Support Vector Machines

Laura Cline

06/11/2021

Contents

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

```
rm(list = ls(all=TRUE))
```

```
libs <- c("tidyverse", "ISLR", "modelr", "e1071", "ROCR")
invisible(lapply(libs, library, character.only=TRUE))
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.0     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

This lab primarily focuses on creating and tweaking a variety of Support Vector Classifiers using the `e1071` library. The text mentions the avaliability of the `LiblineaR` library for very large linear problems, but we will not use it in this lab. We conclude the lab with a discussion/implementation of ROC curves using the `ROCR` library. The topics that will be covered in this lab include:

- Support Vector Classifiers
- Support Vector Machines (SVM)
- ROC Curves
- SVMs with K > 2 Classes

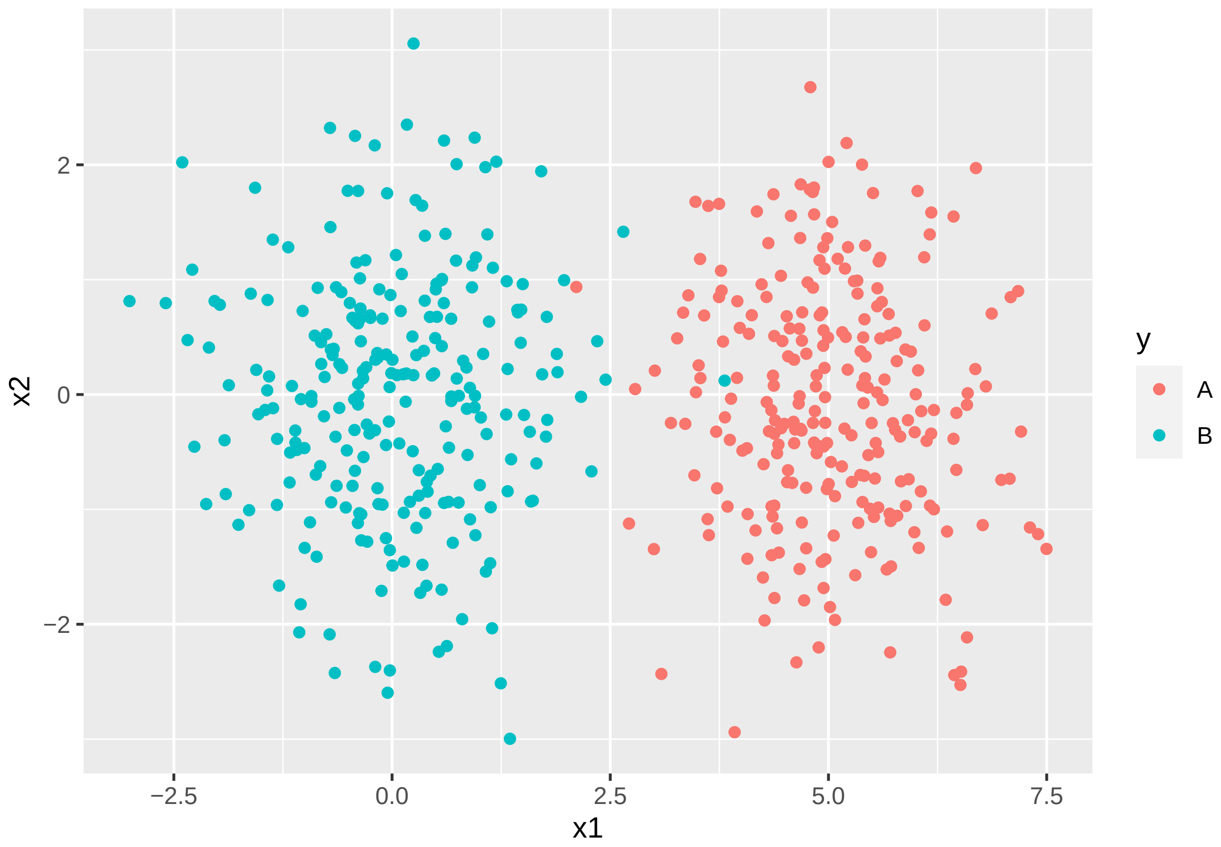
Support Vector Classifier

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. 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 biolate 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 up the toy data to work with
set.seed(1)

dat <- tibble(
  x1 = rnorm(500),
  x2 = rnorm(500),
  y = factor(c(rep('A', 250), (rep('B',250))))
)
```

