Theory: Word2Vec

26 minutes 0 / 5 problems solved

Skip this topic Start practicing

50 users solved this topic. Latest completion was 16 minutes ago.

Current topic:

Word2Vec ···

Topic depends on:

✗ Intro to text representations ···

✗ Text corpora in NLTK ···

Table of contents:

Previously, we've learned the difference between *word embeddings* and count-based *text representations*. In addition, we outlined the general properties of the former models. Now it's time to get involved with Word2Vec, a popular instrument for providing the vectors.

§1. Word2Vec Basics

Word2Vec is an architecture for computing dense word representations. The underlying idea is that the word meaning heavily depends on the context, so Word2Vec approximates the sense of a word through vectors of its surrounding so that we could note synonyms and antonyms, as well as other types of linguistic phenomena and relations (for one, a popular example is that "*man*" is related to "*king*" as "*woman*" relates to "*queen*").

Word2Vec has two models for producing vectors:

- Continuous bag-of-words (CBOW) aims at predicting a word with the help of a given context. For example, if we have "*cat*", "*jumped*", "*over*", "*the*", the model will predict what words may appear along with them; say, the most frequent one will be "*puddle*" — the model will also account for a vector for this term.
- Continuous skip-gram, on the contrary, tries to determine a context with the given words. Following the same example, if we have the "*puddle*", its most common neighbors in the text can be "*cat*", "*jumped*", "*over*", and "*the*", and the model will compute representations for them.

Instead of counting the co-occurrences manually, we could train a model to directly predict if words share the same environment, and adjust the results with a few parameters. For example, we set the **context window size**: if it equals to five, then five words to the left and to the right from the given word will be taken as 'context'. We can also adjust other parameters, they will be described below.

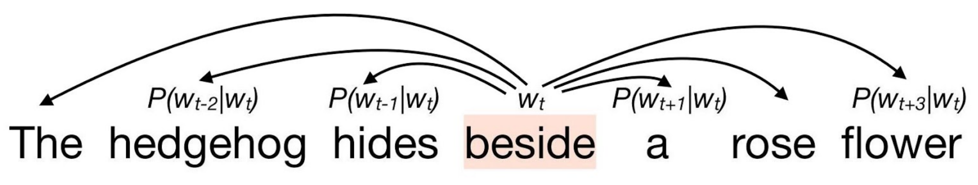
What does Word2Vec do? It calculates word probabilities in a corpus and uses them to create embeddings. As a result, all words from the vocabulary get their own vector representations. Similar words (that is, words appearing in similar contexts) will have similar vectors.

§2. The Prediction

In the following sections, we'll explain the algorithm behind both models in more detail.

First, the model constructs a vocabulary, as in count-based representations on a given corpus. The algorithm will estimate two embeddings for each vector: one for all contexts where the word is a neighbor to a central term, the other for all contexts where it is the central term. During the learning process, the co-occurrence probabilities for words within a defined context window will be increased so the words in the same context window will be closer in their representations.

Below, we illustrate a prediction task for the *skip-gram* model:



The central word, "*beside*", is denoted as w_t. The context window size equals ± 3; so the model will compute conditional probabilities for six pairs: $\mathrm{P}(The|beside)$, $\mathrm{P}(hedgehog|beside)$, $\mathrm{P}(hides|beside)$, $\mathrm{P}(a|beside)$, $\mathrm{P}(rose|beside)$, and $\mathrm{P}(flower|beside)$. Here, the essence of the conditional probabilities is to learn the likelihood of a word occurrence given another term ("*beside*" in our case). Note that the word order inside the context window doesn't influence the results.

Once we estimated representations for "*beside*", we can move one step forward: we take the indefinite article "*a*" as the central term and try to find the embeddings. Then we repeat these operations for all the following words in the corpus ("*rose*", "*flower*", and so on).

§3. Counting Probabilities

Speaking of conditional probabilities, how can we compute, for example, "*hides*" given "*beside*"? To do so, we need to measure the similarity of the corresponding vectors, for example, by taking the dot product.

$$dot\ product(a, b) = \sum_{i=1}^{N} a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_N b_N$$

As you see, we sum the products of all elements of two vectors, a and b. The *continuous bag-of-words* takes several embeddings of the context words and averages them to create a single a vector, while the *skip-gram* uses just one vector of the central term as a. In both cases, we'll take the dot product of this representation with a b embedding of another word from the vocabulary to predict how likely they will appear in the same context.

The dot product ranges from $-\infty$ to ∞, the higher it is, the more similar representations are. This metric is an efficient way to learn the vector similarity, but we need the actual probabilities, so then it is converted into a value between 0 and 1.

