Week 8 Lecture - Support Vector Machines

Unit Co-ordinator: Dr. Liwan Liyanage

School of Computer, Data and Mathematical Sciences

WESTERN SYDNEY UNIVERSITY

Support Vector Machines

Here we approach the ***two-class classification problem*** in a direct way:

We try and find a plane that seperates the classes in feature space.

If we cannot, we get creative in two ways:

- We ***soften*** what we mean by “seperate” and
- We enrich and ***enlarge the feature space*** so that seperation is possible.

Outline

- ***Maximal Margin Classifier*** : requires that the classes be seperable by a linear boundary
- ***Support Vector Classifier*** : extension of the maximal margin classifier that can be applied in a broader range of cases
- ***Support Vector Machine*** : accommodate ***non-linear class boundaries***

Maximal Margin Classifier

What is Hyperplane?

A hyperplane in *p* dimensions is a flat affine subspace of dimension *p-1*.

- In two dimensions, a hyperplane is a flat one-dimensional subspace-in other words, a *line*.
- In three dimensions, a hyperplane is a flat two-dimensional subspace; that is *plane*.
- In *p>3* dimensions, it can be hard to visualize a hyperplane.

WESTERN SYDNEY UNIVERSITY

Maximal Margin Classifier: Mathematical definition of a hyperplane

In two dimensions, a hyperplane is defined by the equation $\beta_0 + \beta_1 X_1 + \beta_2 X_2 = 0$ for parameters $\beta_0, \beta_1 and \beta_2$

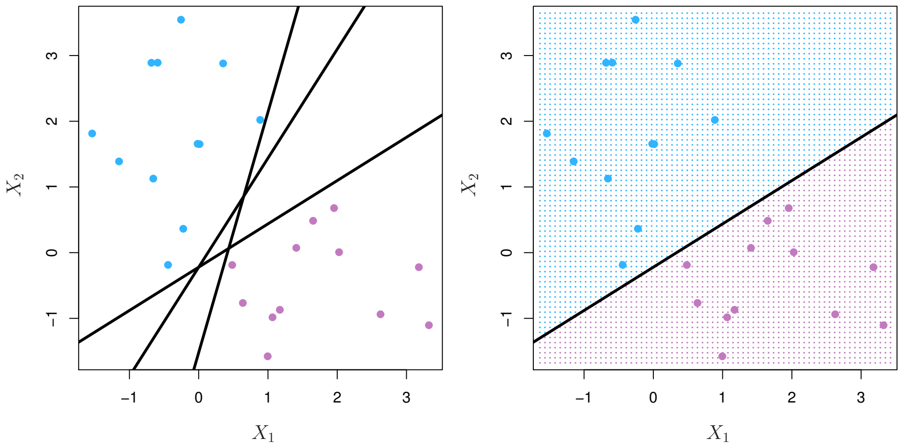
In general extended to the p-dimensional setting the equation for a hyperplane has the form:

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

- If a point $X = (X_1, X_2, X_3, ..., X_p)^T$ in p-dimensional space (i.e. a vector of length $\boldsymbol{p}$) satisfies the above equation, then X lies on the hyperplane
- If $\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p > 0$, then this tells us that X lies to ***one side*** of the hyperplane
- If $\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p < 0$, then X lies on the ***other side*** of the hyperplane.

NOTE:

- One can easily determine on which side of the hyperplane a point lies by simply calculating the sign of $\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p$



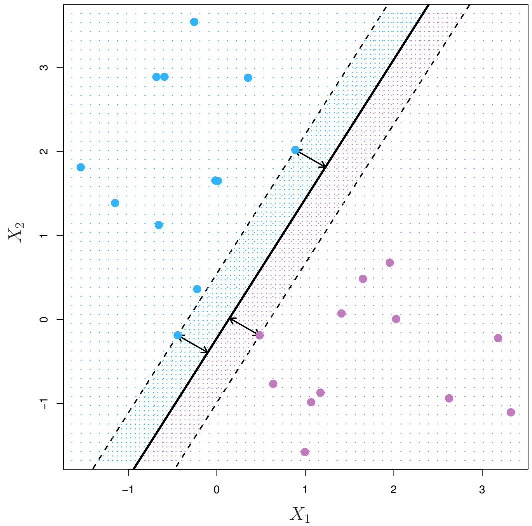
Left: Three seperating hyperplanes out of many

