Sentence Embeddings. Fast, please!

Sentence embeddings are one key ingredient in modern NLP applications. Compute sentence embeddings 38x faster using Gensim, Cython, and BLAS.

Oliver Borchers Follow
Jun 10, 2019 · 10 min read ★

Photo by Bernard Hermant on Unsplash

The fse code in this article is deprecated. Please make sure to use the

updated code as outlined on Github.

Introduction

When working with textual data in a machine learning pipeline, you may come across the need to compute sentence embeddings. Similar to regular word embeddings (like Word2Vec, GloVE, Elmo, Bert, or Fasttext), sentence embeddings embed a **full sentence into a vector space**. In practice, a sentence embedding might look like this:

"I shot the sheriff" → [0.2 ; 0.1 ; -0.3 ; 0.9 ; …]

These sentence embeddings retain some nice properties, as they inherit features from their underlying word embeddings [1]. Thus, we might use sentence embeddings for varying purposes:

- Compute a similarity matrix of sentences based on their embeddings.
- Plot sentences using a common mapping technique, like t-SNE.
- Predict some value for the sentence, i.e. sentiment.

A very simple supervised application of sentence embeddings can be seen in "deep averaging networks" [2], where the authors use sentence embeddings in sentiment analysis and question answering. In fact, sentence embeddings are a *deceptively simple baseline* to start from when you are working with textual data. Fortunately, they do not require any form of gradient based optimization if (pre-trained) word embeddings are already available.

Photo by Álvaro Serrano on Unsplash

This post is intended for a *technical data-science audience*. We are going to explore smooth inverse frequency (SIF) sentence embeddings [1]. Specifically, we are optimizing the computation of SIF embeddings by hand-crafting a function, which is specifically tailored to **compute SIF embeddings as fast as possible**. For this purpose we first use Python, then we migrate to Cython and finally to Basic Linear Algebra Subprograms (BLAS). Cython and BLAS are core ingredients used in Gensims Word2Vec implementation. I have to thank Radim Řehůřek, as Gensims implementation is the foundation of this post. For a similar post, feel free to read the original Word2Vec optimization blog. The following optimizations allow us to speed up the computation of SIF embeddings by a factor of **38 (!)**.

SIF Embeddings

Smooth inverse frequency embeddings were originally conceived by [1] and the corresponding paper has been presented at the 2017 ICLR. The

code for the original paper is available at Github. The authors present a nice probabilistic motivation for the inverse frequency weighted continuous bag-of-words model. We are not going into the technical details of the math, but rather into the optimization of the algorithm for computing the SIF embeddings. If you have to compute the SIF embeddings for millions of sentences, you need a routine to accomplish the task in a lifetime. In this post, we are just looking for **speed**. And Gensim offers quite a lot of it.

Algorithm to compute the SIF embeddings. Source: [1].

Discerning the algorithm, we can infer that most work is to be done in line 1 & 2. While it might seem straightforward, there is plenty of room to optimize line 1 & 2 when you are working with Python. The task is as follows: Compute the SIF embeddings as fast as possible for all sentences in the brown corpus. We will rely on Gensims Word2Vec implementation to obtain word vectors and only use a little preprocessing.

Implementation

The whole code for this project is available on Github. You can install the ***fast sentence embedding library*** which I wrote using pip:

```
pip install fse
```

You will need regular Python packages, specifically Numpy, Scipy, Cython, and Gensim. **TL;DR: If you need sentence embeddings fast, just use:**

```
from gensim.models import Word2Vec
sentences = [["cat", "say", "meow"], ["dog", "say", "woof"]]
model = Word2Vec(sentences, min_count=1)
```

```
from fse.models import Sentence2Vec
se = Sentence2Vec(model)
sentences_emb = se.train(sentences)
```

Optimization

In order to optimize the SIF function, we first need a prototype that is working correctly. Let's have a look at the prototpye:

The code should be pretty self-explanatory. We first define the variable *vlookup*, which points us to gensims vocab class. Hereby we can access a *words index* and *count*. *Vectors* points us to the gensim *wv* class, which contains the word vectors themselves. Then we iterate over all sentences. For each sentence, we sum the vectors of all the words in the sentence up, while multiplying them with their SIF weight. Finally, we divide the vector by the length of the sentence. Nothing too fancy.

We need an embedding to measure the execution time of this function. Let us use a 100 dimensional Word2Vec embedding, that was trained on the commonly available Brown corpus. Then we use the first 400 sentences of the corpus and time the function. **5.70 seconds**.

That is way to slow. The keen eye will have already noticed the pitfall: I've been using a for-loop in the vector addition. That is precisely the reason to

not use python for-loops in such a scenario. They are way too slow. To circumvent this, let us rewrite line 37–40 to use the numpy routines.

The results is astonishing: **0.041434 seconds.** That is **137x faster** than the baseline version. However, we can still do better. Much better. As you might have noticed, vectors[w] acts as a lookup table for the word-vectors. If we *pre-compute* the word indices of the sentence, we can directly use numpy indexing to access all vectors at once, which is much faster. In the same step, we have to pre-compute the SIF weights for the vectors to access them via indexing.

