Lab 10: Support Vector Machine

PSTAT 131/231

Learning Objectives

- Use `svm()` to train a support vector machine with different costs
- Use `tune()` to perform cross-validation to select tuning parameters in SVMs
- Use different kernels: linear, radial and polynomial kernel

---

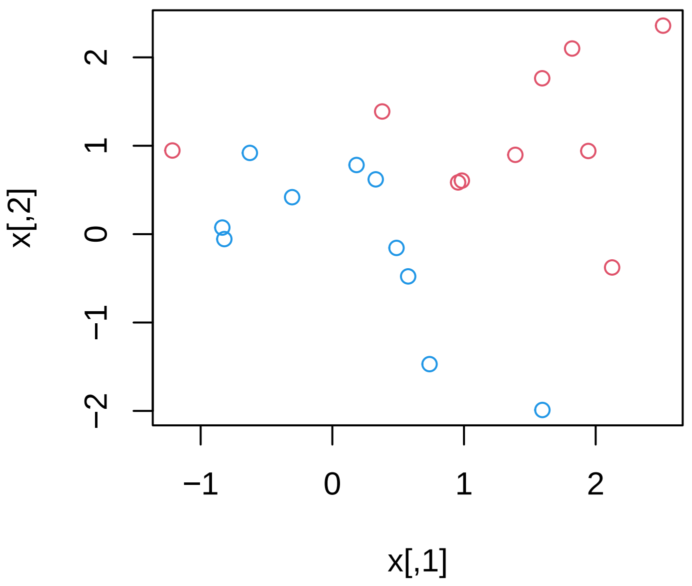
There are multiple R packages implementing SVM. We use the `e1071` library in R to demonstrate the support vector machines and the `svm()` function. Another option is the `LiblineaR` library, which is useful for very large linear problems.

support vector machines

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 machines when the argument `kernel="linear"` is used. As we mentioned in the lecture, this function uses a slightly different formulation from what we covered in class. The major difference is reflected in the `cost` argument in the *`svm()`* function. `cost` allows us to specify the cost of a violation to the margin, and it acts like the reciprocal of `C` in the formulation in class. 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 demonstrate the use of `svm()` with a given value of `cost` on a two-dimensional example so that we can plot the resulting decision boundary. We begin by simulating some observations, which belong to two classes, and checking whether the classes are linearly separable.

```r
# Generate the observations
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))
```



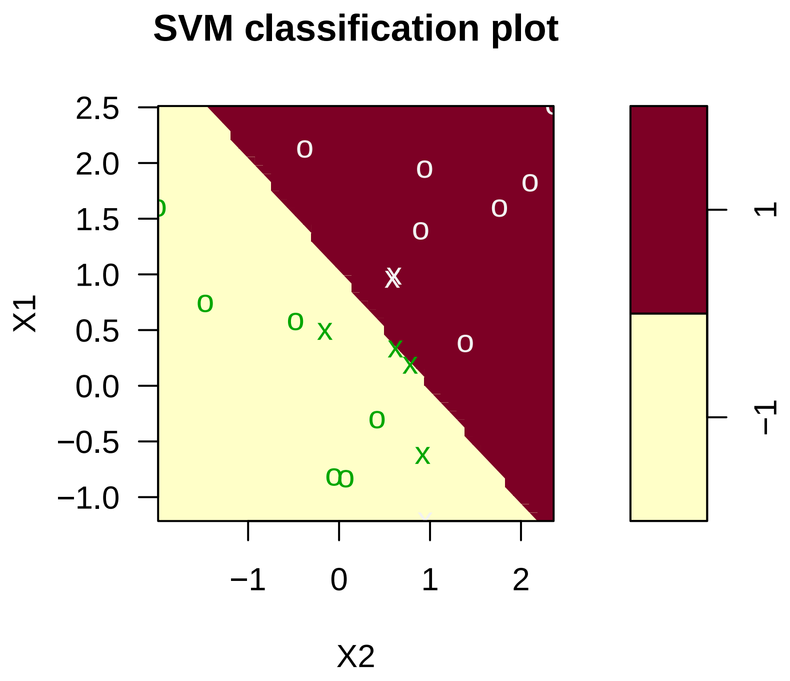
They are not linearly separable. Next, we fit the support vector machines. Note that in order for the `svm` function to perform classification (as opposed to SVM-based regression), we must encode the response as a **factor** variable. We now create a data frame with the response coded as a factor.

```
# Load package
# install.packages("e1071")
library(e1071)

# Generate a data frame for analysis with x and factorized y
dat=data.frame(x, y=as.factor(y))

# cost=10
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 machines obtained:

