CS 221 Project Final Report

Jayant Thatte, Jacob Van Gogh, Ye Yuan

Dec. 10, 2014

1. Problem Description:

Many social networking sites, such as Facebook, Google and etc., have implement social circles to allow users organize their friend lists manually. The scope of this project is to determine users' social circles by studying their profiles and connection with other users. The program should learn based on ground-truth (circles manually specified by users). Such learning will contribute to the process of grouping friends to circles for users whose circles are not known. The performance of the clustering approach is evaluated using the edit distance between the produced output and the correct solution.

In the base line, we had a total training set error of 17165, which is, roughly an average of 286 errors per ego. The error rate is about 103%. The output produced by the algorithm is evaluated by computing the edit distance of the produced output with the ground truth. The edit distance is then scaled divided by the sum of the sizes of ego networks of all users in the data used for testing. This normalizes the edit distance into some sort of an error rate and thus removes the effect of the size of the test data.

2. Model

The user whose friends are being classified into circles is known as the ego. The ego's friend network (egonet) is represented as an undirected graph. The ego's friends are nodes and edges represent two nodes being friends with each other. Users' profile information is encapsulated in a feature vector. The dataset we were given anonymized the values of these features such that they are integers. The profile information being looked at is as follows.

```
Features:
birthday
education;classes;description
education;classes;from;id
…
```

Please find the complete list of features in Appendix I.

3. Literature:

Over the course of our project, we found many different ways of doing graph clustering in the literature. We analyzed the properties and implementation complexity of the algorithms we came across to determine which to implement and try and improve upon. The algorithms considered are listed below.

Mixed Membership Stochastic Blocks (Airoldi et al 2008):

This is a probabilistic model which assigns a probability value to each pair of nodes in a graph sharing membership. This algorithm makes use of graph structure information and allows for intersecting communities, but requires the number of communities to be specified beforehand.

K-means Clustering (Hartigan et all 1979):

This algorithm forms clusters based on distances. K-means is generally easy to implement, but a function would have to be made in order to return distance values from a user's profile information. This algorithm also does not allow for intersecting clusters and requires the number of clusters to be specified beforehand

Clique Percolation (Derényi et all 2005):

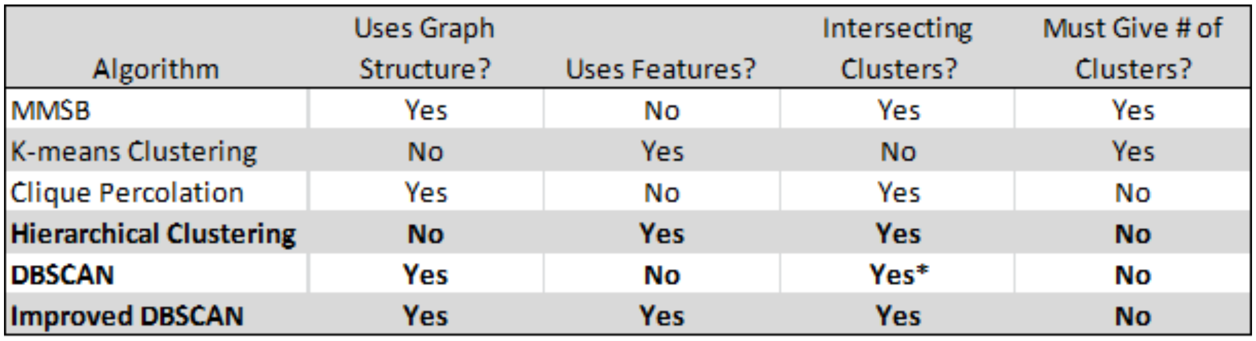
Clique percolation begins with many small, densely connected communities and builds larger communities from these. In this way, the algorithm is able to form clusters that intersect and does not need to be given a cluster number to achieve. It forms these clusters using solely the graph structure information.

Hierarchical Clustering (Johnson 1967):

Hierarchical clustering begins with each node is a singleton cluster. Clusters of size 2 are then formed by combining singleton clusters that pass a given similarity threshold. Larger clusters are formed iteratively using the same process. This algorithm makes use of only profile information and also restricts the types of clusters that can be formed. The nodes are sorted and clusters may only span nodes that are found consecutively in order.

