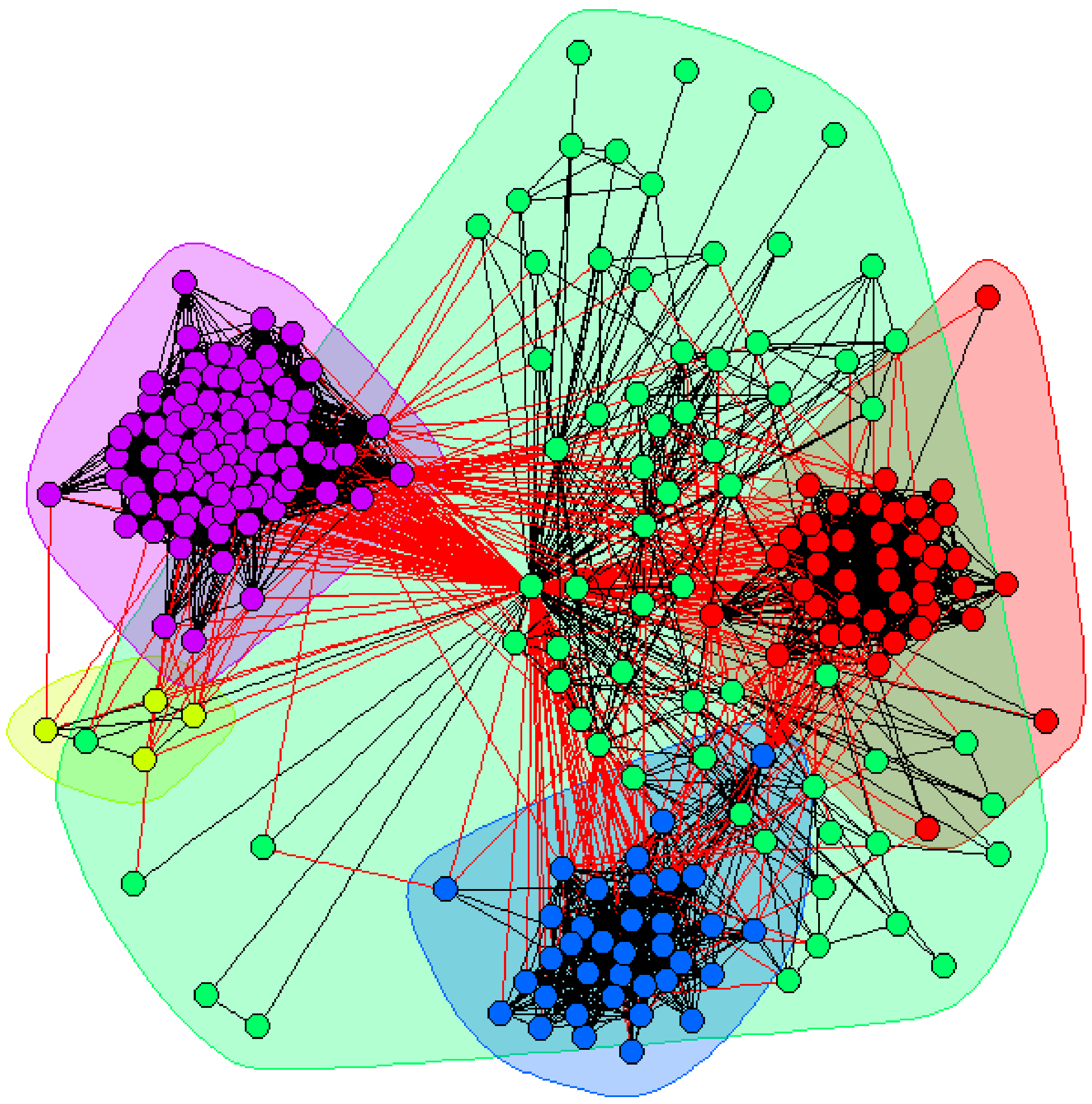
Community Detection in Social Networks

October 19, 2014 generic ML Python, R, social network analysis

In this post I would like to share a very basic approach to **Community Detection in Social Networks**.



I came across this fascinating topic following the superb course on **Mining Massive Datasets** provided on Coursera by Stanford University. The specific field of finding overlapping clusters in graphs is introduced and deeply treated during the third week of classes (links to the PDF slides available for Part 1 and Part 2). I immediately found it extremely interesting and decided to play around by myself. There are at least two very strong reasons to directly check the potentials of these group of algorithms: first of all my complete lack of knowledge in the field and secondly the data I found a couple of weeks ago on a the Kaggle competition "Learning Social Circles in Networks". The contest challenges participants to correctly infer Facebook users' social communities. Such circles may be disjoint, overlap, or be hierarchically nested. To do this, machine learners have access to:

1. A list of the user's friends
2. Anonymized Facebook profiles of each of those friends
3. A network of connections between those friends (their "ego network")

This is exactly what I needed for my learning purposes!

