Exercise 3

Applications of Data Analysis

# Data preprocessing

First of all the data is read with numpys genfromtxt – method into variables y and x. X contains modulator-data and y is metal concentration data.

basepath = os.path.dirname(\_\_file\_\_)

filepath = os.path.abspath(os.path.join(basepath, *"Water\_data.csv"*))

y = np.genfromtxt(filepath, delimiter=*','*, skiprows=1, usecols=range(3, 6))

x = np.genfromtxt(filepath, delimiter=*','*, skiprows=1, usecols=range(0,3))

Data is standardized using z-score normalization.

def **calculateZScore**():

xArr = np.asarray(x)

zScores = (xArr - xArr.mean()) / xArr.std()

return zScores

# Calculating K Nearest Neighbor and predict values

K nearest neighbor is calculated in inferErrors – method. This method calculates Euclidean distance for test instance against every instance of training set. Method returns k nearest neighbor leaving out n nearest neighbor (n = leaveOut attribute). In Leave-one-out the training set is same as test set and leaveOut is 1 to cut out the test instance from neighbors.

Values are predicted in chooseMajorityLabel – method. This method calculates mean for every modulators and returns the prediction based on that.

def **chooseMajorityLabel**(neighbors,k):

predictedOutcome = []

for i in range(3):

sumOfMod = 0.0

for j in range(len(neighbors)):

sumOfMod = sumOfMod + neighbors[j][1][i]

predictedOutcome.append(sumOfMod/k)

return predictedOutcome

def **inferNeighbors**(trainSet,testInstance,labels,k,leaveOut):

distances = []

for x in range(len(trainSet)):

distances.append((ssd.euclidean(trainSet[x], testInstance), labels[x]))

distances.sort(key=operator.itemgetter(0))

return distances[leaveOut:k+leaveOut]

# Leave-<T>-Out cross-validation

def **LooCV**(k):

yPredictions = []

stdX = calculateZScore()

for i in range(len(stdX)):

neighbors = inferNeighbors(stdX,stdX[i],y,k)

yPredictions.append(chooseMajorityLabel(neighbors,k))

cIndexCTotal = calculateCIndex(yPredictions,0,y)

cIndexCd = calculateCIndex(yPredictions,1,y)

cIndexPb = calculateCIndex(yPredictions,2,y)

printCIndexes(cIndexCTotal,cIndexCd,cIndexPb)

LooCV (Leave-One-Out) calculates neighbors for every instance in stardardized training data and majority labels. Majority labels are included to predicted values of y.

The result using LOO:

Leave-one-out cross-validation

C Indexes:

c-total: 0.823497067449

cd: 0.775997528249

pb: 0.819797551789

def **LfoCV**(k):

yPredictions = []

stdX = calculateZScore()

i = 0

kf = KFold(len(stdX),67)

for train\_index, test\_index in kf:

xTrain, xTest = stdX[train\_index], stdX[test\_index]

yTrain, yTest = y[train\_index], y[test\_index]

for i in range(len(xTest)):

neighbors = inferNeighbors(xTrain, xTest[i],yTrain, k)

yPredictions.append(chooseMajorityLabel(neighbors, k))

cIndexCTotal = calculateCIndex(yPredictions,0,yTrain)

cIndexCd = calculateCIndex(yPredictions,1,yTrain)

cIndexPb = calculateCIndex(yPredictions,2,yTrain)

printCIndexes(cIndexCTotal,cIndexCd,cIndexPb)

LfoCV (Leace-Four-Out) divides data set into training and test sets. Test set contains for instances and every test set includes different instances. For every test set calculates neighbors and chooses majority label. Majority labels are included to predicted values of y.

The result using LFO:

Leave-four-out cross-validation

C Indexes:

c-total: 0.784681989924

cd: 0.664510447036

pb: 0.759032141131

# Calculating c-index

def **calculateCIndex**(predictions,index,labels):

n = 0

h\_sum = 0

for i in range(len(labels)):

t = labels[i][index]

p = predictions[i][index]

for j in range(i+1,len(labels)):

nt = labels[j][index]

np = predictions[j][index]

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

C-index values are calculated based on given pseudo-code.