In a simple case, given some vector a, the model learns its dot product with each word in the corpus (so, each word's embedding will be used as b at some point). As a result, the *continuous bag-of-words* model returns one word (a central word) with the highest probability. In the *skip-gram*, the outcome is several context words, and the model treats each of them independently, counting probabilities of their observations one after another. So, to compute $\mathrm{P}([The, hedgehog, hides, a, rose, flower]|beside)$ and predict how likely these words appear within a range before and after "*beside*", we need to consider them separately: $\mathrm{P}(The|beside)$, ..., $\mathrm{P}(flower|beside)$.

§4. Adjusting Vectors and Parameters

With both models, before the actual learning process starts, we need to specify the vector length as a parameter, and the vectors of words are initialized with random weights. Why though? We can't tell in advance which words are related. The algorithm will iteratively adjust the values to bring the words in similar contexts closer and then increase the probability of their co-occurrence.

How exactly does the algorithm do this? We apply an objective function that measures how precise our prediction is and how far it is from the desired outcome. The algorithm evaluates errors in word vectors and then follows an update rule to optimize the objective function: the goal is to learn vectors that can be used to predict neighboring words (as you can remember, the embeddings of similar words are also similar). So, at each iteration of the model, the values are adjusted until they are close to this objective.

The results also depend on the learning parameters. The most important ones that we can set before the training are the following:

- The context window size that we mentioned earlier; a short context window means that the model learns embeddings from the immediate neighboring words, and this results in more syntactic embeddings (that is, the embeddings that tell us more about the compatible words in a context rather than of the possible synonyms);
- The number of dimensions, that is, the vector length, is usually set to a number less than a thousand, so the vectors come out rather short;
- The threshold on the frequency of words that the model ignores: the model takes the words appearing less than a defined value neither as the central term nor as a context. For example, if we set the threshold

as 10, the model will ignore all words occurring less than 10 times in a corpus.

- The **threshold on the frequency of words that the model downsamples**: you may remember that the words of higher frequency often contain little information, and that's a way to deal with them here; the model will emphasize the more important terms (also, it speed up the learning process by ignoring frequent words). Usually, this threshold is set as $t = 10^{-5}$. Then, we can count the probability of downsampling a w_i word in the corpus as shown below:
  $\mathrm{P}(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$
  where $f(w_i)$ is the word's frequency in the text (a probability of observing it). The higher this value is, the more likely w_i will be downsampled.
- The **training method** that defines what objective function we'll have; the popular **negative sampling** method is often used with the skip-gram model. We'll treat each actual context word as a positive training example (in the example above, "***rose***", "***flower***" are the 'correct' predictions for "***beside***"). Now, we'll find multiple negative training examples for each ***context word/central word*** pair. For instance, the terms "***cat***" and "***jumped***" aren't observed in the defined window, so we'll take them. Now, the objective is to make the model distinguish between these two cases. By doing so, we maximize the probability of seeing the central word with its context (by increasing the dot product) and minimize the probability of its occurrence with the non-context examples. This allows us to change a small number of values and speed up the process, that's why it's so widely used.

§5. Training Word2Vec in Gensim

Now, we'll consider the Word2Vec implementation in Gensim, a Python library for NLP. This library is designed for various word vector models and has a lot of convenient methods for text preprocessing and model construction.

First, we need to choose a corpus. Let's train the model with the Brown corpus, integrated with NLTK. We start by importing the required contents:

```
from gensim.models import Word2Vec
from nltk.corpus import brown
```

In general, Word2Vec doesn't require hand-labeled annotation, but it can be useful to restrict the vocabulary size, we can also perform text normalization. Another approach is to ignore the lower-frequency terms. We'll set a threshold on word frequency as a model parameter, `min_count`. We'll also set other parameters:

- `window` is the maximum distance to a context word from the given center,
- `size` is the length of embeddings,
- `sample` is a threshold on frequent word downsampling, in this case, it equals to `6e-5`, which is basically 6×10^{-5}, or 0.00006.
- `negative` is the number of negative training examples (this parameter implies that we use negative sampling as a training method).

```
w2v_model = Word2Vec(min_count=5,
                     window=5,
                     sg=0,
                     size=300,
                     sample=6e-5,
                     negative=20)
```

Note that by default, `negative` equals 0, and the model won't use it in the estimation. Also, there is the `sg` parameter that denotes the model we use. If we specified 1, it would be the skip-gram. Otherwise, the continuous bag-of-words will be applied.