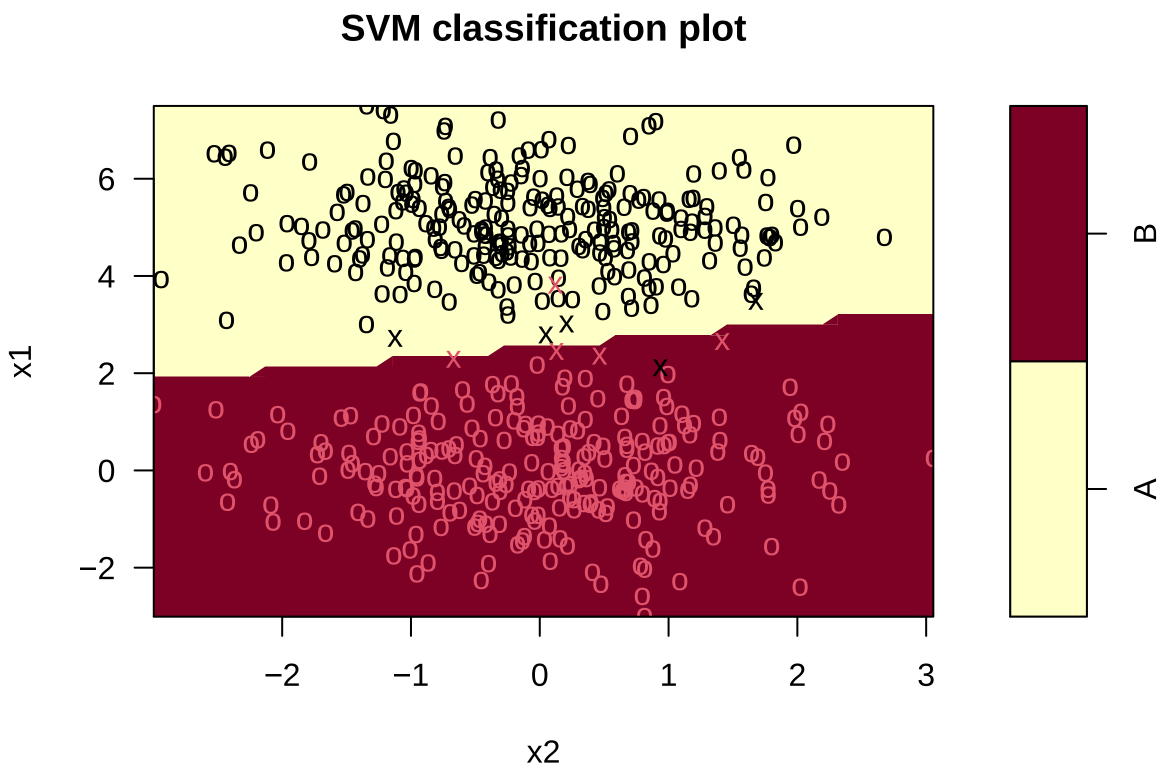
```r
# Add a little separability between the classes
dat$x1[dat$y == 'A'] <- dat$x1[dat$y == 'A'] + 5

# Check to see if classes are linearly seperable
ggplot(dat, aes(x1, x2, col=y)) +
  geom_jitter()
```



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 vector.

Now that we've created our toy data and can clearly see they are not linearly seperable, let's fit the support vector classifier. Note that we need to make sure that `y` is encoded as a factor variable in our formula. We set the `cost` argument to 10 in this example as well (default is 1).

```
svc <- svm(y ~ ., data=dat, kernel='linear',
           cost = 10, scale = FALSE)
plot(svc, dat)
```



```
summary(svc)
```

```
##
## 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:  10
##
##  ( 5 5 )
##
##
## Number of Classes:  2
##
## Levels:
##  A B
```

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

We can plot the support vector classifier obtained.

Note that the two arguments in the `plot.svm()` 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 red, and region that will be assigned the +1 class is shown in yellow. 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 the library the decision boundary looks somewhat jagged in the plot. 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 behaviour of the usual `plot()` function in `R`. The support vectors are plotted as crosses and the remaining observations are plotted as circles; we see here that there are ten support vectors. We can determine their identities as follows:

```
svc$index
```

```
##  [1]  14  24  84 205 232 274 485 486 492 495
```

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

This tells us that a linear kernel was used with `cost=10`, and that there are 10 support vectors, 5 in one class and 5 in the other.

Our support vector classifier works pretty well (as it shou,d the data was made to be very easily separated). Upon closer inspection, we see that there are 10 support vectors, 5 from each class. You will notice that these 10 support vectors are plotted as crosses on the above graph, and everything is left as an open circle.

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

With a smaller cost parameter, our decision boundary gets a little more rigid and will allow for fewer errors in the training set. In sort - smaller margins, fewer support vectors.

```
svc2 <- svm(y ~ ., data=dat, kernel = 'linear',
            cost = 1, scale = FALSE)

summary(svc2)
```

```
##
## 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:  13
##
##  ( 7 6 )
##
##
## Number of Classes:  2
##
## Levels:
##  A B
```

Note 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 witha linear kernel, using a range of values of the `cost` parameter.

How do we determine the best cost parameter to use? You should know by now that cross-validation is pretty much the answer to every parameter-tuning problem. Luckily, the `e1071` library contains the `tune()` function, which will allow us to perform 10-fold cross-validation over a range of cost parameters.

```r
svc_cv <- tune(svm, y ~ ., data=dat, kernel='linear',
               ranges = list(cost = c(0.001, 0.01, 1, 5, 10, 100)))
```

`tune()` returns an object of class `tune`, which contains results from the 10-fold CV, including the best performing parameter values and their respective errors (`best.parameters`, `best.performance`), a data frame of all parameter combinations and their corresponding results (`performances`).

