LAB 2 – COMMUNITY FINDING

COMP-47270 COMPUTATIONAL NETWORK ANALYSIS AND MODELLING

Author :	*Paula Dwan* *paula.dwan@gmail.com*
Student ID :	*13208660*
Lecturer :	*Dr. Neil Hurley* *neil.hurley@ucd.ie*
Due Date :	*Sunday, 19 April 2015*

Table of Contents

PRELIMINARIES

INSTALLING LIBRARIES –

Firstly :

```
✔ ~/_Private/MSc.ASE/Year2-Modules/Comp47270-
ComputationalNetworkAnalysisAndModelling/Labs_CaseStudies/Lab-2-Communities/Lab-2-SourceCode$ sudo
apt-get install -y graphviz libgraphviz-dev pkg-config python-pip
Reading package lists... Done
Building dependency tree
Reading state information... Done
python-pip is already the newest version.
graphviz is already the newest version.
The following packages were automatically installed and are no longer required:
  linux-headers-3.13.0-32 linux-headers-3.13.0-32-generic
  linux-headers-3.13.0-39 linux-headers-3.13.0-39-generic
  linux-image-3.13.0-32-generic linux-image-3.13.0-39-generic
  linux-image-extra-3.13.0-32-generic linux-image-extra-3.13.0-39-generic
Use 'apt-get autoremove' to remove them.
The following extra packages will be installed:
  autotools-dev libltdl-dev libtool libxdot4
Suggested packages:
  libtool-doc autoconf automaken gfortran fortran95-compiler gcj-jdk
The following NEW packages will be installed:
  autotools-dev libgraphviz-dev libltdl-dev libtool libxdot4 pkg-config
0 upgraded, 6 newly installed, 0 to remove and 22 not upgraded.
Need to get 511 kB of archives.
After this operation, 3,119 kB of additional disk space will be used.
Get:1 http://ie.archive.ubuntu.com/ubuntu/ trusty/main autotools-dev all 20130810.1 [44.3 kB]
Get:2 http://ie.archive.ubuntu.com/ubuntu/ trusty-updates/main libxdot4 amd64 2.36.0-0ubuntu3.1 [19.4
kB]
Get:3 http://ie.archive.ubuntu.com/ubuntu/ trusty/main libltdl-dev amd64 2.4.2-1.7ubuntu1 [157 kB]
Get:4 http://ie.archive.ubuntu.com/ubuntu/ trusty-updates/main libgraphviz-dev amd64 2.36.0-0ubuntu3.1
[60.9 kB]
Get:5 http://ie.archive.ubuntu.com/ubuntu/ trusty/main libtool amd64 2.4.2-1.7ubuntu1 [188 kB]
Get:6 http://ie.archive.ubuntu.com/ubuntu/ trusty/main pkg-config amd64 0.26-1ubuntu4 [40.9 kB]
Fetched 511 kB in 0s (670 kB/s)
Selecting previously unselected package autotools-dev.
(Reading database ... 273014 files and directories currently installed.)
Preparing to unpack .../autotools-dev_20130810.1_all.deb ...
Unpacking autotools-dev (20130810.1) ...
Selecting previously unselected package libxdot4.
Preparing to unpack .../libxdot4_2.36.0-0ubuntu3.1_amd64.deb ...
Unpacking libxdot4 (2.36.0-0ubuntu3.1) ...
Selecting previously unselected package libltdl-dev:amd64.
Preparing to unpack .../libltdl-dev_2.4.2-1.7ubuntu1_amd64.deb ...
Unpacking libltdl-dev:amd64 (2.4.2-1.7ubuntu1) ...
Selecting previously unselected package libgraphviz-dev.
Preparing to unpack .../libgraphviz-dev_2.36.0-0ubuntu3.1_amd64.deb ...
Unpacking libgraphviz-dev (2.36.0-0ubuntu3.1) ...
Selecting previously unselected package libtool.
Preparing to unpack .../libtool_2.4.2-1.7ubuntu1_amd64.deb ...
Unpacking libtool (2.4.2-1.7ubuntu1) ...
Selecting previously unselected package pkg-config.
Preparing to unpack .../pkg-config_0.26-1ubuntu4_amd64.deb ...
Unpacking pkg-config (0.26-1ubuntu4) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Setting up autotools-dev (20130810.1) ...
Setting up libxdot4 (2.36.0-0ubuntu3.1) ...
Setting up libltdl-dev:amd64 (2.4.2-1.7ubuntu1) ...
Setting up libgraphviz-dev (2.36.0-0ubuntu3.1) ...
Setting up libtool (2.4.2-1.7ubuntu1) ...
Setting up pkg-config (0.26-1ubuntu4) ...
Processing triggers for libc-bin (2.19-0ubuntu6.4) ...
✔ ~/_Private/MSc.ASE/Year2-Modules/Comp47270-
ComputationalNetworkAnalysisAndModelling/Labs_CaseStudies/Lab-2-Communities/Lab-2-SourceCode$
```

