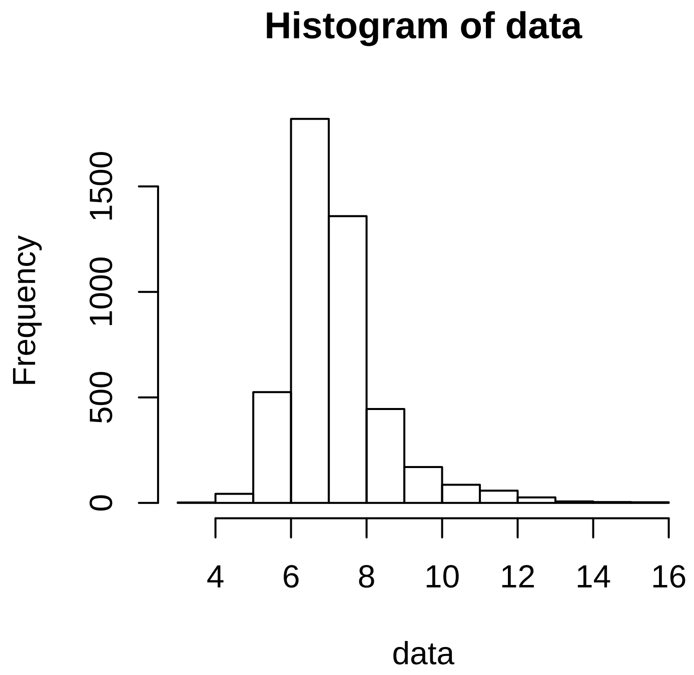
```
weighted.k ← list(rect$k, triangle$k, ep$k, biw$k, triw$k, cosine$k, invert$k,
    gauss$k, rank.weight$k, optimal$k)

opt.weights ← cbind(weighted.k, weighted.accuracy)
opt.kknn ← kknn(Wine.Type ~ ., train = z.train, test = z.test, k = 3, kernel = "inv")
pred.kknn ← predict(opt.kknn, newdata = z.test)
confusion(z.test$Wine.Type, pred.kknn)
```

```
$counts
       pred
true      red white
  red     479     6
  white     5  1459

$acc
[1] 0.9943561

$rates
       pred
true             red       white
  red    0.987628866 0.012371134
  white 0.003415301 0.996584699

$sensitivities
      red     white
0.9876289 0.9965847

$precisions
      red     white
0.9896694 0.9959044

$F1s
      red     white
0.9886481 0.9962445

$cost
[1] 11
```

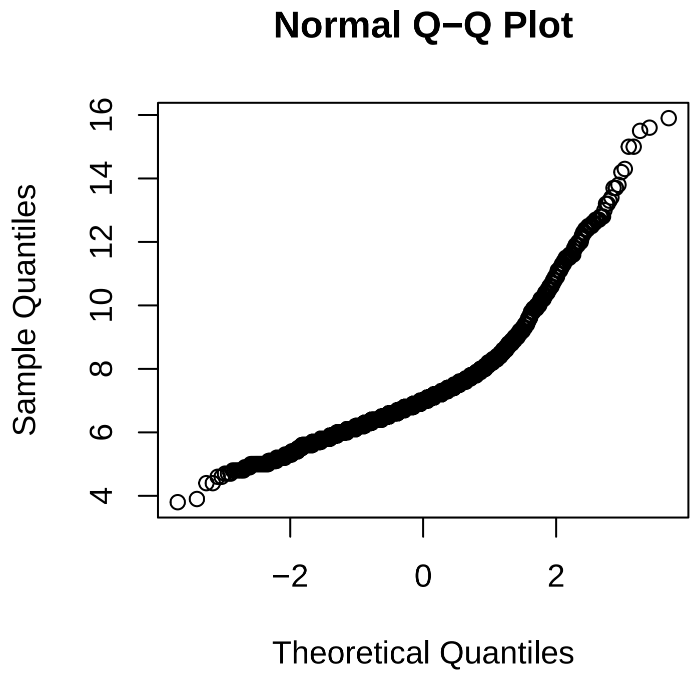
```
# Cross Validation fro Weighted K nearest neighbors
kknn.kflval ← function(k, data) {
    folds = createfolds(nrow(data), k)
    accvector = 1:k
    for (k in 1:k) {
        temptrain = data[folds != k, ]
        temptest = data[folds == k, ]
        tempmodel = kknn(Wine.Type ~ ., train = temptrain, test = temptest,
            k = 3, kernel = "inv")
        temppred = predict(tempmodel, newdata = temptest, type = "raw")
        analysis = confusion(temptest$Wine.Type, temppred)
        accvector[k] = analysis$acc
    }
    return(mean(accvector))
}

kknn.kflval(10, z)
```

```
[1] 0.9941515
```

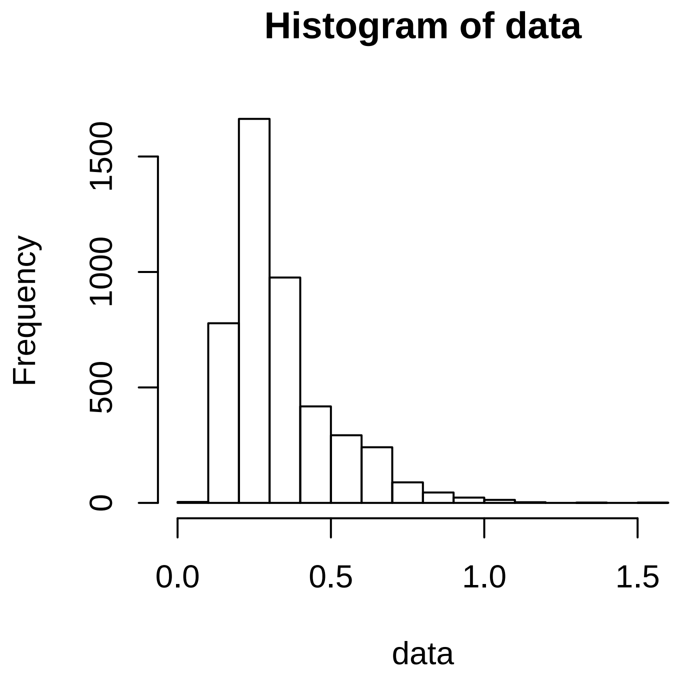
```
# Investigating qualities of data Naive Bayes
normality(Wine.train$fixed.acidity)
```



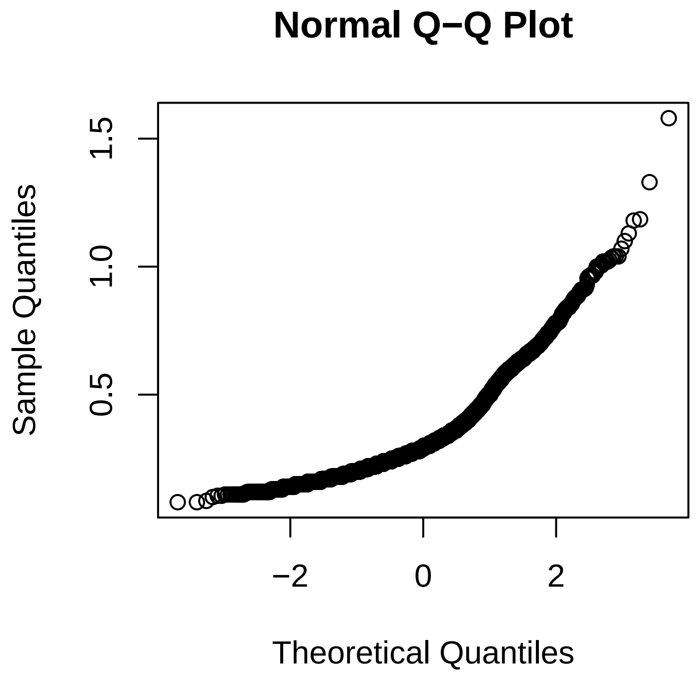
```

	Shapiro-Wilk normality test

data:  data
W = 0.88403, p-value < 2.2e-16
```