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

We'll take a glance at these components and check the results of our best-performing cross-validated model.

```r
summary(svc_cv)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##     5
##
## - best performance: 0.004
##
## - Detailed performance results:
##    cost error dispersion
## 1 1e-03 0.194 0.13368288
## 2 1e-02 0.006 0.01349897
## 3 1e+00 0.006 0.01349897
## 4 5e+00 0.004 0.00843274
## 5 1e+01 0.006 0.01349897
## 6 1e+02 0.006 0.01349897
```

We can see that `cost = 5` results in the lowest cross-validation error rate. The `tune()` function stores the best model obtained, which can be accessed as follows:

```r
best <- svc_cv$best.model
summary(best)
```

```
##
## Call:
## best.tune(method = svm, train.x = y ~ ., data = dat, ranges = list(cost = c(0.001,
##     0.01, 1, 5, 10, 100)), kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  5
##
## Number of Support Vectors:  14
##
##  ( 7 7 )
##
```

```
##
## Number of Classes:  2
##
## Levels:
##  A B
```

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 the test dataset.

Now that we have a working model, let's generate some test data. Keep in mind that we are making our own data to have a pretty well-defined class boundary, so our test error should be practically nil (and it is, as you'll see).

```r
# Change the seed so we don't get the exact same data
set.seed(2)

test <- tibble(
  x1 = rnorm(500),
  x2 = rnorm(500),
  y = factor(c(rep('A', 250), (rep('B',250))))
)

# Add the exact same amount of separability between the classes
test$x1[test$y == 'A'] <- test$x1[test$y == 'A'] + 5
```

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
# Make predictions
predictions <- predict(best, test)
table(predictions, truth = test$y)
```

```
##            truth
## predictions   A   B
##           A 249   1
##           B   1 249
```

Thus, with the value of `cost`, 498 of the test observations are correctly classified.

Support Vector Machine

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"`. In the former case, we also use the `degree` argument to specify a degree for the polynomial kernel and in the latter case we use `gamma` to specify a value of γ for the radial basis kernel.

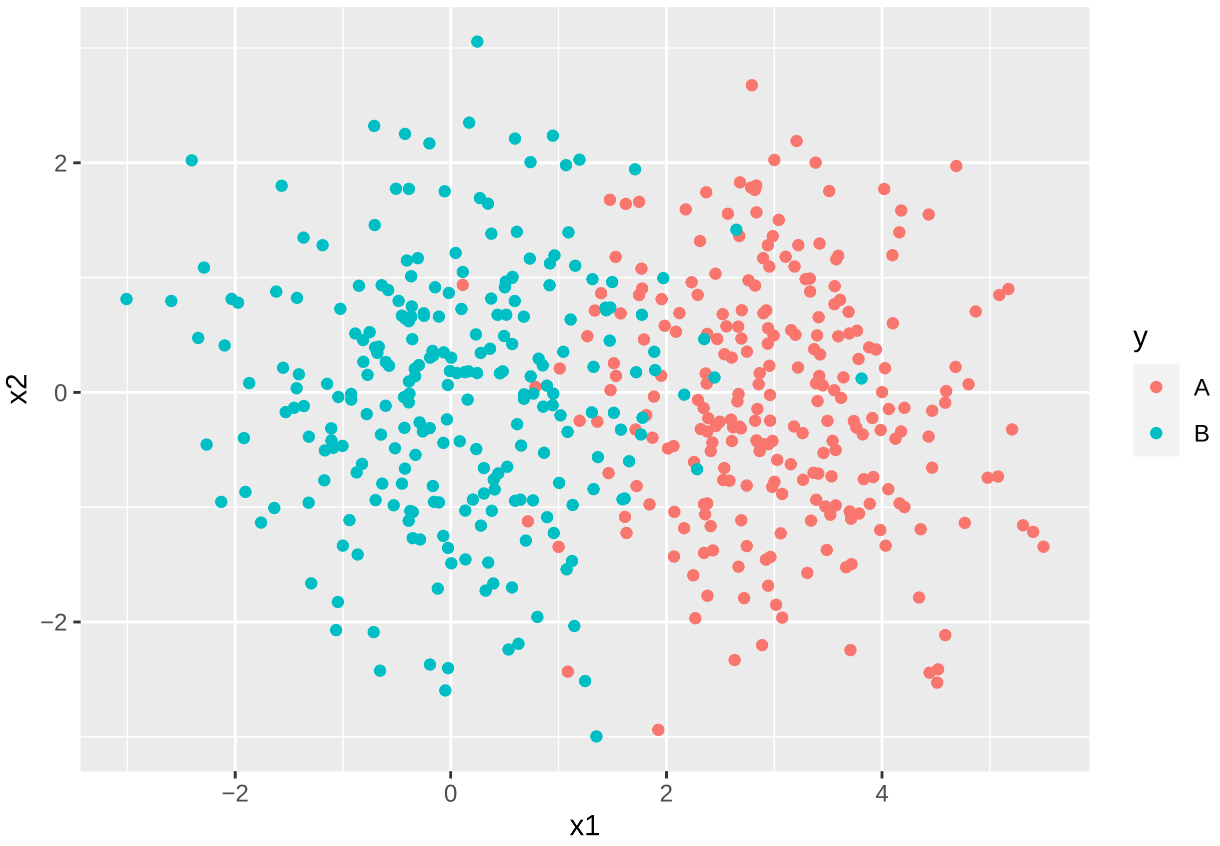
This section is mostly going to be an extension of the previous one, only now we are learning how to use different kernels. In particular, we will do one example with a radial kernel and another with a polynomial kernel. We begin by creating the sample data.

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

