```
import networkx as nx
G=nx.Graph()


G=nx.read_adjlist("/content/Sena_encoded_set.txt")



print(len(G.nodes))
print(len(G.edges))
```

```
7542
9837
```

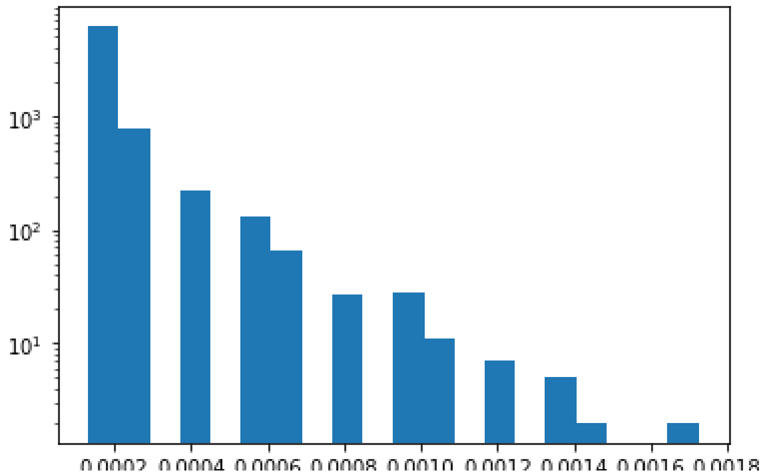
```
for nodes in G.nodes():
  if(nodes[:1]=='h'):
    G.nodes[nodes]['bipartite'] = 'user'
  else:
    G.nodes[nodes]['bipartite'] = 'hotel'
```

```
def get_nodes_from_partition(G,partition):
    # Initialize an empty list for nodes to be returned
    nodes = []
    # Iterate over each node in the graph G
    for n in G.nodes():
        # Check that the node belongs to the particular partition
        if G.nodes[n]['bipartite'] == partition:
            # If so, append it to the list of nodes
            nodes.append(n)
    return nodes
# Print the number of nodes in the 'users' partition
print(len(get_nodes_from_partition(G, 'user')))
print(len(get_nodes_from_partition(G, 'hotel')))
```

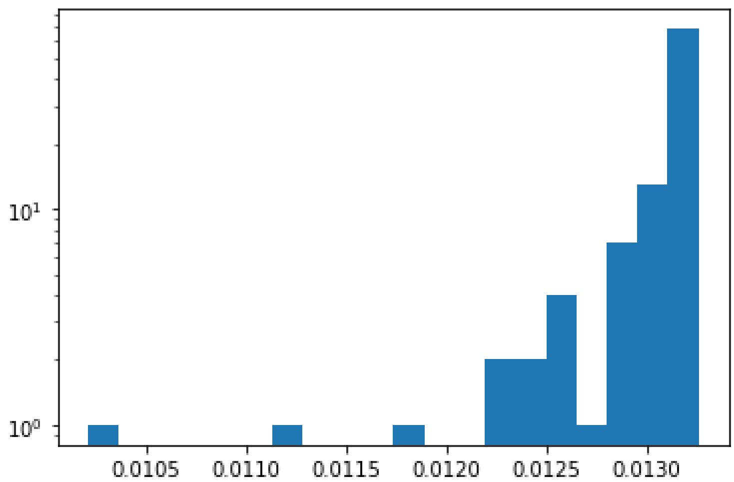
```
7442
100
```

```
# Import matplotlib
import matplotlib.pyplot as plt

# Get the 'users' nodes: user_nodes
user_nodes = get_nodes_from_partition(G,'user')

# Compute the degree centralities: dcs
dcs = nx.degree_centrality(G)

# Get the degree centralities for user_nodes: user_dcs
user_dcs = [dcs[n] for n in user_nodes]

# Plot the degree distribution of users_dcs
plt.yscale('log')
```

```
plt.hist(user_dcs, bins=20)
plt.show()
```

