9 Support Vector Machines

Andrew Liang

11/23/2020

Notes

- classification technique
- considered one of the best "out of the box" classifiers

Maximal Margin Classifier

- first we need to define *hyperplane*
  - p dimensional space
  - flat affine (doesn't need to pass through origin) subspace of a dimension $p-1$
  - the mathematical definition is:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p = 0$$

for parameters $\beta_0, \beta_1, \beta_2$

- for any $X = (X_1, X_2, \ldots, X_p)^T$ that holds the equation above, defines a hyperplane
- if X does NOT satisify the equation above, then:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p > 0$$

OR

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p < 0$$

which tells us that X must lie on either side of the hyperplane

Classification Using a Separating Hyperplane

- suppose we have a *nxp* data matrix $\mathbf{X}$ consisting of n training observations in p-dimensional space:

$$x_1 = \begin{bmatrix} x_{11} \\ \vdots \\ x_{1p} \end{bmatrix}, \ldots, x_n = \begin{bmatrix} x_{n1} \\ \vdots \\ x_{np} \end{bmatrix}$$

and also suppose these observations fall into two distinct classes: $y_1, \dots, y_n \in \{-1, 1\}$.

We are also given a test observation, a p-vector of observed features $x^* = (x_1^x, \dots, x_p^*)^T$. Our goal is to classify test observation using its feature measurements through a concept of *separating hyperplane.*

Not only can we classify observations based on the sign of $f(x^*)$, but also the *magnitude* of it as well.

- if magnitude is large, then we know that the observation x^* lies far from the hyperplane, and vice versa
- separating hyperplane leads to a linear decision boundary

Maximal Margin Classifier

- if the data can be perfectly separated using a hyperlane, then there exists an infinite number of such hyperplanes, as it can be shifted or rotated without coming into contact any of the observations. Must decide which of the infinite separating hyperplanes to use
- leads to the *maximal margin hyperplane* (AKA *optimal separating hyperplane*)
  - this is basically the hyperplane that is furthest from all the training observations
    - determined by the perpendicular distance from each data point to the hyperplane, called the *margin*
    - thus we try to maximize this distance, hence the name *maximal margin classifier*
    - can lead to overfitting when p is large
- the points that are closest and equidistant to this maximal margin hyperplane are called *support vectors*
  - they are vectors in p dimensions
  - thus the hyperplane are "supported" by these points and only depend on them, and not on any other observation
    - a movement in any of these support vectors would move the hyperplane as well

Construction of Maximal Margin Classifier

Consider n training observations $x_1, \dots, x_n \in \mathbb{R}^p$ associated with class labels $y_1, \dots, y_n \in \{-1, 1\}$. We try to solve the optimization:

$$\max_{\beta_0, \beta_1, \dots, \beta_p, M} M$$

subject to:

$$\sum_{j=1}^{p} \beta_j^2 = 1,$$
$$y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq M \qquad \forall i = 1, \dots, n$$

- the second constraint ensures that each observation is on the correct side of hyperplane

Non-separable Case * however, if no hyperplane exists, then maximal margin classifier won't exist

Support Vector Classifiers (AKA soft margin classifier)

- we now consider a classifier that does *not* perfectly separate the classes, but *most* of it, this gives a could of advantages:
  - greater robustness to individual observations
  - better classification of *most* of the training observations

Construction of Support Vector Classifiers

$$\max_{\beta_0,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_p,M} M$$

subject to:

$$\sum_{j=1}^{p} \beta_j^2 = 1,$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq M(1-\epsilon_i)$$

$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C$$

Where C is a non-negative tuning parameter.

- $\epsilon_1, \ldots, \epsilon_n$ are *slack variables* that allow individual observations to be on wrong side of the margin or the hyperlane
  - if $\epsilon_i > 0$, then the ith observation is on wrong side of the margin
  - if $\epsilon_i > 1$, then it is on the wrong side of the hyperplane
- C bounds the sums of ϵ_i's so it determines number and severeity of the violations
  - if $C = 0$ then there is no budget for violations
  - for $C > 0$, no more than C observations can be on wrong side of hyperplane
  - generally chosen via CV
- observations that lie on the margin or violate the margin will affect the hyperplane (known as *support vectors*), and so observations that lie on the correct side of the margin does not affect the support vector classifier
- meaning that it is robust to observations far away from the hyperplane
- different from LDA classifications where it depends on *all* of the observations within each class
- similar to logistic where it is robust to observations far from decision boundary

Support Vector Machines

Classification with Non-linear Decision Boundaries

- can create non-linear boundaries by expanding feature space (ie enabling quadratic and cubic terms)

Rather than fitting support vector classifer using p features, we can fit a support vector classifier using $2p$ features:

$$X_1, X_1^2, \ldots, X_p, X_p^2$$

and so our maximization probleme would be:

$$\max_{\beta_0,\beta_{11},\beta_{12},\ldots,\beta_{p1}\beta_{p2},\epsilon_1,\ldots,\epsilon_p,M} M$$

subject to:

$$y_i(\beta_0 + \sum_{j=1}^{p} \beta_{j1} x_{ij} + \sum_{j=1}^{p} \beta_{j2} x_{ij}^2) \geq M(1-\epsilon_i)$$

$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C, \quad \sum_{j=1}^{p}\sum_{k=1}^{2} \beta_{jk}^2 = 1$$

* in this case, the hyperplane is non-linear because it is a quadratic polynomial within the original feature space

Support Vector Machines (SVMs)

- extension of support vector classifier and uses *kernels* to enlarge feature space
  - allows for efficient computational approach from using kernels
- calculation involves taking *inner products* (dot products) of observations. Inner product of two observations $x_1, x_{i'}$ is given by:

$$\langle x_i, x_{i'} \rangle = \sum_{j=1}^{p} x_{ij} x_{i'j}$$

- and so the linear support vector classifier can be shown as:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle$$

where there are n parameters $\alpha_i, \quad i = 1, \ldots, n$, one per training observation

- to estimate these parameters $\alpha_1, \ldots, \alpha_n$ and β_0, we just need $\binom{n}{2}$ inner products of $\langle x_i, x_{i'} \rangle$ between all pairs of observations

It turns out that α_i is nonzero for only the *support points.* And so if S is the collection of indices for these support points, we can rewrite:

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

which typically involves fewer terms than the previous equation

We replace the inner product calculation with a *generalization* of the inner product in the form:

$$K(x_i, x_{i'})$$

where K is some function that is call the *kernel*

- the kernel is a function that quantifies the similarity of two observations, for example:

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j}$$

is just a support vector classifier (linear). The linear kernel essentially quantifies the similarity of a pair of observations using Pearson correlation. One could also replace the linear kernel with:

$$K(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij} x_{i'j})^d$$

AKA a *polynomial kernel* of degree d, where d is a positive integer. With $d > 1$, the support vector classifier algorithm leads to a more flexible decision boundary

- does this by fitting a support vector classifier in a higher-dimensional space involving polynomials of degree d, rather than in the original feature space
- the process of combining a support vector classifier and a non-linear kernel is called a *support vector machine*
- the function has the form:

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

Another popular choice is the *radial kernel*, which is:

$$K(x_i, x_{i'}) = exp(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2)$$

where γ is a positive constant (essentially a tuning parameter)

