**22c:016 Computer Science I: Foundations**

Project 2: Presidents!

Due Friday, December 10 at 11:59PM

## Introduction

In this project, we're going to be doing some text analysis. The purpose of this project is to build the computational machinery required to efficiently compare the language used in these documents. Ultimately, we'd like to be able to characterize documents by the language they use, which means being able to measure if a document is similar to another. Here, you will be asked to characterize the most similar documents, and then to characterize some new text by which document it is most similar to. The basis we will use for comparison is called the *vector space model*, and was originally introduced by Gerard Salton in 1975 — when there was precious little electronic text around to characterize! It is still one of the models used by information retrieval and/or text analysis systems even today, although there are more sophisticated methods available.

The basic idea is that each document can be represented by an (ordered) list of frequencies, where each frequency corresponds to the number of times its corresponding word appears in the text. If you think of this list of N numbers as an N-dimensional vector, then the notion of a "vector space" makes more sense: a document is just a vector in this N dimensional space. We'll worry about exactly which words are counted and appear in the vector in just a minute, but assuming that the vector is normalized so as to be 1 unit long, two vectors representing documents with very similar word content will have vectors that point in approximately the same direction.

Now this leaves a lot of details to be pinned down. First, which words are included in the vector? How do we normalize vectors to unit length? And are these raw frequencies? Or are they also normalized in some way? How do I compare two vectors? These details are really important, and we'll have to settle them before we can start coding.

## Data

I have obtained from an online archive the text of 894 presidential speeches. Some of these speeches are very short (President Reagan's address at Moscow State University on May 31, 1988 is only 72 words long) and some are very long (President Lincoln's three hour long speech arguing against the Kansas/Nebraska Act on October 16, 1854 clocks in at 32,991 words). I have put the text through a regex filter to remove most of the cruft, but there are still likely to be some unrecognized or missing characters (some of these files were generated by optical character recognition software, which can make this kind of mistake).

I will provide for you a zip file containing all 511 speeches that exceed 2000 words. In addition, I will provide a set of 5 "unknown" speeches to identify by president.

## Document Similarity

We will be using the tf/idf (term frequency/inverse document frequency) weighting scheme originally proposed by Salton and his students for our vectors. Each vector $v$ can be represented as a list of $N$ terms, where a term is essentially a word or word stem and $N$ is a prespecified number of terms we will target in characterizing a document. In general, one picks the $N$ most frequently used words or word stems over the entire document collection or *corpus*; as $N$ gets larger, we get better performance, at least until the words become so infrequent as to add little to performance.

The tf/idf weighting scheme is defined as follows:

$$v = [w_0, w_2, w_3 \cdots w_{(N-1)}]$$

and, for a given document, each weight $w_i$ is computed as follows:

$$w_i = tf_i * \log \frac{|D|}{1 + \{d \in D | t_i \in d\}}$$

Here, for term $t_i$, $tf_i$ is related to the number of times the term appears in the current document, $|D|$ is the number of documents in the entire corpus $D$, and $\{d \in D | t_i \in d\}$ is the number of documents that contain the term $t_i$. The 1 in the denominator saves us from a divide-by-zero issue; the logarithm can be to any base and serves only as a scaling factor. Note that $tf_i$ is not usually just a word count, but is normalized by the length of the document, hence:

$$tf_i = \frac{\textit{number of times ith term appears in document}}{\textit{length of document}}$$

If we characterize each document by its vector $v$, we can compute a measure of similarity by simply taking the scalar product (also called the dot product) of the two vectors. The dot product of two vectors $v_1$ and $v_2$ represented as indexable tuples or lists is simply:

$$\sum_{0 \le i < N} v_1[i] \times v_2[i]$$

A dot product of 1 means the two documents have exactly the same word frequencies, while a dot product of 0 means the two documents share no words in common.

To a first approximation, this model does a pretty good job of ranking similar documents. Of course, it does have its limitations. For example, the model loses any idea of the order in which the words actually occur, which is why this is sometimes called a bag-of-words approach.

## Part 1: Preliminary Processing

For each of the documents provided, you will need to build a document vector to represent it. To build the document vector, you will have to select appropriate terms and, because we are using tf/idf, you will also need to know the frequencies for these terms both within a document and across all documents in the corpus.

Your first function:

```
def extractTerms(filelist, corpusTerms):
```

computes *term frequency dictionaries* (tfd) for each file in filelist, as well as a corpus frequency dictionary (the second argument) for the entire corpus of files. It should be invoked as, *e.g.*,

```
>>> cfd = {}
>>> tfds = extractTerms(['file1', 'file2', 'file3'], cfd)
```

which will return a list of term frequency dictionaries (*i.e.*, dictionaries where the terms are keys and the values are counts), one per file in the input filelist, and update the corpus frequency dictionary (*i.e.* a dictionary where terms are keys and the values are the number of documents where that term occurs) that was passed in as an argument.

More specifically, the function opens each file in the specified filelist, reads the contents, removes stop words (see list provided), applies a stemmer (see below) to each remaining word to produce a term, and then constructs a term frequency dictionary for that file while at the same time adding entries and updating counts in the corpus term frequency dictionary. In this way, once all the files are processed, the corpus frequency dictionary will accurately reflect, for each term, the number of files in which it occurs.

