K-Means R Implementation and Analysis

Christopher Norman

1 K-Means Algorithm

In this project we will create and analyse a K-Means clustering function in R.
Clustering tries to assign objects to a set of groups such that the objects within each of the groups are more similar than the objects outside of the group. These objects are usually unlabeled hence clustering is an "unsupervised learning method". K-Means algorithm is one such way to allocate data points into K clusters. It works by trying to minimising the within cluster variance and maximise the between cluster variance. The steps of the algorithm are stated in Algorithm 1. This algorithm gives a group allocation of each data point and the centres of each of the groups.

Algorithm 1: K-Means Clustering Algorithm

---

1. Input data matrix $\boldsymbol{X}$ with rows corresponding to each data point and choose number of clusters K and randomly assign each data point $\boldsymbol{x}_i$ in $\boldsymbol{X}$ to each of these clusters. Each data point has an index i in $\{1, ..., N\}$ where N is the number of data points and each index will map to a cluster number in $\{1, ..., K\}$.

2. Calculate estimated cluster centres $\hat{\boldsymbol{\mu}} = (\hat{\boldsymbol{\mu}}_1, ..., \hat{\boldsymbol{\mu}}_K)$ by averaging the data points within each cluster:

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{n_k} \sum_{i \in G_k} \boldsymbol{x}_i$$

3. Update data point allocation to the nearest cluster (closest by Euclidean distance).

4. Repeat steps 2 and 3 until convergence (cluster assignment does not change) or until a specified number of updates have been performed.

1.1 Impact of starting conditions

The optimal starting condition for the cluster centres is clearly the population group means for the K clusters $\boldsymbol{\mu}$, however, as we are trying to estimate this value it is not possible.
In the worst case the starting centres lead to having one or more clusters containing zero data points. Bad starting centres are more frequent when the initial assignment of data points $\boldsymbol{x}_i$ are random as the centres therefore are very close global mean of the data and all centres are very similar. Sub optimal initial clustering centres will also increase the number of iterations needed until the clustering converges. Another method of initialising cluster centres is to randomly set them equal a single existing data point. This idea is used in the "K-means++" algorithm [2] which chooses the first centre as a uniformly random data point then chooses the next centres up to K randomly with a probability distribution proportional to the distance between each point and its nearest centre. This requires a large amount of computation initially but decreases the chance of obtaining clusters with zero data points in and the K-means algorithm will converge faster.

2 K-Means Implementation in R

To implement a simple K-means algorithm in R we will create a reference class [1] called `KMeansAlgorithm` the code is shown in Listing 1. A reference class allows object oriented programming within R, this means we can create a single reusable class for the clustering much like the already existing R `kmeans()` function. In this class we use the assignment operator `"<<-"` to assign class variables that can be accessed using `KMeansAlgorithm$VarName` and we use `"="` assignment for local private variables within the class. The constructor of the object must contain at least 4 inputs, these being, `x` - the data matrix, `k` - the number of clusters, `iter.max` - the maximum number of iterations of the algorithm, `single_point_initialisation` - a boolean variable used to indicate how the cluster centres will be initialised.
`KMeansAlgorithm` contains a function called `start()` which acts as an initialisation function and if `single_point_initialisation = FALSE` it assigns random clusters to each data point index in the list `group_allocation` following Step 1 of Algorithm 1. If `single_point_initialisation = TRUE` the centres of the clusters will be set to single random data points at the start of the algorithm. `group_allocation[i]` can be used to return the current cluster number that the data point index i is assigned to. `groups[j]`, where $j \in \{1, ..., K\}$, returns a list of indexes that are currently assigned to the cluster j. It then recursively calls the function `update()` which handles Steps 2 and 3 in Algorithm 1, this iterates until the group allocation has not changed after calling `update()` or reaches the specified number of maximum iterations. The variable `iter` represents the total number of iterations performed before termination.
`update()` first calculates cluster centres then iterates through each of the n data points and updates the cluster assignment to the closest cluster. Finally the function `results()` is called which calculates the total within group sum of squares, between group sum of squares and total sum of squares of the final cluster assignment which will be used for analysis.