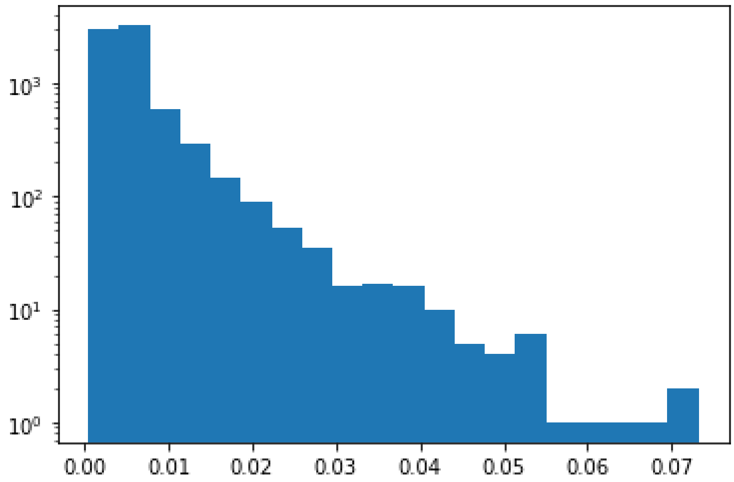



```
# Import matplotlib
import matplotlib.pyplot as plt

# Get the 'users' nodes: user_nodes
project_nodes = get_nodes_from_partition(G,'hotel')

# Compute the degree centralities: dcs
dcs = nx.degree_centrality(G)

# Get the degree centralities for user_nodes: user_dcs
project_dcs = [dcs[n] for n in project_nodes]

# Plot the degree distribution of users_dcs
plt.yscale('log')
plt.hist(project_dcs, bins=20)
plt.show()
```

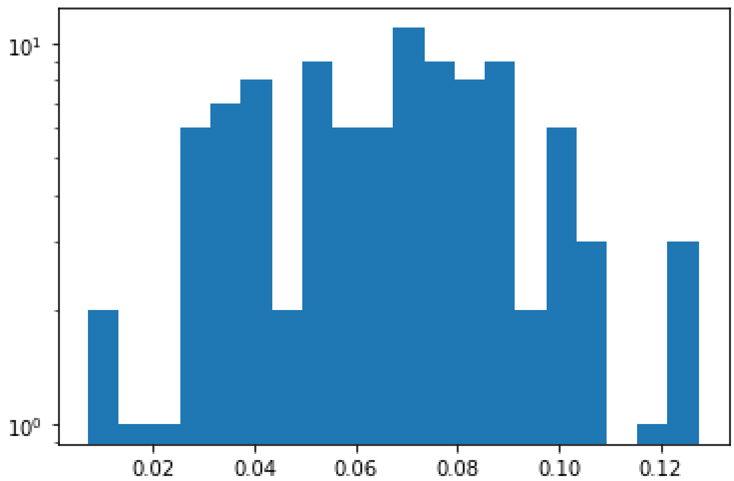



```
user_nodes = get_nodes_from_partition(G,'user')

# Compute the degree centralities: dcs
centrality = nx.eigenvector_centrality(G)

# Get the degree centralities for user_nodes: user_dcs
user_dcs = [centrality[n] for n in user_nodes]
```

```
# Plot the degree distribution of users_dcs
plt.yscale('log')
plt.hist(user_dcs, bins=20)
plt.show()
```

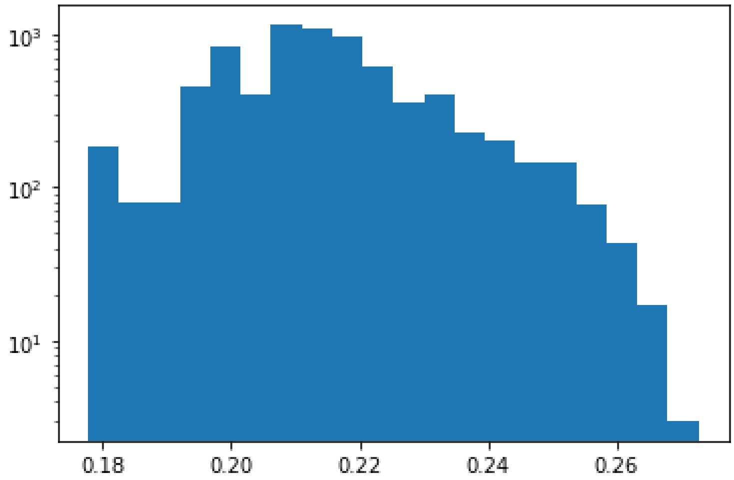



```
project_nodes = get_nodes_from_partition(G,'hotel')

# Compute the degree centralities: dcs
dcs = nx.eigenvector_centrality(G)

# Get the degree centralities for user_nodes: user_dcs
project_dcs = [dcs[n] for n in project_nodes]

# Plot the degree distribution of users_dcs
plt.yscale('log')
plt.hist(project_dcs, bins=20)
plt.show()
```



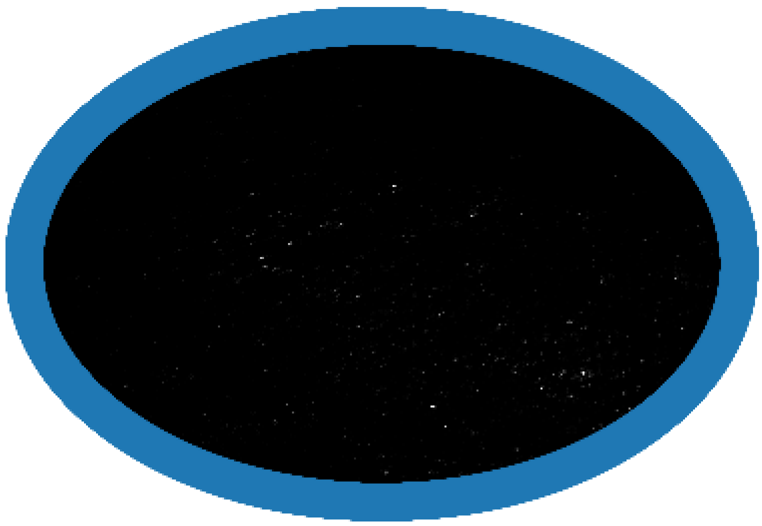

```
user_nodes = get_nodes_from_partition(G,'user')

# Compute the degree centralities: dcs
centrality = nx.closeness_centrality(G)

# Get the degree centralities for user_nodes: user_dcs
```

```
user_dcs = [centrality[n] for n in user_nodes]

# Plot the degree distribution of users_dcs
plt.yscale('log')
plt.hist(user_dcs, bins=20)
plt.show()
```



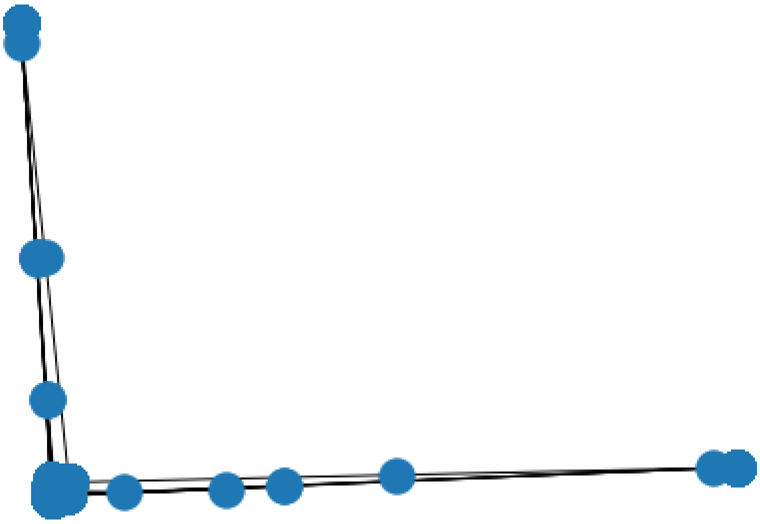

```
pos = nx.bipartite_layout(G,G.nodes())
```

```
nx.draw(G, pos=nx.circular_layout(G,))
```

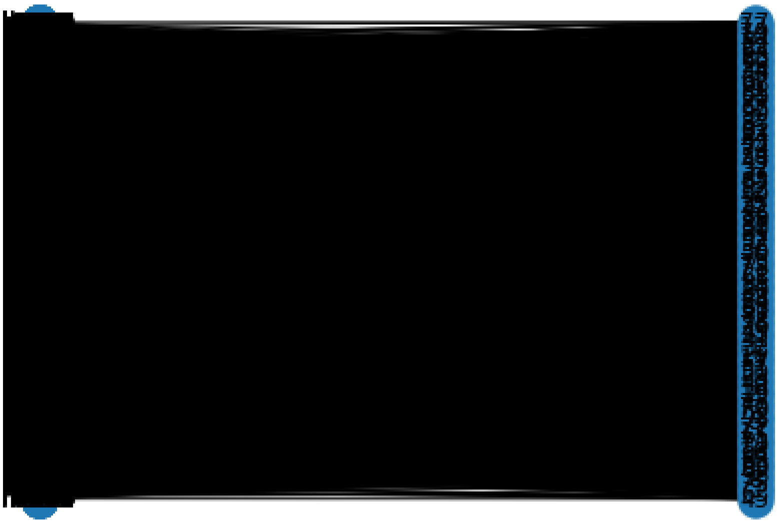


```
nx.draw(G,pos=nx.random_layout(G))
```

```
nx.draw(G,pos=nx.spectral_layout(G))
```



```
top = nx.bipartite.sets(G)[0]
pos = nx.bipartite_layout(G, top)
nx.draw(G, pos, with_labels=True)
```



```
def shared_partition_nodes(G, node1, node2):
    # Check that the nodes belong to the same partition
    assert G.nodes[node1]['bipartite'] == G.nodes[node2]['bipartite']
```

```
    # Get neighbors of node 1: nbrs1
    nbrs1 = G.neighbors(node1)
    # Get neighbors of node 2: nbrs2
    nbrs2 = G.neighbors(node2)

    # Compute the overlap using set intersections
    overlap = set(nbrs1).intersection(nbrs2)
    return overlap
def user_similarity(G, user1, user2, proj_nodes):
    # Check that the nodes belong to the 'users' partition
    assert G.nodes[user1]['bipartite'] == 'user'
    assert G.nodes[user2]['bipartite'] == 'user'

    # Get the set of nodes shared between the two users
    shared_nodes = shared_partition_nodes(G, user1, user2)

    # Return the fraction of nodes in the projects partition
    return len(shared_nodes) / len(proj_nodes)

# Compute the similarity score between users 'u4560' and 'u1880'
project_nodes = get_nodes_from_partition(G, 'hotel')
similarity_score = user_similarity(G, 'h4560', 'h7295', project_nodes)

print(similarity_score)
```

```
0.01
```

```
from collections import defaultdict

def most_similar_users(G, user, user_nodes, proj_nodes):
    # Data checks
    assert G.nodes[user]['bipartite'] == 'user'

    # Get other nodes from user partition
    user_nodes = set(user_nodes)
    user_nodes.remove(user)

    # Create the dictionary: similarities
    similarities = defaultdict(list)
    for n in user_nodes:
        similarity = user_similarity(G, user, n, proj_nodes)
        similarities[similarity].append(n)

    # Compute maximum similarity score: max_similarity
    max_similarity = max(similarities.keys())

    # Return list of users that share maximal similarity
    return similarities[max_similarity]

user_nodes = get_nodes_from_partition(G, 'user')
project_nodes = get_nodes_from_partition(G, 'hotel')

print(most_similar_users(G, 'h4560', user_nodes, project_nodes))
```

```
['h5569', 'h406', 'h2520', 'h3121', 'h553', 'h3872', 'h1408', 'h3667', 'h4633', 'h40
```

```
def recommend_hotel(G, from_user, to_user):
    # Get the set of hotels that from_user has rated
    from_hotel = set(G.neighbors(from_user))
    # Get the set of hotels that to_user has rated
    to_hotel = set(G.neighbors(to_user))

    # Identify hotels that the from_user is connected to that the to_user is not connectec
    return from_hotel.difference(to_hotel)

# Print the repositories to be recommended
print(recommend_hotel(G, 'h790', 'h2148'))
```

```
{'3'}
```

```
user_l=get_nodes_from_partition(G, 'user')
hotel_l=get_nodes_from_partition(G, 'hotel')
for i in range(20):
  preds = nx.preferential_attachment(G,[(user_l[i],hotel_l[i])])
  for u, v, p in preds:
    print(f"({u}, {v}) -> {p}")
```

```
(h4976, 16) -> 100
(h766, 67) -> 100
(h955, 35) -> 100
(h6595, 75) -> 100
(h1617, 64) -> 300
(h3442, 84) -> 100
(h2291, 99) -> 95
(h5322, 76) -> 99
(h344, 45) -> 100
(h5978, 23) -> 98
(h5495, 74) -> 98
(h6272, 13) -> 99
(h2824, 4) -> 100
(h4014, 88) -> 200
(h2513, 3) -> 100
(h99, 47) -> 100
(h5839, 62) -> 300
(h6192, 41) -> 94
(h2025, 0) -> 100
(h6449, 66) -> 97
```

```
for i in range(20):
  preds = nx.jaccard_coefficient(G,[(user_l[i],user_l[i+1])])
  for u, v, p in preds:
    print(f"({u}, {v}) -> {p}")
```

```
(h4976, h766) -> 1.0
(h766, h955) -> 1.0
(h955, h6595) -> 1.0
```

```
    (h6595, h1617) -> 0.3333333333333333
    (h1617, h3442) -> 0.3333333333333333
    (h3442, h2291) -> 1.0
    (h2291, h5322) -> 1.0
    (h5322, h344) -> 1.0
    (h344, h5978) -> 1.0
    (h5978, h5495) -> 1.0
    (h5495, h6272) -> 1.0
    (h6272, h2824) -> 1.0
    (h2824, h4014) -> 0.5
    (h4014, h2513) -> 0.5
    (h2513, h99) -> 1.0
    (h99, h5839) -> 0.3333333333333333
    (h5839, h6192) -> 0.3333333333333333
    (h6192, h2025) -> 1.0
    (h2025, h6449) -> 1.0
    (h6449, h2887) -> 1.0
```

