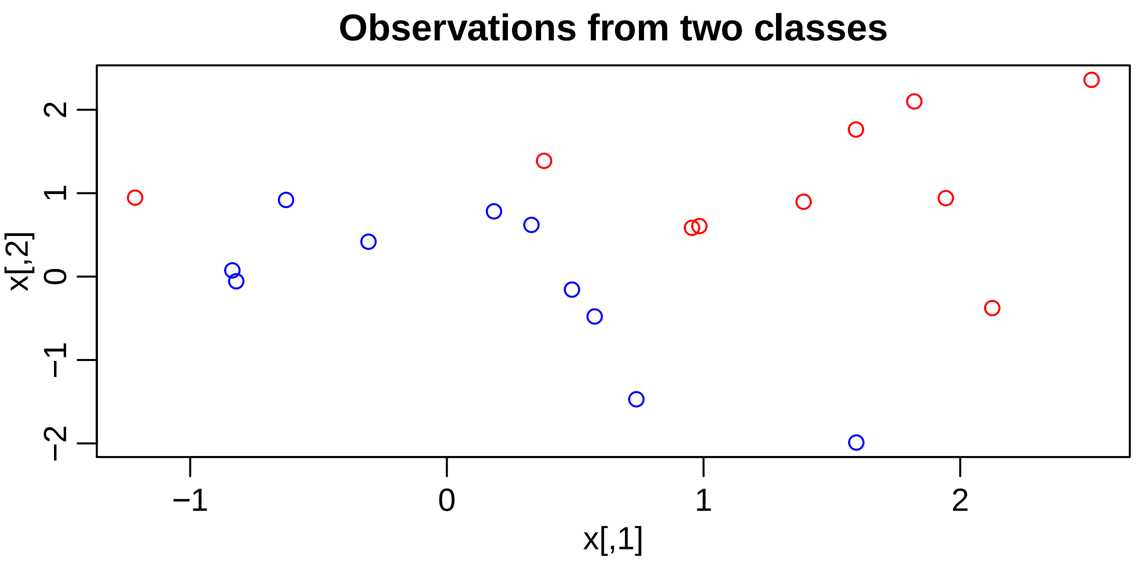
Support Vector Machine

Support Vector Classifier (*soft margin classifier*)

A two-dimensional example

We first generate the observations, which belongs to two classes, and checking whether the two 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
par(mfrow=c(1, 1), mar=c(2.3, 2.1, 1.5, 0) + .5, mgp=c(1.6, .6, 0))
plot(x, col=(3 - y), main='Observations from two classes')
```



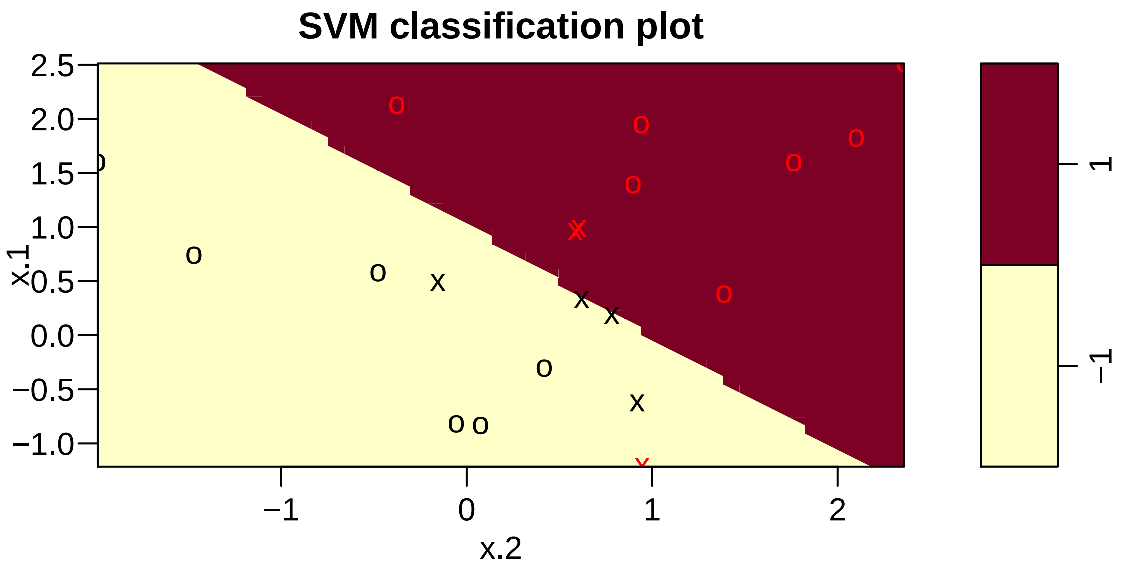
The observations generated are not linear separable. Next, we fit the support vector classifier.

