# STAT 542 / CS 598: Homework 6

*Fall 2019, by John Moran (jfmoran2)*

*Due: Monday, Nov 11 by 11:59 PM Pacific Time*

## Contents

Question 1 [50 Points] Linearly Separable SVM using Quadratic Programming . . . . . . . . . . . . 1
Question 2 [25 Points] Linearly Non-seperable SVM using Penalized Loss . . . . . . . . . . . . . . . 4
Question 3 [25 Points] Nonlinear and Non-seperable SVM using Penalized Loss . . . . . . . . . . . . 7

## Question 1 [50 Points] Linearly Separable SVM using Quadratic Programming

Install the `quadprog` package (there are similar ones in Python too) and utilize the function `solve.QP` to solve SVM. The `solve.QP` function is trying to perform the minimization problem:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\boldsymbol{\beta}^T \mathbf{D}\boldsymbol{\beta} - d^T\boldsymbol{\beta} \\ \text{subject to} \quad & \mathbf{A}^T\boldsymbol{\beta} \geq a \end{aligned}$$

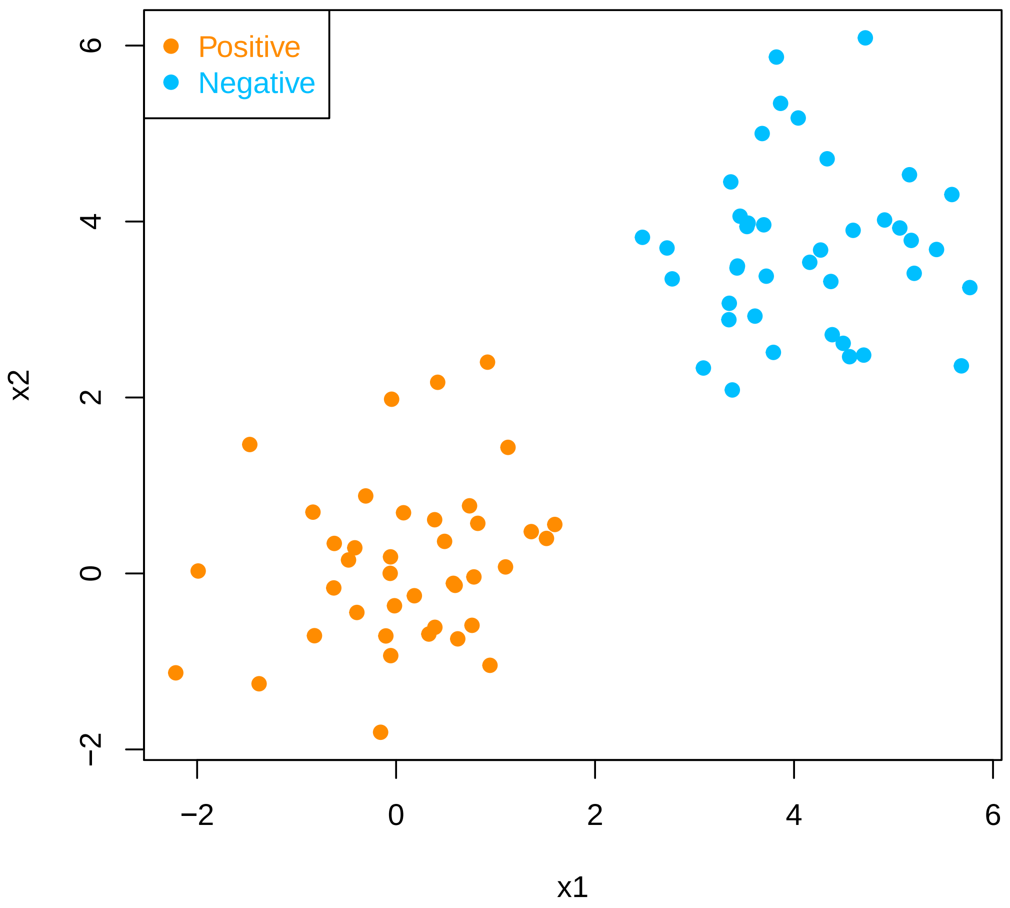
For more details, read the document file of the `quadprog` package on CRAN. Investigate the dual optimization problem of the seperable SVM formulation, and write the problem into the above form by properly defining $\mathbf{D}$, $d$, $A$ and $a$.

**Note**: The package requires $\mathbf{D}$ to be positive definite, while it may not be true in our problem. A workaround is to add a "ridge," e.g., $10^{-5}\mathbf{I}$, to the $\mathbf{D}$ matrix, making it invertible. This may affect your later results, so figure out a way to fix them.

