k-Means Clustering Algorithm

Goutham Swaminathan

#Introduction The *k*-Means Clustering Algorithm is an unsupervised statistical learning method to classify data points into clusters. The term *unsupervised* refers to the fact that the algorithm is not given any verification of whether its result is correct or not.

#Explanation In order to implement this algorithm and apply it to a dataset, we will need to do the following: 1. Simulate data in the form of a 2-dimensional scatter plot. 2. Choose how many clusters we want to have (k). 3. Choose initial coordinates for the k centroids. 4. Create a distance function that calculates the Euclidean distance between a point and a centroid. 5. Apply the distance function to every point and every centroid and record the results. 6. Assign each point to a cluster based on the minimum distance between the point and a centroid. 7. Reassign each centroid to the mean position of its respective cluster. 8. Rerun the algorithm until mean distance between points and centroids is minimized.

```
numPoints = 200
numClusters.data = 5
numClusters = 5
numIterations = 10
spread = 4
meanValue = 50
```

We first initialize variables for the number of points, number of inherent clusters in the data, number of desired clusters, number of iterations OF *k*-means, and the spread of the data. Initializing these variables allows for the most customization of the algorithm when testing it with data.

We will then create a *distance()* function to calculate the distance between every point in an NX2 matrix and a centroid. We will first define the **dist** variable which will hold a matrix of distances, with each row containing the distances for a different centroid. Using the *apply()* function, we will set the **X** parameter to **matrix** and the **MARGIN** parameter to 1 (because we want to apply the function across every row, where each row represents the coordinates of one point). We will then create our own function withIN the *apply()* function, in which we set the **centroid** variable to the first two elements of the **centroid** vector. This is due to the fact that later in the program, we will use the 3rd column of the **centroids** matrix to hold the color of each centroid. In addition, this **centroid** variable is a vector rather than a matrix because in our *kmeans_simulation()* function, we will apply this function to every row of the **centroids** matrix. Afterwards, we will calculate the total square difference between the x and y values for the **centroid** and **x** variables. Finally, we will return the **dist** vector.

```
distance <- function(matrix, centroid){

  dist = apply(matrix, MARGIN = 1, FUN = function(x){
     centroid = centroid[1:2]
     sqrt(sum((centroid - x)^2))})

  return(dist)
}
```

Now, we can begin creating our *kmeans_simulation()* function, which will handle running our *k*-means simulations. We will first create 5 parameters: 1. **npoints** - the numbers of points in the dataset 2. **nclusters** - the number of desired clusters 3. **niterations** - the number of iterations for which we will run our simulations 4. **nclusters.data** - number of inherent clusters in our data 5. **spread** - the spread of our data 6. **meanValue** - the mean value of our data

```
library(ggplot2)
```