3 Results and Analysis

To compare our implementation with the inbuilt `kmeans()` function in R we will create two synthetic data-sets each containing 200 samples (50 from each of the 4 groups), the first containing strong clusters (lower variance) sampled from normal distributions (shown in Figure 1) and the second containing 4 groups sampled from normal distributions with 2 groups having very similar means compared to the other 2 and a higher variance (shown in Figure 2). For better visualisation we will use a multivariate normal distribution with two dimensions and diagonal covariance matrix $= \sigma^2 \boldsymbol{I}$ where $\sigma^2 = 50$ and 80 for data sets 1 and 2 respectively. We will also maintain the same value of $K = 4$ throughout the analysis. The data is generated using a seed value of 158. The code used for the analysis section is shown in Listing 2. In this code we set our K-means result (an object of the class we created before) to `km` and the inbuilt R result to `km.r` for each data set.

3.1 Data Set 1

Firstly running the our implementation on Data Set 1 gives an output shown in Table 3.1. We can see that the cluster allocation is exactly the same other than the label switching of the clusters, each cluster contains 50 samples. The number of iterations performed by our implementation (4) is double that of the R implementation (2), this means that the starting position of the centres is better optimised in the inbuilt function. During testing of our implementation when setting `seed = NULL` in a few cases the program produced an error due to a cluster having zero data points in it. Randomly assigning data points uniformly to each cluster leads to poor initial conditions and hence this error in some cases. This frequency of this error is amplified when the data has a much higher variance and overlapping points like that shown in Figure 2. Instead we can use a second method of initialising the cluster centres, in this method we first assign each of the K centres to a random data point then update the group assignment of

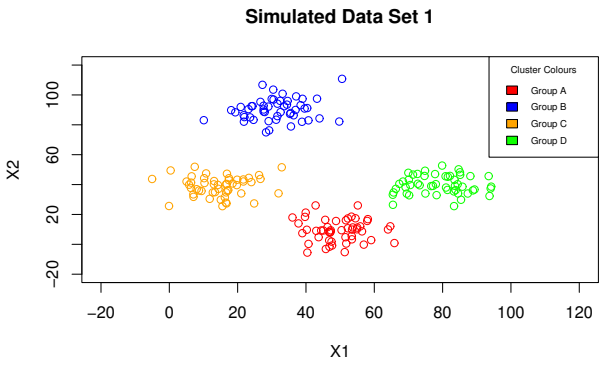

Figure 1: Plot of data set 1

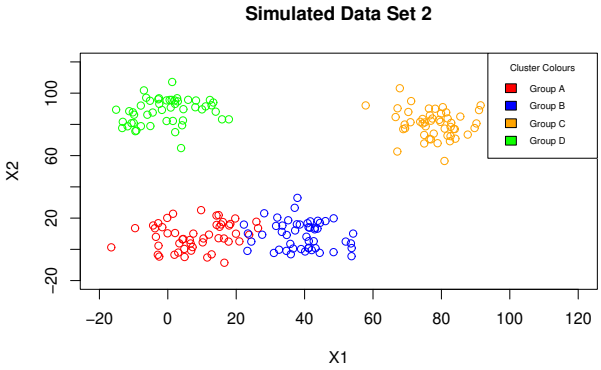

Figure 2: Plot of data set 2

each data point to the closest centre. In Table 3.1 the third column shows the results for this initialisation. We can see that it reduces the number of iterations by 1, in general it reduces the iterations needed and is less likely to produce empty clusters compared to random initial cluster allocation.

DATA SET 1	Our Implementation Random Initialisation	R Implementation	Our Implementation Single Point Centre
Iterations Performed	`km$iter = 4`	`km.r$iter = 2`	`km$iter = 3`
between_SS/total_SS	93.06326 %	93.06326 %	93.06326 %

Cluster Allocation Matrix	`  1  2  3  4`
	`a  0  0 50  0`
	`b 50  0  0  0`
	`c  0  0  0 50`
	`d  0 50  0  0`
		`  1  2  3  4`
		`a  0 50  0  0`
		`b  0  0  0 50`
		`c 50  0  0  0`
		`d  0  0 50  0`
			`  1  2  3  4`
			`a  0 50  0  0`
			`b  0  0  0 50`
			`c 50  0  0  0`
			`d  0  0 50  0`

Cluster Centres	`> km$cluster_means`
	`     [,1]     [,2]`
	`[1,] 31.23366 90.1885`
	`[2,] 78.97612 39.77029`
	`[3,] 49.66798 9.276727`
	`[4,] 14.50959 38.79861`
		`> km.r$centers`
		`      [,1]      [,2]`
		`1 14.50959 38.798611`
		`2 49.66798  9.276727`
		`3 78.97612 39.770286`
		`4 31.23366 90.188498`
			`> km$cluster_means`
			`     [,1]     [,2]`
			`1 31.23366 90.1885`
			`2 49.66798 9.27672`
			`3 14.50959 38.7986`
			`4 78.97612 39.7702`

