HW1 Electronic Commerce

Matan Birnboim - [redacted]

[redacted]

Imports and Setup

In [ ]:
```python
import numpy as np
import networkx as nx
import random
import pandas as pd

inst_minus_1 = pd.read_csv('instaglam_1.csv')
inst_0 = pd.read_csv('instaglam0.csv')
users = np.unique(inst_0['userID'].to_numpy(dtype=str))
users_num = users.shape[0]
```

In [ ]:
```python
def choose_artists():

  """
  Enter your ids below (if you are submitting alone DO NOT CHANGE ID2) and execute the code.
  The list of ids you get is the list of artists you need to promote.
  """

  ####################
  # TODO: change this
  ID1 = '[redacted]'
  ID2 = '[redacted]'
  ####################

  x = (int(ID1[-1]) + int(ID2[-1])) % 5
  y = (int(ID1[-2]) + int(ID2[-2])) % 5
  options = [(70, 150), (989, 16326), (144882, 194647), (389445, 390392), (511147, 532992)]
  y = (y + 1) % 5 if x == y else y
  print("your artists are:")
  artists = [*options[x], *options[y]]
  print(artists)

choose_artists()
```

```
your artists are:
[70, 150, 989, 16326]
```

In [ ]:
```python
artists = [70, 150, 989, 16326]

# Create graphs for times -1 and 0
G_minus_1 = nx.parse_edgelist(list(map(lambda pair: f"{pair[0]} {pair[1]}",inst_minus_1.values.tolist())), nodetype=str)
G_0 = nx.parse_edgelist(list(map(lambda pair: f"{pair[0]} {pair[1]}",inst_0.values.tolist())), nodetype=str)
```

Edge creation

Predicting Edge Creation

Let us examin the factors which might cause an edge to be or not to be created:

- $AC =$ The average degree centrality of both nodes
- $NC =$ Number of common neighbors
- $CNC =$ Common neighbor centrality of both nodes

In [ ]:
```
not_created = list(nx.difference(nx.complete_graph(G_0.nodes), G_0).edges) # List o
 edges that had not been created between times -1 and 0
created = list(nx.difference(G_0, G_minus_1).edges) # List of new edges between times
-1 and 0
new_edges_num = len(created) # Number of edges added between time -1 and 0

"""We will use logistic regression, therefore we should avoid label inbalance, we wil
l randomly choose a number of edges that werent created
   that will prevent the label imbalance"""
rnd_idxs = np.random.randint(0, len(not_created), size=new_edges_num * 2)
not_created_reduced = np.array([not_created[i] for i in rnd_idxs])
not_created_reduced = list(map(lambda x: [0, x[0], x[1]], not_created_reduced))
created = list(map(lambda x: [1, x[0], x[1]], created))

"""edge_data -> A list of lists where every sub list is: [was it created in time 0?
 (bool), 1st node name (str), 2nd node name (str)].
               every sublist represents an edge."""
edge_data = np.concatenate((not_created_reduced, created))
```

In [ ]:
```
def from_e_list_to_cnc(G, e_list):
    return dict(list(map(lambda x: ((x[0], x[1]), x[2]),nx.common_neighbor_centrality(G, e_list))))


def AC_NC_CNC(G, u, v, dcs, cnc):
    """for nodes u and v in graph G it returns the below list:
       [NC, AC, CNC]

    Args:
        G (nx.graph): A graph
        u (str): A node's name
        v (str): A node's name
        dcs (dict): A dictionary containing all nodes which holds the degree centrality of each node
        cnc (dict): A dictionary containing all edges where every edge's value is the common neighbor centrality of both nodes in it

    Returns:
        list: [NC, AC, CNC]
    """
    return len(set(nx.neighbors(G, u)).intersection(set(nx.neighbors(G, v)))), (dcs[u] + dcs[v])/2, cnc[(u, v)]

dcs_minus1 = nx.degree_centrality(G_minus_1)
cnc_minus1 = from_e_list_to_cnc(G_minus_1, list(map(lambda x: (x[1], x[2]), edge_data)))
cnc_minus1_max = max(cnc_minus1.values())

"""For each edge in edge data create: [was it created? (bool), NC (int), AC (float), CNC (float)]"""
X = np.array([[edge[0] , AC_NC_CNC(G_minus_1, edge[1], edge[2], dcs_minus1, cnc_minus1)[0], AC_NC_CNC(G_minus_1, edge[1], edge[2], dcs_minus1, cnc_minus1)[1], AC_NC_CNC(G_minus_1, edge[1], edge[2], dcs_minus1, cnc_minus1)[2]] for edge in edge_data], dtype=float)

X_data = X[:, 1:] # Extract data
X_data[:, 2] = np.log(X_data[:, 2]+0.001) # Turn the CNC to log(CNC) for normalization purposes
y_data = X[:,0] # Extract labels
```

