Homework #3

This homework is complete and will not be changed. The homework does not require a lot of writing, but may require a lot of thinking. It does not require a lot of processing power, but may require efficient programming. It accounts for 12.5% of the course grade. All questions and comments regarding the homework should be directed to Piazza.

Submission details

This homework is due on **May 4th** at 2:00pm, while late days expire on **May 8th** at 1:00pm. The homework must be submitted as a hard-copy in the submission box in front of R 2.49 and also as an electronic version to eUcilnica. It can be prepared in either English or Slovene and either written by hand or typed on a computer. The hard-copy should include (1) this cover sheet with filled out time of the submission and signed honor code, (2) short answers to the questions, which can also demand proofs, tables, plots, diagrams and other, and (3) a printout of all the code required to complete the exercises. The electronic submission should include only (1) answers to the questions in a single file and (2) all the code in a format of the specific programming language. Note that hard-copies will be graded, while electronic submissions will be used for plagiarism detection. The homework is considered submitted only when both versions have been submitted. Failing to include this honor code in the submission will result in **10% deduction**. Failing to submit all the developed code to eUcilnica will result in **50% deduction**.

Honor code

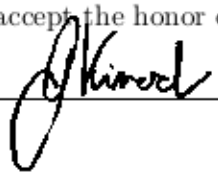
The students are strongly encouraged to discuss the homework with other classmates and form study groups. Yet, each student must then solve the homework by herself or himself without the help of others and should be able to redo the homework at a later time. In other words, the students are encouraged to collaborate, but should not copy from one another. Referring to any solutions obtained from classmates, course books, previous years, found online or other, 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: Jernej Vivod, 63160328

Study group: /

Date & time: 5.5.2020, 11:00

I acknowledge and accept the honor code.

Signature: 

Homework #3

Jernej Vivod
Introduction to Network Analysis

May 5, 2020

Question 1

Show that $L = B^T B$ (L is the graph Laplacian matrix and B is the oriented link incidence matrix). Using this equality further show that all eigenvalues of L are non-negative and that the vector of all ones is an eigenvector of L.

Consider an incidence matrix of a graph where the first node is either the first end-point or the second end point of all links going to other nodes connected with the first node. To recover the node degree from an oriented incidence matrix, we need to sum the absolute values of each column. This is equivalent to taking the dot product of each column with itself. Following the definition of matrix multiplication we can easily see that the elements on the diagonal of the product correspond to the degrees of the nodes.

$$(B^T B)_{i,i} = \sum_j B^T_{ij} B_{ji}$$

On the other hand, taking a dot product of a column with a column corresponding to a different node simply counts the links between the two nodes. Since the first end-point is positive and the second negative, the result of the dot product is equal to the negative sum of the number of links between the two nodes. In the case of a simple graph with no multiple edges, the result will simply be equal to -1 if the corresponding nodes are connected.

Combining these two observations, we can see, that the matrix product $B^T B$ results in a matrix with node degrees on the diagonal and negative values (-1 in the case of a simple graph) on in the i, j-th position, if the j-th and i-th nodes are connected. This can be written as

$$B^T B = D - A,$$

where D is the matrix with node degrees on the diagonal and A is the adjacency matrix. This is equal to the definition of the graph Laplacian matrix L.

A real $n \times n$ symmetric matrix whose eigenvalues are all non-negative is called a positive semi-definite matrix. We know that a matrix $M \in \mathbb{R}^{n \times n}$ is positivie semi-definite if $\forall x \in \mathbb{R}^n, x^T M x \geq 0$.

We can easily show that the Laplacian matrix is positive semi-definite by using its incidence matrix decomposition we proved earlier.

$$x^T L x = x^T B^T B x = (Bx)^t B x = ||Bx||_2 \geq 0$$

Because each row of the graph Laplacian matrix contains the degree of its corresponding node on the diagonal position and negative values on positions corresponding to nodes to which the node corresponding to this node is linked, the values sum up to 0. Multiplying the graph Laplacian matrix with a vector of all ones corresponds to performing a row-wise sum. Therefore, the result will be a vector of all zeros. Thus, a vector of all ones is an eigenvector of the graph Laplacian matrix with an eigenvalue 0.

Question 2

Compute modularity Q of described partition and express it in terms of n_c and n. Find the size of clusters n_c that optimizes modularity Q and express it in terms of n.

We can easily write write the terms in the formula for computing the modularity as functions of n and n_c. We know that the number of links in a community C in such a graph is $n_c - 1$ where n_c is the number of nodes in the community. We also know that the number of nodes is equal to the number of links. and that the total degree is simply two times the number of nodes in each partition.

We can rewrite the equation for computing the modularity as:

$$\begin{aligned} Q &= \sum_C \frac{m_c}{m} - (\frac{k_c}{2m})^2 = \frac{n}{n_c}(\frac{n_c - 1}{n} - (\frac{n_c - 1}{2n})^2) \\ &= (\frac{n_c - 1}{n} - \frac{4n_c^2}{4n})\frac{n}{n_c} = \frac{nn_c - n - nn_c^2}{nn_c} = 1 - \frac{1}{n_c} - \frac{n_c}{n} \end{aligned}$$

We can optimize the modularity with respect to n_c by the standard procedure of differentiating the expression with respect to n_c, setting the derivative equal to 0 and solving for n_c.

$$\begin{aligned} \frac{\partial Q}{\partial n_c} &= 0 - \frac{1}{n_c^2} - \frac{1}{n} = 0 \\ \frac{1}{n_c^2} &= -\frac{1}{n} \\ n_c &= \sqrt{n} \end{aligned}$$

We can see that the modularity is optimized when we set the cluster size n_c equal to the square root of the number of nodes in the graph.

Question 3

Compare community detection methods using specified approach.

Figure 1 shows the visualization of a Girvan-Newman graph with three communities containing 24 nodes each, with expected degree equal to 20 and with the μ parameter equal to 0.1. The code describing the implementation is given at the end of this section.

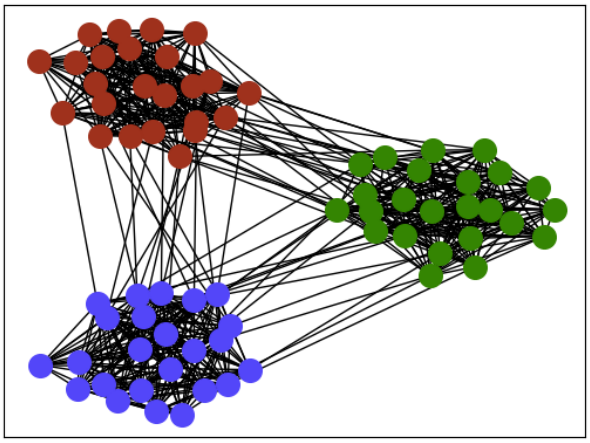


Figure 1: A realization of a Girvan-Newman benchmark graph.

Figure 2 shows the results of applying community detection algorithms on such graphs with varying values of the μ parameter. For each μ parameter value, the results were averaged over 25 iterations of graph construction and community detection. This method was used for all evaluations performed in this section.