Right: The blue and purple grid indicates the decision rule made by a classifier based on a hyperplane

Maximal Margin Classifier: (also known as the optimal separating hyperplane)

- Is the ***separating hyperplane*** that is ***farthest*** from the training observations
- That is, we can compute the (perpendicular) distance from each training observation to a given seperating hyperplane; the smallest such distance is the minimal distance from the observations to the hyperplane, and is known as the ***margin***.
- The maximal margin hyperplane is the separating hyperplane for which the margin is largest-that is, it is the hyperplane that has the farther minimum distance to the training observations.

WESTERN SYDNEY UNIVERSITY


X_1
X_2
−1
0
1
2
3
−1
0
1
2
3

- We can then classify a test observation based on which side of the maximal margin plane it lies. This is known as the ***maximal margin classifier***.
- We hope that a classifier that has a large margin on the training data will also have a large margin on the test data, and hence will classify the test observations correctly.
- Although the maximal margin classifier is often successful, it can also lead to overfitting when p is large.

- In a sense, the maximal margin hyperplane represents the mid-line of the widest “slab” that we can insert between the two classes.
- The maximal margin classifier is a very natural way to perform classification, if a separating hyperplane exists.
- If no separating hypeplane exists, then there is no maximal margin classifier. In this case, we cannot exactly separate the two classes.
- Then we can extend the concept of a separating hyperplane in order to develop a hyperplane that ***almost separates the classes***, using a so-called ***soft-margin*** (***support vector classifier***).

Support Vector Classifier

- The generalization of the maximal margin classifier to the non-separable case is known as the Support Vector Classifier.
- We use the **e1071** library in R to demonstrate the support vector classifier and the SVM.

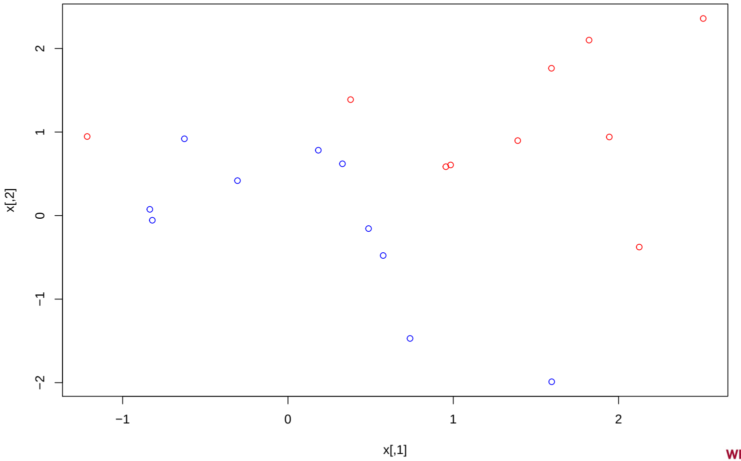
```
install.packages(e1071)
```

we begin by generating the observations, which belong to two classes.

```
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 begin by checking whether the classes are linearly separable.



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.

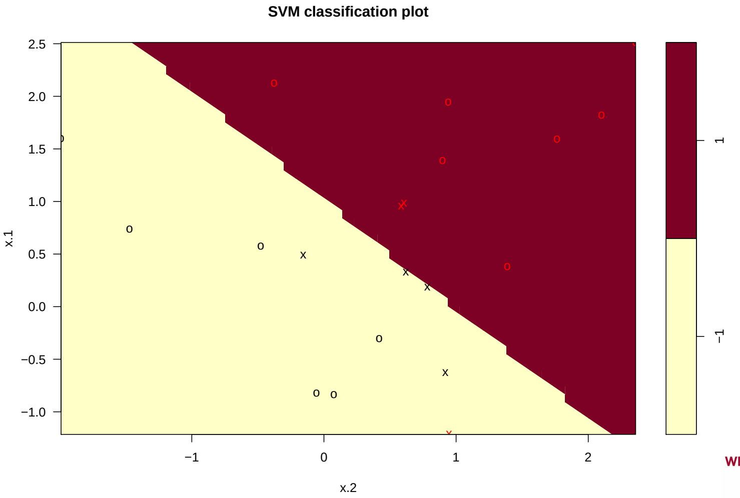
```
dat=data.frame(x=x, y=as.factor(y))
library(e1071)
svmfit1=svm(y~., data = dat, kernel = "linear",
           cost = 10, scale = FALSE)
```

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

WESTERN SYDNEY UNIVERSITY

We can now plot the support vector classifier obtained

