Word embeddings: exploration, explanation, and exploitation (with code in Python)

Galina Olejnik Follow
Dec 3, 2017 · 14 min read

Word embeddings discussion is the topic being talked about by every natural language processing scientist for many-many years, so don't expect me to tell you something dramatically new or 'open your eyes' on the world of word vectors. I'm here to tell some

basic things on word embeddings and describe the most common word embeddings techniques with formulas explained and code snippets attached.

So, as every popular data science book or blog post should always say after introduction part, let's dive in!

Informal definition

As the Wikipedia will point out, word embedding is

> *'the collective name for a set of language modeling and feature learning techniques in natural language processing (NLP) where words or phrases from the vocabulary are mapped to vectors of real numbers'.*

Strictly speaking, this definition is absolutely correct but gives not-so-many insights if the person reading it has never been into natural language processing or machine learning techniques. Being more informal, I can state that word embedding is

> *the vector, which reflects the structure of the word in terms of morphology (Enriching Word Vectors with Subword Information) / word-context(s) representation (word2vec Parameter Learning Explained) / global corpus statistics (GloVe: Global Vectors for Word Representation) / words hierarchy in terms of WordNet terminology (Poincaré Embeddings for Learning Hierarchical Representations) / relationship between a set of documents and the terms they contain (Latent semantic indexing) / etc.*

The idea behind all of the word embeddings is to capture with them as much of the semantical/morphological/context/hierarchical/etc. information as possible, but in practice one methods are definitely better than the other for a particular task (for instance, LSA is quite effective when working in low-dimensional space for the analysis of incoming documents from the same domain zone as the ones, which are already processed and put into the term-document matrix). The problem of choosing the best embeddings for a particular project is always the problem of try-and-fail approach, so realizing why in particular case one model works better than the other sufficiently helps in real work.

In fact, reliable word representation with real-number vector is the goal we're trying to approach. Sounds easy, isn't it?

One-hot encoding (CountVectorizing)

The most basic and naive method for transforming words into vectors is to count occurrence of each word in each document, isn't it? Such an approach is called countvectorizing or one-hot encoding (dependent on the literature).

The idea is to collect a set of documents (they can be words, sentences, paragraphs or even articles) and count the occurrence of every word in them. Strictly speaking, the columns of the resulting matrix are words and the rows are documents.

The code snippet attached is the basic sklearn implementation, full documentation can be found here.

```
from sklearn.feature_extraction.text import CountVectorizer
# create CountVectorizer object
vectorizer = CountVectorizer()

corpus = [
          'Text of first document.',
          'Text of the second document made longer.',
          'Number three.',
          'This is number four.',
]
# learn the vocabulary and store CountVectorizer sparse matrix in X
X = vectorizer.fit_transform(corpus)

# columns of X correspond to the result of this method
vectorizer.get_feature_names() == (
    ['document', 'first', 'four', 'is', 'longer',
     'made', 'number', 'of', 'second', 'text',
     'the', 'this', 'three'])

# retrieving the matrix in the numpy form
X.toarray()

# transforming a new document according to learn vocabulary
vectorizer.transform(['A new document.']).toarray()
```

For instance, to the word 'first' in the given example corresponds vector *[1,0,0,0]*, which is the 2nd column of the matrix *X*. Sometimes the output of this method is called 'sparse matrix' as long as *X* has zeros as the most elements of it and has sparsity as its feature.

TF-IDF transforming

The idea behind this approach is term weighting by exploitation of useful statistical measure called *tf-idf*. Having a large corpus of documents, words like 'a', 'the', 'is', etc. occur very frequently, but they don't carry a lot of information. Using one-hot encoding approach we see that vectors of these words are not-so-sparse, claiming, that these words are important and carry a lot of information if being in so many documents. One of the ways to solve this problem is stopwords filtering, but this solution is discrete and not flexible to the domain zone we're working with.

