STAT 542: Homework 5

Spring 2020, by Ruoqing Zhu (rqzhu)

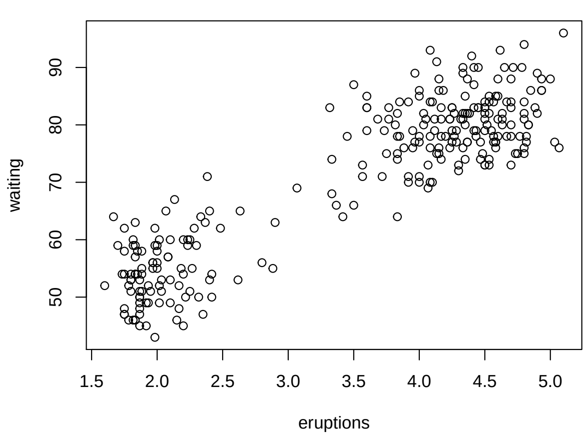
Due: Monday, Apr 13 by 11:59 PM

Contents

Question 1 [40 Points] Two-dimensional Gaussian Mixture Model

If you do not use latex to type your answer, you will lose 2 points. We consider another example of the EM algorithm, which fits a Gaussian mixture model to the Old Faithful eruption data. The data is used in HW4. For a demonstration of this problem, see the figure provided on Wikipedia. As a result, we will use the formula to implement the EM algorithm and obtain the distribution parameters of the two underlying Gaussian distributions. Here is a visualization of the data:

```r
# load the data
load('faithful.Rda')
plot(faithful)
```



We use both variables `eruptions` and `waiting`. The plot above shows that there are two eruption patterns (clusters). Hence, we use a hidden Bernoulli variable $Z_i \sim \text{Bern}(\pi)$, that indicates the pattern when we wait for the next eruption. The corresponding distribution of `eruptions` and `waiting` is a two-dimensional

Gaussian — either $N(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ or $N(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$ — depending on the outcome of Z_i. Here, of course, we do not know the parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1, \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2, \pi\}$, and we want to use the observed data to estimate them.

- [5 points] Based on the above assumption of the description, write down the full log-likelihood $\ell(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta})$. Then, following the strategy of the EM algorithm, in the E-step, we need the conditional expectation

$$g(\boldsymbol{\theta}|\boldsymbol{\theta}^{(k)}) = E_{\mathbf{Z}|\mathbf{x},\boldsymbol{\theta}^{(k)}}[\ell(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta})].$$

The answer is already provided on Wikipedia page. Write down the conditional distribution of $\mathbf{Z}$ given $\mathbf{x}$ and $\boldsymbol{\theta}^{(k)}$ with notations used in our lectures.

$$l(X,Z|\theta) = \sum_{i=1}^{n}[Z_i log(\phi_{\mu_1}(x_i)) + (1-Z_i)log(\phi_{\mu_2}(x_i))] + \sum_{i=1}^{n}[(1-Z_i)(1-\pi) + Z_i\pi]$$

$$P(Z=j|x_i,\theta^{(k)}) = \frac{P(Z=j,x_i|\theta^{(k)})}{P(x_i|\theta^{(k)})} = \frac{P(x_i|Z=j,\theta^{(k)})P(Z=j|\theta^{(k)})}{P(x_i|\theta^{(k)})}$$

$$P(Z=0|x_i,\theta^{(k)}) = \frac{(1-\pi^{(k)})f(x_i,\mu_2^{(k)},\Sigma_2^k)}{\pi^{(k)}f(x_i,\mu_1^{(k)},\Sigma_1^k + (1-\pi^{(k)})f(x_i,\mu_2^{(k)},\Sigma_2^k}$$

$$P(Z=1|x_i,\theta^{(k)}) = \frac{\pi^{(k)}f(x_i,\mu_1^{(k)},\Sigma_1^k)}{\pi^{(k)}f(x_i,\mu_1^{(k)},\Sigma_1^k + (1-\pi^{(k)})f(x_i,\mu_2^{(k)},\Sigma_2^k}$$

Let's assume $P(Z=1|x_i,\theta^{(k)}) = d_i$:

$$g(\theta|\theta^{(k)}) = E_{Z|X}[l(X,Z|\theta)]$$

$$= \sum_{i=1}^{n} d_i^{(t)}[log(\pi^{(k)}) - \frac{1}{2}log(|\Sigma_1|) - \frac{1}{2}(x_i-\mu_1)^T\Sigma_1^{-1}(x_i-\mu_1) - \frac{p}{2}log(2\pi)]$$

$$+(1-d_i^{(t)})[log(1-\pi^{(k)}) - \frac{1}{2}log(|\Sigma_2|) - \frac{1}{2}(x_i-\mu_2)^T\Sigma_2^{-1}(x_i-\mu_2) - \frac{p}{2}log(2\pi)]$$

