University at Buffalo

Project 2

Clustering Algorithms

Team

Jay Bakshi	5020 6954	jaybaksh
Shivam Gupta	5020 6323	sgupta33
Debanjan Paul	5020 8716	dpaul7

1. K-Means Algorithm

1.1 Theory

K-Means is one of the most popular "clustering" algorithms. K-means stores 'K' centroids that it uses to define clusters. Points in the dataset are assigned to a cluster by considering its 'closeness' from a centroid compared to other to other centroids.

K-means involves 3 main steps:

a) K Centroid Initialization
b) Assignment
c) Update centroid

First, we initialize 'k' centroids by randomly choosing any points in the dataset. Second step is the assignment of points to the clusters by comparing their distances from the centroids. During assignment we compare the distance of a point from the K centroids. We assign the point to a specific cluster by selecting the min distance centroid from the previous step.

Next, we iterate through the clusters and perform a *geometric mean* of the points in a cluster. Update the 'k' centroids based on the obtained *mean* result. Compare the euclidean distance of the updated centroids from the old centroids. If the distance is zero that means we do not have to shift the centroids any more and we have obtained the final cluster set. Otherwise, repeat the step (b) and (c).

1.2 Algorithm

```
1: Initialize number of clusters and max number of iterations.

2: function fit(input):

3:    filename = input

4:    old_centroids =  grid(k, attributes in text file)

5:    initialize centroids : getCentroids()

6:    labelData = grid(empty column for cluster number, input)

7:    loop(not shouldStop()):

8:          old_centroids = copy(centroids)

9:          labels = getLabels()

10:         centroids = getCentroids()
```

```
11:       iterations++;
12:  end loop;
13: return;

14: function getCentroids():
15:  if iterations == 0:
16:       initialize centroids with random integer values
17:  else:
18:       for each i in labelData[rows][cluster_i]:
19:            updated centroids = mean(labelData[rows][i])
20:       end loop;
21: return;

22: function getLabels():
23:  foreach i in input[row_i]:
24:       distance =  euclidean distance(i, centroids)
25:       cluster[row_i] = min(distance)
26:       update labelData[row_i][0] = cluster[row_i]
27:  end loop;
28: return labelData;

29: function shouldStop();
30:  check =  if (iterations<maxiterations) AND (distance between
     centroids and old_centroid is 0)
31: return check;
```

1.3 Implementation

Our implementation of the k-means requires a text file and the number of clusters (k) as the input. When the object of the k-means class is instantiated, we define the metrics order and max number of iterations for the algorithm. Later, when the data text file is inputted, k centroids gets initialized with some random values. We'll also initialize a matrix *labelData* of n*(n+1) size, which will store the coordinates of each point with their cluster value in a corresponding column.

Next, we perform the assignment of the data points to the clusters. We compute the distance of each point from k centroids and select the closest centroid i.e. a centroid which is at the min distance from the given point. Update the labelData matrix with the newly assigned cluster for given data point.

Next, we shift the centroids by computing the geometric mean of the points present in the same group/cluster. Compare the updated centroid with the old centroid. Check if the euclidean distance between the two is zero or if we have reached the maximum number of iterations. If any of the conditions matches then we have obtained the final clusters, otherwise repeat the assignment and updation step iteratively.

Once, we obtain the final clusters, we'll compute the Jaccard coefficient and randIndex of the clusters. Jaccard index and randIndex will provide similarity between the clusters as defined by the size of the intersection divided by the size of the union of the sample set.

1.4 Output

- Jaccard coefficients:
    a. new_dataset_1.txt : (0.442 - 0.561)
    b. cho.txt : (0.295 - 0.39)
    c. iyer.txt : (0.27 - 0.34)

- Pros of K-Means:
    a. Very efficient (even if multiple runs are performed).
    b. Can be used for a large variety of data types.

- Cons of K-Means:
    a. Not suitable for all types of data. Eg. irregular shapes and varying density data.
    b. Susceptible to initialization problems and outliers, restricted to data in which there is a notion of a center.

