Web Science: Assignment #8

Alexander Nwala

Apurva Modi

Tuesday, April 30, 2019

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 :
Steps that were taken are mentioned below.

1. I opened Spam folder and promotion folders for all of my emails
2. Created two different folders for Spam and not a Spam.
3. For each of the folder I added 20 email's data into them.
4. Finally I separated them equally into testing and training dataset

There were mostly images as an email into my spam folder so I took source code for those emails.

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

```
import sqlite3 as sqlite
import re
import math
import io

def getwords(doc):
  splitter=re.compile('\\W*')

  # 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
  return dict([(w,1) for w in words])

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):
    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 eTrain(cl):
```

```
    # train on spam
  for i in range(1,11):
    filename = 'Dataset/Training/spam' + str(i) +'.txt'

    with io.open(filename, 'r', encoding='utf-8') as trainFile:
      cl.train(trainFile.read(), 'spam')


    # train on non spam
  for i in range(1,11):
    filename = 'Dataset/Training/notaspam' + str(i) +'.txt'

    with io.open(filename, 'r', encoding='utf-8') as trainFile1:
      cl.train(trainFile1.read(), 'not spam')
```

Listing 2: test.py

```
import docclass
from subprocess import check_output
import io


cl = docclass.naivebayes(docclass.getwords)
#remove previous db file

check_output(['rm', 'apurv.db'])


cl.setdb('apurv.db')
docclass.eTrain(cl)

print("*** Testing for SPAM ***")

for i in range(1,11):
          filename = 'Dataset/Testing/spam' + str(i) +'.txt'

          with io.open(filename, 'r', encoding='utf-8') as testFile:
               print(filename, cl.classify(testFile.read()))

print("*** Testing for NOT A SPAM ***")
for i in range(1,11):
          filename = 'Dataset/Testing/notaspam' + str(i) +'.txt'

          with io.open(filename, 'r', encoding='utf-8') as testFile1:
               print(filename, cl.classify(testFile1.read()))
```

```
[(py2) RocketScientist:A8 apurvamodi$ python test.py
*** Testing for SPAM ***
('Dataset/Testing/spam1.txt', u'spam')
('Dataset/Testing/spam2.txt', u'spam')
('Dataset/Testing/spam3.txt', u'not spam')
('Dataset/Testing/spam4.txt', u'not spam')
('Dataset/Testing/spam5.txt', u'spam')
('Dataset/Testing/spam6.txt', u'not spam')
('Dataset/Testing/spam7.txt', u'spam')
('Dataset/Testing/spam8.txt', u'not spam')
('Dataset/Testing/spam9.txt', u'not spam')
('Dataset/Testing/spam10.txt', u'not spam')
*** Testing for NOT A SPAM ***
('Dataset/Testing/notaspam1.txt', u'not spam')
('Dataset/Testing/notaspam2.txt', u'not spam')
('Dataset/Testing/notaspam3.txt', u'not spam')
('Dataset/Testing/notaspam4.txt', u'not spam')
('Dataset/Testing/notaspam5.txt', u'not spam')
('Dataset/Testing/notaspam6.txt', u'not spam')
('Dataset/Testing/notaspam7.txt', u'spam')
('Dataset/Testing/notaspam8.txt', u'not spam')
('Dataset/Testing/notaspam9.txt', u'not spam')
('Dataset/Testing/notaspam10.txt', u'not spam')
(py2) RocketScientist:A8 apurvamodi$
```

Figure 1: Output stating Spam or Not a Spam

Problem 3

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

SOLUTION

From my dataset I created a confusion matrix in which each row of the matrix represents the instances in a predicted class(Spam or Not a Spam) while each column represents the instances in an actual class(Spam or Not a Spam).

Confusion Matrix	Predicted Spam	Predicted Not a Spam
Spam	4	6
Not a Spam	1	9

Figure 2: 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

As suggested I made use of the following formulas to calculate Precision and Accuracy.

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

Figure 3: Precision Formula

Assigning values to True Positive and False Positive from the confusion matrix.

```
Precision = 0.4
```

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

Figure 4: Accuracy Formula

Assigning values to True Positive, True Negative , False Positive and False Negative from the confusion matrix.

```
Accuracy = 0.65
```

Problem 4 continued on next page. . .

References

1. https://en.wikipedia.org/wiki/Confusion_matrix
2. https://en.wikipedia.org/wiki/Precision_and_recall
3. https://github.com/uolter/PCI