- [10 points] Once we have the $g(\boldsymbol{\theta}|\boldsymbol{\theta}^{(k)})$, the M-step is to re-calculate the maximum likelihood estimators of $\boldsymbol{\mu}_1$, $\boldsymbol{\Sigma}_1$, $\boldsymbol{\mu}_2$, $\boldsymbol{\Sigma}_2$ and π. Again the answer was already provided. However, you need to provide a derivation of these estimators. **Hint**: by taking the derivative of the objective function, the proof involves three tricks:
  - $\text{Trace}(\beta^T\Sigma^{-1}\beta) = \text{Trace}(\Sigma^{-1}\beta\beta^T)$
  - $\frac{\partial}{\partial A}\log|A| = A^{-1}$
  - $\frac{\partial}{\partial A}\text{Trace}(BA) = B^T$

$\pi^{(k+1)}$:

$$\pi^{(k+1)} = \underset{\pi^{(k)}}{\text{argmax}}\ [\sum_{i=1}^{n} d_i^{(k)}]log(\pi^{(k)}) + [\sum_{i=1}^{n}(1-d_i^{(k)})]log(1-\pi^{(k)})$$

$$\frac{\partial g(\theta|\theta^{(k)})}{\partial\pi^{(k)}} = \frac{\sum_{i=1}^{n} d_i^{(k)}}{\pi^{(k)}} - \frac{\sum_{i=1}^{n}(1-d_i^{(k)})}{1-\pi^{(k)}} = 0$$

$$\pi^{(k+1)} = \frac{\sum_{i=1}^{n} d_i^{(k)}}{n}$$

$\mu_1^{(k+1)}$:

$$\mu_1^{(k+1)} = \underset{\mu_1^{(k)}}{\operatorname{argmax}} \sum_{i=1}^{n} d_i^{(k)} [-\frac{1}{2}(x_i - \mu_1^{(k)})^T \Sigma_1^{-1} (x_i - \mu_1^{(k)})]$$

$$\frac{\partial g(\theta|\theta^{(k)})}{\partial \mu_1^{(k)}} = -\sum_{i=1}^{n} d_i^{(k)} [-\frac{1}{2}(x_i - \mu_1^{(k)}) \Sigma_1^{-1}] = 0$$

$$\mu_1^{(k+1)} = \frac{\sum_{i=1}^{n} d_i^{(k)} x_i}{\sum_{i=1}^{n} d_i^{(k)}}$$

$\Sigma_1^{(k+1)}$:

$$\Sigma_1^{(k+1)} = \underset{\mu_1^{(k)}}{\operatorname{argmax}} \sum_{i=1}^{n} d_i^{(k)} [-\frac{1}{2} log(|\Sigma_1^{(k)}|) - \frac{1}{2}(x_i - \mu_1^{(k)})^T \Sigma_1^{-1(k)} (x_i - \mu_1^{(k)})]$$

$$\frac{\partial g(\theta|\theta^{(k)})}{\partial \Sigma_1^{(k)}} = \sum_{i=1}^{n} d_i^{(k)} [-\frac{1}{2} \frac{\partial log(|\Sigma_1^{(k)}|)}{\partial \Sigma_1^{(k)}} - \frac{1}{2} \frac{\partial (x_i - \mu_1^{(k)})^T \Sigma_1^{-1(k)} (x_i - \mu_1^{(k)})}{\partial \Sigma_1^{(k)}}]$$

Also, we know that : $\frac{\partial log(|\Sigma_1^{(k)}|)}{\partial \Sigma_1^{(k)}} = \Sigma_1^{-1}$
$\frac{\partial (x_i - \mu_1^{(k)})^T \Sigma_1^{-1(k)} (x_i - \mu_1^{(k)})}{\partial \Sigma_1^{(k)}} = -\Sigma_1^{-T} (x_i - \mu_1^{(k)})^T (x_i - \mu_1^{(k)} \Sigma_1^{-T}$

Hence,

$$\frac{\partial g(\theta|\theta^{(k)})}{\partial \Sigma_1^{(k)}} = \sum_{i=1}^{n} d_i^{(k)} [-\frac{1}{2} \Sigma_1^{-1} + \frac{1}{2} \Sigma_1^{-T} (x_i - \mu_1^{(k)})^T (x_i - \mu_1^{(k)} \Sigma_1^{-T}] = 0$$

$$\Sigma_1^{(k+1)} = \frac{\sum_{i=1}^{n} d_i^{(k)} (x_i - \mu_1^{(k+1)})(x_i - \mu_1^{(k+1)})^T}{\sum_{i=1}^{n} d_i^{(k)}}$$

