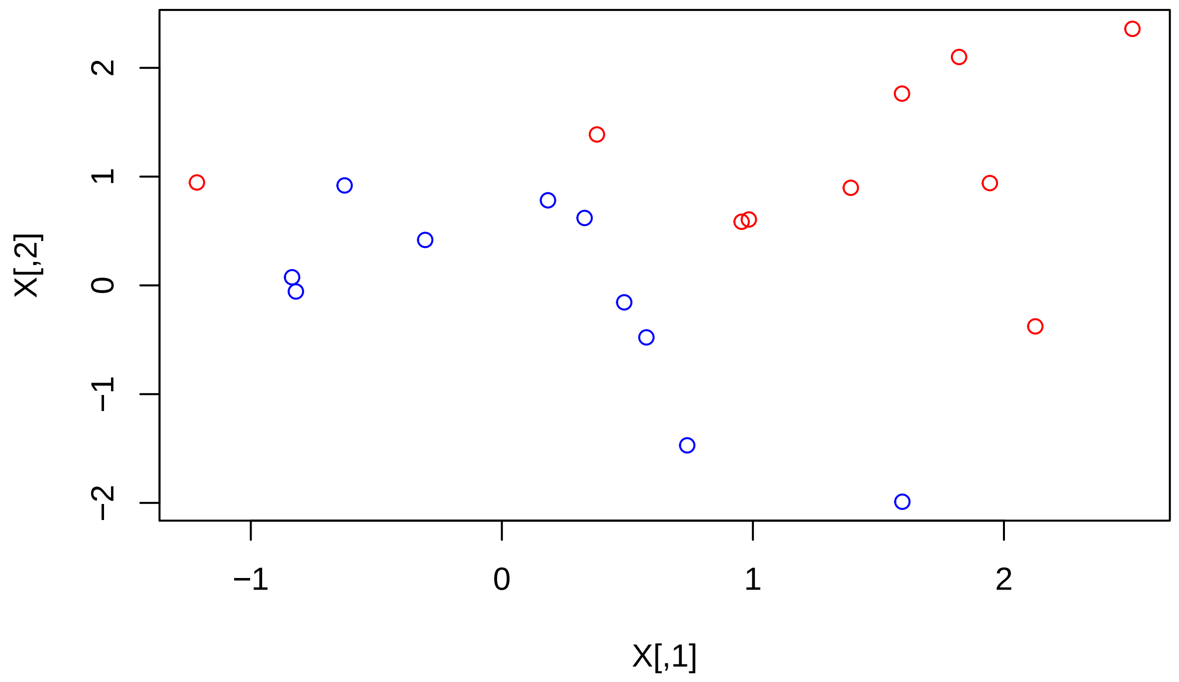
Chapter 9 Lab

Contents

Support Vector Classifier

The library that we use will be `e1071`. An alternative for support vector classification (i.e. linear SVM with linear kernel) is `LiblineaR`.

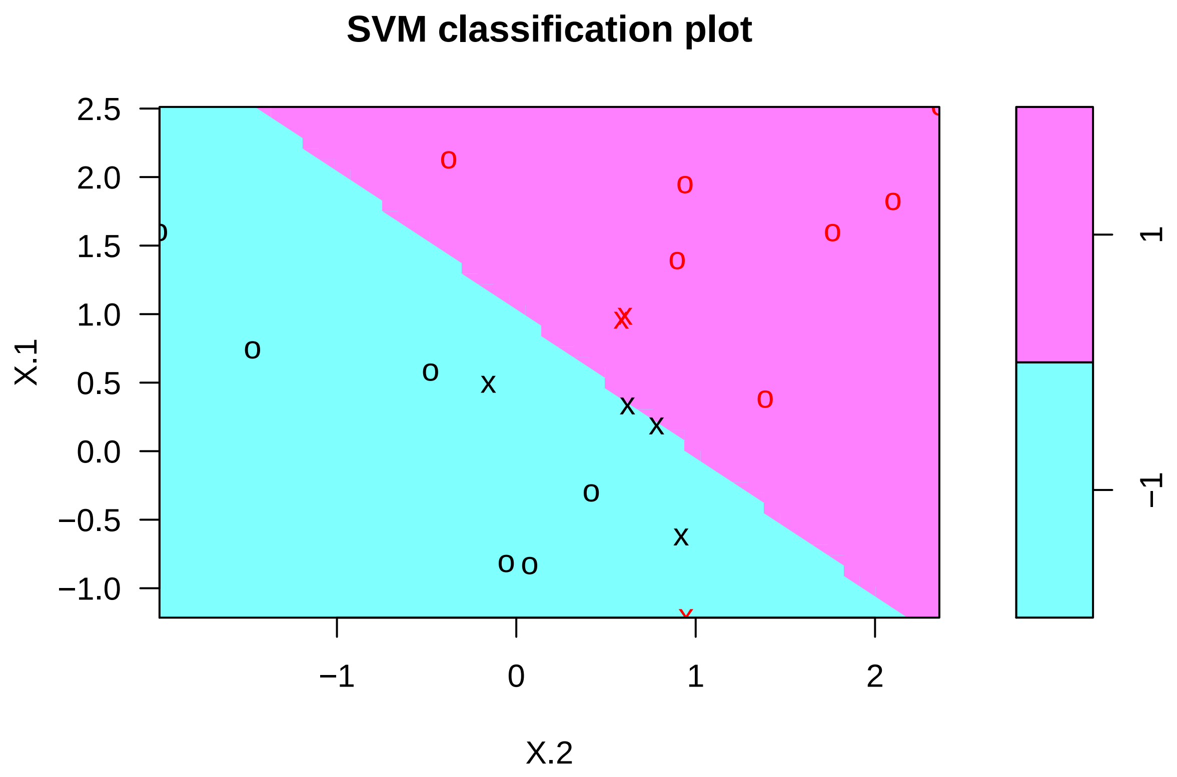
```r
set.seed(1)
X <- matrix(rnorm(20*2), ncol = 2)
y <- c(rep(-1, length = 10), rep(1, length = 10))
X[y == 1, ] <- X[y == 1, ] + 1
plot(X, col = 3 - y)
```



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

We plot the fit:

```
plot(svmfit, data = dat)
```



The support vectors are plotted as crosses, and the second variable is on the horizontal axis.

```
names(svmfit)
```

```
##  [1] "call"            "type"            "kernel"
##  [4] "cost"            "degree"          "gamma"
##  [7] "coef0"           "nu"              "epsilon"
## [10] "sparse"          "scaled"          "x.scale"
## [13] "y.scale"         "nclasses"        "levels"
## [16] "tot.nSV"         "nSV"             "labels"
## [19] "SV"              "index"           "rho"
## [22] "compprob"        "probA"           "probB"
## [25] "sigma"           "coefs"           "na.action"
## [28] "fitted"          "decision.values" "terms"
```