Now we will fit a logistic regression model to predict to probability of an edge to be created

- $P_{(u,v)} = Prob(\text{The edge (u,v) will be created in the next time})$
- $\theta = \text{LR model output} \in R^3$
- $AC = \text{The average degree centrality of both nodes}$
- $NC = \text{Number of common neighbors}$
- $LCNC = \log(\text{Common neighbor centrality of both nodes})$

$$P_{(u,v)} = \frac{e^{\theta^T(PC,NC,LCNC)}}{1+e^{\theta^T(PC,NC,LCNC)}}$$

In [ ]:
```python
def logistic_function(x):
    return np.exp(x) / (1 + np.exp(x))


def compute_cost(theta, x, y):
    m = len(y)
    y_pred = logistic_function(np.dot(x , theta)) >= 1
    cost = np.multiply(y, y_pred) - np.sum(np.log(1+np.exp(y_pred)))
    # FIX MULTIPLY
    gradient = 1 / m * np.dot(x.transpose(), (y_pred - y))
    return cost , gradient

def gradient_descent(x, y, theta, alpha, iterations):
    costs = []
    for i in range(iterations):
        cost, gradient = compute_cost(theta, x, y)
        theta -= (alpha * gradient)
        costs.append(cost)
    return theta, costs

theta_init = np.random.random(X_data.shape[1])
theta, costs = gradient_descent(X_data, y_data, theta_init, 0.002, 4000)
print(theta)
def predict(theta, x):
    y_pred = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        if logistic_function(np.dot(x[i], theta)) >= 1:
            y_pred[i] = 1
    return y_pred
p = predict(theta, X_data)

def compute_stats(original, predicted):
    # F1 computation
    cnt_spec = 0
    cnt = 0
    for i,j in zip(original, predicted):
        if i==1 and j == 1:
            cnt_spec += 1
            cnt += 1
        if i == 0 and j==1:
            cnt += 1

    precision = cnt_spec / cnt

    cnt2_spec = 0
    cnt2 = 0
    for i,j in zip(original, predicted):
        if i==1 and j == 1:
            cnt2_spec += 1
        if i == 1:
            cnt2 += 1

    recall = cnt2_spec / cnt2
    return precision, recall

precision, recall = compute_stats(y_data, p)
```

LR output: $\theta =$ [3.93487307 0.24281832 5.62844561]

F1: 0.9150943396226415

Train Accuracy: 94.39950217797137 %

For every time t we will create another graph called $G\_t\_to\_add$.

$G\_t\_to\_add$ will contain every edge that does not exist in time t and and indicator determining wether the edge will be created in the next time (according to the logistic regression model above).

In [ ]:
```
def possible_edge_indc(G_t):
    """Create a dictionary that contains for each possible edge that had not been created an indicator
        determining if it will be created in the next time

    Args:
        G_t (nx.graph): The graph

    Returns:
        dict: for each edge: key = the edge, value = an indicator determining if it will be created in the next time
    """
    G_t_to_add = nx.difference(nx.complete_graph(G_t.nodes()), G_t)
    bc = nx.betweenness_centrality(G_t, normalized=True)
    dcs_t = nx.degree_centrality(G_t)
    edges_indc_dict = {edge: np.random.rand() <= logistic_function(np.dot(np.array([len(set(nx.neighbors(G_t, edge[0])).intersection(set(nx.neighbors(G_t, edge[1])))) ,
                                                                                              (dcs_t[edge[0]] + dcs_t[edge[1]])/2,
                                                                                              max([bc[edge[0]], bc[edge[1]]]), 1])
                                                                                   , theta))
                           for edge in list(G_t_to_add.edges())}
    return edges_indc_dict
eid = possible_edge_indc(G_minus_1)
```

Improved Triadic Closure Attempt

- $\forall k \in N : Prob$ (edge (u,v) will be created in the next time | u and v have k common friends)

= Proportion of edge that were created between nodes that had k common neighbors between times -1 and 0

