Machine Learning and Algorithms for Data Mining

Assessment 1:

Computing Communities in Large Networks Using Random Walks

Sebastian Borgeaud — spb61@cam.ac.uk

November 29, 2017

Abstract

In this report, I first report the key ideas and concepts introduced in "Computing Communities in Large Networks Using Random Walks" [1]. Next, I present my Python implementation of the algorithm. Finally, I compare the results produced by my algorithm with those reported in [1] and evaluate my algorithms on multiple real world test networks.

1 Main contributions, Key Concepts and Ideas

The paper presents a clustering method for large undirected graphs based on random walks. The paper is based upon the idea that before a random walk converges to the stationary distribution, the random walker spends more time travelling inside clusters than moving between clusters. In fact, we can use this idea to detected clusters in an undirected connected graph.

1.1 Random walks on graphs

More formally, consider an undirected graph $G = (V, E)$ where V are the vertices and E are the edges of G. Let $n = |V|$ and $m = |E|$. This graph has

an **adjacency matrix** A, given by

$$A_{i,j} = \begin{cases} 1, & \text{if } (i,j) \in E, \\ 0 & \text{otherwise} \end{cases}$$

The paper makes two assumptions about the undirected graphs:

1. Every node in the graph is connected to itself.
2. The graph is connected, i.e. any node can be reached from any other node in the graph.

We can define a random walk on this graph using the **transition matrix** P, given by:

$$P_{i,j} = \frac{A_{i,j}}{d(i)} \tag{1}$$

where $d(i) = \Sigma_j A_{i,j}$ denotes the **degree** of node i. The random walk is a random process such that if at time t the random walker is at node i, it will move with probability $P_{i,j}$ to node j. Hence, since $P_{i,j}$ is equal for all the neighbour states j of state i, the random walker moves to a neighbour (or stays in the current state) with uniform probability.

1.2 Distance between vertices and distance between clusters

For a random walk of length t, the probability of starting at node i and ending at node j is given by $P^t_{i,j}$. If t is large enough to gather some information about the structure of the graph, but not too large as to make $P^t_{i,j}$ converge to the stationary distribution, $P^t_{i,j}$ can be used to define a notion of distance between two nodes of the graph, as it has the following desirable properties:

1. If two vertices i and j are in the same cluster, then $P^t_{i,j}$ should be high.
2. The probability of $P^t_{i,j}$ is influenced by $d(j)$ as the walker is more likely to go to high degree vertices.
3. A random walk starting from a vertex i or j of the same cluster, should end in vertex k with similar probability. That is for every vertex k, $P^t_{i,k}$ and $P^t_{j,k}$ should be similar.

We can now define the distance between two vertices i and j:

Definition 1. We can now define the distance between two vertices i and j:

$$r_{i,j} = \sqrt{\sum_{k=1}^{n} \frac{(P_{i,k}^t - P_{j,k}^t)^2}{d(k)}}$$

Note that $r_{i,j}$ is really an euclidian distance as we have that

$$r_{i,j} = \|D^{-\frac{1}{2}} P_{i,\bullet}^t - D^{-\frac{1}{2}} P_{j,\bullet}^t\|$$

where $D_{i,j} = \delta_{i,j} \cdot d(i)$ is the diagonal matrix containing the degrees of the vertices and $P_{i,\bullet}^t$ is the column vector containing probabilities $(P_{i,k}^t)_{1 \leq k \leq n}$.

Next, we can generalise this notion of distance to clusters.