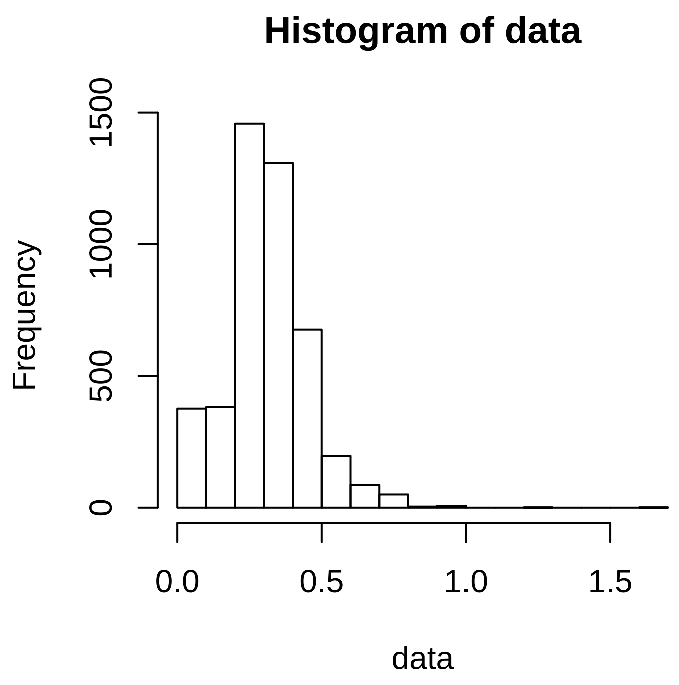
```
normality(Wine.train$volatile.acidity)
```



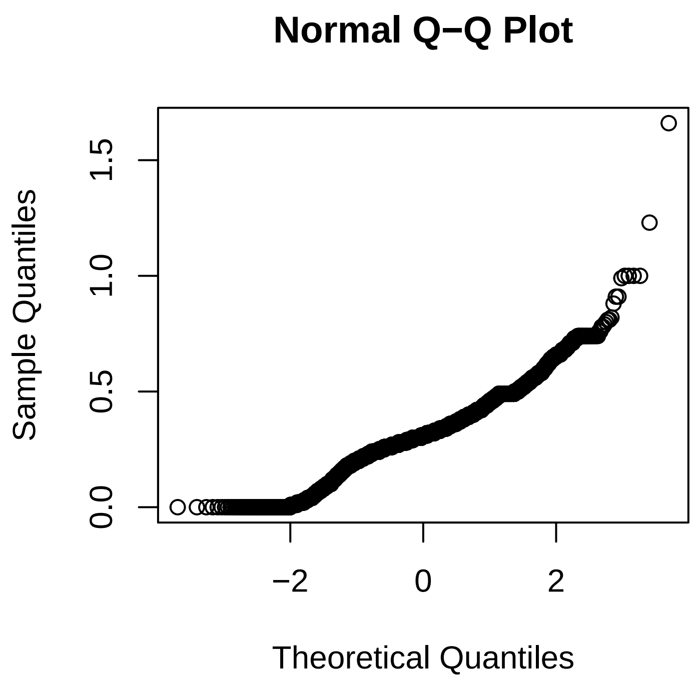
```


        Shapiro-Wilk normality test

data:  data
W = 0.87268, p-value < 2.2e-16
```



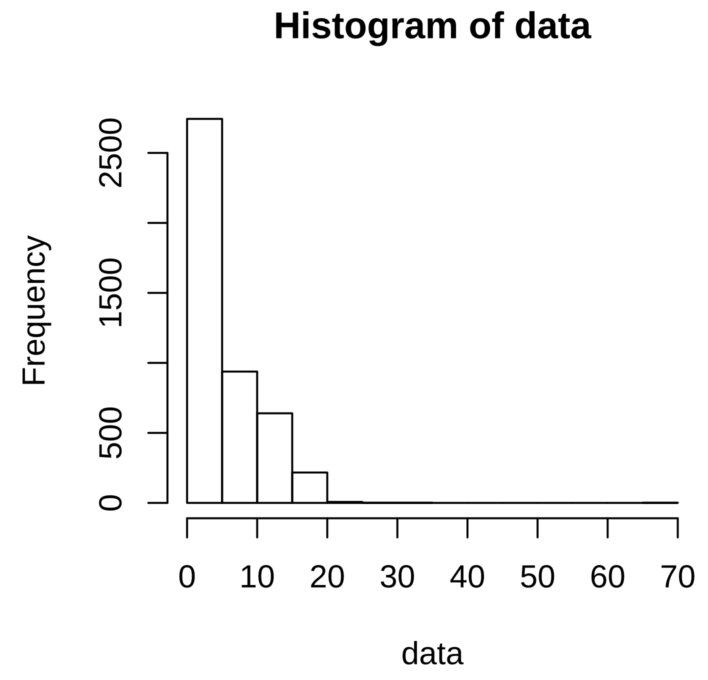
```
normality(Wine.train$citric.acid)
```



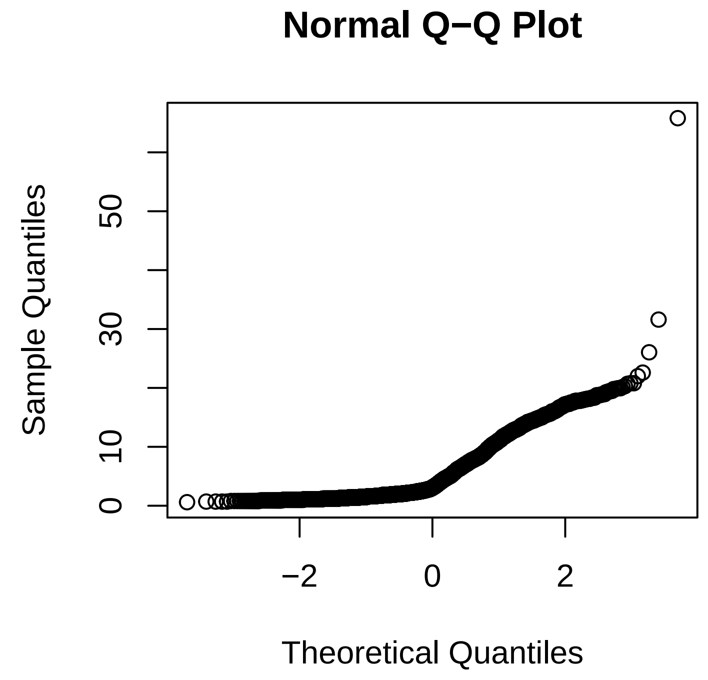
```

	Shapiro-Wilk normality test

data:  data
W = 0.96462, p-value < 2.2e-16
```



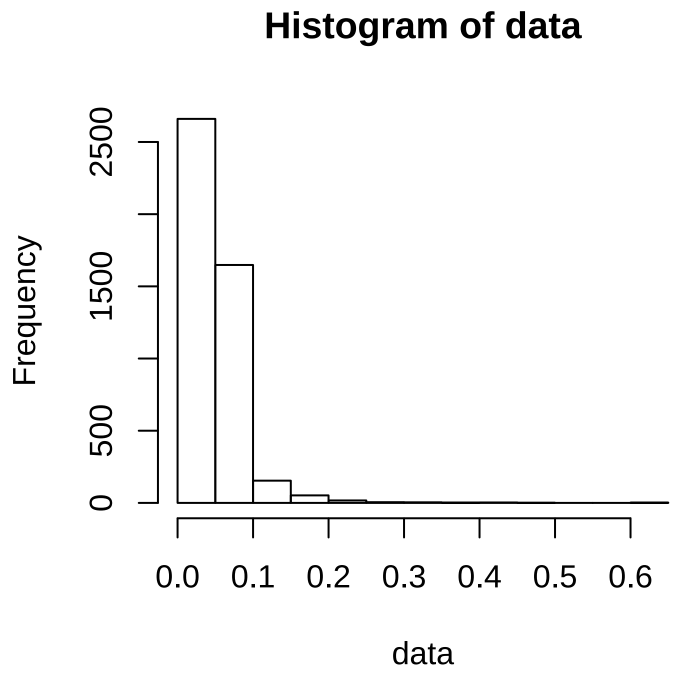
```r
normality(Wine.train$residual.sugar)
```



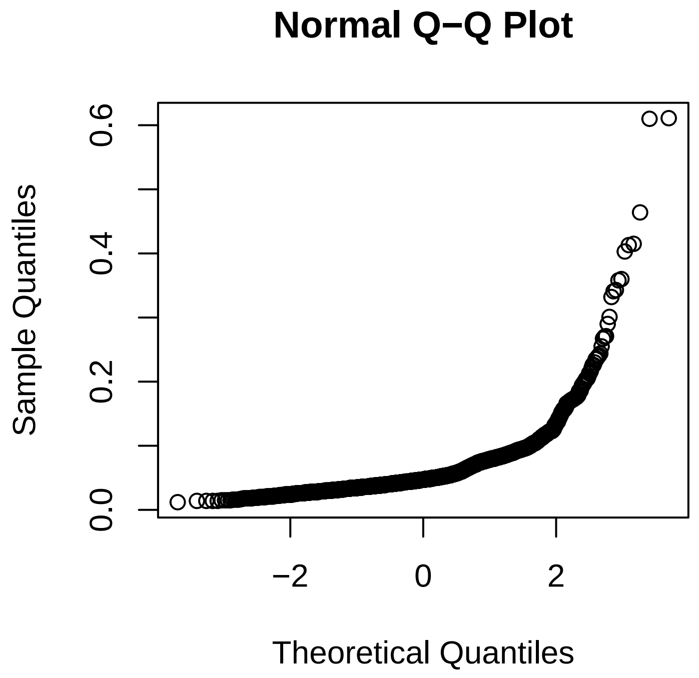
```

        Shapiro-Wilk normality test

data:  data
W = 0.82197, p-value < 2.2e-16
```