As shown below, we could retrieve different variables from the fit, inclduing the index and values of the support vectors:

```
svmfit$index
```

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

```
svmfit$SV
```

```
##           X.1        X.2
## 1  -0.6264538  0.9189774
## 2   0.1836433  0.7821363
## 5   0.3295078  0.6198257
## 7   0.4874291 -0.1557955
## 14 -1.2146999  0.9461950
## 16  0.9550664  0.5850054
## 17  0.9838097  0.6057100
```

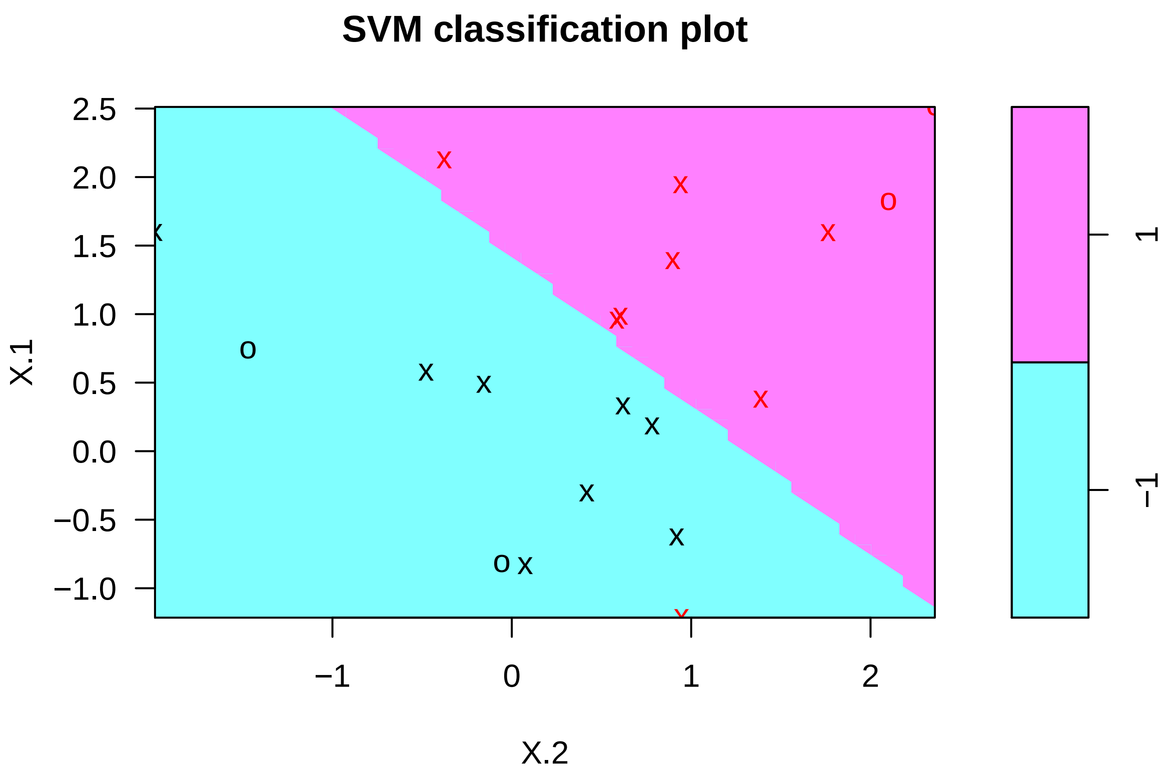
We could also obtain basic information using `summary()` command:

```
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10,
##     scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##       gamma:  0.5
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

It tells us for example that 4 suppport vectors are in the -1 class and 3 are in the 1 class. When we lower the cost, we get more support vectors:

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



```
svmfit$index
```

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

While `svm()` retreives the support vectors, it does not allow us to see the parameters for neither the separating hyperplane nor the margin.

Cross-Validation

`tune` is a built-in function in `e1071` which allows us to cross-validate across a range of models.
Unlike the book, we do impose `scale = FALSE` to be consistent with our later comparision (although :

```
set.seed(3)
tune.out <- tune(svm, y ~ ., data = dat, kernel = "linear",
     ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100)),
     scale = FALSE)
```

The first argument in `tune()` is either the function or the character string name of it. We access the errors for each of the models being evaluated using `summary.tune()`:

```
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  0.60  0.3944053
## 2 1e-02  0.60  0.3944053
## 3 1e-01  0.05  0.1581139
## 4 1e+00  0.15  0.2415229
## 5 5e+00  0.10  0.2108185
## 6 1e+01  0.10  0.2108185
## 7 1e+02  0.10  0.2108185
```

tune() stores the best model:

```
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", scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.1
##       gamma:  0.5
##
## Number of Support Vectors:  16
##
##  ( 8 8 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

Generate test data:

```
# set the seed, so that changing the chunks' order of execution wouldn't change the result:
set.seed(1)
Xtest <- matrix(rnorm(20*2), ncol = 2)
ytest <- sample(c(-1, 1), size = 20, replace = TRUE)
Xtest[ytest == 1, ] = Xtest[ytest == 1, ] + 1
testdat <- data.frame(X = Xtest, y = as.factor(ytest))
```

Test error rate with the best model:

```
ypred <- predict(bestmod, newdata = testdat)
table(predict = ypred, truth = ytest)
```

```
##        truth
## predict -1  1
##      -1 10  1
##      1   1  8
```

Test error rate with the model with `cost = 0.01`:

