Exercise 4

Applications of Data Analysis

Marco Willgren 502606 mahewi@utu.fi

Jarno Vuorenmaa 503618 jkivuo@utu.fi

1 DATA PREPROCESSING AND CREATING DISTANCE MATRIX

Data was given in three csv-files. We read the input data in *x*, output data in *y* and coordinates in *z*. Also the input data is standardized and stored into *stdX*.

```
basepath = os.path.dirname(__file__)
inputpath = os.path.abspath(os.path.join(basepath, "Data4/INPUT.csv"))
outputpath = os.path.abspath(os.path.join(basepath, "Data4/OUTPUT.csv"))
coordinatespath = os.path.abspath(os.path.join(basepath,
"Data4/COORDINATES.csv"))

x = np.genfromtxt(inputpath, delimiter=',')
y = np.genfromtxt(outputpath, delimiter=',')
z = np.genfromtxt(coordinatespath, delimiter=',')

xArr = np.asarray(x)
stdX = (xArr - xArr.mean()) / xArr.std()
```

Distances between every pair of points is stored into a distance matrix. M_{ij} is the euclidean distance between *i*th and *j*th point in data. If i=j, the M_{ij} is set to -1.

```
def calculateDistanceMatrix():
    distanceMatrix = []
    for i in range(len(z)):
        xAxis = []
        for j in range(len(z)):
            if i == j:
                xAxis.append(-1.0)
            else:
                xAxis.append(ssd.euclidean(z[i], z[j]))
        distanceMatrix.append(xAxis)

    return distanceMatrix
```

2 FINDING OUT VALUES IN DEADZONE

The Method *calculateDeadZone* finds the 10 nearest points (where the distance is larger or equal to 0.0) for every data point and sets the distance to those points as -1. Thus our implementation of leave-on-out cross-validation ignores those points (see below).

```
def calculateDeadZone(matrix):
    for i in range(len(matrix)):
        xAxis = matrix[i]
        for _ in range(10):
            minIndex = xAxis.index(min(filter(lambda x:x>=0.0, xAxis)))
            xAxis[minIndex] = -1.0

    return matrix
```

3 CALCULATING 5-NEAREST-NEIGHBOR AND PREDICTING LABEL

The distance between two features is calculated only if the M_{ij} is larger or equals with 0. This limitation is done because we need to leave the test instance out of calculation and also we need to ignore features that are in the deadzone. The distance of a feature with itself and the distances to the features in the deadzone are set to -1. The method returns k-nearest neighbors. The value of k is given as an argument and in this case it is 5.

```
def inferNeighbors(trainSet, testInstance, labels, k, distRow):
    distances = []
    for x in range(len(trainSet)):
        if distRow[x] >= 0.0:
             distances.append((ssd.euclidean(trainSet[x], testInstance),
             labels[x]))

    distances.sort(key=operator.itemgetter(0))
    return distances[0:k]
```

The predicted value for the test instance is the mean value of its neighbors' classes.

```
def chooseMajorityLabel(neighbors, k):
    predictedOutcome = []
    sumOfMod = 0.0
    for i in range(len(neighbors)):
        sumOfMod = sumOfMod + neighbors[i][1]
    predictedOutcome.append(sumOfMod/k)

    return predictedOutcome
```

4 LEAVE-ONE-OUT CROSS-VALIDATION

Leave-one-out cross-validation – method gets the number of nearest neighbors to consider and a distance matrix as its arguments. The distance matrix consists of distances between every pair of points.

The method takes every instance from the training set one by one to use as a test set and calculates the instance's neighbors. The predicted label for the test instance is calculated based on its nearest neighbors and the label is added to a list of predictions made throughout the execution of the method.

After that the c-index is calculated based on pseudo-code we got in last exercise (Third Exercise: Prediction of metal ion content from multi-parameter data).

```
def LooCV(k, distanceMatrix):
    yPredictions = []
    for i in range(len(stdX)):
        neighbors = inferNeighbors(stdX, stdX[i], y, k, distanceMatrix[i])
        yPredictions.append(chooseMajorityLabel(neighbors,k))

    cIndex = calculateCIndex(yPredictions, y)
    printCIndexes(cIndex)

    return cIndex
```