You should generate the data using the following code (or write a similar code in Python). After solving the quadratic programming problem, perform the following:

- Convert the solution into $\beta$ and $\beta_0$, which can be used to define the classification rule
- Plot all data and the decision line
- Add the two separation margin lines to the plot
- Add the support vectors to the plot

```r
set.seed(1)
n <- 40
p <- 2

xpos <- matrix(rnorm(n*p, mean=0, sd=1), n, p)
xneg <- matrix(rnorm(n*p, mean=4, sd=1), n, p)
x <- rbind(xpos, xneg)
y <- matrix(c(rep(1, n), rep(-1, n)))

plot(x,col=ifelse(y>0,"darkorange", "deepskyblue"), pch = 19, xlab = "x1", ylab = "x2")
legend("topleft", c("Positive","Negative"),
       col=c("darkorange", "deepskyblue"), pch=c(19, 19), text.col=c("darkorange", "deepskyblue"))
```



Set up and solve the quadratic programming problem:

```r
library(quadprog)
library(Matrix)

# n above in plot code was for only half the data, i.e. the positve or negative data
n <- 2*n

Dtmp <- sapply(1:n, function(i) y[i]*t(x)[,i])
Dmat <- t(Dtmp) %*% Dtmp

# add a "ridge" to make it Dmat positive definite for solve.QP
diag(Dmat) <- diag(Dmat) + 1e-5

# all ones
dvec <- rep(1, n)

# all zeros
bvec <- rep(0, n+1)
Amat <- rbind(as.vector(y), diag(n))

qp <- solve.QP(Dmat, dvec, t(Amat), bvec=bvec, meq=1)
```

Identify the support vectors and alculate the beta values

```r
idx <- which(abs(qp$solution) > 1e-3)
print("Indices of support vectors:")
```

```
## [1] "Indices of support vectors:"
```

```r
print(idx)
```

```
## [1] 21 44 63
```

```r
sv.lambdas <- qp$solution[idx]
sv.x <- x[idx,]
sv.y <- y[idx]

betas <- rowSums(sapply(1:nrow(sv.x), function(i) sv.lambdas[i]*sv.y[i]*sv.x[i,]))

# calculuate all support vectors beta0, sum them and divide by number of SVs
# numerically safer, even though any one SV is okay
beta0 <- sum(sapply(1:nrow(sv.x), function (i) 1/(sv.y[i]) - betas %*% sv.x[i,]))
beta0 <- beta0/nrow(sv.x)

print(sprintf("beta[1]: %f beta[2]: %f beta0: %f", betas[1], betas[2], beta0))
```

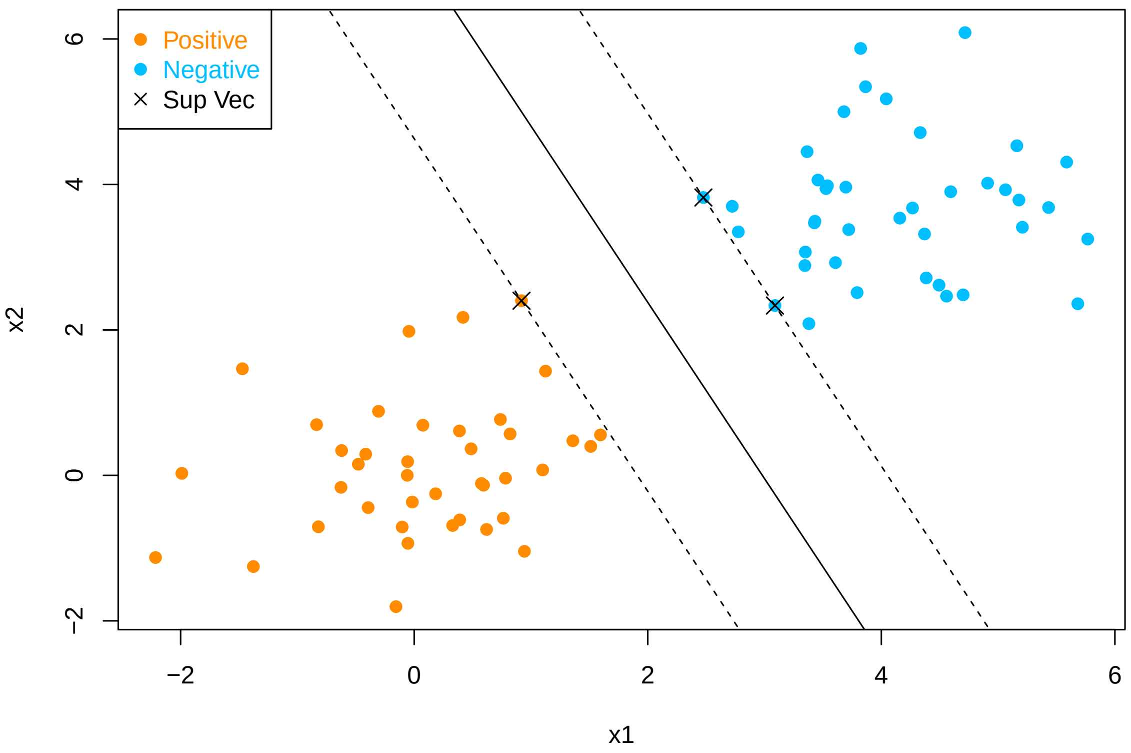
```
## [1] "beta[1]: -0.933432 beta[2]: -0.385075 beta0: 2.782650"
```

