Bioinformatics III

Eigth Assignment

Alexander Flohr (2549738)
Andrea Kupitz (2550260)

June 18, 2018

Exercise 8.1: Data Preprocessing

Listing 1: Listing of source code

(a)

```
import csv
import numpy
import math
from scipy import stats

class DataMatrix:
    def __init__(self, file_path):
        """
        :param file_path: path to the input matrix file
        """
        self.file_path = file_path
        # TODO: define and initialise the class fields you need for your implementation
        # read the matrix in the input file, remove rows with empty values and merge duplicate
        self.cols, self.rows = self.read_data()

    def read_data(self):
        """
        Reads data from a given matrix file, where the first line gives the names of the columns
        gives the names of the rows. Removes rows with empty or non-numerical values and merges
        name into one.
        """
        with open(self.file_path) as tsvfile:
            reader = csv.reader(tsvfile, delimiter='\t')
            cols = next(reader)
            fields = len(cols)
            rows = dict()
            for row in reader:
                if len(row) == fields:
                    floats = []
                    nan = False
                    for i in range(1, fields):
                        try:
                            f = float(row[i])
                            if math.isnan(f):
                                nan = True
                        except ValueError:
                            nan = True
                        floats.append(f)
                    if row[0] not in rows and not nan:
                        rows[row[0]] = [floats]
                    elif not nan:
                        rows[row[0]].append(floats)
            for k, v in rows.items():
                if len(v) > 1:
                    value = []
                    for i in range(0, len(v[0])):
                        mean = []
```

```
                        for j in range(0, len(v)):
                            mean.append(v[j][i])
                        value.append(numpy.mean(mean))
                    rows[k] = [value]
        return cols, rows

    def get_rows(self):
        """
        :return: dictionary with keys = row names, values = list of row values
        """
        rows = dict()
        for k, v in self.rows.items():
            rows[k] = v[0]
        return rows

    def get_columns(self):
        """
        :return: dictionary with keys = column names, values = list of column values
        """
        dic = dict()
        for k, v in self.rows.items():
            for i, col in enumerate(self.cols):
                if i != 0:
                    if col not in dic:
                        dic[col] = []
                    else:
                        dic[col].append(v[0][i-1])
        return dic

    def not_normal_distributed(self, alpha, rows):
        """
        Uses the Shapiro-Wilk test to compute all rows (or columns) that are not normally distr
        :param alpha: significance threshold
        :param rows: True if the Shapiro-Wilk p-values should be computed for the rows, False
        :return: dictionary with keys = row/columns names, values = Shapiro-Wilk p-value
        """
        dic = dict()
        values = self.get_rows() if rows else self.get_columns()
        for k, v in values.items():
            W, p = stats.shapiro(v)
            if p < alpha:
                dic[k] = p
        return dic

    def to_tsv(self, file_path):
        """
        Writes the processed matrix into a tab-separated file, with the same column order as th
        the rows in lexicographical order.
        :param file_path: path to the output file
        """
        with open(file_path, 'w') as f:
            values = [value for (key, value) in sorted(self.get_rows().items())]
            keys = sorted(self.get_rows().keys())
            for i in range(0, len(values)):
                s = keys[i] + '\t'
                for j in values[i]:
                    s += str(j) + '\t'
                s += '\n'
                f.write(s)
```

Listing 1 shows the source code.