- Let T(k) be the fraction of these pairs that have formed an edge by time 0. This is an empirical estimate for the probability that a link will form between two people with k friends in common

In [ ]:
```
import numpy as np
import networkx as nx
import random
import pandas as pd

inst_minus_1 = pd.read_csv('instaglam_1.csv')
inst_0 = pd.read_csv('instaglam0.csv')
users = np.unique(inst_0['userID'].to_numpy(dtype=str))
users_num = users.shape[0]


# Create graphs for times -1 and 0
G_minus_1 = nx.parse_edgelist(list(map(lambda pair: f"{pair[0]} {pair[1]}",inst_minus_1.values.tolist())), nodetype=str)
G_0 = nx.parse_edgelist(list(map(lambda pair: f"{pair[0]} {pair[1]}",inst_0.values.tolist())), nodetype=str)
```

In [ ]:

```
artists = [70, 150, 989, 16326]
nx.set_node_attributes(G_0, 0, name='Nt') # Number of neighbors in time t
nx.set_node_attributes(G_0, 0, name='Bt') # Number of neighbors who have the
nx.set_node_attributes(G_0, {}, name='h')
nx.set_node_attributes(G_0, False, name='bought')

df = pd.read_csv('spotifly.csv')
df['tup'] = list(zip(df[' artistID'], df['#plays']))
spotifly_dict = {k: dict(v) for k, v in df.groupby('userID')['tup'].apply(list).to_di
ct().items()}
def update_nodes_attributes(G_t):
    Bt_dict = {node: sum(
            [G_t.nodes[nei]['bought'] for nei in list(nx.neighbors(G_t, node))])
        for node in list(G_t.nodes)}
    nx.set_node_attributes(G_t, Bt_dict, name='Bt')

update_nodes_attributes(G_0)

for node in list(G_0.nodes):
    G_0.nodes[node]['Nt'] = len(list(G_0.neighbors(node)))
    G_0.nodes[node]['h'] = {
        artist: spotifly_dict[int(node)][artist] if artist in spotifly_dict[int(node
)].keys() else 0 for artist in artists}
```

In [ ]:

```
def edge_creation_probs(not_created_and_common, created_and_common):
  not_created_df = pd.DataFrame(not_created_and_common, columns=['edge', 'common nei
 num'])
  created_df = pd.DataFrame(created_and_common, columns=['edge', 'common nei num'])

  not_created_count = not_created_df.groupby(['common nei num'])['common nei num'].co
unt().to_frame()
  created_count = created_df.groupby(['common nei num'])['common nei num'].count().to
_frame()

  not_created_count = not_created_count.rename({'common nei num': 'not_created_cnt'},
axis='columns')
  created_count = created_count.rename({'common nei num': 'created_cnt'}, axis='colum
ns')
  joined_nei_cnt = not_created_count.join(created_count)
  joined_nei_cnt['created_cnt'] = joined_nei_cnt['created_cnt'].fillna(0)
  joined_nei_cnt['edge_prob'] = joined_nei_cnt['created_cnt'] / (joined_nei_cnt['crea
ted_cnt'] + joined_nei_cnt['not_created_cnt'])
  joined_nei_cnt = joined_nei_cnt.drop(['created_cnt', 'not_created_cnt'], axis=1)
  output = joined_nei_cnt.to_dict()
  return output['edge_prob']
```

In [ ]:
```
def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return array[idx]

def create_edges(G, probs_dict):
  node_with_new_neighbors = []
  can_be_created = list(nx.difference(nx.complete_graph(G), G).edges) # Find all edge
s that had not been created yet
  not_created = can_be_created.copy()
  created = []

  for edge in can_be_created:
    common_nei_num = len(set(nx.neighbors(G, edge[0])).intersection(set(nx.neighbors(
G, edge[1])))) # Number of common neighbors between the 2 edge nodes
    common_nei_closest = find_nearest(list(probs_dict.keys()), common_nei_num) # For
 every number of edges that is not on the probability dictionary, assign the probabil
ity of the nearest number of neighbors value
    indc = np.random.rand() # Simulate the probability
    if indc < probs_dict[common_nei_closest]:
      node_with_new_neighbors.append(edge[0]) # Append both nodes to the list of node
s with new neighbors
      node_with_new_neighbors.append(edge[1])
      G.add_edge(edge[0], edge[1]) # add the edge to the graph
      created.append(edge)
      not_created.remove(edge)

  not_created_and_common = [(edge, len(set(nx.neighbors(G, edge[0])).intersection(set
(nx.neighbors(G, edge[1]))))) for edge in not_created] # [(edge, number of common nei
ghbors between the edge nodes),.... for every edge that was not created]
  created_and_common = [(edge, len(set(nx.neighbors(G, edge[0])).intersection(set(nx.
neighbors(G, edge[1]))))) for edge in created] # [(edge, number of common neighbors b
etween the edge nodes),.... for every edge that was created]
  return node_with_new_neighbors, edge_creation_probs(not_created_and_common, created
_and_common)
```

