Word2Vec Tutorial Part 2 - Negative Sampling

11 Jan 2017

In part 2 of the word2vec tutorial (here's part 1), I'll cover a few additional modifications to the basic skip-gram model which are important for actually making it feasible to train.

When you read the tutorial on the skip-gram model for Word2Vec, you may have noticed something–it's a huge neural network!

In the example I gave, we had word vectors with 300 components, and a vocabulary of 10,000 words. Recall that the neural network had two weight matrices–a hidden layer and output layer. Both of these layers would have a weight matrix with 300 x 10,000 = 3 million weights each!

Running gradient descent on a neural network that large is going to be slow. And to make matters worse, you need a huge amount of training data in order to tune that many weights and avoid over-fitting. millions of weights times billions of training samples means that training this model is going to be a beast.

The authors of Word2Vec addressed these issues in their second paper.

There are three innovations in this second paper:

1. Treating common word pairs or phrases as single "words" in their model.
2. Subsampling frequent words to decrease the number of training examples.
3. Modifying the optimization objective with a technique they called "Negative Sampling", which causes each training sample to update only a small percentage of the model's weights.

It's worth noting that subsampling frequent words and applying Negative Sampling not only reduced the compute burden of the training process, but also improved the quality of their resulting word vectors as well.

Word Pairs and "Phrases"

The authors pointed out that a word pair like "Boston Globe" (a newspaper) has a much different meaning than the individual words "Boston" and "Globe". So it makes sense to treat "Boston Globe", wherever it occurs in the text, as a single word with its own word vector representation.

You can see the results in their published model, which was trained on 100 billion words from a Google News dataset. The addition of phrases to the model swelled the vocabulary size to 3 million words!

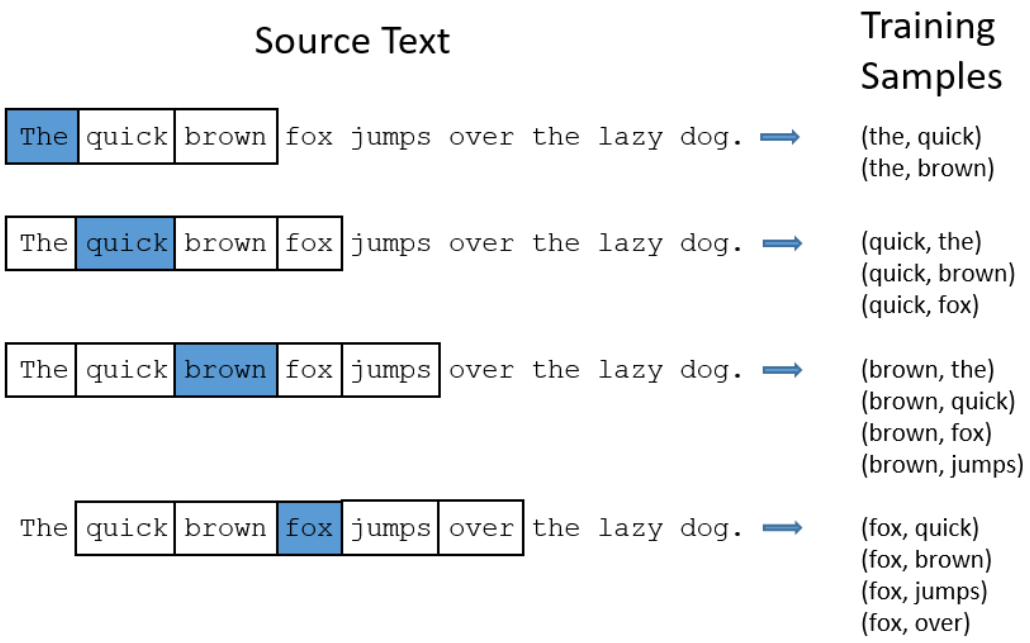
If you're interested in their resulting vocabulary, I poked around it a bit and published a post on it here. You can also just browse their vocabulary here.

I haven't had a chance to investigate their method of phrase detection, but it is covered in the "Learning Phrases" section of their paper. I'm also told the code is available in word2phrase.c of their published code here.

> One thought I had for an alternate phrase recognition strategy would be to use the titles of all Wikipedia articles as your vocabulary.

