Web Science: Assignment #8

Alexander Nwala

Puneeth Bikkasandra

Monday, April 30, 2018

Contents

Problem 1

The Training dataset should:

1. consist of 10 text documents for email messages you consider spam (from your spam folder)
2. consist of 10 text documents for email messages you consider not spam (from your inbox)

The Testing dataset should:

1. consist of 10 text documents for email messages you consider spam (from your spam folder)
2. consist of 10 text documents for email messages you consider not spam (from your inbox)

Upload your datasets on github

SOLUTION :

I have solved the problem as described in the below steps :

1. I have scrolled through my email accounts from gmail, to fetch couple of spam messages and non-spam messages.
2. Created two different folders for testing and training.
3. Each of the testing and training folders has 10 independent files for spam and non-spam messages. Totalling a 40 files of spam and non-spam messages.
4. The system needs to be trained in the beginning with the training dataset and later experimented with the testing dataset.

The spam folder of my gmail account had hundreds of spam messages but most of them had pictures in it. "Show Original" option available for every spam message in gmail was not effective in my case. So i have selected the spam mails with text only across 3 of my gmail accounts.

Problem 2

2. Using the PCI book modified docclass.py code and test.py (see Slack assignment-8 channel)
Use your Training dataset to train the Naive Bayes classifier (e.g., docclass.spamTrain())
Use your Testing dataset to test (test.py) the Naive Bayes classifier and report the classification results.

SOLUTION

The below code files from text **Programming Collective Intelligence** has been modified to train and test the dataset.

Listing 1: docclass.py

```
#from pysqlite2 import dbapi2 as sqlite
import sqlite3 as sqlite
import re
import math

def getwords(doc):
  splitter=re.compile(’\\W*’)
  #print(doc)
  # Split the words by non-alpha characters
  words=[s.lower() for s in splitter.split(doc)
          if len(s)>2 and len(s)<20]

  # Return the unique set of words only
  toreturn = dict([(w,1) for w in words])
  return toreturn

class classifier:
  def __init__(self,getfeatures,filename=None):
    # Counts of feature/category combinations
    self.fc={}
    # Counts of documents in each category
    self.cc={}
    self.getfeatures=getfeatures

  def setdb(self,dbfile):
    self.con=sqlite.connect(dbfile)
    self.con.execute(’create table if not exists fc(feature,category,count)’)
    self.con.execute(’create table if not exists cc(category,count)’)


  def incf(self,f,cat):
    count=self.fcount(f,cat)
    if count==0:
      self.con.execute("insert into fc values (’%s’,’%s’,1)"
                       % (f,cat))
    else:
      self.con.execute(
        "update fc set count=%d where feature=’%s’ and category=’%s’"
        % (count+1,f,cat))

  def fcount(self,f,cat):
```