2 Hierarchical Agglomerative Clustering Algorithm

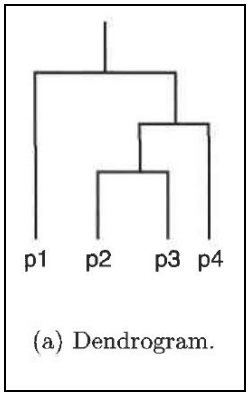
2.1 Theory

Hierarchical clustering techniques are simple and important category for clustering methods. They are also pretty popular and enjoy widespread use till date. There are 2 approaches for the same :

Agglomerative and Divisive. For this project our focus will be on Agglomerative approach only.

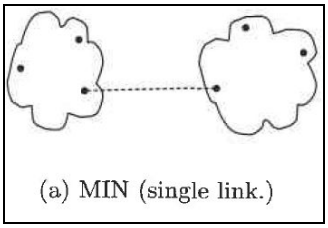
We start with the points as individual clusters and, at each step, merge the closest pair of clusters. This requires us defining cluster proximity.

Also, in general a hierarchical clustering is often displayed graphically using a tree-like diagram called a **dendrogram**, which displays both the cluster-subcluster relationships and the order in which the clusters were merged (for our case of agglomerative view).



(a) Dendrogram.

The key operation in this algorithm is computing the proximity matrix, which can be done using different techniques - MIN, MAX and Group Average. For our project we are focussing on **MIN** only - defined as cluster proximity as the proximity between the closest 2 points that are in different clusters, or using graph terms, the shortest edge between two nodes in different subsets of nodes. It is also known as **Single Link** approach.


(a) MIN (single link.)

The total space complexity is $O(m^2)$. Without modification the time complexity is of $O(m^3)$. With keeping data sorted in a list or heap, overall time complexity is $O(m^2 \log$

m). This complexity can severely limit the size of datasets that can be processed with this algorithm.

2.2 Algorithm

The basic algorithm as described in the reference book Introduction to Data Mining is written below :

```
1: Computer the proximity matrix, if necessary.
2: repeat
3:    Merge the closest two clusters.
4:    Update the proximity matrix to reflect the proximity between
the cluster and the original clusters.
```

2.3 Implementation

Our code implementation requires the dataset in a text file format (.txt). The number of clusters to stop at will be extracted from the given ground truth clusters. If not provided with the same, the execution of algorithm will result in final single cluster with information about the derived hierarchy in it to be extracted as and when required. We proceed with defining the following functions -

I. distance_matrix_cal : takes the input data in an m x n array and returns a m x m array. We're using cdist() from scipy.spatial.distance to calculate the distance matrix using Euclidean distance.
II. find_min : takes m x m distance matrix and a 3 x 1 list storing min_dist and respective indexes. The function scans the lower triangle of the diagonal distance matrix to find the minimum distance between 2 points and records the same in min_dist list and returns the same.
III. update_mat : takes m x m array as distance matrix updist_mat, 3 x 1 list min_dist and a list of list upid. This function is responsible for updating the proximity matrix to reflect the proximity of the newly formed cluster with the original cluster. This function returns a smaller m-1 x m-1 distance matrix, updated min_dist and upid.
IV. listflatten : takes the upid, which is a list of list and flattens the whole n x d list into a n x 2 list. This output of this can be saved and recalled for outputting the final clusters.
V. class METRICS : this is responsible for calculating the Jaccard coefficient and Rand Index.
VI. hexColor : this function is a utility defined to assign colors randomly in the scatter plot.

