SVM Example

The goal is to:

- Learn how manipulate a SVM in R with the package kernlab
- Observe the effect of changing the C parameter and the kernel
- Test a SVM classifier for cancer diagnosis from gene expression data

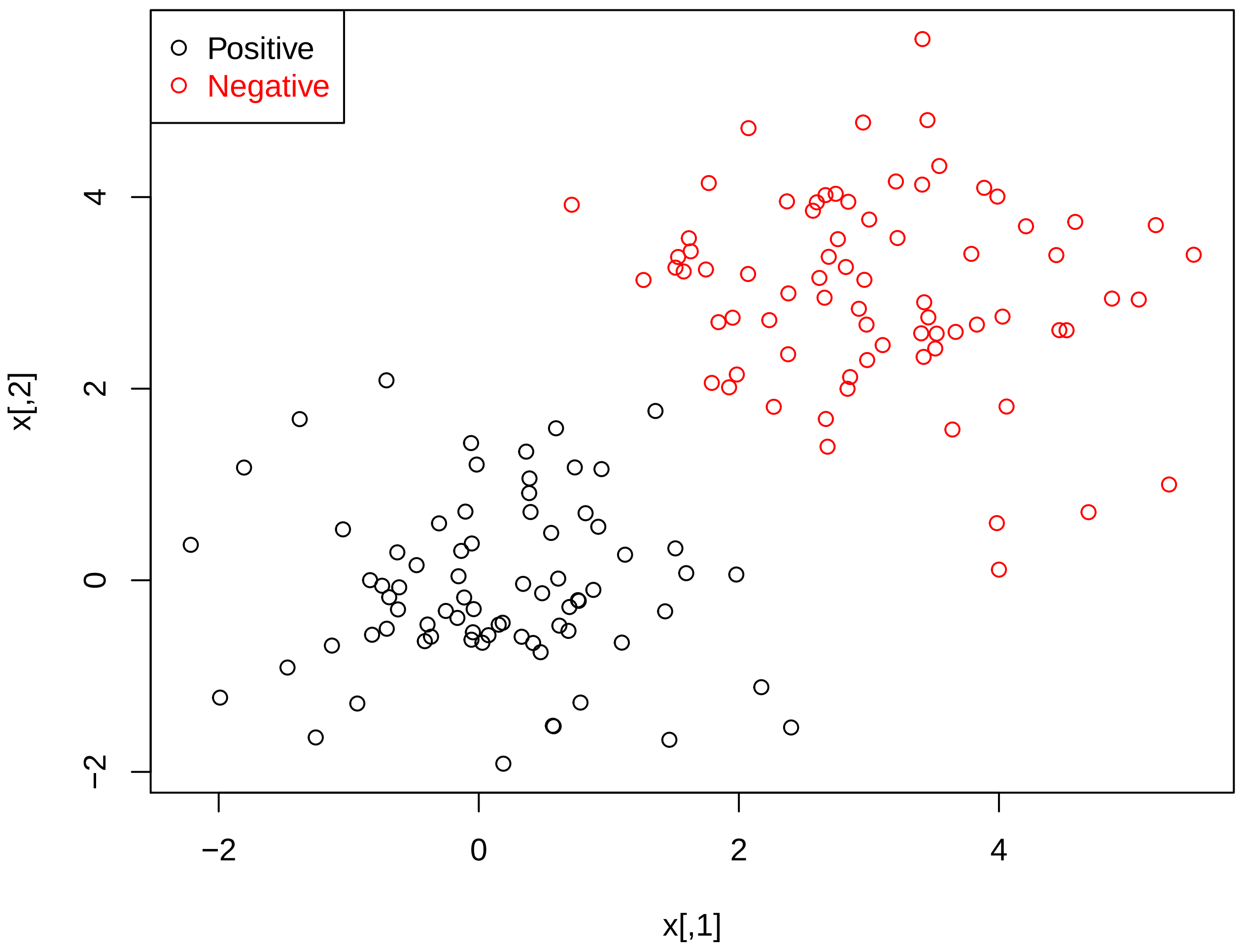
1 Linear SVM

Here we generate a toy dataset in 2D, and learn how to train and test a SVM.

1.1 Generate toy data

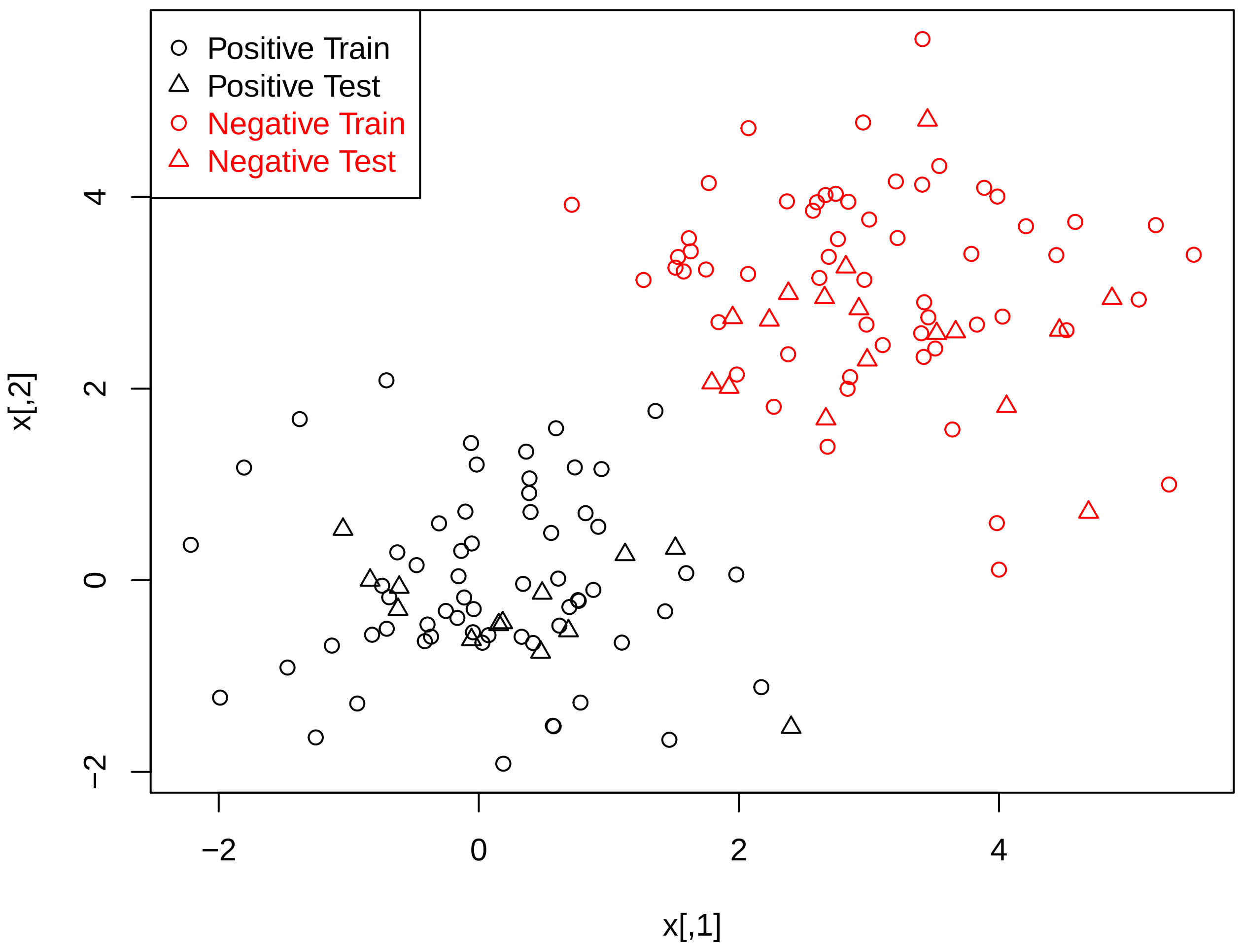
First generate a set of positive and negative examples from 2 Gaussians.

```r
set.seed(1)

n <- 150 #number of data points
p <- 2 # dimension
sigma <- 1 # variance of the distribution
meanpos <- 0 # centre of the distribution of positive examples
meanneg <- 3 # centre of the distribution of negative examples
npos <- round(n / 2) # number of positive examples
nneg <- n - npos # number of negative examples

# Generate the positive and negative examples
xpos <- matrix(rnorm(npos * p, mean = meanpos, sd = sigma), npos, p)
xneg <- matrix(rnorm(nneg * p, mean = meanneg, sd = sigma), npos, p)
x <- rbind(xpos, xneg)

# Generate the labels
y <- matrix(c(rep(1, npos), rep(-1, nneg)))

# Visualize the data
plot(x, col = ifelse(y > 0, 1, 2))
legend("topleft", c("Positive", "Negative"), col = seq(2), pch = 1, text.col = seq(2))
```



Now we split the data into a training set (80%) and a test set (20%)

```
# Prepare a training and a test set
ntrain <- round(n * 0.8) # number of training examples
tindex <- sample(n, ntrain) # indices of training samples
xtrain <- x[tindex, ]
xtest <- x[-tindex, ]
ytrain <- y[tindex]
ytest <- y[-tindex]
istrain <- rep(0, n)
istrain[tindex] <- 1

# Visualize
plot(x, col = ifelse(y > 0, 1, 2), pch = ifelse(istrain == 1,1,2))
legend("topleft", c("Positive Train", "Positive Test", "Negative Train", "Negative Test"), col = c(1, 1
```



1.2 Train a SVM

Now we train a linear SVM with parameter C=100 on the training set.

```
# load the kernlab package
# install.packages("kernlab")
library(kernlab)

# train the SVM
svp <- ksvm(xtrain, ytrain, type = "C-svc", kernel = "vanilladot", C=100, scaled=c())
```

```
##  Setting default kernel parameters
```

```
#Look and understand what svp contains
# General summary
svp
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 100
##
## Linear (vanilla) kernel function.
##
```

```
## Number of Support Vectors : 3
##
## Objective Function Value : -3.7546
## Training error : 0
```

```
# Attributes that you can access
attributes(svp)
```

```
## $param
## $param$C
## [1] 100
##
##
## $scaling
## `\001NULL\001`
##
## $coef
## $coef[[1]]
## [1] -3.5699332  3.7543303 -0.1843971
##
##
## $alphaindex
## $alphaindex[[1]]
## [1]  1 65 90
##
##
## $b
## [1] -6.859059
##
## $obj
## [1] -3.754606
##
## $SVindex
## [1]  1 65 90
##
## $nSV
## [1] 3
##
## $prior
## $prior[[1]]
## $prior[[1]]$prior1
## [1] 62
##
## $prior[[1]]$prior0
## [1] 58
##
##
##
## $prob.model
## $prob.model[[1]]
## NULL
##
##
## $alpha
## $alpha[[1]]
## [1] 3.5699332 3.7543303 0.1843971
##
##
## $type
## [1] "C-svc"
##
## $kernelf
## function (x, y = NULL)
## {
##     if (!is(x, "vector"))
##         stop("x must be a vector")
##     if (!is(y, "vector") && !is.null(y))
##         stop("y must be a vector")
##     if (is(x, "vector") && is.null(y)) {
##         crossprod(x)
##     }
##     if (is(x, "vector") && is(y, "vector")) {
##         if (!length(x) == length(y))
##             stop("number of dimension must be the same on both data points")
##         crossprod(x, y)
##     }
## }
## <environment: 0x438ec38>
## attr(,"kpar")
## list()
## attr(,"class")
## [1] "vanillakernel"
## attr(,"class")attr(,"package")
## [1] "kernlab"
##
## $kpar
## list()
##
## $xmatrix
## $xmatrix[[1]]
##          X1       X2
## 1  1.984153 2.148143
## 65 1.358680 1.767287
## 90 1.266782 3.134448
##
##
## $ymatrix
##   [1] -1 -1  1 -1 -1 -1  1  1 -1  1 -1  1  1 -1 -1  1  1  1 -1  1 -1  1  1
##  [24]  1  1 -1  1 -1  1 -1 -1  1  1 -1  1  1  1 -1 -1  1 -1  1 -1 -1 -1  1
##  [47]  1 -1 -1 -1  1 -1  1  1 -1  1  1 -1 -1 -1  1  1  1 -1  1  1 -1  1 -1
##  [70] -1  1  1  1 -1 -1 -1  1 -1 -1  1  1 -1  1  1  1 -1  1 -1  1 -1  1  1
##  [93]  1  1 -1 -1  1  1 -1  1  1 -1 -1 -1  1 -1 -1  1  1 -1  1  1 -1 -1 -1
## [116] -1 -1 -1  1  1
##
## $fitted
##   [1] -1 -1  1 -1 -1 -1  1  1 -1  1 -1  1  1 -1 -1  1  1  1 -1  1 -1  1  1
##  [24]  1  1 -1  1 -1  1 -1 -1  1  1 -1  1  1  1 -1 -1  1 -1  1 -1 -1 -1  1
##  [47]  1 -1 -1 -1  1 -1  1  1 -1  1  1 -1 -1 -1  1  1  1 -1  1  1 -1  1 -1
##  [70] -1  1  1  1 -1 -1 -1  1 -1 -1  1  1 -1  1  1  1 -1  1 -1  1 -1  1  1
##  [93]  1  1 -1 -1  1  1 -1  1  1 -1 -1 -1  1 -1 -1  1  1 -1  1  1 -1 -1 -1
## [116] -1 -1 -1  1  1
##
## $lev
## [1] -1  1
##
## $nclass
## [1] 2
##
## $error
## [1] 0
##
## $cross
## [1] -1
##
## $n.action
## function (object, ...)
## UseMethod("na.omit")
## <bytecode: 0x3101a28>
## <environment: namespace:stats>
##
## $terms
## `\001NULL\001`
##
## $kcall
## .local(x = x, y = ..1, scaled = ..5, type = "C-svc", kernel = "vanilladot",
##     C = 100)
##
## $class
## [1] "ksvm"
## attr(,"package")
## [1] "kernlab"
```

```r
# For example, the support vectors
alpha(svp)
```

```
## [[1]]
## [1] 3.5699332 3.7543303 0.1843971
```