DBSCAN (Birant et all 2007):

DBSCAN forms clusters in a graph by analyzing the local densities of nodes. In this way, it is able to form clusters. However, the clusters may not intersect.



Algorithm	Uses Graph Structure?	Uses Features?	Intersecting Clusters?	Must Give # of Clusters?
MMSB	Yes	No	Yes	Yes
K-means Clustering	No	Yes	No	Yes
Clique Percolation	Yes	No	Yes	No
Hierarchical Clustering	**No**	**Yes**	**Yes**	**No**
DBSCAN	**Yes**	**No**	**Yes***	**No**
Improved DBSCAN	**Yes**	**Yes**	**Yes**	**No**

Table 1: Algorithm property comparison. The bolded algorithms correspond to those implemented in this project. Note that the traditional implementation of DBSCAN does not allow for intersecting clusters and that this is changed in this project.

From these algorithms, we chose to implement hierarchical clustering and DBSCAN. These two algorithms had good compromises of clusters properties and implementation complexity. However, we altered hierarchical clustering to remove the limitation of consecutive clusters only. Additionally, we altered DBSCAN to allow for cluster intersection. We then combined the ideas learned from these two algorithms to create "improved DBSCAN".

4. Algorithms

There are two potentials we can draw from the data to help constructing the circles: features in user profiles and the connections between the users. Corresponding to each, we derived two approaches from literature to accommodate our problem, hierarchical clustering and Density-based spatial clustering of applications with noise (DBSCAN).

4.1 Hierarchical clustering

Hierarchical clustering is an algorithm that allows us to form circles based on users' features. This algorithm is adapted from that proposed by Johnson (Johnson. "Hierarchical Clustering Schemes". *Psychometrika*). In this algorithm, levels of clusters are formed and then the clusters are analyzed and circles are chosen from them.

Initialize the bottom level to be singleton clusters of all of the users in the egonet. To form the next level, for every cluster in the current level, consider adding each user individually to it. If the new cluster's similarity value (described later) is above the minimum threshold, keep the cluster. Continue doing this for the levels until a level is empty. Once all of the clusters have been created, analyze all of their similarity values again. If a cluster's similarity lies within a pre-determined range, make that cluster a circle.

We define the similarity of a cluster to be the number of features whose value is common across all nodes in the cluster.

Concrete example for the hierarchical clustering analysis:

Example: Consider a simple example with 4 users and 4 features (USER: FEATURE VECTOR)

A: [1, 2, 1, 3]
B: [1, 2, 1, 3]
C: [1, 2, 1, 4]
D: [2, 3, 4, 4]
Similarity range: [2, 3]

The first level of clusters is initialized to be singleton clusters.
Level 0: [(A), (B), (C), (D)]

Consider adding users to the cluster (A). B and A share 4 features in common, A and C 3, and A nd D 0. Thus, (A, B) and (A, C) are added to the next level. B and C share 3 features, so this cluster is added as well. C and D share one feature, but this is below the minimum of the range, so this cluster is not added.

Level 0: [(A), (B), (C), (D)]
Level 1: [(A, B), (A, C), (B, C)]

Adding C to (A, B) gives a cluster with a similarity value of 3, so this cluster is added to the next level. Adding D to the other two clusters gives clusters with similarity values that are too low.

Level 0: [(A), (B), (C), (D)]
Level 1: [(A, B), (A, C), (B, C)]
Level 2: [(A, B, C)]

Adding D to this last cluster does not give a viable cluster. The next level is empty, so the loop terminates. Out of all of these clusters, we choose those with similarity values ranging from 2 to 3 (Singleton clusters are not considered during this process). Therefore, our final circles are:

Circle 1: (A, C). Circle 2: (B, C). Circle 3: (A, B, C).

4.2 DBSCAN

The DBSCAN algorithm we utilized in this project is derived from an original method presented by Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu in proceedings of seconnd International Conference on Knowledge Discovery and Data Mining.[1] This algorithm forms circlers based on the density around each node. It tides nodes very connected to each other into one circle and leaves out the nodes with sparse connection with members in such circle. The way it determine when a node has a decent local density is to count whether it has more than certain number of nodes directly connect to it, defined with "neighbors". In our implementation, the algorithm iterates all unvisited nodes in one ego net and tries to form circles starting at nodes with decent density. It stops when the cluster cannot be expanded any further under current threshold of density, and then turns to the next unvisited node, starting a new circle if possible. The process for one ego is completed after no more unvisited nodes available in its ego network.