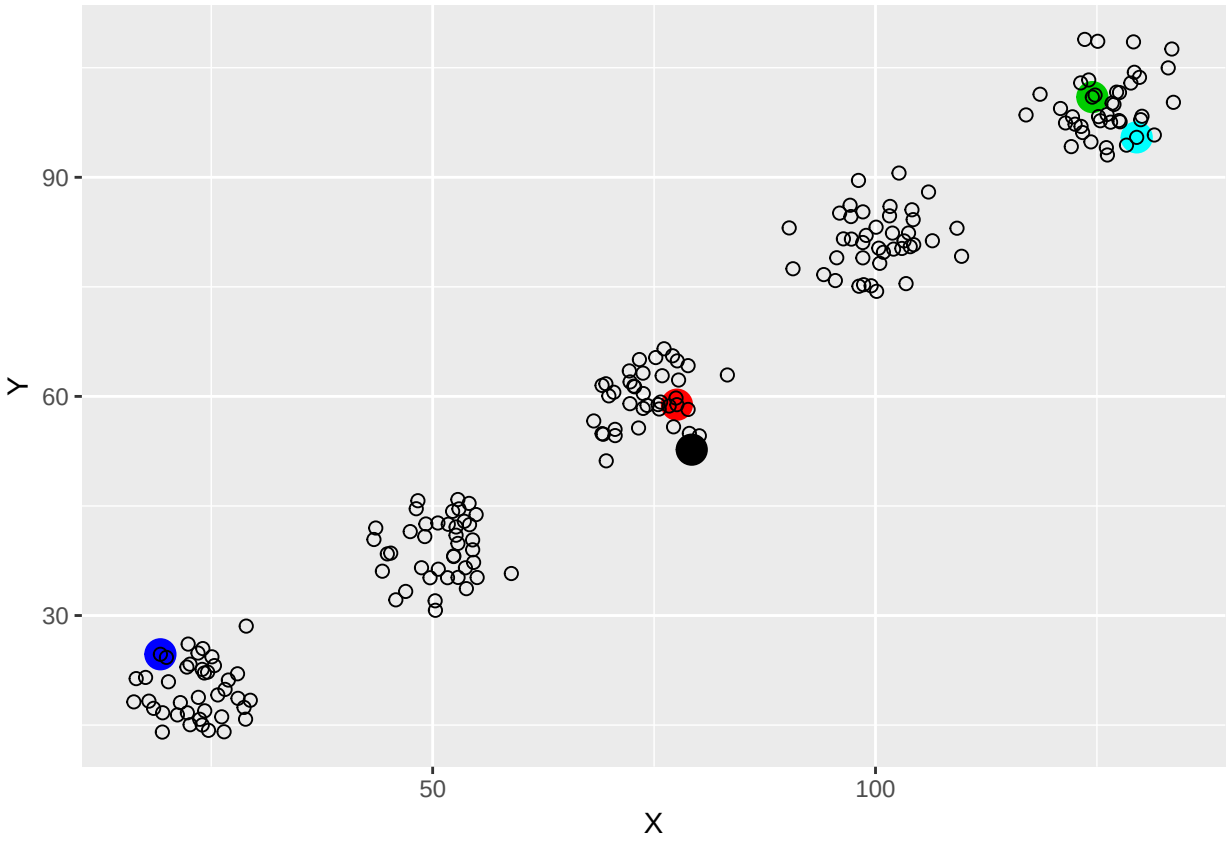
```r
kmeans_simulation = function(npoints, nclusters,
                             niterations, nclusters.data = 4, spread = 5, meanValue = 50){

  xvalues = vector() #initialize a vector of x values
  yvalues = vector() #initialize a vector of y values

  for(i in 1:nclusters.data){ # run following code for every inherent cluster
   ## generates random X and Y values
    xvalues = c(xvalues, rnorm(npoints/nclusters.data,
                               mean = (25*i), sd = spread)) #clustered x values
    yvalues = c(yvalues, rnorm(npoints/nclusters.data, # clustered y values
                               mean = (100*i)/nclusters.data, sd = spread))
    # xvalues = sample(1:100, npoints, replace = TRUE) # scattered x values
    # yvalues = sample(1:100, npoints, replace = TRUE) # scattered y values
  }

  # binds X and Y values into a matrix
  dataset = cbind(xvalues, yvalues)

  xy =  dataset[sample(1:npoints, nclusters),] # sets centroids to random pts from dataset
  color = 1:nrow(xy)
  centroids = cbind(xy, color) # binds color vector for coloring centroids

  centroids_init = centroids # creates initial centroids matrix for plotting
  cost = vector() # initializes cost vector

  for(j in 1:niterations){ # runs for specified number of iterations

    #uses apply() to calculate distances between centroids & datapts
    alldistances = apply(centroids, MARGIN = 1, distance, matrix = dataset)

    # assigns datapt to cluster based on min distance
    clusterAssignment = apply(alldistances, MARGIN = 1, which.min)

    # uses sapply() and sum() to track cost using total distances
    cost[j] = sum(sapply(1:nclusters,
                         function(x)
                           sum(distance(dataset[clusterAssignment == x,], centroids[x,]))
                         ))

    # uses sapply() to reassign centroid coordinates to mean of datapoint coordinates
    centroids[,1:2] = sapply(1:nclusters,
                             function(x) mean(dataset[clusterAssignment == x,1:2]))

  }

  # returns list of various values
  return (list(cost, clusterAssignment, centroids, dataset, centroids_init))
}
```

Now that we have written our function, we can implement it. We will first start by creating a **list** variable which will hold the *list* output of our call to the *kmeans_simulation()* function.

```
list = kmeans_simulation(
  numPoints, numClusters, numIterations, numClusters.data, spread, meanValue
  )
```

Using the *ggplot()* function, we can use *geom_point()* to first plot the initial clusters, which is the 5th element of **list**, as well as the data itself, which is the 4th element of **list**.

```
ggplot(data = NULL) +
  geom_point(size=2, shape="23") + xlab("X") + ylab("Y") +
  geom_point(aes(x=list[[5]][,1], y=list[[5]][,2]), colour=list[[5]][,3], size=5) +
  geom_point(aes(x=list[[4]][,1], y=list[[4]][,2]), colour="black", pch=21, size=2)
```