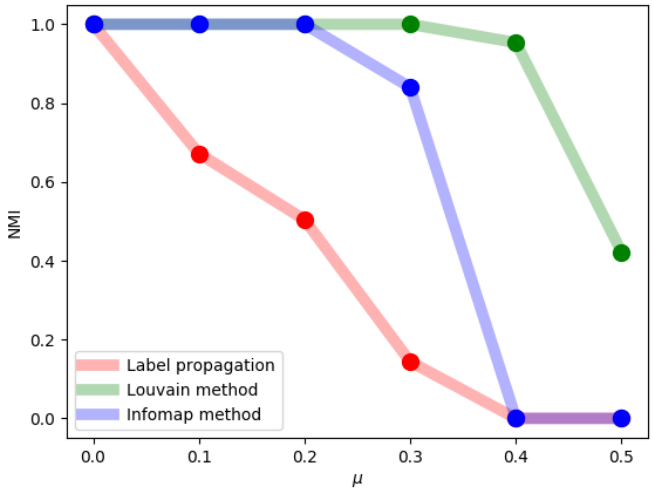


Figure 2: Normalized mutual information between the ground truth and results of community detection algorithms for varying values of the μ parameter of Girvan-Newman benchmark graphs.

We can see that the Louvain method performs best since it reliably detects the correct communities for values of μ lower than 0.4 and then drops sharply. Label propagation performs especially poorly as it descreases steadily for even moderately small values of the μ parameter. We can also see the clear absence of a desired sharp drop in normalized mutual information as the division into communities becomes too blurred to make reliable predictions. The Infomap method is stable up to moderate values of μ and then drops relatively sharply.

Figure 3 shows the results of performing such analysis on Lancichinetti synthetic benchmark graphs.

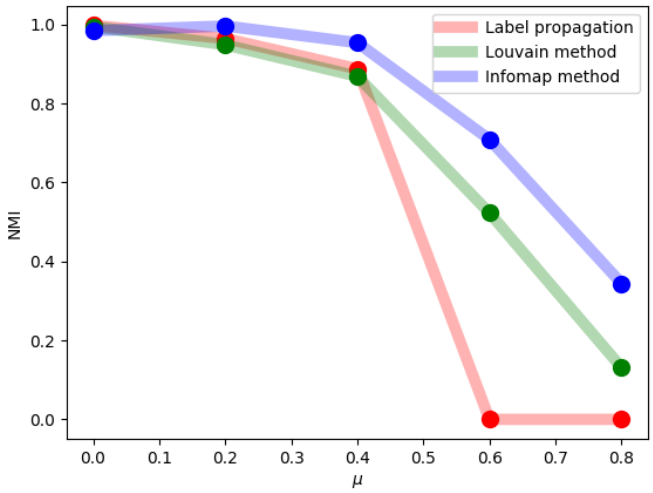


Figure 3: Normalized mutual information between the ground truth and results of community detection algorithms for varying values of the μ parameter of Lancichinetti synthetic benchmark graphs.

We can see that both the Louvain and Infomap method perform similarly and follow a very similar trajectory. All three methods are almost equivalent up to $\mu = 0.4$. The label propagation then drops sharply to 0 which can be seen as desirable at such large values of parameter μ. This makes the label propagation algorithm the algorithm with the most desirable properties when evaluated on such graphs.

Figure 4 shows the normalized variation of information computed using results of detecting communities in random Erdős–Rényi graph realizations,, where each connected component corresponds to a community. The graphs are constructed with 1000 nodes and varying expected average degrees.

We can see that the label propagation method performs best as it best detects that the graph lacks communities and classifies the entire graph (its connected components) as single communities. The infomap method also performs similarly but in our case produces some false detections for very sparse random graphs. On the other hand, the Louvain method does not perform particularly well as it detects non-existent communities in the random graph realizations.

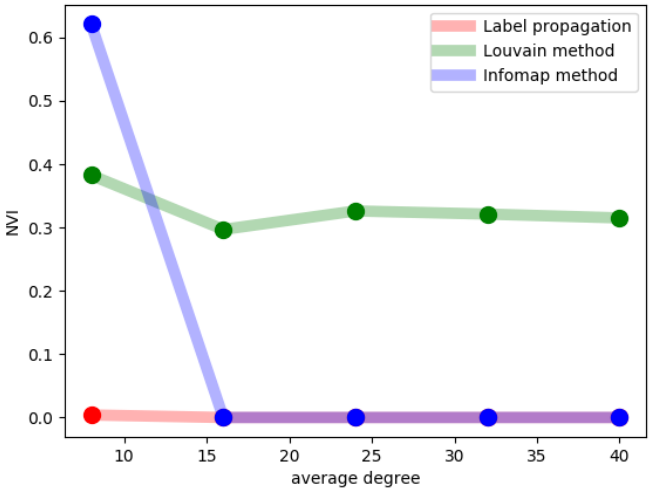


Figure 4: Normalized variation of informaton between the ground truth and results of community detection algorithms for varying expected node degree values of random Erdős–Rényi graph realizations.

Table 1 shows the normalized variation of information computed by considering pairs of runs of community detection algorithms on Lusseau bottlenose dolphins network. We can see that only the Louvain method produces results differing from zero. The other two algorithms perform deterministically and produce same results on each iteration.

method	NVI
Label propagation	0.0
Louvain method	0.1092
Infomap method	0.0

Table 1: Normalized variation of information of results obtained by pairs of sequential algorithm applications.

Overall, all three methods have their advantages and weaknesses. Their usefulness can be determined by studying their known properties and then choosing the method that is best considering what we know about our network. The label propagation method did not produce false detections in the Erdős–Rényi random graphs and had a desirable strong cut-off point when tested on Lancichinetti synthetic graphs. On the other hand, evaluations on simple Girvan-Newman graphs produced an undesirable almost linear drop in its performance that makes results interpretation very difficult. The Louvain method performed very well on Girvan-Newman and Lancichinetti benchmark graphs but suffered from false detections when used on Erdős–Rényi random graphs. In our experiments, the Infomap method performed relatively well on all benchmark graphs and the resuts indicate it to be the best all-round community detection method.

The code written to answer this question is given below.

```python
import networkx as nx
import parse_network
import random
import matplotlib.pyplot as plt


def girvan_newman(num_groups, group_sizes, expected_degree, mu):
    """
    Construct Girvan-Newman benchmark graph with specified properties.

    Args:
        num_groups (int): Number of groups in the benchmark graph
        group_sizes (int): Sizes of groups in the benchmark graph
        expected_degree (int): expected node degree in the benchmark graph
        mu (int): The mu parameter controlling the connectedness of the groups

    Returns:
        (tuple): Constructed graph and ground truth in required format
    """

    # Compute probabilitiy of a link between nodes in same group and
    # link between nodes in different groups.
    p_same = expected_degree*(1-mu)/(group_sizes-1)
    p_other = expected_degree*mu/((num_groups-1)*group_sizes)

    # Initialize empty graph with specified number of nodes.
    graph = nx.empty_graph(n=num_groups*group_sizes, create_using=nx.Graph)

    # Label groups of nodes and construct ground truth in required format.
    nx.set_node_attributes(graph, {idx : {'label' : idx//group_sizes} for idx in range(graph
        .number_of_nodes())})
    attrs = nx.get_node_attributes(graph, 'label')
    ground_truth = [{node_idx for node_idx, label in attrs.items() if label == comm_label}
        for comm_label in set(attrs.values())]

    # Add links.
    node_idxs = list(graph.nodes())
    for idx1 in range(len(node_idxs)-1):
        for idx2 in range(idx1+1, len(node_idxs)):
            if graph.nodes()[idx1]['label'] == graph.nodes()[idx2]['label']:
                # If nodes in same group, add link with probabilitiy p_same.
                if random.random() < p_same:
                    graph.add_edge(idx1, idx2)
            else:
                # If nodes not in same group, add link with probability p_other.
                if random.random() < p_other:
                    graph.add_edge(idx1, idx2)

    # Return constructed graph and ground truth.
    return graph, ground_truth


def draw_girvan_newman(num_groups, group_sizes, expected_degree, mu):
    """
    Draw Girvan-Newman benchmark graph with specified properties and save plot to results
        folder.

    Args:
        num_groups (int): Number of groups in the benchmark graph
        group_sizes (int): Sizes of groups in the benchmark graph
        expected_degree (int): expected node degree in the benchmark graph
        mu (int): The mu parameter controlling the connectedness of the groups

    Returns:
        (int): 0 if success else 1.
    """