4.2.1 Concrete Example with Diagram

Please see Appendix II.

4.2.2 Pseudo code

```
densityClustering(ego, threshold):
friends, neighbours = getFriends(ego)
    for each unvisited point friend in friends
        mark P as visited
        NeighborPts = regionQuery(P, eps)
        if sizeof(neighbours[friend]) < threshold
            cluster = next cluster
            expandCluster(friend, neighbours, cluster, threshold,
unvisited)

expandCluster(friend, neighbours, cluster, threshold, unvisited)
    add friend to cluster
    for each point nbr in neighbours
        if nbr is not a member of cluster
            add nbr to cluster
        if nbr is visited:
             continue
        mark nbr as visited
        if len(neighbours) >= threshold
```

[1] Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." International Conference on Knowledge Discovery and Data Mining

```
              neighbours = union(neighbours, nbr's neighbors)
```

4.3 Improved DBSCAN with customized distance function

Later, we improved DBSCAN, the algorithm in last section, with customized distance function. In original DBSCAN we implemented, the distance function gives one when the two nodes are directly connected, and gives zero otherwise. To accommodate our data with profiles of features for every user, we have our new distance function, which gives value one for a pair of nodes only if they are both graphically connected and have a decent number of similar features. To be specific, we compare the profiles of the two user nodes and count the number of the features they have in common. If the two user nodes are neighbors of each other (graphically connected) and the quantity of their common features is above the threshold (currently set as 6), then our distance function give the value of one.

4.3.1 Pseudo code

```
densityClustering(ego, threshold):
friends, neighbours = getFriends(ego)
    for each unvisited point friend in friends
        mark P as visited
        NeighborPts = regionQuery(P, eps)
        if sizeof(neighbours[friend]) < threshold
            cluster = next cluster
            expandCluster(friend, neighbours, cluster, threshold,
unvisited)

expandCluster(friend, neighbours, cluster, threshold, unvisited)
    add friend to cluster
    for each point nbr in neighbours
        if nbr is not a member of cluster
            add nbr to cluster
        if nbr is visited:
            continue
        mark nbr as visited
        if len(neighbours) >= threshold and
numOfCommonFeatures(friend, nbr) >= featureThreshold
            neighbours = union(neighbours, nbr's neighbors)
```

4.4 Improved DBSCAN with customized distance function and weighted features

In the "improved DBSCAN with customized distance function" in section 3.3, all the features are weighted the same when contributing to the distance function. However, some features are more likely to influence the circles than the others. For example, user nodes with the same last names are possibly family members, so there is a big chance that they are in one circle. Other significant features can be working/studying under the same advisors, studying the same class in the same school, working in the same project in the same company and etc.

For the features that are significantly affecting the formation of circles, we want to give them bigger weights. Thus, we implement a getScore function, which takes the common features and return a weighted score. Thus, instead of using the plain count of the common features in the last section, we have a more advanced estimation of feature similarities.

4.4.1 Pseudo code

```
densityClustering(ego, threshold):
friends, neighbours = getFriends(ego)
    for each unvisited point friend in friends
        mark P as visited
        NeighborPts = regionQuery(P, eps)
        if sizeof(neighbours[friend]) < threshold
            cluster = next cluster
            expandCluster(friend, neighbours, cluster, threshold,
unvisited)

expandCluster(friend, neighbours, cluster, threshold, unvisited)
    add friend to cluster
    for each point nbr in neighbours
        if nbr is not a member of cluster
            add nbr to cluster
        if nbr is visited:
            continue
        mark nbr as visited
        if len(neighbours) >= threshold and
getScore(commonFeatures(friend, nbr)) >= featureThreshold
            neighbours = union(neighbours, nbr's neighbors)
```

5. Error Analysis

Our algorithm is evaluated with testing data and a metric calculating the edit distance. Both "add" and "remove" edits contribute to the edit distance.

In the evaluation process, the top ranking solution has an edit distance of 19247. The test data consists of 60 egos. The sum of all nodes in ego networks of all of these 60 egos yields 14519 nodes.

