9.7 Exercies

- Exercise 4
- Exercise 5
- Exercise 6
- Exercise 7
- Exercise 8

```
# Load all the libraries used in these exercises
library(e1071)
```

```
## Warning: package 'e1071' was built under R version 3.4.4
```

```
library(ISLR)
```

```
## Warning: package 'ISLR' was built under R version 3.4.4
```

Exercise 4

Generate a simulated two-class data set with 100 observations and two features in which there is a visible but non-linear separation between the two classes. Show that in this setting, a support vector machine with a polynomial kernel (with degree greater than 1) or a radial kernel will outperform a support vector classifier on the training data. Which technique performs best on the test data? Make plots and report training and test error rates in order to back up your assertions.

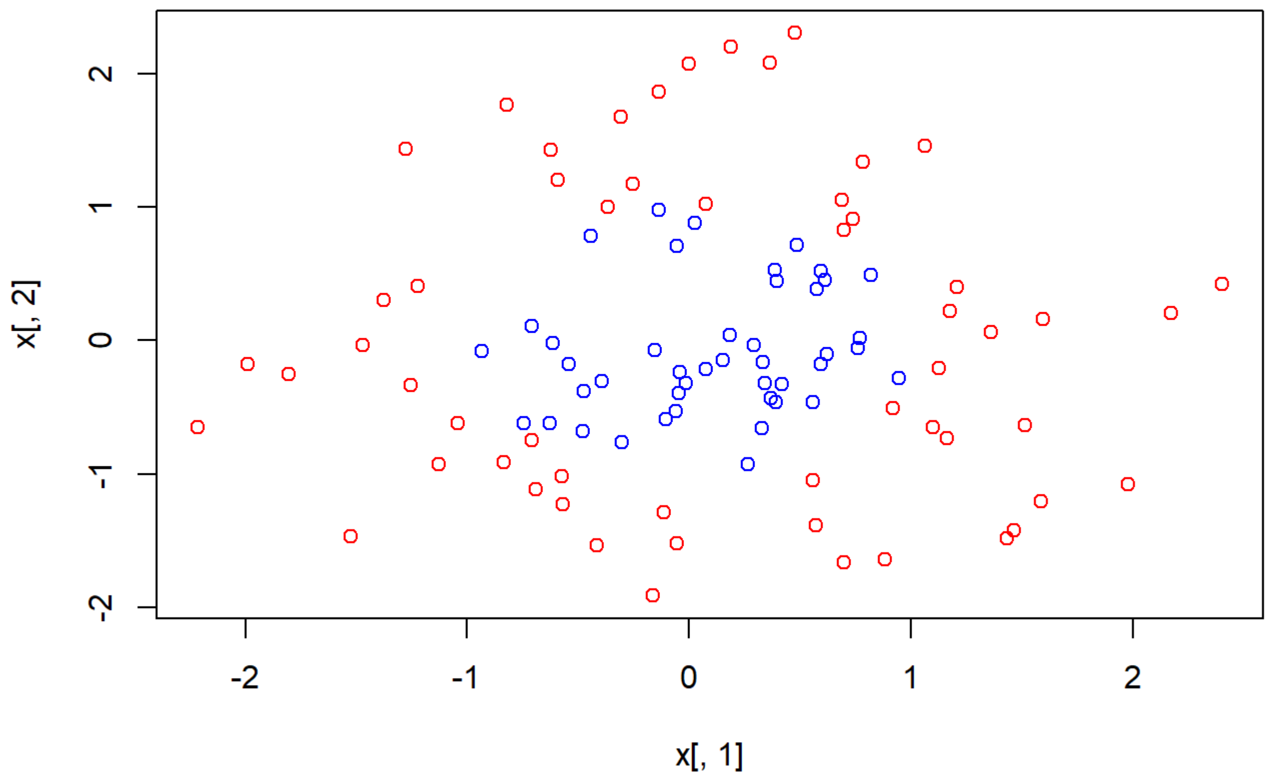
```
set.seed(1)

# create predictor matrix
p = 2
n = 100
x = matrix(data = rnorm(n * p, mean = 0, sd = 1), nrow = n, ncol = p)

# create two classes divided by the circle with origin (0,0) and radius 1
y = rep(-1, n)
y[x[,1]*x[,1] + x[,2]*x[,2] > 1] = 1

# color the classes
color = rep('blue', n)
color[y == 1] = 'red'

# plot data and classes
plot(x[,1], x[,2], col = color)
```



x[, 1]
x[, 2]
-2
-1
0
1
2

```
set.seed(1)

# Divide data into training and test
dat = data.frame(x = x, y = as.factor(y))
train = sample.int(n, size = n * .8, replace = FALSE)

# Fit an SVM with a linear boundary
linear.fit = svm(y ~ ., data = dat[train,], kernel = "linear", cost = 10, scale = FALSE)
linear.error.test  = 1 - mean(predict(linear.fit, dat[-train,]) == dat[-train, 'y'])
linear.error.train = 1 - mean(predict(linear.fit, dat[train,]) == dat[train, 'y'])

# Fit an SVM with a radial boundary
radial.fit = svm(y ~ ., data = dat[train,], kernel = "radial", gamma = 0.1, cost = 10, s
cale = FALSE)
radial.error.test  = 1 - mean(predict(radial.fit, dat[-train,]) == dat[-train, 'y'])
radial.error.train = 1 - mean(predict(radial.fit, dat[train,]) == dat[train, 'y'])

# Fit an SVM with a quadratic boundary
quad.fit = svm(y ~ ., data = dat[train,], kernel = "polynomial", degree = 2, cost = 10,
 scale = FALSE)
quad.error.test  = 1 - mean(predict(quad.fit, dat[-train,]) == dat[-train, 'y'])
quad.error.train = 1 - mean(predict(quad.fit, dat[train,]) == dat[train, 'y'])

# Compare error rates
print(data.frame(
  type = c("linear", "radial", "quadratic"),
  train.error = c(linear.error.train, radial.error.train, quad.error.train),
  test.error = c(linear.error.test, radial.error.test, quad.error.test)
))
```

```
##        type train.error test.error
## 1    linear       0.450        0.4
## 2    radial       0.050        0.2
## 3 quadratic       0.025        0.0
```

Unsuprisingly, the linear boundary is outclassed here. I am suprised that the quadratic decision boundary outperformed the radial boundary. Perhaps if we used cross-validation to select good values for **gamma** *and* **cost** *we would see some different results.*

Exercise 5

We have seen that we can fit an SVM with a non-linear kernel in order to perform classification using a non-linear decision boundary by performing logistic regression using non-linear transformations of the features.