```r
plot(svmfit1,dat)
```



We can determine their identities as follows:

The ***support vectors*** are plotted as *crosses* and the remaining observations are plotted as *circles*

```
svmfit1$index
```

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

We can obtain some basic information about the support vector classifier by using the summary() command:

```
summary(svmfit1)
```

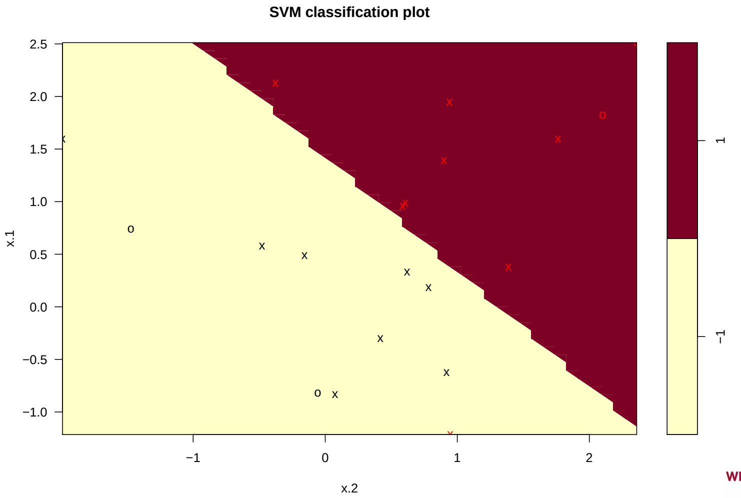
```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10,
##     scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
##
```

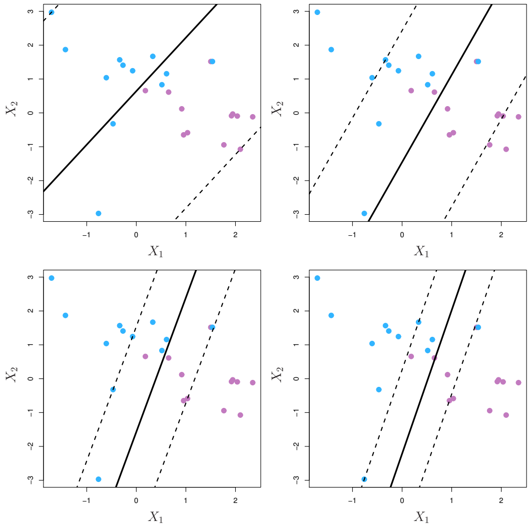
What if we instead used a smaller value of the cost parameter?

```r
svmfit2=svm(y~., data=dat, kernel="linear",
           cost=0.1, scale = FALSE)
summary(svmfit2)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 0.1,
##     scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.1
##
## Number of Support Vectors:  16
##
##  ( 8 8 )
##
```

```
plot(svmfit2, dat)
```





Support vector classifier was fitted using four different values of the tuning parameter c. (largest c in the top left whereas lowest c in the bottom right)

The following command indicates that we want to compare SVMs with a linear kernel, using a range of values of the cost parameter.

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.

```
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)) )
```

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

```
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
```

We see that ***cost = 0.1*** results in the lowest cross-validation error rate. The ***tune()*** function stores the best model obtained, which can be accessed as follows:

```r
best_model = tune_out$best.model
summary(best_model)
```

```
##
## 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
```

Predicting the class labels

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 a test data set.

```
x_test = matrix(rnorm(20*2), ncol = 2)
y_test = sample(c(-1,1), 20, rep = TRUE)
x_test[y_test==1, ] = x_test[y_test ==1, ]+1
test_dat = data.frame(x=x_test, y=as.factor(y_test))
```

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.

```
y_predict = predict(best_model, test_dat)
table(predict = y_predict, truth = test_dat$y)
```

```
##        truth
## predict -1 1
##      -1  9 1
##      1   2 8
```

What if we had instead used ***cost = 0.01***?