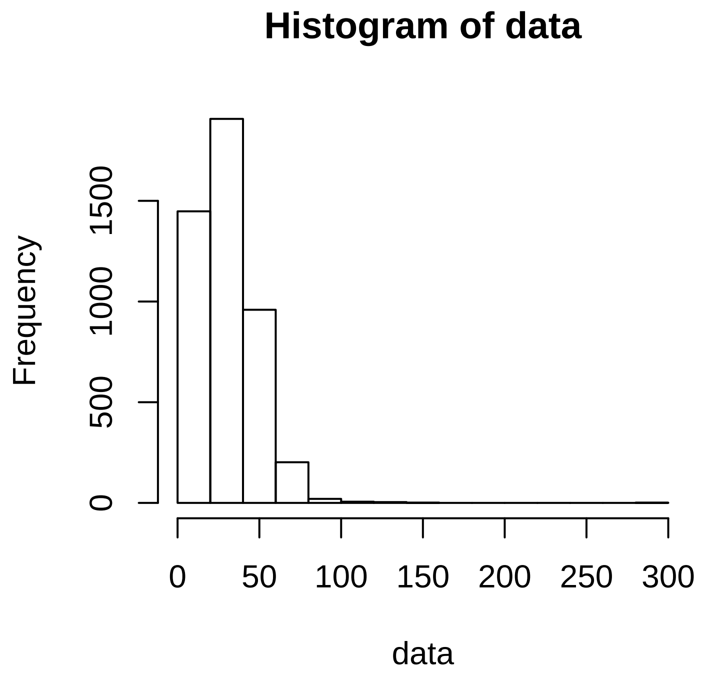
```
normality(Wine.train$chlorides)
```



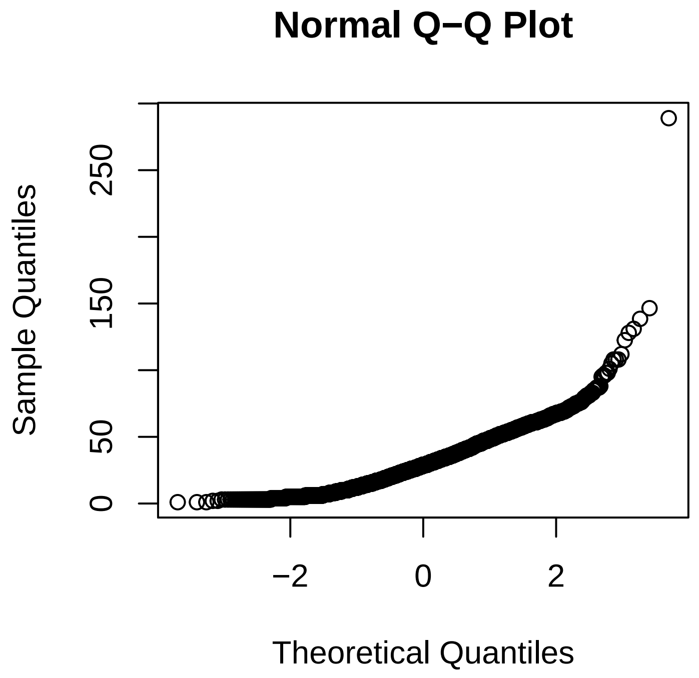
```

	Shapiro-Wilk normality test

data:  data
W = 0.64501, p-value < 2.2e-16

```



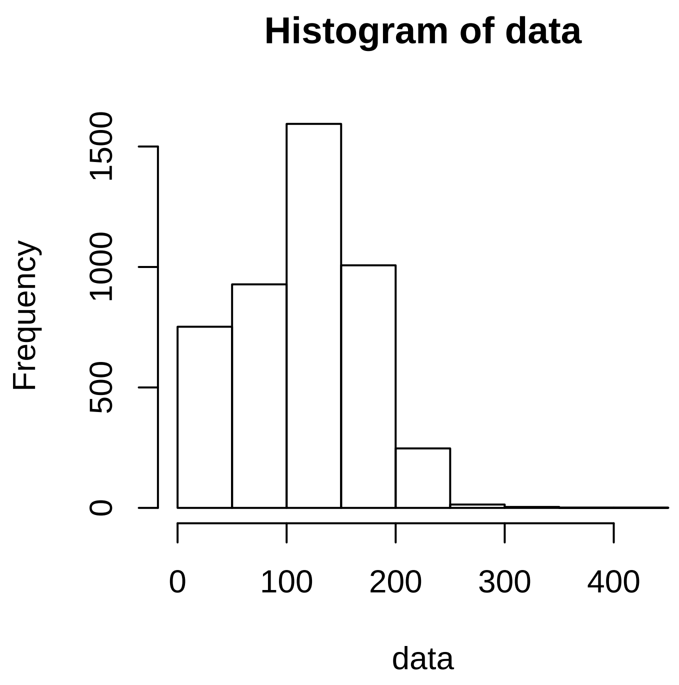
```r

normality(Wine.train$free.sulfur.dioxide)
```



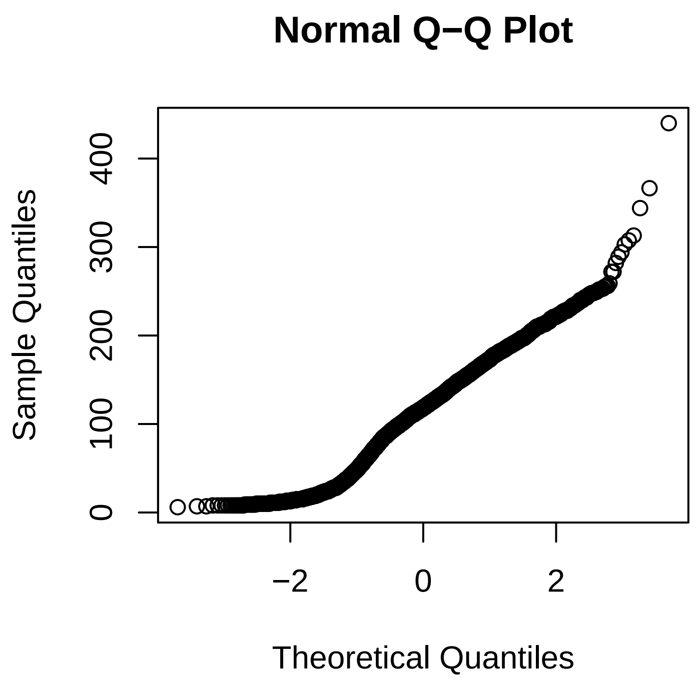
```

        Shapiro-Wilk normality test

data:  data
W = 0.93333, p-value < 2.2e-16
```



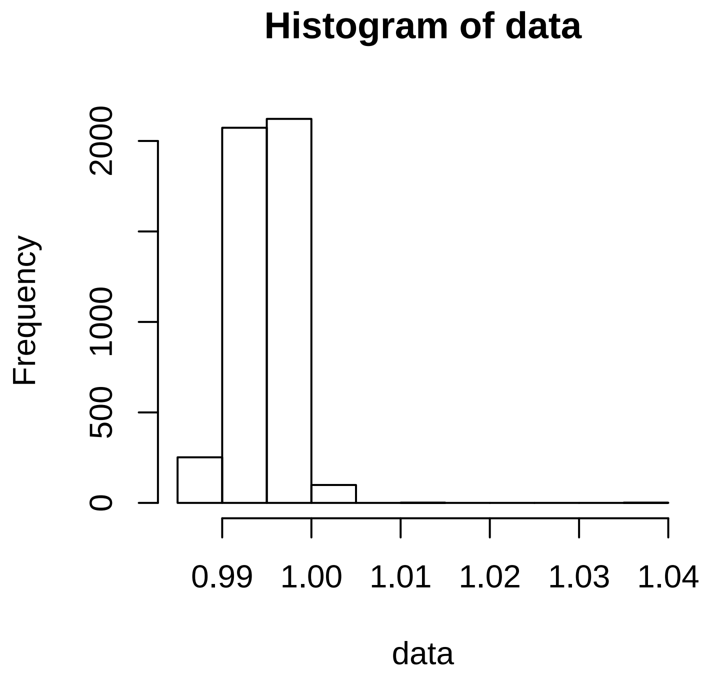
```
normality(Wine.train$total.sulfur.dioxide)
```