```
# plot.svm function in package e1071
plot(svmfit, dat,symbolPalette = terrain.colors(2))
```



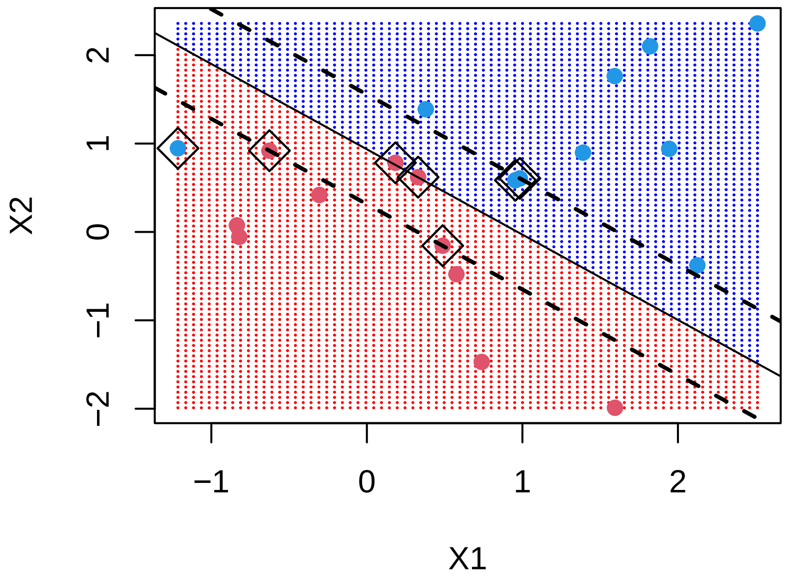
Note that the two arguments to the `plot.svm()` (i.e., `plot()` in `e1071`) function are the output of the call to `svm`, as well as the data used in the call to `svm`. The region of feature space that will be assigned to the −1 class is shown in light yellow, and the region that will be assigned to the +1 class is shown in red. The decision boundary between the two classes is linear (because 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 somewhat jagged in the plot. Check with the previous plot to see how many misclassifications are made.

Note: here the second feature is plotted on the x-axis and the first feature is plotted on the y-axis, in contrast to the behavior of the usual `plot` function in R.) The **support vectors** are plotted as crosses and the remaining observations are plotted as circles.

As mentioned in the the chapter, the plot function is somewhat crude, and plots X2 on the horizontal axis (unlike what R would do automatically for a matrix). Lets see how we might make our own plot.

The first thing we will do is make a grid of values for X1 and X2. We will write a function to do that, in case we want to reuse it. It uses the handy function expand.grid, and produces the coordinates of n*n points on a lattice covering the domain of x. Having made the lattice, we make a prediction at each point on the lattice. We then plot the lattice, color-coded according to the classification. Now we can see the decision boundary.

