Untitled

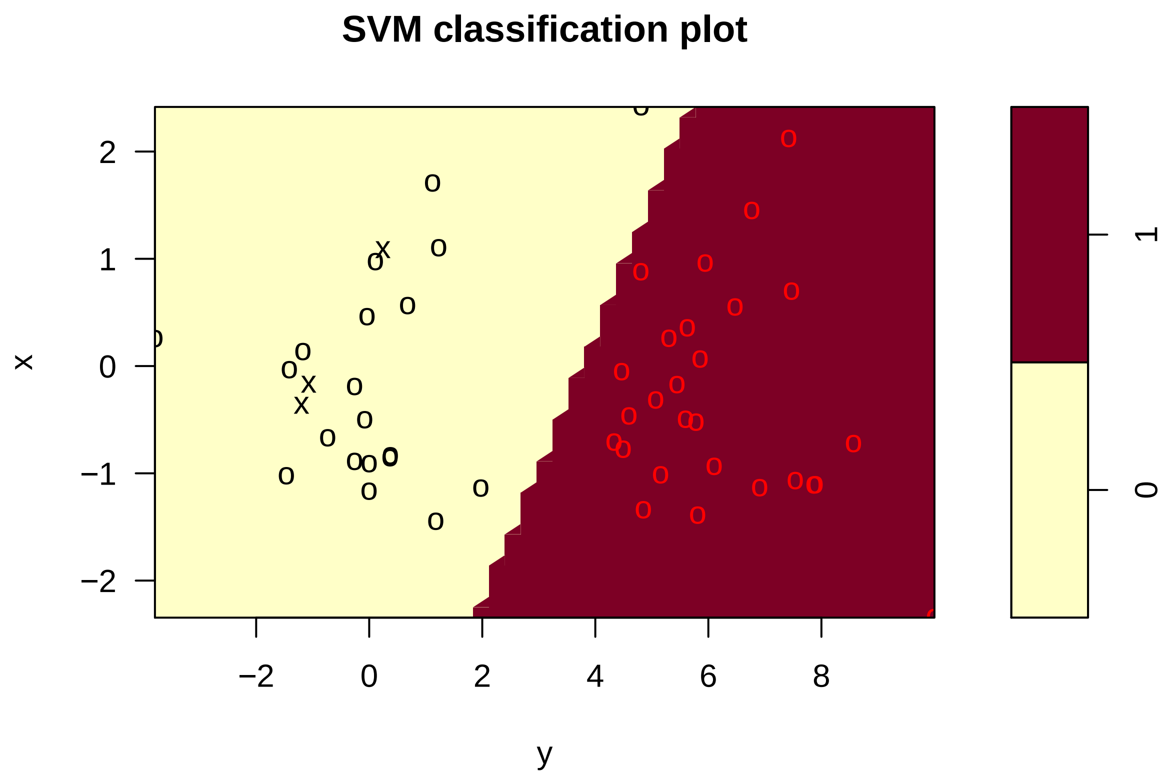
Yawei LI

03/07/2020

Non-linear Separation

Linear

```r
library(e1071)
set.seed(1234)
x = rnorm(100)
y = 2 + x^2 + rnorm(100)
class = sample(100, 50)
y[class] = y[class] + 3
y[-class] = y[-class] - 3
type = rep(0,100)
type[class] = 1
type = as.factor(type)
df = data.frame(x,y,type)
train_i = sample(100,50)
train = df[train_i,]
test = df[-train_i,]
linear_svm = svm(type~., data = df, kernel = "linear", cost = 10)
plot(linear_svm, train)
```