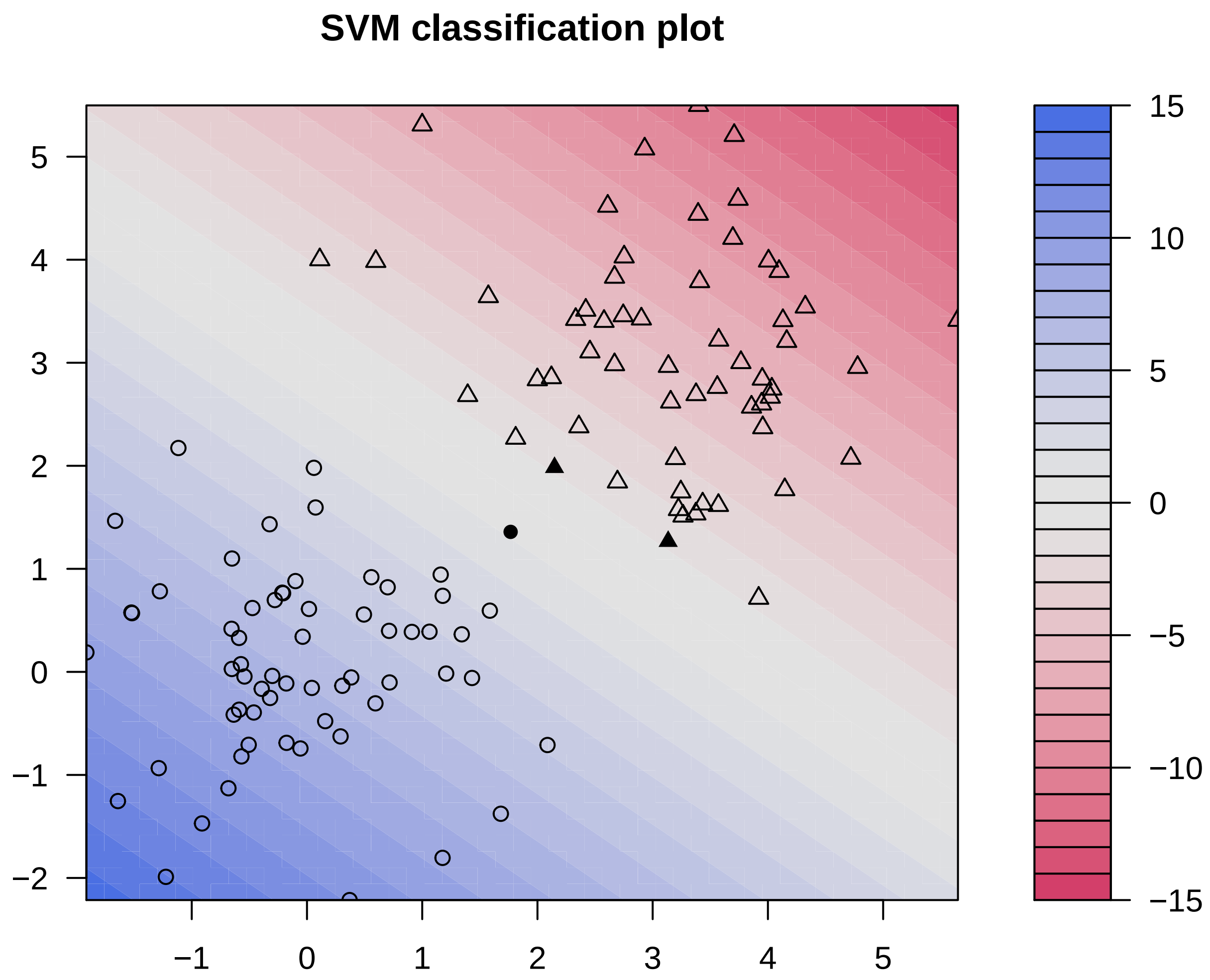
```r
alphaindex(svp)
```

```
## [[1]]
## [1]  1 65 90
```

```r
b(svp)
```

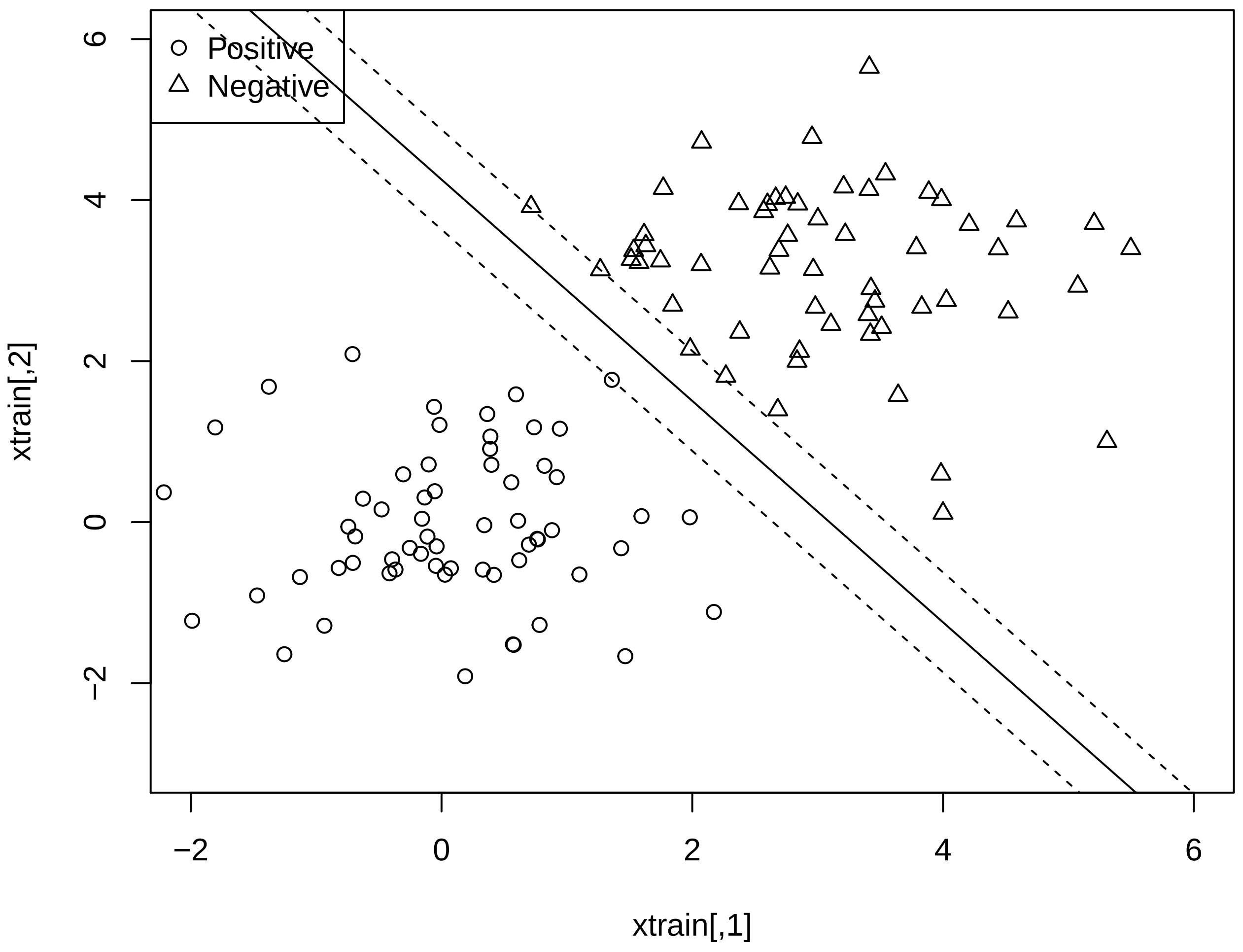
```
## [1] -6.859059
```

```r
# Use the built-in function to pretty-plot the classifier
plot(svp, data = xtrain)
```



Next, we create a function `plotlinearsvm=function(svp,xtrain)` to plot the points and the decision boundaries of a linear SVM.

```
plotlinearsvm <- function(svp, xtrain, plot_legend=T){
  plot(xtrain, pch = ifelse(ytrain > 0, 1, 2), xlim = c(-2, 6), ylim = c(-3, 6))
  if (plot_legend)
    legend("topleft", c("Positive", "Negative"), pch = seq(2))
  w = colSums(unlist(alpha(svp)) * ytrain[unlist(alphaindex(svp))] * xtrain[unlist(alphaindex(svp)),])
  b = - b(svp)
  abline(a= -b / w[2], b = -w[1]/w[2])
  abline(a= (-b+1)/ w[2], b = -w[1]/w[2], lty = 2)
  abline(a= (-b-1)/ w[2], b = -w[1]/w[2], lty = 2)
}

plotlinearsvm(svp, xtrain)
```



The figure represents a linear SVM with decision boundary $f(x) = 0$. Dotted lines correspond to the level $f(x) = 1$ and $f(x) = -1$.

1.3 Predict with a SVM

Now we can use the trained SVM to predict the label of points in the test set, and we analyze the results using variant metrics.

```r
# Predict labels on test
ypred <- predict(svp, xtest)
table(ytest, ypred)
```

```
##      ypred
## ytest -1  1
##    -1 17  0
##    1   0 13
```

```r
# Compute accuracy
sum(ypred == ytest) / length(ytest)
```

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

```r
# Compute at the prediction scores
ypredscore <- predict(svp, xtest, type = "decision")
```

```r
# Check that the predicted labels are the signs of the scores
table(ypredscore > 0, ypred)
```

```
##        ypred
##          -1  1
##   FALSE 17  0
##   TRUE   0 13
```

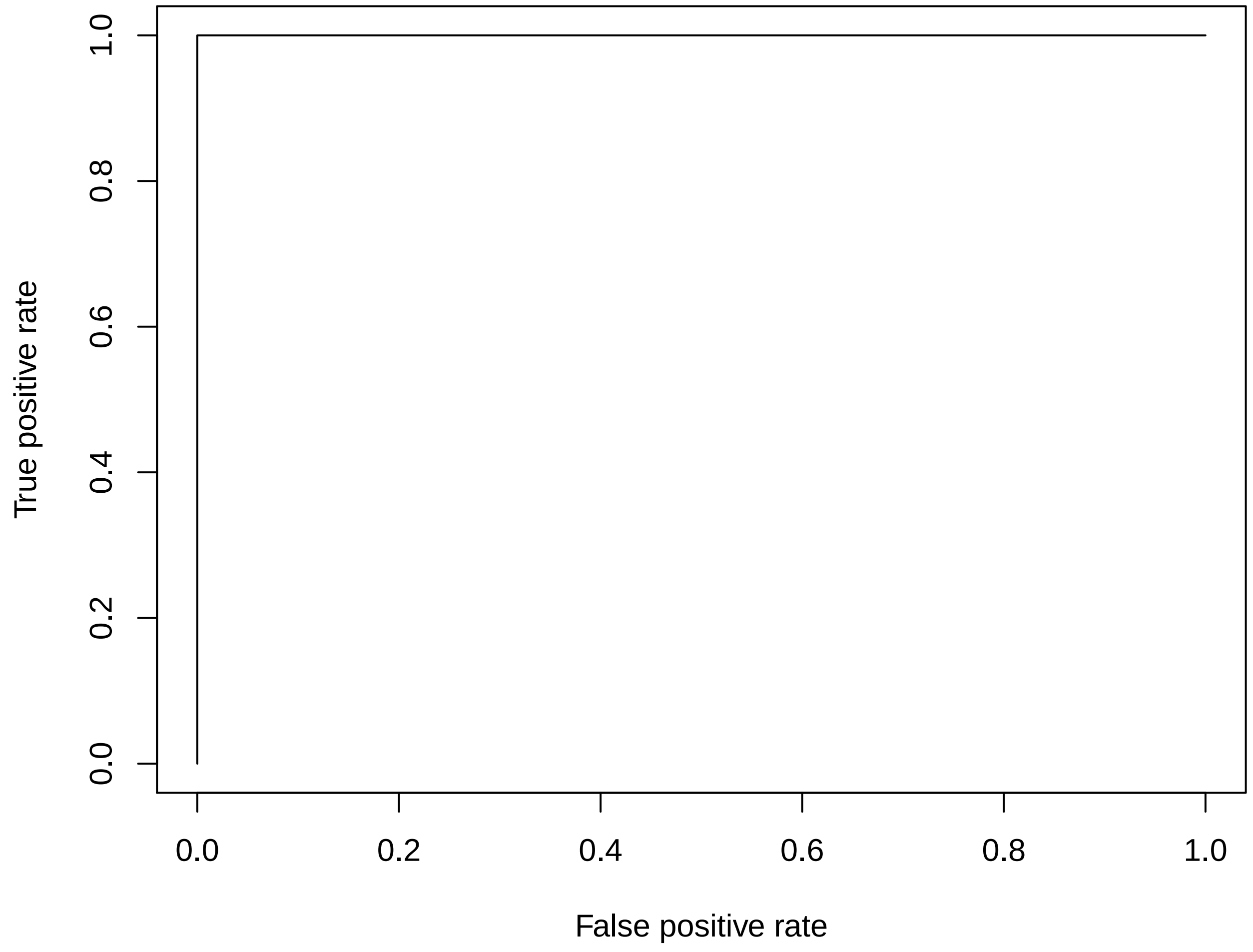
```r
# Package to compute ROC curve, precision-recall etc...
# install.packages("ROCR")
library(ROCR)
```

```
## Loading required package: gplots
```

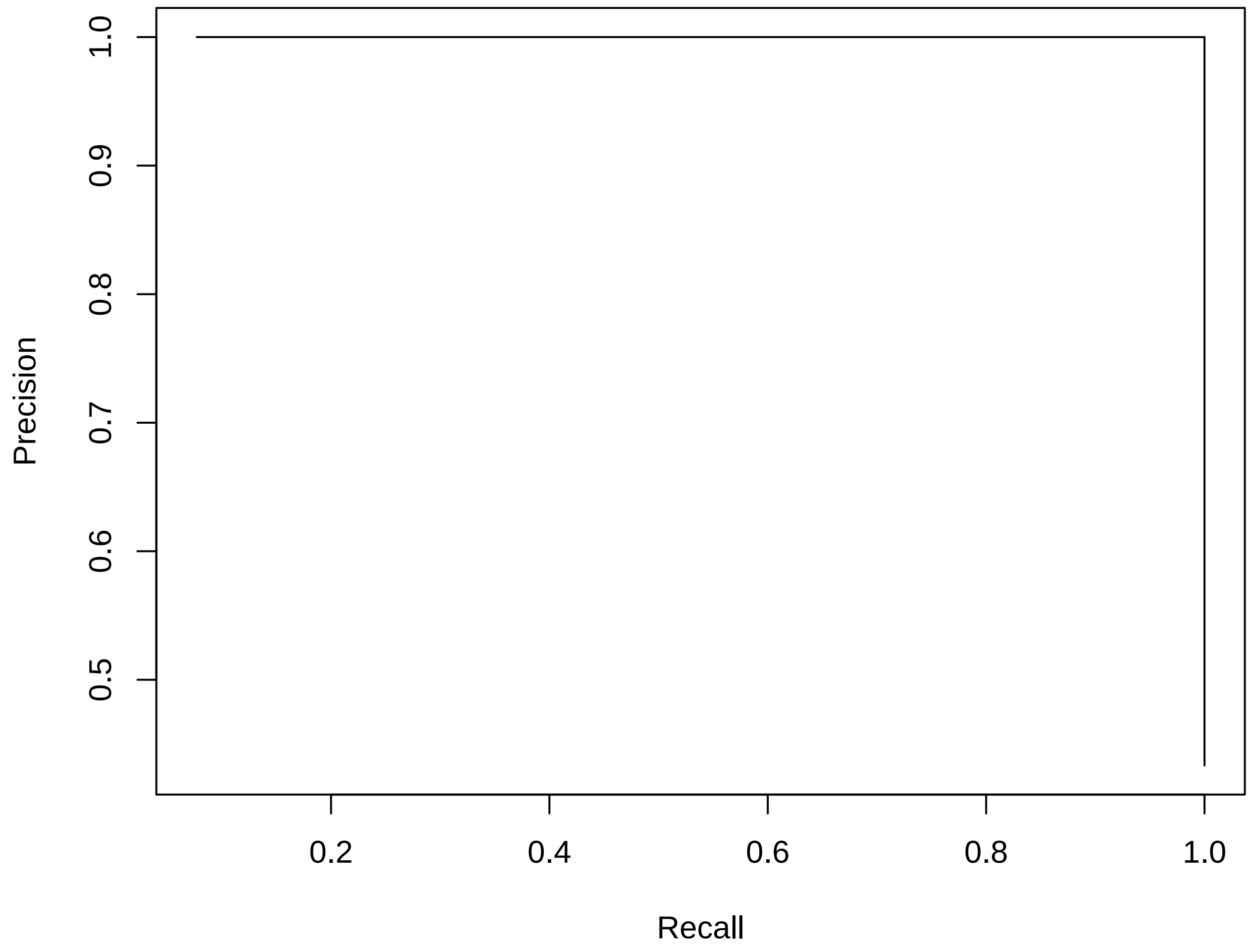
```
##
## Attaching package: 'gplots'
```

```
## The following object is masked from 'package:stats':
##
##     lowess
```