and then :

```
✔ ~/_Private/MSc.ASE/Year2-Modules/Comp47270-
ComputationalNetworkAnalysisAndModelling/Labs_CaseStudies/Lab-2-Communities/Lab-2-SourceCode$ sudo pip
install pygraphviz
Downloading/unpacking pygraphviz
```

```
  Downloading pygraphviz-1.2.tar.gz (90kB): 90kB downloaded
  Running setup.py (path:/tmp/pip_build_root/pygraphviz/setup.py) egg_info for package pygraphviz
    Trying pkg-config
    library_path=
    include_path=/usr/include/graphviz

    warning: no previously-included files matching '*~' found anywhere in distribution
    warning: no previously-included files matching '*.pyc' found anywhere in distribution
    warning: no previously-included files matching '.svn' found anywhere in distribution
    no previously-included directories found matching 'doc/build'
Installing collected packages: pygraphviz
  Running setup.py install for pygraphviz
    Trying pkg-config
    library_path=
    include_path=/usr/include/graphviz
    building 'pygraphviz._graphviz' extension
    x86_64-linux-gnu-gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-
prototypes -fPIC -I/usr/include/graphviz -I/usr/include/python2.7 -c pygraphviz/graphviz_wrap.c -o
build/temp.linux-x86_64-2.7/pygraphviz/graphviz_wrap.o
    pygraphviz/graphviz_wrap.c: In function ‘agattr_label’:
    pygraphviz/graphviz_wrap.c:2855:5: warning: return makes integer from pointer without a cast
[enabled by default]
         return agattr(g, kind, name, val);
         ^
    x86_64-linux-gnu-gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl,-Bsymbolic-functions
-Wl,-z,relro -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes
-D_FORTIFY_SOURCE=2 -g -fstack-protector --param=ssp-buffer-size=4 -Wformat -Werror=format-security
build/temp.linux-x86_64-2.7/pygraphviz/graphviz_wrap.o -lcgraph -lcdt -o build/lib.linux-x86_64-
2.7/pygraphviz/_graphviz.so

    warning: no previously-included files matching '*~' found anywhere in distribution
    warning: no previously-included files matching '*.pyc' found anywhere in distribution
    warning: no previously-included files matching '.svn' found anywhere in distribution
    no previously-included directories found matching 'doc/build'
Successfully installed pygraphviz
Cleaning up...
```

CODE PROVIDED

LAPLACIAN.PY

```
# import the networkx network analysis package
import networkx as nx

# import the graphvisualisation package graphviz
from networkx import graphviz_layout

import pygraphviz
# import the plotting functionality from matplotlib
import matplotlib.pyplot as plt

#import Delaunay tesselation
from scipy.spatial import Delaunay

# import kmeans
from scipy.cluster.vq import vq, kmeans, whiten

import numpy as np
import scipy as sp
import random

def placement():
    num_nodes = 100
    x = [random.random() for i in range(num_nodes)]
    y = [random.random() for i in range(num_nodes)]

    x = np.array(x)
    y = np.array(y)

    # Make a graph with num_nodes nodes and zero edges
    # Plot the nodes using x,y as the node positions

    G = nx.empty_graph(num_nodes)
    print "G.number_of_nodes() = ", G.number_of_nodes()

    pos = dict()
```

```
    for i in range(num_nodes):
        pos[i] = x[i],y[i]

    plot_graph(G, pos, 1)

    # Now add some edges - use Delaunay tesselation to produce a planar graph.
    # Delaunay tesselation covers the convex hull of a set of points with
    # triangular simplices (in 2D)
    #
    # Aside : Paula 13-Jan-2015
    #   planar graph - graph that can be plotted in 2-D with no overlaps.

    points = np.column_stack((x,y))
    dl = Delaunay(points)
    tri = dl.simplices

    edges = np.zeros((2, 6*len(tri)),dtype=int)
    data = np.ones(6*len(points))
    j=0
    for i in range(len(tri)):
        edges[0][j]=tri[i][0]
        edges[1][j]=tri[i][1]
        j = j+1
        edges[0][j]=tri[i][1]
        edges[1][j]=tri[i][0]
        j = j+1
        edges[0][j]=tri[i][0]
        edges[1][j]=tri[i][2];
        j = j+1
        edges[0][j]=tri[i][2]
        edges[1][j]=tri[i][0];
        j = j+1
        edges[0][j]=tri[i][1]
        edges[1][j]=tri[i][2]
        j=j+1
        edges[0][j]=tri[i][2]
        edges[1][j]=tri[i][1]
        j=j+1

    data=np.ones(6*len(tri))
    A = sp.sparse.csc_matrix((data,(edges[0,:],edges[1,:])))

    for i in range(A.nnz):
        A.data[i] = 1.0

    G = nx.to_networkx_graph(A)
    plot_graph(G,pos,2)

    # Use the eigenvectors of the normalised Laplacian to calculate placement positions
    # for the nodes in the graph

    #   eigen_pos holds the positions
    eigen_pos = dict()
    deg = A.sum(0)
    diags = np.array([0])
    D = sp.sparse.spdiags(deg,diags,A.shape[0],A.shape[1])        # diagonal matrix of degrees
    Dinv = sp.sparse.spdiags(1/deg,diags,A.shape[0],A.shape[1])   # inverse of
    # Normalised laplacian : multiply by 1 / Deg previously
    L = Dinv*(D - A)
    E, V = sp.sparse.linalg.eigs(L,3,None,100.0,'SM') # 100x100 matrix -> compress into 100 vector
    V = V.real

    for i in range(num_nodes):
        eigen_pos[i] = V[i,1].real,V[i,2].real

    # for n,nbrsdict in G.adjacency_iter():
    #     for nbr,eattr in nbrsdict.items():
    #         if 'weight' in eattr:
    #             print n,nbr,eattr['weight']

    plot_graph(G,eigen_pos,3)                       # call function

    # Now let's see if the eigenvectors are good for clustering
    # Use k-means to cluster the points in the vector V

    features = np.column_stack((V[:,1], V[:,2]))
    cluster_nodes(G,features,pos,eigen_pos)
```