Calculate the slope, intercept, margin lines and plot them, including identifying the support vectors

```r
# ax + by + c = 0
# slope: -(a/b)
# intercept (set x=0), by + c = 0, y = -c/b
slope <- -(betas[1]/betas[2])
intercept <- -beta0/betas[2]
pos.margin.intercept <- (-beta0 + 1) / betas[2]
neg.margin.intercept <- (-beta0 - 1) / betas[2]

print(sprintf("slope: %f intercept: %f", slope, intercept))
```

```
## [1] "slope: -2.424027 intercept: 7.226257"
```

```r
plot(x,col=ifelse(y > 0,"darkorange", "deepskyblue"), pch = 19, xlab = "x1", ylab = "x2")
legend("topleft", c("Positive","Negative", "Sup Vec"),
       col=c("darkorange", "deepskyblue", "black"), pch=c(19, 19, 4),
       text.col=c("darkorange", "deepskyblue", "black"))

abline(intercept, slope)
abline(pos.margin.intercept, slope, lty = 2)
abline(neg.margin.intercept, slope, lty = 2)

par(cex=1.5)
points(sv.x, col="black", pch = 4)
```



## Question 2 [25 Points] Linearly Non-seperable SVM using Penalized Loss

We also introduced an alternative method to solve SVM. Consider a logistic loss function

$$L(y, f(x)) = \log(1 + e^{-yf(x)})$$

and solve the penalized loss for a linear SVM

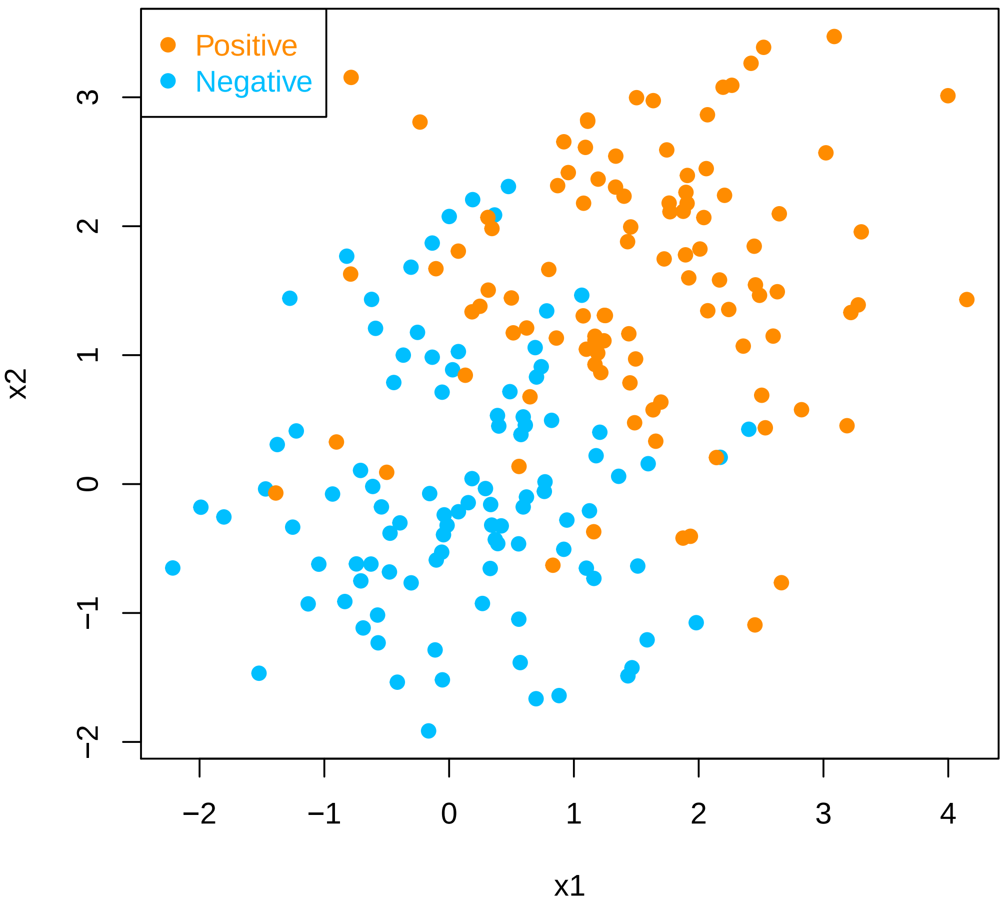
$$\underset{\beta_0,\beta}{\arg\min} \sum_{i=1}^{n} L(y_i, \beta_0 + x^T\beta) + \lambda\|\beta\|^2$$

The rest of the job is to solve this optimization problem. To do this, we will utilize a general-purpose optimization package/function. For example, in `R`, you can use the `optim` function. Read the documentation of this function (or equivalent ones in Python) and set up the objective function properly to solve for the parameters. If you need an example of how to use the `optim` function, read the corresponding part in the example file provide on our course website here (Section 10). You should generate the data using the following code (or write a similar code in Python). Perform the following:

- Write a function to define the objective function (penalized loss). The algorithm may run faster if you further define the gradient function. However, the gradient is not required for completing this homework, but it counts for 2 bonus points.
- Choose a reasonable $\lambda$ value so that your optimization can run properly. In addition, I recommend using the `BFGS` method in the optimization.
- After solving the optimization problem, plot all data and the decision line
- If needed, modify your $\lambda$ so that the model fits reasonably well (you do not have to optimize this tuning), and re-plot

```r
set.seed(1)
n = 100 # number of data points for each class
p = 2 # dimension
```

```
# Generate the positive and negative examples
xpos <- matrix(rnorm(n*p,mean=0,sd=1),n,p)
xneg <- matrix(rnorm(n*p,mean=1.5,sd=1),n,p)
x <- rbind(xpos,xneg)
y <- c(rep(-1, n), rep(1, n))

#reset cex from previous plot
par(cex=1.0)

plot(x,col=ifelse(y>0,"darkorange", "deepskyblue"), pch = 19, xlab = "x1", ylab = "x2")
legend("topleft", c("Positive","Negative"), col=c("darkorange", "deepskyblue"),
       pch=c(19, 19), text.col=c("darkorange", "deepskyblue"))
```



Write function to define penalized loss function

```
f <- function(betas, X, Y, Lambda) {
  beta0 <- betas[1]
  beta <- betas[2:3]

  sum(log(1+exp(-Y*(beta0 + X %*% beta)))) + Lambda*sum(beta^2)
}
```

Choose a lambda and call optim function and plot result, try lambda = 1

```
lambda <- 1
```

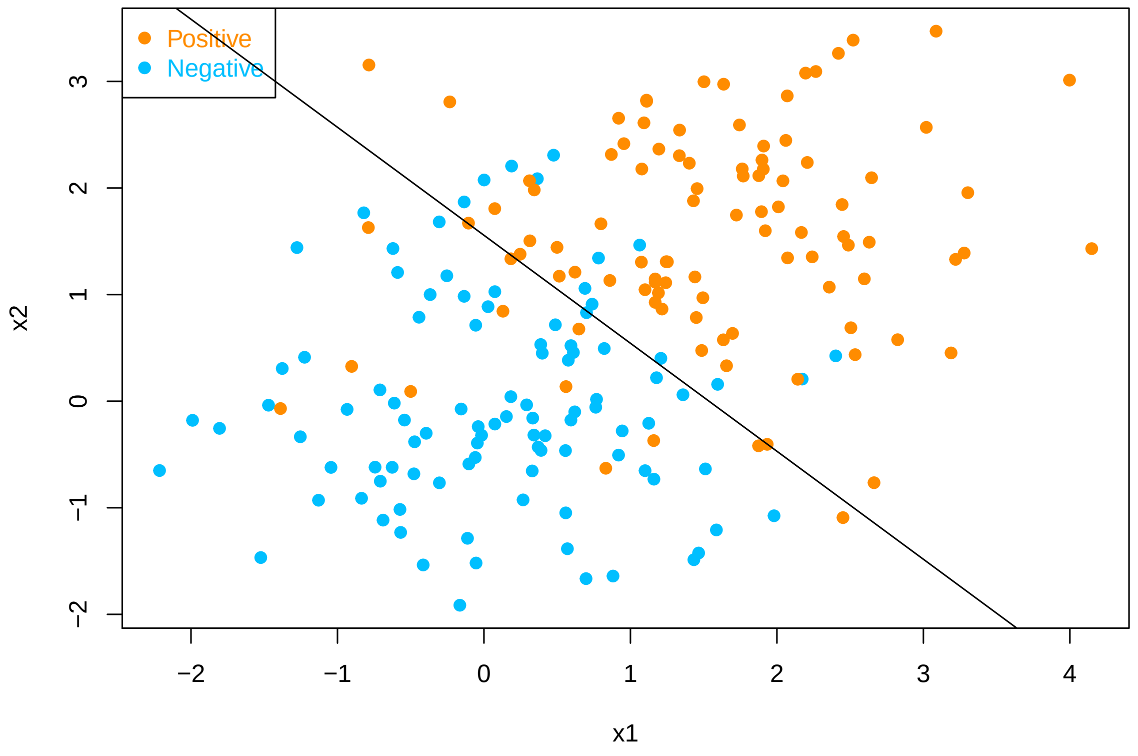
```r
solution <- optim(rep(0,3), f, X = x, Y=y, Lambda = lambda, method = "BFGS")

print(sprintf("Optimized value from optim function: %f", solution$value))
```

```
## [1] "Optimized value from optim function: 69.164314"
```

```r
calc.beta0 <- solution$par[1]
calc.beta1 <- solution$par[2]
calc.beta2 <- solution$par[3]

slope <- -calc.beta1/calc.beta2
intercept <- -calc.beta0 /calc.beta2

print(sprintf("slope: %f intercept: %f", slope, intercept))
```

```
## [1] "slope: -1.013320 intercept: 1.556746"
```

```r
plot(x,col=ifelse(y>0,"darkorange", "deepskyblue"), pch = 19, xlab = "x1", ylab = "x2")
legend("topleft", c("Positive","Negative"), col=c("darkorange", "deepskyblue"),
         pch=c(19, 19), text.col=c("darkorange", "deepskyblue"))

abline(intercept, slope)
```



Try reducing lambda from 1 to 0.1 to see if solution value is lower.