```r
# Set up toy data to work with
set.seed(1)
dat <- tibble(
  x1 = rnorm(500),
  x2 = rnorm(500),
  y = factor(c(rep('A',250), (rep('B',250))))
)
# Add a little of separability between the classes
dat$x1[dat$y == "A"] <-  dat$x1[dat$y == "A"] + 3
```

Plotting the data makes it clear that the class boundary is indeed non-linear:

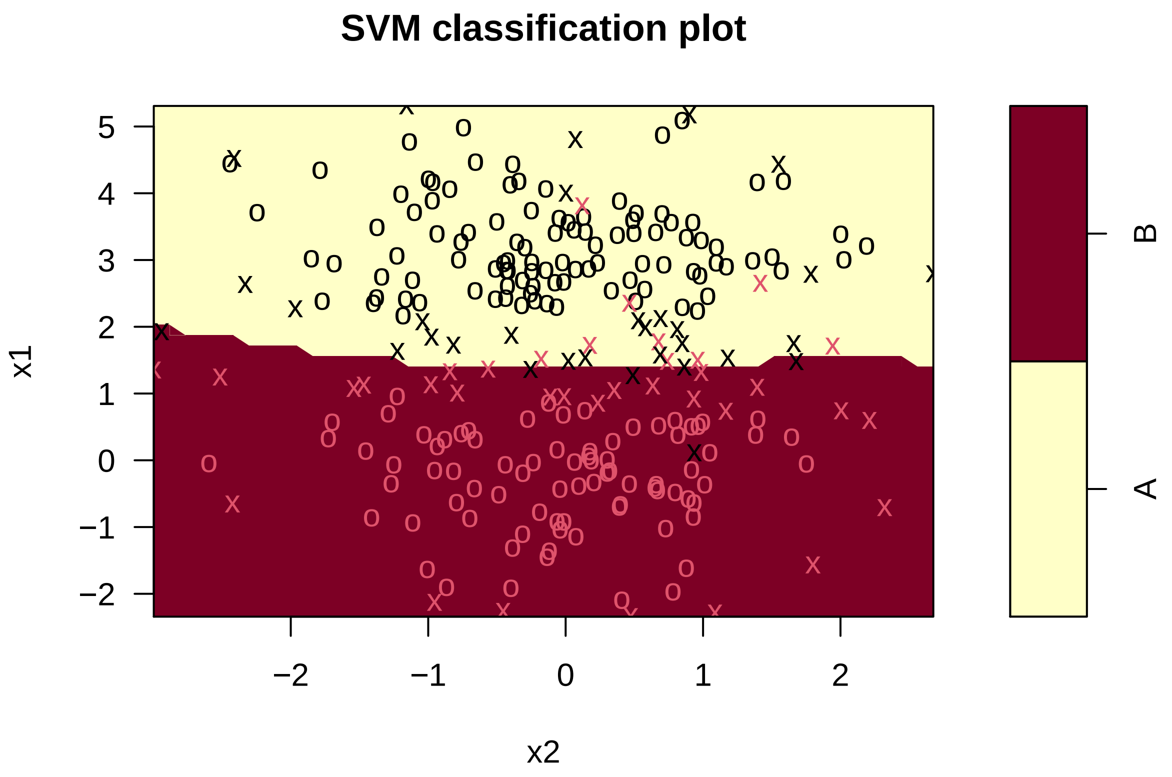
```r
ggplot(dat, aes(x1, x2, col=y)) +
  geom_jitter()
```



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 <- dat %>%
  sample_frac(0.5)

test <- setdiff(dat, train)
```

```r
rad_kern <- svm(y ~ ., data=train, kernel = 'radial',
                gamma = 1, cost = 1)
plot(rad_kern, 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(rad_kern)
```

```
##
## Call:
## svm(formula = y ~ ., data = train, kernel = "radial", gamma = 1,
##     cost = 1)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  66
##
##  ( 35 31 )
##
##
## Number of Classes:  2
##
## Levels:
##  A B
```

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 a price of a more irregular decision boundary that seems to be at risk of overfitting the data.