2.4 Output

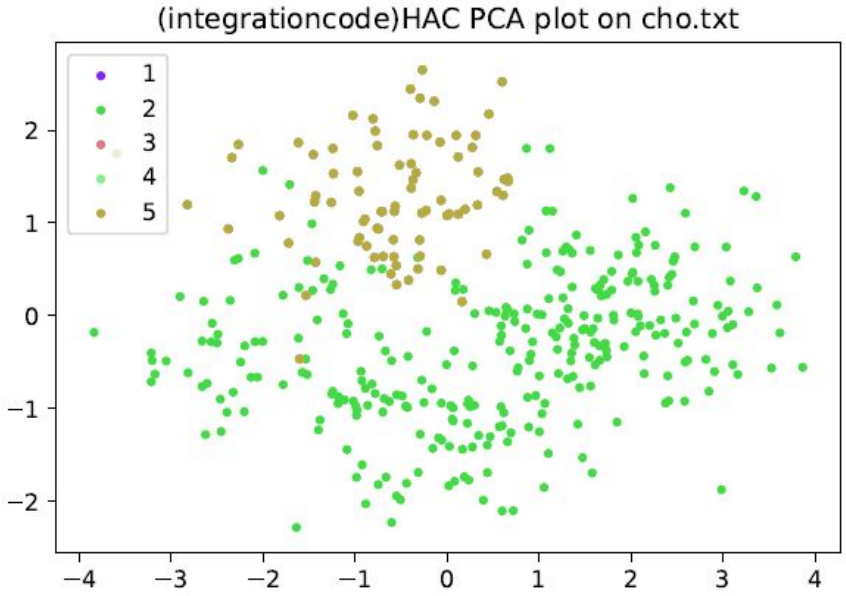
	Jaccard Coefficient	Rand Index
cho.txt	0.26706	0.48562
iyer.txt	0.15846	0.19413
new_dataset_2.txt	1.o	1.0

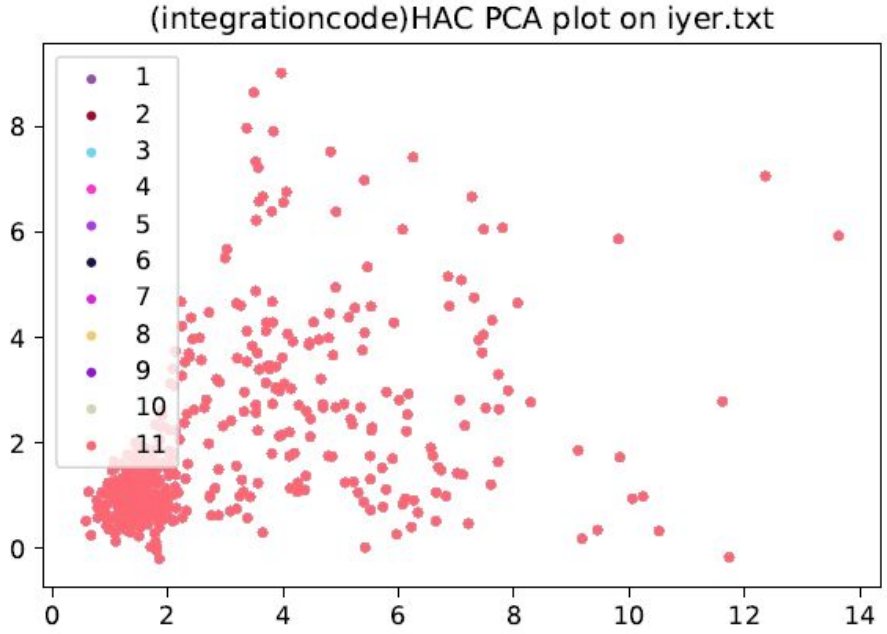
Pros of Hierarchical Agglomerative Clustering :

