Introduction to Network Analysis
Homework 3

Nace Gorenc

April 2021

1 Graph Laplacian matrix

Firstly, let us take a look at the matrix $L = D - A$. On the main diagonal are node degrees $L_{ii} = k_i$ and every other element L_{ij} is either 0 or -1 depending if nodes i and j are adjacent. Now let us observe what we get if we multiply B^T and B. We denote

$$B = [b_1\, b_2\, \cdots\, b_n],$$

where b_i represents all links of the node i. When we multiply i-th row of the matrix B^T and i-th column in the matrix B we get $L_{ii} = b_i^T b_i = k_i$ since we basically count the number of links of the i-th node. On the other hand, we get

$$L_{ij} = b_i^T b_j = B_{1i}B_{1j} + \cdots + B_{mi}B_{mj} = -1$$

for $i \neq j$ due to $B_{li}B_{lj}$ being equal to 0 if nodes i and j are not connected and $B_{li}B_{lj} = -1 \cdot 1 = 1$ if link l connects them (it is arbitrary which one equals to -1 or 1).

Secondly, let us show that all eigenvalues of L are non-negative. From linear algebra we know that the eigenvalues of the matrix $A^H A$, where H denotes conjugate transpose, are non-negative (from $0 \leq ||Ax||_2^2 = (Ax)^H(Ax) = x^H A^H Ax$). In our case we have $L = B^T B = B^H B$ thus all eigenvalues are non-negative.

Lastly, let us show that vector of all ones is an eigenvector of L. If we compute $L \cdot \mathbf{1}$ we get vector of all zeros due to addition of node degree on the diagonal and all adjacent flags -1.

2 Ring graph modularity

2.1 Modularity

$$
\begin{aligned}
Q &= \sum_C \left(\frac{m_C}{m} - \left(\frac{k_C}{m} \right)^2 \right) \\
&= \sum_C \left(\frac{n_C - 1}{n} - \left(\frac{2n_C}{2n} \right)^2 \right) \\
&= \frac{n}{n_C} \left(\frac{n_C - 1}{n} - \left(\frac{n_C}{n} \right)^2 \right) \\
&= \frac{n_C - 1}{n_C} - \frac{n_C}{n} \\
&= 1 - \frac{1}{n_C} - \frac{n_C}{n}
\end{aligned}
$$

2.2 Size that optimizes Q

$$
\frac{\partial Q}{\partial n_C} = \frac{1}{n_C^2} - \frac{1}{n} = 0 \quad \Rightarrow \quad \frac{1}{n_C^2} = \frac{1}{n} \quad \Rightarrow \quad n_C = \sqrt{n}
$$

3 Who's the winner?

For this exercises we use Louvain, Walktrap and Label propagation algorithms from CDlib library.

3.1 Girvan-Newman banchmark

Figure 1 represents community detection accuracy of all three algorithms on Girwan-Newman graph. As we can observe, Louvain and Walktrap algorithms produces the best results. Label propagation produces similar results for ≤ 0.2 and after its NMI score goes to 0. The printout of the code used for this banchmark can be seen below.

```
def gn_benchmark():
    l = [0, 0.1, 0.2, 0.3, 0.4, 0.5]
    l_a = []
    l_b = []
    l_c = []
    for mi in l:
        a = 0
        b = 0
        c = 0
        for _ in range(25):
            g = girvan_newman_graph(mi)
            louvain = algorithms.louvain(g)
```

