

INA - homework 3

Matic Isovski, 63180442

April 2022

Homework #3

This homework is complete and will not be changed. The homework does not require a lot of writing, but may require some thinking. It does not require a lot of processing power, but may require efficient programming. It accounts for 10% of the course grade. Any questions and comments regarding the homework should be directed to [Piazza](#).

Submission details

This homework is due on **April 22nd** at 11:59pm, while late days expire on **April 25th** at 11:00am. The homework must be submitted through (1) [Gradescope](#) (entry code **6PDZD3**) and (2) [eUcilnica](#). (1) Submission to [Gradescope](#) should include answers to all questions, each on a separate page, which may also demand pseudocode, proofs, tables, plots, diagrams and other. (2) Submission to [eUcilnica](#) should include at least this cover sheet with signed honor code and all the programming code used to complete the exercises. The homework is considered submitted only when *both* parts have been submitted. Failing to include this honor code in the submission will result in **10% deduction**. Failing to submit all the developed code will result in **50% deduction**.

Honor code

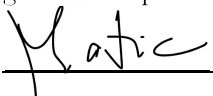
Students are strongly encouraged to discuss the homework with other classmates and form study groups. Yet, each student must then solve the homework by itself without the help of others and should be able to redo the homework at a later time. In other words, students are encouraged to collaborate, but should not copy from one another. Referring to any solutions obtained from classmates, course books, previous years, found online or other material, is considered an honor code violation. Also, stating any part of the solutions in class or on [Piazza](#) is considered an honor code violation. Finally, failing to name the correct study group members, or filling out the wrong date or time of the submission, is also considered an honor code violation. Honor code violation will not be tolerated! Any student violating the honor code will be reported to **faculty disciplinary committee** and vice dean for education.

Name & SID: Matic Isovski tzXs80Wu2yUQE9mgRVJojB7O6GlnZh5edHMTDaxL

Study group: _____

Date & time: 25.04.2022 10:00

I acknowledge and accept the honor code.

Signature: 

1 Ring graph modularity

So we get $Q = 1 - \frac{1}{n_c} - \frac{n_c}{n}$ and optimal size of cluster $n_c = \sqrt{n}$.

I. $Q = \sum_c \frac{m_c}{m} - \left(\frac{k_c}{2m} \right)^2$

$m = n$
 $m_c = n_c - 1$
 $k_c = 2n_c$
 $C = \frac{n}{n_c}$

$$\Rightarrow Q = \frac{n}{n_c} \left(\frac{n_c - 1}{n} - \frac{n_c^2}{n^2} \right)$$

$$= \frac{n_c - 1}{n_c} - \frac{n_c}{n}$$

$n_c = 4$:

$$Q = 1 - \frac{1}{n_c} - \frac{n_c}{n} \quad ; \quad Q = 1 - \frac{1}{4} - \frac{4}{36} = 0,64$$

II. (Find maximum):

$$f(x) = 1 - \frac{1}{n_c} - \frac{n_c}{n} = 1 - \frac{1}{n_c} - \frac{n_c}{36}$$

$$f'(x) = n_c^{-2} - \frac{1}{36}$$

$$n_c^{-2} - \frac{1}{36} = 0$$

$$n_c^2 = 36$$

$$n_c = \pm 6$$

$$\Rightarrow n_c = 6 : Q = 1 - \frac{1}{6} - \frac{6}{36} = 0,66$$

$$\boxed{n_{c_{opt}} = \sqrt{n}}$$

Figure 1: Derived expression and optimal size of clusters n_c .

2 Who's the winner?

2.1 i

In Figure 2, we can clearly see, that the most robust the Walktrap. Robustness of Louvain is very close to Walktrap's, but slightly drops after $\mu > 0.6$. Infomap is the worst, falling already when $\mu > 0.2$.

