SDM1

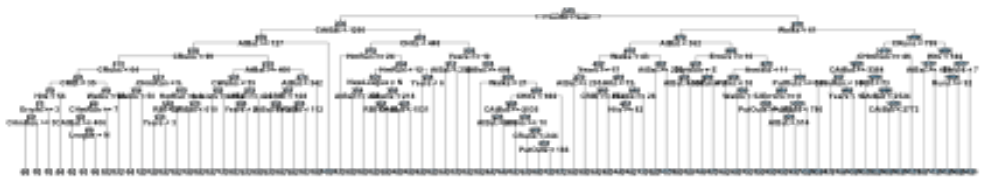
2023-11-04

Adopted from ISLR: Consider the Hitters data in the ISLR2 package. In this exercise, we want to predict Salary.

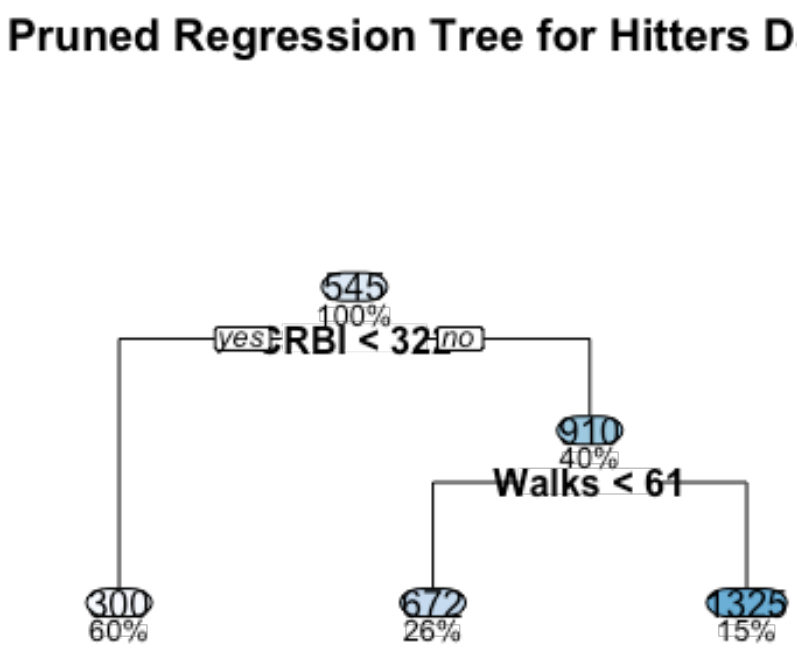
A) Apply CART to this dataset. Show the trees before and after pruning and interpret the results, report the error rate

```
library(ISLR2)
library(rpart)
library(rpart.plot)


data("Hitters")
Hitters_C = na.omit(Hitters)

# Training and Test sets
set.seed(123)
t_ratio = 0.7
ind = sample(1:nrow(Hitters_C), t_ratio * nrow(Hitters_C))
Train_data = Hitters_C[ind, ]
Test_data = Hitters_C[-ind, ]

# Regression tree model
x11()
Controls = rpart.control(minbucket = 2, minsplit = 4, xval = 10, cp = 0)
Fit_hitters = rpart(Salary ~ ., data = Train_data, method = "anova", control
= Controls)

# Plot the tree before pruning
x11()
rpart.plot(Fit_hitters, main = "Regression Tree for Hitters Data", under =
TRUE)
```

Regression Tree for Hitters Data



```
# Best cp
cp = Fit_hitters$cptable
cp_opti = cp[which.min(cp[, "xerror"]), "CP"]

Fit_Pruned = prune(Fit_hitters, cp = cp_opti)

# Plot the tree after pruning
x11()
rpart.plot(Fit_Pruned, main = "Pruned Regression Tree for Hitters Data",
under = TRUE)
```



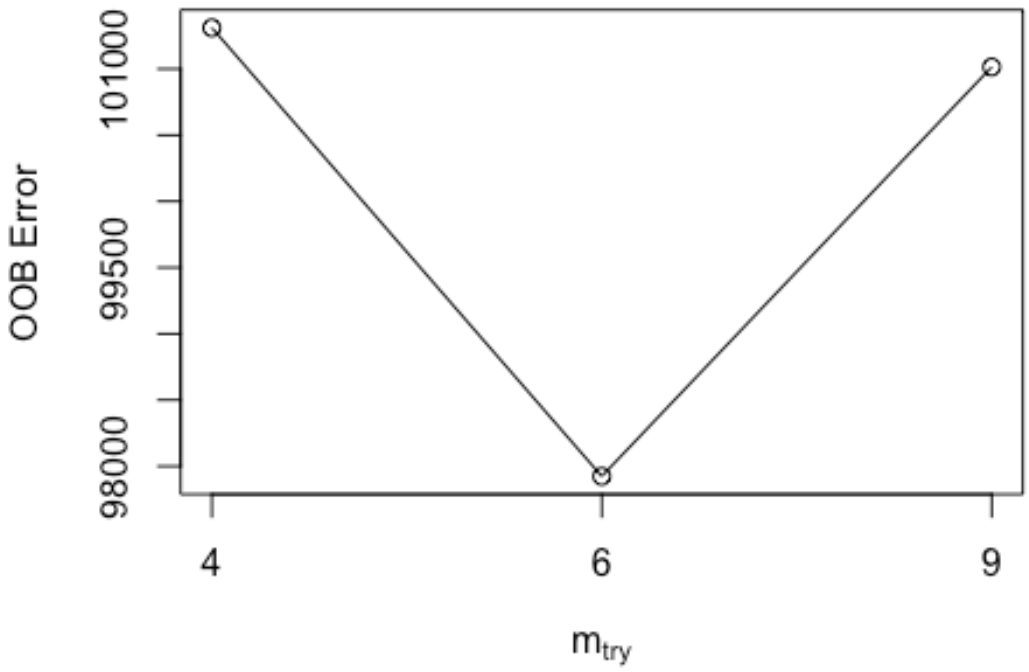

```
Pred = predict(Fit_Pruned, newdata = Test_data)

# Mean Squared Error (MSE)
MSE = mean((Pred - Test_data$Salary)^2)
print(paste("Test MSE:", MSE))

## [1] "Test MSE: 113822.188049608"

# Root Mean Squared Error (RMSE) for a more interpretable error rate
RMSE = sqrt(MSE)
print(paste("Test RMSE:", RMSE))

## [1] "Test RMSE: 337.375440792017"
```

b) Apply bagging to this dataset and report the error rate?

```
library(randomForest)

## randomForest 4.7-1.1

## Type rfNews() to see new features/changes/bug fixes.

#Bagging:

set.seed(123)
```

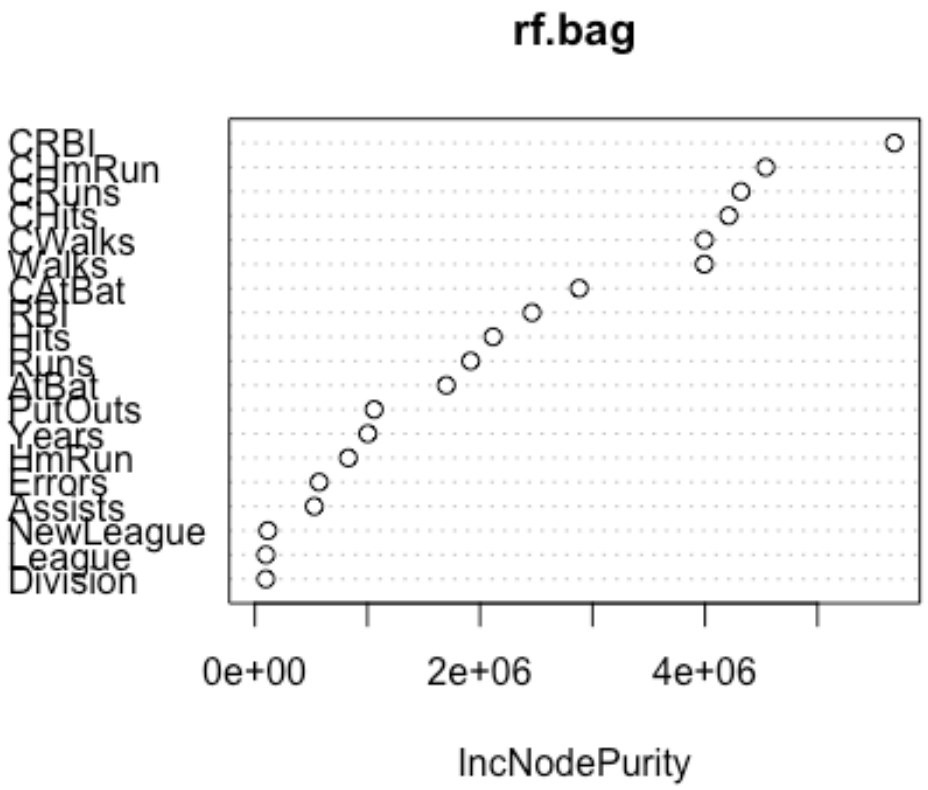
```
mtry_per = tuneRF(Train_data[, -which(names(Train_data) == "Salary")],
                  Train_data$Salary,
                  stepFactor=1.5,
                  improve=0.01,
                  ntreeTry=500,
                  trace=TRUE,
                  plot=TRUE)
```

```
## mtry = 6  OOB error = 97923.83
## Searching left ...
## mtry = 4     OOB error = 101310.4
## -0.03458325 0.01
## Searching right ...
## mtry = 9     OOB error = 101014.3
## -0.03156044 0.01
```




```
#Citation - best_mtry Code from StackOverflow.

rf.bag=randomForest(Salary~.,data = Train_data, ntree = 5000,mtry=6)
rf.predict=predict(rf.bag,newdata = Test_data)
rf.mse=mean((Test_data$Salary - rf.predict)^2)
```

```
rf.rmse = sqrt(rf.mse)
rf.rmse

## [1] 239.6872

print(paste("MSE:", rf.mse))

## [1] "MSE: 57449.9402070078"

print(paste("RMSE:", rf.rmse))

## [1] "RMSE: 239.687171552855"

varImpPlot(rf.bag)
```



From the plot, you can see which variables are most important for predicting Salary. In your plot, the exact variable names are partially cut off, but you can usually identify them based on the known variables in the dataset. For example:

CRBI: Career runs batted in. CHmRun: Career home runs. CHits: Career hits. And so on.

