Homework8

Question 1

a.

```r
distance=function(x,y){
  sqrt(sum((x-y)^2))
}

within_cluster = function(data, Ci, K, mk){
  total_distance = 0
  for (i in 1:K) {
    within = sum(apply(
      data[Ci == i, ],
      MARGIN = 2,
      FUN = distance,
      y = mk[[i]]
    ))
    total_distance = total_distance + within
    total_distance = total_distance/2
  }
  return(total_distance)
}


compare = function(d, data, K){
  #d is a matrix, with K cols n rows
  # output is a vector of length n, indicating the cluster
  index = matrix(nrow = nrow(data), ncol=K)
  shortcut = apply(d, MARGIN=1, FUN=min)

  for (k in 1:K){
    index[,k] = as.vector(shortcut==d[,k])*k
  }

  Ci_new = apply(index, MARGIN = 1, FUN=sum)
  return(Ci_new)
}


main = function(data, K) {
  # initiate everything
  mk = list()
  Ci = sample(1:K, replace = TRUE, size = nrow(data))
  Ci_new = rep(0, length.out = nrow(data))
  switch = TRUE

  while (switch) {

    # C fixed , iterate mk
    for (k in 1:K) {
      mk[[k]] = apply(data[Ci == k,], MARGIN = 2, FUN = mean)
    }

    # mk updated, find new C
    # use compare() function to update new Ci vector of length n
    d = matrix(nrow = nrow(data), ncol = K)
    for (k in 1:K) {
      d[, k] = apply(data,
                     FUN = distance,
                     MARGIN = 1,
                     y = mk[[k]])
    }
    Ci_new = compare(d, data=data, K=K)
    if (sum(Ci_new != Ci) > 0) {
      Ci = Ci_new
      switch = TRUE
    }
    else{
      switch = FALSE
    }
  }

  final_distance = within_cluster(data=data, Ci=Ci, K=K,
                  mk=mk)
  return(list(Ci, final_distance))
}


#within_cluster(data=data, Ci=result[[1]], K=K,
#                mk=result[[2]])

compose_main_within = function(data, K){
  result = main(data=data, K=K)
  distance_value = within_cluster(data=data, Ci=result[[1]], K=K,
                  mk=result[[2]])
  return(distance_value)
}


#replicate for 10 times and pick the best

main_replicate = function(data, K, n) {
  d = c()
  ci_list = list()
  for (i in 1:n) {
    result = main(data = data, K = K)
    ci_list[[i]] = result[[1]]
    d = append(d, unlist(result[[2]]))
  }

  result_ci = ci_list[which(d == min(d))[1]]
  result_d = d[d==min(d)]

  return(list(result_ci,result_d))
}
```

the `replicate_main` function would replicate the `main` function n times and pick the best result.

b. Compare your results with a built-in kmeans algorithm, e.g., kmeans(), on the iris data. Make sure that you

understand the iris data. For both your code and the built-in function, use k=3. Try 1 and 20 random starts and compare the results with different seeds. Do you observe any difference? What is the cause of the difference?

```
set.seed(1)
iris.kmean <- kmeans(iris[, -5], centers = 3, nstart = 1)
set.seed(20)
iris.kmean_20 <- kmeans(iris[, -5], centers = 3, nstart = 20)

iris.kmean$tot.withinss
```

```
## [1] 78.85144
```

```
iris.kmean_20$tot.withinss
```

```
## [1] 78.85144
```

```
iris.kmean_20$cluster
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [43] 1 1 1 1 1 1 1 1 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2
##  [85] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 3 3 3 3 2 3 3 3 3 3 3 2 2 3 3 3 3 2 3 2 3 2 3 3
## [127] 2 2 3 3 3 3 3 2 3 3 3 3 2 3 3 3 2 3 3 3 2 3 3 2
```

```
iris.kmean$cluster
```

```
##   [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [43] 3 3 3 3 3 3 3 3 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1
##  [85] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 2 2 1 2 2 2 2 2 2 1 1 2 2 2 2 1 2 1 2 1 2 2
## [127] 1 1 2 2 2 2 2 1 2 2 2 2 1 2 2 2 1 2 2 2 1 2 2 1
```

```
data=as.matrix(iris[,-5])
colnames(data) <- NULL
```

```r
set.seed(2)
result = main_replicate(data=data, K=3, n=1)

set.seed(1608)
result_20 = main_replicate(data=data, K=3, n=20)
result[[1]]
```

```
## [[1]]
##   [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [43] 2 2 2 2 2 2 2 2 3 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1
##  [85] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 3 3 3 3 1 3 3 3 3 3 3 1 1 3 3 3 3 1 3 1 3 1 3 3
## [127] 1 1 3 3 3 3 3 1 3 3 3 3 1 3 3 3 1 3 3 3 1 3 3 1
```

```r
result[[2]]
```

```
## [1] 61.36439
```

```r
result_20[[1]]
```

```
## [[1]]
##   [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [43] 2 2 2 2 2 2 2 2 3 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1
##  [85] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 3 3 3 3 1 3 3 3 3 3 3 1 1 3 3 3 3 1 3 1 3 1 3 3
## [127] 1 1 3 3 3 3 3 1 3 3 3 3 1 3 3 3 1 3 3 3 1 3 3 1
```

```r
result_20[[2]]
```

```
## [1] 61.36439 61.36439 61.36439
```