We can perform cross-validation using `tune()` to select the best choice of γ and `cost` for a SVM with a radial kernel. Then, we can check how well those parameter values perform on the training set.

```
rad_cv <- tune(svm, y ~ ., data = train, kernel = 'radial',
               ranges = list(cost = c (0.1, 1, 10, 100, 1000),
                             gamma = c (0.5, 1, 2, 3, 4)))
```

```
summary(rad_cv)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##   0.1   0.5
##
## - best performance: 0.056
##
## - Detailed performance results:
##     cost gamma error dispersion
## 1  1e-01   0.5 0.056 0.04299871
## 2  1e+00   0.5 0.064 0.04695151
## 3  1e+01   0.5 0.068 0.04237400
## 4  1e+02   0.5 0.080 0.04618802
## 5  1e+03   0.5 0.084 0.04402020
## 6  1e-01   1.0 0.060 0.04714045
## 7  1e+00   1.0 0.072 0.04541170
## 8  1e+01   1.0 0.072 0.04541170
## 9  1e+02   1.0 0.072 0.04131182
## 10 1e+03   1.0 0.088 0.04131182
## 11 1e-01   2.0 0.072 0.04917090
## 12 1e+00   2.0 0.076 0.04402020
## 13 1e+01   2.0 0.080 0.04216370
## 14 1e+02   2.0 0.080 0.02666667
## 15 1e+03   2.0 0.116 0.02951459
## 16 1e-01   3.0 0.072 0.04917090
## 17 1e+00   3.0 0.080 0.04216370
## 18 1e+01   3.0 0.072 0.04131182
## 19 1e+02   3.0 0.108 0.04237400
## 20 1e+03   3.0 0.128 0.04131182
## 21 1e-01   4.0 0.084 0.05146736
## 22 1e+00   4.0 0.076 0.04402020
## 23 1e+01   4.0 0.072 0.03155243
## 24 1e+02   4.0 0.124 0.05146736
## 25 1e+03   4.0 0.152 0.04917090
```

Therefore, the best choice of parameters involves `cost = 0.1` and `gamma = 0.5`. 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(prediction = predict(rad_cv$best.model),
      actual = train$y)
```

```
##           actual
## prediction   A   B
##           A 127   7
##           B   7 109
```

6% of test observations are misclassified by this SVM.

We'll just quickly run the same analysis with a polynomial kernel so we can compare the results.

```
poly_cv <- tune(svm, y ~ ., data=train, kernel='polynomial',
                ranges = list(cost = c(0.1, 1, 10, 100, 1000),
                              degree = c(2, 3, 4)))
```

```
table(prediction = predict(poly_cv$best.model),
      actual = train$y)
```

```
##            actual
## prediction   A   B
##          A 132  21
##          B   2  95
```

ROC Curves

The ROCR package can be used to produce ROC curves. We first write a short function to plot an ROC curve given a vector containing a numerical score for each observations, pred, and a vector containing the class label for each observation, truth.

In order to produce ROC curves from SVM fits, we need to extract the fitted decision values by setting decision.values = T in svm(). Once we have that, we use the ROCR library to help us build the curves.

For this example, we create the ROC curves using the training data because it is slightly less effort.

```
rocplot=function(pred, truth, ...){
  predob = prediction(pred, truth)
  perf = performance(predob , "tpr", "fpr")
  plot(perf ,...)
  }
```

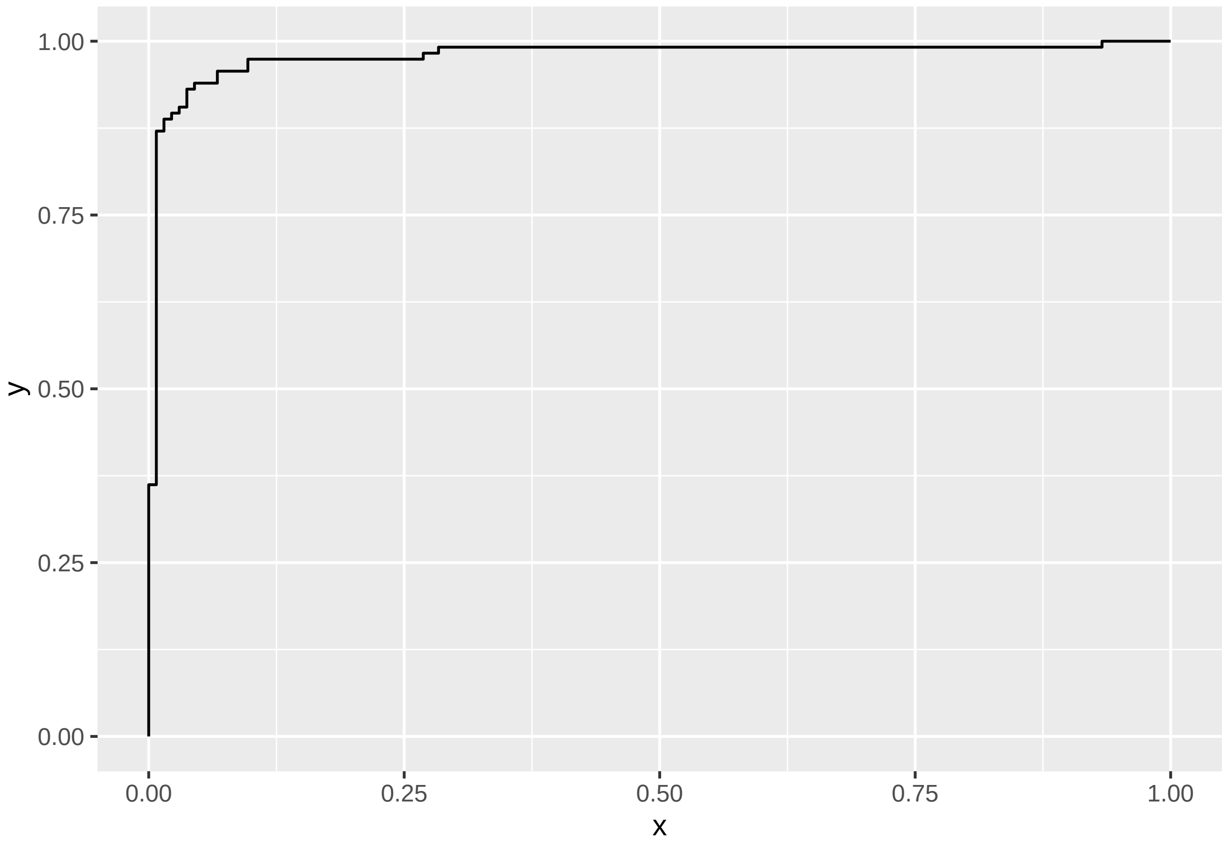
SVMs and support vector classifiers output class labels for each observations. However, it is also possible to obtain *fitted values* for each observation, which are the numerical scores used to obtain the class labels. For instance, in the case of a support vector classifier, the fitted value for an observation $X = (X_1, X_2, ..., X_p)^T$ takes the form $\hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2 + ... + \hat{\beta}_p X_p$. 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 another. In order to obtain the fitted values for a given SVM model fit, we use decision.values=TRUE when fitting svm(). Then the predict() function will output the fitted values.

