Lab 09 - SVMs

Ken Ye

2023-11-08

SVM Lab

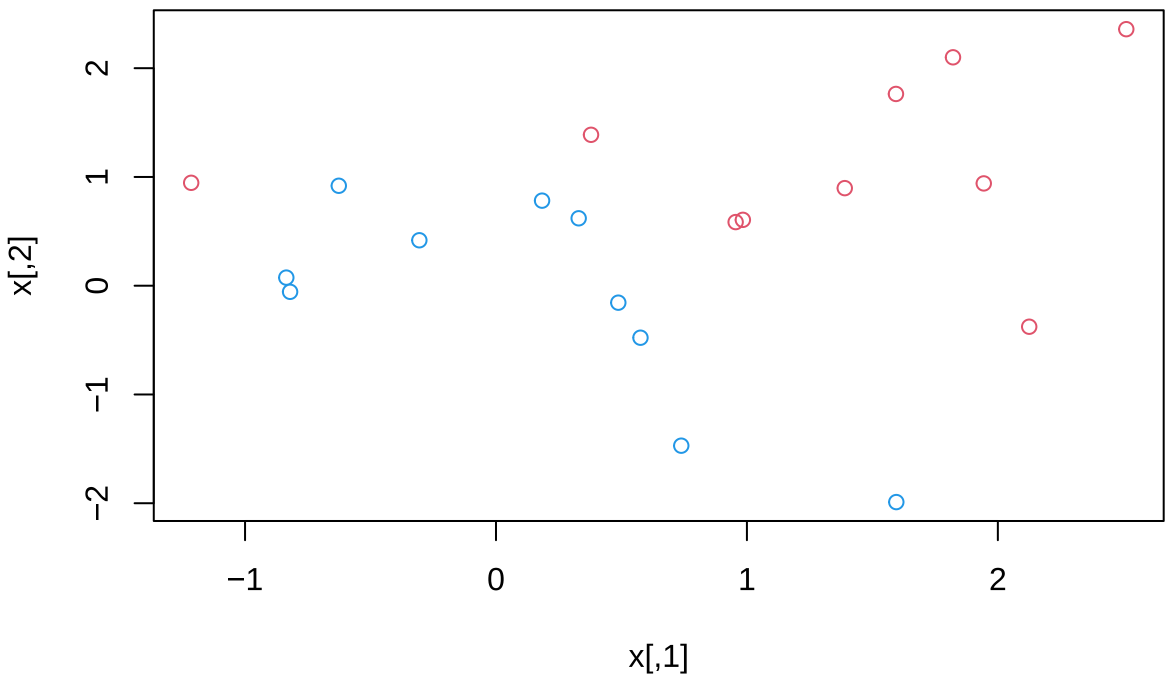
We will use the `e1071` library in `R` to demonstrate the support vector classifier and the SVM. Another option is the `LiblinearR` library, which is useful for very large linear problems.

9.1 Support Vector Classifier

The `e1071` library contains implementations for a number of statistical learning methods. In particular, the `svm()` function can be used to fit a support vector classifier when the argument `kernel = "linear"` is used. This function uses a slightly different formulation for the support vector classifier than the text. A `cost` argument allows us to specify the cost of a violation to the margin. When the `cost` argument is small, then the margins will be wide and many support vectors will be on the margin or will violate the margin. When the `cost` argument is large, then the margins will be narrow and there will be few support vectors on the margin or violating the margin.

We now use the `svm()` function to fit the support vector classifier for a given value of the `cost` parameter. Here we demonstrate the use of this function on a two-dimensional example so that we can plot the resulting decision boundary. We begin by generating the observations, which belong to two classes, and checking whether the classes are linearly separable.

```r
set.seed(1)
x <- matrix(rnorm(20*2), ncol = 2)
y <- c(rep(-1,10), rep(1,10))
x[y==1,] = x[y==1,] + 1
plot(x, col = (3-y))
```



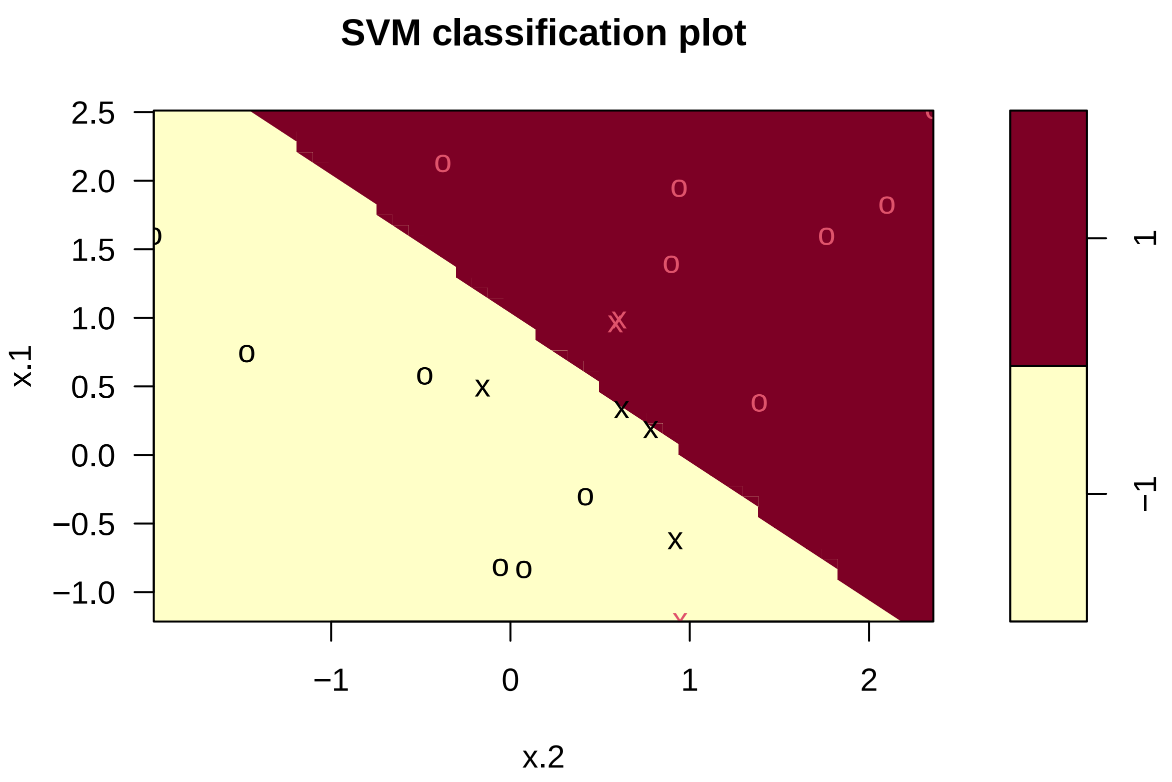
These points are not linearly separable. Next, we fit the support vector classifier. Note that in order for the `svm()` function to perform classification (as opposed to SVM-based regression), we must encode the response as a factor variable. We now create a data frame with the response coded as a factor.

```
dat <- data.frame(x = x, y = as.factor(y))
library(e1071)
svmfit <- svm(y ~ ., data = dat, kernel = "linear", cost = 10,
              scale = FALSE)
```

The argument `scale = FALSE` tells the `svm()` function not to scale each feature to have mean zero or standard deviation one; depending on the application, one might prefer to use `scale = TRUE`.

We can now plot the support vector classifier obtained:

```
plot(svmfit, dat)
```



Note that the two arguments to the `plot()` function are the output of the call to `svm()` and the data used in the call to `svm()`. The region of feature space that will be assigned to the −1 class is shown in yellow and the region that will be assigned to the +1 class is shown in dark red. The decision boundary between the two classes is linear (since we used the argument `kernel = "linear"`), though due to the way in which the plotting function is implemented in this library the decision boundary looks jagged in the plot. We see that in this case only one observation is misclassified. (Note that here the second feature is plotted on the x-axis and the first feature is plotted on the y-axis, in contrast to the usual behavior of `plot()`). The support vectors are plotted as crosses and the remaining observations are plotted as circles; we see here that there are seven support vectors. We can determine their identities as follows:

```
svmfit$index
```

```
## [1]  1  2  5  7 14 16 17
```

We can also obtain some basic information about the support vector classifier fit using the `summary()` command:

```
summary(svmfit)
```