```

```
65      # Initialize Girvan-Newman benchmark graph with specified properties.
66      graph, _ = girvan_newman(num_groups, group_sizes, expected_degree, mu)
67
68      # Get unique group labels.
69      labels = set(nx.get_node_attributes(graph, 'label').values())
70
71      # Set graph position (for plotting).
72      pos=nx.spring_layout(graph)
73
74      # Go over group labels and plot nodes corresponding to that group using random color.
75      for label in labels:
76          nodes_nxt_group = [n for (n, d) in graph.nodes(data=True) if d['label'] == label]
77          nx.draw_networkx_nodes(graph, pos, nodelist=nodes_nxt_group, node_size=200, node_
                color=[[random.random(), random.random(), random.random()]])
78
79      # Plot edges.
80      nx.draw_networkx_edges(graph, pos,width=1.0)
81
82      # Save figure.
83      try:
84          plt.savefig('../results/girvan_newman_benchmark_graph.png')
85          return 0
86      except:
87          return 1
88
89
90  def lancichinetti(mu):
91      """
92      Return Lancichinetti benchmark graph with specified mu parameter.
93
94      Args:
95          mu (float): The mu parameter.
96
97      Returns:
98          (tuple): Parsed graph with specified mu parameter and ground truth in required
                format
99      """
100
101     # Get path.
102     fmt = '{:<04}'
103     f_tmp = fmt.format(mu).replace('.', '')
104     f = 'LFR_' + f_tmp[:2] + '_' + f_tmp[2:]
105
106     # Load and parse graph. Get ground truth in required format.
107     graph = parse_network.parse_network('../data/LFR/' + f, create_using=nx.Graph)
108     attrs = nx.get_node_attributes(graph, 'data')
109     ground_truth = [{node_idx for node_idx, label in attrs.items() if label == comm_label}
            for comm_label in set(attrs.values())]
110
111     # Return graph and ground truth in required format.
112     return graph, ground_truth
113
114
115 def erdos_renyi(num_nodes, average_degree):
116     """
117     Construct Erdos-Renyi random graph with specified number of nodes and specified average
            degree.
118
119     Args:
120         num_nodes (int): Number of nodes in constructed Erdos-Renyi random graph.
121         average_degree (int): Average degree in constructed Erdos-Renyi random graph
122
123     Returns:
124         (tuple): Parsed network and ground truth in required format
125     """
126     # Construct graph and return it along with its connected components as communities (
            ground truth).
127     graph = nx.erdos_renyi_graph(num_nodes, average_degree/num_nodes, directed=False)
```

```
    return graph, list(nx.algorithms.components.connected_components(graph))


def bottlenose_dolphins():
    """
    Parse and return Lusseau bottlenose dolphins network.

    Returns:
        (tuple): Parsed network and ground truth in required format
    """

    # Load and parse graph. Get ground truth in required format.
    graph = parse_network.parse_network('../data/dolphins', create_using=nx.Graph)
    attrs = nx.get_node_attributes(graph, 'data')
    ground_truth = [{node_idx for node_idx, label in attrs.items() if label == comm_label}
        for comm_label in set(attrs.values())]

    # Return graph and ground truth in required format.
    return graph, ground_truth


if __name__ == '__main__':
    # Draw Girvan-Newman benchmark graph and save plot.
    draw_girvan_newman(3, 24, 20, 0.1)
```

```
import community
from cdlib import algorithms
import networkx as nx
import benchmark_graphs
import benchmark_utils
import pickle
import os
import sys


def benchmark_gn():
    """
    Perform benchmarking of algorithms on Girvan-Newman benchmark graph.

    Returns:
        (tuple): Used mu values, results for label propagation algorithm, results for
            Louvain method,
        results for Infomap method.
    """

    NUM_REP = 25  # number of algorithm repetitions (on newly constructed graph)
    GN_NUM_GROUPS = 3  # number of groups in benchmark graph
    GN_GROUP_SIZES = 24  # group sizes in benchmark graph
    GN_EXPECTED_DEGREE = 20  # expected degree in benchmark graph
    gn_mu_vals = (0.0, 0.1, 0.2, 0.3, 0.4, 0.5)  # list of mu values for benchmark graph

    # Initialize lists for storing results for different mu values.
    y_vals_label_prop = []
    y_vals_louvain = []
    y_vals_infomap = []

    # Go over mu values.
    print("Performing benchmarks on Girvan-Newman benchmark graphs")
    for idx, mu in enumerate(gn_mu_vals):

        # Initialize lists for storing results for iterations.
        nmi_label_prop = []
        nmi_louvain = []
        nmi_infomap = []

        # Repeat benchmark graph construction and community detection specified number of
```

```
            times.
        for _ in range(NUM_REP):

            # Construct benchmark graph with specified properties.
            graph, ground_truth = benchmark_graphs.girvan_newman(GN_NUM_GROUPS, GN_GROUP_
                SIZES, GN_EXPECTED_DEGREE, mu)

            # Get detections for algorithms.
            res_label_prop = benchmark_utils.normalize_community_format(nx.algorithms.
                community.label_propagation.label_propagation_communities(graph), 'label_
                propagation')
            res_louvain = benchmark_utils.normalize_community_format(community.best_
                partition(graph, randomize=True), 'louvain')
            res_infomap = benchmark_utils.normalize_community_format(algorithms.infomap(
                graph), 'infomap')

            # Compute NMI values.
            nmi_label_prop.append(benchmark_utils.nmi(res_label_prop, ground_truth))
            nmi_louvain.append(benchmark_utils.nmi(res_louvain, ground_truth))
            nmi_infomap.append(benchmark_utils.nmi(res_infomap, ground_truth))

        # Get mean NMI value and set as value for current mu value.
        y_vals_label_prop.append(sum(nmi_label_prop)/len(nmi_label_prop))
        y_vals_louvain.append(sum(nmi_louvain)/len(nmi_louvain))
        y_vals_infomap.append(sum(nmi_infomap)/len(nmi_infomap))
        print("Done {0}/{1}".format(idx+1, len(gn_mu_vals)))

    # Return data for plotting results.
    return gn_mu_vals, y_vals_label_prop, y_vals_louvain, y_vals_infomap


def benchmark_lancichinetti():
    """
    Perform benchmarking of algorithms on Lancichinetti benchmark graph.

    Returns:
        (tuple): Used mu values, results for label propagation algorithm, results for
            Louvain method,
        results for Infomap method.
    """

    NUM_REP = 25  # number of algorithm repetitions (on newly constructed graph)
    lanc_mu_vals = (0.0, 0.2, 0.4, 0.6, 0.8)  # list of mu values for benchmark graph

    # Initialize lists for storing results for different mu values.
    y_vals_label_prop = []
    y_vals_louvain = []
    y_vals_infomap = []

    # Go over mu values.
    print("Performing benchmarks on Lancichinetti benchmark graphs")
    for idx, mu in enumerate(lanc_mu_vals):
        # Initialize lists for storing results for iterations.
        nmi_label_prop = []
        nmi_louvain = []
        nmi_infomap = []

        # Repeat benchmark graph construction and community detection specified number of
            times.
        for _ in range(NUM_REP):

            # Construct benchmark graph with specified properties.
            graph, ground_truth = benchmark_graphs.lancichinetti(mu)

            # Get detections for algorithms.
            res_label_prop = benchmark_utils.normalize_community_format(\
                    nx.algorithms.community.label_propagation.label_propagation_communities(
                        graph), 'label_propagation')
```

```
            res_louvain = benchmark_utils.normalize_community_format(community.best_
                partition(graph, randomize=True), 'louvain')
            res_infomap = benchmark_utils.normalize_community_format(algorithms.infomap(
