CS4025/CS5057: Continuous Assessment 1
Classifying Emotions

Philip Hale

November 25, 2013

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 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

2.2 System Performance