5 SUMMING ALL THE METHODS

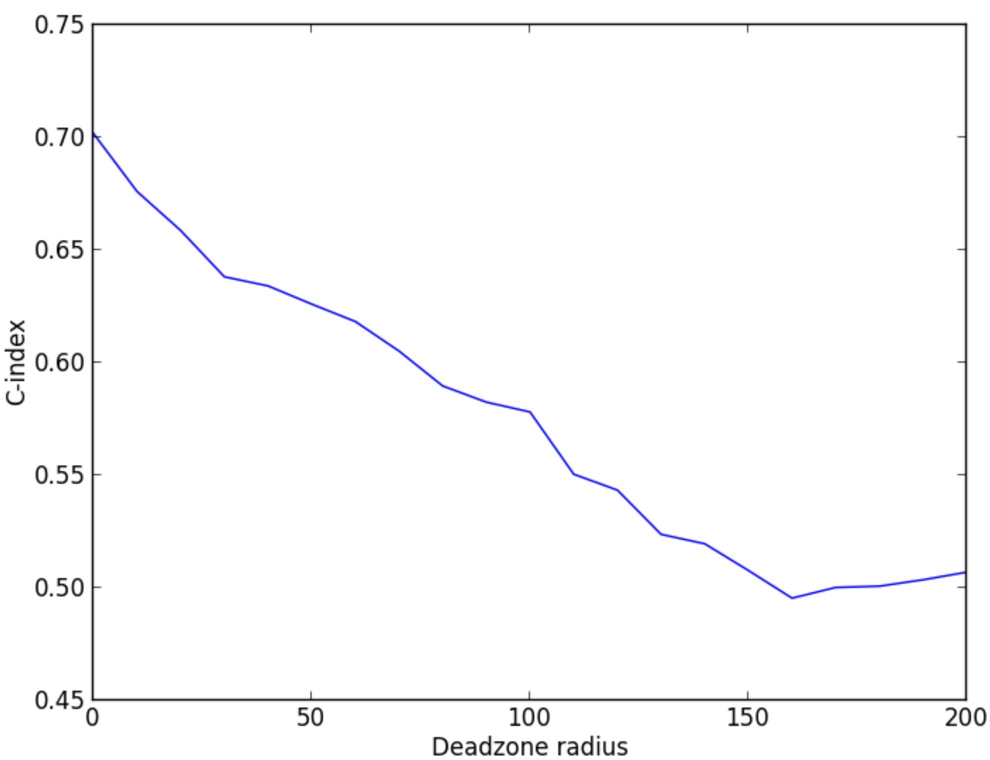
Firstly, the distance matrix is created. This matrix is used and modified during calculation. C-index is calculated for each of the dead zone radius "cases" using leave-one-out implementation of cross validation (see above) together with the pattern recognition algorithmn k(5)-nearest-neighbors. After each dead zone radius case (0,10,20…,200) the distance matrix is modified by disabling the next 10 nearest points. In the first round it disables 1-10 nearest features, in second round 11-20 nearest features and so on.

```
def main():
    startTime = int(round(time.time() * 1000))
    distanceMatrix = calculateDistanceMatrix()
    cIndexes = []
    deadZoneValues = []
    for i in range(21):
        print 'Leave-one-out CV with deadzone radius ' + str(i * 10) + ':'
        cIndexes.append(LooCV(5, distanceMatrix))
        deadZoneValues.append(i * 10)
        distanceMatrix = calculateDeadZone(distanceMatrix)

    endTime = int(round(time.time() * 1000))
    print 'Running time: ' + str(endTime - startTime) + 'ms'

    plotCIndexVsDeadZone(cIndexes, deadZoneValues)
```

All the used deadzone radius and c-indexes are plotted:



And same data as plotted above:

Leave-one-out CV with deadzone radius **0**: C-Index: 0.**701855867534**

Leave-one-out CV with deadzone radius **10**: C-Index: 0.**676058930829**

Leave-one-out CV with deadzone radius **20**: C-Index: 0.**658731830851**

Leave-one-out CV with deadzone radius **30**: C-Index: 0.**638168826969**

Leave-one-out CV with deadzone radius **40**: C-Index: 0.**634122722163**

Leave-one-out CV with deadzone radius **50**: C-Index: 0.**626040661936**

Leave-one-out CV with deadzone radius **60**: C-Index: 0.**618301230945**

Leave-one-out CV with deadzone radius **70**: C-Index: 0.**605187175657**

Leave-one-out CV with deadzone radius **80**: C-Index: 0.**589694664503**

Leave-one-out CV with deadzone radius **90**: C-Index: 0.**582504700215**

Leave-one-out CV with deadzone radius **100**: C-Index: 0.**578135914964**

Leave-one-out CV with deadzone radius **110**: C-Index: 0.**550505789349**

Leave-one-out CV with deadzone radius **120**: C-Index: 0.**543447067108**

Leave-one-out CV with deadzone radius **130**: C-Index: 0.**523801007659**

Leave-one-out CV with deadzone radius **140**: C-Index: 0.**519612461485**

Leave-one-out CV with deadzone radius **150**: C-Index: 0.**507752730185**

Leave-one-out CV with deadzone radius **160**: C-Index: 0.**495467424674**

Leave-one-out CV with deadzone radius **170**: C-Index: 0.**500218736745**

Leave-one-out CV with deadzone radius **180**: C-Index: 0.**500782202599**

Leave-one-out CV with deadzone radius **190**: C-Index: 0.**503624030384**

Leave-one-out CV with deadzone radius **200**: C-Index: 0.**507006575402**

6 CODE

```python
'''

Authors: Marco Willgren, 502606
         Jarno Vuorenmaa, 503618


Task steps:

1. Implement a Leave-One-Out cross validation with deadzone radius R = 0, 10, 20,
..., 200. So you will do 21 analyses in total here.
    Use 5-nearest neighbor as the prediction method. Remember normalization.

2. Calculate the C-index value for each of the deadzone radius cases.

3. Plot the C-index vs. Deadzone radius in a graph to visualize, how the prediction
performance changes with the deadzone radius.
    Set Y-axis to be the C-index and X-axis to be Deadzone radius

4. Return your implementation and the graph in a written report.

'''

import os
import operator
import scipy.spatial.distance as ssd
import numpy as np
import matplotlib.pyplot as pp
import time

if __name__ == '__main__':
    pass

basepath = os.path.dirname(__file__)
inputpath = os.path.abspath(os.path.join(basepath, "Data4/INPUT.csv"))
outputpath = os.path.abspath(os.path.join(basepath, "Data4/OUTPUT.csv"))
coordinatespath = os.path.abspath(os.path.join(basepath, "Data4/COORDINATES.csv"))

x = np.genfromtxt(inputpath, delimiter=',')
y = np.genfromtxt(outputpath, delimiter=',')
z = np.genfromtxt(coordinatespath, delimiter=',')

xArr = np.asarray(x)
stdX = (xArr - xArr.mean()) / xArr.std()


def calculateDistanceMatrix():
    distanceMatrix = []
    for i in range(len(z)):
        xAxis = []
        for j in range(len(z)):
            if i == j:
                xAxis.append(-1.0)
```

```
            else:
                xAxis.append(ssd.euclidean(z[i], z[j]))
        distanceMatrix.append(xAxis)

    return distanceMatrix

def calculateCIndex(predictions, labels):
    n = 0
    h_sum = 0
    for i in range(len(labels)):
        t = labels[i]
        p = predictions[i]
        for j in range(i+1,len(labels)):
            nt = labels[j]
            np = predictions[j]
            if t != nt:
                n = n + 1
                if (p < np and t < nt) or (p > np and t > nt):
                    h_sum = h_sum + 1
                elif (p < np and t > nt) or (p > np and t < nt):
                    h_sum = h_sum + 0
                elif (p == np):
                    h_sum = h_sum + 0.5

    if n == 0:
        return 0
    else:
        return h_sum/n

def LooCV(k, distanceMatrix):
    yPredictions = []
    for i in range(len(stdX)):
        neighbors = inferNeighbors(stdX, stdX[i], y, k, distanceMatrix[i])
        yPredictions.append(chooseMajorityLabel(neighbors,k))

    cIndex = calculateCIndex(yPredictions, y)
    printCIndexes(cIndex)

    return cIndex

def chooseMajorityLabel(neighbors, k):
    predictedOutcome = []
    sumOfMod = 0.0
    for i in range(len(neighbors)):
        sumOfMod = sumOfMod + neighbors[i][1]
    predictedOutcome.append(sumOfMod/k)

    return predictedOutcome


def inferNeighbors(trainSet, testInstance, labels, k, distRow):
    distances = []
    for x in range(len(trainSet)):
        if distRow[x] >= 0.0:
            distances.append((ssd.euclidean(trainSet[x], testInstance), labels[x]))
```

```
    distances.sort(key=operator.itemgetter(0))
    return distances[0:k]

def printCIndexes(cIndex):
    print 'C-Index: {a}'.format(a=cIndex)
    print

def calculateDeadZone(matrix):
    for i in range(len(matrix)):
        xAxis = matrix[i]
        for _ in range(10):
            minIndex = xAxis.index(min(filter(lambda x:x>=0.0, xAxis)))
            xAxis[minIndex] = -1.0

    return matrix

def plotCIndexVsDeadZone(cIndexes, deadZoneValues):
    pp.ylabel('C-index')
    pp.xlabel('Deadzone radius')
    pp.plot(deadZoneValues, cIndexes)
    pp.show()

def main():
    startTime = int(round(time.time() * 1000))
    distanceMatrix = calculateDistanceMatrix()
    cIndexes = []
    deadZoneValues = []
    for i in range(21):
        print 'Leave-one-out CV with deadzone radius ' + str(i * 10) + ':'
        cIndexes.append(LooCV(5, distanceMatrix))
        deadZoneValues.append(i * 10)
        distanceMatrix = calculateDeadZone(distanceMatrix)

    endTime = int(round(time.time() * 1000))
    print 'Running time: ' + str(endTime - startTime) + 'ms'

    plotCIndexVsDeadZone(cIndexes, deadZoneValues)

main()
```