Practical 1: Word Vectors

Deep Natural Language Processing Class, 2022

Report due date - 14.03.2023

The practical report (pdf + code) should be sent via Moodle/Pegaz with a standard due dates policy.

1 Word Vectors

Word Vectors became a fundamental component for downstream NLP tasks, e.g. question answering, text generation, translation, etc., so it is important to build some intuitions as to their strengths and weaknesses. In this practical, you will explore two types of word vectors: those derived from co-occurrence matrices, and those derived via the word2vec algorithm.

Note on terminology: The terms "word vectors" and "word embeddings" are often used interchangeably. The term "embedding" refers to the fact that we are encoding aspects of a word's meaning in a lower dimensional space. As Wikipedia states, "conceptually it involves a mathematical embedding from a space with one dimension per word to a continuous vector space with a much lower dimension".

1.1 Count-based Word Vectors

Many word vector implementations are driven by simple heuristic that the meaning of the word can be extracted from the surrounding context. Moreover, similar word - (quasi)-synonyms, will be used in similar contexts as well. By examining these contexts, we can try to develop embeddings for our words. With this intuition in mind, many "old school" approaches to constructing word vectors relied on word counts. As an example, we will look into one of those strategies, co-occurrence matrices.

1.2 Co-Occurrence Matrix

A co-occurrence matrix counts how often things co-occur in some environment. Given some word occurring in the document, we consider *the context window* surrounding w_i. Supposing our fixed window size is n, then we define the n preceding and n subsequent words in that document as: $w_{i-n}, w_{i-n+1}, ..., w_{i-1}$ and $w_{i+1}, ..., w_{i+n}$. We build a co-occurrence matrix M, which is a symmetric word-by-word matrix in which M_{ij} is the number of times w_j appears inside w_i's window.

Note, In NLP, we often add START and END tokens to represent the beginning and end of sentences, paragraphs or documents. In this case we imagine START and END tokens

*	START	Ala	mieć	kot	i	pies	lubić	END
START	0	2	0	0	0	0	0	0
Ala	2	0	1	0	0	0	1	0
mieć	0	1	0	1	0	0	0	0
kot	0	0	1	0	1	0	1	1
i	0	0	0	1	0	1	0	0
pies	0	0	0	0	1	0	0	1
lubić	0	1	0	1	0	0	0	0
END	0	0	0	1	0	1	0	0

Figure 1: Example: co-cccurrence with a fixed window of $n = 1$. Document 1: "Ala ma kota i psa." and document 2: "Ala lubi kota.". Notice that the word in the table are in canonical forms. We will come back to the problem of morphologically-rich languages like Polish in next practicals.

encapsulating each document, e.g., "START Ala lubi kota.", and include these tokens in our co-occurrence counts.

The rows (or columns) of this matrix provide one type of word vectors (those based on word-word co-occurrence), but the vectors will be large in general (linear in the number of distinct words in a corpus). Thus, our next step is to run dimensionality reduction. In particular, we will run SVD (Singular Value Decomposition), which is a kind of generalized PCA (Principal Components Analysis) to select the top k principal components. This reduced-dimensionality co-occurrence representation preserves semantic relationships between words, e.g. doctor and hospital will be closer than doctor and dog.

If you can barely remember what an eigenvalue is, here's a slow, friendly introduction to SVD: `https://davetang.org/file/Singular_Value_Decomposition_Tutorial.pdf`. For the purpose of this class, you only need to know how to extract the k-dimensional embeddings by utilizing pre-programmed implementations of these algorithms from the `numpy`, `scipy`, or `sklearn` python packages. In practice, it is challenging to apply full SVD to large corpora because of the memory needed to perform PCA or SVD. However, if you only want the top$k$ vector components for relatively small k - known as Truncated SVD — then there are reasonably scalable techniques to compute those iteratively.

1.3 Plotting Co-Occurrence Word Embeddings

We will focus now on the corpus of frequency dictionary of contemporary Polish. The original purpose of the corpus was to create a general frequency dictionary of contemporary Polish. The work started in 1967. Partial results were published between 1972 and 1977, the completed dictionary in 1990. The corpus was later augmented in various respects, both by manual editing and automated procedures.

