Bioinformatics III

Third Assignment

Thibault Schowing (2571837)
Wiebke Schmitt (2543675)

May 2, 2018

Exercise 3.1:

(a) *Given the states of the features, you want to infer if two proteins are likely to physically interact. In practice, log-likelihood ratios are used in binary classification:*

$$log \frac{P(C|S)}{P(\bar{C}|S)}$$

Derive a term that uses observable probabilities such as $P(S_i|C)$ to calculate the loglikelihood ratio from training data. How does the actual classification work?

First we have:

$$P(S_i|C) = \frac{P(C|S_i)P(S_i)}{P(C)}$$

And:

$$P(S_i|\bar{C}) = \frac{P(\bar{C}|S_i)P(S_i)}{P(\bar{C})}$$

Then we develop the desired final output

$$\frac{P(S_i|C)}{P(S_i|\bar{C})} \Longleftrightarrow \frac{P(S_i|C)P(C)}{P(S_i|\bar{C})P(\bar{C})} = \frac{P(C|S_i)P(S_i)}{P(\bar{C}|S_i)P(S_i)} = \frac{P(C|S_i)}{P(\bar{C}|S_i)}$$

$$log \frac{P(C|S)}{P(\bar{C}|S)} = log \prod_i^n \frac{P(S_i|C)P(C)}{P(S_i|\bar{C})P(\bar{C})} = \sum_i^n log \frac{P(S_i|C)P(C)}{P(S_i|\bar{C})P(\bar{C})} = \Lambda(C|S)$$

$$O(C|S) = \Lambda(C|S)O(C)$$

The posterior odd is calculated by the odds of an event ($\frac{p(event)}{1-p(event)}$) multiplied by the likelihood of that event[1].

To do the classification, we must interate through the data and calculate all the priors and likelihood. The prior $P(C)$ is made from an educated guess

[1]Slides V4 - 4

(b) *Shortly discuss: What are the practical advantages of the logarithm and the likelihood ratio within this framework? State two reasons why this particular type of classifier may perform poorly on a real world dataset.*

The logarithm increase is a monotonically increasing function of x hence, for any positive value the maximum value of a function $f(x)$, the maximum of $f(x)$ is equal to the maximum of $log(f(x))$. This simplifies the calculation because we don't need the second derivative. A likelihood function is not concave but the log-likelihood is. Also, as seen in part A, with the log-likelihood we can turn a log of products into a sum of logs. The main inconvenient is that this method assume that all the features are independent and do not take in account the eventual correlations between them.

(c) *Use the file training1.tsv to build a model. This basically means to determine all necessary priors and likelihoods from part (a). The file layout is explained in README.txt. Report $P(C)$ and $P(\bar{C})$ as well as the ten S_i (feature number, variant and log-ratio) with the highest absolute log-likelihood ratios. Examine and comment on the results of the training-phase. Which features seem to be the most helpful?*

Prior probability $P(C) = 0.78$
Prior probability $P(\bar{C}) = 0.22$

Table 1: 10 S_i with the highest absolute log-likelihood ratio

Feature	**Variant**	**log-ratio**
33	0	-3.7214026458194964
11	3	-2.565631943311438
87	1	-2.4686396773241284
53	1	-2.3351082846996056
99	1	-2.3061207478263537
59	1	-2.2779498708596573
80	2	-2.2779498708596573
86	3	-2.2550655770260692
91	3	-2.2173252490432223
97	1	-2.2099451417455995

The log likelihood ratio explains that if you have the variant X you have log(likelohood) more chance to have a complex (C). If negative, it diminishes this chance. To interpret the results above, if for the feature 33 you have variant 0, you have 3.7 more chance that it doesn't make a complex than with variant 1,2 or 3. So here the feature 33 is the most helpful.

Listing 1: bayes.py

```
import math
import copy

# For all features, compute the probability (prior) to have 0, 1, 2 or 3
    depending on the output (0 or 1)
def priors(features, output_indexes):
    priors = {}
    # Start to 1 to match the instructions
    feature_nb = 1
    for feature in features:
        P_Si_Output = {}
        # Values of the feature for a certain output (0 or 1)
        S = [feature[i] for i in output_indexes]
        # for all possible feature values -> [0,1,2,3], set dynamically here
        for value in set(feature):
            # Prob of having this 'value' when output is 0 or 1 (depend on
                output_indexes)
            P_Si_Output[value] = S.count(value) / float(len(S))

        priors[feature_nb] = P_Si_Output
        feature_nb += 1
    return priors


def log_likelihood(Prior_C, Prior_not_C, P_S_C, P_S_notC):
    log_like = [[0.0]*4 for _ in range(len(P_S_C))]

    #For each feature
    # Careful, in P_S_C it's a dict -> start at 1 as "feature 1"
```

```
    # in log_like it's a list of list -> feature 1 == [0]
    for feat in P_S_C:
        for val in [0,1,2,3]:
            p = math.log((P_S_C[feat][val] * Prior_C)/P_S_notC[feat][val] * Prior_not_C)
            log_like[feat-1][val] = p
    return log_like

# Returns the N max likelihood ratios
def getNMaxLikelihoodRatio(likelihoods, N):
    # As we have to loop N times, we'll need to set the max value to zero
    # in order not to pick it more than once.
    likelihoods_copy = copy.deepcopy(likelihoods)
    t = []
    for out in range(N):
        # Will contain (feature number, variant, absolute likelihood ratio)
        info = (0,0,0)
        max = 0

        for feat in range(len(likelihoods_copy)):
            for val in range(len(likelihoods_copy[feat])):
                if abs(likelihoods_copy[feat][val]) > max:
                    max = abs(likelihoods_copy[feat][val])
                    # Max is calculated with the abs, but the real value is stored
                    info = (feat, val, likelihoods_copy[feat][val])
                    likelihoods_copy[feat][val] = 0.0


        t.append(info)
    return t

# Read data file

def readFile(filename):
    lines = []
    with open(filename) as f:
        for line in f:
            line = line.split('\t')
            map(str.strip, line)
            lines.append(line)

    # Convert all the elements in float instead of chars
    lines = [[float(i) for i in line] for line in lines]
    return lines

lines = readFile("data/training1.tsv")

# Number of features
nb_features = len(lines[0]) - 1
print("Nb_features: ", nb_features)

# Get the data by columns: https://stackoverflow.com/questions/44360162/how-to-access-a-column-in-a-list-of-lists-in-python
data_columns = list(zip(*lines))
# Problem, columns are now tuples
data_columns = [list(elem) for elem in data_columns]

# Features only
features = data_columns[1:]

# Output only
outputs = list(data_columns[0])

# Indexes according to outputs (1 or 0, first column)
interaction_indexes = [i for i,x in enumerate(outputs) if x == 1]
```

```
no_interaction_indexes = [i for i,x in enumerate(outputs) if x == 0]

# Prior probabilities

Prior_C = outputs.count(1) / float(len(outputs))
print("Prior probability of having a connection: ", Prior_C)
Prior_not_C = 1 - Prior_C
print("Prior probability of not having a connection: ", Prior_not_C)

# For each feature and possible value, calculate the probability according to
    the output

# P_S_C = Probability of having S (feature) according to output 1
P_S_C = priors(features, interaction_indexes)

# P_S_notC = Probability of having S (feature) according to output 0
P_S_notC = priors(features, no_interaction_indexes)

# Print every probabilities for every feature's values
# print("Features's values's probabilities if connection: \n")
# for p in P_S_C:
#     print("Feature ", p, ": ")
#     for val in P_S_C[p]:
#         print("\tValue: ", val, " prob: ", P_S_C[p][val])
#
# print("Features's values's probabilities if no connection: \n")
# for p in P_S_notC:
#     print("Feature ", p, ": ")
#     for val in P_S_notC[p]:
#         print("\tValue: ", val, " prob: ", P_S_notC[p][val])


# Now we compute the log likelihood for every features and possible output

log_like = log_likelihood(Prior_C, Prior_not_C, P_S_C, P_S_notC)
#print(log_like)

# Get the N (ABSOLUTE) max log-likelihood ratios.
maxLikelihoods = getNMaxLikelihoodRatio(log_like, 10)

# Nice printing
for _ in maxLikelihoods:
    print(_)

# Part D

lines = readFile("data/test1.tsv")
data_columns = list(zip(*lines))
data_columns = [list(elem) for elem in data_columns]
features = data_columns[1:]
outputs = list(data_columns[0])

print("Real test outputs: ")
print(outputs)

predictiontmp = []

for f in range(len(features)):
    tmp_output = 0
    for v in [0,1,2,3]:
        tmp_output += log_like[f][v]
    predictiontmp.append(tmp_output)

prediction = [0 if x < 0 else 1 for x in predictiontmp]
print(prediction)
```

For the train and test 1 and 2 we obtain only 0's as a prediction which is not satisfying according to the real outputs.

Exercise 3.2: Classify real-world network examples

(a) *If one of the nodes has a degree of 1, then $\tilde{C}^{(3)}_{i,j}$ is infinite. What is the maximal finite value that the edge-clustering coefficient can take? For which configuration does this occur? Give an example!*

The edge-clustering coefficient cannot be more than 2. In the layout below we see that the clustering coefficient for the link between node 2 and 3 is equal to $\tilde{C}^{(3)}_{2,3} = \frac{1+1}{min(2-1,6-1)} = \frac{2}{1}$. We see that no matter the degree of node 3, if we connect more node to node 2 the coefficient will decrease even if the number of possible triplet increase.

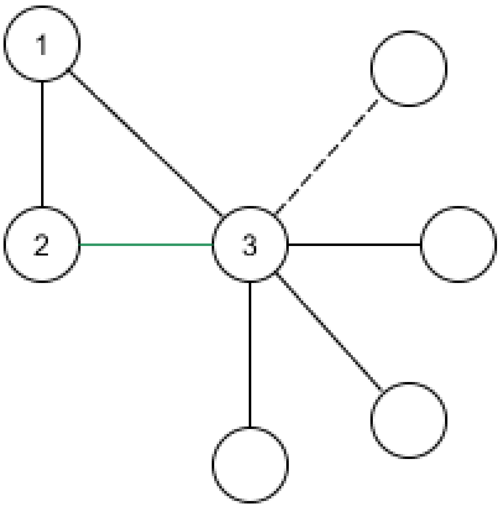

Figure 1:

(b) (1)*Give the links that you deleted from the network in (iii) by printing the names of the two nodes and their current edge-clustering coefficient in the order of their deletion. Of course, add the output to the PDF/sheet that you hand in. Implement this part as a script or class-based, there are no specifications you need to adjust to.*

Table 2: Output - Order of removed links

Tyrion	Sansa	$0.33\bar{3}$
Joffrey	Hound	0.5
Eddard	Robert	0.5
Eddard	Jon	0.5
Joffrey	Jaime	1.0
Hound	Mountain	1.0
Cersei	Tyrion	1.0
Jaime	Cersei	1.0
Catelyn	Baelish	1.0
Sansa	Baelish	1.0
Eddard	Catelyn	1.5
Sansa	Arya	1.5
Eddard	Sansa	1.0
Catelyn	Arya	1.0
Joffrey	Cersei	2.0
Cersei	Robert	1.0
Samwell	Jon	2.0
Joffrey	Robert	∞
Shae	Tyrion	∞
Eddard	Arya	∞
Hound	Arya	∞
Varys	Baelish	∞
Jaime	Tyrion	∞
Cersei	Mountain	∞
Catelyn	Sansa	∞
Samwell	Jeor	∞
Jon	Jeor	∞

Implementation of the network decomposition:

Listing 2: testNetwork.py: decompose the network, output in table 2

```
from GenericNetwork import GenericNetwork
import sys
net = GenericNetwork("GoT.txt")
nb_links = net.nb_links
removed_links_list = [None] * nb_links

print(net)


# for all the links

for current_link in range(nb_links):
    min_coeff = sys.maxsize

    # Iterate through all the links
    for node_a in net.nodes:
        for node_b in net.nodes:
            node1 = net.getNode(node_a)
            node2 = net.getNode(node_b)

            # if the two nodes are different and are linked, check if the
                coeff is smaller or equal than nim_coeff
            if(node1 != node2 and node1.hasLinkTo(node2) and net.
                edgeClusterCoeff(node1, node2) <= min_coeff):
                # We have a temp minimal cluster coeff between node1 and node2
                min_coeff = net.edgeClusterCoeff(node1, node2)
```

```
                    removed_links_list[current_link] = (node1, node2, min_coeff)
                    # Nodes to remove (for now, they will be after the loops)
                    buffer_node1 = node1
                    buffer_node2 = node2

        # Here we have the minimal cluster with (node1, node2, coeff) in removed_links_list

        # To be sure that there is no mistakes
        if(buffer_node1.hasLinkTo(buffer_node2)):
            net.removeLink(buffer_node1, buffer_node2)
            print("Link removed: ", buffer_node1, " ", buffer_node2 , " ", min_coeff)
        else:
            print("Warning: node ", buffer_node1, " and ", buffer_node2, " have no link. ")

        #print(net)
```

Implementation of the other classes. New functions have been added directly in the Network classes.

Listing 3: Node.py

```
# Node class, assignment 1
class Node:
    def __init__(self, identifier):
        """
        Sets node id and initialize empty node list that references its connected nodes
        """
        self.id = identifier
        self.nodelist = []

    def hasLinkTo(self, node):
        """
        Returns True if this node is connected to node asked for,
        False otherwise
        """
        return (node in self.nodelist)

    def addLinkTo(self, node):
        """
        Adds link from this node to parameter ode (only if there is no link connection already),
        does not automatically care for a link from parameter node to this node
        """
        if (~self.hasLinkTo(node)):
            self.nodelist.append(node)

    def degree(self):
        """
        Returns degree of this node
        """
        return len(self.nodelist)

    def __str__(self):
        """
        Returns id of node as string
        """
        return str(self.id)

    def getNodeSet(self):
        return set(self.nodelist)

    """
    Remove node from neighbours list
```

```
    """
    def removeNode(self,node):
        self.nodelist.remove(node)
```

Listing 4: AbstractNetwork.py

```
from Node import Node
import sys

class AbstractNetwork:
    """Abstract network definition, can not be instantiated"""

    def __init__(self, amount_nodes, amount_links):
        """
        Creates empty nodelist and call createNetwork of the extending class
        """
        self.nodes = {}
        self.__createNetwork__(amount_nodes, amount_links)


    def __createNetwork__(self, amount_nodes, amount_links):
        """
        Method overwritten by subclasses, nothing to do here
        """
        raise NotImplementedError

    def appendNode(self, node):
        """
        Appends node to network
        """
        self.nodes[node.id] = node

    def maxDegree(self):
        """
        Returns the maximum degree in this network
        """
        return max([x.degree() for x in self.nodes.values()])

    def size(self):
        """
        Returns network size
        """
        return len(self.nodes)

    def __str__(self):
        '''
        Any string-representation of the network (something simply is enough)
        '''
        # will contain: {identifier : neighbours} -> dict are printed pretty nicely
        self.networkdict = {}
        for n in self.nodes.values():
            # n is a node -> contains identifier and neighbours
            nblist = []
            for elem in n.nodelist:
                nblist.append(elem.id)
            self.networkdict[n.id] = nblist

        niceprint = str(("\n".join("{}\t\t{}".format(k, v) for k, v in self.networkdict.items())) + "\n\n")
        return niceprint

    def getNode(self, identifier):
        """
        Returns node according to key
        """
        if identifier not in self.nodes:
            self.nodes[identifier] = Node(identifier)
```

```python
        return self.nodes[identifier]


    def degreeSum(self):
        '''

        :return: sum of all degrees of the network (a bit unefficient )
        '''
        sum = 0
        for key, node in self.nodes.items():
            sum += node.degree()
        return sum

    def removeLink(self, node1, node2):
        if node1.hasLinkTo(node2) and node2.hasLinkTo(node1):
            node1.removeNode(node2)
            node2.removeNode(node1)
        else:
            print("Cannot remove link between ", str(node1), " and ", str(
                node2))

    def nbTriangle(self, node1, node2):
        if(not node1.hasLinkTo(node2)):
            return 0
        else:
            # Intersection is part of "set"
            intersect = set(node1.nodelist).intersection(node2.nodelist)
            return len(intersect)

    def edgeClusterCoeff(self, node1, node2):
        if(node1.degree() -1 == 0 or node2.degree()-1 == 0):
            return sys.maxsize
        else:
            return (self.nbTriangle(node1, node2) + 1) / float(min(node1.
                degree() - 1, node2.degree() - 1))
```

Listing 5: GenericNetwork.py

```python
from AbstractNetwork import AbstractNetwork
from Node import Node

# from standard library module
from itertools import islice
import sys

class GenericNetwork(AbstractNetwork):


    def __init__(self, filename):
        """
        Create a network from a file
        """

        self.nodes = {}
        self.nb_links = 0

        # We first need to create all Nodes (unique)
        allEntries = []
        pairs = []

        with open(filename) as f:

            # Run through the entire file to make a set of entries
            for line in f:
                line = line.rstrip()
                line_tab = line.split(' ')
                pairs.append(line_tab)
```

```
            allEntries.extend(line_tab)

        allUniqueEntries = set(allEntries)
        for n in allUniqueEntries:
            self.appendNode(Node(n))

        for pair in pairs:
            self.nb_links += 1
            self.getNode(pair[0]).addLinkTo(self.getNode(pair[1]))
            self.getNode(pair[1]).addLinkTo(self.getNode(pair[0]))
```

(2) *Use the links deleted in (1) in reverse order, i.e., the link that was deleted last is now used first to construct the communities.*

Table 3: Inclusion of links, "x" means two subgraphs have been merged.

Link	Merge	Graph
Jon - Jeor		[(Jon - Jeor)]
Samwell - Jeor		[(Jon - Jeor),(Samwell - Jeor)]
Catelyn - Sansa		[(Catelyn - Sansa)]
Cersei - Mountain		[(Cersei - Mountain)]
Jaime - Tyrion		[(Jaime - Tyrion)]
Varys - Baelish		[(Varys - Baelish)]
Hound - Arya		[(Hound - Arya)]
Eddard - Arya		[(Hound - Arya),(Eddard - Arya)]
Shae - Tyrion		[(Jaime - Tyrion),(Shae - Tyrion)]
Joffrey - Robert		[(Joffrey - Robert)]
Samwell - Jon		[(Jon - Jeor),(Samwell - Jeor),(Samwell - Jon)]
Cersei - Robert	x	[(Cersei - Mountain),(Joffrey - Robert), (Cersei - Robert)]
Joffrey - Cersei		[(Cersei - Mountain),(Joffrey - Robert), (Cersei - Robert), (Joffrey - Cersei)]
Catelyn - Arya	x	[(Catelyn - Sansa),(Hound - Arya),(Eddard - Arya), (Catelyn - Arya)]
Eddard - Sansa		[(Catelyn - Sansa),(Hound - Arya),(Eddard - Arya), (Catelyn - Arya),(Eddard - Sansa)]
Sansa - Arya		[(Catelyn - Sansa),(Hound - Arya),(Eddard - Arya), (Catelyn - Arya),(Eddard - Sansa),(Sansa - Arya)]
Eddard - Catelyn		[(Catelyn - Sansa),(Hound - Arya),(Eddard - Arya),(Catelyn - Arya), (Eddard - Sansa),(Sansa - Arya),(Eddard - Catelyn)]
Sansa - Baelish	x	[(Varys - Baelish),(Sansa - Baelish),(Catelyn - Sansa), (Hound - Arya),(Eddard - Arya),(Catelyn - Arya), (Eddard - Sansa),(Sansa - Arya),(Eddard - Catelyn)]
Catelyn - Baelish		[(Varys - Baelish),(Sansa - Baelish),(Catelyn - Sansa), (Hound - Arya),(Eddard - Arya),(Catelyn - Arya), (Eddard - Sansa),(Sansa - Arya),(Eddard - Catelyn), (Catelyn - Baelish)]
Jaime - Cersei	x	[(Jaime - Tyrion),(Shae - Tyrion),(Cersei - Mountain), (Joffrey - Robert), (Cersei - Robert),(Joffrey - Cersei)]
Cersei - Tyrion		[(Jaime - Tyrion),(Shae - Tyrion),(Cersei - Mountain), (Joffrey - Robert), (Cersei - Robert),(Joffrey - Cersei), (Cersei - Tyrion)]

Link	Merge	Graph
Hound - Mountain	x	[(Varys - Baelish),(Sansa - Baelish),(Catelyn - Sansa), (Hound - Arya),(Eddard - Arya),(Catelyn - Arya), (Eddard - Sansa),(Sansa - Arya),(Eddard - Catelyn), (Catelyn - Baelish),(Jaime - Tyrion),(Shae - Tyrion), (Cersei - Mountain),(Joffrey - Robert), (Cersei - Robert), (Joffrey - Cersei),(Cersei - Tyrion)]
Joffrey - Jaime		[(Varys - Baelish),(Sansa - Baelish),(Catelyn - Sansa), (Hound - Arya),(Eddard - Arya),(Catelyn - Arya), (Eddard - Sansa),(Sansa - Arya),(Eddard - Catelyn), (Catelyn - Baelish),(Jaime - Tyrion),(Shae - Tyrion), (Cersei - Mountain),(Joffrey - Robert), (Cersei - Robert), (Joffrey - Cersei),(Cersei - Tyrion),(Joffrey - Jaime)]
Eddard - Jon	x	[(Varys - Baelish),(Sansa - Baelish),(Catelyn - Sansa), (Hound - Arya),(Eddard - Arya),(Catelyn - Arya), (Eddard - Sansa),(Sansa - Arya),(Eddard - Catelyn), (Catelyn - Baelish),(Jaime - Tyrion),(Shae - Tyrion), (Cersei - Mountain),(Joffrey - Robert), (Cersei - Robert), (Joffrey - Cersei),(Cersei - Tyrion),(Joffrey - Jaime), (Jon - Jeor),(Samwell - Jeor),(Samwell - Jon)]
Eddard - Robert		[(Varys - Baelish),(Sansa - Baelish),(Catelyn - Sansa), (Hound - Arya),(Eddard - Arya),(Catelyn - Arya), (Eddard - Sansa),(Sansa - Arya),(Eddard - Catelyn), (Catelyn - Baelish),(Jaime - Tyrion),(Shae - Tyrion), (Cersei - Mountain),(Joffrey - Robert), (Cersei - Robert), (Joffrey - Cersei),(Cersei - Tyrion),(Joffrey - Jaime), (Jon - Jeor),(Samwell - Jeor),(Samwell - Jon), (Eddard - Robert)]
Joffrey - Hound		[(Varys - Baelish),(Sansa - Baelish),(Catelyn - Sansa), (Hound - Arya),(Eddard - Arya),(Catelyn - Arya), (Eddard - Sansa),(Sansa - Arya),(Eddard - Catelyn), (Catelyn - Baelish),(Jaime - Tyrion),(Shae - Tyrion), (Cersei - Mountain),(Joffrey - Robert), (Cersei - Robert), (Joffrey - Cersei),(Cersei - Tyrion),(Joffrey - Jaime), (Jon - Jeor),(Samwell - Jeor),(Samwell - Jon), (Eddard - Robert),(Joffrey - Hound)]
Tyrion - Sansa		[(Varys - Baelish),(Sansa - Baelish),(Catelyn - Sansa), (Hound - Arya),(Eddard - Arya),(Catelyn - Arya), (Eddard - Sansa),(Sansa - Arya),(Eddard - Catelyn), (Catelyn - Baelish),(Jaime - Tyrion),(Shae - Tyrion), (Cersei - Mountain),(Joffrey - Robert), (Cersei - Robert), (Joffrey - Cersei),(Cersei - Tyrion),(Joffrey - Jaime), (Jon - Jeor),(Samwell - Jeor),(Samwell - Jon), (Eddard - Robert),(Joffrey - Hound),(Tyrion - Sansa)]

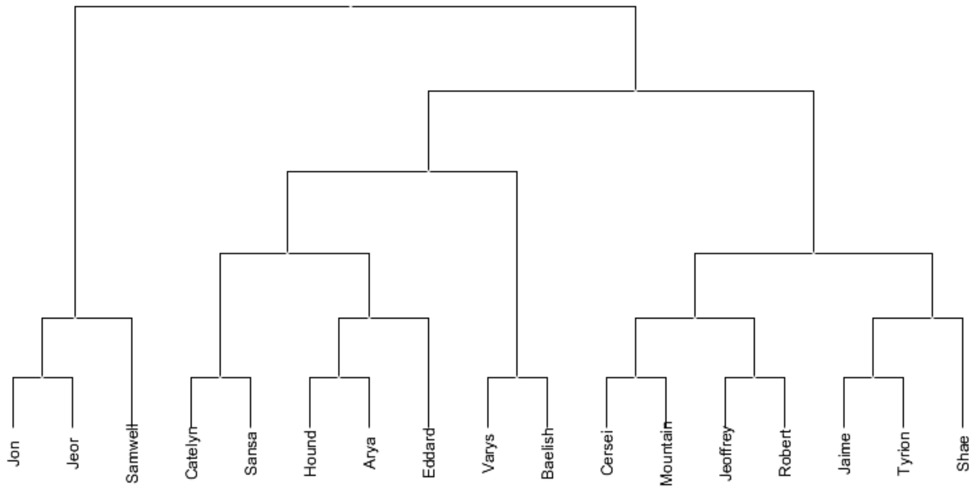

Figure 2: Dendogram

(c) Visualisation of the communities

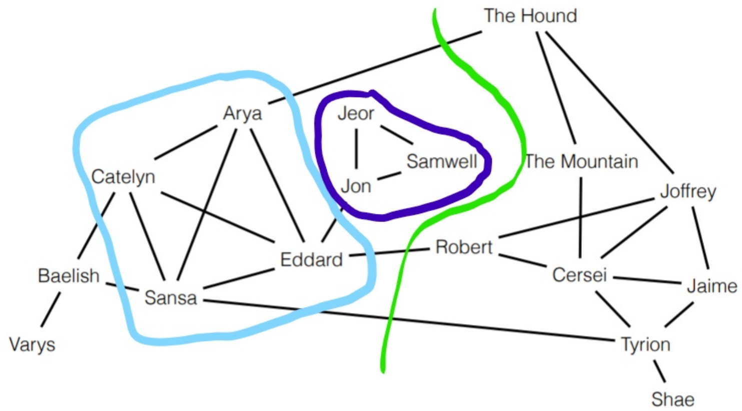

Figure 3: Visualisation of the network

Here we can identify many communities. The two biggest are the ones we can have by separating the network in the middle (In green). All the nodes have more links inside the communities than outside.

Table 4: The two big communities of the network separated by the green line. All the nodes not cited below have $k_{out} = 0$.

Node	k_{in}	k_{out}
Arya	3	1
Eddard	3	1
Sansa	3	1
The Hound	2	1
Robert	2	1
Tyrion	3	1

Here is two disjointed examples:

Table 5: Stark community in light blue. Each member have a k_{in} bigger than the k_{out} so the strong criterion applies.

Node	k_{in}	k_{out}
Catelyn	3	1
Arya	3	1
Eddard	3	2
Sansa	3	2

Table 6: Jon's community in dark blue, the strong criterion applies too.

Node	k_{in}	k_{out}
Jon	2	1
Jeor	2	0
Samwell	2	0