TTIC 31190: Natural Language Processing – Autumn 2023
Assignment 1: Distributional Word Vectors

Instructor: Freda Shi and Jiawei (Joe) Zhou
Assigned: Wednesday, October 4, 2023
Due: 11:59pm CDT/CST, Thursday, October 19, 2023

Submission Instructions

Submit your report (in pdf format) and code through Gradescope. There will be two separate assignments on Gradescope for report and code submission. You can find a link to Gradescope on the Canvas assignment page. Your report should contain all of the required results and analysis.

Lateness Policy

If you turn in an assignment late, a penalty will be assessed. The penalty will be 2% (of the entire point total) per hour late. You will have 4 late days to use as you wish during the quarter. Late days must be used in whole increments (i.e., if you turn in an assignment 6 hours late and want to use a late day to avoid penalty, it will cost an entire late day to do so). If you wish to use a late day for an assignment, indicate that in the comments along with your homework submission through Canvas.

Collaboration Policy

You are welcome to discuss assignments with others in the course, but solutions and code must be written individually.

Distributional Word Vectors

You will implement and experiment with ways of creating word vectors. Let's begin by defining a **vocabulary** V and a **context vocabulary** V_C. The steps below will ask you to implement different ways of building word vectors for the words in V, using context words from V_C. You will be provided with a dataset of sentences from English Wikipedia, along with files that you can use to fill V and V_C. Before describing the ways of computing word vectors, we will discuss evaluation and the provided data files.

Quantitative Evaluation (EVALWS)

You will quantitatively evaluate your word vectors by comparing the word similarities they produce to human-annotated word similarities. That is, you will be provided with files containing pairs of words along with their human-annotated similarity scores. You should use your word vectors (calculated from Wikipedia corpus) to compute the cosine similarity of each word pair in the word similarity files. Use Spearman's rank correlation coefficient (also called Spearman's ρ; see `scipy.stats.spearmanr`) as the evaluation metric. That is, for each word pair in a word similarity dataset, you will compute its cosine similarity using your word vectors, which forms one list of similarities. Then, form another list of similarities for those same word pairs, this time containing the manually-annotated similarities. Compute Spearman's ρ between the two lists of similarities. Do this for two word similarity datasets: MEN and SimLex-999. The files are provided. (If any word pair contains a word that does not appear in the training corpus, simply use a cosine similarity of 0.0 for that pair.) We will refer to this procedure as EVALWS.

Provided Data

The following data files are provided to you for this assignment:

- `wiki-1percent.txt` - sample of English Wikipedia downloaded in May 2015, with punctuation separated from words, all characters converted to lowercase, and with one sentence per line.
- `men.tsv` - MEN word similarity dataset. The first line is a header, then each line has the format "word1`[tab]`word2`[tab]`similarity_score".
- `simlex-999.tsv` - SimLex-999 word similarity dataset. Same format as MEN.
- `vocab-5k.txt` - file containing the 5,000 most common words in Wikipedia, one word per line.
- `vocab-15kws.txt` - file containing the 15,000 most common words in Wikipedia plus the words in the word similarity evaluation datasets, totaling 15,228 words.

Implementation Tips

The most difficult part of the programming required to do this assignment is making your implementation efficient enough to scale up to large corpora and a large context vocabulary size. Below are some tips to keep in mind:

- You only need to do one pass through the corpus to compute the required counts, so you do not need to store the corpus in memory.
- You can use sparse data structures to store the counts (these may have various names depending on the programming language, e.g., maps, hashes, or dictionaries). For example, you can use a sparse data structure to map a word (from V) to a sparse data structure that maps context words (from V_C) to counts. You can also exploit sparsity when computing cosine similarities.
- If you encounter underflow when estimating the small probabilities used for PMI calculation, you can perform the computation in the log domain (though we did not have to do this for standard implementation).