- if given test observation $x^* = (x_1^* \dots x_p^*)^T$ is far from the training observation x_i in terms of Euclidean distance, then $\sum_{j=1}^{p}(x_j^* - x_{ij})^2$ will be very large, and thus $K(x_i, x_{i'}) = exp(-\gamma \sum_{j=1}^{p}(x_{ij} - x_{i'j})^2)$ will be small. Thus, training observations far from x^* will not play a role in the predicted class label for x^*
  - means that the radial kernel has a very *local* behavior
- advantage of using kernels over expanding feature space is that it's more computationally feasible
  - only need to compute $K(x_i, x_{i'})$ for all $\binom{n}{2}$ distinct pairs i, i'

SVMs with More than Two Classes

One-Verses-One Classification (all-pairs)

- suppose we want to classify using SVMs when there are $K > 2$ classes
- this method constructs $\binom{K}{2}$ SVMs, each of which compares a pair of classes
- classify a test observation using each of the $\binom{K}{2}$ classifiers, and tally the number of times the test obs is assigned to each of the K classes
- final classification is performed by assigning the obs to class which it was most frequently assigned in these $\binom{K}{2}$ pairwise classifications
- usually used for smaller number of classes

One-Verses-All Classification

- We fit K SVMs, each time comparing one of the K classes to the remaining $K-1$ classes
- let $\beta_{0k}, \beta_{1k}, \ldots, \beta_{pk}$ denote parameters that result from fitting an SVM comparing the kth class to the others
- let x^* denote test observation, we then assign observation to the class for which $\beta_{0k}+\beta_{1k}x_1^*+\ldots+\beta_{pk}x_p^*$ is largest (classifier that yields the highest margin)
- usually used for a large amount of classes
  - amounts to highest confidence that the test obs belongs to the kth class rather than any other classes

Which Classifier to Use?

- when classes are (nearly) separable, SVM does better than Logistic, and so does LDA
  - if not, then LR (with ridge/lasso penalty) and SVM are very similar
- if you want to estimate probabilities, LR should be the choice
- for nonlinear boundaries, SVMs are popular
  - however, using kernels with LR and LDA can work as well, but computations more expensive
- drawback with SVMs is that it doesn't really select features well like the lasso, all features are used

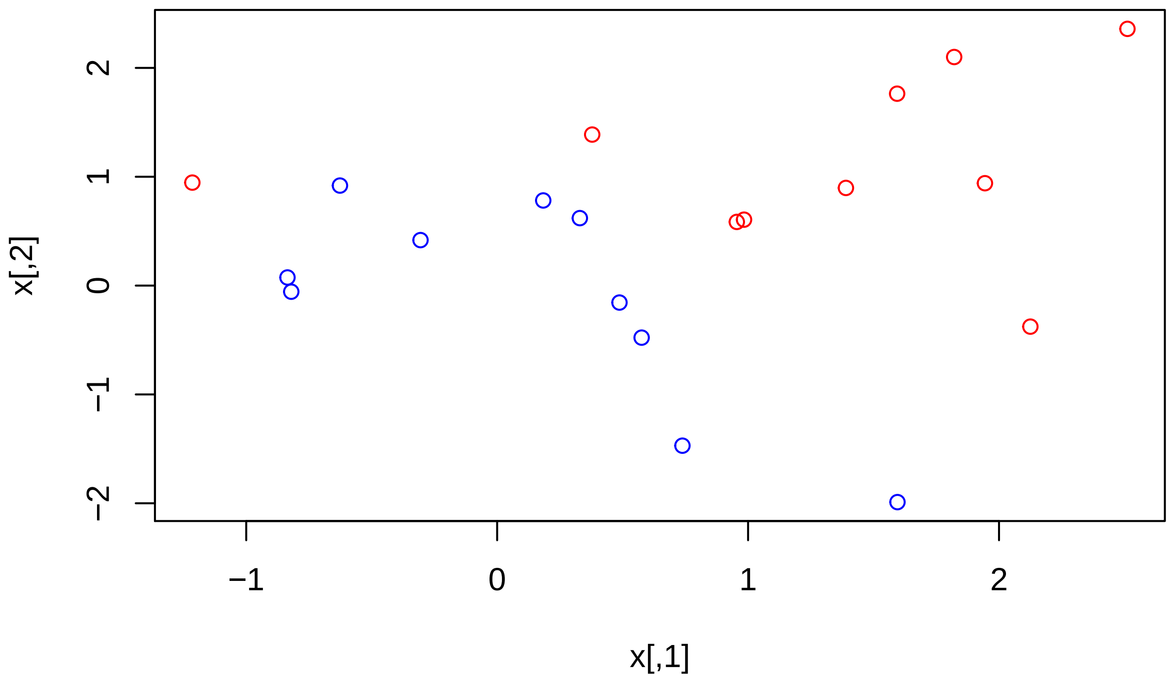
Applied

Support Vector Classifier

We use the svm() function with argument kernel = "linear"

Generate observations that belong to two classes, and checking if classes are linearly separable:

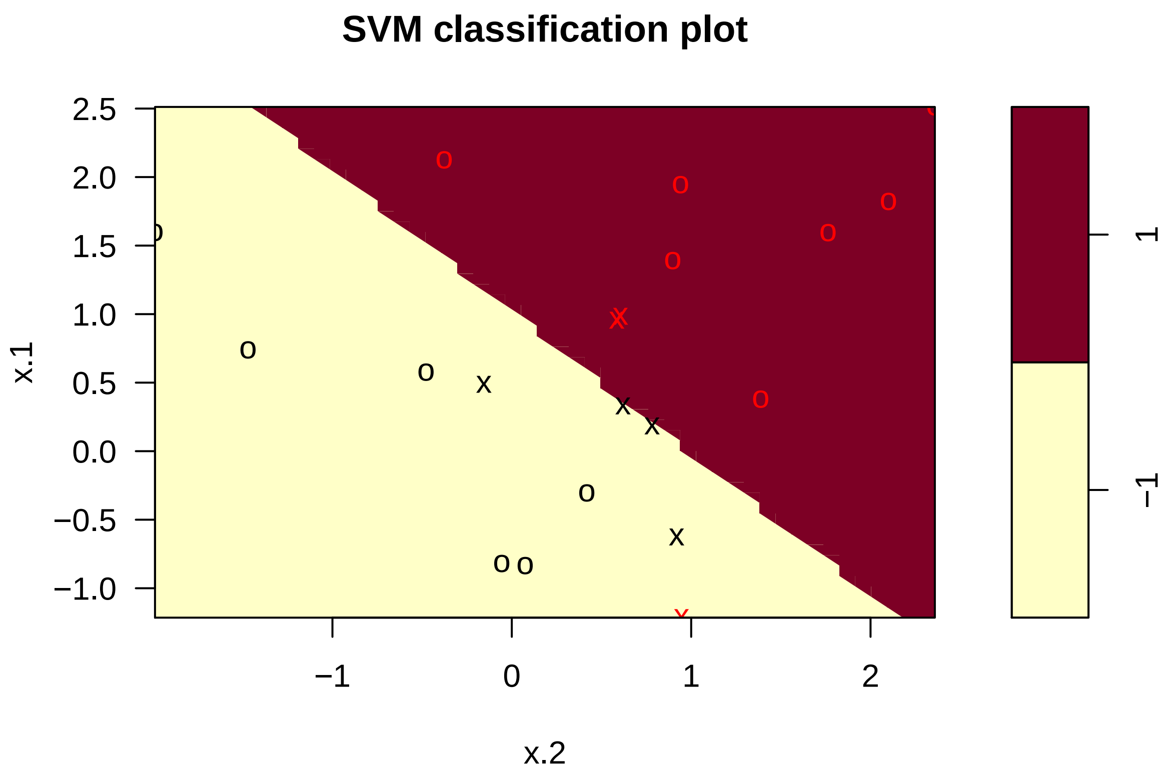
```
set.seed(1)
x <- matrix(rnorm(20*2), ncol = 2)
y <- c(rep(-1,10), rep(1,10))
x[y==1,] <- x[y==1,] + 1
plot(x, col=(3-y))
```



We can see that the observations aren't linearly separable.