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

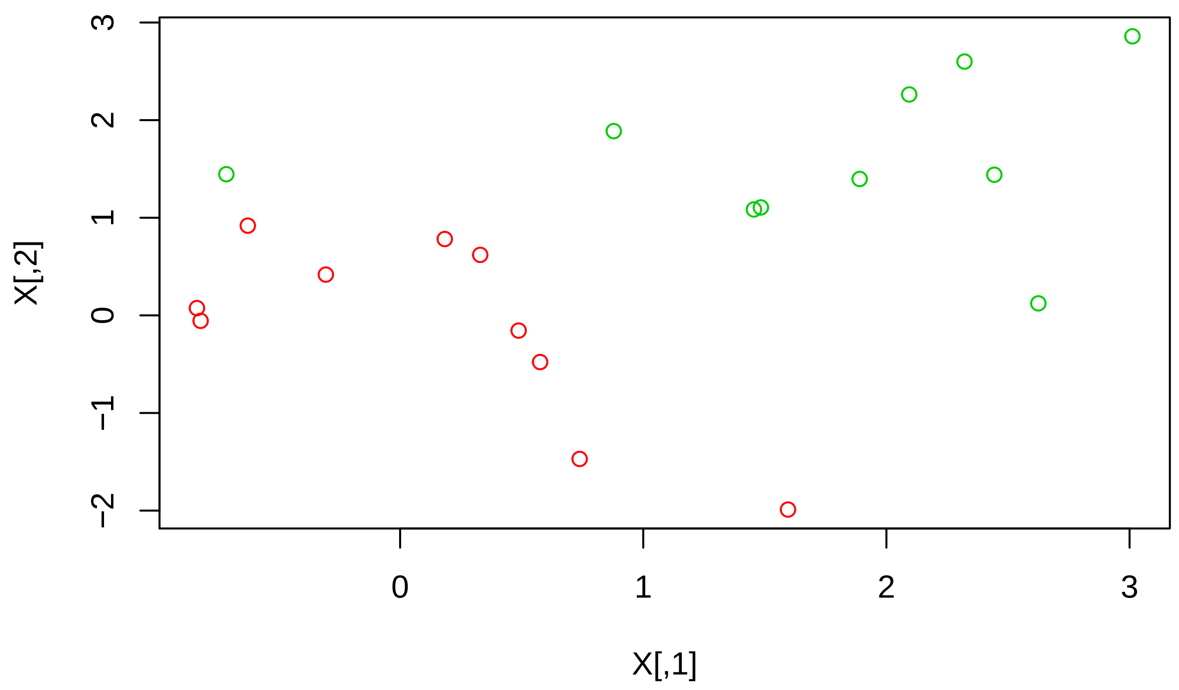
```
##        truth
## predict -1 1
##      -1  8 7
##      1   3 2
```

This model performs much worse than the best model. However, it would have been possible to find a model that outperfroms the best model (according to cross-validation), due to the random nature of cross-validation in choosing the best model and the randomness in the test data.

Linearly Separable Data

To make the classes linearly separable, we increase features corresponding to class +1 another 0.5 point (in addition to the 1 point we added to them already). Be careful to run the below once or improve the naming.