Definition 2. Let $C_1, C_2 \subset V$ be two clusters in G. The distance r_{C_1,C_2} is defined to be:

$$r_{C_1,C_2} = \sqrt{\sum_{k=1}^{n} \frac{(P_{C_1,k}^t - P_{C_2,k}^t)^2}{d(k)}}$$

where the probability $P_{C,k}^t$ to start a random walk of t steps in cluster C and end the walk in vertex k is defined to be

$$P_{C,k}^t = \frac{1}{|C|} \sum_{i \in C} P_{i,k}^t$$

Again, this is really an euclidean distance as

$$r_{C_1,C_2} = \|D^{-\frac{1}{2}} P_{C_1,\bullet}^t - D^{-\frac{1}{2}} P_{C_2,\bullet}^t\|$$

1.3 Clustering algorithm

The distance introduced earlier can be used in a hierarchical clustering algorithm. First, we initialise the initial partition of the graph into n clusters, one cluster per vertex: $\mathcal{P}_1 = \{\{v\} \mid v \in V\}$. Then the algorithm repeatedly merges clusters until only a single cluster is left, by repeating at each step k:

1. Find the two adjacent clusters C_1 and C_2 that minimise the variation $\Delta\sigma(C_1, C_2)$, explained in the next section.

2. Merge C_1 and C_2 into a new cluster C_3.

3. Update the variations $\Delta\sigma(C_3, C)$ between C_3 and the clusters C that are adjacent to C_3.

4. Create a new partition $\mathcal{P}_{k+1} = (\mathcal{P} \setminus \{C_1, C_2\}) \cup C_3$

1.3.1 Choosing two clusters to merge

In order to reduce the complexity, only adjacent clusters are considered for merging. Furthermore, this enforces the desirable property that every cluster connected is, that is any node in a cluster can be reached from every other node in the cluster without leaving the cluster.

The two clusters are then chosen to be those that, when merged, minimise

$$\sigma_k = \frac{1}{n} \sum_{C \in \mathcal{P}_k} \sum_{i \in C} r_{i,C}^2$$

This is usually a NP-hard problem. However, for our definition of $r_{i,C}$ with $r_{i,C} = r_{\{i\},C}$, we can compute the variation $\Delta\sigma(C_1, C_2)$ that would be induced by merging C_1 with C_2, where

$$\Delta\sigma(C_1, C_2) = \frac{1}{n}\Big(\sum_{i \in C_3} r_{i,C_3}^2 - \sum_{i \in C_1} r_{i,C_1}^2 - \sum_{i \in C_2} r_{i,C_2}^2 \Big)$$

Given the probability vectors $P^t_{C_1,\bullet}$ and $P^t_{C_2,\bullet}$, we can compute $\Delta\sigma(C_1, C_2)$ in time $\mathcal{O}(n)$ as one can show that $\Delta\sigma(C_1, C_2)$ is related to r_{C_1,C_2} by:

$$\Delta\sigma(C_1, C_2) = \frac{1}{n} \frac{|C_1||C_2|}{|C_1| + |C_2|} r_{C_1,C_2}^2 \tag{2}$$

1.3.2 Merging the clusters

This step is straightforward as the new cluster C_3 consists of the nodes of C_1 and C_2:

$$C_3 = C_1 \cup C_2$$

The updated probability vector $P^t_{C_3,\bullet}$ can be computed using $P^t_{C_1,\bullet}$ and $P^t_{C_2,\bullet}$:

$$P^t_{C_3,\bullet} = \frac{C_1 P^t_{C_1,\bullet} + |C_2| P^t_{C_2,\bullet}}{|C_1| + |C_2|} \tag{3}$$

1.3.3 Updating the distances

Next, we need to compute the updated variation $\Delta\sigma(C_3, C)$ for every cluster C adjacent to C_3. There are two cases to consider:

1. If C is adjacent to both C_1 and C_2, then we already have the values of $\Delta\sigma(C_1, C)$ and $\Delta\sigma(C_2, C)$. It is then possible to compute $\Delta\sigma(C_3, C)$ in constant time $\mathcal{O}(1)$ using:

$$\Delta\sigma(C_3, C) = \frac{(|C_1| + |C|)\Delta\sigma(C_1, C) + (|C_2| + |C|)\Delta\sigma(C_2, C) + |C|\Delta\sigma(C_1, C_2)}{|C_1| + |C_1| + |C|} \tag{4}$$

2. If C is adjacent to only one of C_1 and C_2, we have to compute $\Delta\sigma(C_3, C)$ using equation 5, that is

$$\Delta\sigma(C_3, C) = \frac{1}{n}\frac{|C_3||C|}{|C_3| + |C|} r^2_{C_3,C} \tag{5}$$

Note that C has to be adjacent to at least one of C_1 and C_2 as C is now adjacent to C_3 and hence these two cases cover all possible cases.