```
dat <- data.frame(x=x, y=as.factor(y))
svmfit <- svm(y ~ ., data=dat , kernel ="linear", cost=10, scale=FALSE)
```

Note. The argument `scale` = `FALSE` tells the `svm`() function not to scale each feature to have mean zero or standard deviation one.

Now, we plot the support vector classifier obtained below.

```
par(mfrow=c(1, 1), mar=c(2, 2, 1.5, 0) + .5, mgp=c(1.6, .6, 0))
plot(svmfit, dat)
```



The separate regions of feature space are assigned to the $+1$ and -1, respectively.

The decision boundary between the two classes is linear (because we used the argument `kernel` = `"linear"`), though due to the way in which the plotting function is implemented in this library the decision boundary looks somewhat jagged in the plot.

The support vectors are plotted as crosses and the remaining observations are plotted as circles; we see here that there are seven support vectors, which lie directly on the margin, or on the wrong side of the margin for their class.

One can determine their identities as follows.

```
svmfit$index
```

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

Moreover, one can obtain some basic information about the support vector classifier fit using the `summary` command.

```
summary(svmfit)
```

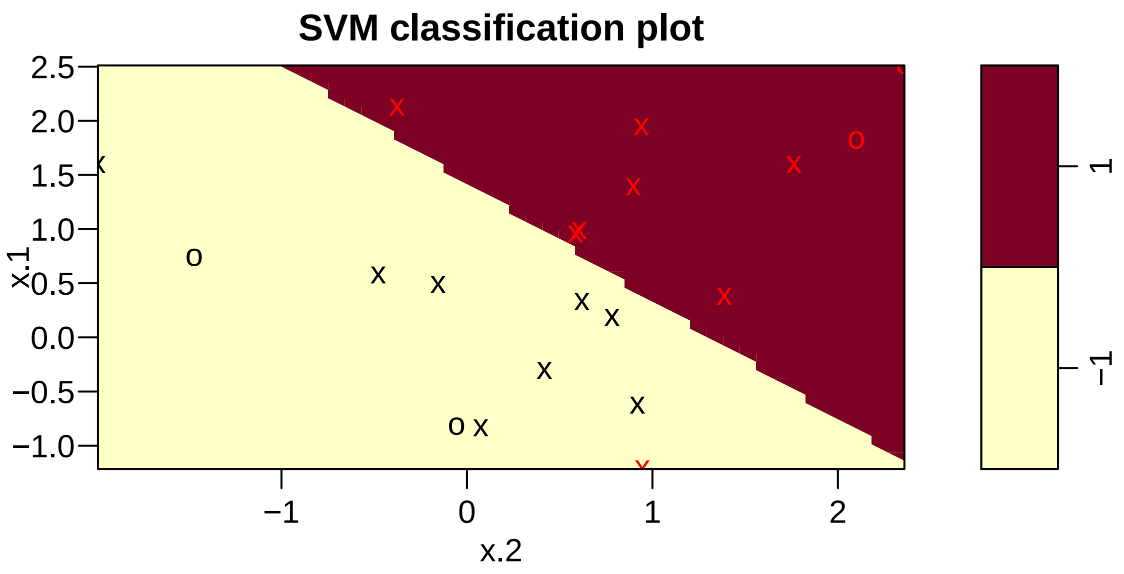
```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10, scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1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

The summary above tell us, for instance, that a linear kernel was used with `cost` = `10`, and that there were seven support vectors, four in one class and three in the other.

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

Here, a linear kernel was used with `cost` = 0.1.

```
svmfit <- svm(y ~ ., data=dat , kernel ="linear", cost =0.1, scale=FALSE)
par(mfrow=c(1, 1), mar=c(2, 2, 1.5, 0) + .5, mgp=c(1.6, .6, 0))
plot(svmfit, dat)
```



With a smaller value of the cost parameter being used, we obtain a larger number of support vector, because the margin is now wider compared to the previous one.