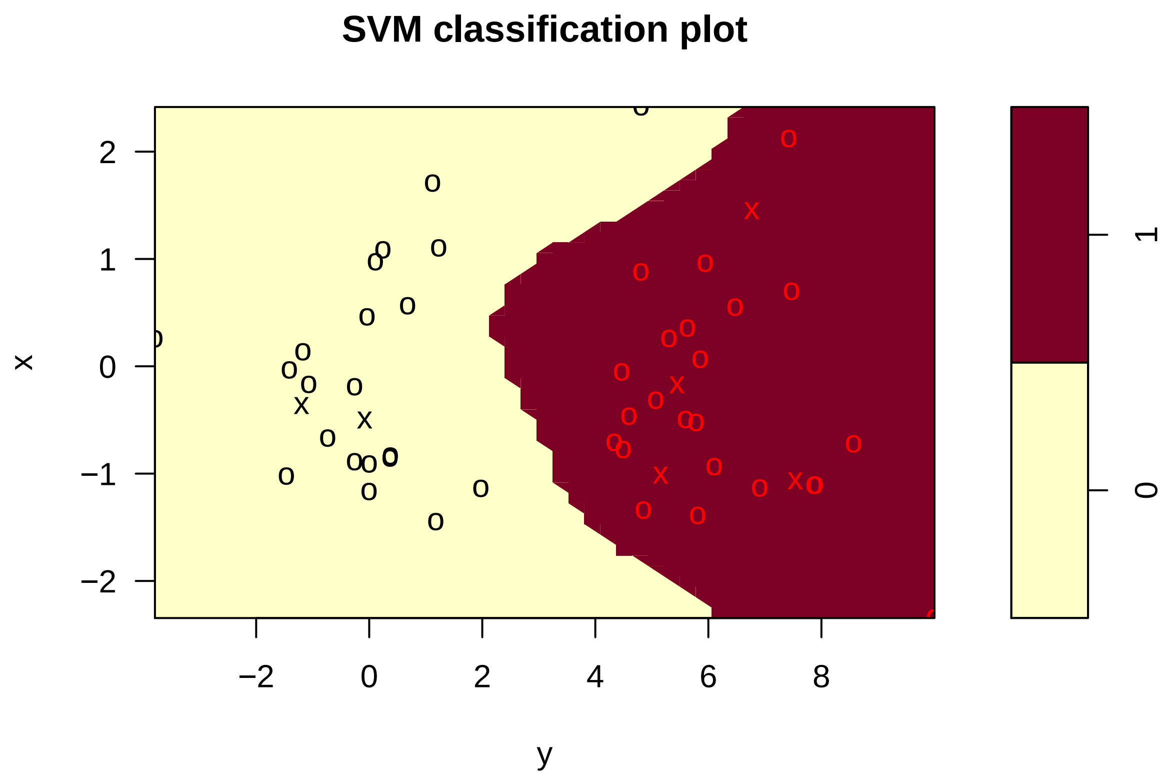


The Linear SVM has 0 error rate on training and 0.04 on test.

```
linear_p_train = predict(linear_svm,train)
linear_p_test = predict(linear_svm,test)
linear_er_train = sum(linear_p_train != train$type)/50
linear_er_test = sum(linear_p_test != test$type)/50
```

radial

```
radial_svm = svm(type~., data = df, kernel = "radial", cost = 10, gamma = 1)
plot(radial_svm, train)
```



The radial SVM has 0 error rate on training and 0 on test.

```
radial_p_train = predict(radial_svm,train)
radial_p_test = predict(radial_svm,test)
radial_er_train = sum(radial_p_train != train$type)/50
radial_er_test = sum(radial_p_test != test$type)/50
```

As can be seen, radial SVM yields a much lower error rate on test data and outperforms the linear SVM.