With a runtime of **0.011987 seconds**, these small additions cause the function to be **476x faster** than the baseline we established.

To further speed up the implementation, we might actually pre-compute all the weighted vectors, as the weights and word vectors themselves do not change. Thereby, we decrease the runtime to **0.008674 seconds**, marking a **658x increase** over the baseline. *But wait. There is a catch:*

By pre-computing the SIF vectors, we might have to compute a fairly large matrix, which is not time-efficient if we only have a few sentences and a large vocabulary. Nor is it space-efficient, as we have to store a separate weighted embedding matrix in RAM. Thus, we are running into a *tradeoff between memory and time efficency*. Pre-computing the weighted vectors does only make sense if the effective number of words in our sentences to train on is larger than the vocabulary.

I will spoil a little bit by stating that this is not necessary due to the BLAS function we are going to use later, but keep it in mind for now.

We have already obtained quite some considerable speedups. The next thing we want to optimize is the summation of the vectors, where we can still obtain a significant speedup. However, to better benchmark the results, we are taking the conversion of the sentences to indices out of the equation. We thus pre-compute all sentence-indices for the following benchmarks:

```
sentences_idx = [np.asarray([int(model.wv.vocab[w].index) for w in s
if w in model.wv.vocab], dtype=np.intc) for s in sentences]
```

You will immediately realize that this is a *pretty bad idea* in practice, as we now essentially have to store a second dataset. However, it serves our benchmarking purposes quite well for the moment.

With **0.004472 seconds** the new function — using pre-computed weighted vectors and sentence indices — clocks in at **1276x** the speed of the baseline implementation. **However, the comparison is a bit off, as the baseline did not use pre-computed indices**. Thus, this is our new benchmark for the moment.

The next step is to migrate to Cython. Note that we are still working with pre-computed indices and weighted vectors. We wrap everything into a nice Cython file and then compile it.

The cdef float[:,:] command gives us access to the memory view of the pre-computed vectors. For a more detailed explanation, the Cython website provides a neat tutorial on memory views. Remember to use

```
# cython: boundscheck=False
# cython: wraparound=False
```

at the beginning of the file. These two commands remove the safety nets, which might possibly slow us down. The former checks whether we are running out of bounds of the array, whilst the latter allows us to perform negative indexing (which we are not going to use). We are now running **2.42x faster** than the previous *pure numpy implementation*, which also relied on pre-computed indices for the sentences. Thus the comparison is fair again.

Note that we have pre-defined all of the necessary structures *as C variables* that we can use in a pure C-loop without any Pythonic interference. Thereby, we can also release the global interpreter lock (GIL), which might come in handy at some future point in time, when we are thinking about a multi-threaded implementation of **fse**. We essentially rewrite the part of the code used to sum all words in a sentence:

Thereby, we decrease the runtime to **0.000805 seconds**, which marks another **2.28 fold** increase over the first Cython implementation.

Finally, we are going for BLAS. The Basic Linear Algebra Subprograms describe low-level linear algebra routines. They are categorized in three levels: Level 1: vector ops, Level 2: vector-matrix ops, Level 3: matrix-matrix ops. We only need access to Level 1 for our purposes. Precisely, we need:

- SAXPY: (Single) constant times a vector plus a vector
- SSCAL: (Single) scales a vector by a constant

Gensims core classes make heavy use of these highly optimized routines to speed up Word2Vec, Doc2Vec, and Fasttext.

Now recall what we wrote about pre-computing the SIF weighted vectors earlier. The SAXPY function takes the argument *a*, which is used in y = a*x + y. Thus, we have to do the weight*vector multiplication anyways. Therefore, pre-computing the SIF vectors is actually an inefficient idea after all.

Additionally, we are dropping Cython memory view for direct unsafe access to the numpy variables via pointers & memory addresses.

We have to define the sscal and saxpy pointers elsewhere, which is spared in the code snippet (usually in a *.pxd file).

What did we get from all of the additional hassle? A runtime of **0.000267 seconds**, which again marks a **3.01x increase(!)** over the previous implementation. If we only look at the functions that used pre-computed vectors and indices, we decreased the runtime from **0.004472** to **0.000267 seconds**, a **16.76x increase** in speed. These values can change significantly as we increase the size of the vectors and the data. In total, we sped up the code over the baseline by a factor of **21,400**, although it is a bit of a comparison of apples and oranges, due to the differences in pre-computation.

Validation & Application

To test if everything is working correctly, we are now looking at two tasks: the STS sentence similarity benchmark [3] and predicting emotions on a sentence level [4]. You can find the scripts in the corresponding Jupyter notebook. Let's use the pre-trained GoogleNews-vectors-negative300 vectors for the estimation of the sentence embeddings.

