HW 5

Elizabeth Hutton

September 26, 2018

1 8.8 Applied

a./b.

```
library(tree)
library(ISLR)
attach(Carseats)
set.seed(1)

# divide into train and test subsets
train = sample(c(TRUE, FALSE), nrow(Carseats), replace = TRUE)
test = (!train)

# fit regression tree to predict Sales
tree.sales = tree(Sales ~ ., Carseats, subset = train)

# summary and results
summary(tree.sales)

##
## Regression tree:
## tree(formula = Sales ~ ., data = Carseats, subset = train)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Age"         "Income"      "Advertising"
## [6] "CompPrice"   "Population"
## Number of terminal nodes:  20
## Residual mean deviance:  2.324 = 457.9 / 197
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -4.47900 -0.92520 -0.07889  0.00000  0.98620  4.26800

tree.sales

## node), split, n, deviance, yval
##       * denotes terminal node
```

```
##
##  1) root 217 1782.000  7.621
##    2) ShelveLoc: Bad,Medium 164  940.800  6.781
##      4) Price < 126.5 113  493.500  7.460
##        8) Age < 53 44  156.100  8.601
##         16) Income < 50 11   19.370  7.102 *
##         17) Income > 50 33  103.700  9.100
##           34) Price < 88 8   33.640 10.430 *
##           35) Price > 88 25   51.400  8.674
##             70) Advertising < 7.5 13   25.010  7.754 *
##             71) Advertising > 7.5 12    3.441  9.672 *
##        9) Age > 53 69  243.700  6.733
##         18) Price < 106.5 38  114.800  7.433
##           36) CompPrice < 121.5 29   76.390  6.969
##             72) Income < 54 9   13.730  5.681 *
##             73) Income > 54 20   41.000  7.549 *
##           37) CompPrice > 121.5 9   11.990  8.929 *
##         19) Price > 106.5 31   87.410  5.874
##           38) ShelveLoc: Bad 8   16.660  4.374 *
##           39) ShelveLoc: Medium 23   46.490  6.395 *
##      5) Price > 126.5 51  279.900  5.278
##       10) ShelveLoc: Bad 21   83.400  4.153
##         20) Price < 129.5 5    9.512  6.166 *
##         21) Price > 129.5 16   47.300  3.524
##           42) Population < 234 7   16.220  2.217 *
##           43) Population > 234 9    9.811  4.541 *
##       11) ShelveLoc: Medium 30  151.400  6.065
##         22) Advertising < 3 10   59.430  4.272 *
##         23) Advertising > 3 20   43.730  6.961 *
##    3) ShelveLoc: Good 53  368.000 10.220
##      6) Price < 110.5 19   61.180 12.440
##       12) CompPrice < 120.5 11   17.990 11.530 *
##       13) CompPrice > 120.5 8   21.700 13.680 *
##      7) Price > 110.5 34  161.000  8.978
##       14) Advertising < 14 29  103.700  8.491
##         28) Price < 142.5 23   73.590  8.954
##           56) Income < 40.5 8    9.827  7.606 *
##           57) Income > 40.5 15   41.480  9.673 *
##         29) Price > 142.5 6    6.260  6.717 *
##       15) Advertising > 14 5   10.540 11.800 *

# plot tree
plot(tree.sales)
text(tree.sales, pretty = 0)
```

