

|  |  |  |  |
| --- | --- | --- | --- |
| DATA SCIENCE – 2FILLING MISSING ATTRIBUTES,FINDING INFLUENCERS Version 1.0 – 14/05/18  Année scolaire 2018-2019 |  |  |  |

|  |
| --- |
| Thibault Dewavrin  Ricardo Moises Rodriguez |

# Introduction

As data scientist at LinkedIn, our colleagues from Marketing want to organize an online marketing campaign for their client, a restaurant in the Bay area. They have read a lot about social network analysis and asked us to find the 5 most influential people on the network who would best promote the restaurant.

When taking a closer look at the dataset we are given, we can see that only 40% of users have provided their location. We will therefore have to fill the uncomplete profiles before finding the influencers.

In order to do that we are given a dataset representing a LinkedIn graph, each node being a person and edges the fact they are connected. People also have three attributes: location (locations they live/have lived in), college (college(s) they went to), employer (company(ies) they worked for). Only 40% of the people in the graph have provided their attribute.

Our aim is to use connections and correlations between those attributes and people in order to fill the missing profiles, and use those results in order to find influencers.

# PROBLEM ENUNCIATION AND STATISTICS

## problem enunciation and task analysis

As data-scientists, we intend to use the CRISP-DM methodology in order to solve the problem, which consists in different phases as presented in the chart beside.



First of all, we needed to do some researches about similar studies, because it may help us understand more the problem and see different solutions. We used several papers but the most useful was [1] User Profiling in an Ego Network: Co-profiling Attributes and Relationships.

We should now list all the factors we think might lead a user to have a certain attribute, as we will use it later. This will help us understand the statistics that are relevant to draw from the graph.

We then have to understand how the dataset is made so that we can use the data properly.

We then draw the statistics and use the statistics to elaborate one or several models for user profiling.

Once it is done, we have to implement the model using the data given and assess the model by comparing our predictions with the ground truth. Once it is done for every model, we will know which one are the best, meaning the one with best accuracies.

We will now have the filled profiles and will be able to use it to establish who are the influencers that might help us find a restaurant at the bay.

## Statistics

We therefore want to draw interesting statistics from the graph that might help us understand what factors make that a node has a certain attribute.

First some basic statistics about the graph:

*Number of users in our graph: 811*

*Number of users with one or more attribute college: 230*

*Number of users with one or more attribute location: 336*

*Number of users with one or more attribute employer: 297*

*We have to find information for 475 users with empty profile*

*Number of edges: 1597*

*Average degree: 3.9383*

*Graph density 0.004862*

*Diameter (maximum eccentricity): 19*

*Radius (minimum eccentricity): 10*

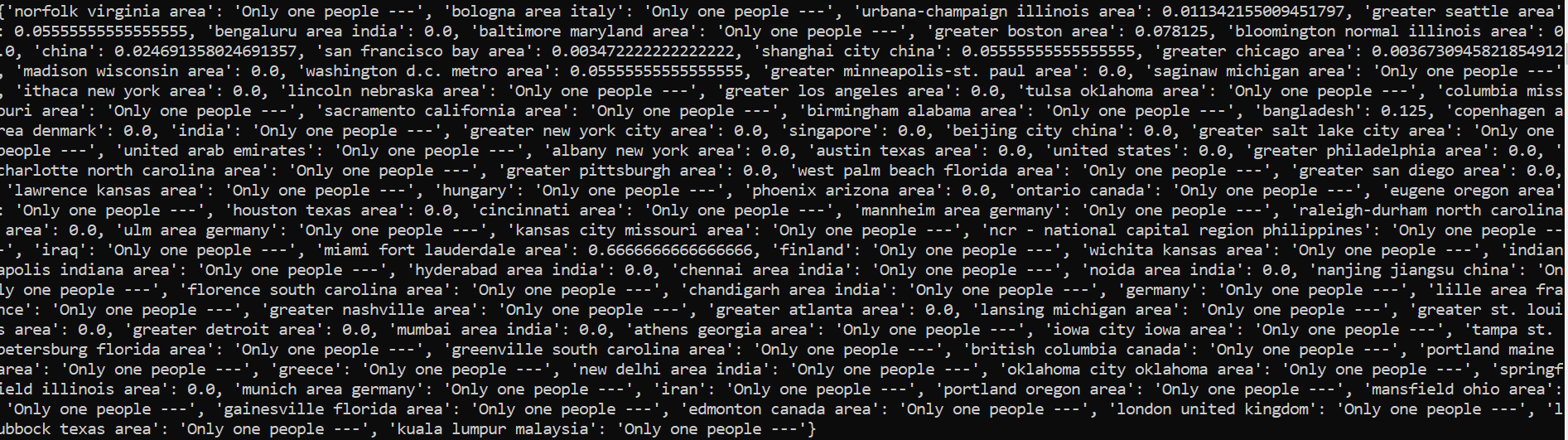
*Mean eccentricity (eccentricity(v) = the maximum distance from v to all other nodes): 13.241676942046857*

*Center is composed of 3 nodes (nodes with eccentricity equal to radius)*

*Periphery is composed of 5 nodes (nodes with eccentricity equal to the diameter)*

*Mean clustering coefficient 0.347288*

*Total number of triangles in graph: 1217*

* We might also want to know, for each location for example, the clustering coefficient for the graph composed only of the nodes that have this value for attribute location. For example, we look at all the people living in location L1 and compute the clustering coefficient for the location L1. If values are high, then we may be able to determine easily whether a user lives in this location or not. Here are the results :

We can see many locations have only one person in it and if they have more than one person, the clustering coefficient is really low. We therefore cannot use this information in our prediction model.

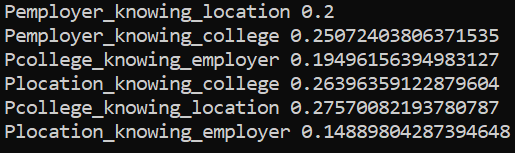
* We can compute the homophily as well:

The homophily for Employer, Location, College are respectively: 0.0338, 0.04320, 0.03569.