```
# Run the first SVM and observe ROC performance
svm_roc <- svm(y ~ ., data = train, kernel = 'radial',
               gamma = 0.5, cost = 0.1, decision.values = T)

fitted <- svm_roc$decision.values
```

Now, we can produce the ROC plot.

```
perform <- prediction (fitted, train$y) %>%
  performance(., 'tpr', 'fpr')

original <- tibble(x = unlist(perform@x.values),
```

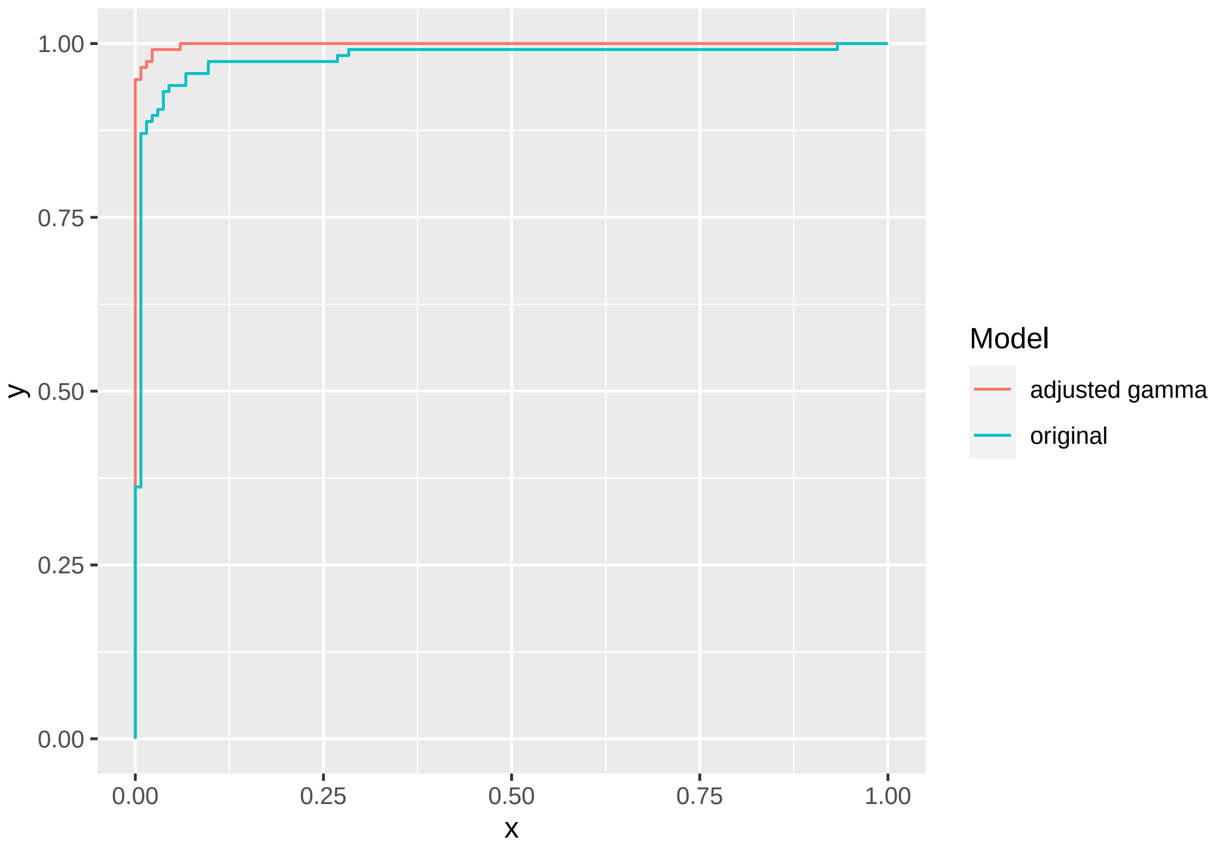
```r
                   y = unlist(perform@y.values))