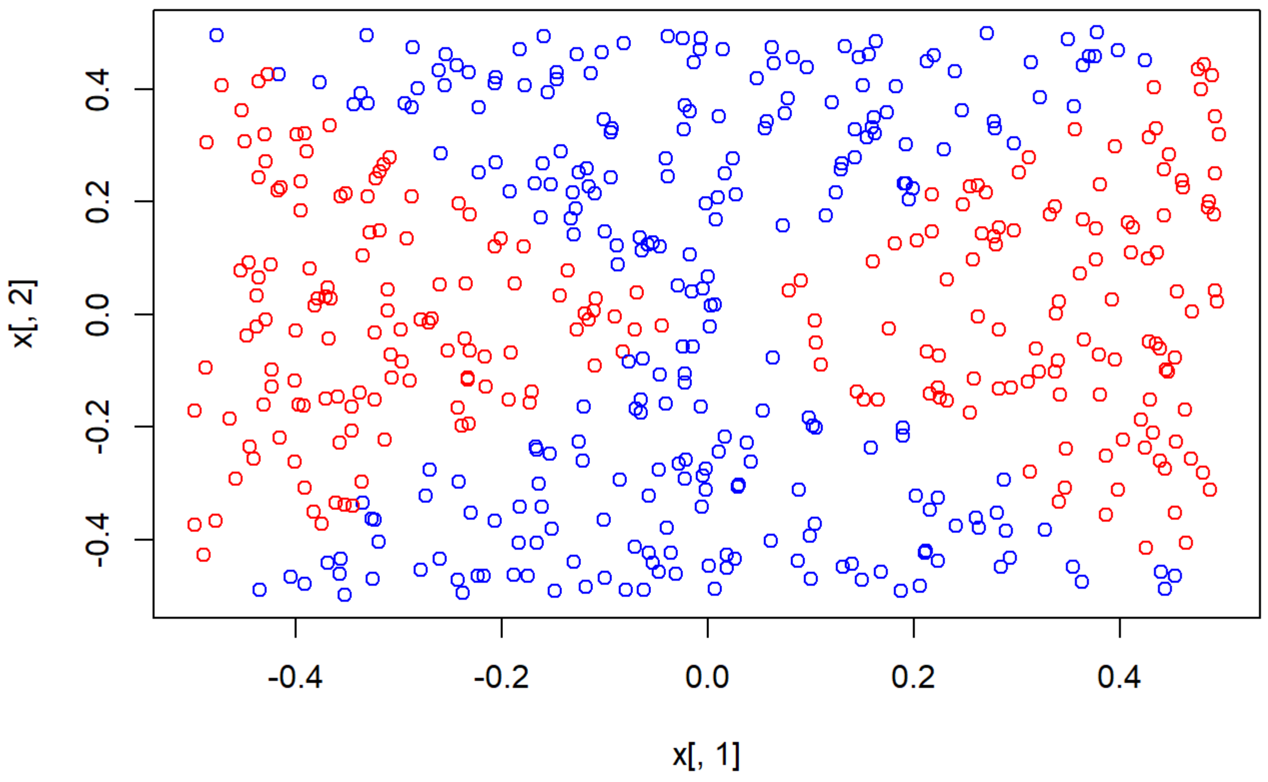
a. Generate a data set with $n = 500$ and $p = 2$, such that the observations belong to two classes with a quadratic decision boundary between them. For instance, you can do this as follows:

```
> x1 = runif(500) - 0.5
> x2 = runif(500) - 0.5
> y = 1 * (x1^2 - x2^2 > 0)
```

```
set.seed(1)
n = 500
p = 2
x = matrix(data = runif(n*p) - 0.5, nrow = n, ncol = p)
y = 1 - 1 * (x[,1]^2 - x[,2]^2 < 0)
```

b. Plot the observations, colored according to their class labels. Your plot should display X_1 on the *x*axis, and X_2 on the *y*-axis.

```
color = rep('blue', n)
color[y == 1] = 'red'
plot(x[,1], x[,2], col = color)
```



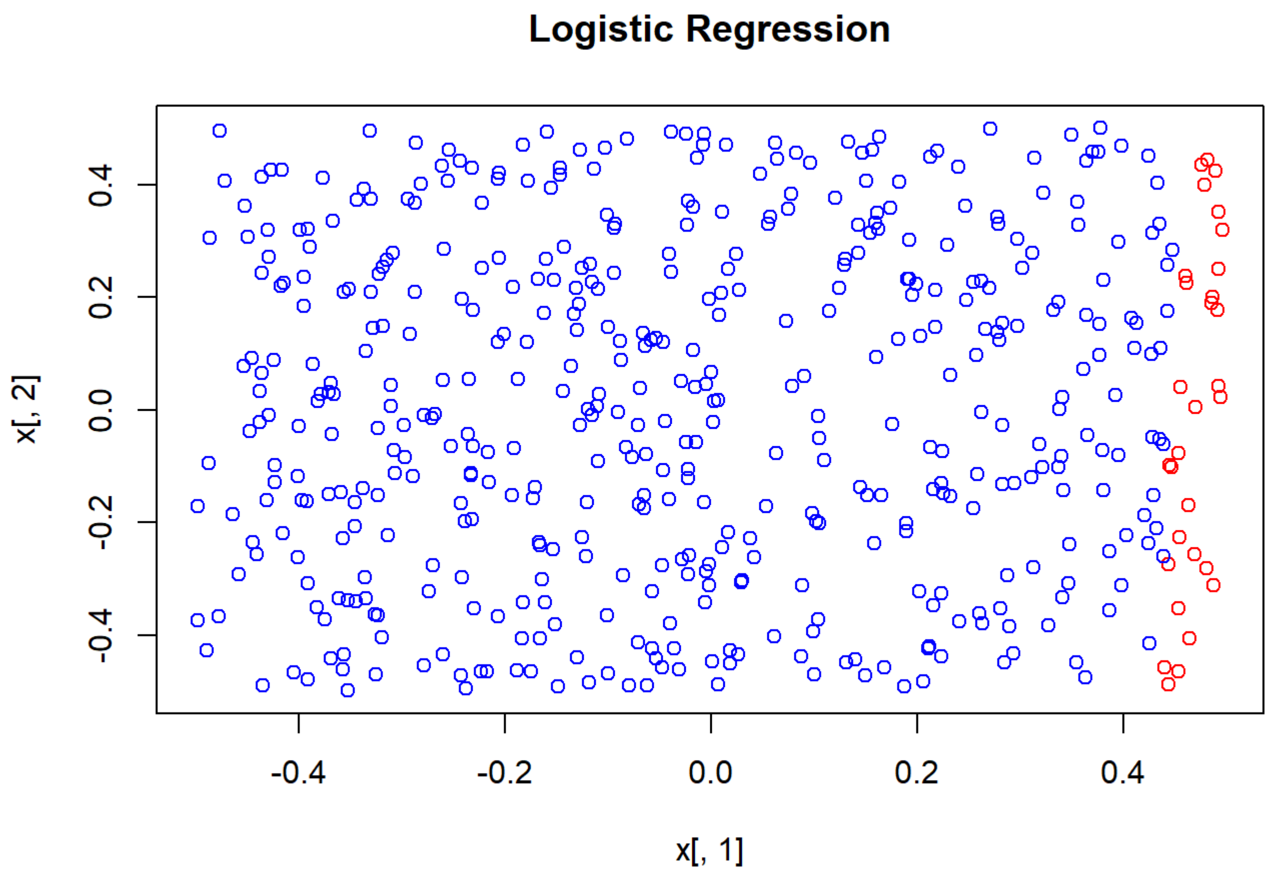
We can see right away that a linear decision boundary won't work.