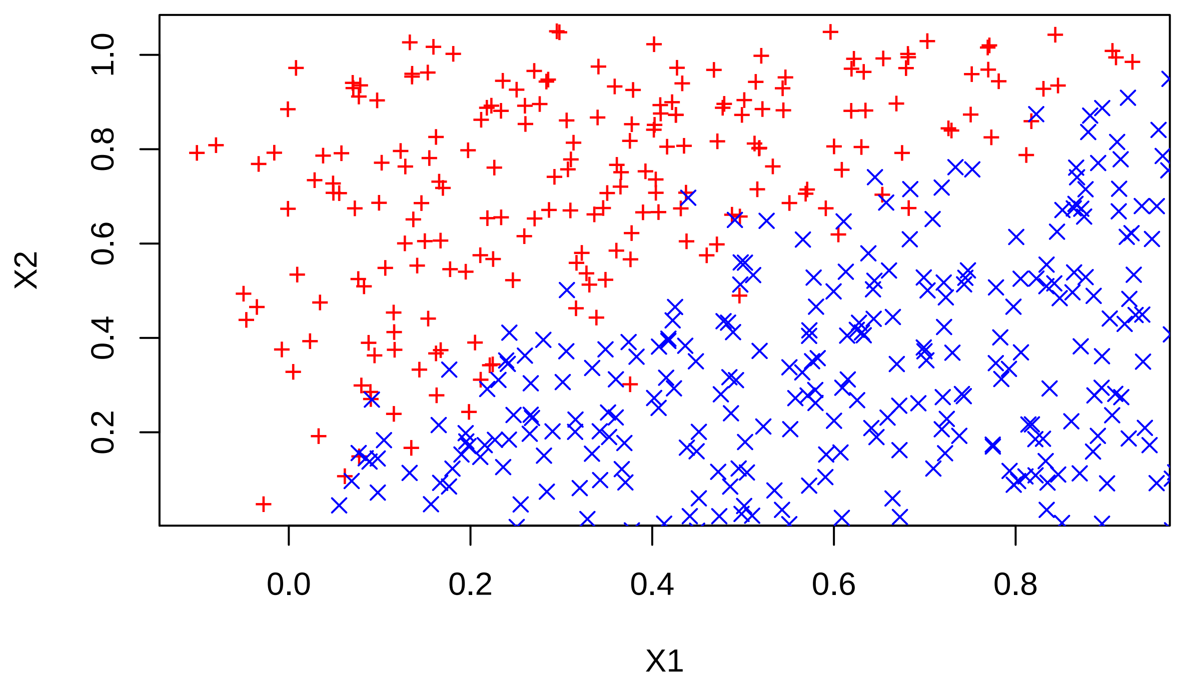
SVM vs Logistic

Generate Datasets

```
set.seed(2234)
x1 = runif(500)
x2 = runif(500)
y = 1 * (x1^3 - x2^4 > 0)
x1 = x1 + rnorm(500, sd = 0.05)
x2 = x2 + rnorm(500, sd = 0.05) # Creating overlapping
```

Ploting the obs

```
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)
```

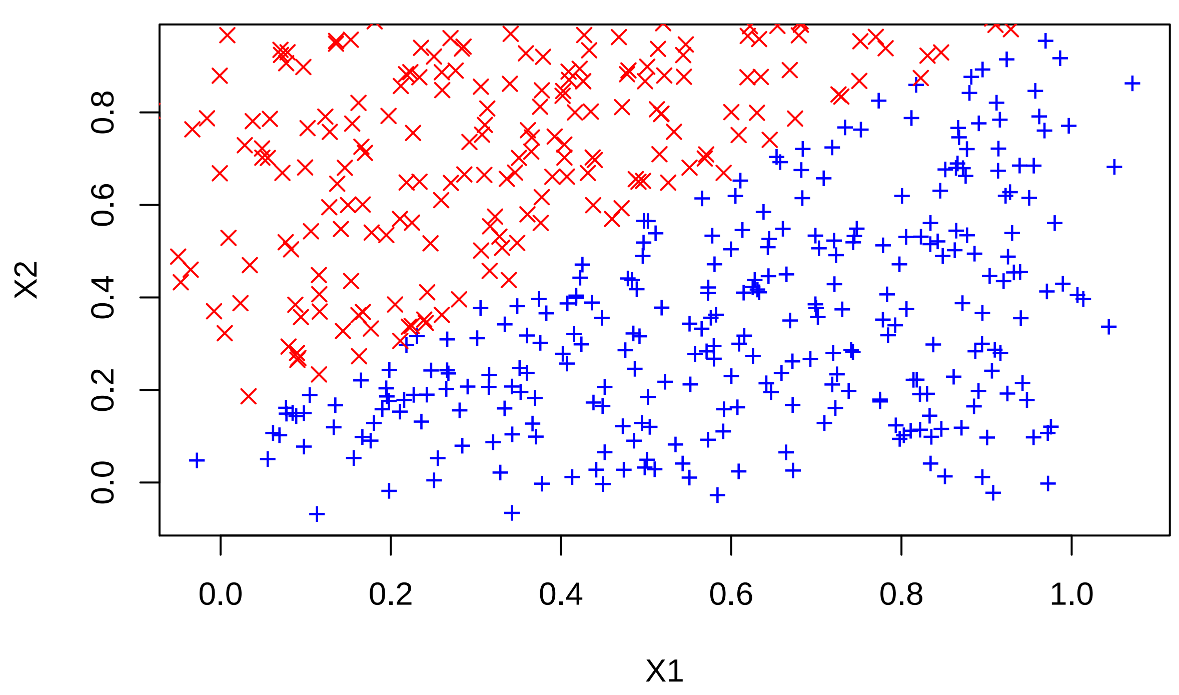


### Fitting Logistic

```
train_i = sample(500,250)
y = as.factor(y)
df = data.frame(x1,x2,y)
train = df[train_i,]
test = df[-train_i,]
logit = glm(y ~ x1 + x2, family = binomial, data = train)
```