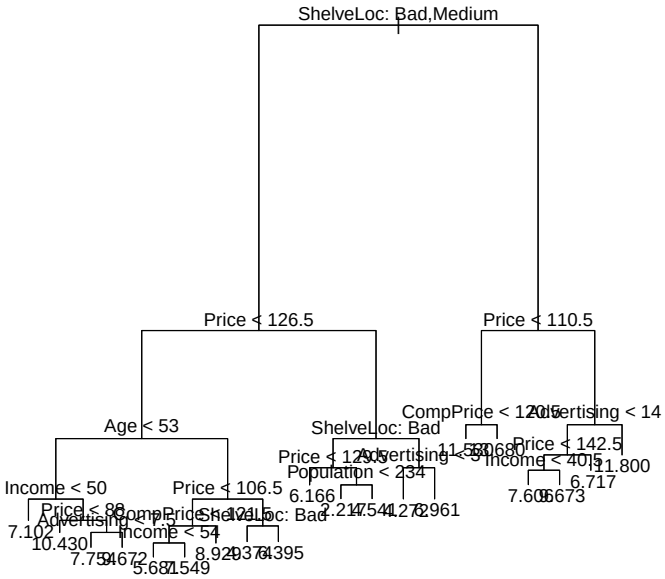


Figure 1: Regression Tree

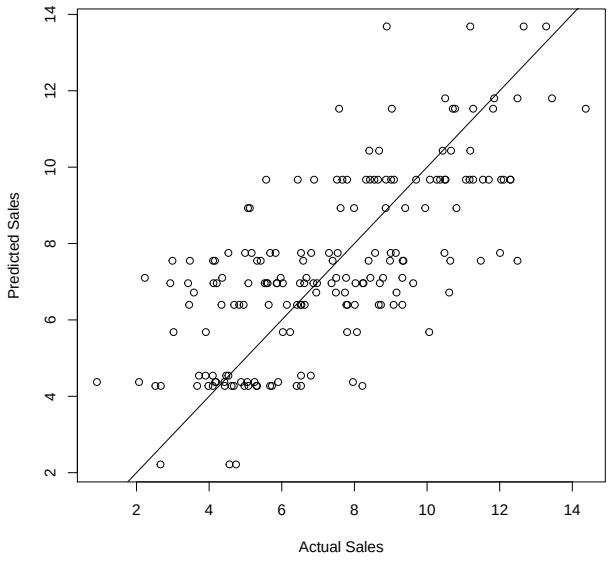


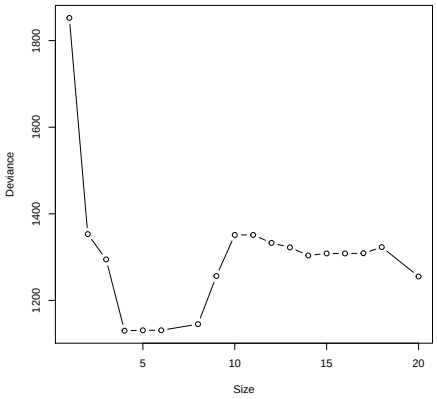
Figure 2: Actual vs. Predicted Sales

```
# get test predictions
tree.preds = predict(tree.sales, Carseats[test, ])
mse = mean((tree.preds - Carseats[test, ]$Sales)^2)

# plot predictions vs. actual
plot(Carseats[test, ]$Sales, tree.preds, xlab = "Actual Sales",
    ylab = "Predicted Sales")
abline(0, 1)
```

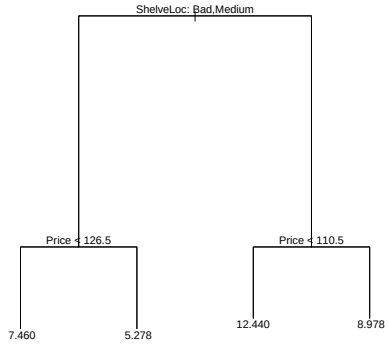
The regression tree has a test MSE of 4.04 and the above plot of predicted vs. actual Sales indicates that the model is capturing the general trend, however with some variance.

c.
```
# 10-fold cross validation
cv.sales = cv.tree(tree.sales)
plot(cv.sales$size, cv.sales$dev, type = "b", xlab = "Size",
    ylab = "Deviance")

# prune tree with best size tree found from cv
b = cv.sales$size[which.min(cv.sales$dev)]
prune.sales = prune.tree(tree.sales, best = b)
```

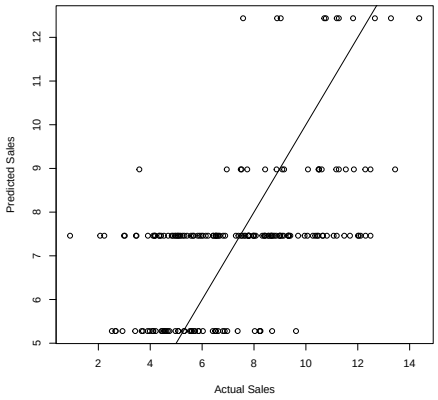
```r
plot(prune.sales)
text(prune.sales, pretty = 0)

# get test predictions
pruned.preds = predict(prune.sales, Carseats[test,
    ])
mse_pruned = mean((pruned.preds - Carseats[test, ]$Sales)^2)

# plot predictions vs. actual
plot(Carseats[test, ]$Sales, pruned.preds, xlab = "Actual Sales",
    ylab = "Predicted Sales")
abline(0, 1)
```