Overview

The approach I propose below is structured in two main parts:

1. **Build the Graph** of the ego-networks extracting nodes and edges from Kaggle data. I implemented this step in Python, generating the graphs with Networkx and saving the Adjiacency matrix of each of them to a separate file.
2. **Community Detection** on top of the undirected graph. I performed this step in R, loading the graphs as Adjiacency matrices and then run a bunch of Clustering Algorithms available in R-igraph.

The use of both Python and R was not planned in the first place. I directly dived into the first of them supported by Neyworkx, but as soon as I started deepening the community detection algorithms I realized that R-igraph had a woderful ensemble of methods directly available. Note that igraph supports Python as well but apparently there are not the same features between the two libraries and the R one seems to be much fancier. I was a bit disappointed at the very beginning but in the end I grabbed the opportunity of learning a new package.

Enough words, I'd say. Let's go for some code.

Building Ego-Networks

The Kaggle data (available here) is organized in 110 .egonet files (corresponding to 110 anonymized facebook users), each containing the network of his friends. A practicle example may help to clarify the data structure.

Let's focus on the file **0.egonet**, which contains all the information on **user 0**'s network. Each row of the file is the list of the friends of the first user in the line who is directly part of the ego's network. Below the first 4 lines are shown (for the purpose of clearness only the first 5 five connections in the line are reported).

```
1: 146 189 229 201 204 ...
2: 146 191 229 201 204 ...
3: 185 80 61 188 22 222 ...
4: 72 61 187 163 177 138 ...
```

0 has **1** as friend who has **146-189-229**… as friends as well.

0 has **2** as friend who has **146-191-229**… as friends as well.

0 has **3** as friend who has **185-80-61**… as friends as well.

0 has **4** as friend who has **72-61-187**… as friends as well.

Well I guess you get the point…

Below I attach the **Python code** which access every egonet file and builds a list of nodes and edges to be fed to the Networkx constructor. Just to be clear [0, 1, 2, 3, 4 …] are vertices of the graph while [(0-1), (1-146), (1-189), (1-229) …] are edges or connections. After a graph has been constructed its adjiacency matrix is computed and saved in a csv file.

