Instituto Politécnico Nacional

Escuela Superior de Cómputo

Regresión Logística y matriz de confusión

Natural Language Processing

Estudiante:
Nicolás Sayago Abigail

Profesora:
Olga Kolesnicova

Mayo 30, 2020

1 Regresión Logística

ITERACIONES

Resultados obtenidos con 10000 iteraciones:

```
TRAINING TEST with 10000 iterations
Iteration: 0 Cost Function value: 0.6275865217749537
Iteration: 500 Cost Function value: 0.2837052720178191
Iteration: 1000 Cost Function value: 0.27191124552954504
Iteration: 1500 Cost Function value: 0.2614080475987347
Iteration: 2000 Cost Function value: 0.251937148105418
Iteration: 2500 Cost Function value: 0.2433182216237096
Iteration: 3000 Cost Function value: 0.23541925645537035
Iteration: 3500 Cost Function value: 0.22813966504320643
Iteration: 4000 Cost Function value: 0.22140025582130374
Iteration: 4500 Cost Function value: 0.2151370056366762
Iteration: 5000 Cost Function value: 0.20929703286029352
Iteration: 5500 Cost Function value: 0.20383589670786886
Iteration: 6000 Cost Function value: 0.19871572491585304
Iteration: 6500 Cost Function value: 0.19390387578921975
Iteration: 7000 Cost Function value: 0.18937195527017114
Iteration: 7500 Cost Function value: 0.18509507637253508
Iteration: 8000 Cost Function value: 0.18105128833296358
Iteration: 8500 Cost Function value: 0.17722112748770408
Iteration: 9000 Cost Function value: 0.17358725745439857
Iteration: 9500 Cost Function value: 0.1701341762480761
Cost Function with Testing set: 0.8243822395313952
```

RESULTADOS TESTING SET

A continuación se muestran algunos valores del testing set con los cuales se obtuvo la matriz de confusión:

```
TESTING SET
Prediction: 0.961989946505473 Real: 1
Prediction: 0.950483515881912 Real: 1
Prediction: 0.9717251593854935 Real: 1
Prediction: 0.9934372544800677 Real: 1
Prediction: 0.9539941863606264 Real: 1
Prediction: 0.9802495307053958 Real: 1
Prediction: 0.8903514388419443 Real: 0
Prediction: 0.9138779122914039 Real: 0
Prediction: 0.6286394526868379 Real: 0
Prediction: 0.6576702240638426 Real: 0
Prediction: 0.7216431090266804 Real: 0
Prediction: 0.6784656368230143 Real: 0
Prediction: 0.5674334399891708 Real: 0
Prediction: 0.6479512512388226 Real: 0
```

MATRIZ DE CONFUSIÓN

Para obtener la **matriz de confusión** tomamos con valor 1 a los valores que fueran mayores o iguales a 0.90.

```
Confusion matrix
------------ 1 -------- 0 -----
--- 1 ----  141 ------- 4 -----
--- 0 -----  15 ------ 237 ----


Precision: 0.9724137931034482
Recall: 0.9038461538461539
F1: 0.9368770764119602
```