Let's fit a support vector classifier:

```
dat <- data.frame(x=x, y=as.factor(y)) # must encode response as a factor

library(e1071)
svmfit <- svm(y~.,
              data = dat,
              kernel = "linear",
              cost = 10,
              scale = F) # scale = F tells function not to scale each feature to have mean zero or sd 1
plot(svmfit, dat)
```



The supported vectors are plotted as crosses and remaining observations plotted as circles. We can identify the support vectors:

```
svmfit$index
```

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

and also get some basic info on the support vector classifier:

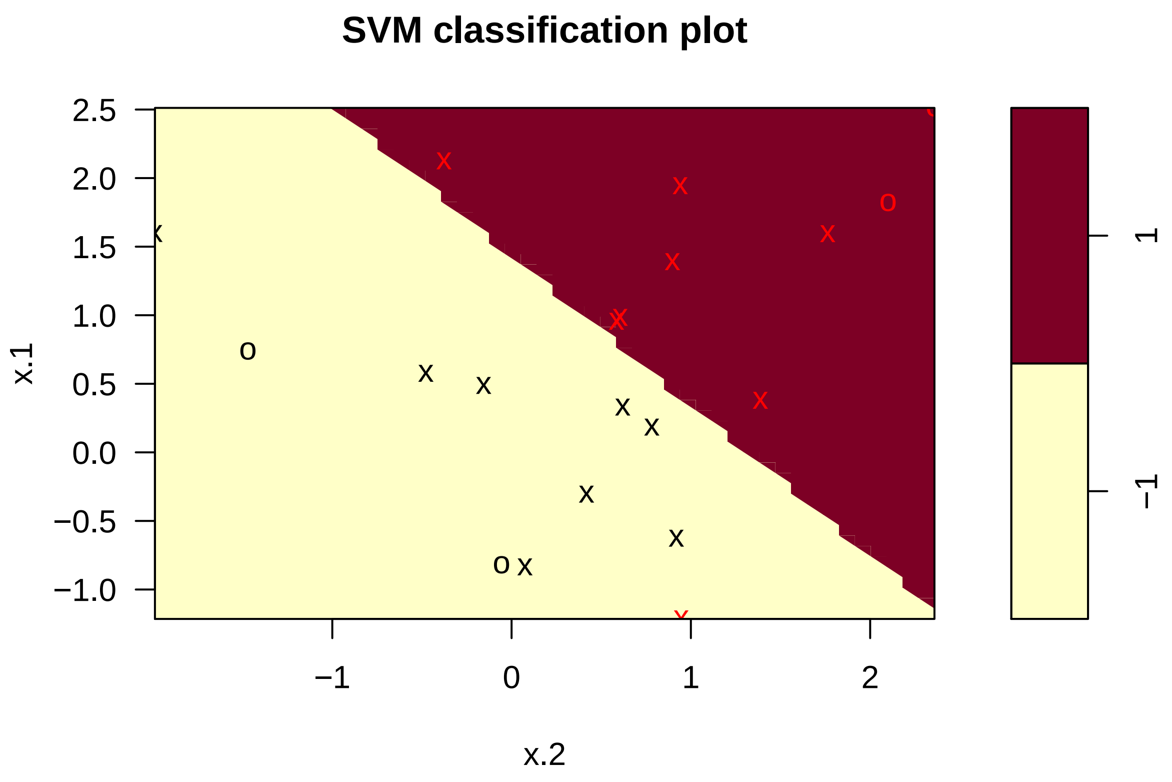
```
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10, scale = F)
##
##
## 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 that four of the support vectors were in class one and three were in class two. What if we used a smaller cost parameter?

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



```
svmfit$index
```

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

```
summary(svmfit)
```

```
##
## Call:
```

```
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 0.1, scale = F)
##
##
## 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
```

With a smaller cost parameter, we obtain a wider margin, hence we get more support vectors. The svm() does not explicilty output coefficients of linear decision boundary nor does it output the width of margin

It does include tune(), which allows us to perform CV on set of models of interest, k = 10 by default:

```r
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)))
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
```

It seems that cost = 0.1 gives lowest CV error. tune() stores the best model obtained, and can be accessed:

```r
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
```

Now we can predict the class label on a set of test observations. Let's generate a test set:

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

Now we predict:

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

```
##        truth
## predict -1 1
##      -1  9 9
##      1   2 0
```

So with cost = 0.1, we can see that it correctly classifies 9 test observations

Now let's try cost = 0.01:

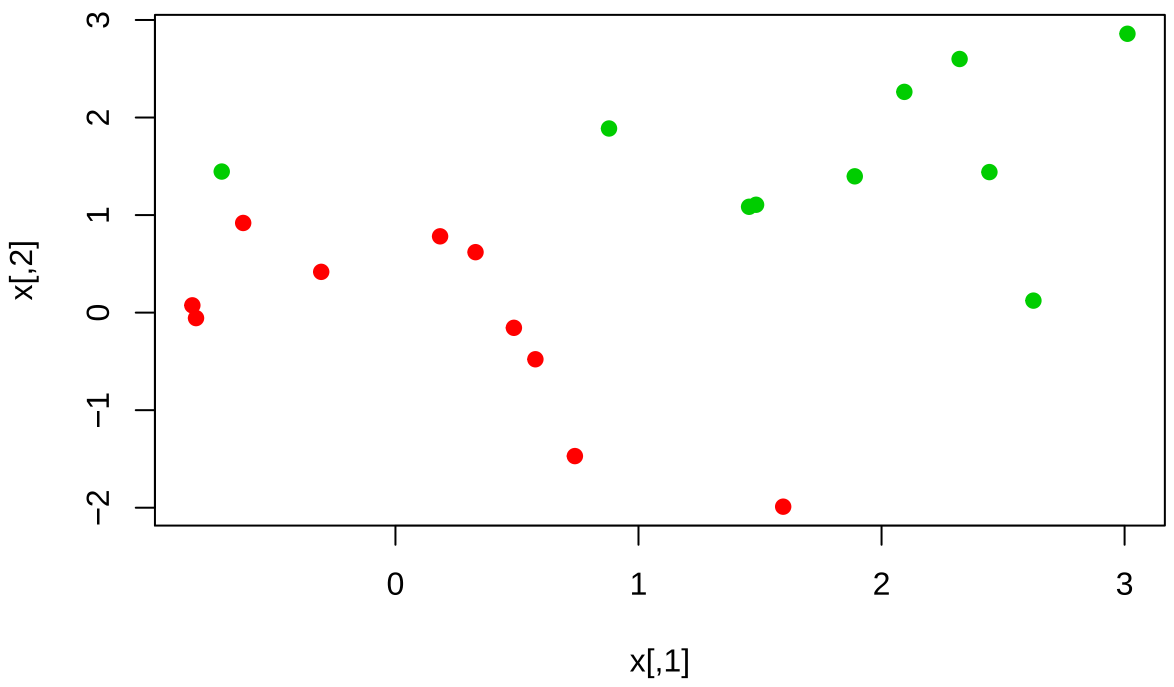
```r
svmfit <- svm(y~.,
              data = dat,
              kernel = "linear",
              cost = 0.01,
              scale = F)
ypred <- predict(svmfit, testdat)
table(predict = ypred, truth = testdat$y)
```