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

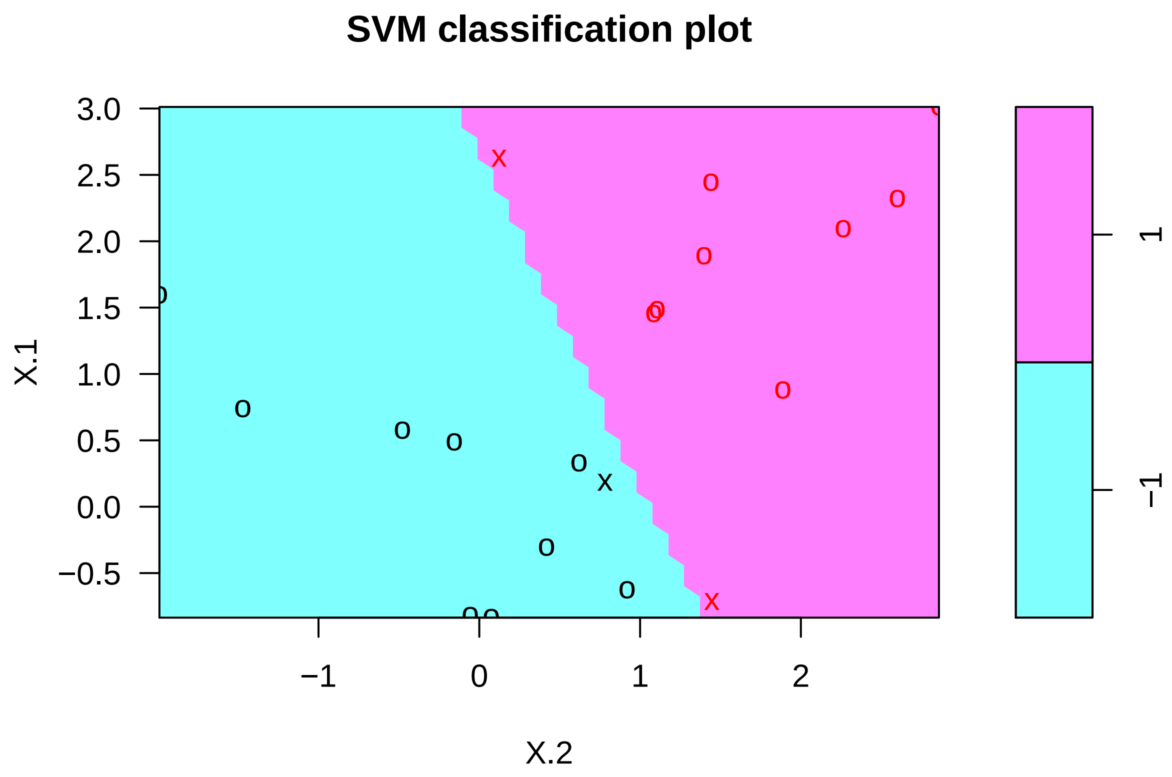


Hard margins:

```r
dat <- data.frame(X = X, y = as.factor(y))
svmfit <- svm(y ~ ., data = dat, kernel = "linear", cost = 1e+5,
              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
##       gamma:  0.5
##
## Number of Support Vectors:  3
##
##  ( 1 2 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

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



This model is fit too closely, which has resulted in only 3 support vectors and a narrow margin.

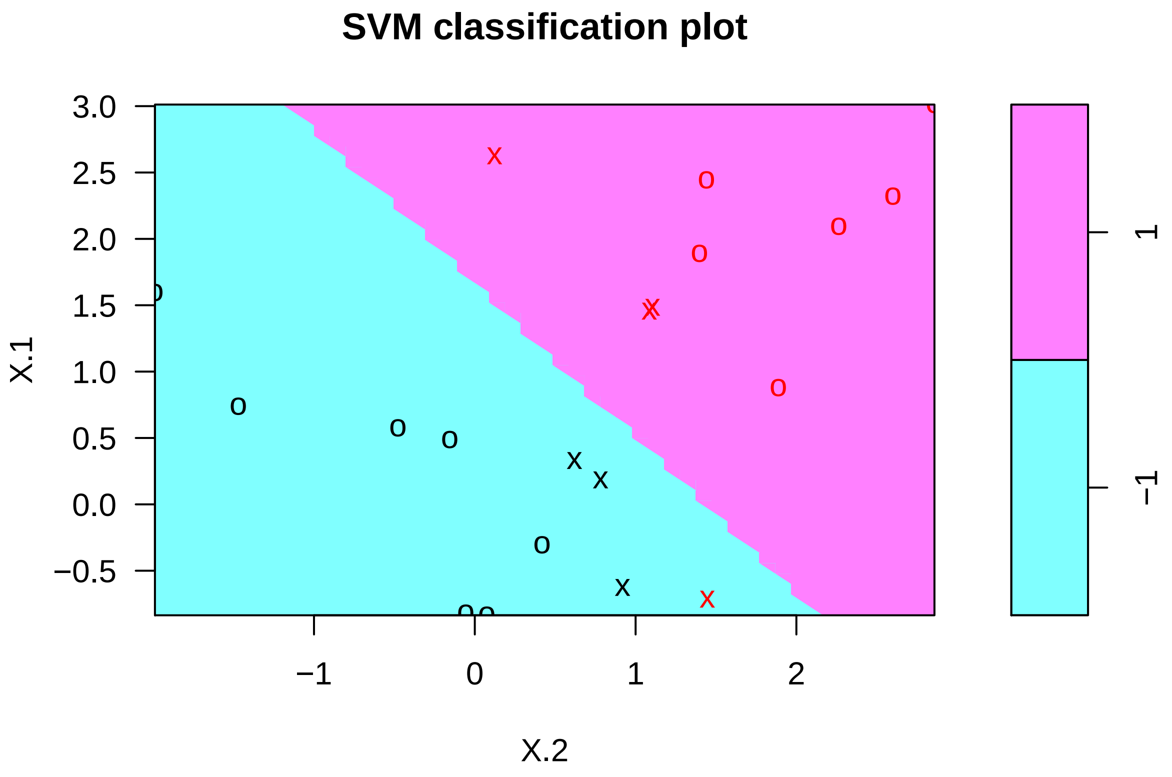
Soft margins:

```
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
##       gamma:  0.5
##
## Number of Support Vectors:  7
##
##  ( 3 4 )
##
##
## Number of Classes:  2
##
```

```
## Levels:
##  -1 1
```

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



Takeaways

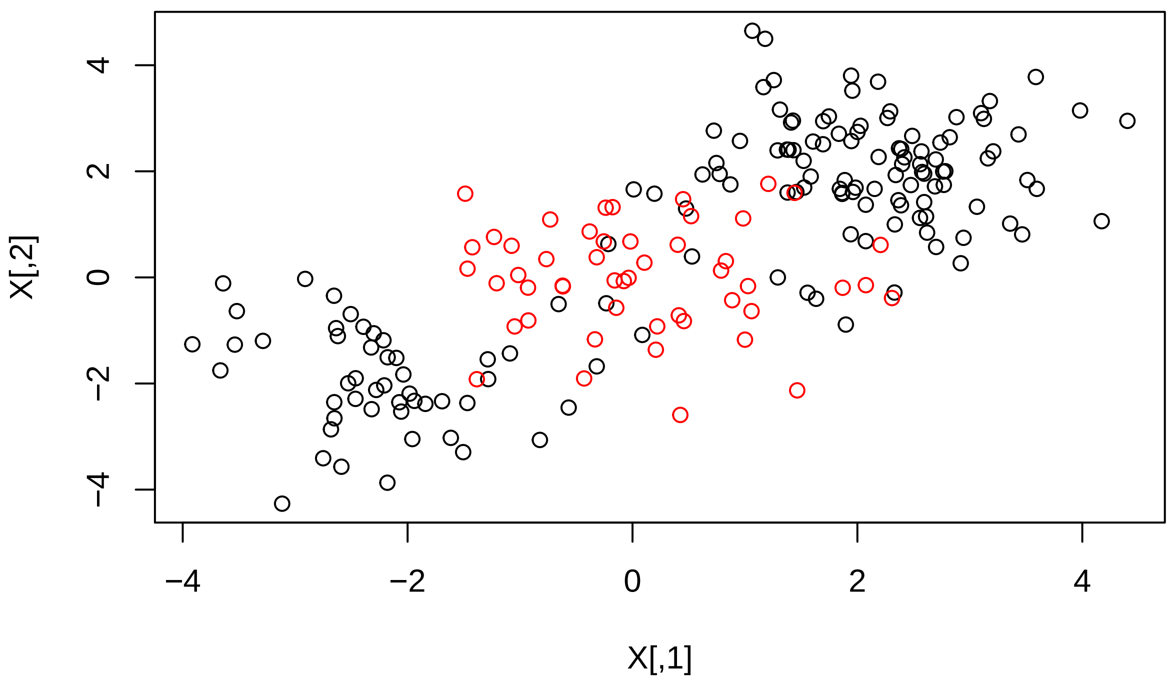
- Make sure the response is a factor variable before using `svm()` classification
- Always name the variables inside `data.frame()` or `list()`
  - Otherwise, the R's default naming may cause confusion later
- Since SVM is distance-based, we should scale the variables to get reasonable results.
  - Otherwise the variables with higher variance will dominate, and the kernel will use measures of similarity based on those variables.

- `plot.svm()` needs a second argument indicating data.
- `summary.svm()` gives us the basic information, while element `index` and `SV` of a fit object give information about the support vectors.
- The first argument in `tune()` is the method
- We can name the dimensions in `table()`

Support Vector Machine

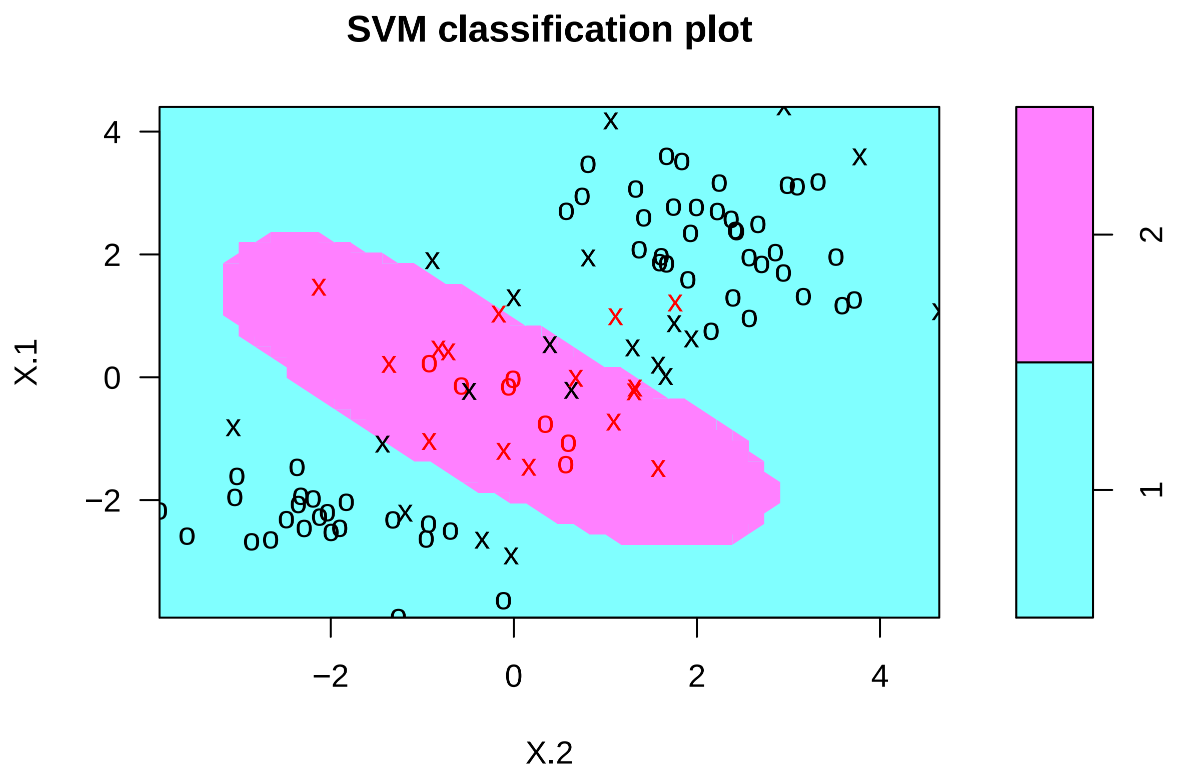
```r
set.seed(1)
# Generate nonlinear data:
```

