Bioinformatics III

First Assignment

Max Jakob (2549155)
Carolin Mayer (2552320)

April 27, 2018

Exercise 1.1: The random network

(a) In Listing 1, our implementation of the required `Node` class is shown.

Listing 1: Source code of Node.py

```python
class Node:
    def __init__(self, identifier):
        """
        Sets node id and initialize empty node list that references its connected nodes
        """
        self.id = identifier
        self.nodes = {}

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

    def addLinkTo(self, node):
        """
        Adds link from this node to parameter node (only if there is no link connection already
        does not automatically care for a link from parameter node to this node
        """
        if not (self.hasLinkTo(node)):
            self.nodes[node.id] = node

    def degree(self):
        """
        Returns degree of this node
        """
        return self.nodes.__len__()

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

(b) In Listing 2, our implementation of the required `AbstractNetwork` class is shown.

Listing 2: Source code of AbstractNetwork.py

```python
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.mdegree = 0
        self.__createNetwork__(amount_nodes, amount_links)
        for node in self.nodes:
            degree = self.nodes[node].nodes.__len__()
            if(degree>self.mdegree):
                self.mdegree = degree


    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
        if(self.mdegree < node.degree()):
            self.mdegree = node.degree()

    def maxDegree(self):
        """
        Returns the maximum degree in this network
        """
        return int(self.mdegree)

    def size(self):
        """
        Returns network size (here: number of nodes)
        """
        return self.nodes.__len__()

    def __str__(self):
        '''
        Any string-representation of the network (something simply is enough)
        '''
        s = " "
        '''
        for node in self.nodes:
            s = s + "{ " + str(node) + " }"
            s = s + " -> { "
            for k in self.nodes[node].nodes:
                s = s + str(k) + " "
            s = s + "}\n"
        '''
        return s

    def getNode(self, identifier):
        """
        Returns node according to key
        """
        return self.nodes[identifier]
```

(c) In Listing 3, our implementation of the required `RandomNetwork` class is shown.

Listing 3: Source code of RandomNetwork.py

```
from AbstractNetwork import AbstractNetwork
from Node import Node
import random # you will need it :-)

class RandomNetwork(AbstractNetwork):
    """Random network implementation of AbstractNetwork"""
```

```
    def __createNetwork__(self, amount_nodes, amount_links): # remaining methods are taken from
        """
        Creates a random network
        1. Build a list of n nodes
        2. For i=#links steps, add a connection between for two randomly chosen nodes that are
        """
        random.seed()

        for i in range(0, amount_nodes):
            AbstractNetwork.appendNode(self, node=Node(i))

        size = AbstractNetwork.size(self)-1
        for i in range(0, amount_links):
            k1 = random.randint(0, size)
            k2 = random.randint(0, size)
            n1 = AbstractNetwork.getNode(self, k1)
            n2 = AbstractNetwork.getNode(self, k2)
            n1.addLinkTo(n2)
            n2.addLinkTo(n1)
```

Exercise 1.2: Degree distribution of random networks

(a) In Listing 4, our implementation of the required `DegreeDistribution` class is shown.

Listing 4: Source code of RandomNetwork.py

```
import numpy

class DegreeDistribution:
    """Calculates a degree distribution for a network"""
    def __init__(self, network):
        """
        Inits DegreeDistribution with a network and calculate its distribution
        """
        size = network.maxDegree() +1
        self.hist = [0] * size
        for node in network.nodes:
            i = network.nodes[node].nodes.__len__()
            self.hist[i] = self.hist[i] + 1

    def getNormalizedDistribution(self):
        '''
        Returns the computed normalized distribution
        '''
        num = numpy.sum(self.hist)
        return [i / num for i in self.hist]
```

(b) In Listing 5, our implementation of the required `Tools` class is shown.

Listing 5: Source code of RandomNetwork.py