```
##         truth
## predict -1  1
##      -1 11  9
##      1   0  0
```

We now see that it actually predicts 10 observations correctly, which is interesting because it outperformed the "best" model

Now we consider a case when two classes are linearly separable:

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



The observations are barely linearly separable

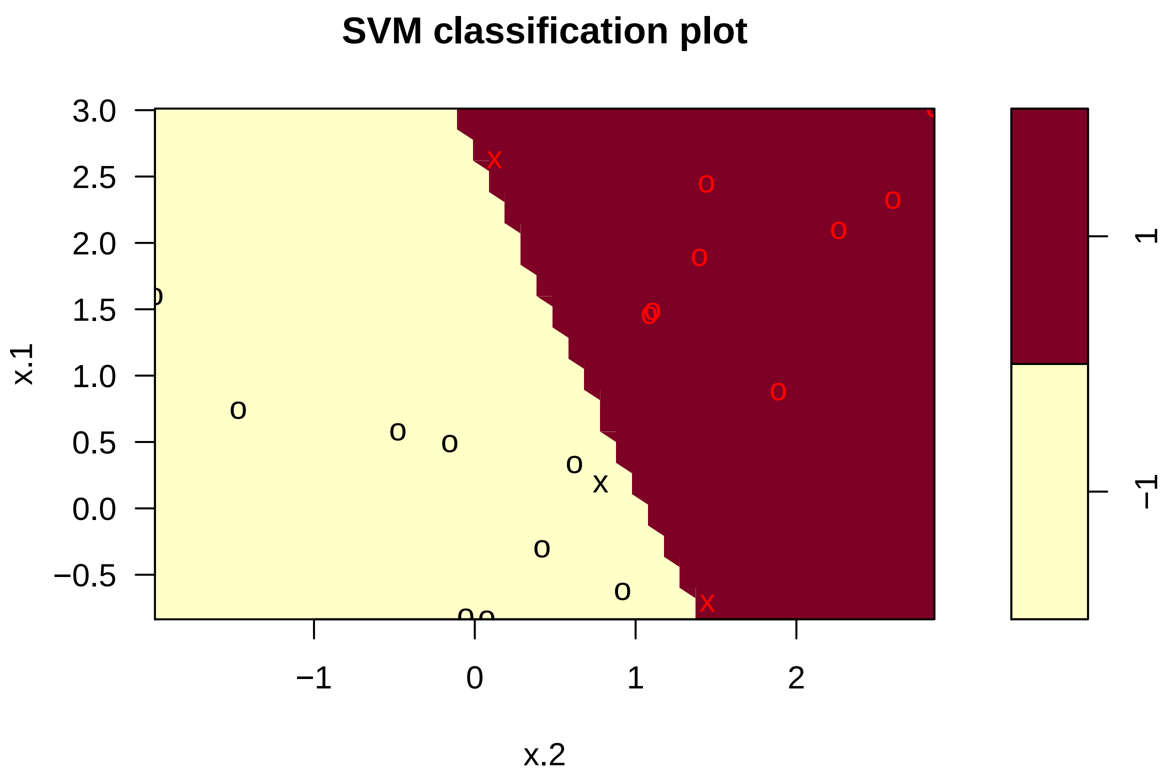
Fitting the support vector classifier using a very large value of cost so no observations are misclassified:

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

```
##
## Call:
```

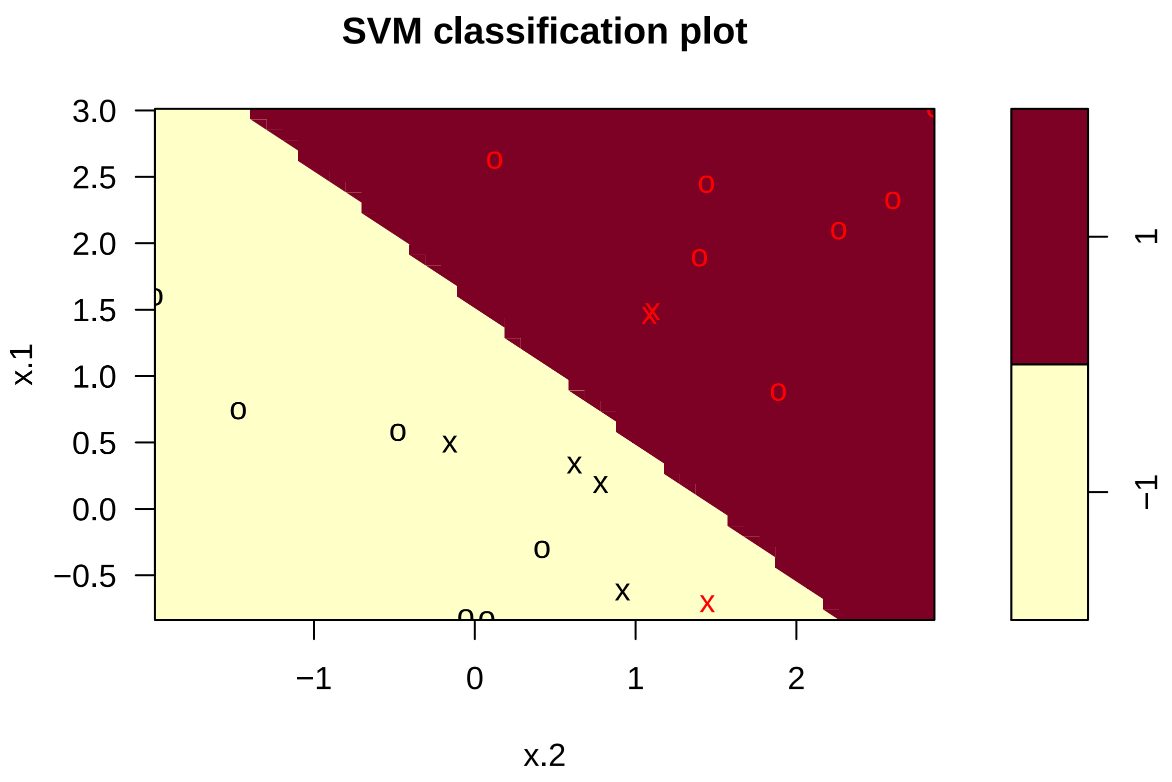
```
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 1e+05)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1e+05
##
## Number of Support Vectors:  3
##
##  ( 1 2 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

