# Word2Vec For Phrases — Learning Embeddings For More Than One Word

Moshe Hazoom Follow

Dec 22, 2018 · 12 min read


Photo by Alexandra on Unsplash

## How to learn similar terms in a given unsupervised corpus using Word2Vec

When it comes to semantics, we all know and love the famous Word2Vec [1] algorithm for creating word embeddings by distributional semantic representations in many NLP applications, like NER, Semantic Analysis, Text Classification and many more.

However, the limitation of the current implementation of Word2Vec algorithm is its **uni-gram natural behavior**. In Word2Vec, we are trying to predict a given word based on its context (CBOW), or predicting a surrounding context based on a given word (Skip-Gram). But what if we would like to embed the term "American Airlines" as its whole? In this post I will explain how to create embeddings for more than uni grams using unsupervised text corpus. If you are familiar with Word2Vec algorithm and word embeddings, you can skip the first part of this post.

Specifically, we will cover:

1. Introduction on words representation in NLP tasks.
2. The Distributional Hypothesis [2] and Word2Vec algorithm.
3. Learning phrases from unsupervised text.
4. How to extract similar phrases to a given phrase.

## Background

The current company I work for, Amenity Analytics, is building Text Analytics products while focusing on the Finance domain. It helps businesses get actionable insights on huge scale. Recently, we release a new search engine based on Elastic Search to help our clients get a more precise and focused view on their data. After looking into users' queries in the search engine, we noticed that many clients are searching for financial terms, while naively performing a Full Text Search with the query is not good enough. For example, one term that came up many times in users' searches is "Inflection Point".

Looking for the definition of "Inflection Point" in Investopedia:

"An inflection point is an event that results in a significant change in the progress of a company, industry, sector, economy or geopolitical situation and can be considered a turning point after which a dramatic change, with either positive or negative results, is expected to result"

Our clients want to see significant events in the companies they are following, thus, we need to search for more terms with the same meaning as “Inflection Point”, like “Turning Point”, “Tipping point”, etc.

. . .

## From Discrete Symbols to Continuous Vectors

### Words Representation

The most granular objects in language are characters, which forms words, or tokens. Words (and character) are discrete and symbolic. There is no way to tell that “Labrador” and “dog” are somehow related to each other just by looking on the words as is, or looking on the characters that compose them.

### Bag of Words (BOW)

The most common feature extraction for NLP tasks is bag-of-words (BOW) approach. In bag-of-words, we look at the histogram of word occurrences in a given corpus, without considering the order. Often, we look for more than just one word, but also on bi-grams (“I want”), tri-gram (“I want to”), or n-grams in the general case. It’s a common approach to normalize the counts for each word because the documents can differ in length (in most cases).

$$tf_{i,j} = \frac{c_{i,j}}{\sum_{w_{i'} \in D_j} c_{i',j}}$$

$$c_{i,j} = count\ of\ term\ i\ in\ document\ j$$

Normalized BOW.

One of the main drawbacks of BOW representation is that it’s discrete and cannot capture semantic relationship between words.

### Term Frequency — Document Inverse Frequency (TF-IDF)

One of the outcomes of BOW representation is that it gives a score for words that appeared many times, but many of them don't give any meaningful information, like "to" and "from". We want to distinguish between words that appear many times and are commons words to words that appear many times but gives information about the specific document. Weighting the BOW vectors is a common practice and one of the most used weighting approach is TF-IDF (Manning et al., 2008).

$$W_{i,j} = tf_{i,j} * \log(\frac{N}{df_i})$$

$$tf_{i,j} = number\ of\ occurences\ of\ term\ i\ in\ document\ j$$

$$df_i = number\ of\ documents\ containing\ term\ i$$

$$N = total\ number\ of\ documents$$

TF-IDF weighting formula. There are many variations for TF-IDF, one can read more about it here.

However, both BOW and TF-IDF cannot capture the semantic meaning of words, because they represents words, or n-grams, in a discrete way.