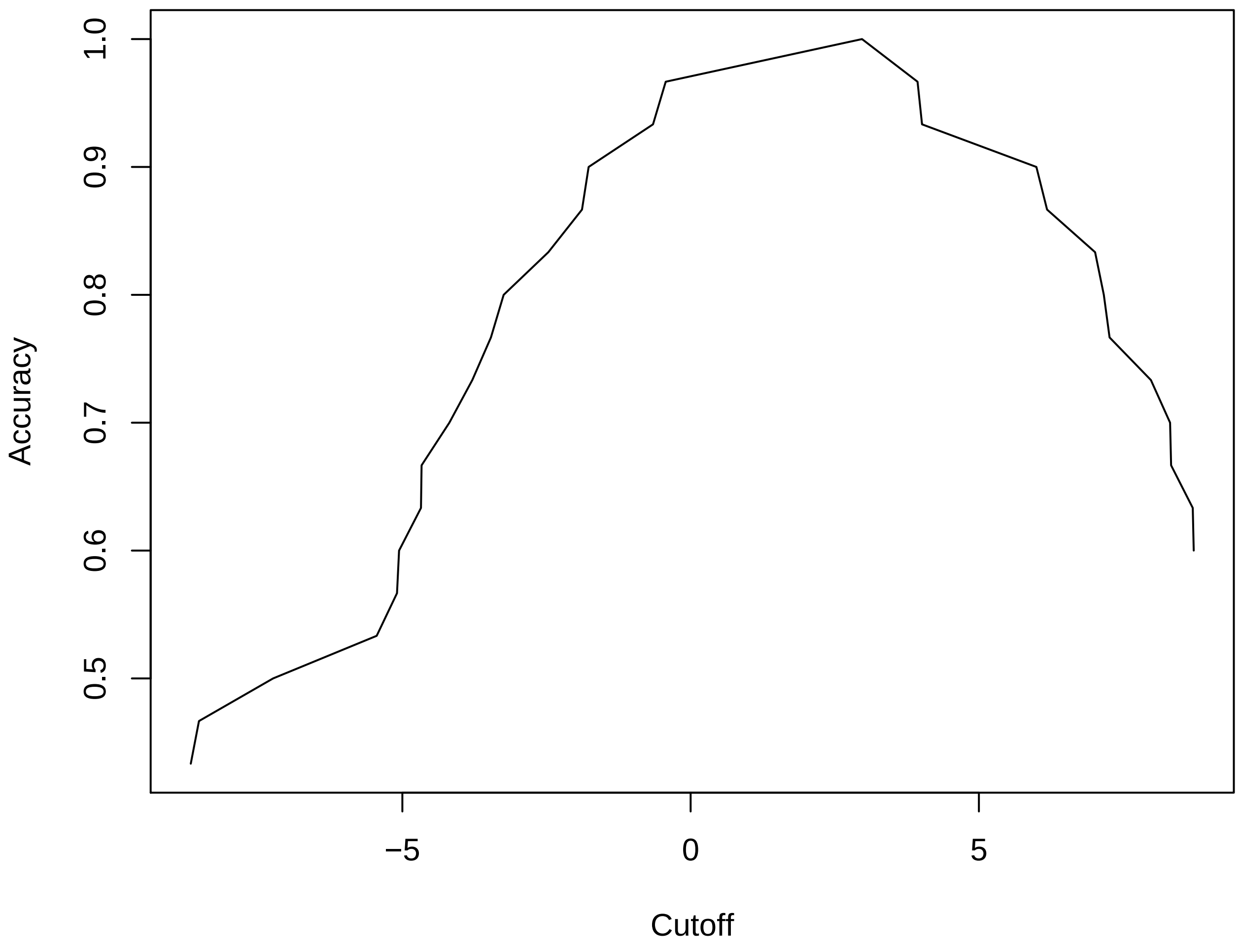
```r
pred <- prediction(ypredscore, ytest)

# Plot ROC curve
perf <- performance(pred, measure = "tpr", x.measure = "fpr")
plot(perf)
```



```r
# Plot precision/recall curve
perf <- performance(pred, measure = "prec", x.measure = "rec")
plot(perf)
```



```r
# Plot accuracy as function of threshold
perf <- performance(pred, measure = "acc")
plot(perf)
```



2 Cross-validation

Instead of fixing a training set and a test set, we can improve the quality of these estimates by running k-fold cross-validation. We split the training set in k groups of approximately the same size, then iteratively train a SVM using k - 1 groups and make prediction on the group which was left aside. When k is equal to the number of training points, we talk of leave-one-out (LOO) cross-validatin. To generate a random split of n points in k folds, we can for example create the following function:

```
cv.folds <- function(y, folds = 3){
  ## randomly split the n samples into folds
  split(sample(length(y)), rep(1:folds, length = length(y)))
}
```

We write a function `cv.ksvm = function(x, y, folds = 3,...)` which returns a vector ypred of predicted decision score for all points by k-fold cross-validation.

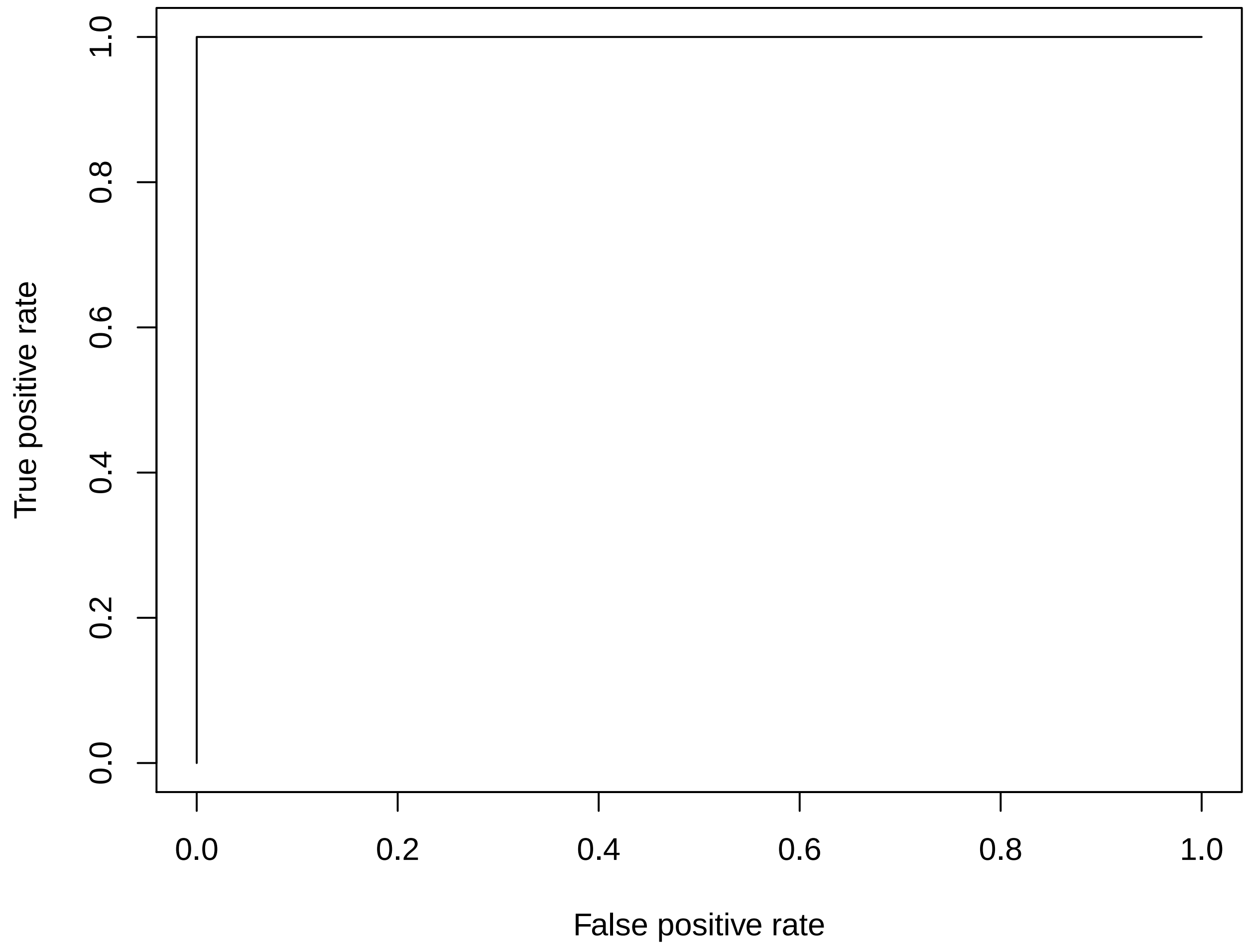
```
cv.ksvm <- function(x, y, folds = 3,...){
  index = cv.folds(y, folds = folds)
  predScore = rep(NA, length(y))
  for (i in 1:folds){
    toTrain = unname(unlist(index[-i]))
    testSet = index[[i]]
    svp = ksvm(x[toTrain, ], y[toTrain], type = "C-svc",
         kernel = "vanilladot", C=100, scaled=c())
    predScore[testSet] = predict(svp, x[unlist(index[[i]]), ], type = "decision")
  }
 predScore
}
```

We compute the various performance of the SVM by 5-fold cross-validation.

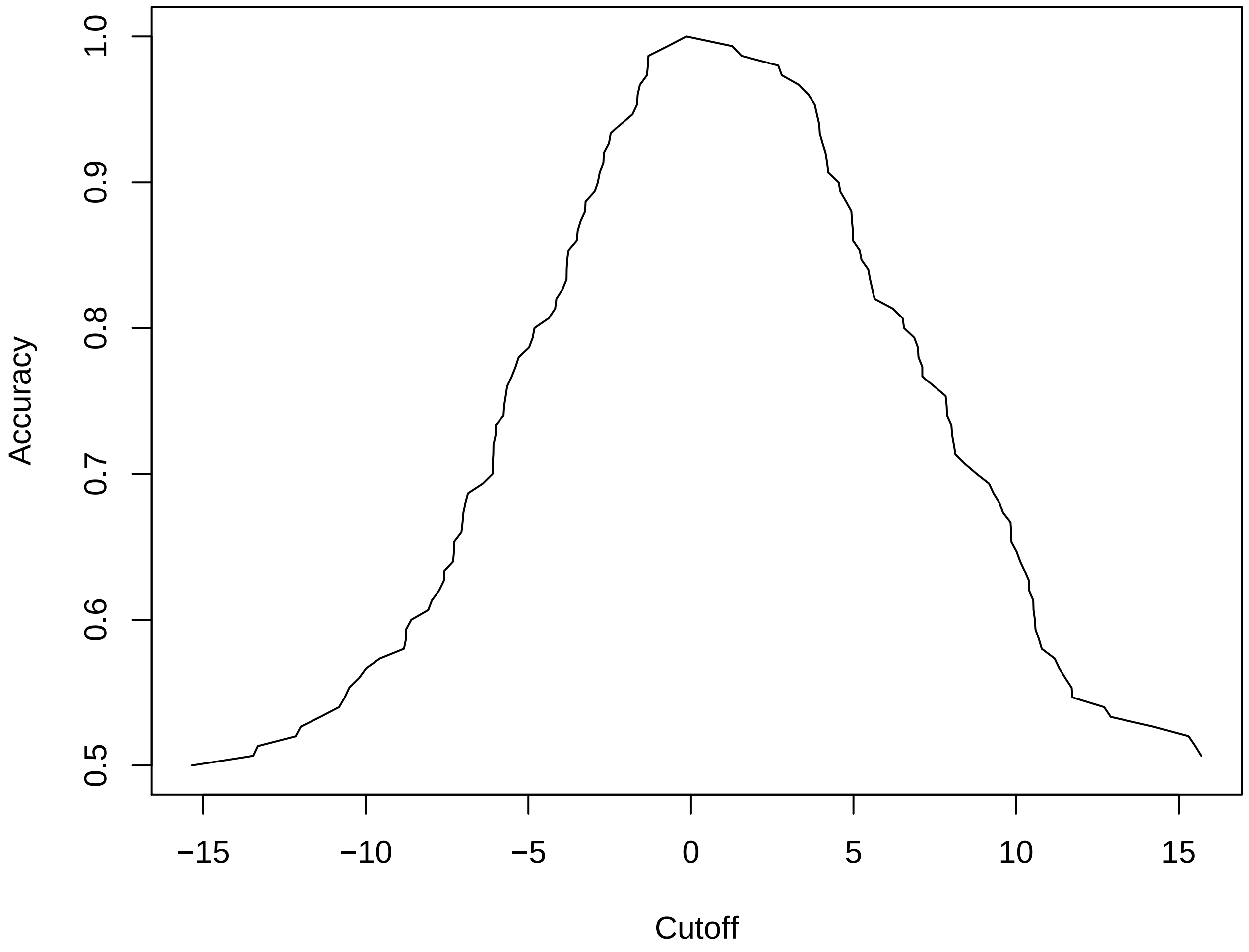
```
ypredscore = cv.ksvm(x, y, folds=5)
```

```
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
```

```
pred = prediction(ypredscore, y)
perf = performance(pred, measure = "tpr", x.measure = "fpr")
plot(perf)
```



```
perf = performance(pred, measure = "acc")
plot(perf)
```



