Jacob Dineen

Text Mining

7/29/18

Homework 2

As we’ve discussed at length, machines contrast from humans in several ways. They lack the ability to do much more than count, which is why we see all text mining and vision related problems translating the training data into a binary, or numeric, representation of the training data. The goal being to approximate a function that maps a set of input variables to a respective class with the lowest error. Homework 1 focused on the pretrained models available to us via NLTK and Sentistrength with each output portraying a unique perspective, and a different output. While NLTK focused mainly on polarity (positive or negative) and subjectivity, assuming as an output of a softmax activation function, Sentistrength was more of an ordinal classification problem, instead focusing on ranking sentiment and feature importance.

Since I haven’t left a whole lot of time to get this assignment written up, I’ll be using Weka (3.8) for vectorization. During the asynchronous session, I had explored using some preprocessing techniques in base python, coupled with some use of NLTK’s module for tokenization (Re: Movie Review slides). While it is certainly more scalable and without some of the memory restrictions that seem to accompany Weka, I am working with a dataset of less than 100 samples and Weka’s GUI for document transformation/vectorization is incredibly intuitive to use and interpret results.

Referencing back to Assignment 1, I had essentially collected two datasets – One that was scraped via Twython for a more robust (in size) sample, and one that was manually collected, which I believe was more representative of public perception. The former was more representative of the types of information dissemination that news outlets and similar entities have adapted Twitter for. The second dataset contained indicators of sentiment that the first lacked, and as such, I could analyze and label a grand truth vector with which I could interpret the success of NLTK in extracting information.

As an exploratory endeavor, I don’t have much inclination to believe that one vectorization technique will work better than another for this specific task – The task ultimately being able to vectorize raw tokens in such a way that a machine will be able to recognize linguistic patterns and perform some type of mapping, as referenced above – so I will iterate through potential solutions such as: Removing stop words, some preprocessing steps like converting to a specific case, variations of different stemmers and tokenizers, and Boolean/TF/TFIDF/Normalization transformations. It should be noted that Weka threw fits trying to read in my original file. I ran my original document through a mini script use the ‘re’ module in python that pretty much stripped any of the excessive characters that I thought may be causing issues (quotations appeared to be a big one, although it’s a bit tricky to debug the error messages). I also had to use the CSV converter to make sure that Weka understood that I was reading in strings from the first-last attribute range.

To begin, I’d like to have a baseline to measure up against. I’ll use default parameters initially – This means that there is no TF/IDF transformation, no stemming, and a default tokenizer. For the purpose of having some control, I’ll be leaving the wordsToKeep parameter set to 1000 for the entirety of this experiment, mainly because I believe that my total unique token count is less than 1000 anyway. This default experiment is using Boolean vectorization, so it’s the equivalent of creating a sparse matrix, and the results are to be expected as I spot-check some unique words. All the histograms noted for the categorical features are left skewed, meaning that they rarely appear in documents. It is likely that a countvectorizer, one that displays frequency counts of unique words, will also be sparse for most words. For exploratory purposes, I included a stemming technique (Lovins) and wasn’t very satisfied with the results. While the vocabulary size saw a notable decrease of 20ish%, the semantic and linguistic interpretation of the Stem makes it difficult to truly understand the document. My thinking is that overtly stemming documents might lead a machine to inaccurately detect a level of information gain that isn’t truly indicated by the tokens in that document.

Switching to countvectorizers, which, in my opinion, help to paint a more complete picture about word distribution in a document, the total vocabulary is similar in size. One of the more interesting things here is that the token ‘AI’ is not present in 58 of the 89 instances. Manually collecting them, I thought that this number would have been higher, leading me to believe that a bigram might be useful for phrases such as artificial intelligence, which would otherwise be broken out into individual tokens. Which leads me to a crucial point that I had missed earlier – It is likely more effective in this instance for me to convert all tokens to lowercase to try to enhance groupings. I could see issues that might arise in this case as far as contextual relevance and deciphering between a proper noun and a different part of speech, but it could work to reduce some of the sparsity and dimensionality through deduping. For the case of this assignment and case matching, our vocabulary is decreased by 10%. Frequency counting on its own doesn’t paint the whole picture. Exporting out a csv file with the tokens and their raw frequency counts shows us that most of the high-frequency words are stop words, highly occurring words that are so inclusive that their unlikely to have provide any information gain over a document. An example seen in Appendix 1.1 shows most of the high frequency words are stopwords. Words that appear to show some indication of being tokens containing a level of polarity aren’t prevalent until the 40-100 range, and even there, the document frequency is relatively low. I think this echoes back upon the discussion during class that a tf matrix with relatively few samples might be better served as type Boolean rather than raw counts, mainly because we’ll want to be able to generalize to unseen test data at a later point.

\*Using an NGram tokenizer expands the total vocabulary 5x, but it’s less likely that you’ll have any number of documents containing the same set of concatenated tokens, leading me to believe that Ngrams might be effective with more data.

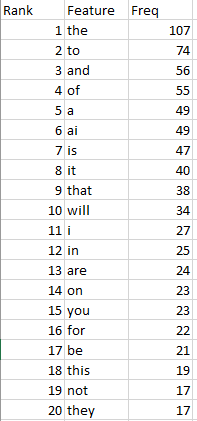
A TF-IDF transformation with stopwords removed, as seen in 1.2, starts to paint a little bit clearer of a picture, although it still isn’t something that a human would be able to readily analyze at scale.

It seems necessary to discuss the idea that it becomes quite difficult to analyze individual tokens when the token count reaches a certain threshold. You almost need to have prior knowledge about which words are likely to have some sentiment indicator and be able to sift to those. A better indicator may be to feed these through some classification algorithm (C.A.R.T. are usually the easiest to interpret through visualization) and understand which words are root nodes, to touch back on feature ranking. A human being, on their own, is unlikely to be able to

Based on this exercise, I would say that, for my data, the best vectorization option would be a Boolean vector with stopwords removed, no stemming, and a generic tokenizer. It isn’t very clear that more information about word frequency would be helpful in discerning a class mapping across a feature space.

**Appendix**

**1.1**



**1.2**

Summed TD-IDF weights across documents w/ stopwords removed.