```r
svmfit3 = svm(y~., data = dat, kernel = "linear",
              cost = 0.01, scale = FALSE)
y_predict = predict(svmfit3, test_dat)
table(predict = y_predict, truth = test_dat$y)
```

```
##        truth
## predict -1  1
##      -1 11  6
##      1   0  3
```

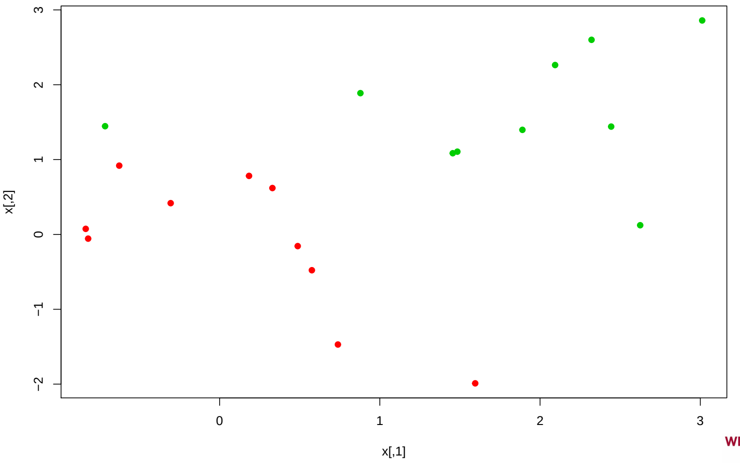
In this case, more observations are misclassified.

Now consider a situation in which the two classes are linearly seperable

Then, we can find a seperating hyperplane using the ***svm()*** function. We first further seperate the two classes in our simulated data so that they are linearly seperable:

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

Plot



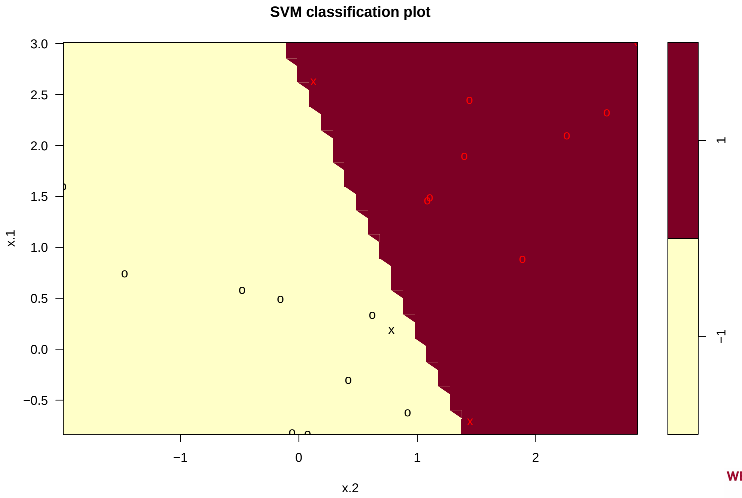
WESTERN SYDNEY UNIVERSITY

We fit the support vector classifier and plot the resulting hyperplane, using a very large value of cost so that no observations are misclassified.

```r
dat = data.frame(x=x, y= as.factor(y))
svmfit4=svm(y~., data = dat, kernel = "linear",
           cost = 1e5)
summary(svmfit4)
```

```
##
## 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
```

```r
plot(svmfit4,dat)
```



```r
#predicting responses for test data
y_predict = predict(svmfit4, test_dat)
table(predict = y_predict, truth = test_dat$y)
```

```
##         truth
## predict -1 1
##      -1  9 4
##      1   2 5
```

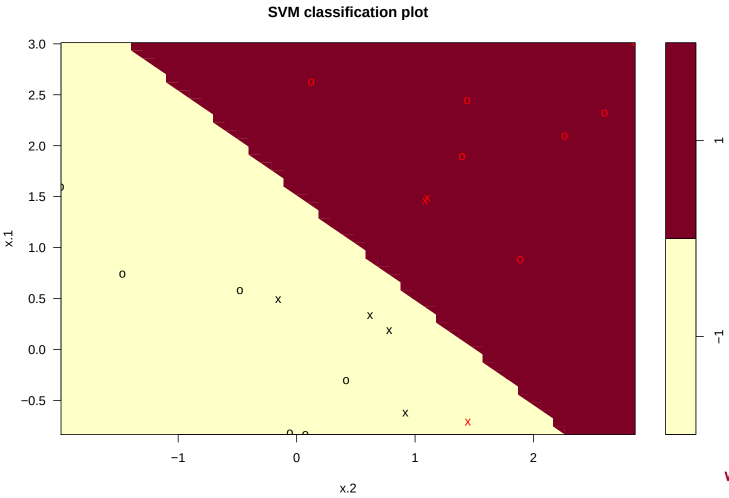
SVM with smaller value of cost function

We now try a smaller value of cost.