Listing 2: Listing of source code

(b)
```
from data_matrix import DataMatrix
from network import CorrelationNetwork
from correlation import CorrelationMatrix
from cluster import CorrelationClustering
```

```
5
def exercise_1():
    DM1 = DataMatrix('expression.tsv')
    DM1.to_tsv('Flohr_Kupitz_expression.tsv')
    print(len(DM1.not_normal_distributed(0.05, True)))
10  DM2 = DataMatrix('methylation.tsv')
    DM2.to_tsv('Flohr_Kupitz_methylation.tsv')
    print(len(DM2.not_normal_distributed(0.05, True)))

def exercise_3(threshold=0.75):
15  CN1 = CorrelationNetwork(CorrelationMatrix(DataMatrix('expression.tsv'), 'Pearson', True),
    CN1.to_sif('Flohr_Kupitz_expression_network_pearson.sif')
    CN1 = CorrelationNetwork(CorrelationMatrix(DataMatrix('expression.tsv'), 'Spearman', True),
    CN1.to_sif('Flohr_Kupitz_expression_network_spearman.sif')
    CN1 = CorrelationNetwork(CorrelationMatrix(DataMatrix('expression.tsv'), 'Kendall', True),
20  CN1.to_sif('Flohr_Kupitz_expression_network_kendall.sif')
    CN1 = CorrelationNetwork(CorrelationMatrix(DataMatrix('methylation.tsv'), 'Pearson', True),
    CN1.to_sif('Flohr_Kupitz_methylation_network_pearson.sif')
    CN1 = CorrelationNetwork(CorrelationMatrix(DataMatrix('methylation.tsv'), 'Spearman', True)
    CN1.to_sif('Flohr_Kupitz_methylation_network_spearman.sif')
25  CN1 = CorrelationNetwork(CorrelationMatrix(DataMatrix('methylation.tsv'), 'Kendall', True),
    CN1.to_sif('Flohr_Kupitz_methylation_network_kendall.sif')

def exercise_4():
    # TODO
30  pass

# only execute the following if this module is the entry point of the program, not when it is i
if __name__ == '__main__':
    exercise_1()
35  exercise_3()
    exercise_4()
```

Listing 2 shows the source code.
The number of genes and samples whose data does not follow a normal distribution for the expression data is 74 and for the methylation data is 66.

Exercise 8.2: Correlation Measures

All subtasks are implemented in `correlation.py`, see listing 3

Listing 3: Listing of source code

```
from itertools import combinations
from copy import copy
import math

def computeMean(vec):
    """
    Compute the mean for a list
    """
    temp = 0
    for i in vec:
        temp += i
    return float(temp)/float(len(vec))

def initializeIndices(x_s):
    """
    Create list of indices for values, where equal values
    in x_s get equal averaged idices
    """
    x_r = list(range(0, len(x_s)))
    # add an terminate-sign to end the algorithm
    x_r.append(-1)
    i = 0
    for j in range(1, len(x_s)):
        if x_s[i] != x_s[j]:
            index = computeMean(x_r[i:j])
            while i < j:
                x_r[i] = index
                i += 1
    # return all but the last termination-sign
    return x_r[0:len(x_s)]

def bubble(x_s):
    """
    Recursive bubble sorting with "rebubbling"
    the indices with respect to multi-occurring
    values
    """
    for i in range(1,len(x_s)):
        if x_s[i-1] < x_s[i]:
            # swap list elements
            temp = x_s[i-1]
            x_s[i-1] = x_s[i]
            x_s[i] = temp
            # next bubble
            x_r = bubble(x_s)
            # reverse bubbleing of the indices
            temp = x_r[i-1]
            x_r[i-1] = x_r[i]
            x_r[i] = temp
            return x_r
    # called only in the deepest recursive step
    # here the indices get initialized
    return initializeIndices(x_s)

def rank(x):
    """
    :param x: a list of values
    :return: ranking of the input list
    """
    # sort the values of x by bubble sort
    # original indices are kept in a second list
```

```
    return bubble(copy(x))

def pearson_correlation(x, y):
    """
    :param x: a list of values
    :param y: a list of values
    :return: Pearson correlation coefficient of X and Y
    """
    mean_x = computeMean(x)
    mean_y = computeMean(y)

    # covariance
    upper = 0
    for i in range(0,len(x)):
        upper += (x[i] - mean_x) * (y[i] - mean_y)
    lower_left = 0

    # standard deviation of x
    for i in range(0,len(x)):
        lower_left += (x[i] - mean_x) * (x[i] - mean_x)
    lower_left = math.sqrt(lower_left)

    # standard deviation of y
    lower_right = 0
    for i in range(0,len(y)):
        lower_right += (y[i] - mean_y) * (y[i] - mean_y)
    lower_right = math.sqrt(lower_right)

    # pearson-correlation formula
    return (upper/(lower_left*lower_right))

def spearman_correlation(x, y):
    """
    :param x: a list of values
    :param y: a list of values
    :return: Spearman correlation coefficient of X and Y
    """
    rank_x = rank(x)
    rank_y = rank(y)
    return pearson_correlation(rank_x, rank_y)


def kendall_correlation(x, y):
    """
    :param x: a list of values
    :param y: a list of values
    :return: Kendall-B correlation coefficient of X and Y
    """
    # processing part (1)
    rank_x = rank(x)
    rank_y = rank(y)

    # processing part (2)
    pairing = []
    for i in range(0, len(x)):
        pairing.append((rank_x[i], rank_y[i]))

    # processing part (3)
    concordant = 0
    discordant = 0
    tied_x = 0
    tied_y = 0
    for i in range(0,len(pairing)):
        for j in range(0,len(pairing)):
            if i != j:
                (a,b) = pairing[i]
                (c,d) = pairing[j]
```

```
                if (a < b and c < d) or (a > b and c > d):
                    concordant += 1
                elif (a < b and c > d) or (a > b and c < d):
                    discordant += 1
                elif (a == b and c != d):
                    tied_x += 1
                elif (a != b and c == d):
                    tied_y += 1

        # processing part (4)
        score1 = float(concordant - discordant)
        score2 = float(concordant + discordant + tied_x)
        score3 = float(concordant + discordant + tied_y)
        if score1*score2*score3 == 0:
            return 0
        else:
            return score1/math.sqrt(score2 * score3)

class CorrelationMatrix(dict):
    """
    This class behaves like a dictionary, where the correlation between two elements 1 and 2 is acc
    cor_matrix[(element_1, element_2)] or cor_matrix[(element_2, element_1)] since the matrix is sy
    It also stores the row (or column) names of the input DataMatrix.
    """
    def __init__(self, data_matrix, method, rows):
        """
        :param data_matrix: a DataMatrix (see data_matrix.py)
        :param method: string specifying the correlation method, must be 'Pearson', 'Spearman' or '
        :param rows: True if the correlation matrix should be constructed for the rows, False if fo
        """
        # initialise the dictionary
        super().__init__(self)

        # if rows = True, then compute the correlation matrix for the row data
        if rows:
            data = data_matrix.get_rows()
        # if rows = False, then compute the correlation matrix for the column data
        else:
            data = data_matrix.get_columns()

        # sorted list of row names (or column names) in the input data matrix
        self.names = list(sorted(data.keys()))

        # compute the correlation between all pairs of rows (or columns)
        for name_1, name_2 in combinations(data.keys(), 2):
            # use the specified correlation method
            if method == 'Pearson':
                correlation = pearson_correlation(data[name_1], data[name_2])
            elif method == 'Spearman':
                correlation = spearman_correlation(data[name_1], data[name_2])
            elif method == 'Kendall':
                correlation = kendall_correlation(data[name_1], data[name_2])
            else:
                raise ValueError('The correlation method not supported must be either Pearson, Spear

            # add the correlation symmetrically
            self[(name_1, name_2)] = correlation
            self[(name_2, name_1)] = correlation
```