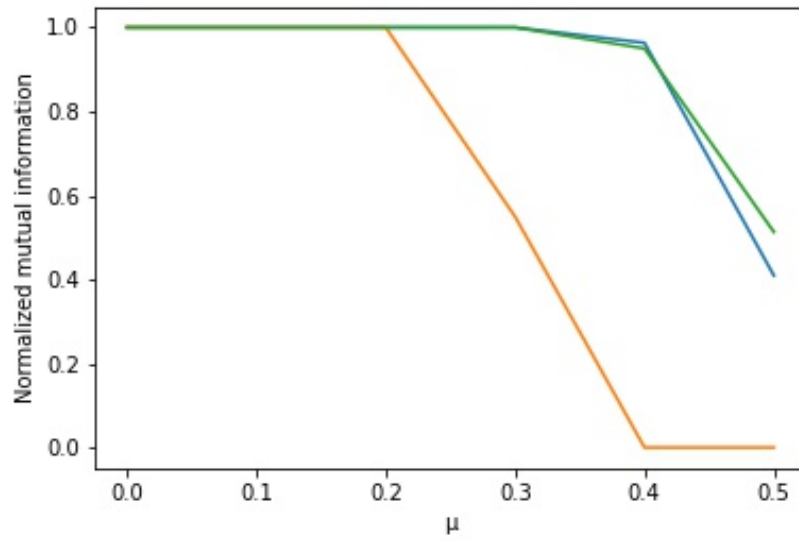


Figure 2: Girvan-Newman benchmark graph.

```

def build_synhtetic_graph(mu = 0.5):
    G = nx.Graph(name = "girvan_newman")
    n = 72
    cluster_div = 24
    for i in range(n):
        G.add_node(i, cluster = i // cluster_div + 1)
    for i in range(n):
        for j in range(i + 1, n):
            if G.nodes[i]['cluster'] == G.nodes[j]['cluster']:
                if random.random() < 20 * (1 - mu) / (cluster_div-1):
                    G.add_edge(i, j)
            else:
                if random.random() < 20 * mu / (n-cluster_div):
                    G.add_edge(i, j)
    return G

def compare_on_synthetic(algo_fn, mus, iter_num=10):
    nmis = []
    for mu in mus:
        NMI = 0
        for _ in range(iter_num):
            G = build_synhtetic_graph(mu)
            clustered_G = algo_fn(G)
            partition = get_graph_ideal_partition(G)
            NMI += clustered_G.normalized_mutual_information(partition).score / iter_num
        nmis.append(NMI)
        print("mu={:.2f}, NMI: {:.3f}".format(mu, NMI))
    return nmis

def get_graph_ideal_partition(G):
    P = {}
    for node in G.nodes(data = True):
        if node[1]['cluster'] not in P:
            P[node[1]['cluster']] = []
        P[node[1]['cluster']].append(node[0])
    node_clusters = P.values()
    return NodeClustering(list(node_clusters), G, 'Ideal')

def run():
    iter_num = 25
    mus = [0.1 * i for i in range(0, 6)]
    algs = {
        "Louvain": algorithms.louvain,
        "Infomap": algorithms.infomap,
        "Walktrap": algorithms.walktrap
    }
    fig = plt.figure()
    plt.xlabel('μ')
    plt.ylabel('Normalized mutual information')
    plt.xticks(mus)

    for algo_name in algs:
        print('ALGORITHM:', algo_name)
        nmis = compare_on_synthetic(algs[algo_name], mus, iter_num)
        plt.plot(mus, nmis, label=algo_name)
        print('=====')

    plt.savefig("images/2_i.jpg")
    plt.legend(loc="best")

run()

```

Figure 3: Benchmark code printout.

2.2 ii

On lancichinetti graph, the best method is Infomap. When linking within the clusters is low, all methods perform quite well, with a slow decline for Louvain and Walktrap. But after $\mu = 0.4$, Louvain drops significantly. When linking within the clusters is high, performance drops for all methods, but we can see that Walktrap overtakes Infomap when $\mu > 0.7$.