In a similar way,

$$\Sigma_2^{(k+1)} = \frac{\sum_{i=1}^{n} (1 - d_i^{(k)})(x_i - \mu_2^{(k+1)})(x_i - \mu_2^{(k+1)})^T}{\sum_{i=1}^{n} (1 - d_i^{(k)})}$$

$$\mu_2^{(k+1)} = \underset{\mu_1^{(k)}}{\operatorname{argmax}} \sum_{i=1}^{n} (1 - d_i^{(k)}) [-\frac{1}{2}(x_i - \mu_2^{(k)})^T \Sigma_2^{-1} (x_i - \mu_2^{(k)})]$$

- [15 points] Implement the EM algorithm using the formulas you have. You need to give a reasonable initial value such that the algorithm will converge. Make sure that you properly document each step and report the final results (all parameter estimates). For this question, you may use other packages to calculate the Gaussian densities.

```
# EM

library(lattice)
library(mvtnorm)

load('faithful.Rda')
```

```r
x = faithful

# Initialize parameters
mu1_hat = c(3, 80)
sigma1_hat = matrix(c(0.1, 0, 0, 10), 2, 2)

mu2_hat = c(3.5, 60)
sigma2_hat = matrix(c(0.1, 0, 0, 50), 2, 2)

pi_hat = 0.5
pi_hat_previous = -1

# Iterations (maximum number of iterations is 1000)
for(iteration in 1:1000){

  cat('\nIteration ', iteration, ': ')

  # E step

  # calculate P(Z = 1, x | theta)
  set.seed(iteration)
  d1 = pi_hat * dmvnorm(x, mean = mu1_hat, sigma = sigma1_hat)

  # calculate P(Z = 0, x | theta)
  set.seed(iteration)
  d2 = (1 - pi_hat) * dmvnorm(x, mean = mu2_hat, sigma = sigma2_hat)

  # update conditional probability of Z = 1, or, P(Z = 1 | theta, x)
  posterior_1 = d1 / (d1 + d2)

  # update conditional probability of Z = 0, or, P(Z = 0 | theta, x)
  posterior_2 = d2 / (d1 + d2)

  # M step

  pi_hat_previous = pi_hat
  # update pi_hat
  pi_hat = mean(posterior_1)

  # update mu1_hat
  mu1_hat = colSums( posterior_1 * x ) / sum(posterior_1)
  # update mu2_hat
  mu2_hat = colSums( posterior_2 * x ) / sum(posterior_2)

  # update sigma1_hat
  sigma1_hat = (t(posterior_1 * sweep(x, 2, mu1_hat)) %*% as.matrix(sweep(x, 2, mu1_hat))) / sum(posteri

  # update sigma2_hat
  sigma2_hat = (t(posterior_2 * sweep(x, 2, mu2_hat)) %*% as.matrix(sweep(x, 2, mu2_hat))) / sum(posteri

  cat('pi_hat', pi_hat, '\n') # final value of pi_hat : 0.6441271

  if(all.equal(pi_hat, pi_hat_previous)==T){
```

```r
      cat('\nmu1_hat : ', '\n')
      print(mu1_hat)
      cat('\nmu2_hat : ', '\n')
      print(mu2_hat)
      cat('\nsigma1_hat : ', '\n')
      print(sigma1_hat)
      cat('\nsigma2_hat : ', '\n')
      print(sigma2_hat)
      break
    }
}
```

```
##
## Iteration  1 : pi_hat 0.1986198
##
## Iteration  2 : pi_hat 0.2336633
##
## Iteration  3 : pi_hat 0.2948807
##
## Iteration  4 : pi_hat 0.3635225
##
## Iteration  5 : pi_hat 0.4255568
##
## Iteration  6 : pi_hat 0.4730465
##
## Iteration  7 : pi_hat 0.5095165
##
## Iteration  8 : pi_hat 0.540693
##
## Iteration  9 : pi_hat 0.569819
##
## Iteration  10 : pi_hat 0.5965163
##
## Iteration  11 : pi_hat 0.6173254
##
## Iteration  12 : pi_hat 0.6310979
##
## Iteration  13 : pi_hat 0.6404256
##
## Iteration  14 : pi_hat 0.6433725
##
## Iteration  15 : pi_hat 0.6439538
##
## Iteration  16 : pi_hat 0.644086
##
## Iteration  17 : pi_hat 0.6441173
##
## Iteration  18 : pi_hat 0.6441248
##
## Iteration  19 : pi_hat 0.6441266
##
## Iteration  20 : pi_hat 0.644127
##
```

