# STAT 542: Homework 10

Arka Mitra (netid - arkam2)

Due: Tuesday, April 13, 11:59 PM CT

## Contents

## About HW10

This homework involves two coding questions. One is discriminant analysis, and another one is logistic regression.

## Question 1 [50 Points] Discriminant Analysis

For this question, you need to write your own code. We will use the handwritten digit recognition data again from the `ElemStatLearn` package. Pool the `zip.train` and `zip.test` data together. Randomly select 1000 observations as the training data and the rest as testing data. No cross-validation is needed. Make sure to save seed.

### a) [25 points] Linear discriminate analysis

Write your own linear discriminate analysis (LDA) code following our lecture note. Use the training data to estimate all parameters and apply them to the testing data to evaluate the performance. You may face computational problems when calculating the log-determinate. Figure out ways to solve that. Report the model fitting results (such as a confusion table and misclassification rates). Which digit seems to get misclassified the most?

```r
library(ElemStatLearn)
set.seed(70)

zip.train = zip.test
zip.test = zip.train
digits = rbind(zip.train, zip.test)
ind = runif(1000, 0, dim(digits)[1])
train = digits[ind,]
test = digits[-ind,]

# Arrays to store model estimates
```

```
pi_k <- rep(0,10)
mu_k <- matrix(0,10,256)
sig1 <- 0
for (k in c(0:9)){
  id_k1 = which(train[,1]==k) # indices for each class
  nk1 = sum(as.numeric(train[,1]==k)) # number of obs in each class
  trn_k1 = train[id_k1,] # training data for each class
  pi_k[k+1] = nk1/nrow(train)
  mu_k[k+1,] = apply(trn_k1[,-1],2,mean)
  s1 = 0
  for (i in 1:nk1){ #variances in each class
    s1 = s1 + (trn_k1[i,-1] - mu_k[k+1,]) %*% t(trn_k1[i,-1] - mu_k[k+1,])
  }
  sig1 = sig1 + s1
}
sig1 = sig1 / (nrow(train) - 10) # the estimated sigma

temp_pred2a = matrix(0,nrow(test),10)
for (j in 1:10){
  temp_mu1 = as.matrix(mu_k[j,])
  temp1 = matrix(t(temp_mu1) %*% solve(sig1) %*% temp_mu1 / (-2) + log(pi_k[j]),nrow(test),1)
  temp_pred2a[,j] = test[,-1] %*% solve(sig1) %*% temp_mu1 + temp1
}
rm(temp1, temp_mu1)

#check levels of misclassification
pred2a = apply(temp_pred2a,1,which.max)-1
print('Mean Misclassification Rate across all classes:')
```

```
## [1] "Mean Misclassification Rate across all classes:"
```

```
mean(pred2a!=test[,1])
```

```
## [1] 0.1397953
```

```
print('Confusion Matrix:')
```

```
## [1] "Confusion Matrix:"
```

```
table(pred_value=pred2a,true_value=test[,1])
```

```
##            true_value
## pred_value   0   1   2   3   4   5   6   7   8   9
##          0 530   0  12   4   0  10   8   0  14   0
##          1   0 411   0   0   8   0   2   2   2   2
##          2   0   2 231  14   0   0   4   0   8   0
##          3   8   0  16 209   0  22   2   1  15   2
##          4   4   6  12   2 253  10   6   4  13  16
##          5   2   0   6  16   0 190  10   0  16   0
##          6   6   4   8   0   5   4 239   0   2   0
##          7   0   0   6   4   5   2   0 186   4   3
##          8   2   0  12   6  12   8   2   2 190   2
##          9   2   5   2   4  22   8   0  14   0 250
```

```
mis2a = matrix(0,1,10)
colnames(mis2a) = seq(0,9)
for (m in c(0:9)){
  mis2a[1,m+1] = mean(pred2a[which(test[,1]==m)]!=test[which(test[,1]==m),1])
}
print('Mean Misclassification Rate for individual classes:')
```

```
## [1] "Mean Misclassification Rate for individual classes:"
```

```
mis2a
```

```
##               0          1        2         3         4         5         6         7
## [1,] 0.0433213 0.03971963 0.242623 0.1930502 0.1704918 0.2519685 0.1245421 0.1100478
##              8          9
## [1,] 0.280303 0.09090909
```

The digit 8 seems to be misclassified the most.

