CS4025/CS5057: Continuous Assessment 1
Classifying Emotions

Philip Hale

November 25, 2013

Contents

1 NGram models **1**
1.1 Six-way Classification. 1
1.1.1 Ruby 1
1.1.2 Python 2
1.2 Combining unigram, bigram and trigram models 2
1.3 System performance 3

2 Classification with WEKA **3**
2.1 Creating the ARFF file 3
2.2 System Performance 4

Appendices **4**

A Python Source **4**

B Functionality of Ruby classifier **8**

C Simple Bayesian Classifier **9**

1 NGram models

1.1 Six-way Classification.

1.1.1 Ruby

My first instinct was to classify the emotions using an existing Bayesian classification library, to see how this problem has been approached in the past. Using the language I'm most familiar with, I wrote `bayesian_classifier.rb` to perform sentiment analysis on two of the emotions files.

Having confirmed that a simple Bayesian classifier is able to produce reasonable results, I moved on to producing my own system.

The first attempt used Ruby's support for functional and object-oriented styles to create an alternative implementation from the `Sentiment.py` I had worked with previously. Typically, functional and OOP styles of programming give you more maintainable and testable code at the expense of speed and simplicity. This program is split into various classes and is supported by comprehensive unit tests.

A DataSet class is used to abstract concerns such as reading from disk, combining the emotions files and partitioning the data into different sets. The held-back set of data is split in two (dev- and test-sets) to allow fine-tuning of the algorithm without introducing bias.

Probabilities and word counts given specific sentiments are generated in a more functional approach, and the results are memoized to reduce running-time. For example, `data_set.word_count(word: 'help', sentiment: :fearful)` will give the number of times 'help' occurs in a fearful context.

1.1.2 Python

Unfortunately, two major problems led me to abandon this approach and start again in Python. Firstly, the system had trouble consistently classifying the dataset in under 10 seconds, slowing down development. More importantly, it was unable to classify to a satisfactory accuracy. This impeded further development, since without a reliable unigram classifier the rest of the objectives would be hard to meet.

I cut my losses and started afresh using the Sentiment.py script from the practicals.

My approach this time aimed to deviate as little as possible from code which I knew could classify correctly. To this end, I replicated the existing code to support additional sentiments, almost doubling LOC count. At this point, only unigrams were supported.

Some changes had to be made other than just adding extra files. The threshold above which a classification is determined to be correct was increased to 0.16. In fact, both the threshold and the initially assigned probability for each sentiment can be given a starting value of $\frac{1}{6}$ since the classification task involves six sentiments.

In order to aid future development and understanding of the script, I reduced much of the duplication by nesting the various data structures in dictionaries, where the keys are the names of the sentiments. This greatly reduces the amount of repetition in the code and means adding or removing sentiments only requires changing code in one place. For example, to extend the classifier to support a seventh emotion, you would only need to add the datafile and its name to the `SENTIMENTS()` function.

1.2 Combining unigram, bigram and trigram models

I was interested in adopting some of the functional patterns from my initial ruby classifier, and so researched functional methods for producing NGrams. Python doesn't quite have the simplicity of `Enumerable#each_cons` for producing NGrams, but a similar result could be achieved with `zip()`.

A few extra lines of code were written to add bigrams and trigrams including the beginning-of-sentence and end-of-sentence tags.

The models were combined by summing the different order NGrams into a single list of 'words'. This has the effect of approximately tripling the number of words in the dictionary. Since the rest of the code is setup to calculate based on the size of this overall list, the calculations still work out.

1.3 System performance

The system is limited by the unadjusted plus-one smoothing approach used for probability discounting. Applying a better discounting algorithm such as Good Turing discounting, in combination with a better combination of NGrams such as Katz Backoff, would have resulted in a higher overall level of accuracy. However, combining the NGrams resulted in a higher overall level of accuracy than any individual NGram regardless of order, suggesting that even a simple approach to combining NGram models yields positive results

Here are the results of running the algorithm on both the training and test set of data:

Listing 1: Training Set

```
SENTIMENT   ACCURACY  (CORRECT/TOTAL)
ALL         0.35      (749/2161)
angry       0.99      (144/145)
disgusted   1.00      (232/232)
fearful     1.00      (154/154)
happy       0.00      (3/834)
sad         0.03      (12/435)
surprised   0.57      (204/361)
```

Listing 2: Test Set

```
SENTIMENT   ACCURACY  (CORRECT/TOTAL)
ALL         0.25      (59/237)
angry       0.93      (14/15)
disgusted   0.43      (13/30)
fearful     0.77      (23/30)
happy       0.02      (2/91)
sad         0.07      (2/28)
surprised   0.12      (5/43)
```

2 Classification with WEKA

2.1 Creating the ARFF file

The ARFF file is generated in a separate subroutine to be run after the classification task. It requires pre-processing the words (strictly speaking NGrams) which are most common in the training data, as a proportion of the sentiments they are found in. This function, `mostUseful()`, takes as arguments the pWord values generated in the training step, and the number of 'top' results to return. Higher numbers give a larger number of features.

A separate function is given to write the actual ARFF file, using this mostUseful data structure. Since computing the data lines for the ARFF file was going to require generating NGrams again, the code for turning a sentence into a large set of uni-, bi- and tri-grams was extracted into a separate method.

Writing the ARFF file takes place in two stages. First, the list of attributes is generated and written. These are the xy most 'influential' NGrams in the data set, where x is the specified number of predictors and y the number of sentiments. In order to produce a total of 600 attributes for a 6-way classification task, x is set to 100. The attribute names are surrounded in quotes to allow for punctuation.

Next, the rows of data are written. This requires iterating through the dataset and ascertaining if each of the most useful NGrams occur in that sentence. There is a 'yes' for every useful NGram which appears in the sentence. Finally, the known sentiment is appended to the end of each line, allowing us to perform cross-validation in WEKA.

2.2 System Performance

With a starting size of 120 attributes, the ID3 classifier was only able to correctly classify 49% of all instances in the training set. Increasing the attributes count to 600 resulted in an increased successful classification rate of 54%. Critically though, this increased the time taken to build the model in WEKA by more than an order of magnitude, from less than a second to over twenty. The time taken to generate the ARFF file is negligible in either case.

WEKA also provides an open-source version of a better decision-tree algorithm than ID3, namely C4.5 (J48). However, despite taking even longer to run, it is unable to improve on the 54% accuracy given by ID3.

Since the ARFF file is built from data generated by the trainBayes() function, it inherits some of its weaknesses. Improving the manner in which NGrams are combined and discounted would likely improve on these results.

Appendices

A Python Source

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import re, random, math, collections, itertools, pdb, operator

# Gives the emotions we are interested in classifying.  To classify an
# additional emotion, add its name here and an accompanying data file with some
# training data.
def SENTIMENTS():
    return ['angry', 'disgusted', 'fearful', 'happy', 'sad', 'surprised']

# The Bayes classifier gives a number between 0 and 1, expressing the
# confidence that the sentence is of a certain sentiment. The threshold
# determines the required certainty for a sentence to be classified as that
# sentiment.
def THRESHOLD():
    return 1.0 / float(len(SENTIMENTS()))

#  Adapted from Scott Triglia. Elegant N-gram Generation in Python. (2013-01-20).
#  URL: http://locallyoptimal.com/blog/2013/01/20/elegant-n-gram-generation-in-python/.
#  Accessed: 2013-11-21.
#  Archived by WebCite at http://www.webcitation.org/6LImC39zN
def makeNgram(wordList, order):
    ngramList = []
    for ngram in zip(*[wordList[i:] for i in range(order)]):
        ngramList.append('_'.join(ngram))
    return ngramList

def makeNGramList(sentence):
```

```
    wordList = re.findall(r"[\w']+", sentence)#collect all words

    unigramList = []
    bigramList = []
    trigramList = []

    unigramList = wordList

    if len(wordList) > 1:
        bigramList = ["<sen>_"+wordList[0]]
    bigramList = bigramList + makeNgram(wordList, 2)
    if len(wordList) > 1:
        bigramList = bigramList + [wordList[-1]+"_</sen>"]