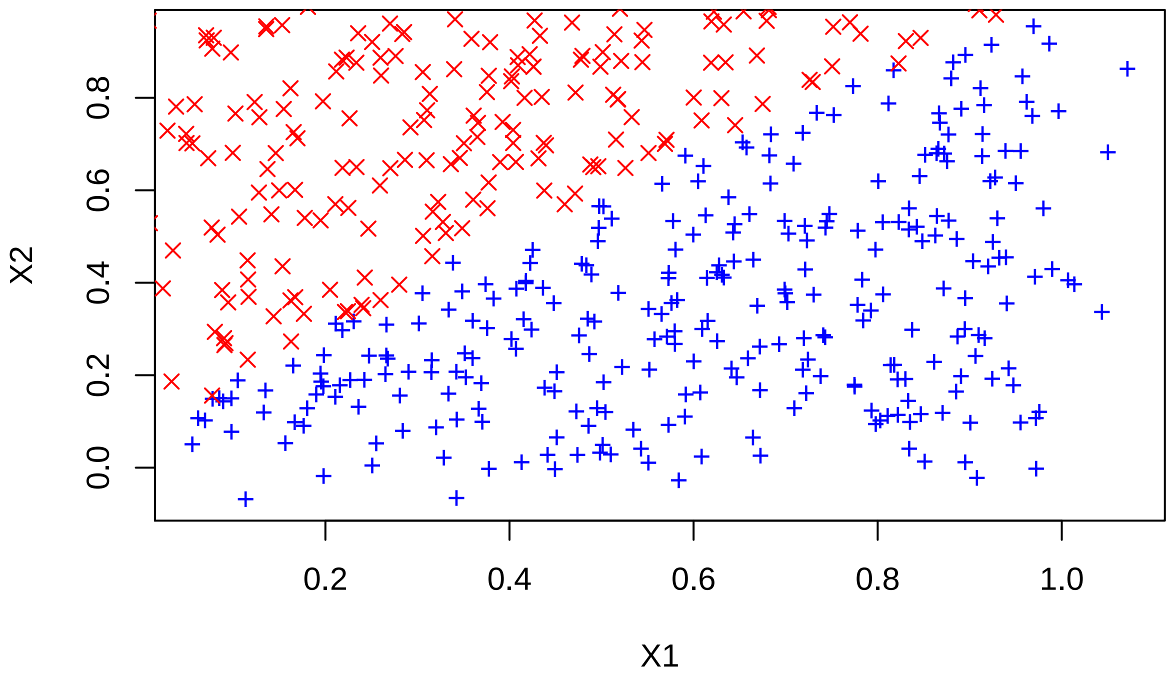
Plotting Logistic and Boundary

```
logit_p = as.numeric(predict(logit, df, type = "response") > 0.5)
blue = df[logit_p == 1, ]
red = df[logit_p == 0, ]
plot(blue$x1, blue$x2, col = "blue", xlab = "X1", ylab = "X2", pch = "+")
points(red$x1, red$x2, col = "red", pch = 4)
```