Alternatively, the ksvm function can automatically compute the k-fold cross-validation accuracy.

```
svp <- ksvm(x, y, type = "C-svc", kernel = "vanilladot", C = 100, scaled=c(), cross = 5)
```

```
##  Setting default kernel parameters
```

```
print(cross(svp))
```

```
## [1] 0.006666667
```

```
print(error(svp))
```

```
## [1] 0
```

2.1 Effect of C

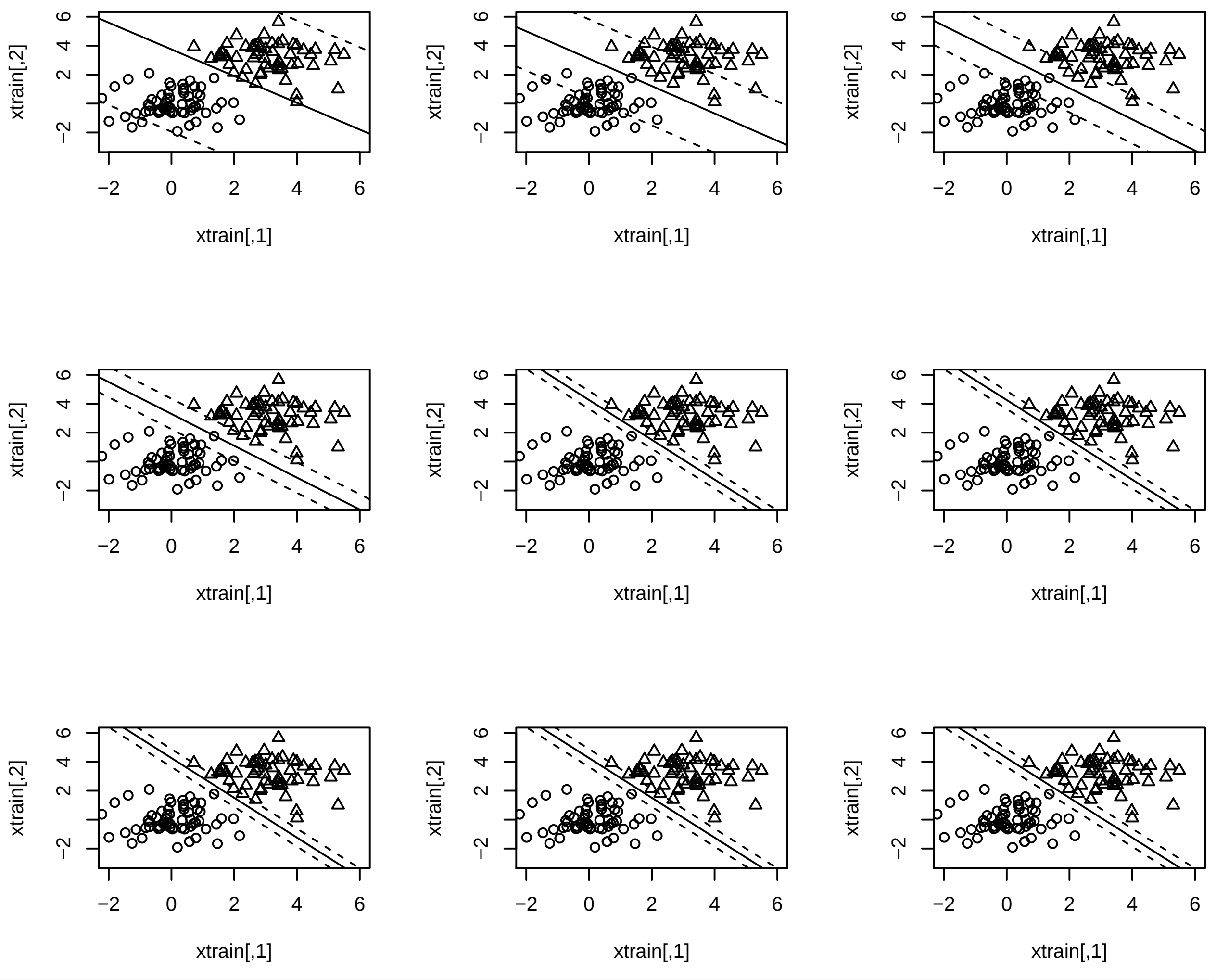
The C parameters balances the trade-off between having a large margin and separating the positive and unlabeled on the training set. It is important to choose it well to have good generalization.

Plot the decision functions of SVM trained on the toy examples for different values of C in the range $2^{seq(-10,14)}$.

```
cost = 2^(seq(-10, 14, by=3))
par(mfrow = c(3,3))
for (c in cost){
  svp = ksvm(xtrain, ytrain, type = "C-svc", kernel = "vanilladot", C=c, scaled=c())
```

```r
  plotlinearsvm(svp, xtrain, plot_legend = F)
}
```

```
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
```



```r
par(mfrow=c(1,1))
```

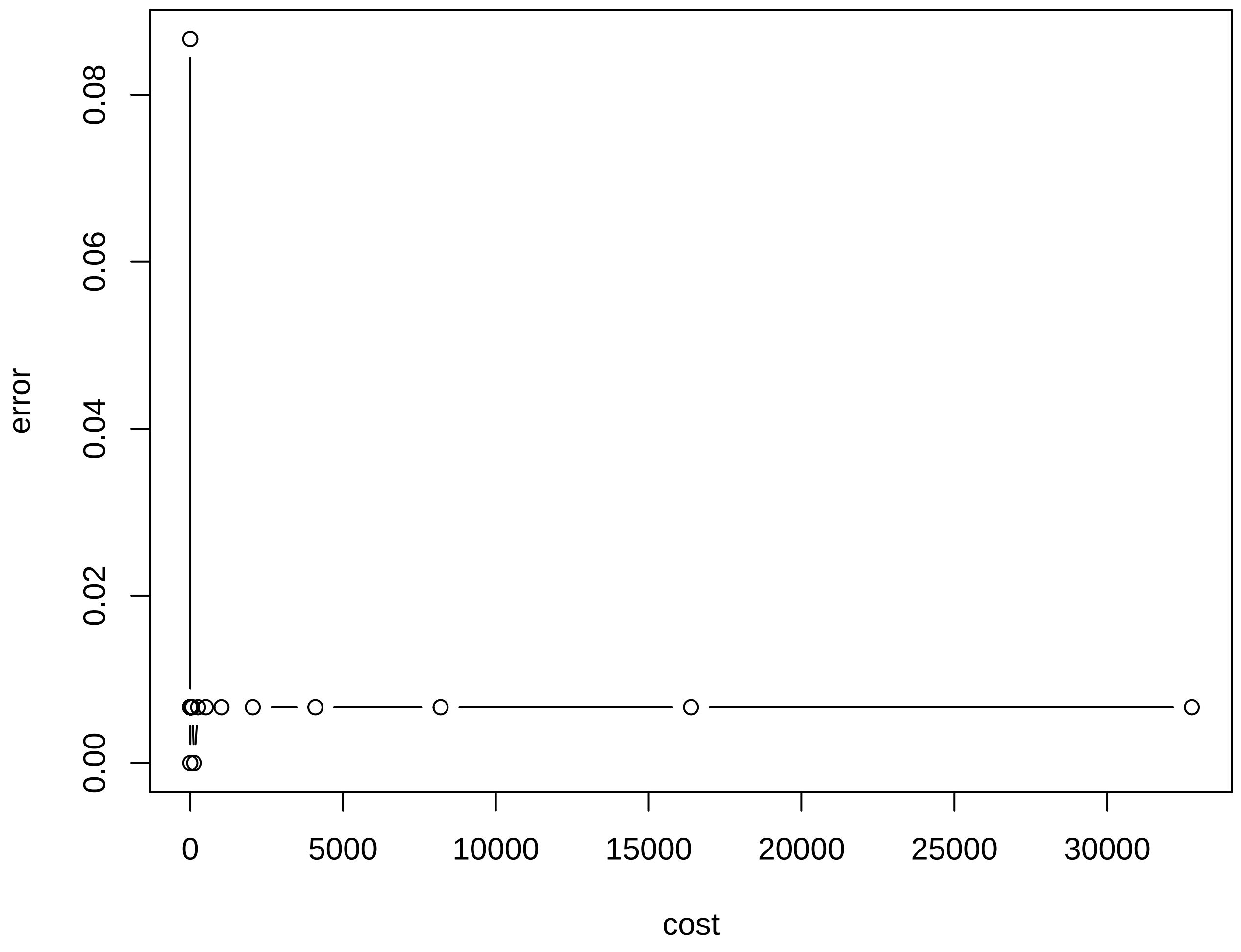
Plot the 5-fold cross-validation error as a function of C.

```r
cost = 2^(seq(-10, 15))
crossError = rep(NA, length(cost))
```

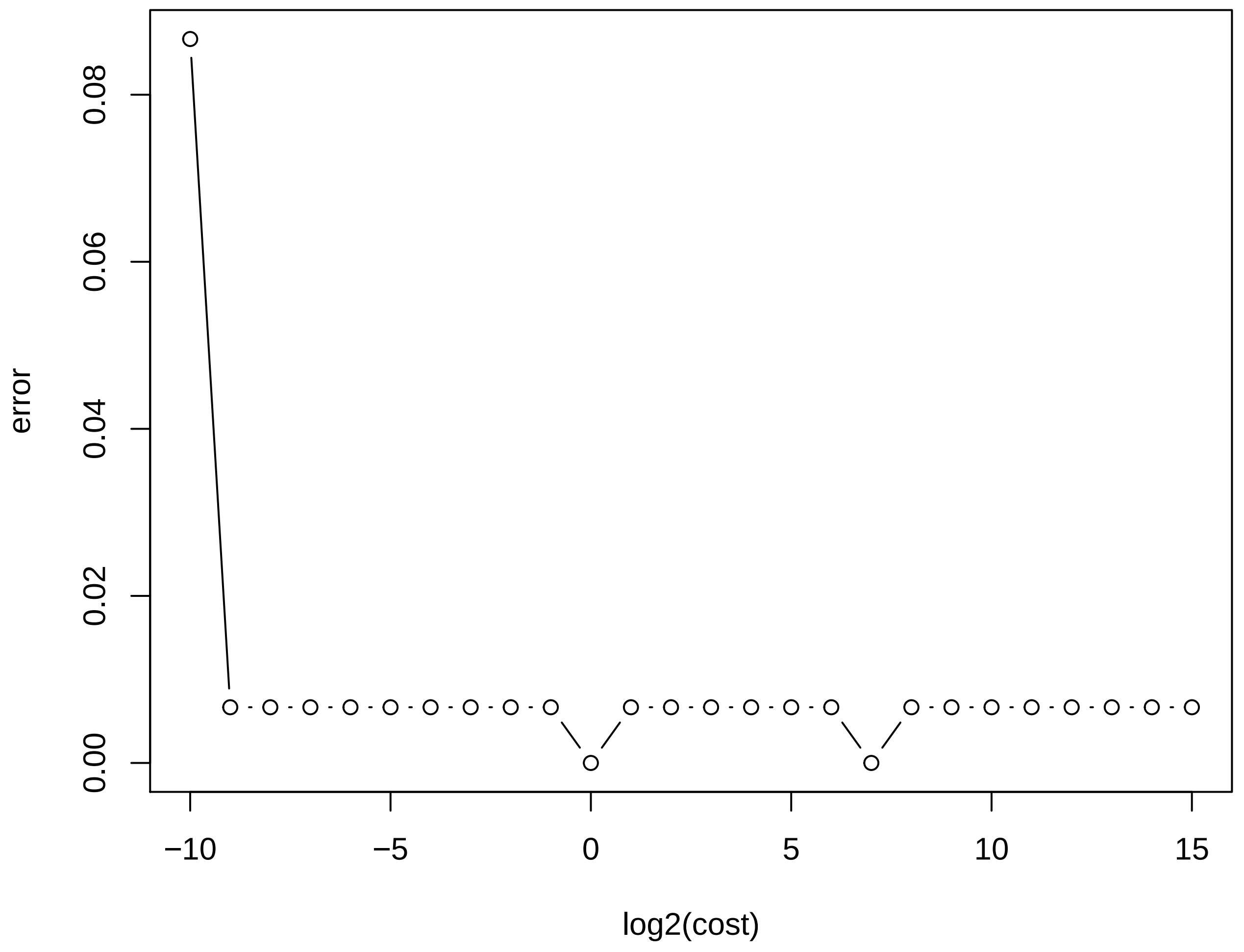
```r
error = sapply(cost, function(c){
  cross(ksvm(x, y, type = "C-svc", kernel = "vanilladot", C = c, scaled=c(), cross = 5))
})
```

```
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
```

```r
plot(cost, error, type='b')
```

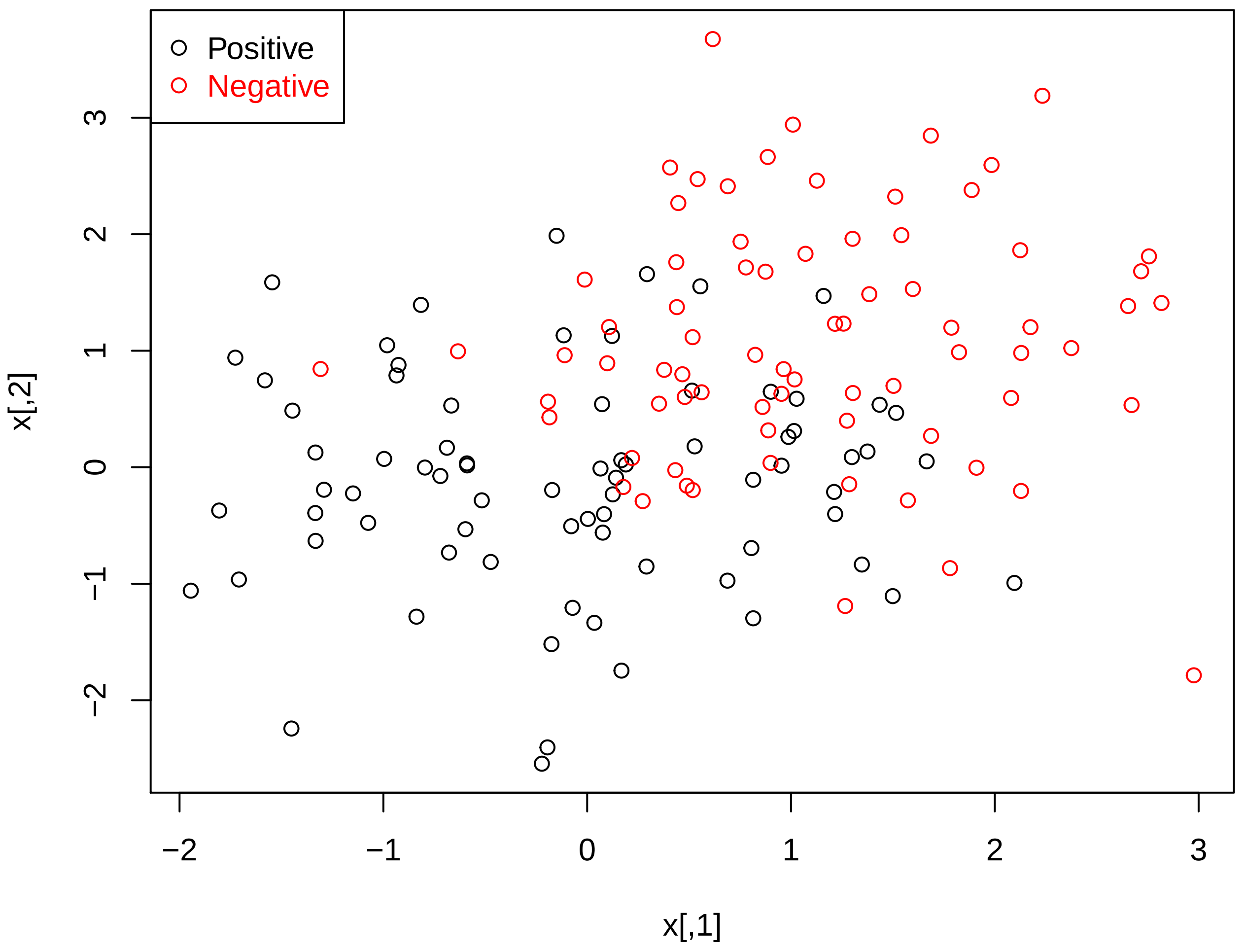


```r
plot(log2(cost), error, type='b')
```



Do the same on data with more overlap between the two classes, e.g., re-generate toy data with meanneg being 1.