    if len(wordList) > 2:
        trigramList = trigramList + ["<sen>_<sen>_"+wordList[0]]
    if len(wordList) > 1:
        trigramList = trigramList + ["<sen>_"+wordList[0]+"_"+ wordList[1]]
    trigramList = trigramList + makeNgram(wordList, 3)
    if len(wordList) > 2:
        trigramList = trigramList + [wordList[-2]+"_"+wordList[-1]+"_</sen>"]
    if len(wordList) > 1:
        trigramList = trigramList + [wordList[-1]+"_</sen>_</sen>"]

    return (unigramList + bigramList + trigramList)

def readFiles(sentencesTrain,sentencesTest):
    for sentiment in SENTIMENTS():
        txt = open('emotions/' + sentiment + '.txt')
        for sentence in re.split(r'\n', txt.read()):
            if random.randint(1,10)<2:
                sentencesTest[sentence] = sentiment
            else:
                sentencesTrain[sentence] = sentiment

# Calculate p(W|Positive), p(W|Negative), p(W) for all words in training data.
def trainBayes(sentencesTrain, pWord, freq):
    for sentiment in SENTIMENTS():
        freq[sentiment] = {}
    dictionary = {}
    wordTotals = { 'all': 0 }
    for sentiment in SENTIMENTS():
        wordTotals[sentiment] = 0

    #iterate through each sentence/sentiment pair in the training data
    for sentence, sentiment in sentencesTrain.iteritems():
        if sentence == '':
            continue
        for word in (makeNGramList(sentence)):
            wordTotals['all'] += 1
            if not dictionary.has_key(word):
                dictionary[word] = 1
            wordTotals[sentiment] += 1
            if not freq[sentiment].has_key(word):
                freq[sentiment][word] = 1
            else:
                freq[sentiment][word] += 1

    # smoothing so min count of each word is 1
    for word in dictionary:
        for sentiment in SENTIMENTS():
            if not freq[sentiment].has_key(word):
```

```
                freq[sentiment][word] = 1
            else:
                freq[sentiment][word] += 1

        # divisor for the p(word) calculation
        freqWordAll = 0
        for sentiment in SENTIMENTS():
            freqWordAll += freq[sentiment][word]

        # p(word|sentiment)
        for sentiment in SENTIMENTS():
            pWord[sentiment][word] = \
                freq[sentiment][word] / float(wordTotals[sentiment])

        #p(word)
        pWord['all'][word] = freqWordAll / float(wordTotals['all'])


#INPUTS:
#  sentences is a dictonary of { sentence: sentiment } for every sentence.
#  pWord is dictionary storing p(word) and p(word|sentiment)
def testBayes(sentences, pWord):
    total = {}
    correct = {}
    total['all'] = 0
    correct['all'] = 0
    for sentiment in SENTIMENTS():
        total[sentiment] = 0
        correct[sentiment] = 0

    #for each sentence, sentiment pair in the dataset
    for sentence, sentiment in sentences.iteritems():
        if sentence == '':
            continue
        p = {}
        for s in SENTIMENTS():
            p[s] = THRESHOLD()

        for word in (makeNGramList(sentence)):
            if pWord['all'].has_key(word):
                for s in SENTIMENTS():
                    if pWord[s][word] > 0.00000001:
                        p[s] *= pWord[s][word] * 10000

        total['all'] += 1
        total[sentiment] += 1

        totalProb = float(sum(p.itervalues()))
        prob = 0
        prob = p[sentiment] / totalProb
        if prob >= THRESHOLD():
            correct['all'] += 1
            correct[sentiment] += 1

    accuracy = {}
    accuracy['all'] = correct['all'] / float(total['all'])

    print " (ALL)=%0.2f" % accuracy['all'] + \
            " (%d" % correct['all'] + "/%d" % total['all'] + ")"
    for sentiment in SENTIMENTS():
        accuracy[sentiment] = correct[sentiment] /  float(total[sentiment])
        print " (" + sentiment + ")=%0.2f" % accuracy[sentiment] + \
```

