**Data set （这个是公共的。。）**

the data set is crawled from foursquare and twitter. We exploited foursquare endpoint API to get user profile, friendship graph, venues profile and checkin data. Checkin data can be considered as a unary rating for venues: if user chooses to checkin at a venue, he/she must at least consider it’s worth sharing.

**user-based collaborative filtering**

With checkin data collected, if we consider checkin as an action of showing interest, the whole checkin data set can be considered a labeled training set. Using this data, we can do a user-based collaborative filtering: recommend venues according to other users’ checkin records.

In our implementation, we used classic kNN algorithm to define a neighborhood of users similar to the user who we want to suggest venues to. The neighborhood is defined on the basis of the cosine similarity measure:

where vector u1 and u2 represent user’s rating of a venue, ratings are 1 if user checkins in a venue, 0 otherwise.

Testing was done by randomly selecting 30% of the users that are “active”: active users are users who had checkined at least two venues, and removing half their checkins: removing checkins means to set corresponding bit to 0, randomly. Then for each user, we do recommendations according to remaining data, then compare our recommendation to the real checkins data that was set aside at beginning, and calculate precision and recall. Do this for several times and calculate curriculum precision and recall.

To run this, just change directory into final/recommend/ , do “python collabrativeFiltering.py”, it will do recommendations and tests, and print out curriculum precision and recall.

**friend recommendation**

This friend recommendation implementation uses simple graph concept. The program first load from user.dat and friendship.dat the nodes and edges of a social graph, then implement two APIs recommend\_by\_number\_of\_common\_friends and recommend\_by\_influence.

Recommend by number of friends is intuitively easy: If non-friend Y is your friend's friend, then maybe Y should be your friend too. If person Y is the friend of many of your friends, then Y is an even better recommendation. The best friend recommendation is the person with whom you have the largest number of mutual friends.

A different way of recommending friend is: if a person has great influence, meaning he/she has a large number of friends, it may be meaningful to introduce him/her to a user.

Calling these functions by passing in a user id, they will return 10 recommendations for given user.

Link prediction:

From the perspective of Graph Theory, we can look at the set of users U and the places P, and the associate between them L, as a bipartite graph G =(U,P,L), where U intersect P is empty. All the edges are unweight and the graph represents the complete information of the input data, an edge e belongs L connect u in U and p in P means that the users u has checked in the place p. We can now apply the graph algorithm to gain insight into this graph. We can transform the problem to the link prediction problem, where a recommendation is a link that is likely to appear as the network evolves over time.

There are many ways to predict the connections for this bipartite graph.

Shortest path approach:

The idea is that we rank pairs (x,y) by the length of their shortest path in the graph. x in users U and y in places P. The approach follows the notion that collaboration networks are “small worlds,” where individuals are related through shortest path length. Let me give you an example, suppose we had three users A,B and C. A B both check in the place 1,2 and 3. C only check in at place 1 and 2. We will predict that C will also check in at place 3 in the future. Suppose we want to find out the score of A to 3, we can easily figure that the path C->1->A->3 or C->1->B->3… are all shortest path from C to 3. We will keep the shortest path (the length) as the score for this edge and sorted them and find the most likely path that C will go to for the next step.

Explain the code:

In the shortestpath.py file, we implement the link prediction using shortest path approach. First I build the adjacency list to store all the node and edges between them. In the build graph function, we will first read in the file, and build the bipartite graph using both the dictionary and list structure in python.

For the shortest path part, we can use Dijksta’s algorithm to compute every path between two nodes in the both sides, but we figure that for a particular user, we can just use the breakfast search approach to compute all the path from that user to every places nodes. The score for every path is the length between the origin and the destination.

After that we can sort according to the score we compute and keep the top results. We will use 50% data from a particular user as the training set and and 50% of the data as ground true so that we can use that to see if our prediction actually works. For each of them, we calculate recall and precision.

We will run this for every users in our data set and compute the total average recall and precision.

Result:

Recall: 1.3%

Precision: 1.6%

There are also many other ways to predict the link in this network, for example using the common neighbors, rooted PageRank, or SimRank. But none of them are very good for Link prediction in Bipartite Network. The following I will present how to use internal links to do link prediction in Bipartite Networks.