Subsampling Frequent Words

In part 1 of this tutorial, I showed how training samples were created from the source text, but I'll repeat it here. The below example shows some of the training samples (word pairs) we would take from the sentence "The quick brown fox jumps over the lazy dog." I've used a small window size of 2 just for the example. The word highlighted in blue is the input word.



There are two "problems" with common words like "the":

1. When looking at word pairs, ("fox", "the") doesn't tell us much about the meaning of "fox". "the" appears in the context of pretty much every word.
2. We will have many more samples of ("the", ...) than we need to learn a good vector for "the".

Word2Vec implements a "subsampling" scheme to address this. For each word we encounter in our training text, there is a chance that we will effectively delete it from the text. The probability that we cut the word is related to the word's frequency.

If we have a window size of 10, and we remove a specific instance of "the" from our text:

1. As we train on the remaining words, "the" will not appear in any of their context windows.
2. We'll have 10 fewer training samples where "the" is the input word.

Note how these two effects help address the two problems stated above.

Sampling rate

The word2vec C code implements an equation for calculating a probability with which to keep a given word in the vocabulary.

w_i is the word, $z(w_i)$ is the fraction of the total words in the corpus that are that word. For example, if the word "peanut" occurs 1,000 times in a 1 billion

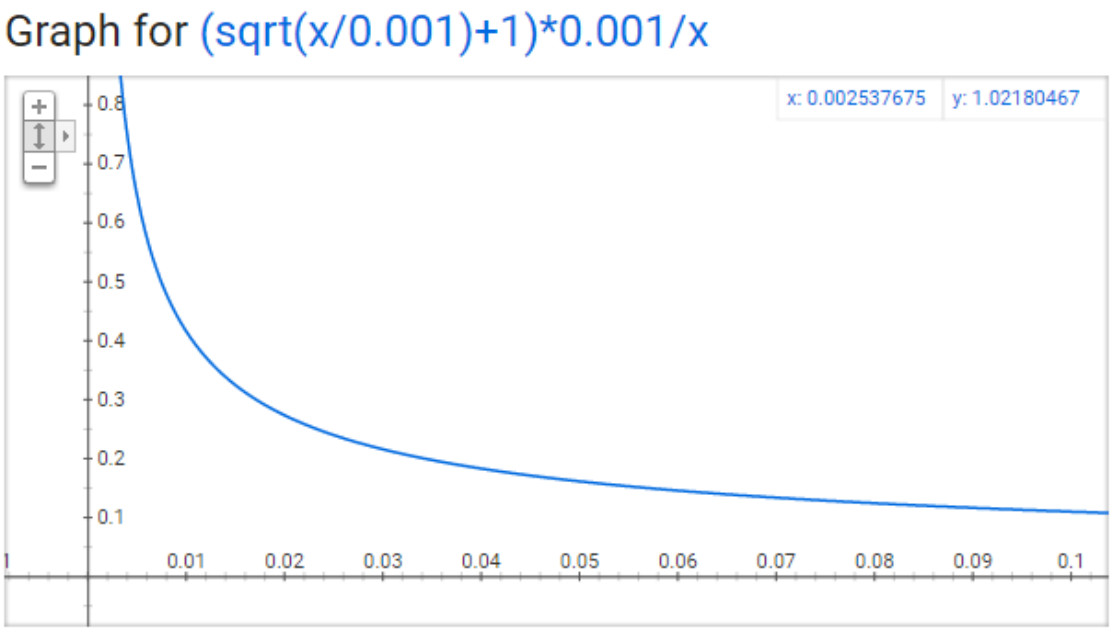
word corpus, then z('peanut') = 1E-6.

There is also a parameter in the code named 'sample' which controls how much subsampling occurs, and the default value is 0.001. Smaller values of 'sample' mean words are less likely to be kept.

$P(w_i)$ is the probability of *keeping* the word:

$$P(w_i) = (\sqrt{\frac{z(w_i)}{0.001}} + 1) \cdot \frac{0.001}{z(w_i)}$$

You can plot this quickly in Google to see the shape.



No single word should be a very large percentage of the corpus, so we want to look at pretty small values on the x-axis.

Here are some interesting points in this function (again this is using the default sample value of 0.001).

- $P(w_i) = 1.0$ (100% chance of being kept) when $z(w_i) <= 0.0026$.
  - This means that only words which represent more than 0.26% of the total words will be subsampled.
- $P(w_i) = 0.5$ (50% chance of being kept) when $z(w_i) = 0.00746$.
- $P(w_i) = 0.033$ (3.3% chance of being kept) when $z(w_i) = 1.0$.
  - That is, if the corpus consisted entirely of word w_i, which of course is ridiculous.