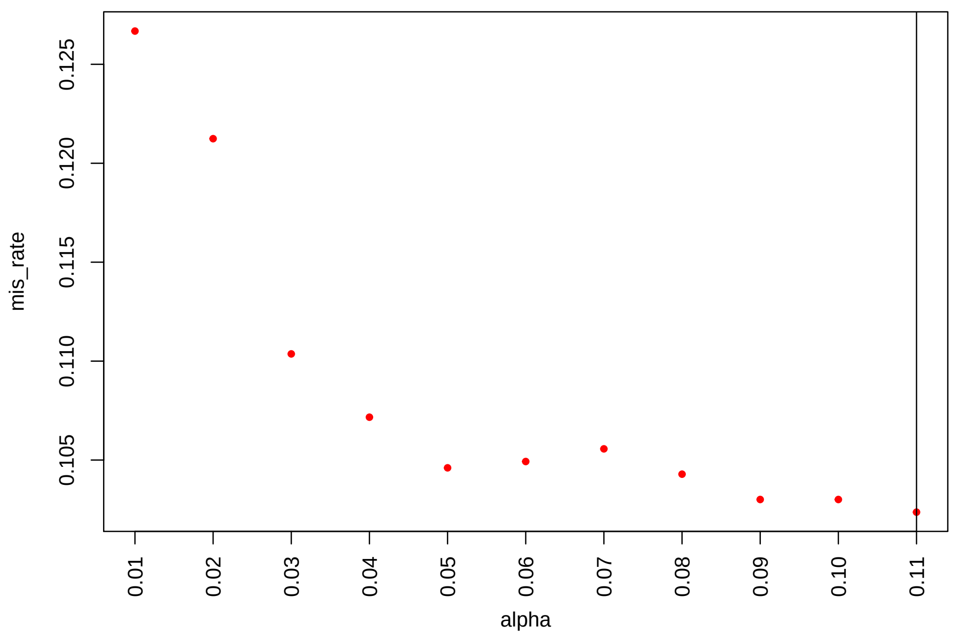
### b) [25 points] Regularized quadratic discriminate analysis

QDA does not work directly in this example because we do not have enough samples to provide invertible class-specific sample covariance matrix. An alternative idea to fix this issue is to consider a regularized QDA method, which uses

$$\widehat{\Sigma}_k(\alpha) = \alpha\widehat{\Sigma}_k + (1-\alpha)\widehat{\Sigma}$$

