ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
Τμήμα Πληροφορικής

Εργασία Μαθήματος ***«Ανάλυση Κοινωνικών Δικτύων (6ο εξ)»***

Όνομα φοιτητή	Μίμογλου Γεώργιος
Αρ. Μητρώου	Π17073
Ημερομηνία παράδοσης	17/7/2020

Εισαγωγή

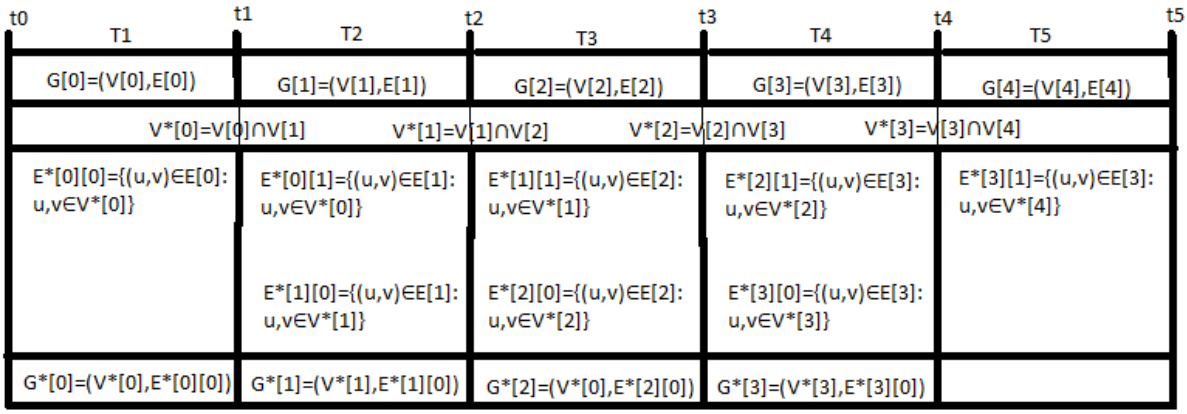
Δομές δικτύων εμφανίζονται από τα αρχαία χρόνια, αλλά η μελέτη τους ξεκινάει από το 1736 όταν ο Euler όρισε και έλυσε το. “Seven Bridges problem of Königsberg”. Έκτοτε, η μελέτη τους συνεχίστηκε από μαθηματικούς δημιουργώντας σιγά-σιγά την Θεωρία των Γράφων. Ύστερα, αναπτύχθηκαν τα complex networks, όμως παρέμειναν στα «χαρτιά», αφού τότε δεν υπήρχαν τα απαραίτητα πραγματικά δεδομένα. Φτάνουμε στο σήμερα, όπου με την βοήθεια των κοινωνικών δικτύων έχουμε πλέον τα κατάλληλα δεδομένα και η υπάρχουσα θεωρία άρχισε να ενσαρκώνεται. Ένα σημαντικό πεδίο των complex networks αποτελεί το Link Prediction. Ο χαρακτηρισμός «σημαντικό» δεν είναι καθόλου τυχαίος αν παρατηρήσουμε τις εφαρμογές που έχει το εν λόγω πεδίο. Πληροφορική, βιολογία, ψυχολογία, κοινωνιολογία, εγκληματολογία είναι μερικές επιστήμες που μελετούν δίκτυα και προβλέπουν την εξέλιξή τους μέσα στο χρόνο.

Σκοπός της εργασίας

Με αυτή την εργασία, οι φοιτητές έρχονται σε επαφή με έννοιες και εργαλεία για την μελέτη κοινωνικών δικτύων. Πιο συγκεκριμένα, ζητείται ο υπολογισμός μέτρων κεντρικότητας και η πρόβλεψη δημιουργίας ακμών, σε σύνολα δικτύων αλληλεπιδράσεων χρηστών σε διαδοχικές χρονικές στιγμές. Η εκφώνηση καθοδηγεί τον φοιτητή ώστε να μοντελοποιήσει το πρόβλημα και δημιουργήσει τις κατάλληλες δομές.

Δομές και ορισμοί

Μπορούμε να κατανοήσουμε καλύτερα τις δομές και τις έννοιες που χρησιμοποιούνται μέσω ενός παραδείγματος. Για ευκολία, Θέτουμε N = 5.



Όπου:

- G: Λίστα που περιέχει τους μη-κατευθυνόμενους γράφους για κάθε μια χρονική περίοδο.
- V*: Λίστα που περιέχει την τομή των συνόλων των κορυφών 2 διαδοχικών (χρονικά) γράφων.

- E*: Δισδιάστατη λίστα που περιέχει για κάθε διαδοχικό ζεύγος γράφων, τις ακμές που υπάρχουν μεταξύ δύο κόμβων, οι οποίοι ανήκουν σε δυο διαδοχικούς γράφους, αλλά δημιουργούνται – οι ακμές – μόνο στον πρώτο (πρώτη στήλη της λίστας). Αντίστοιχα, η δεύτερη στήλη της λίστας, περιέχει αυτές που δημιουργούνται στον δεύτερο γράφο.
- G*: Λίστα που περιέχει τους μη-κατευθυνόμενους γράφους των οποίων οι κόμβοι ανήκουν σε δυο διαδοχικούς κόμβους και ακμές που δημιουργήθηκαν τη προηγούμενη χρονική περίοδο. $G_j^* = (V_j^*, E_{j0}^*)$

Υλοποίηση και εκτέλεση της εργασίας

Η εργασία υλοποιήθηκε σε Python Notebook, χρησιμοποιώντας Python 3.7.7. Εκτός από το .ipynb αρχείο παραδίδεται και αντίστοιχο .py αρχείο το οποίο όπως παρουσιάζει σφάλματα κατά την εκκίνηση λόγω ενός dll. Προτείνεται λοιπόν η χρήση του Notebook.

Για την εκτέλεση του .py, γράψτε την εντολή:

```
python SNAProject.py <N> <Pgd> <Pcn> <Pjc> <Paa> <Ppa> [<Data Percentage>]
```

όπου N: αριθμός διαστημάτων,
Pgd: ποσοστό κορυφαίων τιμών ομοιότητας για το Sgd
Pcn: ποσοστό κορυφαίων τιμών ομοιότητας για το Scn
Pjc : ποσοστό κορυφαίων τιμών ομοιότητας για το Sjc
Paa: ποσοστό κορυφαίων τιμών ομοιότητας για το Saa
Ppa: ποσοστό κορυφαίων τιμών ομοιότητας για το Spa
Data Percentage (optional): ποσοστό των νεότερων δεδομένων που θα χρησιμοποιηθούν. (default 0.005%)

Για την εκτέλεση του .ipynb, ανοίξτε το αρχείο από Visual Code ή Jupyter Notebook. Στο τρίτο κελί εισάγεται τις επιθυμητές τιμές στις παραμέτρους.

```
N = 200
Pgd = 0.6
Pcn = 0.6
Pjc = 0.6
Paa = 0.6
Ppa = 0.6
PERCENTAGE = 0.005
```

Εκτελέστε κάθε κελί με τη σειρά που βρίσκονται.

Διαδικασίες

0. Φορτώνουμε τα επιθυμητά δεδομένα σε ένα DataFrame.

```
ALL_RECORDS = 63497049
# Keep only 0.5% of samples.

skip = ceil(ALL_RECORDS*(1-PERCENTAGE))
print("Loading data...")
data = pd.read_csv('sx-stackoverflow.txt', sep=" ", header=0, names=['source', 'target', 'timestamp'],
dtype={'source': np.int32, 'target': np.int32, 'timestamp': np.int32}, skiprows=skip)
print("Done!")
```

```
Loading data...
Done!
```

1. Εξάγουμε τις τιμές t_{min}, t_{max}

```
# 1 Calculating t_min t_max
t_min = data.iloc[0,2]
t_max = data.iloc[-1,2]
print(f"t_min: {t_min}, t_max: {t_max}")
```

```
t_min: 1456431641, t_max: 1457273428
```

2. Υπολογίζουμε τη διαμέριση του διαστήματος $T = [t_{min}, t_{max}]$ στα υποδιαστήματα $\{T_1, T_2, \dots, T_j, \dots, T_N\}$ και τις αντίστοιχες χρονικές στιγμές $\{t_0, t_1, \dots, t_{j-1}, t_j \dots t_{N-1}, t_N\}$.

```
# 2 Calculating time intervals

Dt = t_max - t_min
dt = ceil(Dt/N)

t = [t_min + j * dt for j in range(N + 1)]
#print(t)
T = [[t[i], t[i + 1] - 1] for i in range(N)]
T[-1][1] = t_max # eliminate remainders
print(f"N: {N}, Dt: {Dt}, dt: {dt}")
for i in range(len(T)):
    pass
    print(f"T[{i}, {i+1}]: [{T[i][0]}, {T[i][1]}]")
```

```
N: 200, Dt: 841787, dt: 4209
T[0, 1]: [1456431641, 1456435849]
T[1, 2]: [1456435850, 1456440058]
T[2, 3]: [1456440059, 1456444267]
T[3, 4]: [1456444268, 1456448476]
T[4, 5]: [1456448477, 1456452685]
T[5, 6]: [1456452686, 1456456894]
T[6, 7]: [1456456895, 1456461103]
T[7, 8]: [1456461104, 1456465312]
T[8, 9]: [1456465313, 1456469521]
T[9, 10]: [1456469522, 1456473730]
T[10, 11]: [1456473731, 1456477939]
T[11, 12]: [1456477940, 1456482148]
T[12, 13]: [1456482149, 1456486357]
T[13, 14]: [1456486358, 1456490566]
```

3. Δημιουργούμε μια λίστα από τα υποδίκτυα κάθε χρονικής περιόδου.

```
# 3 Calculating undirected Graphs
print(f"Calculating undirected graphs for {N} time intervals.")

G = []
ti = 0
g = nx.Graph()
for i, r in data.iterrows():
    if r['timestamp'] < T[ti][1]:
        g.add_edge(r['source'], r['target'])
    else:
        G.append(g)
        g = nx.Graph()
        g.add_edge(r['source'], r['target'])
        ti += 1
print("Done!")
```

```
Calculating undirected graphs for 200 time intervals.
Done!
```

4. Για κάθε υποδίκτυο, υπολογίζουμε ορισμένες κεντρικότητες. Η κεντρικότητα αποτελεί ένα δείκτη σημαντικότητας ενός κόμβου. Κάθε δείκτης κεντρικότητας ορίζει διαφορετικά την σημαντικότητα που υπολογίζει. Να σημειωθεί πως στην συγκεκριμένη υλοποίηση οι δείκτες κεντρικότητας είναι κανονικοποίημενοι ως προς την μέγιστη τιμή κεντρικότητας κάθε υποδικτύου. Οι δείκτες κεντρικότητας που υπολογίζονται είναι:
   - Degree Centrality: Ορίζεται ως ο βαθμός του κόμβου.
   - In-Degree Centrality: Ορίζεται ως ο αριθμός των εισερχόμενων κόμβων (σε κατευθυνόμενο γράφο).
   - Out- Degree Centrality: Ορίζεται ως ο αριθμός των εξερχόμενων κόμβων (σε κατευθυνόμενο γράφο).
   - Closeness Centrality: Ορίζεται ως το μέσο μήκος των συντομότερων μονοπατιών προς όλους τους κόμβους (σε μη-κατευθυνόμενο γράφο).
   - Betweenness Centrality: Ορίζεται ως το άθροισμα του κλάσματος των συντομότερων μονοπατιών όλων των ζευγών που διέρχονται από τον κόμβο.
   - Eigenvector Centrality: Υπολογίζει την σημαντικότητα ενός κόμβου βάσει την σημαντικότητα των γειτόνων του. Χρησιμοποιείται η εξίσωση του ιδιοδιανύσματος για τον υπολογισμού του, εξού και το όνομα του δείκτη, $Ax = \lambda x$ όπου A η μήτρα γειτνίασης, λ η ιδιοτιμή και x το διάνυσμα των δεικτών κεντρικότητας των κόμβων.
   - Katz Centrality: Αποτελεί μια γενίκευση του Eigenvector Centrality. Υπολογίζει την σημαντικότητα ενός κόμβου βάσει την σημαντικότητα των γειτόνων του.

```python
def get_centralities(func, G):

    if func == nx.in_degree_centrality or func == nx.out_degree_centrality:
        G = nx.DiGraph(G)

    if func == nx.eigenvector_centrality or func == nx.katz_centrality:
        centr = func(G, max_iter=10000)
    else:
        centr = func(G)

    return centr
```

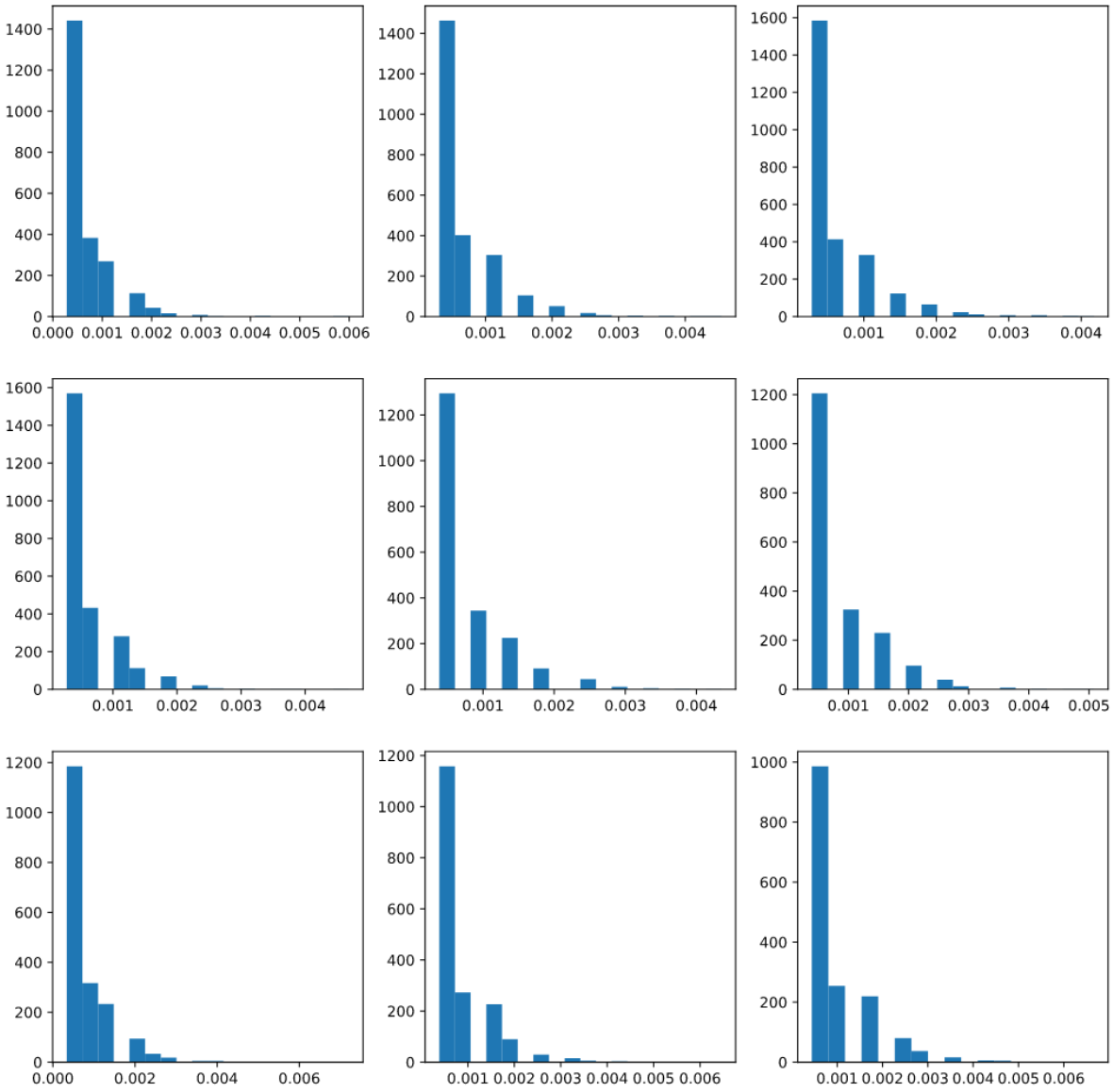
```
# 4 Calculating centrality metrics.
degree_centralities = []
in_degree_centralities = []
out_degree_centralities = []
closeness_centralities = []
betweenness_centralities = []
eigenvector_centralities = []
katz_centralities = []

start = time.time()
print("Calculating degree centralities for each graph.")
for i in range(len(G)):
    degree_centralities.append(get_centralities(nx.degree_centrality, G[i]))
print("Done! Execution time:", time.time() - start, '\n')

start = time.time()
print("Calculating in degree centralities for each graph. (Converted into directed graphs)")
for i in range(len(G)):
    in_degree_centralities.append(get_centralities(nx.in_degree_centrality, G[i]))
print("Done! Execution time:", time.time() - start, '\n')

start = time.time()
print("Calculating out degree centralities for each graph. (Converted into directed graphs)")
for i in range(len(G)):
    out_degree_centralities.append(get_centralities(nx.out_degree_centrality, G[i]))
print("Done! Execution time:", time.time() - start, '\n')

start = time.time()
print("Calculating closeness centralities for each graph. (Estimated execution time: 14'')")
for i in range(len(G)):
    closeness_centralities.append(get_centralities(nx.closeness_centrality, G[i]))
print("Done! Execution time:", time.time() - start, '\n')

start = time.time()
print("Calculating betweenness centralities for each graph. (Estimated execution time: 250'')")
for i in range(len(G)):
    betweenness_centralities.append(get_centralities(nx.betweenness_centrality, G[i]))
print("Done! Execution time:", time.time() - start, '\n')

start = time.time()
print("Calculating eigenvector centralities for each graph. (Estimated execution time: 84'')")
for i in range(len(G)):
    eigenvector_centralities.append(get_centralities(nx.eigenvector_centrality, G[i]))
print("Done! Execution time:", time.time() - start, '\n')

start = time.time()
print("Calculating katz centralities for each graph. (Estimated execution time: 12'')")
for i in range(len(G)):
    katz_centralities.append(get_centralities(nx.katz_centrality, G[i]))
print("Done! Execution time:", time.time() - start, '\n')
```

```
Calculating degree centralities for each graph.
Done! Execution time: 0.13660478591918945

Calculating in degree centralities for each graph. (Converted into directed graphs)
Done! Execution time: 1.5468928813934326

Calculating out degree centralities for each graph. (Converted into directed graphs)
Done! Execution time: 1.609691858291626

Calculating closeness centralities for each graph. (Estimated execution time: 14'')
Done! Execution time: 13.502883911132812

Calculating betweenness centralities for each graph. (Estimated execution time: 250'')
Done! Execution time: 222.1573040485382

Calculating eigenvector centralities for each graph. (Estimated execution time: 84'')
Done! Execution time: 85.26569414138794

Calculating katz centralities for each graph. (Estimated execution time: 12'')
Done! Execution time: 12.35606050491333
```

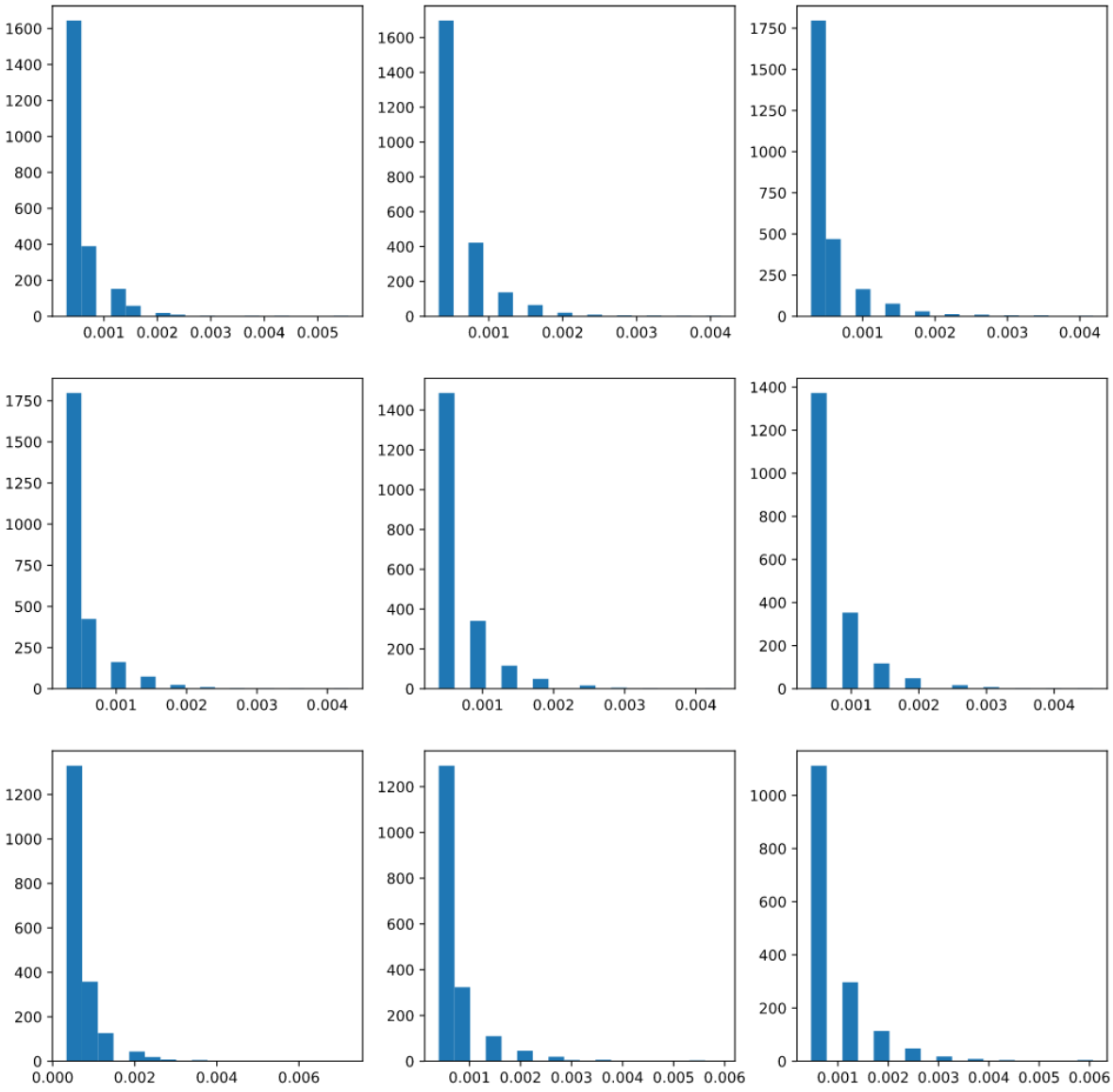
Σε περίπτωση *PowerIterationFailedConvergence* αυξήστε το max_iter στη συνάρτηση *get_centralities.*

Στη συνέχεια παρουσιάζονται γραφικά οι κατανομές κάθε δείκτη των 9 τελευταίων υποδικτύων.

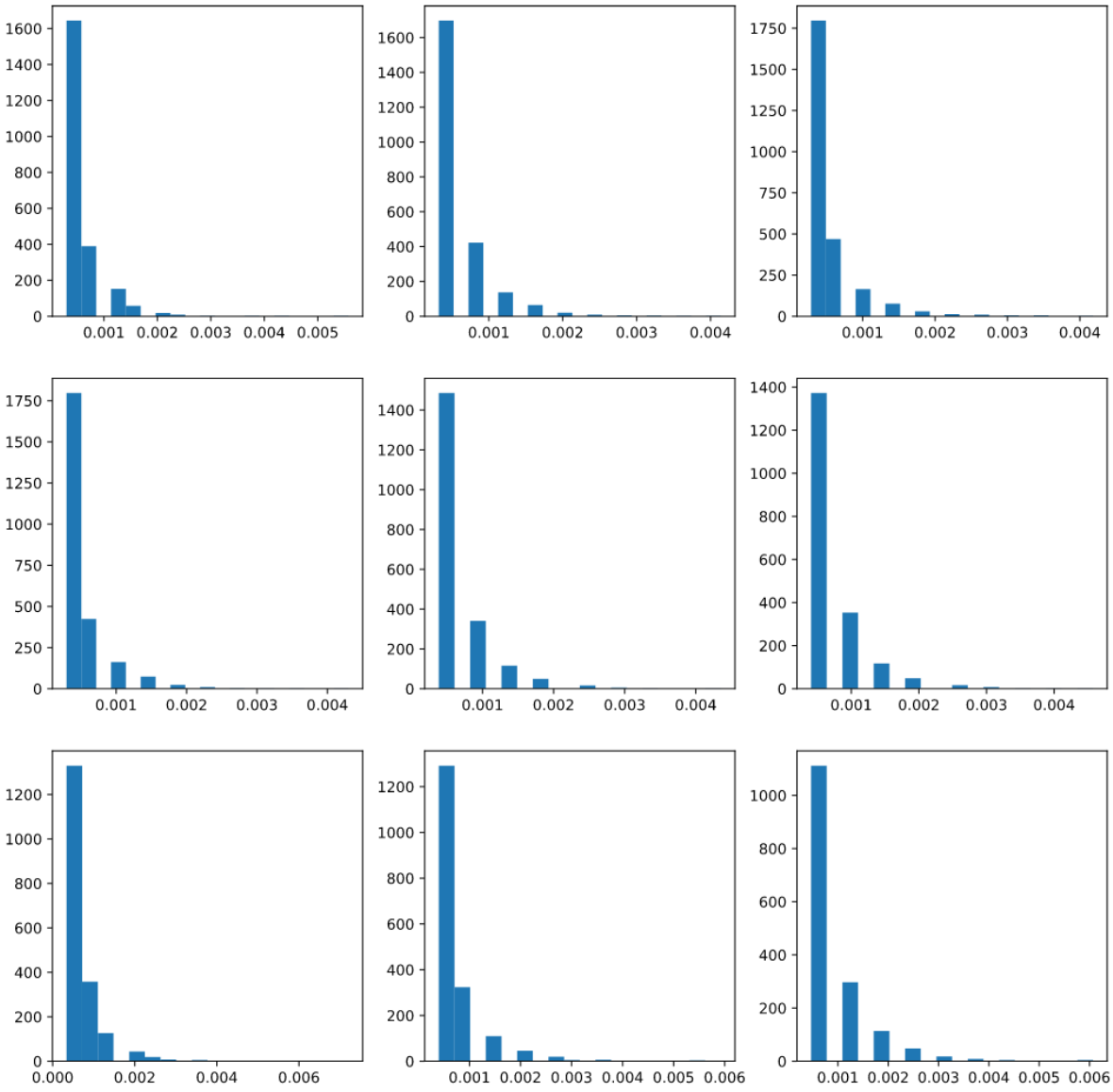
- Degree Centrality



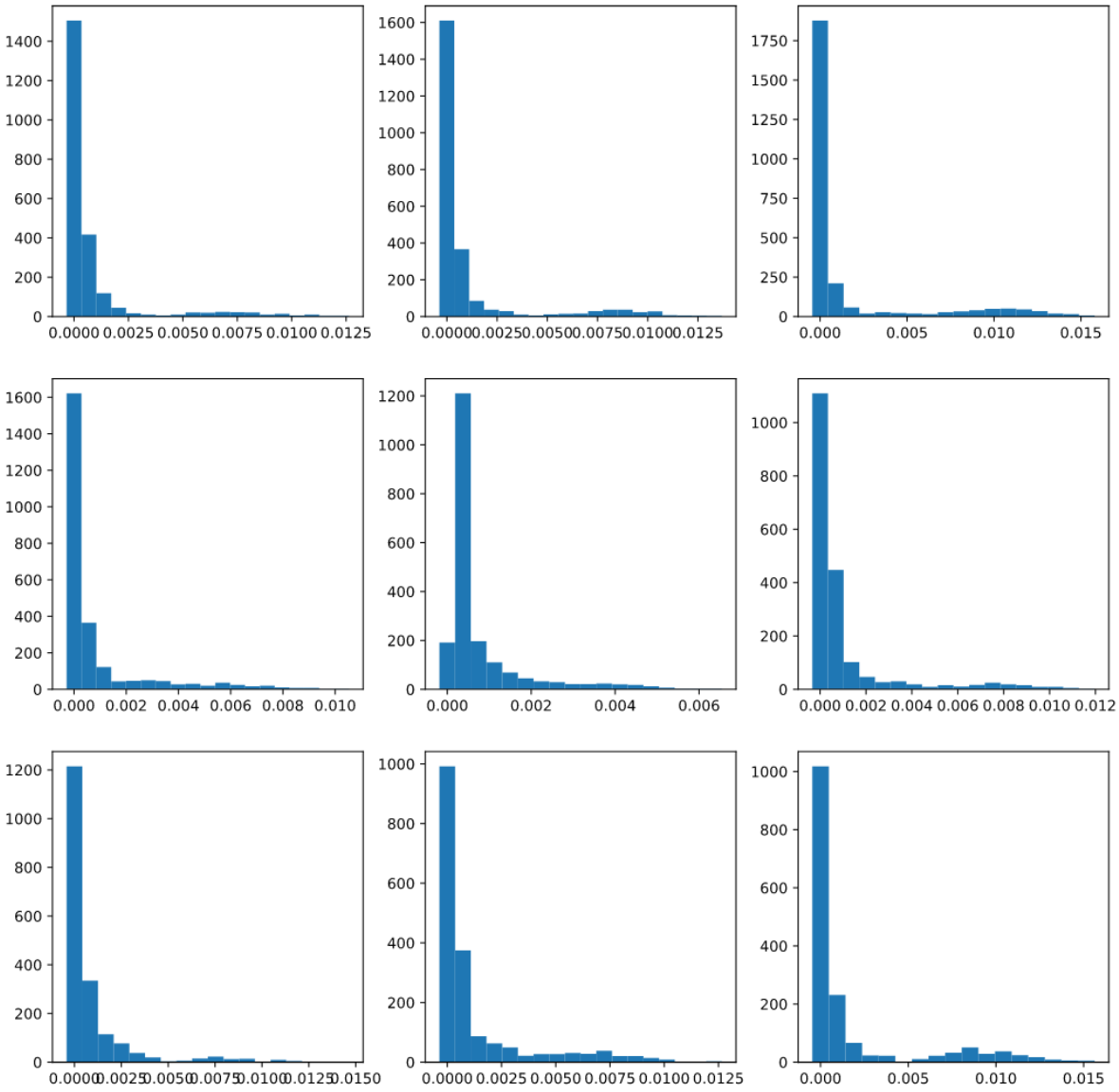
- In-Degree Centrality



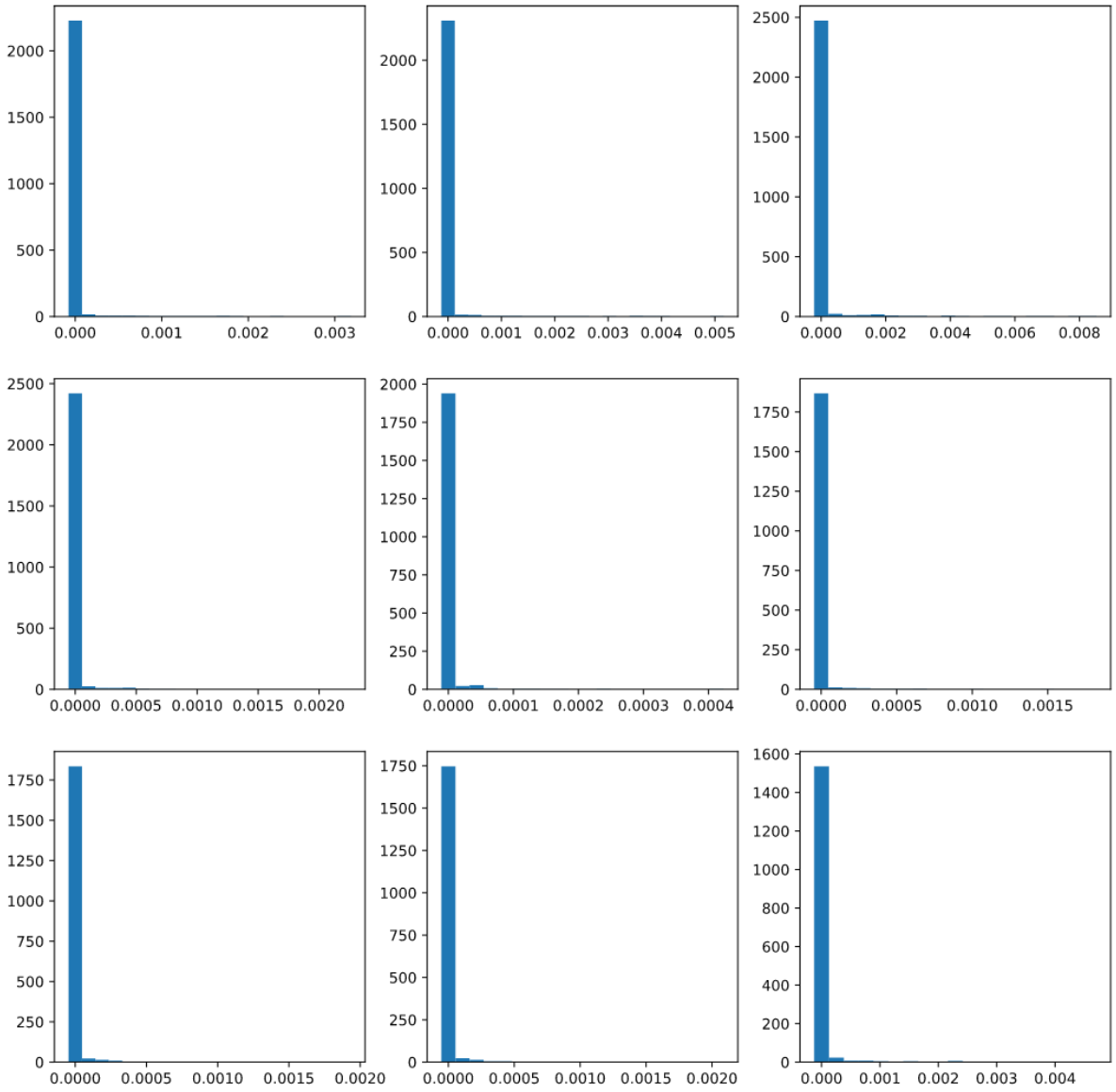
- Out-Degree Centrality



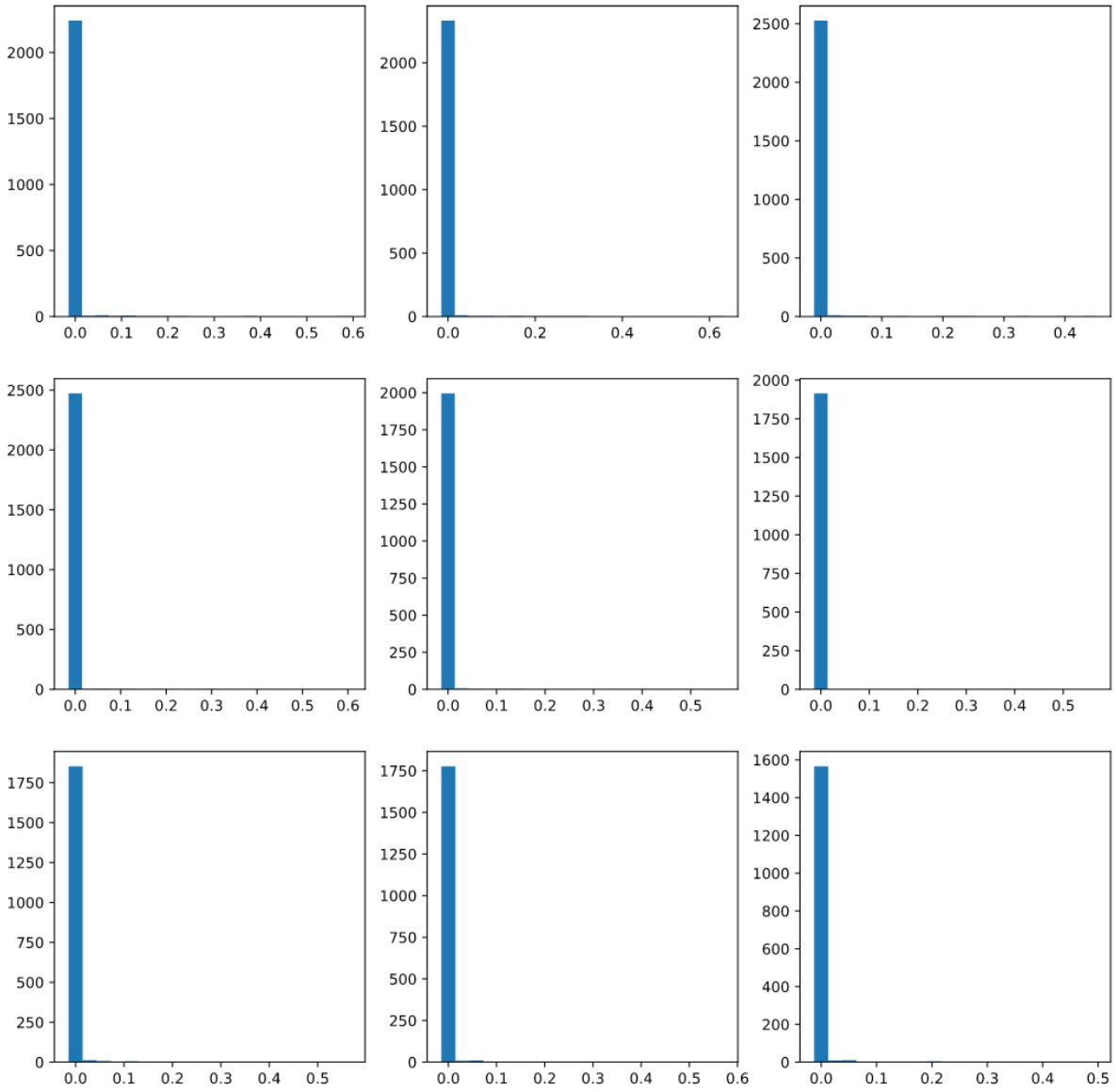
- Closeness Centrality



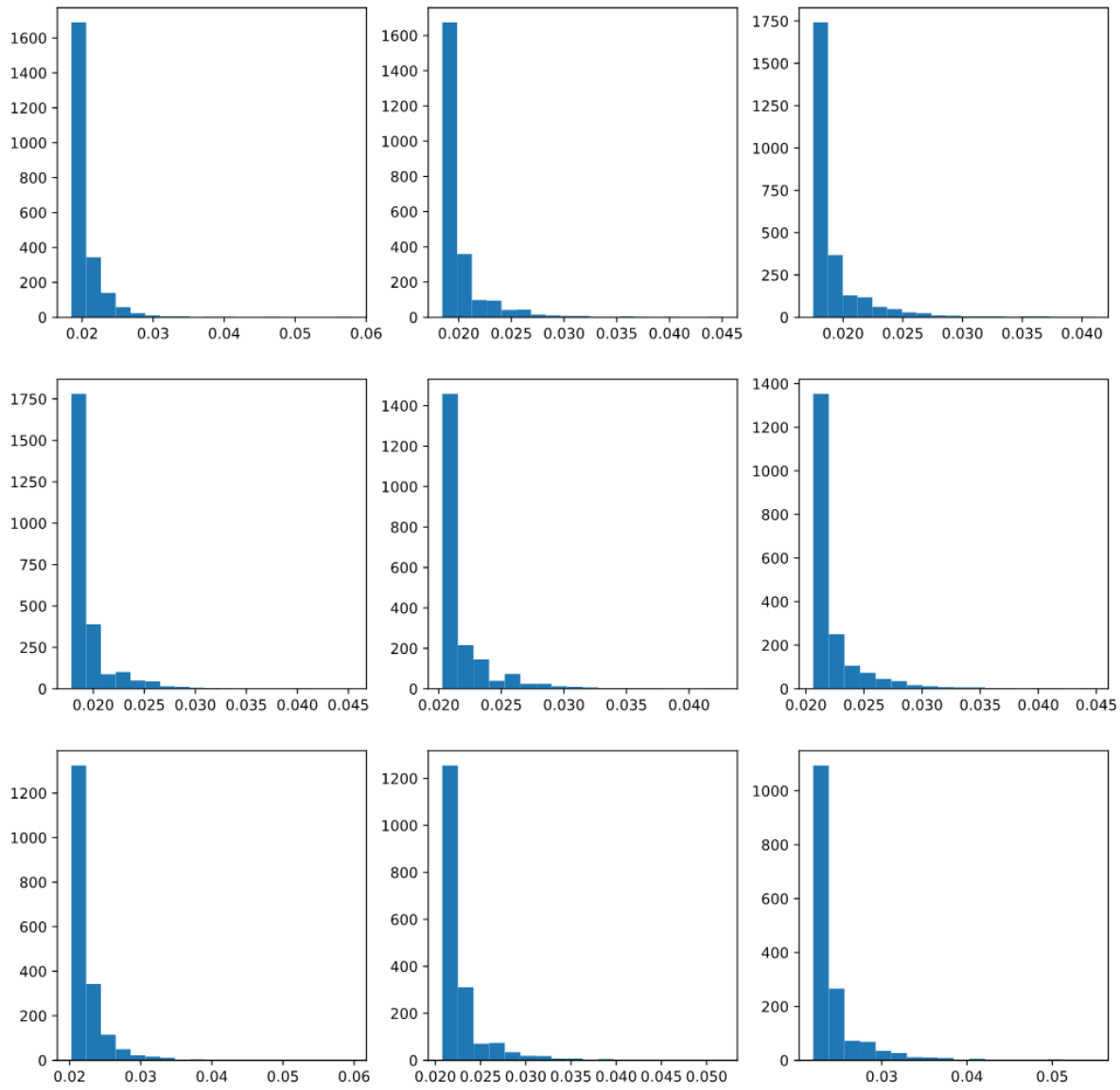
- Betweenness Centrality



- Eigenvector Centrality



- Katz Centrality



Παρατηρούμε πως όλες για όλους του δείκτες οι κατανομές είναι όμοιες, με τις μικρές τιμές να υπερτερούν αυτών των μεγαλύτερων όπως ήταν αναμενόμενο. Με άλλα λόγια, οι πολύ σημαντικοί κόμβοι σε ένα δίκτυο είναι κατά πολύ λιγότεροι από τους «ασήμαντους».

5. Υπολογίζουμε τα σύνολα $V^*[t_{j-1}, t_{j+1}], E^*[t_{j-1}, t_j], E^*[t_j, t_{j+1}]$ για κάθε ζεύγος διαδοχικών υποδικτύων.

```
# 5 Calculating V*
print("Calculating V* for every two consecutive intervals.")
V_star = []
for i in range(1, N - 1):
    v_0 = set(G[i-1].nodes)
    v_1 = set(G[i].nodes)
    intersection = list(v_0 & v_1)
    V_star.append(intersection)
print("Done!\n")
```

```
Calculating V* for every two consecutive intervals.
Done!
```

```
#  Calculating E*
# E[j][0] the previous time interval (set 7 in SocialNetworksAnalysisAssignment.pdf),
# E[j][1] the next time interval (set 8 in SocialNetworksAnalysisAssignment.pdf)

print("Calculating E* for each time interval.")
E_star = []
j = 0
for i in range(1, N-1):
    e_0 = []
    for e in list(G[i-1].edges):
        u,v = e
        if u in V_star[j] and v in V_star[j]:
            e_0.append((u,v))
    e_1 = []
    for e in list(G[i].edges):
        u,v = e
        if u in V_star[j] and v in V_star[j]:
            e_1.append((u,v))
    E_star.append([e_0, e_1])
    j += 1
print("Done!\n")
```

```
Calculating E* for each time interval.
Done!
```

6. Τα μέτρα ομοιότητας μας βοηθούν στη πρόβλεψη δημιουργίας νέας ακμής. Ορίζονται διάφοροι δείκτες ομοιότητας, οι οποίοι ορίζουν τον δικό τους τρόπου υπολογισμού της ομοιότητας. Τα μέτρα υπολογίζονται στα υποδίκτυα με κόμβους το σύνολο $V^*[t_{j-1}, t_{j+1}]$ άλλα πάνω στη βάση του συνόλου των προηγούμενων ακμών $E^*[t_{j-1}, t_j]$. Έτσι, ορίζουμε τα αντίστοιχα υποδίκτυα $G^*[t_{j-1}, t_{j+1}] = (V^*[t_{j-1}, t_{j+1}], E^*[t_{j-1}, t_j])$.

```
# 6 Creating G*s, where G*_j = (V*_j, E*_j[0])
print("Calclating G* for each V*. [G*_j = (V*_j, E*_j[0])]")
G_star = []
for i in range(len(V_star)):
    g = nx.Graph()
    for node in V_star[i]:
        g.add_node(node)
    for source, target in E_star[i][0]:
        g.add_edge(source, target)
    G_star.append(g)
print("Done!")
```

```
Calclating G* for each V*. [G*_j = (V*_j, E*_j[0])]
Done!
```

```python
def get_similarity_dict(func, G):
    if func == nx.all_pairs_shortest_path_length:
        sim_dict = dict(func(G))

    elif func == nx.common_neighbors:
        sim_dict = {}
        for u in G.nodes:
            dict_u = {}
            for v in G.nodes:
                cnbors  = func(G, u, v)
                n_cnbors = sum(1 for _ in cnbors)
                dict_u[v] = n_cnbors
            sim_dict[u] = dict_u

    elif func == nx.jaccard_coefficient or func == nx.preferential_attachment:
        results = func(G, ebunch=G.edges)
        sim_dict = {k: {v: s} for k,v,s in results}

    elif func == nx.adamic_adar_index:
        G = G.copy()
        G.remove_edges_from(nx.selfloop_edges(G))
        results = func(G, ebunch=G.edges)
        sim_dict = {k: {v: s} for k,v,s in results}

    else:
        raise Exception("Not implemented for this function")
    return sim_dict
```

Υπολογίζονται οι παρακάτω μήτρες ομοιότητας:

- Graph Distance (Sgd): Ορίζεται ως το μήκος του συντομότερου μονοπατιού μεταξύ δύο κόμβων.
- Common Neighbors (Scn): Ορίζεται ως το πλήθος των κοινών γειτόνων δύο κόμβων.
- Jaccard's Coefficient (Sjc): Ορίζεται ως ο λόγος του πλήθους των κοινών γειτόνων δύο κόμβων προς το συνολικό πλήθος των γειτόνων τους.
- Adamic / Adar (Saa): Υπολογίζεται βάσει του πλήθους των γειτόνων των από κοινού γειτονικών κόμβων μεταξύ δύο κόμβων. Δίνει έμφαση στις ακμές των οποίων οι κόμβοι έχουν από κοινού «αδύναμους» γείτονες.
- Preferential Attachment (Spa): Ορίζεται ως το γινόμενο του πλήθους των γειτόνων των δύο κόμβων. Ακολουθεί την λογική "the rich are getting richer".

```
M↓
start = time.time()
print("Calculating Sdg for each G*.")
Sgd = [get_similarity_dict(nx.all_pairs_shortest_path_length, G_star[i]) for i in range(len(G_star))]
print("Done! Execution time: ", time.time()-start, "\n")

start = time.time()
print("Calculating Scn for each G*. (Estimated execution time: 120'')")
Scn = [get_similarity_dict(nx.common_neighbors, G_star[i]) for i in range(len(G_star))]
print("Done! Execution time: ", time.time()-start, "\n")

start = time.time()
print("Calculating Sjc for each G*.")
Sjc = [get_similarity_dict(nx.jaccard_coefficient, G_star[i]) for i in range(len(G_star))]
print("Done! Execution time: ", time.time()-start, "\n")

start = time.time()
print("Calculating Saa for each G*.")
Saa = [get_similarity_dict(nx.adamic_adar_index, G_star[i]) for i in range(len(G_star))]
print("Done! Execution time: ", time.time()-start, "\n")

start = time.time()
print("Calculating Spa for each G*.")
Spa = [get_similarity_dict(nx.preferential_attachment, G_star[i]) for i in range(len(G_star))]
print("Done! Execution time: ", time.time()-start, "\n")
```

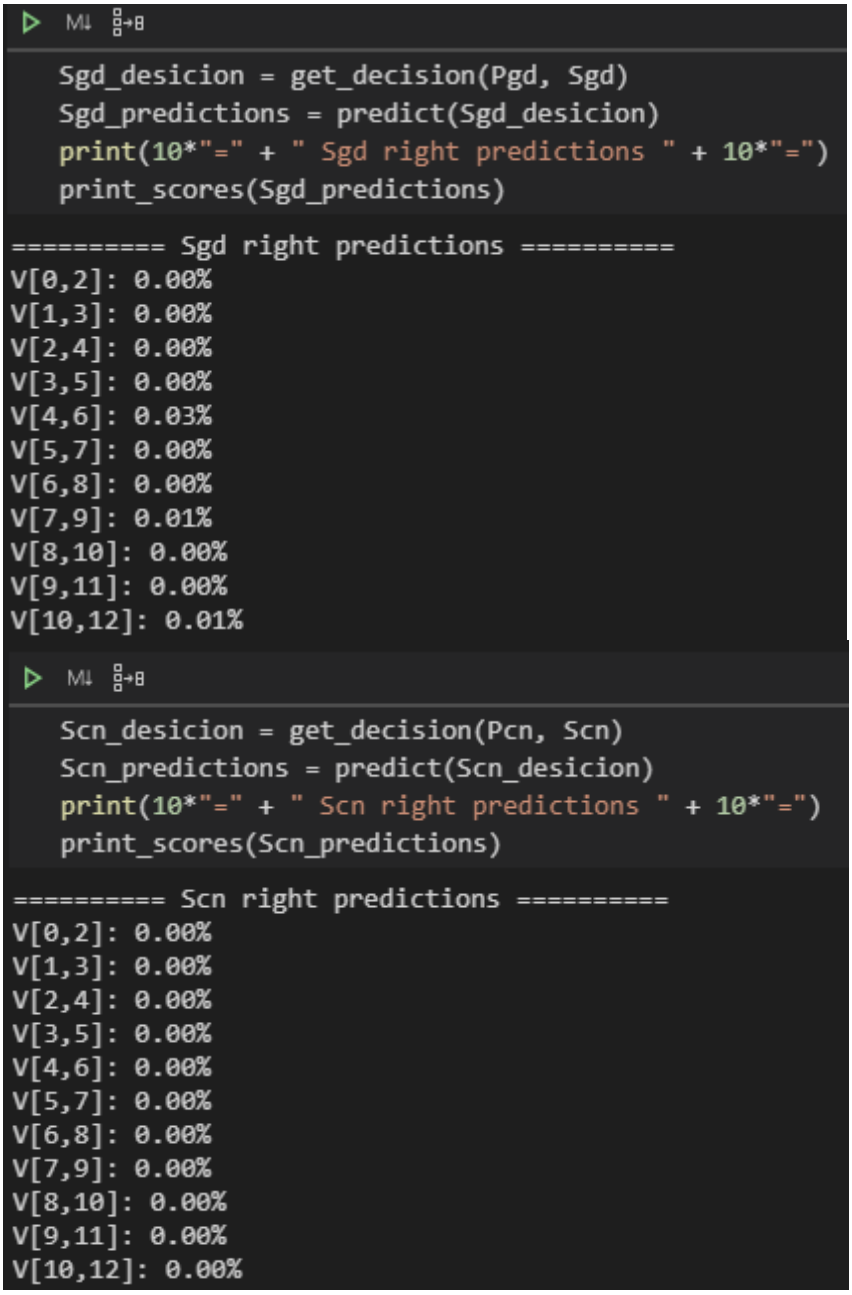
```
Calculating Sdg for each G*.
Done! Execution time:  0.5426616668701172

Calculating Scn for each G*. (Estimated execution time: 120'')
Done! Execution time:  128.16038060188293

Calculating Sjc for each G*.
Done! Execution time:  0.48148393630981445

Calculating Saa for each G*.
Done! Execution time:  0.44892334938049316

Calculating Spa for each G*.
Done! Execution time:  0.17852330207824707
```

Κάθε λίστα S περιέχει Dictionary of Dictionaries, αντί δισδιάστατη μήτρα, με την τιμή ομοιότητας μεταξύ δύο κόμβων. Είναι δηλαδή της μορφής [{source : {target : value}}]. Πλέον, όταν λέμε μήτρα ομοιότητας, θα αναφερόμαστε σε ένα τέτοιο dictionary.

7. Τέλος, ορίζονται συναρτήσεις απαραίτητες για την τελική πρόβλεψη.

```
M↓
def get_min_max_value_of_similarity(similarity):
    val = []
    for u in similarity:
        for v in similarity[u]:
            val.append(similarity[u][v])
    return (min(val), max(val))


def get_decision(p, s):
    s_decision = []
    for i in range(len(s)):
        min_val, max_val = get_min_max_value_of_similarity(s[i])
        decision = []
        for u in s[i].keys():
            for v in s[i][u].keys():
                if u == v:
                    continue
                value = s[i][u][v]
                if value >= p * max_val:
                    decision.append((u,v))
        s_decision.append(decision)
    return s_decision


def predict(decision):
    right_prediction = []
    for i in range(len(decision)):
        edges_exist = 0
        for u,v in decision[i]:
            if (u,v) in E_star[i][1]:
                edges_exist +=1

        if edges_exist == 0 and len(decision[i]) == 0 :
            right_prediction.append(0.0)
        else:
            right_prediction.append(edges_exist/len(decision[i]))
    return right_prediction


def print_scores(prediction):
    for i in range(len(prediction)):
        print('V[%d,%d]: %.2f%%' % (i, i+2,prediction[i]))
```

- get_min_max_value_of_similarity: Επιστρέφει τη μέγιστη και την ελάχιστη τιμή ομοιότητας μιας μήτρας ομοιότητας.
- get_decision: Δέχεται ως ορίσματα το ποσοστό των κορυφαίων τιμών ομοιότητας και τη μήτρα ομοιότητας, p και s αντίστοιχα. Επιστρέφει μια λίστα απόφασης με τις ακμές της προηγούμενης χρονικής περιόδου, που έχουν τιμή ομοιότητας μεγαλύτερη ή ίση από την επιθυμητή p * max_val όπου max_val η μέγιστη τιμή ομοιότητας της μήτρας αυτής.
- predict: Δέχεται ως όρισμα μια λίστα απόφασης (όπως περιγράφηκε παραπάνω) και επιστρέφει μια λίστα με τα ποσοστό των ακμών που υπάρχουν στην επόμενη χρονική περίοδο μέσα στη λίστα απόφασης, για κάθε υποδίκτυο.
- print_scores: Δέχεται ως όρισμα μια λίστα ποσοστών ορθών προβλέψεων και την εμφανίζει.

