R Notebook

***5. We have seen that we can fit an SVM with a non-linear kernel in order to perform classification using a non-linear decision boundary. We will now see that we can also obtain a non-linear decision boundary by performing logistic regression using non-linear transformations of the features.***

1. Generate a data set with n = 500 and p = 2, such that the observations belong to two classes with a quadratic decision boundary between them. For instance, you can do this as follows: > x1 <- runif (500) - 0.5 > x2 <- runif (500) - 0.5 > y <- 1 \* (x1^2 - x2^2 > 0)

set.seed(421)  
x1 = runif(500) - 0.5  
x2 = runif(500) - 0.5  
y = 1 \* (x1^2 - x2^2 > 0)

1. Plot the observations, colored according to their class labels. Your plot should display X1 on the x-axis, and X2 on the yaxis.

plot(x1[y == 0], x2[y == 0], col = "red", xlab = "X1", ylab = "X2", pch = "+")  
points(x1[y == 1], x2[y == 1], col = "blue", pch = 4)

The plot clearly shows non-linear decision boundary.

1. Fit a logistic regression model to the data, using X1 and X2 as predictors.

lm.fit = glm(y ~ x1 + x2, family = binomial)  
summary(lm.fit)

##   
## Call:  
## glm(formula = y ~ x1 + x2, family = binomial)  
##   
## Deviance Residuals:   
## Min 1Q Median 3Q Max   
## -1.278 -1.227 1.089 1.135 1.175   
##   
## Coefficients:  
## Estimate Std. Error z value Pr(>|z|)  
## (Intercept) 0.11999 0.08971 1.338 0.181  
## x1 -0.16881 0.30854 -0.547 0.584  
## x2 -0.08198 0.31476 -0.260 0.795  
##   
## (Dispersion parameter for binomial family taken to be 1)  
##   
## Null deviance: 691.35 on 499 degrees of freedom  
## Residual deviance: 690.99 on 497 degrees of freedom  
## AIC: 696.99  
##   
## Number of Fisher Scoring iterations: 3

Both variables are insignificant for predicting y.

1. Apply this model to the training data in order to obtain a predicted class label for each training observation. Plot the observations, colored according to the predicted class labels. The decision boundary should be linear

data = data.frame(x1 = x1, x2 = x2, y = y)  
lm.prob = predict(lm.fit, data, type = "response")  
lm.pred = ifelse(lm.prob > 0.52, 1, 0)  
data.pos = data[lm.pred == 1, ]  
data.neg = data[lm.pred == 0, ]  
plot(data.pos$x1, data.pos$x2, col = "blue", xlab = "X1", ylab = "X2", pch = "+")  
points(data.neg$x1, data.neg$x2, col = "red", pch = 4)

With the given model and a probability threshold of 0.5, all points are classified to single class and no decision boundary can be shown. Hence we shift the probability threshold to 0.52 to show a meaningful decision boundary. This boundary is linear as seen in the figure.

1. Now fit a logistic regression model to the data using non-linear functions of X1 and X2 as predictors (e.g. X2 1 , X1×X2, log(X2), and so forth).

We use squares, product interaction terms to fit the model.

lm.fit = glm(y ~ poly(x1, 2) + poly(x2, 2) + I(x1 \* x2), data = data, family = binomial)

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

1. Apply this model to the training data in order to obtain a predicted class label for each training observation. Plot the observations, colored according to the predicted class labels. The decision boundary should be obviously non-linear. If it is not, then repeat (a)-(e) until you come up with an example in which the predicted class labels are obviously non-linear.

lm.prob = predict(lm.fit, data, type = "response")  
lm.pred = ifelse(lm.prob > 0.5, 1, 0)  
data.pos = data[lm.pred == 1, ]  
data.neg = data[lm.pred == 0, ]  
plot(data.pos$x1, data.pos$x2, col = "blue", xlab = "X1", ylab = "X2", pch = "+")  
points(data.neg$x1, data.neg$x2, col = "red", pch = 4)

This non-linear decision boundary closely resembles the true decision boundary.