A native solution to the stopwords issue is using statistical quantity, which looks like:

$$tfidf(term, document) = tf(term, document) \cdot idf(term)$$

The first part of it is *tf*, which means 'term frequency'. By saying it we simply mean the number of times the word occurs in the document divided by the total number of words in the document:

$$tf(term, document) = \frac{n_i}{\sum_{k=1}^{V} n_k}$$

The second part is *idf*, which stands for 'inverse document frequency', interpreted like inversed number of documents, in which the term we're interested in occurs. We're also taking logarithm of this component:

$$idf(term) = \log \frac{N}{n_t}$$

We're done with the formula, but how do we use it? In our previous method reviewed, we're having word as a column *j* occurring *n* times in the document as a row *i*. We're taking the same CountVectorizer matrix calculated earlier and replacing each cell of it by tf-idf score for this term and this document.

```
from sklearn.feature_extraction.text import TfidfTransformer

# create tf-idf object
transformer = TfidfTransformer(smooth_idf=False)
```

```
# X can be obtained as X.toarray() from the previous snippet
X = [[3, 0, 1],
     [5, 0, 0],
     [3, 0, 0],
     [1, 0, 0],
     [3, 2, 0],
     [3, 0, 4]]

# learn the vocabulary and store tf-idf sparse matrix in tfidf
tfidf = transformer.fit_transform(counts)

# retrieving matrix in numpy form as we did it before
tfidf.toarray()
```

Full documentation on this sklearn class can be found here, there are many interesting parameters there that can be used.

Word2Vec (word2vec parameter learning explained)

As I would say, here the fun begins! Word2Vec is the first neural embedding model (or at least the first, which gained its popularity in 2013) and still the one, which is used by the most of researchers. Doc2Vec, its child, is also the most popular model for paragraphs representation, which was inspired by Word2Vec. In fact, many of the concepts we will be reviewing later are based on the Word2Vec prerequisites, so be sure to pay enough attention to this embeddings type.

There are 3 different types of Word2Vec parameter learning, and all of them are based on the neural network model, so this paragraph will be created with the assumption, that you know what it is.

One-word context

The intuition behind it is the fact that we're considering one word per one context (we're predicting one word given only one word); this approach is often referred to as CBOW model. The architecture of our neural network is that we're having a one-hot encoded vector as the input of size $V \times 1$, input $\rightarrow$ hidden layer weights matrix W of size $V \times N$, hidden layer $\rightarrow$ output layer weights matrix W' of size $N \times V$ and softmax function as a final activation step. Our goal is to calculate the following probability distribution, which is the vector representation of the word with index I:

$$p(w_j|w_I)$$

We're assuming that we call our input vector *x*, with all zeros in it and only one 1 at the position *k*. Hidden layer *h* is computed with:

$$h = W^T x := v_{W_I}^T$$

Speaking about this notation, we're can consider *h* to be 'input vector' of the word *x*. Every word in our vocabulary has input and output representations; formally, row *i* of weights matrix *W* is our 'input' vector representation of the word *i*, so we're using colon sign to avoid misunderstandings.

As the next step of the neural network, we take vector *h* and do the following computations:

$$u_j = {v'_{w_j}}^T h$$

Our *v'* is the output vector of the word *w* with index *j*, and for every entry *u* with index *j* we do this multiplication operation.

As we've said before, the activation step is calculated with standard softmax (negative sampling or hierarchical softmax techniques are welcome):

$$p(w_j|w_I) = y_j = \frac{exp(u_j)}{\sum_{j'=1}^{V} exp(u_{j'})}$$

The diagram on the method captures all of the steps described.