Exercise 8.3: Gene Co–Expression Networks

Listing 4: Listing of source code

(a)

```
from Node import Node

class CorrelationNetwork:
    def __init__(self, correlation_matrix, threshold):
        """
        Constructs a co-expression network from a correlation matrix by adding edges between no
        correlation bigger than the given threshold.
        :param correlation_matrix: a CorrelationMatrix (see correlation.py)
        :param threshold: a float between 0 and 1
        """
        self.cor = correlation_matrix
        self.nodes = {}
        for entry, cor in correlation_matrix.items():
            node1 = Node(entry[0])
            node2 = Node(entry[1])
            self.appendNode(node1)
            self.appendNode(node2)
            if abs(cor) > threshold:
                node1.addLinkTo(node2)
                node2.addLinkTo(node1)

    def appendNode(self, node):
        """
        Appends node to network
        """
        if node.id not in self.nodes:
            self.nodes[node.id] = node

    def to_sif(self, file_path):
        """
        Write the network into a simple interaction file (SIF).
        Column 0: label of the source node
        Column 1: interaction type
        Columns 2+: label of target node(s)
        :param file_path: path to the output file
        """
        with open(file_path, 'w') as f:
            values = [value for (key, value) in sorted(self.nodes.items())]
            keys = sorted(self.nodes.keys())
            written = []
            s = ''
            for position, value in enumerate(keys):
                for node2id in values[position].nodelist:
                    if node2id not in written: #and node2id != value:
                        s = str(value) + '\t'
                        s += str(round(self.cor[(value, node2id.id)], 2)) + '\t'
                        s += str(node2id) + '\n'
                    written.append(node2id)
                f.write(s)
```

Listing 4 shows the source code.

(b) Listing 2 shows the source code.

(c)

Exercise 8.4: Hierarchical Clustering

(a)

(b)

(c)