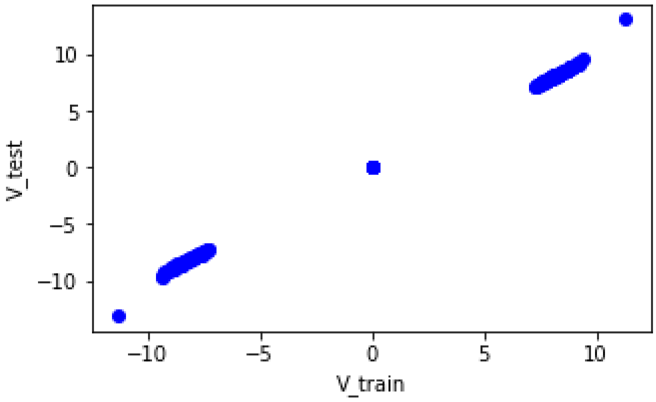
```
import random
import numpy as np
import scipy as sc
import matplotlib.pyplot as plt
from sklearn import metrics
from scipy.optimize import curve_fit
from tqdm import tqdm
```

```
A = nx.adjacency_matrix(G)
A = A.toarray()
proportion_edges = 0.3
edge_subset = random.sample(G.edges(), int(proportion_edges * G.number_of_edges()))
G_train = G.copy()
G_train.remove_edges_from(edge_subset)
A_train = nx.adjacency_matrix(G_train)
A_train = A_train.toarray()
G_test = nx.Graph()
G_test.add_edges_from(edge_subset)
#--------------
# eigenvalue decomposition
V_train, U_train = np.linalg.eig(A_train)
# U.T * Atest * U
target_V = U_train.T @ A @ U_train
# take only the diagonals
target_V = np.diag(target_V)
# plot the pattern
plt.figure(figsize=(5, 3))
plt.xlabel("V_train")
plt.ylabel("V_test")
plt.scatter(V_train, target_V, c='b')
plt.show()
#------
class OddPathCountingKernel:
    def __init__(self):
        self.a1 = 0
        self.a3 = 0
        self.a5 = 0
```