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



We can see there are no training errors and 3 support vectors were used. Margin is narrow as there are circles very close to decision boundaries; this might suggest that it will perform poorly on test set. Let's try a smaller value of cost:

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



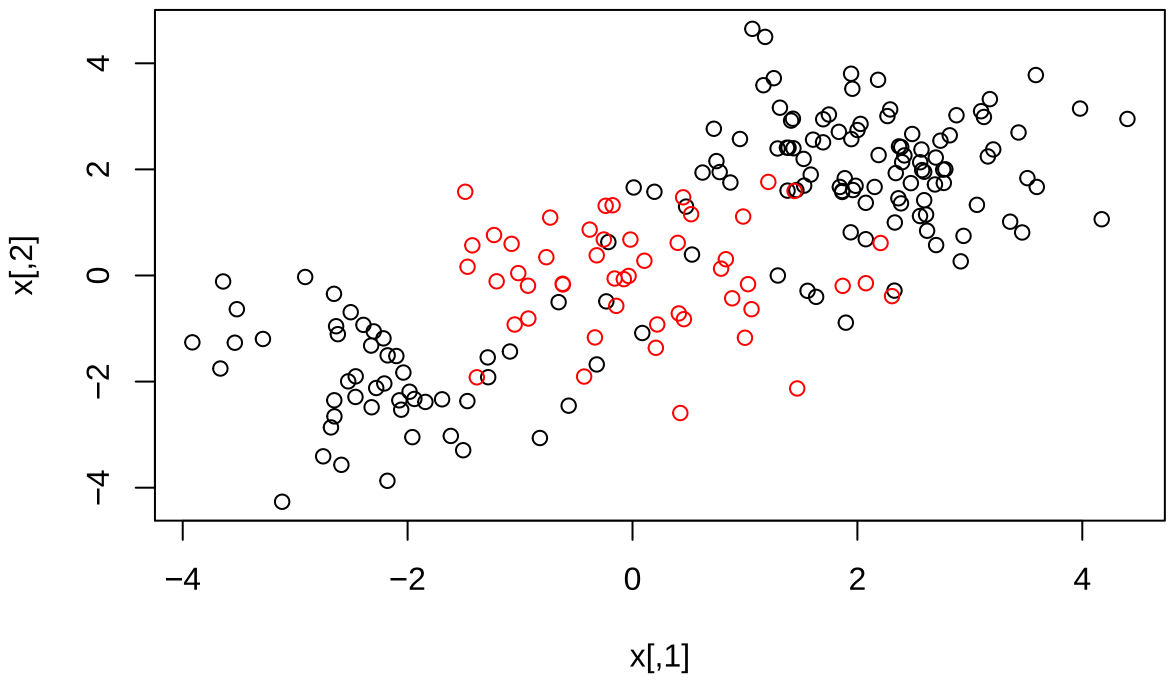
We can see that it missclassified a training observation, but we obtained a much wider margin and make use of 7 support vectors. This might suggest it will perofrm better on a test set

Support Vector Machine

We can use kernel = "polynomial" or kernel = "radial" to fit a SVM with a polynomial or radial kernel respectively

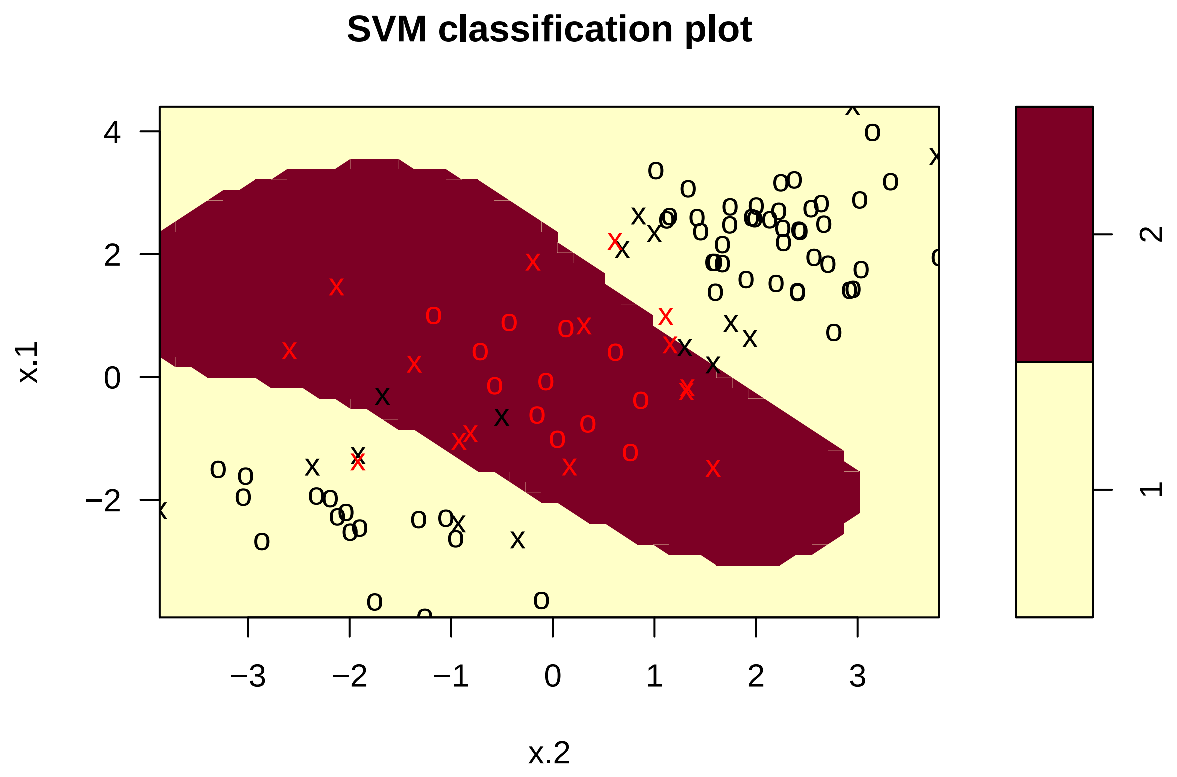
Let's generate some data with non-linear class boundaries:

```
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))

plot(x, col=y)
```



We can see that the boundaries are non-linear. Let's split a train/test, and fit a radial kernel with $\gamma = 1$:

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

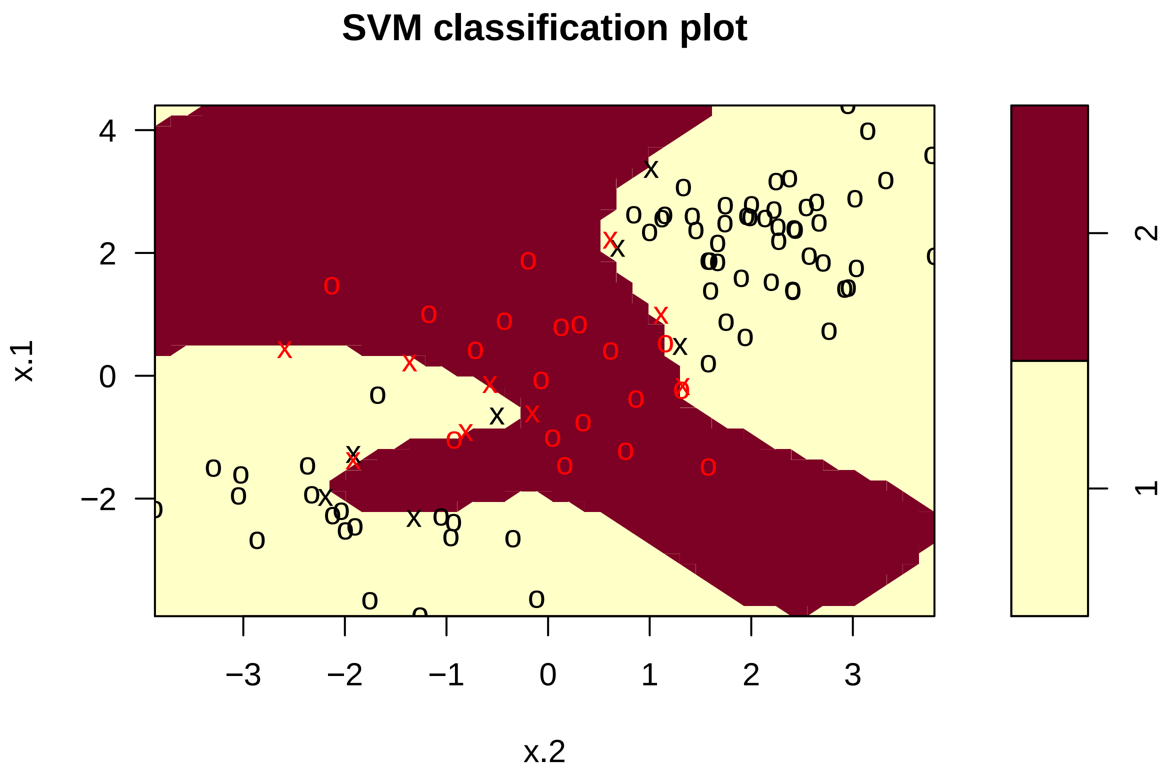


```r
summary(svmfit)
```

```
##
## 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 that there are quite a few training errors in this SVM fit. Let's increase the value of cost to fix this:

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



Even though the training errors are reduced, it comes at a price of a more irregular decision boundary due to overfitting

Let's tune γ through CV:

```
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
```

It seems the best choice of parameters involve cost = 1 and gamma = 2.

Now let's predict a test 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
```

It seems the best model correctly predicted total of 89 predictions out of 100, which is a pretty good fit (of course "good" being subjective)

ROC Curves

First, we write a function to plot an ROC curve given a vector containing numerical score for each observation: pred, and a vector containing class label for each observation: truth,

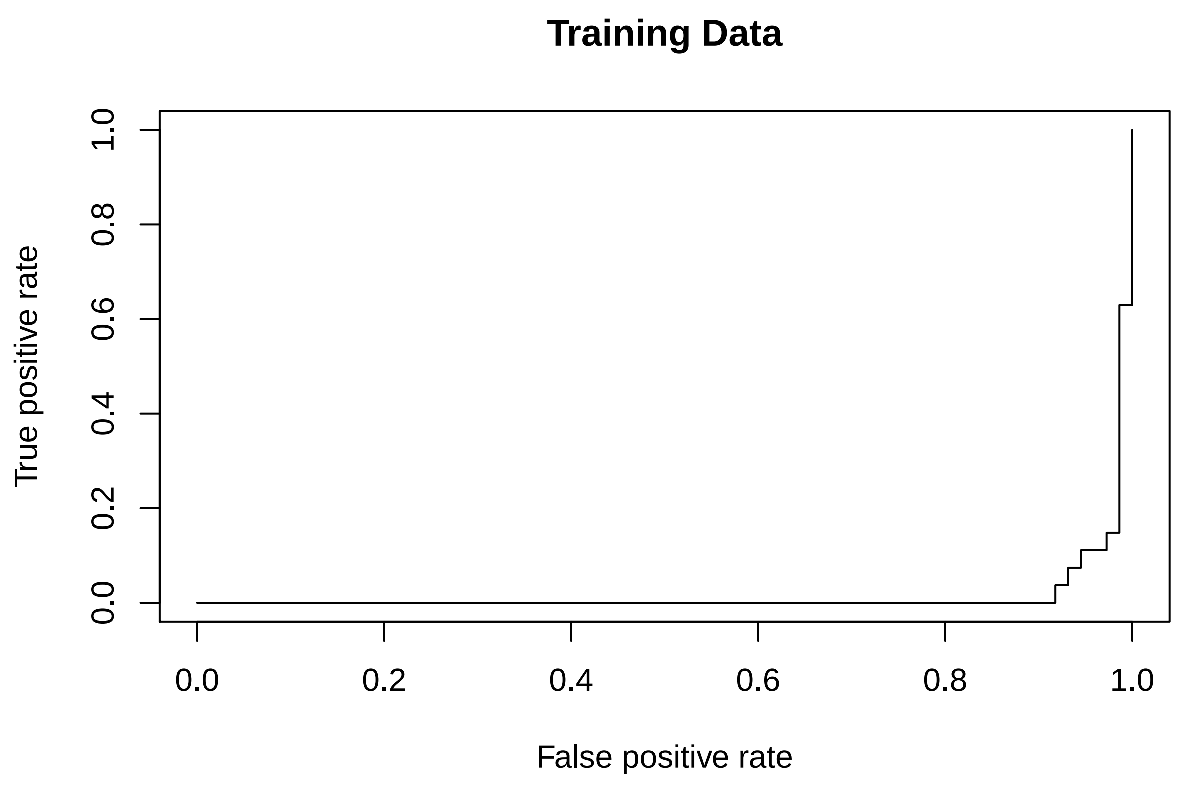
```r
library(ROCR)
rocplot <- function(pred,truth,...){
  predob <- prediction(pred,truth)
  perf <- performance(predob, "tpr","fpr")
  plot(perf,...)
}
```

Now we try to obtain the fitted values for a given SVM model fit, we use decision.values=TRUE when fitting svm(). Then the predict() function will outpu the fitted values:

```r
svmfit.opt <- svm(y~.,
                  data = dat[train,],
                  kernel= "radial",
                  gamma = 2,
                  cost = 1,
                  decision.values = T)
fitted1 <- attributes(predict(svmfit.opt,dat[train,],decision.values=T))$decision.values
```