c. Fit a logistic regression model to the data, using X_1 and X_2 as predictors.

```
glm.fit = glm(y ~ x[,1] + x[,2], family = binomial)
```

d. Apply this model to the *training data* in order to obtain a predicted class label for each training observation. Plot the observations, colored according to the *predicted* class labels. The decision boundary should be linear.

```
glm.prob = predict(glm.fit, type = 'response')
glm.pred = rep('blue', n)
glm.pred[glm.prob > .5] = 'red'
plot(x[,1], x[,2], col = glm.pred, main = 'Logistic Regression')
```



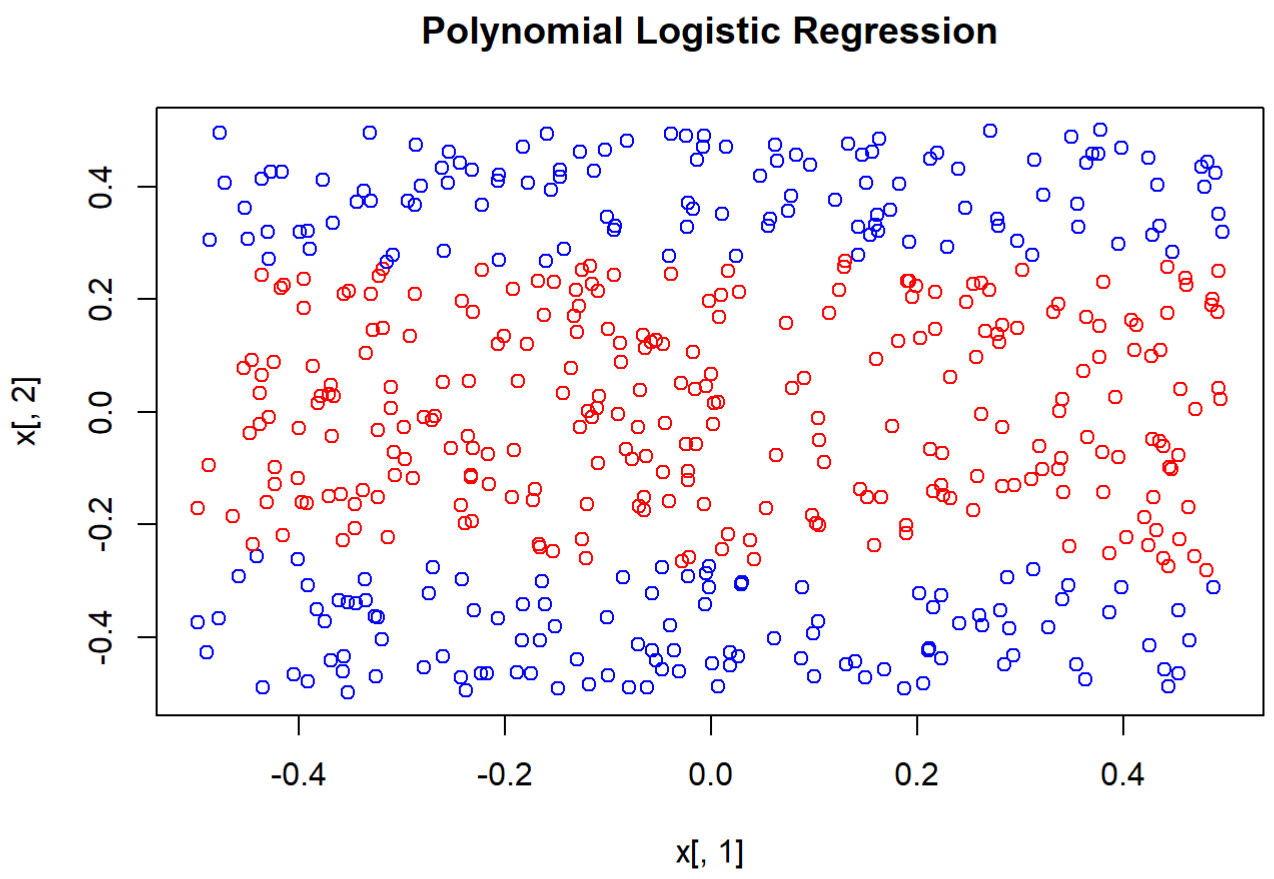
The linear logistic regression doesn't come close.

e. Now fit a logistic regression model to the data using non-linear functions of X_1 and X_2 as predictions (e.g. X_1^2, X_1X_2, $\log X_2$, and so forth).

```
poly.fit = glm(y ~ x[,1] + I(x[,2]^2), family = binomial)
```

f. Apply this model to the *training data* in order to obtain a predicted class label for each training observation. Plot the observations, colored according to the *predicted* class labels. The decision boundary should be obviously non-linear. If it is not, then repeat (a)-(e) until you come up with an example in which the predicted class labels are obviously non-linear.

