Assignment 2: Vector Space Models (due 10 March 2020, 11.59 pm EST)

The goal of this assignment is to implement many of the things you learned in the lecture on Vector Space Models.

The goals of this assignment are:

1. To create a term-document and term-context matrices from a given corpus.
2. To implement weighting methods and similarity metrics.
3. To use the resulting vectors to measure how similar some documents are to each other and how similar some words are to each other.

The materials provided In this zip file are:

```
|-- code
|   |-- data
|   |   |-- processed
|   |   |   |-  data.pkl
|   |   |-- raw
|   |       |-- alt.atheism
|   |       |-- rec.autos
|   |       ...
|   |       |-  talk.religion.misc
|   |-- main.py
|   |-- preprocessing.py
|   |-- vec_spaces.py
|-- cs505_hw2.pdf
```

Note on dataset and terminology

In this assignment, you are provided with a subset of the "20newsgroups" dataset, which, as the name suggests, contains posts from 20 different categories. The subset we have provided contains only a few of the 20 categories.

Additionally, the data has been preprocessed and stored in a pickle file under `data/processed/`. The code that was used to process the data is provided in `preprocessing.py`. The default arguments to `numerize_data()` were used to produce this pickle. Read the documentation strings for `numerize_data()` in `preprocessing.py` and `create_term_document_matrix()` in `vec_spaces.py` to understand how the data was preprocessed. You should not need to do anything to use the preprocessed data except to call `preprocessing.read_processed_data()`.

Finally, note that, in this assignment, the word "document" has been conflated with "newsgroup".

Deliverables

Here are the deliverables that you will need to submit in a zip file:

```
|-- code
|   |-- main.py
|   |-- vec_spaces.py
|-- writeup.pdf
```

- `writeup.pdf` or `writeup.txt` for your discussion of interesting findings, and a section that includes the output of running the `main.py` that you will complete.
- The files `main.py` and `vec_spaces.py`, with code implementations.

1 Term-document matrix

In a term-document matrix, each row represents a word in the vocabulary and each column represents a document. For example, this is a small selection from a term-document matrix showing the occurrence of four words in five documents represents the number of times a particular word (defined by the row) occurs in a particular document (defined by the column).

	alt.atheism	rec.motorcycles	rec.sport.hockey	talk.politics.guns	talk.politics.misc
evolution	34	2	0	0	0
amendment	10	0	4	74	37
theory	64	5	4	8	16
constitution	10	0	0	200	100

The dimensions of a term-document matrix will be the number of documents (in this case, the number of newsgroups provided in the corpus) by the number of unique word types $|V|$ in the corpus.

1.1 Writing `create_term_document_matrix`

In addition to implementing returning the word document matrix described above, you will implement using tf-idf weighing for a term document matrix. The function signature for `create_term_document_matrix` has a flag called `tf_idf_weighing` that indicates whether the returned matrix should contain raw counts, or instead, tf-idf values.

NOTE: Feel free to ignore `tf_idf_weighing` variable until you get to the section "Interesting findings".

Tasks(15 points):

1. Write the body of the function `create_term_document_matrix` in `vec_spaces.py`.

1.2 Similar newsgroups and words

1.2.1 Cosine similarity

Note that each column in a term document matrix can be interpreted as a $|V|$ dimensional vector representing the newsgroup. This naturally leads us to ask, can we compute the similarity between the newsgroups?

The cosine, like most measures for vector similarity used in NLP, is based on the dot product operator from linear algebra.

The dot product acts as a similarity metric because it will tend to be high just when the two vectors have large values in the same dimensions. Alternatively, vectors that have zeros in different dimensions (orthogonal vectors) will have a dot product of 0, representing their strong dissimilarity.

This raw dot-product, however, has a problem as a similarity metric: it favors "long" vectors. For example, computing the dot product between "rec.sport.hockey" and "talk.politics.guns" above would yield a value twice as high as the dot product as between "rec.sport.hockey" and "talk.politics.misc", even though both "talk.politics.misc" and "talk.politics.guns" have the same probability distribution over the four given words.

The simplest way to modify the dot product to normalize for the vector length is to divide the dot product by the lengths of each of the two vectors. This normalized dot product turns out to be the same as the cosine of the angle between the two vectors.

The cosine value ranges from 1 for vectors pointing in the same direction, through 0 for vectors that are orthogonal, to -1 for vectors pointing in opposite directions. Since our term-document matrix contains raw frequency counts, pairwise cosine similarities are non-negative. 1 means that the vectors are identical, 0 means that they are totally dissimilar.

Tasks(5 points):

1. Implement `compute_cosine_similarity` in `vec_spaces.py`.

A sanity check is to make sure that a vector has a cosine of 1 with itself. If that's not the case, then you've messed something up.

1.2.2 Similar newsgroups analysis

Each column in a term document matrix can be considered a vector representation of the document. Using the term-document matrix, you will figure out which newsgroups are most similar to each other. Score how similar each newsgroup is to compared to other newsgroups. Print out the newsgroup that yields the highest similarity score for each newsgroup.

