Exercise 6

Applications of Data Analysis

Marco Willgren 502606 [mahewi@utu.fi](mailto:mahewi@utu.fi)

Jarno Vuorenmaa 503618 [jkivuo@utu.fi](mailto:jkivuo@utu.fi)

# Variance of cross-validation

def **nonSignalCV**(size):

predictions = []

cIndexes = []

for \_ in range(100):

features, labels = generateRandomData(size)

for j in range(len(features)):

predictions.append(leaveOneOutWithKNN(features, labels, j))

cIndexes.append(calculateCIndex(predictions, labels))

predictions = []

mean, variance = calculateCIndexMeanAndVariance(cIndexes)

performance = inferPerformance(cIndexes) \* 100

print *'Random data matrix size: '* + str(size)

print *'Mean: '* + str(mean)

print *'Variance: '* + str(variance)

print *'%-tage of C-Indexes over 0.7: '* + str(performance) + *'%'*

print

print

pp.hist(cIndexes, 10)

pp.xlabel(*'C-Index'*)

pp.ylabel(*'Frequency'*)

pp.show()

## Generating data

The method generates random features and for each of them a label. Feature values are between 1 and 49 and labels are 1 or 0. Half of the labels have value of 1. Features and labels don’t correlate in any way because data is randomized separately.

def **generateRandomData**(size):

features = []

labels = []

half = size / 2

for i in range(size):

features.append([rand(1, 50)])

if i < half:

labels.append(0)

else:

labels.append(1)

shuf(labels)

return features, labels

## Leave-one-out with k-nearest neighbor

The test instance is removed from the feature and label data and we used scipy’s method to calculate 3-nearest neighbor. The method returns the prediction based on classifier model.

def **leaveOneOutWithKNN**(features, labels, indexOfTest):

featuresTemp = list(features)

labelsTemp = list(labels)

testInstance = features[indexOfTest]

del featuresTemp[indexOfTest]

del labelsTemp[indexOfTest]

neigh = KNeighborsClassifier(n\_neighbors=3)

neigh.fit(featuresTemp, labelsTemp)

return neigh.predict(testInstance)

## Results

First of all c-indexes are calculated and stored into array. Then the mean and variance values are calculated from this array. Also percentage of c-indexes over 0.7 is calculated. The c-indexes are plotted as a histogram.

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 **calculateCIndexMeanAndVariance**(cIndexes):

mean = np.mean(cIndexes)

variance = np.mean((cIndexes - mean)\*\*2)

return mean, variance

def **inferPerformance**(cIndexes):

count = 0.0

for i in range(len(cIndexes)):

if (cIndexes[i] > 0.7):

count = count + 1.0

return count / len(cIndexes)

Non-signal data learning

Random data in range (1, 49)

Labels are binary (0, 1)

