# Project 2: Clustering Algorithms

## Visualizing given data

First step is to apply dimensionality reduction on given dataset using PCA.

```
pca = PCA(n_components=2)
pca.fit(features)
features = np.array(pca.transform(features))

features_x = []
features_y = []

for ele in features:
    features_x.append(ele[0])
    features_y.append(ele[1])
```

Fig: Python code snippet for PCA

Next is to prepare dataset for plotting using ground truth.

```
predictions = []
for i in range(1, no_of_objects + 1):
    predictions.append(id_ground_truth_map[i])

# get colors to represent labels
colors_selected = random.sample(colors, len(set(predictions)))

# assigning colors to labels
cluster_no_color_map = {}
cluster_colors = []
for cluster_no in predictions:
    if cluster_no not in cluster_no_color_map:
        color = colors_selected.pop(0)
        cluster_no_color_map[cluster_no] = color
        cluster_colors.append(color)
    else:
        cluster_colors.append(cluster_no_color_map[cluster_no])
```

Fig: Python code snippet to assign colors to clusters

Finally, we plot the datasets.

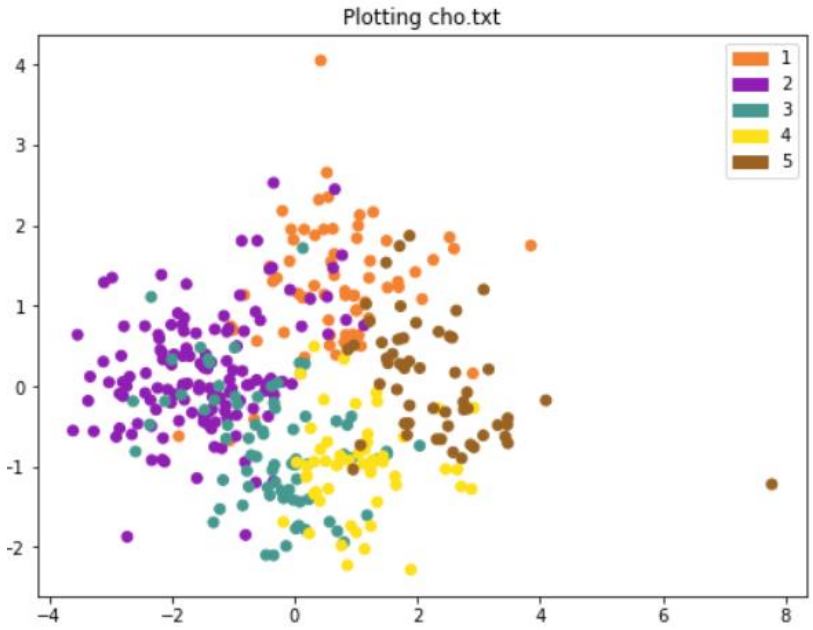


Fig: Plotting cho.txt with ground truth clusters

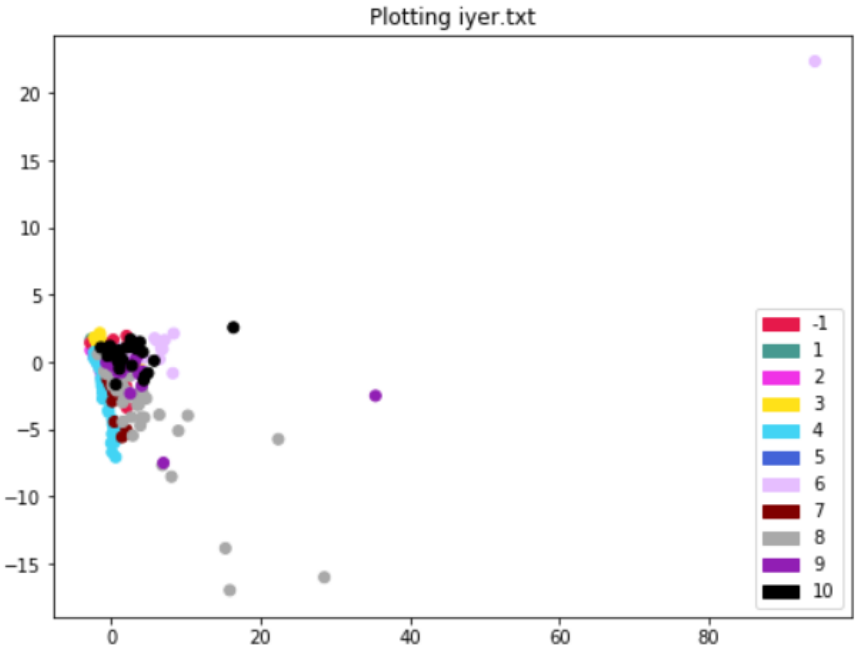


Fig: Plotting iyer.txt with ground truth clusters

## K-Means Clustering

Select initial centroids from user input or random selection.

**Note**: If centroids not given by user, max cluster number in ground truth is selected as no of clusters and then random selection is done.

```
if initial_centroids_given == False:
    no_of_clusters = np.max(ground_truths)
    centroids = choices(features, k = no_of_clusters)
else:
    no_of_clusters = len(initial_centroids[file_index])
    centroids = []
    for id in initial_centroids[file_index]:
        centroids.append(id_feature_map[id])
```

Fig: Python code snippet for selecting centroids

Assign points to nearest centroids.

```
for id in ids:
    min_dist = -1
    cluster_selected = -1
    for centroid_index, centroid in enumerate(centroids):
        diff = np.subtract(id_feature_map[id], centroid)
        diff = [pow(x, 2) for x in diff]
        curr_dist = np.sum(diff)
        if min_dist == -1:
            min_dist = curr_dist
            cluster_selected = centroid_index + 1
        elif curr_dist < min_dist:
            min_dist = curr_dist
            cluster_selected = centroid_index + 1
    clusters[cluster_selected].append(id)
```

Fig: Python code snippet for assigning points to nearest centroids

Update centroids as average of points in cluster.

```
# update centroids
for centroid_index, centroid in enumerate(prev_centroids):
    features_in_cluster = clusters[centroid_index+1]
    if len(features_in_cluster) > 0:
        features_in_cluster = [id_feature_map[x] for x in features_in_cluster]
        features_in_cluster = pd.DataFrame(features_in_cluster)
        no_of_features_in_cluster = len(features_in_cluster)
        temp = []
        for i in range(0, no_of_attributes):
            temp.append(np.sum(features_in_cluster[i])/no_of_features_in_cluster)
        new_centroids.append(temp)
    else:
        new_centroids.append(centroid)
```

Fig: Python code snippet for updating centroids

If previous and new centroids match, convergence is reached and no further calculation is needed.

```
for index, new_centroid in enumerate(new_centroids):
    convergence_reached = centroids_equal(new_centroid, prev_centroids[index])
    if not convergence_reached:
        break
```

Fig: Python code snippet for convergence check

Else calculation is continued till max iteration is reached.