Then, we need to pass the corpus as a list of sentences. We can easily divide it into such units using the NLTK corpus method `sents()`.

```
data = brown.sents()
```

Before model training, we need to build the vocabulary using `build_vocab()`, and then we can train the algorithm with the help of `train()`:

```
w2v_model.build_vocab(data)
w2v_model.train(data, total_examples=w2v_model.corpus_count, epochs=15)
```

For precise calculations, you should specify the number of sentences in the corpus with the `total_examples` parameter or the word counts with the `total_words` parameter (we did the first thing above). Another parameter, the `epochs`, is the number of iterations for the model. It significantly influences the results: iterations denote how many times the model adjusts the vectors.

You should keep in mind that Word2Vec models can take a long time to train, especially with large corpuses.

§6. Gensim Methods for Word2Vec

Finally, let's apply several methods to get the different results from the model.

- The `most_similar()` method in Gensim outputs the most similar terms for a word together with their probabilities. We can indicate how many predicted words we need through the `topn` parameter.

  ```
  w2v_model.most_similar('university', topn=5)
  # [('faculty', 0.9572603702545166),
  # ('colleges', 0.953723669052124),
  # ('student', 0.9516825079917908),
  # ('organizations', 0.9459941387176514),
  # ('universities', 0.9424967169761658)]
  ```

- The `similarity()` method computes how close given terms are and returns the probability of their co-occurrence.

  ```
  w2v_model.similarity('jazz', 'music')
  # 0.9146379
  ```

- The `doesnt_match()` method outputs a word that doesn't match with other words in a given sequence.

  ```
  w2v_model.doesnt_match(['pine', 'fir', 'coconut'])
  # coconut
  ```

In this case, we compared representations of three words, but it is possible to include more terms in the list.

We can save the results with:

```
w2v_model.save('word2vec_model')
```

If we want to use these embeddings later, we can load the model as follows:

```
model = Word2Vec.load('word2vec_model')
```

Of course, these are the most basic example of how you can use this implementation. You can explore it better by looking at the Gensim tutorial on Word2Vec.

§7. Pre-trained Embeddings

In Gensim, we can also work with the pre-trained word representations. Sometimes, it is easier to use them as your Word2Vec model can take too long to train.

NLTK has a sample of a bigger Word2Vec model trained on a hundred billion words from the Google News dataset. To access it, we will import the following contents:

```
import gensim
from nltk.data import find
```

We need the `find()` method, which takes the path to the file as its argument. With its help, we can access the model as follows:

```
word2vec_sample = find('models/word2vec_sample/pruned.word2vec.txt')
```

Now, we'll use the `KeyedVectors` structure. It allows us to access the word embeddings in the trained model by the word corresponding to this vector. We'll load the representations via `load_word2vec_format()`:

```
w2v_model = gensim.models.KeyedVectors.load_word2vec_format(word2vec_sample, binar
y=False)
```

The keyword `False` in the `binary` parameter indicates that we access plaintext data. Alternatively, it could be `True`, if the data was previously saved in the binary format.

We can apply the very same methods to the model and then compare its results with the ones from the model we trained. In this example, we use `most_similar()` to learn about resembling terms for "*university*".

```
w2v_model.most_similar('university', topn=5)
# [('universities', 0.7003918886184692),
# ('faculty', 0.6780906915664673),
# ('undergraduate', 0.6587096452713013),
# ('campus', 0.6434987783432007),
# ('college', 0.638526976108551)]
```

Note that the probabilities are lower here than the ones we obtained from our model. It is so because the NLTK model was trained on a larger number of contexts, and in general there were more examples with the "*university*" in the dataset.

§8. Conclusion

Now, let's go over the main points we've touched in this topic. Word2Vec is an architecture for computing dense vectors; it provides two models for this purpose, the continuous bag-of-words, and the continuous skip-gram. The former one predicts a word given a context, the latter model does the opposite — it takes a word and learns a context through its representation. The algorithms are similar to each other:

- We initialize random vectors for each term in the vocabulary;
- On the basis of the word's usage in a corpus, we adjust its embeddings to make them similar to vectors of neighboring words. We use the dot product to measure the similarity of representations, and the higher it is, the better;
- We repeat this operation for every word from the dictionary.

Moreover, we took a look at the Word2Vec implementation in Gensim. It allows us to train the algorithm with either of two models and use several metrics to approximate the meaning more precisely. At the end of the topic, we also discussed the possibility to use pre-trained embeddings in Gensim. It can be useful when one doesn't have a large dataset or their own model takes too long to train.

Report a typo

6 users liked this theory. 1 didn't like it. What about you?

Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion