Background/Motivation

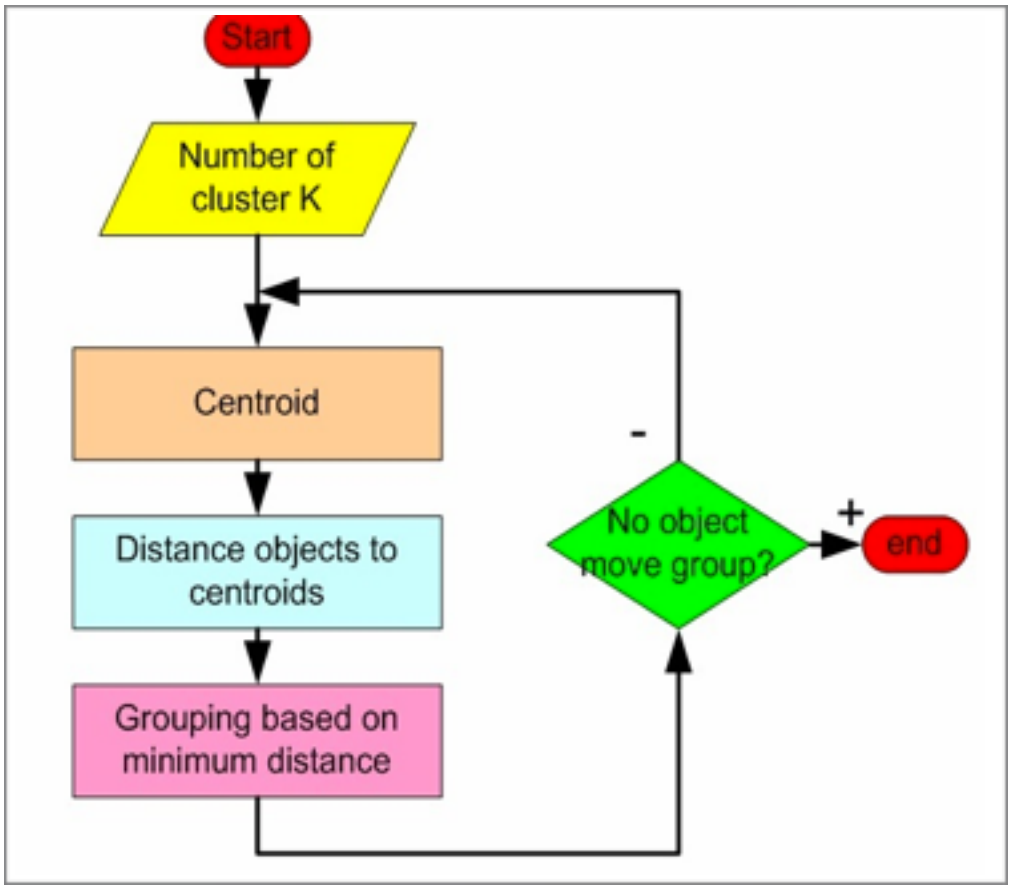
In machine learning and statistics, unsupervised learning is the practice of performing inference on data that are not structured in the typical sense of a response and predictor variables. Techniques in this domain are generally more reliant on visualization or some type of simplification of the data. Examples of unsupervised learning includes dimension reduction, classification and a wide variety of visualization.

Classification has a breadth of applications in very practical domains such as business, healthcare, technology and more. The idea is generally to group data together into similar groups using some sort of algorithmic method. This can be done heuristically, and doesn't need to include complex mathematics at all. For instance, consider a company segmenting their customers into different groups based on interests or past locations. This could be done entirely based on simple decision rules, i.e. how many times the person went to a yoga class, went shopping, went to the movies, etc.

The more interesting part of this domain from a statistical sense are the problems that require some sort of rigorous, algorithmic method of classification or clustering. There is certainly no shortage of clustering methods in both the literature and applied in practice. Some important examples are methods such as DBSCAN, Principal Components Analysis, HCS, k-means and many more.