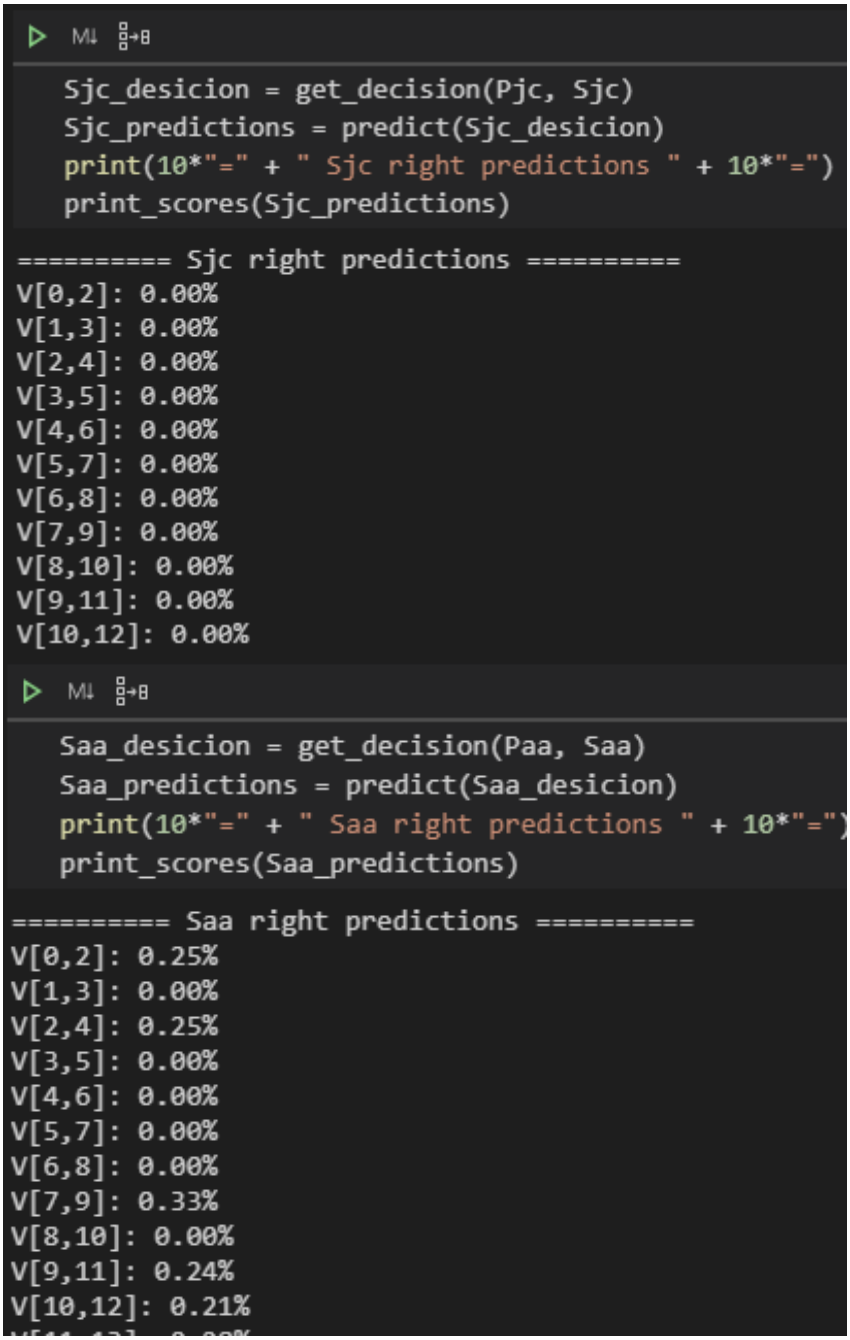


```
Sgd_desicion = get_decision(Pgd, Sgd)
Sgd_predictions = predict(Sgd_desicion)
print(10*"=" + " Sgd right predictions " + 10*"=")
print_scores(Sgd_predictions)
```

```
========== Sgd right predictions ==========
V[0,2]: 0.00%
V[1,3]: 0.00%
V[2,4]: 0.00%
V[3,5]: 0.00%
V[4,6]: 0.03%
V[5,7]: 0.00%
V[6,8]: 0.00%
V[7,9]: 0.01%
V[8,10]: 0.00%
V[9,11]: 0.00%
V[10,12]: 0.01%
```

```
Scn_desicion = get_decision(Pcn, Scn)
Scn_predictions = predict(Scn_desicion)
print(10*"=" + " Scn right predictions " + 10*"=")
print_scores(Scn_predictions)
```

```
========== Scn right predictions ==========
V[0,2]: 0.00%
V[1,3]: 0.00%
V[2,4]: 0.00%
V[3,5]: 0.00%
V[4,6]: 0.00%
V[5,7]: 0.00%
V[6,8]: 0.00%
V[7,9]: 0.00%
V[8,10]: 0.00%
V[9,11]: 0.00%
V[10,12]: 0.00%
```

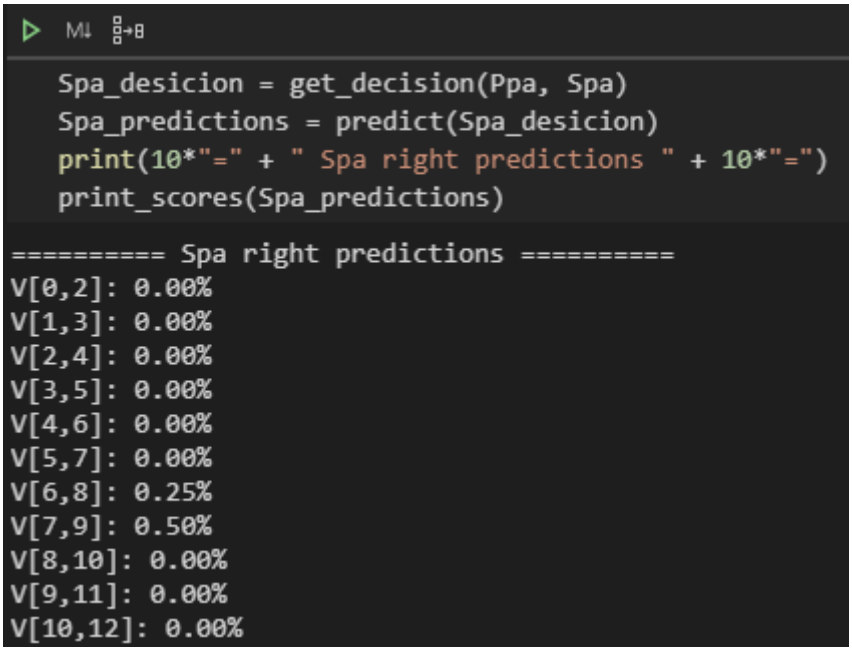


```
Sjc_desicion = get_decision(Pjc, Sjc)
Sjc_predictions = predict(Sjc_desicion)
print(10*"=" + " Sjc right predictions " + 10*"=")
print_scores(Sjc_predictions)
```

```
========== Sjc right predictions ==========
V[0,2]: 0.00%
V[1,3]: 0.00%
V[2,4]: 0.00%
V[3,5]: 0.00%
V[4,6]: 0.00%
V[5,7]: 0.00%
V[6,8]: 0.00%
V[7,9]: 0.00%
V[8,10]: 0.00%
V[9,11]: 0.00%
V[10,12]: 0.00%
```

```
Saa_desicion = get_decision(Paa, Saa)
Saa_predictions = predict(Saa_desicion)
print(10*"=" + " Saa right predictions " + 10*"=")
print_scores(Saa_predictions)
```

```
========== Saa right predictions ==========
V[0,2]: 0.25%
V[1,3]: 0.00%
V[2,4]: 0.25%
V[3,5]: 0.00%
V[4,6]: 0.00%
V[5,7]: 0.00%
V[6,8]: 0.00%
V[7,9]: 0.33%
V[8,10]: 0.00%
V[9,11]: 0.24%
V[10,12]: 0.21%
```



```
    Spa_desicion = get_decision(Ppa, Spa)
    Spa_predictions = predict(Spa_desicion)
    print(10*"=" + " Spa right predictions " + 10*"=")
    print_scores(Spa_predictions)
```

```
========== Spa right predictions ==========
V[0,2]: 0.00%
V[1,3]: 0.00%
V[2,4]: 0.00%
V[3,5]: 0.00%
V[4,6]: 0.00%
V[5,7]: 0.00%
V[6,8]: 0.25%
V[7,9]: 0.50%
V[8,10]: 0.00%
V[9,11]: 0.00%
V[10,12]: 0.00%
```

Τα ποσοστά δεν είναι ενθαρρυντικά. Ίσως ευθύνεται ο διαχωρισμός των χρονικών περιόδων και η επιλογή του τελευταίου 0.005% του συνόλου. Δυστυχώς, το υλικό δεν βοήθησε στο να γίνουν αρκετές δοκιμές με μεγαλύτερο μέγεθος συνόλου δεδομένων. Αυτό γιατί απαιτείται αρκετή μνήμη για τις δομές, καθώς και υπολογιστική ισχύς για τους αλγορίθμους, για παράδειγμα Closeness Centrality αφού υπολογίζει όλα τα συντομότερα μονοπάτια για κάθε κόμβο.

Ευχαριστώ πολύ!