Conclusion

The improved triadic closure and the logistic regression model produced similar accuracy on the given graphs with a slight advantage to the improved triadic closure. Therfore, we chose to use the improved triadic closure method which also was much faster.

Influencers selection model

We will use the greedy hill climbing algorithm as follows:

- Define the final influencers list
- for influencer_num in 1:5:

  A. for person in graph:

    1. infect the person
    2. for every simulation:

       2.1. for t in 1:6:
       - Update Bt and Nt for all nodes
       - Start infection proccess for the current time according to the probability mentioned in the exercise
       - Create edges according to the edge creation model

  B. Choose the person with that produces the biggest number of infeted nodes and append him to the influencers list

In [ ]:
```
def initialize(G_0, G_minus_1):
  not_created = list(nx.difference(nx.complete_graph(G_0.nodes), G_0).edges) # List o
f edges that had not been created between times -1 and 0
  created = list(nx.difference(G_0, G_minus_1).edges) # List of new edges between tim
es -1 and 0

  not_created_and_common = [(edge, len(set(nx.neighbors(G_minus_1, edge[0])).intersec
tion(set(nx.neighbors(G_minus_1, edge[1]))))) for edge in not_created]
  created_and_common = [(edge, len(set(nx.neighbors(G_minus_1, edge[0])).intersection
(set(nx.neighbors(G_minus_1, edge[1]))))) for edge in created]
  return edge_creation_probs(not_created_and_common, created_and_common)
```

In [ ]:
```
def infect(G, artist):
    infected_nodes = []
    for person in list(G.nodes()): # Run over all nodes and try to infect them
        if G.nodes[person]['bought']: continue # Each person can buy only once
        prob = float(G.nodes[person]['Bt']) / float(G.nodes[person]['Nt']) # The defa
ult probability is Bt/Nt
        if G.nodes[person]['h'][artist] > 0:
            prob = prob * float(G.nodes[person]['h'][artist]) / 1000 # If the person
 listened to the artist's songs' the probability should be Bt*h/(Nt*1000)
        rnd = np.random.rand() # Simulate probability
        if rnd <= prob:
            G.nodes[person]['bought'] = True # infect the person
            infected_nodes.append(person) # add the person to the newly infected list
    return infected_nodes
```

Simplifying Assumption

After multiple attemps of running the code we noticed that an insignificant number of edges is created in each iteration of every simulation in the graph. Furthermore, the edge creation proccess was way too long to iterate over all people 5 times for every artist. So we decided to ignore the edge creation proccess and apply only the infection proccess which highly improved the running time of the script.

Code witout edge creation (That we ran in practice)

In [ ]:
```python
nodes_with_new_neighbors_init = list(sum(list(nx.difference(G_0, G_minus_1).edges(
()))

artists = [70, 150, 989, 16326]
def find_my_influencers_without_edge_creation(artists):
  output = {artist: [] for artist in artists}
  for artist in artists: # Choose influencers for every artist
    influencers = []
    num_simulations = 10
    for inf_num in range(5): # Find 5 best influencers
        print(f"artist: {artist}, influ: {inf_num}")
        best_current_influencer = 0
        max_infection = 0
        i = 0
        for person in list(G_0.nodes()): # Out of all people find the current best influencer combined with the influencers that were already chosen
            if i % 100 == 0:
              print(i)
            i += 1
            if person in influencers: continue # If we encounter a person that was already chosen, skip him
            G_t = G_0.copy() # Create a new copy of the graph in time 0
            G_t.nodes[person]['bought'] = True # Infect the current person checked

            for nei in list(G_t.neighbors(person)): # Infecting all neighbors of the current person
                G_t.nodes[nei]['Bt'] += 1

            all_sim_infection_sum = 0 # Sum the number of infected people by the checked person in all simulations

            nodes_with_new_neighbors = nodes_with_new_neighbors_init # All nodes with new neighbors between times -1 and 0
            for sim in range(num_simulations): # The proccess is stochastic, therefore, we run multiple simulations to get closer to the expectation
              for t in range(6): # Simulate the infection proccess combined with the edge creation proccess

                  for node in infect(G_t, artist): # Update Bt of all nodes and infect
                    G_t.nodes[node]['Bt'] = sum([G_t.nodes[nei]['bought'] for nei in list(G_t.neighbors(node))])

              all_sim_infection_sum += sum([G_t.nodes[person]['bought'] for person in list(G_t.nodes())])

            if max_infection < all_sim_infection_sum / num_simulations:  # If the current person achieved better average infection than the current best, make him the current best influencer
                best_current_influencer = person
                max_infection = all_sim_infection_sum / num_simulations
        print(best_current_influencer)
        influencers.append(best_current_influencer) # Add the best
        G_0.nodes[best_current_influencer]['bought'] = True
    for inf in influencers: # return the graph at time 0 to the initial state (in term of infected nodes)
      G_0.nodes[best_current_influencer]['bought'] = False
    output[artist] = influencers
  return output
```