The difference between 1 and 20 trials are very minimal in this case. In fact, most of the results are the same. This is probably because the data has low dimensionality and the number of observations is only 150, hence it's likely that the convergence stays very consistant.

One difference between my function and the built-in `kmeans` function is that the total within-group variance of my function is smaller. This may have something to do with the built-in correction algorithm of the `kmeans` function to reduce the impact of outlier and so on. Another big difference is that my function is very sensitive to random seed when the number of trials gets big. There is a chance that my kmeans function doesn't converge and when the number of trials gets big(such as 20), the probability that the function spits an warning get very high.

Question 2

```r
library(ElemStatLearn)
library(plyr)
set.seed(3)

data("zip.train")
data("zip.test")
```

```r
set.seed(2)
K = 3
index_train = zip.train[, 1] == 1 | zip.train[, 1] == 4 | zip.train[, 1] == 8
train = zip.train[index_train, -1]
train_w_label = zip.train[index_train, ]

index_test = zip.test[, 1] == 1 | zip.test[, 1] == 4 | zip.test[, 1] == 8
test = zip.test[index_test, -1]
test_w_label = zip.test[index_test, ]
rm(zip.train)
rm(zip.test)


result = main_replicate(data=train, K=3, n=10)
head(unlist(result[[1]]))
```

```
## [1] 3 1 1 1 1 2
```

```r
print(result[[2]])
```

```
## [1] 4742.306
```

The head of the index function value is as above and the within cluster distance is 4742.306.

a.

1.Given your clustering results, can you propose a method to assign class labels to each of your clusters?

```r
determin_label = function(result, cover) {
  ci = unlist(result[[1]])
  vote = as.factor(train_w_label[ci == cover, 1])
  vote_freq = count(vote)

  label = vote_freq[vote_freq$freq == max(vote_freq$freq), ]$x
  return(label)
}

determin_label(result=result, cover=3)
```

```
## [1] 4
## Levels: 4 8
```

My method is simply voting within the same cluster. The 'cover' input is the cluster name, the the output gives the estimated label. In this example I estimated that the third cluster has label 4.

2. With your assigned cluster labels, how to predict labels on the zip.test data?

```
K=3
x_new = test[400,]
data = train
#result

predict_label = function(x_new, result, data, K) {
  ci = unlist(result[[1]])
  mk = list()
  for (k in 1:K) {
    mk[[k]] = apply(data[ci == k, ], MARGIN = 2, FUN = mean)
  }
  dis = sapply(mk,  FUN = distance, y = x_new)
  cover = which(dis == min(dis))
  label = determin_label(result = result, cover = cover)
  return(as.numeric(as.character(label)))
}

#predict_label(x_new=x_new, data=data, result=result,K=3)

y_hat_test = apply(test, MARGIN = 1, FUN = predict_label, data=train, result=result, K=3)
head(y_hat_test)
```

```
## [1] 4 1 4 8 4 1
```

The label of test data is predicted by which cluster mean is closest to each observation, then this observation is assigned to this cluster, whose label is assigned in the previous question. Here the `y_hat_test` is the predicted label of test data.