```r
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)
```



Radial kernel:

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



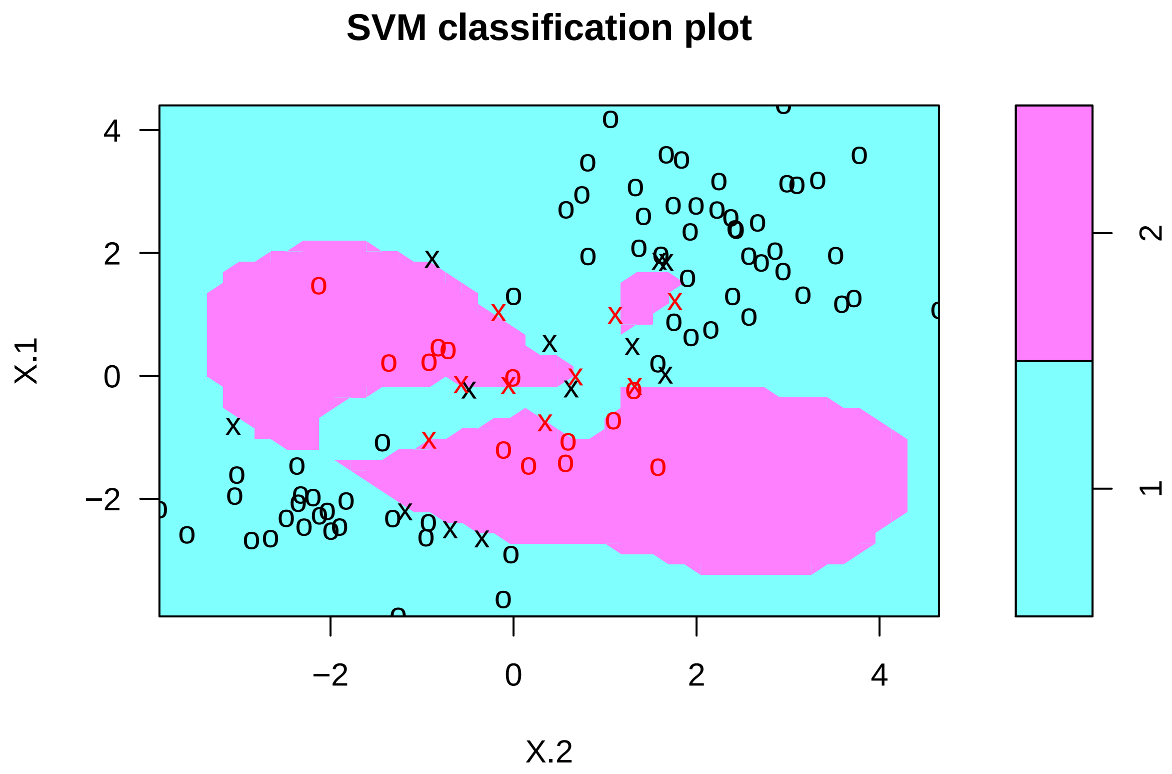
The red markers belong to the purple area and the black markers to the green area. Those points that are support vectors (in the Hilbert feature space) are identified by crosses.

```
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat[train, ], cost = 1, kernel = "radial",
##     gamma = 1)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  1
##
## Number of Support Vectors:  35
##
##  ( 20 15 )
##
##
## Number of Classes:  2
##
## Levels:
##  1 2
```

There are a fair number of training errors. To reduce them we could increase the cost of violating the margin, but that would come at the cost of a more irregular decision boundary and the risk of overfitting:

```
svmfit <- svm(y ~ ., data = dat[train, ], cost = 1e+5, kernel = "radial",
              gamma = 1)
plot(svmfit, data = dat[train, ])
```



Cross-Validation

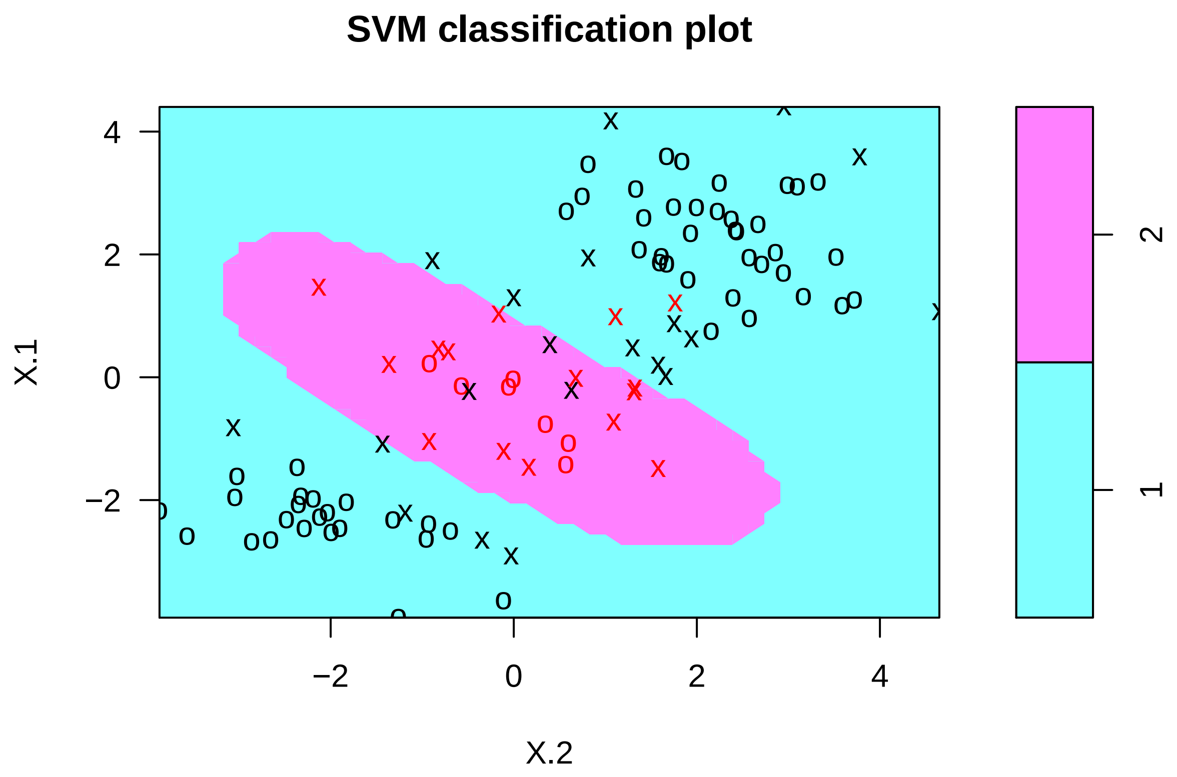
```
set.seed(1)
tune.out <- tune(svm, y ~ ., data = dat[train, ], kernel = "radial",
                 ranges = list(gamma = c(0.5, 1, 2, 3, 4), cost = c(0.1, 1, 10, 100, 1000)))
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  gamma cost
##      1    1
##
## - best performance: 0.1
##
```

```
## - Detailed performance results:
##    gamma  cost error dispersion
## 1    0.5 1e-01  0.22 0.13984118
## 2    1.0 1e-01  0.22 0.13984118
## 3    2.0 1e-01  0.22 0.13984118
## 4    3.0 1e-01  0.22 0.13984118
## 5    4.0 1e-01  0.22 0.13984118
## 6    0.5 1e+00  0.11 0.12866839
## 7    1.0 1e+00  0.10 0.10540926
## 8    2.0 1e+00  0.10 0.12472191
## 9    3.0 1e+00  0.12 0.12292726
## 10   4.0 1e+00  0.13 0.12516656
## 11   0.5 1e+01  0.11 0.12866839
## 12   1.0 1e+01  0.11 0.12866839
## 13   2.0 1e+01  0.12 0.13165612
## 14   3.0 1e+01  0.13 0.10593499
## 15   4.0 1e+01  0.14 0.10749677
## 16   0.5 1e+02  0.10 0.12472191
## 17   1.0 1e+02  0.12 0.13165612
## 18   2.0 1e+02  0.17 0.10593499
## 19   3.0 1e+02  0.13 0.08232726
## 20   4.0 1e+02  0.14 0.08432740
## 21   0.5 1e+03  0.13 0.14181365
## 22   1.0 1e+03  0.16 0.10749677
## 23   2.0 1e+03  0.13 0.09486833
## 24   3.0 1e+03  0.18 0.11352924
## 25   4.0 1e+03  0.18 0.11352924
```

```
gamma_best <- summary(tune.out)$best.parameters[1]
cost_best <- summary(tune.out)$best.parameters[2]
```

```
plot(tune.out$best.model, data = dat[train, ])
```



Test Error Rate

```r
ypred <- predict(tune.out$best.model, newdata = dat[-train, ])
table(predict = ypred, truth = y[-train])
```

```
##        truth
## predict  1  2
##       1 69  8
##       2  3 20
```

The test error rate is

```r
mean(ypred != y[-train])
```

```
## [1] 0.11
```

Summary and Takeaways

- The additional argument for `kernel = "polynomial'` is `degree` and for `kernel = "radial"` is `gamma`
- in the `plot.svm()`, the red markers belong to the purple area, while black markers to the green area. The support vectors are identified by crosses.
- Be careful to use `plot.svm()` with only the training data or the test data, e.g. `plot(svmfit, data = dat[train, ])`
- `tune()` can choose the best model on grids of `cost` and `gamma` at the same time, e.g. `ranges = list(gamma = c(...), cost = c(...)))`

  - There may be different number of values for `gamma` and `cost`
  - `tune()` considers all combinations of `gamma` and `cost`
- The book sets the `cost` argument in `tune` to change exponentially, which reminds me of how we set the grid for `lambda` in `glmnet()`.

ROC Curves

`prediction()` and `performance()` are the main functions in the `ROCR` package. `prediction()` is used to transform the input dat to a standardized format, and `performance()` is used for predictor evaluation.

```
library(ROCR)
```

```
## Loading required package: gplots

##
## Attaching package: 'gplots'

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

```
rocplot <- function(pred, truth, ...) {
  predob <- prediction(predictions = pred, labels = truth,
                       label.ordering = c(2, 1))
  perf <- performance(predob, measure = "tpr", x.measure = "fpr")
  plot(perf, ...)
}
```

Note the argument `label.ordering = c(2, 1)` that we use, which is not used in ISLR. If we do not use it, we would get an inverted ROC curve. Apart from the purposes for which we use `ROCR` here, the package can be used to find the optimal cutoff, e.g. when costs of false positive and false negative are different. You can find a good introduction to the `ROCR` package here.
Before plotting the ROC curves, we obtain the fitted values $f(x) = \beta_0 + x^T\beta$:

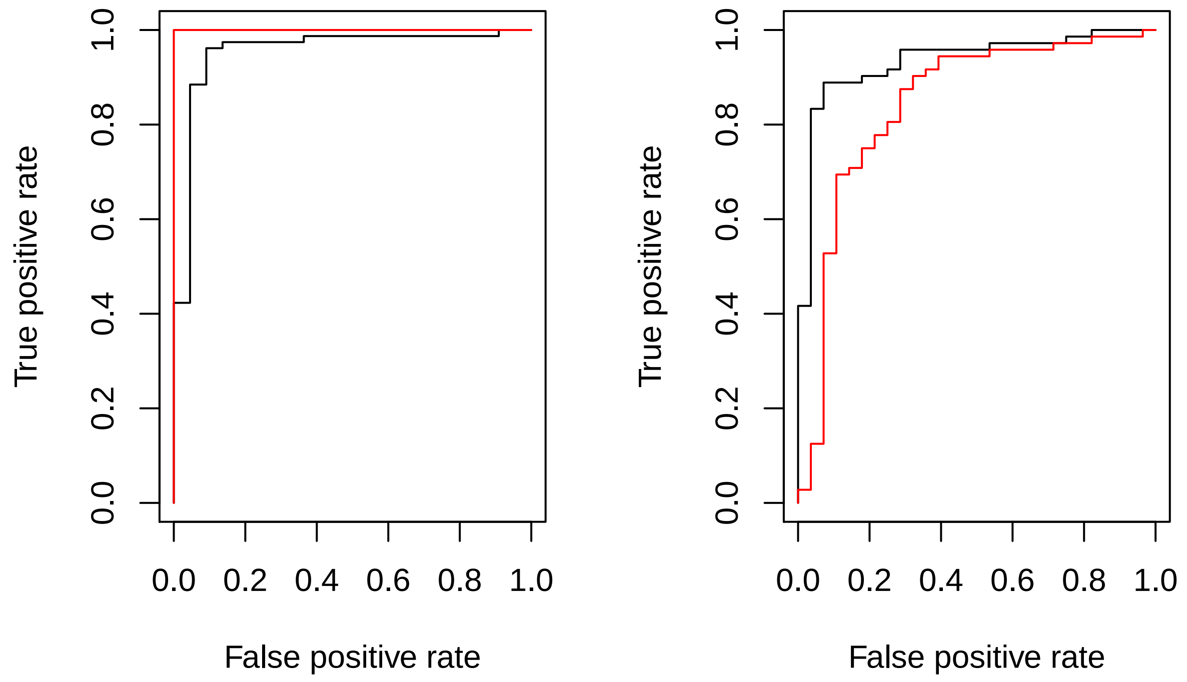
```
svmfit.opt <- svm(y ~ ., data = dat[train, ], cost = cost_best, kernel = "radial",
                  gamma = gamma_best)
pred.out <- predict(svmfit.opt, newdata = dat[train, ], decision.values = TRUE)
fitted <- attr(pred.out, "decision.values")
```

Best model's ROC for training data

```
par(mfrow = c(1, 2))
rocplot(pred = fitted, truth = dat[train, "y"])
# A flexible model's ROC for training data
svmfit.flex <- svm(y ~ ., data = dat[train, ], cost = cost_best,
                   kernel = "radial", gamma = 50)
pred.flex <- predict(svmfit.flex, newdata = dat[train, ], decision.values = TRUE)
fitted.flex <- attr(pred.flex, "decision.values")
rocplot(pred = fitted.flex, truth = dat[train, "y"], add = TRUE, col = "red")

# Best model's ROC for test data
pred.opt.test <- predict(svmfit.opt, newdata = dat[-train, ],
                         decision.values = TRUE)
fitted.opt.test <- attr(pred.opt.test, "decision.values")
rocplot(pred = fitted.opt.test, truth = dat[-train, "y"])
# A flexible model's ROC for test data
```

```r
pred.flex.test <- predict(svmfit.flex, newdata = dat[-train, ],
                          decision.values = TRUE)
fitted.flex.test <- attr(pred.flex.test, "decision.values")
rocplot(pred = fitted.flex.test, truth = dat[-train, "y"], add = TRUE, col = "red")
```

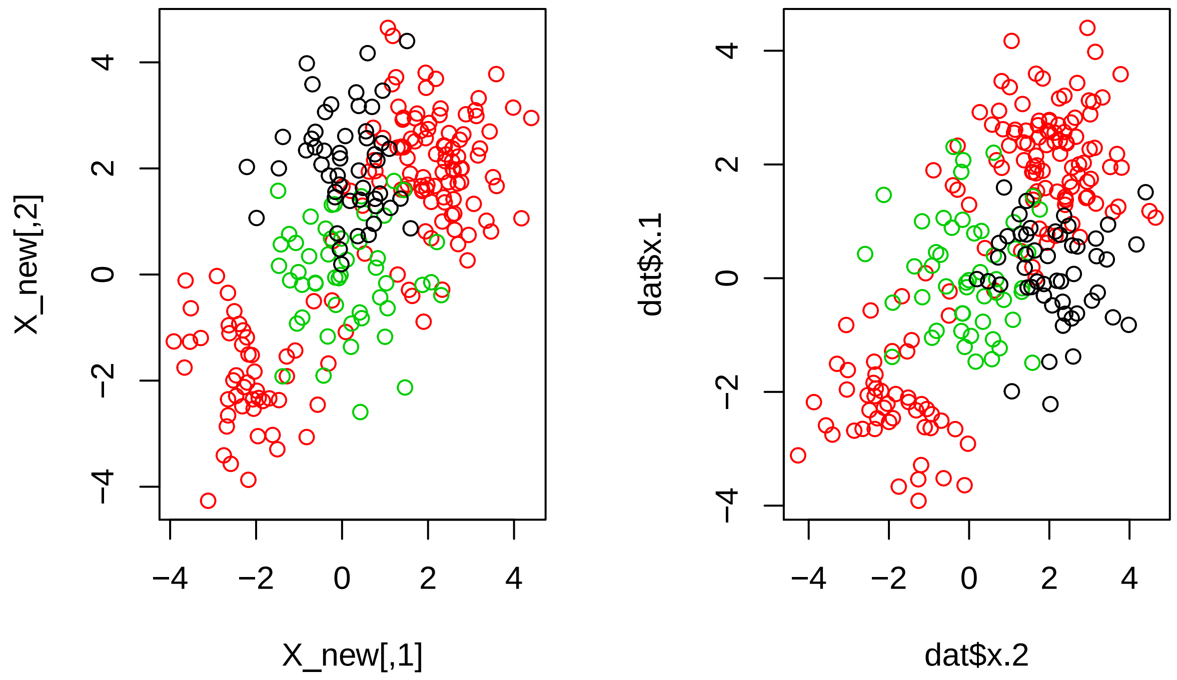


SVM with Multiple Classes

The introduction of variables `X_tmp` and `X_new` below makes the results robust to multiple execution of the chunk below:

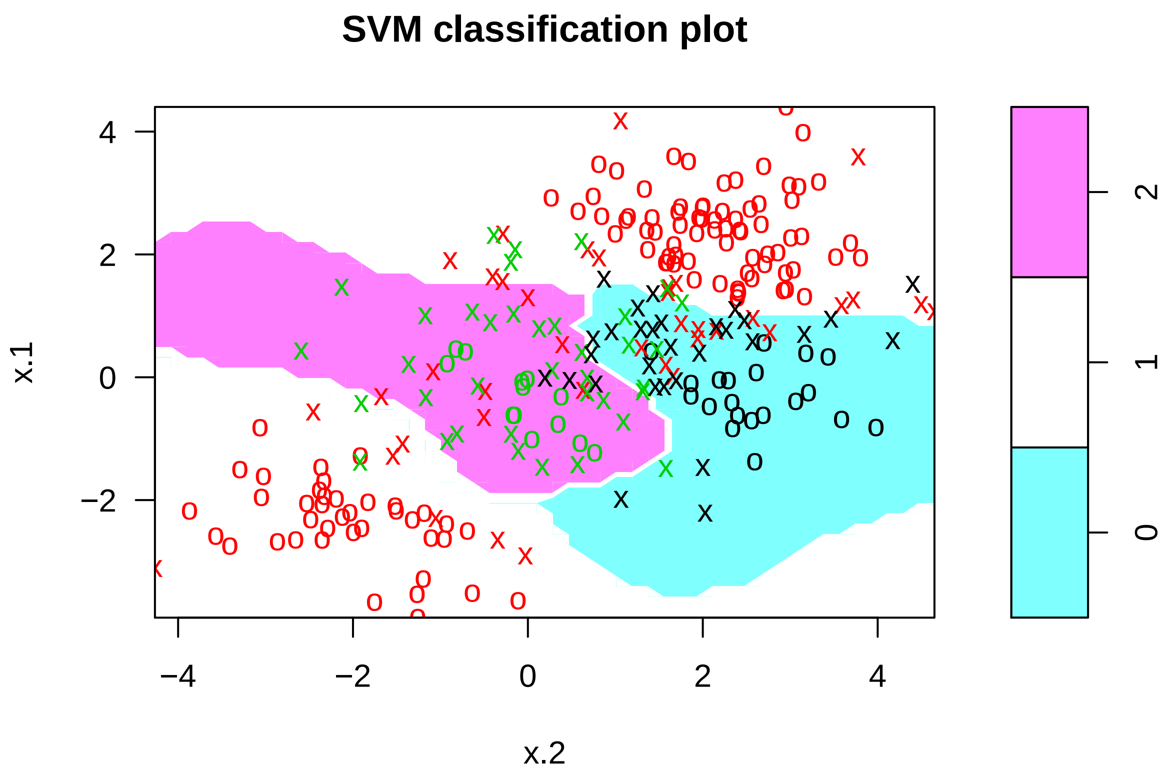
```r
# add a class
set.seed(1)
# temporary variables
X_tmp <- X
y_tmp <- y
# update X and y
X_tmp <- rbind(X_tmp, matrix(rnorm(50*2), ncol = 2))
y_tmp <- c(y_tmp, rep(0, length = 50))
# shift only the second variable up
X_tmp[y_tmp == 0, 2] <- X_tmp[y_tmp == 0, 2] + 2
# define updated variables
X_new <- X_tmp
y_new <- y_tmp
# rewrite the data frame
```

```r
dat <- data.frame(x = X_new, y = as.factor(y_new))
# plot
par(mfrow = c(1, 2))
plot(X_new, col = y_new + 1)
plot(dat$x.2, dat$x.1, col = as.numeric(as.character(dat$y)) + 1)
```



The second plot above flips the variables on the axes to correspond to the plot depicted below for the SVM fit to the whole data:

```r
svmfit_3classes <- svm(y ~ ., data = dat, cost = 10,
                       kernel = "radial", gamma = 1)
plot(svmfit_3classes, data = dat)
```



Application to Gene Expression Data

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

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

```r
str(Khan)
```

```
## List of 4
##  $ xtrain: num [1:63, 1:2308] 0.7733 -0.0782 -0.0845 0.9656 0.0757 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:63] "V1" "V2" "V3" "V4" ...
##   .. ..$ : NULL
##  $ xtest : num [1:20, 1:2308] 0.14 1.164 0.841 0.685 -1.956 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:20] "V1" "V2" "V4" "V6" ...
##   .. ..$ : NULL
##  $ ytrain: num [1:63] 2 2 2 2 2 2 2 2 2 2 ...
##  $ ytest : num [1:20] 3 2 4 2 1 3 4 2 3 1 ...
```

```r
summary(Khan)
```

```
##        Length Class  Mode
## xtrain 145404 -none- numeric
## xtest   46160 -none- numeric
## ytrain     63 -none- numeric
## ytest      20 -none- numeric
```

We see that `Khan` is a list.

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

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

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

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

```r
length(Khan$ytrain)
```

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

```r
length(Khan$ytest)
```

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

There are 63 training observations and 2308 features.

```r
str(Khan$ytrain)
```

```
##  num [1:63] 2 2 2 2 2 2 2 2 2 2 ...
```

```r
unique(Khan$ytrain)
```

```
## [1] 2 4 3 1
```

Next step is to see how mnay observation of each response we have. This is especially important since we have few observations. If we have for instance only one response value we could not fit any model.

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

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

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

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

SVM:

```r
# training data
dat_train <- data.frame(x = Khan$xtrain, ytrain = as.factor(Khan$ytrain))
svmfit_gene <- svm(ytrain ~ ., data = dat_train, cost = 10, kernel = "linear")
summary(svmfit_gene)
```

```
##
## Call:
## svm(formula = ytrain ~ ., data = dat_train, cost = 10, kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##       gamma:  0.0004332756
```

```
##
## Number of Support Vectors:  58
##
##  ( 20 20 11 7 )
##
##
## Number of Classes:  4
##
## Levels:
##  1 2 3 4
```

```
pred_train <- predict(svmfit_gene, newdata = dat_train)
table(pred = pred_train, truth = dat_train$ytrain)
```

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

Test data:

```
dat_test <- data.frame(x = Khan$xtest, y = as.factor(Khan$ytest))
pred_test <- predict(svmfit_gene, newdata = dat_test)
table(pred = pred_test, truth = Khan$ytest)
```

```
##     truth
## pred 1 2 3 4
##    1 3 0 0 0
##    2 0 6 2 0
##    3 0 0 4 0
##    4 0 0 0 5
```