(Compositional) Distributed Semantics Exercise Methods in Computational Linguistics

Franziska Weeber

December 05, 2025

This exercise is based on the material provided by Eva Maria Vecchi

Introduction

The main goal of this lab session is to explore compositional distributional word representations. We will start with subsets of pre-trained embeddings and produce phrase representations using various composition approaches discussed in class.

The participation in this lab session is voluntary: you do not have to submit any result. The work in this session should be done individually, but you are encouraged to discuss with your colleagues. If you want to get feedback, please ask questions during the session.

You do not need to install any additional packages for this exercise. Also, there is a python file on Ilias you can use to structure your code and to get both typing hints and doc strings. As always, you are free to choose any other implementation or to ignore this python file. There is not a single correct implementation!

Word Embeddings

We will use the subsets of pre-trained word embeddings provided on Ilias. You are welcome to work with just one embedding space, or compare performance across the two embedding spaces. Both spaces include an overlapping vocabulary defined as the top 2K most frequent words in the GoogleNews corpus.

These provided embeddings files include:

- w2v_top2kGN.tar.gz – a subset of the w2v embeddings covering the 2K most frequent words in the GoogleNews corpus (Mikolov et al., 2013, `https://code.google.com/archive/p/word2vec/`)
- GloVe_top2kGN.tar.gz – a subset of 1K word embeddings from the Common Crawl 42B-token GloVe vectors with 300 dims (Pennington et al., 2014, `https://nlp.stanford.edu/projects/glove/`)

You are also encouraged to use the word representations built in previous exercises.

Using `python`, load the embeddings spaces so you can access each word vector easily. I recommend a dictionary with words as keys and the embedding as a list of floats (numeric!) as values. You can of course also use another solution.

Vector arithmetics

In this section, we will perform simple vector arithmetic on word embeddings and then search our corpus for the word whose embedding is most similar to the resulting vector. To analyze the results, print the equation of words and the most similar word.

Define analogies

We will focus on analogies of the following form (with $\mathbf{v}_{\text{word}}$ denotes the word embedding for a word from one of the provided corpora):

$$\mathbf{v}_{\text{base\_term}} - \mathbf{v}_{\text{subtract\_term}} + \mathbf{v}_{\text{add\_term}} \approx \mathbf{v}_{\text{result\_term}}$$

So for example, we would expect the following analogy to result from our formula:

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$$

You can - and are encouraged to - test your own analogies. Just make sure all three input words are included in the 2,000 word sample provided to you.

If you don't want to analyze your own analogies, here are some examples to test:

- ("man", "woman", "he"),
- ("her", "him", "she"),
- ("father", "mother", "son"),
- ("students", "student", "school"),
- ("buying", "buy", "pay"),
- ("older", "old", "good"),
- ("fully", "full", "final"),

Find matching words

Now, calculate the result vector for the analogy task. You can use the function `compute_analogy_vector` for this. Given the three terms defined above, return the result vector of applying the analogy equation defined above. You calculate the result of the equation by applying the formula element-wise.

Second, you compute the similarity ranking of all words to this new vector. You can use the function `compute_similarity_ranking` that takes a vector $\mathbf{v}$ and the word to embedding

$\mathbf{v}_{\text{word}}$ dictionary as input and returns a list of $(\text{word}, \text{sim}(\mathbf{v}, \mathbf{v}_{\text{word}}))$ tuples, sorted such that the highest cosine similarity is first and the lowest is last, i. e., non-ascendingly. The code for the sorting is provided.

Use your functions on the analogies you previously defined and print the result. Are your results as expected?

Phrase Semantics

In this section, you will examine three different methods to compose word embeddings into phrase embeddings.

Load data and select vocabulary

Then select a handful of nouns, adjectives, and verbs. For example:

- **Nouns:** ["car", "child", "world", "business"]
- **Adjectives:** ["big", "small", "red", "bad"]
- **Verbs:** ["drive", "see"]

When selecting your own words, you need to make sure that each of these words has an embedding. You only got a sample that contains embeddings of the 2,000 most frequent words in the GoogleNews corpus. If the word is not in that sample, remove it from your list. There is a function provided by me that you can use for that.

Simple Phrase Composition

You will now form simple two-word phrases (e. g., *"red car"*, *"small business"*, *"see world"*) and compute their embeddings using different composition methods. I will use all adjective - noun combinations as well as all verb - noun combinations.

- **Addition** Element-wise addition of the component vectors to produce a composed phrase representation

$$\mathbf{v}_{\text{red car}} = \mathbf{v}_{\text{red}} + \mathbf{v}_{\text{car}}$$

- **Multiplication** Element-wise multiplication of component vectors.

$$\mathbf{v}_{\text{red car}} = \mathbf{v}_{\text{red}} \odot \mathbf{v}_{\text{car}}$$

- **Concatenation** This doubles the dimensionality but preserves each vector in its own "slot." Such an approach might facilitate learning a weighting mechanism over this higher-dimensional space for ML approaches.

$$\mathbf{v}_{\text{red car}} = \text{concat}(\mathbf{v}_{\text{red}}, \mathbf{v}_{\text{car}})$$

Store these phrase embeddings in the same way you stored the word embeddings.

Compare the Results

Here are some example analyses you can implement to compare the results. To answer the questions, print the results for all your target phrases.

- **Semantic Similarity to Constituents** Once you have produced phrase embeddings, calculate the similarity of the phrase vectors to the the embedding vectors of its constituents. Compare these similarities across different composition methods. Does *"red car"* end up closer to *"car,"* or closer to *"red"*? How does that differ for addition vs. multiplication?

$$\text{sim}(\mathbf{v}_{\text{red car}}, \mathbf{v}_{\text{car}}) \quad \text{and} \quad \text{sim}(\mathbf{v}_{\text{red car}}, \mathbf{v}_{\text{red}})$$

- **Nearest Neighborhoods** Find the **top k nearest neighbors** among all word embeddings. For example, compute cosine similarity between the phrase vector and every word vector in the embedding space, then sort by descending similarity. (Hint: If you followed the structure from the python file on Ilias, you already have a function for that).

  Compare how the top nearest neighbors differ across composition methods. E. g., does the additive vector for *"red car"* cluster near color-related words, while the multiplicative vector clusters more tightly around "car"-related words?

  What do these differences reveal about the kind of *meaning* each composition function is capturing?

References

Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.