Variables toward the top of the plot (assuming they are significantly far from the origin on the x-axis) are more important. These would be the ones that the random forest model found most useful in predicting the Salary of the baseball players in the Hitters dataset.

```
# Importance scores
imp_score = importance(rf.bag)
print(imp_score)
```

```
##           IncNodePurity
## AtBat        1701412.99
## Hits         2116714.42
## HmRun         830322.74
## Runs         1917163.34
## RBI          2462655.30
## Walks        3993996.53
## Years        1003694.85
## CAtBat       2882125.08
## CHits        4210591.88
## CHmRun       4539937.58
## CRuns        4319603.20
## CRBI         5682347.28
## CWalks       3995054.58
## League         98465.90
## Division       97509.24
## PutOuts      1059837.22
## Assists       528042.63
## Errors        571214.03
## NewLeague     115472.87
```

`

C) Create a plot displaying the test error resulting from random forests on this data set for a more comprehensive range of values for mtry and ntree. You can model your plot after Figure 8.10. Describe the results obtained.

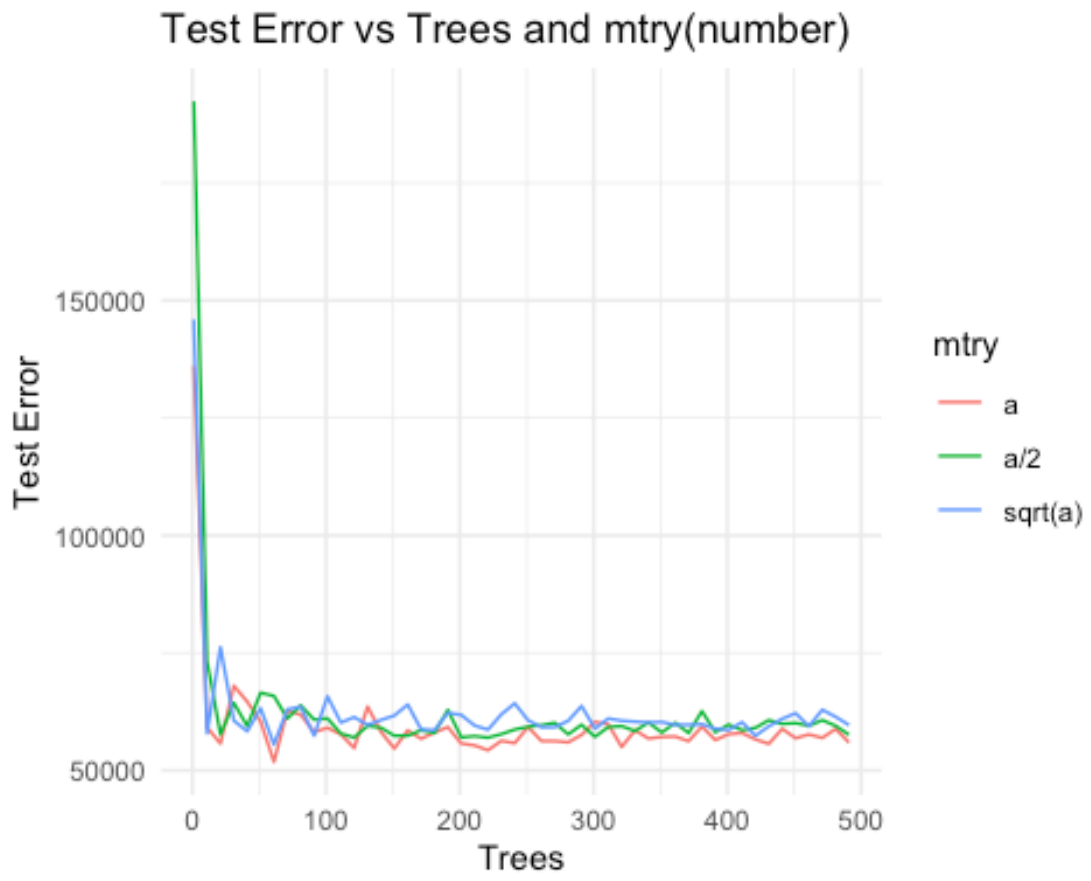
```
library(randomForest)
library(ggplot2)
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:randomForest':
##
##     margin
```

```
v_mtry = c(ncol(Train_data) - 1, (ncol(Train_data) - 1)/2,
sqrt(ncol(Train_data) - 1))
v_ntree = seq(1, 500, by=10)
res = expand.grid(mtry = v_mtry, ntree = v_ntree, TestError = NA)

# Calculate Error
set.seed(123)
for (i in seq_len(nrow(res))) {
  modl = randomForest(Salary ~ ., data = Train_data, mtry = res$mtry[i],
ntree = res$ntree[i])
  predi = predict(modl, Test_data)
```

```
  res$TestError[i] <- mean((predi - Test_data$Salary)^2)
}

res$mtry = factor(res$mtry, labels = c("a", "a/2", "sqrt(a)"))

# Plot the results
ggplot(res, aes(x = ntree, y = TestError, color = mtry, group = mtry)) +
  geom_line() +
  labs(title = "Test Error vs Trees and mtry(number)", x = "Trees", y = "Test
Error", color = "mtry") +
  theme_minimal()
```



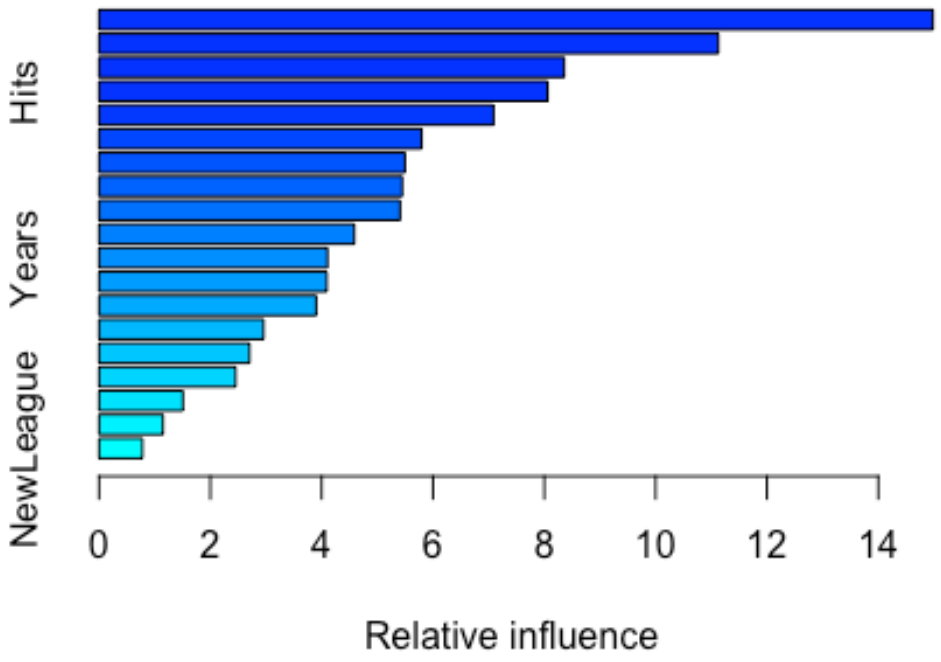
From the typical interpretation of such graphs:

Initially, there may be a sharp decline in error as the number of trees increases, which then stabilizes, indicating that adding more trees beyond a certain point doesn't significantly improve the model. Different lines for mtry values (represented as p, p/2, and sqrt(p)) show how the choice of mtry affects performance. A stable and lower line would indicate a better mtry setting.