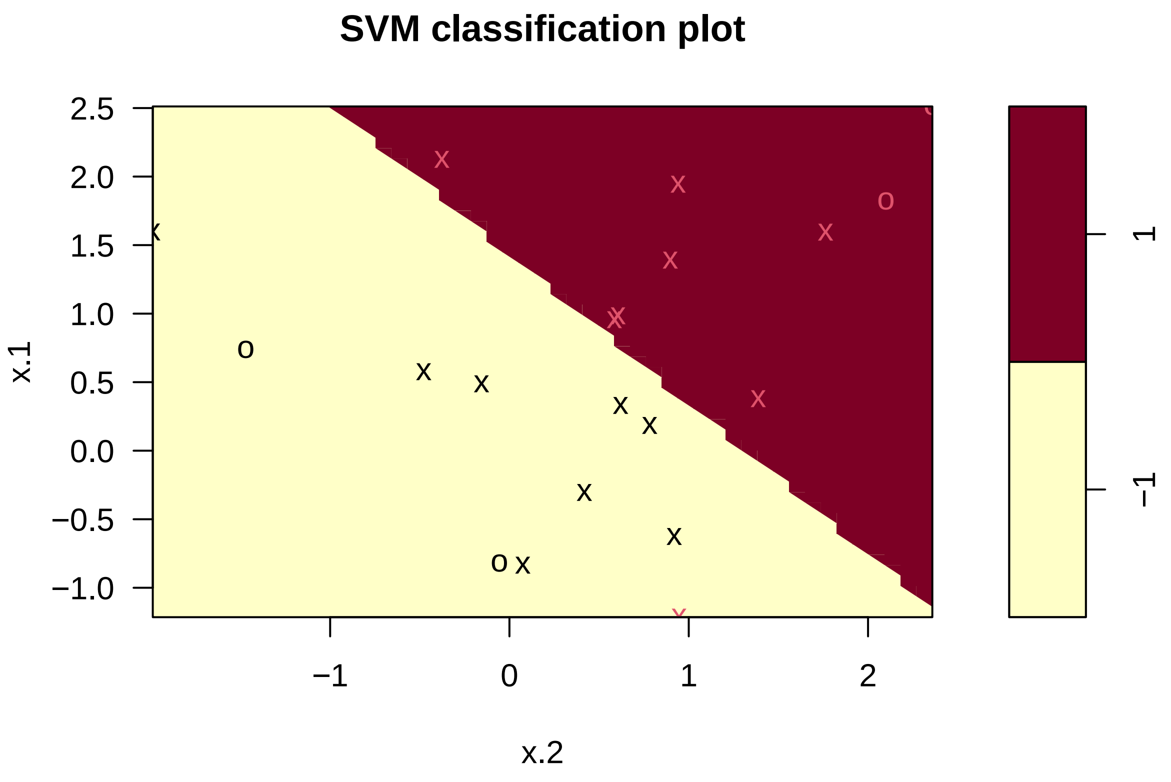
```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10, scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
```

```
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

This tells us, for instance, that a linear kernel was used with `cost = 10`, and that there were seven support vectors, four in one class and three in the other.

What if we instead used a smaller value of the cost parameter?

```
svmfit <- svm(y ~ ., data = dat, kernel = "linear",
              cost = 0.1, scale = FALSE)
plot(svmfit, dat)
```



```
svmfit$index
```

```
##  [1]  1  2  3  4  5  7  9 10 12 13 14 15 16 17 18 20
```

Now that a smaller value of the cost parameter is being used, we obtain a larger number of support vectors, because the margin is now wider. Unfortunately, the `svm()` function does not explicitly output the coefficients of the linear decision boundary obtained when the support vector classifier is fit, nor does it output the width of the margin.

The `e1071` library includes a built-in function, `tune()`, to perform cross-validation. By default, `tune()` performs ten-fold cross-validation on a set of models of interest. In order to use this function, we pass in relevant information about the set of models that are under consideration. The following command indicates that we want to compare SVMs with a linear kernel, using a range of values of the `cost` parameter.

```
set.seed(1)
tune.out <- tune(svm, y ~ ., data = dat, kernel = "linear",
                 ranges = list(cost = c(0.001, 0.01, 0.1,
                                        1, 5, 10, 100)))
```

We can easily access the cross-validation errors for each of these models using the `summary()` command:

```
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##   0.1
##
## - best performance: 0.05
##
## - Detailed performance results:
##    cost error dispersion
## 1 1e-03  0.55  0.4377975
## 2 1e-02  0.55  0.4377975
## 3 1e-01  0.05  0.1581139
## 4 1e+00  0.15  0.2415229
## 5 5e+00  0.15  0.2415229
## 6 1e+01  0.15  0.2415229
## 7 1e+02  0.15  0.2415229
```

We see that `cost = 0.1` results in the lowest cross-validation error rate. The `tune()` function stores the best model obtained, which can be accessed as follows:

```
bestmod <- tune.out$best.model
summary(bestmod)
```

```
##
## Call:
## best.tune(METHOD = svm, train.x = y ~ ., data = dat, 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:  0.1
##
```

```
## Number of Support Vectors:  16
##
##  ( 8 8 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

The `predict()` function can be used to predict the class label on a set of test observations, at any given value of the cost parameter. We begin by generating a test data set.

```
xtest <- matrix(rnorm(20*2), ncol = 2)
ytest <- sample(c(-1,1), 20, rep = TRUE)
xtest[ytest == 1, ] = xtest[ytest == 1, ] + 1
testdat <- data.frame(x = xtest, y = as.factor(ytest))
```

Now, we predict the class labels of these test observations. Here we use the best model obtained through cross-validation in order to make the predictions.

```
ypred <- predict(bestmod, testdat)
table(predict = ypred, truth = testdat$y)
```

```
##        truth
## predict -1 1
##      -1  9 1
##      1   2 8
```

Thus, with this value of `cost`, 17 of the test observations are classified correctly. What if we had instead used `cost = 0.01`?

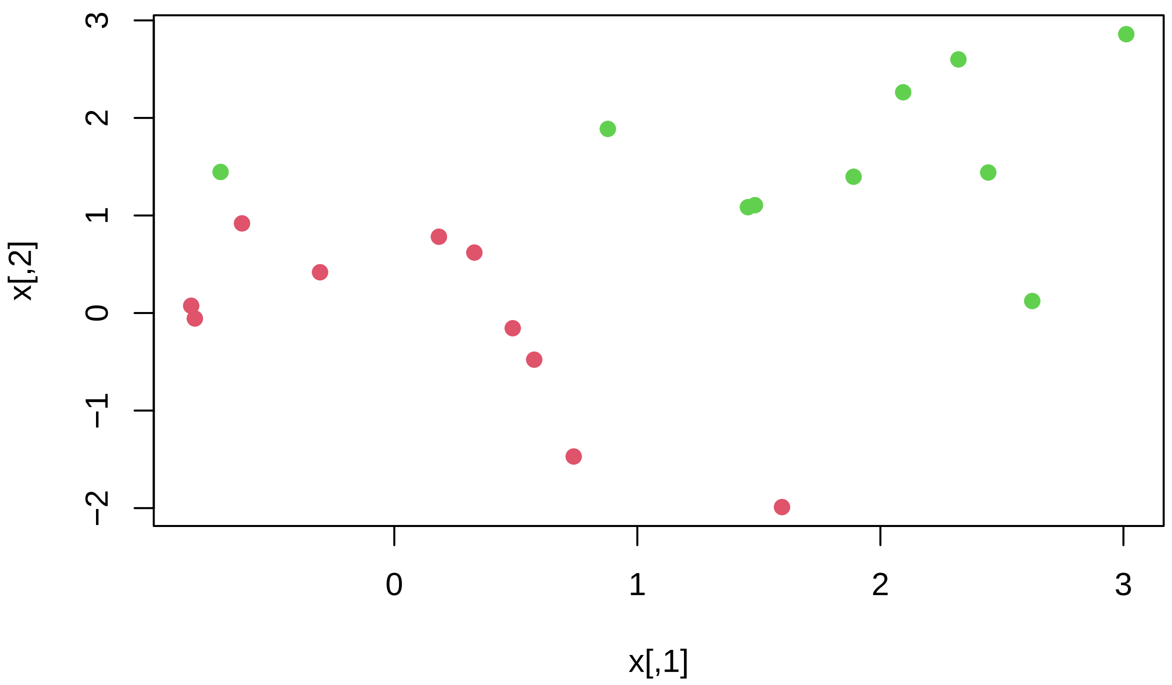
```
svmfit <- svm(y ~ ., data = dat, kernel = "linear", cost = 0.01,
              scale = FALSE)
ypred <- predict(svmfit, testdat)
table(predict = ypred, truth = testdat$y)
```

```
##        truth
## predict -1  1
##      -1 11  6
##      1   0  3
```

Now, 3 additional points are misclassified.

Now consider a situation in which the two classes are linearly separable. Then, we can find a separating hyperplane using the `svm()` function. We first further separate the two classes in our simulated data so that they are linearly separable.

```
x[y==1, ] <- x[y==1,] + 0.5
plot(x, col = (y+5)/2, pch = 19)
```