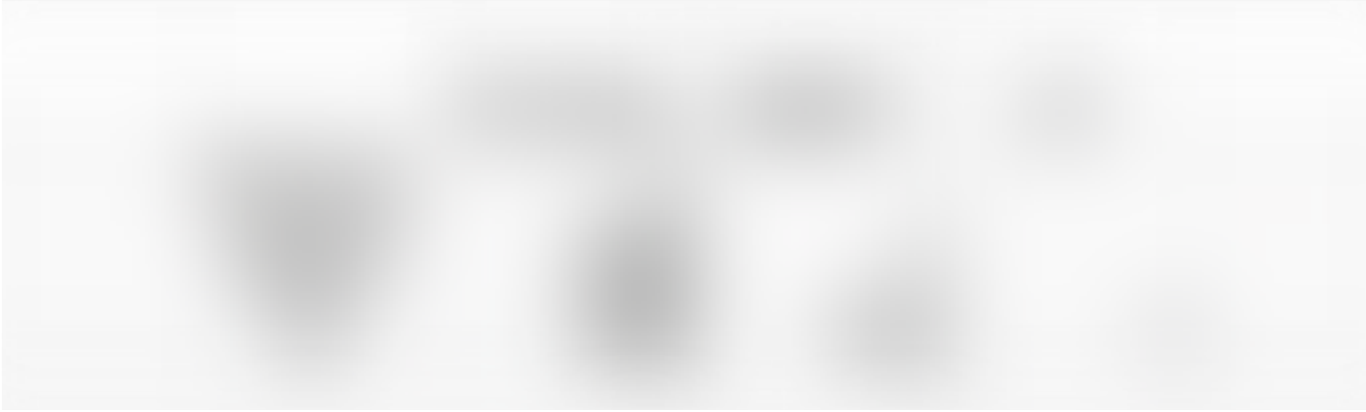
The reddit dataset for predicting emotions contains four classes: creepy, gore, happy, and rage. After some preprocessing we end up with 2,460 sentences and an equal balance of all four emotion classes.

As for timing: Computing the sentence embeddings using our 2nd variant developed earlier takes about 991 ms ± 19.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each). The final implementation that is contained in fse.models.Sentence2Vec completes the task in 25.9 ms ± 615 µs per loop (mean ± std. dev. of 7 runs, 10 loops each), a **38x speed increase** and a fair comparison.

Source: Author.

In terms of predicting emotions we are also able to accomplish the task pretty well. Note: Multinomial Logistic Regression, Train/Test split 50%.

Finally we approach the sentence similarity benchmark. The data is available here. We only look at the DEV set. The original paper reported a spearman correlation of 71.7, but they used Glove vectors. According to the other benchmarks, the implementation works as expected.



Source: Author.

Conclusion

Sentence embeddings are an essential part of NLP pipelines. This blog post shows how to implement average and SIF-weighted CBOW embeddings,

which can serve as a nice baseline for subsequent tasks. Due to their simplicity and modularity, we anticipate a broad variety of applications. The corresponding **fse** package is available on pip / Github and offers a fast way of computing sentence embeddings for data scientists.

For inquiries and questions, feel free to contact me.

Additional information

The code for this post is available on Github and via pip:

```
pip install fse
```

Literature

1. Arora S, Liang Y, Ma T (2017) A Simple but Tough-to-Beat Baseline for Sentence Embeddings. Int. Conf. Learn. Represent. (Toulon, France), 1–16.

2. Iyyer M, Manjunatha V, Boyd-Graber J, Daumé III H (2015) Deep Unordered Composition Rivals Syntactic Methods for Text Classification.

Proc. 53rd Annu. Meet. Assoc. Comput. Linguist. 7th Int. Jt. Conf. Nat. Lang. Process., 1681–1691.

3. Eneko Agirre, Daniel Cer, Mona Diab, Iñigo Lopez-Gazpio, Lucia Specia. Semeval-2017 Task 1: Semantic Textual Similarity Multilingual and Crosslingual Focused Evaluation. Proceedings of SemEval 2017.

4. Duong, Chi Thang, Remi Lebret, and Karl Aberer. “Multimodal Classification for Analysing Social Media.” The 27th European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD), 2017

Disclaimer

Opinions expressed are solely my own and do not express the views or opinions of my employer. The author assumes no responsibility or liability for any errors or omissions in the content of this site. The information contained in this site is provided on an “as is” basis with no guarantees of completeness, accuracy, usefulness or timeliness.

Machine Learning NLP Word Embeddings Artificial Intelligence Towards Data Science

263 claps

WRITTEN BY

Oliver Borchers

Follow

PHD Candidate | Machine Learning | Marketing | Data Science | University of Mannheim

Towards Data Science

Follow

A Medium publication sharing concepts, ideas, and codes.

See responses (3)

More From Medium

Coding Mistakes I Made As A Junior Developer

Sorry, Online Courses Won't Make you a Data Scientist

Chris in Towards Data Science

Ramshankar Yadhunath in Towards Data Science

Lesser known Python Features

James Briggs in Towards Data Science

Why Data Science might just not be worth it

Dario Radečić in Towards Data Science

5 Books That Will Teach You the Math Behind Machine Learning

Tivadar Danka in Towards Data Science

Data Classes in Python

Halil Yıldırım in Towards Data Science

Machine Learning Engineer vs Data Scientist (Is Data Science Over?)

Jason Jung in Towards Data Science

5 Spectacular Books For Learning New Programming Languages

Emmett Boudreau in Towards Data Science