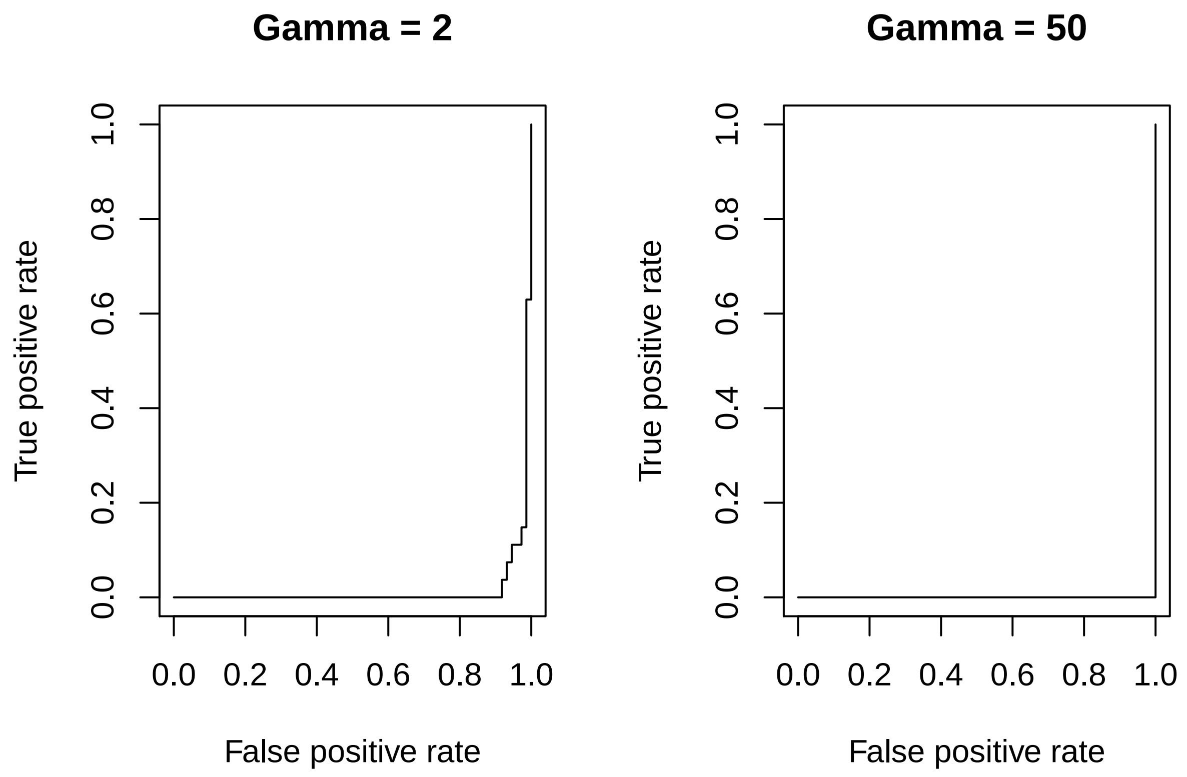
Now we can fit ROC curves:

```r
rocplot(fitted1,dat[train,"y"], main = "Training Data")
```



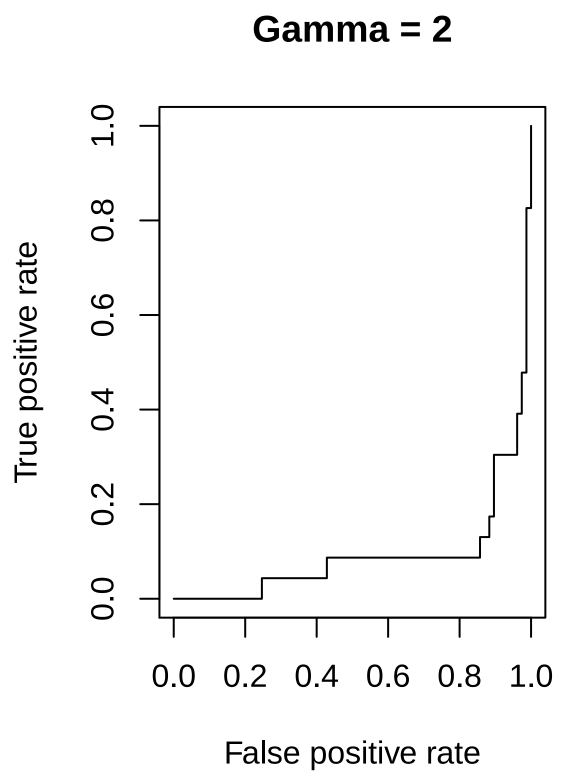
It appears the SVM model is producing accurate predictions. We can increase λ to produce a more flexible fit and generate further improvements in accuracy:

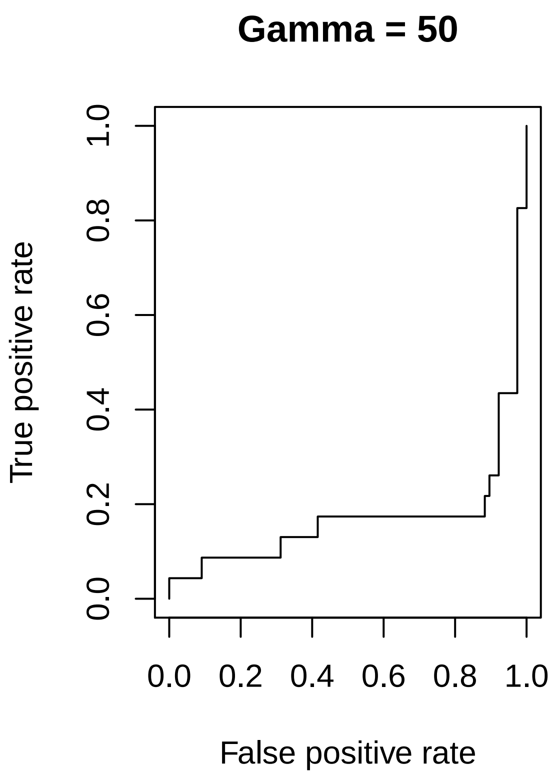
```r
svmfit.flex <- svm(y~.,
                   data = dat[train,],
                   kernel = "radial",
                   gamma = 50,
                   cost = 1,
                   decision.values = T)
fitted2 <- attributes(predict(svmfit.flex, dat[train,],decision.values = T))$decision.values

par(mfrow = c(1,2))
rocplot(fitted1, dat[train,"y"], main = "Gamma = 2")
rocplot(fitted2, dat[train,"y"], main = "Gamma = 50")
```



We can see that a higher γ values improves ROC curve. However this is on the train data. We are much more interested in the prediction accuracy on the test data, so let's try the test data:

```r
fitted1 <- attributes(predict(svmfit.opt,dat[-train,],decision.values=T))$decision.values
fitted2 <- attributes(predict(svmfit.flex,dat[-train,],decision.values=T))$decision.values

par(mfrow=c(1,2))
rocplot(fitted1, dat[-train,"y"], main = "Gamma = 2")
rocplot(fitted2, dat[-train,"y"], main = "Gamma = 50")
```