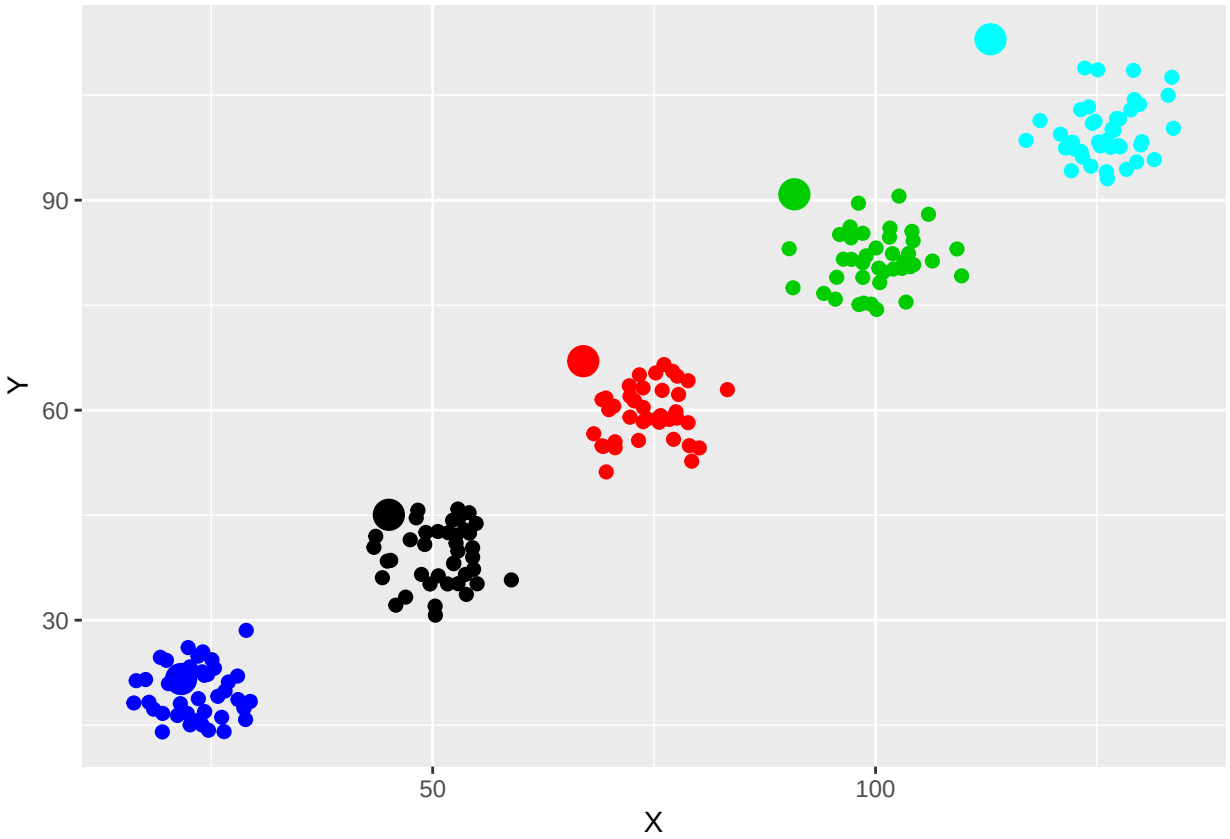


The *Sys.sleep()* function allows us to delay execution of a code sample by a specified period of time so that we can view a previous plot (in this case) before a new plot is generated. In this case, since we will use *Sys.sleep(2)*, the program will delay generating the next plot by 2 seconds.

```
Sys.sleep(2)
```

Now, we can plot the final clustering of the data. Using *ggplot()* and *geom_point()* again, we will plot the final centroids, which is the 3th element of **list**, as well as the original data, but we will specify the *color* parameter to be *list[[2]]*, which is the clusterAssignment vector. As a result, each data point will be color coded based on its cluster assignment.

```
ggplot(data = NULL) +
  geom_point(aes(x=list[[3]][,1], y=list[[3]][,2]), colour=list[[3]][,3], size=5) +
  geom_point(aes(x=list[[4]][,1], y=list[[4]][,2]), colour=list[[2]], size=2) +
  xlab("X") + ylab("Y")
```