Code with edge creation

In [ ]:

```
nodes_with_new_neighbors_init = list(sum(list(nx.difference(G_0, G_minus_1).edges()
()))

artists = [150, 989]
def find_my_influencers(artists):
  output = {artist: [] for artist in artists}
  for artist in artists: # Choose influencers for every artist
    influencers = []
    num_simulations = 10
    probs_0 = initialize(G_0, G_minus_1) # Initialize the probabilities for edge crea
tion according to the graph in time 0
    for inf_num in range(5): # Find 5 best influencers
        print(f"artist: {artist}, influ: {inf_num}")
        best_current_influencer = 0
        max_infection = 0
        person_list = list(G_0.nodes())
        random.shuffle(person_list)
        i = 0
        for person in person_list: # Out of all people find the current best influenc
er combined with the influencers that were already chosen
            if i % 100 == 0:
              print(i)
            i += 1
            if person in influencers: continue # If we encounter a person that was al
ready chosen, skip him
            G_t = G_0.copy() # Create a new copy of the graph in time 0
            G_t.nodes[person]['bought'] = True # Infect the current person checked

            for nei in list(G_t.neighbors(person)): # Infecting all neighbors of the
 current person
                G_t.nodes[nei]['Bt'] += 1

            all_sim_infection_sum = 0 # Sum the number of infected people by the chec
ked person in all simulations

            nodes_with_new_neighbors = nodes_with_new_neighbors_init # All nodes with
new neighbors between times -1 and 0
            for sim in range(num_simulations): # The proccess is stochastic, therefor
e, we run multiple simulations to get closer to the expectation
              probs_t = probs_0.copy()
              for t in range(6): # Simulate the infection proccess combined with the
 edge creation proccess
                  for node in nodes_with_new_neighbors: # Update Nt  of all relevant
 nodes
                      G_t.nodes[node]['Nt'] = G_t.nodes[node]['Nt'] + 1

                  for node in infect(G_t, artist): # Update Bt of all nodes and infec
t
                    G_t.nodes[node]['Bt'] = sum([G_t.nodes[nei]['bought'] for nei in
list(G_t.neighbors(node))])

                  nodes_with_new_neighbors, probs_t = create_edges(G_t, probs_t)
              all_sim_infection_sum += sum([G_t.nodes[person]['bought'] for person in
list(G_t.nodes())])

            if max_infection < all_sim_infection_sum / num_simulations:  # If the cur
rent person achieved better average infection than the current best, make him the cur
rent best influencer
                best_current_influencer = person
                max_infection = all_sim_infection_sum / num_simulations
        print(best_current_influencer)
        influencers.append(best_current_influencer) # Add the best
        G_0.nodes[best_current_influencer]['bought'] = True
    for inf in influencers: # return the graph at time 0 to the initial state (in ter
m of infected nodes)
```

```
        G_0.nodes[best_current_influencer]['bought'] = False
      output[artist] = influencers
    return output
```

In [ ]:

```
output = find_my_influencers(artists)
print(output)

with open('313358343_212724462.csv', 'w') as f:
    f.write("artist Id,influencer 1,influencer 2,influencer 3,influencer 4,influencer 5\n")
    for key in output.keys():
        print(key, output[key])
        f.write("%s,%s,%s,%s,%s,%s\n"%(key,output[key][0],output[key][1],output[key][2],output[key][3],output[key][4]))
```