## Learning Word From its Context

### "Tell me who your friends are, and I will tell you who you are."

**The Distributional Hypothesis** is that words that occur in the same contexts tend to have similar meanings [2]. It's the basis for semantic analysis of text. The idea behind the hypothesis, is that we can learn words meaning by looking on the context they appear at. One can easily tell that the word "play" in the sentence "The boy loves to play outside" has a different meaning than the word "play" in the sentence "The play was fantastic". In general, words that are close to the target word are more informative, but in some cases there are long dependencies in the sentences between the target word and words that "far" from it. Many approaches for learning word from its context have been

developed during the years, among them the famous Word2Vec, which will be covered in this post because its massive popularity both in the academia and the industry.

### Word2Vec

The Distributional Hypothesis is the main idea behind Word2Vec. In Word2Vec, we have a large unsupervised corpus and for each word in the corpus, we try to predict it by its given context (CBOW), or trying to predict the context given a specific word (Skip-Gram). Word2Vec is a (shallow) neural network with one hidden layer (with dimension d) and optimization function of Negative-Sampling or Hierarchical Softmax (One can read this paper for more details). The training phase we iterate through the tokens in the corpus (the target word) and look at a window of size k (k words to each side of the target word, with the values between 2–10 in general).

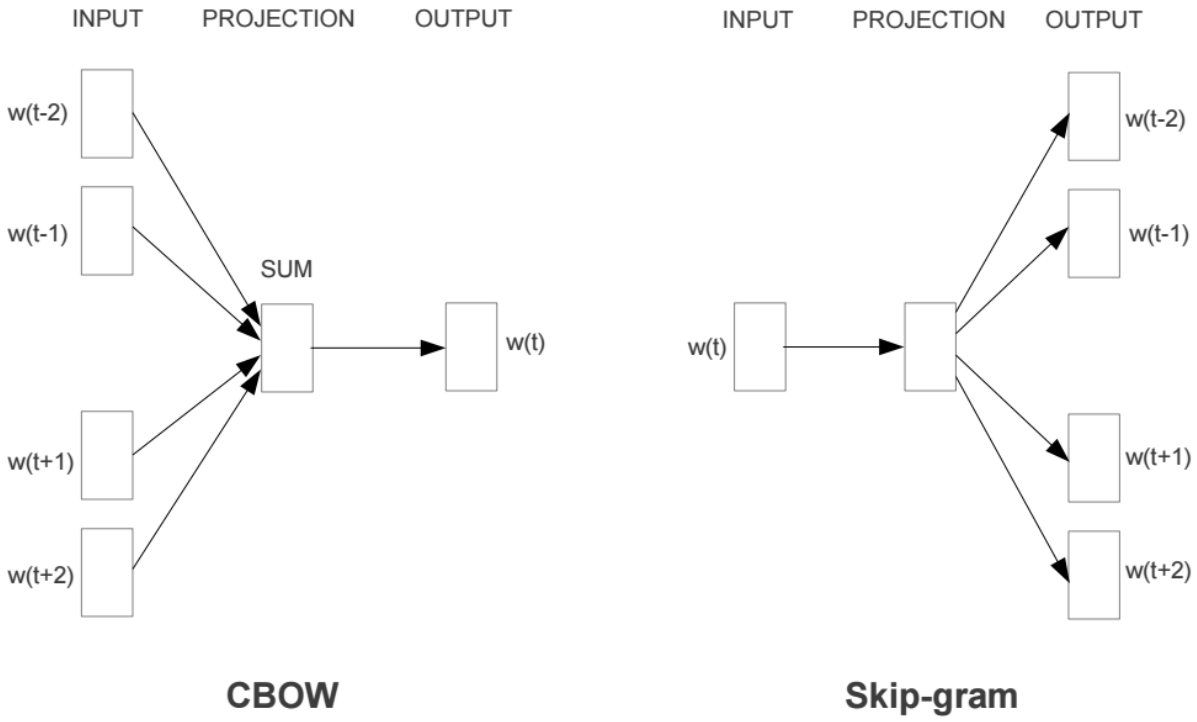


Image source

At the end of the training, we will get from the network the following embedding matrix:

$$E \in R^{|V|xd}$$

$$|V| = vocabulary\ size$$

Emdedding matrix after Word2Vec training

Now, each word will not be represented by a discrete and sparse vector, but by a d-dimension continuous vector, and the meaning of each word will be captured by its relation to other words [5]. The reason behind this is that in training time, if two target words share the some context, intuitively the weight of the network for this two target words will be close to each other and thus their matching vectors. Thus, we get a distributional representation for each word in the corpus, in contrast to count based approaches (like BOW and TF-IDF). Because of the distributional behavior, a specific dimension in the vector doesn't give any valuable information, but looking the (distributional) vector as a whole, one can perform many similarity tasks. For example, we get that V("King")-V("Man")+V("Woman) ~= V("Queen") and V("Paris")-V("France)+V("Spain") ~= V("Madrid"). In addition, we can perform similarity measures, like cosine-similarity, between the vectors and get that the vector of the word "president" will be close to "Obame", "Trump", "CEO", "chairman", etc.

As seen above, we can perform many similarity tasks on words using Word2Vec. But, as we mentioned above, we want to do the same for more than one word.

. . .

## Learning Phrases From Unsupervised Text (Collocation Extraction)

We can easily create bi-grams with our unsupervised corpus and take it as an input to Word2Vec. For example, the sentence "I walked today to the park" will be converted to "I_walked walked_today today_to to_the the_park" and each bi-gram will be treated as a uni-gram in the Word2Vec training phrase. It will work, but there are some problems with this approach:

1. It will learn embeddings **only** for bi-grams, while many of this bi-grams are not really meaningful (for example, "walked_today") and we will miss embeddings for

uni-gram, like “walked” and “today”.

2. Working only with bi-grams creates a very sparse corpus. Think for example about the above sentence “I walked today to the park”. Let’s say the target word is “walked_today”, this term is not very common in the corpus and we will not have many context examples to learn a representative vector for this term.

So, how we overcome this problem? how do we extract only meaningful terms while keeping words as uni-gram if their mutual information is strong enough? As always, the answer is inside the question — **mutual information**.

**Mutual Information (MI)**

Mutual information between two random variables X and Y is a measure of the dependence between X and Y. Formally:

$$MI(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) * \log \frac{p(x,y)}{p(x)p(y)}$$

$$p(x,y) = joint\ probablity\ of\ x\ and\ y$$

$$p(x) = probablity\ of\ x$$

Mutual Information (MI) of random variables X and Y.

In our case, X and Y represents all bi-grams in corpus such that y comes right after x.

**Pointwise Mutual Information (PMI)**

PMI is a measure of the dependence between a concrete occurrences of x of y. For example: x=walked, y=today. Formally:

$$PMI(x;y) = \log \frac{p(x,y)}{}$$

$$p(x)p(y)$$

PMI of concrete occurrences of x and y.

It's easy to see that when two words x and y appear together many times, but not alone, PMI(x;y) will have a high value, while it will have a value of 0 if x and y are completely independent.

**Normalized Pointwise Mutual Information (NPMI)**

While PMI is a measure for the dependence of occurrences of x and y, we don't have an upper bound on its values [3]. We want a measure that can be compared between all bi-grams, thus we can choose only bi-grams above a certain threshold. We want the PMI measure to have a maximum value of 1 on perfectly correlated words x and y. Formally:

$$NPMI(x;y) = \frac{\log \frac{p(x,y)}{p(x)p(y)}}{-\log p(x,y)}$$

Normalized Pointwise Mutual Information of x and y.

**Data-driven Approach**

Another way to extract phrases from text is by using the next formula [4] that takes into account the uni-grams and bi-grams count and a discounting coefficient for preventing of creation of bi-grams of too rare words. Formally:

$$score(x;y) = \frac{count(x,y) - \delta}{count(x) * count(y)}$$

Read this article for more details.