1 Distributional Counting (24 points)

Write code to count the number of times that word y appears in a context window with size w centered at the word x, using the provided `wiki-1percent.txt` corpus. A context window contains up to w words to either side of the center word, so it contains up to $2w + 1$ words in total (including the center word). If y appears multiple times in a single window, count each occurrence separately. We will use the notation $\#(x, y)$ to denote the count of the tuple $\langle x, y \rangle$, i.e., the number of times that word y appeared within w words to the left or right of x. Keep in mind that the tuples are *ordered*: the first item in the tuple $\langle x, y \rangle$ is the **center word** and the second item is the **context word**.

The easiest way to implement this is as follows. Iterate through all valid windows in each line (a valid window is one that is centered on a token in a line). For each valid window centered on word x, iterate through the w words to the left of x and the w words to the right of x. For each such word y, increment the count for the tuple $\langle x, y \rangle$. It does not matter whether y is before or after x in the window. Either way, you should still increment the count for the tuple $\langle x, y \rangle$.

If there are fewer than w words on a side, only iterate over the words that are present. Do not pad with additional tokens in order to fill up the window. Each line is a separate sentence. Do not merge lines in order to fill up windows.

Use V and V_C when computing these counts. Only compute counts for $\langle x, y \rangle$ tuples if both $x \in V$ and $y \in V_C$. If $x \notin V$ or $y \notin V_C$, skip the tuple. If $V = V_C$, then $\#(a, b) = \#(b, a)$, but this is not necessarily the case when $V \neq V_C$.

Consider the following line as an example:

rest for the rest of the day

Table 1 shows the counts we would compute from this line for $w = 3$ for two different choices of the context word vocabulary V_C. Note in this example how $\#(x, y) = \#(y, x)$ when $V = V_C$ but this is not necessarily the case when $V \neq V_C$.

$V = \{$day, for, of, rest, the$\}$
$V_C = \{$day, for, of, rest, the$\}$

tuple	count
#(day, of)	1
#(day, rest)	1
#(day, the)	1
#(for, of)	1
#(for, rest)	2
#(for, the)	1
#(of, day)	1
#(of, for)	1
#(of, rest)	1
#(of, the)	2
#(rest, day)	1
#(rest, for)	2
#(rest, of)	1
#(rest, rest)	2
#(rest, the)	3
#(the, day)	1
#(the, for)	1
#(the, of)	2
#(the, rest)	3
#(the, the)	2

$V = \{$day, for, of, rest, the$\}$
$V_C = \{$day, for, rest$\}$

tuple	count
#(day, rest)	1
#(for, rest)	2
#(of, day)	1
#(of, for)	1
#(of, rest)	1
#(rest, day)	1
#(rest, for)	2
#(rest, rest)	2
#(the, day)	1
#(the, for)	1
#(the, rest)	3

Table 1: Nonzero counts for the example sentence when using $w = 3$ with two different choices for the context vocabulary V_C.

After having computed these counts, we can use them to form word vectors for the words in V. The word vector for a word $x \in V$ is $|V_C|-$dimensional; that is, it has an entry for each context word $y \in V_C$ with value $\#(x, y)$. Therefore, the counts $\#(\cdot, \cdot)$ can be viewed as representing word vectors for the words in V. Note that zero counts do not need to be represented explicitly in the counts $\#(\cdot, \cdot)$. You can just store nonzero counts and assume that all other tuples have count zero. By doing so, word vectors can be represented sparsely, saving memory and speeding up the computation of word similarities.

1.1 (12 points) Implement distributional counting as described above for a provided w, V, and V_C. Submit your code.

1.2 (6 points) Using `vocab-15kws.txt` to populate V and `vocab-5k.txt` to populate V_C, **use your code to report $\#(x, y)$ for the pairs in the following table for both $w = 3$ and $w = 6$**. Some counts have already been filled in for you which you can use to check your code:

	$w = 3$	$w = 6$
#(chicken, the)	52	103
#(chicken, wings)	6	7
#(chicago, chicago)	38	122
#(coffee, the)		
#(coffee, cup)		
#(coffee, coffee)		

1.3 (6 points) Using $w = 3$ (and again using `vocab-15kws.txt` for V and `vocab-5k.txt` for V_C), **evaluate your count-based word vectors using EVALWS and report your results on MEN and SimLex-999**. As a sanity check, your Spearman correlation for MEN should be close to 0.22.