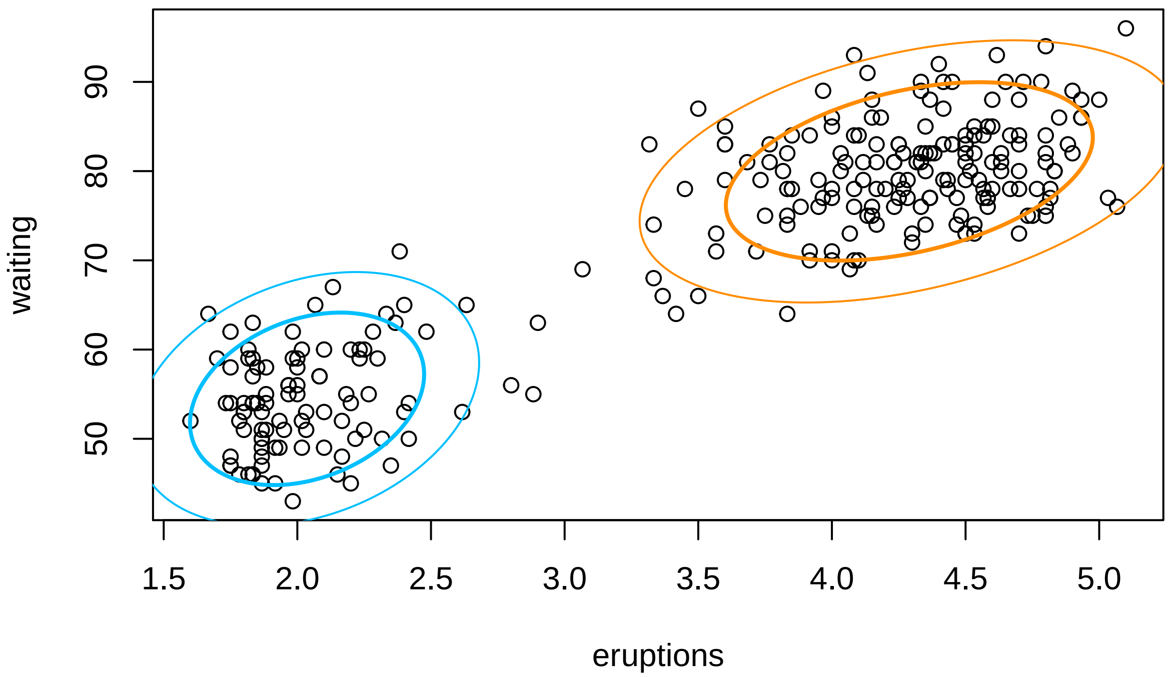
```
## Iteration  21 : pi_hat 0.6441271
##
## Iteration  22 : pi_hat 0.6441271
##
## Iteration  23 : pi_hat 0.6441271
##
## mu1_hat :
## eruptions   waiting
##  4.289662 79.968115
##
## mu2_hat :
## eruptions   waiting
##  2.036388 54.478516
##
## sigma1_hat :
##           eruptions    waiting
## eruptions 0.1699684  0.9406093
## waiting   0.9406093 36.0462106
##
## sigma2_hat :
##            eruptions    waiting
## eruptions 0.06916768  0.4351677
## waiting   0.43516766 33.6972823
```

- [10 points] Plot your final results intuitively. You may borrow the idea at the Wikipedia page, or use the following code, which plots the contours of a Gaussian distribution.

```r
# plot the current fit
library(mixtools)
plot(faithful)

addellipse <- function(mu, Sigma, ...)
{
  ellipse(mu, Sigma, alpha = .05, lwd = 1, ...)
  ellipse(mu, Sigma, alpha = .25, lwd = 2, ...)
}

addellipse(mu1_hat, sigma1_hat, col = "darkorange")
addellipse(mu2_hat, sigma2_hat, col = "deepskyblue")
```



Question 2 [45 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. We only consider the train-test split, with the pre-defined `zip.train` and `zip.test`, and no cross-validation is needed. However, **use `zip.test` as the training data, and `zip.train` as the testing data!**