✓Código fuente

```
import nltk
import re
import math
from bs4 import BeautifulSoup
from pickle import dump, load
from nltk.corpus import cess_esp
from nltk.corpus import PlaintextCorpusReader
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import numpy as np

##############################################
#                            NORMALIZATION
##############################################
#Parameters: File path, encoding
#Return: String with only lower case letters
#Notes: path =
 ↪  '/Users/27AGO2019/Desktop/AbiiSnn/GitHub/Natural-Language-Processing/corpus/e961024
def getText(corpusRoot, code):
        f = open(corpusRoot, encoding = code) #Cod: utf-8, latin-1
        text = f.read()
        f.close()
        soup = BeautifulSoup(text, 'lxml')
        text = soup.get_text()
        text = text.lower()
        return text
```

```
#Parameters: Text
#Return: List of original tokens
def getTokens(text):
        tokens = nltk.word_tokenize(text)
        return tokens

def getWords(fpath, code):
        f = open(fpath, encoding = code) #Cod: utf-8, latin-1
        text = f.read()
        f.close()

        words = re.sub(" ", " ",  text).split()
        return words

##############################################
#                                     LEMMAS
##############################################
# Return: Dictionary
def createDicLemmas(tokensLemmas):
        lemmas = {}
        j = 0
        for i in range(0, len(tokensLemmas)- 2, 3):
                word = tokensLemmas[i]
                tag = tokensLemmas[i+1]
                val = tokensLemmas[i+2]
                l = (word, tag[0].lower())
                lemmas[l] = val
                j = j+1
        return lemmas

##############################################
#                                FRECUENCY
##############################################
def getVectors(vocabulary, matrix):
        vectors = []
        for x in matrix:
                vector = []
                for word in vocabulary:
                        frec = x.count(word)
                        vector.append(frec)
                vectors.append(vector)
        return vectors

def getFrecuency(vectors):
```

```
        matrix = []
        for vector in vectors:
                aux = np.array(vector)
                total = np.sum(aux)
                p = []
                p.append(1)
                for element in vector:
                        ans = element / total
                        p.append(ans)
                auxNP = np.array(p)
                matrix.append(auxNP)
        m = np.array(matrix)
        return m

##############################################
#                                    TAGGING
##############################################
def tag(tokens):
        s_tagged = nltk.pos_tag(tokens)
        l = list()
        for tag in s_tagged:
                pos = 'n'
                if len(tag[1][0]) > 0:
                        pos = tag[1][0].lower()
                tu = (tag[0], pos)
                l.append(tu)
        return l

def getVocabulary(matrix):
        s = set()
        for i in matrix:
                for j in i:
                        s.add(j)
        vocabulary = sorted(s)
        return vocabulary

##############################################
#                    LOGISTIC REGRESSION
##############################################
def getHypothesis(product):
        H_theta = product.transpose()
        H_theta = 1 / (1 + np.exp(-H_theta))
        return H_theta.transpose()
```

```
def costFunction(H_theta, Y):
        m = len(Y[0])
        a = np.multiply(Y, np.log(H_theta))
        b = np.multiply(1-Y, np.log(1 - H_theta))
        cost = (-1 / m) * np.sum(a+b)
        return cost

def gradientDescent(theta, H_theta, Y, matrix, learningRate):
    # theta: n x 1, H_theta: 1 x m, Y: 1xm, X: n x m
    m = len(Y[0])
    aux = (1/m) * np.dot((H_theta - Y), matrix.transpose()) # 1 x n
    aux = aux.transpose() # n x 1
    thetaTemp = theta - (learningRate * aux)
    return thetaTemp # n x 1

##############################################
##############################################
##############################################

# Get Tokens by Generate.txt to create dictionary of lemmas
#Read file spam/ham
fpathCorpus =
 ↪  '/Users/abiga/Desktop/AbiiSnn/GitHub/Natural-Language-Processing/Practice/22/corpus
code = 'ISO-8859-1'
corpus = getText(fpathCorpus, code)
tokens = getTokens(corpus)

#Matrix and Y
matrix = list()
auxY = list()
xi = list()
for token in tokens:
        y = list()
        if token != 'spam' and token != 'ham':
                xi.append(token)
        else:
                typeMessage = 0
                if token == 'ham':
                        typeMessage = 1
                y.append(typeMessage) #Create Y
                ynp = np.array(y)
                auxY.append(ynp)
                matrix.append(xi) #Create X
                xi = list()
```

```
Yn = np.array(auxY)
Y = Yn.transpose()

#Tagging
matrixTag = list()
for i in range(0, len(matrix)):
        auxTag = tag(matrix[i])
        matrixTag.append(auxTag)

#Lemmatize
matrixLem = list()
wnl = WordNetLemmatizer()
for i in range(0, len(matrixTag)):
        l = list()
        for j in range(0, len(matrixTag[i])):
                lemma = wnl.lemmatize(matrixTag[i][j][0])
                t = (lemma, matrixTag[i][j][1])
                l.append(t)
        matrixLem.append(l)

vocabulary = getVocabulary(matrixLem)

# Sacar frecuencia y obtener matrix
vectors = getVectors(vocabulary, matrixLem)
frecuency = getFrecuency(vectors)

Ybackup = Y.transpose()
YTest = Ybackup[928:]
Y = Ybackup[:928]
frecuencyTraining = frecuency[:928]
frecuencyTesting = frecuency[928:]


YTest = YTest.transpose()
Y = Y.transpose()

X = frecuencyTraining.transpose() # n x m
theta = np.zeros(shape = (len(X), 1)) # n x 1
thetaT = theta.transpose() # 1 x n
mul = thetaT.dot(X)
H_theta = getHypothesis(mul) # 1 x m (1 x 928)
# y = 1 x m (1 x 928)
cost = costFunction(H_theta, Y) # escalar
learningRate = 0.4
```

```
print("TRAINING TEST with 10000 iterations")
for ite in range(0, 10000):
        # theta: n x 1, H_theta: 1 x m, Y: 1xm, X: n x m
    tempTheta = gradientDescent(theta, H_theta, Y, X, learningRate)
    theta = tempTheta
    thetaT = tempTheta.transpose()
    mul = thetaT.dot(X)
    H_theta = getHypothesis(mul)
    cost = costFunction(H_theta, Y)
    if((ite % 500) == 0):
            print("Iteration:", ite, "Cost Function value:", cost)

#H(theta) = thetaT * matrix
matrixTest = frecuencyTesting
thetaT = theta.transpose() #Now, this has been trained 1 x m
mT = frecuencyTesting.transpose()
mul = thetaT.dot(mT)
H_theta = getHypothesis(mul)
cost = costFunction(H_theta, YTest)
print("Cost Function with Testing set:", cost)

Ytest = list()
for i in YTest:
    for j in i:
        Ytest.append(j)

print("TESTING SET")
j = 0
for i in range(0, len(matrixTest)):
    prediction = thetaT.dot(matrixTest[i])
    ans = 1 / (1 + math.exp(-1*prediction[0]))
    if((i % 30) == 0):
            print("Prediction:", ans, "Real:", Ytest[j])
    j = j + 1

# Creating confusion matrix
j = 0
true_pos = 0
true_neg = 0
false_pos = 0
false_neg = 0
for i in range(0, len(matrixTest)):
    prediction = thetaT.dot(matrixTest[i])
```

```
    ans = 1 / (1 + math.exp(-1*prediction[0]))
    if ans >= .90:
            ans = 1
    else:
            ans = 0

    if ans == 1 and Ytest[j] == 1:
            true_pos = true_pos + 1
    if ans == 1 and Ytest[j] == 0:
            false_pos = false_pos + 1
    if ans == 0 and Ytest[j] == 1:
            false_neg = false_neg + 1
    if ans == 0 and Ytest[j] == 0:
            true_neg = true_neg + 1
    j = j + 1

print("\n\nConfusion matrix")
print("------------ 1 -------- 0 -----")
print("--- 1 ---- ", true_pos, "-------", false_pos, "-----")
print("--- 0 ----- ", false_neg, "------", true_neg, "----")


P = true_pos / (true_pos + false_pos)
R = true_pos / (true_pos + false_neg)
F1 = 2 * ((P * R) / (P + R))

print("\n\nPrecision:", P)
print("Recall:", R)
print("F1:", F1)
```