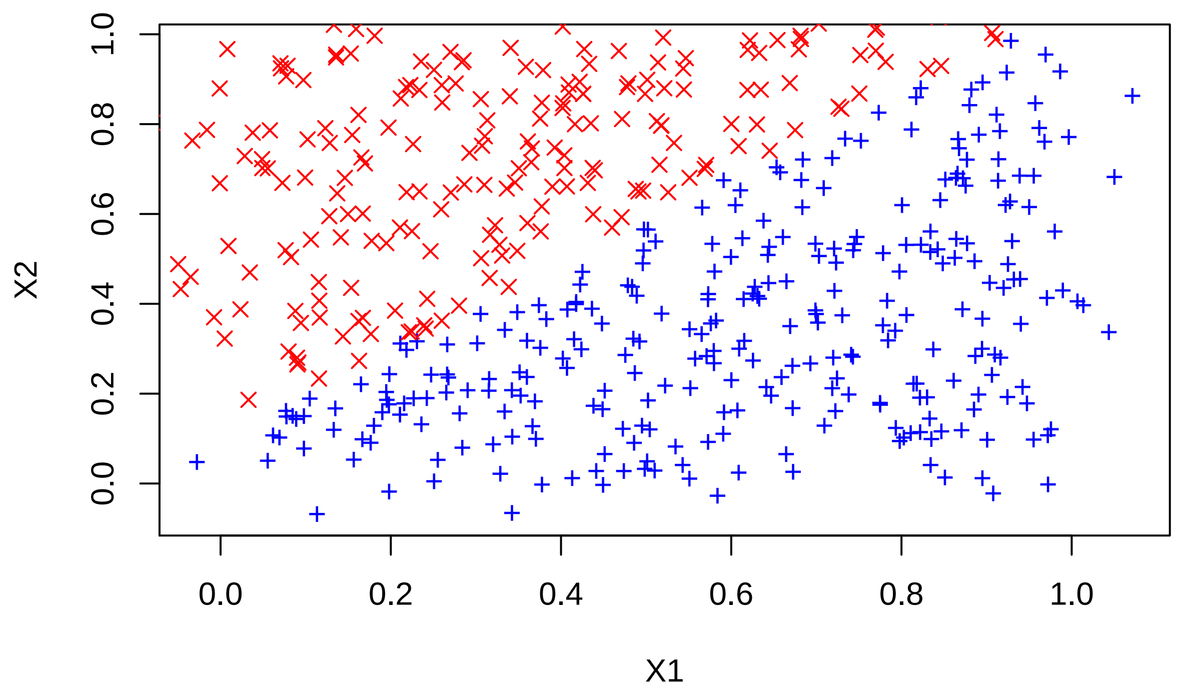
Non-Linear Logistic

```
logit_nl = glm(y ~ poly(x1, 2) + poly(x2, 2) + I(x1 * x2) , family = binomial, data = train)
logit_nl_p = as.numeric(predict(logit_nl, df, type = "response") > 0.5)
blue = df[logit_nl_p == 1, ]
red = df[logit_nl_p == 0, ]
plot(blue$x1, blue$x2, col = "blue", xlab = "X1", ylab = "X2", pch = "+")
points(red$x1, red$x2, col = "red", pch = 4)
```



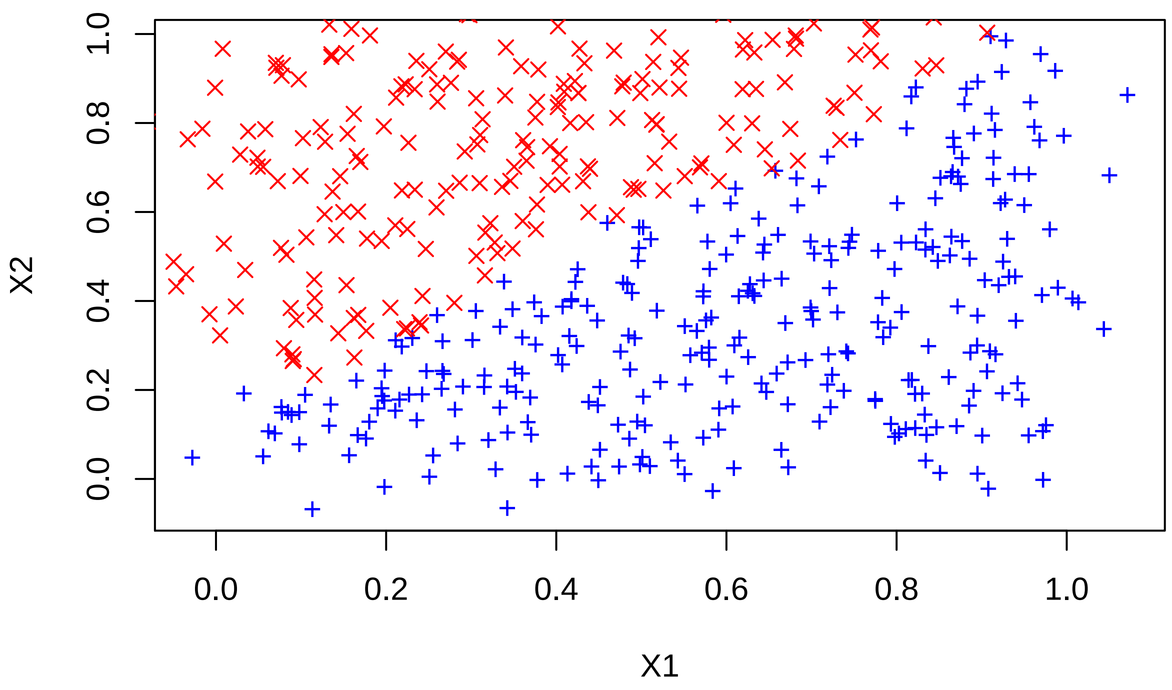
SVM Linear

```
svm1 = svm(y ~ x1 + x2, train, kernel = 'linear')
svm_p = predict(svm1, df)
blue = df[svm_p == 1, ]
red = df[svm_p == 0, ]
plot(blue$x1, blue$x2, col = "blue", xlab = "X1", ylab = "X2", pch = "+")
points(red$x1, red$x2, col = "red", pch = 4)
```