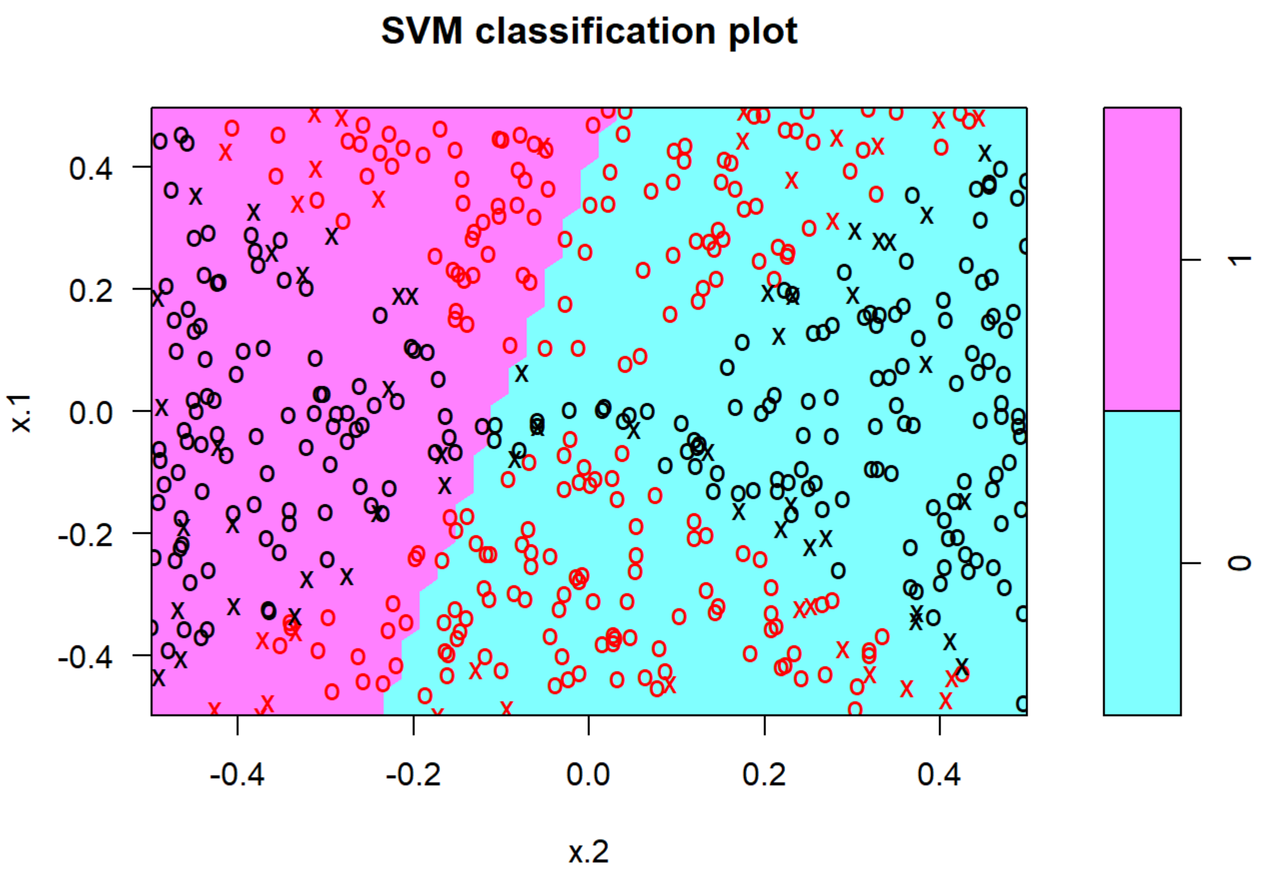
```
poly.prob = predict(poly.fit, type = 'response')
poly.pred = rep('blue', n)
poly.pred[poly.prob > .5] = 'red'
plot(x[,1], x[,2], col = poly.pred, main = 'Polynomial Logistic Regression')
```



The transformation of simply changed the X_2 term to an X_2^2 term greatly improves the fit.

g. Fit a support vector classifier to the data with X_1 and X_2 as predictors. Obtain a class prediction for each training observation. Plot the observations, colored according to the *predicted class labels*.

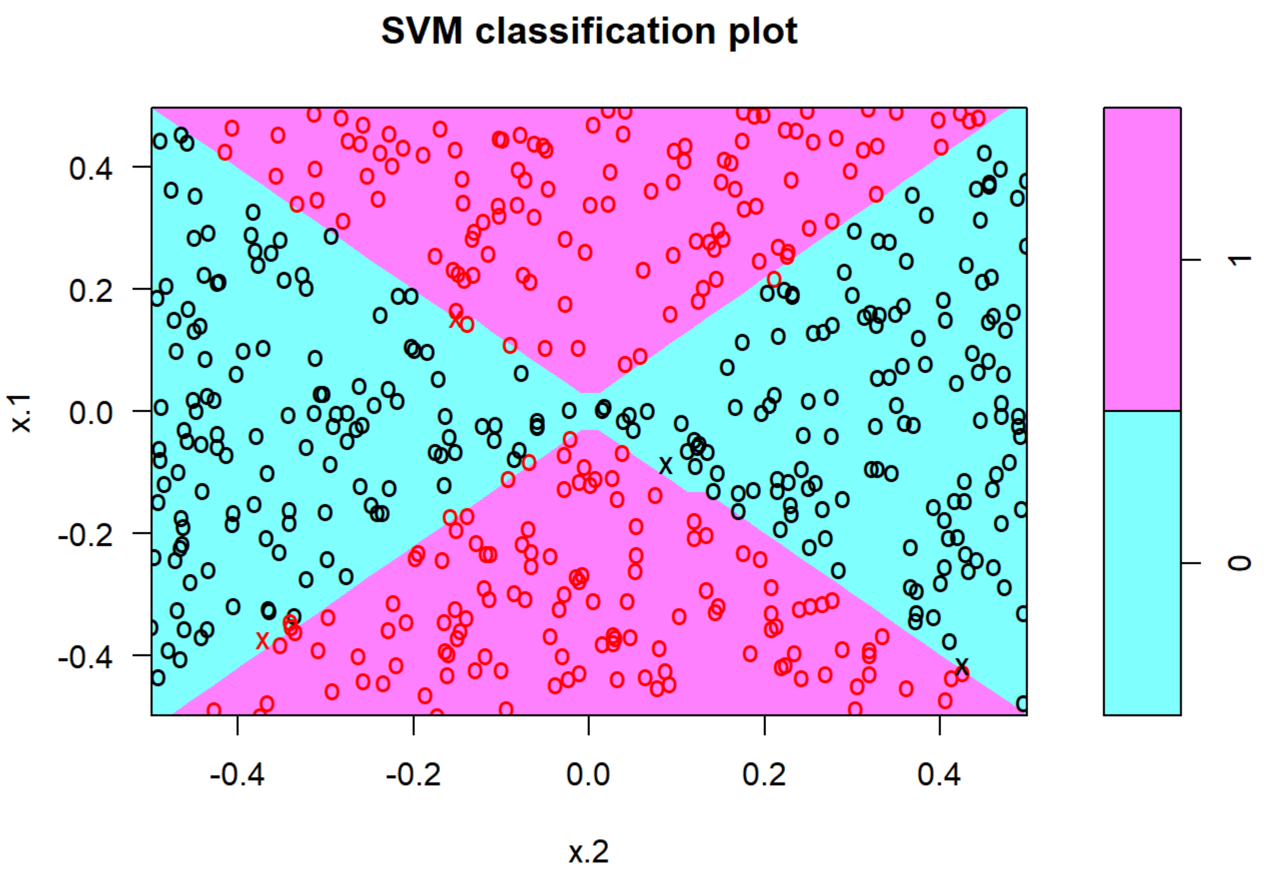
```
dat = data.frame(x = x, y = as.factor(y))
svm.fit = svm(y ~ ., data = dat, kernel = "linear", cost = 1e9, scale = FALSE)
plot(svm.fit, dat)
```



A linear support vector classifier does not work well either.

h. Fit a SVM using a non-linear kernel to the data. Obtain a class prediction for each training observation. Plot the observations, colored according to the *predicted class labels*.

```
svm.fit = svm(y ~ ., data = dat, kernel = "polynomial", degree = 2, cost = 1e9, scale =
FALSE)
plot(svm.fit, dat)
```



The 2nd degree polynomial SVM fit works almost perfectly on the training data.

i. Comment on your results.