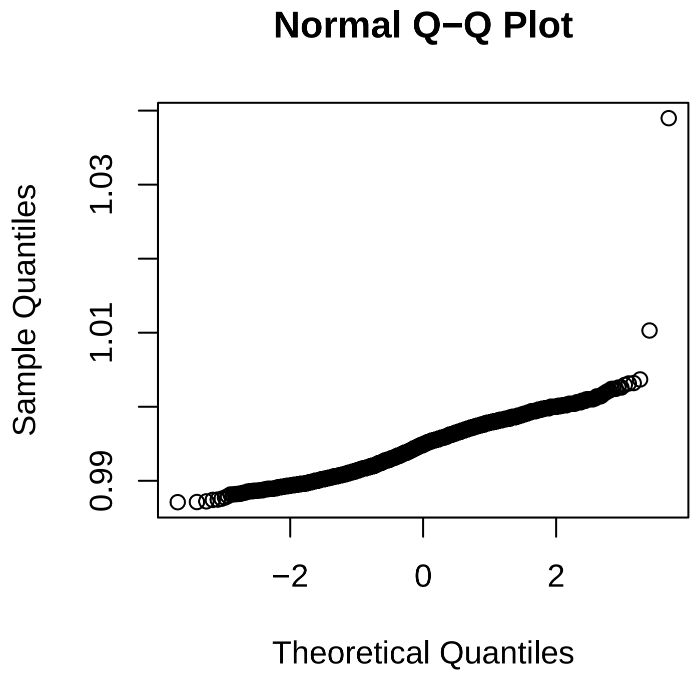
```

	Shapiro-Wilk normality test

data:  data
W = 0.98268, p-value < 2.2e-16
```



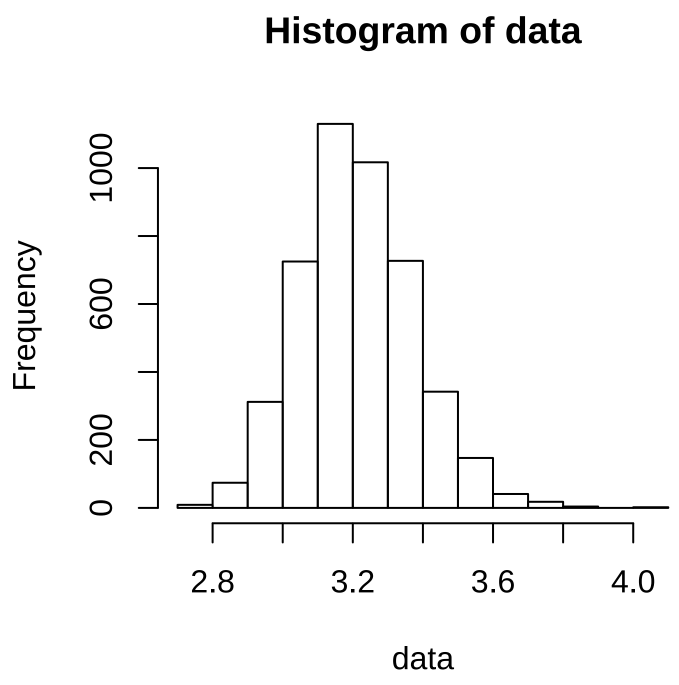
```r
normality(Wine.train$density)
```



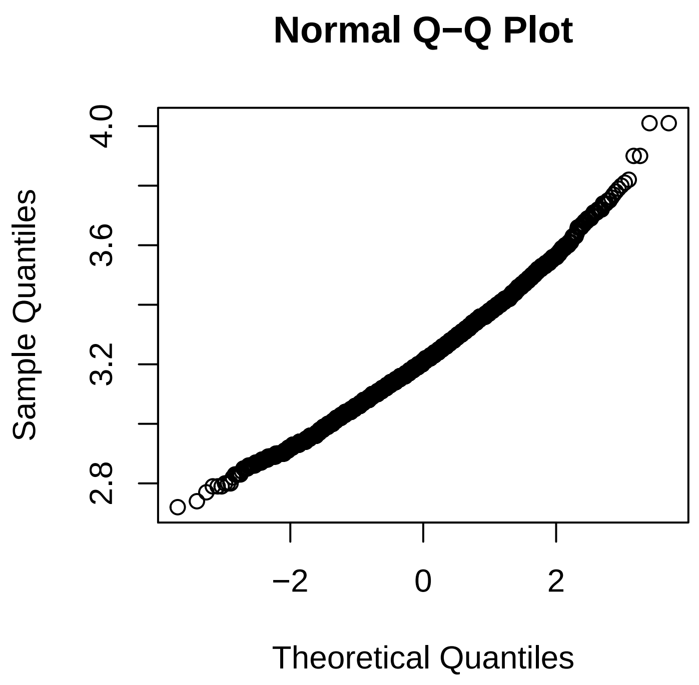
```
        Shapiro-Wilk normality test

data:  data
W = 0.95975, p-value < 2.2e-16
```



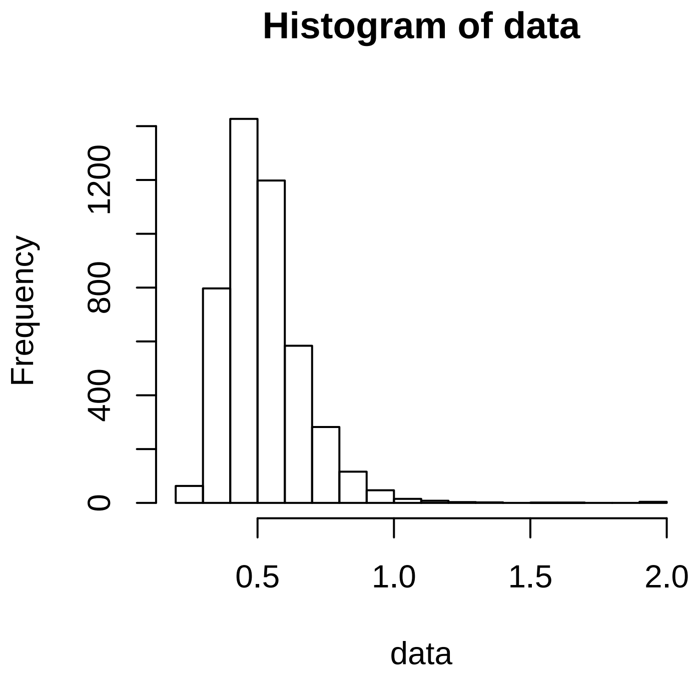
```
normality(Wine.train$pH)
```



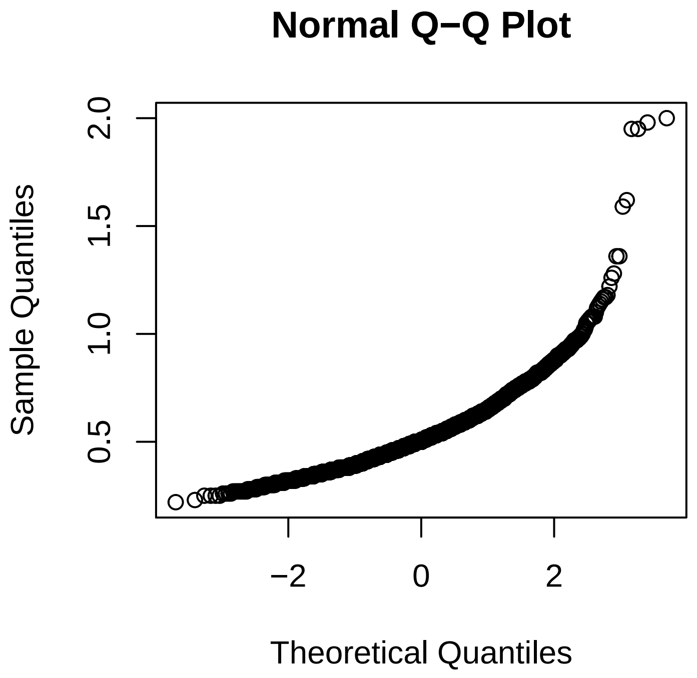
```
        Shapiro-Wilk normality test

data:  data
W = 0.99181, p-value = 1.555e-15
```