3.What is the classification error based on your model?

```
sum(test_w_label[,1]!=y_hat_test)/nrow(test_w_label)
```

```
## [1] 0.05555556
```

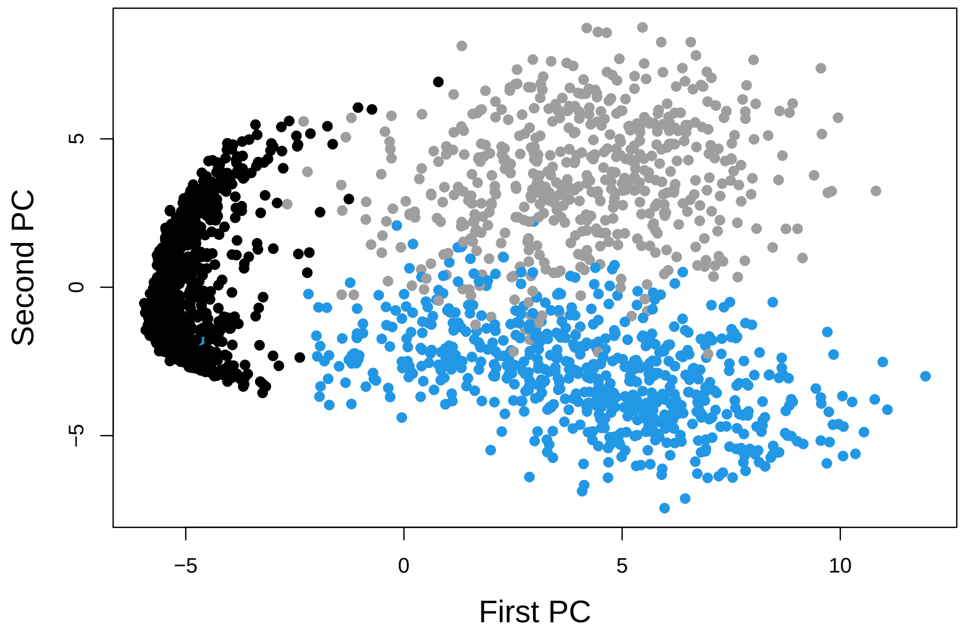
the classification error is about 0.05555556.

b.

1.Process your data using PCA. Plot the data on a two-dimensional plot using the first two PC's. Mark the data points with different colors to represent their digits

```
pcafit = princomp(train)

plot(
  pcafit$scores[, 1],
  pcafit$scores[, 2],
  xlab = "First PC",
  ylab = "Second PC",
  pch = 19,
  cex.lab = 1.5,
  col = train_w_label[, 1]
)
```



2.Based on the first two PCs, redo the k-means algorithm. Again, assign each cluster a label of the digit, and predict the label on the testing data. You need to do this prediction properly, meaning that your observed testing data is still the full data matrix, and you should utilize the information of PCA from the training data to construct new features of the testing data.

```
feature = pcafit$score[,1:2]
test_feature = test %*% pcafit$loadings[,1:2]

result = main_replicate(data=feature, K=3, n=10)

y_hat_test = apply(test_feature, MARGIN = 1, FUN = predict_label, data=feature, result=result, K=3)

head(y_hat_test)
```

```
## [1] 1 1 8 8 1 1
```

The `y_hat_test` is the new predicted labels of the compressed matrix.

3. Compare your PCA results with the results obtained by using the entire dataset. How much classification accuracy is sacrificed by using just two dimensions?

```
sum(test_w_label[,1]!=y_hat_test)/nrow(test_feature)
```

```
## [1] 0.3190476
```

the error with compressed data is 0.3206349. So about 26% of the accuracy is compromised during the process.

4.Comment on the potential strength and drawback of this dimension reduction approach compared with the raw data approach.

The strength of this dimension reduction approach is that it significantly reduce computaional and memory burden, and also if the raw data is highly correlated then the dimension reduction method won't lose too much accuracy. However the drawback is that when the data has low correlation then as a result the error may raise by a huge amount. And also it's not clear how many principle components to choose to make a better trade off.