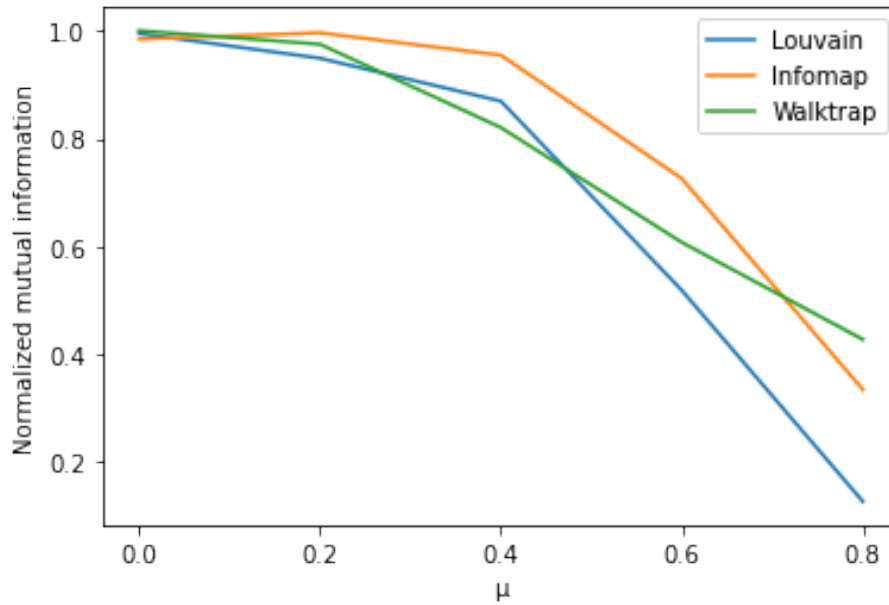


Figure 4: Lancichinetti benchmark graph.

2.3 iii

On a random structure, Infomap is not robust at all. Louvain and Walktrap are doing much better. When node degrees are smaller, Walktrap outperforms both Louvain and Infomap, but when degree gets higher, Louvain and Walktrap are quite close. Overall, Walktrap is the best in this scenario.

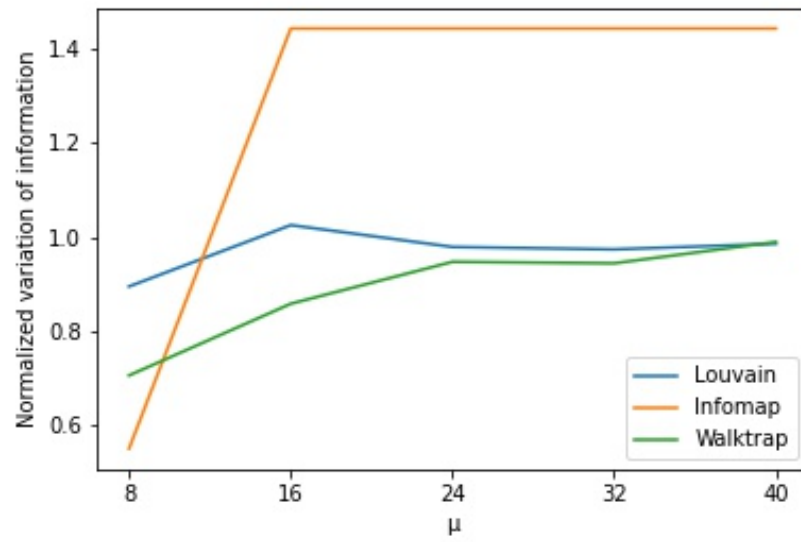


Figure 5: Random structure benchmark graph.

2.4 iv

Louvain performs poorly, on the other hand, Infomap and Walktrap perform well.

```
ALGORITHM: Louvain
NVI: 0.149
=====
ALGORITHM: Infomap
NVI: 0.0
=====
ALGORITHM: Walktrap
NVI: 0.0
```

Figure 6: Results.

2.5 v

Overall, the best method was Walktrap, because it kept the performance even in high values of μ . For the second place, I would choose Louvain, because it was quite close to Walktrap, but with drop-off when dealing with high μ . Infomap performed well on a random structure, but for my project, I would not choose it.

3 Peers, ties and the Internet

3.1 x

Considering the density of real networks, expected accuracy of a method that simply predicts that no links will occur, should be quite high. Real-world networks are very sparse, so classifying into the majority class (no link) is a good "strategy".