3.2 Data Set 2

When running our implementation with the data shown in Figure 2 and random initial assignment an error is thrown due to a cluster containing zero elements. We use a `try-catch` statement to catch this error. Instead we will use the second method of centre initialisation where each centre is set to a single random data point. Figure 3 shows the result of our implementation and Figure 4 shows the result of the inbuilt R implementation. We can see that the inbuilt version is much better in this case as in our implementation it has split the top left cluster into two whereas Figure 4 is much closer to the original data in Figure 2. Repeating this with other `seed = NULL` sometimes results in a much better clustering. Clusters containing zero elements almost never appear due to the better cluster initialisation although it is not as good as the inbuilt R function.

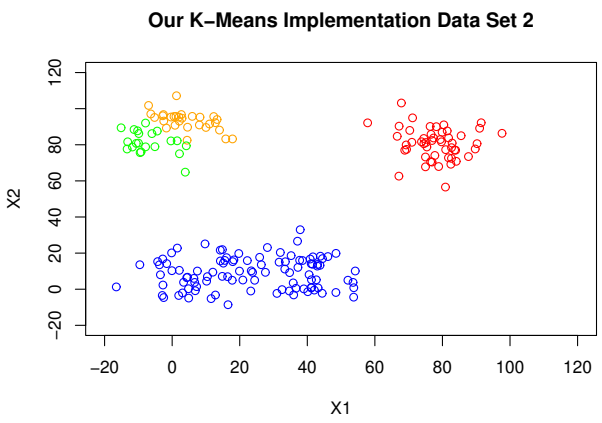

Figure 3: Plot of data set 2 results own implementation

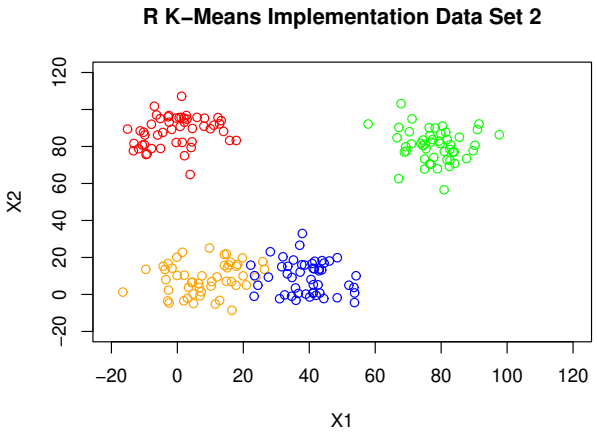

Figure 4: Plot of data set 2 results from inbuilt R k-means

3.3 Conclusion

To conclude we have found that clustering results are highly sensitive to the initial conditions. For large data-sets it is optimal to use a initialisation such as K-means++ and to run the clustering algorithm many times on the same data (hence slightly different starting conditions) to avoid the error introduced by this sensitivity. Label switching must be considered when evaluating a number of different clustering results.

References

[1] Hadley Wickham. Advanced r. http://adv-r.had.co.nz/R5.html, 2021. [Online; accessed 20-November-2021].

[2] Wikipedia contributors. k-means++ — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/K-means%2B%2B, 2021. [Online; accessed 1-December-2021].