                graph), 'infomap')

            # Compute NMI values.
            nmi_label_prop.append(benchmark_utils.nmi(res_label_prop, ground_truth))
            nmi_louvain.append(benchmark_utils.nmi(res_louvain, ground_truth))
            nmi_infomap.append(benchmark_utils.nmi(res_infomap, ground_truth))

        # Get mean NMI value and set as value for current mu value.
        y_vals_label_prop.append(sum(nmi_label_prop)/len(nmi_label_prop))
        y_vals_louvain.append(sum(nmi_louvain)/len(nmi_louvain))
        y_vals_infomap.append(sum(nmi_infomap)/len(nmi_infomap))
        print("Done {0}/{1}".format(idx+1, len(lanc_mu_vals)))

    # Return data for plotting results.
    return lanc_mu_vals, y_vals_label_prop, y_vals_louvain, y_vals_infomap


def benchmark_er():
    """
    Perform benchmarking of algorithms on Erdos-Renyi random graph.

    Returns:
        (tuple): Used average node degrees, results for label propagation algorithm, results
             for Louvain method,
        results for Infomap method.
    """

    NUM_REP = 25  # number of algorithm repetitions (on newly constructed graph)
    NUM_NODES = 1000  # number of nodes in benchmark graph
    er_average_degrees = (8, 16, 24, 32, 40)  # list of average degrees for benchmark graph

    # Initialize lists for storing results for different mu values.
    y_vals_label_prop = []
    y_vals_louvain = []
    y_vals_infomap = []

    # Go over mu values.
    print("Performing benchmarks on Erdos-Renyi random graphs")
    for idx, av_deg in enumerate(er_average_degrees):

        # Initialize lists for storing results for iterations.
        nvi_label_prop = []
        nvi_louvain = []
        nvi_infomap = []

        # Repeat benchmark graph construction and community detection specified number of
            times.
        for _ in range(NUM_REP):

            # Construct benchmark graph with specified properties.
            graph, ground_truth = benchmark_graphs.erdos_renyi(NUM_NODES, av_deg)

            # Get detections for algorithms.
            res_label_prop = benchmark_utils.normalize_community_format(\
                    nx.algorithms.community.label_propagation.label_propagation_communities(
                        graph), 'label_propagation')
            res_louvain = benchmark_utils.normalize_community_format(community.best_
                partition(graph, randomize=True), 'louvain')
            res_infomap = benchmark_utils.normalize_community_format(algorithms.infomap(
                graph), 'infomap')

            # Compute NMI values.
            nvi_label_prop.append(benchmark_utils.nvi(res_label_prop, ground_truth))
            nvi_louvain.append(benchmark_utils.nvi(res_louvain, ground_truth))
            nvi_infomap.append(benchmark_utils.nvi(res_infomap, ground_truth))
```

```
161
162
163             # Get mean NMI value and set as value for current mu value.
164             y_vals_label_prop.append(sum(nvi_label_prop)/len(nvi_label_prop))
165             y_vals_louvain.append(sum(nvi_louvain)/len(nvi_louvain))
166             y_vals_infomap.append(sum(nvi_infomap)/len(nvi_infomap))
167             print("Done {0}/{1}".format(idx+1, len(er_average_degrees)))
168
169     # Return data for plotting results.
170     return er_average_degrees, y_vals_label_prop, y_vals_louvain, y_vals_infomap
171
172
173 def benchmark_dolphins():
174     """
175     Perform benchmarking of algorithms on Lusseau bottlenose dolphins network.
176
177     Returns:
178         (tuple): results for label propagation algorithm, results for Louvain method,
179         results for Infomap method.
180     """
181
182     NUM_REP = 25  # number of algorithm repetitions
183
184     # Initialize lists for storing results.
185     det_label_prop = []
186     det_louvain = []
187     det_infomap = []
188
189     # Repeat benchmark graph construction and community detection specified number of times.
190     print("Performing benchmarks on Lusseau bottlenose dolphins network")
191     for idx in range(NUM_REP):
192
193         # Construct benchmark graph with specified properties.
194         graph, ground_truth = benchmark_graphs.bottlenose_dolphins()
195
196         # Get detections for algorithms.
197         res_label_prop = benchmark_utils.normalize_community_format(\
198                 nx.algorithms.community.label_propagation.label_propagation_communities(
                        graph), 'label_propagation')
199         res_louvain = benchmark_utils.normalize_community_format(community.best_partition(
                graph, randomize=True), 'louvain')
200         res_infomap = benchmark_utils.normalize_community_format(algorithms.infomap(graph),
                'infomap')
201
202         # Add detections to results list.
203         det_label_prop.append(res_label_prop)
204         det_louvain.append(res_louvain)
205         det_infomap.append(res_infomap)
206
207         print("Done {0}/{1}".format(idx+1, NUM_REP))
208
209     # Initialize lists for computing pairwise NVI for detections.
210     pairwise_nvi_label_prop = []
211     pairwise_nvi_louvain = []
212     pairwise_nvi_infomap = []
213
214     # Compute NVI of detections (pairwise).
215     for idx in range(NUM_REP-1):
216         pairwise_nvi_label_prop.append(benchmark_utils.nvi(det_label_prop[idx], det_label_
                prop[idx+1]))
217         pairwise_nvi_louvain.append(benchmark_utils.nvi(det_louvain[idx], det_louvain[idx
                +1]))
218         pairwise_nvi_infomap.append(benchmark_utils.nvi(det_infomap[idx], det_infomap[idx
                +1]))
219
220     # Get mean NVI values and set as results
221     res_label_prop = sum(pairwise_nvi_label_prop)/len(pairwise_nvi_label_prop)
222     res_louvain = sum(pairwise_nvi_louvain)/len(pairwise_nvi_louvain)
```

```
    res_infomap = sum(pairwise_nvi_infomap)/len(pairwise_nvi_infomap)

    # Return data for plotting results.
    return res_label_prop, res_louvain, res_infomap


def plot_results(x, y, labels_y, x_label, y_label, file_name):
    """
    Plot benchmarking results.

    Args:
        x (list): x-axis values
        y (list): y-axis values (list of lists)
        labels_y (list): Lables for the drawn lines (for legend)
        x_label (str): The x-axis label
        y_label (str): The y-axis label
        file_name (str): The file name for the saved plot
    """

    import matplotlib.pyplot as plt

    # Plot results.
    fig = plt.figure()
    plt.plot(x, y[0], 'r.', markersize=20)
    plt.plot(x, y[0], 'r-', label=labels_y[0], alpha=0.3, linewidth=7)
    plt.plot(x, y[1], 'g.', markersize=20)
    plt.plot(x, y[1], 'g-', label=labels_y[1], alpha=0.3, linewidth=7)
    plt.plot(x, y[2], 'b.', markersize=20)
    plt.plot(x, y[2], 'b-', label=labels_y[2], alpha=0.3, linewidth=7)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.legend()

    # Save plot to file.
    plt.savefig('../results/' + file_name)


def perform_benchmarking():
    """
    Perform implemented benchmarks.

    Returns:
        (int): 0 if successful else 1
    """

    try:

        # Compute results using benchmarking functions.
        gn_mu_vals, y_vals_label_prop_gn, y_vals_louvain_gn, y_vals_infomap_gn = benchmark_
            gn()
        lanc_mu_vals, y_vals_label_prop_lc, y_vals_louvain_lc, y_vals_infomap_lc = benchmark
            _lancichinetti()
        er_average_degrees, y_vals_label_prop_er, y_vals_louvain_er, y_vals_infomap_er =
            benchmark_er()
        res_label_prop_dolph, res_louvain_dolph, res_infomap_dolph = benchmark_dolphins()

        # Build dictionary for results.
        res = {'gn_mu_vals' : gn_mu_vals,
               'y_vals_label_prop_gn' : y_vals_label_prop_gn,
               'y_vals_louvain_gn' : y_vals_louvain_gn,
               'y_vals_infomap_gn' : y_vals_infomap_gn,
               'lanc_mu_vals' : lanc_mu_vals,