Here we focus on a relatively simple, but nonetheless powerful and well-documented method of clustering: k-means. The objective of k-means is to separate data into k distinct groups, where membership of a group is based on the distance (using some distance metric, usually euclidian distance) to the group centroid (a multi-dimensional generalization of the mean). K-means was introduced in the 1960's and popularized with advancements in computing powers that began in the 80's.

The basic algorithm is actually quite simply and can be visualized as follows:



As it is such a quintessential method, standard libraries exist to implement it in most programming languages geared towards working with data. However, the best way to truly understand a machine learning algorithm is to implement it from scratch. In this way, it's implementation serves to clarify the algorithm and reveals pitfalls and advantages of using it.

Problem Formulation

The problem that is generally solved with clustering algorithms can be simply stated, but solved or made clearer by dozens of complex methods. Generally speaking, we have data that represent something, i.e. people, places, or some description of a physical or mechanical process, and we want to group similar observations together.

This can be accomplished in a multitude of ways: connectivity, density, distribution, proximity to a center point or vector, groups, graph, subspaces, etc, etc. In k-means the idea is quite simple: we want to group points into k groups based on their proximity (using a variety of potential distance metrics) to the centroid of each group, which must be first guessed and then iteratively computed.

Some challenges that arise when formulating a solution to this include, but are not limited to: how do we measure the proximity, i.e. which distance formula is most appropriate for the

situation? How do we best choose the number of groups, k? Too few groups could result in a meaningless analysis, and too many could segment things in a way that isn't helpful either. This will depend largely on the data we are working with as well as what we want to accomplish from the analysis, and how specific or general we wish for our inferences to be. Another problem that may arise could be the initial guess for the centroids of groups. Of course after we make an initial guess we also want to update group membership in a sensible way that minimizes the error from our initial cluster estimate.

Design & Architecture

In terms of solving the clustering problem by implementing the k-means algorithm, there were a few pitfalls in writing out the algorithm. The main problem encountered was through the initial selection of centroids. They were first selected using the first k data points from the data set, which was not ideal because it resulted in no points being assigned to a group. When this happened the new mean was updated as the point (0,0) and successively had no data points in its cluster. This resulted in having to set k as k+1 in order to perform k-means clustering. For instance, to group data into three group, k would have to be set as four. This would have obvious limitations as k increased.

The fix to this was to design the algorithm in such a way that it chooses k random indices (without replacement) from the data and uses those points as the centroids. In this way there is an element of randomness to it, if a cluster happens to have zero data points for some reason (maybe the random indices were points close together and they all go grouped in a cluster with a slightly closer centroid), it can simply be rerun. In this way it is done without a random seed, so that values of the random indices are bound to change each time it is run. The k-means algorithm is then implemented as such:

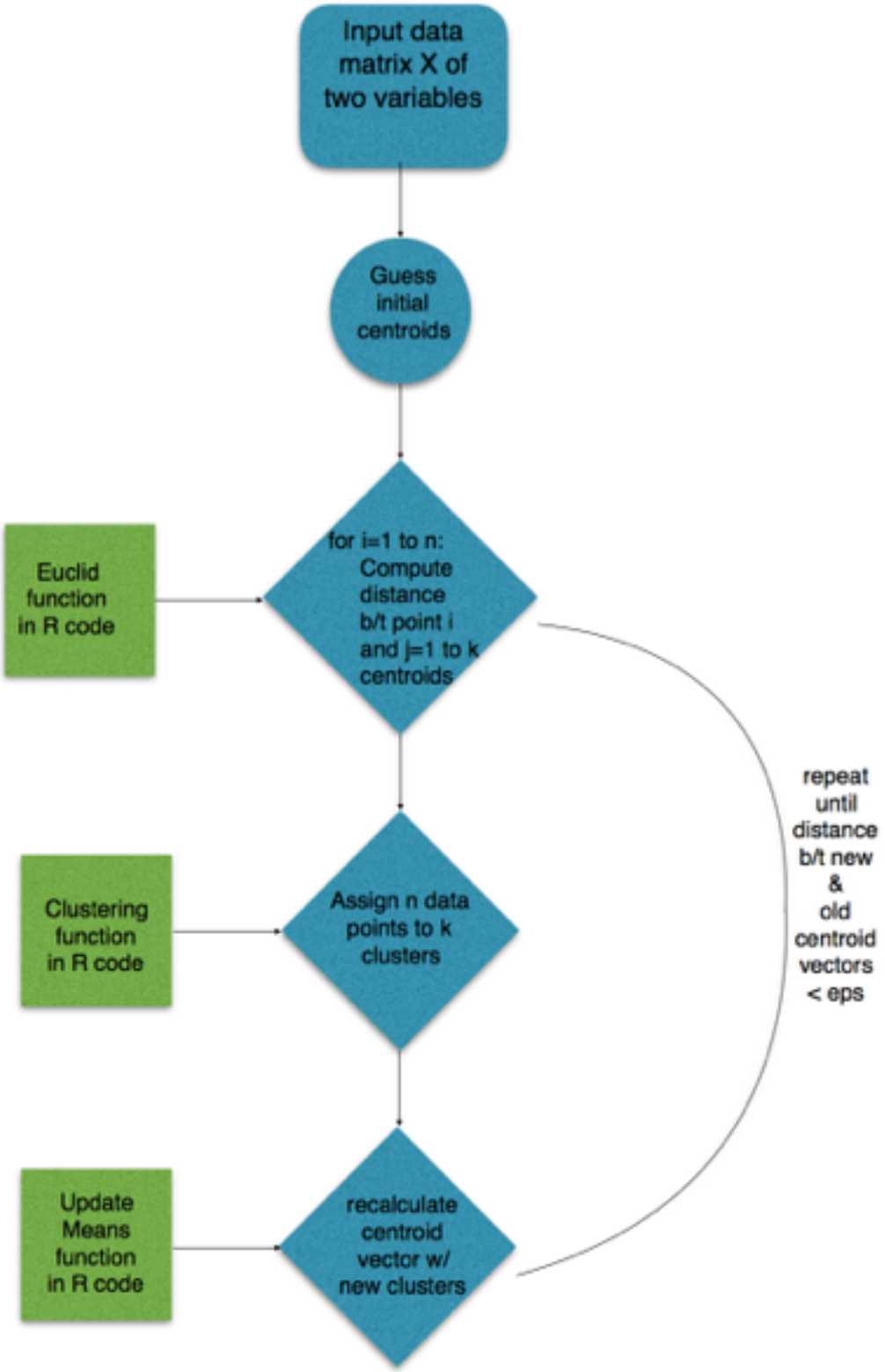
Given a two column matrix of data to cluster into k categories with a tolerance of epsilon,

1. Randomly choose k indices between one and n.
2. Use the points at these indices as the k initial centroid guesses.
3. For all n data points, compute the euclidean distance between all k centroids.
4. Assign new clusters to each point based on the minimum distance between the point and cluster k.
5. Recalculate the k centroids based on the new cluster assignments.
6. Calculate the distance between the current and previous mean vectors.

7. If this distance is greater than epsilon, continue iterating steps three through six until it drops below epsilon.

8. Output the clusters, number of iterations, final centroid vector and final epsilon.

The algorithm is implemented in R, and can be visualized using the following diagram:



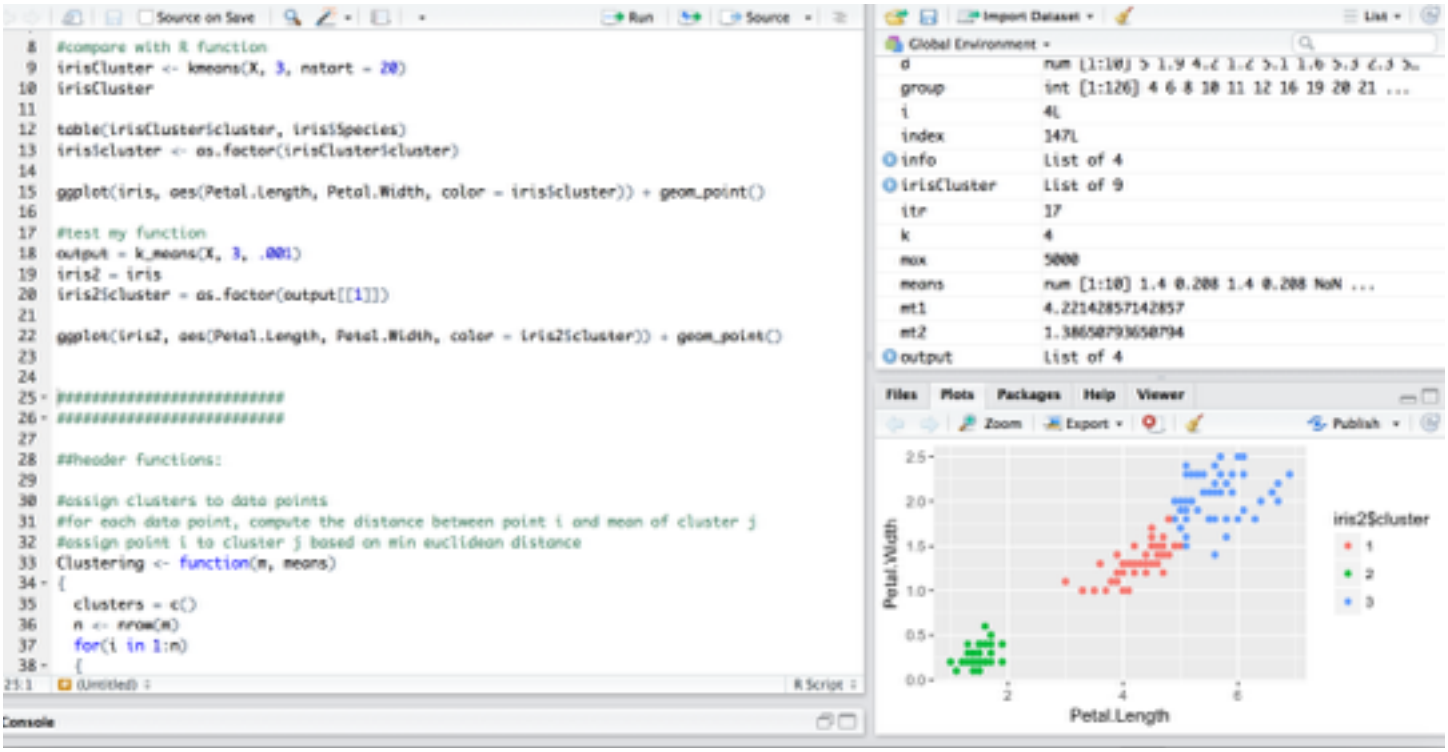
Dataset

The dataset used to test this implementation is the class “iris” dataset from R.A. Fisher. This dataset describes three species of flowers, setosa, versicolor, and virginica by attributes such as both length and width of their sepals and petals. This is a useful example for clustering because of the more structured nature of the data: we can use k=3 and examine the performance of the algorithm against the actual species of flowers. We can also experiment with higher or lower values of k and see how the algorithm clusters the data into a higher or lower number of groups.