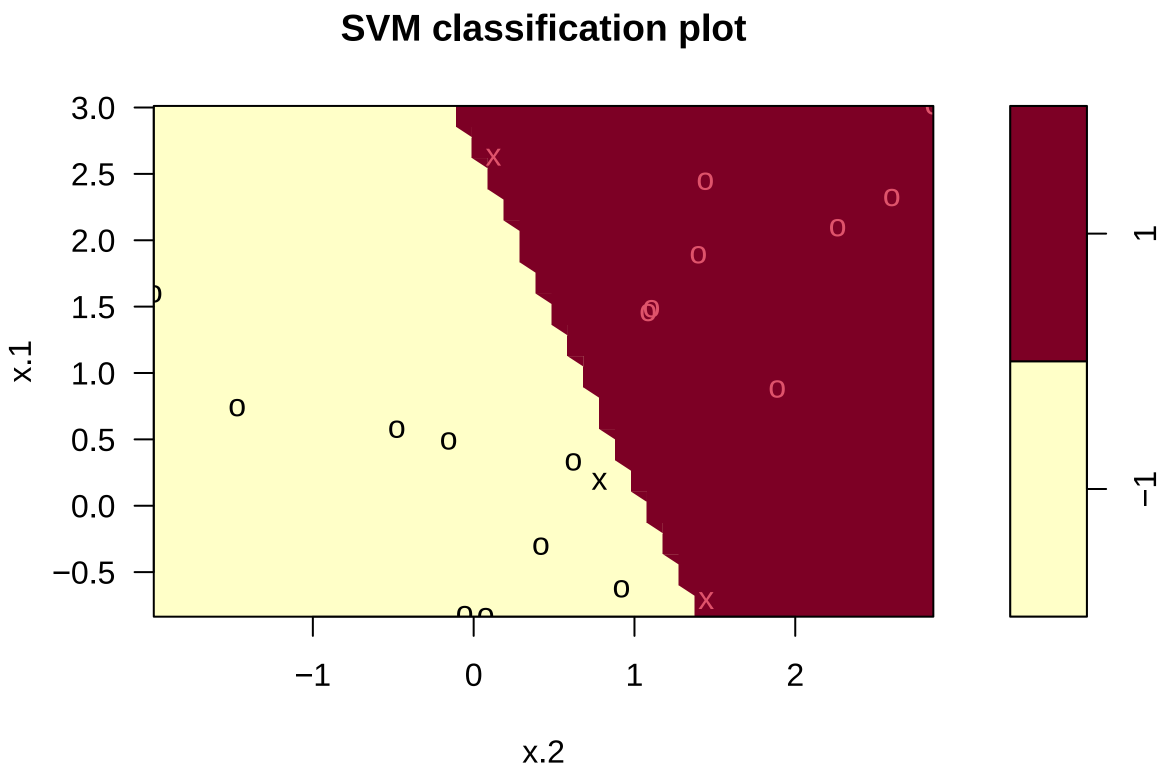


Now the observations are just barely linearly separable. We fit the support vector classifier and plot the resulting hyperplane, using a very large value of `cost` so that no observations are misclassified.

```r
dat <- data.frame(x = x, y = as.factor(y))
svmfit <- svm(y ~., data = dat, kernel = "linear",
              cost = 1e5)
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 1e+05)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1e+05
##
## Number of Support Vectors:  3
##
##  ( 1 2 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

```r
plot(svmfit, dat)
```

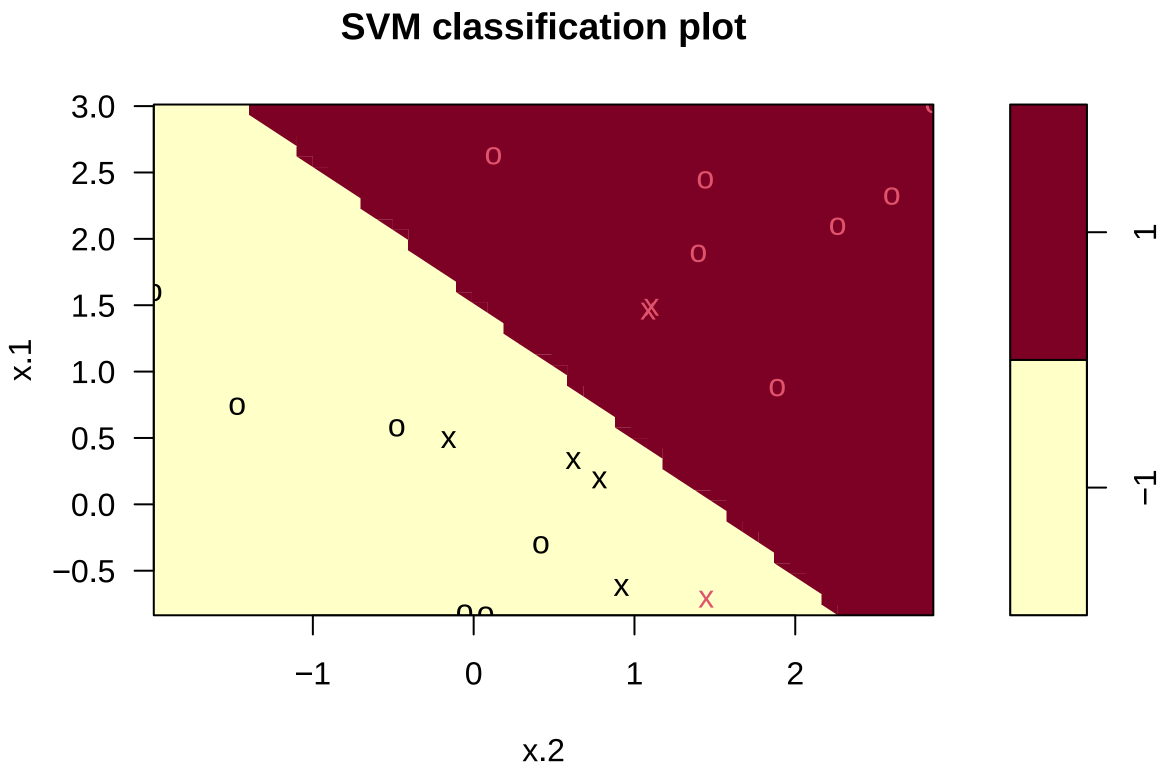


No training errors were made and only three support vectors were used. However, we can see from the figure that the margin is narrow (because the observations that are not support vectors, indicated as circles, are close to the decision boundary). It seems likely that this model will perform poorly on test data. We now try with a smaller value of `cost`:

```r
svmfit <- svm(y~., data = dat, kernel = "linear",
              cost = 1)
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 1)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

```
plot(svmfit, dat)
```



Using `cost = 1`, we misclassify a training observation, but we also obtain a much wider margin and make use of seven support vectors. It seems likely that this model will perform better on test data than the model with `cost = 1e5`.

9.2 Support Vector Machine

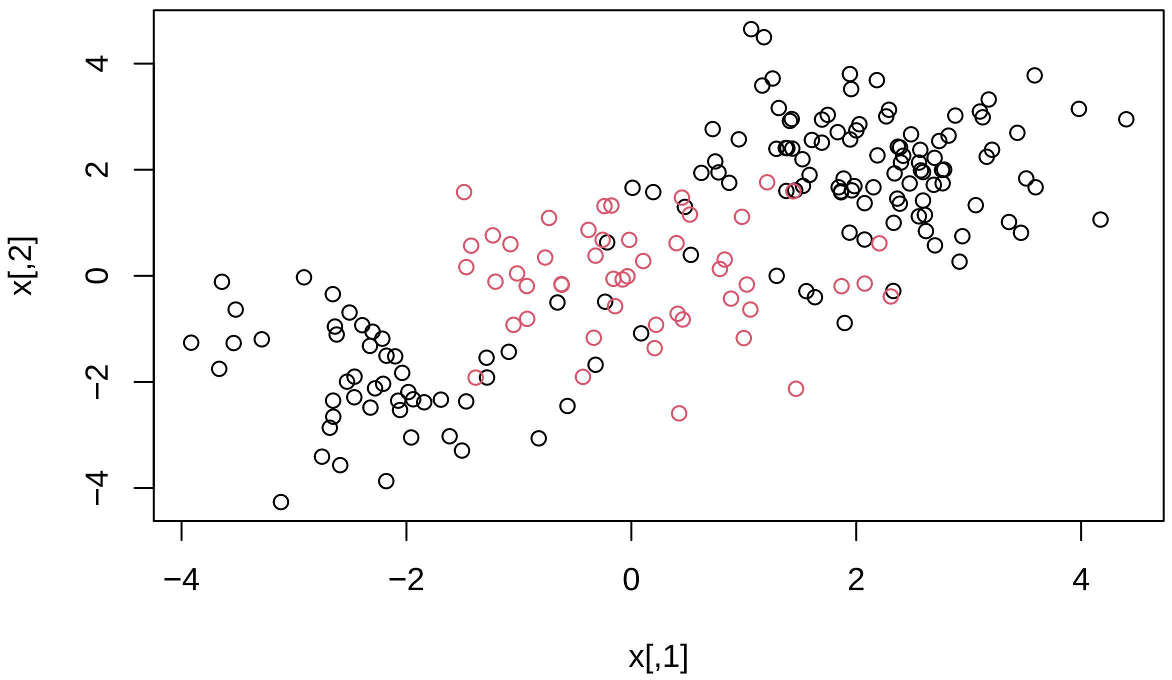
In order to fit an SVM using a non-linear kernel, we again use the `svm()` function. However, now we use a different value of the parameter `kernel`. To fit an SVM with a polynomial kernel, we use `kernel = "polynomial"` and to fit an SVM with a radial kernel, we use `kernel = "radial"`. In the former case we also use the `degree` argument to specify a degree for the polynomial kernel (this is the d parameter in equation 9.22), and in the latter case, we use `gamma` to specify a value of γ for the radial basis kernel.