# Code

*'''*

*Authors: Marco Willgren, 502606*

*Jarno Vuorenmaa, 503618*

*Exercise 3: Applications of data analysis*

*The Water\_data.csv file is a multi-parameter dataset consisting of 268 samples obtained from 67 mixtures of Cadmium, Lead, and tap water.*

*Three features (attributes) where measured for each samples (Mod1, Mod2, Mod3).*

*Tasks*

*Use K-Nearest Neighbor Regression to predict total metal concentration (c\_total), concentration of Cadmium (Cd) and concentration of Lead (Pb), for each sample.*

*- The data should be normalized using z-score.*

*- Implement Leave-One-Out Cross Validation approach and calculate the C-index for each output (c-total, Cd, Pb).*

*- Implement Leave-Four-Out Cross Validation and calculate the C-index for each output (c-total, Cd, Pb).*

*'''*

import os

import numpy as np

import scipy.spatial.distance as ssd

import operator

from sklearn.cross\_validation import KFold

if \_\_name\_\_ == *'\_\_main\_\_'*:

pass

basepath = os.path.dirname(\_\_file\_\_)

filepath = os.path.abspath(os.path.join(basepath, *"Water\_data.csv"*))

y = np.genfromtxt(filepath, delimiter=*','*, skiprows=1, usecols=range(3, 6))

x = np.genfromtxt(filepath, delimiter=*','*, skiprows=1, usecols=range(0,3))

def **LfoCV**(k):

yPredictions = []

stdX = calculateZScore()

i = 0

kf = KFold(len(stdX),67)

for train\_index, test\_index in kf:

xTrain, xTest = stdX[train\_index], stdX[test\_index]

yTrain, yTest = y[train\_index], y[test\_index]

for i in range(len(xTest)):

neighbors = inferNeighbors(xTrain, xTest[i],yTrain, k)

yPredictions.append(chooseMajorityLabel(neighbors, k))

cIndexCTotal = calculateCIndex(yPredictions,0,yTrain)

cIndexCd = calculateCIndex(yPredictions,1,yTrain)

cIndexPb = calculateCIndex(yPredictions,2,yTrain)

printCIndexes(cIndexCTotal,cIndexCd,cIndexPb)

def **LooCV**(k):

yPredictions = []

stdX = calculateZScore()

for i in range(len(stdX)):

neighbors = inferNeighbors(stdX,stdX[i],y,k)

yPredictions.append(chooseMajorityLabel(neighbors,k))

cIndexCTotal = calculateCIndex(yPredictions,0,y)

cIndexCd = calculateCIndex(yPredictions,1,y)

cIndexPb = calculateCIndex(yPredictions,2,y)

printCIndexes(cIndexCTotal,cIndexCd,cIndexPb)

def **printCIndexes**(cTotal,cd,pb):

print *'C Indexes:'*

print *'c-total: {a}'*.format(a=cTotal)

print *'cd: {b}'*.format(b=cd)

print *'pb: {c}'*.format(c=pb)

def **calculateCIndex**(predictions,index,labels):

n = 0

h\_sum = 0

for i in range(len(labels)):

t = labels[i][index]

p = predictions[i][index]

for j in range(i+1,len(labels)):

nt = labels[j][index]

np = predictions[j][index]

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 **chooseMajorityLabel**(neighbors,k):

predictedOutcome = []

for i in range(3):

sumOfMod = 0.0

for j in range(len(neighbors)):

sumOfMod = sumOfMod + neighbors[j][1][i]

predictedOutcome.append(sumOfMod/k)

return predictedOutcome

def **inferNeighbors**(trainSet,testInstance,labels,k):

distances = []

for x in range(len(trainSet)):

distances.append((ssd.euclidean(trainSet[x], testInstance), labels[x]))

distances.sort(key=operator.itemgetter(0))

return distances[1:k+1]

def **calculateZScore**():

xArr = np.asarray(x)

zScores = (xArr - xArr.mean()) / xArr.std()

return zScores

LooCV(10)

LfoCV(10)

‘