               'y_vals_label_prop_lc' : y_vals_label_prop_lc,
               'y_vals_louvain_lc' : y_vals_louvain_lc,
               'y_vals_infomap_lc' : y_vals_infomap_lc,
               'er_average_degrees' : er_average_degrees,
               'y_vals_label_prop_er' : y_vals_label_prop_er,
               'y_vals_louvain_er' : y_vals_louvain_er,
```

```
                'y_vals_infomap_er' : y_vals_infomap_er,
                'res_label_prop_dolph' : res_label_prop_dolph,
                'res_louvain_dolph' : res_louvain_dolph,
                'res_infomap_dolph' : res_infomap_dolph,
                }

        # Save dictionary.
        with open('../results/cached_data_3.p', 'wb') as f:
            pickle.dump(res, f, pickle.HIGHEST_PROTOCOL)
        return 0
    except:
        return 1


if __name__ == '__main__':
    import argparse

    # Parse optional flag for parsing cached results.
    parser = argparse.ArgumentParser()
    parser.add_argument('--load-cached', action='store_true')
    args = parser.parse_args()

    # If not loading cached results, perform benchmarking.
    if not args.load_cached:
        res = perform_benchmarking()
        if res == 0:
            print("Benchmarking completed")
        else:
            print("Something went wrong during the benchmarking process")

    if os.path.isfile('../results/cached_data_3.p'):

        # If cached data exists, load it and visualize results.
        with open('../results/cached_data_3.p', 'rb') as f:
            res = pickle.load(f)

        # Plot results for Girvan-Newman benchmark graph.
        plot_results(res['gn_mu_vals'], [res['y_vals_label_prop_gn'], res['y_vals_louvain_gn
            '], res['y_vals_infomap_gn']],
                ['Label propagation', 'Louvain method', 'Infomap method'], x_label=r'$\mu$',
                    y_label='NMI', file_name='benchmark_gn.png')

        # Plot results for Lancichinetti benchmark graph.
        plot_results(res['lanc_mu_vals'], [res['y_vals_label_prop_lc'], res['y_vals_louvain_
            lc'], res['y_vals_infomap_lc']],
                ['Label propagation', 'Louvain method', 'Infomap method'], x_label=r'$\mu$',
                    y_label='NMI', file_name='benchmark_lc.png')

        # Plot results for Erdos-Renyi benchmark (random) graph.
        plot_results(res['er_average_degrees'], [res['y_vals_label_prop_er'], res['y_vals_
            louvain_er'], res['y_vals_infomap_er']],
                ['Label propagation', 'Louvain method', 'Infomap method'], x_label='average
                    degree', y_label='NVI', file_name='benchmark_er.png')

        # Save results for dolphin network in form of Markdown table.
        with open('../results/res_dolphins.txt', 'w') as f:
            f.write('| method              | NVI    |\n')
            f.write('|---------------------|--------|\n')
            f.write('| Label propagation | {0:.4f} |\n'.format(res['res_label_prop_dolph']))
            f.write('| Louvain method    | {0:.4f} |\n'.format(res['res_louvain_dolph']))
            f.write('| Infomap method    | {0:.4f} |\n'.format(res['res_infomap_dolph']))

        sys.exit(0)
    else:
        print("Cached data does not exist. Please run script without the --load-cached flag.
            ")
        sys.exit(1)
```

```python
import math
import scipy as sp
from sklearn.metrics import normalized_mutual_info_score
from pyitlib import discrete_random_variable as drv


def nmi(prediction, ground_truth):
    """
    Compute normalized mutual information of predicted communities with ground truth.

    Args:
        prediction (list): List of sets representing found communities
        ground_truth (list): List of sets representing ground truth

    Returns:
        (float): Computed normalized mutual information score
    """

    # Assign labels to nodes, sort by node index and get labels.
    first_partition_c = [x[1] for x in sorted([(node, nid) for nid, cluster in enumerate(
        prediction) for node in cluster], key=lambda x: x[0])]
    second_partition_c = [x[1] for x in sorted([(node, nid) for nid, cluster in enumerate(
        ground_truth) for node in cluster], key=lambda x: x[0])]

    # Compute normalized mutual information.
    return normalized_mutual_info_score(first_partition_c, second_partition_c)


def nvi(prediction, ground_truth):
    """
    Compute normalized variation of information of predicted communities with ground truth.

    Args:
        prediction (list): List of sets representing found communities
        ground_truth (list): List of sets representing ground truth

    Returns:
        (float): Computed normalized variation of information score
    """

    # Assign labels to nodes, sort by node index and get labels.
    first_partition_c = [x[1] for x in sorted([(node, nid) for nid, cluster in enumerate(
        prediction) for node in cluster], key=lambda x: x[0])]
    second_partition_c = [x[1] for x in sorted([(node, nid) for nid, cluster in enumerate(
        ground_truth) for node in cluster], key=lambda x: x[0])]

    # Compute normalized mutual information.
    return drv.information_variation(first_partition_c, second_partition_c, base=math.e)/
        math.log(len(first_partition_c))


def normalize_community_format(res, method):
    """
    Normalize predicted communities format for all methods to list of sets.

    Args:
        res (list): Predicted communities as returned by method
        method (str): Method used to predict the communities ('label_propagation', 'louvain
            ', 'infomap')

    Returns:
        (list): List of sets representing found communities
    """

    if method == 'label_propagation':
        return list(res)
    elif method == 'louvain':
```

```
        return [{node_idx for node_idx, label in res.items() if label == com_label} for com_
            label in set(res.values())]
    elif method == 'infomap':
        return list(map(lambda x: set(x), res.communities))
```

```
import networkx as nx
import re


def parse_network(path, *args, **kwargs):
    """
    Parse network and add associated data. The data should be specified using the LNA format
        .
    Author: Jernej Vivod

    Args:
        path (str): Path to the data file.
        args (tuple): Arguments for the networkx read_edgelist method.
        **kwargs (dict): Keyworkd arguments for the networkx read_edgelist method.

    Returns:
        (obj): Networkx graph representation with added node names and data.
    """

    def parse_line_data(line):
        node_idx = line[:line.index('"')].split(' ')[1]
        node_name = re.findall(r'"([^"]*)"', line)[0]
        node_data = line.split(" ")[-1].strip()
        return node_idx, node_name, node_data

    # Parse graph from edge list.
    graph = nx.read_edgelist(path, *args, **kwargs)

    # Initialize dictionaries for parsing data.
    names = dict()
    data = dict()

    # Flag indicating the start of parsing.
    parse = False
    with open(path, 'r') as f:

        # Go over lines.
        for line in f:

            # If parsing.
            if parse:
                if len(line.split(" ")) == 1:
                    # If found last delimiter, add data to graph and return.
                    nx.set_node_attributes(graph, names, 'name')
                    nx.set_node_attributes(graph, data, 'data')
                    return graph
                else:
                    # Parse node index, name and associated data and add to dictionaries.
                    node_idx, node_name, node_data = parse_line_data(line)
                    data[node_idx] = node_data
                    names[node_idx] = node_name
            else:
                if len(line.split(" ")) == 1:
                    # If at first delimiter, set parse flag.
                    parse = True
                else:
                    pass
```