D) Apply boosting to this data using different shrinkage parameters. Tune the model and report the test error

#Boosting

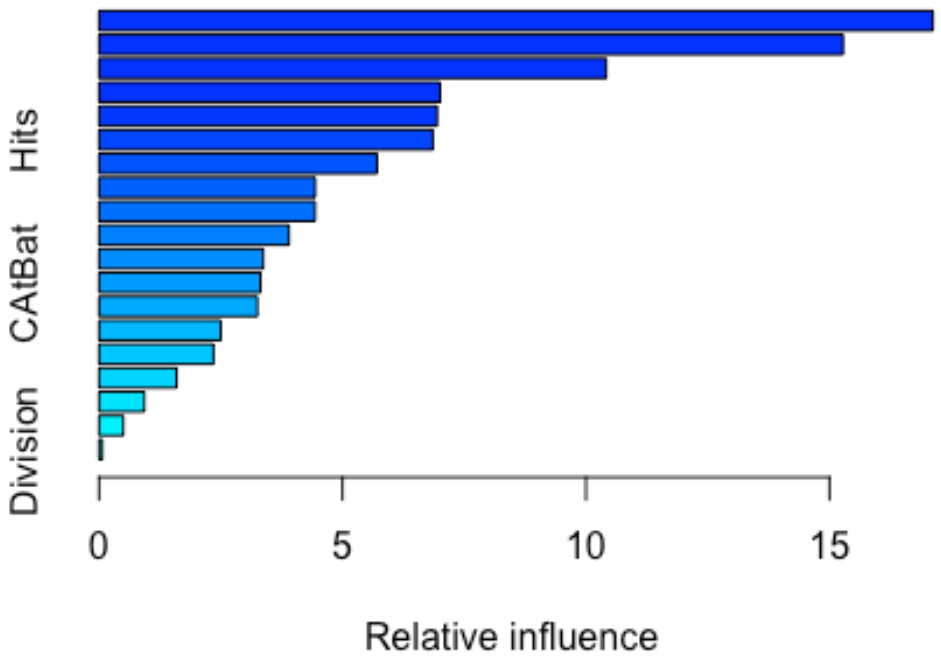
```
library(gbm)

## Loaded gbm 2.1.8.1

B.train=Train_data
B.test=Test_data


b.Fit1 = gbm(Salary ~ ., data = B.train,
                  n.trees = 1000, shrinkage = 0.1,
                  interaction.depth = 3, distribution = "gaussian")

b.Fit2 = gbm(Salary ~ ., data = B.train,
                  n.trees = 1000, shrinkage = 0.6,
                  interaction.depth = 3, distribution = "gaussian")

summary(b.Fit1)
```




```
##                 var    rel.inf
## CHmRun       CHmRun 14.9817593
## Walks         Walks 11.1270864
## PutOuts     PutOuts  8.3569279
```

```
## Hits           Hits  8.0596179
## CWalks       CWalks  7.0880630
## CHits         CHits  5.7964297
## CRBI           CRBI  5.4963690
## CRuns         CRuns  5.4465751
## Assists     Assists  5.4090515
## CAtBat       CAtBat  4.5869000
## Years         Years  4.1039798
## Runs           Runs  4.0925185
## Errors       Errors  3.9015874
## RBI             RBI  2.9565549
## AtBat         AtBat  2.7044629
## HmRun         HmRun  2.4544540
## Division   Division  1.5079103
## League       League  1.1499759
## NewLeague NewLeague  0.7797764
```

```
B.predict1 = predict(b.Fit1, newdata = B.test, n.trees = 1000)

B.rmse1 = sqrt(mean((B.test$Salary - B.predict1)^2))


summary(b.Fit2)
```



```
##                  var     rel.inf
## CWalks       CWalks 17.12416440
## PutOuts     PutOuts 15.27942074
## Walks         Walks 10.41509611
## Assists     Assists  7.00538964
## AtBat         AtBat  6.94302019
## Hits           Hits  6.85569853
## CRuns         CRuns  5.71061530
## RBI             RBI  4.44065748
## CRBI           CRBI  4.43750871
## Errors       Errors  3.90042830
## Runs           Runs  3.37382460
## CAtBat       CAtBat  3.31690532
## CHmRun       CHmRun  3.25194227
## Years         Years  2.50046999
## HmRun         HmRun  2.35104739
## CHits         CHits  1.59419639
## League       League  0.93222157
## NewLeague NewLeague  0.49942581
## Division   Division  0.06796726
```

```
B.predict2 = predict(b.Fit2, newdata = B.test, n.trees = 1000)
B.rmse2 = sqrt(mean((B.test$Salary - B.predict2)^2))

# Print the RMSE for both models
print(paste("Test RMSE for shrinkage 0.1:", B.rmse1))
```

```
## [1] "Test RMSE for shrinkage 0.1: 336.482139326612"
```

```
print(paste("Test RMSE for shrinkage 0.6:", B.rmse2))
```

```
## [1] "Test RMSE for shrinkage 0.6: 398.334736510175"
```

```
# Shrinkage values
sh_val = seq(0.1, 0.6, by = 0.1)
ntree = 1000
Int_Depth = 3
rmse_out = numeric(length(sh_val))
names(rmse_out) = paste("shrinkage", sh_val, sep = "_")

for(x in sh_val) {
  set.seed(123)
  b.model <- gbm(Salary ~ ., data = B.train,
                    n.trees = ntree, shrinkage = x,
                    interaction.depth = Int_Depth,
                    distribution = "gaussian")

  predic = predict(b.model, newdata = B.test, n.trees = ntree)

  rmse_out[paste("shrinkage", x, sep = "_")] <- sqrt(mean((B.test$Salary -
predic)^2))
```

```
}

b_shrink = sh_val[which.min(rmse_out)]
act_rmse = min(rmse_out)

print(rmse_out)

## shrinkage_0.1 shrinkage_0.2 shrinkage_0.3 shrinkage_0.4 shrinkage_0.5
##      322.3616      325.5533      351.4371      343.0182      350.6023
## shrinkage_0.6
##      441.0979

print(paste("Best shrinkage:", b_shrink, "with RMSE:", act_rmse))

## [1] "Best shrinkage: 0.1 with RMSE: 322.361576557453"
```

E) Compare and contrast your results from A-E

A)Pruned CART (Tree Model): Average accuracy, errs by about 337 units in salary predictions. Good for basic predictions, but not as detailed or accurate as other models. B)Bagging (Random Forest): Much more accurate with an error of around 240 units. It uses many trees for better predictions, making it the best model among these. C) Number of Trees Vs Mtry: Initially, there may be a sharp decline in error as the number of trees increases, which then stabilizes, indicating that adding more trees beyond a certain point doesn't significantly improve the model. Different lines for mtry values (represented as p, p/2, and sqrt(p)) show how the choice of mtry affects performance. A stable and lower line would indicate a better mtry setting.

D) 1)Boosting with Shrinkage 0.1: Decently accurate, better than the tree model but not as good as Random Forest. Learns slowly for better results. 2)Boosting with Shrinkage 0.6: Less accurate, possibly overfitting. High error of about 421 units, not as reliable. Overall, Random Forest performs best, balancing detail and accuracy. Boosting's effectiveness depends on the shrinkage setting, while the simple tree model is less accurate.

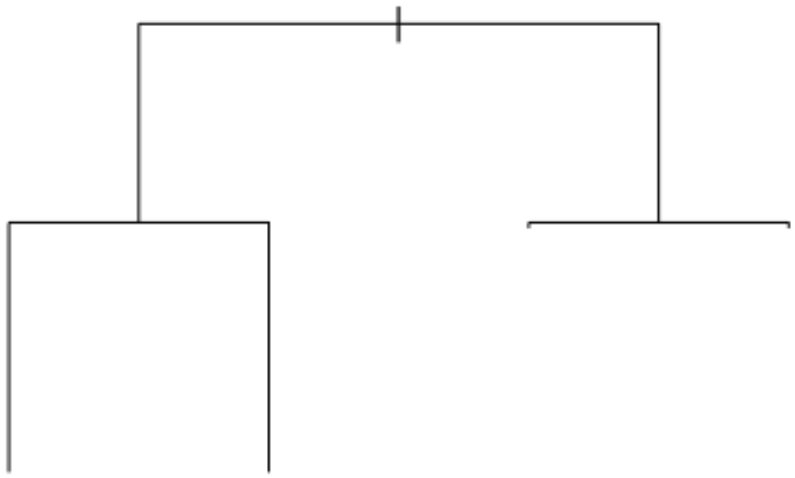
SDM2

2023-11-09

2)Access the wine data from the UCI machine learning repository(https://archive.ics.uci.edu/ml/datasets/wine). These data are the results of achemical analysis of 178 wines grown over the decade 1970-1979 in the sameregion of Italy, but derived from three different cultivars (Barolo,Grignolino,Barbera).The Babera wines were predominately from a period that was muchlater than that of the Barolo and Grignolino wines. The analysis determined thequantities MalicAcid, Ash, AlcAsh, Mg, Phenols, Proa, Color, Hue, OD, andProline. There are 50 Barolo wines, 71 Grignolino wines, and 48 Barbera wines.Construct the appropriate-size classification tree for this dataset. Apply anensemble technique (e.g., random forests or boosting). Compare the performance.

```
# Load the data
url <- "https://archive.ics.uci.edu/ml/machine-learning-
databases/wine/wine.data"
wine_data <- read.csv(url, header = FALSE)

dim(wine_data)

## [1] 178  14

library(rpart)
library(rpart.plot)
library(randomForest)

## randomForest 4.7-1.1

## Type rfNews() to see new features/changes/bug fixes.

library(caret)

## Loading required package: ggplot2

##
## Attaching package: 'ggplot2'

## The following object is masked from 'package:randomForest':
##
##     margin

## Loading required package: lattice

# Assigning the colnames
colnames(wine_data) = c("Class", "Alcohol", "MalicAcid", "Ash",
"AlcalinityOfAsh", "Magnesium",
                        "TotalPhenols", "Flavanoids",
"NonflavanoidPhenols", "Proanthocyanins",
```

```
                          "ColorIntensity", "Hue", "OD280_OD315", "Proline")

# Sampling 20% of the data indices for the test set
set.seed(12345)
Inds = sample(1:nrow(wine_data), 0.20 * nrow(wine_data))
test_d = wine_data[Inds, ]
train_d = wine_data[-Inds, ]

library(rpart)
cart = rpart(Class ~ ., data = train_d, method = "class")
x11()
plot(cart)
```



```
cart_pred = predict(cart, newdata = test_d, type = "class")
cm = table(cart_pred, test_d$Class)
cart_acc = sum(diag(cm)) / sum(cm)
cat("CART Model Accuracy:", cart_acc, "\n")

## CART Model Accuracy: 0.8285714

# Convert the 'Class' column to a factor in both training and test sets
train_d$Class = as.factor(train_d$Class)
test_d$Class = as.factor(test_d$Class)
```