Corpus data contain $10,000$ samples divided into 5 parts: essays, news, scientific texts, fiction and plays. Every sample is approximately 50 words long, they all come from texts published between 1963 and 1967 and contain bibliographic description of its source. Each

word is tagged with its base form and some morphological properties. Sentence boundaries are also marked.

For more details, please see `http://clip.ipipan.waw.pl/PL196x`. We provide a `read_corpus_pl` function below that pulls out the data. The function also adds START and END tokens to each of the documents, and lowercases words.

Read the corpus and analyze what these documents are like.

a) Implement `distinct_words` function (1 point)

Write a method to work out the distinct words (word types) that occur in the corpus.

You can run sanity checks to test the implementation.

b) Implement `compute_co_occurrence_matrix` function (1.5 points)

Write a method that constructs a co-occurrence matrix for a certain window-size n with a default of 4, considering words n before and n after the word in the center of the window.

You can run sanity checks to test the implementation.

c) Implement `reduce_to_k_dim` function (0.5 point)

Construct a method that performs dimensionality reduction on the matrix to produce k-dimensional embeddings. Use SVD to take the top k components and produce a new matrix of k-dimensional embeddings.

All of `numpy`, `scipy`, and `scikit-learn` (sklearn) provide some implementation of SVD, but only `scipy` and `sklearn` provide an implementation of Truncated SVD, and only `sklearn` provides an efficient randomized algorithm for calculating large-scale Truncated SVD. So please use `sklearn.decomposition.TruncatedSVD`.

d) Implement `plot_embeddings` function ((0.5 point)

Write a function to plot a set of 2D vectors in 2D space. For graphs, you should use Matplotlib (plt).

e) Co-Occurrence Plot Analysis (1.5 points)

Now we will put together all the parts you have written. We will compute the co-occurrence matrix with fixed window of 4, over the PL1968x corpus. Then we will use TruncatedSVD to compute 2-dimensional embeddings of each word.

Run the functions `plot_unnormalized` and `plot_normalized` to produce two plots. It'll

probably take a few seconds to run. What clusters together in 2-dimensional embedding space? What doesn't cluster together that you might think should have?
TruncatedSVD returns $U \times S$, so we normalize the returned vectors in the second plot, so that all the vectors will appear around the unit circle. Is normalization necessary?

Answer in no more than 3 sentences per plot.

2 Prediction-Based Word Vectors

As discussed in class, more recently prediction-based word vectors have come into fashion, e.g. word2vec. Here, we shall explore the embeddings produced by word2vec. Please revisit the class notes and lecture slides for more details on the word2vec algorithm. It is highly recommended to try to read the original paper. Should you have any questions or problems, feel free to ask them during lectures/practicals or via email.

We will switch now to Polish word2vec embeddings. Download them from this link. They were produced by Sławomir Dadas. See more other Polish resources at his Github repository here.

Due to issues with `nltk` with Python versions above 3.5 we have to do a small change in the reader code. Replace the following line in the `_resolve` method in `YOUR-PYTHON-PATH/site-packages/nltk/corpus/reader/pl196x.py`:

```
if len(filter(lambda accessor: accessor is None,
(fileids, categories, textids))) != 1:
```

by:

```
if len(list(filter(lambda accessor: accessor is None,
(fileids, categories, textids)))) != 1:
```

a) Reducing dimensionality of Word2Vec Word Embeddings (1.5 points)

Let's directly compare the word2vec embeddings to those of the co-occurrence matrix. Run `get_matrix_of_vectors` and `reduce_to_k_dim` functions to reduce 300-Dimensinal word embeddings. Plot embeddings given the words and using previously created plotting function.

What clusters together in 2-dimensional embedding space? What doesn't cluster together that you might think should have? How is the plot different from the one generated earlier from the co-occurrence matrix?

2.1 Cosine Similarity

Now that we have word vectors, we need a way to quantify the similarity between individual words, according to these vectors. One such metric is cosine-similarity. We will be using this to find words that are "close" and "far" from one another.