We first generate some data with a non-linear class boundary as follows:

```
set.seed(1)
x <- matrix(rnorm(200*2), ncol = 2)
x[1:100,] <- x[1:100,] + 2
x[101:150, ] <- x[101:150,] - 2
y <- c(rep(1,150), rep(2,50))
dat <- data.frame(x = x, y = as.factor(y))
```

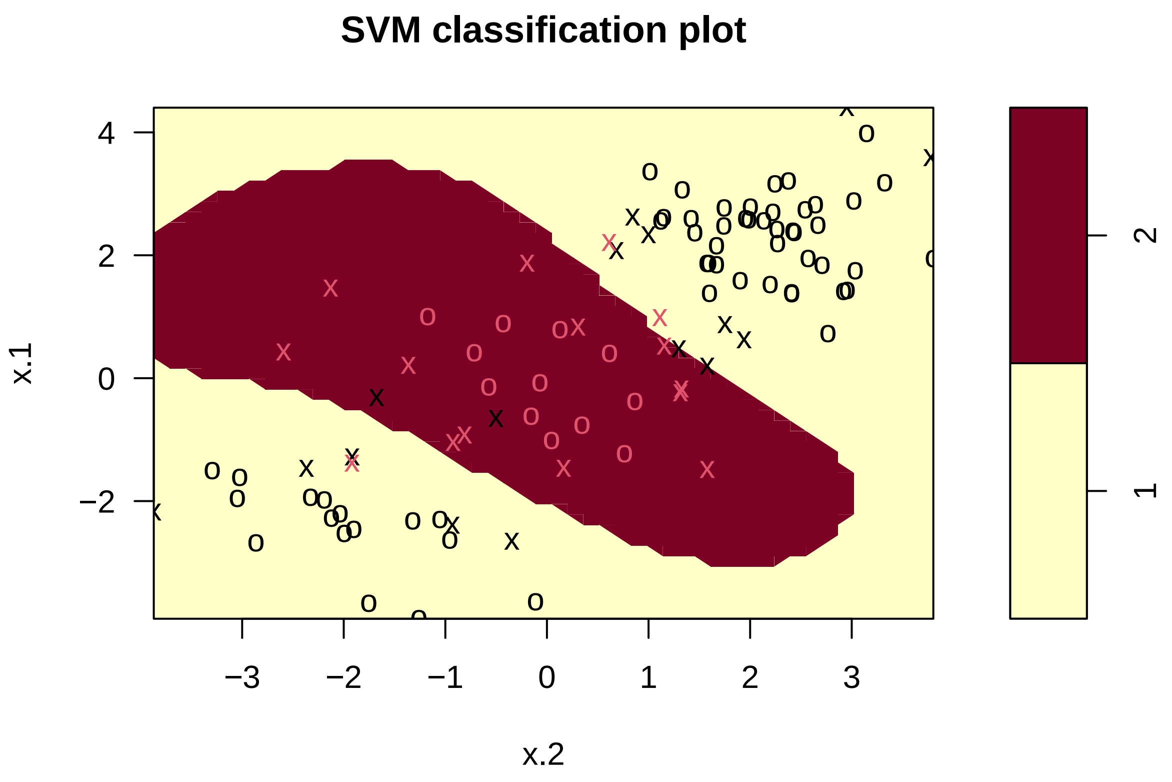
We can plot the data to confirm that the class boundary is indeed non-linear:

```
plot(x, col = y)
```



The data is randomly split into training and testing groups. We then fit the training data using the `svm()` function with a radial kernel and $\gamma = 1$:

```r
train <- sample(200,100)
svmfit <- svm(y ~ ., data = dat[train,], kernel = "radial",
              gamma = 1, cost = 1)
plot(svmfit, dat[train,])
```



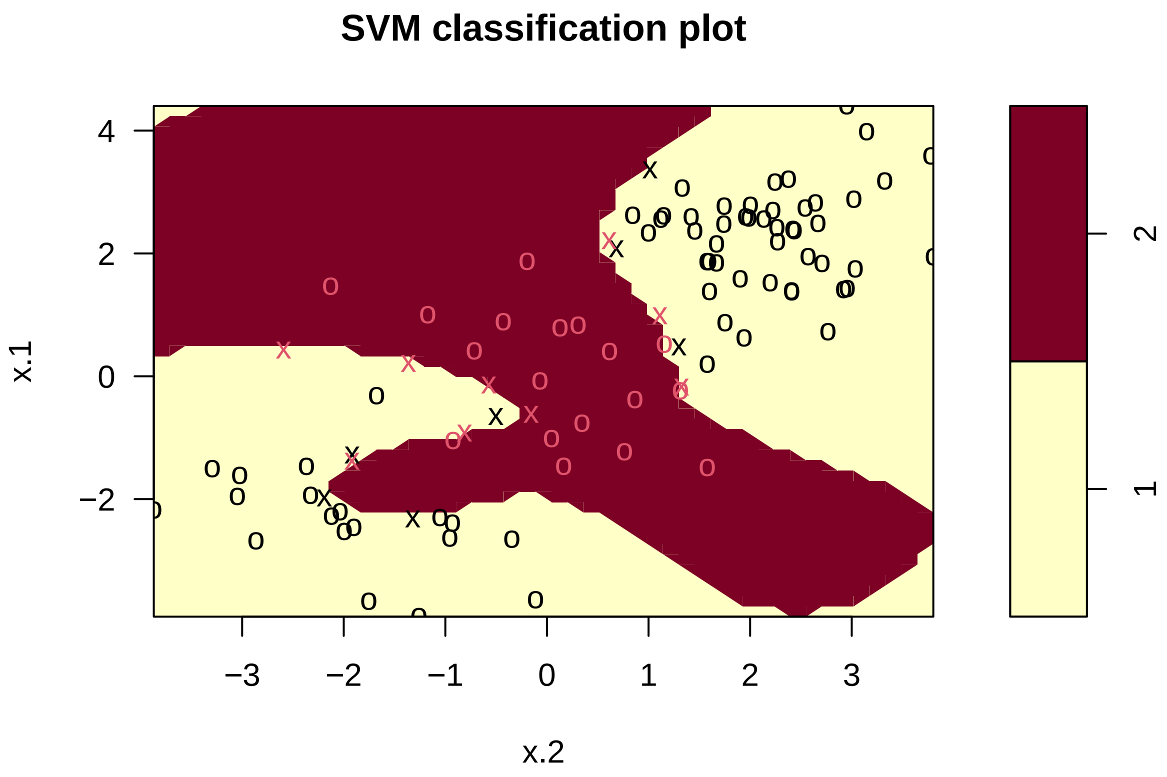
The plot shows that the resulting SVM has a decidedly non-linear boundary. The `summary()` function can be used to obtain some information about the SVM fit:

```
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat[train, ], kernel = "radial", gamma = 1,
##     cost = 1)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  31
##
##  ( 16 15 )
##
##
## Number of Classes:  2
##
## Levels:
##  1 2
```

We can see from the figure that there are a fair number of training errors in the SVM fit. If we increase the value of `cost`, we can reduce the number of training errors. However, this comes at the prince of a more irregular decision boundary that seems to be at risk of overfitting the data.

```
svmfit <- svm(y ~ ., data = dat[train,], kernel = "radial",
              gamma = 1, cost = 1e5)
plot(svmfit, dat[train,])
```



We can perform cross-validation using `tune()` to select the best choice of γ and `cost` for an SVM with a radial kernel:

```r
set.seed(1)
tune.out <- tune(svm, y ~ ., data = dat[train, ],
                 kernel = "radial",
                 ranges = list(cost = c(0.1, 1, 10, 100, 1000),
                               gamma = c(0.5, 1, 2, 3, 4)))
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##     1   0.5
##
## - best performance: 0.07
##
## - Detailed performance results:
##     cost gamma error dispersion
## 1  1e-01   0.5  0.26 0.15776213
## 2  1e+00   0.5  0.07 0.08232726
## 3  1e+01   0.5  0.07 0.08232726
## 4  1e+02   0.5  0.14 0.15055453
## 5  1e+03   0.5  0.11 0.07378648
## 6  1e-01   1.0  0.22 0.16193277
## 7   1e+00   1.0  0.07 0.08232726
## 8   1e+01   1.0  0.09 0.07378648
## 9   1e+02   1.0  0.12 0.12292726
## 10  1e+03   1.0  0.11 0.11005049
## 11  1e-01   2.0  0.27 0.15670212
## 12  1e+00   2.0  0.07 0.08232726
## 13  1e+01   2.0  0.11 0.07378648
## 14  1e+02   2.0  0.12 0.13165612
## 15  1e+03   2.0  0.16 0.13498971
## 16  1e-01   3.0  0.27 0.15670212
## 17  1e+00   3.0  0.07 0.08232726
## 18  1e+01   3.0  0.08 0.07888106
## 19  1e+02   3.0  0.13 0.14181365
## 20  1e+03   3.0  0.15 0.13540064
## 21  1e-01   4.0  0.27 0.15670212
## 22  1e+00   4.0  0.07 0.08232726
## 23  1e+01   4.0  0.09 0.07378648
## 24  1e+02   4.0  0.13 0.14181365
## 25  1e+03   4.0  0.15 0.13540064
```

The best choice of parameters involves `cost = 1` and `gamma = 2`. We can view the test set predictors for this model by applying the `predict()` function to the data. Notice that to do this we subset the dataframe `dat` using `-train` as an index set.

```
tab <- table(true = dat[-train, "y"],
      pred = predict(tune.out$best.model, newdata = dat[-train,]))
tab
```

```
##     pred
## true  1  2
##    1 67 10
##    2  2 21
```

```
sum(diag(tab))/sum(tab) ## accuracy
```

```
## [1] 0.88
```

9.3 ROC Curves

The `pROCR` package can be used to produce ROC curves such as those in Figure 9.10 and 9.11. We first write a short function to plot an ROC curve given a vector containing a numerical score for each observation, `pred` and a vector containing the class label for each observation, `truth`.