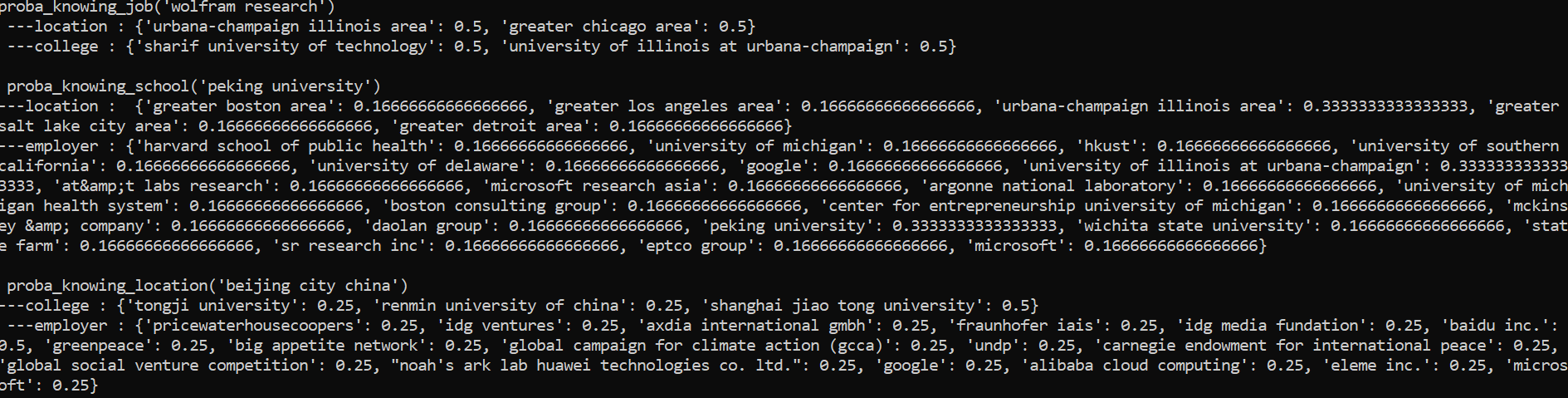
Theses values are low due to the fact very few people have completed their profiles!

* We also wanted to know if sharing an attribute might help sharing another one. For example : if i and j are in the same college, what is the probability they have the same employer ?

Here’s our result



We can see the ¼ of the people from the same location share the same college and vice-versa, which is an interesting point for the models we might want to implement.

Other interesting conditional probabilities would be the probability for users with the same attribute (ex : location) to have another attribute ( ex: employer ). Using it, we might be able to know, for example, that every people who have been to a certain college have a certain attribute location, etc. Below is a sample form our results (for a given job, given college and given location)

We can see this information can be useful, because sometimes, knowing an attribute of a user might help us define the most probable values for his other attributes.

# PREDICTION MODELS AND HYPOTHESeS

## PREAMBULE : ACCURACY EVALUTATION

We will need a precise way to compute the accuracy of our model, by comparing the predictions with the ground truth. Our aim is to fill the profile, but for an attribute, there can be several values (mostly there is only one value but it isn’t impossible that there are more than one value). In a first time, we predict only one value for each attribute, but later we might predict several values.

The evaluation process will be done this way : we will compare the predicted results and ground truth, and return the number of correct predictions divided by the number of predictions.

## THE NAIVE METHOD

We can use the notion of homophily to help us begin solving this problem. Indeed homophily is the ‘tendency of individuals to associate and bond with similar others’ (Wikipedia). If similarities breed connections, we might assume that a proportion of people that are connected are connected because they have similarities. From a logical point of view:

We have ‘A & B sharing attributes => A probably linked to B’ (with a probability equals to the homophily for the attributes shared)

Which is equivalent to ‘A,B not linked => A&B are unlikely to share attributes’.

What we want to do is consider that people who are connected are likely to share one or more attribute, which is not directly equivalent to the concept of homophily but a hypothesis.

Considering this, the first idea that came through our mind was, for a given node, to predict the value of one of his attributes by the most represented value amongst his neighbours’ attributes. (ex: I have 5 neighbours, 3 of them went to college C1, 2 of them went to college C2, our model will predict that I went to college C1).

The results are not very good. (30%,20%,30%). It is logical, if we consider the statistics (for the medium-size network) : the average degree is ~4, which means the sample we use for the prediction isn’t big enough. We need to think how we can extend it without decreasing the precision (which might happen if we use a too large sample) .

## COMMUNITY DETECTION BASED METHOD

After seeing the results from the naive method, we’ve thought about extending the method to a larger part of the network. The next idea that came to our minds was that we could detect communities from the graph. Once we have our partition of the graph and the communities, we know in which community the user is and can predict the value of one of his attributes by the most represented value amongst the people in his community’s attribute.

In order to make the partition and split the graph into different communities, we used the Louvain algorithm and the heuristic provided by the package NetworkX:

*partition = community.best\_partition(G)*

We then had to look what community each unfilled node was in, and browse through the community, take the number of times each location is represented in the community and predict the location corresponding to the most represented area. We can draw a list of the occurrences for each location, compute its maximum and standard deviation, and consider as probable all the locations with occurrence superior or equal to ‘*maximum – standard deviation’*.

The accuracy is : 

The accuracy is the best for college, and the least for employer.

In order to increase the accuracy, we tried to do the prediction by browsing through both the community and the first level ego network. By doing it we’ve had



We can see the prediction is much better for college and location but is worse for the employer.

We will therefore use it only for the prediction of the college and the location.

## USING CONDITIONNAL PROBABILITIES

After drawing the statistics, and a few experiments on the data we’ve had the idea to use one’s known – or predicted – attribute to predict his other attributes.

As college is our most accurate prediction, we wanted to use the predicted or known value of a user’s college in order to predict his location and employer.

For a given user, we take his value for attribute college. If it is missing, we take the predicted value(s). We search for everyone that has been to this college (let’s call it C1) and store their locations. For each location in the stored locations, we compute the probability of living there knowing we went to C1. We then have a list of all the locations where the people who also went to this college usually live in and the probability (Pli they live in this location.