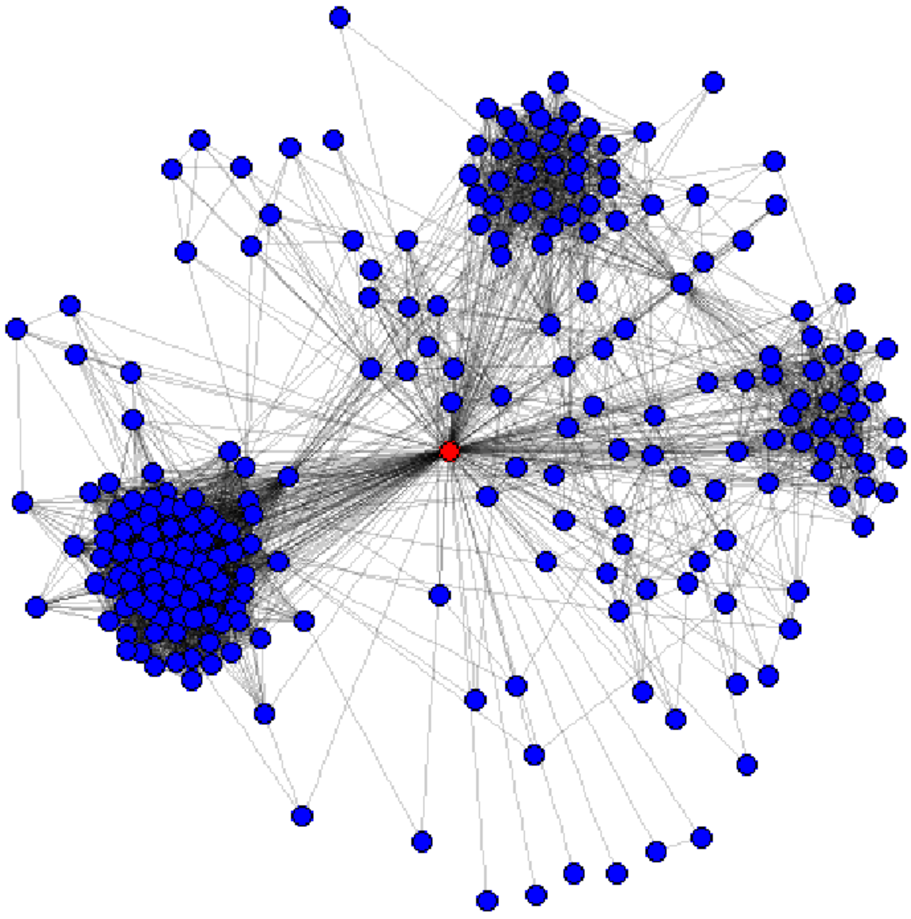
```
import networkx as nx
from os import listdir
from os.path import isfile, join
import itertools
import matplotlib.pyplot as plt
import os
import re
import scipy
from scipy.sparse import *
from operator import itemgetter
from sklearn.cluster import KMeans
import numpy as np
import sys
import pandas as pd


def load_egonet_files(path):
    """
    given the path to the .egonet files returns a list with all the files.
    """
    onlyfiles = [fyle for fyle in listdir(path) if fyle.endswith('.egonet')]
    return onlyfiles

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

def build_graph(n):
    """
    In the Kaggle Competition Learning Social Circles in Networks (https://www.kag
    110 ego-ketworks are available. The function takes as argument an integer in t
    a networkx graph of the friends network of the user.
    """
    edges = []
    nodes = []
    path = 'D:\KaggleGraphs\egonets\egonets'
    egonets = load_egonet_files(path)
    for egonet in egonets[n:n+1]:
        ego = int(re.match( r'([0-9]+).egonet', egonet).group(1))
        m = open(os.path.join(path,egonet), "r")
        friends = [line[:-1].replace(':','').split(' ') for line in m.readlines()]
        friends = [map(int, friend[:1]) if friend[1] == '' else map(int, friend) f
        edges += [(ego,friend[0]) for friend in friends]

        for friend in friends:
            edges += [(friend[0], user) for user in friend[1:] if len(friend)>1]

        nodes += list(itertools.chain.from_iterable(friends)) + [ego]
    edges = list(set(tuple(sorted(edge)) for edge in edges))
    nodes = list(set(nodes))
    G = nx.Graph()
    G.add_nodes_from(nodes)
    G.add_edges_from(edges)

    return G, nodes

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

def to_R(n):
    """
    generates a CSV file with the Adjacency matrix representation of G.
    """
    G, nodes = build_graph(n)
    A = nx.to_numpy_matrix(G)
    df = pd.DataFrame(A, index=nodes, columns=nodes)
    df.to_csv('graph-{}.csv'.format(n),index=True)

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

if __name__ == '__main__':

    for n in range(110):
        to_R(n)
```

The result of the provided code are 110 CSV files containing the adjiacency matrices of each ego network graph. Let's move to the real Clustering part.

Detecting Communities

First of all let's plot a graph and see how it looks like before clustering detection. Below the **R code** to load the data from CSV file, build the network (we stick to the 0.egonet) and draw it.

```
library(igraph)
# read graph from csv file
dat = read.csv('graph-0.csv', header=TRUE, row.names=1, check.names=FALSE)
m = as.matrix(dat)
# build graph from adjacency matrix
g = graph.adjacency(m,mode="undirected",weighted=NULL)

# plots the graph
E(g)$color <- rgb(0,0,0,alpha=.2)
ego <- names(which.max(degree(g)))
V(g)[V(g) != ego]$color = 'blue'
V(g)[ego]$color = 'red'

windows()
plot(g, vertex.label=NA, vertex.size=5, layout=layout.fruchterman.reingold)
```



Time for some clustering.

R-igraph provides several powerful community detection algorithms. Each of them works in a different way and I highly encourage you to have a look at this very informative post on Stack Overflow describing all of them in detail. I decided to go for the whole bunch of algorithms as I wanted to somewhat compare their performances, which I did with the help of **Modularity**. This metrics measures the strength of division of a network into modules. Networks with high modularity have dense connections between the nodes within modules but sparse connections between nodes in different modules.

Modularity is basically the fraction of the edges that fall within the given groups minus the expected such fraction if edges were distributed at random. So the higher the better.

Here you find the results on the user-0-network.