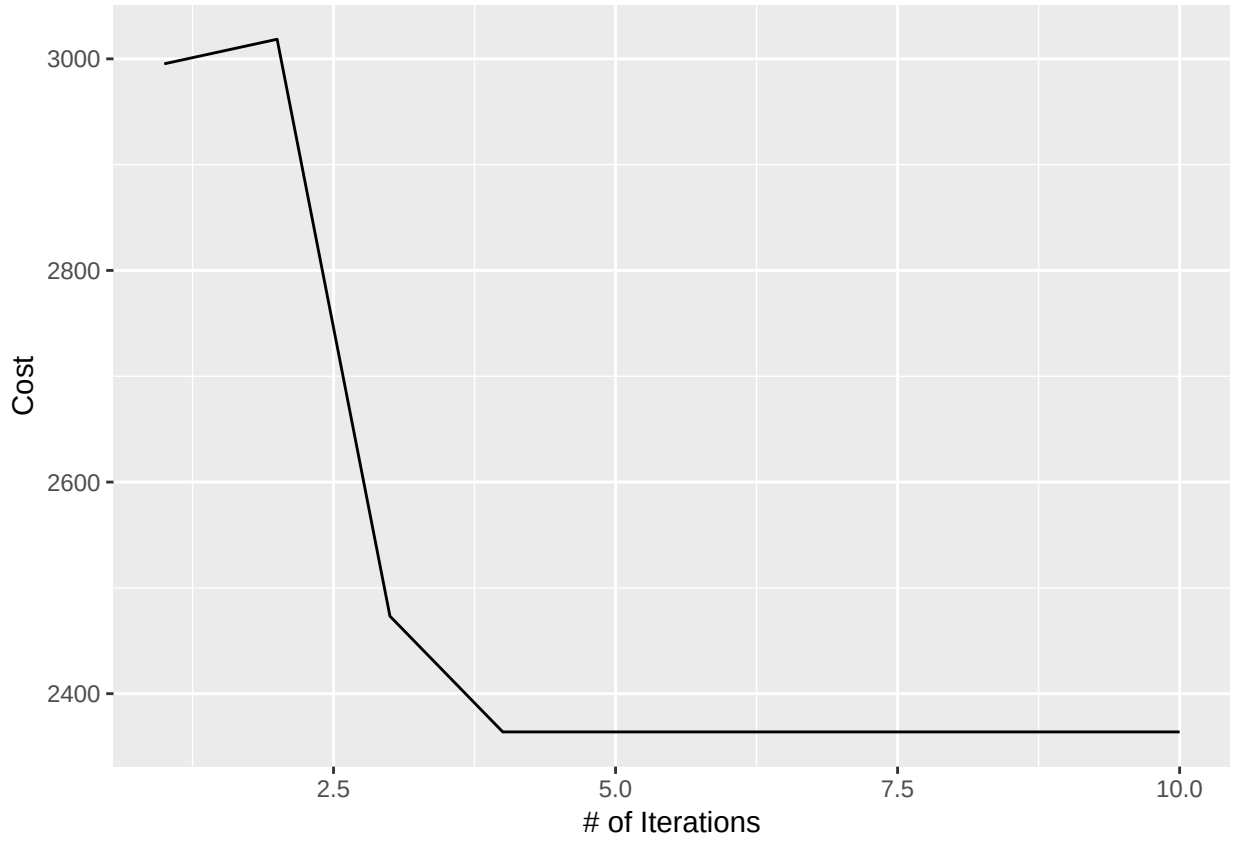
As we can see, using the clustered dataset that we created in the beginning of the *kmeans_simulation()* function to test the algorithm has proved that the algorithm can correctly identify the clusters in the data.

```
Sys.sleep(2)
```

Finally, we will plot the **cost** value with respect to the number of iterations and the number of clusters. This step is important because it allows us to identify the optimal number of clusters and iterations to include in our data based on the number of inherent clusters (*nclusters.data*).

For the "# of Iterations vs. Cost" graph, we will use the *ggplot()* and *geom_line()* functions and index our **list** variable to the 1st index, which contains our cost values.

```
ggplot(data = NULL) +
  geom_line(aes(x=1:numIterations, y=list[[1]])) +
  xlab("# of Iterations") + ylab("Cost")
```



As we can see, the graph seems to plateau at 8 iterations, meaning that 8 iterations would be the optimal number for this simulation.