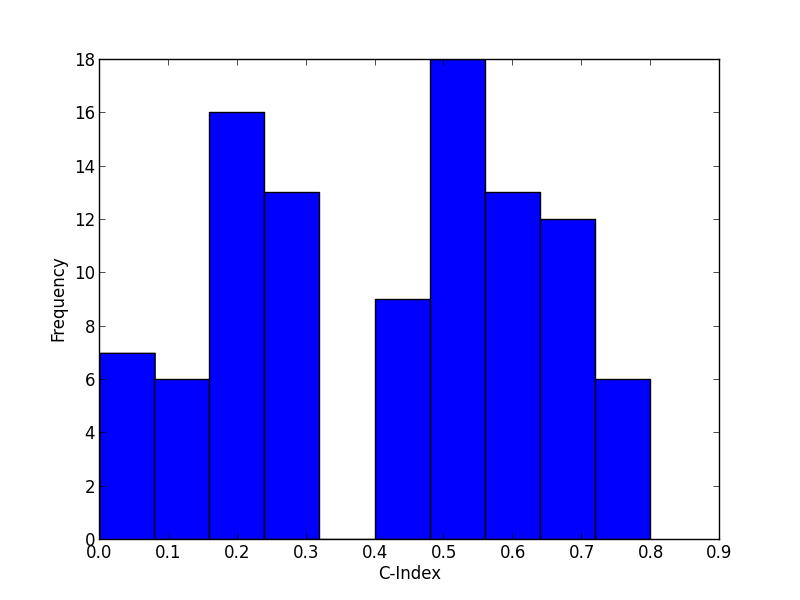
-----------------------------

Random data matrix size: 10

Mean: 0.413

Variance: 0.051531

%-tage of C-Indexes over 0.7: 6.0%

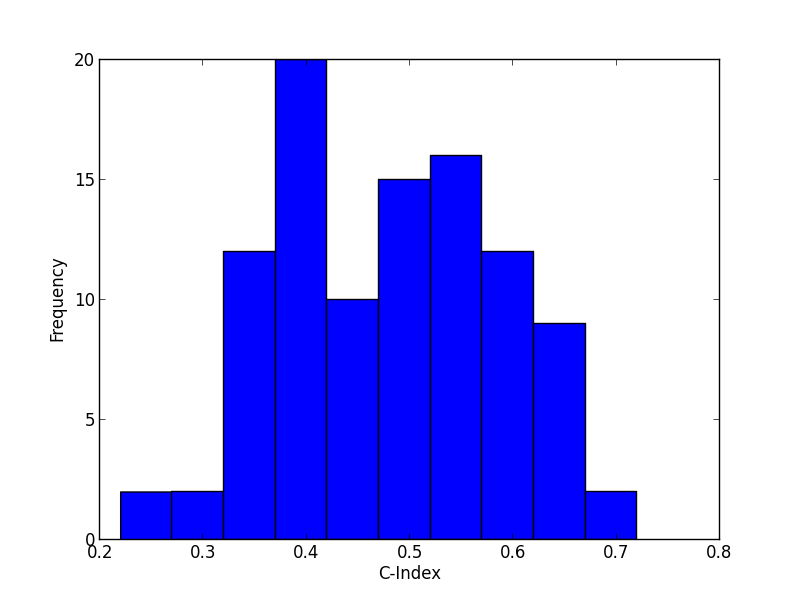


Random data matrix size: 50

Mean: 0.4814

Variance: 0.01101804

%-tage of C-Indexes over 0.7: 1.0%

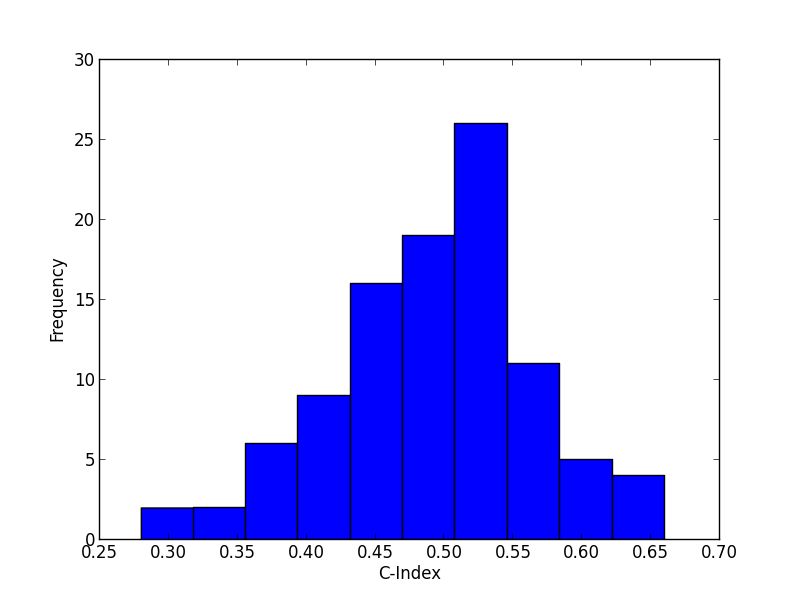


Random data matrix size: 100

Mean: 0.4933

Variance: 0.00522411

%-tage of C-Indexes over 0.7: 0.0%

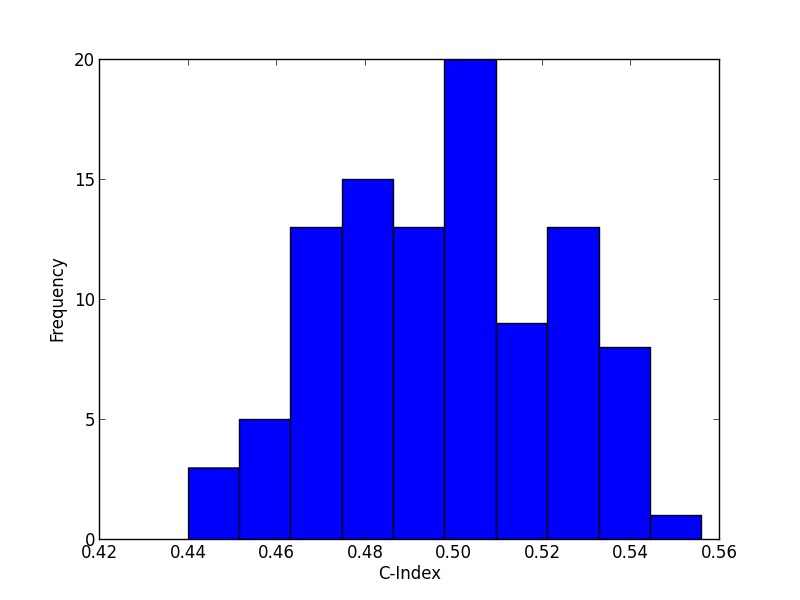


Random data matrix size: 500

Mean: 0.49636

Variance: 0.0006244304

%-tage of C-Indexes over 0.7: 0.0%



## Analysing the results

When the data size is increasing the mean value stabilizes near 0.5, giving as a hint that the data is random/non-signal. And also the variance is decreasing as data size is increasing. This can be also seen from the histograms.

# Mis-using feature selection

def **featureSelectedCV**(rightWay):

features, labels = generateLoadsOfRandomData()

predictions = []

if not rightWay:

bestFeatures, labels = selectBestCorrelations(features, labels, 0, rightWay, 10)

for i in range(len(features)):

if rightWay:

bestFeatures, labels = selectBestCorrelations(features, labels, i, rightWay, 10)

predictions.append(leaveOneOutWithKNN(bestFeatures, labels, i))

if not rightWay:

print *'C-Index (wrong way): '* + str(calculateCIndex(predictions, labels))

else:

print *'C-Index (right way): '* + str(calculateCIndex(predictions, labels))

Main method gets a Boolean value as an argument, which tells whether to include the test instance in feature selection or not. Method uses same methods for predicting labels and calculating the c-index as the first assignment.

First we need to generate random data with sample size of 50 and including 1000 features. Labels are binaries and divides equally.