## Extract Similar Phrases

Now that we have a way to extract meaningful bi-grams from out large unsupervised corpus, we can replace bi-grams with a NPMI above a certain threshold to one uni-gram, for example: “inflection point” will be transformed to “inflection_point”. It’s easy to create tri-grams by using the transformed corpus with bi-grams and running again the process (with a lower threshold) for form tri-grams. Similarly, we can continue this process to n-grams with a decreasing threshold.

Our corpus consists of ~60 million sentences that contain 1.6 billion words in total. It took us 1 hour to construct bi-grams using the data-driven approach. Best results achieved with a threshold of 7 and a minimum term count of 5.

We measured the results using an evaluation set that contains important bi-grams that we want to identify, like financial terms, people names (mostly CEOs and CFOs) cities, countries, etc. The metric we used is a simple recall: from our extracted bi-grams, what is the coverage in the evaluation test. In this specific task, we care more about the recall instead of the precision so we allowed our self to use a relatively small threshold when extracting the bi-grams. We do take in consider that our precision might get worse when lowering the threshold and in turn we might extract bi-grams that are not very valuable, but that’s preferable than missing important bi-grams, when performing Query Expansion task.

**Example Code**

Reading corpus line by line (we assume each line contain one sentence) in a memory efficient approach:

```
def get_sentences(input_file_pointer):
    while True:
        line = input_file_pointer.readline()
        if not line:
            break

        yield line
```

Clean sentences by trimming leading and trailing spaces, lower case, remove punctuation, remove unnecessary characters and reduce duplicate space into a single space (note that this is not really necessary because we later on tokenize our sentence by space character):

```
import re

def clean_sentence(sentence):
    sentence = sentence.lower().strip()
    sentence = re.sub(r'[^a-z0-9\s]', '', sentence)
    return re.sub(r'\s{2,}', ' ', sentence)
```

Tokenize each line by a simple space delimiter (more advanced techniques for tokenization exist, but tokenize by a simple space gave us good results and works well n practice), and remove stop-words. Removing stop-words is task dependent and in some NLP tasks, keeping the stop-words yields better results. One should evaluate both approaches. For this task, we used Spacy's stop-word set.

```
from spacy.lang.en.stop_words import STOP_WORDS

def tokenize(sentence):
    return [token for token in sentence.split() if token not in
STOP_WORDS]
```

Now, that we have a representations of our sentences by a 2-d matrix of cleaned tokens, we can build bi-grams. We will use Gensim library that is really recommended for NLP semantic tasks. Fortunately, Genim has an implementation for phrases extraction, both with NPMI and the above data-driven approach of Mikolov et al. One can control the hyperparameters easily, like determining the minimum term count, threshold and scoring ('default' for data-driven approach and 'npmi' for NPMI). Note that values are different between the two approaches and one needs to take it into account.

```
from gensim.models.phrases import Phrases, Phraser

def build_phrases(sentences):
    phrases = Phrases(sentences,
```

```
                        min_count=5,
                        threshold=7,
                        progress_per=1000)
    return Phraser(phrases)
```

After we finish building the phrases model, we can save it easily and load it later:

```
phrases_model.save('phrases_model.txt')

phrases_model= Phraser.load('phrases_model.txt')
```

Now that we have a phrases model, we can use it to extract bi-grams for a given sentence:

```
def sentence_to_bi_grams(phrases_model, sentence):
    return ' '.join(phrases_model[sentence])
```

We want to create, based on our corpus, a new corpus with meaningful bi-grams concatenated together for later use:

```
def sentences_to_bi_grams(n_grams, input_file_name,
output_file_name):
    with open(input_file_name, 'r') as input_file_pointer:
        with open(output_file_name, 'w+') as out_file:
            for sentence in get_sentences(input_file_pointer):
                cleaned_sentence = clean_sentence(sentence)
                tokenized_sentence = tokenize(cleaned_sentence)
                parsed_sentence = sentence_to_bi_grams(n_grams,
tokenized_sentence)
                out_file.write(parsed_sentence + '\n')
```

. . .