Compared to the error rate (edit distance) in baseline, 17265, the output in Hierarchical Clustering algorithm has a higher error, an edit distance of 17646. The output in DBSCAN has even higher error, an edit distance of 19247. Actually, even an empty result (containing no circles at all) has an edit distance of 17101, which is lower than the result from DBSCAN.

The improved DBSCAN algorithms perform much better. The DBSCAN with customized distance function has an edit distance of 17086. The number of elements in the ground truth which our algorithm didn't put them in the circles (number of "add" edits) is 17015. The number of elements that aren't in the ground truth which our algorithm mistakenly includes (number of "remove" edits) is 71.

The DBSCAN with customized distance function and weighted features has an edit distance of 17086. The number of elements in the ground truth which our algorithm didn't put them in the circles (number of "add" edits) is 17079. The number of elements that aren't in the ground truth which our algorithm mistakenly includes (number of "remove" edits) is 7.

Appendix I:

Features:
birthday
education;classes;description
education;classes;from;id
education;classes;from;name
education;classes;id
education;classes;name
education;classes;with;id
education;classes;with;name
education;concentration;id
education;concentration;name
education;degree;id
education;degree;name
education;school;id
education;school;name
education;type
education;with;id
education;with;name
education;year;id
education;year;name
first_name
gender
hometown;id
hometown;name
id
languages;id
languages;name
last_name
locale
location
location;id
location;name
middle_name
name
political
religion
work;description
work;employer;id
work;employer;name
work;end_date
work;from;id
work;from;name
work;location;id
work;location;name
work;position;id
work;position;name
work;projects;description
work;projects;end_date
work;projects;from;id
work;projects;from;name
work;projects;id
work;projects;name
work;projects;start_date
work;projects;with;id
work;projects;with;name
work;start_date
work;with;id
work;with;name

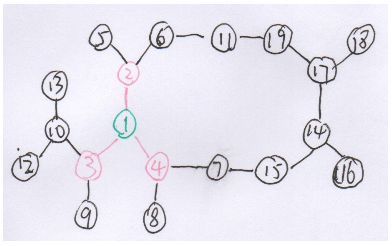
Appendix II

A concrete example for DBSCAN algorithm:

One concrete example for our DBSCAN algorithm can be: We have an ego, whose ID is 0, with a list of friends in his/her ego net, whose ID's are: 1, 2, 3, ..., 19. We also acknowledge their friends in the scope of this ego net, which in terms of the graph structure, their neighbors.

Friend	List of his/her
1	[2. 3. 4]
2	[1. 5. 6]
3	[1. 9. 10]
4	[1. 7. 8]
5	[2]
6	[2. 11]
7	[4. 15]
8	[4]
9	[3]
10	[3. 12. 13]
11	[6. 19]
12	[10]
13	[10]
14	[15. 16. 17]
15	[7. 14]
16	[14]
17	[18. 19]
18	[17]
19	[11. 17]

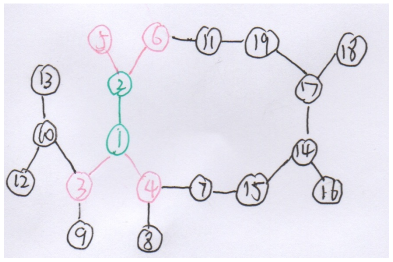
We keep a bitmap "unvisited" to track whether a node has been visited before. It is initialized with {1:1, 2:1, 3:1, 4:1, 5:1, 6:1, 7:1, 8:1, 9:1, 10:1, 11:1, 12:1, 13:1, 14:1, 15:1, 16:1, 17:1, 18:1, 19:1}. It's implemented as a dictionary in our module. The value "1" stands for "unvisited" and "0" for "visited". We also keep variables such as clusters (list of clusters) and currNbr (a queue of neighbors that we traverse and meanwhile append new entries to in the process.



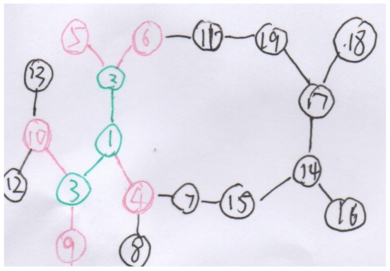
We start with any available unvisited node. In this case, we traverse the bitmap “unvisited” and find the first available unvisited node: 1. The threshold for local density is set at 3. We can see that node 1 has three neighbours, 2, 3 and 4. 1 is not a noise point, so we start a new cluster with 1 as its first member and then add 2, 3, and 4 to the currNbr. Now we have cluster as [1] and currNbr as [2, 3, 4]

The rest of process for this new cluster is to repeat the following in a loop

1. get a node from currNbr
2. add the node to cluster if it has at least three neighbours to the cluster and its neighbours to the currNbr.