1.4 Evaluating the partitions

The final step of the algorithm is to choose the partition $\mathcal{P}_k$ that best captures the notion of community for our graph, i.e. the partition that has the "best" clustering of the nodes. The paper presents two methods for this:

1. Choose the partition $\mathcal{P}$ that maximises the modularity Q, defined as

$$Q(\mathcal{P}) = \sum_{C \in \mathcal{P}} e_C - a_C^2$$

where e_C is the fraction of edges inside cluster C and a_C is the fraction of edges bound to cluster C.

2. An alternative is to choose the partition $\mathcal{P}$ associated with the largest value of the increase ratio

$$\eta_k = \frac{\Delta\sigma_k}{\Delta\sigma_{k-1}}$$

where $\Delta\sigma_k = \sigma_{k+1} - \sigma_k$. This relies on the idea that if $\Delta\sigma_k$ is large then we are merging very different clusters, whilst if $\Delta\sigma_{k-1}$ is small, then the clusters at k should be relevant.

2 Implementation

I implemented this algorithm in Python, making use of the `numpy` package to handle the computations over the matrices and vectors.

2.1 Initialisation of P^t

The first step is to compute the probability vectors $P^t_{i,\bullet}$ from the adjacency matrix A. This can be done efficiently with numpy as

$$P^t = P \cdot P^{t-1}$$

starting with $P^0 = I$, where I is the identity matrix. In python, using numpy we have:

```
P_t = np.eye(N)
for i in range(t):
    P_t = np.dot(P, P_t)
```

The vector $P^t_{i,\bullet}$, then simply corresponds to the i$^{\text{th}}$ row of the matrix `P_t`, `P_t[i]`.

2.2 Initialisation of the clusters

The second step is to initialise the clusters. For each cluster C, we need to keep the following information:

- The nodes contained in the cluster
- The probability vector $P^t_{C,\bullet}$
- The neighbouring clusters

Furthermore, to keep the hierarchical clustering algorithm simple, I also store in cluster C the variation $\Delta\sigma(C, C')$ for each cluster C' adjacent to C. This is slightly inefficient as we have duplicate information as $\Delta\sigma(C, C') = \Delta\sigma(C', C)$, but allows for a simpler implementation of the algorithm.

The clusters are therefore stored in a dictionary, `clusters`, with cluster C_i located at `clusters[i]`. Each cluster C_i is itself a dictionary, where

- `clusters[i]['nodes']` contains a Python set of the nodes in C_i.

- `clusters[i]['P_t']` contains a numpy vector representing $P^t_{C_i,\bullet}$.
- `clusters[i]['neighbours']` contains a dictionary mapping every neighbouring cluster C_j of C_i to $\Delta\sigma(C_i, C_j)$, that is

$$\texttt{clusters[i]['neighbours']['j']} = \Delta\sigma(C_i, C_j)$$

2.3 Hierarchical clustering

The hierarchical clustering algorithm is implemented using the two helper functions detailed in the next two sections.

2.3.1 Finding two clusters C_i and C_j to merge

The first helper function takes the dictionary of the current clusters in the partition and returns the ids of the two clusters C_i and C_j to merge in this step together with $\Delta\sigma(C_i, C_j)$.

This step is straightforward using an $\mathcal{O}(M)$ implementation where M is the number of edges in the graph. I simply iterate over all pairs of neighbouring clusters and find the two clusters whose variation is the smallest.

```
def find_to_merge(clusters):
    min_Δσ = None
    to_merge = None
    for i, cluster in clusters.items():
        for j, dist in cluster['neighbours'].items():
            if min_Δσ == None or dist < min_Δσ:
                min_Δσ = dist
                to_merge = (i, j)
    return (to_merge[0], to_merge[1], min_Δσ)
```

2.3.2 Merging two clusters C_i and C_j

The second helper function takes in the ids of two clusters to merge and a fresh identity `new_id` and updates the dictionary `clusters` by merging the two clusters into a new cluster, associating the new cluster with `new_id`.