1. Fit a support vector classifier to the data with X1 and X2 as predictors. Obtain a class prediction for each training observation. Plot the observations, colored according to the predicted class labels.

library(e1071)

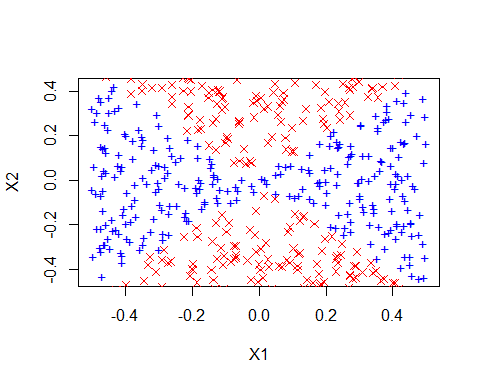
## Warning: package 'e1071' was built under R version 4.2.1

svm.fit = svm(as.factor(y) ~ x1 + x2, data, kernel = "linear", cost = 0.1)  
svm.pred = predict(svm.fit, data)  
data.pos = data[svm.pred == 1, ]  
data.neg = data[svm.pred == 0, ]  
plot(data.pos$x1, data.pos$x2, col = "blue", xlab = "X1", ylab = "X2", pch = "+")  
points(data.neg$x1, data.neg$x2, col = "red", pch = 4)

A linear kernel, even with low cost fails to find non-linear decision boundary and classifies all points to a single class.

1. Fit a SVM using a non-linear kernel to the data. Obtain a class prediction for each training observation. Plot the observations, colored according to the predicted class labels.

svm.fit = svm(as.factor(y) ~ x1 + x2, data, gamma = 1)  
svm.pred = predict(svm.fit, data)  
data.pos = data[svm.pred == 1, ]  
data.neg = data[svm.pred == 0, ]  
plot(data.pos$x1, data.pos$x2, col = "blue", xlab = "X1", ylab = "X2", pch = "+")  
points(data.neg$x1, data.neg$x2, col = "red", pch = 4)

 Again, the non-linear decision boundary on predicted labels closely resembles the true decision boundary.

1. Comment on your results.

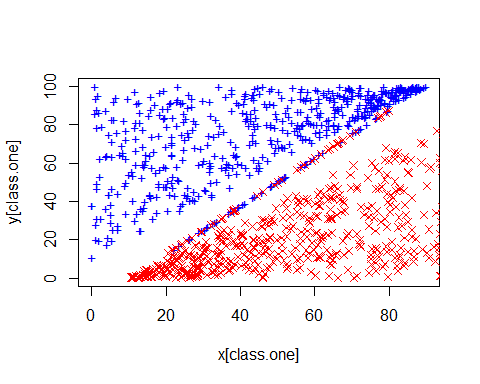
This experiment demonstrates the effectiveness of SVMs with non-linear kernels for locating non-linear boundaries. SVMs using linear kernels and logistic regression with no interactions both fall short in locating the decision border. Logistic regression appears to have the same power as radial-basis kernels when interaction factors are included. However, choosing the proper interaction terms requires some manual work and fine adjustment. With a lot of features, this endeavor may be impossible. On the other hand, radial basis kernels just need to have the gamma value tuned, which can be done quickly via cross-validation.

***6. At the end of Section 9.6.1, it is claimed that in the case of data that is just barely linearly separable, a support vector classifier with a small value of cost that misclassifies a couple of training observations may perform better on test data than one with a huge value of cost that does not misclassify any training observations. You will now investigate this claim.***

1. Generate two-class data with p = 2 in such a way that the classes are just barely linearly separable.

We randomly generate 1000 points and scatter them across line x=y with wide margin. We also create noisy points along the line 5x−4y−50=0. These points make the classes barely separable and also shift the maximum margin classifier.

set.seed(3154)  
# Class one  
x.one = runif(500, 0, 90)  
y.one = runif(500, x.one + 10, 100)  
x.one.noise = runif(50, 20, 80)  
y.one.noise = 5/4 \* (x.one.noise - 10) + 0.1  
  
# Class zero  
x.zero = runif(500, 10, 100)  
y.zero = runif(500, 0, x.zero - 10)  
x.zero.noise = runif(50, 20, 80)  
y.zero.noise = 5/4 \* (x.zero.noise - 10) - 0.1  
  
# Combine all  
class.one = seq(1, 550)  
x = c(x.one, x.one.noise, x.zero, x.zero.noise)  
y = c(y.one, y.one.noise, y.zero, y.zero.noise)  
  
plot(x[class.one], y[class.one], col = "blue", pch = "+", ylim = c(0, 100))  
points(x[-class.one], y[-class.one], col = "red", pch = 4)

 The plot shows that classes are barely separable. The noisy points create a fictitious boundary 5x−4y−50=0.

1. Compute the cross-validation error rates for support vector classifiers with a range of cost values. How many training errors are misclassified for each value of cost considered, and how does this relate to the cross-validation errors obtained?