A Code Appendix

```
KMeansAlgorithm = setRefClass("KMeansAlgorithm", fields = list(x = "matrix", k = "
    numeric", iter.max = "numeric", nstart = "numeric", groups = "list", group_
    allocation = "vector", cluster_means = "matrix", iter = "numeric", tot.withinss = "
    numeric", betweenss = "numeric", tot.ss = "numeric", single_point_initialisation = "
    logical"),
  methods = list(
      # Initialises and runs the K-Means algorithm
      start = function() {
        groups <<- vector(mode = "list", length = k)  #  Holds  the k current  groups
    andthe  respective  indexes  assigned  to them

        if(single_point_initialisation == TRUE){  # Random single data point centres
          number_variables = dim(x)[2] # Number of variables for each data point
          cluster_means <<- matrix(list(), nrow=k, ncol=number_variables)

          for(i in 1:k){
            cluster_means <<- x[sample(nrow(x),size=k,replace=FALSE),]
```

```
      }

      for (i in 1:(nrow(x))){  # Check each data point for it's closest cluster mean
        closest_group = NULL
        closest_distance = 10000000  # Arbitrary large value
        for (group_number in 1:k) {
          if(norm(x[i,] - unlist(cluster_means[group_number,]), type="2") < closest_
distance){
            closest_group = group_number
            closest_distance = norm(x[i,] - unlist(cluster_means[group_number,]),
type="2")
          }
        }
        group_allocation[i] <<- closest_group  # Update group assignment
      }

    } else {  # Randomly assign each data point to k groups
      group_allocation <<- sample(1:(k), nrow(x), replace=T)

      for(i in 1:(nrow(x))){  # Each data point index will be assigned to each group
        groups[[group_allocation[i]]] <<- c(groups[[group_allocation[i]]], i)
      }
    }

    iter <<- 0
    for(iteration in 1:iter.max){  # Iterate through algorithm iter.max times
      previous_group_allocation = group_allocation
      update()
      iter <<- iter + 1  # Increment iteration count by 1
      if(identical(previous_group_allocation, group_allocation)){
        break  # End loop when iteration doesn't change the group allocation
      }
    }
    results()  # Get final variation calculations
  },

  # Updates the group assignments
  update = function() {
    number_variables = dim(x)[2] # Number of variables for each data point

    groups <<- vector(mode = "list", length = k)
    for(i in 1:(nrow(x))){  # Update sets of clusters with the ID's that are
currently assigned to that cluster
      groups[[group_allocation[i]]] <<- c(groups[[group_allocation[i]]], i)
    }

    # First estimate group means
    cluster_means <<- matrix(list(), nrow=k, ncol=number_variables)
    for(i in 1:k){  # Append the values of all the points currently in each cluster
      current_group_data = c()
      for(j in groups[[i]]){
        current_group_data = rbind(current_group_data, x[j,])
      }
      # Group mean estimators = (1/number of samples currently in the group)*(sum of
 variables)
      cluster_means[i,] <<- (1/length(groups[[i]]))*colSums(current_group_data)
    }

    for (i in 1:(nrow(x))){  # Check each data point for it's closest cluster mean
      closest_group = NULL
      closest_distance = 10000000  # Arbitrary large value
```

```
          for (group_number in 1:k) {
            if(norm(x[i,] - unlist(cluster_means[group_number,]), type="2") < closest_
    distance){
              closest_group = group_number
              closest_distance = norm(x[i,] - unlist(cluster_means[group_number,]), type
    ="2")
            }
          }
          group_allocation[i] <<- closest_group  # Update group assignment
        }
      },

      results = function(){
        # Calculates within cluster and between cluster sum of squares
        # Total within group sum of squares
        tot.withinss <<- 0
        for(group_number in 1:k){
          for(i in groups[[group_number]]){
            tot.withinss <<- tot.withinss + as.numeric(t((x[i,] - unlist(cluster_means[
    group_number,])))%*%(x[i,] - unlist(cluster_means[group_number,])))
          }
        }
        betweenss <<- 0  # Between group sum of squares
        global_mean = (1/nrow(x))*colSums(x)
        for(group_number in 1:k){
          betweenss <<- betweenss + as.numeric(length(groups[[group_number]])%*%t(unlist
    (cluster_means[group_number,]) - global_mean)%*%(unlist(cluster_means[group_number
    ,]) - global_mean))
        }
        tot.ss <<- betweenss + tot.withinss
      }
      )
  )
```

