Machine Learning
Project 1

Documentation and Research

By
Joel Satkauskas
R00116315

Project Description:

Building the model, I began in the main method by iterating through every review in both positive and negative reviews. It reads the content from the review as a string and proceeds to perform preprocessing on the string in an attempt to improve the accuracy of the model.

```
for eachFile in posListing:
    PosRecords+=1
    PosFile = open(posPath+eachFile, "r")
    PosContent = PosFile.read().decode('utf8')
    PosContent = PosContent.lower()
    PosContent = replaceAcronym(PosContent)
    PosContent = re.sub(r''+regexString, '', PosContent)
    PosContent = removeStopWord(PosContent)
    PosContent = ps.stem(PosContent)
    AllPosWords += (PosContent.split())

print 'All pos words =',len(AllPosWords)

for eachFile in negListing:
    NegRecords+=1
    NegFile = open(negPath+eachFile, "r")
    NegContent = NegFile.read().decode('utf8')
    NegContent  = NegContent.lower()
    NegContent = replaceAcronym(NegContent)
    NegContent  = re.sub(r''+regexString, '', NegContent)
    NegContent = removeStopWord(NegContent)
    NegContent = ps.stem(NegContent)
    AllNegWords += (NegContent.split())

print 'All neg words =',len(AllNegWords)
```

Once it has finished preprocessing, it adds the string onto a list. This way by the end of the records, there are 2 lists, one with every single word from all positive reviews and one for negative reviews.

To get the vocabulary, I cast these lists into sets. I then create a union of the sets.

```
vobalularyPos = set(AllPosWords)
vobalularyNeg = set(AllNegWords)
vobalulary = vobalularyPos | vobalularyNeg
```

It then creates 2 dictionaries that will store the frequency of each word in positive and negative reviews.

```
"""
Get the frequency of each word in positive and negative reviews.
"""
posDict = {}
negDict = {}

for word in AllPosWords:
    if word in posDict:
        posDict[word] = posDict[word]+1
    else:
        posDict[word] = 1

for word in AllNegWords:
    if word in negDict:
        negDict[word] = negDict[word]+1
    else:
        negDict[word] = 1
```

I then make sure that both dictionaries have the same vocabulary, making the model more accurate.

```
"""
Make sure that both dictionaries have the same words.
"""
for key in posDict:
    if key not in negDict:
        negDict[key] = 0

for key in negDict:
    if key not in posDict:
        posDict[key] = 0
```

At this point, it has everything it needs to use the multinomial Naive Bayes formula.

- $P(w \mid c) = \frac{count(w,c)+1}{count(c)+|V|}$

count(w, c) is the number of occurrences of the word w in all documents of class c.

count(c) The total number of words in all documents of class c (including duplicates).

|V| The number of words in the vocabulary

It create 2 new dictionaries and stores the probability of each word for positive and negative respectfully.

```
ProbPosDict = {}
ProbNegDict = {}

for word in posDict:
    ProbPosDict[word] = ((posDict[word]+1.0)/((len(AllPosWords)+len(vobalulary))))

for word in negDict:
    ProbNegDict[word] = ((negDict[word]+1.0)/((len(AllNegWords)+len(vobalulary))))
```

From here, it's using these dictionaries to test our test data and see how accurate the model is.

$$P(c \mid W) = \log P(c) + \sum_{w \in W} \log P(w \mid c)$$

For all positive and negative reviews, it iterates through each one and applies the same preprocessing algorithmes it used when making the model. It will then use the above formula to get the probability of negative or positive.

```
for eachFile in posTestListing:
    PosTestRecords+=1
    PosTestFile = open(testPosPath+eachFile, "r")
    PosTestContent = PosTestFile.read().decode('utf8')
    PosTestContent = PosTestContent.lower()
    PosTestContent = replaceAcronym(PosTestContent)
    PosTestContent = re.sub(r''+regexString, '', PosTestContent)
    PosTestContent = removeStopWord(PosTestContent)
    PosTestContent = ps.stem(PosTestContent)
    AllTestPosWords = (PosTestContent.split())

    testPosTotal = math.log(PPos)
    testNegTotal = math.log(PNeg)

    for word in AllTestPosWords:
        if word in posDict:
            testPosTotal += math.log(ProbPosDict[word])
        if word in negDict:
            testNegTotal += math.log(ProbNegDict[word])

    if testPosTotal > testNegTotal:
        TestAmountOfPosRight+=1

print 'Total pos file = ', PosTestRecords
print 'Files predicted to be positive = ',TestAmountOfPosRight
print 'Accuracy of model = ', (TestAmountOfPosRight*100.0/PosTestRecords)
```