> You may notice that the paper defines this function a little differently than what's implemented in the C code, but I figure the C implementation is the more authoritative version.

Negative Sampling

Training a neural network means taking a training example and adjusting all of the neuron weights slightly so that it predicts that training sample more accurately. In other words, each training sample will tweak *all* of the weights in the neural network.

As we discussed above, the size of our word vocabulary means that our skip-gram neural network has a tremendous number of weights, all of which would be updated slightly by every one of our billions of training samples!

Negative sampling addresses this by having each training sample only modify a small percentage of the weights, rather than all of them. Here's how it works.

When training the network on the word pair ("fox", "quick"), recall that the "label" or "correct output" of the network is a one-hot vector. That is, for the output neuron corresponding to "quick" to output a 1, and for *all* of the other thousands of output neurons to output a 0.

With negative sampling, we are instead going to randomly select just a small number of "negative" words (let's say 5) to update the weights for. (In this context, a "negative" word is one for which we want the network to output a 0 for). We will also still update the weights for our "positive" word (which is the word "quick" in our current example).

> The paper says that selecting 5-20 words works well for smaller datasets, and you can get away with only 2-5 words for large datasets.

Recall that the output layer of our model has a weight matrix that's 300 x 10,000. So we will just be updating the weights for our positive word ("quick"), plus the weights for 5 other words that we want to output 0. That's a total of 6 output neurons, and 1,800 weight values total. That's only 0.06% of the 3M weights in the output layer!

In the hidden layer, only the weights for the input word are updated (this is true whether you're using Negative Sampling or not).

Selecting Negative Samples

The “negative samples” (that is, the 5 output words that we’ll train to output 0) are chosen using a “unigram distribution”.

Essentially, the probability for selecting a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples.

In the word2vec C implementation, you can see the equation for this probability. Each word is given a weight equal to it’s frequency (word count) raised to the 3/4 power. The probability for a selecting a word is just it’s weight divided by the sum of weights for all words.

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n}\left(f(w_j)^{3/4}\right)}$$

The decision to raise the frequency to the 3/4 power appears to be empirical; in their paper they say it outperformed other functions. You can look at the shape of the function–just type this into Google: “plot y = x^(3/4) and y = x” and then zoom in on the range x = [0, 1]. It has a slight curve that increases the value a little.

The way this selection is implemented in the C code is interesting. They have a large array with 100M elements (which they refer to as the unigram table). They fill this table with the index of each word in the vocabulary multiple times, and the number of times a word’s index appears in the table is given by $P(w_i)$ * table_size. Then, to actually select a negative sample, you just generate a random integer between 0 and 100M, and use the word at that index in the table. Since the higher probability words occur more times in the table, you’re more likely to pick those.

Other Resources

For the most detailed and accurate explanation of word2vec, you should check out the C code. I’ve published an extensively commented (but otherwise unaltered) version of the code here.

I’ve also created a post with links to and descriptions of other word2vec tutorials, papers, and implementations.

Join the discussion…

Yueting Liu • 2 months ago

I have got a virtual map in my head about word2vec within a couple hours thanks to your posts. The concept doesn't seem daunting anymore. Your posts are so enlightening and easily understandable. Thank you so much for the wonderful work!!!

5 • Reply • Share ›

Chris McCormick Mod → Yueting Liu • 2 months ago

Awesome! Great to hear that it was so helpful--I enjoy writing these tutorials, and it's very rewarding to hear when they make a difference for people!

2 • Reply • Share ›

Jane • 4 months ago

so aweeesome! Thanks Chris! Everything became soo clear! So much fun learn it all!

2 • Reply • Share ›

Chris McCormick Mod → Jane • 4 months ago

Haha, thanks, Jane! Great to hear that it was helpful.

• Reply • Share ›

Ben Bowles • 2 months ago

Thanks for the great tutorial.

About this comment "Recall that the output layer of our model has a weight matrix that's 300 x 10,000. So we will just be updating the weights for our positive word ("quick"), plus the weights for 5 other words that we want to output 0. That's a total of 6 output neurons, and 1,800 weight values total. That's only 0.06% of the 3M weights in the output layer!"