In order to do the prediction, we used the community detection- based method and took all the occurrences multiplied them by the probability Pli, and only took the occurrences\*Pli superior to ‘*maximum – standard deviation’*

Results:

Conditional probabilities applied only on location using the predicted colleges :

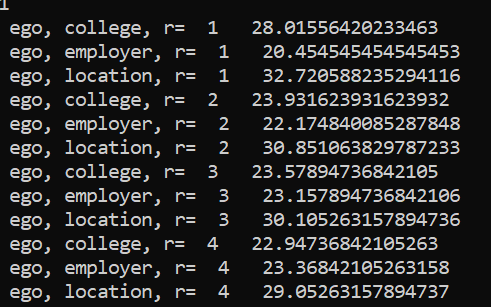


The results are pretty poor, and there is no apparent implementation error, thus leading us to wonder whether this method is accurate. We’ve decided not to use it. Another possibility was to put a coefficient in order to limit the impact of the probability on the predicted result, and do ‘mixed algorithm’ but seeing how poor the results were, we didn’t want to continue on this model, because this model is based on an assumption, that a user goes to a college C1, but seeing the accuracy, the hypothesis is only 41.6% of the time true…

## DIFFERENT LEVELS EGO NETWORK

We wanted to go back to the naive method and extend it using the ego network. There are different levels for the ego network and we can extend the naive method to the level 2, level 3 or level 4 ego network. (level 2 meaning the neighbours of the neighbours, level 3 their neighbours also, etc).

The results we obtained are the following : (r is the radius of the ego network)



We can see the ego network-based method is quite performant, however it is still less than the combination of Louvain and ego network for college and location.

# 

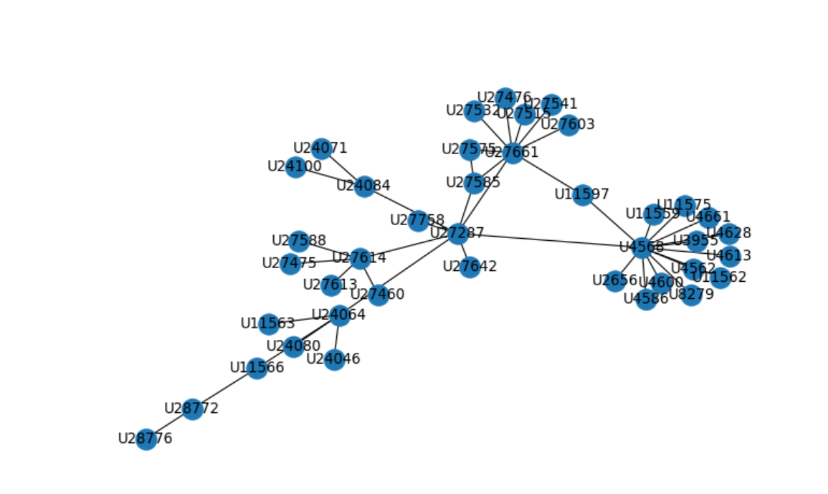
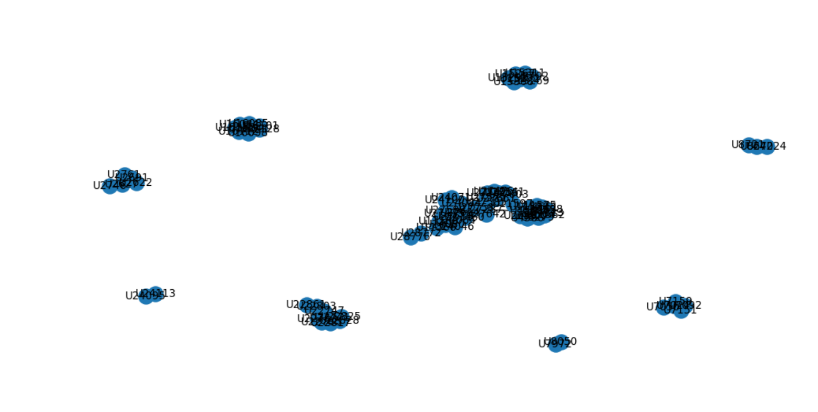
# FINDING INFLUENCERS

In order to find influencers in the San Francisco bay area, we will have to take as a sample all the nodes that live or have a neighbor who lives in this area. Indeed our hypothesis is that people that are in the ego network of someone who works in the bay might be able to know more people that live in the bay than him. Therefore we will do our calculations as such.

The sample we will use will be composed of the people who filled their locations and also the predictions

In order to find what nodes can have the most influence, we have to compute the centrality of each node of the sample. In order to do so, we will use the algorithm provided by networkX that enables us to compute the degree centrality, which is what we are looking for. *( ‘nx.degree\_centrality(G)’* ).

Below the graph of the people who live in san francisco bay area, with a different view for the second one. We can see the graph is composed of several connected components.



The result we have from this simulation is : 

And they are the people that should be contacted in order to promote a restaurant in the bay area. We can see them on the graph, and the result seems coherent because they have a high degree. The results depend on the prediction that depends itself on the partition chosen by our Louvain algorithm heuristic.

# CONCLUSION

As we can see with the different models, our accuracy at its best is a little below 33% for location, 43% for college and 26% for employer.

These results aren’t very accurate but it is difficult knowing we have only 40% of the profiles filled, thus making it hard to make a valid model based on the statistics. The smallest the sample, the less precise the probabilities are!

In order to have a better accuracy, we should focus on the dependency between the attributes. Our method for conditional probabilities wasn’t really good but it surely is possible to find a model that describe better the dependencies.

As the aim of filling the incomplete profiles was to find influencers, it would have been interesting to find a way to give more weigh to the predictions with San Francisco Bay Area in location, in doubt we are wrong. We haven’t found a way yet but it could be the aim of a new study.

Another interesting thing would have been to look for the different circles in an ego network with a radius of 1 as the study [1] does. Our problem was that the average degree was 4 which is too little to have circles, moreover amongst those 4, only 1.6 (in average) would have filled his profile: we lack too many attributes in order to do it.

|  |
| --- |
| BIBLIOGRAPHIE [1*] User Profiling in an Ego Network: Co-profiling Attributes and Relationships* ∗ Rui Li†, Chi Wang†, Kevin Chen-Chuan Chang†,‡ {ruili1, chiwang1, kcchang}@illinois.edu † Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA ‡ Advanced Digital Sciences Center, Illinois at Singapore, Singapore ANNEXES: Le code est accessible sur le drive suivant :  <https://drive.google.com/drive/folders/1teKC7WZ0b1m36gRmYvGNFw5aOYUwMKY_?usp=sharing> |

Ou bien copié collé ici :

import networkx as nx

import matplotlib.pyplot as plt

from matplotlib import pylab

import numpy as np

import pickle

from collections import Counter

from networkx.algorithms import community

import community

college={}

location={}

employer={}

# The dictionaries are loaded as dictionaries from the disk (see pickle in Python doc)

with open('mediumCollege\_60percent\_of\_empty\_profile.pickle', 'rb') as handle:

college = pickle.load(handle)

with open('mediumLocation\_60percent\_of\_empty\_profile.pickle', 'rb') as handle:

location = pickle.load(handle)

with open('mediumEmployer\_60percent\_of\_empty\_profile.pickle', 'rb') as handle:

employer = pickle.load(handle)

def properties(g):

"""

Computes simple and classic graph metrics.

Parameters

----------

g : graph

A networkx graph

"""

# networkx short summary of information for the graph g

print(nx.info(g))

# Draw the degree distribution. Powerlow distribution for a real (complex) network

plt.figure(num=None)

fig = plt.figure(1)

degree\_sequence=[d for n, d in g.degree()] # degree sequence

# print("Degree sequence %s" % degree\_sequence)

plt.hist(degree\_sequence, bins='auto')

plt.title("powerlaw degree distribution")

plt.ylabel("# nodes")

plt.xlabel("degree")

# plt.show()

pylab.close()

del fig

precomputed\_eccentricity = nx.eccentricity(g) # costly step, we save time here!

print("Graph density %f" % nx.density(g))

print("Diameter (maximum eccentricity): %d" % nx.diameter(g,precomputed\_eccentricity))

print("Radius (minimum eccentricity): %d" % nx.radius(g,precomputed\_eccentricity)) #The radius is the minimum eccentricity.

print("Mean eccentricity (eccentricity(v) = the maximum distance from v to all other nodes): %s" % np.mean(list(precomputed\_eccentricity.values())))

print("Center is composed of %d nodes (nodes with eccentricity equal to radius)" % len(nx.center(g, precomputed\_eccentricity)))

print("Periphery is composed of %d nodes (nodes with eccentricity equal to the diameter)" % len(nx.periphery(g,precomputed\_eccentricity)))

print("Mean clustering coefficient %f" % np.mean(list(nx.clustering(g).values())))

total\_triangles=sum(nx.triangles(g).values())/3

print("Total number of triangles in graph: %d" % total\_triangles)

#### Computing the homophily

def homophily(G):

similar\_neighbors\_E=0

similar\_neighbors\_L=0

similar\_neighbors\_C=0

total\_number\_neighbors=0

for n in G.nodes():

for nbr in G.neighbors(n):

total\_number\_neighbors+=1

if n in employer and nbr in employer:

if len([val for val in employer[n] if val in employer[nbr]]) > 0:

similar\_neighbors\_E+=1

if n in college and nbr in college:

if len([val for val in college[n] if val in college[nbr]]) > 0:

similar\_neighbors\_C+=1

if n in location and nbr in location:

if len([val for val in location[n] if val in location[nbr]]) > 0:

similar\_neighbors\_L+=1

homophily\_E=similar\_neighbors\_E/total\_number\_neighbors

homophily\_C=similar\_neighbors\_C/total\_number\_neighbors

homophily\_L=similar\_neighbors\_L/total\_number\_neighbors

print("\n The homophily for E,L,C are respectively : ",homophily\_E,homophily\_L,homophily\_C)

return homophily\_E,homophily\_L,homophily\_C

def college\_shared(i,j):

att\_shared=[]

if i in college:

if j in college:

for attj in college[j]:

for atti in college[i]:

if atti==attj:

att\_shared.append(atti)

return (att\_shared)

def location\_shared(i,j):

att\_shared=[]

if i in location:

if j in location:

for attj in location[j]:

for atti in location[i]:

if atti==attj:

att\_shared.append(atti)

return (att\_shared)

def employer\_shared(i,j):

att\_shared=[]

if i in employer:

if j in employer:

for attj in employer[j]:

for atti in employer[i]:

if atti==attj:

att\_shared.append(atti)

return (att\_shared)

def proba\_knowing\_job(job):

loc={}

col={}

nb\_e=0

for e in employer:

if job in employer[e]:

if e in location and e in college :

nb\_e+=1

if e in location:

for l in location[e]:

if l in loc:

loc[l]+=1

else:

loc[l]=1

if e in college:

for c in college[e]:

if c in col:

col[c]+=1

else:

col[c]=1

for i in loc:

loc[i]/=nb\_e

for i in col:

col[i]/=nb\_e

return loc,col

def proba\_knowing\_school(school):

loc={}

emp={}

nb\_c=0

for c in college:

if school in college[c]:

if c in location and c in employer :

nb\_c+=1

if c in location:

for l in location[c]:

if l in loc:

loc[l]+=1

else:

loc[l]=1

if c in employer:

for e in employer[c]:

if e in emp:

emp[e]+=1

else:

emp[e]=1

for i in loc:

loc[i]/=nb\_c

for i in emp:

emp[i]/=nb\_c

return loc,emp

def proba\_knowing\_location(loca):

col={}

emp={}

nb\_l=0

for l in location:

if loca in location[l]:

if l in college and l in employer :

nb\_l+=1

if l in college:

for c in college[l]:

if c in col:

col[c]+=1

else:

col[c]=1

if l in employer:

for e in employer[l]:

if e in emp:

emp[e]+=1

else:

emp[e]=1

for i in col:

col[i]/=nb\_l

for i in emp:

emp[i]/=nb\_l

return col,emp

def probas\_conditionnelles(G):

location\_knowing\_college=0

employer\_knowing\_college=0

employer\_knowing\_location=0

nb\_total\_college=0

nb\_total\_location=0

nb\_total\_employer=0

total\_college\_shared=0

total\_location\_shared=0

total\_employer\_shared=0

for i in G.nodes:

for j in G.nodes:

if i!=j:

if i in college:

if j in college:

nb\_total\_college+=1

if i in location:

if j in location:

nb\_total\_location+=1

if i in employer:

if j in employer:

nb\_total\_employer+=1

if len(college\_shared(i,j))!=0:

total\_college\_shared+=1

if len(location\_shared(i,j))!=0:

location\_knowing\_college+=1

if len(employer\_shared(i,j))!=0:

employer\_knowing\_college+=1

if len(location\_shared(i,j))!=0:

total\_location\_shared+=1

if len(employer\_shared(i,j))!=0:

employer\_knowing\_location+=1

if len(employer\_shared(i,j))!=0:

total\_employer\_shared+=1

Pc=total\_college\_shared/nb\_total\_college

Pl=total\_location\_shared/nb\_total\_location

Pe=total\_employer\_shared/nb\_total\_employer

Pl\_w\_c=location\_knowing\_college/total\_college\_shared

Pe\_w\_c=employer\_knowing\_college/total\_college\_shared

Pe\_w\_l=employer\_knowing\_location/total\_location\_shared

Pl\_w\_e=Pe\_w\_l\*(Pe/Pl)

Pc\_w\_l=Pl\_w\_c\*(Pl/Pc)

Pc\_w\_e=Pe\_w\_c\*(Pe/Pc)

return(Pe\_w\_l,Pe\_w\_c,Pc\_w\_e,Pl\_w\_c,Pc\_w\_l,Pl\_w\_e)

# -\*- coding: utf-8 -\*-

"""

Created on Wed Jul 26 16:09:11 2017

@author: cbothore

"""

from statistiques import \*

import networkx as nx

import matplotlib.pyplot as plt

from matplotlib import pylab

import numpy as np

import pickle

from collections import Counter

from networkx.algorithms import community

import community

import operator

def naive\_method(graph, empty, attr):

""" Predict the missing attribute with a simple but effective

relational classifier.

The assumption is that two connected nodes are

likely to share the same attribute value. Here we chose the most frequently

used attribute by the neighbors

Parameters

----------

graph : graph

A networkx graph

empty : list

The nodes with empty attributes

attr : dict

A dict of attributes, either location, employer or college attributes.

key is a node, value is a list of attribute values.

Returns

-------

predicted\_values : dict

A dict of attributes, either location, employer or college attributes.

key is a node (from empty), value is a list of attribute values. Here

only 1 value in the list.

"""

predicted\_values={}

for n in empty:

nbrs\_attr\_values=[]

for nbr in graph.neighbors(n):

if nbr in attr:

for val in attr[nbr]:

nbrs\_attr\_values.append(val)

predicted\_values[n]=[]

if nbrs\_attr\_values: # non empty list

# count the number of occurrence each value and returns a dict

cpt=Counter(nbrs\_attr\_values)

# take the most represented attribute value among neighbors

a,nb\_occurrence=max(cpt.items(), key=lambda t: t[1])

predicted\_values[n].append(a)

return predicted\_values

def evaluation\_accuracy(groundtruth, pred):

""" Compute the accuracy of your model.

The accuracy is the proportion of true results.

Parameters

----------

groundtruth : : dict

A dict of attributes, either location, employer or college attributes.

key is a node, value is a list of attribute values.

pred : dict

A dict of attributes, either location, employer or college attributes.

key is a node, value is a list of attribute values.

Returns

-------

out : float

Accuracy.

"""

#it improved the impact of a good answer in the different values for a prediction

true\_positive\_prediction=0

total\_predictions=0

for p\_key,p\_val in pred.items():

if p\_key in groundtruth:

for p\_value in pred[p\_key]:

total\_predictions+=1

# if prediction is no attribute values, e.g. [] and so is the groundtruth

# May happen

if not p\_value and not groundtruth[p\_key]:

true\_positive\_prediction +=1

# counts the number of good prediction for node p\_key

# here len(p\_value)=1 but we could have tried to predict more values

if p\_value in groundtruth[p\_key]:

true\_positive\_prediction += 1

#len([c for c in p\_value if c in groundtruth[p\_key]])

# no else, should not happen: train and test datasets are consistent

return true\_positive\_prediction\*100/total\_predictions

# load the graph

G = nx.read\_gexf("mediumLinkedin.gexf")

print("Nb of users in our graph: %d" % len(G))

# load the profiles. 3 files for each type of attribute

# Some nodes in G have no attributes

# Some nodes may have 1 attribute 'location'

# Some nodes may have 1 or more 'colleges' or 'employers', so we

# use dictionaries to store the attributes

college={}

location={}

employer={}

# The dictionaries are loaded as dictionaries from the disk (see pickle in Python doc)

with open('mediumCollege\_60percent\_of\_empty\_profile.pickle', 'rb') as handle:

college = pickle.load(handle)

with open('mediumLocation\_60percent\_of\_empty\_profile.pickle', 'rb') as handle:

location = pickle.load(handle)

with open('mediumEmployer\_60percent\_of\_empty\_profile.pickle', 'rb') as handle:

employer = pickle.load(handle)

print("Nb of users with one or more attribute college: %d" % len(college))

print("Nb of users with one or more attribute location: %d" % len(location))

print("Nb of users with one or more attribute employer: %d" % len(employer))

# here are the empty nodes for whom your challenge is to find the profiles

empty\_nodes=[]

with open('mediumRemovedNodes\_60percent\_of\_empty\_profile.pickle', 'rb') as handle:

empty\_nodes = pickle.load(handle)

print("Your mission, find attributes to %d users with empty profile" % len(empty\_nodes))

# --------------------- Baseline method -------------------------------------#

# Try a naive method to predict attribute

# This will be a baseline method for you, i.e. you will compare your performance

# with this method

# Let's try with the attribute 'employer'

employer\_predictions=naive\_method(G, empty\_nodes, employer)

location\_prediction =naive\_method(G,empty\_nodes, location)

groundtruth\_employer={}

with open('mediumEmployer.pickle', 'rb') as handle :

groundtruth\_employer = pickle.load(handle)

with open('mediumLocation.pickle', 'rb') as handle :

groundtruth\_location = pickle.load(handle)

with open('mediumCollege.pickle', 'rb') as handle :

groundtruth\_college = pickle.load(handle)

result=evaluation\_accuracy(groundtruth\_employer,employer\_predictions)

#result=evaluation\_accuracy(groundtruth\_location, location\_prediction)

print("%f%% of the predictions are true" % result)

# --------------------- Now your turn -------------------------------------#

# Explore, implement your strategy to fill empty profiles of empty\_nodes

# and compare with the ground truth (what you should have predicted)

# user precision and recall measures

print ( "\n------------------------- Our answers -----------------------")

ListsOfSchools={}

for j in G.nodes():

if j in college:

for k in college[j]:

if k not in ListsOfSchools:

ListsOfSchools[k]=[]

ListsOfSchools[k].append(j)

ListsOfJobs={}

for j in G.nodes:

if j in employer:

for k in employer[j]:

if k not in ListsOfJobs:

ListsOfJobs[k]=[]

ListsOfJobs[k].append(j)

ListOfLocations={}

for j in G.nodes:

if j in location:

for k in location[j]:

if k not in ListOfLocations:

ListOfLocations[k]=[]

ListOfLocations[k].append(j)

properties(G)

#compute the clustering coeff for each attribute will help us define where each one lives.

#print("Clustering coefficients %f", list(nx.clustering(G).values()))

#plt.hist(list(nx.clustering(G).values()))

#plt.show()

""" how to use the clustering coefficient ???

how to use homophily ??

if people go to the same school, they might be connected.

then ( contraposée ) if they are not connected it means they were probably not in the same school -> use 1-homophily again !!!

Can we rely on homophily as we computed it ? or should we compute another version of homophily computing it for only people we know have filled in

the information concerning the attribute ?

"""

#let's take the people who live at each location

ListOfLocations={}

LocationsAndClusters={}

for j in G.nodes:

if j in location:

for k in location[j]:

if k not in ListOfLocations:

ListOfLocations[k]=[]

ListOfLocations[k].append(j)

for l in ListOfLocations:

connected=0

total=0

only\_one\_people=0

for k in ListOfLocations[l]:

for m in ListOfLocations[l]:

if m in G.neighbors(k):

connected+=1

total+=1

LocationsAndClusters[l]=connected/total

if len(ListOfLocations[l])==1:

LocationsAndClusters[l]="Only one people ---"

only\_one\_people+=1

print('----',only\_one\_people)

#print(LocationsAndClusters) # the "clustering coefficient" for each location

""" functions """

#let's determine where are the communities

#print(community.k\_clique\_communities(G,5))

#print(partition)

def extracting\_communities(partition):

communities={}

for i in G.nodes:

if partition[i] in communities:

communities[partition[i]].append(i)

else:

communities[partition[i]]=[i]

return(communities)

def voir\_partition(partition,G):

size = float(len(set(partition.values())))

pos = nx.spring\_layout(G)

count = 0.

for com in set(partition.values()) :

count = count + 1.

list\_nodes = [nodes for nodes in partition.keys()

if partition[nodes] == com]

nx.draw\_networkx\_nodes(G, pos, list\_nodes, node\_size = 20,

node\_color = str(count / size))

nx.draw\_networkx\_edges(G, pos, alpha=0.5)

plt.show()

#print(location)

### USING LOUVAIN COMMUNITY - NAIVE MODEL

partition =community.best\_partition(G)

communities=extracting\_communities(partition)

#print(extracting\_communities(partition))

"""echo\_graph ??????????????????????????????????????????????"""

def keywithmaxval(d):

max=0

for p\_key,p\_value in d.items():

if p\_value>=max:

max=p\_value

key=p\_key

return p\_key

def louvain\_naive(G):

predicted\_location={}

predicted\_employer={}

predicted\_college={}

for i in empty\_nodes :

if i not in college:

possible\_colleges={}

for j in communities[partition[i]]:

if j in college:

for c in college[j]:

if c in possible\_colleges:

possible\_colleges[c]+=1

else:

possible\_colleges[c]=1

occurences=[]

for p\_key in possible\_colleges:

occurences.append(possible\_colleges[p\_key])

predicted\_college[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_colleges:

if possible\_colleges[pl]>=maxi-ecart\_type:

predicted\_college[i].append(pl)

if possible\_colleges[pl]==maxi:

maxkey=pl

predicted\_college[i]=[maxkey] # if we use only one attribute

if i not in location:

possible\_locations={}

for j in communities[partition[i]] :

if j in location:

for l in location[j]:

if l in possible\_locations:

possible\_locations[l]+=1

else:

possible\_locations[l]=1

occurences=[]

for p\_key in possible\_locations:

occurences.append(possible\_locations[p\_key])

predicted\_location[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_locations:

if possible\_locations[pl]>=maxi-ecart\_type:

predicted\_location[i].append(pl)

if possible\_locations[pl]==maxi:

maxkey=pl

predicted\_location[i]=[maxkey] # if we use only one attribute as answer

if i not in employer:

possible\_employers={}

for j in communities[partition[i]]:

if j in employer:

for e in employer[j]:

if e in possible\_employers:

possible\_employers[e]+=1

else:

possible\_employers[e]=0

occurences=[]

for p\_key in possible\_employers:

occurences.append(possible\_employers[p\_key])

predicted\_employer[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_employers:

if possible\_employers[pl]>=maxi-ecart\_type :

predicted\_employer[i].append(pl)

if possible\_employers[pl]==maxi:

maxkey=pl

predicted\_employer[i]=[maxkey] # if we use only one attribute

return predicted\_college,predicted\_location,predicted\_employer

#predicted\_college,predicted\_location,predicted\_employer=louvain\_naive(G)

#print(empty\_nodes)

#print(predicted\_employer)

#print(groundtruth\_employer)

#result=evaluation\_accuracy\_several\_attributes(groundtruth\_location,predicted\_location)

#print(result)

#print(groundtruth)

def ego\_niveau2(i):

niveau2=[]

for j in G.neighbors(i):

niveau2.append(j)

for k in G.neighbors(j):

if k!=i:

niveau2.append(k)

return niveau2

def louvain\_and\_ego(G):

predicted\_location={}

predicted\_employer={}

predicted\_college={}

for i in empty\_nodes :

if i not in college:

possible\_colleges={}

for j in communities[partition[i]] and nx.ego\_graph(G,i,1):

if j in college:

for c in college[j]:

if c in possible\_colleges:

possible\_colleges[c]+=1

else:

possible\_colleges[c]=1

occurences=[]

for p\_key in possible\_colleges:

occurences.append(possible\_colleges[p\_key])

predicted\_college[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_colleges:

if possible\_colleges[pl]>=maxi-ecart\_type:

predicted\_college[i].append(pl)

if possible\_colleges[pl]==maxi:

maxkey=pl

predicted\_college[i]=[maxkey] # if we use only one attribute

if i not in location:

possible\_locations={}

for j in communities[partition[i]] and nx.ego\_graph(G,i,1):

if j in location:

for l in location[j]:

if l in possible\_locations:

possible\_locations[l]+=1

else:

possible\_locations[l]=1

occurences=[]

for p\_key in possible\_locations:

occurences.append(possible\_locations[p\_key])

predicted\_location[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_locations:

if possible\_locations[pl]>=maxi-ecart\_type:

predicted\_location[i].append(pl)

if possible\_locations[pl]==maxi:

maxkey=pl

predicted\_location[i]=[maxkey] # if we use only one attribute as answer

if i not in employer:

possible\_employers={}

for j in communities[partition[i]] and nx.ego\_graph(G,i,1):

if j in employer:

for e in employer[j]:

if e in possible\_employers:

possible\_employers[e]+=1

else:

possible\_employers[e]=0

occurences=[]

for p\_key in possible\_employers:

occurences.append(possible\_employers[p\_key])

predicted\_employer[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_employers:

if possible\_employers[pl]>=maxi-ecart\_type :

predicted\_employer[i].append(pl)

if possible\_employers[pl]==maxi:

maxkey=pl

predicted\_employer[i]=[maxkey] # if we use only one attribute

return predicted\_college,predicted\_location,predicted\_employer

def comparing(uniname):

if 'at' in uniname:

for i in range(len(uniname)):

if uniname[i]=='a':

if uniname[i+1]=='t':

if uniname[i+2]==' ':

return uniname[i+3:]

return False

def maxi(predicted,list):

max=0

for k in list:

if list[k]>=max:

max=list[k]

predicted\_college[i]=k

def louvain\_and\_conditionnal(G):

predicted\_location={}

predicted\_employer={}

predicted\_college={}

for i in empty\_nodes :

if i not in college:

possible\_colleges={}

for j in communities[partition[i]] and nx.ego\_graph(G,i,1):

if j in college:

for c in college[j]:

if c in possible\_colleges:

possible\_colleges[c]+=1

else:

possible\_colleges[c]=1

occurences=[]

for p\_key in possible\_colleges:

occurences.append(possible\_colleges[p\_key])

predicted\_college[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_colleges:

if possible\_colleges[pl]>=maxi-ecart\_type:

predicted\_college[i].append(pl)

if possible\_colleges[pl]==maxi:

maxkey=pl

predicted\_college[i]=[maxkey] # if we use only one attribute

if i not in location:

possible\_locations={}

for j in communities[partition[i]] and nx.ego\_graph(G,i,1) :

if j in location:

for l in location[j]:

if l in possible\_locations:

possible\_locations[l]+=1

else:

possible\_locations[l]=1

occurences=[]

for p\_key in possible\_locations:

occurences.append(possible\_locations[p\_key])

predicted\_location[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_locations:

if possible\_locations[pl]>=maxi-ecart\_type:

predicted\_location[i].append(pl)

if possible\_locations[pl]==maxi:

maxkey=pl

predicted\_location[i]=[maxkey] # if we use only one attribute as answer

if i not in employer:

possible\_employers={}

for j in communities[partition[i]]:

if j in employer:

for e in employer[j]:

if e in possible\_employers:

possible\_employers[e]+=1

else:

possible\_employers[e]=0

occurences=[]

for p\_key in possible\_employers:

occurences.append(possible\_employers[p\_key])

predicted\_employer[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_employers:

if possible\_employers[pl]>=maxi-ecart\_type :

predicted\_employer[i].append(pl)

if possible\_employers[pl]==maxi:

maxkey=pl

predicted\_employer[i]=[maxkey] # if we use only one attribute

###adding the conditionnal probas:

if i in college:

loc\_prob,emp\_prob=proba\_knowing\_school(college[i])

if i not in location:

for l in possible\_locations:

pl=loc\_prob[l]

possible\_locations[l]\*=pl\*\*(8)

occurences=[]

occurences.append(possible\_locations[l])

predicted\_location[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_locations:

if possible\_locations[pl]>=maxi-ecart\_type :

predicted\_location[i].append(pl)

if possible\_locations[pl]>=maxi:

maxkey=pl

#predicted\_location[i]=[maxkey]

if i in predicted\_college:

loc\_prob,emp\_prob=proba\_knowing\_school(predicted\_college[i])

if i not in location:

for l in possible\_locations:

pl=1

if l in loc\_prob:

pl=loc\_prob[l]

possible\_locations[l]\*=1

occurences=[]

occurences.append(possible\_locations[l])

predicted\_location[i]=[]

if len(occurences)!=0:

ecart\_type=np.std(occurences)

maxi=np.max(occurences)

for pl in possible\_locations:

if possible\_locations[pl]>=maxi-ecart\_type:

predicted\_location[i].append(pl)

if possible\_locations[pl]==maxi:

maxkey=pl

#predicted\_location[i]=[maxkey]

return predicted\_college,predicted\_location,predicted\_employer

def ego\_graph\_method(graph, empty, attr,r):

predicted\_values={}

for n in empty:

nbrs\_attr\_values=[]

for nbr in nx.ego\_graph(G,n,r):

if nbr in attr:

for val in attr[nbr]:

nbrs\_attr\_values.append(val)

predicted\_values[n]=[]

if nbrs\_attr\_values: # non empty list

# count the number of occurrence each value and returns a dict

cpt=Counter(nbrs\_attr\_values)

# take the most represented attribute value among neighbors

a,nb\_occurrence=max(cpt.items(), key=lambda t: t[1])

predicted\_values[n].append(a)

return predicted\_values

"""

for r in range(1,5) :

college\_ego\_r= ego\_graph\_method(G,empty\_nodes,college,r)

print("----- ego, college, r= ",r," ",evaluation\_accuracy(groundtruth\_college,college\_ego\_r))

employer\_ego\_r= ego\_graph\_method(G,empty\_nodes,employer,r)

print("----- ego, employer, r= ",r," ",evaluation\_accuracy(groundtruth\_employer,employer\_ego\_r))

location\_ego\_r= ego\_graph\_method(G,empty\_nodes,location,r)

print("----- ego, location, r= ",r," ",evaluation\_accuracy(groundtruth\_location,location\_ego\_r))

"""

#print(groundtruth)

#starting to use conditionnal porbabilities

##print(employer)

#print("proba\_knowing\_job('wolfram research')","\n ---location :", proba\_knowing\_job('wolfram research')[0],"\n","---college :",proba\_knowing\_job('wolfram research')[1])

#print("\n proba\_knowing\_school('peking university')","\n---location : ",proba\_knowing\_school('peking university')[0],"\n---employer :",proba\_knowing\_school('peking university')[1])

#print("\n proba\_knowing\_location('beijing city china')","\n---college :", proba\_knowing\_location('beijing city china')[0],"\n ---employer :",proba\_knowing\_location('beijing city china')[1])

#print(ListsOfSchools)

""" use assortativity degree ? """

##print(" Pemployer\_knowing\_location",Pemployer\_knowing\_location,"\n", "Pemployer\_knowing\_college",Pemployer\_knowing\_college,"\n","Pcollege\_knowing\_employer",Pcollege\_knowing\_employer,"\n","Plocation\_knowing\_college",Plocation\_knowing\_college,"\n",

# "Pcollege\_knowing\_location",Pcollege\_knowing\_location,"\n","Plocation\_knowing\_employer",Plocation\_knowing\_employer,"\n")

#print(groundtruth\_location,groundtruth\_employer,groundtruth\_college)

predicted\_college,predicted\_location,predicted\_employer=louvain\_naive(G)

result\_louvain\_naive=(evaluation\_accuracy(groundtruth\_college,predicted\_college),evaluation\_accuracy(groundtruth\_employer,predicted\_employer), evaluation\_accuracy(groundtruth\_location,predicted\_location))

print("college, employer, location",result\_louvain\_naive)

predicted\_college,predicted\_location,predicted\_employer=louvain\_and\_ego(G)

result\_louvain\_ego=(evaluation\_accuracy(groundtruth\_college,predicted\_college),evaluation\_accuracy(groundtruth\_employer,predicted\_employer), evaluation\_accuracy(groundtruth\_location,predicted\_location))

print("college, employer, location",result\_louvain\_ego)

#print(predicted\_location)

predicted\_college,predicted\_location,predicted\_employer=louvain\_and\_conditionnal(G)

result\_louvain\_conditionnal=(evaluation\_accuracy(groundtruth\_college,predicted\_college),evaluation\_accuracy(groundtruth\_employer,predicted\_employer), evaluation\_accuracy(groundtruth\_location,predicted\_location))

print("college, employer, location",result\_louvain\_conditionnal)

#--------------------------------------- influencers

predicted\_college,predicted\_location,predicted\_employer=louvain\_and\_ego(G) # the most accurate prediction

def draw\_graph(g, node\_attribute=None, list\_of\_values\_of\_attributes=None):

"""

Draw the graph g.

Parameters

----------

g : graph

A networkx graph

node\_attribute : string

The name of the node attribute used to assign colors to the drawing

list\_of\_values\_of\_attributes : list

A list of all the potential values of node\_attribute to assign one color

per value.

"""

#initialze Figure

plt.figure(num=None, figsize=(20, 20), dpi=80)

plt.axis('off')

fig = plt.figure(1)

pos = nx.spring\_layout(g, iterations=100)

if node\_attribute and list\_of\_values\_of\_attributes:

# To associate colors to nodes according to an attribute, here college

# build a color\_map, one for each college

color\_map={}

i=0.0

for s in list\_of\_values\_of\_attributes:

color\_map[s]=i

i+=1/len(list\_of\_values\_of\_attributes)

color\_map[None]=1 # for nodes without values for the attribute node\_attribute

# The values supplied to node\_color should be in the same order as the nodes

# listed in G.nodes(). We take an arbitrary mapping of values color\_map and

# generate the values list in the correct order

#values = [color\_map[G.node[node].get(node\_attribute)] for node in G.nodes()] # for attributes encoded in the graph

values=[]

for node in G.nodes():

if node in node\_attribute:

if node\_attribute[node]:

# we arbitrarily take the first value

values.append(color\_map[node\_attribute[node][0]])

else:

values.append(1)

nx.draw\_networkx\_nodes(g,pos, cmap=plt.get\_cmap('jet'), node\_color=values)

else:

nx.draw\_networkx\_nodes(g,pos)

nx.draw\_networkx\_edges(g,pos)

nx.draw\_networkx\_labels(g,pos)

cut = 1.00

xmax = cut \* max(xx for xx, yy in pos.values())

ymax = cut \* max(yy for xx, yy in pos.values())

plt.xlim(0, xmax)

plt.ylim(0, ymax)

plt.show()

pylab.close()

del fig

#Completing the global atributes of location

p\_location=predicted\_location

location\_updated={}

for i in G.nodes():

if i in location:

location\_updated[i]=location[i]

if i not in location :

if i in p\_location:

location\_updated[i]=p\_location[i]

#print("-----",ListOfLocations["san francisco bay area"])

location\_bay={}

n\_in\_bay=0

for n in ListOfLocations["san francisco bay area"]:

n\_in\_bay+=1

location\_bay[n]=[]

location\_bay[n]=location\_updated[n]

print(n\_in\_bay)

F=nx.Graph() # getting the graph directed for another approximation

for i in G.nodes():

if i in location\_bay:

F.add\_node(i)

for j in G.neighbors(i):

F.add\_edge(i,j)

# if j in location\_bay:

# F.add\_edge(i,j)

#draw\_graph(G, node\_attribute=location\_bay, list\_of\_values\_of\_attributes=list\_of\_different\_attribute\_values(location\_bay))

centrality = nx.degree\_centrality(F)

#print(centrality)

print(sorted(centrality,key=centrality.\_\_getitem\_\_)[len(centrality)-5:len(centrality)])

draw\_graph(F)