- [15 points] 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. Report the model fitting results (such as a confusion table and misclassification rates). Which digit seems to get misclassified the most?

```
# LDA
# Handwritten Digit Recognition Data
library(ElemStatLearn)

# this is the training data!
train = zip.test

# this is the testing data!
test = zip.train
```

```
# LDA
```

```r
prior_prob = rep(0, 10) # prior_prob[i] represents prior_prob for digit (i-1)

for(K in 0:9){
  prior_prob[K + 1] = sum(train[,1]==(K)) / nrow(train)
}

# class_mean[i, ] represents class_mean for digit (i-1)
class_mean = matrix(0, nrow = 10, ncol = ncol(train)-1)

for(K in 0:9){
  class_mean[K + 1, ] = colMeans(train[train[,1]==(K), 2:ncol(train)])
}

variance = matrix(0, nrow = 256, ncol = 256)

for(K in 0:9){
    rows = train[train[,1]==K, 2:ncol(train)]
    diff_from_mean = sweep(rows, 1,  class_mean[K+1, ] )
    variance = variance + (t(diff_from_mean) %*% diff_from_mean)
}

variance = variance / (nrow(train) - 10)

variance_inv = solve(variance)

yhat = rep(-1, nrow(test))

for(i in 1:nrow(test)){
  x_ = test[i, 2:ncol(test)]

  max_a_posteriori = rep(0, 10)

  for(K in 0:9){
    digit_ = K
    class_mean_ = class_mean[digit_+1, ]
    prior_prob_ = prior_prob[digit_+1]

    distance_from_class_mean = (x_ - class_mean_)
    distance_from_class_mean = matrix(distance_from_class_mean, 1, 256)
    product_term = distance_from_class_mean %*% variance_inv %*% t(distance_from_class_mean)
    product_term = -0.5 * product_term
    log_term = log(prior_prob_)
    total = product_term + log_term

    max_a_posteriori[K + 1] = total
  }

  yhat[i] = (order(max_a_posteriori)[10] - 1)
}

accuracy = mean(yhat==test[,1])
error_rate = 1 - accuracy
```

```r
cat('Error rate : ', round(error_rate*100, 2), '%') # 12.3 %
```

```
## Error rate :  12.3 %
```

```r
library(caret)
confusionMatrix_ = confusionMatrix(data = as.factor(yhat),
                 reference = as.factor(test[,1]))

print(confusionMatrix_$table)
```

```
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0 1135    0   11   10    1   15   15    1   11    2
##          1    1  998   16    2   30    3    9    6    9    2
##          2    5    1  591   16   14    5   15    1    7    1
##          3   12    0   23  555    0   48    1    3   25    2
##          4    5    0   25    2  537   23   17    7   15   49
##          5   12    1    2   20    0  432    9    1   11    2
##          6   13    1    8    0    8   14  581    0    4    0
##          7    0    0   11   14    1    2    1  564    3   27
##          8    7    1   29   31    9   10   14    2  450    8
##          9    4    3   15    8   52    4    2   60    7  551
```

```r
accuracy_of_each_class = diag(confusionMatrix_$table) / colSums(confusionMatrix_$table)

print('Digit with least accuracy : ')
```

```
## [1] "Digit with least accuracy : "
```

```r
print(names(accuracy_of_each_class[order(accuracy_of_each_class)][1])) # 5
```

```
## [1] "5"
```