```
# Plot regions
make.grid = function(x, n = 75) {
    grange = apply(x, 2, range)
    x1 = seq(from = grange[1, 1], to = grange[2, 1], length = n)
    x2 = seq(from = grange[1, 2], to = grange[2, 2], length = n)
    expand.grid(X1 = x1, X2 = x2)
}

make.plot = function(svmfit,dat){
  xgrid = make.grid(dat[,-3])
  ygrid = predict(svmfit, xgrid)
  colors = ifelse(as.numeric(dat[,3])==1,2,4)
  plot(xgrid, col = c("red", "blue")[as.numeric(ygrid)], pch = 20, cex = 0.1)
  points(dat[,-3], col = colors, pch = 19)
  points(x[svmfit$index, ], pch = 5, cex = 2)
  # Plot decision boundary and margins
  beta = drop(t(svmfit$coefs) %*% x[svmfit$index, ])
  beta0 = svmfit$rho
  points(dat[svmfit$index,-3], pch = 5, cex = 2)
  abline(beta0/beta[2], -beta[1]/beta[2])
  abline((beta0 - 1)/beta[2], -beta[1]/beta[2], lty = 2,lwd=2)
  abline((beta0 + 1)/beta[2], -beta[1]/beta[2], lty = 2,lwd=2)

}

make.plot(svmfit,dat)
```



The support points (points on the margin, or on the wrong side of the margin) are indexed in the $index component of the fit.

```
svmfit$index
```

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

We can obtain some basic information about the support vector machines fit using the `summary` command:

```
summary(svmfit)
```

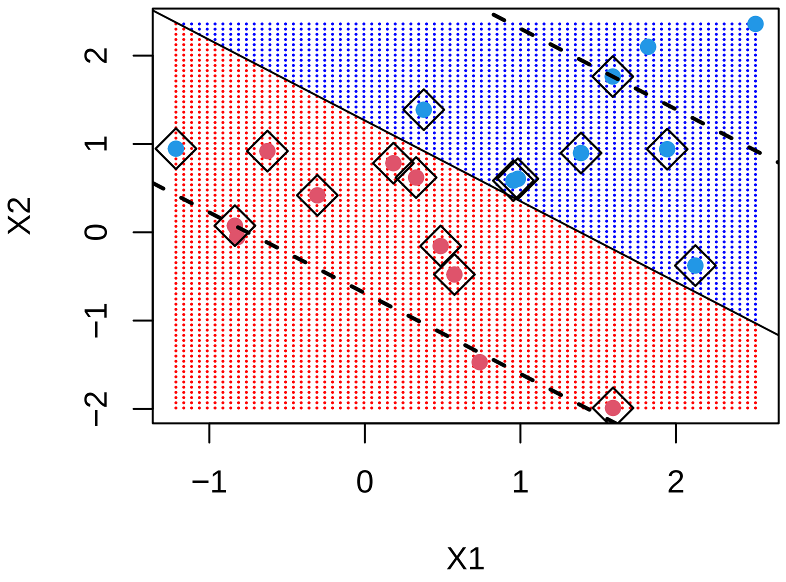
```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10, scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
```

```
##
## 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)
make.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 machines is fit, nor does it output the width of the margin.

Cross-validation in `e1071`

Recall from the lecture, we can treat the value of `C` (and correspondingly, `cost`) as a tuning parameter in a SVM formulation. In practice, we don't know the optimal value of `cost`, and thus need to perform cross-validation.

The `e1071` library includes a built-in function, `tune()`, to perform cross-validation. By default, `tune()` performs 10-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.

```
# Generate test set
set.seed(123)
```

```r
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))
testdat=data.frame(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 predictions.

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