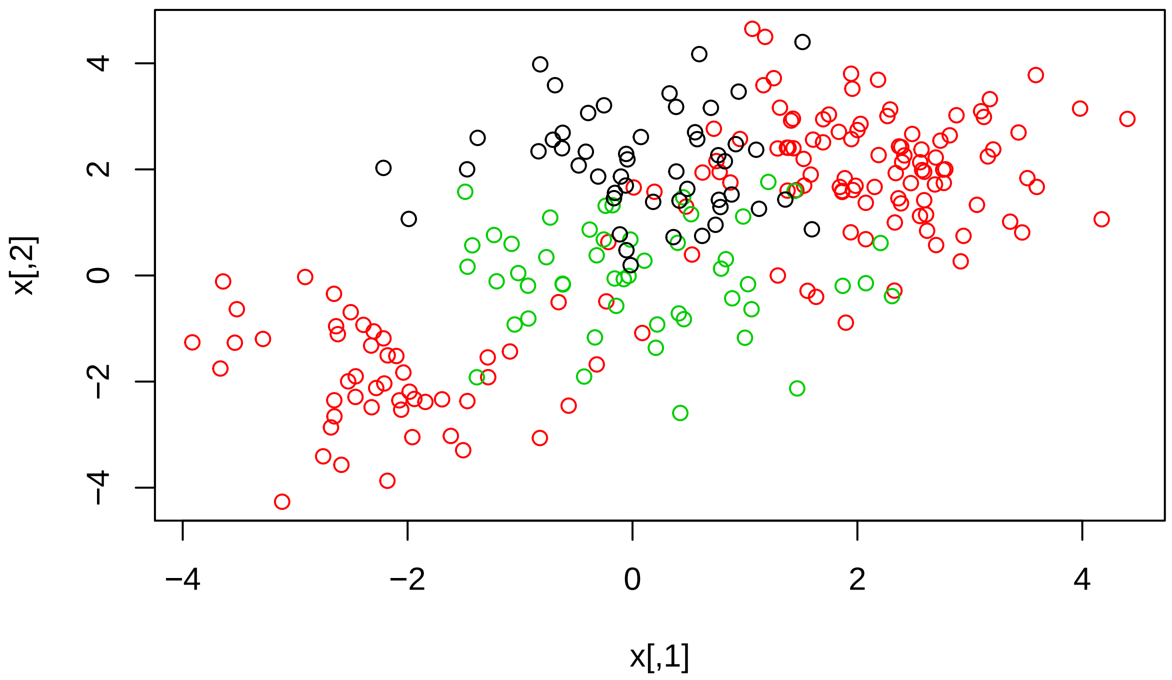


We can see that the SVM with $\gamma = 2$ performs much better on the less flexible model

SVM with More Than Two Classes

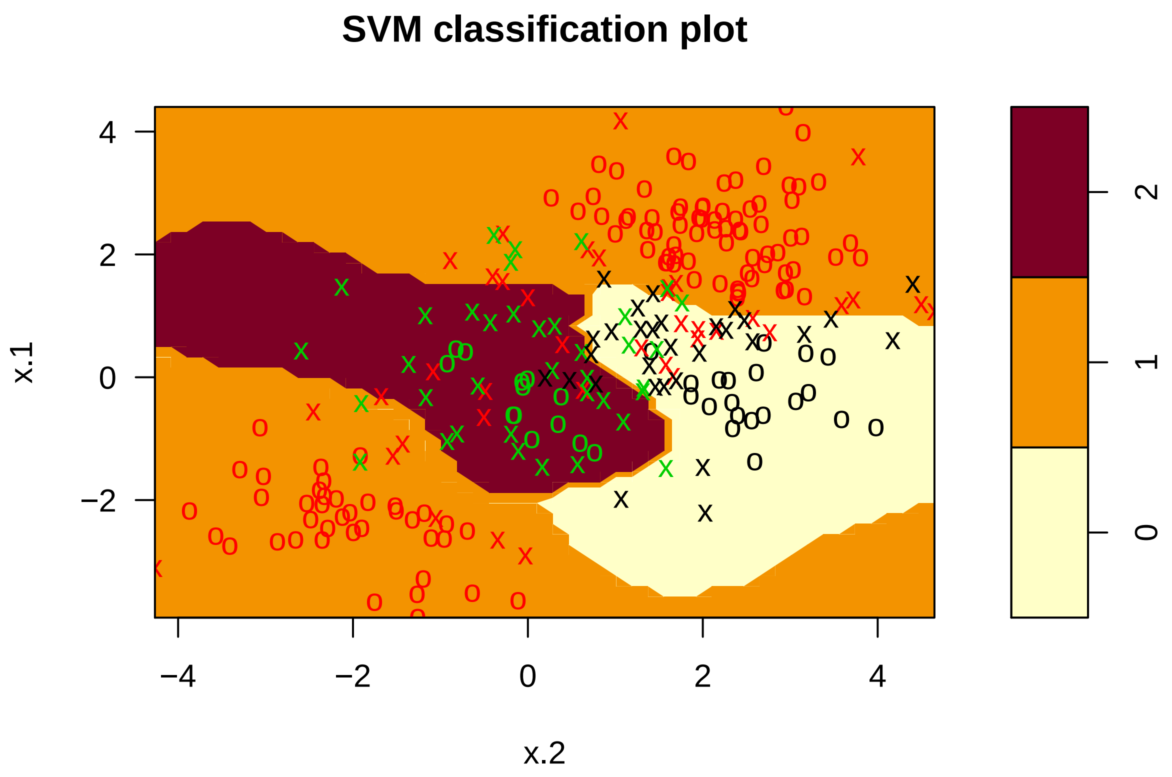
The svm() function will perofrm multi-class classification using OVO approach. Let's generate a third class of observations:

```
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))
plot(x,col=(y+1))
```



Let's fit a SVM to the data:

```r
svmfit <- svm(y~.,
              data = dat,
              kernel = "radial",
              cost = 10,
              gamma = 1)
plot(svmfit, dat)
```



Application to Gene Data

Let's explore the Khan dataset, which consists of number of tissue samples corresponding to four distinct types of small round blue cell tumors. The dataset already consists of train and test data:

```
library(ISLR)
names(Khan)
```

```
## [1] "xtrain" "xtest"  "ytrain" "ytest"
```

```
dim(Khan$xtrain)
```

```
## [1]   63 2308
```

```
dim(Khan$xtest)
```

```
## [1]   20 2308
```

```
table(Khan$ytrain)
```

```
##
##  1  2  3  4
##  8 23 12 20
```

```r
table(Khan$ytest)
```

```
##
## 1 2 3 4
## 3 6 6 5
```

So we have 2308 genes, and the train and test set consists of 63 and 20 obs respectively

We can see that there are a very large number of features relative to number of obs. This suggests we use a linear kernel, since additional flexibility will cause overfitting:

```r
dat <- data.frame(x=Khan$xtrain, y=as.factor(Khan$ytrain))
out <- svm(y~.,
           data = dat,
           kernel = "linear",
           cost = 10)
summary(out)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##
## Number of Support Vectors:  58
##
##  ( 20 20 11 7 )
##
##
## Number of Classes:  4
##
## Levels:
##  1 2 3 4
```

```r
table(out$fitted, dat$y)
```

```
##
##      1  2  3  4
##   1  8  0  0  0
##   2  0 23  0  0
##   3  0  0 12  0
##   4  0  0  0 20
```

Amazingly, there are *no* training errors. Usually with large number of variables relative to obs, it implies it is easy to find a hyperplane to fully separate the classes.

Now let's see how it fits on the test:

```
dat.te <- data.frame(x=Khan$xtest, y=as.factor(Khan$ytest))
pred.te <- predict(out, newdata = dat.te)
table(pred.te, dat.te$y)
```

```
##
## pred.te 1 2 3 4
##       1 3 0 0 0
##       2 0 6 2 0
##       3 0 0 4 0
##       4 0 0 0 5
```

It seems we've only misclassified 2 observations using cost = 10.