- [15 points] QDA does not work directly in this example because we do not have enough samples to estimate the inverse 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 estimation 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 performence of the model?

```r
# QDA

list_of_cov_matrices = list()

for(K in 0:9){
  rows = train[train[,1]==K, 2:ncol(train)]
  class_mean_ = class_mean[K+1, ]
```

```r
  class_mean_ = matrix(class_mean_, 1, 256)
  diff = sweep(rows, 2, class_mean_)
  list_of_cov_matrices[[K + 1]] = (t(diff) %*% diff) / (nrow(rows)-1)
}

list_of_adj_cov_matrices = list()
list_of_det_of_adj_cov_matrices = list()
list_of_inv_of_cov_matrices = list()

alpha_values = alpha_grid = seq(0,0.99,0.03)

for(alpha in alpha_values){

  for(K in 0:9){
    variance_mat = (alpha * list_of_cov_matrices[[K + 1]]) + ((1 - alpha) * variance)
    list_of_adj_cov_matrices[[K + 1]] = variance_mat
    list_of_det_of_adj_cov_matrices[[K + 1]] = -0.5*log(abs(det(variance_mat)))
    variance_inv = solve(variance_mat)
    list_of_inv_of_cov_matrices[[K + 1]] = variance_inv
  }

  yhat = rep(-1, nrow(test))

  for(i in 1:nrow(test)){
    x_ = test[i, 2:ncol(test)]

    max_a_posteriori = rep(0, 10)

    for(K in 0:9){
      digit_ = K
      class_mean_ = class_mean[digit_+1, ]
      prior_prob_ = prior_prob[digit_+1]

      distance_from_class_mean = (x_ - class_mean_)
      distance_from_class_mean = matrix(distance_from_class_mean, 1, 256)
      product_term = distance_from_class_mean %*% list_of_inv_of_cov_matrices[[K + 1]] %*% t(distance_f
      product_term = -0.5 * product_term
      total = list_of_det_of_adj_cov_matrices[[K + 1]] + product_term + log(prior_prob_)

      max_a_posteriori[K + 1] = total
    }

    yhat[i] = (order(max_a_posteriori)[10] - 1)
  }

  cat('alpha', alpha, ': Accuracy', mean(yhat==test[,1]), '\n')
  # optimal alpha 0.12 : Accuracy 0.9264847
}
```

```
## alpha 0 : Accuracy 0.8769716
## alpha 0.03 : Accuracy 0.914415
## alpha 0.06 : Accuracy 0.9227815
## alpha 0.09 : Accuracy 0.9251132
```

```
## alpha 0.12 : Accuracy 0.9264847
## alpha 0.15 : Accuracy 0.9259361
## alpha 0.18 : Accuracy 0.9262104
## alpha 0.21 : Accuracy 0.9253875
## alpha 0.24 : Accuracy 0.9257989
## alpha 0.27 : Accuracy 0.9248388
## alpha 0.3 : Accuracy 0.9244274
## alpha 0.33 : Accuracy 0.9237416
## alpha 0.36 : Accuracy 0.9225072
## alpha 0.39 : Accuracy 0.9226444
## alpha 0.42 : Accuracy 0.9212728
## alpha 0.45 : Accuracy 0.920587
## alpha 0.48 : Accuracy 0.9203127
## alpha 0.51 : Accuracy 0.9194898
## alpha 0.54 : Accuracy 0.9196269
## alpha 0.57 : Accuracy 0.9189412
## alpha 0.6 : Accuracy 0.9186668
## alpha 0.63 : Accuracy 0.9189412
## alpha 0.66 : Accuracy 0.9183925
## alpha 0.69 : Accuracy 0.9183925
## alpha 0.72 : Accuracy 0.9181182
## alpha 0.75 : Accuracy 0.9179811
## alpha 0.78 : Accuracy 0.1378412
## alpha 0.81 : Accuracy 0.1378412
## alpha 0.84 : Accuracy 0.1378412
## alpha 0.87 : Accuracy 0.08832808
## alpha 0.9 : Accuracy 0.08832808
## alpha 0.93 : Accuracy 0.08832808
## alpha 0.96 : Accuracy 0.08832808
## alpha 0.99 : Accuracy 0.08832808
```

```r
# Optimal alpha value
alpha = 0.12
```

```r
# Performance on optimal alhpa value

for(K in 0:9) {
  variance_mat = (alpha * list_of_cov_matrices[[K + 1]]) + ((1 - alpha) * variance)
  list_of_adj_cov_matrices[[K + 1]] = variance_mat
  list_of_det_of_adj_cov_matrices[[K + 1]] = -0.5 * log(abs(det(variance_mat)))
  variance_inv = solve(variance_mat)
  list_of_inv_of_cov_matrices[[K + 1]] = variance_inv
}

yhat = rep(-1, nrow(test))

for (i in 1:nrow(test)) {
  x_ = test[i, 2:ncol(test)]

  max_a_posteriori = rep(0, 10)

  for (K in 0:9) {
    digit_ = K
    class_mean_ = class_mean[digit_ + 1,]
```

```r
    prior_prob_ = prior_prob[digit_ + 1]

    distance_from_class_mean = (x_ - class_mean_)
    distance_from_class_mean = matrix(distance_from_class_mean, 1, 256)
    product_term = distance_from_class_mean %*% list_of_inv_of_cov_matrices[[K + 1]] %*% t(distance_from
    product_term = -0.5 * product_term
    total = list_of_det_of_adj_cov_matrices[[K + 1]] + product_term + log(prior_prob_)

    max_a_posteriori[K + 1] = total
  }

  yhat[i] = (order(max_a_posteriori)[10] - 1)
}

accuracy = mean(yhat==test[,1])
error_rate = 1 - accuracy

cat('Error rate : ', round(error_rate*100, 2), '%') # 7.35 %
```

```
## Error rate :  7.35 %
```

```r
library(caret)
confusionMatrix_ = confusionMatrix(data = as.factor(yhat),
                reference = as.factor(test[,1]))

print(confusionMatrix_$table)
```

```
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0 1167    0    2    4    0    8   12    0    7    2
##          1    4 1003   10    2   19    0    7    5    7    1
##          2    4    2  688   16    8    4    3    1    2    0
##          3    2    0    6  587    0   44    0    1   10    1
##          4    2    0    7    0  565   11    3    3    2   27
##          5    1    0    1   23    2  456    2    2   11    3
##          6    9    0    1    0    4   17  631    0    3    0
##          7    1    0    2    7    1    1    0  575    2   16
##          8    4    0   11   16    5   12    5    2  491    2
##          9    0    0    3    3   48    3    1   56    7  592
```

```r
accuracy_of_each_class = diag(confusionMatrix_$table) / colSums(confusionMatrix_$table)

print('Digit with least accuracy : ')
```

```
## [1] "Digit with least accuracy : "
```

```r
print(names(accuracy_of_each_class[order(accuracy_of_each_class)][1])) # 5
```

```
## [1] "5"
```

Since the optimal value of alpha is 0.12 in regularized discriminant analysis, we can conclude following about the underlying data :
- The optimal alpha is close to 0 (instead of being close to 1), which means the regularized covariance matrix tends to look like that of the common covariance matrix of LDA.
- The is mainly because the number of samples per class (for each digit) is low as compared to the dimension of the data. Hence, the sample covariance matrix can be accurately estimated for each class. A biased estimator will be favored due to the bias-variance trade-off. Hence, the optimal value of alpha is shifted toward 0.

```
for(K in 0:9){
   cat('Digit', K, ':', sum(train[,1]==(K)), '\n')
 }
```

```
## Digit 0 : 359
## Digit 1 : 264
## Digit 2 : 198
## Digit 3 : 166
## Digit 4 : 200
## Digit 5 : 160
## Digit 6 : 170
## Digit 7 : 147
## Digit 8 : 166
## Digit 9 : 177
```