```
##         truth
## predict -1 1
##      -1  8 3
##      1   1 8
```

What if we had instead used `cost=0.01`?

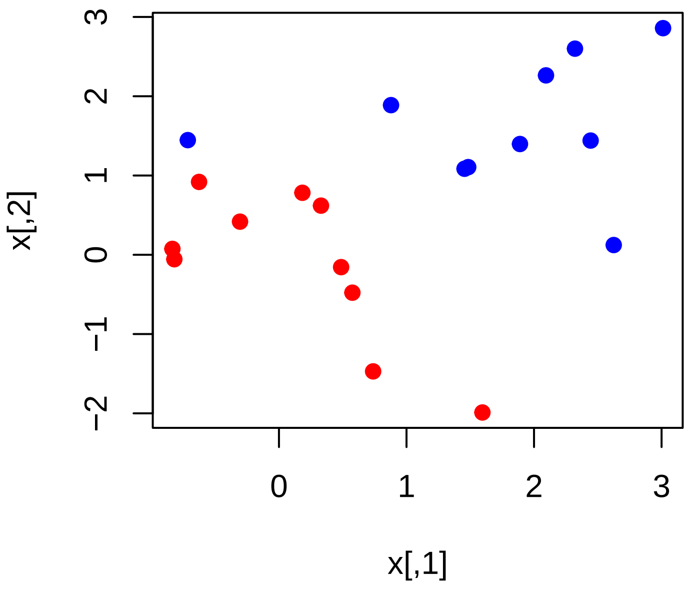
```r
svmfit=svm(y~., data=dat, kernel="linear", cost=.01,scale=FALSE)
ypred=predict(svmfit,testdat)
table(predict=ypred, truth=testdat$y)
```

```
##         truth
## predict -1 1
##      -1  8 7
##      1   1 4
```

In this case more misclassifications are made.

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:

```r
# Check the linear separation in the data
x[y==1,]=x[y==1,]+0.5

plot(x, col=ifelse(as.numeric(dat[,3])==1, "red", "blue"), pch=19)
```

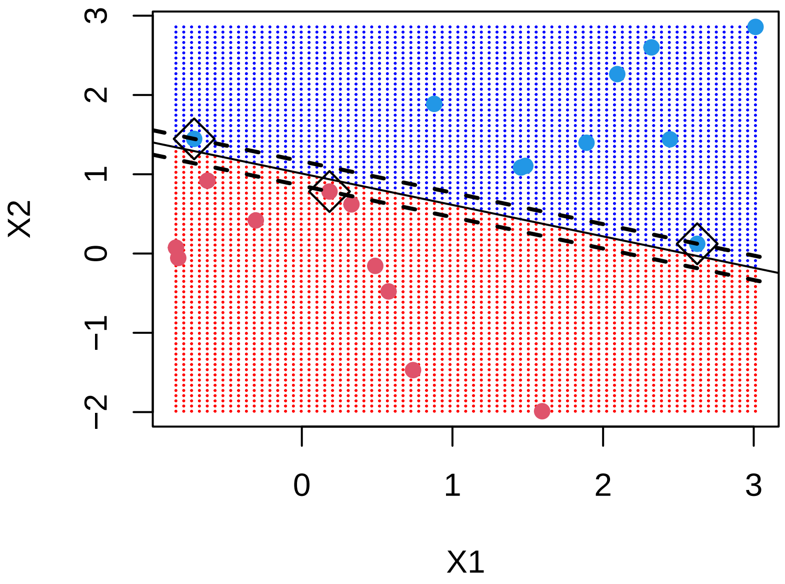


Now the observations are just barely linearly separable. We fit the support vector machines and plot the resulting hyperplane, using a very large value of cost so that no observations are misclassified.

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

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 1e+05,
##     scale = FALSE)
##
##
## 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)
make.plot(svmfit, dat)
```