Whichever is higher, has a better chance of being the class. Do this for all records and keep track of the amount identified to be of that class. Then display the accuracy of the model.

```
((TestAmountOfNegRight+TestAmountOfPosRight)*100.0)/(PosTestRecords+NegTestRecords)
```

Using the larger data set with no preprocessing I'm getting a 82.2% accuracy for my model.

```
Total pos file =  1000
Files predicted to be positive =  784
Accuracy of model =  78.4

Total neg file =  1000
Files predicted to be negative =  860
Accuracy of model =  86.0

Average accurecy =  82.2
```

Research:

For the research into improving the accuracy of the model, I have tried using lower casing and regular expression, removing stop words, stemming and acronym replacement.

1. **Data Cleaning(Lowercase and regular expression.)**

Lower Casing:
A very simple technique, I tried to lower the case of all words every time it took in a record. The theory being that, instead of having 2 sets of the same words, it would have one set with the added value of both, eg, instead of having “Bad” = 250, “bad” = 250, it would have “bad” = 500.

Initially this improves the accuracy by 0.15%. Additionally, testing using it with other algorithms i've found that it improves their accuracy a lot more instead of not lower casing anything.

```
Total pos file =  1000
Files predicted to be positive =  784
Accuracy of model =  78.4

Total neg file =  1000
Files predicted to be negative =  863
Accuracy of model =  86.3

Average accurecy =  82.35
```

Regular expression:
With regular expression, I used a pattern that would match anything that wasn't a word, space or comma (‘)

```
regexString = '[^a-zA-z\'\s]'
```

I then made it subtract whatever it matched from the string for both negative and positive and for the model and test records.

```
for eachFile in posListing:
    PosRecords+=1
    PosFile = open(posPath+eachFile, "r")
    PosContent = PosFile.read()#.decode('utf8')
    #PosContent = PosContent.lower()
    #PosContent = replaceAcronym(PosContent)
    PosContent = re.sub(r''+regexString, '', PosContent)
    #PosContent = removeStopWord(PosContent)
    #PosContent = ps.stem(PosContent)
    AllPosWords += (PosContent.split())
```

The theory behind this is similar to the lower casing. I thought that instead of having many occurences of the same word but slightly differently, eg “-bad”, “bad,”, “/bad”, clean up the punctuation so that all these words get caught as one and therefore increase the weight of the word in the model and use it properly in the test.

Using this without lowercasing it actually drops the accuracy of the model by 1.5%.

```
Total pos file =  1000
Files predicted to be positive =  768
Accuracy of model =  76.8

Total neg file =  1000
Files predicted to be negative =  846
Accuracy of model =  84.6

Average accurecy =  80.7
```

I've found that it always bring the accuracy down when used with the other algorithms so I won't be using it when mentioning the other algorithms.

2. Stop Words

The theory behind this is that certain words add no weight to the classification of documents. Words used to construct and link sentences such as “the”, “and”, “or”, “are”, ect. Removing these words from the vocabulary would increase the weight of other words that actually add to the classification and therefore increase the accuracy of the model.

I implement this approach by importing a file of 666 stop words and creating a method that takes in a string and removes these words from the string.

```
stopWordPath = 'C:\\Users\\Admin\\Desktop\\stopWords.txt'
stopWordFile = open(stopWordPath)
stopWordsStr = stopWordFile.read()
stopWordsList = stopWordsStr.split()
def removeStopWord(contentString):
    for word in stopWordsList:
        if word in contentString:
            contentString = contentString.replace(' '+word+' ', ' ')

    return contentString
```

Without lowering the case, it improves the accuracy by 0.7%

```
Total pos file =  1000
Files predicted to be positive =  803
Accuracy of model =  80.3

Total neg file =  1000
Files predicted to be negative =  859
Accuracy of model =  85.9

Average accurecy =  83.1
```

And lowering the case adds a further 0.8% onto that (increases stop word improvement by 0.65% since lowercaseing by itself adds 0.15%).

```
Total pos file =  1000
Files predicted to be positive =  807
Accuracy of model =  80.7

Total neg file =  1000
Files predicted to be negative =  871
Accuracy of model =  87.1

Average accurecy =  83.9
```

3. Stemming

Stemming has a similar intention as the above methods, it will try to group similar words together as one and therefore increase the weight of words in the model and their frequency in the test content.

For this I imported and used PortersStemmer. Passing it a string will return the same string but with the words stemmed, eg, ‘Fished’, ‘Fishing’, ‘Fisher’ would return as ‘Fish’,‘Fish’,‘Fish’.