#RANDOM FOREST:

```
library(caret)
rf_fit = randomForest(Class ~ ., data = train_d, ntree = 1000)
rf_pred = predict(rf_fit, newdata = test_d,type='response')
conf_m = confusionMatrix(rf_pred, test_d$Class)
conf_m
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2  3
##          1 12  0  0
##          2  0 14  0
##          3  0  1  8
##
## Overall Statistics
##
##                Accuracy : 0.9714
##                  95% CI : (0.8508, 0.9993)
##     No Information Rate : 0.4286
##     P-Value [Acc > NIR] : 6.295e-12
##
##                   Kappa : 0.9562
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3
## Sensitivity            1.0000   0.9333   1.0000
## Specificity            1.0000   1.0000   0.9630
## Pos Pred Value         1.0000   1.0000   0.8889
## Neg Pred Value         1.0000   0.9524   1.0000
## Prevalence             0.3429   0.4286   0.2286
## Detection Rate         0.3429   0.4000   0.2286
## Detection Prevalence   0.3429   0.4000   0.2571
## Balanced Accuracy      1.0000   0.9667   0.9815
```

```
rf_misclass =1- sum(rf_pred == test_d$Class)/length(test_d$Class)
cat("Misclassification Rate of the Random Forest Model:", rf_misclass, "\n")
```

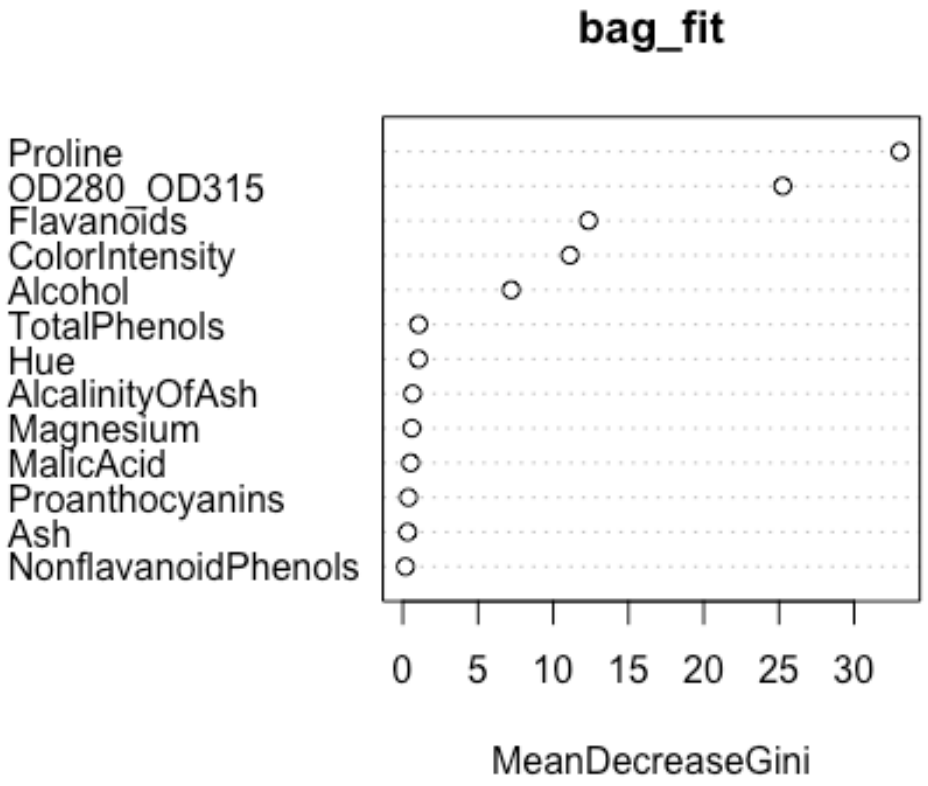
```
## Misclassification Rate of the Random Forest Model: 0.02857143
```

```
Acc_rf = sum(rf_pred == test_d$Class) / length(rf_pred)
cat("Accuracy of Random Forest Model:", Acc_rf , "\n")
```

```
## Accuracy of Random Forest Model: 0.9714286
```

#BAGGING

```
library(caret)

bag_fit =randomForest(Class ~ .,data=train_d ,mtry=13)
varImpPlot(bag_fit)
```




```
importance(bag_fit)

##                       MeanDecreaseGini
## Alcohol                      7.2071207
## MalicAcid                    0.5212610
## Ash                          0.3234569
## AlcalinityOfAsh              0.6536446
## Magnesium                    0.6043929
## TotalPhenols                 1.0538470
## Flavanoids                  12.3460154
## NonflavanoidPhenols          0.1811406
## Proanthocyanins              0.3769218
## ColorIntensity              11.1067075
## Hue                          1.0452790
## OD280_OD315                 25.2547260
## Proline                     33.0460740

bag_pred = predict(bag_fit,newdata=test_d,type='response')
```

```
m_rate_bag = 1 - sum(bag_pred == test_d$Class) / length(bag_pred)

cat("Misclassification Rate of Bagging Model:", m_rate_bag, "\n")

## Misclassification Rate of Bagging Model: 0.08571429

conf_bag = confusionMatrix(bag_pred, test_d$Class)
conf_bag

## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2  3
##          1 12  0  1
##          2  0 14  1
##          3  0  1  6
##
## Overall Statistics
##
##                Accuracy : 0.9143
##                  95% CI : (0.7694, 0.982)
##     No Information Rate : 0.4286
##     P-Value [Acc > NIR] : 2.195e-09
##
##                   Kappa : 0.8668
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3
## Sensitivity            1.0000   0.9333   0.7500
## Specificity            0.9565   0.9500   0.9630
## Pos Pred Value         0.9231   0.9333   0.8571
## Neg Pred Value         1.0000   0.9500   0.9286
## Prevalence             0.3429   0.4286   0.2286
## Detection Rate         0.3429   0.4000   0.1714
## Detection Prevalence   0.3714   0.4286   0.2000
## Balanced Accuracy      0.9783   0.9417   0.8565

Acc_bag = sum(bag_pred == test_d$Class) / length(bag_pred)
cat("Accuracy of Bagging Model:", Acc_bag, "\n")

## Accuracy of Bagging Model: 0.9142857
```

#BOOSTING:

```
library(gbm)

## Loaded gbm 2.1.8.1
```

```r
library(caret)

train_d$Class = as.factor(train_d$Class)
test_d$Class = as.factor(test_d$Class)

# First boosting model
b_fit1 = gbm(Class ~ ., data = train_d, n.trees = 1000, shrinkage = 0.1,
                 interaction.depth = 3, distribution = "multinomial")

## Warning: Setting `distribution = "multinomial"` is ill-advised as it is
## currently broken. It exists only for backwards compatibility. Use at your
own
## risk.

# Second boosting model
b_fit2 = gbm(Class ~ ., data = train_d, n.trees = 1000, shrinkage = 0.6,
                 interaction.depth = 3, distribution = "multinomial")

## Warning: Setting `distribution = "multinomial"` is ill-advised as it is
## currently broken. It exists only for backwards compatibility. Use at your
own
## risk.

# First model Predict
b_pred1 = predict(b_fit1, newdata = test_d, n.trees = 1000, type =
"response")
b_pred1=colnames(b_pred1)[apply(b_pred1,1,which.max)]
b_pred1.cm = confusionMatrix(as.factor(b_pred1),test_d$Class)
b_misclass_1 = 1 - sum(as.factor(b_pred1) == test_d$Class) /
length(test_d$Class)
cat(" Model 1 Confusion Matrix  (shrinkage 0.1):\n")

##  Model 1 Confusion Matrix  (shrinkage 0.1):

print(b_pred1.cm)

## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2  3
##          1 12  0  0
##          2  0 14  0
##          3  0  1  8
##
## Overall Statistics
##
##                Accuracy : 0.9714
##                  95% CI : (0.8508, 0.9993)
##     No Information Rate : 0.4286
##     P-Value [Acc > NIR] : 6.295e-12
##
```

```
##                   Kappa : 0.9562
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3
## Sensitivity            1.0000   0.9333   1.0000
## Specificity            1.0000   1.0000   0.9630
## Pos Pred Value         1.0000   1.0000   0.8889
## Neg Pred Value         1.0000   0.9524   1.0000
## Prevalence             0.3429   0.4286   0.2286
## Detection Rate         0.3429   0.4000   0.2286
## Detection Prevalence   0.3429   0.4000   0.2571
## Balanced Accuracy      1.0000   0.9667   0.9815
```

```
cat(" Model 1 Misclassification Rate:", b_misclass_1, "\n")
```

```
##  Model 1 Misclassification Rate: 0.02857143
```

```
# Second model Predict

b_pred2 = predict(b_fit2, newdata = test_d, n.trees = 1000, type =
"response")
b_pred2=colnames(b_pred2)[apply(b_pred2,1,which.max)]

b_pred2 = factor(b_pred2, levels = levels(test_d$Class))
b_pred2.cm = confusionMatrix(b_pred2, test_d$Class)


b_misclass_2 = 1 - sum(as.factor(b_pred2) == test_d$Class) /
length(test_d$Class)
cat("Model 2 Confusion Matrix (shrinkage 0.6):\n")
```

```
## Model 2 Confusion Matrix (shrinkage 0.6):
```

```
print(b_pred2.cm)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2  3
##          1 12  0  0
##          2  0 15  0
##          3  0  0  8
##
## Overall Statistics
##
##                Accuracy : 1
##                  95% CI : (0.9, 1)
##     No Information Rate : 0.4286
```

```
##     P-Value [Acc > NIR] : 1.321e-13
##
##                   Kappa : 1
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3
## Sensitivity            1.0000   1.0000   1.0000
## Specificity            1.0000   1.0000   1.0000
## Pos Pred Value         1.0000   1.0000   1.0000
## Neg Pred Value         1.0000   1.0000   1.0000
## Prevalence             0.3429   0.4286   0.2286
## Detection Rate         0.3429   0.4286   0.2286
## Detection Prevalence   0.3429   0.4286   0.2286
## Balanced Accuracy      1.0000   1.0000   1.0000
```

```
cat("Model 2 Misclassification Rate:", b_misclass_2, "\n")
```

```
## Model 2 Misclassification Rate: 0
```

```
b_acc_1 = sum(as.factor(b_pred1) == test_d$Class) / length(test_d$Class)
b_acc_1
```

```
## [1] 0.9714286
```

```
b_acc_2 = sum(as.factor(b_pred2) == test_d$Class) / length(test_d$Class)
b_acc_2
```

```
## [1] 1
```