An example is given later in this document. **A good place to start are the readInput() and parse() functions we wrote while studying War and Peace in class. You will want to handle punctuation, contractions and any other issues, such as dates and the like, that may arise when you examine the output of your extractTerms() function on the documents provided.**

Your code should make use of a stemmer:

```
def stemmer(word):
```

to reduce each word to its basic form. Your stemmer should extend the stemmer used in class:

```
def stemmer(word):
    if word[-3:] == 'ies' and word[-4:] not in ['eies', 'aies']:
        word = word[:-3] + 'y'
    if word [-2:] == 'es' and word[-3:] not in ['aes', 'ees', 'oes']:
        word = word[:-1]
    if word [-1:] == 's' and word[-2:] not in ['us', 'ss', 'ys']:
        word = word[:-1]
    return(word)
```

which is based on Harman's *weak stemmer,* to also remove the following endings from words: *-able, -al, -ance, -ant, -ar, -ary, -ate, -ement, -ence, -ent, -er, -ess, -ible, -ic, -ify, -ine, -ion, -ism, -iti, -ity, -ive, -ize, -ly, -ment, -or, -ou, -ous, -th*, and *-ure*. **You may wish to rewrite/recast the stemmer to use regular expressions and incorporate additional features, but this isn't required.**

## Part 2: Vector Models

Once you have created the word distributions for each of the documents provided, you can use the document dictionary to find the k=100 most frequently used words in the corpus; these will be the terms in your vector space. You are now ready to convert the list of dictionaries returned from extractWords() into a list of tuples representing the tf/idf encoding of each document according to the $k$ most commonly used terms within the corpus (*i.e.*, the terms with the highest values in the corpus frequency dictionary).

```
def createModels(tfds, cfd, k):
```

should take take the list of term frequency dictionaries returned from extractWords(), the corpus frequency dictionary, an integer k, and return two values: a $k$-tuple of word terms and a list of $N$ $k$-tuples, representing the vector model of each document built according to the word term $k$-tuple.

An example will help to clarify. Consider a corpus consisting of only three documents, *file1*, *file2*, and *file3* containing, respectively, 'a a a b b', 'b b b b b c c', and 'a a b b b b b b b'. Here, we'll assume each letter from [abc] represents a single word in the document, there are no stop words, and because stemming isn't an issue, terms are simply single characters.

```
>>> cfd={}
>>> tfds=extractTerms(['file1','file2','file3'], cfd)
>>> cfd
{'a':2 ,'b':3 ,'c':1}
```

Note that the corpus frequency dictionary correctly states that 'a' appears in 2 documents, 'b' in 3 documents, and 'c' in 1 document.

```
>>> tfds[0]
{'a':3, 'b':2}
>>> tfds[1]
{'b':5, 'c':2}
```

```
>>> tfds[2]
{'b':7, 'a':2}
```

Also show the correct character counts from each file.

Now if I invoke createModels() as follows:

```
>>> (terms, models)=createModels(tfds, cfd, 2)
```

the first value returned, terms, will be a list of the terms used to construct each document's vector model. Here, because $k = 2$, there will be exactly two terms, and they will correspond to the most terms with the highest values in cfd:

$$('b', 'a')$$

**Note: this was the part that was totally screwed up the first time:** For the first document, the tfd vector will be:

$$(\frac{2}{5} \times \log(\frac{3}{3}), \frac{3}{5} \times \log(\frac{3}{4}))$$

For the second document:

$$(\frac{5}{7} \times \log(\frac{3}{6}), \frac{0}{7} \times \log(\frac{3}{7}))$$

And for the third document:

$$(\frac{7}{9} \times \log(\frac{3}{8}), \frac{2}{9} \times \log(\frac{3}{3}))$$

**Moreover, we will want these vectors to be unit vectors, so they must be normalized accordingly. To normalize a vector $[x_1, x_2, x_3]$, you will want to divide each term in the vector by the sum of the terms squared. In other words, the normalized form of $[x_1, x_2, x_3]$ is:**

$$[\frac{x_1}{u}, \frac{x_2}{u}, \frac{x_3}{u}]$$

**where $u = \sqrt{x_1^2 + x_2^2 + x_3^2}$.**

## Part 3: Presidential Similarity

Now that you have your machinery in place, for each of the 43 presidents provided, compute the 6 possible similarities between the 4 speeches provided for that president; that is, for each of the 4 speeches, compute the similarity with each of the other 3 speeches: when you ignore ordering, there will be 6 dot products to compute. Report the average dot product for each president, and plot, using matplotlib, a histogram of the 43 averages obtained.

## Part 4: Attributing Unknowns

For the last part of the assignment, you will compute the vector model of the 5 unknown documents provided and compute their similarity with each of the presidential speeches in your corpus. For each unknown speech, report the 10 closest matches with existing speeches. Can you guess who wrote those speeches? (You will not be graded on accuracy, but once the assignment is complete, I will reveal the answers).