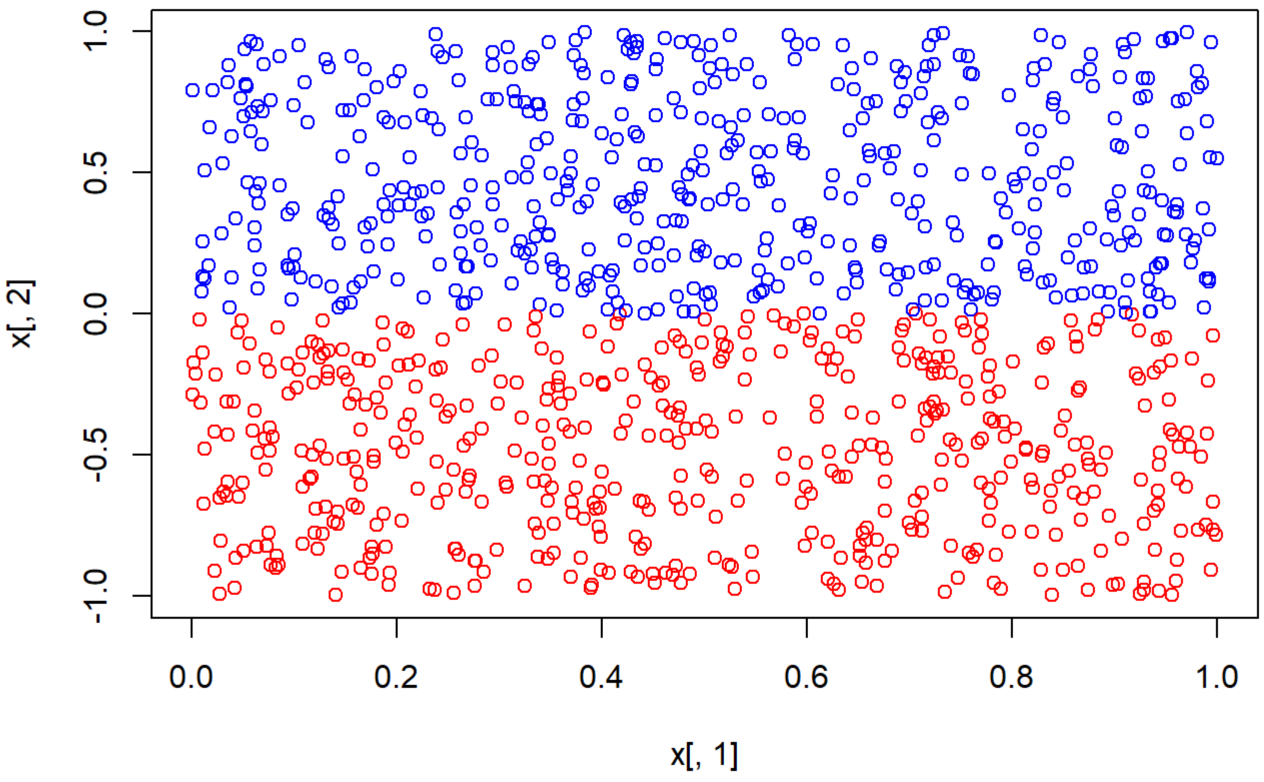
The linear fits using logistic regression and support vector classifier were both very bad at predicting the response for the training data. The polynomial fit using logistic regression performed much better, and the polynomial fit using SVM fit the training data almost perfectly.

Exercise 6

At the end of Section 9.6.1, it is claimed that in the case of data that is just barely linearly separable, a support vector classifier with a small value of **cost** that misclassifies a couple of training observations may perform better on test data than one with a huge value of **cost** that does not misclassify any training observations. You will now investigate this claim.

a. Generate two-class data with $p = 2$ in such a waay that the classes are just barely linearly separable.

```
set.seed(1)
n = 1000
p = 2
x = matrix(data = runif(n * p, min = 0, max = 1), nrow = n, ncol = p)
class.index = sample.int(n, size = n/2, replace = FALSE)
x[class.index, 2] = x[class.index, 2] - 1
color = rep('blue', n)
color[class.index] = 'red'
plot(x[,1], x[,2], col = color)
```



b. Compute the cross-validation error rates for support vector classifiers with a range of **cost** values. How many training errors are misclassified for each value of **cost** considered, and how does this relate to the cross-validation errors obtained?

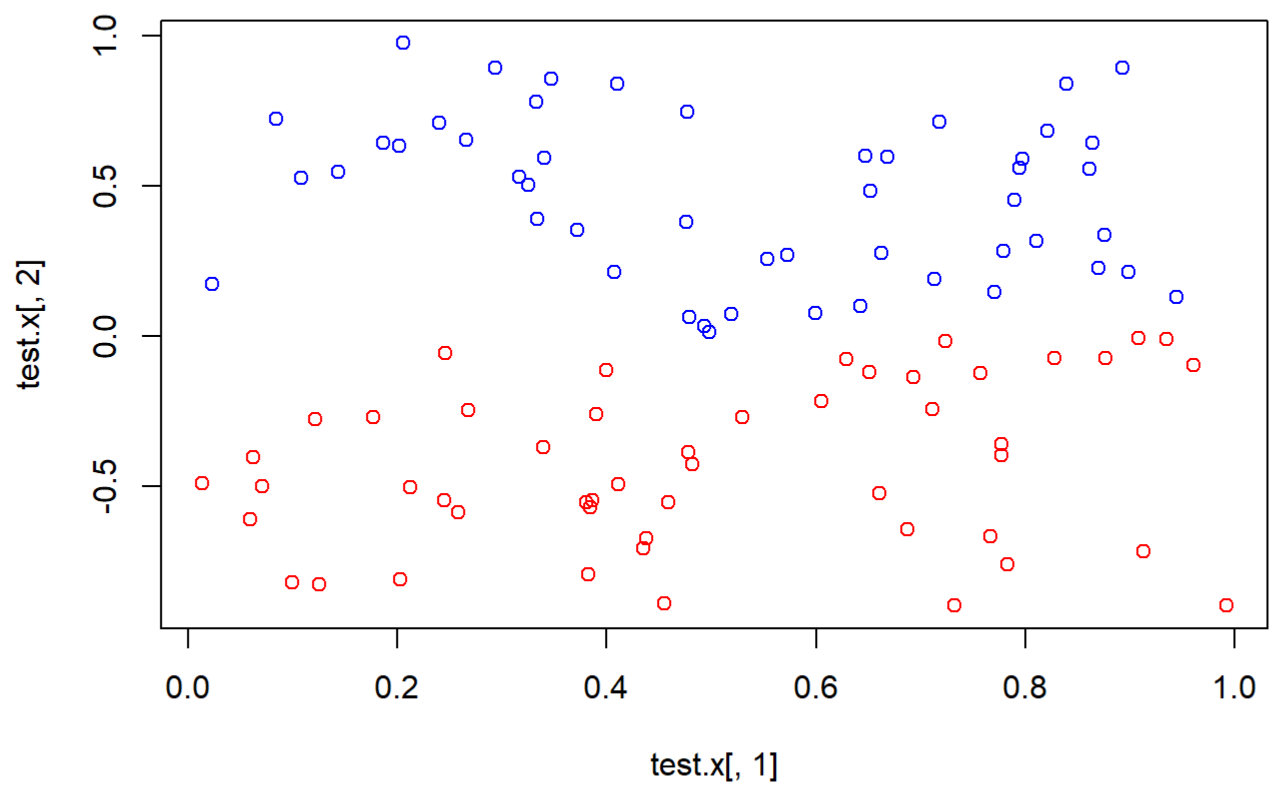
```
set.seed(1)
dat = data.frame(x = x, y = as.factor(color))
cost.list = c(0.001, 1, 100, 10000)
tune.out = tune(svm, y ~ ., data = dat, kernel = "linear", ranges = list(cost = cost.lis
t))
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   cost
##  10000
##
## - best performance: 0.002
##
## - Detailed performance results:
##    cost error  dispersion
## 1 1e-03 0.010 0.008164966
## 2 1e+00 0.006 0.008432740
## 3 1e+02 0.004 0.005163978
## 4 1e+04 0.002 0.004216370
```