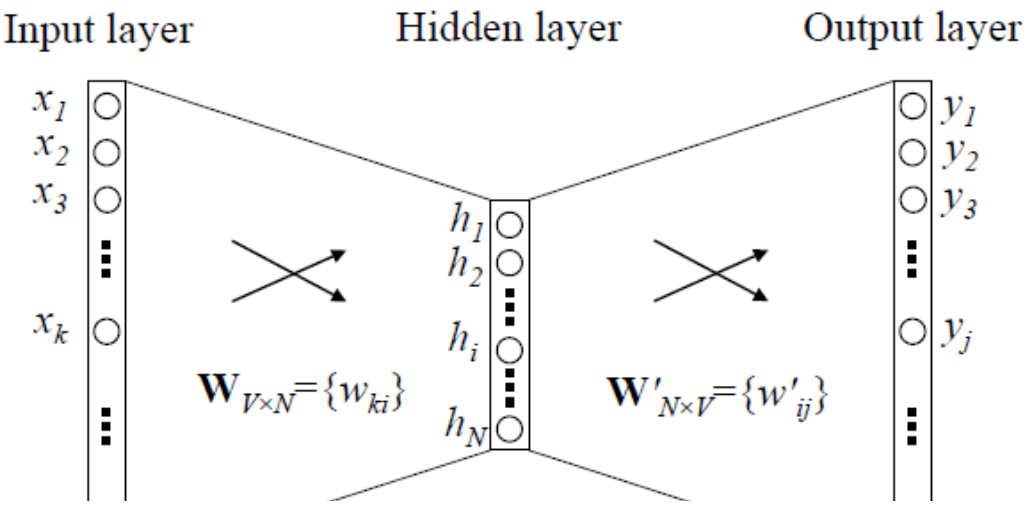


Figure 1: A simple CBOW model with only one word in the context

Multi-word context

This model has no differences from the one-word context, except the type of probability distribution we want to obtain and the type of hidden layer we're having. Interpretation of multi-word context is the fact that we'd like to predict multinomial distribution given not only one context word but rather many of them to store information about the relation of our target word to other words from the corpus.

Our probability distribution now looks this way:

$$p(w_O|w_{1,1}, ..., w_{i,c})$$

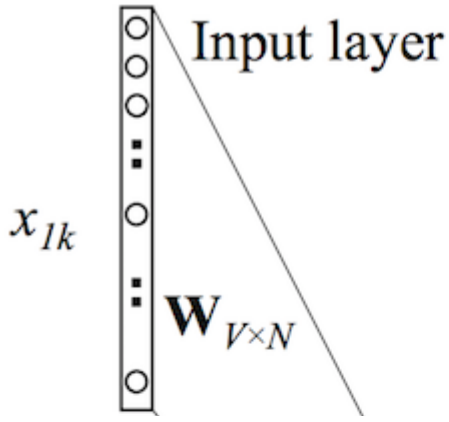
To obtain it, we're changing our hidden layer function to:

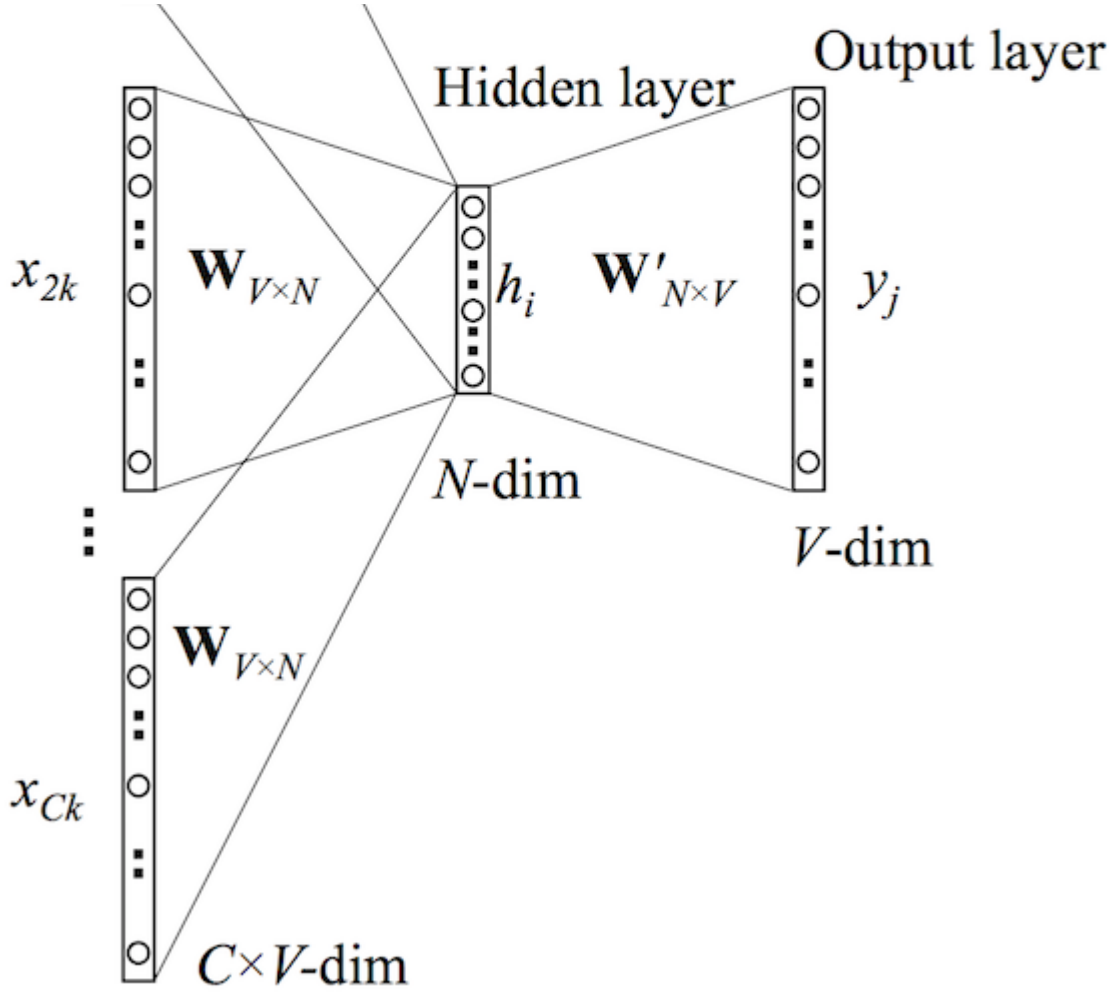
$$h = \frac{1}{C}W(x_1 + x_2 + ... + xc)$$