- [15 points] Naive Bayes is another approach that can be used for discriminant analysis. Instead of jointly modeling the density of pixels as multivariate Gaussian, we treat them independently.

$$f_k(x) = \prod_j f_{kj}(x_j).$$

  However, this brings up other difficulties, such as the dimensionality problem. Hence, we consider first to reduce the dimension of the data and then use independent Gaussian distributions to model the density. It proceeds with the following:

  - Perform PCA on the training data and extract the first ten principal components. For this question, you can use a built-in function.
  - model the density based on the principal components using the Naive Bayes approach. Assume that each $f_{kj}(x_j)$ is a Gaussian density and estimate their parameters.
  - Apply the model to the testing data and evaluate the performance. Report the results similarly to the previous two parts.

```
# Naive Bayes

# PCA on train data
train_pca = prcomp(train[, 2:ncol(train)], rank. = 10, center = TRUE, scale. = FALSE)
train_pca_x = train_pca$x

col_means = matrix(0, nrow = 10, ncol = 10)
col_sd = matrix(0, nrow = 10, ncol = 10)

for(K in 0:9){
  rows = (train[,1]==K)
  col_means[K + 1, ] = colMeans(train_pca_x[rows, ])
  col_sd[K + 1, ] = apply(train_pca_x[rows, ], 2, sd)
```

```r
}

yhat = rep(-1, nrow(test))

# PCA on test data
test_pca_x = test[,-1] %*% train_pca$rotation

for(i in 1:nrow(test)){
      x_ = test_pca_x[i, ]

      max_a_posteriori = rep(0, 10)

      for(K in 0:9){
        digit_ = K
        prior_prob_ = prior_prob[digit_+1]
        col_mean_ = col_means[K+1, ]
        col_sd_ = col_sd[K+1, ]

        prod_ = prod(dnorm(x_,  mean = col_mean_, sd = col_sd_))
        total = prior_prob_*prod_

        max_a_posteriori[K + 1] = total
      }

      yhat[i] = (order(max_a_posteriori)[10] - 1)
}

accuracy = mean(yhat==test[,1])
error_rate = 1 - accuracy

cat('Error rate : ', round(error_rate*100, 2), '%') # 30.67 %
```

```
## Error rate :  30.67 %
```

```r
library(caret)
confusionMatrix_ = confusionMatrix(data = as.factor(yhat),
                reference = as.factor(test[,1]))

print(confusionMatrix_$table)
```

```
##            Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0 1092  601   13   21    6   48  130    3   42    1
##          1    0  284    0    0    2    0    1    1    0    0
##          2    6    1  534   20   10    4   12    3    3    0
##          3    9    0   19  498    0   26    0    0   15    0
##          4    7    0   56    0  250   10    2    1    0    9
##          5   15    9   20   69    0  418    3    1   47    1
##          6   65   92   39    6   38   28  514    2   19    0
##          7    0    0   20   19    7    2    0  534    2   42
##          8    0    5   16    5    9    2    0    1  342    2
##          9    0   13   14   20  330   18    2   99   72  589
```

```r
accuracy_of_each_class = diag(confusionMatrix_$table) / colSums(confusionMatrix_$table)

print('Digit with least accuracy : ')
```

```
## [1] "Digit with least accuracy : "
```

```r
print(names(accuracy_of_each_class[order(accuracy_of_each_class)][1])) # 1
```

```
## [1] "1"
```

Question 3 [15 Points] Penalized Logistic Regression

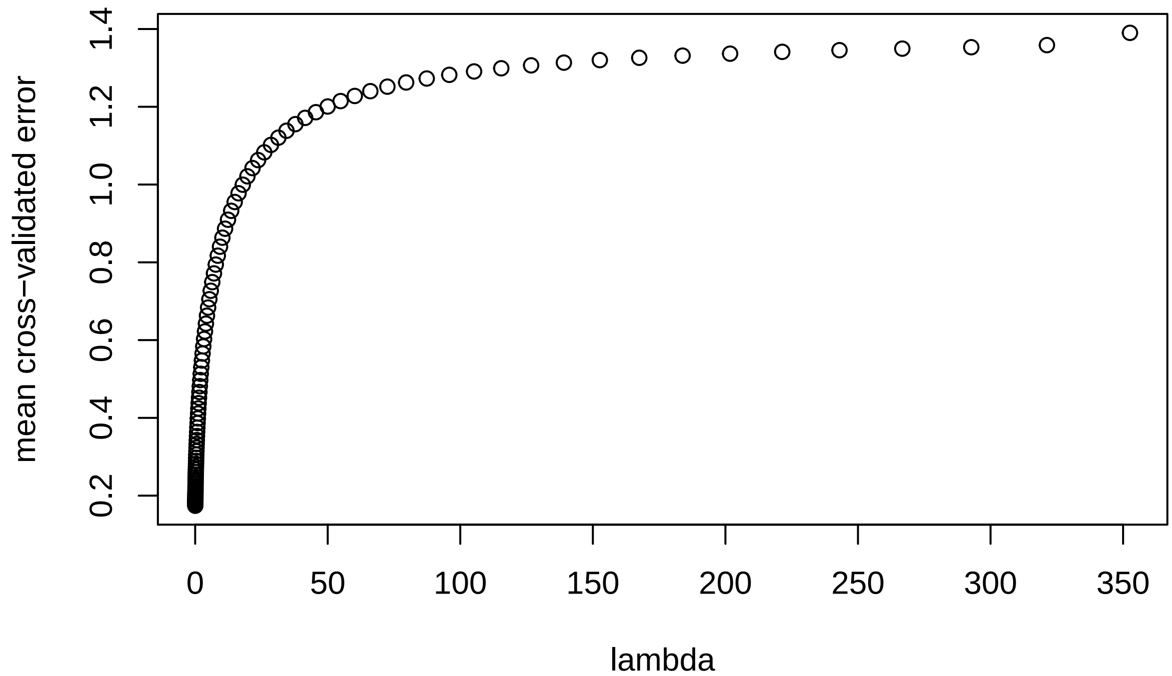
Take digits 2 and 4 from the handwritten digit recognition data and perform logistic regression using penalized linear regression (with Ridge penalty). Use the same train-test setting we had in Question 2. However, for this question, you need to perform cross-validation on the training data to select the best tuning parameter. This can be done using the `glmnet` package. Evaluate and report the performance on the testing data, and summarize your fitting result. A researcher is interested in which regions of the pixel are most relevant in differentiating the two digits. Use an intuitive way to present your findings.

```r
# Penalized Logistic Regression

library(glmnet)
train_subset = train[(train[,1]==2) | (train[,1]==4),]
test_subset = test[(test[,1]==2) | (test[,1]==4),]

set.seed(1)
cv_fit = cv.glmnet(train_subset[, 2:ncol(train_subset)],
                   train_subset[, 1],
                   alpha = 0,
                   family = 'binomial')

plot(cv_fit$lambda, cv_fit$cvm, xlab = 'lambda', ylab = 'mean cross-validated error')
```




```
# Optimal lambda
cv_fit$lambda.min # 0.0352596
```

```
## [1] 0.0352596
```

```
set.seed(1)
x_standardized = scale(train_subset[, 2:ncol(train_subset)], center = FALSE, scale = TRUE)
m1 = glmnet(
  x_standardized,
  train_subset[, 1],
  alpha = 0,
  family = 'binomial',
  lambda = cv_fit$lambda.min,
  standardize = FALSE
)

yhat = predict(m1, test_subset[, 2:ncol(test_subset)], type='class')

accuracy = mean(yhat==test_subset[,1])
error_rate = 1 - accuracy

cat('Error rate : ', round(error_rate*100, 2), '%') # 2.39 %
```

```
## Error rate :  2.39 %
```

```r
library(caret)
confusionMatrix_ = confusionMatrix(data = as.factor(yhat),
                reference = as.factor(test_subset[,1]))

print(confusionMatrix_)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   2   4
##          2 714  16
##          4  17 636
##
##                Accuracy : 0.9761
##                  95% CI : (0.9667, 0.9835)
##     No Information Rate : 0.5286
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9521
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.9767
##             Specificity : 0.9755
##          Pos Pred Value : 0.9781
##          Neg Pred Value : 0.9740
##              Prevalence : 0.5286
##          Detection Rate : 0.5163
##    Detection Prevalence : 0.5278
##       Balanced Accuracy : 0.9761
##
##        'Positive' Class : 2
##
```

```r
accuracy_of_each_class = diag(confusionMatrix_$table) / colSums(confusionMatrix_$table)
accuracy_of_each_class
```

```
##         2         4
## 0.9767442 0.9754601
```

```r
# Most important 20 pixels in order
important_pixels = rev(order(m1$beta))[1:20]
important_pixels
```

```
##  [1] 115 116 138 117 130 114 100 105 154  99 121 218 139  29  28  73 101  11 122 129
```

```r
pixel_map = matrix(0,nrow = 1,ncol = 256)
for(i in important_pixels){
  pixel_map[i] = 1
}
image(matrix(pixel_map, 16, 16), useRaster=TRUE, axes=FALSE)
```