The high cost model performed the best on the training data, misclassifying only 2 training observations with **cost=10000**. *With* **cost=100**, *4 training observations were misclassified. With* **cost=1**, *6 training observations were misclassified. With* **cost=0.001**, *10 training observations were misclassified. So, as expected, increasing* **cost** *always yields better performance on the training data.*

c. Generate an appropriate test data set, and compute the test errors corresponding to each of the values of **cost** considered. Which value of **cost** leads to the fewest test errors, and how does this compare to the values of **cost** that yield the fewest training errors and the fewest cross-validation errors?

```
set.seed(1)
test.n = 100
test.x = matrix(data = runif(test.n * p, min = 0, max = 1), nrow = test.n, ncol = p)
test.class = sample.int(test.n, size = test.n / 2, replace = F)
test.color = rep('blue', test.n)
test.color[test.class] = 'red'
test.x[test.class, 2] = test.x[test.class, 2] - 1
test.dat = data.frame(x = test.x, y = as.factor(test.color))
plot(test.x[,1], test.x[,2], col = test.color)
```



So now we have our test data - time to make predictions.

```r
for (cost in cost.list) {
  svm.fit = svm(y ~ ., data = dat, kernel = "linear", cost = cost)
  test.pred = predict(svm.fit, test.dat)
  test.error = mean(test.pred != test.dat$y)
  print(paste('cost:', cost, ' --- ', 'test error:', test.error))
}
```

```
## [1] "cost: 0.001  ---  test error: 0.02"
## [1] "cost: 1  ---  test error: 0.02"
## [1] "cost: 100  ---  test error: 0"
## [1] "cost: 10000  ---  test error: 0"
```

d. Discuss your results.

So in this case, the high cost model still performed better on the test data. This result is only because the test data that I generated so closely mirrors the training data. The general rule of using a lower cost model for barely separable classes doesn't apply if the training data and test data both obey the same margins and there is a sufficient amount of training data.

Exercise 7

In this problem, you will use support vector approaches in order to predict whether a given car gets high or low gas mileage based on the **Auto** data set.

```
fh='D:\\GoogleDrive\\Introduction to Statistical Learning with Applications in R\\data-s
ets\\Auto.csv'
Auto = read.csv(file=fh,header=TRUE)
```

a. Create a binary variable that takes on a 1 for cars with gas mileage above the median, and a 0 for cars with gas mileage below the median.

```
mpg.median = median(Auto$mpg)
n = nrow(Auto)
# note that any mpg that is exactly equal to the median will also yield a 1
mpg.factor = rep(1, n)
mpg.factor[Auto$mpg < mpg.median] = 0
```

b. Fit a support vector classifier to the data with various values of **cost**, in order to predict whether a car gets high or low gas mileage. Report the cross-validation errors associated with different values of this parameter. Comment on your results.

```
set.seed(1)
# need to remove mpg in the training observations since you really wouldn't use the othe
r factors to predict mpg if you already have mpg available
dat = data.frame(x = as.matrix(subset(Auto, select = -c(mpg))), y = as.factor(mpg.facto
r))
cost.list = c(0.001, 0.01, 0.1, 1, 10, 100, 1000)
linear.out = tune(svm, y ~ ., data = dat, kernel = "linear", ranges = list(cost = cost.l
ist))
summary(linear.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##     1
##
## - best performance: 0.09064103
##
## - Detailed performance results:
##    cost      error dispersion
## 1 1e-03 0.53410256 0.03405530
## 2 1e-02 0.09314103 0.04406587
## 3 1e-01 0.09564103 0.04519299
## 4 1e+00 0.09064103 0.03954689
## 5 1e+01 0.09064103 0.03954689
## 6 1e+02 0.09064103 0.03954689
## 7 1e+03 0.09064103 0.03954689
```

As expected, training error goes down as **cost** *increases. However, this has diminishing returns. Therefore, we can conclude that certainly any* **cost** *greater than 1 is overfitting the training data.*