Which is simply the average of our context vectors x from 1 to C. Cost function now takes the form of:

$$-\log p(w_O|w_{1,1}, ..., w_{i,c})$$

All of the other components are the same for this architecture.





Skip-gram model

Imagine the situation opposite to CBOW multi-word model: we'd like to predict *c* context words having one target word on the input. Then, our objective we're trying to approach changes dramatically:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c}\log p(w_{t+j}|w_t)$$

-c and *c* are limits of our context window and word with index *t* is every word from the corpus we're working with.

Our first step we're doing to obtain hidden layer is the same as for two previous cases:

$$h = W^T x := v_{W_I}^T$$

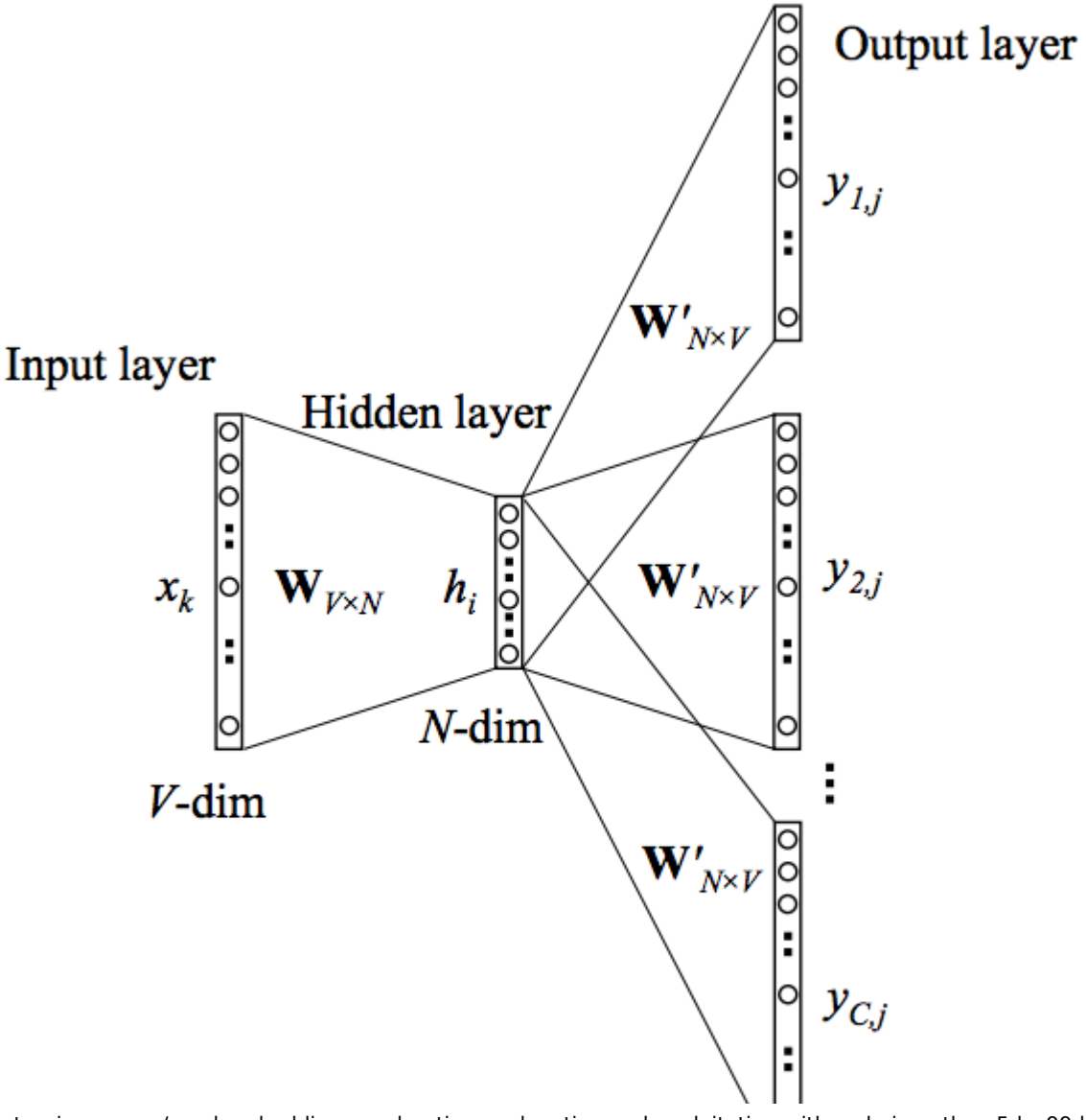
Our output layer (without activation) is calculated with:

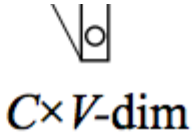
$$u_{c,j} = u_j = {v'_{w_j}}^T h$$

On the output layer, we're computing c multinomial distribution; each output panel shares the same weights from the hidden layer → output layer weights matrix *W'*. As the activation of output we're also using softmax with a bit of changed notation according to rather *c* panels, but not one output panel as we had earlier:

$$p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = \frac{exp(u_{c,j})}{\sum_{j'=1}^{V} exp(u_{j'})}$$

Illustration on the skip-gram calculation replicates all of the stages performed.





Basic implementation of Word2Vec model can be performed with gensim; full documentation is here.

```
from gensim.models import word2vec

corpus = [
          'Text of the first document.',
          'Text of the second document made longer.',
          'Number three.',
          'This is number four.',
]
# we need to pass splitted sentences to the model
tokenized_sentences = [sentence.split() for sentence in corpus]

model = word2vec.Word2Vec(tokenized_sentences, min_count=1)
```

GloVe (Glove: Global Vectors for Word Representation)

The approach of global word representation is used to capture the meaning of one word embedding with the structure of the whole observed corpus; word frequency and co-occurence counts are the main measures on which the majority of unsupervised algorithms are based on. GloVe model trains on global co-occurrence counts of words and makes a sufficient use of statistics by minimizing least-squares error and, as result, producing a word vector space with meaningful substructure. Such an outline sufficiently preserves words similarities with vector distance.

To store this information we use co-occurrence matrix X, each entry of which corresponds to the number of times word j occurs in the context of word i. As the consequence:

$$P_{ij} = P(j|i) = \frac{X_{ij}}{X_i}$$

is the probability that word with index j occurs in the context of word i.