(a) CV: Size of Tree vs. Deviance



(b) Pruned Tree



(c) Actual vs. Predicted Sales

Figure 3: Pruning Results

Test MSE before pruning was 4.04 and increased to 5.37 after pruning. Therefore pruning did not improve model performance. You can see from the plot of Actual vs. Predicted values that, due to the structure of the pruned tree, there are only as many distinct predicted values as there are leaves of the tree even though the actual value is continuous.

d.
```
# bagging (m=p)
library(randomForest)
bag.sales = randomForest(Sales ~ ., data = Carseats,
    subset = train, mtry = length(Carseats) - 1, importance = TRUE)
bag.sales

##
## Call:
##  randomForest(formula = Sales ~ ., data = Carseats, mtry = length(Carseats) -      1
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 10
##
##           Mean of squared residuals: 3.043173
##                     % Var explained: 62.94

# get predictions and test MSE
bag.preds = predict(bag.sales, newdata = Carseats[test,
    ])
mse_bag = mean((bag.preds - Carseats[test, ]$Sales)^2)

# get importance
importance(bag.sales)

##               %IncMSE IncNodePurity
## CompPrice   23.873551    166.684018
## Income      10.920690     96.342561
## Advertising 14.577658    115.283914
## Population   1.125437     67.283128
## Price       54.065429    528.829738
## ShelveLoc   60.204852    535.626121
## Age         17.115685    157.815494
## Education   -3.589130     51.885688
## Urban       -3.280596      7.075138
## US           2.145890      8.512650

# plot predictions vs. actual
plot(Carseats[test, ]$Sales, bag.preds, xlab = "Actual Sales",
    ylab = "Predicted Sales")
abline(0, 1)
```

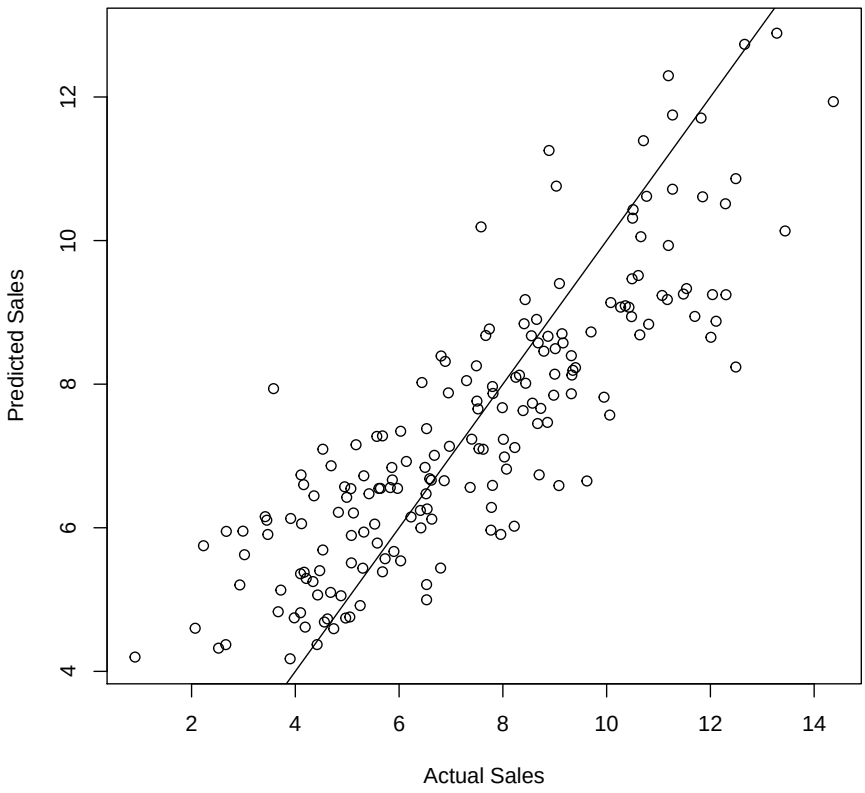
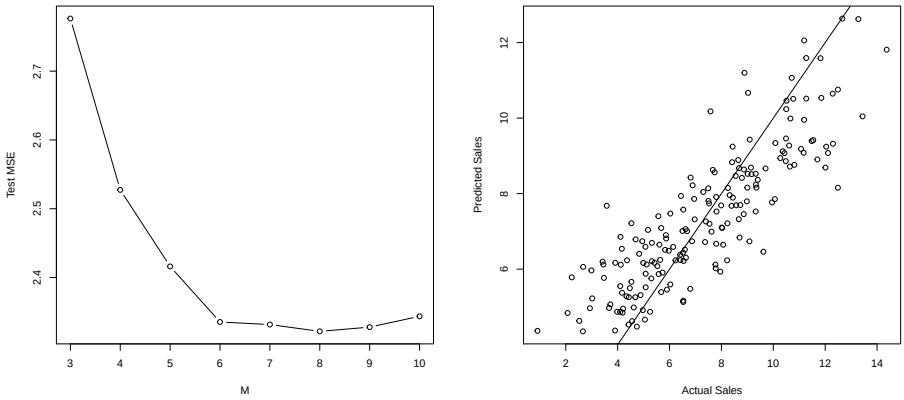