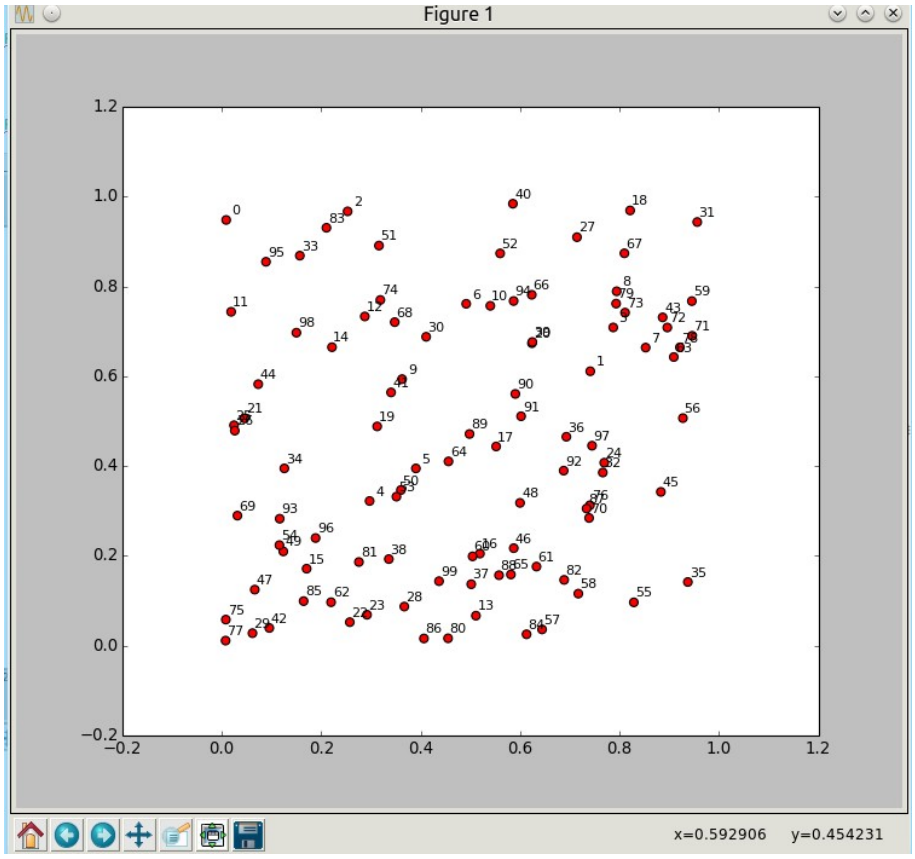
```
    # Finally, use the columns of A directly for clustering
    raw_input("Press Enter to Continue ...")

    cluster_nodes(G,A.todense(),pos,eigen_pos)      # call function


def plot_graph(G,pos,fignum):

    label = dict()
    labelpos=dict()
    for i in range(G.number_of_nodes()):
        label[i] = i
        labelpos[i] = pos[i][0]+0.02, pos[i][1]+0.02

    fig=plt.figure(fignum,figsize=(8,8))
    fig.clf()
    nx.draw_networkx_nodes(G,
                           pos,
                           node_size=40,
                           hold=False,
                       )

    nx.draw_networkx_edges(G,pos, hold=True)
    nx.draw_networkx_labels(G,
                           labelpos,
                           label,
                           font_size=10,
                           hold=True,
                       )
    fig.show()


def cluster_nodes(G, feat, pos, eigen_pos):
    book,distortion = kmeans(feat,3)
    codes,distortion = vq(feat, book)

    nodes = np.array(range(G.number_of_nodes()))
    W0 = nodes[codes==0].tolist()
    W1 = nodes[codes==1].tolist()
    W2 = nodes[codes==2].tolist()
    print "W0 = ", W0
    print "W1 = ", W1
    print "W2 = ", W2

    plt.figure(3)
    nx.draw_networkx_nodes(G,
                           eigen_pos,
                           node_size=40,
                           hold=True,
                           nodelist=W0,
                           node_color='m'
                       )
    nx.draw_networkx_nodes(G,
                           eigen_pos,
                           node_size=40,
                           hold=True,
                           nodelist=W1,
                           node_color='b'
                       )

    plt.figure(2)
    nx.draw_networkx_nodes(G,
                           pos,
                           node_size=40,
                           hold=True,
                           nodelist=W0,
                           node_color='m'
                       )
    nx.draw_networkx_nodes(G,
                           pos,
                           node_size=40,
                           hold=True,
                           nodelist=W1,
                           node_color='b'
                       )
```

```
if __name__ == '__main__':
    placement()
```