So if there were 2 sentences, such as:
“I hated that movie” and “I hate this movie”, Stemming would group them as one as they mean the same thing. This increase the weight of “hate” in the model and its frequency in the test data.

```
for eachFile in posListing:
    PosRecords+=1
    PosFile = open(posPath+eachFile, "r")
    PosContent = PosFile.read().decode('utf8')

    #(1) Lower case and Regular expression
    #PosContent = PosContent.lower()
    #PosContent = re.sub(r''+regexString, '', PosContent)
    #/(1) Lower case and Regular expression

    #(4) Replace Acronyms
    #PosContent = replaceAcronym(PosContent)
    #/(4) Replace Acronyms

    #(2) Remove Stop Words
    #PosContent = removeStopWord(PosContent)
    #/(2) Remove Stop Words

    #(3) Stem word
    PosContent = ps.stem(PosContent)
    #/(3) Stem word

    AllPosWords += (PosContent.split())

print 'All pos words =',len(AllPosWords)
```

Surprisingly, with no other preprocessing, this results to a loss of 0.1%.

```
Total pos file =  1000
Files predicted to be positive =  783
Accuracy of model =  78.3

Total neg file =  1000
Files predicted to be negative =  859
Accuracy of model =  85.9

Average accurecy =  82.1
```

But with lowercasing and removing stop words before stemming, it improves the accuracy by 0.1%. (On top of the accuracy of lowercasing and removing stop words)

```
Total pos file =  1000
Files predicted to be positive =  810
Accuracy of model =  81.0

Total neg file =  1000
Files predicted to be negative =  870
Accuracy of model =  87.0

Average accurecy =  84.0
```

4. **Acronym**

Since these these movie reviews were created and shared over the internet, we can presume that a certain amount of acronyms were used as its common in IOT communication and culture.

The idea behind this preprocessing method is to find any acronyms and replace it with the full text of what it means. Eg, “lol” would be replaced with “lots of laughs”. So instead of acronyms being a feature, they’re full meaning would be added to the frequency of other words.

I did this by importing a file with 124 acronyms and their full meaning. I read this file into a dictionary and made a method that would check if a string has this acronym. If so, it would replace with the text.

```
acroymPath = 'C:\\Users\\Admin\\Desktop\\acr.txt'
acroymDict = {}
with open(acroymPath) as f:
    for line in f:
        splitLine = line.split()
        acroymDict[(splitLine[0])] = ",".join(splitLine[1:])


def removeStopWord(contentString):
    for word in stopWordsList:
        if word in contentString:
            contentString = contentString.replace(' '+word+' ', ' ')

    return contentString
```

```
for eachFile in posListing:
    PosRecords+=1
    PosFile = open(posPath+eachFile, "r")
    PosContent = PosFile.read().decode('utf8')

    #(1) Lower case and Regular expression
    PosContent = PosContent.lower()
    #PosContent = re.sub(r''+regexString, '', PosContent)
    #/(1) Lower case and Regular expression

    #(4) Replace Acronyms
    PosContent = replaceAcronym(PosContent)
    PosContent = PosContent.replace('_', ' ')
    #/(4) Replace Acronyms

    #(2) Remove Stop Words
    #PosContent = removeStopWord(PosContent)
    #/(2) Remove Stop Words

    #(3) Stem word
    PosContent = ps.stem(PosContent)
    #/(3) Stem word

    AllPosWords += (PosContent.split())

print 'All pos words =',len(AllPosWords)
```

With lowercasing, this actually decreases the accuracy by 0.2%.

```
Total pos file =  1000
Files predicted to be positive =  785
Accuracy of model =  78.5

Total neg file =  1000
Files predicted to be negative =  858
Accuracy of model =  85.8

Average accurecy =  82.15
```

And with stop word removal and stemming it reduces the accuracy by 0.15%.

```
Total pos file =  1000
Files predicted to be positive =  806
Accuracy of model =  80.6

Total neg file =  1000
Files predicted to be negative =  871
Accuracy of model =  87.1

Average accurecy =  83.85
```

Conclusion:

In conclusion, I increased the accuracy of the model by 1.8% by using lowercaseing, stop word removal and stemming, up to 84% accuracy.

Regular expression and acronyms replacement seemed to of lowered the accuracy. I'm not sure why regular expression lowered the accuracy but I can see why acronym replacement might.

Certain acronyms might be used that give the opposite effect on the class. For example in a negative review, there might be the acronym “bff” (best friend forever). The method would change it to that line and the frequency calculation would take the word ‘best’ in and increase its weight resulting to a lower accuracy for negative reviews.

The method that increased the accuracy more than others were the combination of lower casing and stop words removal which raised the accuracy by 1.55%.