Snapshot

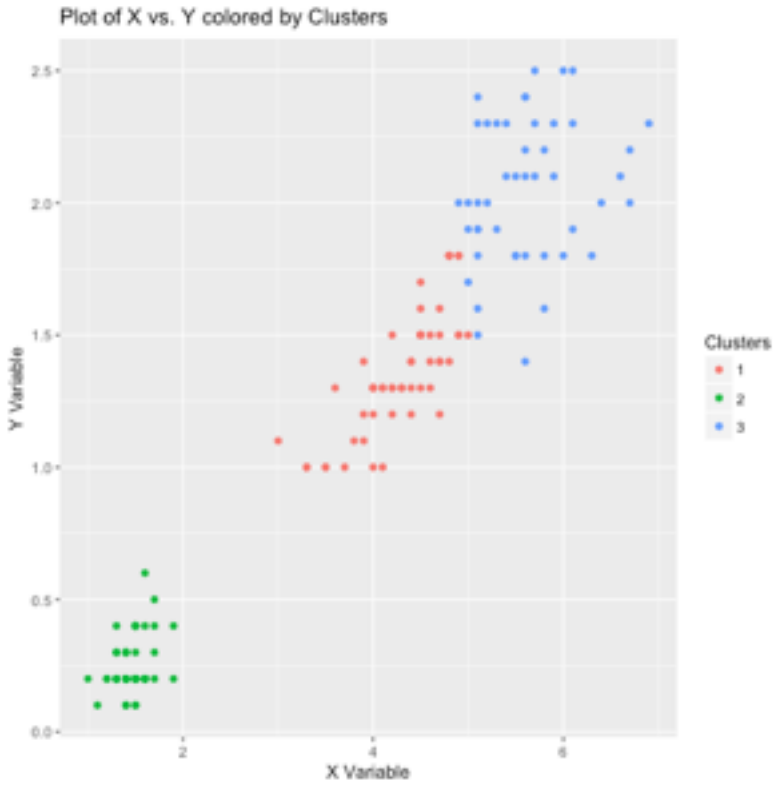
Below is a screenshot of the algorithm as it is run in R, including a graph of the final resulting clusters applied to the aforementioned iris dataset. The algorithm was compared to the R function kmeans() and achieved similar results on the same dataset.

The algorithm is made more versatile using the linux command line. In takes inputs of: the data file (.csv), x column index, y column index, k, epsilon and an output filename to output the dataset with a column of clusters appended (as a .csv) and the graph of data with color coded clusters (as a .png).



Which results in the following outputted files:

	A	B	C	D	E	F
1	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	cluster
2	5.1	3.5	1.4	0.2	setosa	2
3	4.9	3	1.4	0.2	setosa	2
4	4.7	3.2	1.3	0.2	setosa	2
5	4.6	3.1	1.5	0.2	setosa	2
6	5	3.6	1.4	0.2	setosa	2
7	5.4	3.9	1.7	0.4	setosa	2
8	4.6	3.4	1.4	0.3	setosa	2
9	5	3.4	1.5	0.2	setosa	2
10	4.4	2.9	1.4	0.2	setosa	2
11	4.9	3.1	1.5	0.1	setosa	2
12	5.4	3.7	1.5	0.2	setosa	2
13	4.8	3.4	1.6	0.2	setosa	2
14	4.8	3	1.4	0.1	setosa	2
15	4.3	3	1.1	0.1	setosa	2
16	5.8	4	1.2	0.2	setosa	2
17	5.7	4.4	1.5	0.4	setosa	2
18	5.4	3.9	1.3	0.4	setosa	2
19	5.1	3.5	1.4	0.3	setosa	2
20	5.7	3.8	1.7	0.3	setosa	2
21	5.1	3.8	1.5	0.3	setosa	2
22	5.4	3.4	1.7	0.2	setosa	2
23	5.1	3.7	1.5	0.4	setosa	2
24	4.6	3.6	1	0.2	setosa	2
25	5.1	3.3	1.7	0.5	setosa	2
26	4.8	3.4	1.9	0.2	setosa	2
27	5	3	1.6	0.2	setosa	2
28	5	3.4	1.6	0.4	setosa	2
29	5.2	3.5	1.5	0.2	setosa	2
30	5.2	3.4	1.4	0.2	setosa	2
31	4.7	3.2	1.6	0.2	setosa	2



Alternatively, one could use it in an R Shiny app that allows the user to input a .csv file of data, choose the two variables, k, and epsilon and output the graph with color coded clusters of the final result.

Use Instructions (for R users)

To use the algorithm, open the .R file.

Read the data in as a data frame to R.

Subset the data frame based on the two variables you wish to cluster.

Load the library ggplot2 to visualize the results.