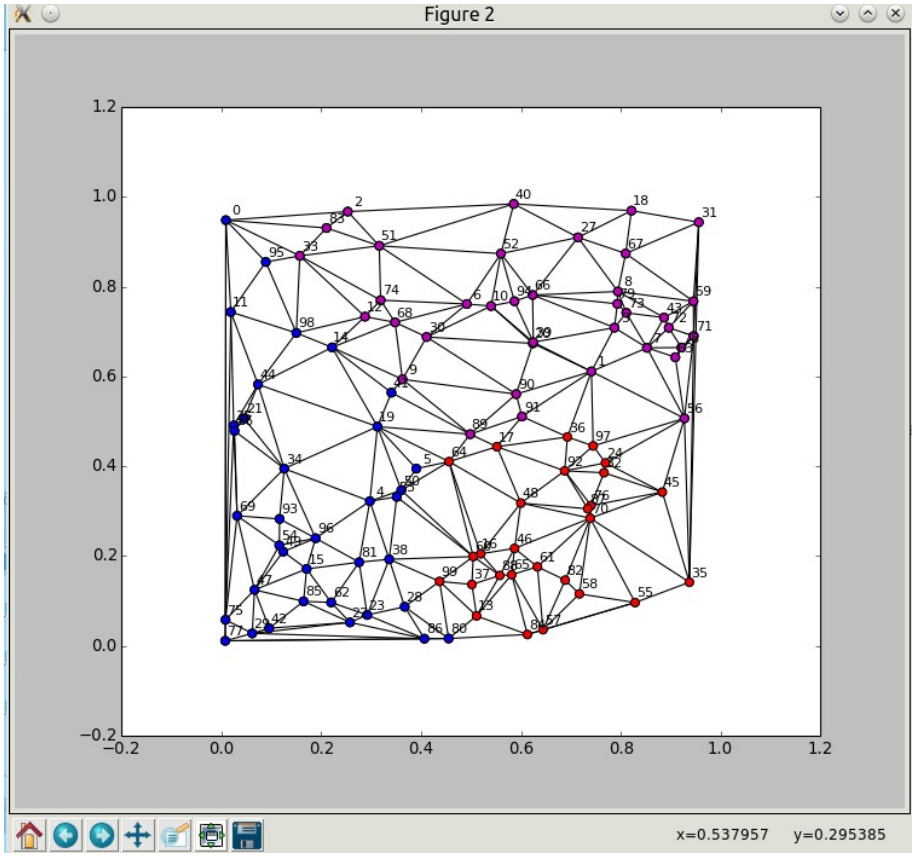
LAPLACIAN.PY – OUTPUT TO CONSOLE

```
G.number_of_nodes() =  100

cluster_nodes for e-vector values :-
W0 =     [1, 2, 3, 6, 7, 8, 9, 10, 12, 18, 20, 27, 30, 31, 33, 39, 40, 43, 51, 52, 56, 59, 63, 66, 67,
         68, 71, 72, 73, 74, 78, 79, 83, 89, 90, 91, 94]
W1 =     [0, 4, 5, 11, 14, 15, 19, 21, 22, 23, 25, 26, 28, 29, 34, 38, 41, 42, 44, 47, 49, 50, 53, 54,
         62, 69, 75, 77, 80, 81, 85, 86, 93, 95, 96, 98]
W2 =     [13, 16, 17, 24, 32, 35, 36, 37, 45, 46, 48, 55, 57, 58, 60, 61, 64, 65, 70, 76, 82, 84, 87,
         88, 92, 97, 99]
Press Enter to Continue ...

cluster_nodes for Delauney tessalation values :-
W0 =     [0, 2, 11, 15, 21, 22, 23, 25, 26, 29, 33, 34, 42, 44, 47, 49, 54, 62, 69, 75, 77, 81, 83, 85,
         86, 93, 95, 96, 98]
W1 =     [4, 5, 6, 9, 10, 12, 13, 14, 16, 17, 18, 19, 20, 24, 28, 30, 32, 35, 36, 37, 38, 40, 41, 45,
         46, 48, 50, 51, 52, 53, 55, 57, 58, 60, 61, 64, 65, 68, 70, 74, 76, 80, 82, 84, 87, 88, 89,
         90, 91, 92, 94, 97, 99]
W2 =     [1, 3, 7, 8, 27, 31, 39, 43, 56, 59, 63, 66, 67, 71, 72, 73, 78, 79]
Press Enter to Continue ...
```

