

Project 4: Vector Space Models

Due: Tuesday Mar. 29th, 11:00am (Eastern Time)

Deliverables

- A completed `vectors.py` file. This includes filling in 8 functions:
 - Four for creating matrices: `create_term_document_matrix`, `create_term_context_matrix`, `create_PPMI_matrix`, and `compute_tf_idf_matrix`
 - Two for computing similarities: `compute_cosine_similarity` and `compute_jaccard_similarity`
 - Two for ranking outputs: `rank_plays` and `rank_words`
- `writeup.tex` and a corresponding `writeup.pdf` with answers to the 16 Lab questions.

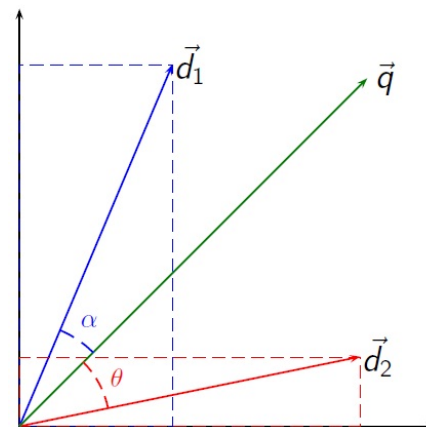
Be sure that your final submission code runs on the department machines and when executed from an arbitrary directory (not just the one that contains your code). Include your names at the top of your writeup file.

Introduction

In this assignment you will implement many of the things you learned in [Chapter 6 of the textbook](#). If you haven't read it yet, now would be a good time to do that.

I'll wait.

Done? Great, now we can begin!



I have provided a corpus of Shakespeare plays available at (`/data/cs65-S22/vectors`), which you will use to create a term-document matrix and a term-context matrix. You'll implement a selection of the weighting methods and similarity metrics defined in the textbook. Ultimately, your goal is to use the resulting vectors to measure how similar Shakespeare plays are to each other, and to find words that are used in a similar fashion. All (or almost all) of the code you write will be direct implementations of concepts and equations described in [Chapter 6, sections 6.3-6.7](#).

All of the data you'll need for this project is outlined below and already included in the associated GitHub repo or in the source directory (`/data/cs65-S22/vectors`) on the department machines:

Data

- Starter code is available in the `vectors.py` Python file of the Lab4 [GitHub repo](#).
- `will_play_text.csv` is a csv with the complete works of Shakespeare.
- `vocab.txt` is a text file with the vocabulary we'll be using for the project.
- `play_names.txt` is a text file with the list of all the plays in the Shakespeare dataset.

1 Term document matrix

You will write code to compile a term-document matrix for Shakespeare's plays, following the description in the textbook.

In a term-document matrix, each row represents a word in the vocabulary and each column represents a document from some collection of documents. The table below shows a small sample from a term-document matrix showing the occurrence of four words in four plays by Shakespeare. Each cell in this matrix represents the number of times a particular word (defined by the row) occurs in a particular document (defined by the column). Thus *fool* appeared 58 times in *Twelfth Night*.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

The dimensions of your term-document matrix will be the number of documents **D** (in this case, the number of Shakespeare's plays that I provided in the corpus) by the number of unique word types $|V|$ in that collection. The columns represent the documents, and the rows represent the words, and each cell represents the frequency of that word in that document.

Questions

1. Write the `create_term_document_matrix` function and describe your approach in the writeup.

This lets us answer questions like *how many words did Shakespeare use?*¹

¹The table will also tell you how many words Shakespeare used only once. Did you know that there's a technical term for that? In corpus linguistics they are called *hapax legomena*, but I prefer the term *singleton*, because there's no

1.1 Comparing plays

The term-document matrix will also let us do cool things like figure out which plays are most similar to each other, by comparing the column vectors. We could even look for outliers to see if some plays are so dissimilar from the rest of the canon that **maybe they weren't authored by Shakespeare after all**.

Let's begin by considering the column representing each play. Each column is a $|V|$ -dimensional vector. By far the most common similarity metric is the cosine of the angle between the vectors. The cosine similarity metric is defined in Section 6.4 of the textbook:

The cosine — like most measures for vector similarity used in NLP — is based on the dot product operator from linear algebra, also called the inner product:

$$\text{dot product}(\vec{v}, \vec{w}) = \vec{v} \cdot \vec{w} = \sum_{i=1}^N v_i w_i = v_1 w_1 + v_2 w_2 + \dots + v_N w_N$$

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.