```r
svmfit5=svm(y~., data = dat, kernel = "linear",
          cost = 1)
summary(svmfit5)
```

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

```
plot(svmfit5,dat)
```



```r
#predicting responses for test data
y_predict = predict(svmfit5, test_dat)
table(predict = y_predict, truth = test_dat$y)
```

```
##         truth
## predict -1 1
##      -1  9 2
##      1   2 7
```

Support Vector Machine

In order to fit a 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 a SVM with a ***polynomial kernel***, we use "**kernel = polynomial**"
- To fit a 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 (this is **d**), and in the latter case, we use ***gamma*** to specify a value of γ for the radial basis kernel.

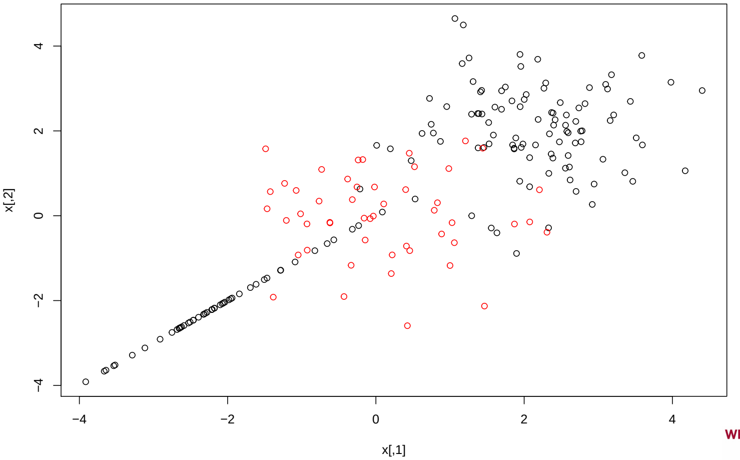
WESTERN SYDNEY UNIVERSITY

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

```r
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))
```

```r
plot(x, col = y)
```

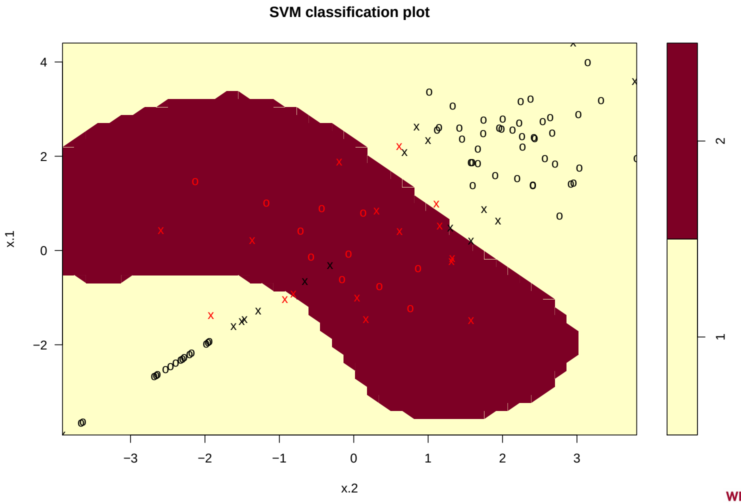
Plotting the data makes it clear that the class boundary is indeed non-linear;



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 = sample(200, 100)
svmfit6 = svm(y~. , data = dat[train, ],
             kernel = "radial", gamma = 1, cost = 1)
plot(svmfit6, dat[train, ])
```

SVM Classification:



```r
#predicting responses for test data
y_predict = predict(svmfit6, dat[-train, ])
table(predict = y_predict, truth = dat[-train, ]$y)
```