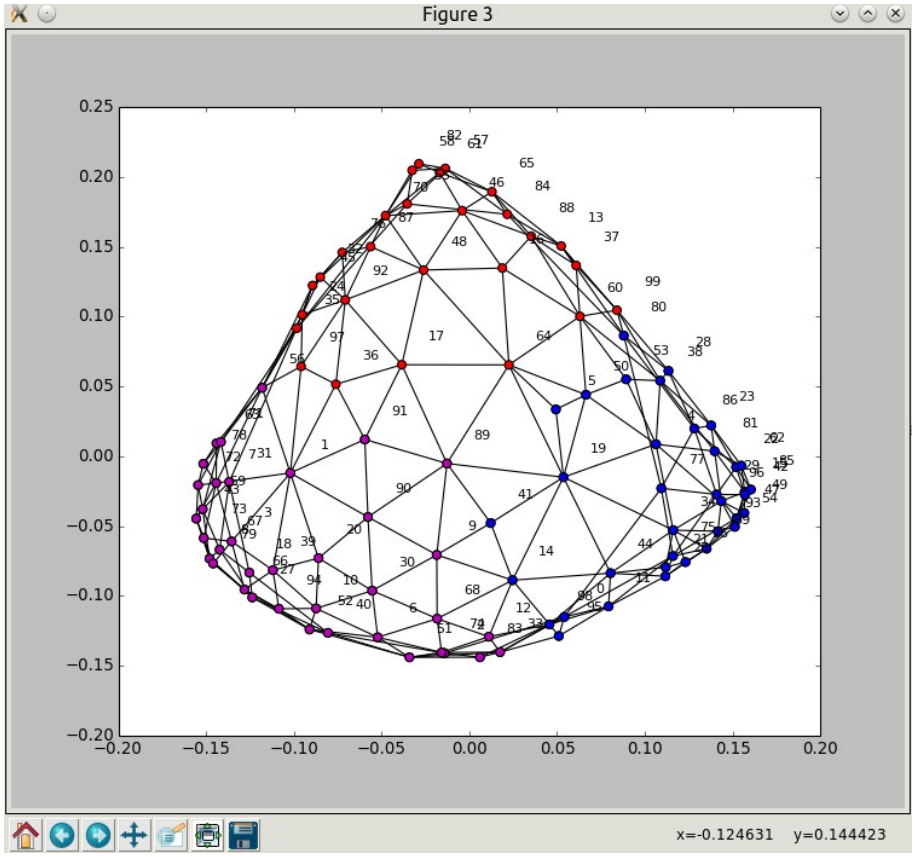
LAPLACIAN.PY – OUTPUT TO FIGURE (1) – E-VECTORS



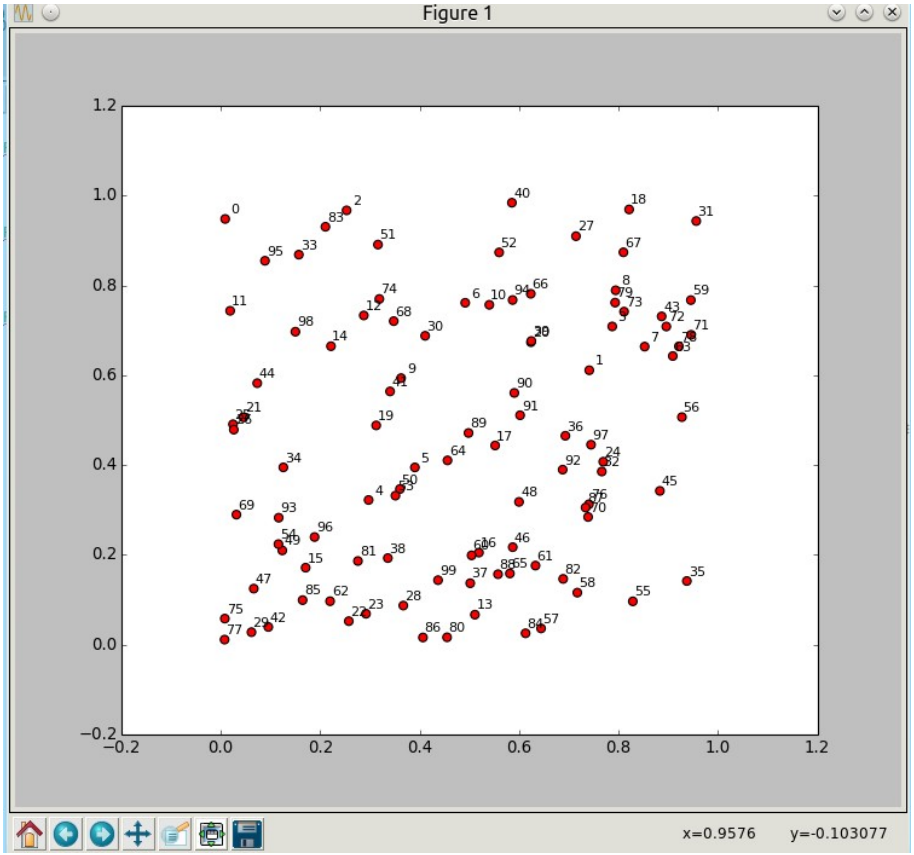
LAPLACIAN.PY – OUTPUT TO FIGURE (2) – E-VECTORS



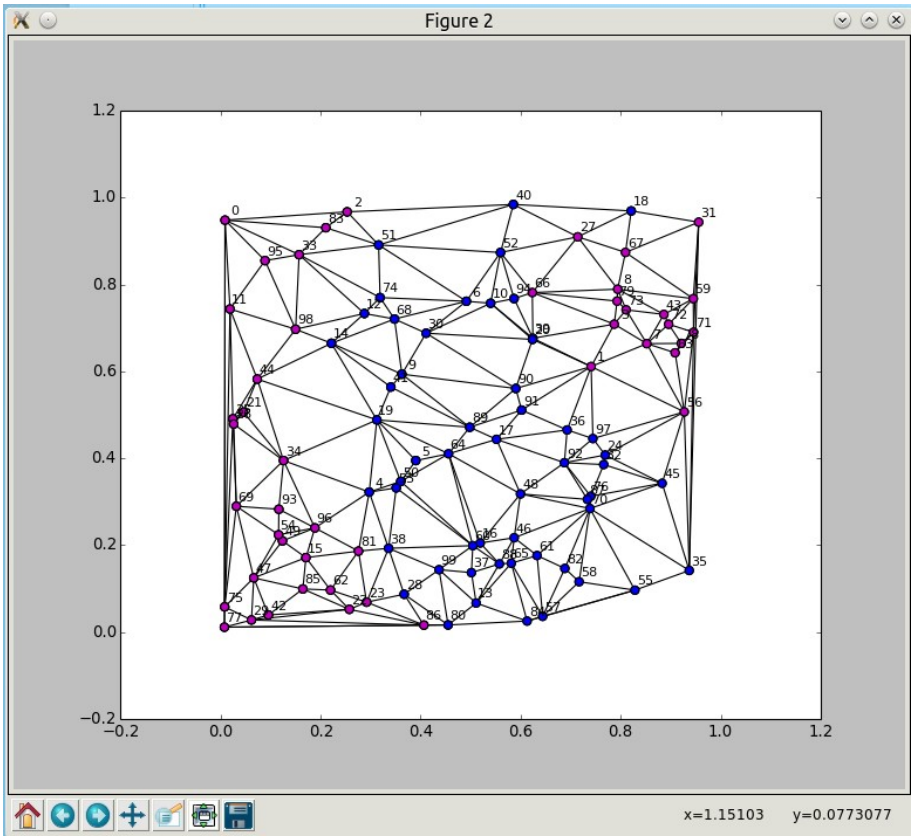
LAPLACIAN.PY – OUTPUT TO FIGURE (3) – E-VECTORS



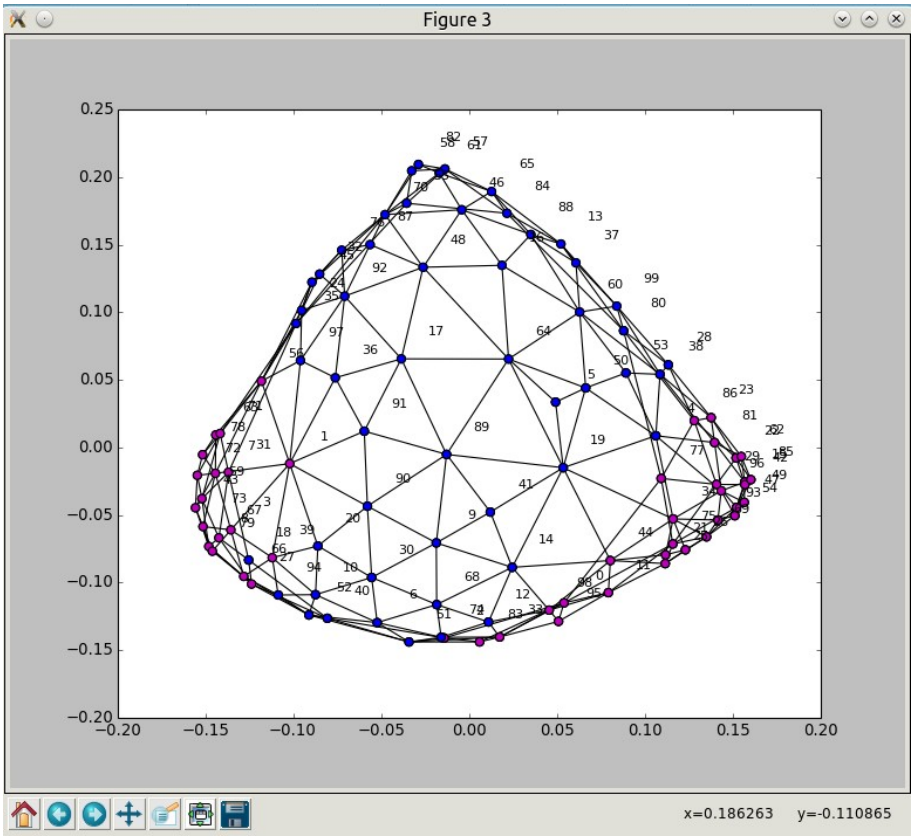
LAPLACIAN.PY – OUTPUT TO FIGURE (1) – DELAUNEY TESSELLATION



LAPLACIAN.PY – OUTPUT TO FIGURE (2) – DELAUNEY TESSELLATION



LAPLACIAN.PY – OUTPUT TO FIGURE (3) – DELAUNEY TESSELLATION



Laboratory 2 – Basic Requirements

- Write some functions to evaluate the quality of a community:
  1. An edge-cut function counts the total number of edges that are cut by a partitioning.
  2. The normalised mutual information function compares two community partitionings and determines how alike they are.
  3. Newman's modularity function computes how cohesive the communities in a community partitioning are.
- Using some of the SNAP graphs that you downloaded in Lab 1, apply the k-means clustering methods in ***laplacian.py*** to partition the graph into k ≥ 2 communities, where the k-means vectors are generated from eigenvectors and from rows of the adjacency matrix.
- Compute how similar the partitionings are by using NMI.
- Compute how good the partitionings are by using modularity and edge-cut.

Laboratory 2 – Case Study