The dot product is higher if a vector is longer, with higher values in each dimension. More frequent words have longer vectors, since they tend to co-occur with more words and have higher co-occurrence values with each of them. The raw dot product thus will be higher for frequent words. But this is a problem; we'd like a similarity metric that tells us how similar two words are regardless of their frequency.

We modify the dot product to normalize for the vector length by dividing 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, following from the definition of the dot product between two vectors \vec{a} and \vec{b} :

$$\vec{a} \cdot \vec{b} = |\vec{a}| |\vec{b}| \cos \theta$$

$$\frac{\vec{a} \cdot \vec{b}}{|\vec{a}| |\vec{b}|} = \cos \theta$$

The cosine similarity metric between two vectors \vec{v} and \vec{w} thus can be computed as:

$$\text{cosine}(\vec{v}, \vec{w}) = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}| |\vec{w}|} = \frac{\sum_{i=1}^N v_i w_i}{\sqrt{\sum_{i=1}^N v_i^2} \sqrt{\sum_{i=1}^N w_i^2}} \quad (1)$$

need to make this sound more technical than it really is.

The cosine value ranges from 1 for vectors pointing in the same direction, through 0 for orthogonal vectors, to -1 for vectors pointing in opposite directions. But since raw frequency values are non-negative, the cosine for these vectors ranges from 0-1.

Questions

2. Write the `compute_cosine_similarity` function and describe your approach in the writeup.
3. For each play in the corpus, score how similar each other play is to it.
 - (a) Which plays are the closest to each other in vector space (ignoring self similarity)?
 - (b) Which plays are the most distant from each other?

You might be wondering: *“But how do I know if my rankings are good or not?”* If you’ve read a subset of the plays, you can simply use your intuition. Alternatively, you might take a look at this [grouping of Shakespeare’s plays into Tragedies, Comedies and Histories](#). Do plays that are thematically similar to the one that you’re ranking appear among its most similar plays, according to cosine similarity? Another clue that you’re doing the right thing is if a play has a cosine of 1 with itself. If that’s not the case, then you’ve messed something up. Another good hint, is that there are a ton of plays about Henry. They’ll probably be similar to each other.

2 Measuring word similarity

Next, we’re going to see how we can represent words as vectors in vector space. This will give us a way of representing some aspects of the *meaning* of words, by measuring the similarity of their vectors.

Questions

4. In our term-document matrix, the rows are word vectors. Instead of a $|V|$ -dimensional vector, these row vectors only have D dimensions. Do you think that’s enough to represent the meaning of words? Try it out. In the same way that you computed the similarity of the plays, you can compute the similarity of the words in the matrix.
 - (a) Pick some words and compute 10 words with the highest cosine similarity between their row vector representations.
 - (b) Are those 10 words good synonyms?

2.1 Term-context matrix

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.

For this part of the assignment, you'll write the `create_term_context_matrix` function. This function 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 4, then you will use 4 words to the left of the target word, and 4 words to its right for the context. In this case, the cell represents the number of times in Shakespeare's plays the column word occurs in ± 4 word window around the row word.

Questions

5. Implement the `create_term_context_matrix` function and describe your approach in the writeup.
6. Re-compute the most similar words for your test words that you chose above in Question 4a but now using the row vectors in your term-context matrix instead of your term-document matrix.
 - (a) What is the dimensionality of your word vectors now?
 - (b) Do the most similar words make more sense than before (i.e. does this approach perform better)?

3 Weighting terms

Your term-context matrix contains the raw frequency of the co-occurrence of two words in each cell. Raw frequency turns out not to be the best way of measuring the association between words². There are several methods for weighting words that get better results. You'll implement two different weighting schemes: *Positive pointwise mutual information (PPMI)* — see Section 6.7 of the textbook — and *Term frequency inverse document frequency (tf-idf)* — see Section 6.5 of the textbook.

3.1 tf-idf

For a word t in a document d , then term-frequencies is just the log-transformed count of its occurrences (plus a pseudo-count parameter, here just 1, in order to prevent us trying to get the log of zero).

$$\text{tf}_{t,d} = \log(\text{count}(t,d) + 1) \quad (2)$$

The inverse document frequency (idf) is defined using the fraction $\frac{N}{df_t}$, where N is the total number of documents in the collection, and df_t is the number of documents in which the term t

²To be less generous, it's a terrible way of doing so.

occurs.

$$\text{idf}_t = \log \left(\frac{N}{df_t} \right) \quad (3)$$

The tf-idf is simply a weighted value that combines our tf and idf terms.

$$w_{t,d} = \text{tf}_{t,d} \times \text{idf}_t \quad (4)$$

3.2 PPMI

Pointwise mutual information (PMI) is a measure of how often two events (in our case a word and a context) occur $P(w, c)$, compared with what we would expect if they were independent $P(w)P(c)$. While PMI values can range from negative to positive infinity, for our purposes we want to ignore the negative values (see Section 6.7 in the textbook). Thus in Positive PMI (PPMI) we simply replace any negative PMI value with zero.