Should this actually be 3600 weights total for each training example, given that we have an embedding matrix and an matrix of weights, and BOTH involve updating 1800 weights (300 X 6 neurons)? (Both of which should be whatever dimension you are using for your embeddings multiplied by vocab size)?

1 • Reply • Share ›

Chris McCormick Mod → Ben Bowles • 2 months ago

Hi Ben, thanks for the comment.

In my comment I'm talking specifically about the output layer. If you

include the hidden layer, then yes, there are more weights updated. The number of weights updated in the hidden layer is only 300, though, not 1800, because there is only a single input word.

So the total for the whole network is 2,100. 300 weights in the hidden layer for the input word, plus 6 x 300 weights in the output layer for the positive word and five negative samples.

And yes, you would replace "300" with whatever dimension you are using for your word embeddings. The vocabulary size does *not* factor into this, though--you're just working with one input word and 6 output words, so the size of your vocabulary doesn't impact this.

Hope that helps! Thanks!

▲ ▼ • Reply • Share ›

Ben Bowles ➚ Chris McCormick • 2 months ago

This is super helpful, I appreciate this. My intuition (however naive it may be) was that the embeddings in the hidden layer for the negative sample words should also be updated as they are relevant to the loss function. Why is this not the case? I suppose I may have to drill down into the equation for backprop to find out. I suppose it has to do with the fact that when the one-hot vector is propagated forward in the network, it amounts to selecting only the embedding that corresponds to the target word.

▲ ▼ • Reply • Share ›

Chris McCormick Mod ➚ Ben Bowles • 2 months ago

That's exactly right--the derivative of the model with respect to the weights of any other word besides our input word is going to be zero.

Hit me up on LinkedIn!

▲ ▼ • Reply • Share ›

Rumesa Farooq • 6 days ago

Hello Chris
I have gone through your commented C code of word2vec at github.It was really nice.But I have a problem.When I run the code with CBOW=1, it creates vectors and then display cousine distance or similarity by entering a word.But when I run it in skipgram mode i.e CBOW=0, it also creates vectors but donot display the cousine distance for any word.Any suggestion? How to use distance file for skipgram?

▲ ▼ • Reply • Share ›

Jay • 7 days ago

Very nice. Would be helpful to add a short section on updating the weights and showing them converge.

Reply • Share ›

Chris McCormick Mod → Jay • 6 days ago

Thanks, Jay!

Reply • Share ›

Chen Lu • 23 days ago

Thanks Chris, it is really a very useful tutorial. One thing (which may be too naive, I am not able to get yet, is the one-hot vector part, which part of the code provided in the (github/word2vec_commented) is doing that part?

Reply • Share ›

Chris McCormick Mod → Chen Lu • 16 days ago

Hi Chen,

The one-hot vector is part of the mathematical formulation of the skip-gram architecture, but in code, you don't need it. All the one-hot vector does is select the corresponding word vector from the middle layer, so you won't find any one-hot vectors in the code.

Reply • Share ›

Mahesh Govind • 2 months ago

Hi Chris,

Thank you for the good tutorial .

A query about visualising word vector. When they visualize the vectors , are we doing a dimensionality reduction and plotting them. And when we plot them , ie the word vectors , those vectors which are similar (dot product value) are displayed around same area.

Reply • Share ›

Chris McCormick Mod → Mahesh Govind • a month ago

Hi Mahesh, thanks. I'm not sure how they do the visualizations, but I would guess the same as you, that they're doing something like PCA to reduce to 2 dimensions.

Reply • Share ›

Mahesh Govind → Chris McCormick • a month ago

Thank you for the reply . One more foolish query . Word2vec will provide one word co-related to the input word as the result, irrespective of the window size ? The window size will have impact only on creating a number of training tuples ? RNN is used to provide the context for a sentence . Is this intuition correct

Reply • Share ›

Chris McCormick Mod → Mahesh Govind • a month ago

Yes, training is done with one output word at a time. Increasing the window size can affect the resulting word vectors, since it will add farther away words to the context of the input word.

Reply • Share ›

Huichang Han • 2 months ago

Great article, this is the best tutorial for me to learn the insight of word2vec.

Reply • Share ›

Chris McCormick Mod → Huichang Han • 2 months ago

Thank you :)

Reply • Share ›

Walker • 2 months ago

Is tf.nn.sampled_softmax_loss() an implementation of negative sampling?