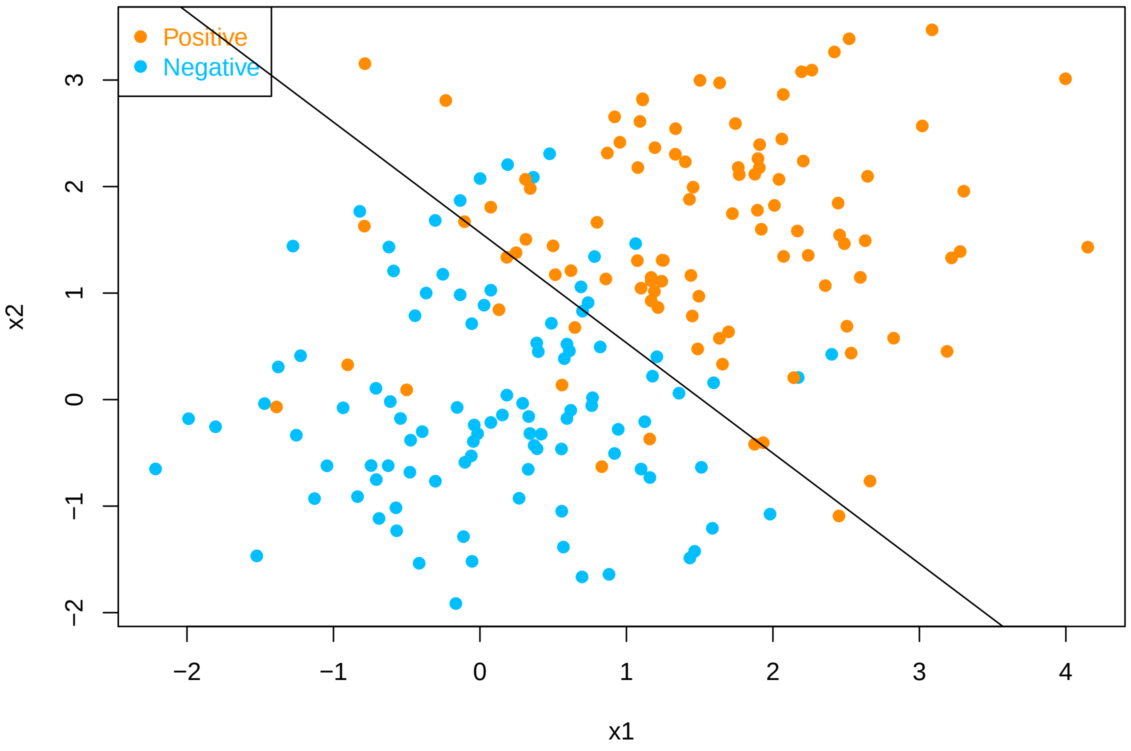
```r
lambda <- 0.1

solution <- optim(rep(0,3), f, X = x, Y=y, Lambda = lambda, method = "BFGS")

print(sprintf("Optimized value from optim function: %f", solution$value))
```

```
## [1] "Optimized value from optim function: 65.546412"
```

```r
calc.beta0 <- solution$par[1]
calc.beta1 <- solution$par[2]
calc.beta2 <- solution$par[3]

slope <- -calc.beta1/calc.beta2
intercept <- -calc.beta0 /calc.beta2

print(sprintf("slope: %f intercept: %f", slope, intercept))
```

```
## [1] "slope: -1.036099 intercept: 1.568848"
```

```r
plot(x,col=ifelse(y>0,"darkorange", "deepskyblue"), pch = 19, xlab = "x1", ylab = "x2")
legend("topleft", c("Positive","Negative"), col=c("darkorange", "deepskyblue"),
         pch=c(19, 19), text.col=c("darkorange", "deepskyblue"))

abline(intercept, slope)
```



The solution value return from the optim function is lower for the lambda of 0.1 and the fitted line seems to divide the data little better, so lambda of 0.1 looks like a better parameter choice than lambda of 1.