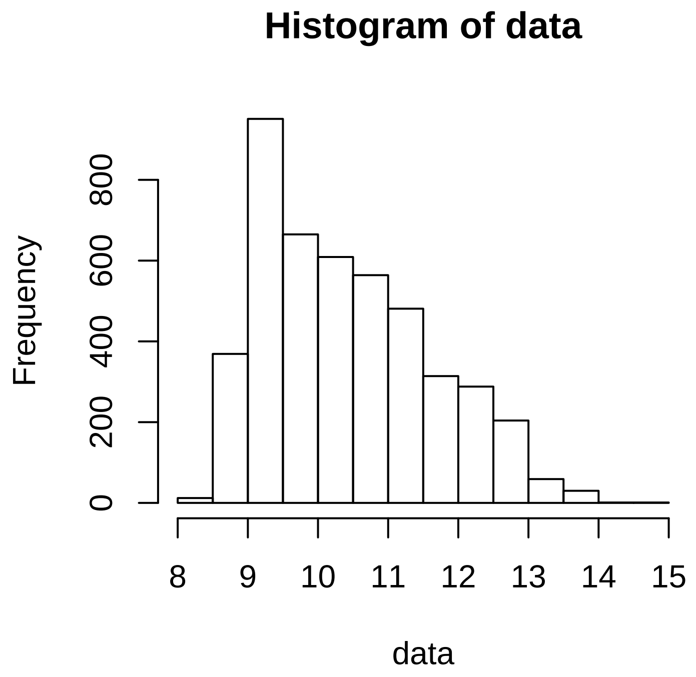
```
normality(Wine.train$sulphates)
```



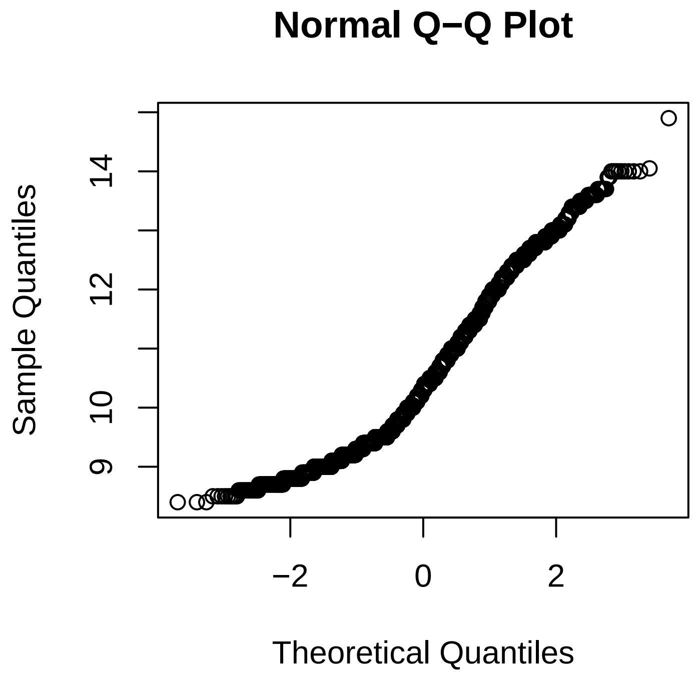
```

	Shapiro-Wilk normality test

data:  data
W = 0.89657, p-value < 2.2e-16
```



```
normality(Wine.train$alcohol)
```



```

	Shapiro-Wilk normality test

data:  data
W = 0.95221, p-value < 2.2e-16
```

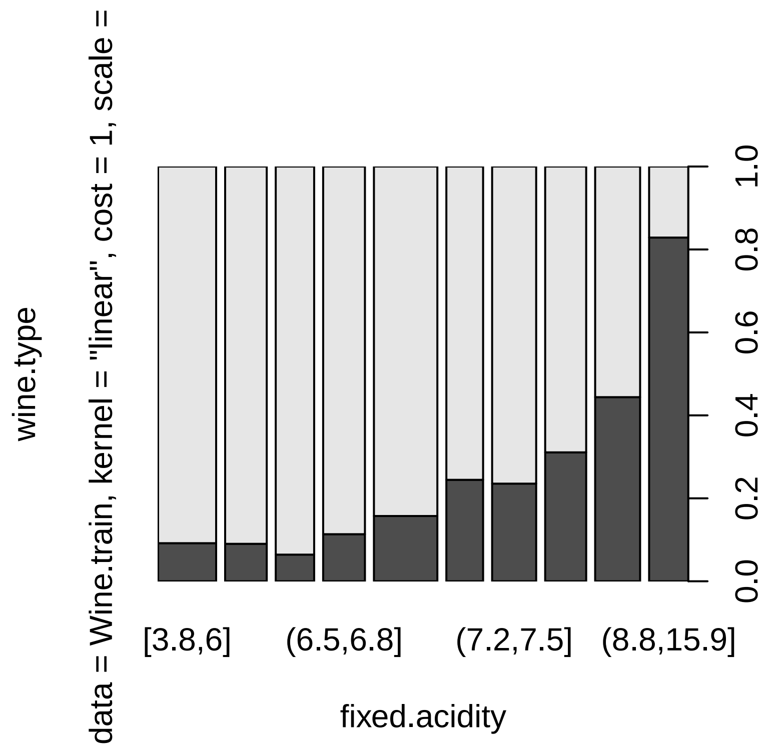


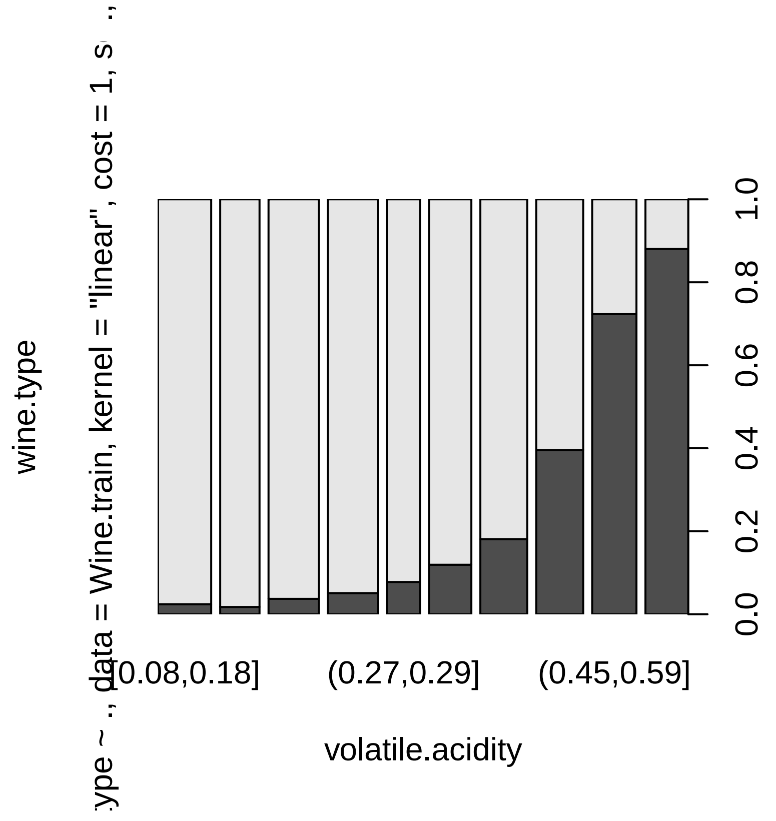
```r
# Discretizing data
Total_Wine$fixed.acidity <- mycut(Total_Wine$fixed.acidity , 10)
Total_Wine$volatile.acidity <- mycut(Total_Wine$volatile.acidity , 10)
```