Problem 2 continued on next page. . .

```
    res=self.con.execute(
      'select count from fc where feature="%s" and category="%s"'
      %(f,cat)).fetchone()
    if res==None: return 0
    else: return float(res[0])

  def incc(self,cat):
    count=self.catcount(cat)
    if count==0:
      self.con.execute("insert into cc values ('%s',1)" % (cat))
    else:
      self.con.execute("update cc set count=%d where category='%s'"
                       % (count+1,cat))

  def catcount(self,cat):
    res=self.con.execute('select count from cc where category="%s"'
                         %(cat)).fetchone()
    if res==None: return 0
    else: return float(res[0])

  def categories(self):
    cur=self.con.execute('select category from cc');
    return [d[0] for d in cur]

  def totalcount(self):
    res=self.con.execute('select sum(count) from cc').fetchone();
    if res==None: return 0
    return res[0]


  def train(self,item,cat):
    features=self.getfeatures(item)
    # Increment the count for every feature with this category
    for f in features:
      self.incf(f,cat)

    # Increment the count for this category
    self.incc(cat)
    self.con.commit()

  def fprob(self,f,cat):
    if self.catcount(cat)==0: return 0

    # The total number of times this feature appeared in this
    # category divided by the total number of items in this category
    return self.fcount(f,cat)/self.catcount(cat)

  def weightedprob(self,f,cat,prf,weight=1.0,ap=0.5):
    # Calculate current probability
    basicprob=prf(f,cat)

    # Count the number of times this feature has appeared in
    # all categories
    totals=sum([self.fcount(f,c) for c in self.categories()])

    # Calculate the weighted average
    bp=((weight*ap)+(totals*basicprob))/(weight+totals)
    return bp




class naivebayes(classifier):

  def __init__(self,getfeatures):
    classifier.__init__(self,getfeatures)
    self.thresholds={}

  def docprob(self,item,cat):
    features=self.getfeatures(item)

    # Multiply the probabilities of all the features together
    p=1
    for f in features: p*=self.weightedprob(f,cat,self.fprob)
    return p

  def prob(self,item,cat):
    catprob=self.catcount(cat)/self.totalcount()
    docprob=self.docprob(item,cat)
    return docprob*catprob

  def setthreshold(self,cat,t):
    self.thresholds[cat]=t

  def getthreshold(self,cat):
    if cat not in self.thresholds: return 1.0
    return self.thresholds[cat]

  def classify(self,item,default=None):
    probs={}
    # Find the category with the highest probability
    max=0.0
    for cat in self.categories():
      probs[cat]=self.prob(item,cat)
      if probs[cat]>max:
        max=probs[cat]
        best=cat

    # Make sure the probability exceeds threshold*next best
    for cat in probs:
      if cat==best: continue
      if probs[cat]*self.getthreshold(best)>probs[best]: return default
    return best

class fisherclassifier(classifier):
  def cprob(self,f,cat):
    # The frequency of this feature in this category
    clf=self.fprob(f,cat)
    if clf==0: return 0

    # The frequency of this feature in all the categories
    freqsum=sum([self.fprob(f,c) for c in self.categories()])

    # The probability is the frequency in this category divided by
    # the overall frequency
    p=clf/(freqsum)

    return p
  def fisherprob(self,item,cat):
    # Multiply all the probabilities together
    p=1
    features=self.getfeatures(item)
    for f in features:
      p*=(self.weightedprob(f,cat,self.cprob))

    # Take the natural log and multiply by -2
    fscore=-2*math.log(p)

    # Use the inverse chi2 function to get a probability
    return self.invchi2(fscore,len(features)*2)
  def invchi2(self,chi, df):
    m = chi / 2.0
    sum = term = math.exp(-m)
    for i in range(1, df//2):
        term *= m / i
        sum += term
    return min(sum, 1.0)
  def __init__(self,getfeatures):
    classifier.__init__(self,getfeatures)
    self.minimums={}

  def setminimum(self,cat,min):
    self.minimums[cat]=min

  def getminimum(self,cat):
    if cat not in self.minimums: return 0
    return self.minimums[cat]
  def classify(self,item,default=None):
    # Loop through looking for the best result
    best=default
    max=0.0
    for c in self.categories():
      p=self.fisherprob(item,c)
      # Make sure it exceeds its minimum
      if p>self.getminimum(c) and p>max:
        best=c
        max=p
    return best

```

```

def sampletrain(cl):
  cl.train(’Nobody owns the water.’,’good’)
  cl.train(’the quick rabbit jumps fences’,’good’)
  cl.train(’buy pharmaceuticals now’,’bad’)
  cl.train(’make quick money at the online casino’,’bad’)
  cl.train(’the quick brown fox jumps’,’good’)

def spamTrain(cl):
  cl.train(’the the’, ’not spam’)
  cl.train(’cheap cheap cheap banking the’, ’spam’)
  cl.train(’the’, ’not spam’)
  cl.train(’cheap cheap banking banking banking the the’, ’spam’)
  cl.train(’cheap cheap cheap cheap cheap buy buy the’, ’spam’)
  cl.train(’banking the’, ’not spam’)
  cl.train(’buy banking the’, ’not spam’)
  cl.train(’the’, ’not spam’)
  cl.train(’the’, ’not spam’)
  cl.train(’cheap buy dinner the the’, ’not spam’)

def testEmail(cl):

    # training for non-spam
    for i in range(1,11):
        filename = ’Training/notspam’ + str(i) +’.txt’
        with open(filename, ’r’) as nonspam:
            cl.train(nonspam.read(), ’not spam’)

    # training for spam
    for i in range(1,11):
        filename = ’Training/spam’ + str(i) +’.txt’
        with open(filename, ’r’) as spam:
            cl.train(spam.read(), ’spam’)
```