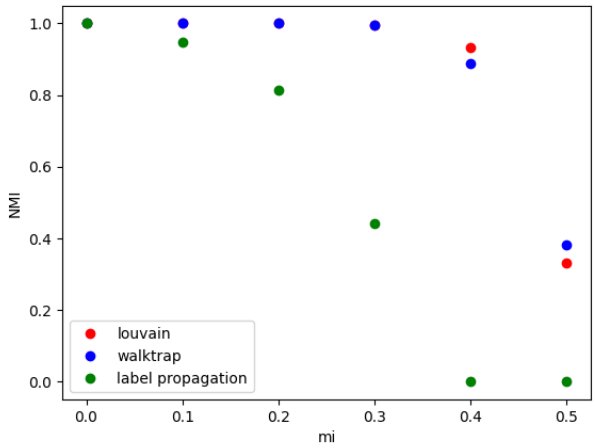

Figure 1: Community detection accuracy on Girwan-Newman graph.

```
            walktrap = algorithms.walktrap(g)
            label_prop = algorithms.label_propagation(g)
            true_labels = classes.NodeClustering([[3*i + j for i in
            ↪  range(24)] for j in range(3)], g)

            a +=
            ↪  evaluation.normalized_mutual_information(true_labels,
            ↪  louvain).score
            b +=
            ↪  evaluation.normalized_mutual_information(true_labels,
            ↪  walktrap).score
            c +=
            ↪  evaluation.normalized_mutual_information(true_labels,
            ↪  label_prop).score
        l_a.append(a/25)
        l_b.append(b/25)
        l_c.append(c/25)
    plt.plot(l, l_a, 'ro')
    plt.plot(l, l_b, 'bo')
    plt.plot(l, l_c, 'go')
    plt.ylabel('NMI')
    plt.xlabel('mi')
    plt.legend(['louvain', 'walktrap', 'label propagation'],
    ↪  loc='lower left')
    plt.savefig('plot31.png')
    plt.show()
```

3.2 Lancichinetti benchmark

Figure 2 represents community detection accuracy of all three algorithms on Lancichinetti graph. In spite of the fact that Label propagation yields the best

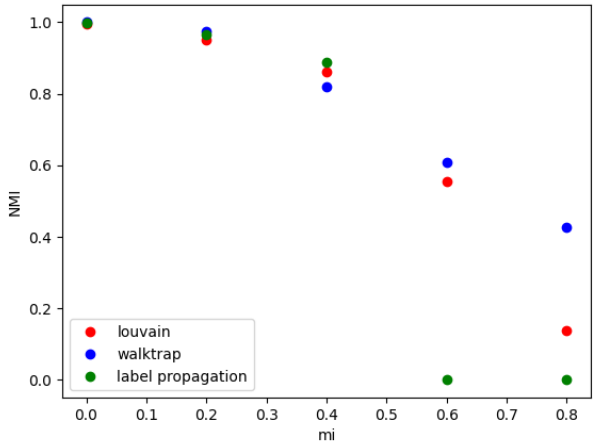

Figure 2: Community detection accuracy on Lancichinetti graph.

results at the < 0.6 it produces the worst results after that boundary. The Walktrap produces the most constantly good results and the Louvain plummets at $= 0.8$.

3.3 Erdős-Rényi benchmark

Figure 3 represents community detection robustness of all three algorithms on Erdős-Rényi graph. Due to the lack of community structures we only have one big community. As we can observe, the most robust algorithm is Label propagation with a constant NVI score of 0. On the other hand, Walktrap and Louvain detects communities in the random graph which is not good.

3.4 Lusseau bottlenose dolphins network

Table 1 represents community detection uncertainty of all three algorithms on Lusseau bottlenose dolphins network. We could say that Walktrap an Lable propagation algorithms produced the best resoults. However, Lable propagation in CDlib library does not support randomization. On the other hand, Walktrap algorithm is based on random walks therefore each run of the algorithm should produce slightly different result. Since it does not, we could declare it for the most deterministic.

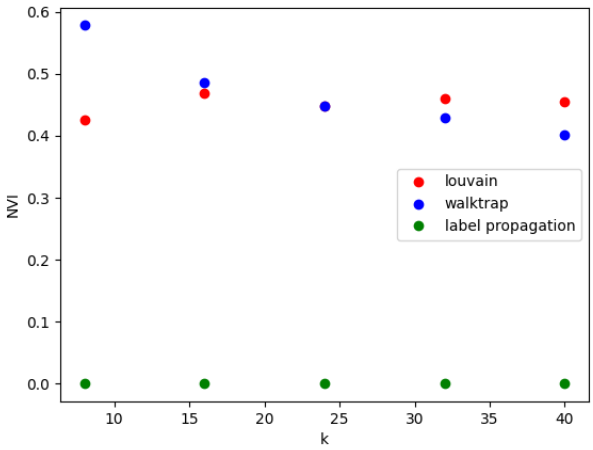

Figure 3: Community detection accuracy on Erdős-Rényi graph.

Algorithm	Community detection uncertainty
Louvain	0.158
Walktrap	0.0
Lable propagation	0.0

Table 1: Community detection uncertainty on Lusseau bottlenose dolphins network.

3.5 Best algorithm

Firstly, let us take a look at the weaknesses of all three algorithms. Label propagation preformed poorly with higher on Girvan-Newman and Lancichinetti benchmarks thus it has the lowest accuracy. Louvain and Walktrap detected communities in random graph hence they are not that robust. Moreover, Louvain preformed slightly worse than Walktrap in Lancichinetti benchmark. Finally, if we consider that the result of uncertainty for Walktrap is accurate, the best algorithm would be Walktrap, as long as we do not try to detect communities in random networks.