Question 4

Answer the given question concerning link prediction in real-world networks. Implement a framework for evaluating link prediction methods and use it in conjunction with specified sample netwotks.

We know that real networks are usually sparse. A single node is only connected to a few other nodes. Real networks are also usually large. Assuming these two properties, a link between two randomly chosen nodes in such a network is very unlikely. The classification accuracy of a method which simply predicts that no link will occur will therefore tend to 1.0 as the size of the network increases.

Table 2 shows the average AUC obtained by running the algorithms on each network 10 times.

method	preferential attachment	Adamic-Adar	Community
Erdős-Rényi	0.5109	0.5006	0.6548
Gnutella	0.7169	0.5128	0.7866
Facebook	0.8292	0.9925	0.9513
nec	0.8181	0.6854	0.9280

Table 2: AUC scores obtained by benchmarking different link prediction methods on sample networks.

For the Erdős-Rényi graph, the community algorithm proves most successful. As all connected components of the random graph consist of a single community, the indices will be the same for each node as long as the community detection algorithm successfully classifies the connected components as single communities. This can be problematic when using the Louvain method. The equal indices best capture the random structure of this kind of graph. For the Gnutella network, the community index and the preferential attachment models perform well. By plotting a portion of the network, we can clearly see, that nodes tend to form communities centered around hubs that have a few edges leading out to other communities. The degree distribution of the network also follows the power-law which is why the preferential attachment assumption also yields satisfactory results. The absence of triadic closure makes the Adamic-Adar index unsuitable for such a network. On the other hand, the Adamic-Adar index performs very well on the Facebook social circles networks as triadic closure or abundance of triangles is a notable property of such social networks. The network also follows a power-law degree distribution and contains notable communities, which is why the preferential attachment index and the community index also perform relatively well. Similarly as the Gnutella network, the nec overlay map lacks triadic closures, which is to be expected for these types of networks as such connections would introduce redundancy. The degree distribution again follows an approximate power-law distribution and the nodes again form communities with central hubs which is why the preferential attachment and community indices perform relatively well with the community index best capturing the nature of this particular network.

The code written to answer this question is given below.

```python
import random
import math
import community
from scipy.special import comb
from collections import Counter
import parse_network


def link_prediction_auc(network, prediction_func):
    """
    Perform link prediction using specified method on specified network and
    return AUC.

    Args:
        network (object): The network on which to evaluate the link prediction mechanism
        prediction_func (function): The function implementing link prediction index
            computation

    Returns:
        (float): AUC score of method
    """

    # Randomly sample m/10 pairs of nodes that are not yet
    # linked and store them into L_{N}.
    negative_examples = []
    while len(negative_examples) < math.ceil(network.number_of_edges()/10):
        pair = tuple(random.sample(network.nodes, 2))
        if not network.has_edge(*pair):
            negative_examples.append(pair)

    # Randomly sample m/10 links from the network, remove them from the
    # network and store them into L_{P}.
    positive_examples = random.sample(network.edges, math.ceil(network.number_of_edges()/10)
        )
    network.remove_edges_from(positive_examples)


    # Compute the link prediction index s for all pairs of nodes in union of L_{N}
    # and L_{P}.
    # link_prediction_indices = lp_idx(negative_examples + positive_examples)
    lp_ind_n = [prediction_func(network, link) for link in negative_examples]
    lp_ind_p = [prediction_func(network, link) for link in positive_examples]

    # Sample m/10 pairs from L_{N} and m/10 pairs from L_{P} with repetitions.
    samp_lp_ind_n = random.choices(lp_ind_n, k=int(math.ceil(network.number_of_edges()/10)))
    samp_lp_ind_p = random.choices(lp_ind_p, k=int(math.ceil(network.number_of_edges()/10)))
    comp = [val_p - val_n for val_p, val_n in zip(samp_lp_ind_p, samp_lp_ind_n)]
    num_p_larger = sum(el > 0 for el in comp)
    num_eq = sum(el == 0 for el in comp)

    return (num_p_larger + num_eq/2)/(math.ceil(network.number_of_edges()/10))


def get_index_func(kind, network):
    """
    Get specified function for computing link prediction index.

    Args:
        kind (str): Which link prediction index function to return
        network (object): The network for which link prediction will be run

    Returns:
        (function): Specified function for computing link prediction index.
    """

    ### Function used to compute the link indices ###

