ISTA 331 HW: TEXT MINING WITH COSINE SIMILARITY

Due Thursday, 3/27, 2020, 11:59 PM

1. Introduction

Cosine similarity is a way of measuring the similarity between two vectors by looking at the angle between them. Vectors that point in the same direction have an angle close to 0 between them; vectors that point in opposite directions have an angle close to π radians (or 180°), and vectors that are perpendicular have an angle near $\pi/2$ or 90°.

The measure of similarity is called *cosine similarity* because instead of measuring the angle directly, we find the cosine of the angle. This can be conveniently calculated using a dot product. Remember that for vectors $\mathbf{u}, \mathbf{v}$, the dot product is defined by the formula

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=0}^{m-1} u_i v_i$$

where m is the dimension (number of elements in the vectors). It can be proved that the dot product can also be expressed in a geometric form:

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos\theta$$

where θ is the angle between the vectors.

Application: related documents. Our application is going to be to measure how similar documents are to one another. This can be used in practice to categorize documents by subject or style, or even attempt to determine who wrote a particular text.[1]

1.1. Application to text analysis.

We'll use cosine similarity along with the bag-of-words approach that we used in the naïve Bayes spam filter in HW2 to measure the similarity between pairs of documents. For a simplified version, consider the two sentences:

```
'I like rich chocolate cake more than I like apple pie.'
'My dog likes peanut butter more than he likes apple pie.'
```

Intuitively, these sentences are *similar* in terms of the language they use, their topic, etc. We can measure this by converting each sentence into a vector counting how many times each word appears:

```
word  | apple butter chocolate cake dog he I like likes ...
count1|   1     0        1       1   0  0  2   2    0   ...
count2|   1     1        0       0   1  1  0   0    2   ...
```

This is a "bag of words" because the only thing we consider is how many times each word appears in the sentence.

[1] Since writing style has to do with much more than just word choice, usually cosine similarity is just a part of a larger strategy in authorship analysis.

To improve this analysis, we can drop very common words (like I, my, he, etc.) because they aren't very informative, and treat words such as 'like' and 'likes' as equivalent. By the end, the vectors might look something like this:

```
word  | apple butter chocolate cake dog like peanut pie rich
count1|   1     0        1       1   0    2     0    1   1
count2|   1     1        0       0   1    2     1    1   0
```

Then, we could calculate the dot product of these vectors to be 6, and the magnitude of each vector to be 3. Applying the formula above,

$$\mathbf{u} \cdot \mathbf{v} = 6 = 3 \times 3 \times \cos\theta$$

so that $\cos\theta = 2/3$. So we would say that the cosine similarity of these two sentences is $2/3$. Notice we never actually calculate the angle; this is partly because it doesn't contain any additional information, and partly because a scale running from -1 to 1 is more intuitive than a scale running from 0 to π.[2]

This is the rough idea; there are a couple of refinements we want to use when implementing this in practice.

1.2. **Stop words and stemming.** Above, we stripped out some common words and also collapsed two similar words, 'like' and 'likes', to the same category. Common words we want to ignore are called *stop words* in natural language processing. We'll use a standard list of them in `sklearn.feature_extraction`. Include the following import:

```
from sklearn.feature_extraction import stop_words
```

Then, the list of words we want to ignore is found in `stop_words.ENGLISH_STOP_WORDS`.

The next step is *stemming*, which refers to chopping off suffixes so that different forms of the same word get counted together. In the example above, this corresponds to counting 'likes' as the same word as 'like'. The Python package `nltk` has a few canned solutions for stemming; the one we'll use is called `SnowballStemmer`. The way we use it is to initialize an instance like

```
stemmer = SnowballStemmer("english")
```

after which we can call the `stem` method and have it return the reduced word:

```
In [3]: stemmer.stem("likes")
Out[3]: 'like'
```

In natural language processing, the results of stemming are often called *tokens* to distinguish them from the unprocessed words.

1.3. **TF-IDF.** The last adjustment we need to make to the calculation above is to weight each word by a quantity called *inverse document frequency* (IDF). This gives extra weight to uncommon words, because otherwise more common words would dominate the calculation (even though uncommon words are often more distinctive, and so more informative). The formula for the IDF of a token t is

$$IDF(t) = 1 + \log_2\left(\frac{\text{total \# of documents}}{\text{\# of docs containing } t}\right)$$

In the instructions below you'll be asked to use a slightly different formula.

[2]You might notice that in our application, the scale actually runs from 0 to 1; none of these vectors have negative entries so the dot product is always ≥ 0. But in some situations, negative similarity may be possible.

The TF part of TF-IDF is *term frequency*, which is just the number of times a word appears in a document – i.e., just the same count we described above. The metric we'll use in the end is just the product of these two numbers, $TF \times IDF$.