For more cluster information refer: **K_Means_Clusters.txt**

Generate ground truth matrix, prediction matrix from id_ground_truth_map and id_prediction_map.

```
def get_prediction_matrix(no_of_objects, id_prediction_map):
    columns = list(np.arange(1, no_of_objects+1))
    columns = [int(x) for x in columns]
    rows = list(np.arange(1, no_of_objects+1))
    rows = [int(x) for x in rows]

    prediction_matrix = pd.DataFrame(columns = columns, index = rows)

    for row in range(1, no_of_objects + 1):
            for column in range(1, row+1):
                if row == column:
                    prediction_matrix[column][row] = 1
                else:
                    if id_prediction_map[column] == id_prediction_map[row]:
                        prediction_matrix[column][row] = 1
                    else:
                        prediction_matrix[column][row] = 0
    return prediction_matrix
```

Fig: Python code snippet for generating prediction matrix

Evaluate clusters using external index.

```
def evaluate_clusters(no_of_objects, ground_truth_matrix, prediction_matrix):
    m11 = 0
    m00 = 0
    m10 = 0
    m01 = 0
    for row in range(1, no_of_objects + 1):
        for column in range(1, row+1):
            if ground_truth_matrix[column][row] == 1 and prediction_matrix[column][row] == 1:
                m11 = m11 + 1
            elif ground_truth_matrix[column][row] == 0 and prediction_matrix[column][row] == 0:
                m00 = m00 + 1
            elif ground_truth_matrix[column][row] == 1 and prediction_matrix[column][row] == 0:
                m10 = m10 + 1
            elif ground_truth_matrix[column][row] == 0 and prediction_matrix[column][row] == 1:
                m01 = m01 + 1
    jaccard_coefficient = m11/(m11+m10+m01)
    print("==>Jaccard Coefficient: "+str(round(jaccard_coefficient, 2)))
    rand_index = (m11+m00)/(m11+m00+m10+m01)
    print("==>Rand Index: "+str(round(rand_index, 2)))
```

Fig: Python code snippet for cluster evaluation

**Pros:**

- Efficient: O(tkn), where n is # objects, k is # clusters, and t is # iterations.
- Easy to implement.

**Cons:**

- Need to specify K, the number of clusters.
- Initialization matters.
- Problems with clusters of varying densities and shapes.

**Findings**:

**<u>Case 1: K-means observations for user given centroids</u>:**

initial centroids = [50,100,150,200,250], max iteration = 100

| **Cho.txt** |
|---|

==>**Centroids**
[0.11, -0.14, -0.6, -0.68, -0.57, -0.42, -0.35, -0.1, 0.33, 0.62, 0.35, 0.08, -0.04, 0.06, 0.17, 0.44]
[-0.57, -0.46, -0.17, 0.29, 0.62, 0.56, 0.29, 0.05, -0.18, -0.36, -0.05, 0.23, 0.27, 0.14, 0.11, -0.15]
[-0.43, 0.24, 1.4, 0.72, -0.12, -0.28, -0.66, -1.04, -0.8, 0.94, 0.78, 0.35, -0.01, -0.4, -0.76, -0.56]
[-0.31, -0.1, 0.71, 0.41, 0.1, -0.12, -0.31, -0.44, -0.36, 0.45, 0.42, 0.17, -0.05, -0.24, -0.33, -0.24]
[-0.83, -0.89, -1.01, -0.64, -0.25, 0.21, 0.76, 0.98, 0.81, -0.13, -0.38, -0.13, 0.06, 0.39, 0.58, 0.48]

==>**Jaccard Coefficient: 0.34**
==>**Rand Index: 0.78**

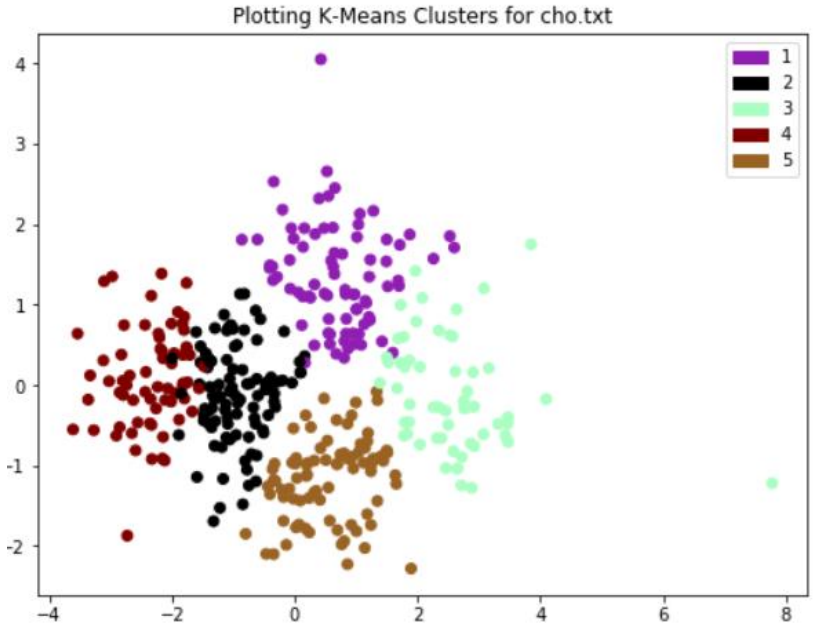

Fig: Plot for K-means for cho.txt with user selected centroids

| **Iyer.txt** |
|---|

==>**Centroids**
[1.0, 0.96, 0.95, 0.97, 0.92, 0.76, 0.64, 0.59, 0.6, 0.82, 0.85, 0.92]
[1.0, 2.67, 2.41, 3.75, 6.18, 38.8, 86.92, 25.58, 14.81, 6.73, 2.72, 2.42]
[1.0, 1.12, 2.2, 3.78, 3.39, 10.6, 13.69, 14.4, 8.97, 8.4, 7.62, 6.56]
[1.0, 1.76, 3.42, 5.61, 4.01, 2.72, 2.43, 2.34, 1.59, 1.38, 1.31, 1.24]
[1.0, 0.98, 1.14, 1.23, 1.35, 2.32, 2.55, 2.81, 2.43, 2.19, 2.37, 2.63]