Note: If there is a word in a word similarity dataset that has no counts in the Wikipedia corpus, its length will be zero and therefore any cosine similarity computed for it will be undefined due to division by zero. You can handle this case by just defining cosine similarity involving such words to be zero.

2 Combining Counts with Inverse Document Frequency (IDF) (10 points)

One problem with using raw counts is that they place too much weight on extremely common words. We'll try to address this by using **inverse document frequency (IDF)**. IDF was developed in the context of document retrieval, but here we're working with sentences so we'll be applying the idea to sentences rather than documents.

Let S denote the set of sentences in the corpus. Then, instead of defining word vector entries using counts $\#(x, y)$, we will define them as follows. The word vector for a word $x \in V$ has an entry for each word $y \in V_C$ with value given by:

$$\#(x, y) \times \frac{|S|}{|\{s \in S : s \text{ contains } y\}|} \qquad (1)$$

The first term above is the "term frequency" (TF) and the second is the inverse of the "sentence frequency" for the context word. Note that if a word vector entry was previously zero when only using counts, it is still zero when using the above formula, so we can still use sparse data structures.

2.1 (10 points) Extend your implementation to be able to compute IDF-based word vectors using Eq. 1. Using $w = 3$, `vocab-15kws.txt` to populate V, and `vocab-5k.txt` to populate V_C, **evaluate (EVALWS) your IDF-based word vectors and report your results**. You should see improvements in Spearman correlation for both MEN and SimLex-999. (Note: There are several other ways of computing IDF, including using logarithm transformations and other sorts of scaling. Feel free to experiment with other ways of computing IDF if you like. Just be sure to clearly describe the versions for which you report results.)

3 Pointwise Mutual Information (PMI) (14 points)

While IDF helps by downweighting frequent words, the IDF for a context word is computed independently of the center word it is paired with. Next, we'll define word vectors using **pointwise mutual information (PMI)**, which captures some information about the relationship between the center word and context word in a single number.

We define two random variables: X is a random variable representing the center word and Y is a random variable representing the context word. The set of possible events for X is the vocabulary V, while the set of possible events for Y is V_C. PMI is defined for the event pair $X = x, Y = y$ as follows:

$$\mathrm{pmi}(x, y) = \log_2 \frac{p_{X,Y}(x, y)}{p_X(x)p_Y(y)}$$

where $p_X(x)$ and $p_Y(y)$ are the probability mass functions for X and Y, and $p_{X,Y}(x, y)$ is the joint distribution.

We will compute PMIs by using the counts $\#(x, y)$ that we computed above. First, we define the total count N:

$$N = \sum_x \sum_y \#(x, y)$$

Then, we estimate the joint probabilities as follows:

$$p_{X,Y}(x, y) = \frac{\#(x, y)}{N}$$

We estimate the two other probabilities as follows:

$$p_X(x) = \frac{\sum_y \#(x, y)}{N}$$

$$p_Y(y) = \frac{\sum_x \#(x, y)}{N}$$

Plugging these into our formula for PMI gives us

$$\mathrm{pmi}(x, y) = \log_2 \frac{\#(x, y)N}{\sum_{y'} \#(x, y') \sum_{x'} \#(x', y)}$$

3.1 (8 points) Implement the capability of computing PMIs. Use your code to calculate PMIs for $w = 3$ when using `vocab-15kws.txt` to populate V and `vocab-5k.txt` to populate V_C. Note that since we are using different vocabularies for center words and context words, $\mathrm{pmi}(a, b)$ will not necessarily equal $\mathrm{pmi}(b, a)$ (though they will be similar). (If there is a word in V that has no counts, the numerator and denominator for all of its PMI values will be zero, so you can just define all such PMIs to be zero.) **For center word $x =$ "coffee", print the 10 context words with the largest PMIs and the 10 context words with the smallest PMIs.** Print both the words and the PMI values. (Note: using my implementation, the highest-PMI context word was "tea" with PMI 8.166, and the lowest-PMI context word was "he" with PMI -2.26034.)