Tasks(5 points):

1. Complete `test_newsgroup_similarity` in `main.py` to achieve the above.
2. Call `test_newsgroup_similarity` on a (raw count, not tf-idf) term document matrix in the `main()` function.

1.2.3 Similar words analysis

Another interpretation of the term document matrix is that each row is a vector representation for a word.

In this section, you will choose a few words (ten to twenty) that you hypothesize will yield interesting insights. You will then find and print the words that are most similar to each of the word that you picked.

Tasks(5 points):

1. Complete `test_word_similarity` in `main.py` to achieve the above.
2. Call `test_word_similarity` on a (raw count) term document matrix in the `main()` function.

How do I know if my rankings are good?

Obviously, some pairs of newsgroups are more closely related. Take a look at this[1] grouping of the newsgroups if you don't trust your judgement.

The word rankings might, in fact, not be good. That is, unless you choose your words carefully. Don't worry about choosing good words too much until the "Interesting findings" section.

2 Term-context matrix

In a term-document matrix, the rows are word vectors. Instead of a $|V|$ -dimensional vector, these row vectors only have $D($ where D is the number of documents). Do you think that's enough to represent the meaning of words?

[1] http://qwone.com/~jason/20Newsgroups/

Instead of using a term-document matrix, a more common way of computing word similarity is by constructing a term-context matrix (also called a word-word matrix), where columns are labeled by words rather than documents. The dimensionality of this kind of a matrix is $|V|$ by $|V|$. Each cell represents how often the word in the row (the target word) co-occurs with the word in the column (the context) in a training corpus.

	evolution	amendment	theory	constitution
evolution	0	1	68	10
amendment	1	0	5	100
theory	68	5	0	1
constitution	10	100	1	0

2.1 Writing `create_term_context_matrix`

The `window_size` argument specifies the size word window around the target word that you will use to gather its contexts. For instance, if you set that variable to be 5, then you will use 5 words to the left of the target word, and 5 words to its right for the context.

We've seen in class what Positive Pointwise Mutual Information(PPMI) is. The `ppmi_weighing` argument specifies whether you return raw co-occurrence counts, or PPMI values instead.

NOTE: Feel free to ignore `ppmi_weighing` variable until you get to the "Interesting findings" section.

Tasks(20 points):

1. Write the body of the function `create_term_context_matrix` in `vec_spaces.py`.

2.2 Similar words analysis

You can now re-compute the most similar words for your test words using the row vectors in your term-context matrix instead of your term-document matrix.

Tasks:

1. Call `test_word_similarity` on (raw co-occurrence count) term context matrix in the `main()` function.

3 Word2Vec

NOTE: You will have to install `gensim` into your Python environment.

In this part, we will use `gensim` library's word2vec implementation to get 100 dimensional word vectors. (How big were our previous word vectors?)

In addition, we will use this matrix to analyze word similarity again.

Tasks(15 points):

1. Write the body of the function `create_word2vec_matrix` in `vec_spaces.py`.
2. Write the body of `test_word2vec_word_similarity` in `main.py`.
3. Call `test_word2vec_similarity` on this matrix in `main()` in `main.py`.

Note that, since we're using a library implementation of word2vec, the code patterns in this section will be a bit different. Notably:

- The `create_word2vec_matrix` doesn't take as argument the data that was in the pickle file provided. Instead, it takes an instance of a `DataLoader()` that has already been written for you, that will act essentially as a Python list that contains the data. Read the documentation of `DataLoader` to get a better idea.

- The `create_word2vec_matrix` returns a tuple of the word2vec matrix and a dictionary where the keys are indices $[0, V)$, and the values are words.

4 Additional similarity measures

There are several ways of computing the similarity between two vectors. In addition to writing a function to compute cosine similarity you will also write functions to Jaccard similarity and Dice similarity.

Tasks(10 points):

1. Write the body of the function `compute_dice_similarity` in `vec_spaces.py`.
2. Write the body of the function `compute_jaccard_similarity` in `vec_spaces.py`.

5 Interesting findings

(20 points), bonus points: 5

Play with different vector representations and different similarity functions. Does one combination appear to work better than another? Do any interesting patterns emerge? Discuss your findings in your write up. To qualify for the full credit for this section, you must discuss at least two interesting findings. A really interesting finding will get an extra 5 bonus points.

Note: Don't be worried about finding the "right" answer, but make sure that the answer you give can be backed up by examples.

Some patterns you must discuss:

- When computing word similarity, how does each of the term document matrix, term context matrix, and the word2vec matrix compare to each other? Is there a clear winner?
- What are the effects of the `tf_idf_weighing` and the `ppmi_weighing` flags? Do they improve(or worsen) the numerical methods' ability to approximate our intuitive sense of "similarity" between words and between newsgroups? Why/how?

Some patterns you can discuss:

- What are the effects of using different similarity measures?
- Can you find instances of bias in word representations(for example, are masculine words associated more with certain adjectives)?
- Etc.