==>**Jaccard Coefficient: 0.27**
==>**Rand Index: 0.61**

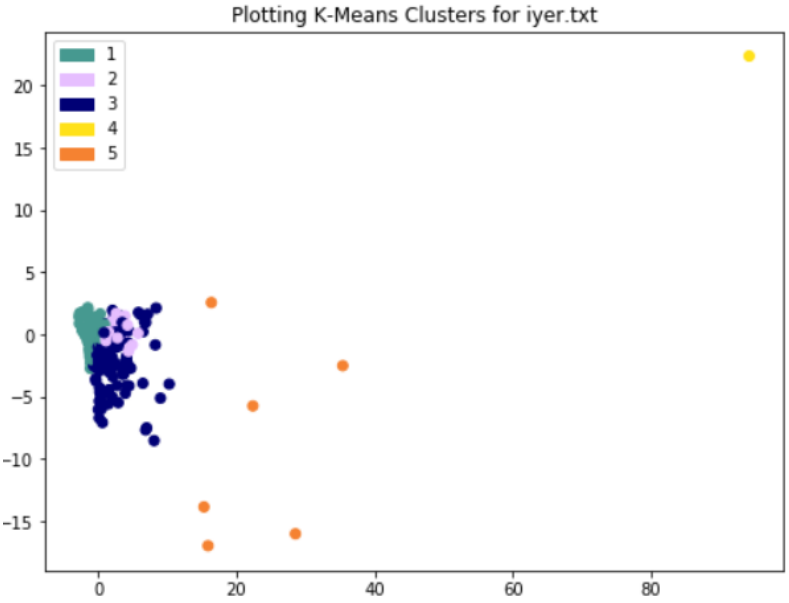


Fig: Plot for K-means for iyer.txt with user selected centroids

**Case 2: K-means observations for random centroids:**

| **Cho.txt** |
|---|

No of clusters = 5, max iteration = 100

==>**Centroids**
[-0.53, -0.45, -0.21, 0.27, 0.61, 0.57, 0.3, 0.06, -0.16, -0.39, -0.08, 0.22, 0.27, 0.14, 0.11, -0.13]
[-0.41, 0.25, 1.4, 0.71, -0.12, -0.28, -0.66, -1.03, -0.79, 0.93, 0.77, 0.34, -0.01, -0.41, -0.76, -0.56]
[-0.38, -0.14, 0.72, 0.45, 0.13, -0.1, -0.29, -0.43, -0.37, 0.45, 0.43, 0.18, -0.03, -0.22, -0.31, -0.25]
[0.12, -0.11, -0.56, -0.64, -0.54, -0.42, -0.35, -0.12, 0.3, 0.61, 0.35, 0.09, -0.04, 0.04, 0.13, 0.42]
[-0.82, -0.89, -1.03, -0.67, -0.27, 0.19, 0.73, 0.97, 0.82, -0.1, -0.36, -0.13, 0.07, 0.39, 0.59, 0.5]

==>**Jaccard Coefficient: 0.34**
==>**Rand Index: 0.78**

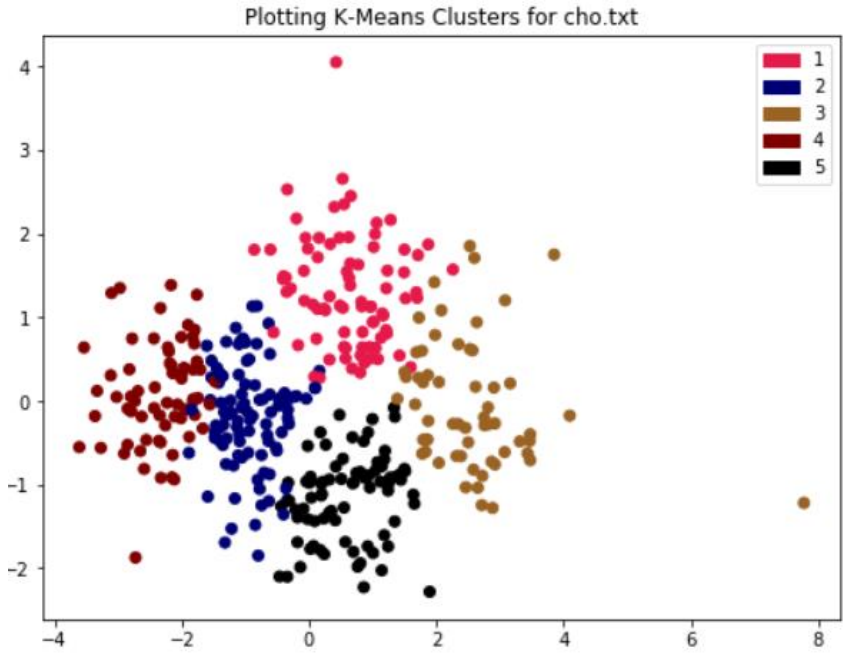


Fig: Plot for K-means for cho.txt with random centroids

After multiple runs,

**Jaccard coefficient was in range:** **0.34 – 0.41**

**Rand Index was in range:** **0.78 – 0.8**

| **Iyer.txt** |
|---|

No of clusters = 10, max iteration = 100

==>**Centroids**
[1.0, 0.94, 0.94, 0.91, 0.88, 0.69, 0.56, 0.48, 0.45, 0.65, 0.62, 0.65]
[1.0, 2.67, 2.41, 3.75, 6.18, 38.8, 86.92, 25.58, 14.81, 6.73, 2.72, 2.42]
[1.0, 1.17, 1.3, 1.15, 1.22, 1.08, 1.15, 1.23, 1.54, 2.97, 4.51, 6.7]
[1.0, 0.95, 0.98, 1.05, 1.11, 1.35, 1.5, 1.82, 2.03, 2.14, 2.35, 2.61]
[1.0, 1.8, 3.83, 6.76, 5.12, 4.17, 3.51, 3.37, 1.81, 1.56, 1.46, 1.34]
[1.0, 0.99, 1.15, 1.23, 1.5, 2.85, 4.39, 5.77, 4.57, 3.21, 3.03, 2.66]
[1.0, 1.0, 0.9, 0.93, 0.92, 0.73, 0.63, 0.63, 0.74, 1.11, 1.23, 1.36]
[1.0, 1.24, 1.94, 2.49, 2.05, 1.65, 1.55, 1.64, 1.45, 0.97, 0.94, 0.89]
[1.0, 1.02, 2.03, 3.52, 1.78, 10.2, 14.35, 15.17, 10.08, 9.86, 8.97, 7.7]
[1.0, 0.96, 1.15, 1.43, 1.57, 3.67, 3.38, 2.96, 1.94, 1.45, 1.36, 1.27]