def **generateLoadsOfRandomData**():

features = []

labels = []

for i in range(50):

col = []

if (i < 25):

labels.append(0)

else:

labels.append(1)

for \_ in range(1000):

col.append(rand(1, 50))

features.append(col)

shuf(labels)

return features, labels

## Selecting 10 best features

At first method leaves out test instance if the Boolean argument (rightWay) is true. Otherwise this step is not included. Method uses scipy.stats.Kendalltau to calculate correlations between each column and labels. Correlation values are sorted in decreasing order and 10 features mapped to highest correlation values are returned.

def **selectBestCorrelations**(features, labels, i, rightWay, selectCount):

tauVals = []

bestFeatures = []

features = np.array(features)

if rightWay:

tempFeatures = np.array(filterTestInstance(features, i))

tempLabels = np.array(filterTestInstance(labels, i))

else:

tempFeatures = features

tempLabels = labels

for i in range(1000):

tauVal, \_ = tau(tempFeatures[:,i], tempLabels)

tauVals.append((abs(tauVal), i))

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

tauVals = tauVals[::-1]

tauVals = tauVals[:10]

for i in range(len(tauVals)):

bestFeatures.append(features[:,tauVals[i][1]])

return np.transpose(bestFeatures), labels

## Analysing the results

Mis-using feature selection

Random data in range (1, 49)

Labels are binary (0, 1)

-----------------------------

C-Index (wrong way): 0.78

C-Index (right way): 0.28

Using the wrong way the results are biased because the test instance is included in selecting best features. Thus the results seem good as c-index is over 0.7. This happens because method chooses better features knowing the test instance also.