## Learning Similar Phrases

After this above phrase, our corpus contains phrases and we can use it as input into Word2Vec training (maybe changing hyper-parameters will be necessary) , same like before. The training phrase will consider “inflection_point” as one word and will learn a distributed d-dimensional vector that will be close to the vectors of terms like “tipping_point” or “inflection”, which is our goal!

On our 1.6 billion words corpus, it took us 1 hour to construct bi-grams and another 2 hours to train Word2Vec (with batch Skip-Gram, 300 dimension, 10 epochs, context of k=5 , negative sampling of 5, learning rate of 0.01 and minimum word count of 5) on a machine with 16 CPUs and 64 RAM using AWS Sagemaker service. A great Notebook example of how to use AWS Sagemaker service to train Word2Vec can be found here.

One can also use Gensim library to train Word2Vec model, for example here.

For example, when giving the term “Inflection Point”, we get back the following related terms, ordered by their cosine-similarity score from their represented vector and the vector of “inflection_point”:


```
"terms": [
        {
          "term": "inflection",
          "score": 0.741
        },
        {
          "term": "tipping_point",
          "score": 0.667
        },
        {
          "term": "inflexion_point",
          "score": 0.637
        },
        {
          "term": "hit_inflection",
          "score": 0.624
        },
        {
          "term": "inflection_points",
          "score": 0.606
        },
        {
          "term": "reached_inflection",
          "score": 0.583
        },
        {
```

```
            "term": "cusp",
            "score": 0.567
          },
          {
            "term": "reaching_inflection",
            "score": 0.546
          },
          {
            "term": "reached_tipping",
            "score": 0.518
          },
          {
            "term": "hitting_inflection",
            "score": 0.501
          }
        ]
```

Some of our clients wanted to see the affect of Black Friday on companies' sales, so when giving the term "Black Friday" we get:

```
"terms": [
          {
            "term": "cyber_monday",
            "score": 0.815
          },
          {
            "term": "thanksgiving_weekend",
            "score": 0.679
          },
          {
            "term": "holiday_season",
            "score": 0.645
          },
          {
            "term": "thanksgiving_holiday",
            "score": 0.643
          },
          {
            "term": "valentine_day",
            "score": 0.628
          },
          {
            "term": "mother_day",
            "score": 0.628
          },
          {
            "term": "christmas",
            "score": 0.627
          },
          {
```

```
      "term": "shopping_cyber",
      "score": 0.612
    },
    {
      "term": "holiday_shopping",
      "score": 0.608
    },
    {
      "term": "holiday",
      "score": 0.605
    }
  ]
```

Pretty cool, isn't it?

. . .

## Conclusion

In this post we covered different approaches for word representation in NLP tasks (BOW, TF-IDF and Word Embeddings), learnt how to learn word representation from its context using Word2Vec, saw how we can extract meaningful phrases from a given corpus (NPMI and data-driven approach) and how to transform a given corpus in order to learn similar terms/words for each one of extracted terms/words using Word2Vec algorithm. The results of this process can be used in a downstream task, like Query Expansion in Information Extraction tasks, Document Classification, Clustering, Question-Answering and many more.

Thanks for reading!

## References

[1] Mikolov, T., Chen, K., Corrado, G.S., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *CoRR, abs/1301.3781*.

[2] Harris, Z. (1954). Distributional structure. *Word*, 10(23): 146–162.

[3] Bouma, G. (2009). Normalized ( Pointwise ) Mutual Information in Collocation Extraction.

[4] Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *NIPS*.

[5] Goldberg, Y., Hirst, G., Liu, Y., & Zhang, M. (2017). Neural Network Methods for Natural Language Processing. *Computational Linguistics, 44*, 193–195.

)

Machine Learning NLP Deep Learning Python Towards Data Science