```

```
66      # Compute preferential attachment index.
67      def preferential_attachment_index(network, link):
68          return network.degree[link[0]]*network.degree[link[1]]
69
70      # Compute Adamic-Adar index.
71      def adamic_adar_index(network, link):
72          return sum(1/math.log(network.degree(x))
73                  for x in set(network.neighbors(link[0])).intersection(network.neighbors(link
                        [1])))
74
75      # Compute community index.
76      def community_index(network, communities, nc, mc, link):
77          if communities[link[0]] != communities[link[1]]:
78              return 0
79          else:
80              return mc[communities[link[0]]]/comb(nc[communities[link[0]]], 2)
81
82      ################################################
83
84      # Get counts of edges in communities.
85      def get_mc(network, communities):
86          counts = dict.fromkeys(set(communities.values()), 0)
87          for edge in network.edges():
88              if communities[edge[0]] == communities[edge[1]]:
89                  counts[communities[edge[0]]] += 1
90          return counts
91
92
93      # Return specified function.
94      if kind == 'preferential-attachment':
95          return preferential_attachment_index
96      elif kind == 'adamic-adar':
97          return adamic_adar_index
98      elif kind == 'community':
99          communities = community.best_partition(network)
100         nc = Counter(communities.values())
101         mc = get_mc(network, communities)
102         return lambda network, link: community_index(network, communities, nc, mc, link)
103     else:
104         raise(ValueError('Unknown index function specified.'))
105
106
107 def get_benchmark_network(name):
108     """
109     Get benchmark networks for comparing link prediction methods.
110
111     Args:
112         name (str): Name of the benchmark network to get
113
114     Returns:
115         (object): Parsed network
116     """
117
118     if name == 'erdos-renyi':
119         # Erdos-Renyi random graph
120         NUM_NODES = 25000
121         AVERAGE_DEGREE = 10
122         from benchmark_graphs import erdos_renyi
123         network, _ = erdos_renyi(num_nodes=NUM_NODES, average_degree=AVERAGE_DEGREE)
124         return network
125
126     elif name == 'gnutella':
127         # Gnutella peer-to-peer file sharing network
128         return parse_network.parse_network('../data/gnutella', create_using=nx.Graph)
129
130     elif name == 'facebook':
131         # Facebook social circles network
132         return parse_network.parse_network('../data/circles', create_using=nx.Graph)
```

```
133
134     elif name == 'nec':
135         # nec overlay map
136         return parse_network.parse_network('../data/nec', create_using=nx.Graph)
137
138     else:
139         raise(ValueError('Unknown network specified'))
140
141
142 def main():
143     """
144     Perform benchmarking of link prediction methods and save results.
145     """
146
147     # Number of runs of each method to perform.
148     NUM_RUNS = 1
149
150     ### RUN EVALUTATIONS AND WRITE RESULTS TO FILE ###
151     # Initialize lists for storing results for runs. Perform evaluations and save average
            result.
152
153     auc_pref_vals = []
154     auc_ad_vals = []
155     auc_comm_vals = []
156     network = get_benchmark_network('erdos-renyi')
157     for idx in range(NUM_RUNS):
158         auc_pref_vals.append(link_prediction_auc(network, get_index_func('preferential-
                attachment', network)))
159         auc_ad_vals.append(link_prediction_auc(network, get_index_func('adamic-adar',
                network)))
160         auc_comm_vals.append(link_prediction_auc(network, get_index_func('community',
                network)))
161     with open('../results/res_auc.txt', 'w') as f:
162         f.write("|             | preferential attachment | adamic-Adar | community |\n")
163         f.write("|-------------|-------------------------|-------------|-----------|\n")
164         f.write("| Erdos-Renyi | {0:.4f}                 | {1:.4f}     | {2:.4f}    |\n".
                format(sum(auc_pref_vals)/len(auc_pref_vals),
165             sum(auc_ad_vals)/len(auc_ad_vals), sum(auc_comm_vals)/len(auc_comm_vals)))
166
167     auc_pref_vals = []
168     auc_ad_vals = []
169     auc_comm_vals = []
170     network = get_benchmark_network('gnutella')
171     for idx in range(NUM_RUNS):
172         auc_pref_vals.append(link_prediction_auc(network, get_index_func('preferential-
                attachment', network)))
173         auc_ad_vals.append(link_prediction_auc(network, get_index_func('adamic-adar',
                network)))
174         auc_comm_vals.append(link_prediction_auc(network, get_index_func('community',
                network)))
175
176     with open('../results/res_auc.txt', 'a') as f:
177         f.write("| gnutella    | {0:.4f}                 | {1:.4f}     | {2:.4f}    |\n".
                format(sum(auc_pref_vals)/len(auc_pref_vals),
178             sum(auc_ad_vals)/len(auc_ad_vals), sum(auc_comm_vals)/len(auc_comm_vals)))
179
180     auc_pref_vals = []
181     auc_ad_vals = []
182     auc_comm_vals = []
183     network = get_benchmark_network('facebook')
184     for idx in range(NUM_RUNS):
185         auc_pref_vals.append(link_prediction_auc(network, get_index_func('preferential-
                attachment', network)))
186         auc_ad_vals.append(link_prediction_auc(network, get_index_func('adamic-adar',
                network)))
187         auc_comm_vals.append(link_prediction_auc(network, get_index_func('community',
                network)))
188
```

```
    with open('../results/res_auc.txt', 'a') as f:
        f.write("| facebook    | {0:.4f}                  | {1:.4f}      | {2:.4f}    |\n".
            format(sum(auc_pref_vals)/len(auc_pref_vals),
            sum(auc_ad_vals)/len(auc_ad_vals), sum(auc_comm_vals)/len(auc_comm_vals)))

    auc_pref_vals = []
    auc_ad_vals = []
    auc_comm_vals = []
    network = get_benchmark_network('nec')
    for idx in range(NUM_RUNS):
        auc_pref_vals.append(link_prediction_auc(network, get_index_func('preferential-
            attachment', network)))
        auc_ad_vals.append(link_prediction_auc(network, get_index_func('adamic-adar',
            network)))
        auc_comm_vals.append(link_prediction_auc(network, get_index_func('community',
            network)))

    with open('../results/res_auc.txt', 'a') as f:
        f.write("| nec         | {0:.4f}                  | {1:.4f}      | {2:.4f}    |".
            format(sum(auc_pref_vals)/len(auc_pref_vals),
            sum(auc_ad_vals)/len(auc_ad_vals), sum(auc_comm_vals)/len(auc_comm_vals)))


if __name__ == '__main__':
    main()
```

Question 5

Propose a strategy of predicting journals corresponding to physics papers that gives at least $\approx 70\%$ classification accuracy. The strategy can use any network analysis method or other approach as long as it scales better than $\mathcal{O}(n^2)$ in real networks.

We can solve this task by splitting the dataset (our network) into training and test sets. In this case, the training set consists of nodes corresponding to papers published in years before 2013 and the test set of papers published in year 2013. We can extract useful features from these nodes such as the node degree, the mean degree of its neighbours, the maximum degree of its neighbours, the minimum degree of its neighbours, the standard deviation of the degree of its neighbours, the number of neighbours with the same label, the number of triangles the node is part of, the maximum number of triangles a neighbour is part of, the minimum number of triangles the neighbour is part of, the mean number of triangles a neighbour is part of and the number of times each label (journal title) appears in the neighbourhood.

This yields a vector describing each node in the network. We can compute such features for each node in the training and test sets and get a canonical matrix form where each row corresponds to a node.

Standard machine learning algorithms can now be used to predict the node labels (titles of journals). We used a random-forest classifier to perform the classifications. We compared the results with a baseline classifier that predicts the label (corresponding journal) of each node to be the most common label in its neighbourhood.

The average classification accuracy of our approach is **0.88**. The average classification accuracy of the baseline approach is **0.66**.

Table 3 shows the classification accuracies and other statistics of our approach. Table 4 shows the same statistics for the baseline approach. We can clearly see that our approach significantly outperforms the baseline.

Label	Precision	Recall	f1-score	Support
PhysRevA	0.86	0.89	0.88	2385
PhysRevB	0.87	0.95	0.91	4090
PhysRevC	0.91	0.93	0.92	883
PhysRevD	0.96	0.96	0.96	2868
PhysRevE	0.86	0.80	0.83	1875
PhysRevLett	0.86	0.80	0.83	3336
PhysRevSTAB	0.89	0.74	0.81	129
PhysRevX	0.81	0.26	0.39	86
RevModPhys	0.96	0.51	0.67	43

Table 3: Precision, recall, f1-score and support for our approach.

Label	Precision	Recall	f1-score	Support
PhysRevA	0.77	0.61	0.880.68	23852385
PhysRevB	0.79	0.62	0.69	4090
PhysRevC	0.86	0.82	0.84	883
PhysRevD	0.88	0.89	0.89	2868
PhysRevE	0.85	0.52	0.64	1875
PhysRevLett	0.36	0.60	0.45	3336
PhysRevSTAB	0.80	0.60	0.69	129
PhysRevX	0.00	0.00	0.00	86
RevModPhys	0.00	0.00	0.00	43

Table 4: Precision, recall, f1-score and support for baseline approach.