ggplot(original, aes(x, y)) +
  geom_line()
```



SVM appears to be producing accurate predictions. By increasing γ, we can produce a more flexible fit and generate further improvements in accuracy.

```r
# Rerun SVM with more liberal gamma values
svm_roc2 <- svm(y ~ ., data=train, kernel='radial',
                gamma = 50, cost = 0.1, decision.values = T)

fitted2 <- svm_roc2$decision.values

perform2 <- prediction(fitted2, train$y) %>%
  performance(., 'tpr', 'fpr')

high_gamma <- tibble(x = unlist(perform2@x.values),
                   y = unlist(perform2@y.values))
```

```r
# Graphically compare the two models
bind_rows("original" = original, "adjusted gamma" = high_gamma, .id = "Model") %>%
  ggplot(., aes(x, y, col=Model)) +
  geom_line()
```



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 to get more accurate results.

SVM With Multiple Classes (K > 2 Classes)

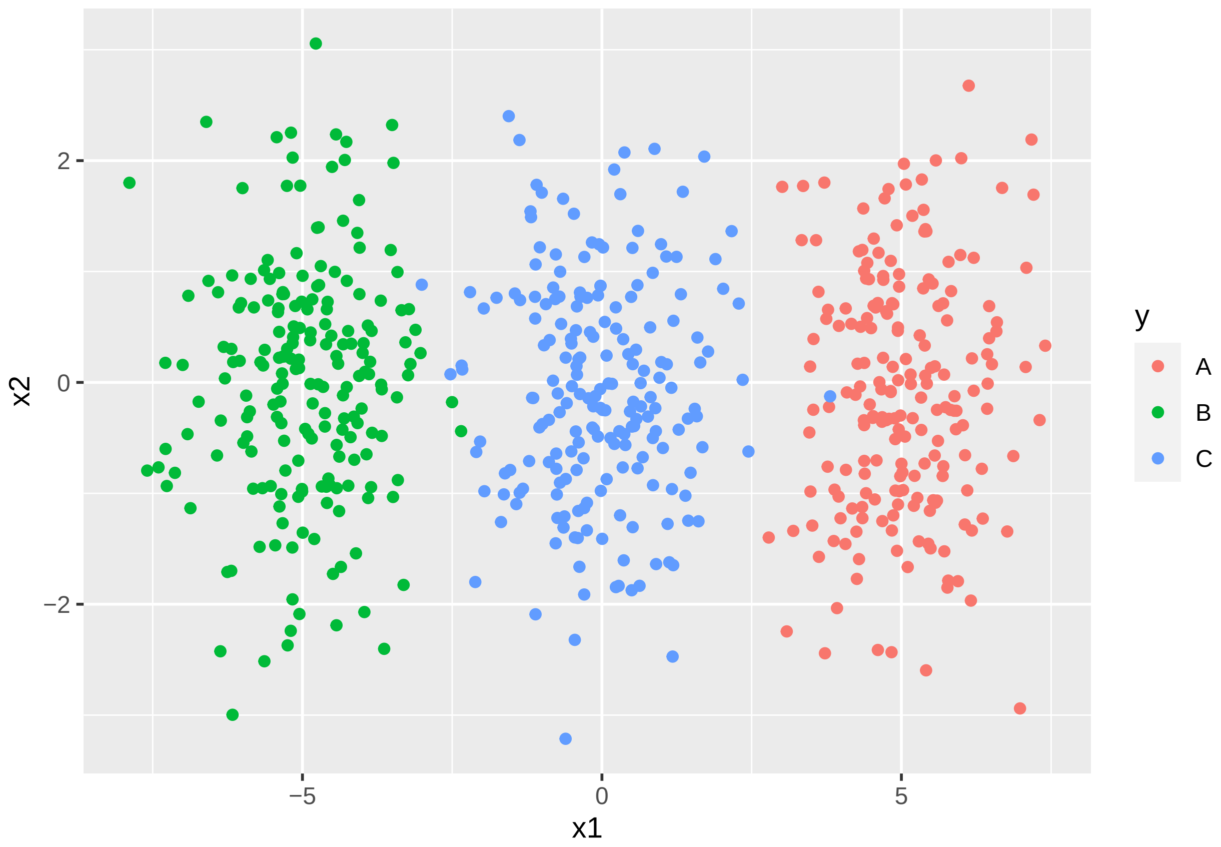
If the response is a factor containing more than two levels, then the `svm()` function will perform multi-class classification using the one-versus-one approach. We explore that setting here by generating a third class of observations.

When our dependent variable is categorical with more than two classes, `svm()` will perform classification using the one-versus-one approach. Recall that this requires $\binom{K}{2}$ SVMs for each possible comparison, so it might be kind of slow.