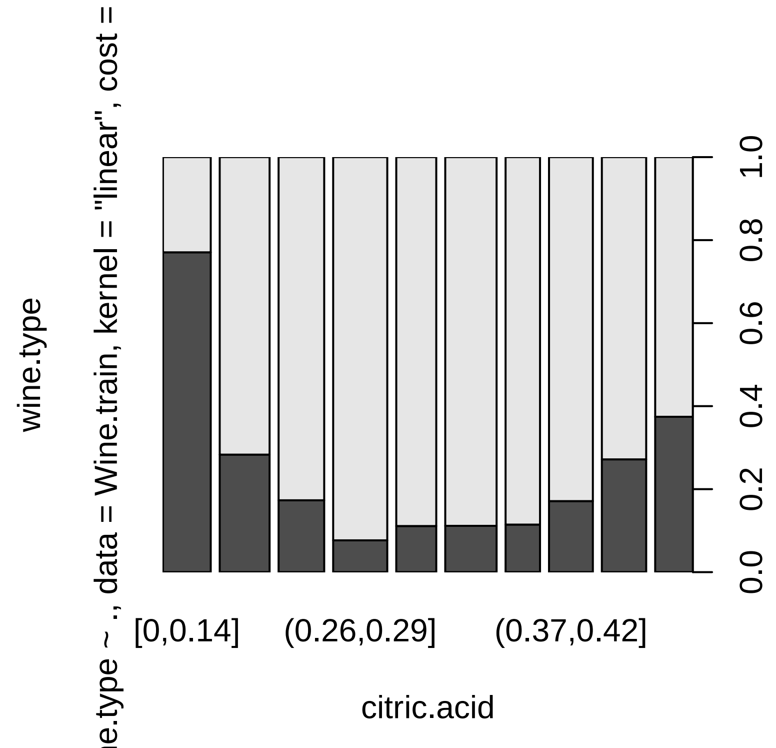
```
Total_Wine$citric.acid ← mycut(Total_Wine$citric.acid , 10)
Total_Wine$residual.sugar ← mycut(Total_Wine$residual.sugar , 10)
Total_Wine$chlorides ← mycut(Total_Wine$chlorides , 10)
Total_Wine$free.sulfur.dioxide ← mycut(Total_Wine$free.sulfur.dioxide , 10)
Total_Wine$total.sulfur.dioxide ← mycut(Total_Wine$total.sulfur.dioxide , 10)
Total_Wine$density ← mycut(Total_Wine$density , 10)
Total_Wine$pH ← mycut(Total_Wine$pH, 10)
Total_Wine$sulphates ← mycut(Total_Wine$sulphates , 10)
Total_Wine$alcohol ← mycut(Total_Wine$alcohol , 10)
Total_Wine$quality ← as.factor(Total_Wine$quality)

Wine.Split ← splitdata(Total_Wine, 0.7)
Wine.train ← Wine.Split$traindata
Wine.test ← Wine.Split$testdata

# Applying Naive Bayes
nb.wine ← naiveBayes(wine.type ~ ., data = Wine.train)
pred.nb.wine ← predict(nb.wine, newdata = Wine.test, type = "class")
nb.conf ← confusion(Wine.test$wine.type, pred.nb.wine)

# Cross Validation
nb.kflval ← function(k, data) {
    folds = createfolds(nrow(data), k)
    accvector = 1:k
    for (k in 1:k) {
        temptrain = data[folds != k, ]
        temptest = data[folds == k, ]
        tempmodel = naiveBayes(wine.type ~ ., data = temptrain)
        temppred = predict(tempmodel, newdata = temptest, type = "class")
        analysis = confusion(temptest$wine.type, temppred)
        accvector[k] = analysis$acc
    }
    return(mean(accvector))
}

nb.kflval(10, Total_Wine)
```

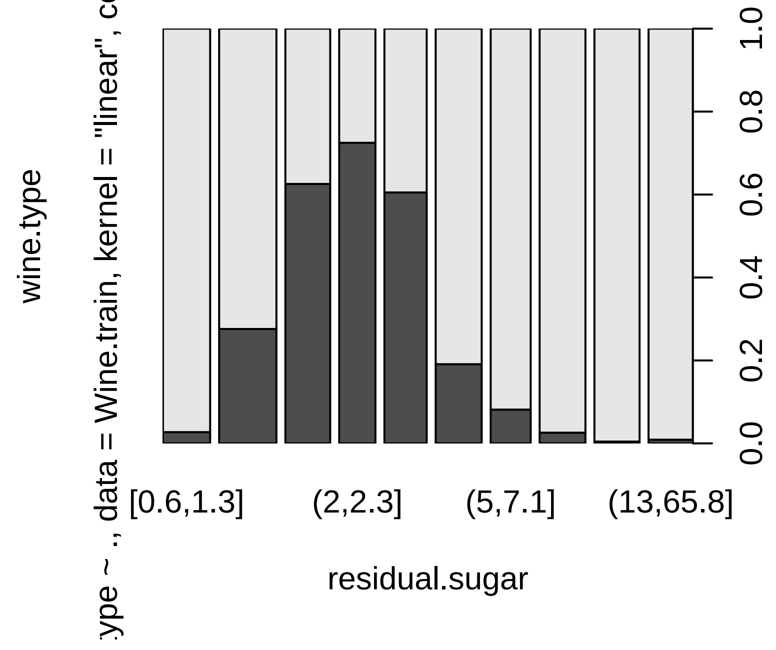
```
[1] 0.9898424
```

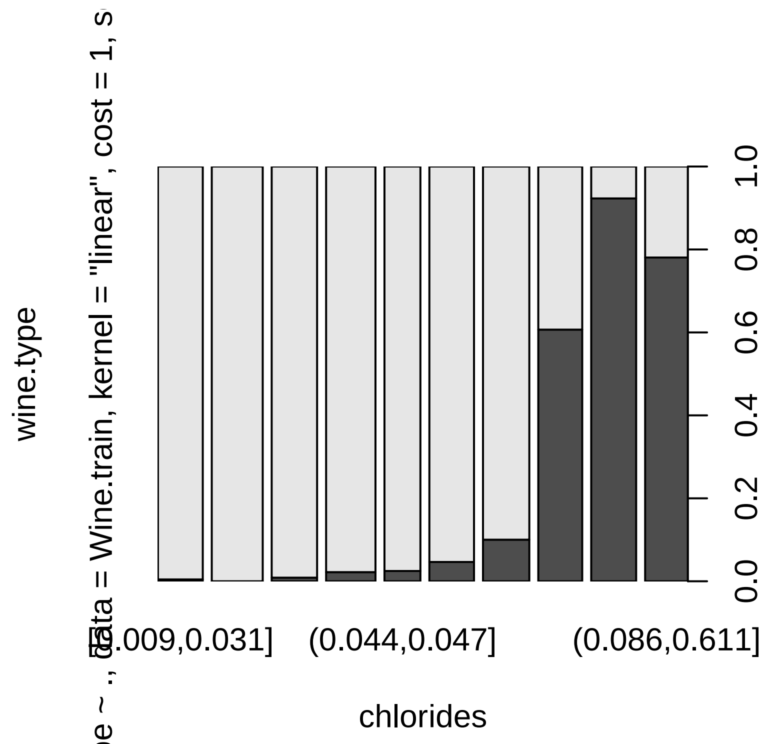
```
# Support Vector Machines
rm(Total_Wine)
Total_Wine ← rbind(Red_Wine, White_Wine)
wine.svm ← svm(wine.type ~ ., data = Wine.train, kernel = "linear", cost = 1,
    scale = T)
pred.svm ← predict(wine.svm, newdata = Wine.test)
svm.conf ← confusion(Wine.test$wine.type, pred.svm)
plot(wine.type ~ ., wine.svm, data = Wine.train)
```

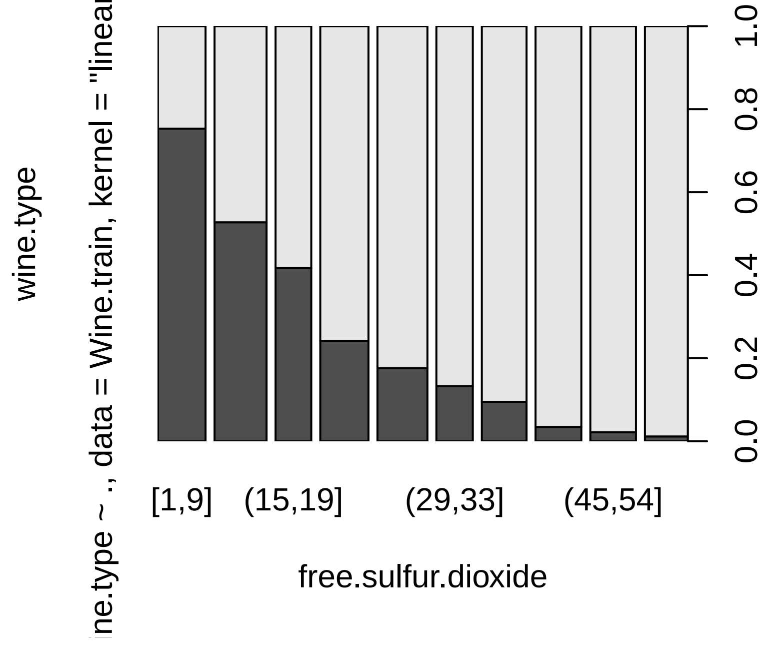

type ~ ., data = Wine.train, kernel = "linear", cost = 1, scale =
wine.type
fixed.acidity
[3.8,6]
(6.5,6.8]
(7.2,7.5]
(8.8,15.9]
0.0
0.2
0.4
0.6
0.8
1.0