SVM Non Linear

```
svm2 = svm(y ~ x1 + x2, train, gamma = 1)
svm_p = predict(svm2, df)
blue = df[svm_p == 1, ]
red = df[svm_p == 0, ]
plot(blue$x1, blue$x2, col = "blue", xlab = "X1", ylab = "X2", pch = "+")
points(red$x1, red$x2, col = "red", pch = 4)
```



Discussion

It's clear that SVM outperforms logistic regression in our case. And by allowing non-linear boundary, we are trading bias for less variance and our approach works well in non-linear formula.

Tuning Cost

Generate Dataset

Barely-Linear Data generation learned from https://rstudio-pubs-static.s3.amazonaws.com/65566_a44a67a726284943b8f1ec986bf9642d.html

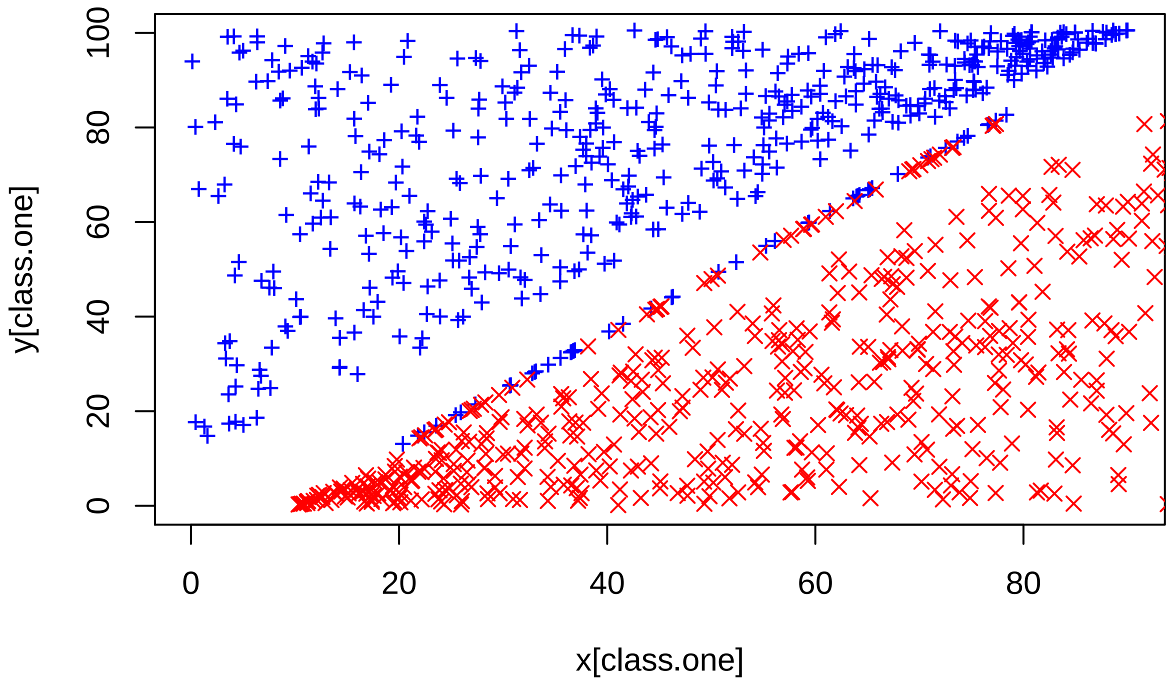
```
set.seed(3234)
x.one <- runif(500, 0, 90)
y.one <- runif(500, x.one + 10, 100)
x.one.noise <- runif(50, 20, 80)
y.one.noise <- 6/5 * (x.one.noise - 10) + 0.1

x.zero <- runif(500, 10, 100)
y.zero <- runif(500, 0, x.zero - 10)
x.zero.noise <- runif(50, 20, 80)
y.zero.noise <- 6/5 * (x.zero.noise - 10) - 0.1

class.one <- seq(1, 550)
```

```r
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)
```