The code written to answer this question is given below.

```

import networkx as nx
import numpy as np
import re
import parse_network
from sklearn import preprocessing
from sklearn.metrics import classification_report


def get_tts(network):
    """
    Get train-test split for network nodes corresponding to papers published in 2013
    are added to the test set and nodes corresponding to papers published earlier are
    added to the training set.

    Args:
        network (object): Network on which to perform the train-test split.

    Returns:
```

```
        (tuple): Indices of nodes training nodes and indices of test nodes.
    """

    # Find nodes corresponding to papers published in 2013 (test set) and nodes
        corresponding to papers
    # published earlier (train set).
    train_idxs = [idx for idx, data in dict(network.nodes(data=True)).items() if data['name'
        ][-4:] != '2013']
    test_idxs = [idx for idx, data in dict(network.nodes(data=True)).items() if data['name'
        ][-4:] == '2013']

    # Return training data indices and test data indices.
    return train_idxs, test_idxs


def get_features_node(node, network, bow):
    """
    Get features for specified node.

    Args:
        node (str): Node index
        network (object): The network the node is part of
        bow (list): List of all node labels

    Returns:
        (numpy.ndarray): Vector of features for the current node
    """

    # Get paper of node.
    name = network.node[node]['name']

    # Get target value.
    target = re.findall('[a-zA-Z]+', name)[0]

    # Compute degree, mean degree of neighbors, number of triangles the
    # node forms with neighbors.
    degree = network.degree[node]
    neighbors = [n for n in network.neighbors(node)]
    neighbors_names = [re.findall('[a-zA-Z]+', network.node[neigh]['name'])[0] for neigh in
        neighbors]

    # Compute bag-of-words features (number of neighbors with each label).
    bow_feature = np.zeros(len(bow), dtype=int)
    for idx, w in enumerate(bow):
        bow_feature[idx] = neighbors_names.count(w)


    # Compute mean degree of neighbors.
    mean_degree_neigh = np.mean([network.degree[neigh] for neigh in neighbors])

    # Compute maximum degree of neighbors.
    max_degree_neigh = np.max([network.degree[neigh] for neigh in neighbors])

    # Compute minimum degree of neighbors.
    min_degree_neigh = np.min([network.degree[neigh] for neigh in neighbors])

    # Compute standard deviation of degree of neighbors.
    std_degree_neigh = np.std([network.degree[neigh] for neigh in neighbors])

    # Compute number of neighbors with same target.
    num_neighbors_same_target = 0
    for neigh in neighbors:
        if re.findall('[a-zA-Z]+', network.node[neigh]['name'])[0] == target:
            num_neighbors_same_target += 1

    # Compute number of triangles including current nodes.
    num_triangles_this = nx.triangles(network, node)

```

```
    # Compute mean number of triagles including one of the neighbors.
    triangles_neighbors = [nx.triangles(network, neigh) for neigh in neighbors]

    # Compute mean number of triangles of neighbors.
    mean_triangles_neigh = np.mean(triangles_neighbors)

    # Compute maximum number of triangles of neighbors.
    max_triangles_neigh = np.max(triangles_neighbors)

    # Compute minimum number of triangles of neighbors.
    min_triangles_neigh = np.min(triangles_neighbors)

    # Construct features vectors.
    feature_vec = np.append(np.array([degree,
                              mean_degree_neigh,
                              max_degree_neigh,
                              min_degree_neigh,
                              std_degree_neigh,
                              num_neighbors_same_target,
                              num_triangles_this,
                              mean_triangles_neigh,
                              max_triangles_neigh,
                              min_triangles_neigh]), bow_feature)

    # Return features vector and target variable.
    return feature_vec, target


def get_features(network, node_idxs):
    """
    Get features for specified node indices.

    Args:
        network (object): The network containing the nodes
        node_idxs (list): List of node indices for which to compute
        features

    Returns:
        (tuple): numpy array containing features and numpy array
        containing the target variable values.
    """

    # Define and initialize data and target variables.
    data = None
    target = []

    # Get label encoder and "bag-of-words".
    le, bow = get_label_encoder_and_bow(network)

    # Go over specified nodes and compute features.
    for idx, node in enumerate(node_idxs):
        print('done {0}/{1}'.format(idx, len(node_idxs)))
        feature_vec_nxt, target_nxt = get_features_node(node, network, bow)
        target.append(target_nxt)
        if data is None:
            data = feature_vec_nxt
        else:
            data = np.vstack((data, feature_vec_nxt))

    # Perform label encoding for target variable.
    target = le.transform(target)

    # Return data and target arrays.
    return data, target


def get_label_encoder_and_bow(network):
    """
```

```
    Get label encoder for target variables and "bag-of-words".

    Args:
        (network): network from which to take the labels.

    Returns:
        (tuple): Fitted LabelEncoder instance and "bag-of-words" list sorted in
        alphabetical order
    """

    # Get labels found in network.
    names = nx.get_node_attributes(network, 'name').values()

    # Get unique labels as "bag-of-words".
    bow = list(map(lambda x: re.findall('[a-zA-Z]+', x)[0], names))

    # Fit label-encoder on bag-of-words.
    le = preprocessing.LabelEncoder().fit(bow)

    # Return fitted label encoder and sorted "bag-of-words".
    return le, sorted(list(set(bow)))


def majority_neigh(network, node_idxs):
    """
    Get results for baseline classifier that predicts the label to be the most
    common label among the neighbors.

    Args:
        network (object): The network containing the nodes
        node_idxs (list): List of node indices for which to make the predictions

    Returns:
        (list): List of label predictions for nodes in node_idxs list
    """

    # Get label encoder.
    le, _ = get_label_encoder_and_bow(network)

    # Initialize list for storing the results.
    res = []

    # Go over nodes and perform classification.
    for node in node_idxs:
        neighbors = network.neighbors(node)
        neighbor_targets = [re.findall('[a-zA-Z]+', network.node[neigh]['name'])[0] for
            neigh in neighbors]
        res.append(max(set(neighbor_targets), key=neighbor_targets.count))

    # Return classifications.
    return le.transform(res)


def evaluate_model(data_train, target_train, data_test, target_test, clf):
    """
    Evaluate model using specified classifier.

    Args:
        data_train (numpy.ndarray): Training data
        target_train (numpy.ndarray): Training target values
        data_test (numpy.ndarray): Test data
        target_test (numpy.ndarray): Test training variables
        clf (object): Classifier to evaluate

    Returns:
        (str): Classification report
    """

```

```
    # Fit classifier.
    clf.fit(data_train, target_train)

    # Score predictions and create classification report.
    pred = clf.predict(data_test)
    return classification_report(target_test, pred)


def main():
    """
    Split data into training and test sets, construct features and evaluate classifiers.
    """

    from sklearn.ensemble import RandomForestClassifier


    # Parse network.
    network = parse_network.parse_network('../data/aps_2008_2013', create_using=nx.Graph)

    import pdb
    pdb.set_trace()

    le, _ = get_label_encoder_and_bow(network)

    # Get training and test data.
    train_idxs, test_idxs = get_tts(network)
    data_train, target_train = get_features(network, train_idxs)
    data_test, target_test = get_features(network, test_idxs)

    # Initialize random-forest classifier.
    clf_rf = RandomForestClassifier()

    # Compute classification repots and write to file.
    clf_report_rf = evaluate_model(data_train, target_train, data_test, target_test, clf_rf)
    clf_report_maj = classification_report(target_test, majority_neigh(network, test_idxs))

    with open('../results/res_classification.txt', 'w') as f:
        f.write(clf_report_rf + '\n')
        f.write(clf_report_maj)


if __name__ == '__main__':
    main()
```