We can think of n-dimensional vectors as points in n-dimensional space. If we take this perspective, L1 and L2 distances help quantify the amount of space "we must travel" to get between these two points. Another approach is to examine the angle between two vectors. From trigonometry we know that:

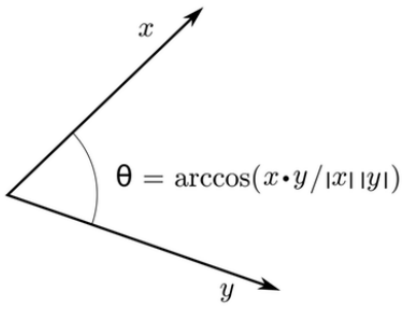

Figure 2: Cosine similarity.

Instead of computing the actual angle, we can leave the similarity in terms the cosine of the angle. Formally the Cosine Similarity between two vectors u and v is defined as:

$$\texttt{sim}(u, v) = \frac{u \cdot v}{||u||||v||}$$

b) Polysemous Words (1 point)

Find a polysemous word (for example, "stówa") such that the top-10 most similar words (according to cosine similarity) contains related words from both meanings. For example, "stówa" has both "wers" and "cent" in the top 10. You will probably need to try several polysemous words before you find one. Please state the polysemous word you discover and the multiple meanings that occur in the top 10. Why do you think many of the polysemous words you tried didn't work?

Note: You should use the `wv_from_bin_pl.most_similar(word)` function to get the top 10 similar words. This function ranks all other words in the vocabulary with respect to their cosine similarity to the given word. For further assistance please check the GenSim documentation.

c) Synonyms & Antonyms (1 point)

When considering Cosine Similarity, it's often more convenient to think of Cosine Distance, which is simply 1 - Cosine Similarity. Find three words (w_1, w_2, w_3) where w_1 and w_2 are synonyms and w_1 and w_3 are antonyms, but Cosine Distance(w_1, w_3) < Cosine Distance(w_1, w_2). For example, w_1="radosny" is closer to w_3="smutny" than to w_2="pogodny".

Once you have found your example, please give a possible explanation for why this counter-intuitive result may have happened.

You should use the the `wv_from_bin_pl.distance(w1, w2)` function here in order to compute the cosine distance between two words. Please see the GenSim documentation for further assistance.

Solving Analogies with Word Vectors

Word2Vec vectors have been shown to sometimes exhibit the ability to solve analogies. As an example, for the analogy:

"man : king :: woman : x", what is x?

The `most_similar` function finds words that are most similar to the words in the positive list and most dissimilar from the words in the negative list. The answer to the analogy will be the word ranked most similar (largest numerical value).

d) Finding Analogies (1 point)

Find an example of analogy that holds according to these vectors (i.e. the intended word is ranked top). In your solution please state the full analogy in the form x:y :: a:b. If you believe the analogy is complicated, explain why the analogy holds in one or two sentences.

Note: You may have to try many analogies to find one that works!

e) Incorrect Analogies (0.5 point)

Find an example of analogy that does not hold according to these vectors. In your solution, state the intended analogy in the form x:y :: a:b, and state the (incorrect) value of b according to the word vectors.

f) Guided Analysis of Bias in Word Vectors (0.5 point)

It's important to be aware of limitations implicit to our word embeddings. Run the code under Section f).

1) which terms are most similar to "kobieta" and "szef" and most dissimilar to "mezczyzna", and 2) which terms are most similar to "mezczyzna" and "prezes" and most dissimilar to "kobieta". What do you find in the top 10?

g) Independent Analysis of Bias in Word Vectors (1 point)

Use the `most_similar` function to find another case where some bias is exhibited by the vectors. Briefly explain the example of bias that you discover.

h) The source of bias in word vectors (0.5 point)

What might be the cause of these biases in the word vectors?

i) English versus Polish (5 points)

Load vectors for English and run similar analysis for points from b) to g) with exactly the same examples (with respect to translation). Have you observed any qualitative differences? Answer with up to 7 sentences.