HW5_dbagchi2

Diptendra Nath Bagchi (dbagchi2@illinois.edu)

13 April, 2020

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:

a. [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 Z given x and $\boldsymbol{\theta}^{(k)}$ with notations used in our lectures.

The pdf for a 2 dimensional guassian Mixture model is given as:

$p(x) = (1 - \pi)\, \phi_{\mu_1,\Sigma_1}(x) + \pi\, \phi_{\mu_2,\Sigma_2}(x)$

Thus the log likelihood will be:

$\ell(x|\theta) = \sum_{i=1}^{n} \log[(1 - \pi)\, \phi_{\mu_1,\Sigma_1}(x) + \pi\, \phi_{\mu_2,\Sigma_2}(x)]$

Let Z be the latent group variable from which the observed values comes from. Hence, for a two variable case, the latent variable follows a bernoulli random variable with probability, $P(Z_i = 1) = \pi$.

Then the likelihood of the complete data is the follwing:

$\ell(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta}) = \sum_{i=1}^{n} \left[(1 - z_i) \log \phi_{\mu_1,\Sigma_1}(x) + z_i \log \phi_{\mu_2,\Sigma_2}(x)\right] + \sum_{i=1}^{n} \left[(1 - z_i) \log(1 - \pi) + z_i \log \pi\right]$

where - $\mathbf{x}$: Vector of all observed values

- $\mathbf{z}$: Vector of the latent variables
- $\boldsymbol{\theta}$: Vector of all unknown parameters.

The complete data likelihood is difficult to optimize using standard optimization techniques because the summation is inside the log. Hence, we need another technique that can find a local optima for this likelihood. It turns out that we can solve this problem iteratively using E-M Algorithm - which is defined below.

E-Step

We do not know the exact value of the z_i for each x_i. Hence, we will calculate the probability, $P(Z_i = 1|\boldsymbol{\theta}, \mathbf{x})$. We use the Bayes Theorem to calculate that - which is the follwoing.

$p_i = p(Z_i = 1|\boldsymbol{\theta}, \mathbf{x}) = \frac{p(Z_i=1,x_i|\boldsymbol{\theta})}{p(x_i|\boldsymbol{\theta})}$
$= \frac{p(Z_i=1,x_i|\boldsymbol{\theta})}{p(Z_i=1,x_i|\boldsymbol{\theta})+p(Z_i=0,x_i|\boldsymbol{\theta})}$

b. [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:

Derivation:

From part 1a, we get the following complete data likelihood,

$\ell(x_i, p_i|\boldsymbol{\theta}) = \sum_{i=1}^n \left[(1-p_i)\log\phi_{\mu_1,\Sigma_1}(x_i) + p_i \log\phi_{\mu_2,\Sigma_2}(x_i)\right]$
$+\sum_{i=1}^n \left[(1-p_i)\log(1-\pi) + p_i\log\pi\right]$

Solve for μ_1. As we can see from the above likelihood, that the parameters are additive and hence it is easy to differentiate it with one parameter at a time.

$$\ell(x_i, p_i|\boldsymbol{\theta}) = \sum_{i=1}^n (1-p_i)\left(-\frac{1}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma_1| - \frac{1}{2}(\mathbf{x_i}-\mu_\mathbf{1})^\mathbf{T}\boldsymbol{\Sigma}_\mathbf{1}^{-\mathbf{1}}(\mathbf{x_i}-\mu_\mathbf{1})\right)$$

Derivating wrt μ_1, we get

$$\frac{\partial \mathbf{w}^\mathbf{T}\mathbf{A}\mathbf{w}}{\partial \mathbf{w}} = \mathbf{2Aw}$$

$\frac{\partial}{\partial\mu}\ell(\mathbf{x_i}, p_i|\boldsymbol{\theta}) = \sum_{i=1}^n 2(\mathbf{1}-\mathbf{p_i})\boldsymbol{\Sigma}_\mathbf{1}^{-\mathbf{1}}(\mathbf{x_i}-\mu_\mathbf{1}) = 0$
$\sum_{i=1}^n (\mathbf{1}-\mathbf{p_i})\boldsymbol{\Sigma}_\mathbf{1}^{-\mathbf{1}}(\mathbf{x_i}-\mu_\mathbf{1}) = 0$
$\sum_{i=1}^n (1-p_i)(x_i) - \sum_{i=1}^n (1-p_i)(\mu_1) = 0$
$\widehat{\mu}_1 = \frac{\sum_{i=1}^n (1-\widehat{p}_i)x_i}{\sum_{i=1}^n (1-\widehat{p}_i)}$

Similarly,

$$\widehat{\mu}_2 = \frac{\sum_{i=1}^n \widehat{p}_i x_i}{\sum_{i=1}^n \widehat{p}_i}$$

Derivating wrt to Σ_1, we get the following,

$$\frac{\partial}{\partial A}x^t A x = \frac{\partial}{\partial A}tr[x^T x A] = [xx^t]^T = x^{TT}x^T = xx^T$$

$\ell(\mathbf{x_i}, p_i|\boldsymbol{\theta}) = \mathrm{C} - \frac{(1-p_i)}{2}\log|\Sigma_1| - \frac{1}{2}\sum_{i=1}^n (1-p_i)(\mathbf{x_i}-\mu_\mathbf{1})^\mathbf{T}\boldsymbol{\Sigma}_\mathbf{1}^{-\mathbf{1}}(\mathbf{x_i}-\mu_\mathbf{1})$
$= \mathrm{C} + \frac{(1-p_i)}{2}\log|\Sigma_1^{-1}| - \frac{1}{2}\sum_{i=1}^n (1-p_i)tr[(\mathbf{x_i}-\mu_\mathbf{1})(\mathbf{x_i}-\mu_\mathbf{1})^\mathbf{T}\boldsymbol{\Sigma}_\mathbf{1}^{-\mathbf{1}}]$
$\frac{\partial}{\partial\Sigma_1^{-1}}\ell(\mathbf{x_i}, p_i|\boldsymbol{\theta}) = \frac{(1-p_i)}{2}\Sigma_1 - \frac{1}{2}\sum_{i=1}^n (1-p_i)(\mathbf{x_i}-\mu_\mathbf{1})(\mathbf{x_i}-\mu_\mathbf{1})^T$

Because, $\Sigma_1^T = \Sigma_1$

Equating it to 0 and solving for the Σ_1 gives us,

$$\widehat{\Sigma}_1 = \frac{\sum_{i=1}^n (1-\widehat{p}_i)(\mathbf{x_i}-\mu_\mathbf{1})(\mathbf{x_i}-\mu_\mathbf{1})^T}{\sum_{i=1}^n (1-\widehat{p}_i)}$$

Similarly,

$$\widehat{\Sigma}_2 = \frac{\sum_{i=1}^n (1-\widehat{p}_i)(\mathbf{x_i}-\mu_\mathbf{2})(\mathbf{x_i}-\mu_\mathbf{2})^T}{\sum_{i=1}^n (1-\widehat{p}_i)}$$

Finally π,

Differentiating wrt π, we get,

$$\frac{\partial}{\partial \pi}\ell(\mathbf{x_i}, p_i|\boldsymbol{\theta}) = \sum_{i=1}^{n} \frac{-(1-p_i)}{(1-\pi)} + \frac{p_i}{\pi}$$

Equating to zero and solving for pi, we get

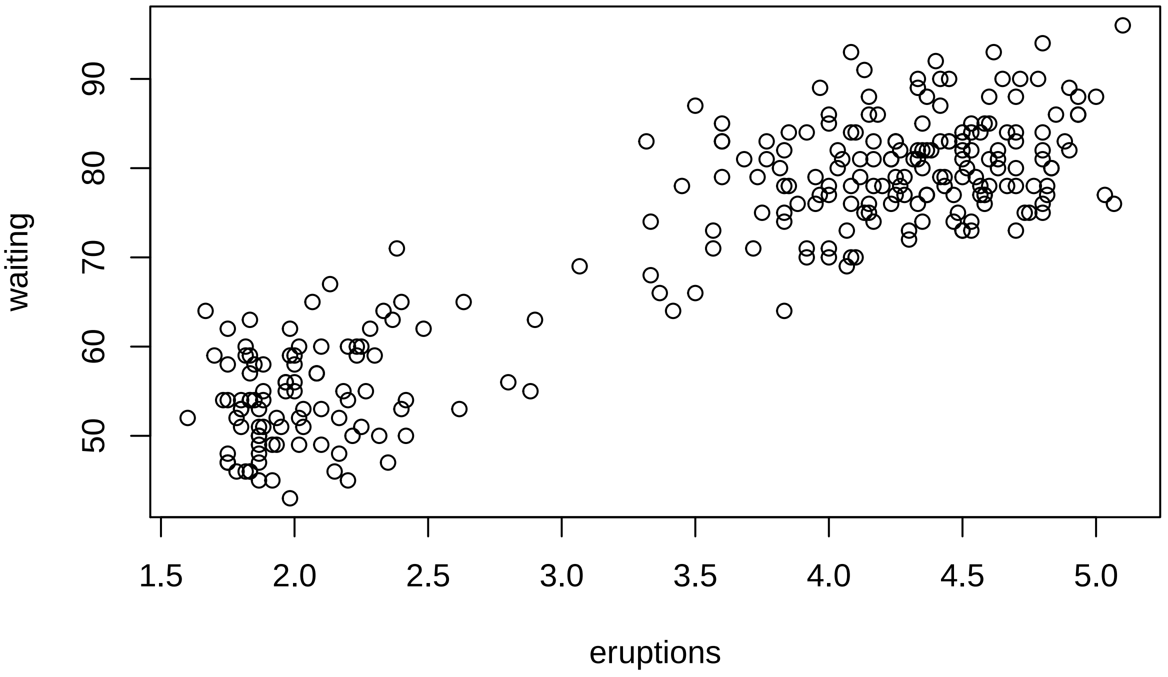
$\frac{n\pi}{1-n\pi} = \sum_{i=1}^{n} \frac{p_i}{1-p_i}$

$\widehat{\pi} = \sum_{i=1}^{n} \widehat{p}_i/n$

Hence Proved.

c. [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.

```r
faithful = read.csv("data/faithful.csv")
faithful = faithful[, -1]
plot(faithful)
```



```r
# Expectation Step in the EM Algorithm
e_step = function(x, mu1, mu2, cov1, cov2, pi1, pi2)  {
  # Using the bayes rule, we have
  prob_x_given_pi1_mul_pi1 = mvtnorm::dmvnorm(x = x, mean = mu1, sigma = cov1) * pi1
  prob_x_given_pi2_mul_pi2 = mvtnorm::dmvnorm(x = x, mean = mu2, sigma = cov2) * pi2
  # Denominatior / Normalizer
  denominator = prob_x_given_pi1_mul_pi1 + prob_x_given_pi2_mul_pi2
  # Posterior probability (Probability of x_i belongs to pi_i given x_i)
  post_pi1_given_xi = prob_x_given_pi1_mul_pi1 / denominator
  post_pi2_given_xi = prob_x_given_pi2_mul_pi2 / denominator
  # Complete expecation step
  total = 0
  for (i in 1:nrow(x)) {
    temp_mu1 = post_pi1_given_xi[i] *
      (log(pi1) - (0.5 * determinant(cov1, logarithm = TRUE)$modulus) -
      (0.5 * as.matrix(x[1, ] - mu1) %*% solve(cov1) %*% as.matrix(t(x[1, ] - mu1))) -
      log(2 * pi)
      )

    temp_mu2 = post_pi2_given_xi[i] *
      (log(pi2) - (0.5 * determinant(cov2, logarithm = TRUE)$modulus) -
      (0.5 * as.matrix(x[1, ] - mu2) %*% solve(cov2) %*% as.matrix(t(x[1, ] - mu2))) -
      log(2 * pi)
      )
    internal_addition = temp_mu1 + temp_mu2
    total = total + internal_addition
  }
  # Return the list of both the posterior probability
  return(list("post1_gaussian" = post_pi1_given_xi,
              "post2_gaussian" = post_pi2_given_xi,
              "q" = total))
}

# Maximization Step in the EM Algorithm
m_step = function(x, posterior_probability) {
  x = as.matrix(x, nrow = nrow(x))

  # Calculate the sum of the probability from the E step
  sum_posterior_prob_1 = sum(posterior_probability[[1]])
  sum_posterior_prob_2 = sum(posterior_probability[[2]])
  # Calculate the mean of the gaussian distribution
  mu1 = (1 / sum_posterior_prob_1) * (matrix(posterior_probability[[1]], nrow = 1) %*% x)
  mu2 = (1 / sum_posterior_prob_2) * (matrix(posterior_probability[[2]], nrow = 1) %*% x)

  # Standard Deviation
  x_minus_mu1 = t(t(apply(x, 1, function(x) x - mu1)))
  x_minus_mu2 = t(t(apply(x, 1, function(x) x - mu2)))

  sigma1 = ((x_minus_mu1 %*% diag(posterior_probability[[1]])) %*% t(x_minus_mu1)) / sum_posterior_prob
  sigma2 = ((x_minus_mu2 %*% diag(posterior_probability[[2]])) %*% t(x_minus_mu2)) / sum_posterior_prob

  # Tao t+1
  tao1 = sum_posterior_prob_1 / length(posterior_probability[[1]])
  tao2 = sum_posterior_prob_2 / length(posterior_probability[[2]])


  return(list("mu1" = as.vector(mu1), "mu2" = as.vector(mu2),
              "sigma1" = sigma1, "sigma2" = sigma2,
              "tao1" = tao1, "tao2" = tao2))
}
```

Table 1: Mean and Sigma for Cluster 1

Variable	Mean	Variable	Eruptions	Waiting
Eruptions	2.04	Eruptions	0.07	0.44
Waiting	54.48	Waiting	0.44	33.70

```r
for (i in 1:50) {
  if (i == 1) {
    # Initialization
    e.step = e_step(x = faithful, mu1 = c(2, 55), mu2 = c(4.5, 80),
                    cov1 = matrix(c(1, 0.5, 0.5, 1), nrow = 2),
                    cov2 = matrix(c(1, 0.5, 0.5, 1), nrow = 2),
                    pi1 = 0.5, pi2 = 0.5)

    m.step = m_step(x = faithful, posterior_probability = e.step)
    cur_loglik = e.step$q
    loglik_vector = e.step$q
  } else {
    # Repeat E and M steps till convergence
    e.step = e_step(x = faithful, mu1 = m.step$mu1, mu2 = m.step$mu2,
                     cov1 = m.step$sigma1, cov2 = m.step$sigma2,
                     pi1 = m.step$tao1, pi2 = m.step$tao2)
    m.step = m_step(x = faithful, posterior_probability = e.step)
    loglik_vector = c(loglik_vector, e.step$q)
    loglik_diff = abs((cur_loglik - e.step$q))
    if(loglik_diff < 1e-6) {
      break
    } else {
      cur_loglik = e.step$q
    }
  }
}
```

```r
mean1 = data.frame("Variable" = c("Eruptions", "Waiting"),
                "Mean" = m.step$mu1)
sigma1 = data.frame("Variable" = c("Eruptions", "Waiting"),
                "Eruptions" = m.step$sigma1[, 1], "Waiting" = m.step$sigma1[, 2])

kable(list(mean1,sigma1), digits = 2,
  caption = 'Mean and Sigma for Cluster 1',
  booktabs = TRUE, valign = "t", align = "c", ) %>%
  kable_styling("striped", full_width = FALSE, position = "center")
```

```r
mean2 = data.frame("Variable" = c("Eruptions", "Waiting"),
                "Mean" = m.step$mu2)
sigma2 = data.frame("Variable" = c("Eruptions", "Waiting"),
                "Eruptions" = m.step$sigma2[, 1], "Waiting" = m.step$sigma2[, 2])

kable(list(mean2,sigma2), digits = 2,
  caption = 'Mean and Sigma for Cluster 2',
  booktabs = TRUE, valign = "t") %>%
  kable_styling("striped", full_width = FALSE, position = "center")
```

Table 2: Mean and Sigma for Cluster 2

Variable	Mean	Variable	Eruptions	Waiting
Eruptions	4.29	Eruptions	0.17	0.94
Waiting	79.97	Waiting	0.94	36.05

```r
# load the data
#faithful = read.table("..//data//faithful.txt")
# the parameters
mu1 = m.step$mu1
mu2 = m.step$mu2
Sigma1 = m.step$sigma1
Sigma2 = m.step$sigma2

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

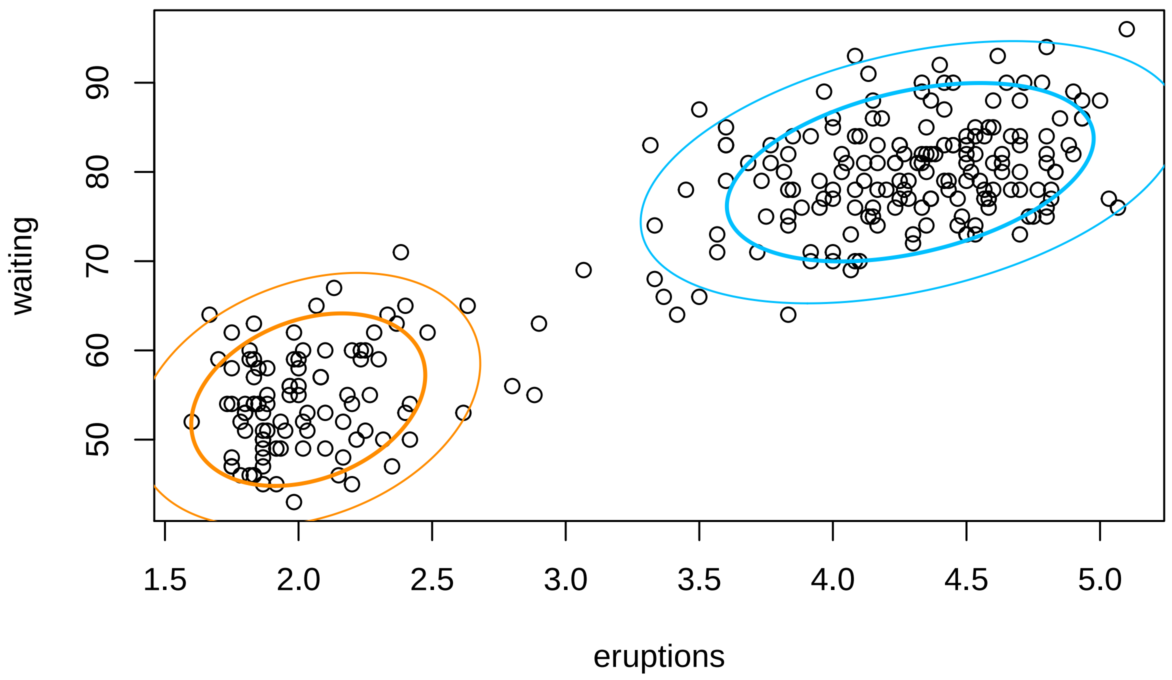
```
## mixtools package, version 1.2.0, Released 2020-02-05
## This package is based upon work supported by the National Science Foundation under Grant No. SES-051
```

```
##
## Attaching package: 'mixtools'
```

```
## The following objects are masked from 'package:mvtnorm':
##
##     dmvnorm, rmvnorm
```

```r
plot(faithful)

addellipse <- function(mu, Sigma, ...)
{
  ellipse(mu, Sigma, alpha = .05, lwd = 1, ...)
  ellipse(mu, Sigma, alpha = .25, lwd = 2, ...)
}
addellipse(mu1, Sigma1, col = "darkorange")
addellipse(mu2, Sigma2, 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!

a. [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?

```r
# Handwritten Digit Recognition Data
library(ElemStatLearn)
# this is the training data!
train = zip.test
# this is the testing data!
test = zip.train
```

```r
k = 0:9
n = nrow(train)
d = ncol(train) - 1
pi = matrix(0, nrow = length(k), ncol = 2)
mu = matrix(0, nrow = length(k), ncol = d + 1)
sigma = matrix(0, nrow = d, ncol = d)
sigma_k = list()
for (i in k) {
  # x for only one kind of k
  x_k = train[train[, 1] == i, -1]
  # Mean and the prior probability
  mu[i + 1, ] = c(i, apply(X = x_k, MARGIN = 2, FUN = mean))
  pi[i + 1, ] = c(i, nrow(x_k) / n)
  # X - MuK
  x_minus_muk = (x_k - mu[i + 1, -1])
  # Variance and Covarinace
  inner_product = (t(x_minus_muk) %*% x_minus_muk)
  sigma_k[[i + 1]] = inner_product / (nrow(x_k) - 1) # This is to be used in QDA
  sigma = sigma + inner_product
}
sigma = sigma / (n - length(k))
sigma_inverse = solve(sigma)
```

```r
calc_accuracy = function(actual, predicted) {
  return(mean(actual == predicted))
}

predict_lda = function(x) {
  si_k = rep(0, 10)
    for (k in 0: 9) {
      si_k[k + 1] = log(pi[k + 1, 2]) - (0.5 *(t(x - mu[k + 1, -1]) %*%
                                           sigma_inverse %*% (x - mu[k + 1, -1])))
    }
  return(which.max(si_k) - 1)
}
final = apply(test[, -1], 1, predict_lda)
accuracy_lda = calc_accuracy(actual = test[, 1], predicted = final)
```

The accuracy 0.877 and the confusion table is given below.

```r
kable(table(test[, 1], final) , digits = 2, row.names = TRUE, align = "c") %>%
  kable_styling("striped", full_width = FALSE) %>%
  add_header_above(c("Confusion Matrix for LDA" = 11)) %>%
  column_spec(column = 1, bold = TRUE)
```

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

b. [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

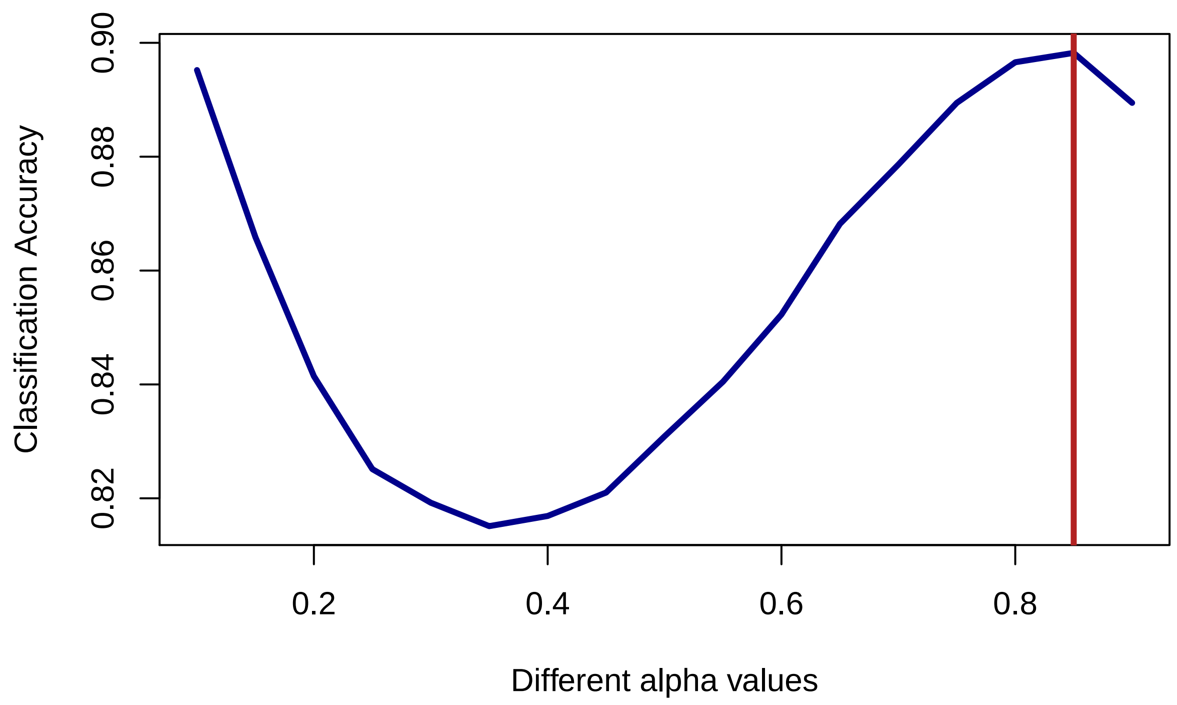
$$\widehat{\Sigma}_k(\alpha) = \alpha\widehat{\Sigma}_k + (1-\alpha)\widehat{\Sigma}$$

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
# Function to predict the qda
predict_qda = function(x) {
  si_k = rep(0, 10)
    for (k in 0: 9) {
      si_k[k + 1] = log(pi[k + 1, 2]) -
        (0.5 *(t(x - mu[k + 1, -1]) %*% sigma_k_for_each_alpha_inverse[[k + 1]] %*%
                 (x - mu[k + 1, -1]))) -
        (0.5 * sigma_k_for_each_alpha_log_det[[k + 1]])
    }
  return(which.max(si_k) - 1)
}

k = seq(0.1, 0.9, by = 0.05)
accuracy = rep(0, length(k))
for(i in 1:length(k)) {
  alpha = k[i]
  sigma_k_for_each_alpha = lapply(sigma_k, function(x) (alpha * x) + (1 - alpha) * sigma)
  sigma_k_for_each_alpha_inverse = lapply(sigma_k_for_each_alpha, function(x) solve(x))
  sigma_k_for_each_alpha_log_det = lapply(sigma_k_for_each_alpha,
                                          function(x) determinant(x, logarithm = TRUE)$modulus)
  predicted_qda_value = apply(test[, -1], 1, predict_qda)
  accuracy[i] = calc_accuracy(test[, 1], predicted_qda_value)
}
```

```r
best_alpha_qda = k[which.max(accuracy)]
plot(k, accuracy, type = "l",
xlab = "Different alpha values", ylab = "Classification Accuracy",
col = "darkblue",
lwd = 3)
abline(v = best_alpha_qda, col = "firebrick", lwd = 3)
```



The best tuning parameter is 0.85. For the best tuning parameter, the classification accuracy is 0.898. The accuracy of the QDA model is more than the LDA model - although by a very small amount. Ideally, this should have been the case as LDA assumes a very simple assumption that each class follows the same covarince structure - which is a very naive assumption to make in real data.

On the other hand, QDA provides a more flexible covariance structure for each class and hence a better accuracy for the best tuning parameter.

Having said that, in our case we can see the graph above - which shows us clearly that in our case the accuracy of the LDA is comparable to the QDA primarily because the covariance structure is very similiar for each class as the hand-written digits are all written in the middle of the picture. Hence, we see such close accuracy for both LDA and QDA.

Also, the confusion matrix is given below.

```r
# Confusion matrix for the QDA
kable(table(test[, 1], predicted_qda_value) , digits = 2, row.names = TRUE, align = "c") %>%
  kable_styling("striped", full_width = FALSE) %>%
  add_header_above(c("Confusion Matrix for QDA" = 11)) %>%
  column_spec(column = 1, bold = TRUE)
```

Confusion Matrix for QDA										
	0	1	2	3	4	5	6	7	8	9
0	1188	5	0	0	0	0	0	0	1	0
1	0	1001	2	0	1	0	0	0	0	1
2	33	3	675	0	9	1	1	0	4	5
3	92	1	36	423	5	69	1	1	24	6
4	15	4	6	0	602	1	0	2	3	19
5	45	0	6	14	8	458	8	0	12	5
6	54	12	7	0	3	4	578	0	4	2
7	12	8	6	0	23	2	0	538	1	55
8	29	9	9	1	12	13	2	1	452	14
9	9	4	0	0	47	3	0	10	1	570

c. [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.

```r
# Step1: Perform PCA and take the top 10 components and create a
pca_fit = prcomp(x = train[, -1], center = TRUE)
train_set_pca = cbind(train[, 1], pca_fit$x[, 1:10])
# Step2: Model the density based on PC's and NB
list_of_df = split(x = as.data.frame(train_set_pca), f = as.factor(train_set_pca[, 1]))
f_k_j = lapply(list_of_df,
               function(x) apply(x[, 2:ncol(x)], MARGIN = 2,
                                 function(y) c(mean(y), sd(y))) )

# Step 3: Prediction
test_set_pca = cbind(test[, 1], test[, -1] %*% pca_fit$rotation[, 1:10])
f_k = function(x, y) {
  mu = x[1, ]
  sigma = x[2, ]
  val = prod(dnorm(y, mean = mu, sd = sigma))
  return(val)
}
max_pi_k_f_k = function(one_row){
 return(which.max(sapply(f_k_j, f_k, y = one_row, simplify = TRUE) * pi[, 2]) - 1)
}
nb_predicted = apply(test_set_pca[, -1], MARGIN = 1, max_pi_k_f_k)
```

```r
nb_accuracy = calc_accuracy(actual = test_set_pca[, 1], predicted = nb_predicted)
```

The accuracy from the Naive Bayes model is less than compared to the LDA & QDA model. The accuracy is 0.693.

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.

```
# Only digit 2 and 4 are taken for train and test data
train_set_logistic = train[train[, 1] %in% c(2, 4), ]
test_set_logistic = test[test[, 1] %in%  c(2, 4), ]
# Use the 10-fold-CV to find the optimal lambda
penalized_logistic = cv.glmnet(x = train_set_logistic[, -1],
                               y = train_set_logistic[, 1],
                               alpha = 0,
                               family = 'binomial')
predicted_logistic = predict(penalized_logistic, test_set_logistic[, -1], type = "class")
accuracy_logistic = calc_accuracy(actual = test_set_logistic[, 1],
                                  predicted = as.numeric(predicted_logistic))
confusion_matirx= confusionMatrix(data = as.factor(predicted_logistic),
                                  reference = as.factor(as.character(test_set_logistic[, 1])))$table
```

The accuracy for the logistic regression with ridge penalty is 0.977. The confusion matrix is given below.

```
kable(confusion_matirx , digits = 2, row.names = TRUE, align = "c") %>%
  kable_styling("striped", full_width = FALSE) %>%
  add_header_above(c("Confusion Matrix for Penalized Logistic Regression" = 3)) %>%
  column_spec(column = 1, bold = TRUE)
```

Confusion Matrix for Penalized Logistic Regression		
	2	4
2	713	14
4	18	638

The most intuitive way to find out the right pixels which are importance in differentiating the two kind of hand-written digits is to only take out the pixels where the pixel values are different. Another way of saying that is to pick only the most important variables from the data set.

Hence, the statistical way to achieve that is by using a penalized logistic regression with lasso penalty and subset only those vairables which have non-zero coefficients in it.

```
penalized_logistic = cv.glmnet(x = train_set_logistic[, -1],
                               y = train_set_logistic[, 1],
                               alpha = 1,
                               family = 'binomial')
myCoefs = coef(penalized_logistic, penalized_logistic$lambda.1se)
#feature names: intercept included
regions_of_importance = as.data.frame(myCoefs@Dimnames[[1]][which(myCoefs != 0 ) ])
colnames(regions_of_importance) = c("Most important pixel regions")
kable(regions_of_importance , digits = 2, row.names = FALSE, align = "c") %>%
  kable_styling("striped", full_width = FALSE) %>%
  column_spec(column = 1, bold = TRUE)
```

Most important pixel regions
(Intercept)
V8
V11
V23
V24
V28
V29
V81
V100
V101
V105
V107
V114
V115
V116
V117
V121
V123
V129
V138
V154
V161
V177
V178
V181
V191
V195
V197
V207
V212
V218
V221
V229
V230
V250