==>**Jaccard Coefficient: 0.31**
==>**Rand Index: 0.78**

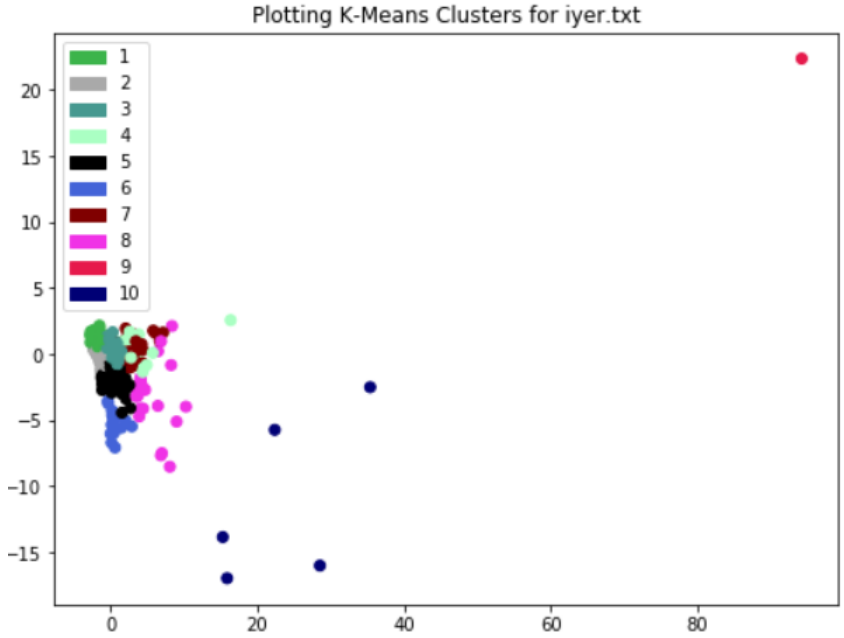


Fig: Plot for K-means for iyer.txt with random centroids

After multiple runs,

**Jaccard coefficient was in range:** **0.3 – 0.35**

**Rand Index was in range:** **0.72 – 0.78**

## Hierarchical Agglomerative Single Link Clustering

**Note**: Max cluster number in ground truth is selected as number of clusters

Rename column and row indices to start from 1 instead of 0.

```
features = data.iloc[:,2:no_of_columns]
features.columns = [x for x in range(1, no_of_columns - 1)]
features.index = np.arange(1, no_of_objects+1)
```

Fig: Python code snippet for column and row indices renaming

Generate a distance matrix for the dataset with min value at row and column intersection.

```
min_distance = -1
min_row = -1
min_column = -1

for row in range(1, no_of_objects + 1):
    for column in range(1, row+1):
        if row == column:
            # pd accessed by column, row
            base_distance_matrix[column][row] = 0
        else:
            feature1 = id_feature_map[column]
            feature2 = id_feature_map[row]
            diff = np.subtract(feature1, feature2)
            diff_square = [pow(x, 2) for x in diff]
            total = np.sum(diff_square)
            if min_distance == -1:
                min_distance = total
                min_row = row
                min_column = column
            else:
                if total < min_distance:
                    min_distance = total
                    min_row = row
                    min_column = column
            base_distance_matrix[column][row] = total
```

Fig: Python code snippet for distance matrix with min value

Combine min_row and min_column as single cluster and update distance matrix.

```
for row in rows:
    for column in columns:
        if row == column:
            distance_matrix[column][row] = 0
            break
        else:
            if isinstance(column, int) and isinstance(row, int):
                dist_val = base_distance_matrix[column][row]
                distance_matrix[column][row] = dist_val
                if min_distance == -1:
                    min_distance = dist_val
                    min_row = row
                    min_column = column
                else:
                    if dist_val < min_distance:
                        min_distance = dist_val
                        min_row = row
                        min_column = column
```

Fig: Python code snippet for updating distance matrix

Repeat above steps until only one column and one row is left that has all points.

To get clusters, clip the dendogram at a level that has correct number of clusters.

For example: Consider input 1,2,3,4,5

Initially every point is in its own cluster: {1} {2} {3} {4} {5}

If 1,2 has min distance:

{1,2} {3} {4} {5}

{1,2,3} {4} {5} ← Number of clusters = 3

{1,2,3} {4,5}

{1,2,3,4,5}

If no_of_clusters = 3 then return {1,2,3} {4} {5}

```
temp = clusters[len(clusters)-1]
temp.remove([min_row])
temp.remove([min_column])
temp.append([cluster])
cluster_no_ids_map[len(temp)] = str(temp)
```

Fig: Python code snippet for storing cluster no and ids in that cluster

For more cluster information refer: **Agg_Clustering_Single_Link_Clusters.txt**

Generating ground truth matrix, prediction matrix and evaluation of algorithm is same as k-means.

**Pros**:

- No need to specify cluster numbers as clusters are obtained by cutting of dendogram.
- Better suited for classification problems.
- Handle non-elliptical shapes.

**Cons**:

- O(N^2) space and O(N^3) time complexity
- Sensitive to noise and outliers.

**Findings**:

| **Cho.txt** |
|---|

==>**Jaccard Coefficient: 0.23**
==>**Rand Index: 0.24**

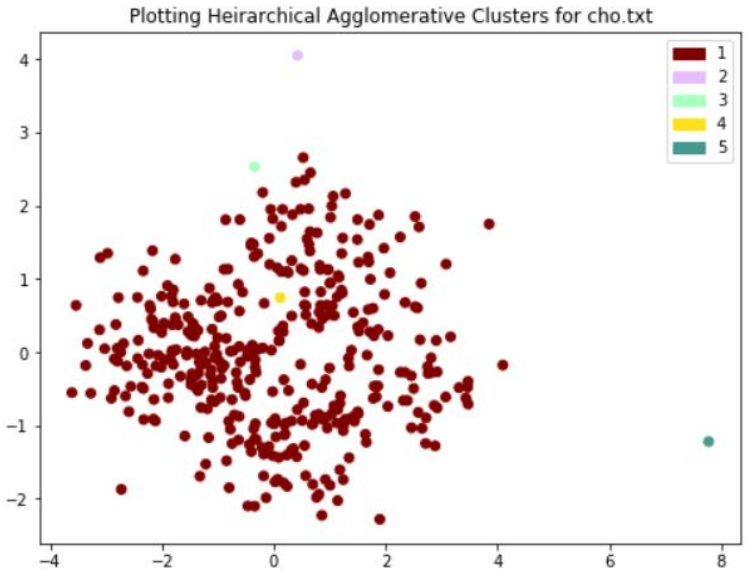


Fig: Plotting hierarchical single link clusters for cho.txt

==>**Jaccard Coefficient: 0.16**
==>**Rand Index: 0.19**

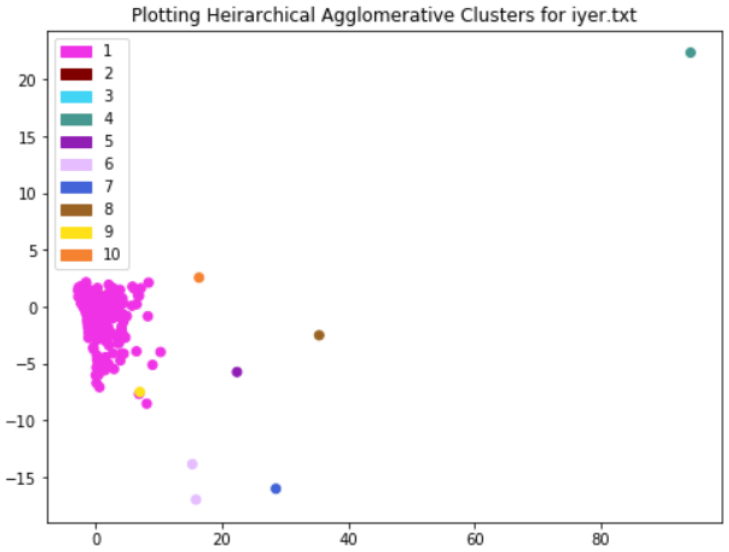


Fig: Plotting hierarchical single link clusters for iyer.txt

## Density Based Clustering

**Note**: No need to give number of clusters.

Maintain visited and added to cluster flags for each point

```
for index, id in enumerate(ids):
    # loc gives row with row_label as argument
    id_feature_map[id] = features[index]
    id_flags_map[id] = [False, False]
```

Fig: Python code snippet of storing flags

For each unvisited point, get its neighbors and expand cluster if min_pts count is satisfied.