```
                " (%d" % correct[sentiment] + "/%d" % total[sentiment] + ")"

def mostUseful(pWord, usefulWords, predictors):
    proportion = {}
    for sentiment in SENTIMENTS():
        proportion[sentiment] = {}
        for word in pWord[sentiment]:
            proportion[sentiment][word] = \
                    pWord[sentiment][word] / pWord['all'][word]
        topWords = sorted(proportion[sentiment].iteritems(),
                key=operator.itemgetter(1), reverse=True)[:predictors]
        usefulWords[sentiment] = topWords
        usefulWords['all'] += topWords

def writeArff(mostUsefulAll, sentences):
    f = open("emotion.arff", "w")

    f.write("@relation emotion\n\n")
    for feature in mostUsefulAll:
        f.write("@attribute \"" + feature[0] + "\" {yes,no}\n")
    f.write("@attribute emotion {" + ",".join(SENTIMENTS()) + "}\n\n@data\n")

    for sentence,sentiment in sentences.iteritems():
        if sentence == '':
            continue
        nGrams = makeNGramList(sentence)
        for word in mostUsefulAll:
            if word[0] in nGrams:
                f.write('yes, ')
            else:
                f.write('no, ')
        f.write(str(sentiment) + "\n")
    f.close()

#---------- Main Script --------------------------

#initialise datasets and dictionaries
sentencesTrain={}
sentencesTest={}
freq = {}
pWord={ 'all': {} }
for sentiment in SENTIMENTS():
    pWord[sentiment] = {}

# split the sentiment files into training and test sets
readFiles(sentencesTrain,sentencesTest)

#build conditional probabilities using training data
trainBayes(sentencesTrain, pWord, freq)

# establish the most predictive patterns
usefulWords = { 'all': [] }
for sentiment in SENTIMENTS():
    usefulWords[sentiment] = []

mostUseful(pWord, usefulWords, 100)

writeArff(usefulWords['all'], sentencesTrain)

#run naive bayes classifier on datasets
print "Naive Bayes"
print "Train Accuracy"
```

```
testBayes(sentencesTrain, pWord)
print "Test Accuracy"
testBayes(sentencesTest, pWord)
```

B Functionality of Ruby classifier

```
classification on real data
  #classify
    should classify known fearful-text as fearful
  #classify_dev_set
    doesn't crash at runtime

ArrayExtensions#first_percent
  rounds up
  supports empty arrays
  with 0 gives the empty array
  with 10 gives the first 10% of an array
    should eq [1]
  with 100 returns the array
    should eq [:foo, :bar]
  graceful degradation
    treats >100 percentages as 100
    treats <0 percentages as 0

EmotionClassifier::Classifier
  should be initialised with some sentiments
  initialized with data
    has a training set
    has a dev-set of data to test against
    has a held-back set of test data
  #probability
    #probability with only word argument gives proportion of that word in the dataset
    #probability with word and sentiment arguments gives proportion of that word for that sentiment

EmotionClassifier::DataSet
  #sentences strips the sentence of punctuation and downcases
  splits the data into 80/10/10 for training/test/dev
  can give all the data
  assigns a sentiment to each sentence
  #with_sentiment gives the sentences with a given sentiment
  uses unigrams by default
  #set_ngram_order(3) makes the data set return trigrams
  unigrams
    #words gives an array with every word by default
    #words with sentiment argument gives words in that sentiment
  bigrams
    #words gives an array of bigrams from all sentences

EmotionClassifier::Emotion
  #to_string gives the name of the emotion
  #negate_to_s gives a string representing the opposite emotion
  #== is true when emotions have the same name

EmotionClassifier::Ngram
  #bigrams gives array of bigrams
  #trigrams gives array of trigrams
  can give arbitrary numbered n-grams
  treats word+punctuation as different from word
  #ngrams works with arrays as well as sentences
```

```

Finished in 3.16 seconds
33 examples, 0 failures
```

C Simple Bayesian Classifier

```ruby
require 'classifier'

classifier = Classifier::Bayes.new "Angry", "Fearful"

File.open('emotions/angry.txt').each { |grr| classifier.train_angry grr }
File.open('emotions/fearful.txt').each { |eek| classifier.train_fearful eek }

puts classifier.classify "I hate you!"
#=> Angry
puts classifier.classify "Please don't hurt me"
#=> Fearful
```