c. Now repeat (b), this time using SVMs with radial and polynomial basis kernels, with different values of **gamma** and **degree** and **cost**. Comment on your results.

```
set.seed(1)
gamma.list = c(0.5, 1, 2, 3, 4)
radial.out = tune(svm, y ~ ., data = dat, kernel = "radial", ranges = list(cost = cost.l
ist, gamma = gamma.list))
summary(radial.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##     1   0.5
##
## - best performance: 0.08307692
##
## - Detailed performance results:
##     cost gamma      error dispersion
## 1  1e-03   0.5 0.53410256 0.03405530
## 2  1e-02   0.5 0.53410256 0.03405530
## 3  1e-01   0.5 0.53410256 0.03405530
## 4  1e+00   0.5 0.08307692 0.02368462
## 5  1e+01   0.5 0.08307692 0.02896099
## 6  1e+02   0.5 0.08307692 0.02896099
## 7  1e+03   0.5 0.08307692 0.02896099
## 8  1e-03   1.0 0.53410256 0.03405530
## 9  1e-02   1.0 0.53410256 0.03405530
## 10 1e-01   1.0 0.53410256 0.03405530
## 11 1e+00   1.0 0.46641026 0.13042249
## 12 1e+01   1.0 0.44128205 0.12523088
## 13 1e+02   1.0 0.44128205 0.12523088
## 14 1e+03   1.0 0.44128205 0.12523088
## 15 1e-03   2.0 0.53410256 0.03405530
## 16 1e-02   2.0 0.53410256 0.03405530
## 17 1e-01   2.0 0.53410256 0.03405530
## 18 1e+00   2.0 0.53153846 0.03247413
## 19 1e+01   2.0 0.53153846 0.03247413
## 20 1e+02   2.0 0.53153846 0.03247413
## 21 1e+03   2.0 0.53153846 0.03247413
## 22 1e-03   3.0 0.53410256 0.03405530
## 23 1e-02   3.0 0.53410256 0.03405530
## 24 1e-01   3.0 0.53410256 0.03405530
## 25 1e+00   3.0 0.53153846 0.03247413
## 26 1e+01   3.0 0.53153846 0.03247413
## 27 1e+02   3.0 0.53153846 0.03247413
## 28 1e+03   3.0 0.53153846 0.03247413
## 29 1e-03   4.0 0.53410256 0.03405530
## 30 1e-02   4.0 0.53410256 0.03405530
## 31 1e-01   4.0 0.53410256 0.03405530
## 32 1e+00   4.0 0.53153846 0.03247413
## 33 1e+01   4.0 0.53153846 0.03247413
## 34 1e+02   4.0 0.53153846 0.03247413
## 35 1e+03   4.0 0.53153846 0.03247413
```

With the radial kernel, we again observe **cost=1** *as fitting the data best. In this case,* **gamma=0.5** *also best fits the training data. When using the radial kernel, increasing* **cost** *too much actually starts to increase the training error obtained from the cross-validation.*

```
set.seed(1)
degree.list = c(2, 3, 4, 5)
poly.out = tune(svm, y ~ ., data = dat, kernel = "polynomial", ranges = list(cost = cos
t.list, degree = degree.list))
summary(poly.out)
```

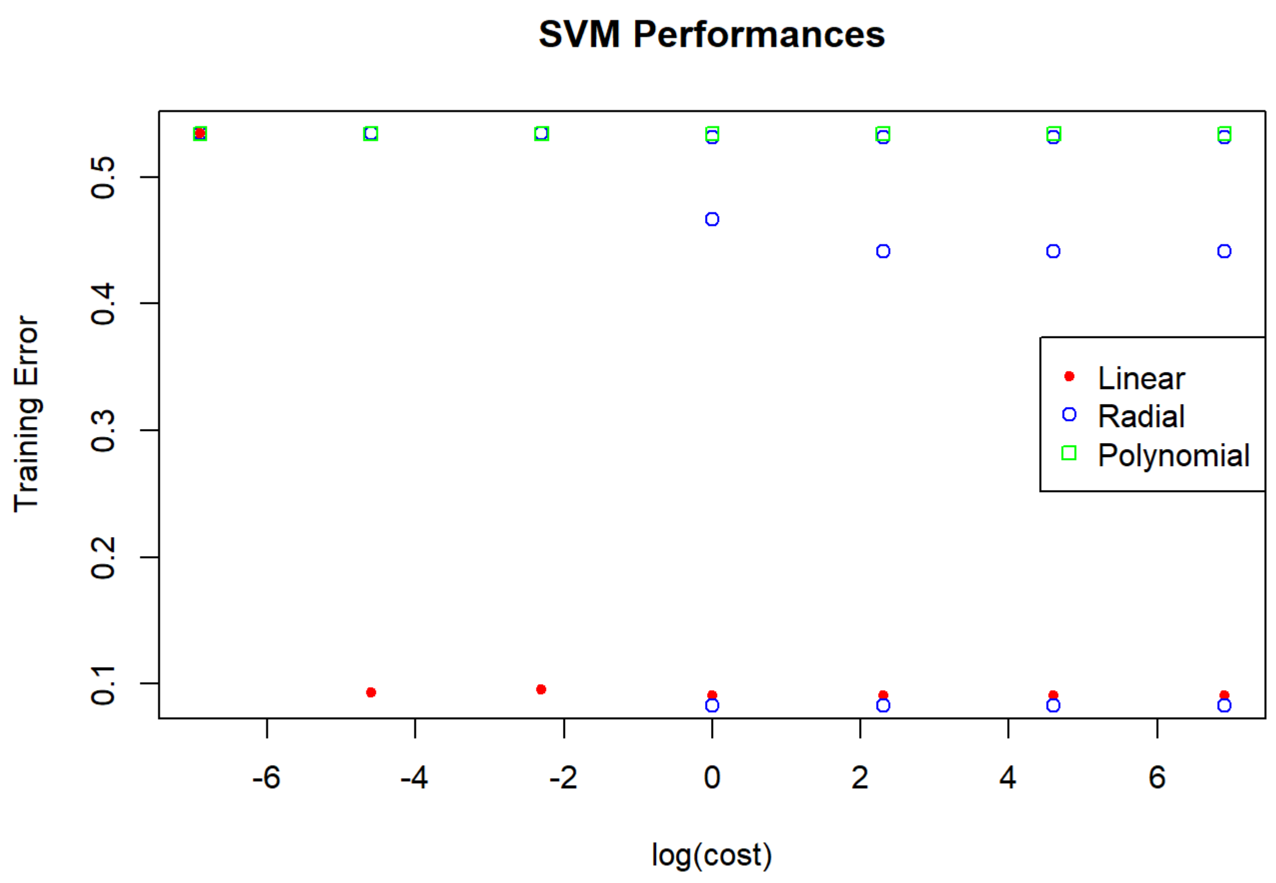
```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   cost degree
##  0.001      2
##
## - best performance: 0.5341026
##
## - Detailed performance results:
##     cost degree     error dispersion
## 1  1e-03      2 0.5341026  0.0340553
## 2  1e-02      2 0.5341026  0.0340553
## 3  1e-01      2 0.5341026  0.0340553
## 4  1e+00      2 0.5341026  0.0340553
## 5  1e+01      2 0.5341026  0.0340553
## 6  1e+02      2 0.5341026  0.0340553
## 7  1e+03      2 0.5341026  0.0340553
## 8  1e-03      3 0.5341026  0.0340553
## 9  1e-02      3 0.5341026  0.0340553
## 10 1e-01      3 0.5341026  0.0340553
## 11 1e+00      3 0.5341026  0.0340553
## 12 1e+01      3 0.5341026  0.0340553
## 13 1e+02      3 0.5341026  0.0340553
## 14 1e+03      3 0.5341026  0.0340553
## 15 1e-03      4 0.5341026  0.0340553
## 16 1e-02      4 0.5341026  0.0340553
## 17 1e-01      4 0.5341026  0.0340553
## 18 1e+00      4 0.5341026  0.0340553
## 19 1e+01      4 0.5341026  0.0340553
## 20 1e+02      4 0.5341026  0.0340553
## 21 1e+03      4 0.5341026  0.0340553
## 22 1e-03      5 0.5341026  0.0340553
## 23 1e-02      5 0.5341026  0.0340553
## 24 1e-01      5 0.5341026  0.0340553
## 25 1e+00      5 0.5341026  0.0340553
## 26 1e+01      5 0.5341026  0.0340553
## 27 1e+02      5 0.5341026  0.0340553
## 28 1e+03      5 0.5341026  0.0340553
```

Not very good performance here. All of the polynomial decision boundaries of order greater than a linear boundary seem to have the same training error. We can therefore conclude that these boundaries have small higher degree terms. That is to say that they are approximately linear.