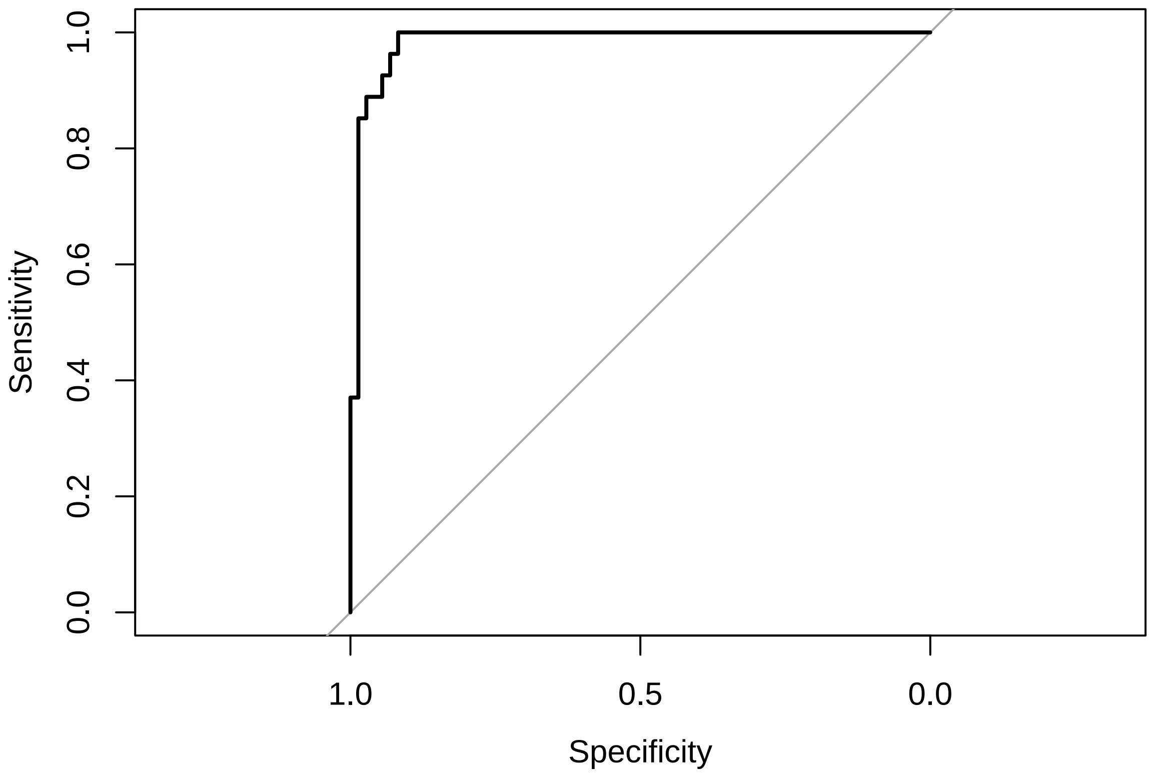
SVMs and support vector classifiers output class labels for each observation. However, it is also possible to obtain *fitted values* for each observation, which are the numerical scores used to obtain the class labels. For instance, in the case of a support vector classifier, the fitted value for an observation $X = (X_1, X_2, \dots, X_p)^T$ takes the form $\hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2 + \dots + \hat{\beta}_p$. For an SVM with a non-linear kernel, the equation that yields the fitted value here is given as:

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x, x_i).$$

In essence, the sign of the fitted value determines on which side of the decision boundary the observation lies. Therefore, the relationship between the fitted value and the class prediction for a given observation is simple: if the fitted value exceeds zero then the observation is assigned to one class, and if it is less than zero then it is assigned to the other. In order to obtain the fitted values for a given SVM model fit, we can use `decision.values = TRUE` when fitting `svm()`. Then, the `predict()` function will output the fitted values.

```
svmfit.opt <- svm(y ~ ., data = dat[train,], kernel = "radial",
                  gamma = 2, cost = 1, decision.values = TRUE)
fitted <- attributes(predict(svmfit.opt, dat[train,],
                             decision.values=TRUE))$decision.values
```

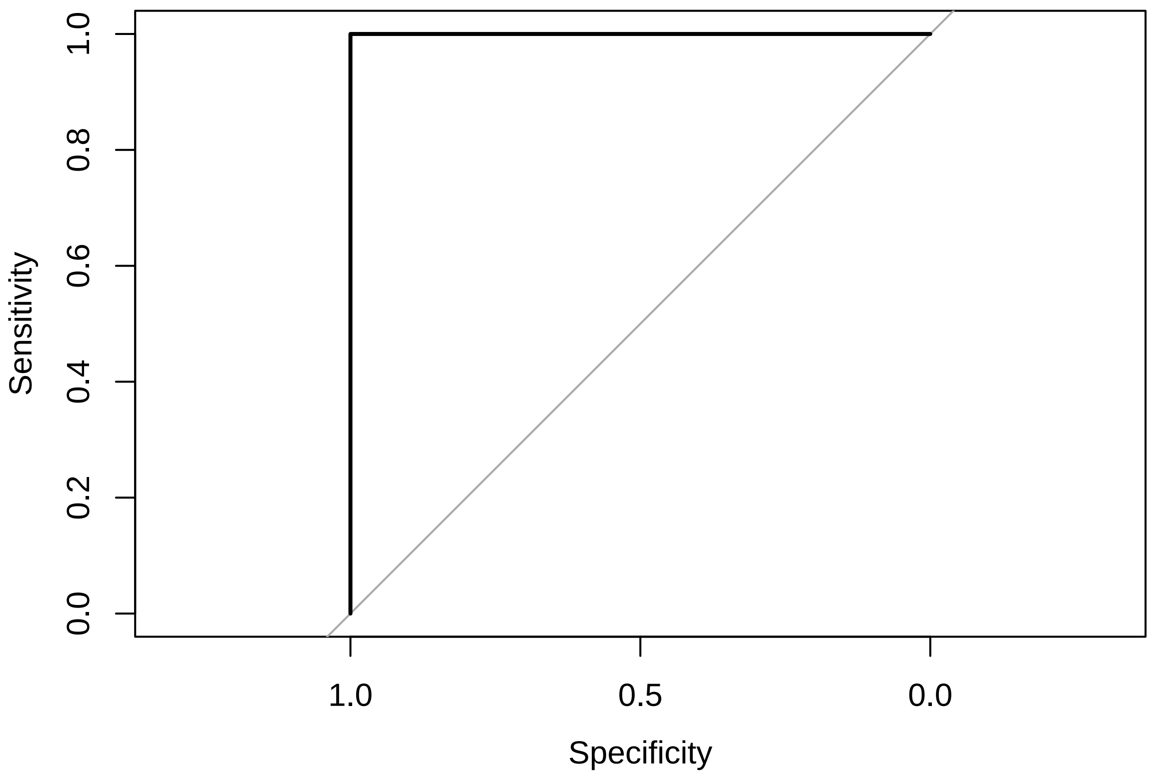
```
library(pROC)
ROC_svm <- roc(dat[train, "y"], fitted)
plot(ROC_svm)
```



```
svmfit.flex <- svm(y ~ ., data = dat[train, ], kernel = "radial",
                   gamma = 50, cost = 1, decision.values = TRUE)
fitted <- attributes(predict(svmfit.flex, dat[train, ],
                             decision.values = T))$decision.values
```

Now we can produce the ROC plot:

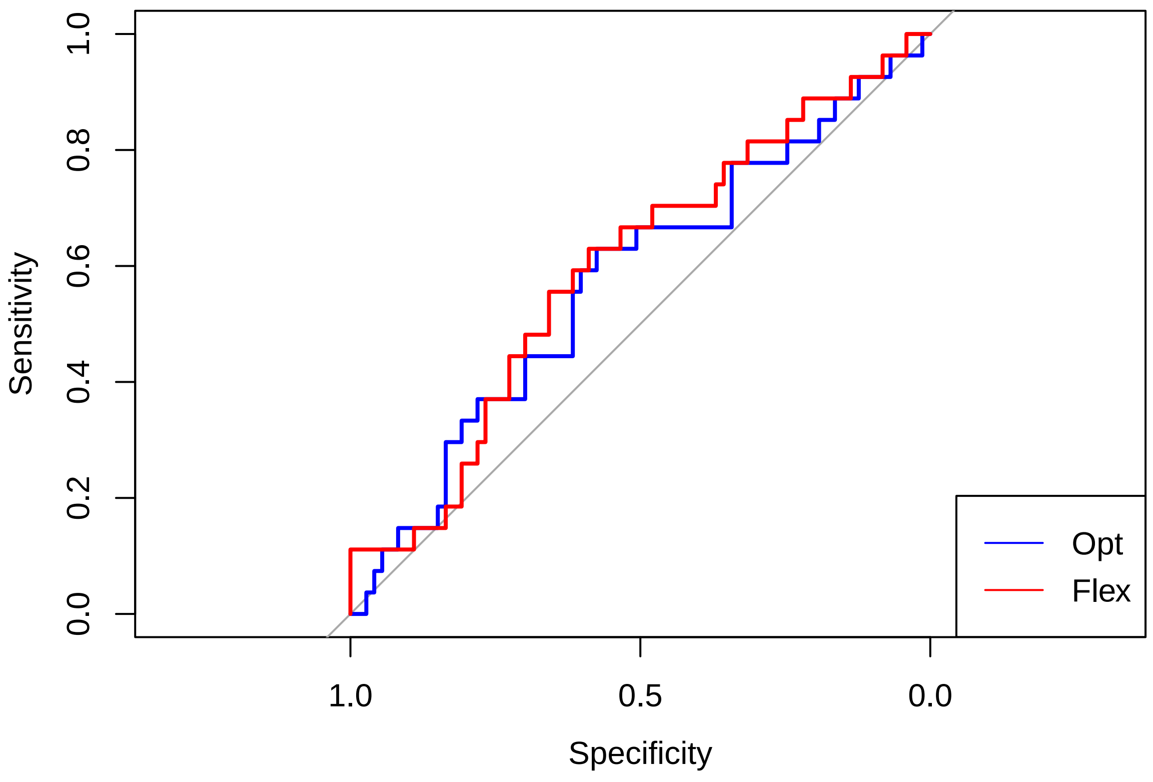
```
ROC_svm <- roc(dat[train, "y"], fitted)
plot(ROC_svm)
```