Expand cluster means for each point in the neighbor, find its neighbors and if min_pts count is satisfied add it to the initial list of neighbors.

```
for id in ids:
    cluster = []
    flags = id_flags_map[id]
    if flags[0] == False:
        id_flags_map[id] = [True, flags[1]]
        neighbors = getNeighborPts(id, epsilon)
        if len(neighbors) < min_pts:
            # add to noise
            noise.append(id)
        else:
            cluster = expandCluster(id, neighbors, epsilon, min_pts)
    if len(cluster) > 0:
        print("==>"+str(cluster))
        clusters.append(cluster)
```

Fig: Python code snippet of density based clustering

Helper functions

```
def getNeighborPts(id, epsilon):
    neighbors = []
    for ele in ids:
        given_point = id_feature_map[id]
        ele_in_list = id_feature_map[ele]
        diff = np.subtract(given_point, ele_in_list)
        diff_square = [pow(x,2) for x in diff]
        total = np.sum(diff_square)
        if total <= epsilon:
            neighbors.append(ele)
    return neighbors

def expandCluster(id, neighbors, epsilon, min_pts):
    cluster = []
    cluster.append(id)
    id_flags_map[id] = [True, True]
    for neighbor in neighbors:
        flags = id_flags_map[neighbor]
        if flags[0] == False:
            id_flags_map[neighbor] = [True, flags[1]]
            new_neighbors = getNeighborPts(neighbor, epsilon)
            if len(new_neighbors) >= min_pts:
                for new_neighbor in new_neighbors:
                    if new_neighbor not in neighbors:
                        neighbors.append(new_neighbor)
        if flags[1] == False:
            cluster.append(neighbor)
            id_flags_map[neighbor] = [True, True]
    return cluster
```

Fig: Python code snippet of density based clustering helper functions

Points that don't satisfy min_pts count are added to noise.

For more cluster information refer: **Density_Based_Clusters.txt.**

Generating ground truth matrix, prediction matrix and evaluating clusters is same as earlier algorithms.

**Pros**:

- Resistant to noise.
- Handle clusters with different shapes and sizes.

**Cons**:

- Sensitive to parameters.
- Cannot handle varying densities.

**Findings**:

| **Cho.txt** |
|---|

epsilon = 1
min_pts = 2

Objects classified as noise: 206

==>**Jaccard Coefficient: 0.2**
==>**Rand Index: 0.57**

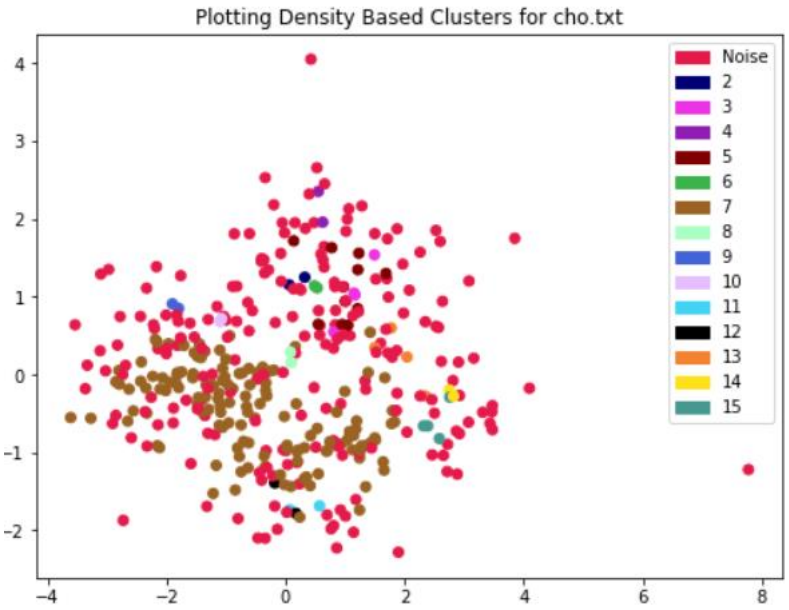

Fig: Plotting density based clusters for cho.txt

Reducing epsilon to 0.5 but maintaining min_pts at 2

Objects classified as noise: 349
==>**Jaccard Coefficient: 0.21**
==>**Rand Index: 0.32**

The external index drops as distance is reduced, as less neighbors are found that doesn't meet min_pts. Also more points are added to noise.

Increasing epsilon to 1.2 but maintaining min_pts at 2

Objects classified as noise: 155
==>**Jaccard Coefficient: 0.2**
==>**Rand Index: 0.58**

Expected reduction in number of points added to noise and increase in index values as more points are added as neighbors due to large epsilon value.

If we keep on increasing epsilon further, lesser number of clusters is generated and index values goes down.

| **Iyer.txt** |
|---|

epsilon = 1
min_pts = 2

Objects classified as noise: 139

==>**Jaccard Coefficient: 0.29**
==>**Rand Index: 0.67**

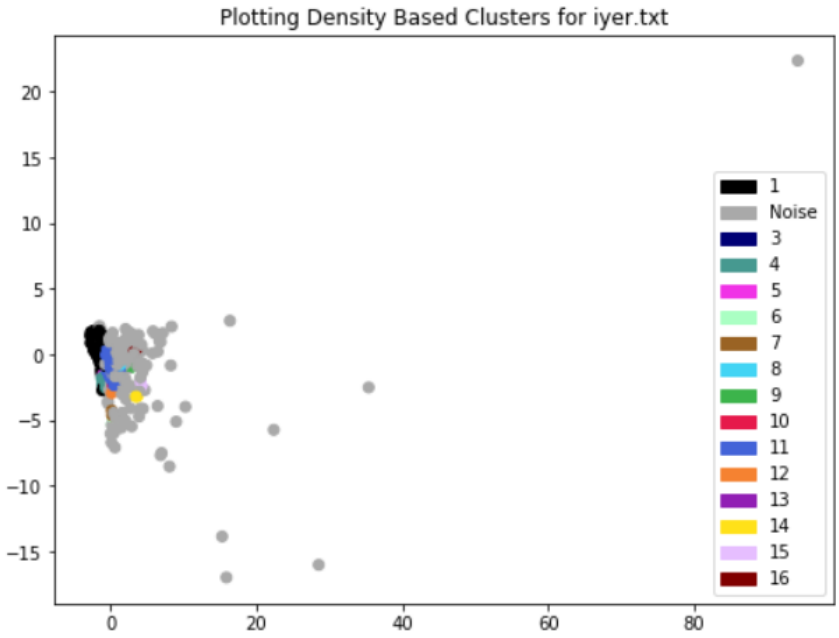


Fig: Plotting density based clusters for iyer.txt

Reducing epsilon to 0.5 but maintaining min_pts at 2

Objects classified as noise: 215
==>**Jaccard Coefficient: 0.29**
==>**Rand Index: 0.65**

Same behavior as cho.txt

Increasing epsilon to 1.2 but maintaining min_pts at 2

Objects classified as noise: 126
==>**Jaccard Coefficient: 0.24**
==>**Rand Index: 0.57**

Above we can observe the dip in index values when we increase epsilon as mentioned earlier.