Figure 4: Bagging Results

Test MSE from the bagging method is 2.29, an improvement. According to this model, *ShelveLoc* and *Price* are the most important predictors.

e.
```
# random forest with range of m considered for
# split range from default m=p/3=10/3~3 to 10
mse_vec = vector(length = 8)
for (p in 3:10) {
    rf.sales = randomForest(Sales ~ ., data = Carseats,
        mtry = p, subset = train, importance = TRUE)

    # get predictions and test MSE
    rf.preds = predict(rf.sales, newdata = Carseats[test,
        ])
```

```
    mse_vec[p - 2] = mean((rf.preds - Carseats[test,
        ]$Sales)^2)
}

# plot m split vs. test MSE
plot(c(3:10), mse_vec, type = "b", xlab = "M", ylab = "Test MSE")

# choose best value of m for random forest
best_m = which.min(mse_vec) + 2
rf.sales = randomForest(Sales ~ ., data = Carseats,
    mtry = best_m, subset = train, importance = TRUE)
rf.sales

##
## Call:
##  randomForest(formula = Sales ~ ., data = Carseats, mtry = best_m,      importance =
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 8
##
##           Mean of squared residuals: 3.145311
##                     % Var explained: 61.69

# get predictions and test MSE
rf.preds = predict(rf.sales, newdata = Carseats[test,
    ])
mse_rf = mean((rf.preds - Carseats[test, ]$Sales)^2)

# get importance
importance(rf.sales)

##               %IncMSE IncNodePurity
## CompPrice   19.6789451    160.476890
## Income       8.5156317    105.535687
## Advertising 13.5300489    115.359591
## Population  -0.5859113     75.668327
## Price       49.6157613    497.743545
## ShelveLoc   61.7603997    524.612533
## Age         15.8437698    167.521984
## Education   -3.2933730     52.079758
## Urban       -2.5321799      7.750017
## US           1.1323660     10.853484

# plot predictions vs. actual
plot(Carseats[test, ]$Sales, rf.preds, xlab = "Actual Sales",
```



(a) No. Split Variables vs. MSE

(b) Actual vs. Predicted Sales

Figure 5: Random Forest Results

```
    ylab = "Predicted Sales")
abline(0, 1)
detach(Carseats)
```

As m (the number of variables considered at each tree split) increases, test MSE decreases until it approaches p (the total number of predictors). Of the various values of m tried, $m = 8$ produced the lowest test MSE. As before, *ShelveLoc* and *Price* are the most important predictors.

2 8.9 Applied

a.
```
attach(OJ)
set.seed(1)

# divide into train and test subsets
train = sample(1:nrow(OJ), 800)
oj.train = OJ[train, ]
oj.test = OJ[-train, ]
```