3.2 y

```
def leiden(G, node_pairs):
    coms = algorithms.leiden(G)
    C = dict()
    for i in range(len(coms.communities)):
        for node in coms.communities[i]:
            C[node] = i
    part_G = community_louvain.induced_graph(C, G)

    scores = []
    for n1, n2 in node_pairs:
        score = 0
        if C[n1] == C[n2]:
            try:
                nc = len(coms.communities[C[n1]])
                mc = part_G[C[n1]][C[n2]]['weight']
                score = mc / (nc * (nc-1) / 2)
            except KeyError:
                score = 0
        scores.append([n1, n2, score])
    return scores

def sample(G, p):
    nodes = list(G.nodes())
    N = int(G.number_of_edges()*p)

    Ln = []
    while len(Ln) != N:
        node1, node2 = random.choice(nodes), random.choice(nodes)
        if not G.has_edge(node1, node2) and (node1, node2) not in Ln:
            Ln.append((node1, node2))

    Lp = random.sample(list(G.edges()), N)
    G.remove_edges_from(Lp)

    return G, [*Ln, *Lp], len(Ln), N

def AUC(G, algs):
    G, nodes, l, N = sample(G.copy(), 0.1)

    scores = []
    for algo_name in algs:
        predicted = list(algs[algo_name](G, nodes))
        m1, m2 = 0, 0
        for n in range(N):
            s1 = random.sample(predicted[:l], 1)[0]
            s2 = random.sample(predicted[l:], 1)[0]
            if s2[2] > s1[2]:
                m1 += 1
            elif s1[2] == s2[2]:
                m2 += 1
        scores.append((m1 + m2/2) / N)
    return np.array(scores)

def AUC_runs(G, name, algs, n=10):
    auc_l, auc_p, auc_a = 0, 0, 0
    for i in range(n):
        scores = AUC(G, algs)
        auc_l += scores[0]
        auc_p += scores[1]
        auc_a += scores[2]
    print(name + ':')
    print(" Leiden:", auc_l/n)
    print(" Preferential attachment:", auc_p/n)
    print(" Adamic adar index:", auc_a/n)
```

Figure 7: AUC implementation code printout.

3.3 z

On real networks, methods performed really well, while quite bad on random graphs.

For predicting links in the Facebook network, the best method is Adamic adar index. This is because edges mostly appear between common neighbors.

For predicting links in the Gnutella network, the best method is Preferential attachment. This is because of its power law degree distribution, so edges appear between nodes with a high node degree.

For predicting links in the Internet network, the best method is Leiden. This is because Internet most link are within the communities. The second-best method is Preferential attachment, because it is also a scale free network.

```
Gnutella:
  Leiden: 0.5331090675502063
  Preferential attachment: 0.717225640678883
  Adamic adar index: 0.51465278247346
Facebook:
  Leiden: 0.9562337073557747
  Preferential attachment: 0.8306471721636631
  Adamic adar index: 0.9934886093165589
Internet:
  Leiden: 0.8991366040692956
  Preferential attachment: 0.8242758389073913
  Adamic adar index: 0.6970109988525371
Random:
  Leiden: 0.49984799999999996
  Preferential attachment: 0.497544
  Adamic adar index: 0.5000439999999999
```

Figure 8: Results.

4 Get at least 70% right!

For classification problem, I searched existing functions from NetwrokX library. I found two Semi-supervised learning functions: Harmonic Function, and Local and Global Consistency. I ran both 10 times. With Harmonic Function accuracy of 67,95% was obtained, while 72,89% with Local and Global Consistency. So both have accuracy $\geq 65\%$, and LaGC is better than HF. HF is also much slower than LaGC.

```
def read_and_split(filepath):
    G = nx.Graph()

    with open(filepath, 'r') as f:
        f.readline()
        test = []
        for line in f:
            if line.startswith("**"):
                continue

            l = line.split()
            if len(l) == 3:
                if l[1][-3:-1] == '13':
                    G.add_node(int(l[0]))
                    test.append([int(l[0]), l[2]])
                else:
                    G.add_node(int(l[0]), label=l[2])
            else:
                G.add_edge(int(l[0]), int(l[1]))

    return G, test

G, test = read_and_split("data/aps_2008_2013.net")
iter_num = 10
acc = 0
for _ in range(iter_num):
    #predicted = node_classification.harmonic_function(G)
    predicted = node_classification.local_and_global_consistency(G)
    count = 0
    for (node, label) in test:
        if predicted[node-1] == label:
            count += 1

    acc += count
acc /= len(test)*iter_num
print("Accuracy:", acc)
```

Figure 9: Code printout.