**Comparison of algorithms**:

| **Criteria** | **K- Means** | **Hierarchical (Min)** | **Density Based** |
|---|---|---|---|
| # of clusters required | Yes | No | No |
| Initialization matters | Yes | No | Yes |
| Shape of clusters | Elliptical | Non Elliptical also | Various |
| Noise clusters | No | No | Yes |

## Map Reduce K-Means

Using Hadoop infrastructure and Linux.

**Pre requisites**:
Working Hadoop environment is setup.

Start the single node dfs:
/hadoop-2.7.7/sbin$ ./start-dfs.sh

Make directories for input and output:
/hadoop-2.7.7/bin$ ./hdfs dfs -mkdir /input
/hadoop-2.7.7/bin$ ./hdfs dfs -put /home/satvinder/workspace/kmeans/iyer.txt /input
/hadoop-2.7.7/bin$ ./hdfs dfs -put /home/satvinder/workspace/kmeans/cho.txt /input

**Map Task**:

- Each map task receives single line from the input file and initial set of centroids from a list.
- Emits out centroid value selected for the point and the point.

**Ex**:
(1, 2) ←Line from file
[(1,1),(5,5)] ←List of centroids

Emit ((1,1), (1,2))

```
public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter)
List<String[]> list_of_centroids = new ArrayList<>();
for(int i=0; i<initial_centroids.size(); i++){
   list_of_centroids.add(initial_centroids.get(i).split("\t"));
}
String input = value.toString();
String[] temp = input.split("\t");
String[] feature = new String[temp.length-2];
for(int i=2; i<temp.length; i++){
 feature[i-2] = temp[i];
}
double min_distance = -1;
double curr_distance = 0;
String[] curr_centroid = null;
String[] min_centroid = null;
for(int i=0; i<list_of_centroids.size(); i++){
 curr_centroid = list_of_centroids.get(i);
 curr_distance = 0;
 for(int j=0; j<curr_centroid.length; j++){
   curr_distance = curr_distance + Math.pow(Double.parseDouble(feature[j])-Double.parseDouble(curr_c
 }
 curr_distance = Math.sqrt(curr_distance);
 if(min_distance == -1){
   min_distance = curr_distance;
   min_centroid = curr_centroid;
 }else{
   if(curr_distance < min_distance){
     min_distance = curr_distance;
     min_centroid = curr_centroid;
   }
 }
}
   output.collect(new Text(Arrays.toString(min_centroid)), new Text(Arrays.toString(feature)));
```

Fig: Java code snippet for map task

**Reduce Task**:

- Each reduce task receives points assigned to a particular cluster.
- If 10 centroids, so 10 reducers.
- Emits out updated centroid as average of all points.

**Ex**:
Reducer for centroid (1,1)
((1,1), (1,2))
((1,1), (3,2))

Emit ((2, 2), “NEW_CENTROID”)

```
public void reduce(Text key, Iterator<Text> values, OutputCollector<Text, Text> output, Reporter
 //System.out.println("Reducer reached for centroid "+key);
 double[] acc_feature = null;
 String[] temp = null;
 String feature = null;
 int no_of_items = 0;
 while(values.hasNext()){
   no_of_items++;
   feature = values.next().toString();
   feature = feature.replace("[","");
   feature = feature.replace("]","");
   temp = feature.split(",");
   if(acc_feature == null){
     acc_feature = new double[temp.length];
     for(int i=0; i<temp.length; i++){
       acc_feature[i] = Double.parseDouble(temp[i]);
     }
   }else{
     for(int i=0; i<temp.length; i++){
       acc_feature[i] = acc_feature[i]+Double.parseDouble(temp[i]);
     }
   }
 }
 for(int i=0; i<acc_feature.length; i++){
   acc_feature[i] = Double.parseDouble(decimal_format.format(acc_feature[i]/no_of_items));
 }
 output.collect(new Text(Arrays.toString(acc_feature)), new Text("NEW_CENTROID"));
```

Fig: Java code snippet for reduce task

**Note**: Combiner could have been used for local aggregation but data is not that huge so we can avoid it.

**Repeat Map-Reduce**:

Use new centroids and repeat map - reduce tasks until convergence or max iteration is reached.

```
while(true){

 if(iteration_no > max_iterations){
   System.out.println("Centroids after Max Iteration");
       for(int i=0; i<initial_centroids.size(); i++){
         System.out.println(initial_centroids.get(i));
       }
       System.out.println("Output at directory "+args[1]+String.valueOf(timestamp)+(iteration_no-1));
   break;
 }

  JobConf conf = new JobConf(Kmeans.class);
    conf.setJobName("kmeans"+iteration_no);

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(Text.class);

    conf.setMapperClass(Map.class);
    // conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]+String.valueOf(timestamp)+iteration_no));

    JobClient.runJob(conf);

    System.out.println("Iteration "+iteration_no+" completed");
```

Fig: Java code snippet for map-reduce iterations

**Note**: For complete code refer: **Kmeans.java**

**After code compile, generate JAR files for execution:**
jar -cvf /home/satvinder/workspace/wordcount.jar -C wordcount_classes/ .

**Execute map reduce:**
/hadoop-2.7.7/bin$ ./hadoop jar /home/satvinder/workspace/kmeans/kmeans.jar org.myorg.Kmeans h dfs:/input/iyer.txt hdfs:/output > /home/satvinder/logs
**OR**
/hadoop-2.7.7/bin$ ./hadoop jar /home/satvinder/workspace/kmeans/kmeans.jar org.myorg.Kmeans h dfs:/input/iyer.txt hdfs:/output 50,100,150,200,250 > /home/satvinder/logs

**Get centroids from algorithm**:

Refer the console logs or get file from HDFS.

```
Iteration 22 completed
Centroids updated
Iteration 23 completed
Convergence reached
Centroids
1.0 0.96  0.95  0.97  0.92  0.76  0.64  0.59  0.6 0.82  0.85  0.92
1.0 0.98  1.14  1.23  1.35  2.32  2.55  2.81  2.43  2.19  2.37  2.63
1.0 1.12  2.2 3.78  3.39  10.6  13.69 14.4  8.97  8.4 7.62  6.56
1.0 1.76  3.42  5.61  4.01  2.72  2.43  2.34  1.59  1.38  1.31  1.24
1.0 2.67  2.41  3.75  6.18  38.8  86.92 25.58 14.81 6.73  2.72  2.42
Output at directory hdfs:/output154077466085023
```

Fig: Log sample of map reduce

Use the path in output directory and file name as part-00000 to get file from HDFS:
/hadoop-2.7.7/bin$ ./hdfs dfs -get hdfs:/ output154065484183510/part-00000 /home/satvinder/worksp ace/kmeans/outputs

Copy centroid files in python directory to get cluster points and evaluate the algorithm.
**Note**: Rename output files from map reduce.
Ex: part-00000 for cho.txt → cho_mapreduce.txt

```
mapreduce_file = filename.replace(".txt","")+"_mapreduce.txt"
mapreduce_output = pd.read_csv(mapreduce_file, delimiter="\t", header=None)
centroids = np.array(mapreduce_output[0])
for index, ele in enumerate(centroids):
    temp = ele.replace("[","")
    temp = temp.replace("]","")
    temp = temp.split(",")
    temp = [float(x) for x in temp]
    centroids[index] = temp
```

Fig: Python code snippet for reading map reduce output file

Getting clusters, generating ground truth and prediction matrix, evaluation is all same as k-means.