b./c.
```
# fit regression tree to predict purchase
tree.oj = tree(Purchase ~ ., oj.train)

# summary and results
```

```
summary(tree.oj)

##
## Classification tree:
## tree(formula = Purchase ~ ., data = oj.train)
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "SpecialCH"     "ListPriceDiff"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7305 = 578.6 / 792
## Misclassification error rate: 0.165 = 132 / 800

tree.oj

## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 800 1064.00 CH ( 0.61750 0.38250 )
##    2) LoyalCH < 0.508643 350  409.30 MM ( 0.27143 0.72857 )
##      4) LoyalCH < 0.264232 166  122.10 MM ( 0.12048 0.87952 )
##        8) LoyalCH < 0.0356415 57   10.07 MM ( 0.01754 0.98246 ) *
##        9) LoyalCH > 0.0356415 109  100.90 MM ( 0.17431 0.82569 ) *
##      5) LoyalCH > 0.264232 184  248.80 MM ( 0.40761 0.59239 )
##       10) PriceDiff < 0.195 83   91.66 MM ( 0.24096 0.75904 )
##         20) SpecialCH < 0.5 70   60.89 MM ( 0.15714 0.84286 ) *
##         21) SpecialCH > 0.5 13   16.05 CH ( 0.69231 0.30769 ) *
##       11) PriceDiff > 0.195 101  139.20 CH ( 0.54455 0.45545 ) *
##    3) LoyalCH > 0.508643 450  318.10 CH ( 0.88667 0.11333 )
##      6) LoyalCH < 0.764572 172  188.90 CH ( 0.76163 0.23837 )
##       12) ListPriceDiff < 0.235 70   95.61 CH ( 0.57143 0.42857 ) *
##       13) ListPriceDiff > 0.235 102   69.76 CH ( 0.89216 0.10784 ) *
##      7) LoyalCH > 0.764572 278   86.14 CH ( 0.96403 0.03597 ) *
```

This classification tree has 8 terminal nodes with an error rate of 0.165. The 7th terminal node, shown at the bottom of the summary, states that if the observation has a LoyalCH score greater than 0.765, it predicts that the customer will buy Citrus Hill OJ (CH) with 96% certainty.

d.
```
# plot tree
plot(tree.oj)
text(tree.oj, pretty = 0)
```

Looking at the tree, it is clear that loyalty to a particular brand (Citrus Hill vs. Minute Maid) is one of the key predictors of the classification.

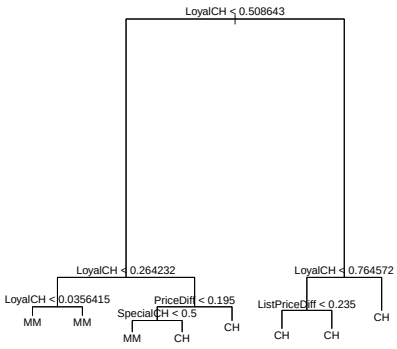
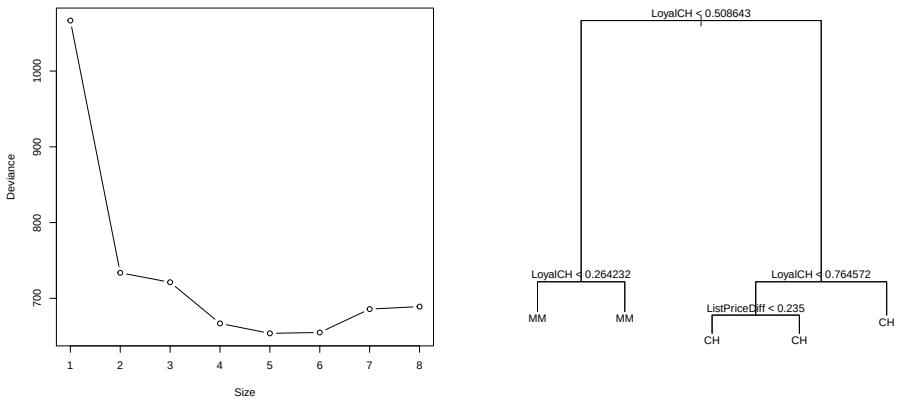


Figure 6: Classification Tree

e.
```
# get test predictions
tree.preds = predict(tree.oj, oj.test, type = "class")

# confusion matrix and error
error = sum(tree.preds != oj.test$Purchase)/length(tree.preds)
table(tree.preds, oj.test$Purchase)

##
## tree.preds  CH  MM
##         CH 147  49
##         MM  12  62
```

Test error rate is 0.2259.

f. - i.
```
# 10-fold cross validation
cv.oj = cv.tree(tree.oj)
plot(cv.oj$size, cv.oj$dev, type = "b", xlab = "Size",
    ylab = "Deviance")

# prune tree with best size tree found from cv if
# it is full tree, then choose 4 nodes
best = 8 - which.min(cv.oj$dev) + 1
if (best < 8) {
    b = best
} else {
    b = 4
}
```