Formally, if we have a co-occurrence matrix F with W rows (words) and C columns (contexts), where f_{ij} gives the number of times word w_i occurs in context c_j . Then the frequency values f_{ij} are converted to $PPMI_{ij}$ values as follows:

$$p_{ij} = \frac{f_{ij}}{\sum_{i=1}^W \sum_{j=1}^C f_{ij}} \quad (5)$$

$$p_{i*} = \frac{\sum_{j=1}^C f_{ij}}{\sum_{i=1}^W \sum_{j=1}^C f_{ij}} \quad (6)$$

$$p_{*j} = \frac{\sum_{i=1}^W f_{ij}}{\sum_{i=1}^W \sum_{j=1}^C f_{ij}} \quad (7)$$

$$PPMI_{ij} = \arg \max(\log_2 \frac{p_{ij}}{p_{i*}p_{*j}}, 0) \quad (8)$$

Questions

7. Implement the `create_PPMI_matrix` function and describe your approach in the writeup.
8. Implement the `compute_tf_idf_matrix` function and describe your approach in the writeup.

A word of warning about runtimes here: calculating PPMI for your whole $|V|$ -by- $|V|$ matrix might be slow. My implementation for PPMI takes about 10 minutes to compute all values. But you can assume that I always write perfectly optimized code on my first try. You may improve performance by using matrix operations in NumPy.³

4 Other similarity functions

There are several ways of computing the similarity between two vectors. In addition to writing a function to compute cosine similarity, you should also write functions to compute **Jaccard similarity**. The Jaccard similarity between two vectors \vec{v} and \vec{w} can be computed as:

$$\text{Jaccard}(\vec{v}, \vec{w}) = \frac{\sum_{i=1}^N \min(v_i, w_i)}{\sum_{i=1}^N \max(v_i, w_i)} \quad (9)$$

Questions

9. Implement the `compute_jaccard_similarity` function.

5 Ranking

Questions

10. Implement the `rank_plays` function.
11. Experiment: pick a couple plays to ranks the others with respect to it. Try running different vector space models and report the results. Each possible model uses either (a) cosine or Jaccard similarity, (b) PPMI or tf-idf, and (c) chose two different window sizes so you should run a total of eight different models all together. Provide a couple sentences intuition about your findings.
12. Implement the `rank_words` function.
13. Repeat the above experiment but with words rather than plays. Again you should be running and reporting the outputs of eight different vector space models and providing some brief intuitions about the comparative outcomes.

³This is why a bunch of engineering tools use MATLAB by the way.

6 Feedback

Please provide answers to these in the writeup just like the rest of the questions. There are no right or wrong answers here of course :)

Questions

14. Approximately how long did you spend on this assignment?
15. Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?
16. Which aspects of this assignment did you like? Is there anything you would have changed?

7 Extra credit: Comparing vector spaces

Quantifying the goodness of one vector space representation over another can be very difficult to do. It might ultimately require testing how the different vector representations change the performance when used in a downstream task like question answering (this is a form of *extrinsic evaluation*). A common way of quantifying the goodness of word vectors is to use them to compare the similarity of words with human similarity judgments, and then calculate the correlation of the two rankings.

If you would like extra credit on this assignment, you can quantify the goodness of each of the eight different vector space models that you produced. You can calculate their scores on the [SimLex999 data set](#), and compute their correlation with human judgments using [Kendall's Tau](#).

Add a section to your writeup explaining what experiments you ran, and which setting had the highest correlation with human judgments.



FAQs

- When finding the top 10 similar items for a given target element, should I count the target element?

No, do not count the target element.

- How can I improve the performance and efficiency of my code?

Try to use [vectorized code](#) wherever possible instead of using loops. You can refer to this resource on [vectorized code](#).

- How many documents should I consider for reporting similarity scores in the writeup?

You need not report similarity scores for every pair of documents. A subset of similarity scores should be sufficient. For instance, you can include the top 10 of one or two documents.

- How can I compute the similarity scores on the SimLex999 data set, and compute their correlation with human judgments using Kendall's Tau?

You can use [simlex data](#) to get a ranking list with your model and calculate the number of concordant and discordant pairs.