**Improving the performance**:

1. Instead of creating a file for initial centroids and updating that file in HDFS, we create a global list that is available to all mappers and mappers read centroids from that global list.
2. Of course, updating of centroids is also done in that global list.

```
static List<String> initial_centroids = new ArrayList<>();
```

Fig: Java code snippet for global list for centroids.

```
if(no_of_matches == initial_centroids.size()){
  System.out.println("Convergence reached");
  System.out.println("Centroids");
  for(int i=0; i<new_centroids.size(); i++){
    System.out.println(new_centroids.get(i));
  }
  System.out.println("Output at directory "+args[1]+String.valueOf(timestamp)+iteration_no);
  break;
}else{
   initial_centroids.clear();
 for(int i=0; i<new_centroids.size(); i++){
  initial_centroids.add(new_centroids.get(i));
 }
 System.out.println("Centroids updated");
```

Fig: Java code snippet for updating centroids.

**Findings**:

**<u>Case 1: Map reduce K-means observations for user given centroids</u>:**

initial centroids = [50,100,150,200,250], max iteration = 100

| **Cho.txt** |
|---|

Convergence reached

==>**Centroids**
[-0.31, -0.1, 0.71, 0.41, 0.1, -0.12, -0.31, -0.44, -0.36, 0.45, 0.42, 0.18, -0.05, -0.24, -0.33, -0.24]
[-0.43, 0.24, 1.4, 0.72, -0.12, -0.28, -0.66, -1.04, -0.8, 0.94, 0.78, 0.35, -0.01, -0.4, -0.76, -0.56]
[-0.57, -0.46, -0.17, 0.29, 0.62, 0.56, 0.29, 0.05, -0.18, -0.36, -0.05, 0.23, 0.27, 0.14, 0.11, -0.15]
[-0.81, -0.88, -1.02, -0.64, -0.25, 0.21, 0.76, 0.97, 0.81, -0.12, -0.38, -0.13, 0.05, 0.38, 0.57, 0.47]
[0.1, -0.14, -0.6, -0.68, -0.57, -0.43, -0.36, -0.11, 0.32, 0.62, 0.35, 0.08, -0.04, 0.06, 0.17, 0.45]

==>**Jaccard Coefficient: 0.34**
==>**Rand Index: 0.78**

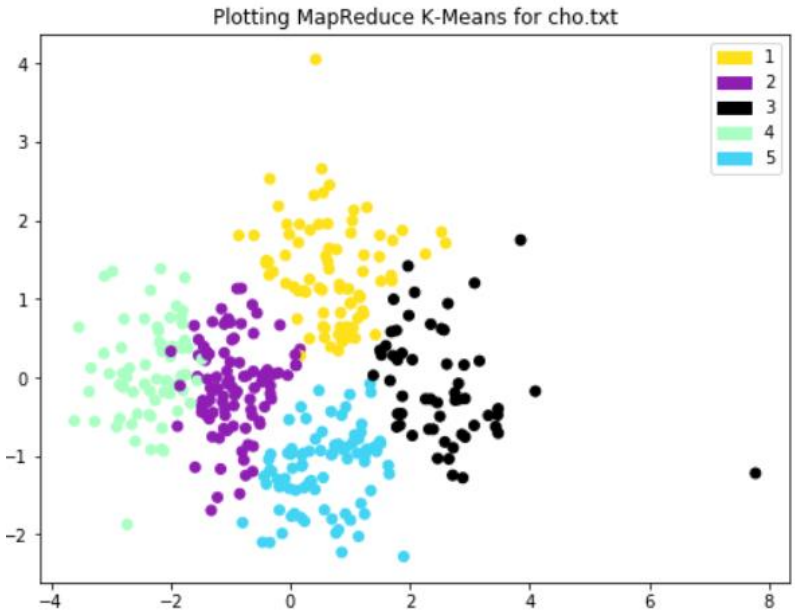


Fig: Plotting map reduce k-means for cho.txt

| **Iyer.txt** |
|---|

Convergence reached

==>**Centroids**
[1.0, 0.96, 0.95, 0.97, 0.92, 0.76, 0.64, 0.59, 0.6, 0.82, 0.85, 0.92]
[1.0, 0.98, 1.14, 1.23, 1.35, 2.32, 2.55, 2.81, 2.43, 2.19, 2.37, 2.63]
[1.0, 1.12, 2.2, 3.78, 3.39, 10.6, 13.69, 14.4, 8.97, 8.4, 7.62, 6.56]
[1.0, 1.76, 3.42, 5.61, 4.01, 2.72, 2.43, 2.34, 1.59, 1.38, 1.31, 1.24]
[1.0, 2.67, 2.41, 3.75, 6.18, 38.8, 86.92, 25.58, 14.81, 6.73, 2.72, 2.42]

==>**Jaccard Coefficient: 0.27**
==>**Rand Index: 0.61**

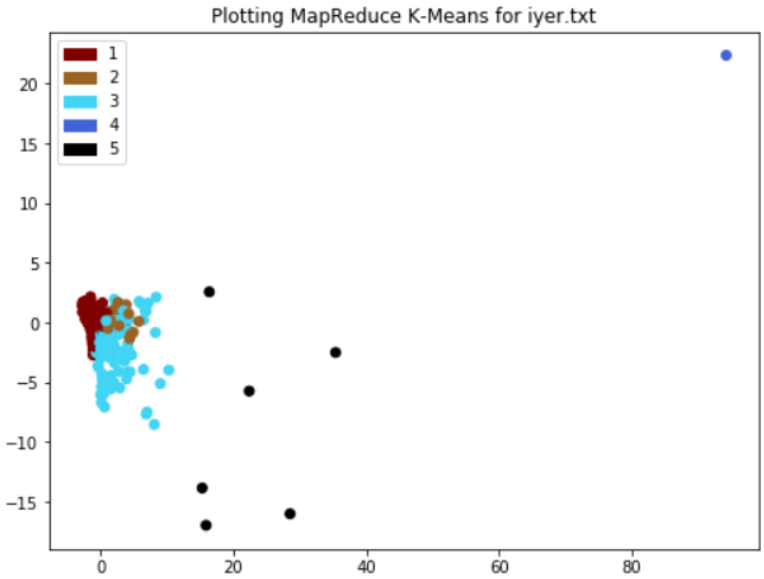

Fig: Plotting map reduce k-means for iyer.txt

As expected, k-means results as indicated by external index are same with user given centroids whether computed by parallel or non-parallel methods.