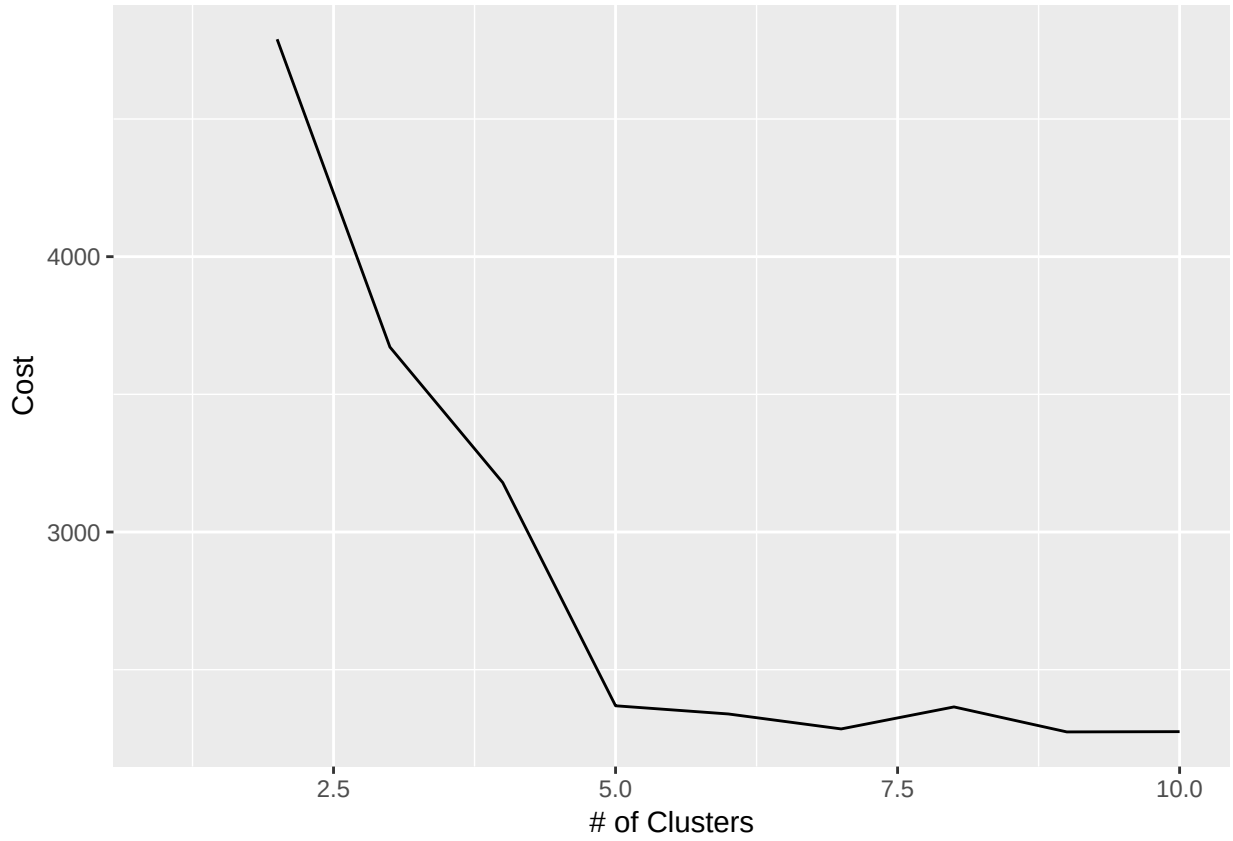
We will now plot "# of Clusters vs. Cost" by first creating a **costList** vector to store the cost values. We will then run a loop from 2:10 in which we call the *kmeans_simulation()* function however, rather than using the **numClusters** variable, we will use **i** instead. We will then index this function call to return the cost and assign it to the **i**th index of the **costList** vector.

```
costList = vector()

for(i in 2:10){
  costList[i] = kmeans_simulation(
    numPoints, i, numIterations, numClusters.data, spread)[[1]][numIterations]
}
```

We will now plot "# of Clusters vs. Cost" in the same way as before, using *ggplot()* and *geom_line()* and setting the **x** value from 1 to *length(costList)*, which represents the range of the number of clusters.

```
ggplot(data = NULL) +
  geom_line(aes(x = 1:length(costList), y = costList)) +
  xlab("# of Clusters") + ylab("Cost")
```

```
## Warning: Removed 1 rows containing missing values (geom_path).
```



As we can see, the graph looks like an elbow, with the "elbow joint" appearing at 6 clusters. This "elbow plot" shows that after more than 6 clusters, the cost does not decrease notably, so the optimal number of clusters for this data is 6.

#Conclusion Thank you for taking the time to read this explanation for the "K Means Clustering Algorithm". For more information about the algorithm, please visit this link. Link: https://www.naftaliharris.com/blog/visualizing-k-means-clustering/. This explanation and the included code is **100% my original work**. Please feel free to visit my GitHub page at http://github.com/goutham1220 where I will be posting more explanations as well as other statistics and data science-related resources. In addition, please feel free to visit my YouTube channels "GSDataScience" (http://bit.ly/gsdatascience), where I will be posting more data science and statistics-related videos, and "Gooth" (http://youtube.com/gooth), where I post more cinematic-style, slice-of-life videos.