Prof. R. Rojas

Mustererkennung, WS17/18

Übungsblatt 2

Boyan Hristov, Nedeltscho Petrov

31. Oktober 2017

Link zum Git Repository: https://github.com/BoyanH/Freie-Universitaet-Berlin/tree/master/MachineLearning/Homework2

Score

Das ist die Ausgabe des Programs und damit auch das Score

```
Score for 3 vs 7: 100.0%
Score for 3 vs 8: 100.0%
Score for 5 vs 7: 100.0%
Score for 5 vs 8: 100.0%
Score for 7 vs 8: 100.0%
```

Lineare Regression

Wir benutzen die folgende Formel für lineare Regression mit Least Squares

$$\vec{B} = (X^T X)^{-1} X^T \vec{y} \tag{1}$$

wobei $\vec{B}$ die Lösung von der folgenden Gleichung ist: (im besten Fall, falls Matrix mit vollem Rank)

$$y_i = B_0 + B_i X_{11} + B_i X12 + ... + B_n X_{in} \tag{2}$$

dabei ist X die Eingabematrix mit je Zeile ein Punkt im n-Dimensionalen Raum. Damit wir diese Gleichung lösen können, mit B_0 als 1. Glied, müssen wir erstmal eine Spalte mit Einsen einfügen.

```
ones = np.ones((len(self.trainData), 1), dtype=float)
X = np.append(ones, self.trainData, axis = 1)
```

Weiter, um die 2 Klassen gleichwertig in der linearen Regression zu betrachten, dürfen wir natürlich nicht gleich die Labels benutzen, sondern diese erstmal normalisieren (so zu sagen). Dabei mapen wir die eine Labels zu -1 und die andere zu 1. Z.B wenn wir zwischen 3 und 5 unterscheiden wollen, wird das je Label 3 zu -1 und je 5 zu 1. Das passiert folgendermaßen

```
 def normalizeLabels(self, labels):
  return list(map(lambda x: -1 if int(x) == self.classA else 1, labels))
```

Und damit unsere fit Methode

```
 def fit(self):
  ones = np.ones((len(self.trainData), 1), dtype=float)
  X = np.append(ones, self.trainData, axis = 1)
  xtxInversed = LinearRegressionClassifier.pseudoInverse(X.T.dot(X))
  normalizedLabels = self.normalizeLabels(self.trainLabels)
  self.beta = xtxInversed.dot(X.T).dot(normalizedLabels)
```

Pseudoinverse

Wir benutzen die Moore-Penrose Pseudoinverse. Nach der Folmel gilt:

$$A^+ = \lim_{\delta \to 0} A^*(AA^* + \delta E)^{-1} \tag{3}$$

Wobei A^* die adjungierte Matrix ist. Da wir aber in dem Bereich der reelen Zahlen uns befinden, ist die adjungierte Matrix identisch zu der transponierten. Deswegen gilt:

$$A^+ = \lim_{\delta \to 0} A^T(AA^T + \delta E)^{-1} \tag{4}$$

Damit unsere Implementierung

```
@staticmethod
 def pseudoInverse(X):
  delta = np.nextafter(np.float16(0), np.float16(1)) # as close as we can get to lim delta
    -> 0
  pseudoInverted = X.T.dot(np.linalg.inv(X.dot(X.T) + delta * np.identity(len(X))))
```