Cost Tuning with CV

As can be seen, the number of misclasses changes with the cross-validation error rate in the same direction yet their rates of change differ.

```r
set.seed(4234)
z <- rep(0, 1100)
z[class.one] <- 1
z <- as.factor(z)
levels(z) = c('Class1','Class0')
data <- data.frame(x = x, y = y, z = z)

tune.out <- tune(svm, z ~ ., data = data, kernel = "linear", ranges = list(cost = c(0.25, 0.50, 1, 2, 4
results <- data.frame(cost = tune.out$performance$cost,
               misclass = tune.out$performance$error * 1100
               )
results
```

```
##      cost misclass
## 1    0.25       51
## 2    0.50       50
## 3    1.00       49
## 4    2.00       47
## 5    4.00       48
## 6    8.00       54
## 7   16.00       55
## 8   32.00       58
## 9   64.00       58
## 10 128.00       57
```

```r
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##     2
##
## - best performance: 0.04272727
##
## - Detailed performance results:
##      cost      error dispersion
## 1    0.25 0.04636364 0.01629052
## 2    0.50 0.04545455 0.01603486
## 3    1.00 0.04454545 0.01790189
## 4    2.00 0.04272727 0.01716875
## 5    4.00 0.04363636 0.01472116
## 6    8.00 0.04909091 0.01149919
## 7   16.00 0.05000000 0.01071374
## 8   32.00 0.05272727 0.01703451
## 9   64.00 0.05272727 0.01858147
## 10 128.00 0.05181818 0.02057484
```

Test

As can be seen from the table, cost at 1 and 2 that is tuned from cross-validation also are among the best-performers on the test set.

```r
set.seed(6234)
x.one <- runif(500, 0, 90)
y.one <- runif(500, x.one + 10, 100)
x.one.noise <- runif(50, 20, 80)
y.one.noise <- 6/5 * (x.one.noise - 10) + 0.1

x.zero <- runif(500, 10, 100)
y.zero <- runif(500, 0, x.zero - 10)
x.zero.noise <- runif(50, 20, 80)
y.zero.noise <- 6/5 * (x.zero.noise - 10) - 0.1

class.one <- seq(1, 550)
x <- c(x.one, x.one.noise, x.zero, x.zero.noise)
```

```r
y <- c(y.one, y.one.noise, y.zero, y.zero.noise)

z <- rep(0, 1100)
z[class.one] <- 1
z <- as.factor(z)
levels(z) = c('Class1','Class0')
test <- data.frame(x = x, y = y, z = z)

costs = results$cost
error_rate <- rep(NA, 10)
for (cost in costs){
  svm.fit <- svm(z~., data = data, kernel = 'linear', cost = cost)
  pred = predict(svm.fit, newdata = test)
  error_rate[match(cost, costs)] <- sum(pred != test$z)/1100
}

data.frame(costs,error_rate)
```

```
##     costs error_rate
## 1    0.25 0.04727273
## 2    0.50 0.04545455
## 3    1.00 0.04454545
## 4    2.00 0.04454545
## 5    4.00 0.04545455
## 6    8.00 0.04454545
## 7   16.00 0.05090909
## 8   32.00 0.05090909
## 9   64.00 0.05090909
## 10 128.00 0.05000000
```

Discussion

Our SVM models all appeared to be robust with a test error rate only slightly above training error rate, and the accuracy is satisfactory. Still, it's worthwhile to further investigate why during training stage the best tuned value (cost = 1) from cross validation does not yield the least misclassifcation.