I always think they are different. Sampled softmax loss is a fast way to compute softmax (because the denominator of the softmax is huge) and negative sampling is ... emm ... conceptually "randomly select just a small number of “negative” words (let’s say 5) to update the weights for", i don't really know how to implement it. It's like a big black box to me. And just now some friend mentioned that we can use sampled softmax loss to implement negative sampling. I was shocked and did a little research about it.

It seems like they are from different paper and https://www.tensorflow.org/... describes that NCE is a generalized version of subsampled softmax and NEG is simplifying case of the NCE... this is just quite confusing. So I guess this is not the same but they are closely related and comparable?

Reply • Share ›

Chris McCormick Mod → Walker • 2 months ago

Haha, yeah, it is quite a confusing mess! If you want a detailed implementation of negative sampling, check out my commented word2vec code here. I'm not familiar with those other methods, but I think your insight is correct--they are all similar approaches.

Reply • Share ›

yoch melka • 2 months ago

Thank you very much for this great explanation !

> Their paper doesn’t go in to detail about how they performed phrase

detection, other than to say that it was a “data driven” approach.

The second paper ('Distributed Representations of Words and Phrases and their Compositionality', section 4) give the details about phrasing, and the corresponding source code is available in word2phrase.c

Reply • Share ›

Chris McCormick Mod → yoch melka • 2 months ago

Thank you! I will update the post to point to those, and maybe write them up at some point.

Reply • Share ›

yoch melka → Chris McCormick • 2 months ago

Note that, according to the source code, equation (6) in the paper seems incomplete. The complete form is

and it's very similar to PMI (without log).

Reply • Share ›

Vicky Zheng • 3 months ago

Thanks for the explanation! Quick question - could you explain the idea behind the negative sampling probability equation? Why raise the frequency and sum of frequencies to the 3/4 power?

Reply • Share ›

Chris McCormick Mod → Vicky Zheng • 2 months ago

Hi, Vicky - I believe that was arrived upon empirically; from their paper: "We investigated a number of choices for P_n(w) and found that the unigram distribution U(w) raised to the 3/4rd power (i.e., U(w)^(3/4)/Z) outperformed significantly the unigram and the uniform distributions"

You can look at the shape of the function, type this into google: "plot y = x^(3/4) and y = x" and then zoom in on the range x = [0, 1]. It has a slight curve that increases the value a little.

Reply • Share ›

Jacob Tao • 3 months ago

Hi Chris, thanks for the post which really clarifies a lot for the paper. However, I'm still confused of the purpose of negative sampling. Is it just aiming for computational efficiency? In the original paper, the author points

out hierarchal softmax as an "efficient approximation" of the softmax, and that's how I deduct that maybe negative sampling is only used for improving computation? I've also read paper "natural-language-understanding-almost-from-scratch", where the negative sampling is necessary for the training, which has nothing to do with computation limits. Therefore, I was wondering whether the accuracy would improve ultimately if we remove the negative sampling and update weights for every other words during training.

Reply • Share ›

Chris McCormick Mod → Jacob Tao • 2 months ago

Hi Jacob,

I do think that the primary purpose of negative sampling is a dramatic improvement in computational efficiency. That's a very relevant quality, as it allows the model to be trained on enormous datasets (they trained on 33 billion words!), which does in turn improve accuracy.

Updating every weight may be computationally intractable--in my simple example it would be 3 orders of magnitude slower!

They claim in their paper that negative sampling improves the accuracy over hierarchical softmax (which is also an attempt at efficiency): "In addition, we present a simplified variant of Noise Contrastive Estimation (NCE) [4] for training the Skip-gram model that results in faster training and better vector representations for frequent words, compared to more complex hierarchical softmax that was used in the prior work [8]".

I don't see any comments in the paper, though, addressing whether negative sampling trades accuracy for efficiency. I honestly don't know the answer to that.

Reply • Share ›

itzjustricky • 3 months ago

Absolutely AMAZINGG! I am very sad this is not the first word2vec explanation I arrived upon. Super clear.

Reply • Share ›

Chris McCormick Mod → itzjustricky • 3 months ago

Good to hear, thank you!

Reply • Share ›

edwin guo • 3 months ago

Hi Chris, thanks for the excellent tutorial! A quick question, is subsampling happens when you clean the dataset? And every single word has its chance to be dropped? Or use nltk corpus to eliminate the stopwords

chance to be dropped? Or use nltk.corpus to eliminate the stopwords ahead of time? what is the best practice?