## Question 3 [25 Points] Nonlinear and Non-seperable SVM using Penalized Loss

We can further use the kernel trick to solve for a nonlinear decision rule. The optimization becomes

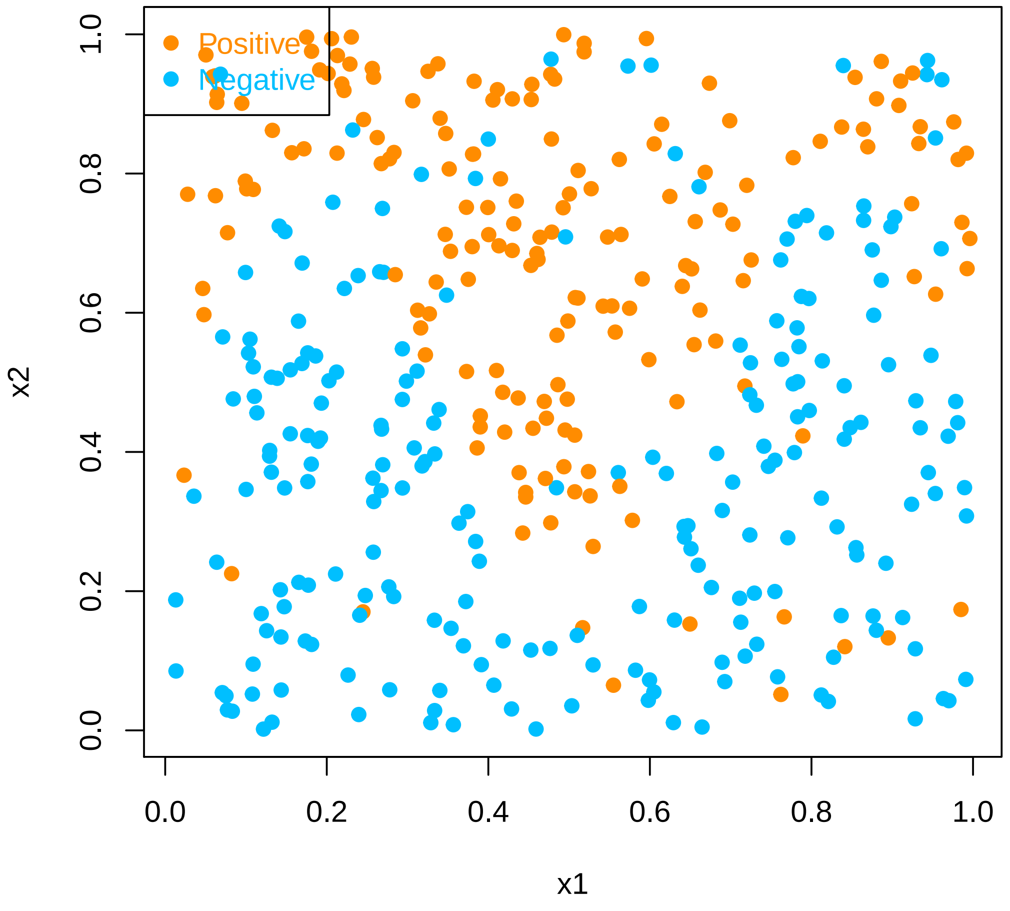
$$\sum_{i=1}^{n} L(y_i, K_i^T \beta) + \lambda \beta^T K \beta$$

where $K_i$ is the $i$th column of the $n \times n$ kernel matrix $K$. For this problem, we consider the Gaussian kernel (you do not need an intercept). Again, we can use the

You should generate the data using the following code (or write a similar code in Python). Perform the following:

- Pre-calculate the $n \times n$ kernel matrix $K$ of the observed data
- Write a function to define the objective function (this should not involve the original $x$, but uses $K$). Again, the gradient is not required for completing this homework. However, it counts for 3 bonus points.
- Choose a reasonable $\lambda$ value so that your optimization can run properly
- After solving the optimization problem, plot **fitted** labels (in-sample prediction) for all subjects
- If needed, modify your $\lambda$ so that the model fits reasonably well (you do not have to optimize this tuning), and re-plot
- Summarize your in-sample classification error

```r
set.seed(1)
n = 400
p = 2 # dimension

# Generate the positive and negative examples
x <- matrix(runif(n*p), n, p)
side <- (x[, 2] > 0.5 + 0.3*sin(3*pi*x[, 1]))
y <- sample(c(1, -1), n, TRUE, c(0.9, 0.1))*(side == 1) + sample(c(1, -1), n,
           TRUE, c(0.1, 0.9))*(side == 0)

plot(x,col=ifelse(y>0,"darkorange", "deepskyblue"), pch = 19, xlab = "x1", ylab = "x2")
legend("topleft", c("Positive","Negative"),
     col=c("darkorange", "deepskyblue"), pch=c(19, 19), text.col=c("darkorange", "deepskyblue"))
```



Calculate K, the Gaussian kernel and write the objective function using only K and no x.