Application

Model Fitting

```r
train = read.csv('gss_train.csv')
train = data.Normalization(train)
test = read.csv('gss_test.csv')
test = data.Normalization(test)

train$colrac = as.factor(train$colrac)
levels(train$colrac) = c('Prohibit','Allow')
test$colrac = as.factor(test$colrac)
levels(test$colrac) = c('Prohibit','Allow')

set.seed(5234)
costs = c(0.01,0.1,1,10,100)
```

```
tune_cost <- tune(svm, colrac ~ ., data = train, kernel = 'linear', ranges = list(cost = costs))
summary(tune_cost)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##     1
##
## - best performance: 0.2045665
##
## - Detailed performance results:
##    cost     error dispersion
## 1 1e-02 0.2045846 0.02081492
## 2 1e-01 0.2052422 0.03036338
## 3 1e+00 0.2045665 0.03050103
## 4 1e+01 0.2059178 0.02987057
## 5 1e+02 0.2065935 0.03103234
```

Polynomial

```
set.seed(2234)
tune.out <- tune(svm, colrac ~ ., data = train, kernel = "polynomial", ranges = list(cost = costs, degr
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost degree
##     1      3
##
## - best performance: 0.1991747
##
## - Detailed performance results:
##     cost degree     error dispersion
## 1  1e-02      2 0.4746599 0.03543964
## 2  1e-01      2 0.4233312 0.04087944
## 3  1e+00      2 0.2984355 0.02135761
## 4  1e+01      2 0.3409668 0.03986553
## 5  1e+02      2 0.3457101 0.03998311
## 6  1e-02      3 0.4746599 0.03543964
## 7  1e-01      3 0.2444268 0.03032256
## 8  1e+00      3 0.1991747 0.01700321
## 9  1e+01      3 0.2099719 0.03299259
## 10 1e+02      3 0.2180845 0.03316866
## 11 1e-02      4 0.4746599 0.03543964
## 12 1e-01      4 0.4733085 0.03633010
## 13 1e+00      4 0.2639942 0.02412371
## 14 1e+01      4 0.2754761 0.02899975
## 15 1e+02      4 0.2754807 0.02344995
```

Radial

```
set.seed(2234)
tune.out <- tune(svm, colrac ~ ., data = train, kernel = "radial", ranges = list(cost = costs, gamma = 
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##     1  0.01
##
## - best performance: 0.2011836
##
## - Detailed performance results:
##     cost gamma     error dispersion
## 1  1e-02 1e-02 0.4746599 0.03543964
## 2  1e-01 1e-02 0.2180800 0.02687312
## 3  1e+00 1e-02 0.2011836 0.03201683
## 4  1e+01 1e-02 0.2072828 0.02579054
## 5  1e+02 1e-02 0.2228007 0.03034346
## 6  1e-02 1e-01 0.4746599 0.03543964
## 7  1e-01 1e-01 0.4746599 0.03543964
## 8  1e+00 1e-01 0.3186650 0.03014254
## 9  1e+01 1e-01 0.2997642 0.03065090
## 10 1e+02 1e-01 0.2997642 0.03065090
## 11 1e-02 1e+00 0.4746599 0.03543964
## 12 1e-01 1e+00 0.4746599 0.03543964
## 13 1e+00 1e+00 0.4746599 0.03543964
## 14 1e+01 1e+00 0.4746599 0.03543964
## 15 1e+02 1e+00 0.4746599 0.03543964
## 16 1e-02 5e+00 0.4746599 0.03543964
## 17 1e-01 5e+00 0.4746599 0.03543964
## 18 1e+00 5e+00 0.4746599 0.03543964
## 19 1e+01 5e+00 0.4746599 0.03543964
## 20 1e+02 5e+00 0.4746599 0.03543964
## 21 1e-02 1e+01 0.4746599 0.03543964
## 22 1e-01 1e+01 0.4746599 0.03543964
## 23 1e+00 1e+01 0.4746599 0.03543964
## 24 1e+01 1e+01 0.4746599 0.03543964
## 25 1e+02 1e+01 0.4746599 0.03543964
## 26 1e-02 1e+02 0.4746599 0.03543964
## 27 1e-01 1e+02 0.4746599 0.03543964
## 28 1e+00 1e+02 0.4746599 0.03543964
## 29 1e+01 1e+02 0.4746599 0.03543964
## 30 1e+02 1e+02 0.4746599 0.03543964
```

Discussion

As can be seen from the model summaries, the Polynomial Kernel yields the best accuracy. However, it does not surpass its peers by far. In addition, the runtime of polynomial kernel model on my machine is like one tenth compared to the other two methods, meaning it's much more computationally efficient. This accuracy is also superior to all models I trained for the last problem set, yet given a rather large number of features, I do expect higher performance.