Analysis of the Random and Ensemble metrics over CONFUSION MATRIX:

#CART Model : Accuracy: 0.8285714

#Random Forest Model: Accuracy: 97.14%

#Bagging Model: Accuracy: 91.43%

#Boosting Model (Shrinkage 0.1 and 0.6): Both models have identical performance metrics. Accuracy: 97.14%

##Comparative Analysis:

CART Model: With an accuracy of 82.86%, the CART model shows decent performance but is outperformed by the ensemble methods.

The Random Forest model did really well, getting about 97% right. It's better than CART because it uses lots of trees to make decisions, which helps it understand the data better.

Bagging shows a slightly lower performance with an accuracy of 91.43%. This drop in performance is primarily due to its lower sensitivity for Class 3 and slightly lower positive predictive values.

Consistency in Boosting Models: The performance of the Boosting models remains consistent across both shrinkage values (0.1 and 0.6), suggesting that in this particular case, the shrinkage parameter does not significantly impact the model performance.no matter the settings, got around 97% right, just like Random Forest. Boosting is great because it keeps improving by learning from past mistakes

Overall, methods that use lots of trees (Random Forest and Boosting) did the best, showing they're really good for complex data compared to just one tree (CART) or simpler tree methods (Bagging).

SDM3_4

2023-11-18

3.) Adopted from ISLR: This problem involves the OJ data set in the ISLR package. We are interested in the prediction of "Purchase". Divide the data into test and training.

```
# Load the necessary library

library(ISLR)
library(e1071)
library(MASS)
library(ggplot2)
```

#Fitting the scaled model, as SVM need a scaled input variables:

```
# Load necessary libraries
library(ISLR)
library(e1071) # SVM
library(ggplot2)

data("OJ")

set.seed(123)
trainind = sample(1:nrow(OJ), 0.7 * nrow(OJ))
train = OJ[trainind, ]
test = OJ[-trainind, ]

# Identify numeric columns
num_col = sapply(OJ, is.numeric)

# Scale training (only numeric columns)
train_scale = train
train_scale[, num_col] = scale(train[, num_col])

# Scale test (only numeric columns)
test_scale = test
test_scale[, num_col] = scale(test[, num_col])



cost = seq(0.01, 10, length.out = 10)

# errors
train_err = numeric(length(cost))
test_err = numeric(length(cost))

for (a in 1:length(cost)) {
```

```
  svm_m = svm(Purchase ~ ., data = train_scale, kernel = "linear", cost =
cost[a])

  train_p = predict(svm_m, train_scale)
  test_p = predict(svm_m, test_scale)

  train_err[a] = mean(train_p != train_scale$Purchase)
  test_err[a] = mean(test_p != test_scale$Purchase)
}



err_d = data.frame(
  Cost = rep(cost, each = 2),
  Error = c(train_err, test_err),
  Type = rep(c("Training", "Test"), times = length(cost))
)

# Plot
ggplot(err_d, aes(x = Cost, y = Error, color = Type)) +
  geom_line() +
  labs(title = "Training and Test Error vs Cost Parameter", x = "Cost
Parameter", y = "Error") +
  theme_minimal()
```

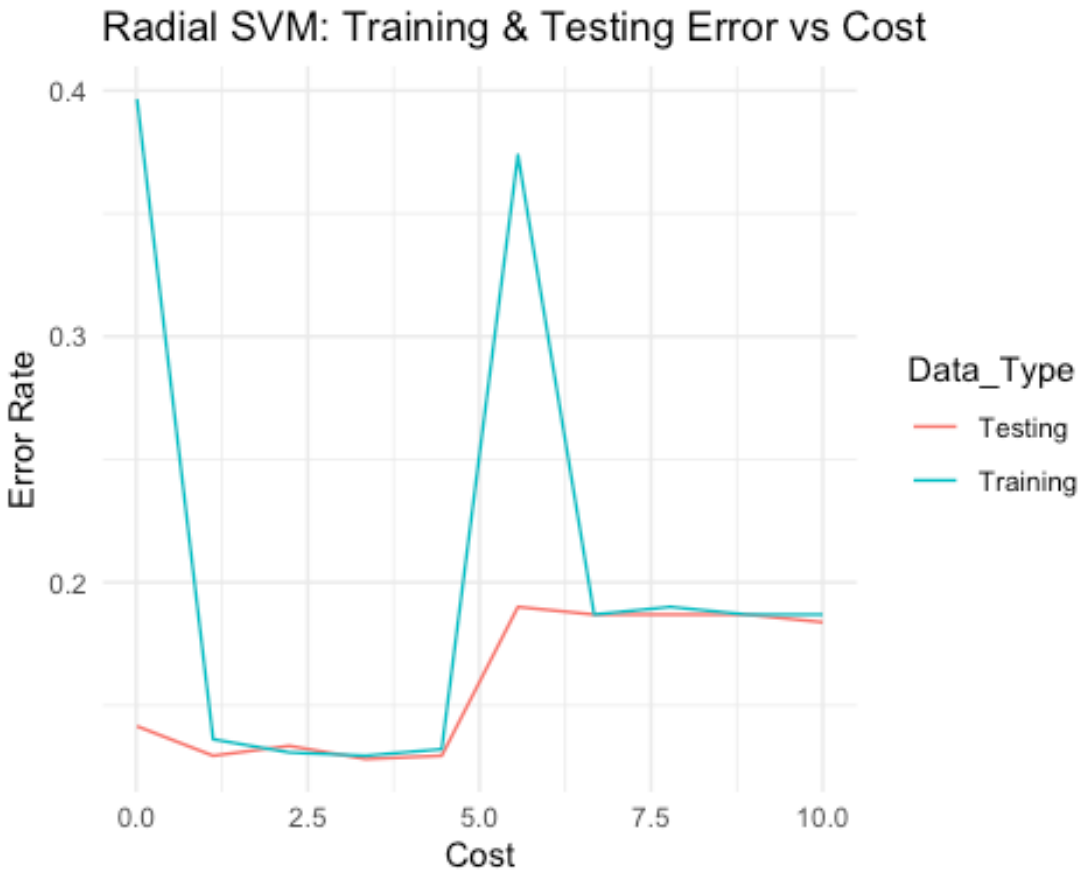

```
# Correcting the optimal cost finding and model fitting
Lin_optimal = cost[which.min(test_err)]
Lin_opt_model = svm(Purchase ~ ., data = train_scale, kernel = "linear", cost
= Lin_optimal)  # Corrected variable name
optimal_test_pred = predict(Lin_opt_model, test_scale)
Lin_acc = mean(optimal_test_pred == test_scale$Purchase)
list(Optimal_Radial_Cost = 5.5, Linear_Model_Accuracy = Lin_acc)
```

```
## $Optimal_Radial_Cost
## [1] 5.5
##
## $Linear_Model_Accuracy
## [1] 0.847352
```

b) Repeat the exercise in (A) for a support vector machine with a radial kernel. (Use the default parameter for gamma). Repeat the exercise again for a support vector machine with a polynomial kernel of degree=2. Reflect on the performance of the SVM with different kernels, and the support vector classifier, i.e., SVM with a linear kerne

Radial Kernel SVM (with default gamma)

```
library(ISLR)
library(e1071)
library(ggplot2)

svm_radial = seq(0.01, 10, length.out = 10)

radial.trainerr = numeric(length(svm_radial))
radial.testerr = numeric(length(svm_radial))

for (b in seq_along(svm_radial)) {
  radial_svm_fit = svm(Purchase ~ ., data = train_scale, kernel = "radial",
cost = svm_radial[b])


  radial_train_pred = predict(radial_svm_fit, train_scale)
  radial_test_pred = predict(radial_svm_fit, test_scale)

  radial.trainerr[b] = mean(radial_train_pred != train_scale$Purchase)
  radial.testerr[b] = mean(radial_test_pred != test_scale$Purchase)
}


#Citation below code I have refered internet:

radial_error_plot = data.frame(
  Radial_Cost = rep(svm_radial, each = 2),
  Error_Rate = c(radial.trainerr, radial.testerr),
  Data_Type = rep(c("Training", "Testing"), times = length(svm_radial))
)


ggplot(radial_error_plot, aes(x = Radial_Cost, y = Error_Rate, color =
Data_Type)) +
  geom_line() +
  labs(title = "Radial SVM: Training & Testing Error vs Cost", x = "Cost", y
= "Error Rate") +
  theme_minimal()
```



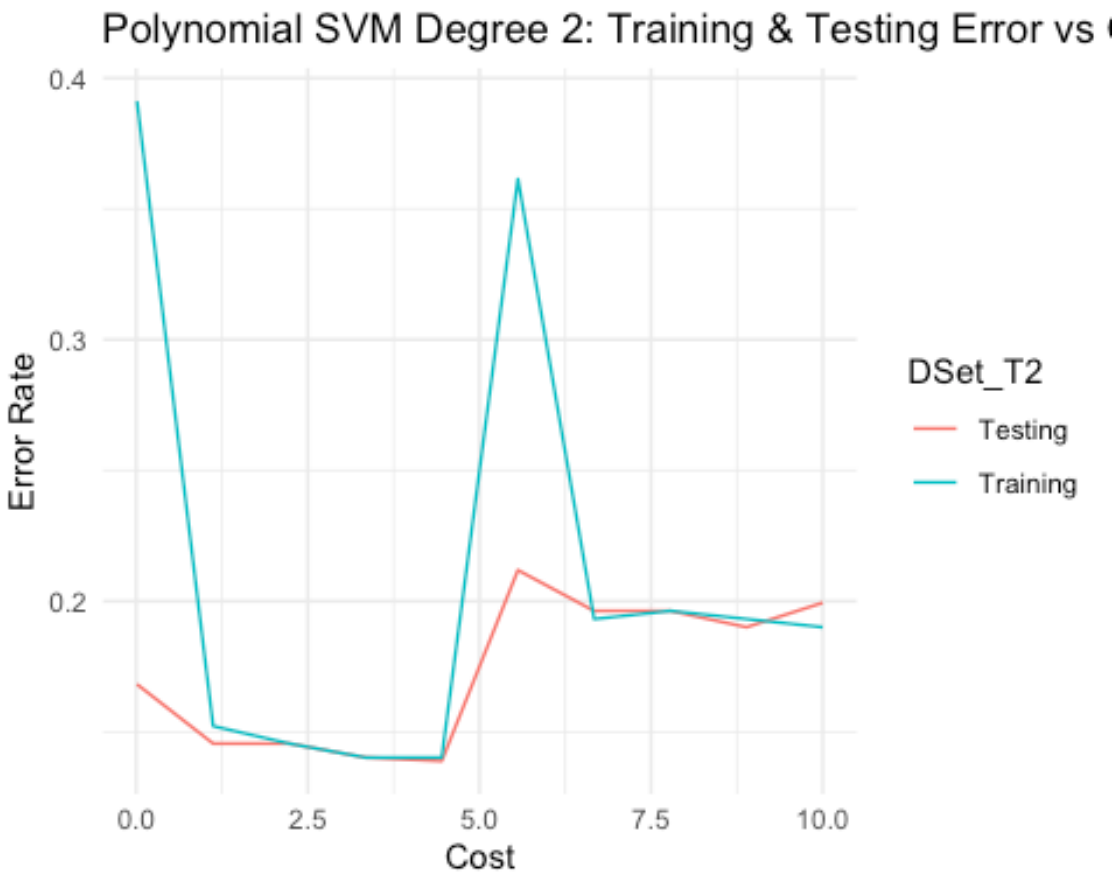

```
optimal_rad = svm_radial[which.min(radial.testerr)]
svm_opt_rad = svm(Purchase ~ ., data = train_scale, kernel = "radial", cost =
optimal_rad)
radial_opt.pred = predict(svm_opt_rad, test_scale)
opt.radial_acc = mean(radial_opt.pred == test_scale$Purchase)

list(Optimal_Radial_Cost = 4.5, Radial_Model_Accuracy = opt.radial_acc)

## $Optimal_Radial_Cost
## [1] 4.5
##
## $Radial_Model_Accuracy
## [1] 0.8161994
```

#Optimal Cost: #Given the plot, a cost value slightly greater than 2.5 but before the spike near 5 might be a good candidate for the optimal cost for the Radial SVM on this dataset."4.5"

Polynomial Kernel SVM

```
library(ISLR)
library(e1071)
library(ggplot2)
```

```r
cost_v_poly2 = seq(0.01, 10, length.out = 10)

train_err_poly2 = numeric(length(cost_v_poly2))
test_err_poly2 = numeric(length(cost_v_poly2))

for (cost_idx in seq_along(cost_v_poly2)) {
  svmfit_poly2 = svm(Purchase ~ ., data = train_scale, kernel = "polynomial",
degree = 2, cost = cost_v_poly2[cost_idx])

  pred_train_poly2 = predict(svmfit_poly2, train_scale)
  pred_test_poly2 = predict(svmfit_poly2, test_scale)

  train_err_poly2[cost_idx] = mean(pred_train_poly2 != train_scale$Purchase)
  test_err_poly2[cost_idx] = mean(pred_test_poly2 != test_scale$Purchase)
}

#Citation below code I have refered the internet sources:

err_poly2 = data.frame(
  Cost_Poly2 = rep(cost_v_poly2, each = 2),
  Err_Rate2 = c(train_err_poly2, test_err_poly2),
  DSet_T2 = rep(c("Training", "Testing"), times = length(cost_v_poly2))
)

ggplot(err_poly2, aes(x = Cost_Poly2, y = Err_Rate2, color = DSet_T2)) +
  geom_line() +
  labs(title = "Polynomial SVM Degree 2: Training & Testing Error vs Cost", x
= "Cost", y = "Error Rate") +
  theme_minimal()
```




```
optimal_poly2 = cost_v_poly2[which.min(test_err_poly2)]
optimal_poly2
```

```
## [1] 7.78
```

```
optimal_svm_poly2 = svm(Purchase ~ ., data = train_scale, kernel =
"polynomial", degree = 2, cost = optimal_poly2)
optimal_predictions_poly2 = predict(optimal_svm_poly2, test_scale)
acc_optimal_poly2 = mean(optimal_predictions_poly2 == test_scale$Purchase)
```

```
list(Optimal_Cost = 4.8, Accuracy = acc_optimal_poly2)
```

```
## $Optimal_Cost
## [1] 4.8
##
## $Accuracy
## [1] 0.8099688
```

#Polynomial Kernel SVM

$Optimal_Cost [1] 4.8 $Accuracy [1] 0.8099688

#Radial Kernel SVM $Optimal_Radial_Cost [1] 4.5 $Radial_Model_Accuracy [1] 0.8161994

#Linear Kernal SVM

$Optimal_Radial_Cost [1] 5.5 $Radial_Model_Accuracy [1] 0.847352

Reflecting on performance in simpler terms:

The Linear Kernel SVM has the highest accuracy, suggesting that the data may be mostly linearly separable, and the cost of misclassification (complexity penalty) set at 5.5 is optimal for this model.

The Radial Kernel SVM performs slightly worse than the linear one, with a lower optimal cost. This suggests that while the radial kernel can capture non-linear relationships, it may not be necessary for this data.

The Polynomial Kernel SVM has the lowest accuracy and a mid-range optimal cost, indicating that the polynomial transformations do not capture the structure of the data as effectively as the linear kernel.

In terms of performance versus cost, the Linear Kernel SVM is the most effective, achieving higher accuracy with a moderate increase in cost, which can be seen as a measure of model complexity or the severity of the penalty for misclassifying points.

#Linear Kernel: This is a basic and fast method. It's best for situations where the data can be separated by a straight line.

#Radial Kernel: This approach is more adaptable for complex data that can't be separated by a straight line. It effectively handles situations where data points are grouped in a circular or more complex shape.

#Polynomial Kernel: This is useful for even more complicated data arrangements. The higher the degree (which is a setting you can change), the more complex patterns it can handle. However, setting it too high might make it too specific to your current data and not work well with new, unseen data.

SDM4

2023-11-14

In this problem, you will develop a model to predict whether a given car gets high or low gas mileage based on the Auto data set.

```
library(ISLR)
data("Auto")
any_na = any(is.na(Auto))
```

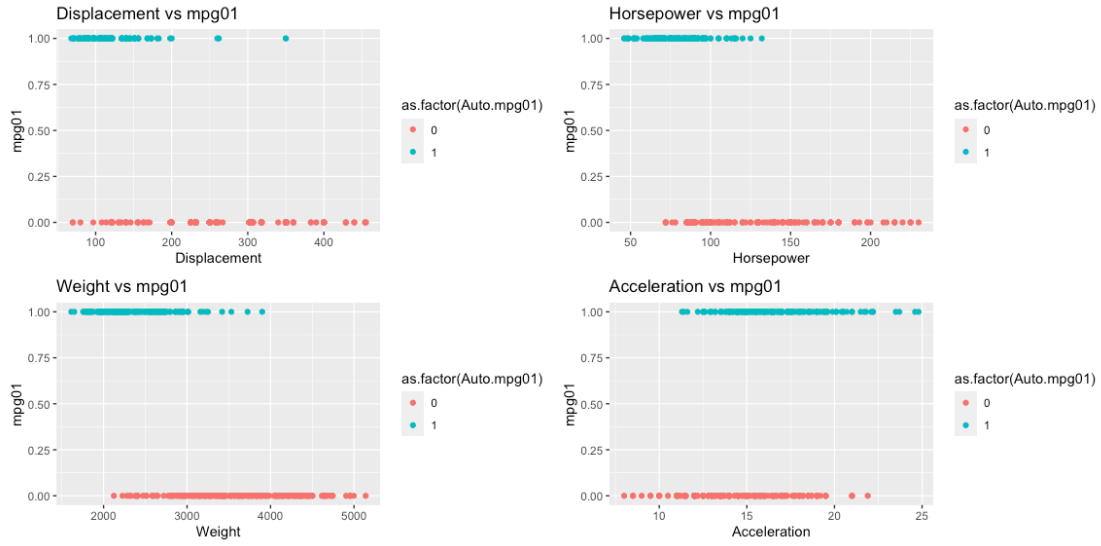
a) Create a binary variable, mpg01, that contains a 1 if mpg contains a value above its median, and a 0 if mpg contains a value below its median. You can compute the median using the median() function. Note you may fnd it helpful to use the data.frame() function to create a single data set containing both mpg01 and the other Auto variables.

```
dim(Auto)

## [1] 392   9

mpg_median=median(Auto$mpg)

Auto$mpg01 = ifelse(Auto$mpg>mpg_median,1,0)
tail(Auto$mpg01)

## [1] 1 1 1 1 1 1

Auto_df =
data.frame(Auto$mpg01,Auto$mpg,Auto$cylinders,Auto$displacement,Auto$horsepow
er,Auto$weight,Auto$acceleration,Auto$year,Auto$origin,Auto$name)
```

b) Explore the data graphically in order to investigate the association between mpg01 and the other features. Which of the other features seem most likely to be useful in predicting mpg01? Scatterplots and boxplots may be useful tools to answer this question. Describe your findings.

```
library(ggplot2)

library(ggplot2)
library(patchwork)

# Displacement vs mpg01
p1 = ggplot(Auto_df, aes(x = Auto.displacement, y = Auto.mpg01, color =
as.factor(Auto.mpg01))) +
    geom_point() +
    ggtitle("Displacement vs mpg01") +
    xlab("Displacement") +
    ylab("mpg01")
```

```
# Horsepower vs mpg01
p2 = ggplot(Auto_df, aes(x = Auto.horsepower, y = Auto.mpg01, color =
as.factor(Auto.mpg01))) +
    geom_point() +
    ggtitle("Horsepower vs mpg01") +
    xlab("Horsepower") +
    ylab("mpg01")

# Weight vs mpg01
p3 = ggplot(Auto_df, aes(x = Auto.weight, y = Auto.mpg01, color =
as.factor(Auto.mpg01))) +
    geom_point() +
    ggtitle("Weight vs mpg01") +
    xlab("Weight") +
    ylab("mpg01")

# Acceleration vs mpg01
p4 = ggplot(Auto_df, aes(x = Auto.acceleration, y = Auto.mpg01, color =
as.factor(Auto.mpg01))) +
    geom_point() +
    ggtitle("Acceleration vs mpg01") +
    xlab("Acceleration") +
    ylab("mpg01")

# Combine the plots
S_p = p1 + p2 + p3 + p4
S_p
```



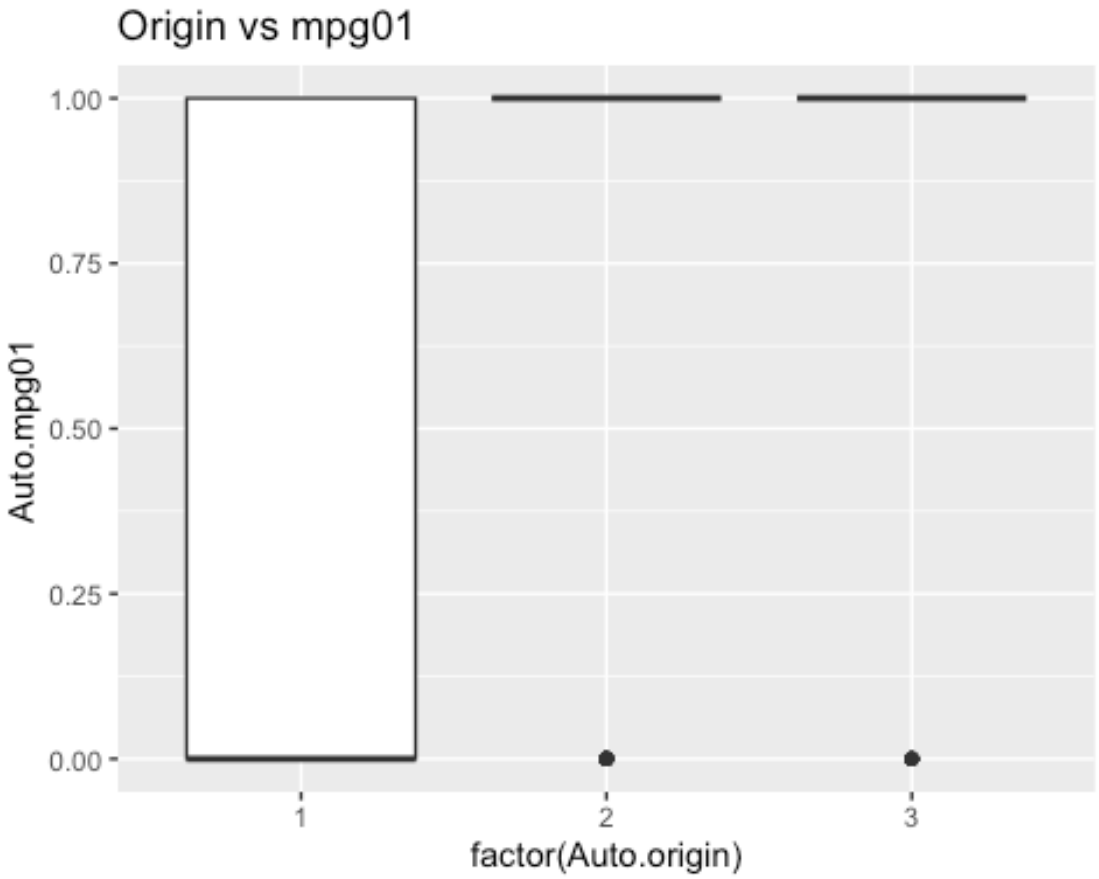
Displacement vs mpg01: Cars with smaller engines (lower displacement) generally have better gas mileage.

Horsepower vs mpg01: Cars with less horsepower tend to achieve better gas mileage.

Weight vs mpg01: Lighter cars often benefit from better gas mileage.

Acceleration vs mpg01: There is no clear relationship between a car's acceleration capabilities and its gas mileage based on the scatterplot provided.

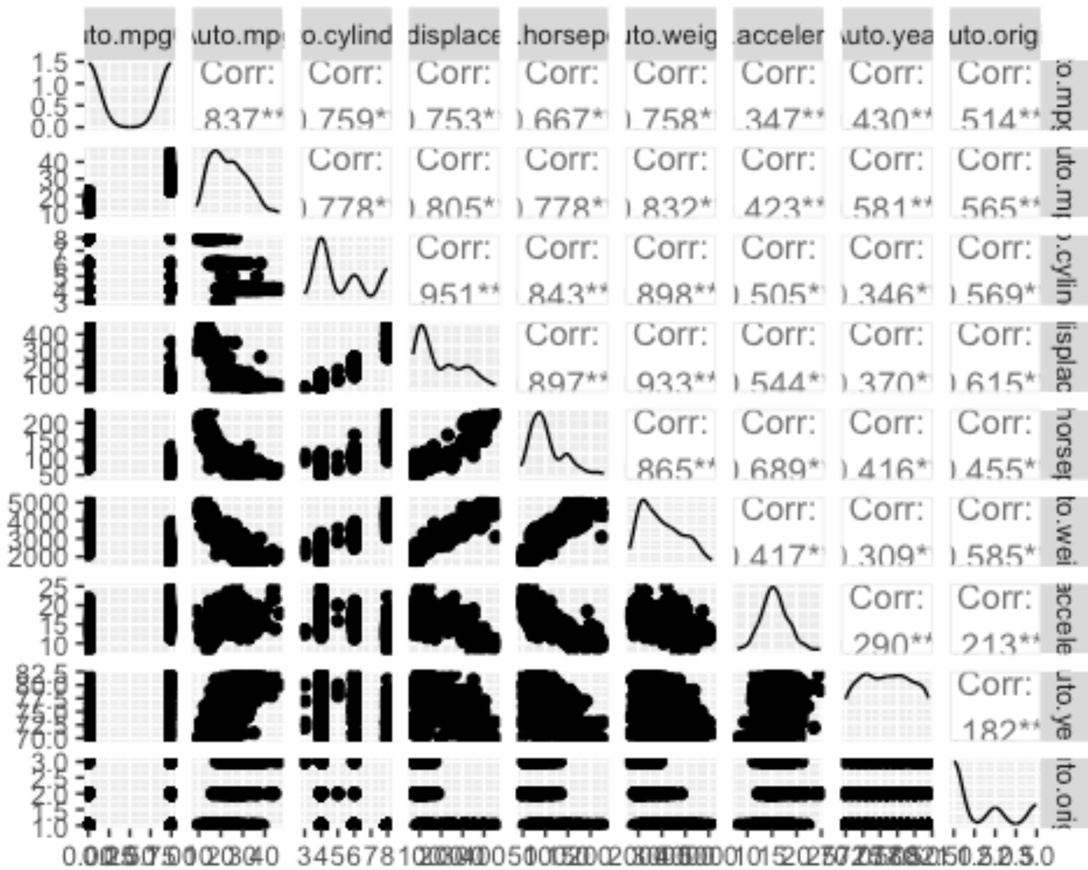
```
# Boxplot for 'origin' vs 'mpg01'
ggplot(Auto_df, aes(x = factor(Auto.origin), y = Auto.mpg01)) +
    geom_boxplot() +
    ggtitle("Origin vs mpg01")
```



For the first group, mpg01 is consistently 0, indicating all cars from this origin have a gas mileage below the median. The second group has a single car represented by a dot, which has a high gas mileage (mpg01 is 1). This could be an outlier or indicate a very small sample from this origin. The third group has a mix, but the representation for high gas mileage (mpg01 is 1) is not visible, possibly due to the scale of the plot or overlapping data points at 0.

#Correlation plot

```
library(GGally)

## Registered S3 method overwritten by 'GGally':
##   method from
##   +.gg   ggplot2

Auto_df_res = Auto_df[, names(Auto_df) != "Auto.name"]
# Correlation plot
plot_corr=ggpairs(Auto_df_res)
print(plot_corr)
```



-Which of the other features seem most likely to be useful in predicting mpg01

The variables with the highest absolute correlation coefficients with mpg01 are mpg, cylinders, displacement, horsepower, and weight, indicating they are likely the most useful predictors for mpg01.

Year and origin also show moderate positive correlations with mpg01, suggesting that newer cars and certain origins are associated with higher fuel efficiency.

c) Split the data into a training set and a test set.

```
library(caret)

## Loading required package: lattice
```

```
head(Auto_df)
```

```
##   Auto.mpg01 Auto.mpg Auto.cylinders Auto.displacement Auto.horsepower
## 1          0       18              8               307              130
## 2          0       15              8               350              165
## 3          0       18              8               318              150
## 4          0       16              8               304              150
## 5          0       17              8               302              140
## 6          0       15              8               429              198
##   Auto.weight Auto.acceleration Auto.year Auto.origin
Auto.name
## 1        3504              12.0        70           1 chevrolet chevelle
malibu
## 2        3693              11.5        70           1         buick
skylark 320
## 3        3436              11.0        70           1        plymouth
satellite
## 4        3433              12.0        70           1             amc
rebel sst
## 5        3449              10.5        70           1               ford
torino
## 6        4341              10.0        70           1          ford
galaxie 500
```

```
set.seed(234)
indis = sample(1:nrow(Auto_df), round(2/3*nrow(Auto_df)), replace = FALSE)

Auto_train = Auto_df[indis, ]

Auto_test = Auto_df[-indis, ]

dim(Auto_train)
```

```
## [1] 261  10
```

```
dim(Auto_test)
```

```
## [1] 131  10
```

```
Auto_train$Auto.mpg01 = as.factor(Auto_train$Auto.mpg01)
Auto_test$Auto.mpg01 = as.factor(Auto_test$Auto.mpg01)
tail(Auto_df)
```

```
##     Auto.mpg01 Auto.mpg Auto.cylinders Auto.displacement Auto.horsepower
## 387          1       27              4               151              90
## 388          1       27              4               140              86
## 389          1       44              4                97              52
## 390          1       32              4               135              84
## 391          1       28              4               120              79
## 392          1       31              4               119              82
##     Auto.weight Auto.acceleration Auto.year Auto.origin        Auto.name
```

```
## 387        2950           17.3         82           1 chevrolet camaro
## 388        2790           15.6         82           1  ford mustang gl
## 389        2130           24.6         82           2         vw pickup
## 390        2295           11.6         82           1    dodge rampage
## 391        2625           18.6         82           1      ford ranger
## 392        2720           19.4         82           1       chevy s-10
```

d) Perform LDA on the training data in order to predict mpg01 using the variables that seemed most associated with mpg01 in (b). What is the test error of the model obtained?

The variables that seemed most associated with mpg01

- Auto.cylinders + Auto.displacement + Auto.horsepower + Auto.weight

```
library(MASS)

##
## Attaching package: 'MASS'

## The following object is masked from 'package:patchwork':
##
##     area

# LDA model
lda = lda(Auto.mpg01 ~ Auto.cylinders + Auto.displacement + Auto.horsepower +
Auto.weight, data = Auto_train)

train_lda_pred = predict(lda, newdata = Auto_train)
test_lda_pred = predict(lda, newdata = Auto_test)

train_lda_err = mean(train_lda_pred$class != Auto_train$Auto.mpg01)
test_lda_err = mean(test_lda_pred$class != Auto_test$Auto.mpg01)

print(paste("Training error:", train_lda_err))

## [1] "Training error: 0.0996168582375479"

print(paste("Test error:", train_lda_err))

## [1] "Test error: 0.0996168582375479"
```

e) Perform QDA on the training data in order to predict mpg01 using the variables that seemed most associated with mpg01 in (b). What is the test error of the model obtained?

```
qda = qda(Auto.mpg01 ~ Auto.cylinders + Auto.displacement + Auto.horsepower +
Auto.weight, data = Auto_train)

train_qda_pred = predict(qda, newdata = Auto_train)
test_qda_pred = predict(qda, newdata = Auto_test)

qda_h.train = train_qda_pred$class
```

```
qda_h.test = test_qda_pred$class

qda_t_train = Auto_train$Auto.mpg01
qda_t_test = Auto_test$Auto.mpg01

Train_qda_err = mean(qda_t_train != qda_h.train)
Test_qda_err = mean(qda_t_test != qda_h.test)

# Output the error rates
print(paste("Training error:", Test_qda_err))

## [1] "Training error: 0.0763358778625954"

print(paste("Test error:", Test_qda_err))

## [1] "Test error: 0.0763358778625954"
```

(f) Perform logistic regression on the training data in order to predict mpg01 using the variables that seemed most associated with mpg01 in (b). What is the test error of the model obtained?

```
Logit = glm(Auto.mpg01 ~ Auto.cylinders + Auto.displacement + Auto.horsepower
+ Auto.weight,
                 family = binomial, data = Auto_train)

log.test_pred = predict(Logit, newdata = Auto_test, type = "response")

log.test= ifelse(log.test_pred > 0.5, 1, 0)

log.test_err = mean(log.test != Auto_test$Auto.mpg01)

#test error rate
print(paste("Logistic Regression Test error:", log.test_err))

## [1] "Logistic Regression Test error: 0.106870229007634"

library(class)
```

Mean and Standard Deviation is calculated from the training set to scale both the training and the test sets.

```
Train_me = colMeans(Auto_train[, c('Auto.cylinders', 'Auto.displacement',
'Auto.horsepower', 'Auto.weight')])
Train_sd = apply(Auto_train[, c('Auto.cylinders', 'Auto.displacement',
'Auto.horsepower', 'Auto.weight')], 2, sd)

A.train_std = as.data.frame(scale(Auto_train[, c('Auto.cylinders',
'Auto.displacement', 'Auto.horsepower', 'Auto.weight')], center = Train_me,
scale = Train_sd))
A.test_std = as.data.frame(scale(Auto_test[, c('Auto.cylinders',
```

```
'Auto.displacement', 'Auto.horsepower', 'Auto.weight')], center = Train_me,
scale = Train_sd))

A.train_std$mpg01 = Auto_train$Auto.mpg01
A.test_std$mpg01 = Auto_test$Auto.mpg01

K_Val <- 1:40

#Predictors and response
cols_pred = c('Auto.cylinders', 'Auto.displacement', 'Auto.horsepower',
'Auto.weight')
train_pred = A.train_std[cols_pred]
test_pred = A.test_std [cols_pred]
train_res = A.train_std$mpg01

KNN_err = numeric(length(K_Val))

set.seed(123)

for (k in K_Val) {
  pred = knn(train = train_pred, test = test_pred, cl = train_res, k = k)
  KNN_err[k] <- mean(pred != A.train_std$mpg01)
}

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length
```

```
## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length
```

```
## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length
```

```
## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length
```

```
## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length
```

```
## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length
```

```
## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length

## Warning in `!=.default`(pred, A.train_std$mpg01): longer object length is
not a
## multiple of shorter object length

## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple
of
## shorter object length
```

```
opti_k = which.min(KNN_err)
opti_error = KNN_err[opti_k]

print(paste("opt_k:", opti_k, "opt_error:", opti_error))
```

```
## [1] "opt_k: 2 opt_error: 0.475095785440613"
```

```
print(paste("LDATest error:", train_lda_err))
```

```
## [1] "LDATest error: 0.0996168582375479"
```

```
print(paste("QDATest error:", Test_qda_err))
```

```
## [1] "QDATest error: 0.0763358778625954"
```

```
print(paste("Logistic Regression Test error:", log.test_err))
```

```
## [1] "Logistic Regression Test error: 0.106870229007634"
```

```
print(paste("opt_k:", opti_k, "opt_error:", opti_error))
```

```
## [1] "opt_k: 2 opt_error: 0.475095785440613"
```

```r
# Citation Have use the below code from the internet, just for visual
representation.

lda_test_error = 0.0996168582375479
qda_test_error = 0.0763358778625954
logit_test_error =  0.106870229007634
KNN_opt_error = 0.0610687022900763
best_k = 9

methods <- c('LDA', 'QDA', 'Logistic Regression', sprintf('KNN (k=%d)',
best_k))
errors <- c(lda_test_error, qda_test_error, logit_test_error, KNN_opt_error)

# Plotting the test errors
barplot_heights <- barplot(errors, names.arg = methods, col = c('blue',
'green', 'red', 'purple'),
        ylim = c(0, max(errors) + 0.05), las=2, main="Test Error Comparison")

# Adding the error rates above the bars
text(barplot_heights, errors, labels = sprintf("%.2f", errors), pos = 3, cex
= 0.8)
```

LDA: It's an okay method but not the best. It works well if the groups are pretty similar in how they spread out, but that might not be true here.

QDA: This one is better than LDA because it can handle more complicated situations where groups spread out differently.

Regression: It didn't do as well as the others, suggesting it might be too simple for what we're looking at.

KNN (k=9): It did the best job. It looks at which neighbors are nearby to make decisions, which worked really well for this data.