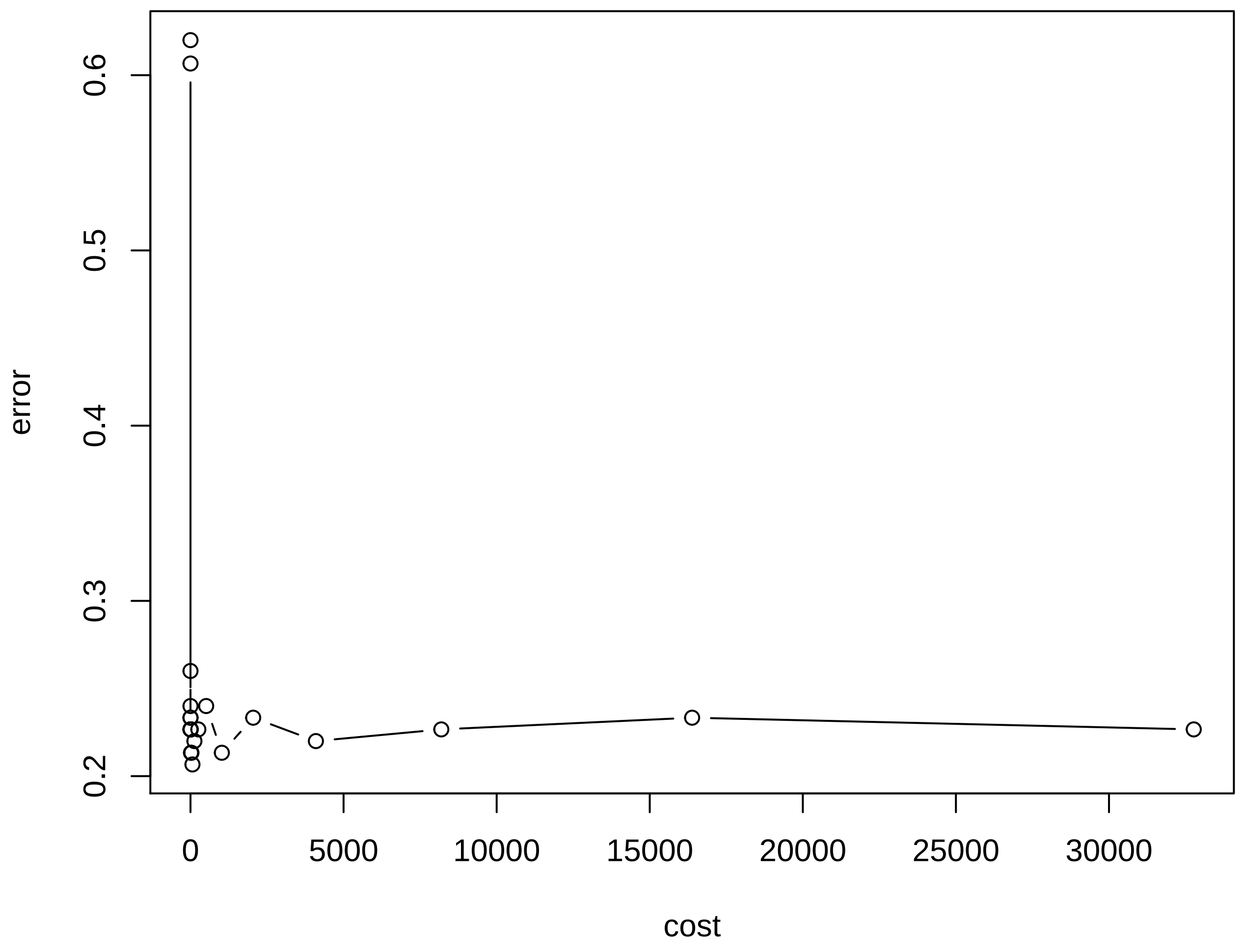
```r
n <- 150 #number of data points
p <- 2 # dimension
sigma <- 1 # variance of the distribution
meanpos <- 0 # centre of the distribution of positive examples
meanneg <- 1 # centre of the distribution of negative examples
npos <- round(n / 2) # number of positive examples
nneg <- n - npos # number of negative examples

# Generate the positive and negative examples
xpos <- matrix(rnorm(npos * p, mean = meanpos, sd = sigma), npos, p)
xneg <- matrix(rnorm(nneg * p, mean = meanneg, sd = sigma), npos, p)
x <- rbind(xpos, xneg)

# Generate the labels
y <- matrix(c(rep(1, npos), rep(-1, nneg)))

# Visualize the data
plot(x, col = ifelse(y > 0, 1, 2))
legend("topleft", c("Positive", "Negative"), col = seq(2), pch = 1, text.col = seq(2))
```



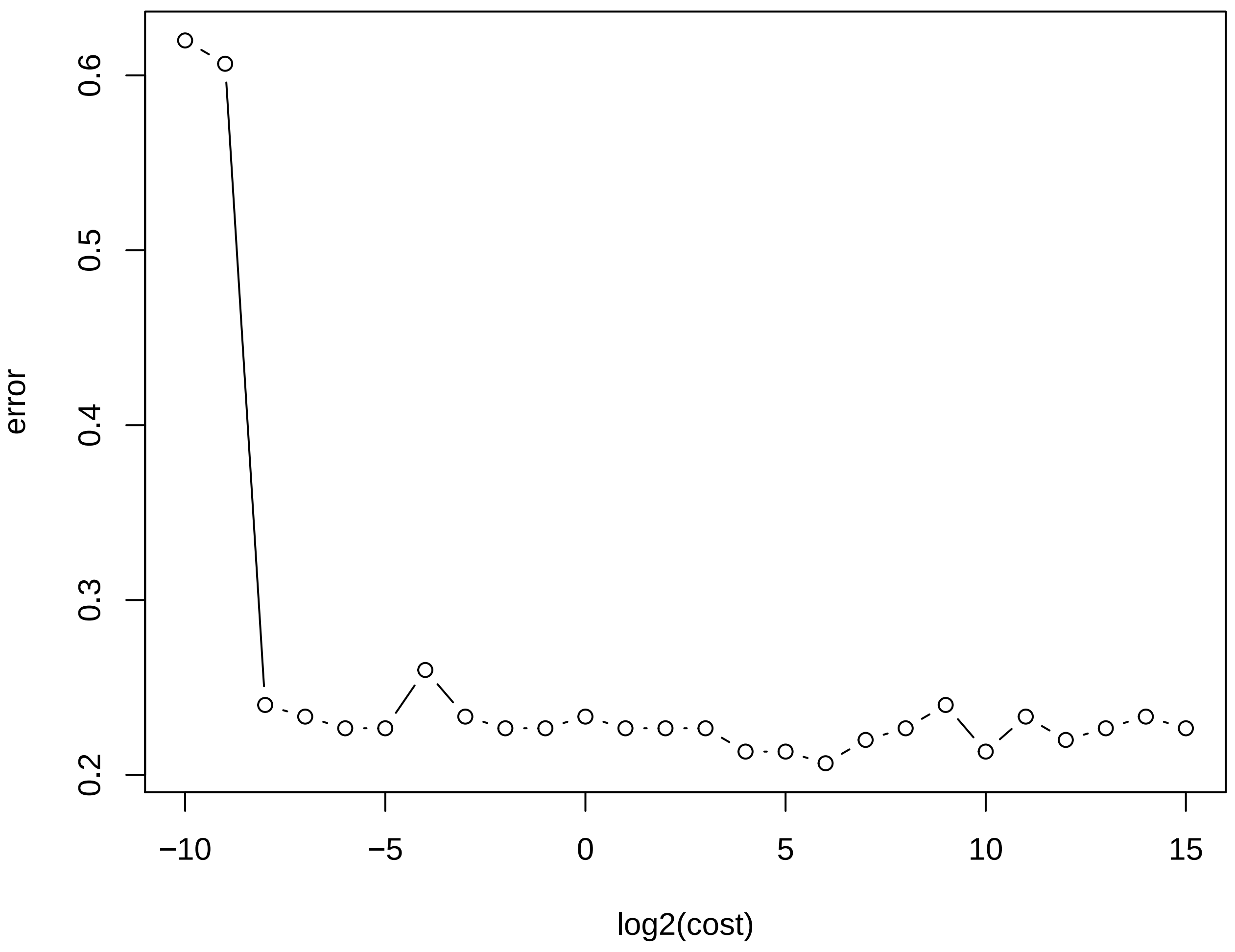
```r
# generate training/testing
ntrain <- round(n * 0.8) # number of training examples
tindex <- sample(n, ntrain) # indices of training samples
xtrain <- x[tindex, ]
xtest <- x[-tindex, ]
ytrain <- y[tindex]
ytest <- y[-tindex]
istrain <- rep(0, n)
istrain[tindex] <- 1


# cost cross validation
cost = 2^(seq(-10, 15))
error = sapply(cost, function(c){
  cross(ksvm(x, y, type = "C-svc", kernel = "vanilladot", C = c, scaled=c(),
             cross = 5))
})
```

```
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
## Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
```

```
plot(cost, error, type='b')
```



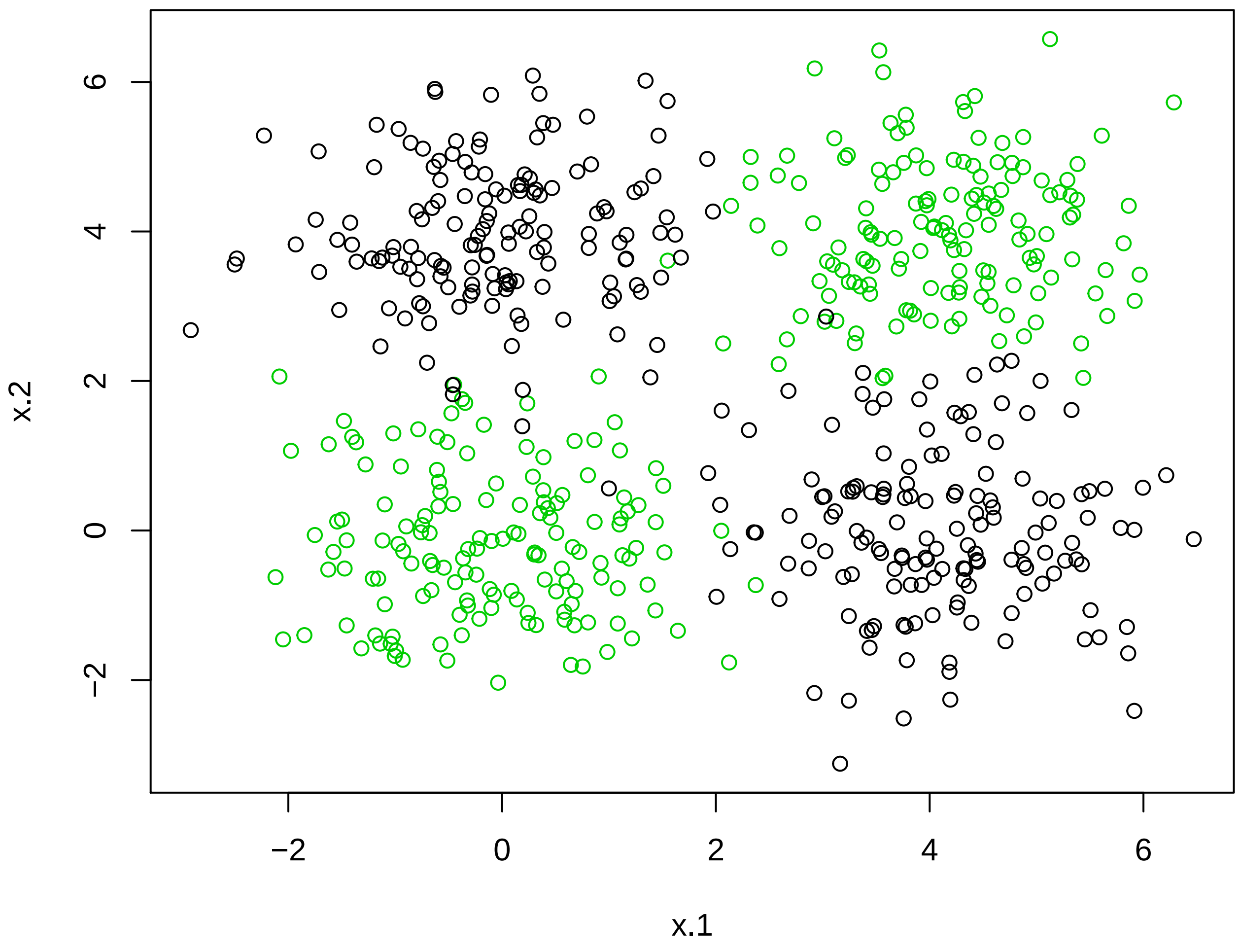
```r
plot(log2(cost), error, type='b')
```



3 Nonlinear SVM

Sometimes linear SVM are not enough. For example, generate a toy dataset where positive and negative examples are mixture of two Gaussians which are not linearly separable.