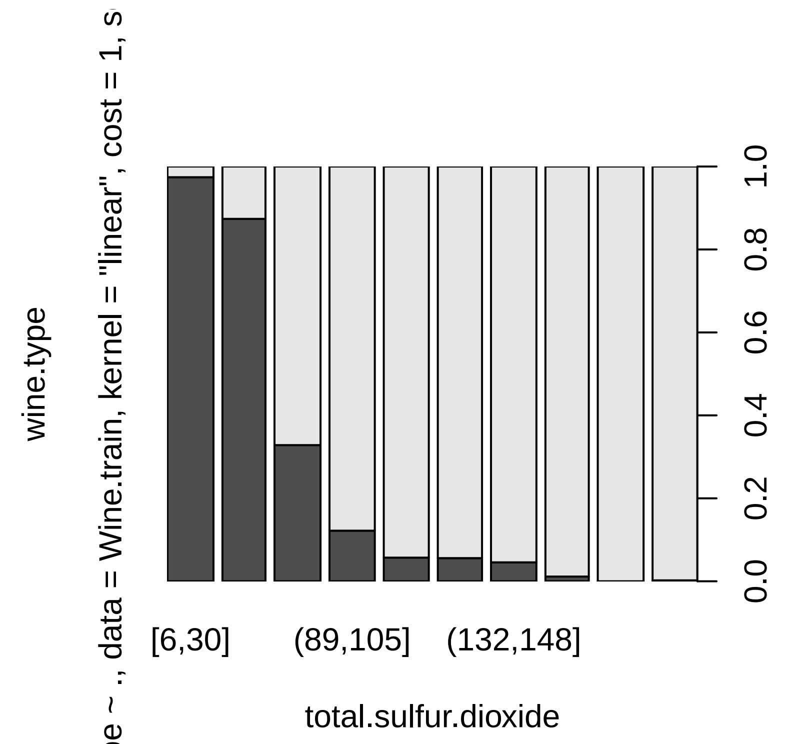

type ~ ., data = Wine.train, kernel = "linear", cost = 1, s
wine.type
volatile.acidity
[0.08,0.18]
(0.27,0.29]
(0.45,0.59]
0.0
0.2
0.4
0.6
0.8
1.0

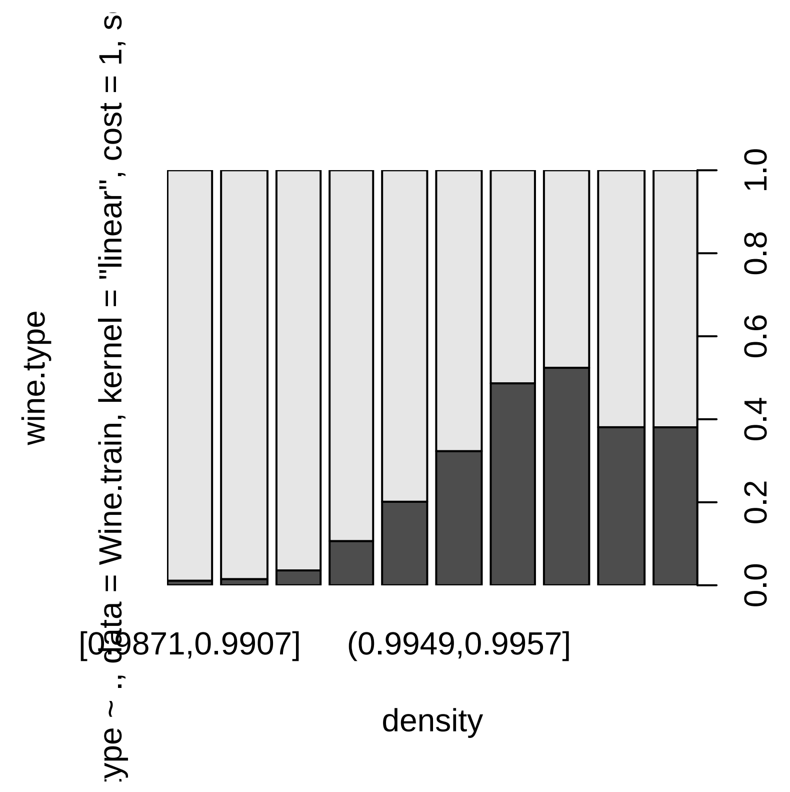

ine.type ~ ., data = Wine.train, kernel = "linear", cost =
wine.type
1.0
0.8
0.6
0.4
0.2
0.0
[0,0.14]
(0.26,0.29]
(0.37,0.42]
citric.acid

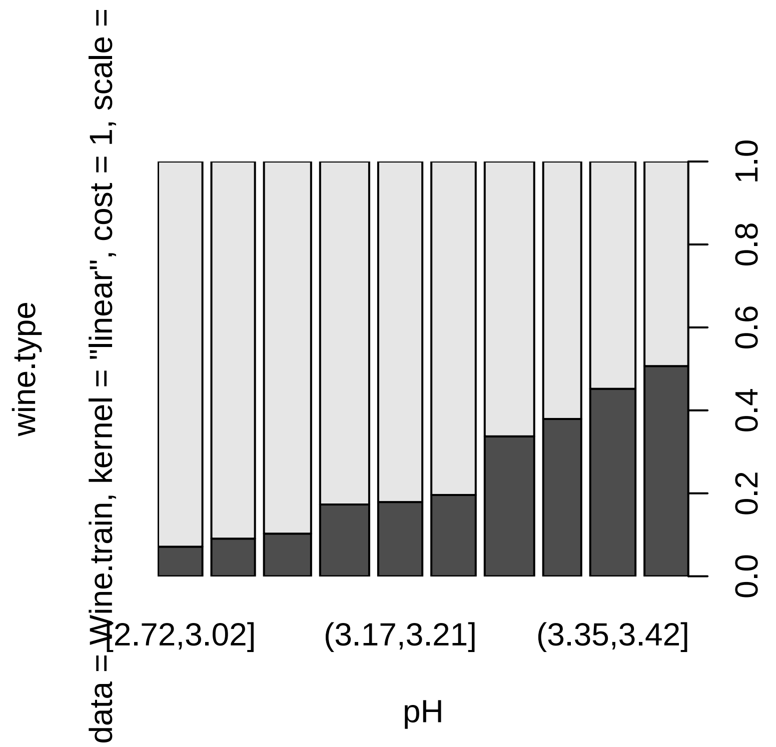

ype ~ ., data = Wine.train, kernel = "linear", cost = 1, s
wine.type
1.0
0.8
0.6
0.4
0.2
0.0
[0.6,1.3]
(2,2.3]
(5,7.1]
(13,65.8]
residual.sugar

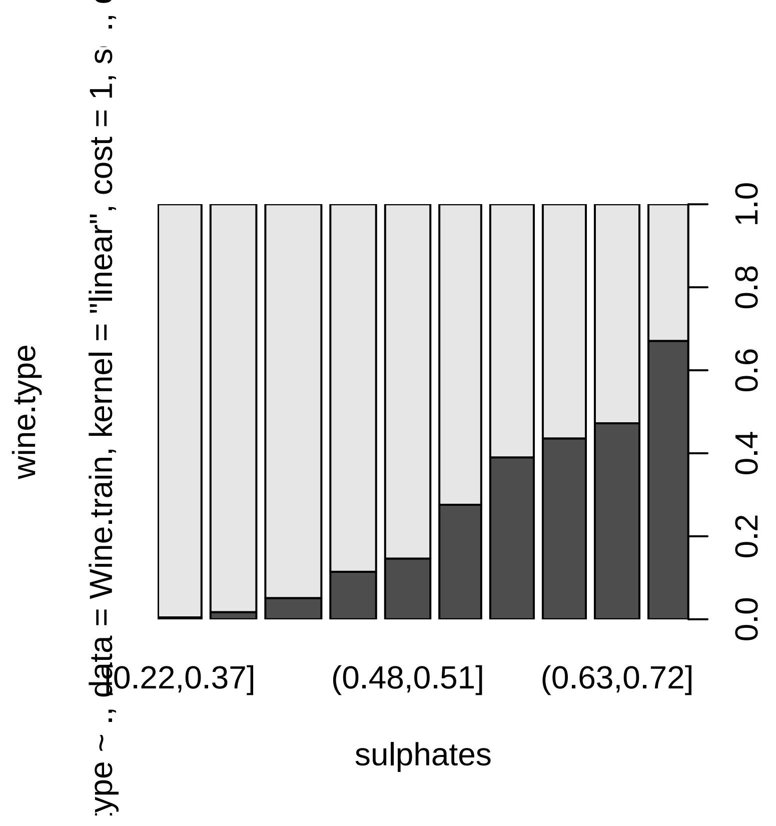

wine.type
data = Wine.train, kernel = "linear", cost = 1, s
[0.009,0.031]
(0.044,0.047]
(0.086,0.611]
chlorides
0.0
0.2
0.4
0.6
0.8
1.0