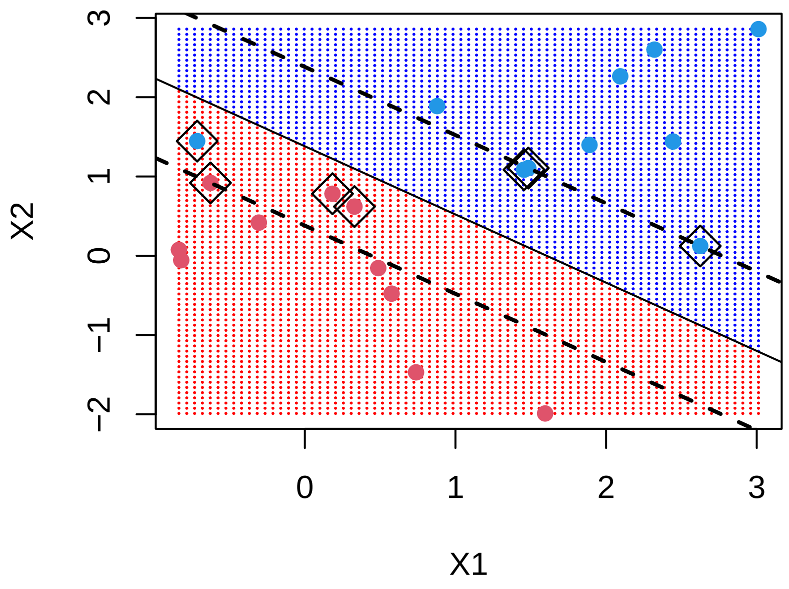


No training errors were made and only three support vectors were used. However, we can see from the figure that the margin is very narrow (because the observations that are not support vectors, indicated as without square boxes, are very close to the decision boundary). It seems likely that this model will perform poorly on test data. We now try a smaller value of cost:

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

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

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



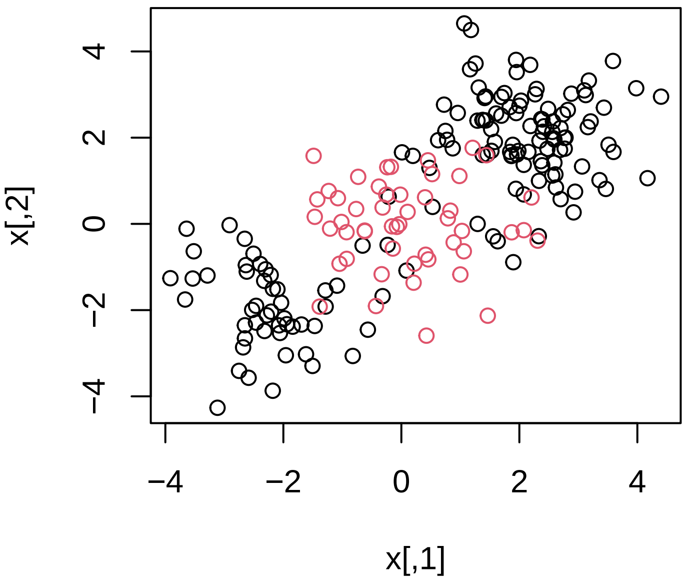
Using `cost=1`, we misclassify a training observation, but we also obtain a much wider margin and make use of seven support vectors. It seems likely that this model will perform better on test data than the model with `cost=1e5`.

Support Vector Machine with other kernels

In order to fit an SVM using a non-linear kernel, we once 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"`.

