# Instituto Politécnico Nacional

## Escuela Superior de Cómputo

# Regresión Logística

## Natural Language Processing

*Estudiante:*
Nicolás Sayago Abigail

*Profesora:*
Olga Kolesnicova

06 Abril, 2020

## RESULTADOS TRAINING SET

Decidí hacer 10000 iteraciones y usar un **learning rate** de 0.4 para ver si se obtenían mejores resultados. Estas son los primeros y últimos resultados de cada 100 iteraciones. Como se puede observar, nuevamente los resultados van en decremento y en las dos primeras iteraciones mostradas se ve un cambio brusco y después un cambio pequeño.

```
TRAINING TEST:
Iteration: 0 Cost Function value: 0.6275865217749537
Iteration: 100 Cost Function value: 0.294388478821439
Iteration: 200 Cost Function value: 0.2915563750158737
Iteration: 300 Cost Function value: 0.28886573203638605
Iteration: 400 Cost Function value: 0.28625019646606364
Iteration: 500 Cost Function value: 0.2837052720178191
Iteration: 600 Cost Function value: 0.28122694608670207
Iteration: 700 Cost Function value: 0.27881156915203725
Iteration: 800 Cost Function value: 0.27645581312754847
Iteration: 900 Cost Function value: 0.2741566350597488
Iteration: 1000 Cost Function value: 0.27191124552954504
Iteration: 1100 Cost Function value: 0.26971708110012466
Iteration: 1200 Cost Function value: 0.2675717802441455
Iteration: 1300 Cost Function value: 0.26547316226664963
Iteration: 1400 Cost Function value: 0.263419208811379
Iteration: 1500 Cost Function value: 0.2614080475987347
Iteration: 1600 Cost Function value: 0.2594379380950245
Iteration: 1700 Cost Function value: 0.2575072588562345
Iteration: 1800 Cost Function value: 0.2556144963265357
Iteration: 1900 Cost Function value: 0.2537582349030994
Iteration: 2000 Cost Function value: 0.251937148105418
Iteration: 2100 Cost Function value: 0.2501499907099557
Iteration: 2200 Cost Function value: 0.2483955917301951
Iteration: 2300 Cost Function value: 0.246672848138541
```

```
Iteration: 7400 Cost Function value: 0.1859311408187227
Iteration: 7500 Cost Function value: 0.18509507637253508
Iteration: 7600 Cost Function value: 0.1842683276390183
Iteration: 7700 Cost Function value: 0.18345073170387544
Iteration: 7800 Cost Function value: 0.18264212948577757
Iteration: 7900 Cost Function value: 0.18184236561765846
Iteration: 8000 Cost Function value: 0.18105128833296358
Iteration: 8100 Cost Function value: 0.18026874935658083
Iteration: 8200 Cost Function value: 0.17949460380019475
Iteration: 8300 Cost Function value: 0.17872871006182733
Iteration: 8400 Cost Function value: 0.17797092972934223
Iteration: 8500 Cost Function value: 0.17722112748770408
Iteration: 8600 Cost Function value: 0.17647917102979996
Iteration: 8700 Cost Function value: 0.17574493097063998
Iteration: 8800 Cost Function value: 0.17501828076476783
Iteration: 8900 Cost Function value: 0.17429909662672213
Iteration: 9000 Cost Function value: 0.17358725745439857
Iteration: 9100 Cost Function value: 0.1728826447551738
Iteration: 9200 Cost Function value: 0.17218514257465864
Iteration: 9300 Cost Function value: 0.17149463742795834
Iteration: 9400 Cost Function value: 0.17081101823332245
Iteration: 9500 Cost Function value: 0.1701341762480761
Iteration: 9600 Cost Function value: 0.16946400500672928
Iteration: 9700 Cost Function value: 0.16880040026116747
Iteration: 9800 Cost Function value: 0.16814325992283194
Iteration: 9900 Cost Function value: 0.16749248400680417
```

### RESULTADOS TESTING SET

Después de haber entrenado, se aplica la hipótesis y se compara con el valor real del conjunto de prueba. Podemos observar que en la mayoría de los casos cuando la predicción es mayor a 0.95, el resultado real es 1. Cuando el resultado real es 0, la mayoría de los valores son menores a 0.90.

```
Cost Function with Testing set: 0.8243822395313952


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
```

## ✓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

#############################################
```

```python
#                                  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/18/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:")
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 % 100) == 0):
            print("Iteration:", ite, "Cost Function value:", cost)

#H(theta) = thetaT * matrix
matrixTest = frecuencyTesting
```

```python
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
```