Evaluate some community-finding methods on SNAP data and on simulated networks with embedded communities:

1. Implement your own community-finding algorithm.
2. Compare with at least two other algorithms.
3. Compute their relative performance in terms of quality (NMI, modularity) and run-time.

Community Finding Algorithms

1 : Louvain method – finding communities in large networks

Overview

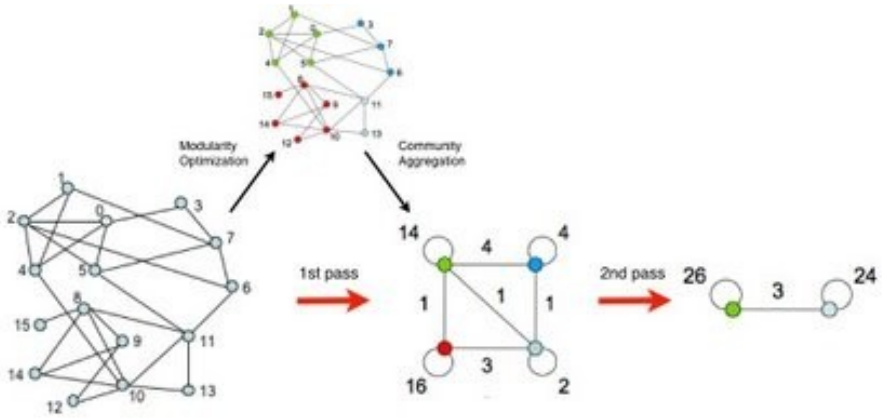
A Multi-Level Aggregation Method for optimizing modularity : in the last few years, there have been many attempts to uncover communities in large networks. By large, we mean systems composed of millions of nodes, which cannot be visualized nor analyzed at the level of single nodes and therefore require a coarse-grained description.

Our method, that we call Louvain Method (because, even though the co-authors now hold positions in Paris, London and Louvain, the method was devised when they all were in Louvain), outperforms other methods in terms of computation time, which allows us to analyze networks of unprecedented size (e.g. the analysis of a typical network of 2 million nodes only takes 2 minutes). The Louvain method has also been to shown to be very accurate by focusing on ad-hoc networks with known community structure. Moreover, due to its hierarchical structure, which is reminiscent of renormalization methods, it allows to look at communities at different resolutions.