Reply • Share ›

Chris McCormick Mod → edwin guo • 3 months ago

The Word2Vec C implementation performs the subsampling "on the fly". That is, it removes the words as part of the training. And, yes, every word has a chance of being dropped, but with a probability related to its frequency in the dataset (less frequent words are more likely to be kept).

As for whether or not to remove stop words, I don't know what the best practices are on this, sorry...

Reply • Share ›

jayanth reddy Regatti • 3 months ago

Hi Chris, thanks for the post. I have a question. You mentioned in negative sampling that the label or correct output of the network is a one hot vector. However in your part-1 you mentioned that it is a vector of probabilities adding up to 1. Am I missing something?

Reply • Share ›

Chris McCormick Mod → jayanth reddy Regatti • 3 months ago

The training output is a one hot vector, but that's not the ultimate desired output of the network. The output should be a probability distribution over all the words in the vocabulary. The network learns these statistics from the training data.

Reply • Share ›

Cuong Nguyen Ngoc • 3 months ago

Hi Chris. Thank you so much. It's so clear to understand, this helped me a lot. Can I ask you a question about Google news pretrained word2vec data? How did you can get these words to put here https://github.com/chrisjmc... ?

Reply • Share ›

Chris McCormick Mod → Cuong Nguyen Ngoc • 3 months ago

Thanks! I included my code for generating those files in the project.

Reply • Share ›

Max Fomitchev • 4 months ago

Hi Chris, thanks for the awesome article, everything is easy to understand.

I was studying the original word2vec code and found 2 small differences in hierarchical softmax and negative sampling parts of gradient calculation.

Probably you can clarify is there any error or not.

The first looks like a small bug -
gradient calculating in HS:
if (f <= -MAX_EXP) continue;
else if (f >= MAX_EXP) continue;
else f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];

(f == -MAX_EXP) makes
(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2)) == 0 (correct index)
and
(f == MAX_EXP) makes

see more

• Reply • Share ›

Johan Falkenjack • 4 months ago

Hi Chris, awesome tutorial. This actually made most of word2vec actually "click" for me. Thank you!

However some of the nitty gritty is still a bit unclear to me.

In negative sampling, you compute the activations for the correct context word and for n random "context word candidates", right? But how do you actually do this? Do you use masking with a softmax layer the size of your vocabulary? Or do you do something similar to the Collobert & Weston approach and feed in both the current word (from which the prediction is computed), the correct context word plus n extra random context word candidates and then use a softmax layer of the width 1+n to compute the most probable among the context word candidates?

• Reply • Share ›

Chris McCormick Mod → Johan Falkenjack • 4 months ago

Thanks, Johan! You've got the right idea.

It may be easier to think of from a software perspective than a mathematical one, I'm not sure.

When you calculate a gradient update, it's always with respect to a single parameter, and you just do it for every parameter in the network. Here, what you're doing is just omitting 99.94% of the output layer parameters from the gradient update.

So, in software, you only calculate the gradients of the output weights for those handful of words, and you don't touch the rest.

Hope that helps!

• Reply • Share ›

Shayne Miel • 5 months ago

The insights into the C code are so valuable. Thank you!

Reply • Share ›

Chris McCormick Mod → Shayne Miel • 5 months ago

Thanks, Shayne!

I'm actually hoping to publish an unaltered but commented and document version of Google's original C implementation. Maybe in the next month or so!

1 Reply • Share ›

Saurabh Thakur → Chris McCormick • 4 months ago

Chris! Loved your explanation. What is the parameter name for negative sampling in gensim word2vec model?

Reply • Share ›

Chris McCormick Mod → Saurabh Thakur • 4 months ago

It's the parameter 'negative', which defaults to 5. See the source here.

Reply • Share ›

Bob • 5 months ago

Hi, Thanks for the post !
Feel free correct me.
Question :
Regarding the example of "the" :

A) "....effectively delete it from the text. The probability that we CUT the word is related to the word's frequency."

On the other hand,

B) "... the probability for SELECTing a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples."

A and B seem to contradict with each other.

If my limited understanding is not wrong, in negative sampling, "the" WILL be selected. It should not be cut off.

Related posts

Concept Search on Wikipedia 22 Feb 2017
Getting Started with mlpack 01 Feb 2017
DBSCAN Clustering 08 Nov 2016