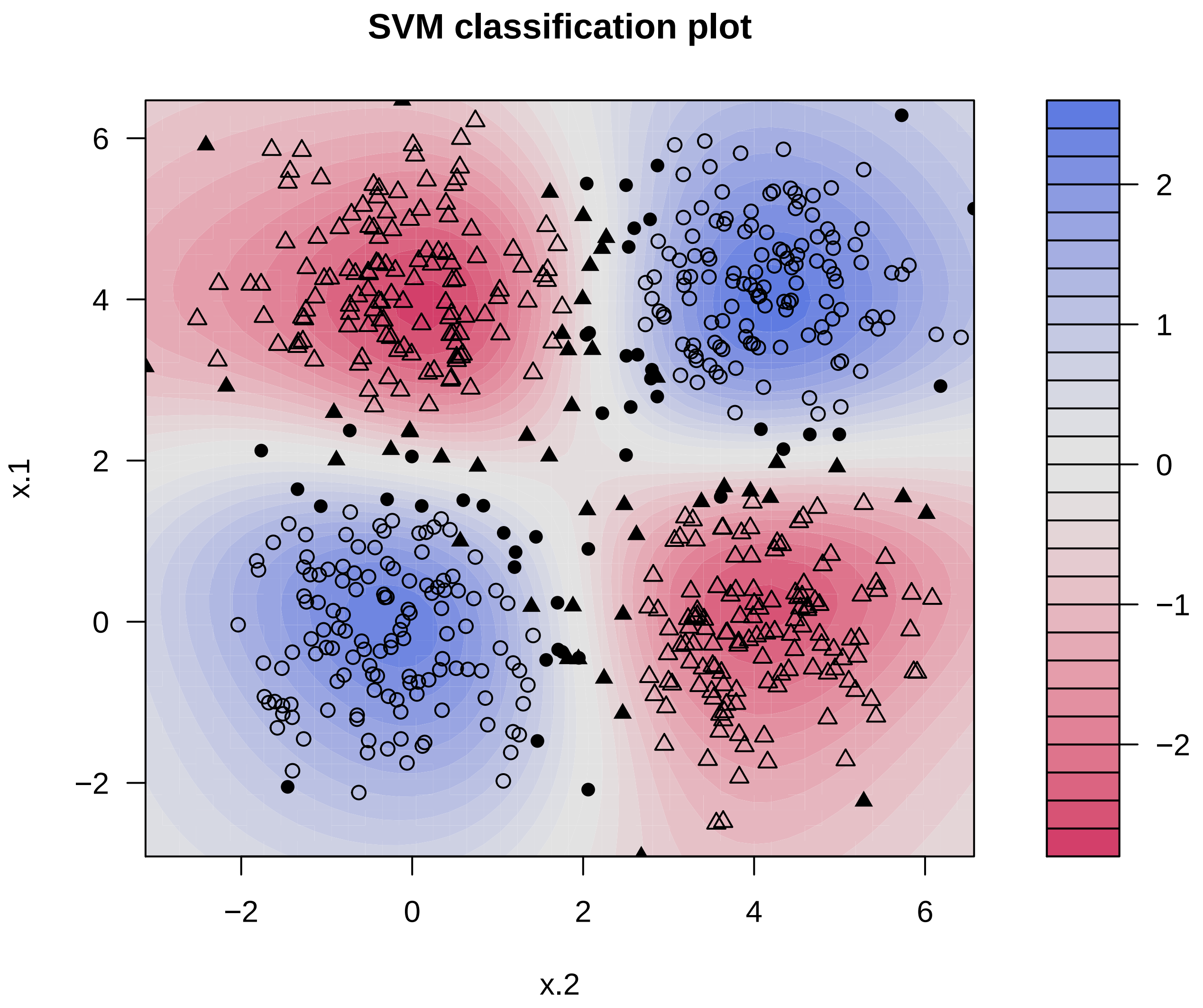
```r
RandomMatrix <- function( dist, n, p, ... ) {
  rs <- dist( n*p, ... )
  matrix( rs, n, p )
}

GenerateDatasetNonlinear <- function( n, p ) {
  bottom.left <- RandomMatrix( rnorm, n, p, mean=0, sd=1 )
  upper.right <- RandomMatrix( rnorm, n, p, mean=4, sd=1 )
  tmp1 <- RandomMatrix( rnorm, n, p, mean=0, sd=1 )
  tmp2 <- RandomMatrix( rnorm, n, p, mean=4, sd=1 )
  upper.left <- cbind( tmp1[,1], tmp2[,2] )
  bottom.right <- cbind( tmp2[,1], tmp1[,2] )
  y <- c( rep( 1, 2 * n ), rep( -1, 2 * n ) )
  idx.train <- sample( 4 * n, floor( 3.5 * n ) )
  is.train <- rep( 0, 4 * n )
  is.train[idx.train] <- 1
  data.frame( x=rbind( bottom.left, upper.right, upper.left, bottom.right ), y=y, train=is.train )
}

data = GenerateDatasetNonlinear(150, 2)
plot(data[,1:2], col = data[,3] + 2)
```



```r
x = as.matrix(data[,1:2])
y = matrix(data[,3])
```

To solve this problem, we should instead use a nonlinear SVM. This is obtained by simply changing the kernel parameter. For example, to use a Gaussian RBF kernel with $\sigma = 1$ and $C = 1$:

```r
# Train a nonlinear SVM
svp <- ksvm(x, y, type = "C-svc", kernel="rbf", kpar = list(sigma = 1), C = 1)

# Visualize it
plot(svp, data = x)
```



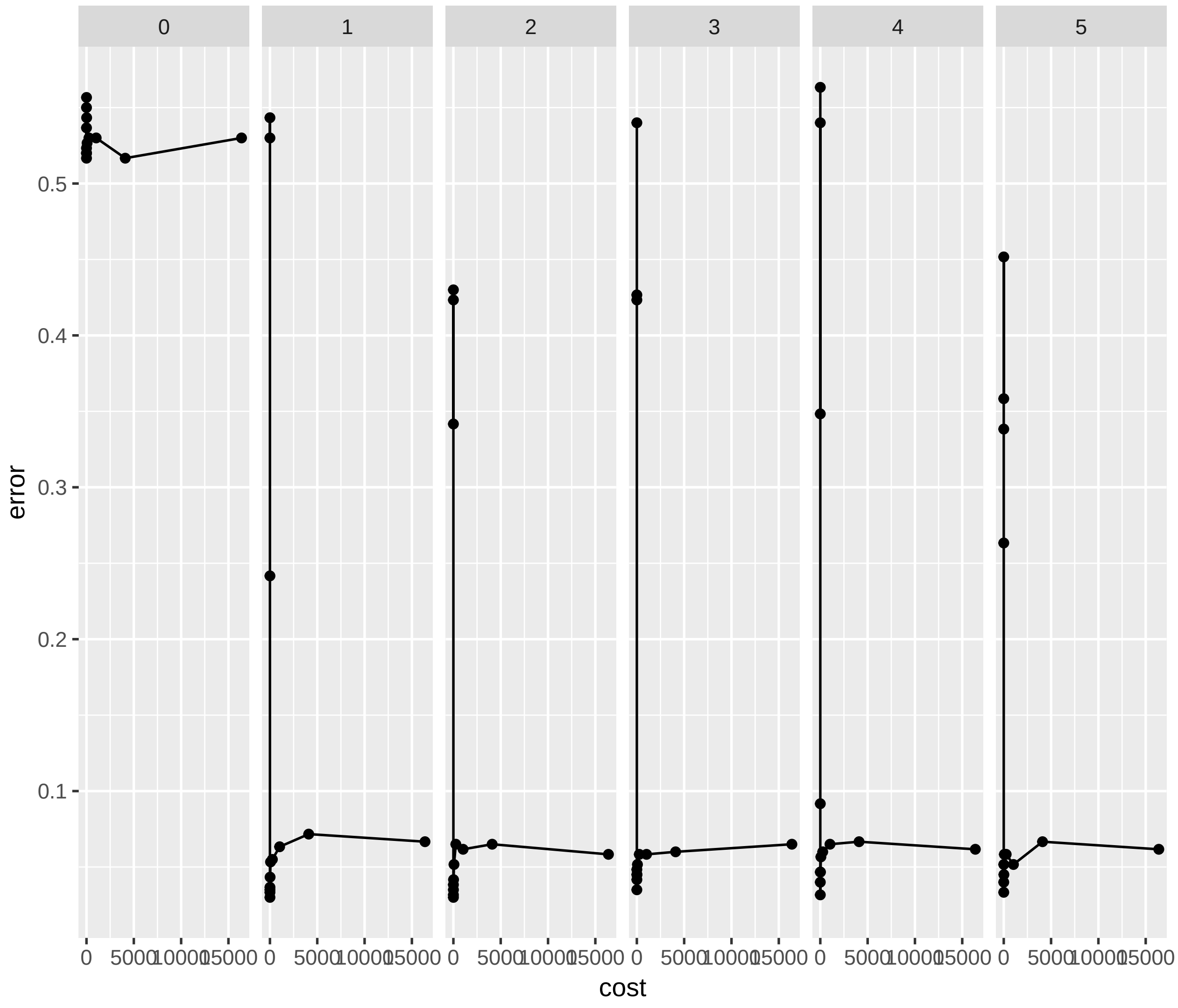
You should obtain something that look like Figure 3. Much better than the linear SVM, no? The nonlinear SVM has now two parameters: σ and C. Both play a role in the generalization capacity of the SVM.

Visualize and compute the 5-fold cross-validation error for different values of C and σ. Observe their influence.

```
library(ggplot2)
```

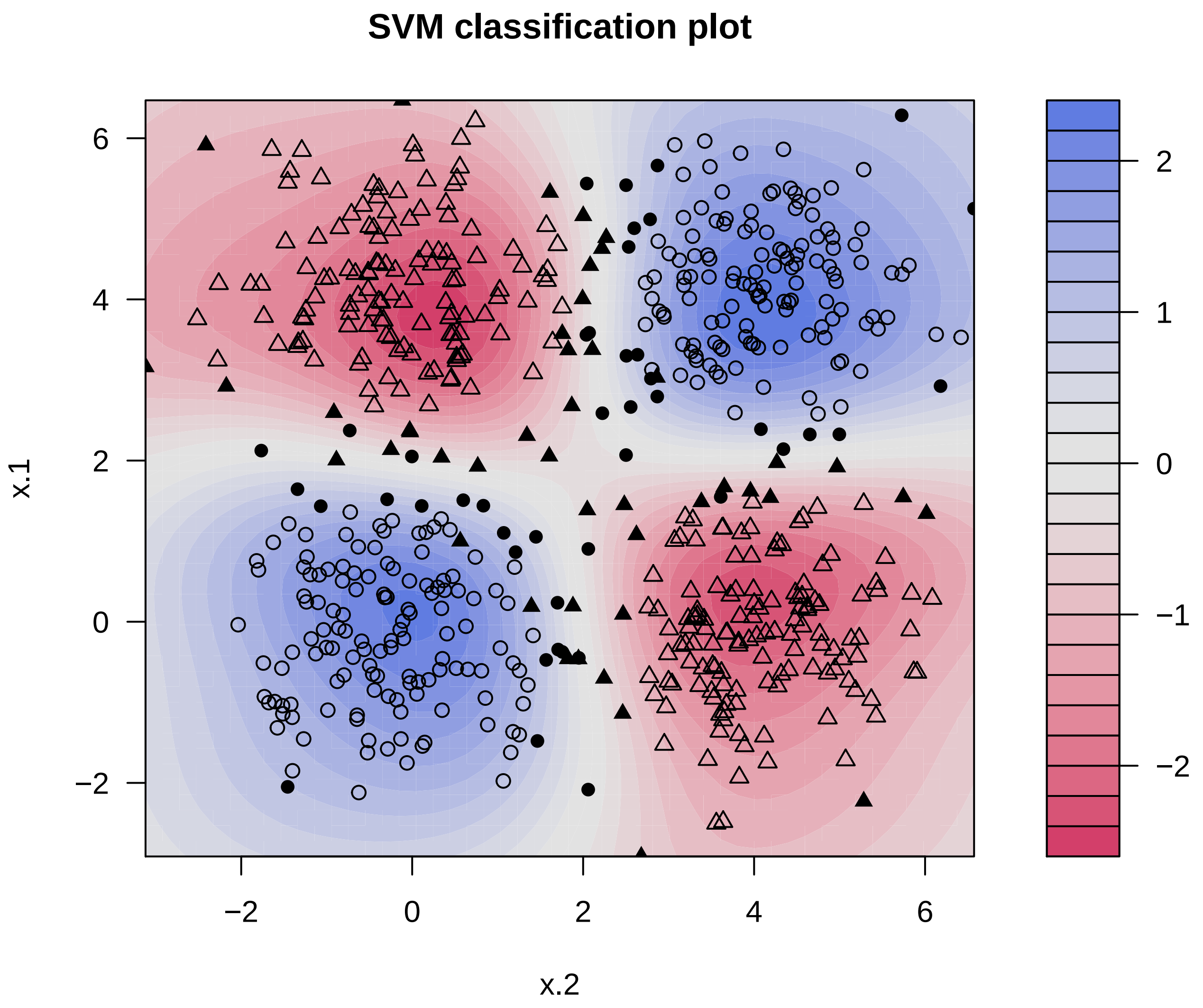
```
##
## Attaching package: 'ggplot2'

## The following object is masked from 'package:kernlab':
##
##     alpha
```

```
cost = 2^(seq(-10, 15, by=2))
sigma = 0:5
error = sapply(cost, function(c){
  sapply(sigma, function(s){
    cross(ksvm(x, y, type = "C-svc", kernel="rbf", kpar = list(sigma = s), C = c,
               scaled=c(), cross = 5))
  })
})
toPlotError = data.frame(sigma = rep(sigma, length(cost)),
                                   cost = rep(cost, each = length(sigma)),
                                   error = as.vector(error))

ggplot(data = toPlotError, aes(x=cost, y=error)) + geom_point() + geom_line() +
  facet_grid(.~sigma)
```