```
import matplotlib.pyplot as plt
import numpy as np
import math

def plotDistributionComparison(histograms, legend, title):
    '''
    Plots a list of histograms with matching list of descriptions as the legend
    '''
    # adjust size of elements in histogram
    maxlen = 0
    for h in histograms:
        if len(h) > maxlen:
            maxlen = len(h)

    for h in histograms:
```

```
        while len(h) != maxlen:
            h.append(0.0)

    # plots histograms
    for h in histograms:
        plt.plot(range(len(h)), h, marker = 'x')

    # remember: never forget labels! :-)
    plt.xlabel('Degree of k')
    plt.ylabel('Density')

    # you don't have to do something here
    plt.legend(legend)
    plt.title(title)
    plt.tight_layout()# might throw a warning, no problem
    plt.show()

def getPoissonDistributionHistogram(num_nodes, num_links, k):
    '''
    Generates a Poisson distribution histogram up to k
    '''
    lam = 2 * num_links / num_nodes
    res = [0]*k
    for i in range(0, k):
        res[i] = poisson(lam, i)
    return res

def poisson(lam, k):

    if(k == 0):
        return math.exp(-lam)
    else:
        return (lam/k * poisson(lam,k-1))
```

All missing entries in the histogram where extended with 0.0. This has been done for the purpose of shrinking all histograms to the same length, thats to say to the same number ob buckets they contain.