We create a z variable according to classes.

library(e1071)

set.seed(555)  
z = rep(0, 1100)  
z[class.one] = 1  
data = data.frame(x = x, y = y, z = z)  
tune.out = tune(svm, as.factor(z) ~ ., data = data, kernel = "linear", ranges = list(cost = c(0.01,   
 0.1, 1, 5, 10, 100, 1000, 10000)))  
summary(tune.out)

##   
## Parameter tuning of 'svm':  
##   
## - sampling method: 10-fold cross validation   
##   
## - best parameters:  
## cost  
## 10000  
##   
## - best performance: 0   
##   
## - Detailed performance results:  
## cost error dispersion  
## 1 1e-02 0.05818182 0.01926091  
## 2 1e-01 0.04727273 0.01648663  
## 3 1e+00 0.04636364 0.01738137  
## 4 5e+00 0.05000000 0.01975516  
## 5 1e+01 0.04909091 0.01973190  
## 6 1e+02 0.04818182 0.02012359  
## 7 1e+03 0.04181818 0.03005963  
## 8 1e+04 0.00000000 0.00000000

data.frame(cost = tune.out$performances$cost, misclass = tune.out$performances$error \*   
 1100)

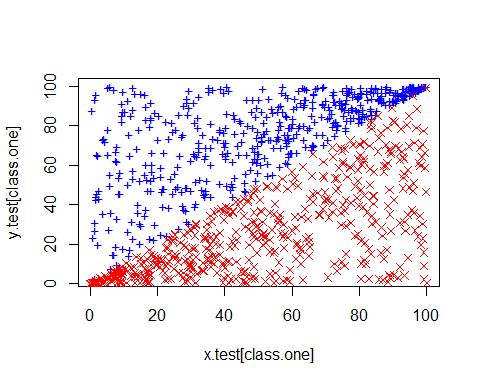
## cost misclass  
## 1 1e-02 64  
## 2 1e-01 52  
## 3 1e+00 51  
## 4 5e+00 55  
## 5 1e+01 54  
## 6 1e+02 53  
## 7 1e+03 46  
## 8 1e+04 0

The table above shows train-misclassification error for all costs. A cost of 10000 seems to classify all points correctly. This also corresponds to a cross-validation error of 0.

1. Generate an appropriate test data set, and compute the test errors corresponding to each of the values of cost considered. Which value of cost leads to the fewest test errors, and how does this compare to the values of cost that yield the fewest training errors and the fewest cross-validation errors?

We now generate a random test-set of same size. This test-set satisfies the true decision boundary x=y.

set.seed(1111)  
x.test = runif(1000, 0, 100)  
class.one = sample(1000, 500)  
y.test = rep(NA, 1000)  
# Set y > x for class.one  
for (i in class.one) {  
 y.test[i] = runif(1, x.test[i], 100)  
}  
# set y < x for class.zero  
for (i in setdiff(1:1000, class.one)) {  
 y.test[i] = runif(1, 0, x.test[i])  
}  
plot(x.test[class.one], y.test[class.one], col = "blue", pch = "+")  
points(x.test[-class.one], y.test[-class.one], col = "red", pch = 4)

 We now make same predictions using all linear svms with all costs used in previous part.

set.seed(30012)  
z.test = rep(0, 1000)  
z.test[class.one] = 1  
all.costs = c(0.01, 0.1, 1, 5, 10, 100, 1000, 10000)  
test.errors = rep(NA, 8)  
data.test = data.frame(x = x.test, y = y.test, z = z.test)  
for (i in 1:length(all.costs)) {  
 svm.fit = svm(as.factor(z) ~ ., data = data, kernel = "linear", cost = all.costs[i])  
 svm.predict = predict(svm.fit, data.test)  
 test.errors[i] = sum(svm.predict != data.test$z)  
}  
  
data.frame(cost = all.costs, `test misclass` = test.errors)

## cost test.misclass  
## 1 1e-02 51  
## 2 1e-01 14  
## 3 1e+00 6  
## 4 5e+00 0  
## 5 1e+01 0  
## 6 1e+02 183  
## 7 1e+03 205  
## 8 1e+04 205

cost=10 seems to be performing better on test data, making the least number of classification errors. This is much smaller than optimal value of 10000 for training data.

1. Discuss your results.

We observe overfitting for linear kernel once more. The train data is overfit because a huge cost tries to fit accurately classify noisy-points. However, a modest cost works better on test data and makes a few mistakes on the noisy test points.