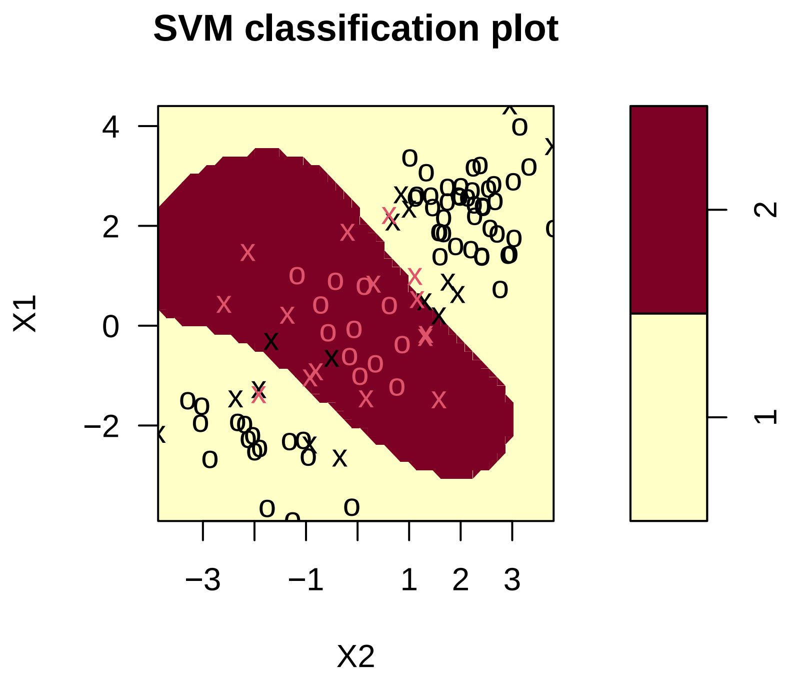
We first generate some data with a non-linear class boundary, as follows:

```r
# Generate the data
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))
dat=data.frame(x,y=as.factor(y))

# Check the non-linear class boundary
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.

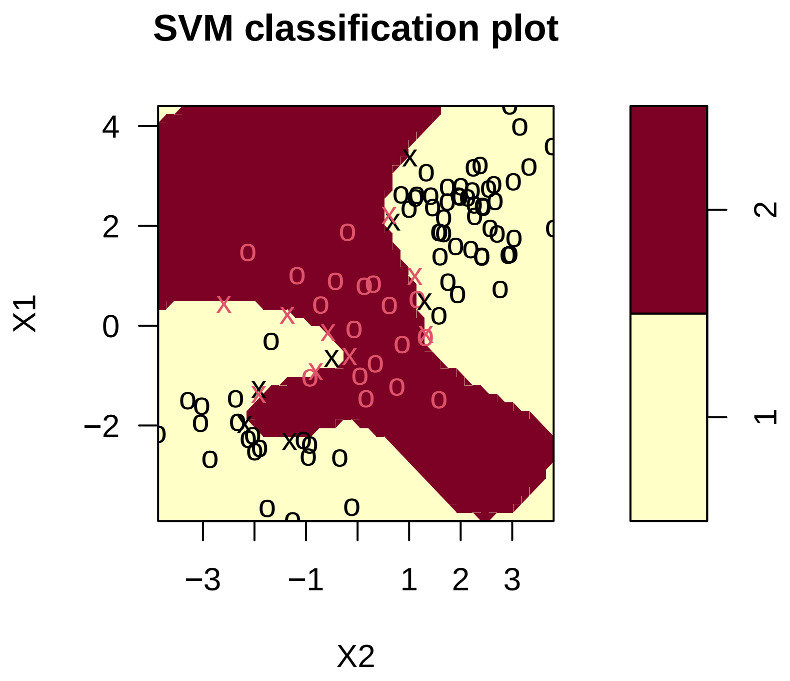
The `summary` function can be used to obtain some information about the SVM fit:

```
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 this SVM fit. If we increase the value of `cost`, we can reduce the number of training errors. However, this comes at the price 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:

```
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)$"best.parameters"
```

```
##   cost gamma
## 2    1   0.5
```

```
summary(tune.out)$"best.performance"
```

```
## [1] 0.07
```

Therefore, the best choice of parameters involves `cost=1` and $\gamma = 2$. We can view the test set predictions 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.

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

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

Your turn

Train a SVM on the training set with the following parameters:

- Use a quadratic kernel (i.e., polynomial with degree=2)
- $\gamma = 1$
- $cost = 1$

Predict the labels for response and construct confusion matrix

```
# trainig set
dat[train,]

# SVM


# Predict the true labels


# Construct the confusion matrix
```

Credit: Material adopted from: > - Chapter 9, *Introduction to Statistical Learning in R* by James, Witten, Hastie and Tibshirani > - https://lagunita.stanford.edu/c4x/HumanitiesScience/StatLearning/asset/ch9.html