In the following, I will present how to use internal links to predict potential links in Bipartite Networks. I read some of the papers and get the idea of how to use link prediction on bipartite graphs.

There are two concepts that we should know about for the internal links method, the induced link and the internal link. First we begin with a bipartite graph G = (A,B,E) where A and B are the two distinct groups of the nodes and E is the set of edges where every edge (a,b) belongs E and a belongs to A and b belongs to B. In order to make the algorithm works, we need a projection graph, Gp = (V,Ep) where V belongs to A. we create Gp from, using the notion of induced links. An induced link (u,v) belongs to Ep where u belongs to V and A, and v is also belongs to V and A. The induced link is created when the condition that there exists i belongs to B where (u,i) belongs to E and (v,i) also belongs to E. In other words, an induced link is a link added to Gp between two nodes in the same subset of G, where both of the nodes have at least one edge to a node in the other subset of G. Gp is made entirely of induced links, and the represents the nodes within G that share endpoints.

After we get the projection graph Gp, the internal links can be identified. An internal link is a link (a,b) not belongs to E. a belongs to A and b belongs to B such that (a,b) were added, the induced links added to Gp would not changed. It means that an internal link is a link within the bipartite graph that does not add any new indeced links to a projection graph.

Next I will explain the detail of the codes(in the internallinks.py):

First of all, we will use the same Bipartite graph as before. This adjacency list graph will help us to the figure the link between user and the check-in places. After we get that, we can create the projection graph.

The algorithm goes like this, we loop through all the users and places they went to, for each user u1 and for each places p, we get which users u2 also link to places p, if u1!= u2, then we create the induced link, an edge within the Gp graph. Once we have iterated through each user, we create the entire graph.

Pgraph = {}

for u1 in user\_list:

for p in adj\_list[u1]:

for u2 in adj\_list[p]:

if u2 != u1:

addedge(Pgraph,u1,u2)

return Pgraph

Once we get the projection graph, we can weight the induced edges. Because in the final step, we need to score all the internal links and get our best prediction, and the score for the internal links depends on the weight of the induced edges. So in the next step, we need to figure out the weight for the induced edges. The method I used to sum up the number of common places by the two users.

The algorithm is very simple, we find out the intersection of two users’ places they went to and find the length of the result list. In the code, I will compare two users because I just want to keep one of the record in the result list.

def score(u1,u2,adj\_list,score\_list):

score = len(set(adj\_list[u1]) & set(adj\_list[u2]))

if u1 > u2:

temp = u1

u1 = u2

u2 = temp

score\_list[(u1,u2)] = score

return score

After we have created the weighted projection graph, we want to identify internal links by iterating through the users and places and find the links (user, place) that induce edges which already exist within Gp. This is the definition of internal link.

The algorithm goes like this, we loop through the user list, and for each user, we loop through the places, if the user did not went to a particular place p, we set the internal to True, we then check all the users that went to p, user\_p, if the user u and user\_p do not have a link in the Gp, we set the internal value to false, and break the loop. At the end, if the internal value is still true, we know that this is the internal link.

def internallinks(user\_list,venue\_list,adj\_list,Pgraph):

internal\_links = {}

for u in user\_list:

for p in venue\_list:

if p not in adj\_list[u]:

internal = True

for user\_p in adj\_list[p]:

if user\_p not in Pgraph[u]:

internal = False

break

if internal:

addedge(internal\_links,u,p)

return internal\_links

After we get all the internal nodes. We can score them, the way we score the internal link is to sum up all the induced links associate with this internal links score. What I mean by associate induced link is that for each internal link, we check other users lets say v that go to that place, if there is a link between user u and user v in Gp (induced link) then we add up the score.

def score\_internal\_link(score\_list,link,adj\_list):

place = link[1]

user = link[0]

score\_total = 0

for u in adj\_list[place]:

if u != user:

score\_total = score\_total+ score(u,user,adj\_list,score\_list)

return score\_total

After that we can sort according to the score we compute and keep the top results. We will use 50% data from a particular user as the training set and and 50% of the data as ground true so that we can use that to see if our prediction actually works. For each of them, we calculate recall and precision.

We will run this for every users in our data set and compute the total average recall and precision.

Result:

Recall: 16.32%

Precision: 24.78%