# Code

*'''*

*Authors: Marco Willgren, 502606*

*Jarno Vuorenmaa, 503618*

*'''*

from sklearn.neighbors import KNeighborsClassifier

from random import randint as rand

from random import shuffle as shuf

import numpy as np

from matplotlib import pyplot as pp

from scipy.stats import kendalltau as tau

import operator

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

pass

def **generateRandomData**(size):

features = []

labels = []

half = size / 2

for i in range(size):

features.append([rand(1, 50)])

if i < half:

labels.append(0)

else:

labels.append(1)

shuf(labels)

return features, labels

def **generateLoadsOfRandomData**():

features = []

labels = []

for i in range(50):

col = []

if (i < 25):

labels.append(0)

else:

labels.append(1)

for \_ in range(1000):

col.append(rand(1, 50))

features.append(col)

shuf(labels)

return features, labels

def **selectBestCorrelations**(features, labels, i, rightWay, selectCount):

tauVals = []

bestFeatures = []

features = np.array(features)

if rightWay:

tempFeatures = np.array(filterTestInstance(features, i))

tempLabels = np.array(filterTestInstance(labels, i))

else:

tempFeatures = features

tempLabels = labels

for i in range(1000):

tauVal, \_ = tau(tempFeatures[:,i], tempLabels)

tauVals.append((abs(tauVal), i))

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

tauVals = tauVals[::-1]

tauVals = tauVals[:10]

for i in range(len(tauVals)):

bestFeatures.append(features[:,tauVals[i][1]])

return np.transpose(bestFeatures), labels

def **leaveOneOutWithKNN**(features, labels, indexOfTest):

featuresTemp = list(features)

labelsTemp = list(labels)

testInstance = features[indexOfTest]

del featuresTemp[indexOfTest]

del labelsTemp[indexOfTest]

neigh = KNeighborsClassifier(n\_neighbors=3)

neigh.fit(featuresTemp, labelsTemp)

return neigh.predict(testInstance)

def **nonSignalCV**(size):

predictions = []

cIndexes = []

for \_ in range(100):

features, labels = generateRandomData(size)

for j in range(len(features)):

predictions.append(leaveOneOutWithKNN(features, labels, j))

cIndexes.append(calculateCIndex(predictions, labels))

predictions = []

mean, variance = calculateCIndexMeanAndVariance(cIndexes)

performance = inferPerformance(cIndexes) \* 100

print *'Random data matrix size: '* + str(size)

print *'Mean: '* + str(mean)

print *'Variance: '* + str(variance)

print *'%-tage of C-Indexes over 0.7: '* + str(performance) + *'%'*

print

print

pp.hist(cIndexes, 10)

pp.xlabel(*'C-Index'*)

pp.ylabel(*'Frequency'*)

pp.show()

def **calculateCIndexMeanAndVariance**(cIndexes):

mean = np.mean(cIndexes)

variance = np.mean((cIndexes - mean)\*\*2)

return mean, variance

def **inferPerformance**(cIndexes):

count = 0.0

for i in range(len(cIndexes)):

if (cIndexes[i] > 0.7):

count = count + 1.0

return count / len(cIndexes)

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 **filterTestInstance**(listA, i):

listTemp = list(listA)

del listTemp[i]

return listTemp

def **featureSelectedCV**(rightWay):

features, labels = generateLoadsOfRandomData()

predictions = []

if not rightWay:

bestFeatures, labels = selectBestCorrelations(features, labels, 0, rightWay, 10)

for i in range(len(features)):

if rightWay:

bestFeatures, labels = selectBestCorrelations(features, labels, i, rightWay, 10)

predictions.append(leaveOneOutWithKNN(bestFeatures, labels, i))

if not rightWay:

print *'C-Index (wrong way): '* + str(calculateCIndex(predictions, labels))

else:

print *'C-Index (right way): '* + str(calculateCIndex(predictions, labels))

def **printNSDHeader**():

print *'Non-signal data learning'*

print *'Random data in range (1, 49)'*

print *'Labels are binary (0, 1)'*

print *'-----------------------------'*

print

print

def **printFSHeader**():

print *'Mis-using feature selection'*

print *'Random data in range (1, 49)'*

print *'Labels are binary (0, 1)'*

print *'-----------------------------'*

print

print

def **main**():

printNSDHeader()

nonSignalCV(10)

nonSignalCV(50)

nonSignalCV(100)

nonSignalCV(500)

printFSHeader()

featureSelectedCV(False)

featureSelectedCV(True)

main()