Run the functions that the main algorithm depends on: Clustering, UpdateMeans, dist, createMeanMatrix, euclid, euclid2 & delta.

Run the k_means() function block of code.

If everything runs without any errors, the algorithm can be applied by running k_means(X, k, eps) with inputted X matrix, k number of clusters and tolerance of eps.

Save the output from this function, which will be a list.

Add the first element of outputted list to the original dataset as a factor named ‘cluster’.

Graph the data in ggplot with color based on cluster factor.

Rerun with various cluster sizes (k) or tolerances (eps) to vary results.

Use Instructions (command line)

Make sure all of the necessary files (data as a .csv, R script as a .R file) are in the same directory.

Navigate to this directory using the command: cd ‘/directory/goes/here/‘

Run the R script with the following command, supplying the necessary arguments specificed:

Rscript ‘R Script Filename.R’ [input_file.csv X_column_index Y_column_index k tolerance output_file_name]

note that the output file name should not include any file extension, these will be appended as necessary in the R script.

Findings, lessons, experiences

Implementing the k-means algorithm was incredibly beneficial in terms of understanding how it works, and the specific mechanics of it. It was also useful to have a greater level of flexibility in terms of specifying initial centroids, which could easily be adapted to suit a particular problem and involve educated guesses based on cluster size k instead of relying on an algorithmic method as described above.

In working with the initial centroid guesses, it was helpful that the first method of using the first k observations resulted in an error so that a moderately more robust method could be used. This stressed the importance of a random initialization that would change on a different running of the code, so that users could bypass errors with little or no knowledge of the “under the hood” workings of k-means.

Another takeaway was the distance metric used in the algorithm. While standard euclidean distance was used in this case, it was interesting to see the ease at which any other distance function could be used depending on the specific use case.

The experience also highlighted the simplicity of k-means, despite it’s usefulness and power in data science. It also showcased the versatile nature of the algorithm, with modifications

and customizations able to be done at every step without loss of functionality in previous or successive steps.

Overall, the implementation was relatively straightforward and enlightening in terms of thinking about more complex algorithms and models.

Conclusions

As alluded to throughout the report, there were some limitations and upgrades that could be made to the algorithm as it currently stands. The main limitations lie in the initial centroid guess, which could still give errors depending on the use case. This would ideally be overcome by adapting the algorithm to work with edge cases or having some sort of internal check that guarantees the desired result is achieved as much as is feasible.

Another case that isn't covered in this algorithm is if k=1 or k>n, which would throw off the results, and are somewhat assumed that the intended user is aware of the error in this thinking.

It would also be nice to have a wrapper that allowed for less technical users to benefit from the power of this algorithm and code. For less technically inclined users, an R Shiny application would be a great way to allow a sort of rudimentary GUI that allowed for uploading of data, specification of the k-means paramters, and iterative running of the code with an outputted visualization.

Much can be said about the various modifications that could also be made to the initial guess, distance metrics, and edge cases that more proficient users and developers could certainly improve upon in this code.