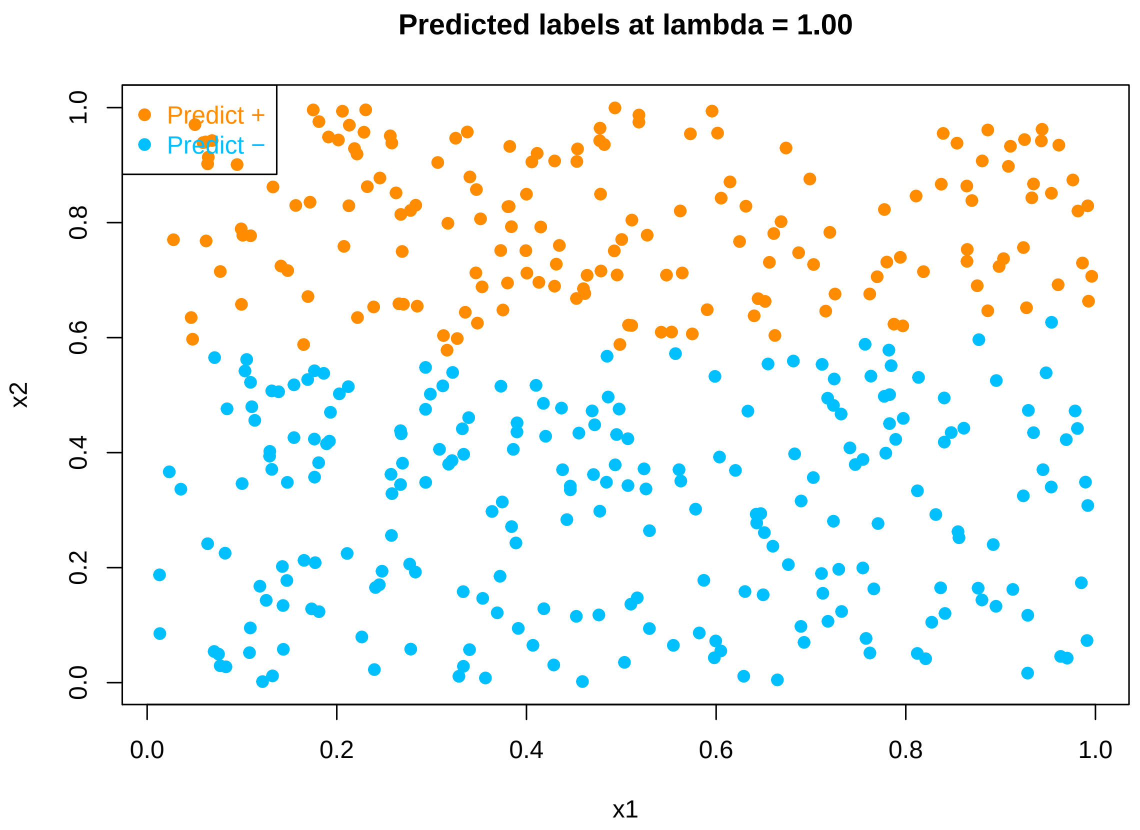
```
sigma <- 1

gauss.kernel <- function(aRow, entireX, sigma) {
  m1 <- sweep(entireX,2,aRow,"-")
  m2 <- (sweep(m1,2,sigma,"/"))^2
  return( exp(-0.5*rowSums(m2)))
}
K <- apply(x, 1, gauss.kernel, entireX=x, sigma=sigma)

f <- function(alpha, K, Y, Lambda) {
  Ka <- crossprod(K,alpha)
  sum(log(1+exp(-Y*Ka))) + Lambda*t(alpha) %*% Ka
}
```

**Start note on code efficiency**
Initially the gauss_kernel function used for loops. Changing the code to use the *sweep* function, sped up the calculation significantly, though it is neglible for n=400, but testing with much larger n showed many orders of magnitude improvement in speed.

Also, in the objective function, I was calculating the cross product of K,alpha twice in the *sum* line of code. By storing it in the *Ka* variable and moving it to the line above, it only calculates it once. This cut the run time of the program down from 27 seconds to 16 seconds.
**End note on code efficiency**

Set lambda to 1, and call the optim function, calculate the prediction values and plot the predicted labels.

```
lambda <- 1
alpha <- as.matrix(rep(0,n))
solution <- optim(alpha, f, K = K, Y=y, Lambda = lambda, method = "BFGS")

alphas <- solution$par

# the prediction is just the sign of K %*% alphas
pred <- K %*% alphas
ind <- which(pred > 0)
pred[ind] <- 1
pred[-ind] <- -1

top.title <- sprintf("Predicted labels at lambda = %0.2f", lambda)
plot(x,col=ifelse(pred > 0,"darkorange", "deepskyblue"), pch = 19,
     xlab = "x1", ylab = "x2", main=top.title)
legend("topleft", c("Predict +","Predict -"),
       col=c("darkorange", "deepskyblue"), pch=c(19, 19), text.col=c("darkorange", "deepskyblue"))
```



The prediction accuracy using sigma = 1.0 and lambda = 1.0 is:

```r
accuracy <- sum(y == pred)/n
print(sprintf("Accuracy of predictions: %f", accuracy))
```

```
## [1] "Accuracy of predictions: 0.772500"
```

and the solution value returned from optim is:

```r
print(sprintf("Solution value: %f", solution$value))
```

```
## [1] "Solution value: 226.360930"
```