This helper function needs to create a new cluster `C3` from the two cluster `C1` and `C2`. First, I create the set of nodes in `C3` by computing the union of the set of nodes in `C1` and the set of nodes in `C2`.

Next, I compute the new probability vector $P^t_{\texttt{C3},\bullet}$ from $P^t_{\texttt{C1},\bullet}$ and $P^t_{\texttt{C2},\bullet}$ using equation 3:

$$P^t_{C_3,\bullet} = \frac{C_1 P^t_{C_1,\bullet} + |C_2| P^t_{C_2,\bullet}}{|C_1| + |C_2|}$$

Finally, we update the variations $\Delta\sigma(\texttt{C3}, \texttt{C})$ for every cluster `C` now adjacent to `C3`. To do this we first compute the set of new neighbours of `C3`, `new_neighbours`. This set consists of all clusters `C` that were adjacent to `C1` or adjacent to `C2`. Then we compute $\Delta\sigma(\texttt{C3}, \texttt{C})$ using either equation 4 or equation 5 depending on whether `C` was adjacent to both `C1` and `C2`, or to only one of the two clusters. Note that the dictionary of neighbours for every cluster `C` also has to be updated. The entries for `C1` and `C2` are removed, and a new entry for `C3` is added containing $\Delta\sigma(\texttt{C3}, \texttt{C})$.

The full implementation of this function is given next:

```
def merge_clusters(clusters, i, j, new_id):
    C1 = clusters[i]
    C2 = clusters[j]
    l1 = len(C1['nodes'])
    l2 = len(C2['nodes'])

    C3 = {}
    C3['nodes'] = C1['nodes'].union(C2['nodes'])
    C3['P_t'] = (l1 * C1['P_t'] + l2 * C2['P_t']) / (l1 + l2)

    new_neighbours = set(C1['neighbours'].keys()).union(set(C2[
        'neighbours'].keys()))
    new_neighbours.remove(i)
    new_neighbours.remove(j)

    C3['neighbours'] = {}
    for n in new_neighbours:
        C = clusters[n]
        l3 = len(C['nodes'])
        if n in C1['neighbours'].keys() and n in C2['neighbours
            '].keys():
            x_A = (l1 + l3) * C1['neighbours'][n]
            x_B = (l2 + l3) * C2['neighbours'][n]
            x_C = (l3) * C1['neighbours'][j]
```

```
            new_Δσ = (x_A + x_B - x_C) / (l1 + l2 + l3)
        else:
            new_Δσ = Δσ(l1+l2, C3['P_t'], l3, C['P_t'])

        C3['neighbours'][n] = new_Δσ
        C['neighbours'].pop(i, None)
        C['neighbours'].pop(j, None)
        C['neighbours'][new_id] = new_Δσ

    clusters[new_id] = C3
    clusters.pop(i)
    clusters.pop(j)
```

2.3.3 Hierarchical clustering algorithm

Using the two helper functions as stated above, the main loop of the algorithm becomes very simple, repeatedly finding two clusters to merge and then merging them:

```
new_id = N #next cluster starts with id N

while(len(clusters) > 1):
    # find clusters to merge
    (i,j, min_Δσ) = find_to_merge(clusters)
    # Compute new partion
    merge_clusters(clusters, i, j, new_id)
    # increment id for next cluster
    new_id += 1
```

However, as we need to be able to rebuild the dendogram, to compute the modularity and to compute the increase ratio at each step, we need to keep some additional information at each step k. First, to be able to later reconstruct the dendogram, at each time step I append the tuple `(new_id, i, j, cum_dist)` to a list `build_tree`, representing the fact that clusters with ids `i` and `j` have been merged into a cluster with id `new_id`. The value `cum_dist` represent the cumulative variation that has occurred since up to this merging, i.e. $\texttt{cum\_dist} = \sum_{i=1}^{k} \texttt{min\_}\Delta\sigma_k$.