However, these ROC curves are all on the training data. We are really more interested in the level of prediction accuracy on the test data, the model with $\gamma = 2$ appears to provide the most accurate results.

```
fitted <- attributes(predict(svmfit.opt, dat[-train, ],
                              decision.values=TRUE))$decision.values
ROC_svm1 <- roc(dat[train, "y"], fitted)

fitted <- attributes(predict(svmfit.flex, dat[-train, ],
                              decision.values=TRUE))$decision.values
ROC_svm2 <- roc(dat[train, "y"], fitted)
plot(ROC_svm1, col = "blue")
lines(ROC_svm2, col = "red")
legend("bottomright", legend = c("Opt", "Flex"),
       col = c("blue", "red"), lty = 1)
```



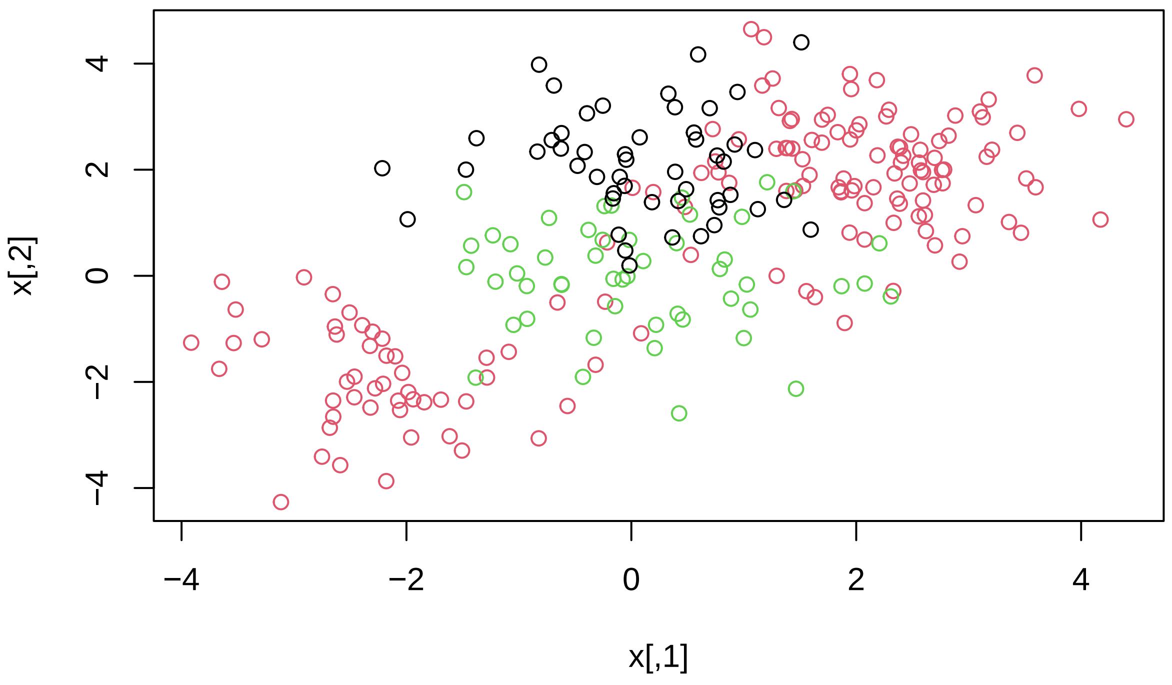
9.4 SVM with Multiple Classes

If the response is a factor containing more than two levels, then the `svm()` function will perform multi-class classification using the one-versus-one approach. We explore that setting here by generating a third class of observations.

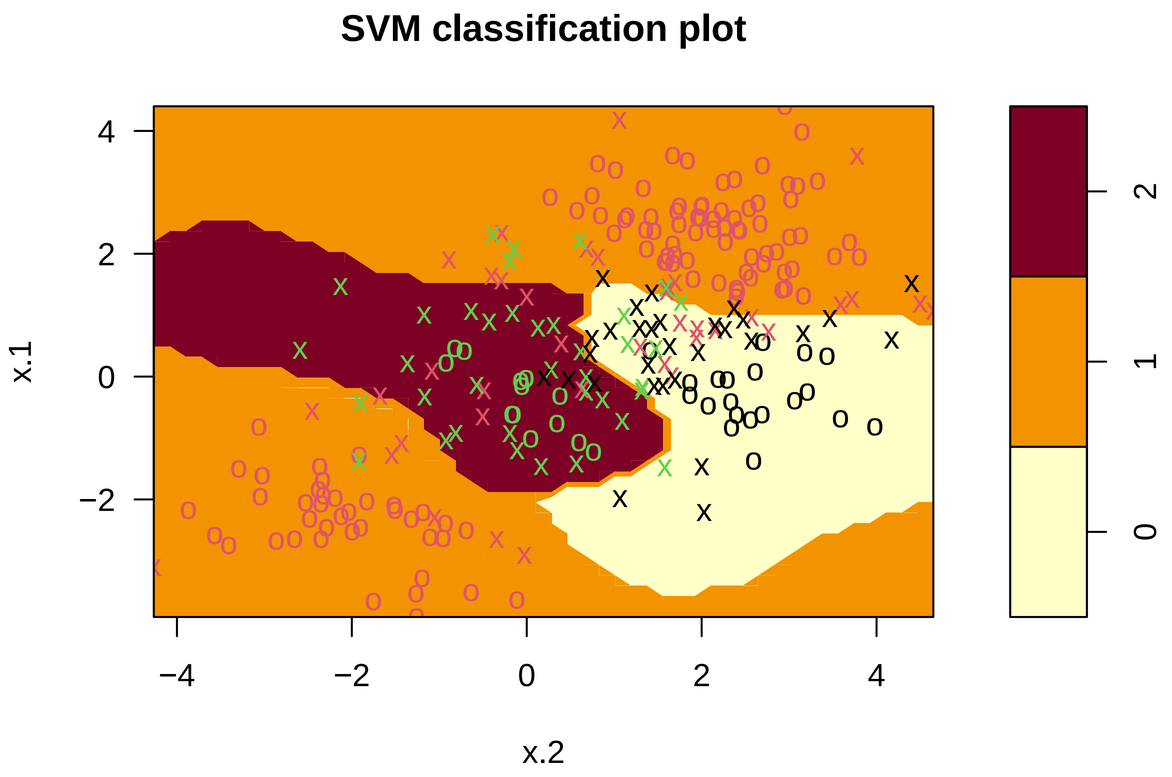
```
set.seed(1)
x <- rbind(x, matrix(rnorm(50*2), ncol = 2))
y <- c(y, rep(0, 50))
x[y==0,2] <- x[y==0,2] + 2
dat <- data.frame(x = x, y = as.factor(y))
```

We can fit an SVM to this data:

```
svmfit <- svm(y ~ ., data = dat, kernel = "radial",
              cost = 10, gamma = 1)
par(mfrow = c(1,1))
plot(x, col = (y+1))
```



```r
plot(svmfit, dat)
```



The `e1071` library can also be used to perform support vector regression, if the response vector that is passed to `svm()` is numerical rather than a factor.

Problems

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.

(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.

```
library(ISLR)
data("Auto")
attach(Auto)
```

```
median <- median(mpg)
Auto$above_median <- ifelse(mpg > median, 1, 0)
Auto$above_median <- as.factor(Auto$above_median)
head(Auto)
```

```
##   mpg cylinders displacement horsepower weight acceleration year origin
## 1  18         8          307        130   3504         12.0   70      1
## 2  15         8          350        165   3693         11.5   70      1
## 3  18         8          318        150   3436         11.0   70      1
## 4  16         8          304        150   3433         12.0   70      1
## 5  17         8          302        140   3449         10.5   70      1
## 6  15         8          429        198   4341         10.0   70      1
##                        name above_median
## 1 chevrolet chevelle malibu            0
## 2         buick skylark 320            0
## 3        plymouth satellite            0
## 4             amc rebel sst            0
## 5               ford torino            0
## 6          ford galaxie 500            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)
tune.out <- tune(svm, above_median~.,
                 data = Auto,
                 kernel = "linear",
                 ranges = list(cost = c(0.001, 0.01, 0.1,
                                        1, 5, 10, 100)))
```

```
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##      1
##
## - best performance: 0.01025641
##
## - Detailed performance results:
##    cost      error dispersion
## 1 1e-03 0.09442308 0.04519425
## 2 1e-02 0.07653846 0.03617137
## 3 1e-01 0.04596154 0.03378238
## 4 1e+00 0.01025641 0.01792836
## 5 5e+00 0.02051282 0.02648194
## 6 1e+01 0.02051282 0.02648194
## 7 1e+02 0.03076923 0.03151981
```

```
bestmod.linear <- tune.out$best.model
summary(bestmod.linear)
```

```
##
## Call:
## best.tune(METHOD = svm, train.x = above_median ~ ., data = Auto,
##     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:  1
##
## Number of Support Vectors:  56
##
##  ( 26 30 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

After performing the CV, we find that that `cost = 1` results in the lowest cross-validation error rate, which is 0.01025641.

(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.