**<u>Case 2: Map reduce K-means observations for random centroids</u>:**

| **Cho.txt** |
|---|

Number of clusters = 5, max iteration = 100

Convergence reached.

==>**Centroids**
[-0.35, 0.09, 1.1, 0.58, -0.02, -0.21, -0.51, -0.75, -0.58, 0.71, 0.61, 0.25, -0.05, -0.34, -0.56, -0.4]
[-0.6, -0.47, -0.06, 0.35, 0.6, 0.47, 0.23, -0.01, -0.23, -0.25, 0.04, 0.25, 0.25, 0.1, 0.05, -0.2]
[-0.91, -0.94, -1.08, -0.71, -0.28, 0.21, 0.78, 0.98, 0.86, -0.13, -0.36, -0.08, 0.13, 0.43, 0.62, 0.49]
[0.0, -0.47, -0.83, -0.72, -0.54, -0.35, -0.52, -0.48, 0.11, 0.96, 0.64, 0.38, 0.15, 0.1, 0.15, 0.51]
[0.17, 0.17, -0.2, -0.44, -0.42, -0.35, -0.05, 0.29, 0.42, 0.17, -0.07, -0.27, -0.25, -0.03, 0.11, 0.3]

==>**Jaccard Coefficient: 0.41**
==>**Rand Index: 0.8**

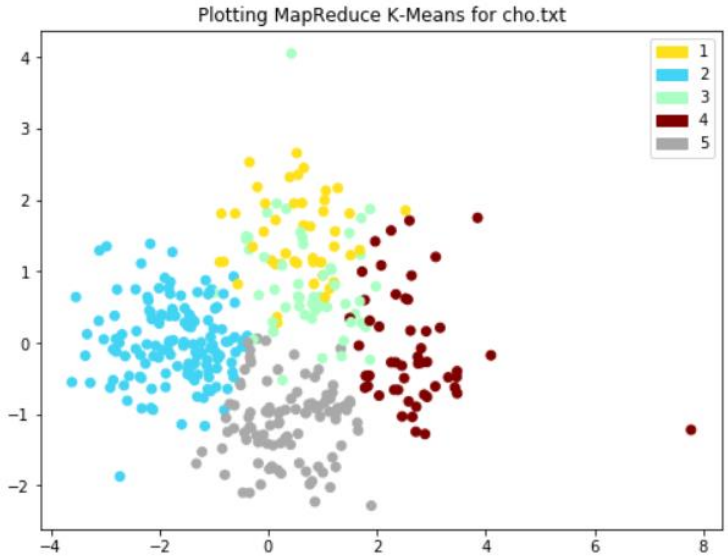


Fig: Plotting map reduce k-means for cho.txt

After multiple runs,

**Jaccard coefficient was in range:** **0.33 – 0.41**

**Rand Index was in range:** **0.78 – 0.8**

| **Iyer.txt** |
|---|

Number of cluster = 10, max iteration = 100

Convergence reached.

==>**Centroids**
[1.0, 0.89, 1.1, 1.1, 1.12, 1.67, 1.87, 2.32, 2.53, 2.03, 1.98, 1.89]
[1.0, 0.95, 0.94, 0.94, 0.9, 0.72, 0.6, 0.54, 0.52, 0.74, 0.73, 0.78]
[1.0, 0.96, 1.12, 1.45, 1.71, 3.93, 3.49, 2.92, 1.73, 1.36, 1.31, 1.25]
[1.0, 0.99, 1.15, 1.23, 1.5, 2.85, 4.39, 5.77, 4.57, 3.21, 3.03, 2.66]
[1.0, 1.02, 2.03, 3.52, 1.78, 10.2, 14.35, 15.17, 10.08, 9.86, 8.97, 7.7]
[1.0, 1.09, 0.91, 0.96, 1.0, 0.94, 0.93, 0.99, 1.19, 1.84, 2.32, 2.92]
[1.0, 1.17, 1.3, 1.15, 1.22, 1.08, 1.15, 1.23, 1.54, 2.97, 4.51, 6.7]
[1.0, 1.37, 2.58, 3.89, 2.94, 2.12, 2.01, 1.84, 1.4, 1.1, 1.16, 1.2]
[1.0, 2.01, 4.25, 7.39, 5.47, 4.3, 3.61, 3.63, 1.92, 1.63, 1.52, 1.34]
[1.0, 2.67, 2.41, 3.75, 6.18, 38.8, 86.92, 25.58, 14.81, 6.73, 2.72, 2.42]

==>**Jaccard Coefficient: 0.32**
==>**Rand Index: 0.72**

After multiple runs,

**Jaccard coefficient was in range:** **0.3 – 0.4**

**Rand Index was in range:** **0.71 – 0.83**

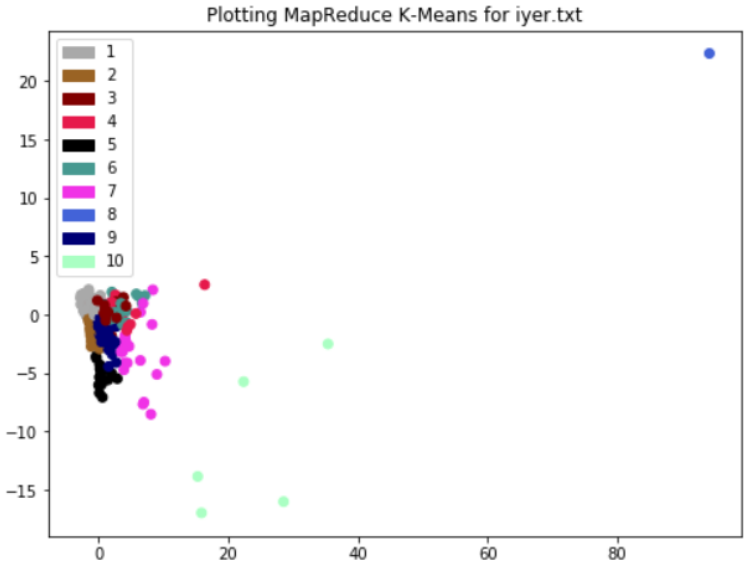


Fig: Plotting map reduce k-means for iyer.txt

**Difference in execution of parallel and non-parallel instance**:

It has been observed that although map reduce runs in parallel it is slower to execute for the given dataset than its non-parallel instance.

Maybe it is due to the heavy map and reduce framework that causes lots of overhead.

Parallel execution might improve if the dataset is too huge with which the overhead of implementing map reduce might seem negligible.

## Conclusion:

K-means can be an efficient algorithm to find clusters but is sensitive to initialization of parameters.

Hierarchical clustering on the other hand requires no initial input for number of centroids.

Both above methods are sensitive to noise and outliers.

Density based clustering can determine noise clusters but is sensitive to selection of initial parameters.

Map Reduce K-Means yields same results as its non-parallel instance.

Map reduce actually slows down the process if used on a small dataset.