Next, the partition {clusters[i]['nodes'] | i ∈ clusters} is appended to a list named partitions representing $\mathcal{P}$.

Finally, to be able to compute the increase ration η_k, I keep the minimum variation min_$\Delta\sigma$ obtained at each step in a list $\Delta\sigma$s.

The full version of the main loop is therefore:

```
new_id = N
build_tree = []
Δσs = []
partitions = [create_partition(clusters)]
cum_dist = 0

while(len(clusters) > 1):
    # find clusters to merge
    (i,j, min_Δσ) = find_to_merge(clusters)
    # Compute new partion
    merge_clusters(clusters, i, j, new_id)

    # For rebuilding the dendogram
    build_tree.append((new_id, i, j, cum_dist))
    cum_dist += min_Δσ

    # Keep track of partitions
    partitions.append([C['nodes'] for C in clusters.values()])

    # For evaluation of partitions
    s.append(min_Δσ)

    new_id += 1
```

3 Results & analysis

3.1 Example graph from the paper

First of all, I verify that my implementation of the distance functions and the hierarchical clustering algorithm matches the one given in the paper, by testing it on the simple example network consisting of 16 nodes (see figure

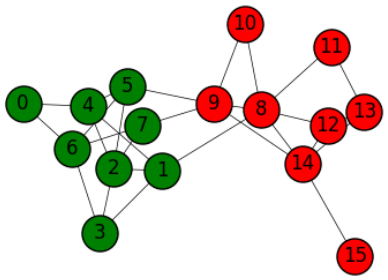

Figure 1: Simple graph with 16 vertices given in [1].

1) given on page 11 of the paper [1]. Running my implementation on this graph gives the dendogram shown in figure 2. As expected, this is the same dendogram as the one given in the paper. Furthermore, the increase ratios and the modularities for each partition $\mathcal{P}_k$, see figure 3, also match the values given in the paper. This suggests that my implementation is indeed correct. [h]

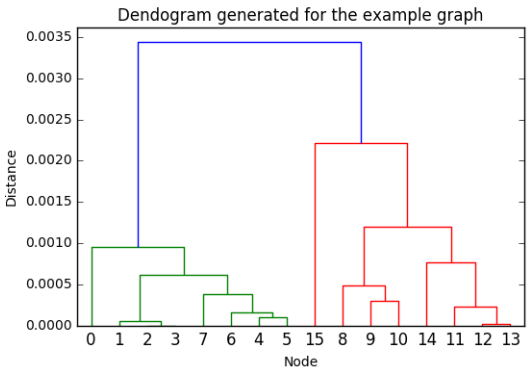

Figure 2: Dendogram resulting from runnign my implementation of the clustering algorithm on the simple toy graph. The resulting dendogram matches the one given in the paper.

Finally, using the partition into two clusters as suggested by both the

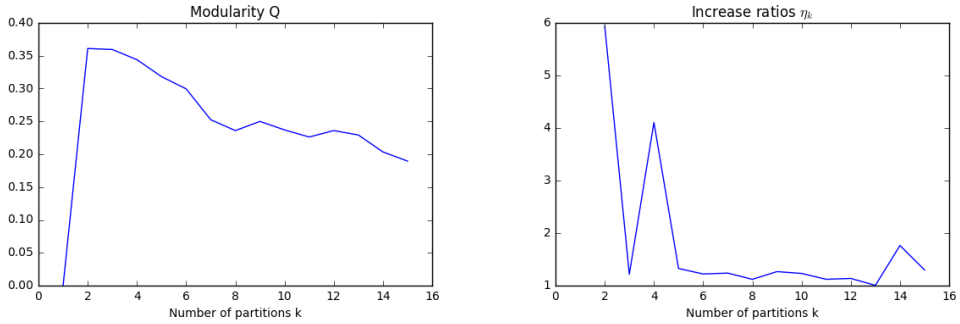

Figure 3: Modularity and increase ratios for the toy example graph given in the paper. The curves match those given in the paper.

modularity and the increase ratios, we can recover the same partition as the one given in the paper, i.e. the partition into the two clusters denoted by green and red nodes as shown in figure 1.