Listing 1: Code for reference class `KMeansAlgorithm`

```
source("k-means/KMeansAlgorithm.R")  # This may need to be updated depending where you
    run this file from
library("mnormt") # R package for multivariate normal distribution

set.seed(158)  # Set seed for reproducible results
# Data 1 4 multivariate normal distributions with equal variance
mu1 = c(50, 10)
mu2 = c(30, 90)
mu3 = c(15, 40)
mu4 = c(80, 40)
sigma = 50*diag(2)

group1 = rmnorm(n = 50, mu1, sigma)
group2 = rmnorm(n = 50, mu2, sigma)
group3 = rmnorm(n = 50, mu3, sigma)
group4 = rmnorm(n = 50, mu4, sigma)

plot(group1, xlim=range(-20,120), ylim=range(-20,120), main="Simulated Data Set 1", xlab
    ="X1", ylab="X2", col="red")
points(group2, col="blue")
points(group3, col="orange")
points(group4, col="green")
legend("topright", title = "Cluster Colours",legend= c("Group A", "Group B", "Group C",
    "Group D")
       ,fill = c("red", "blue", "orange", "green"), cex = 0.6)
data = rbind(group1, group2, group3, group4)
```

```
# Create K-Means object from implementation in KMeansAlgorithm.R
km = KMeansAlgorithm(x = data, k = 4, iter.max = 20, nstart = 3,single_point_
    initialisation = FALSE)
km$start()  # Initalise algorithm

cluster_allocation = rep(c("a","b","c","d"),each=50)

print("OUR IMPLEMENTAION DATA 1")
cat("between_SS / total_SS = ", (km$betweenss/(km$tot.withinss + km$betweenss)*100), "%"
    )
km$cluster_means
print("Cluster assignment matrix:")
table(cluster_allocation, km$group_allocation)
cat("Number of iterations:", km$iter)


print("R IMPLEMENTAION DATA 1")
km.r <- kmeans(data, 4, iter.max = 20)  # K-Means R package to compare
cat("between_SS / total_SS = ", (km.r$betweenss/(km.r$tot.withinss + km.r$betweenss)*
    100), "%")
km.r$centers
print("Cluster assignment matrix:")
table(cluster_allocation, km.r$cluster)
cat("Number of iterations:", km.r$iter)


print("OUR IMPLEMENTAION DATA 1 SINGLE POINT CENTRES")
km = KMeansAlgorithm(x = data, k = 4, iter.max = 20, nstart = 3,single_point_
    initialisation = TRUE)
km$start()  # Initalise algorithm

cat("between_SS / total_SS = ", (km$betweenss/(km$tot.withinss + km$betweenss)*100), "%"
    )
km$cluster_means
print("Cluster assignment matrix:")
table(cluster_allocation, km$group_allocation)
cat("Number of iterations:", km$iter)

# Data 2
mu1 = c(10, 10)
mu2 = c(40, 10)
mu3 = c(80, 80)
mu4 = c(0, 90)
sigma = 80*diag(2)

group1 = rmnorm(n = 50, mu1, sigma)
group2 = rmnorm(n = 50, mu2, sigma)
group3 = rmnorm(n = 50, mu3, sigma)
group4 = rmnorm(n = 50, mu4, sigma)

plot(group1, xlim=range(-20,120), ylim=range(-20,120), main="Simulated Data Set 2", xlab
    ="X1", ylab="X2", col="red")
points(group2, col="blue")
points(group3, col="orange")
points(group4, col="green")
legend("topright", title = "Cluster Colours",legend= c("Group A", "Group B", "Group C",
    "Group D")
       ,fill = c("red", "blue", "orange", "green"), cex = 0.6)
data_2 = rbind(group1, group2, group3, group4)
```

```
try({  # Catch the error thrown when a cluster is empty
# Create K-Means object from implementation in KMeansAlgorithm.R
km = KMeansAlgorithm(x = data_2, k = 4, iter.max = 20, nstart = 3, single_point_
    initialisation = FALSE)
km$start()  # Initalise algorithm
}, {print("Error, a cluster is empty")}, silent = FALSE)

km = KMeansAlgorithm(x = data_2, k = 4, iter.max = 20, nstart = 3, single_point_
    initialisation = TRUE)
km$start()  # Initalise algorithm

print("OUR IMPLEMENTAION DATA 2")
cat("between_SS / total_SS = ", (km$betweenss/(km$tot.ss)*100), "%")
km$cluster_means
print("Cluster assignment matrix:")
table(cluster_allocation, km$group_allocation)
cat("Number of iterations:", km$iter)


print("R IMPLEMENTAION DATA 2")
km.r <- kmeans(data, 4, iter.max = 20)  # K-Means R package to compare
cat("between_SS / total_SS = ", (km.r$betweenss/(km.r$totss)*100), "%")
km.r$centers
print("Cluster assignment matrix:")
table(cluster_allocation, km.r$cluster)
cat("Number of iterations:", km.r$iter)

colour_list = c("red", "blue", "orange", "green")
# PLOT OUR RESULT
plot(c(), xlim=range(-20,120), ylim=range(-20,120), main="Our K-Means Implementation
    Data Set 2", xlab="X1", ylab="X2", col="red")
for (index in 1:200){
  colour = colour_list[km$group_allocation[index]]
  points(data_2[index,1], data_2[index, 2], col = colour)
}

# PLOT R RESULT
plot(c(), xlim=range(-20,120), ylim=range(-20,120), main="R K-Means Implementation Data
    Set 2", xlab="X1", ylab="X2", col="red")
for (index in 1:200){
  colour = colour_list[km.r$cluster[index]]
  points(data_2[index,1], data_2[index, 2], col = colour)
}
```

Listing 2: Code for analysis: `KMeansAnalysis.R`