Ratios of co-occurrence probabilities are the appropriate starting point to begin word embedding learning. We firstly define a function *F* as:

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

which is dependent on 2 word vectors with indexes *i* and *j* and separate context vector with index *k*. F encodes the information, present in the ratio; the most intuitive way to represent this difference in vector form is to subtract one vector from another:

$$F(w_i - w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

Now in the equation, the left-hand side is the vector, while the right-hand side is the scalar. To avoid this we can calculate the product of 2 terms (product operation still allows us to capture the information we need):

$$F((w_i - w_j)^T \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

As long as in word-word co-occurrence matrix the distinction between context words and standard words is arbitrary, we can replace the probabilities ratio with:

$$F((w_i - w_j)^T \widetilde{w}_k) = \frac{F(w_i^T \widetilde{w}_k)}{F(w_j^T \widetilde{w}_k)}$$

and solve the equation:

$$F(w_i^T \widetilde{w}_k) = P_{ik} = \frac{X_{ik}}{X_i}$$

If we assume that *F* function is *exp()*, then the solution becomes:

$$w_i^T \widetilde{w}_k = \log P_{ik} = \log X_{ik} - \log X_i$$

This equation does not preserve symmetry, so we absorb 2 of the terms into biases:

$$w_i^T \widetilde{w}_k + b_i + \widetilde{b_k} = \log X_{ik}$$

Now our loss function we're trying to minimize is the linear regression function with some of the modifications:

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \tilde{w}_j + b_i + b_j - \log X_{ij})^2$$

where *f* is the weighting function, which is defined manually.

GloVe is also implemented with gensim library, its basic functionality to train on standard corpus is described with this snippet

```
import itertools
from gensim.models.word2vec import Text8Corpus
from glove import Corpus, Glove

# sentences and corpus from standard library
sentences = list(itertools.islice(Text8Corpus('text8'),None))
corpus = Corpus()

# fitting the corpus with sentences and creating Glove object
corpus.fit(sentences, window=10)
glove = Glove(no_components=100, learning_rate=0.05)

# fitting to the corpus and adding standard dictionary to the object
glove.fit(corpus.matrix, epochs=30, no_threads=4, verbose=True)
glove.add_dictionary(corpus.dictionary)
```

FastText (Enriching Word Vectors with Subword Information)

What if we want to take into account morphology of words? To do so, we can still use skip-gram model (remember I told you that skip-gram baseline is used in many other related works) together with negative sampling objective.

Let us consider that we are given a scoring function which maps pairs of the (word, context) to real-valued scores. The problem of predicting context words can be viewed as a sequence of binary classification tasks (predict presence or absence of context words).

For word at position *t* we consider all context words as positive examples and sample negative examples at random from the dictionary.

Now our negative sampling objective is:

$$\sum_{t=1}^{T}[\sum_{c \epsilon c_t} l(s(w_t, w_c)) + \sum_{n \epsilon N_{t,c}} l(-s(w_t, n))]$$

The FastText model takes into account internal structure of words by splitting them into a bag of character n-grams and adding to them a whole word as a final feature. If we denote n-gram vector as *z* and *v* as output vector representation of word w (context word):

$$s(w, c) = \sum_{g \epsilon G_w} z_g^T v_c$$

We can choose n-grams of any size, but in practice size from 3 to 6 is the most suitable one.

All of the documentation and examples on FastText implementation with the Facebook library are available here.

Poincaré embeddings (Poincaré Embeddings for Learning Hierarchical Representations)

Poincaré embeddings are the latest trend in the natural language processing community, based on the fact, that we're using hyperbolic geometry to capture hierarchical properties of the words we can't capture directly in Euclidean space. We need to use such kind of geometry together with Poincaré ball to capture the fact, that distance from the root of the tree to its leaves grows exponentially with every new child, and hyperbolic geometry is able to represent this property.

Notes on hyperbolic geometry

Hyperbolic geometry studies non-Euclidean spaces of constant negative curvature. Its main 2 theorems are that:

- *for every line l and every point P not on l there pass through P at least two distinct parallels through. Moreover there are infinitely many parallels to l through P;*
- *all triangles have angle sum less than 180 degrees.*

For 2-dimensional hyperbolic space we see that area *s* and length *l* both grow exponentially with formulas:

$$l = 2\pi sinh(r)$$

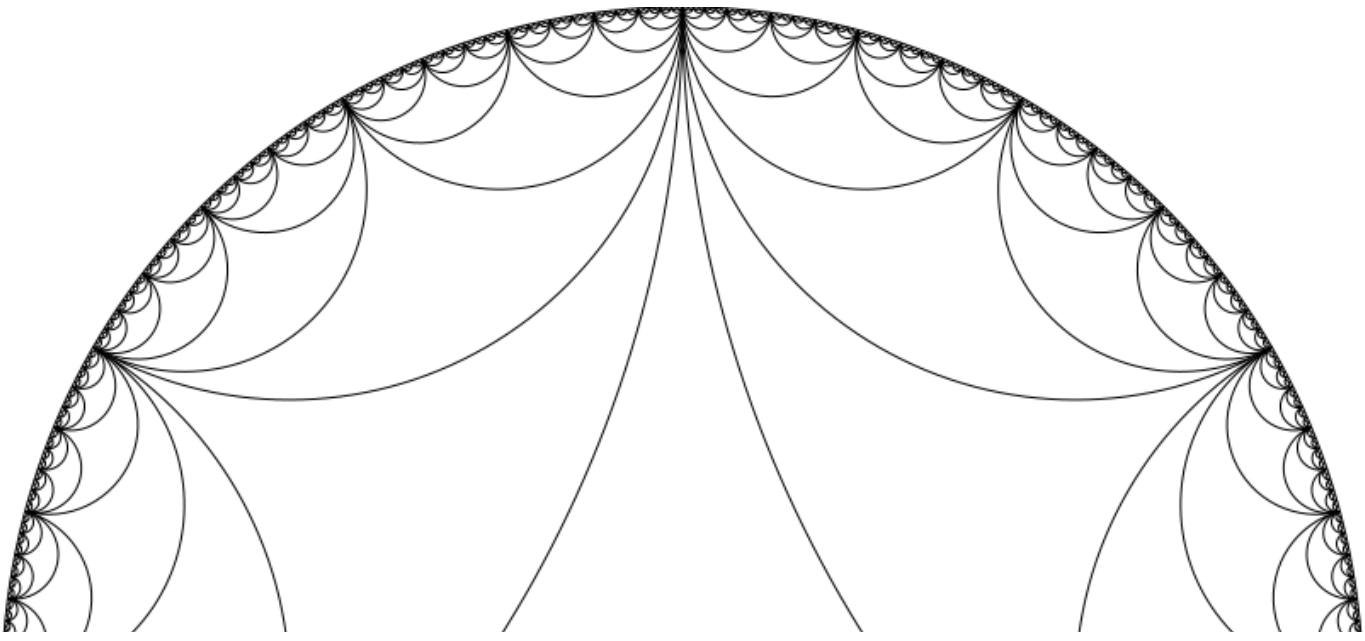
$$s = 2\pi(cosh(r) - 1)$$

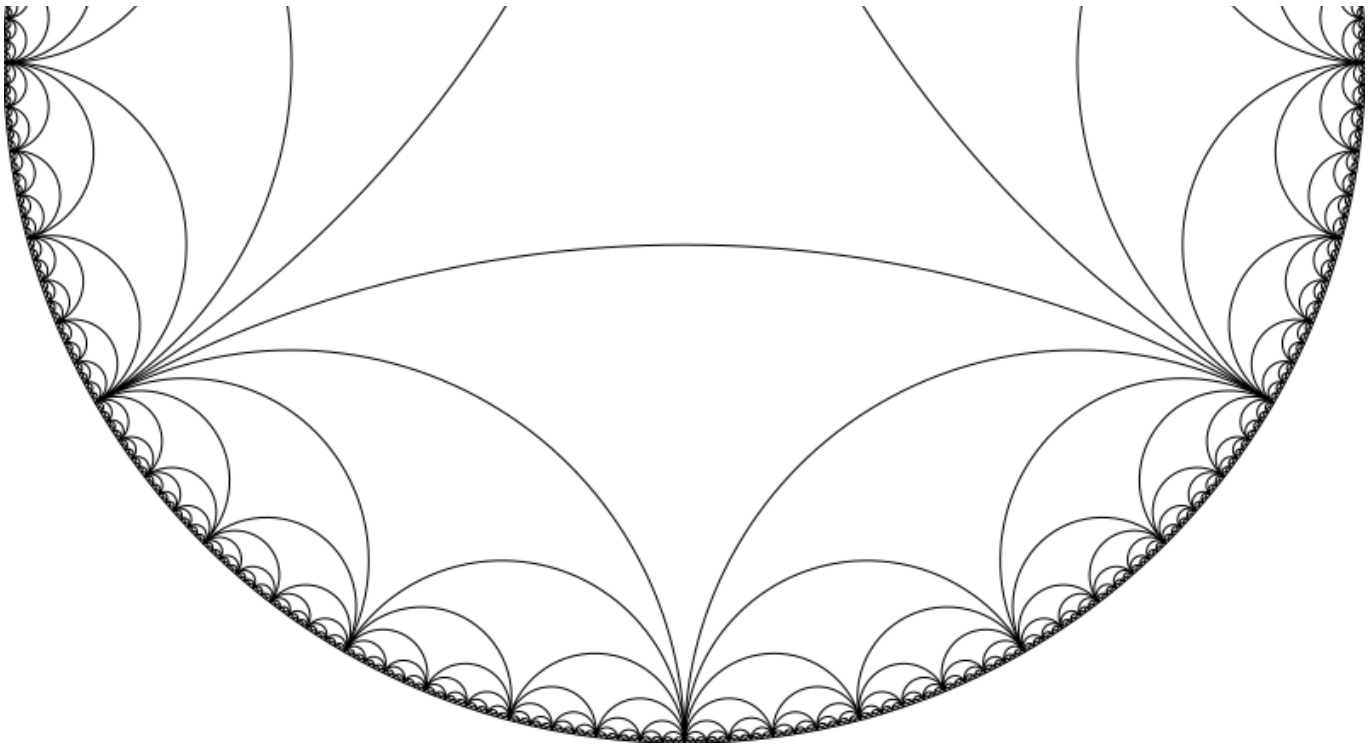
Hyperbolic geometry is designed the way that we can use the distance in the embedding space we create to reflect the semantic similarity of the words.