```
# radial
set.seed(1)
tune.out <- tune(svm, above_median~.,
                 data = Auto,
                 kernel = "radial",
                 ranges = list(cost = c(0.1, 1, 10, 100, 1000),
                               gamma = c(0.5, 1, 2, 3, 4)))
summary(tune.out)
```

```
##
```

```
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##    10   0.5
##
## - best performance: 0.04865385
##
## - Detailed performance results:
##     cost gamma      error dispersion
## 1  1e-01   0.5 0.07910256 0.04234351
## 2  1e+00   0.5 0.05115385 0.02716416
## 3  1e+01   0.5 0.04865385 0.03075209
## 4  1e+02   0.5 0.05121795 0.03424201
## 5  1e+03   0.5 0.05121795 0.03424201
## 6  1e-01   1.0 0.55115385 0.04366593
## 7  1e+00   1.0 0.06384615 0.04375618
## 8  1e+01   1.0 0.05884615 0.04020934
## 9  1e+02   1.0 0.05884615 0.04020934
## 10 1e+03   1.0 0.05884615 0.04020934
## 11 1e-01   2.0 0.55115385 0.04366593
## 12 1e+00   2.0 0.14019231 0.07984711
## 13 1e+01   2.0 0.13512821 0.08055403
## 14 1e+02   2.0 0.13512821 0.08055403
## 15 1e+03   2.0 0.13512821 0.08055403
## 16 1e-01   3.0 0.55115385 0.04366593
## 17 1e+00   3.0 0.41326923 0.14331350
## 18 1e+01   3.0 0.38025641 0.14908523
## 19 1e+02   3.0 0.38025641 0.14908523
## 20 1e+03   3.0 0.38025641 0.14908523
## 21 1e-01   4.0 0.55115385 0.04366593
## 22 1e+00   4.0 0.47705128 0.05783758
## 23 1e+01   4.0 0.47705128 0.06151011
## 24 1e+02   4.0 0.47705128 0.06151011
## 25 1e+03   4.0 0.47705128 0.06151011
```

```r
bestmod.radial <- tune.out$best.model
summary(bestmod.radial)
```

```
##
## Call:
## best.tune(METHOD = svm, train.x = above_median ~ ., data = Auto,
##     ranges = list(cost = c(0.1, 1, 10, 100, 1000), gamma = c(0.5,
##         1, 2, 3, 4)), kernel = "radial")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  10
##
## Number of Support Vectors:  259
```

```
##
##  ( 127 132 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

For the Radial Kernel, we find that `cost = 10` and `gamma = 0.5` result in the lowest cross-validation error rate, which is 0.04865385.

```
# polynomial
set.seed(1)
tune.out <- tune(svm, above_median~.,
                 data = Auto,
                 kernel = "polynomial",
                 ranges = list(cost = c(0.1, 1, 10, 100, 1000),
                               degree = c(1, 2, 3),
                               gamma = c(0.5, 1, 2, 3, 4)))
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost degree gamma
##     1      1     1
##
## - best performance: 0.01025641
##
## - Detailed performance results:
##     cost degree gamma      error dispersion
## 1  1e-01      1   0.5 0.05365385 0.02830539
## 2  1e+00      1   0.5 0.01282051 0.01813094
## 3  1e+01      1   0.5 0.02051282 0.02648194
## 4  1e+02      1   0.5 0.03076923 0.03151981
## 5  1e+03      1   0.5 0.03076923 0.03151981
## 6  1e-01      2   0.5 0.17358974 0.03983401
## 7  1e+00      2   0.5 0.14532051 0.03783687
## 8  1e+01      2   0.5 0.17096154 0.02736052
## 9  1e+02      2   0.5 0.19891026 0.03316037
## 10 1e+03      2   0.5 0.19891026 0.03316037
## 11 1e-01      3   0.5 0.04608974 0.04327907
## 12 1e+00      3   0.5 0.04352564 0.03835271
## 13 1e+01      3   0.5 0.04358974 0.03636247
## 14 1e+02      3   0.5 0.04358974 0.03636247
## 15 1e+03      3   0.5 0.04358974 0.03636247
## 16 1e-01      1   1.0 0.04596154 0.03378238
## 17 1e+00      1   1.0 0.01025641 0.01792836
## 18 1e+01      1   1.0 0.02051282 0.02648194
## 19 1e+02      1   1.0 0.03076923 0.03151981
## 20 1e+03      1   1.0 0.03076923 0.03151981
## 21 1e-01      2   1.0 0.14038462 0.04576941
## 22 1e+00      2   1.0 0.16070513 0.02091410
## 23 1e+01      2   1.0 0.19891026 0.03316037
## 24 1e+02      2   1.0 0.19891026 0.03316037
## 25 1e+03      2   1.0 0.19891026 0.03316037
## 26 1e-01      3   1.0 0.04096154 0.04047941
## 27 1e+00      3   1.0 0.04358974 0.03636247
## 28 1e+01      3   1.0 0.04358974 0.03636247
## 29 1e+02      3   1.0 0.04358974 0.03636247
## 30 1e+03      3   1.0 0.04358974 0.03636247
## 31 1e-01      1   2.0 0.02814103 0.01893035
## 32 1e+00      1   2.0 0.01282051 0.02179068
## 33 1e+01      1   2.0 0.03076923 0.03151981
## 34 1e+02      1   2.0 0.03076923 0.03151981
## 35 1e+03      1   2.0 0.03076923 0.03151981
## 36 1e-01      2   2.0 0.15307692 0.02099852
## 37 1e+00      2   2.0 0.17858974 0.02709415
## 38 1e+01      2   2.0 0.19891026 0.03316037
## 39 1e+02      2   2.0 0.19891026 0.03316037
## 40 1e+03      2   2.0 0.19891026 0.03316037
## 41 1e-01      3   2.0 0.04615385 0.03585671
## 42 1e+00      3   2.0 0.04358974 0.03636247
## 43 1e+01      3   2.0 0.04358974 0.03636247
## 44 1e+02      3   2.0 0.04358974 0.03636247
## 45 1e+03      3   2.0 0.04358974 0.03636247
## 46 1e-01      1   3.0 0.02557692 0.02093679
## 47 1e+00      1   3.0 0.01282051 0.02179068
## 48 1e+01      1   3.0 0.03076923 0.03151981
## 49 1e+02      1   3.0 0.03076923 0.03151981
## 50 1e+03      1   3.0 0.03076923 0.03151981
## 51 1e-01      2   3.0 0.15814103 0.01581211
## 52 1e+00      2   3.0 0.19891026 0.03316037
## 53 1e+01      2   3.0 0.19891026 0.03316037
## 54 1e+02      2   3.0 0.19891026 0.03316037
## 55 1e+03      2   3.0 0.19891026 0.03316037
## 56 1e-01      3   3.0 0.04358974 0.03636247
## 57 1e+00      3   3.0 0.04358974 0.03636247
## 58 1e+01      3   3.0 0.04358974 0.03636247
## 59 1e+02      3   3.0 0.04358974 0.03636247
## 60 1e+03      3   3.0 0.04358974 0.03636247
## 61 1e-01      1   4.0 0.01538462 0.01792836
## 62 1e+00      1   4.0 0.01538462 0.02162241
## 63 1e+01      1   4.0 0.03076923 0.03151981
## 64 1e+02      1   4.0 0.03076923 0.03151981
## 65 1e+03      1   4.0 0.03076923 0.03151981
## 66 1e-01      2   4.0 0.16333333 0.02503157
## 67 1e+00      2   4.0 0.19891026 0.03316037
## 68 1e+01      2   4.0 0.19891026 0.03316037
## 69 1e+02      2   4.0 0.19891026 0.03316037
## 70 1e+03      2   4.0 0.19891026 0.03316037
## 71 1e-01      3   4.0 0.04358974 0.03636247
## 72 1e+00      3   4.0 0.04358974 0.03636247
## 73 1e+01      3   4.0 0.04358974 0.03636247
## 74 1e+02      3   4.0 0.04358974 0.03636247
## 75 1e+03      3   4.0 0.04358974 0.03636247
```

```
bestmod.poly <- tune.out$best.model
summary(bestmod.poly)
```

```
##
## Call:
## best.tune(METHOD = svm, train.x = above_median ~ ., data = Auto,
##     ranges = list(cost = c(0.1, 1, 10, 100, 1000), degree = c(1,
##         2, 3), gamma = c(0.5, 1, 2, 3, 4)), kernel = "polynomial")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  polynomial
##        cost:  1
##      degree:  1
##      coef.0:  0
##
## Number of Support Vectors:  56
##
##  ( 26 30 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

For the Polynomial Kernel, we find that `cost = 1`, `degree = 1`, and `gamma = 1`, result in the lowest cross-validation error rate, which is 0.01025641.

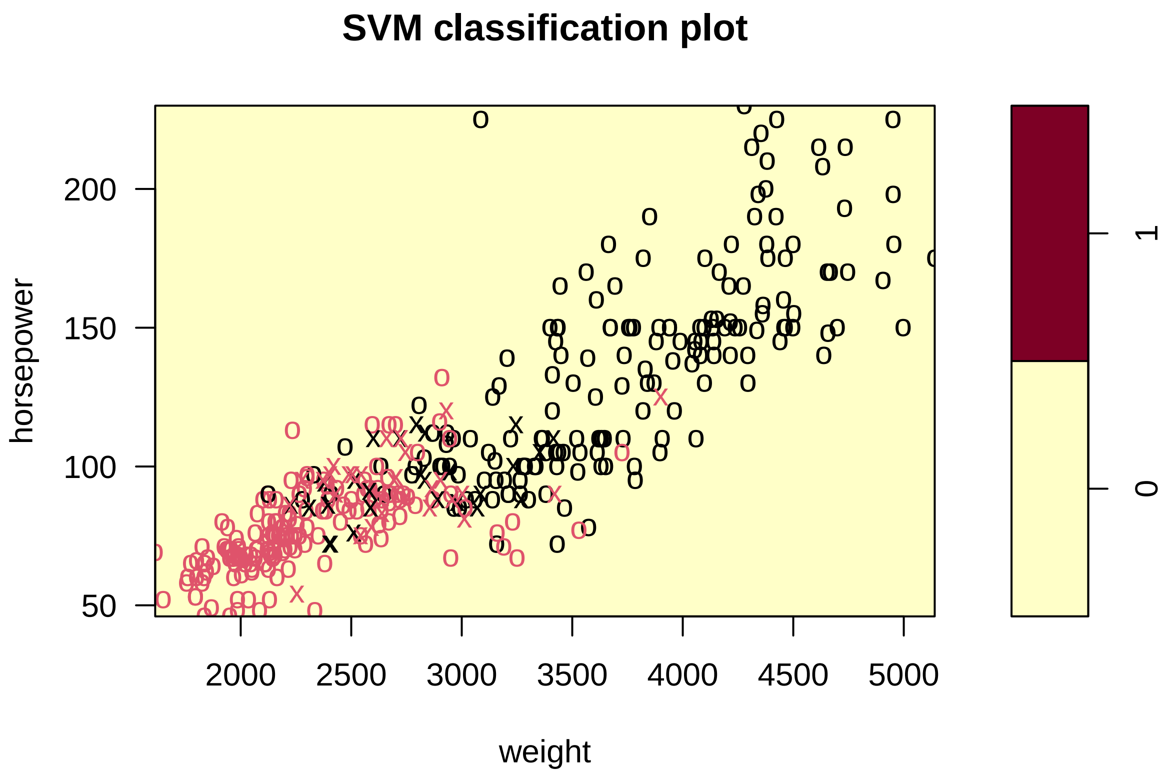
(d) Make some plots to back up your assertions in (b) and (c).