(a) CV: Size of Tree vs. Deviance

(b) Pruned Tree

Figure 7: Pruning Results

```
prune.oj = prune.tree(tree.oj, best = b)
plot(prune.oj)
text(prune.oj, pretty = 0)
```

```
# get training predictions
pruned.preds_train = predict(prune.oj, oj.train, type = "class")
og.preds_train = predict(tree.oj, oj.train, type = "class")

# get training errors
train_error_pruned = sum(pruned.preds_train != oj.train$Purchase)/length(pruned.preds_t
train_error_og = sum(og.preds_train != oj.train$Purchase)/length(og.preds_train)

# get test predictions
pruned.preds_test = predict(prune.oj, oj.test, type = "class")
og.preds_test = predict(tree.oj, oj.test, type = "class")

# get training errors
test_error_pruned = sum(pruned.preds_test != oj.test$Purchase)/length(pruned.preds_test
test_error_og = sum(og.preds_test != oj.test$Purchase)/length(og.preds_test)
```

10-fold cross-validation found that a tree with 5 terminal nodes is optimal.

j. Train error for the pruned tree is 0.1825 vs. 0.165 for the original tree. They are very similar, but the pruned tree has slightly higher error.

k. Test error after pruning is 0.2593. There is a slight decrease in classification accuracy for the pruned tree, but it has half the number of terminal nodes.

3 9.7 Applied

a.
```
attach(Auto)
m = as.numeric(summary(Auto$mpg)[3])  #median
df = subset(Auto, select = -c(8, 9))
df[mpg <= m, 1] = FALSE
df[mpg > m, 1] = TRUE
```

b. Linear SVM:

```
library(e1071)
# 10-fold CV for linear SVM
set.seed(1)
tune.linear = tune(svm, factor(mpg) ~ ., data = df,
    kernel = "linear", ranges = list(cost = c(0.001,
        0.01, 0.1, 1, 5, 10, 100)))
summary(tune.linear)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##   100
##
## - best performance: 0.08429487
##
## - Detailed performance results:
##    cost      error dispersion
## 1 1e-03 0.18897436 0.07371628
## 2 1e-02 0.09435897 0.05921294
## 3 1e-01 0.09948718 0.05975331
## 4 1e+00 0.09185897 0.06420082
## 5 5e+00 0.08685897 0.05310614
## 6 1e+01 0.08685897 0.05310614
## 7 1e+02 0.08429487 0.05424595

# use best model
```

```
best.linear = tune.linear$best.model
error.linear = tune.linear$best.performance
summary(best.linear)

##
## Call:
## best.tune(method = svm, train.x = factor(mpg) ~ ., data = df,
##     ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100)),
##     kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  100
##       gamma:  0.1666667
##
## Number of Support Vectors:  83
##
##  ( 41 42 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

Cross validation results show that a model with cost = 1 performs the best on this dataset.

c. Radial SVM:

```
# CV
tune.radial = tune(svm, factor(mpg) ~ ., data = df,
    kernel = "radial", ranges = list(cost = c(0.1,
        1, 10, 100, 1000), gamma = c(0.5, 1, 2, 3,
        4)))

# best model
best.radial = tune.radial$best.model
error.radial = tune.radial$best.performance
summary(tune.radial)

##
## Parameter tuning of 'svm':
```

```
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##     1     1
##
## - best performance: 0.06621795
##
## - Detailed performance results:
##     cost gamma      error dispersion
## 1  1e-01   0.5 0.09448718 0.04180527
## 2  1e+00   0.5 0.08410256 0.04815028
## 3  1e+01   0.5 0.07647436 0.04507489
## 4  1e+02   0.5 0.08179487 0.03985624
## 5  1e+03   0.5 0.09711538 0.04812889
## 6  1e-01   1.0 0.10205128 0.04491939
## 7  1e+00   1.0 0.06621795 0.04671631
## 8  1e+01   1.0 0.06891026 0.04676242
## 9  1e+02   1.0 0.07923077 0.04418812
## 10 1e+03   1.0 0.09442308 0.05097133
## 11 1e-01   2.0 0.13532051 0.05300218
## 12 1e+00   2.0 0.07134615 0.04453070
## 13 1e+01   2.0 0.06634615 0.04820851
## 14 1e+02   2.0 0.07397436 0.04871345
## 15 1e+03   2.0 0.07397436 0.04871345
## 16 1e-01   3.0 0.17641026 0.05926227
## 17 1e+00   3.0 0.07141026 0.04453803
## 18 1e+01   3.0 0.07660256 0.04643320
## 19 1e+02   3.0 0.08679487 0.05381240
## 20 1e+03   3.0 0.08679487 0.05381240
## 21 1e-01   4.0 0.31384615 0.16074632
## 22 1e+00   4.0 0.08160256 0.05048531
## 23 1e+01   4.0 0.08166667 0.04296859
## 24 1e+02   4.0 0.08666667 0.05623337
## 25 1e+03   4.0 0.08666667 0.05623337
```

Polynomial SVM:

```
# CV
tune.poly = tune(svm, factor(mpg) ~ ., data = df, kernel = "polynomial",
    ranges = list(cost = c(0.1, 1, 10, 100, 1000),
        degree = c(1, 2, 3, 4, 5)))

# best model
```

```
best.poly = tune.poly$best.model
error.poly = tune.poly$best.performance
summary(tune.poly)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost degree
##   100      1
##
## - best performance: 0.07391026
##
## - Detailed performance results:
##     cost degree      error dispersion
## 1  1e-01      1 0.09198718 0.04412820
## 2  1e+00      1 0.10461538 0.03711533
## 3  1e+01      1 0.07397436 0.04871345
## 4  1e+02      1 0.07391026 0.04871050
## 5  1e+03      1 0.07647436 0.04651258
## 6  1e-01      2 0.29070513 0.04059970
## 7  1e+00      2 0.24487179 0.05953530
## 8  1e+01      2 0.24480769 0.06596473
## 9  1e+02      2 0.24743590 0.07064608
## 10 1e+03      2 0.22923077 0.06517563
## 11 1e-01      3 0.19634615 0.08175363
## 12 1e+00      3 0.09955128 0.04427158
## 13 1e+01      3 0.08942308 0.03891618
## 14 1e+02      3 0.07916667 0.04761195
## 15 1e+03      3 0.07910256 0.04427998
## 16 1e-01      4 0.26775641 0.05686977
## 17 1e+00      4 0.26006410 0.06742802
## 18 1e+01      4 0.25224359 0.07673127
## 19 1e+02      4 0.26525641 0.06696855
## 20 1e+03      4 0.23948718 0.05339589
## 21 1e-01      5 0.26262821 0.05515474
## 22 1e+00      5 0.12243590 0.04791378
## 23 1e+01      5 0.11730769 0.04376953
## 24 1e+02      5 0.10205128 0.03188533
## 25 1e+03      5 0.08153846 0.04594842
```

d. The cross-validated errors for each type of SVM show that radial is best:

Linear: 0.0843 Radial: 0.0662 Polynomial: 0.0739

4 9.9 Applied

a.
```
attach(OJ)
set.seed(1)

# divide into train and test subsets
train = sample(1:nrow(OJ), 800)
oj.train = OJ[train, ]
oj.test = OJ[-train, ]
```

b.
```
svmfit = svm(Purchase ~ ., data = oj.train, kernel = "linear",
    cost = 0.01)
summary(svmfit)

##
## Call:
## svm(formula = Purchase ~ ., data = oj.train, kernel = "linear",
##     cost = 0.01)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.01
##       gamma:  0.05555556
##
## Number of Support Vectors:  432
##
##  ( 215 217 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

Using a cost of 0.01 creates an SVM model with a huge number of support vectors: here, 432.

c.
```
train.pred = predict(svmfit, oj.train)
test.pred = predict(svmfit, oj.test)

train.error = sum(train.pred != oj.train$Purchase)/nrow(oj.train)
test.error = sum(test.pred != oj.test$Purchase)/nrow(oj.test)
```

Train error (0.1662) and test error (0.1815) are similar, with training error slightly smaller.