The model from the hyperbolic geometry we're interested in is the Poincaré ball model, stated as:

$$B^d = \{x \epsilon R^d \mid\mid\mid x \mid\mid < 1\}$$

where the norm we're using is the standard Euclidean norm. Poincaré ball model can be drawn as a disk where the straight lines consist of all segments of circles contained within the disk that are orthogonal to the boundary of the disk, plus all diameters of the disk.





Poincare embeddings baseline

Consider our task to model a tree such that we do it in a metric space and the structure of it is reflected in the embedding; as we know, the number of children in the tree grows exponentially with the distance from the root.

Regular tree with branching factor b can be modeled in hyperbolic geometry in 2 dimensions, such that nodes l levels below the root are located on the sphere with radius $l = r$ and nodes that are less than l levels below the root are located within this sphere. Hyperbolic spaces can be thought as continuous versions of trees, and trees — as discrete hyperbolic spaces.

The distance measure between 2 embeddings we're defining in hyperbolic space is:

$$d(u, v) = arcosh(1 + 2\frac{\| u - v \|^2}{(1 - \| u \|^2)(1 - \| v \|^2)})$$

which gives us the ability not only to capture the similarity between embedding effectively (through their distance) but also preserves their hierarchy (through their norm), which we take from WordNet taxonomy.

We're minimizing the loss function with respect to theta (element from the Poincaré ball, its norm should be no bigger than one):

$$L(\Theta) = \sum_{(u,v)\epsilon D} \log \frac{exp(-d(u,v))}{\sum_{v'\epsilon N(u)} exp(-d(u,v'))}$$

where *D* is the set of all observed hyponymy relations and *N(u)* is the set of negative examples for u.

Training details

- *initialize all embeddings randomly from the uniform distribution;*
- *learn embeddings for all symbols in D such that related objects are close in the embedding space.*

Conclusions

I consider this article to be a short intro to word embeddings, briefly describing the most common natural language processing techniques, their peculiarities and theoretical foundations. For more details on every method, a link to each of the original papers is attached; most of the formulas are pointed out, but more of the explanations on the notation can be found in the same place.

I haven't mentioned some of the basic word embedding matrix factorization methodologies like latent semantic indexing and hasn't paid much of the attention on real-life applications of each of the approaches as long as it all depends on the task and given corpus; for instance, the creators of GloVe claim that their approach worked well on named entity recognition task with CoNNL dataset, but it doesn't mean that it will work best in the case of unstructured data coming from different domain zones.

Also, paragraph embeddings are not lightened in this article, but this is another story… Is it worth telling it?

Machine Learning Word Embeddings NLP Calculus