3.2 (6 points) Now, define word vectors using PMI. That is, the word vector for a word $x \in V$ has an entry for each word $y \in V_C$ with value given by $\mathrm{pmi}(x, y)$. As above, use $w = 3$, `vocab-15kws.txt` to populate V, and `vocab-5k.txt` to populate V_C. **Evaluate (EVALWS) your PMI-based word vectors and report your results.**

4 Quantitative Comparisons (12 points)

4.1 (8 points) Evaluate the word vectors (EVALWS) corresponding to the three ways of computing vectors (counts, IDF, and PMI), three values of w (1, 3, and 6), and two context vocabularies (`vocab-15kws.txt` and `vocab-5k.txt`). For all cases, use `vocab-15kws.txt` for V. **Report the results (there should be 36 correlations in all) and describe your findings.** What happens as window size changes for different methods of creating word vectors? What happens when context vocabulary changes? Why do you think you observe the trends you see? Do you see the same trends for MEN and SimLex or do they differ?

4.2 (4 points) You should observe systematic trends in terms of correlation as window size changes which should differ for MEN and SimLex-999. **Look at some of the manually-annotated similarities in the MEN and SimLex-999 datasets and describe why you think the two datasets show the trends they do.** Are these two datasets encoding the same type of similarity? How does the notion of similarity differ between them?

5 Qualitative Analysis (25 points)

In this component, you will analyze your word vectors qualitatively by computing nearest neighbors for particular query words and comparing them across window sizes. You may also find it helpful to manually inspect instances of certain words in the original corpus to find typical contexts for each word.

For this section, use PMI vectors with `vocab-15kws.txt` used to populate both V and V_C. Use two sets of PMI vectors, one with $w = 1$ and the other with $w = 6$. To find the k nearest neighbors for a query word $q \in V$, compute cosine similarity between q and all words in V and then keep the k words with highest cosine similarity. When doing so, omit the query word q from the list of nearest neighbors since it will always have a cosine similarity of 1.

5.1 (5 points) For the two window sizes $w = 1$ and $w = 6$, compute and print the 10 nearest neighbors for the query word *judges*. (Hint: using my implementation, the nearest neighbor for both window sizes is *judge*, followed by *justices* for $w = 1$ and *appeals* with $w = 6$. Your nearest neighbor lists may differ slightly from mine, but hopefully these words are high up in your lists.)

5.2 (10 points) Do nearest neighbors tend to have the same part-of-speech tag as the query word, or do they differ? Does the pattern differ across different part-of-speech tags for the query word? How does window size affect this? Explore these questions by choosing query words with different parts of speech and computing their nearest neighbors. When choosing query words, consider nouns, verbs, adjectives, and prepositions. (Hint: when considering verbs, use inflected forms like *transported*.) Try a few query words from each part of speech category and see if you can find any systematic patterns when comparing their nearest neighbors across window sizes 1 and 6. When the neighbors differ between window sizes, how do they differ? Can you find any query words that have almost exactly the same nearest neighbors with the two window sizes? **Discuss your findings, showing examples of nearest neighbors for particular words to support your claims.**

5.3 (10 points) Now try choosing words with multiple senses (e.g., *bank*, *cell*, *apple*, *apples*, *axes*, *frame*, *light*, *well*, etc.) as query words. What appears to be happening with multisense words based on the nearest neighbors that you observe? What happens when you compare the neighbors with different window sizes ($w = 1$ vs. $w = 6$)? **Discuss your findings, showing examples of nearest neighbors for particular words to support your claims.**