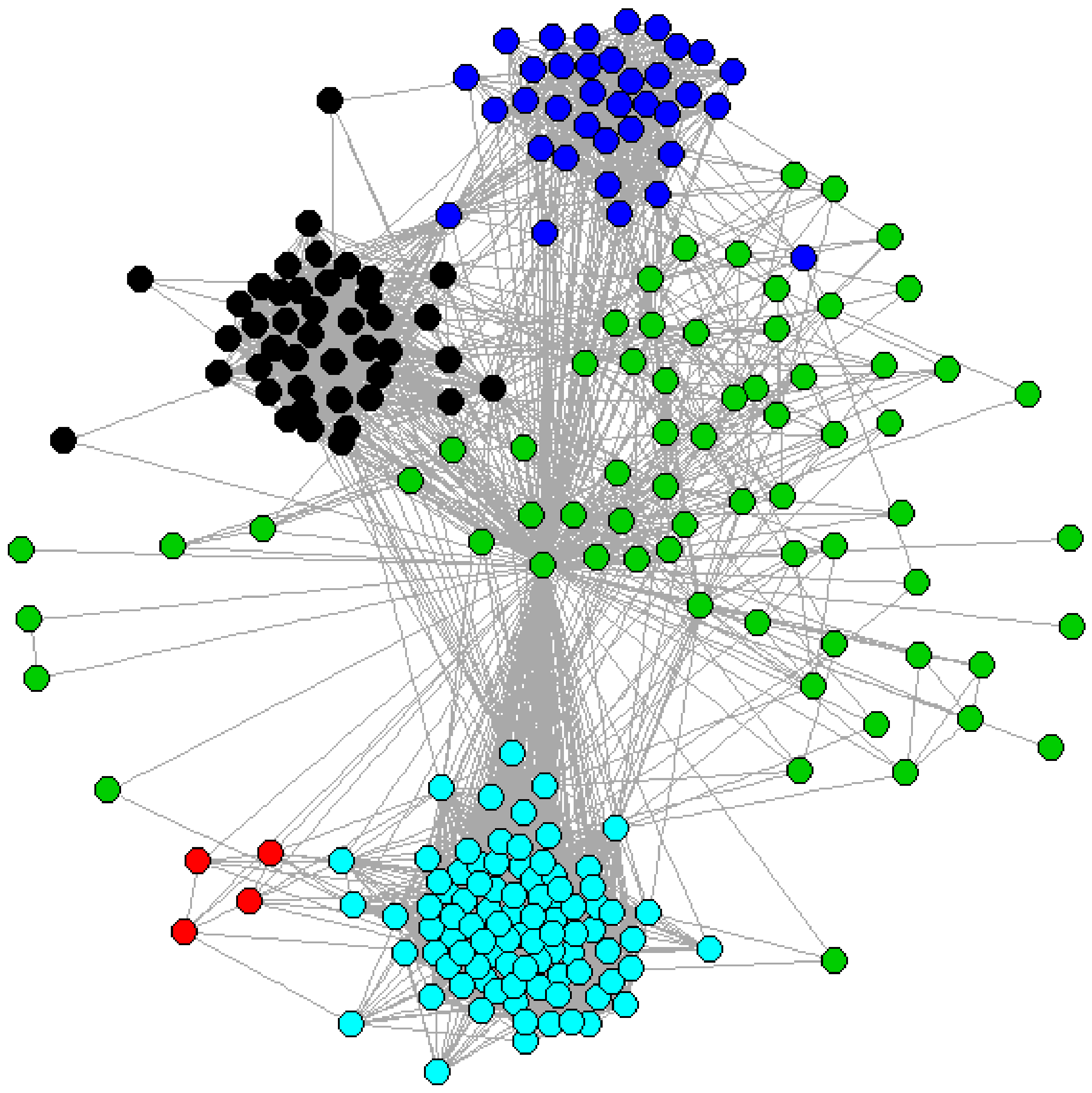
```
> wc <- walktrap.community(g)
> modularity(wc)
```

```
[1] 0.4629543

> wc <- fastgreedy.community(g)
> modularity(wc)
[1] 0.4463902

> wc <- edge.betweenness.community(g)
> modularity(wc)
[1] 0.4330911

> wc <- spinglass.community(g)
> modularity(wc)
[1] 0.4649535

> wc <- leading.eigenvector.community(g)
> modularity(wc)
[1] 0.4511259

> wc <- label.propagation.community(g)
> modularity(wc)
[1] 0.4314803
```

The spinglass.community algorithm (based on a statistical physics approach) is the best one, with a modularity of 0.4649. Turns out that for this particular problem of community detection in small ego-social-networks the spinglass method beats the others in all the 110 egonet graphs.

Below you can find a nice visualization of the detected clusters, in R as well. By the way the plot at the top of the post is exactly the same as the following one visualized in a fancier way.

```
plot(g, vertex.label=NA, vertex.size=5, vertex.color=membership(wc),
     layout=layout.fruchterman.reingold)
```



Share! 0 0 3 1

by ***Francesco Pochetti***

Comments

0 comments