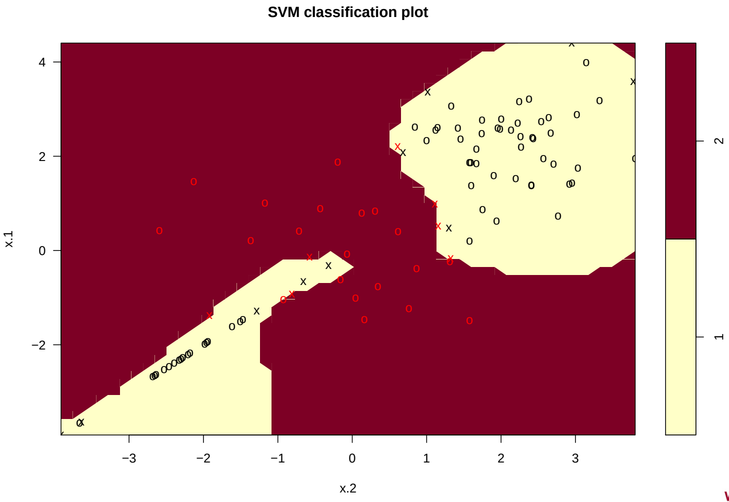
```
##        truth
## predict  1  2
##       1 65  3
##       2 12 20
```

Increasing the cost

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 error. However, this comes at the price of a more irregeular decision boundary.

```
svmfit7 = svm(y~., data = dat[train, ],
             kernel = "radial", gamma = 1, cost = 1e5)
plot(svmfit7, dat[train, ])
```

SVM Classification:



```r
#predicting responses for test data
y_predict = predict(svmfit7, dat[-train, ])
table(predict = y_predict, truth = dat[-train, ]$y)
```

```
##        truth
## predict  1  2
##       1 63  3
##       2 14 20
```

Support Vector Machine: cross-validation

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_out2 = 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_out2)
```

```
##
## 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  0.27 0.15670212
## 2  1e+00   0.5  0.07 0.08232726
## 3  1e+01   0.5  0.08 0.07888106
## 4  1e+02   0.5  0.11 0.07378648
## 5  1e+03   0.5  0.12 0.10327956
## 6  1e-01   1.0  0.26 0.15776213
## 7  1e+00   1.0  0.08 0.07888106
## 8  1e+01   1.0  0.10 0.08164966
## 9  1e+02   1.0  0.15 0.10801234
## 10 1e+03   1.0  0.15 0.13540064
## 11 1e-01   2.0  0.26 0.15776213
## 12 1e+00   2.0  0.09 0.07378648
## 13 1e+01   2.0  0.11 0.07378648
## 14 1e+02   2.0  0.18 0.13984118
## 15 1e+03   2.0  0.14 0.15776213
## 16 1e-01   3.0  0.27 0.15670212
## 17 1e+00   3.0  0.09 0.07378648
## 18 1e+01   3.0  0.10 0.09428090
## 19 1e+02   3.0  0.16 0.15055453
## 20 1e+03   3.0  0.15 0.15092309
## 21 1e-01   4.0  0.27 0.15670212
## 22 1e+00   4.0  0.09 0.07378648
## 23 1e+01   4.0  0.12 0.10327956
```

Support Vector Machine: prediction

We can view the test set prediction 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(true=dat[-train,"y"],
        pred=predict(tune_out2$best.model))
```

```
##      pred
## true  1  2
##    1 54 23
##    2 17  6
```

40% of test observations are misclassified by this SVM.

ROC curves and AUC

- ROC (Receiver Operating Characteristics) curve is plotted with TPR (True Positive Rate) against the FPR (False Positive Rate).
- This is frequently used to show the trade-off between sensitivity (TPR) and specificity (TNR or 1-FPR).
- ROC-AUC(area under the curve) represents degree or measure of separability. It tells how much model is capable of distinguishing between classes.
- The ROCR package can be used to produce ROC curves.
- Higher the AUC, better the model

We first write a short function to plot an ROC curve given a vector containing a numerical score for each observation, **pred**, and a vector containing the class label for each observation, **truth**.