Appendix

```
#!/usr/bin/env Rscript
args = commandArgs(trailingOnly=TRUE)

#script will take args:
#data file (csv), x column, y column, k, tolerance, output csv filename.

if (length(args)<6) {
  stop("Too few arguments, try again", call.=FALSE)
}

data = args[1]
```

```
x_col = as.numeric(args[2])
y_col = as.numeric(args[3])
ind = c(x_col, y_col)
k = as.numeric(args[4])
tolerance = as.numeric(args[5])
output = args[6]

print(paste("File used: ", data))
print(paste("Columns for k means: ", x_col, y_col))
print(paste("K means with parameters: ", k, tolerance))
print(paste("Outputting files with path: ", output))

library(datasets)
library(ggplot2)


setwd("/Users/stevenhurwitt/Downloads/Upwork/k means/")

#iris = iris
#write.csv(iris, file = "iris.csv", col.names = T, row.names = F)
input = read.csv(data, header = T)
X = input[,ind]

#compare with R function
#irisCluster <- kmeans(X, 3, nstart = 20)
#irisCluster

#table(irisCluster$cluster, iris$Species)
#iris$cluster <- as.factor(irisCluster$cluster)

#ggplot(iris, aes(Petal.Length, Petal.Width, color = iris$cluster)) + geom_point()


############################
############################

##header functions:

#assign clusters to data points
#for each data point, compute the distance between point i and mean of cluster j
```

```
#assign point i to cluster j based on min euclidean distance
Clustering <- function(m, means)
{
  clusters = c()
  n <- nrow(m)
  for(i in 1:n)
  {
    distances = c()
    k <- nrow(means)
    for(j in 1:k)
    {
      di <- m[i,] - means[j,]
      ds<-euclid2(di)
      distances <- c(distances, ds)
    }
    minDist <- min(distances)
    cl <- match(minDist, distances)
    clusters <- c(clusters, cl)
  }
  return (clusters)
}

#update means for clusters
#calc new mean vector (length k) based on new cluster assignments
UpdateMeans <- function(m, cl, k)
{
  means <- c()
  for(c in 1:k)
  {
    # get the point of cluster c
    group <- which(cl == c)

    # compute the mean point of all points in cluster c
    mt1 <- mean(m[group,1])
    mt2 <- mean(m[group,2])
    vMean <- c(mt1, mt2)
    means <- c(means, vMean)
  }
  means <- createMeanMatrix(means)
  means[is.na(means)] <- 0
```

```
  return(means)
}

#distance function for euclidean dist b/t two points
dist <- function(x,y)
{
  d<-sqrt( sum((x - y) **2 ))
}

#create matrix for distances
createMeanMatrix <- function(d)
{
  matrix(d, ncol=2, byrow=TRUE)
}

# compute euclidean distance
euclid <- function(a,b){
  d<-sqrt(a**2 + b**2)
}
euclid2 <- function(a){
  d<-sqrt(sum(a**2))
}

#compute difference between new means and old means
delta <- function(oldMeans, newMeans)
{
  a <- newMeans - oldMeans
  max(euclid(a[, 1], a[, 2]))
}

##############K Means#################


k_means <- function(m, k, tol)
{
  #initialization for k means: k random observations from data
  x <- m[, 1]
  y <- m[, 2]
  d=matrix(data=NA, ncol=0, nrow=0)
  threshold = tol
```

```
  samp_ind = sample(1:length(x), k, replace = F)
  for(i in 1:k){
    index = samp_ind[i]
    d <-  c(d, c(x[index], y[index]))}

  init <- matrix(d, ncol=2, byrow=TRUE)
  oldMeans <- init
  oldMeans
  cl <- Clustering(m, oldMeans)
  cl
  means <- UpdateMeans(m, cl, k)
  thr <- delta(oldMeans, means)
  itr <- 1
  print("initial centroids calculated")

  #main algorithm: cluster points into k groups based on
  #distance to mean of group, continue until distance between
  #mean vectors of successive runs is less than a tolerance, epsilon
  print("updating centroids...")
  while(thr > threshold)
  {
    cl <- Clustering(m, means)
    oldMeans <- means
    means <- UpdateMeans(m, cl, k)
    thr <- delta(oldMeans, means)
    itr <- itr+1
    print(paste("iteration ", itr))
  }
  print("... done")
  info = list(cl, thr, means, itr)
  return(info)
}


####################
###test function####

#test my function
```

```
print("Running k-means algorithm...")
km_output = k_means(X, k, tolerance)
output_data = input
output_data$cluster = as.factor(km_output[[1]])

print("Saving plot...")
png(paste(output, ".png"))
plt = ggplot(output_data, aes(output_data[,x_col], output_data[,y_col], color =
output_data$cluster)) + geom_point()
plt = plt + xlab("X Variable") + ylab("Y Variable")
plt = plt + labs(title = "Plot of X vs. Y colored by Clusters", colour = "Clusters")
print(plt)
dev.off()

print("Saving csv....")
write.csv(output_data, file = paste(output, ".csv"), col.names = T, row.names = F)
```