3.2 Test on randomly generated graph

Testing a graph clustering algorithm is a hard task as we need to evaluate the resulting partition. A common way to do this is to use randomly generated graphs for which the partition is already known.

3.3 Zachary's karate club [2]

The next graph I use to test my implementation of the algorithm is the Zachary's karate club network, drawn in figure 4. The graph represents the friendship of the 34 members of a karate club.

As a result of a dispute, the club split into two groups, the members of each group starting their own club. The problem is to recover these two groups using only the information contained in the graph.

Running my implementation of the hierarchical graph clustering algorithm on this graph, results with the dendogram given in figure 5.

Analysing the partitions using modularity (figure 7) suggest to divide the graph into 4 clusters. This partition is shown in figure 6. Although the algorithm was not able to recover that the graph should be partitioned into 2 clusters, the 4 clusters conform very closely with the partitioning into 2 clusters given by the club members: the green and yellow clusters together

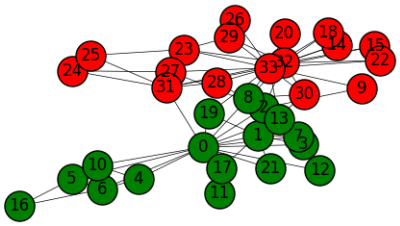

Figure 4: Zachary's karate club network.

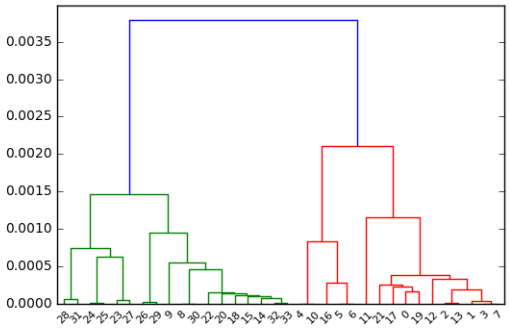

Figure 5: Dendogram resulting from using our algorithm on Zachary's karate club network.

represent one group and the the red and blue clusters together represent the other group with the exception of node 8 which is misclassified. In fact, inspecting the partition into 2 clusters given by the algorithm reveals that all nodes are labelled correctly with the exception of node 8.

3.4 American college football [3]

Finally, I test the algorithm on the American College Football network [3]. The graph consists of 155 nodes, each node being assigned into one of 12 conferences. Similarly as for the Karate Club network, the problem is to recover the 12 conferences by clustering the nodes of the graph. Running my im-

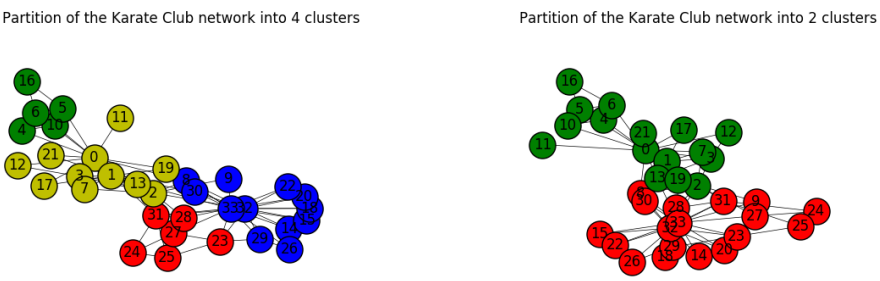

Figure 6: Partitioning of the Karate Club network into 4 and 2 clusters. The 4 cluster partitioning maximises the modularity and conforms very closely with the partitioning into 2 clusters given by the club members: the green and yellow clusters together represent one group and the the red and blue clusters together represent the other group with the exception of node 8 which is misclassified. The partitioning into 2 clusters is almost perfect, except for node 8.