```
# linear
svmfit.linear <- svm(above_median~.,
                     data = Auto,
                     kernel = "linear",
                     cost = 1)
summary(svmfit.linear)
```

```
##
## Call:
## svm(formula = above_median ~ ., data = Auto, kernel = "linear", cost = 1)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1
##
## Number of Support Vectors:  56
```

```
##
##  ( 26 30 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

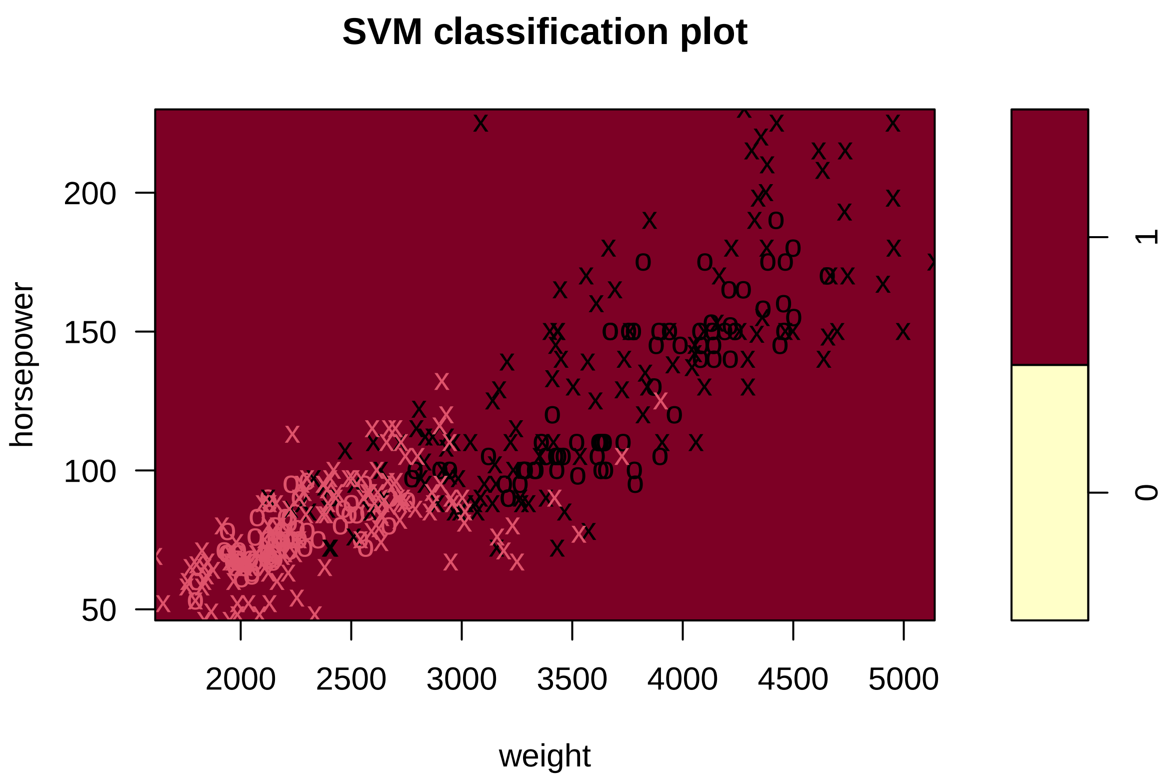
```
plot(svmfit.linear, Auto,  horsepower~weight)
```



```
# radial
svmfit.radial <- svm(above_median~.,
                     data = Auto,
                     kernel = "radial",
                     cost = 10,
                     gamma = 0.5)
summary(svmfit.radial)
```

```
##
## Call:
## svm(formula = above_median ~ ., data = Auto, kernel = "radial", cost = 10,
##     gamma = 0.5)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  10
##
## Number of Support Vectors:  259
##
##  ( 127 132 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

```
plot(svmfit.radial, Auto,  horsepower~weight)
```

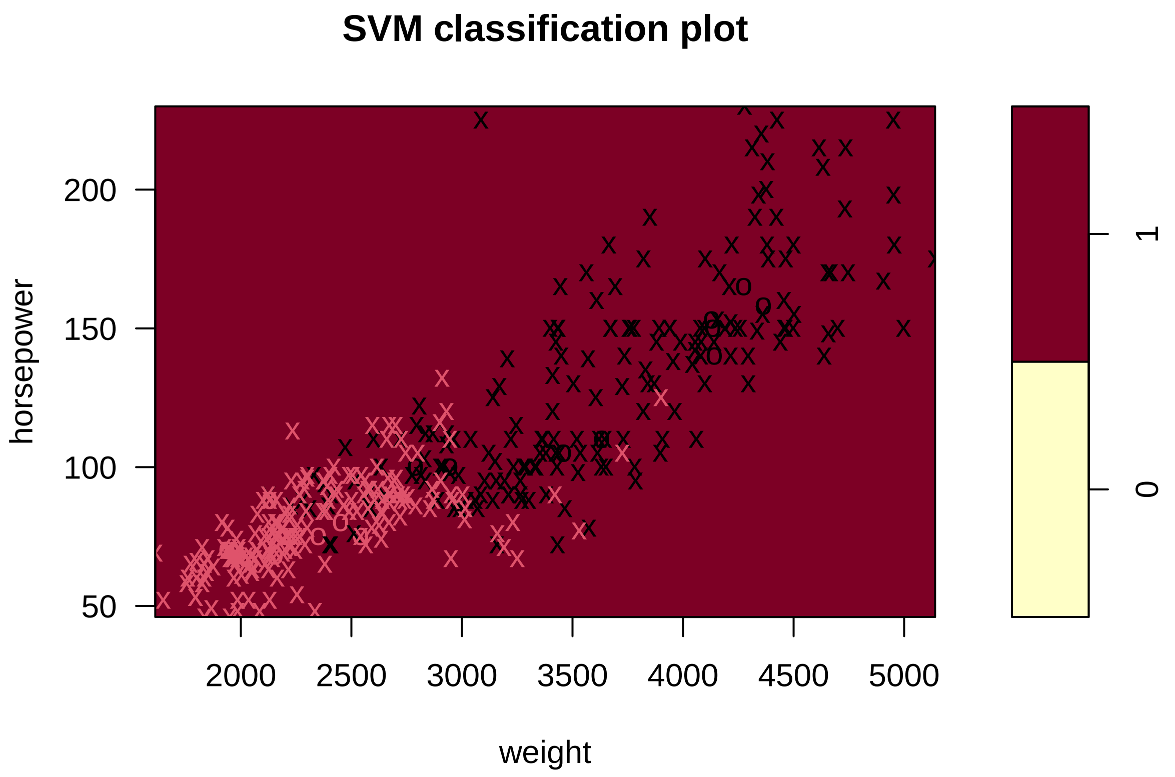


```
# polynomial
svmfit.poly <- svm(above_median~.,
                     data = Auto,
                     kernel = "radial",
                     cost = 1,
                     gamma = 1,
                     d = 1)
summary(svmfit.poly)
```

```
##
## Call:
## svm(formula = above_median ~ ., data = Auto, kernel = "radial", cost = 1,
##     gamma = 1, d = 1)
##
```

```
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  377
##
##  ( 187 190 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

```
plot(svmfit.poly, Auto,  horsepower~weight)
```



Hint: In the lab, we used the plot() function for svm objects only in cases with p = 2. When p > 2, you can use the plot() function to create plots displaying pairs of variables at a time. Essentially, instead of typing `> plot(svmfit , dat)` where svmfit contains your fitted model and dat is a data frame containing your data, you can type `> plot(svmfit , dat , x1 x4)` in order to plot just the first and fourth variables. However, you must replace x1 and x4 with the correct variable names. To find out more, type `?plot.svm`.