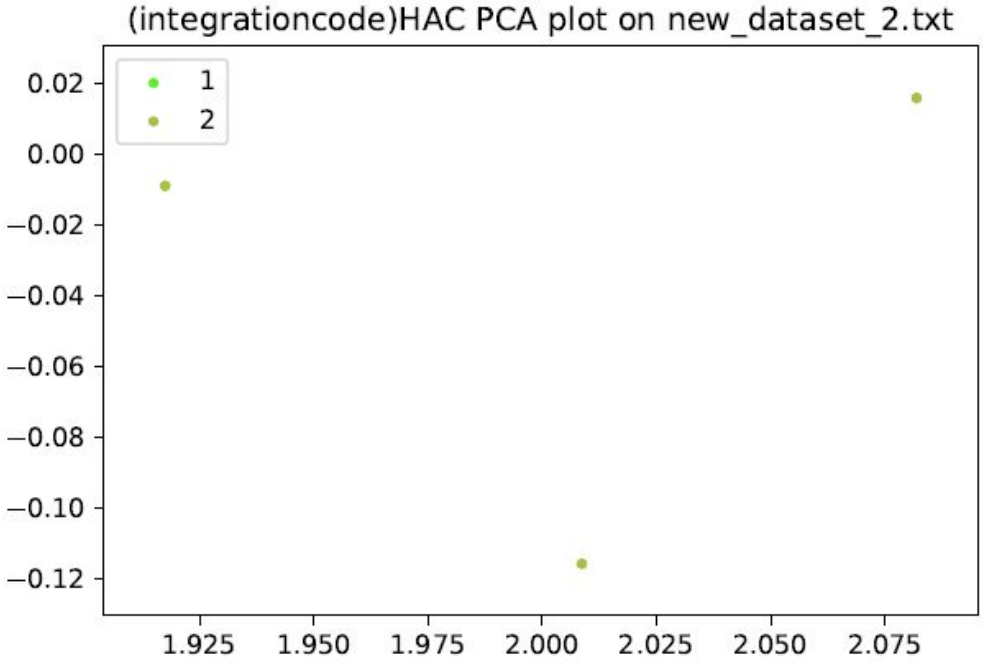
A. They are very well developed ideas and have been in use for years over a varied types of data.
B. There is no need to specify the number of clusters required since the ultimate goal of this clustering is to derive and present a hierarchy of all the clusters also singleton ones.
C. They can correspond to meaningful taxonomies across industries.

Cons of Hierarchical Agglomerative Clustering :

A. The time complexity as discussed is quadratic $O(n^2)$, so hierarchical clustering can be impractical on very big datasets say e.g. 1 million or more records.
B. Our metric for clustering, MIN is susceptible to noise.
C. Once a decision is made to combine 2 clusters, it cannot be undone.
D. No objective function is directly minimized.




(integrationcode)HAC PCA plot on iyer.txt
1
2
3
4
5
6
7
8
9
10
11
0
2
4
6
8
10
12
14


(integrationcode)HAC PCA plot on new_dataset_2.txt
1
2
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
1.925
1.950
1.975
2.000
2.025
2.050
2.075

3 Density based Clustering (DBSCAN)

3.1 Theory

Density-based spatial clustering of applications with noise (DBSCAN) is a data clustering algorithm proposed by Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu in 1996. It is a density-based clustering algorithm: given a set of points in some space, it groups together points that are closely packed together (points with many nearby neighbors), marking as outlier points that lie alone in low-density regions (whose nearest neighbors are too far away). [1]

3.1.1 Basic Idea

3.2 Algorithm

```
1: DBSCAN (D, eps, MinPts)
2:    C = 0
3:    for each unvisited point P in dataset D
4:          mark P as visited
5:          NeighborPts = regionQuery(P, eps)
6:          if sizeof(NeighborPts) < MinPts
7:                mark P as NOISE
8:          else
9:                C = next cluster
10:               expandCluster(P, NeighborPts, C, eps, MinPts)

11: expandCluster (P, NeighborPts, C, eps, MinPts)
12:   add P to cluster C
13:   for each point P' in NeighborPts
14:         if P' is not visited
15:               mark P' as visited
16:               NeighborPts' = regionQuery(P', eps)
17:               if sizeof(NeighborPts') >= MinPts
18:                  NeighborPts = NeighborPts joined with
NeighborPts'
19:         if P' is not yet member of any cluster
20:               add P' to cluster C

21:   regionQuery (P, eps)
22:         return all points within P's eps-neighborhood (including
P)
```

3.3 Implementation

The DBSCAN object takes in minimum points and epsilon value as input when creating the object model. Once this is done, any dataset formatted for the input of the object can be fit into it, using the *fit(<dataset>)* method.

The model scans for points starting from any arbitrary point, considering the radius given by *eps*. If there are more than minPts number of items in the vicinity of any point, it is labelled as a core point, otherwise, those points are assigned as border points.

The model also assigns noise points as core/border points at a later stage, if the point is seen in the vicinity of another core point.

The model is also capable of using different metric orders to calculate Manhattan, Euclidean, or Minkowski distance.

3.5 Output

4. MapReduce