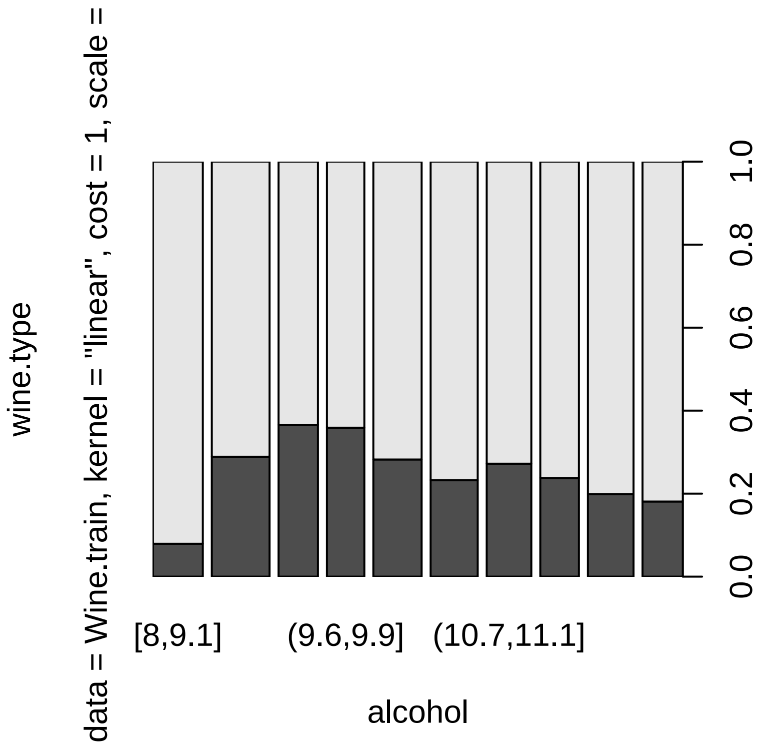

wine.type
ne.type ~ ., data = Wine.train, kernel = "linear", cost =
[1,9]
(15,19]
(29,33]
(45,54]
free.sulfur.dioxide
0.0
0.2
0.4
0.6
0.8
1.0

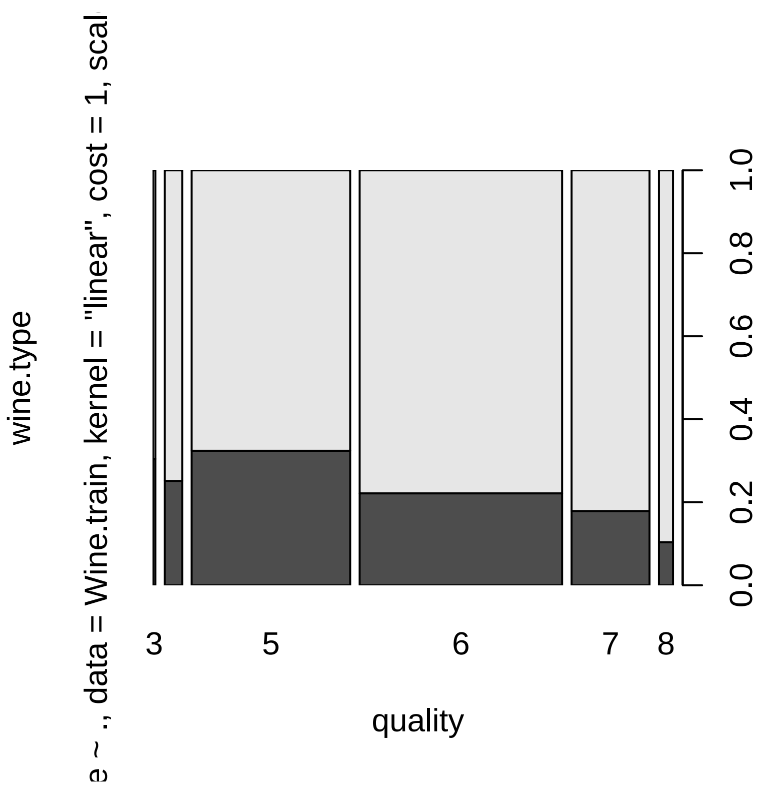

ype ~ ., data = Wine.train, kernel = "linear", cost = 1, s
wine.type
total.sulfur.dioxide
[6,30]
(89,105]
(132,148]
0.0
0.2
0.4
0.6
0.8
1.0


ype ~ ., data = Wine.train, kernel = "linear", cost = 1, s
wine.type
density
[0.9871,0.9907]
(0.9949,0.9957]
0.0
0.2
0.4
0.6
0.8
1.0


wine.type
Wine.train, kernel = "linear", cost = 1, scale =
[2.72,3.02]
(3.17,3.21]
(3.35,3.42]
pH
0.0
0.2
0.4
0.6
0.8
1.0


wine.type
ype ~ ., data = Wine.train, kernel = "linear", cost = 1, s
[0.22,0.37]
(0.48,0.51]
(0.63,0.72]
sulphates
0.0
0.2
0.4
0.6
0.8
1.0





```
# Tuning gamma and cost on SVM
svm.tuning ← tune.svm(wine.type ~ ., data = Total_Wine, gamma = 10^(−6:−3),
    cost = 10^(1:2))
svm.tuning$best.parameters
```

```
  gamma cost
8 0.001  100
```

```
svm.tuning.2 ← tune.svm(wine.type ~ ., data = Total_Wine, gamma = 10^(−3:2),
    cost = 10^(2:4))

# Optmal SVM
opt.svm ← svm(wine.type ~ ., data = Wine.train, kernel = "linear", gamma =
    svm.tuning.2$best.parameters$gamma,
    cost = svm.tuning.2$best.parameters$cost)

# Cross Validation of SVM
svm.kflval ← function(k, data) {
    folds = createfolds(nrow(data), k)
    accvector = 1:k
    for (k in 1:k) {
        temptrain = data[folds != k, ]
        temptest = data[folds == k, ]
        tempmodel = svm(wine.type ~ ., data = Wine.train, kernel = "linear",
            gamma = svm.tuning.2$best.parameters$gamma, cost = svm.tuning.2$best.parameters$cost)
        temppred = predict(tempmodel, newdata = temptest)
        analysis = confusion(temptest$wine.type, temppred)
        accvector[k] = analysis$acc
    }
    return(mean(accvector))
}

svm.kflval(10, Total_Wine)
```

```
Error in predict.svm(tempmodel, newdata = temptest): test data does not match model !
```

```
# Neural Networks
wine.nnet ← nnet(wine.type ~ ., data = Wine.test, size = 10, linout = FALSE,
    maxit = 500)
```

```
Error in nnet.default(x, y, w, entropy = TRUE, ...): too many (1071) weights
```

```
pred.nnet ← predict(wine.nnet, newdata = Wine.test, type = "class")
```

```
Error in predict(wine.nnet, newdata = Wine.test, type = "class"): object 'wine.nnet' not found
```

```
confusion(Wine.test$wine.type, pred.nnet)
```

```
Error in table(true, pred): object 'pred.nnet' not found
```

```
nnet.tuning ← tune.nnet(wine.type ~ ., data = Total_Wine, size = 1:15, trace = FALSE,
    tunecontrol = tune.control(nrepeat = 5, sampling = "cross", cross = 10,
        ))
1 − mean(nnet.tuning$performances$error)
```

```
[1] 0.9754159
```

```
# Plotting Neaural Networks import the function from Github
library(devtools)
source_url("https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1a4684a5/nnet_
```

```
SHA-1 hash of file is 74c80bd5ddbc17ab3ae5ece9c0ed9beb612e87ef
```

```
plot.nnet(wine.nnet)
```

```
Loading required package: scales
```

```


Attaching package: 'scales'
```

```
The following object is masked from 'package:kernlab':

    alpha
```

```
Error in match(x, table, nomatch = 0L): object 'wine.nnet' not found
```