Cross Validation

Here, we use the built-in function, `tune`(), to perform cross-validation using a range of values of the `cost` parameters.

```
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
```

The model with `cost` = 0.1 results in the lowest cross-validation error rate, which can be accessed as follows.

```
bestmod <- tune.out$best.model
summary(bestmod)
```

```
##
## Call:
## best.tune(method = svm, train.x = y ~ ., data = dat, ranges = list(cost = c(0.001,
##     0.01, 0.1, 1, 5, 10, 100)), kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.1
##
## Number of Support Vectors:  16
##
##  ( 8 8 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

Prediction

Here we use the function `predict`() to predict the class label on a set of test observations. We first generate a test data set.

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

Here we use the best model obtained through cross-validation in order to make predictions.

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

```
##        truth
## predict -1 1
##      -1  9 3
##      1   2 6
```

Thus, with `cost` = 0.1, 19 of the test observations are correctly classified. What if we had instead used `cost` = 0.01?

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

```
##        truth
## predict -1  1
##      -1 10  4
##      1   1  5
```

In this case, one addition observation from `class` = −1 is correctly classified and one additional observation from `class` = 1 is mis-classified.

Support Vector Machine (*non-linear classifier*)

The linear classifier is a natural approach for classification if the boundary between the two classes is linear. However, in practice we are often faced with non-linear class boundaries.

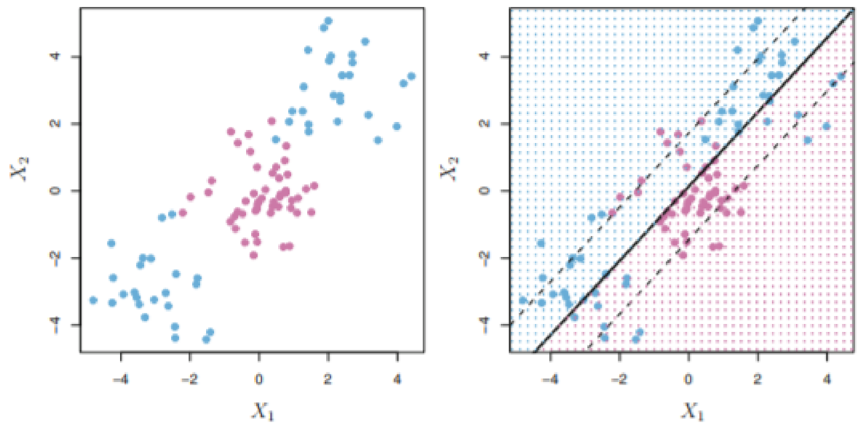


Figure 1: Left: The observations fall into two classes, with a non-linear boundary between them. Right: The support vector classifier seeks a linear boundary, and consequently performs very poorly.

We may want to enlarge our feature space in order to accommodate a non-linear boundary between the classes.

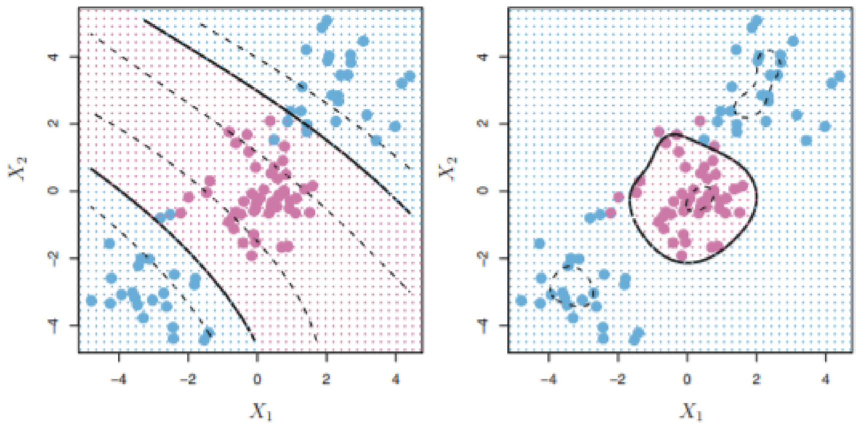
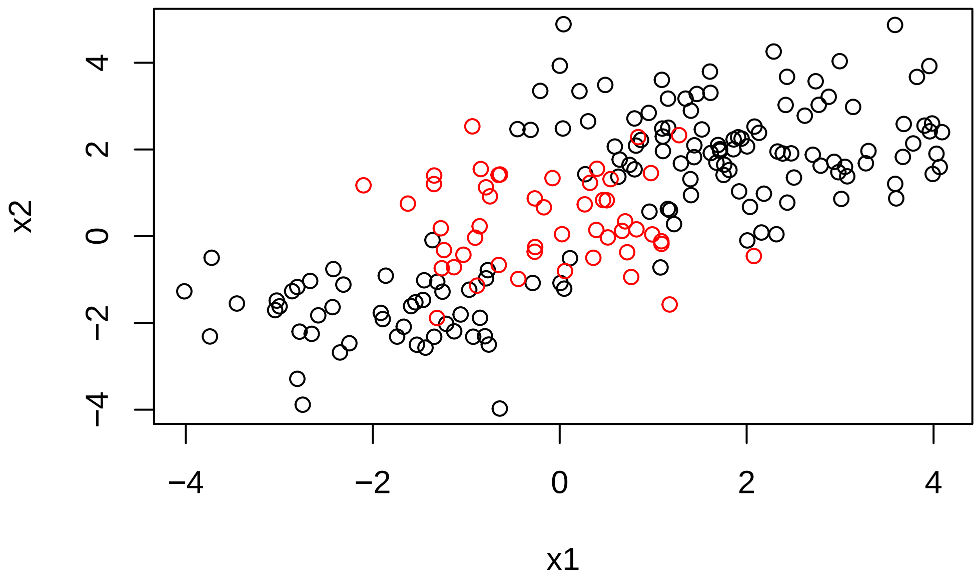


Figure 2: Left: An SVM with a polynomial kernel of degree 3 is applied, resulting in a far more appropriate decision rule. Right: An SVM with a radial kernel is applied. In this example, either kernel is capable of capturing the decision boundary.

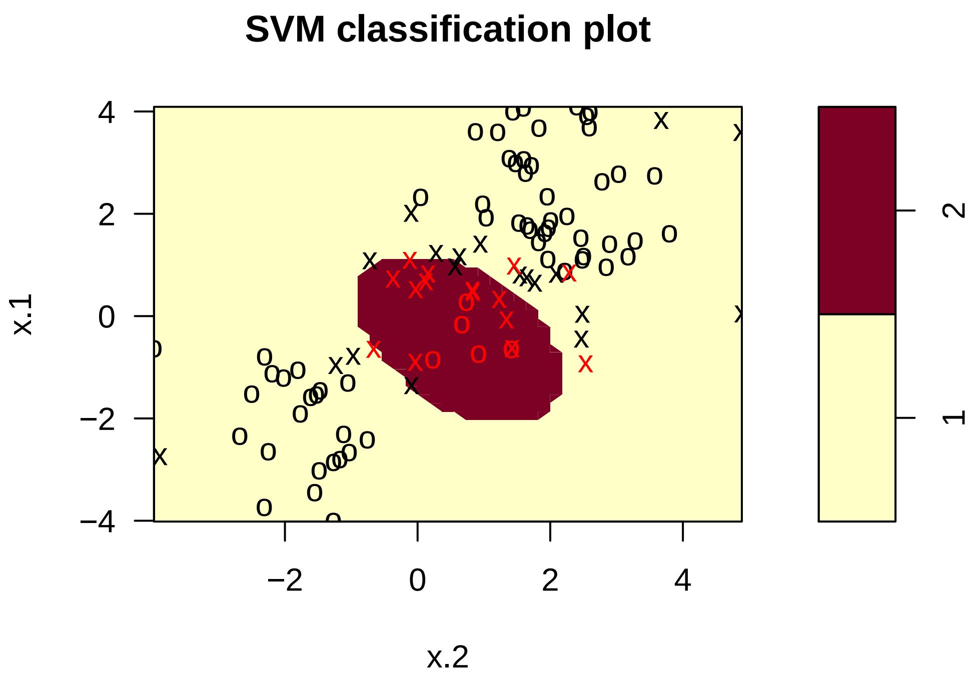
We first generate some data with a non-linear class boundary.

```
set.seed(2)
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, xlab = "x1", ylab = "x2")
```



The data is randomly split into training and testing groups. 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. We then fit the training data using the `svm()` function with a radial kernel and $\gamma = 1$:

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



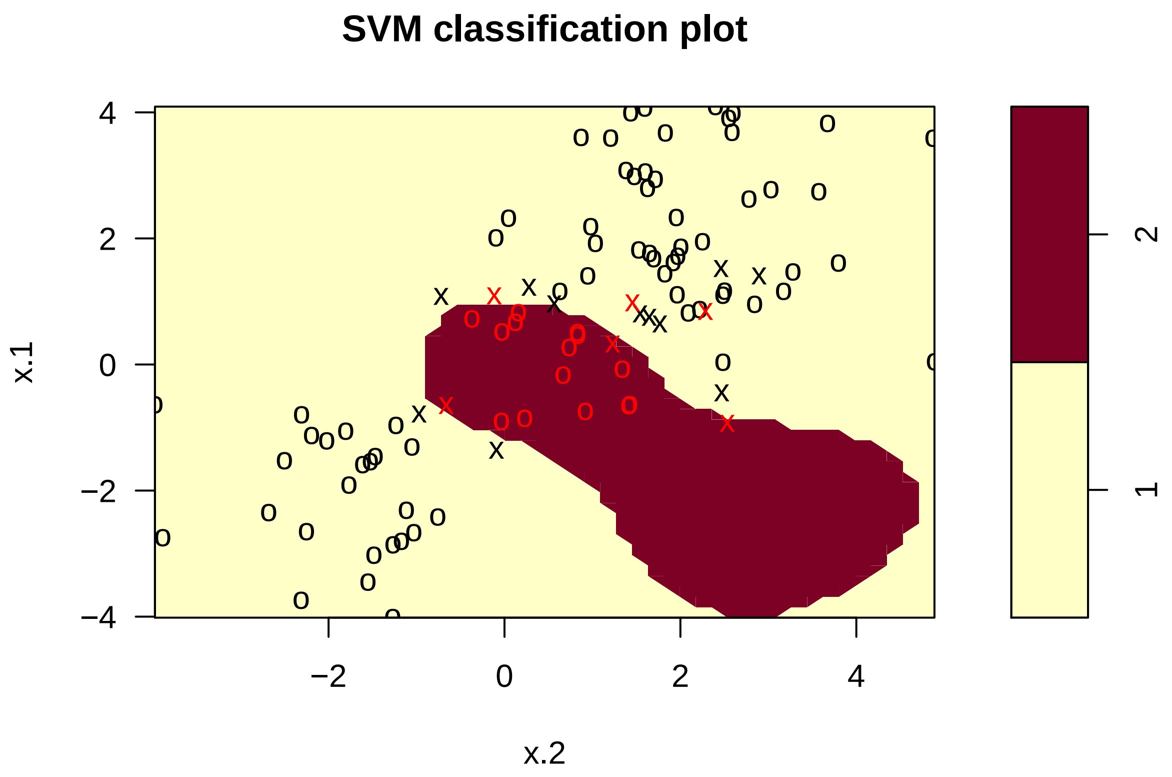
The `summary()` function can be used to obtain some information about the SVM fit:

```
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat[train, ], kernel = "radial", gamma = 1,
##     cost = 1)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  34
##
##  ( 19 15 )
##
##
## Number of Classes:  2
##
## Levels:
##  1 2
```

We can see from the figure that there are a fair number of training errors in this SVM fit. If we increase the value of cost, we can reduce the number of training errors. However, this comes at the price of a more irregular decision boundary that seems to be at risk of overfitting the data.

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



We can perform cross-validation using `tune()` to select the best choice of γ and cost for an SVM with a radial kernel:

```
set.seed(1)
tune.out = tune(svm, y ~ ., data = dat[train ,], kernel ="radial",
                ranges = list(cost=c(0.01, 0.1, 1, 10, 100), gamma = c(0.5,1,2,3)))
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##    10     3
##
## - best performance: 0.07
##
## - Detailed performance results:
##     cost gamma error dispersion
## 1  1e-02   0.5  0.20 0.10540926
## 2  1e-01   0.5  0.20 0.10540926
## 3  1e+00   0.5  0.09 0.09944289
## 4  1e+01   0.5  0.10 0.10540926
## 5  1e+02   0.5  0.09 0.09944289
## 6  1e-02   1.0  0.20 0.10540926
## 7  1e-01   1.0  0.20 0.10540926
## 8  1e+00   1.0  0.10 0.09428090
## 9  1e+01   1.0  0.08 0.10327956
## 10 1e+02   1.0  0.10 0.09428090
## 11 1e-02   2.0  0.20 0.10540926
## 12 1e-01   2.0  0.20 0.10540926
## 13 1e+00   2.0  0.09 0.09944289
## 14 1e+01   2.0  0.08 0.10327956
## 15 1e+02   2.0  0.09 0.08755950
## 16 1e-02   3.0  0.20 0.10540926
## 17 1e-01   3.0  0.20 0.10540926
## 18 1e+00   3.0  0.08 0.09189366
## 19 1e+01   3.0  0.07 0.09486833
## 20 1e+02   3.0  0.11 0.08755950
```

Therefore, the best choice of parameters involves `cost = 1` and `gamma = 2`. We can view the test set predictions for this model by applying the `predict()` function to the data.

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

```
##     pred
## true  1  2
##    1 68  2
##    2 15 15
```