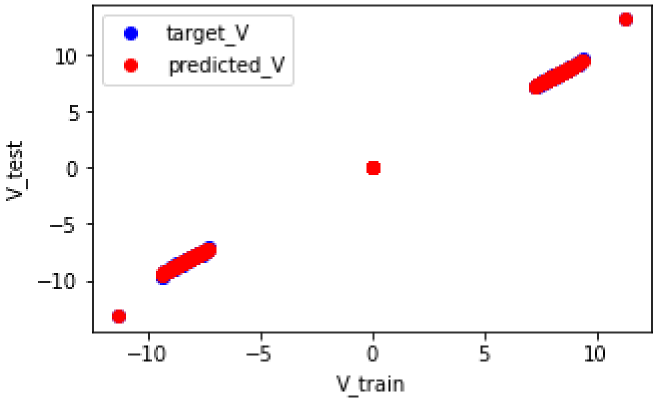
```
        self.a7 = 0

    def func(self, V, a1, a3, a5, a7):
        return V * a1 + V**3 * a3 + V**5 * a5 + V**7 * a7
    def fit(self, V_train, target_V):
        # do curve fitting
        popt, pcov = curve_fit(self.func, V_train, target_V)
        self.a1, self.a3, self.a5, self.a7 = popt
    def pred(self, V_train):
        return self.func(V_train, self.a1, self.a3, self.a5, self.a7)
#----------------
class SinhPseudokernel:
    def __init__(self):
        self.alpha = 0

    def func(self, V, alpha):
        return np.array([
            alpha * (np.exp(lamb) - np.exp(-lamb)) for lamb in V
        ])
    def fit(self, V_train, target_V):
        # do curve fitting
        popt, pcov = curve_fit(self.func, V_train, target_V)
        self.alpha, = popt
    def pred(self, V_train):
        return self.func(V_train, self.alpha)
#------------------
class OddNeumannPseudokernel:
    def __init__(self):
        self.alpha = 0

    def func(self, V, alpha):
        return np.array([
            alpha * (1/(1-lamb) - 1/(1+lamb)) for lamb in V
        ])
    def fit(self, V_train, target_V):
        # do curve fitting
        popt, pcov = curve_fit(self.func, V_train, target_V)
        self.alpha, = popt
    def pred(self, V_train):
        return self.func(V_train, self.alpha)
# fit kernel function
for kernel in [OddPathCountingKernel(), SinhPseudokernel(), OddNeumannPseudokernel()]:
    print(kernel)
    # fit the kernel to the data
    kernel.fit(V_train, target_V)
    # predict the output
    V_pred = kernel.pred(V_train)
    # assume our function is exponential V_train with alpha = 0.6
    plt.figure(figsize=(5, 3))
    plt.xlabel("V_train")
    plt.ylabel("V_test")
    plt.scatter(V_train, target_V, c='b', label="target_V")
    plt.scatter(V_train, V_pred, c='r', label="predicted_V")
    plt.legend()
    plt.show()
```

```
    # transformation
    Apred = U_train @ np.diag(V_pred) @ U_train.T
    Apred = Apred.real
    # make edges of prediction
    pred = [(i, j, Apred[i, j]) for i in range(Apred.shape[0]) for j in range(Apred.shape[
    # create graph
    G_pred = nx.Graph()
    G_pred.add_weighted_edges_from(pred)
#---------
```

```
/usr/local/lib/python3.7/dist-packages/matplotlib/collections.py:153: ComplexWarning
  offsets = np.asanyarray(offsets, float)
```




```
/usr/local/lib/python3.7/dist-packages/numpy/lib/function_base.py:486: ComplexWarnin
  a = asarray(a, dtype=dtype, order=order)
<__main__.OddPathCountingKernel object at 0x7f83a61a43d0>
```



10m 44s completed at 11:28 AM