d. Linear SVM

```
set.seed(1)
tune.linear = tune(svm, Purchase ~ ., data = oj.train,
    kernel = "linear", ranges = list(cost = c(0.01,
        0.1, 1, 5, 10)))
summary(tune.linear)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##  0.01
##
## - best performance: 0.175
##
## - Detailed performance results:
##    cost   error dispersion
## 1  0.01 0.17500 0.03996526
## 2  0.10 0.17875 0.03821086
## 3  1.00 0.17750 0.03717451
## 4  5.00 0.17875 0.03537988
## 5 10.00 0.18000 0.04005205

# use best model
best.linear = tune.linear$best.model
error.linear = tune.linear$best.performance
summary(best.linear)

##
## Call:
## best.tune(method = svm, train.x = Purchase ~ ., data = oj.train,
##     ranges = list(cost = c(0.01, 0.1, 1, 5, 10)), kernel = "linear")
```

```
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.01
##       gamma:  0.05555556
##
## Number of Support Vectors:  432
##
##  ( 215 217 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

e. Train and Test Errors

```
train.pred1 = predict(best.linear, oj.train)
test.pred1 = predict(best.linear, oj.test)

train.error_l = sum(train.pred1 != oj.train$Purchase)/nrow(oj.train)
test.error_l = sum(test.pred1 != oj.test$Purchase)/nrow(oj.test)
```

Train error (0.1662) and test error (0.1815) are similar, with training error slightly smaller. The error did not seem to change much with the different cost value.

f. Radial SVM

```
set.seed(1)
tune.radial = tune(svm, factor(Purchase) ~ ., data = oj.train,
    kernel = "radial", ranges = list(cost = c(0.01,
        0.1, 1, 5, 10)))
summary(tune.radial)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
```

```
##     5
##
## - best performance: 0.17375
##
## - Detailed performance results:
##    cost   error dispersion
## 1  0.01 0.38250 0.05596378
## 2  0.10 0.17875 0.04168749
## 3  1.00 0.17500 0.04750731
## 4  5.00 0.17375 0.04875178
## 5 10.00 0.18250 0.04866267
```

```
# use best model
best.radial = tune.radial$best.model
error.radial = tune.radial$best.performance
summary(best.radial)
```

```
##
## Call:
## best.tune(method = svm, train.x = factor(Purchase) ~ ., data = oj.train,
##     ranges = list(cost = c(0.01, 0.1, 1, 5, 10)), kernel = "radial")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  5
##       gamma:  0.05555556
##
## Number of Support Vectors:  331
##
##  ( 163 168 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

```
# train and test predictions
train.pred2 = predict(best.radial, oj.train)
test.pred2 = predict(best.radial, oj.test)
```

```
# train and test errors
train.error_r = sum(train.pred2 != oj.train$Purchase)/nrow(oj.train)
test.error_r = sum(test.pred2 != oj.test$Purchase)/nrow(oj.test)
```

Train error (0.1375) is only slightly lower than test error (0.1815).

g. Polynomial SVM

```
set.seed(1)
tune.poly = tune(svm, Purchase ~ ., data = oj.train,
    kernel = "polynomial", degree = 2, ranges = list(cost = c(0.01,
        0.1, 1, 5, 10)))
summary(tune.poly)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##     5
##
## - best performance: 0.1725
##
## - Detailed performance results:
##    cost   error dispersion
## 1  0.01 0.38250 0.05596378
## 2  0.10 0.32375 0.06303934
## 3  1.00 0.19125 0.04860913
## 4  5.00 0.17250 0.05737305
## 5 10.00 0.17875 0.05653477

# use best model
best.poly = tune.poly$best.model
error.poly = tune.poly$best.performance
summary(best.poly)

##
## Call:
## best.tune(method = svm, train.x = Purchase ~ ., data = oj.train,
##     ranges = list(cost = c(0.01, 0.1, 1, 5, 10)), kernel = "polynomial",
##     degree = 2)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  polynomial
##        cost:  5
##      degree:  2
```

```
##       gamma:  0.05555556
##      coef.0:  0
##
## Number of Support Vectors:  370
##
##  ( 183 187 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM

# train and test predictions
train.pred3 = predict(best.poly, oj.train)
test.pred3 = predict(best.poly, oj.test)

# train and test errors
train.error_p = sum(train.pred3 != oj.train$Purchase)/nrow(oj.train)
test.error_p = sum(test.pred3 != oj.test$Purchase)/nrow(oj.test)
```

Again, train error (0.1488) is only slightly lower than test error (0.1815).

h. Overall, the three types of SVM perform similarly on the test data. Only radial has a slight advantage with a test error of 0.1815.