Try again with sigma = 0.5 and lambda = 0.01

```r
sigma <- 0.5
lambda <- 0.01

K <- apply(x, 1, gauss.kernel, entireX=x, sigma=sigma)

alpha <- as.matrix(rep(0,n))
solution <- optim(alpha, f, K = K, Y=y, Lambda = lambda, method = "BFGS")

alphas <- solution$par

# the prediction is just the sign of K %*% alphas
pred <- K %*% alphas
ind <- which(pred > 0)
pred[ind] <- 1
pred[-ind] <- -1
```

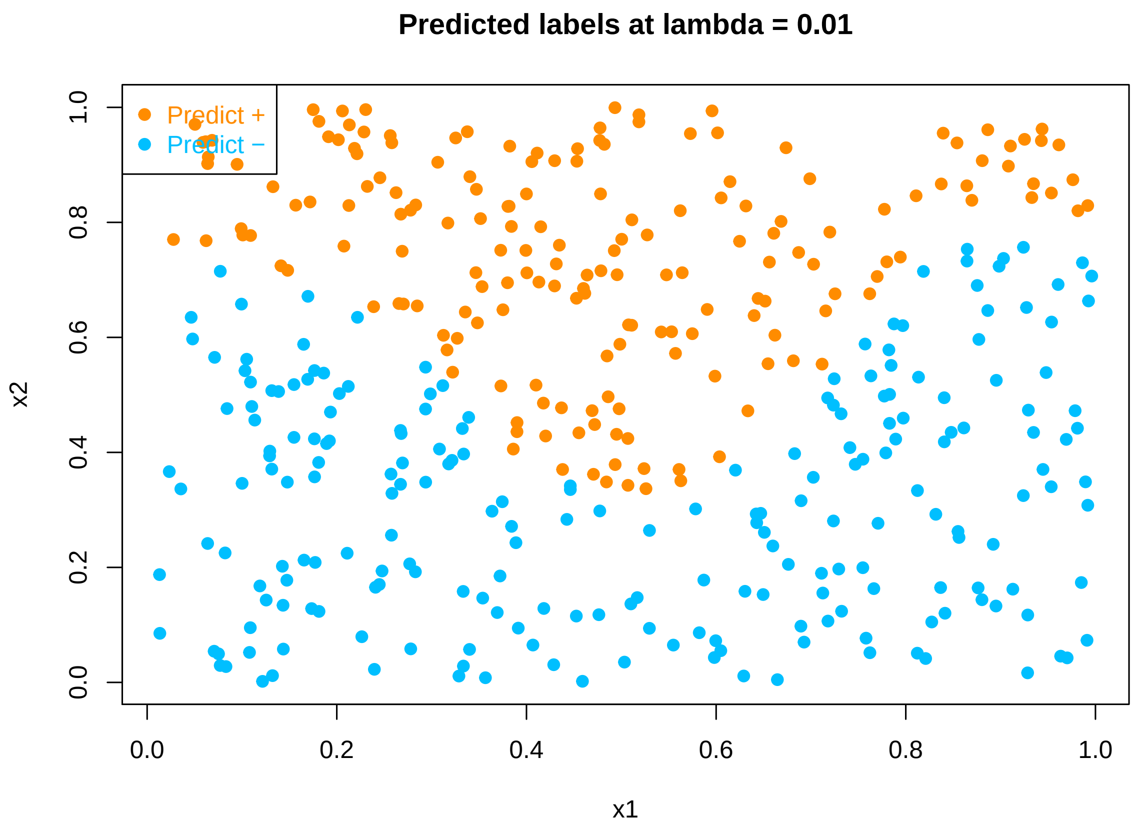
```r
accuracy <- sum(y == pred)/n
print(sprintf("Accuracy of predictions: %f", accuracy))
```

```
## [1] "Accuracy of predictions: 0.850000"
```

```r
print(sprintf("Solution value: %f", solution$value))
```

```
## [1] "Solution value: 172.024598"
```

```r
top.title <- sprintf("Predicted labels at lambda = %0.2f", lambda)
plot(x,col=ifelse(pred > 0,"darkorange", "deepskyblue"), pch = 19,
     xlab = "x1", ylab = "x2", main=top.title)
legend("topleft", c("Predict +","Predict -"),
       col=c("darkorange", "deepskyblue"), pch=c(19, 19),
       text.col=c("darkorange", "deepskyblue"))
```



Experimenting with different values for lambda and sigma, I found that as sigma was lowered, the prediction accuracy went up, but you could tell by the graph that the algorithm was overfitting as sigma went lower. A sigma value of 0.01 produced almost 100% accuracy, but it was grossly overfitting.

After some experimentation with sigma and lambda parameter values, I determined a reasonable selection of parameter values was **lambda = 0.01** and **sigma = 0.5**, shown in the plot above. These parameters values resulted in a accuracy of **85%** which seems to be a reasonable accuracy that can be obtained without overfitting the data. Better values of lambda and sigma could likely be found with further tuning.