for some $\alpha \in (0,1)$. Here $\widehat{\Sigma}$ is the estimator from the LDA method. Implement this method and select the best tuning parameter (on a grid) based on the testing error. You should again report the model fitting results similar to the previous part. What is your best tuning parameter, and what does that imply in terms of the underlying data and the performance of the model?

```
sig2 <- list()
for (k in c(0:9)){
  id_k2 = which(train[,1]==k) # indices for each class
  nk2 = sum(as.numeric(train[,1]==k)) # number of obs in each class
  trn_k2 = train[id_k2,] # training data for each class
  s2 = 0
  for (i in 1:nk2){  # variances for each class
    s2 = s2 + as.matrix(trn_k2[i,-1] - mu_k[k+1,]) %*% t(as.matrix(trn_k2[i,-1] - mu_k[k+1,]))
  }
  sig2[[k+1]] = s2 / (nk2 - 1)
  # the sigma for each class
}

# We use alphas between 0.01 and 0.11 because for higher alpha
# misclassification rates increase monotonically
alpha = seq(0.01,0.11,by=0.01)
temp_predb = matrix(0,nrow(test),10)
pred2 = matrix(NA,nrow(test),length(alpha))
mis_rate = rep(0,length(alpha))
for (l in 1:length(alpha)){
  a = alpha[l]
  for (j in 1:10){
    temp_sig2 = a*sig2[[j]] + (1-a)*sig1

    #tempa & tempb are components in discriminant function

    tempa = matrix(determinant(temp_sig2)$modulus[1]/(-2) + log(pi_k[j]),nrow(test),1)
    tempx = test[,-1]-matrix(mu_k[j,],nrow(test),256,byrow=T)
    tempb = as.matrix(diag(tempx%*%solve(temp_sig2)%*%t(tempx)),nrow(test),1)

    temp_predb[,j] = tempb /(-2) + tempa
  }
  pred2[,l] = apply(temp_predb,1,which.max)-1
  mis_rate[l] = mean(pred2[,l]!=test[,1])
}

#plot for testing error vs tuning parameter
plot(alpha,mis_rate,col="red",pch=20,xaxt="n")
axis(1,at=seq(0.01,0.11,by=0.01),las=2)
abline(v=alpha[which.min(mis_rate)])
```



```
pred2_m = pred2[,which.min(mis_rate)]
print('Mean Misclassification Rate across all classes:')
```

```
## [1] "Mean Misclassification Rate across all classes:"
```

```
mean(pred2_m!=test[,1])
```

```
## [1] 0.1023672
```

```r
print('Confusion Matrix:')
```

```
## [1] "Confusion Matrix:"
```

```r
table(pred_value=pred2_m,true_value=test[,1])
```

```
##           true_value
## pred_value   0   1   2   3   4   5   6   7   8   9
##          0 538   0   4   8   0  18  10   0  22   2
##          1   0 422   0   0   4   0   0   2   0   0
##          2   4   2 269  12   8   0  10   0   2   0
##          3   4   0  10 213   0  18   2   2  12   2
##          4   4   0   6   4 265   4   4   2   0   6
##          5   0   0   2  12   0 200   6   0  12   2
##          6   2   4   2   0   0   0 239   0   2   0
##          7   0   0   4   4   2   4   0 193   2   4
##          8   0   0   6   4   8   4   2   2 210   2
##          9   2   0   2   2  18   6   0   8   2 257
```

```r
mis2b = matrix(0,1,10)
colnames(mis2b) = seq(0,9)
for (m in c(0:9)){
  mis2b[1,m+1] = mean(pred2_m[which(test[,1]==m)]!=test[which(test[,1]==m),1])
}
print('Mean Misclassification Rate for individual classes:')
```

```
## [1] "Mean Misclassification Rate for individual classes:"
```

```r
mis2b
```

```
##               0          1         2         3         4         5         6          7
## [1,] 0.02888087 0.01401869 0.1180328 0.1776062 0.1311475 0.2125984 0.1245421 0.07655502
##              8          9
## [1,] 0.2045455 0.06545455
```

The best tuning parameter is 0.11. Misclassification rates between 0.05 and 0.11 is nearly constantly low. Beyond 0.11, misclassification increases monotonically.

With a small tuning from basic LDA, QDA already gives us an improved classification.

## Question 2 [50 Points] Logistic Regression

We consider a logistic regression problem using the South Africa heart data as a demonstration. The goal is to estimate the probability of `chd`, the indicator of coronary heart disease. The following code is used to prepare the data and fit the logistic regression. Please note that the factor variable `famhist` has been converted to a dummy.

```r
    library(ElemStatLearn)
    data(SAheart)

    heart = SAheart
    heart$famhist = as.numeric(heart$famhist)-1
    n = nrow(heart)
    p = ncol(heart)

    heart.full = glm(chd~., data=heart, family=binomial)

    # fitted value
    yhat = (heart.full$fitted.values>0.5)
    table(yhat, SAheart$chd)
```

```
##
## yhat      0   1
##   FALSE 256  77
##   TRUE   46  83
```

```r
    # data we use
    X <- as.matrix(cbind("intercept" = 1, heart[,-10]))
    Y <- as.matrix(heart$chd)
```