plementation of the algorithm gives the modularities and the increase ratios shown in figure 8. Both the modularity and the increase ratio are maximised at a partitioning into 12 clusters, with a modularity of 0.6. Indeed, using the generated partition for 12 clusters, the algorithm is able to approximately recover the 12 conferences, with only 10 nodes being misclassified. Another way to compare the partitions is to use the correct Rand index $\mathcal{R}'$, which can be computed using

$$\mathcal{R}'(\mathcal{P}_1, \mathcal{P}_2) = \frac{N^2 \sum_{i,j} |C_i^1 \cap C_j^2|^2 - \sum_i |C_i^1|^2 \sum_j |C_j^2|^2}{\frac{1}{2}N^2\big(\sum_i |C_i^1|^2 + \sum_j |C_j^2|^2\big) - \sum_i |C_i^1|^2 \sum_j |C_j^2|^2}$$

where C_i^x is the i$^{\text{th}}$ community of partition $\mathcal{P}_x$.

Comparing the partitioning given by our algorithm with the 12 conferences using the corrected Rand index gives a value of $\mathcal{R}' = 0.906$, which suggests that the algorithm is indeed able to closely recover the 12 conferences.

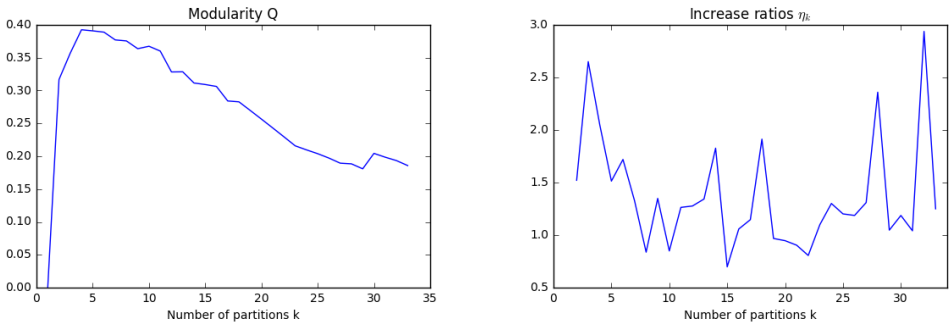

Figure 7: Modularity and increase ratios for the Karate Club network. The modularity suggests a partition into 4 clusters. The increase ration suggests a partition into 3 clusters, as we can disregard clustering the network into more than 10 clusters.

References

[1] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *ISCIS*, volume 3733, pages 284–293, 2005.

[2] Wayne W Zachary. An information flow model for conflict and fission in small groups. *Journal of anthropological research*, 33(4):452–473, 1977.

[3] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

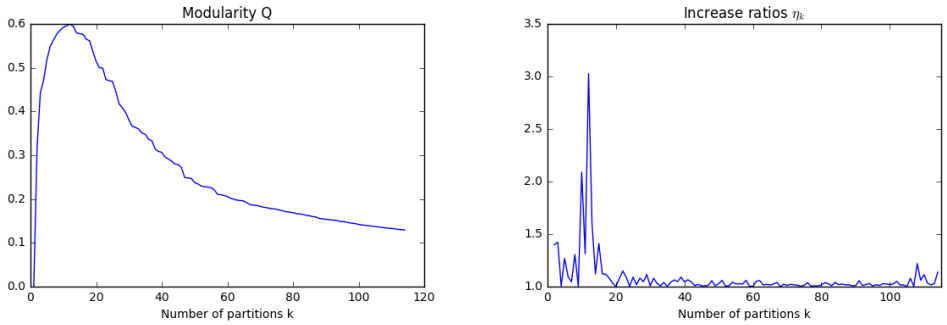

Figure 8: Modularity and increase ratios for the American College Football network. The modularity and increase ratios are both maximised at a partitioning into 12 clusters, which is the correct number of conferences.