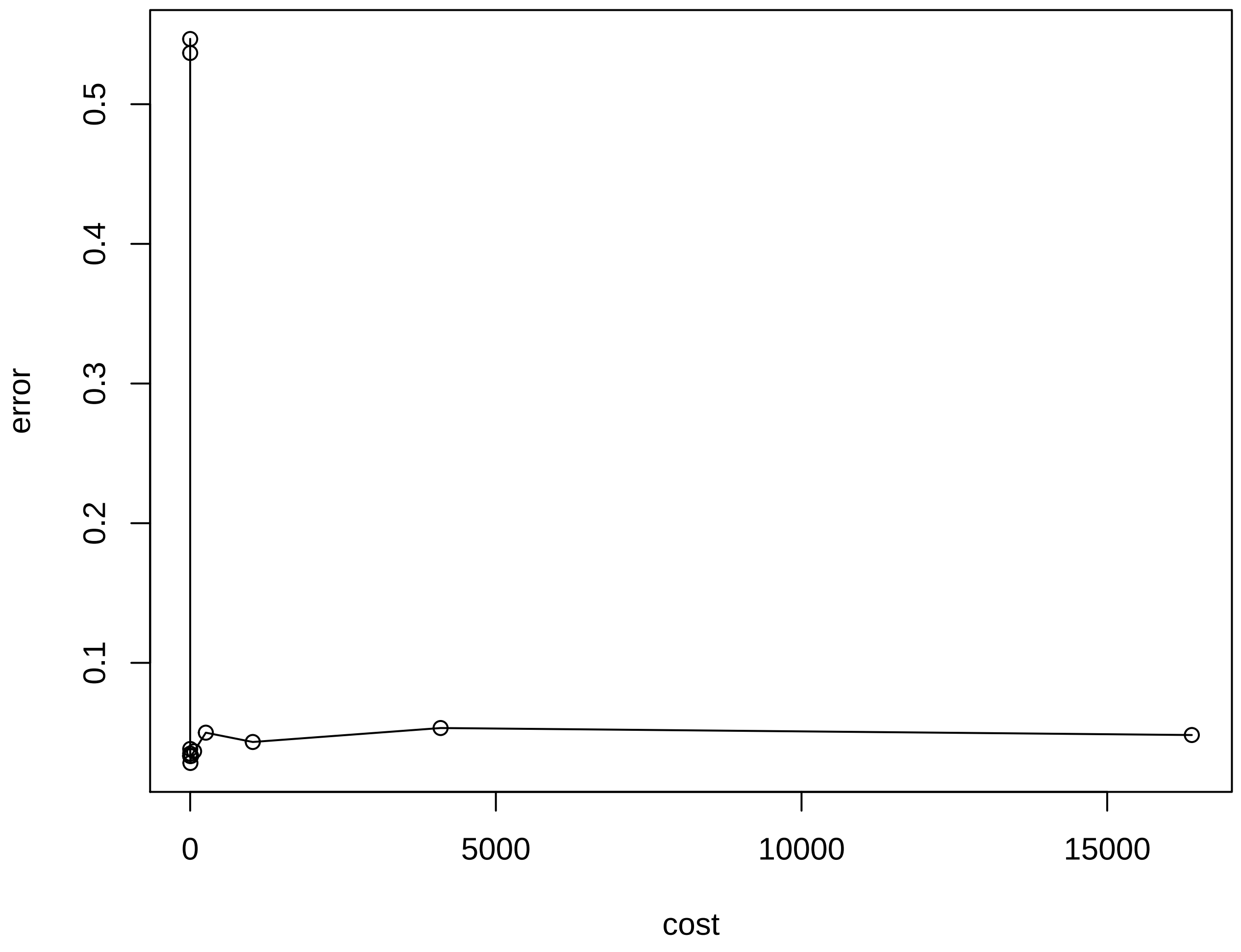
A useful heuristic to choose σ is implemented in kernlab. It is based on the quantiles of the distances between the training point.

```
# Train a nonlinear SVM with automatic selection of sigma by heuristic
svp <- ksvm(x, y, type = "C-svc", kernel = "rbf", C = 1)

# Visualize it
plot(svp, data = x)
```



Train a nonlinear SVM with various of C with automatic determination of σ. In fact, many other nonlinear kernels are implemented. Check the documentation of kernlab to see them: `?kernels`

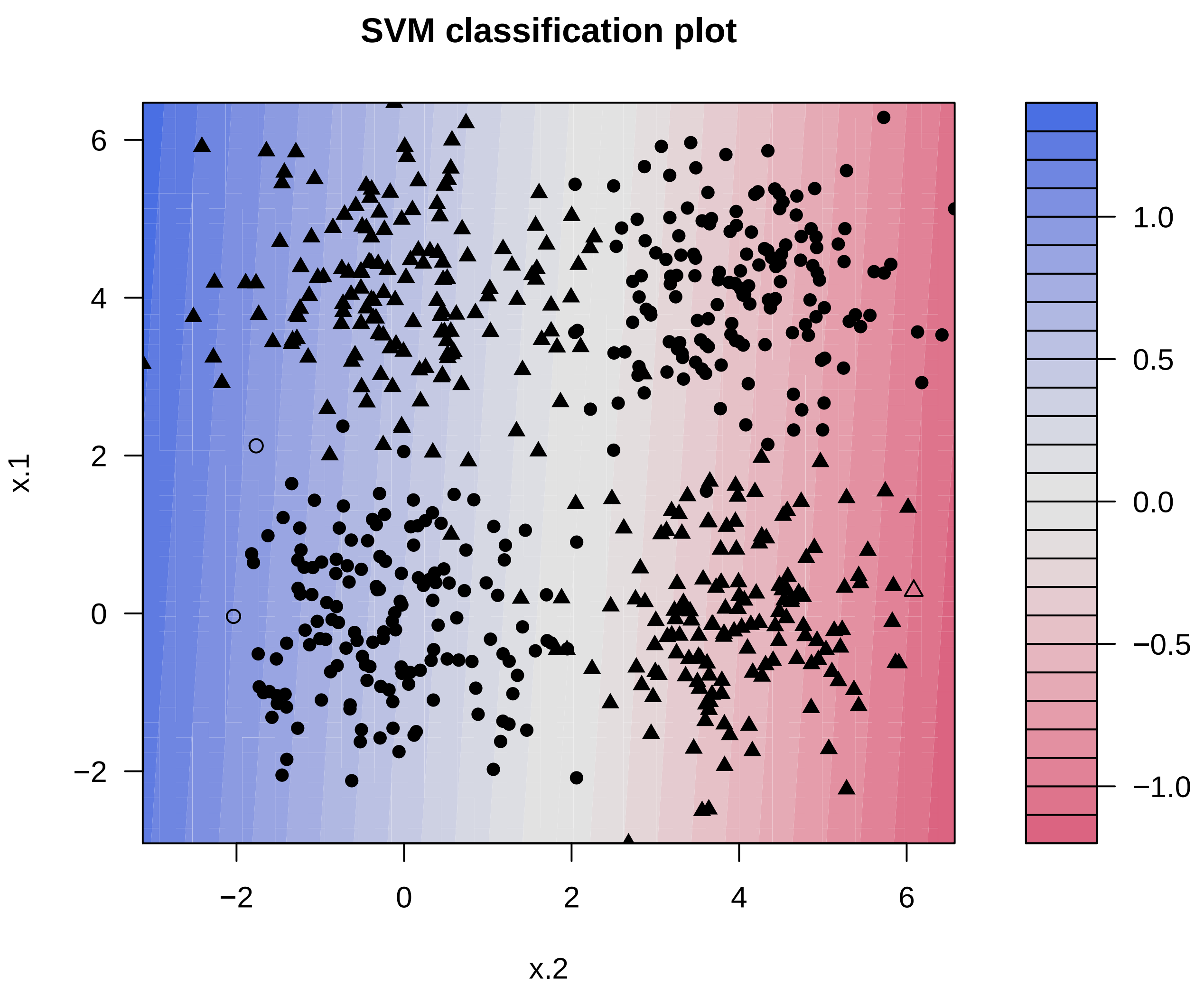
```r
library(ggplot2)
cost = 2^(seq(-10, 15, by=2))
error = sapply(1:length(cost), function(i){
    svp = ksvm(x, y, type = "C-svc", kernel = "rbf", C = cost[i], cross = 5)
    cross(svp)
})
plot(cost, error, type="o")
```



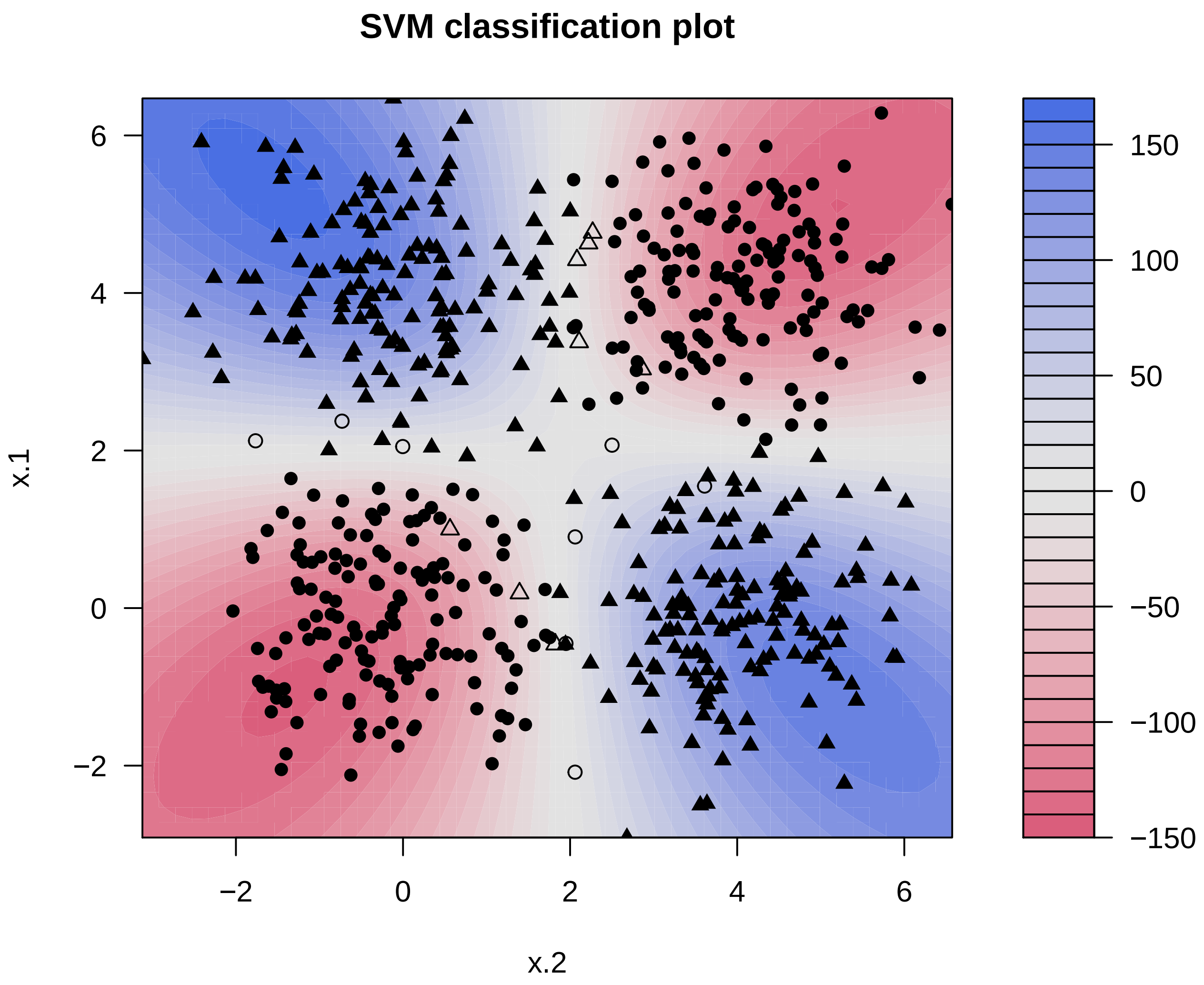
Test the polynomial, hyperbolic tangent, Laplacian, Bessel and ANOVA kernels on the toy examples.

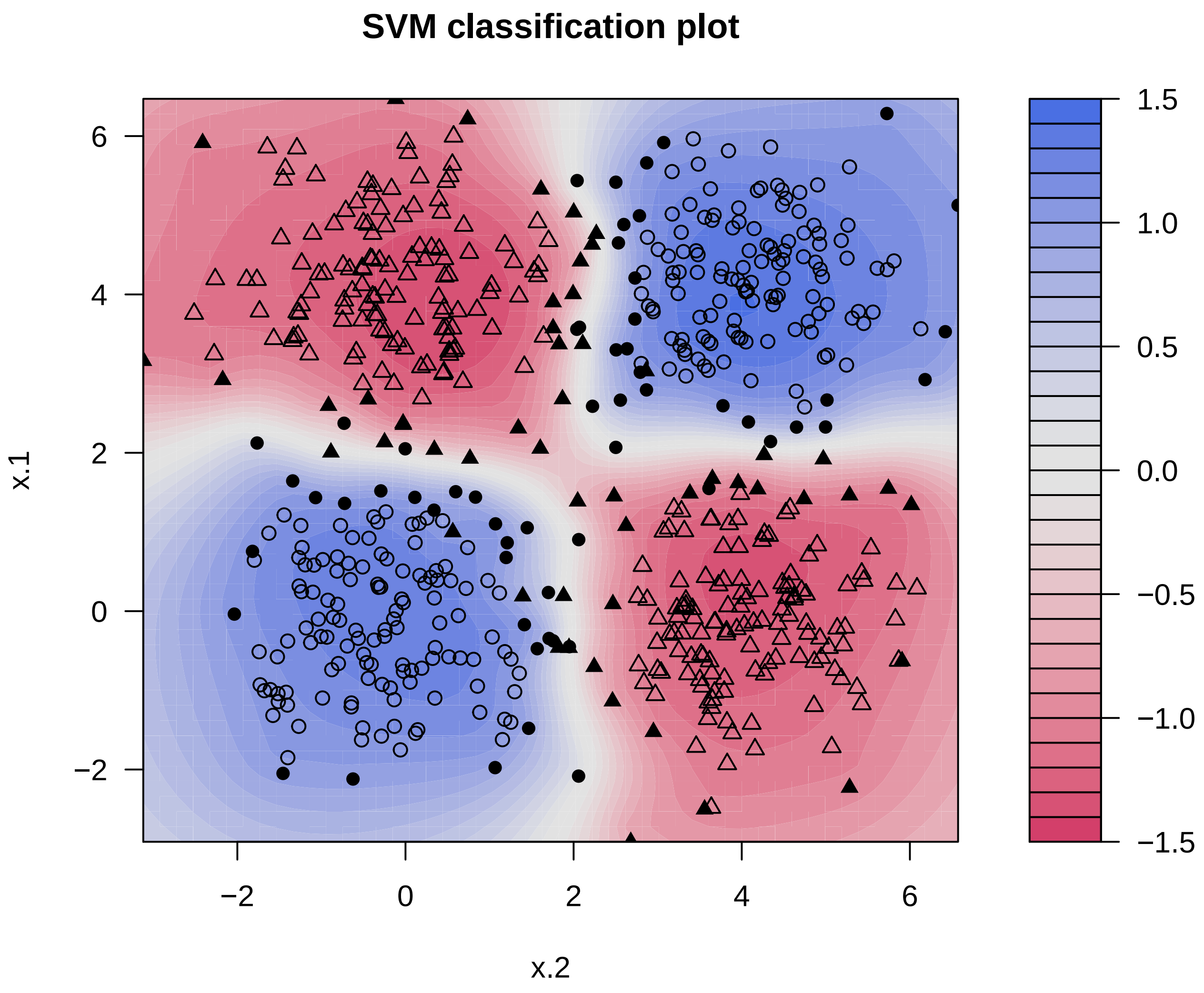
```
myKernels = c("polydot", "tanhdot", "laplacedot", "besseldot", "anovadot")
for (kernel in myKernels){
  plot(ksvm(x, y, type = "C-svc", kernel = kernel, C = 1), data=x)
}
```

```
##  Setting default kernel parameters
```