4 Peers, ties and the Internet

4.1 x

Considering the fact that real networks are not dense, the expected classification accuracy of this method would be quite high.

4.2 y

The printout of the code used for this problem can be seen below.

```
class AUC:
    def __init__(self, g):
        self.g = g.copy()
        self.m = self.g.number_of_edges()
        self.n = self.g.number_of_nodes()
        self.communities = {k: v for v in
        ↪  algorithms.louvain(self.g).communities for k in v}
        x = 1

    def preferential_attachment_index(self, i, j):
        return self.g.degree[i] * self.g.degree[j]

    def adamic_adar_index(self, i, j):
        n = nx.common_neighbors(self.g, i, j)
        return sum([1 / log(self.g.degree(k)) for k in n])

    def community_index(self, i, j):
        if self.communities[i] != self.communities[j]:
            return 0
        return
        ↪  self.g.subgraph(self.communities[i]).number_of_edges()
        ↪  / binom(len(self.communities[i]), 2)

    def negative(self, m):
        l = set()
        while len(l) < m:
            a, b = sample(self.g.nodes(), 2)
            if a not in self.g.neighbors(b):
                l.add((a, b))
        return list(l)

    def positive(self, m):
        return sample(self.g.edges(), m)

    def run(self, it):
        size = int(self.m/10)
        l_n = self.negative(size)
        l_p = self.positive(size)
        self.g.remove_edges_from(l_p)

        pa_m1, pa_m2 = 0, 0
        aa_m1, aa_m2 = 0, 0
        com_m1, com_m2 = 0, 0

        with tqdm(total=size, desc='Iteracija: ' + str(it),
        ↪  unit='calc') as prog_bar:
```

```
            for _ in range(size):
                n = choice(l_n)
                p = choice(l_p)

                if self.preferential_attachment_index(*n) <
                ↪  self.preferential_attachment_index(*p):
                    pa_m1 += 1
                else:
                    pa_m2 += 1

                if self.adamic_adar_index(*n) <
                ↪  self.adamic_adar_index(*p):
                    aa_m1 += 1
                else:
                    aa_m2 += 1

                if self.community_index(*n) <
                ↪  self.community_index(*p):
                    com_m1 += 1
                else:
                    com_m2 += 1

                prog_bar.update(1)

        return (pa_m1 + pa_m2 / 2) / size, (aa_m1 + aa_m2 / 2) /
        ↪  size, (com_m1 + com_m2 / 2) / size
```

4.3 z

Table 2 represents the average AUC score for different networks.

Network	Preferential a. AUC	Adamic-Adar AUC	Community AUC
Erdős-Rényi	0.506	0.495	0.502
Gnutella	0.754	0.515	0.575
Facebook	0.915	0.996	0.967
Nec	0.91	0.71	0.978

Table 2: AUC for all three algorithms on different networks.

Erdős-Rémyi: All three methods produce result around 0.5 which is expected for a random network.

Gnutella: The best method uses preferential attachment index. This is due to nature of p2p file sharing network which creates a connection between one user that requests a specific file and all of the available users that has that file.

Facebook: The best method uses Adamic-Adar index since this network is small-world network.

Nec: The best method uses community index (Leiden-Louvain). This is due to nec being a real network which consists of communities of densely linked nodes with only few links between the communities.

5 Get at least 70% right!

The strategy:

- find the communities,
- each paper in specific community is most likely published in the same journal,
- for each community calculate which journal has the highest frequency,
- classify new articles in each community based on the prevailing journal.

The classification accuracy of this method over 10 runs is 0.7025 which is slightly better than 0.65 which is given by the method described in exercise. That small difference is probably due to similarity between those two methods.

The printout of the code used for this problem can be seen below.

```
def citation():
    g = nx.read_pajek('data/aps_2008_2013.net')
    truth = gt('data/aps_2008_2013')
    t = 0
    all = 0
    for _ in range(10):
        walktrap = algorithms.walktrap(g).communities
        d = {k: [] for k in range(len(walktrap))}
        for k, coms in enumerate(walktrap):
            for art in coms:
                if "2013" not in art:
                    d[k].append(truth[art])
        for k, coms in enumerate(walktrap):
            if len(d[k]) != 0:
                com, _ = max(Counter(d[k]).items(), key=lambda
                ↪  x:x[1])
                for a in coms:
                    if "2013" in a:
                        if truth[a] == com:
                            t += 1
                        all += 1
            else:
                all += len(coms)

    print(t/all)
```