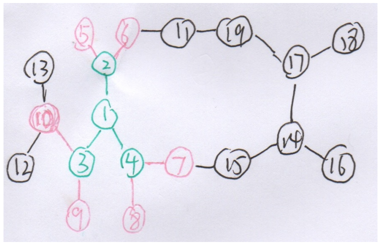


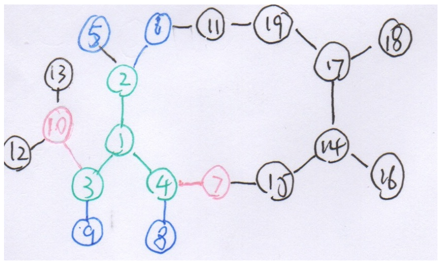
Thus, we get 2 from the currNbr, [2, 3, 4], and decide that since 2 is not in the cluster and 2 has three neighbours, 2 also goes to the cluster and we add 2’s neighbours, 1, 5 and 6 to currNbr. Now we have cluster as [1, 2] and currNbr as [3, 4, 1, 5, 6]



The step, we look at 3 from currNbr, add 3 to the cluster, which is [1, 2, 3] now, and append 3’s neighbours, 1, 9, and 10 to currNbr, which will be [4, 1, 5, 6, 1, 9, 10]

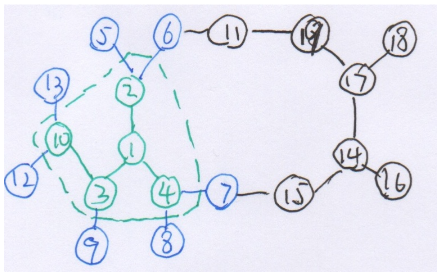
We continue the process for one more step, and get cluster = [1, 2, 3, 4] and currNbr = [1, 5, 6. 1, 9, 10, 7, 8]

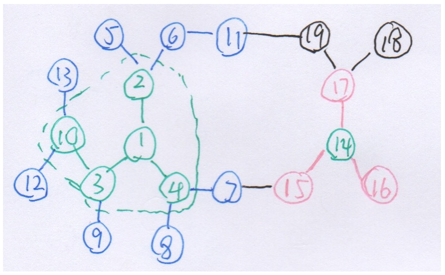




Then, if we get the first one in currNbr, we get 1, which is visited, so we won’t consider 1’s neighbors again. Also, 1 is already in the cluster, so we won’t 1 to the cluster again. Next, we hit 5 in the currNbr. Since 5 doesn’s have three neighbors, we won’t add 5 to the cluster or 5’s neighbors to the currNbr. We simply continue over 5. Same for 6 and 9. At this point, the cluster = [1, 2, 3, 4] and the currNbr = [10, 7, 8]

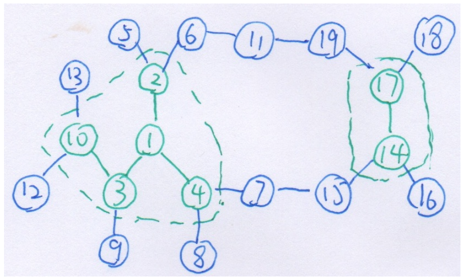
We continue this process until currNbr is empty, then we have cluster = [1, 2, 3, 4, 10]





At this point, we complete forming the first cluster! Store this cluster as the first entry of clusters, and continue with next available unvisited node, which is 11. 11 doesn't have three neighbors, so we continue and find 14 to start with.

Again, playing the very same rules, we get another cluster = [14, 17].



Thus, for ego 0, we have two circles: [[1, 2, 3, 4, 10], [14, 17]]. The module loops through all the egos in the data set.

For the clarity, one thing I mispresented in the example is that the variable currNbr. When we get candidate from currNbr, we don't remove/pop the entry from currNbr. We simply traverse it from left to right and new entries are appended to the right.