2. Instructions

Create a module named `hw5.py`. There are six text files on D2L; three are books from a popular science fiction series, and the other three are books from a popular fantasy series. We are going to create a matrix of the similarities of these documents to see if this technique can show which documents belong together. Code up the 9 functions as specified below, and upload your module to the D2L HW4 assignments folder.

You will need the following imports (don't copy and paste these–copying from PDFs can cause formatting issues):

```
import string
import pandas as pd
import numpy as np
from sklearn.feature_extraction import stop_words
from nltk.stem import SnowballStemmer
```

2.1. **Testing.** Download `hw5_test.py` and auxiliary testing files and put them in the same folder as your `hw5.py` module.

2.2. **Documentation.** Your module must contain a header docstring containing your name, your section leader's name, the date, ISTA 331 Hw5, and a brief summary of the module. Each function must contain a docstring. Each docstring should include a description of the function's purpose, the name, type, and purpose of each parameter, and the type and meaning of the function's return value.

2.3. **Grading.** Your module will be graded on correctness, documentation, and coding style. Code should be clear and concise. You will only lose style points if your code is a real mess. Include inline comments to explain tricky lines and summarize sections of code.

2.4. **Collaboration.** Collaboration is allowed. You are responsible for your learning. Depending too much on others will hurt you on the tests. "Helping" others too much harms them in reality. Cite any sources/collaborators in your header docstring. Leaving this out is dishonest.

2.5. **Resources.**

- `https://textminingonline.com/dive-into-nltk-part-iv-stemming-and-lemmatization`
- `https://stackoverflow.com/questions/10554052/what-are-the-major-differences-and-benefits-of-p`
- `http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html`
- `http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics.pairwise`
- `https://pypi.python.org/pypi/editdistance`
- `https://docs.python.org/3/library/stdtypes.html#mapping-types-dict`
- `https://docs.python.org/3/library/stdtypes.html#string-methods`

3. Function specifications

Define the following functions:

- `dot_product`: takes two word vectors in the form of dictionaries and returns their dot products. (We only have to consider the words that exist in both dictionaries – why?)
- `magnitude`: takes a single word vector in the form of a dictionary and returns its magnitude.
- `cosine_similarity`: takes two word vectors in the form of dictionaries and returns their cosine similarity (use the previous two functions).
- `get_text`: This function takes the filename of a text file and returns a single string containing the cleaned-up contents of the file. Here's what we want to do to clean it:
  - remove every occurrence of the string `"n't"`;
  - remove every occurrence of a character from `string.punctuation`;
  - make everything lowercase; and,
  - get rid of all characters representing digits.
- `vectorize`: this function takes a filename, a stopwords list, and a stemmer, and returns a dictionary representing a wordcount vector mapping cleaned words from the file to word-counts. In other words, you should get the text from the file, break it into a list of tokens, pass each token through the stemmer, and if it is not a stop word, add it to or increment its count in the vector.

  Make sure that the empty string is not a key in the vector.
- `get_doc_freqs`: this function takes a list of wordcount vectors (i.e. dictionaries) and returns a dictionary that maps each key from all of the vectors to the number of vectors that word appears in.
- `tfidf`: this function takes a list of wordcount vectors (i.e. dictionaries) and replaces the word counts with TF-IDF measurements (see above). Use the formula

$$IDF(t) = 1 + \log_2\left(\text{scale} \times \frac{\text{total \# of documents}}{\text{\# of documents containing } t}\right)$$

  The scale parameter is not standard; it is included here to compensate for a low number of documents. Use scale 1 if there are 100 or more documents; otherwise, scale = 100/(# of documents).

Then, define the following function to test our similarity measure and compare our example documents:

- `get_similarity_matrix`: this function takes a list of filenames, a stopwords list, and a stemmer, and returns a `DataFrame` containing the matrix of document similarities. Your `DataFrame` should use the filenames for both its index and its column names, and the `[doc_a, doc_b]` entry of the matrix should be the cosine similarity between the TF-IDF vectors of documents `doc_a` and `doc_b`.

  Note: you can solve this by looping over the full row and column indices and calculating the similarity for each pair of indices, but this is somewhat inefficient. We know in advance that:
  - the matrix is symmetric (i.e. $similarity(A, B) = similarity(B, A)$)
  - the diagonal entries are all 1 since the cosine similarity of any vector with itself is 1

  This means that you can calculate the upper triangle of the matrix ($A_{i,j}$ where $j > i$) and then copy those values to the lower triangle, and fill the diagonal with 1s. Doing this will skip unnecessary calculations and make your function run about twice as fast, which will save you testing time because the documents we're using are pretty big.

Finally, define a `main` function that uses `get_similarity_matrix` to compute the similarity matrix for the five sample documents, `00001.txt`, `00002.txt`, etc. These six documents contain the text of three books each from two popular science fiction/fantasy series. Based only on the similarity matrix, can you group the books into their respective series?