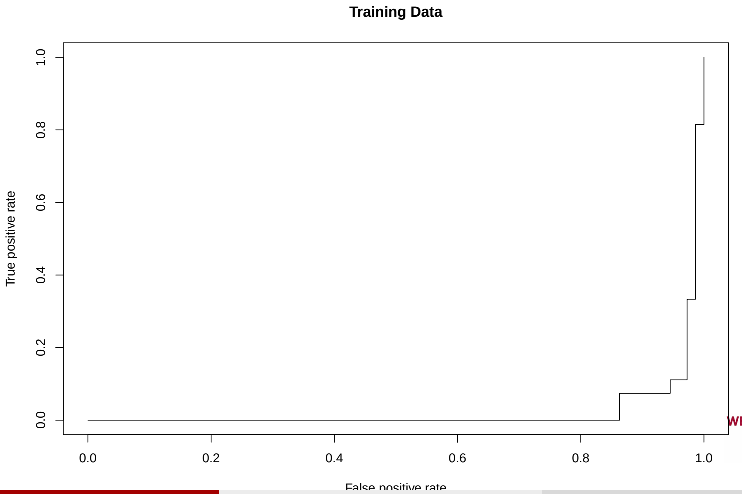
```r
library(ROCR)
rocplot = function(pred, truth, ...){
predob = prediction(pred, truth)
perf = performance (predob, "tpr", "fpr")
plot(perf, ...)
}
```

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.

```r
svmfit_opt = svm(y~., data = dat[train, ], kernel = "radial",
                 gamma = 0.5, cost = 1, decision.values = TRUE)
fitted = attributes(predict(svmfit_opt, dat[train, ],
                            decision.values = TRUE))$decision.values
```

Now we can plot ROC for Training Data using Optimal Model creates above (svmfit_opt)

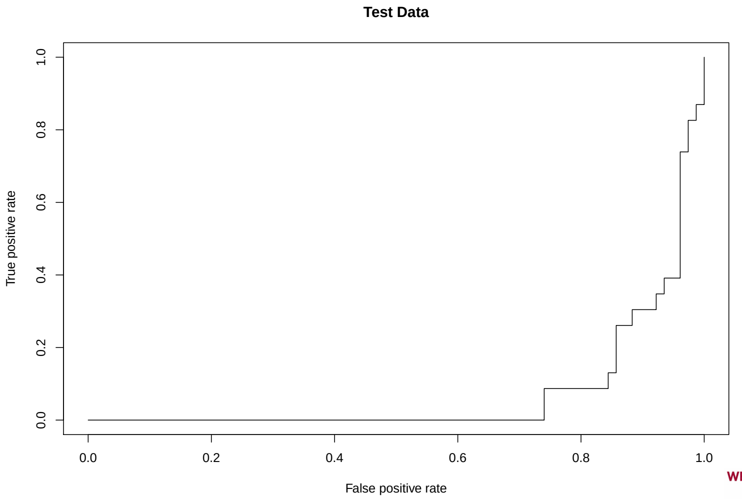
```r
p1 <- rocplot(fitted, dat[train, "y"], main = "Training Data")
```



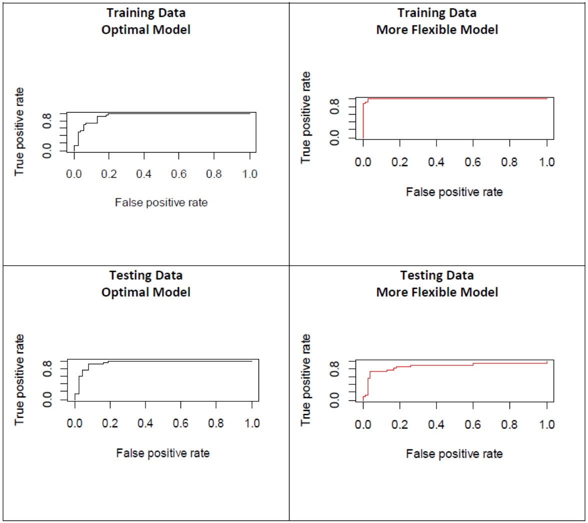
ROC Plot for Test Data using Optimal Model

```
fitted = attributes(predict(svmfit_opt, dat[-train,],
                            decision.values=T))$decision.values
```

```r
p3 <- rocplot(fitted, dat[-train,"y"], main="Test Data")
```



ROC curves: example



TEXT BOOK

Lecture notes are based on the textbook.

For further reference refer;

Prescribed Textbook - Chapter 9

- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications* in R Springer.