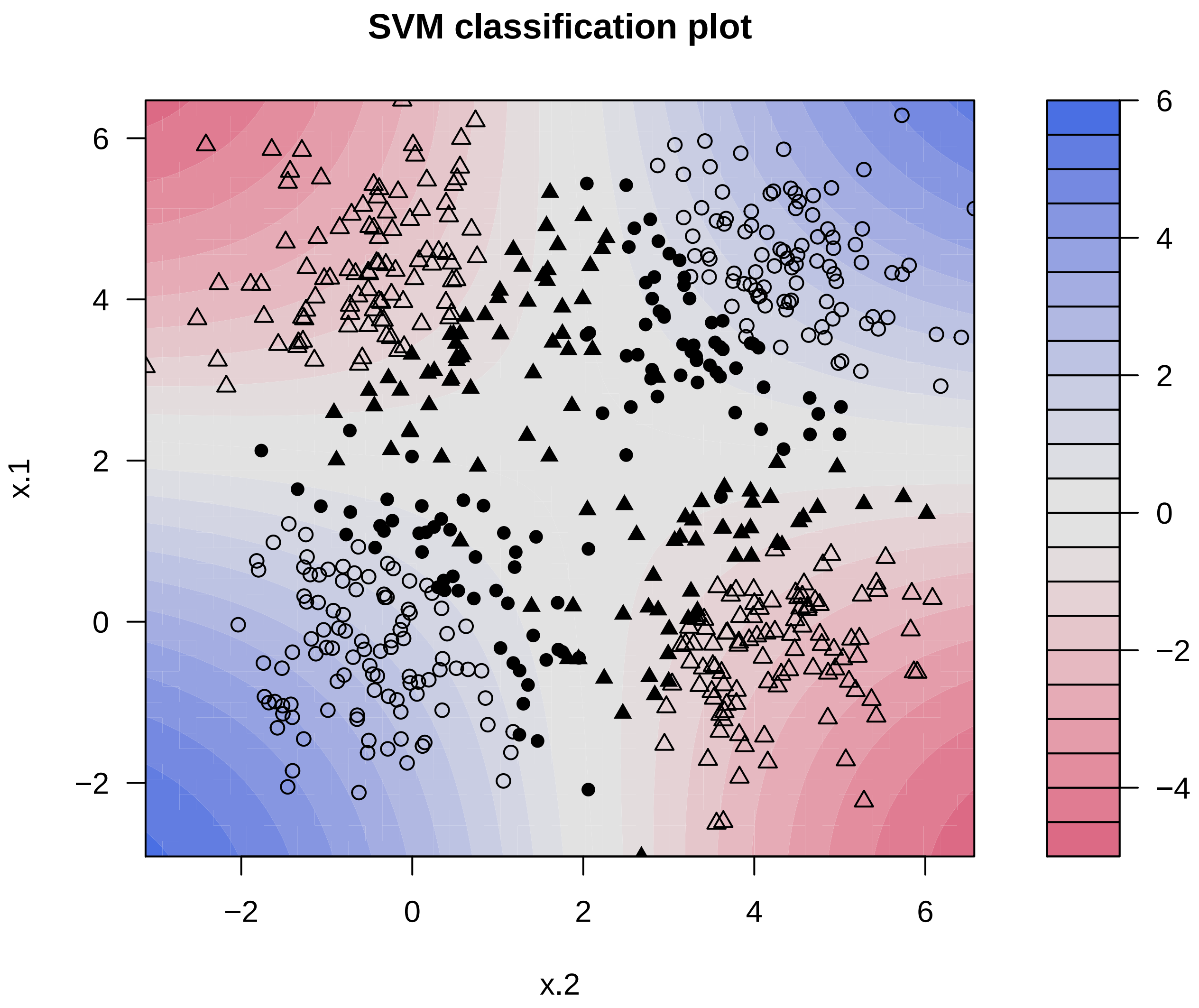


```
##  Setting default kernel parameters
```

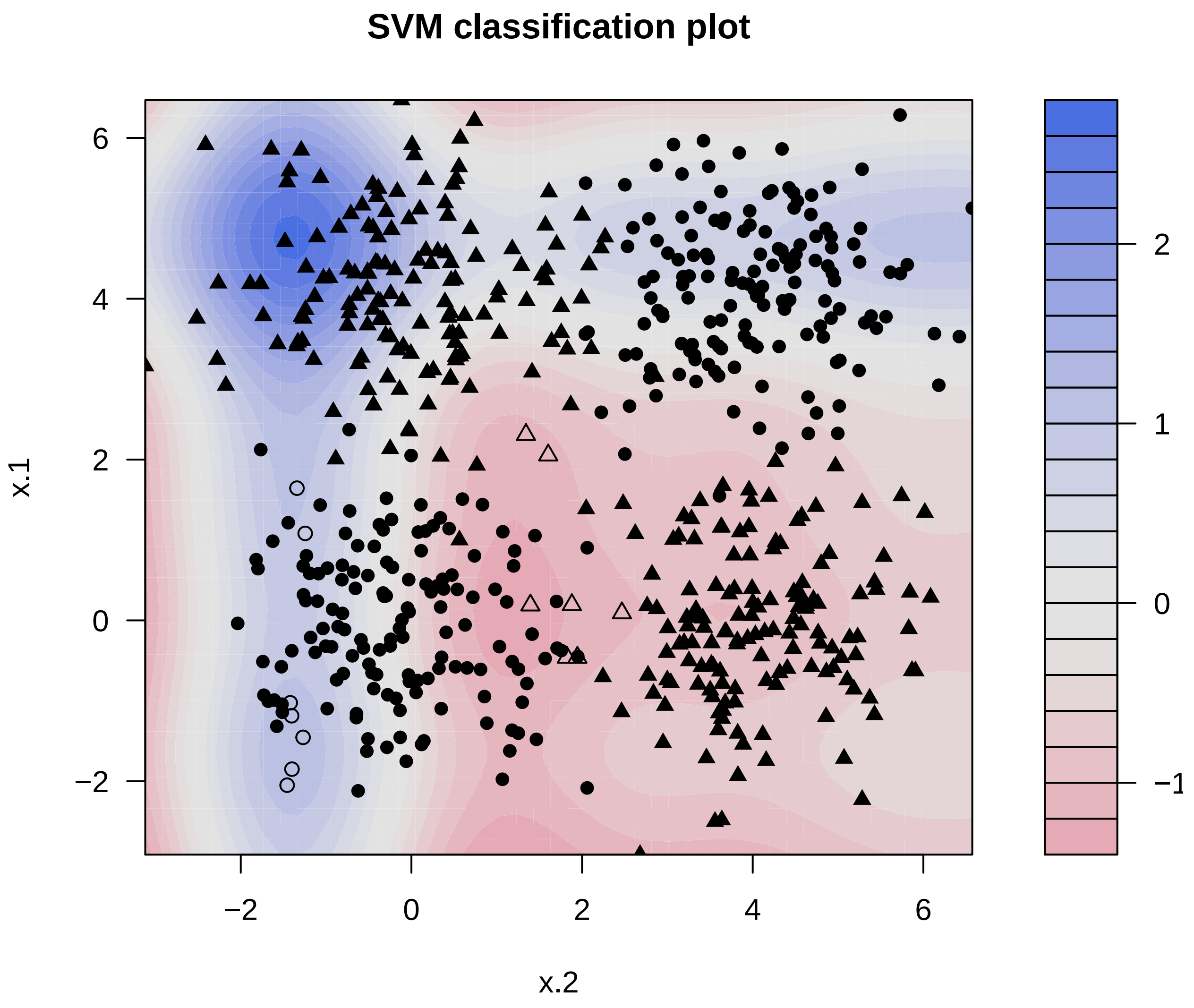

SVM classification plot
x.1
x.2
−2
0
2
4
6
150
100
50
0
−50
−100
−150



```
##  Setting default kernel parameters
```



```
##  Setting default kernel parameters
```


SVM classification plot
x.1
x.2
−2
0
2
4
6
−1
1

4 Application: cancer diagnosis from gene expression data

As a real-world application, let us test the ability of SVM to predict the class of a tumour from gene expression data. We use a publicly available dataset of gene expression data for 128 different individuals with acute lymphoblastic leukemia (ALL).

```
# To install this package run
#
# source("https://bioconductor.org/biocLite.R")
# biocLite("ALL")
#
#
# Load the ALL dataset
library(ALL)
```

```
## Loading required package: Biobase

## Loading required package: BiocGenerics

## Loading required package: parallel

##
## Attaching package: 'BiocGenerics'

## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB

## The following objects are masked from 'package:stats':
##
##     IQR, mad, xtabs

## The following objects are masked from 'package:base':
##
##     anyDuplicated, append, as.data.frame, cbind, colnames,
##     do.call, duplicated, eval, evalq, Filter, Find, get, grep,
##     grepl, intersect, is.unsorted, lapply, lengths, Map, mapply,
##     match, mget, order, paste, pmax, pmax.int, pmin, pmin.int,
##     Position, rank, rbind, Reduce, rownames, sapply, setdiff,
##     sort, table, tapply, union, unique, unsplit, which, which.max,
##     which.min

## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with
##     'browseVignettes()'. To cite Bioconductor, see
##     'citation("Biobase")', and for packages 'citation("pkgname")'.
```

```
data(ALL)

# Inspect them
?ALL
show(ALL)
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 12625 features, 128 samples
##   element names: exprs
## protocolData: none
## phenoData
##   sampleNames: 01005 01010 ... LAL4 (128 total)
##   varLabels: cod diagnosis ... date last seen (21 total)
##   varMetadata: labelDescription
## featureData: none
## experimentData: use 'experimentData(object)'
##   pubMedIds: 14684422 16243790
## Annotation: hgu95av2
```

```r
print(summary(pData(ALL)))
```

```
##      cod              diagnosis            sex          age
##  Length:128         Length:128         F   :42   Min.   : 5.00
##  Class :character   Class :character   M   :83   1st Qu.:19.00
##  Mode  :character   Mode  :character   NA's: 3   Median :29.00
##                                                  Mean   :32.37
##                                                  3rd Qu.:45.50
##                                                  Max.   :58.00
##                                                  NA's   :5
##        BT     remission      CR              date.cr
##  B2     :36   CR  :99   Length:128         Length:128
##  B3     :23   REF :15   Class :character   Class :character
##  B1     :19   NA's:14   Mode  :character   Mode  :character
##  T2     :15
##  B4     :12
##  T3     :10
##  (Other):13
##   t(4;11)         t(9;22)        cyto.normal        citog
##  Mode :logical   Mode :logical   Mode :logical   Length:128
##  FALSE:86        FALSE:67        FALSE:69        Class :character
##  TRUE :7         TRUE :26        TRUE :24        Mode  :character
##  NA's :35        NA's :35        NA's :35
##
##
##
##      mol.biol    fusion protein   mdr          kinet       ccr
##  ALL1/AF4:10   p190     :17     NEG :101   dyploid:94   Mode :logical
##  BCR/ABL :37   p190/p210: 8     POS : 24   hyperd.:27   FALSE:74
##  E2A/PBX1: 5   p210     : 8     NA's:  3   NA's   : 7   TRUE :26
##  NEG     :74   NA's     :95                             NA's :28
##  NUP-98  : 1
##  p15/p16 : 1
##
##   relapse        transplant          f.u            date last seen
##  Mode :logical   Mode :logical   Length:128         Length:128
##  FALSE:35        FALSE:91        Class :character   Class :character
##  TRUE :65        TRUE :9         Mode  :character   Mode  :character
##  NA's :28        NA's :28
##
##
##
```

Here we focus on predicting the type of the disease (B-cell or T-cell). We get the expression data and disease type as follows

```r
x <- t(exprs(ALL))
y <- substr(ALL$BT,1,1)
```

Test the ability of a SVM to predict the class of the disease from gene expression. Check the influence of the parameters.

```r
# You actually need to play a lot with the parameters: kernels, costs, ...

x <- t(exprs(ALL))
y <- substr(ALL$BT,1,1)

# train and test sets
n = length(y)
ntrain <- round(n * 0.8) # number of training examples
tindex <- sample(n, ntrain) # indices of training samples
xtrain <- x[tindex, ]
xtest <- x[-tindex, ]
ytrain <- y[tindex]
ytest <- y[-tindex]

# train svm on train set
svp = ksvm(xtrain, ytrain, type = "C-svc", kernel = "rbf", C = 10)

# predict on test set
(pred = predict(svp, xtest))
```

```
##  [1] B B B B B B B B B B B B B B B B B B T T B T B T T T
## Levels: B T
```

```r
(acc = sum(as.vector(pred) == ytest) / length(ytest))
```

```
## [1] 0.9230769
```

Finally, we may want to predict the type and stage of the diseases. We are then confronted with a multi-class classification problem, since the variable to predict can take more than two values:

```r
y <- ALL$BT
print(y)
```

```
##   [1] B2 B2 B4 B1 B2 B1 B1 B1 B2 B2 B3 B3 B3 B2 B3 B  B2 B3 B2 B3 B2 B2 B2
##  [24] B1 B1 B2 B1 B2 B1 B2 B  B  B2 B2 B2 B1 B2 B2 B2 B2 B2 B4 B4 B2 B2 B2
##  [47] B4 B2 B1 B2 B2 B3 B4 B3 B3 B3 B4 B3 B3 B1 B1 B1 B1 B3 B3 B3 B3 B3 B3
##  [70] B3 B3 B1 B3 B1 B4 B2 B2 B1 B3 B4 B4 B2 B2 B3 B4 B4 B4 B1 B2 B2 B2 B1
##  [93] B2 B  B  T  T3 T2 T2 T3 T2 T  T4 T2 T3 T3 T  T2 T3 T2 T2 T2 T1 T4 T
## [116] T2 T3 T2 T2 T2 T2 T3 T3 T3 T2 T3 T2 T
## Levels: B B1 B2 B3 B4 T T1 T2 T3 T4
```

Fortunately, kernlab implements automatically multi-class SVM by an all-versus-all strategy to combine several binary SVM.

Test the ability of a SVM to predict the class and the stage of the disease from gene expression.

```r
# now y is multiclass
x <- t(exprs(ALL))
y <- ALL$BT

# train and test sets (same as previous question)
n = length(y)
```

```r
ntrain <- round(n * 0.8) # number of training examples
tindex <- sample(n, ntrain) # indices of training samples
xtrain <- x[tindex, ]
xtest <- x[-tindex, ]
ytrain <- y[tindex]
ytest <- y[-tindex]

# train svm on train set
# type of svm is now able to handle multiclass
svp = ksvm(xtrain, ytrain, type = "kbb-svc", kernel = "rbf", C = 10)

# predict on test set
(pred = predict(svp, xtest))
```

```
##  [1] T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1
## [24] T1 T1 T1
## Levels: B B1 B2 B3 B4 T T1 T2 T3 T4
```

```r
(acc = sum(as.vector(pred) == ytest) / length(ytest))
```

```
## [1] 0.03846154
```