The method consists of two phases.

1. First, it looks for "small" communities by optimizing modularity in a local way.
2. Second, it aggregates nodes of the same community and builds a new network whose nodes are the communities.

These steps are repeated iteratively until a maximum of modularity is attained.



The output of the program therefore gives several partitions. The partition found after the first step typically consists of many communities of small sizes. At subsequent steps, larger and larger communities are found due to the aggregation mechanism. This process naturally leads to hierarchical decomposition of the network.

This is obviously an approximate method and nothing ensures that the global maximum of modularity is attained, but several tests have confirmed that our algorithm has an excellent accuracy and often provides a decomposition in communities that has a modularity that is close to optimality.

The method was first published in : “Fast unfolding of communities in large networks” (http://arxiv.org/abs/0803.0476), Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre, Journal of Statistical Mechanics: Theory and Experiment 2008 (10), P1000

More information

Two C++ codes, the original (https://sites.google.com/site/findcommunities/Community_BGLL_CPP.zip?attredirects=0) and an updated (https://sites.google.com/site/findcommunities/newversion/community.tgz?attredirects=0) versions, are freely available for download. More information can be found in the readme file included in both distributions and also at http://www.inma.ucl.ac.be/~blondel/research/louvain.html. A preliminary matlab version can be obtained on demand.

Details about our method can be found http://arxiv.org/abs/0803.0476.

This project has been developed by Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre

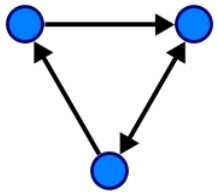
REFERENCES

OVERVIEW

Datasets : http://snap.stanford.edu/data/index.html

NETWORK TYPES

- **Directed** : directed network (http://en.wikipedia.org/wiki/Directed_graph)



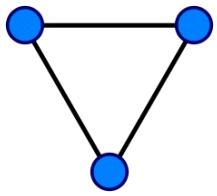
In mathematics, and more specifically in graph theory, a directed graph (or digraph) is a graph, or set of nodes connected by edges, where the edges have a direction associated with them. In formal terms, a digraph is a pair G=(V,A) (sometimes G=(V,E)) of:[1]

- a set V, whose elements are called vertices or nodes,
- a set A of ordered pairs of vertices, called arcs, directed edges, or arrows (and sometimes simply edges with the corresponding set named E instead of A).

It differs from an ordinary or undirected graph, in that the latter is defined in terms of unordered pairs of vertices, which are usually called edges.

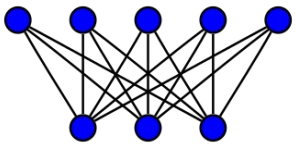
A digraph is called "simple" if it has no loops, and no multiple arcs (arcs with same starting and ending nodes). A directed multigraph, in which the arcs constitute a multiset, rather than a set, of ordered pairs of vertices may have loops (that is, "self-loops" with same starting and ending node) and multiple arcs. Some, but not all, texts allow a digraph, without the qualification simple, to have self loops, multiple arcs, or both.

- **Undirected** : undirected network (http://en.wikipedia.org/wiki/Graph_(mathematics)#Undirected_graph)



An undirected graph is one in which edges have no orientation. The edge (a, b) is identical to the edge (b, a), i.e., they are not ordered pairs, but sets {u, v} (or 2-multisets) of vertices. The maximum number of edges in an undirected graph without a self-loop is n(n - 1)/2.

- **Bipartite** : bipartite network (http://en.wikipedia.org/wiki/Bipartite_graph)



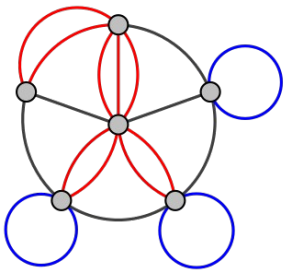
A complete bipartite graph with m = 5 and n = 3

In the mathematical field of graph theory, a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets U and V (that is, U and V are each independent sets) such that every edge connects a vertex in U to one in V. Vertex set U and V are often denoted as partite sets. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles.[1][2]

The two sets U and V may be thought of as a coloring of the graph with two colors: if one colors all nodes in U blue, and all nodes in V green, each edge has endpoints of differing colors, as is required in the graph coloring problem. [3][4] In contrast, such a coloring is impossible in the case of a non-bipartite graph, such as a triangle: after one node is colored blue and another green, the third vertex of the triangle is connected to vertices of both colors, preventing it from being assigned either color.

One often writes G=(U,V,E) to denote a bipartite graph whose partition has the parts U and V, with E denoting the edges of the graph. If a bipartite graph is not connected, it may have more than one bipartition;[5] in this case, the (U,V,E) notation is helpful in specifying one particular bipartition that may be of importance in an application. If $|U|=|V|$, that is, if the two subsets have equal cardinality, then G is called a balanced bipartite graph.[3] If all vertices on the same side of the bipartition have the same degree, then G is called biregular.

- **Multigraph** : network has multiple edges between a pair of nodes (http://en.wikipedia.org/wiki/Multigraph)



A multigraph with multiple edges (red) and several loops (blue). Not all authors allow multigraphs to have loops.

In mathematics, and more specifically in graph theory, a multigraph is a graph which is permitted to have multiple edges (also called parallel edges[1]), that is, edges that have the same end nodes. Thus two vertices may be connected by more than one edge.

There are two distinct notions of multiple edges:

- Edges without own identity: The identity of an edge is defined solely by the two nodes it connects. In this case, the term "multiple edges" means that the same edge can occur several times between these two nodes.
- Edges with own identity: Edges are primitive entities just like nodes. When multiple edges connect two nodes, these are different edges.