Vollständige Implementierung (Ohne Classifier Klasse)

```
import pandas as pd
import numpy as np
from operator import itemgetter
import matplotlib.pyplot as plt
import os
from Classifier import Classifier

import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt

class LinearRegressionClassifier(Classifier):
```

```
# B = (X^TX)^-1X^T*y

# hat H = X*B^ =` X(X^TX)-1X^T

@staticmethod
def pseudoInverse(X):
 # Determining whether a matrix is invertable or not with gaus elimination
 # (most efficient from the naive approaches) takse n^3 time, which
 # makes the whole programm slower for not much precision gain
 # therefore we simply calculate the pseudo invsersed matrix every time

 # if LinearRegressionClassifier.isInvertable(X):
 #  return np.linalg.inv(X)

 # calculate pseudo-inverse A+ of a matrix A (X in our case)
 # A+ = lim delta->0 A*(A.A* + delta.E)^(-1) where A* is the conjugate transpose
 # and E ist the identity matrix

 # In our case, we are working with real numbers, so A* = A^T
 # so the formula is A^T(A.A^T + delta.E)^(-1)

 delta = np.nextafter(np.float16(0), np.float16(1)) # as close as we can get to lim delta -> 0
 pseudoInverted = X.T.dot(np.linalg.inv(X.dot(X.T) + delta * np.identity(len(X))))

 return pseudoInverted

@staticmethod
def isInvertable(X):
 # apply gaus elimination
 # if the matrix is transformable in row-echelon form
 # then it is as well inverable

 X = np.copy(X) # don't really change given matrix
 m = len(X)
 n = len(X[0])
 for k in range(min(m, n)):
  # Find the k-th pivot:
  # i_max  = max(i = k ... m, abs(A[i, k]))
  i_max = k
  max_value = X[k][k]
  for i in range(k, m):
   if X[i][k] > max_value:
    max_value = X[i][k]
    i_max = i
  if X[i_max][k] == 0:
   return False
  for i in range(n):
   temp = X[k][i]
   X[k][i] = X[i_max][i]
   X[i_max][i] = temp
  # Do for all rows below pivot:
  for i in range(k+1, m):
   # for i = k + 1 ... m:
   f = X[i][k] / X[k][k]
   # Do for all remaining elements in current row:
   for j in range(k + 1, n):
    X[i][j]  = X[i][j] - (X[k][j] * f)
   X[i][k] = 0

 return True

def __init__(self, trainSet, testSet, classA, classB):
 self.classA = classA
 self.classB = classB

 trainSet = self.filterDataSet(trainSet)
 testSet = self.filterDataSet(testSet)
```

```
  self.trainData = list(map(lambda x: x[1:], trainSet))
  self.trainLabels = list(map(itemgetter(0), trainSet))
  self.testSet = list(map(lambda x: x[1:], testSet))
  self.testLabels = list(map(itemgetter(0), testSet))

  self.fit()

 def filterDataSet(self, dataSet):
  return list(filter(lambda x: int(x[0]) in [self.classA, self.classB], dataSet))

 def fit(self):
  # fill X with (1,1...,1) in it's first column to be able to get the
  # wished yi = B0 + B1Xi1 + B2Xi2 + ... + BnXin
  ones = np.ones((len(self.trainData), 1), dtype=float)
  X = np.append(ones, self.trainData, axis = 1)

  # and then used the following formula to calculate our closest possible B
  # which solves best our least squares regression
  # B = (X^TX)^(-1)X^Ty
  # where y = (-1, 1, -1, 1, ..., -1) (for example) is a vector
  # of the labels corresponding to the given data points

  xtInversed = LinearRegressionClassifier.pseudoInverse(X.T.dot(X))

  # normalize y, so the two possible classes are maped to -1 or 1
  normalizedLabels = self.normalizeLabels(self.trainLabels)
  self.beta = xtInversed.dot(X.T).dot(normalizedLabels)

 def predictSingle(self, X):
  X = np.append(np.array([1]), np.array(X), axis=0)
  return self.classA if (X.dot(self.beta) < 0) else self.classB

 def predict(self, X):
  return np.array(list(map(lambda x: self.predictSingle(x), X)))

 def test(self):
  print('Score for {} vs {}: {}%'.format(
   self.classA, self.classB, self.score(self.testSet, self.testLabels) * 100))
  self.printConfusionsMatrix(self.confusion_matrix(self.testSet, self.testLabels))


 def normalizeLabels(self, labels):
  return list(map(lambda x: -1 if int(x) == self.classA else 1, labels))

 def printConfusionsMatrix(self, matrix):
  explicitImgPath = os.path.join(dir_path, './Plots/confusion_matrix_for_{}vs_{}.png'.format(
   self.classA, self.classB))
  digits = [str(x) for x in range(10)];

  df_cm = pd.DataFrame(matrix, index = digits,
  columns = digits )

  plt.figure(figsize = (11,7))
  heatmap = sn.heatmap(df_cm, annot=True)

  heatmap.set(xlabel='Klassifiziert', ylabel='Erwartet')

  plt.savefig(explicitImgPath, format='png')

def extractDataFromLine(line):
 line = line.replace(' \n', '') # clear final space and new line chars
 return list(map(float, line.split(' '))); # map line to a list of floats

def parseDataFromFile(file, dataArr):
 for line in file:
  line = line.replace(' \n', '') # clear final space and new line chars
```

```
  currentDigitData = extractDataFromLine(line)
  dataArr.append(currentDigitData)


trainSet = []
testSet = []

dir_path = os.path.dirname(os.path.realpath(__file__))
explicitPathTrainData = os.path.join(dir_path, './Dataset/train')
explicitPathTestData = os.path.join(dir_path, './Dataset/test')
trainFile = open(explicitPathTestData, 'r')
testFile = open(explicitPathTestData, 'r')

parseDataFromFile(trainFile, trainSet)
parseDataFromFile(testFile, testSet)

LinearRegressionClassifier(trainSet, testSet, 3, 5).test()
LinearRegressionClassifier(trainSet, testSet, 3, 7).test()
LinearRegressionClassifier(trainSet, testSet, 3, 8).test()
LinearRegressionClassifier(trainSet, testSet, 5, 7).test()
LinearRegressionClassifier(trainSet, testSet, 5, 8).test()
LinearRegressionClassifier(trainSet, testSet, 7, 8).test()
```