Of the three kernel types, a radial kernel yields the best performance on the training data. However, that training error rate is only slightly lower than the linear decision SVM. Therefore, we may still prefer the linear SVM if we want to reduce variability.

d. Make some plots to back up your assertions in (b) and (c).

```
plot(log(linear.out$performances$cost), linear.out$performances$error, xlab = "log(cos
t)", ylab = "Training Error", main = "SVM Performances", pch = 20, col = 'red')
points(log(radial.out$performances$cost), radial.out$performances$error, pch = 21, col =
'blue')
points(log(poly.out$performances$cost), poly.out$performances$error, pch = 22, col = 'gr
een')
legend('right', legend = c('Linear', 'Radial', 'Polynomial'), pch = c(20, 21, 22), col =
c('red', 'blue', 'green'))
```



On this plot, we can see that linear and radial SVMs yield the smallest training error rates, and the radial error rate is only slightly less than the linear error rate.

Exercise 8

This problem involves the **OJ** dat set which is part of the **ISLR** package.

```
?OJ
```

```
## starting httpd help server ... done
```

a. Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations.

```
set.seed(1)
n = nrow(OJ)
train = sample(n, size = 800)
```

b. Fit a support vector classifier to the training data using **cost=0.01**, with **Purchase** as the response and the other variables as predictors. Use the **summary()** function to produce summary statistics, and describe the results obtained.

```
svm.fit = svm(Purchase ~ ., data = OJ[train,], kernel = "linear", cost = 0.01)
summary(svm.fit)
```

```
##
## 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
```

More than half of the training observations are support vectors, so we can say that there is probably a high degree of variability in this model.

c. What are the training and test error rates?

```
train.error = mean(predict(svm.fit, OJ[train,]) != OJ$Purchase[train])
test.error = mean(predict(svm.fit, OJ[-train,]) != OJ$Purchase[-train])
print(paste('train error:', train.error))
```

```
## [1] "train error: 0.16625"
```

```
print(paste('test error:', test.error))
```

```
## [1] "test error: 0.181481481481481"
```

d. Use the **tune()** function to select an optimal **cost**. Consider values in the range 0.01 to 10.

```
set.seed(1)
cost.list = c(0.01, 0.1, 1, 10)
tune.out = tune(svm, Purchase ~ ., data = OJ[train,], kernel = "linear", ranges = list(c
ost = cost.list))
summary(tune.out)
```

```
##
## 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 10.00 0.18000 0.04005205
```

e. Compute the training and test error rates using this new value for **cost**.

```
train.error = mean(predict(tune.out$best.model, OJ[train,]) != OJ$Purchase[train])
test.error = mean(predict(tune.out$best.model, OJ[-train,]) != OJ$Purchase[-train])
print(paste('train error:', train.error))
```

```
## [1] "train error: 0.16625"
```

```
print(paste('test error:', test.error))
```

```
## [1] "test error: 0.181481481481481"
```

Since the cross-validation selected the same value for **cost**, *it isn't surprising that the error rates are exactly the same as before.*

f. Repeat parts (b) through (e) using a support vector machine with a radial kernel. Use the default value for **gamma**.

```
svm.fit = svm(Purchase ~ ., data = OJ[train,], kernel = "radial", cost = 0.01)
train.error = mean(predict(svm.fit, OJ[train,]) != OJ$Purchase[train])
test.error = mean(predict(svm.fit, OJ[-train,]) != OJ$Purchase[-train])
print('Without cross-validation:')
```

```
## [1] "Without cross-validation:"
```

```
print(paste('train error:', train.error))
```

```
## [1] "train error: 0.3825"
```

```
print(paste('test error:', test.error))
```

```
## [1] "test error: 0.411111111111111"
```

```
set.seed(1)
tune.out = tune(svm, Purchase ~ ., data = OJ[train,], kernel = "radial", ranges = list(c
ost = cost.list))
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##     1
##
## - best performance: 0.175
##
## - 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 10.00 0.18250 0.04866267
```

```
train.error = mean(predict(tune.out$best.model, OJ[train,]) != OJ$Purchase[train])
test.error = mean(predict(tune.out$best.model, OJ[-train,]) != OJ$Purchase[-train])
print('With cross-validation:')
```

```
## [1] "With cross-validation:"
```

```
print(paste('train error:', train.error))
```

```
## [1] "train error: 0.145"
```

```
print(paste('test error:', test.error))
```

```
## [1] "test error: 0.17037037037037"
```

g. Repeat parts (b) through (e) using a support vector machine with a polynomial kernel. Set **degree=2**.

```
svm.fit = svm(Purchase ~ ., data = OJ[train,], kernel = "polynomial", degree = 2, cost =
0.01)
train.error = mean(predict(svm.fit, OJ[train,]) != OJ$Purchase[train])
test.error = mean(predict(svm.fit, OJ[-train,]) != OJ$Purchase[-train])
print('Without cross-validation:')
```

```
## [1] "Without cross-validation:"
```

```
print(paste('train error:', train.error))
```

```
## [1] "train error: 0.3825"
```

```
print(paste('test error:', test.error))
```

```
## [1] "test error: 0.411111111111111"
```

```
set.seed(1)
tune.out = tune(svm, Purchase ~ ., data = OJ[train,], kernel = "polynomial", degree = 2,
ranges = list(cost = cost.list))
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##    10
##
## - best performance: 0.17875
##
## - 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 10.00 0.17875 0.05653477
```

```
train.error = mean(predict(tune.out$best.model, OJ[train,]) != OJ$Purchase[train])
test.error = mean(predict(tune.out$best.model, OJ[-train,]) != OJ$Purchase[-train])
print('With cross-validation:')
```

```
## [1] "With cross-validation:"
```

```
print(paste('train error:', train.error))
```

```
## [1] "train error: 0.145"
```

```
print(paste('test error:', test.error))
```

```
## [1] "test error: 0.185185185185185"
```

h. Overall, which approach seems to give the best results on this data?

All the different SVM types preferred a different value of **cost**. *The best performance on the training data came from the quadratic decision boundary SVM with* **cost=10**. *The best performance on the test data came from the radial SVM with* **cost=1**. *The cross-validation improved the performances with the radial and polynomial models.*