```
# Create the sample data
set.seed(1)
dat <- tibble(
  x1 = rnorm(600),
  x2 = rnorm(600),
  y = factor(c(rep('A', 200), rep('B', 200), rep('C', 200)))
)
```

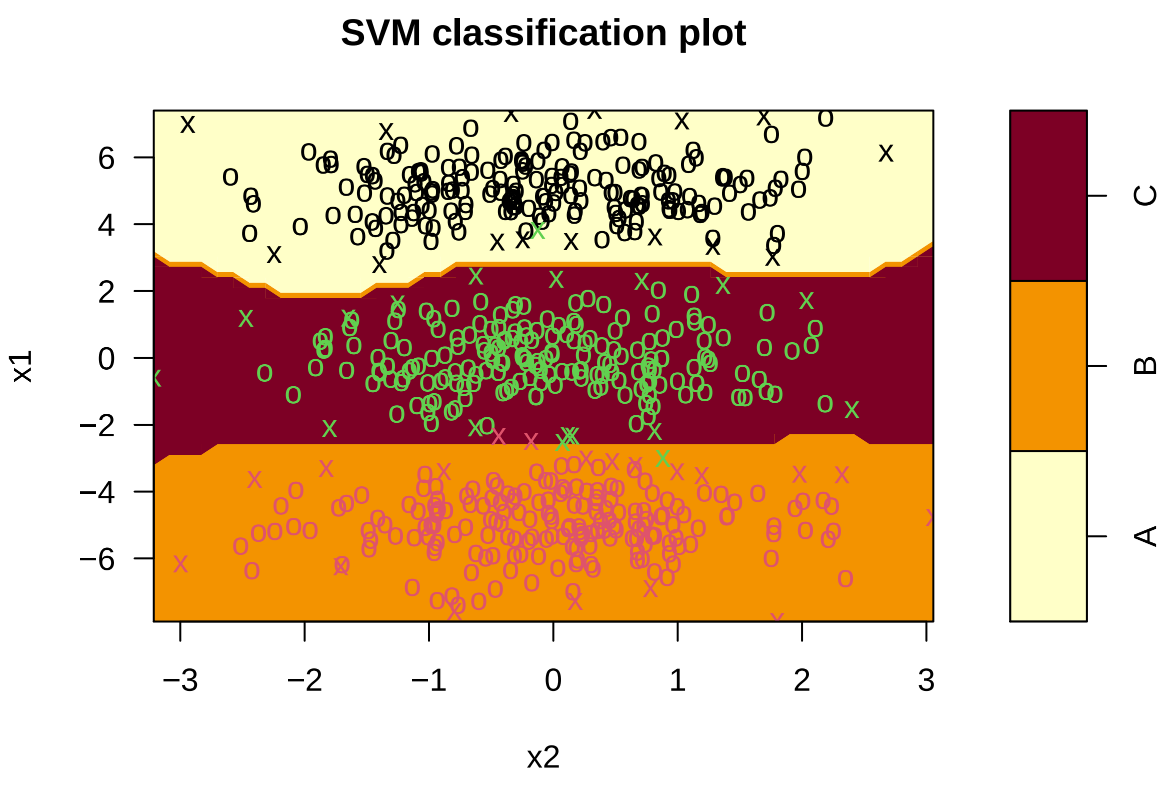
```
# Artificially add separability between the classes
dat$x1[dat$y == "A"] <- dat$x1[dat$y == "A"] + 5
dat$x1[dat$y == "B"] <- dat$x1[dat$y == "B"] - 5
```

```
ggplot(dat, aes(x1, x2, col=y)) +
  geom_jitter()
```



Now we fit an SVM to the data:

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



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

Application to Gene Expression Data

We now examine the `Khan` dataset, which consists of a number of tissue samples corresponding to four distinct types of small round blue cell tumours. For each tissue sample, gene expression measurements are avaliable. The dataset consists of training data, `xtrain` and `ytrain`, and testing data, `xtest` and `ytest`.

We examine the dimension of the data:

```
names(Khan)
```

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

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

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

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

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

The dataset consists of expression measurements for 2,308 genes. The training and test set consist of 62 and 20 observations respectively.

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

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

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

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

We will use a support vector approach to predict cancer subtype using gene expression measurements. In this dataset, there are a very large number of features relative to the number of observations. This suggests that we should use a linear kernel, because the additional flexibility that will result from using a polynomial or radial kernel is unnecessary.

```
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
```

```
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
```

We can see that there are *no* training errors. In fact, this is not surprising because the large number of variables relative to the number of observations implies that it is easy to find hyperplanes that fully separate the classes. We are most interested not in the support vector classifier's performance on the training observations, but rather its performance on the test observations.

```
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
```

We can see that using `cost=10` yields twotest errors on this data.