(c) In Figure 1 and Figure 2, the results of the `createAndPlotNetworks.py` script that plots the Poisson distribution (p) together with the degree distributions (r) with density against degree of k, is shown for each generated random network.
In Figure 1 one can see that all function are steadily rising (except r:50/100) until they reach their peaks always close to k = 4 and density around 0.2. After that, the functions are falling steadily until they slowly reach the density zero. One can see that the Poisson distribution of all networks shows a bell curve, whereas the degree distribution approaches to a bell curve as the number of edges and vertical increases. Hence, one can say that as higher the number of verticals and edges as closer the approximation to the bell curve.
In Figure 2, the number of nodes is constantly 20000 for each network, whereas the number of edges increases. Hence, the peak of the function are shifted at the x-axes. Furthermore, one can see that the peaks approach to 0.2 as the number of edges increase. In this plot, r and p are equal, which demonstrates the approximation of r to p as the network grows.
Hence, as conclusion we can say that as the number of nodes and edges increase, the degree distribution of the random network approach to the Poisson distribution, which one can see in the approaching bell curve of it.

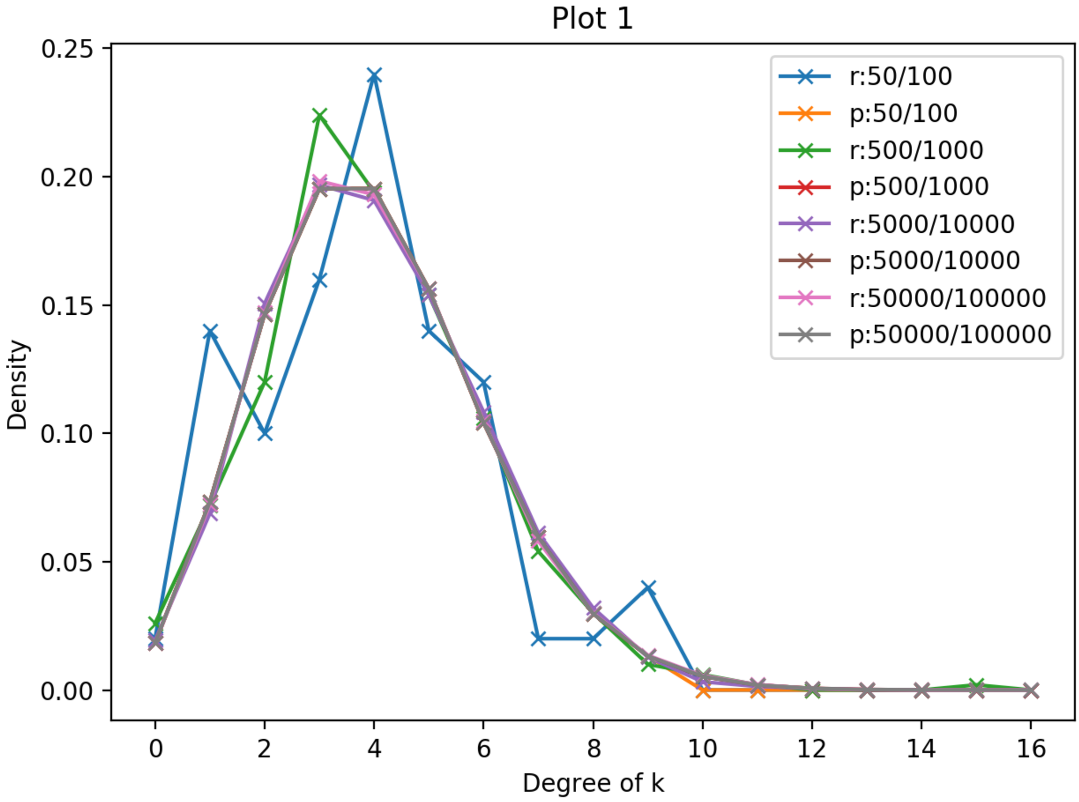

Figure 1: Plot 1

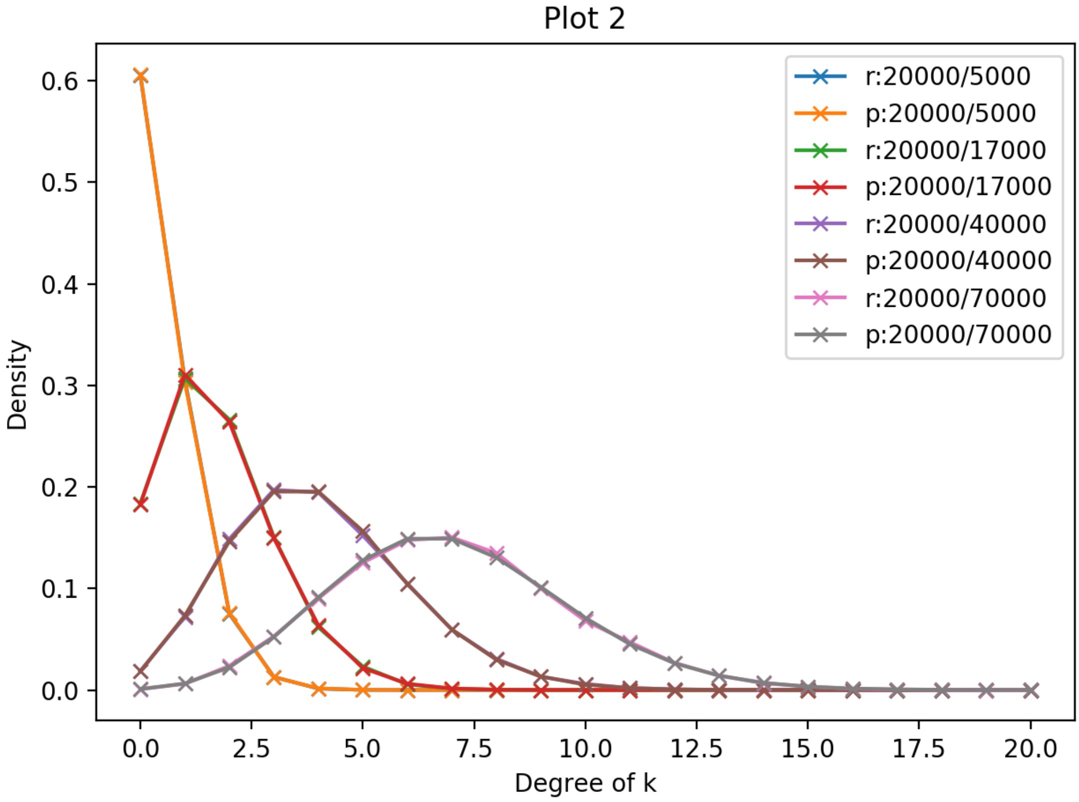

Figure 2: Plot 2