A multigraph is different from a hypergraph, which is a graph in which an edge can connect any number of nodes, not just two.

For some authors, the terms pseudograph and multigraph are synonymous. For others, a pseudograph is a multigraph with loops.

- **Temporal** : for each node/edge we know the time when it appeared in the network

- **Labeled** : network contains labels (weights, attributes) on nodes and/or edges

Network statistics

Dataset statistics	
Nodes	Number of nodes in the network
Edges	Number of edges in the network
Nodes in largest WCC	Number of nodes in the largest weakly connected component
Edges in largest WCC	Number of edges in the largest weakly connected component
Nodes in largest SCC	Number of nodes in the largest strongly connected component
Edges in largest SCC	Number of edges in the largest strongly connected component
Average clustering coefficient	Average clustering coefficient *In graph theory, a clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. Evidence suggests that in most real-world networks, and in particular social networks, nodes tend to create tightly knit groups characterised by a relatively high density of ties; this likelihood tends to be greater than the average probability of a tie randomly established between two nodes (Holland and Leinhardt, 1971;[1] Watts and Strogatz, 1998[2]).* *Two versions of this measure exist: the global and the local. The global version was designed to give an overall indication of the clustering in the network, whereas the local gives an indication of the embeddedness of single nodes.*
Number of triangles	Number of triples of connected nodes (considering the network as undirected)
Fraction of closed triangles	Number of connected triples of nodes / number of (undirected) length 2 paths
Diameter (longest shortest path)	Maximum undirected shortest path length (sampled over 1,000 random nodes)
90-percentile effective diameter	90th percentile of undirected shortest path length distribution (sampled over 1,000 random nodes)

Social circles: Facebook (Social / Undirected)

Dataset information

http://snap.stanford.edu/data/egonets-Facebook.html

This dataset consists of 'circles' (or 'friends lists') from Facebook. Facebook data was collected from survey participants using this Facebook app. The dataset includes node features (profiles), circles, and ego networks.

Facebook data has been anonymized by replacing the Facebook-internal ids for each user with a new value. Also, while feature vectors from this dataset have been provided, the interpretation of those features has been obscured. For instance, where the original dataset may have contained a feature "political=Democratic Party", the new data would simply contain "political=anonymized feature 1". Thus, using the anonymized data it is possible to determine whether two users have the same political affiliations, but not what their individual political affiliations represent.

Data is also available from Google+ and Twitter.

Dataset statistics	
Nodes	4039
Edges	88234
Nodes in largest WCC	4039 (1.000)
Edges in largest WCC	88234 (1.000)
Nodes in largest SCC	4039 (1.000)
Edges in largest SCC	88234 (1.000)
Average clustering coefficient	0.6055
Number of triangles	1612010
Fraction of closed triangles	0.2647
Diameter (longest shortest path)	8
90-percentile effective diameter	4.7

Note that these statistics were compiled by combining the ego-networks, including the ego nodes themselves (along with an edge to each of their friends).

Source (citation)

J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

Files

File	Description
facebook.tar.gz	Facebook data (10 networks, anonymized)
facebook_combined.txt.gz	Edges from all egonets combined
readme-Ego.txt	Description of files

Social circles: Google+ (Social / Directed)

Dataset information

http://snap.stanford.edu/data/egonets-Gplus.html

This dataset consists of 'circles' from Google+. Google+ data was collected from users who had manually shared their circles using the 'share circle' feature. The dataset includes node features (profiles), circles, and ego networks.

Data is also available from Facebook and Twitter.

Dataset statistics	
Nodes	107614
Edges	13673453
Nodes in largest WCC	107614 (1.000)
Edges in largest WCC	13673453 (1.000)
Nodes in largest SCC	69501 (0.646)
Edges in largest SCC	9168660 (0.671)
Average clustering coefficient	0.4901
Number of triangles	1073677742
Fraction of closed triangles	0.6552
Diameter (longest shortest path)	6
90-percentile effective diameter	3

Source (citation)

J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

Files

File	Description
gplus.tar.gz	Google+ (132 networks)
gplus_combined.txt.gz	Edges from all egonets combined
readme-Ego.txt	Description of files

Social circles: Twitter (Social / Directed)

Dataset information

http://snap.stanford.edu/data/egonets-Twitter.html

This dataset consists of 'circles' (or 'lists') from Twitter. Twitter data was crawled from public sources. The dataset includes node features (profiles), circles, and ego networks.

Data is also available from Facebook and Google+.

Dataset statistics	
Nodes	81306
Edges	1768149
Nodes in largest WCC	81306 (1.000)
Edges in largest WCC	1768149 (1.000)
Nodes in largest SCC	68413 (0.841)
Edges in largest SCC	1685163 (0.953)
Average clustering coefficient	0.5653
Number of triangles	13082506
Fraction of closed triangles	0.06415
Diameter (longest shortest path)	7
90-percentile effective diameter	4.5

Source (citation)

J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

Files

File	Description
twitter.tar.gz	Twitter data (973 networks)
twitter_combined.txt.gz	Edges from all egonets combined
readme-Ego.txt	Description of files

Citing SNAP

We encourage you to cite our datasets if you have used them in your work.

You can use the following BibTeX citation:

```
@misc{snapnets,
  author       = {Jure Leskovec and Andrej Krevl},
  title        = {{SNAP Datasets}: {Stanford} Large Network Dataset Collection},
  howpublished = {\url{http://snap.stanford.edu/data}},
  month        = jun,
  year         = 2014
}
```