We can obtain the logistic regression fitting by write our own code. For example, in our R-intro file, section 10, there is an example of solving the $\beta$ parameters in a linear regression. We are going to adapt that strategy by:

- Defining the log-likelihood function of a logistic regression
- Further incorporating the gradient function to improve the computational efficiency

First, write a function `logisticLL(b, X, Y)` that calculates the log-likelihood function of $n$ observations given the design matrix `X`, a vector of binary outcomes `Y` and any given vector of parameter values `b`. Please note that the first element in `b` is the intercept since the first column if `X` should be 1, as provided in our code above. Then, perform the optimization of the logistic regression using the `optim` function. There are a few things you should be careful about:

- Are you maximizing or minimizing the objective function? Is that specified correctly in the `optim` function
- Use `method = "BFGS"` for the optimization

```r
# the negative log-likelihood function of logistic regression
logisticLL <- function(b, x, y)
    {
        bm = as.matrix(b)
        xb =  x %*% bm
        # this returns the negative loglikelihood
        return(sum(y*xb) - sum(log(1 + exp(xb))))
    }

# prepare the data matrix, I am adding a column of 1 for intercept
```

```r
x = as.matrix(cbind("intercept" = 1, heart[, 1:9]))
y = as.matrix(heart[,10])

# check my function
b = rep(0, ncol(x))
logisticLL(b, x, y) # scaler
```

```
## [1] -320.234
```

```r
# check the optimal value and the likelihood
logisticLL(heart.full$coefficients, x, y)
```

```
## [1] -236.07
```

```r
# Use a general solver to get the optimal value
opt1 = optim(b, fn = logisticLL, method = "BFGS", x = x, y = y, control = list("fnscale" = -1))

opt1$par
```

```
##  [1] -6.133036522  0.006357461  0.079318574  0.173889757  0.018589725  0.925594371
##  [7]  0.039552578 -0.062806483  0.000140873  0.045278400
```

```r
opt1$value
```

```
## [1] -236.0704
```

By default optim performs minimization, but it will maximize if control$fnscale is negative, which is the case here. So, we are maximizing the objective function.

In the second step, you should write a gradient function `logisticG(b, X, Y)` that calculates the gradient vector. Then call the `optim` again to incorporate this gradient. For both steps, compare your results to the output from the `glm()` function in terms of the estimated parameters and the training data accuracy.

```r
    # Gradient
    logisticG <- function(b, x, y)
    {
        bm = as.matrix(b)
        expxb =  exp(x %*% bm)
        return(t(X) %*% (y - expxb/(1+expxb)))
    }

opt2 =    optim(b, fn = logisticLL, gr = logisticG,
          method = "BFGS", x = x, y = y, control = list("fnscale" = -1))
opt2$par
```

```
##  [1] -6.150733305  0.006504017  0.079376464  0.173923988  0.018586578  0.925372019
##  [7]  0.039595096 -0.062909867  0.000121675  0.045225500
```

```
opt2$value
```

```
## [1] -236.07
```

```
    heart.full = glm(chd~., data=heart, family=binomial)
    round(summary(heart.full)$coef, dig=3)
```

```
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   -6.151      1.308  -4.701    0.000
## sbp            0.007      0.006   1.135    0.256
## tobacco        0.079      0.027   2.984    0.003
## ldl            0.174      0.060   2.915    0.004
## adiposity      0.019      0.029   0.635    0.526
## famhist        0.925      0.228   4.061    0.000
## typea          0.040      0.012   3.214    0.001
## obesity       -0.063      0.044  -1.422    0.155
## alcohol        0.000      0.004   0.027    0.978
## age            0.045      0.012   3.728    0.000
```

The solution from our logistic regression functions (both with and without gradient) closely matches the solution from glm(), with the gradient solution being exactly equal.