Listing 2: test.py

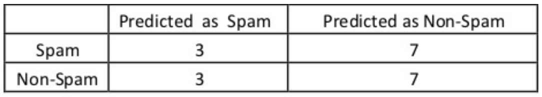
```
import docclass
from subprocess import check_output

import numpy as np



def compareSample(file, pred):

    with open(file, ’r’) as filename:
        result = cl.classify(filename.read())
        if result == ’spam’:
            pred.append(1)
        else:
            pred.append(0)

def emailTest(cl):
    outcome = []
    standard = np.array([1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0])

    # testing spam
    try:
        for i in range(1, 11):
            filename = ’Testing/spam’ + str(i) + ’.txt’

            compareSample(filename, outcome)

    # testing non-spam
        for i in range(1, 11):
            filename = ’Testing/notspam’ + str(i) + ’.txt’
            compareSample(filename, outcome)

    except:
        print (filename)

    print (’STANDARD is:’)
    print(standard)
    outcome = np.array(outcome)

    print (’OUTCOME  after Comparison is:’)
    print(outcome)

    truePositive = len(np.where(outcome[np.where(standard == 1)] == 1)[0])
    trueNegative = len(np.where(outcome[np.where(standard == 0)] == 0)[0])
    falsePositive = len(np.where(standard == 1)[0]) - truePositive
    falseNegative = len(np.where(standard == 1)[0]) - trueNegative

    confusionMatrix = [[truePositive, falsePositive], [falseNegative, trueNegative]]
    print(’CONFUSION MATRIX is :’)
    print (confusionMatrix)

    precision = float(truePositive) / (truePositive + trueNegative)
    accuracy = float(truePositive + falsePositive)/(truePositive + trueNegative +
    falsePositive + falseNegative)
    print(’PRECISION’,float(precision))
    print(’ACCURACY’,float(accuracy))


cl = docclass.naivebayes(docclass.getwords)
check_output([’rm’, ’spamCheck.db’])
cl.setdb(’spamCheck.db’)
docclass.testEmail(cl)
emailTest(cl)
```

Problem 3

3. Draw a confusion matrix for your classification results
(see: https://en.wikipedia.org/wiki/Confusion_matrix)

SOLUTION

With above generated dataset and the below formulae, i have arrived at the below confusion matrix. Where 7 out of 10 spam email messages were predicted incorrectly as non-spam and only 3 of them were predicted as spam.



	Predicted as Spam	Predicted as Non-Spam
Spam	3	7
Non-Spam	3	7

Figure 1: Confusion Matrix

Problem 3 continued on next page. . .

Problem 4

4. Report the precision and accuracy scores of your classification results
(see: https://en.wikipedia.org/wiki/Precision_and_recall)

SOLUTION

To calculate the Precision and Accuracy for my dataset, i have made use of the below formulae from the suggested wikipedia page.

$$\text{Precision} = \frac{tp}{tp + fp}$$

Figure 2: Precision Formula

Applying the parameters from the previous computation,i have arrived at below Precision

```
Precision = 0.3
```

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

Figure 3: Accuracy Formula

Applying the parameters from the previous computation,i have arrived at below Accuracy

```
Accuracy = 0.5
```

References

1. https://en.wikipedia.org/wiki/Confusion_matrix

2. https://en.wikipedia.org/wiki/Precision_and_recall)