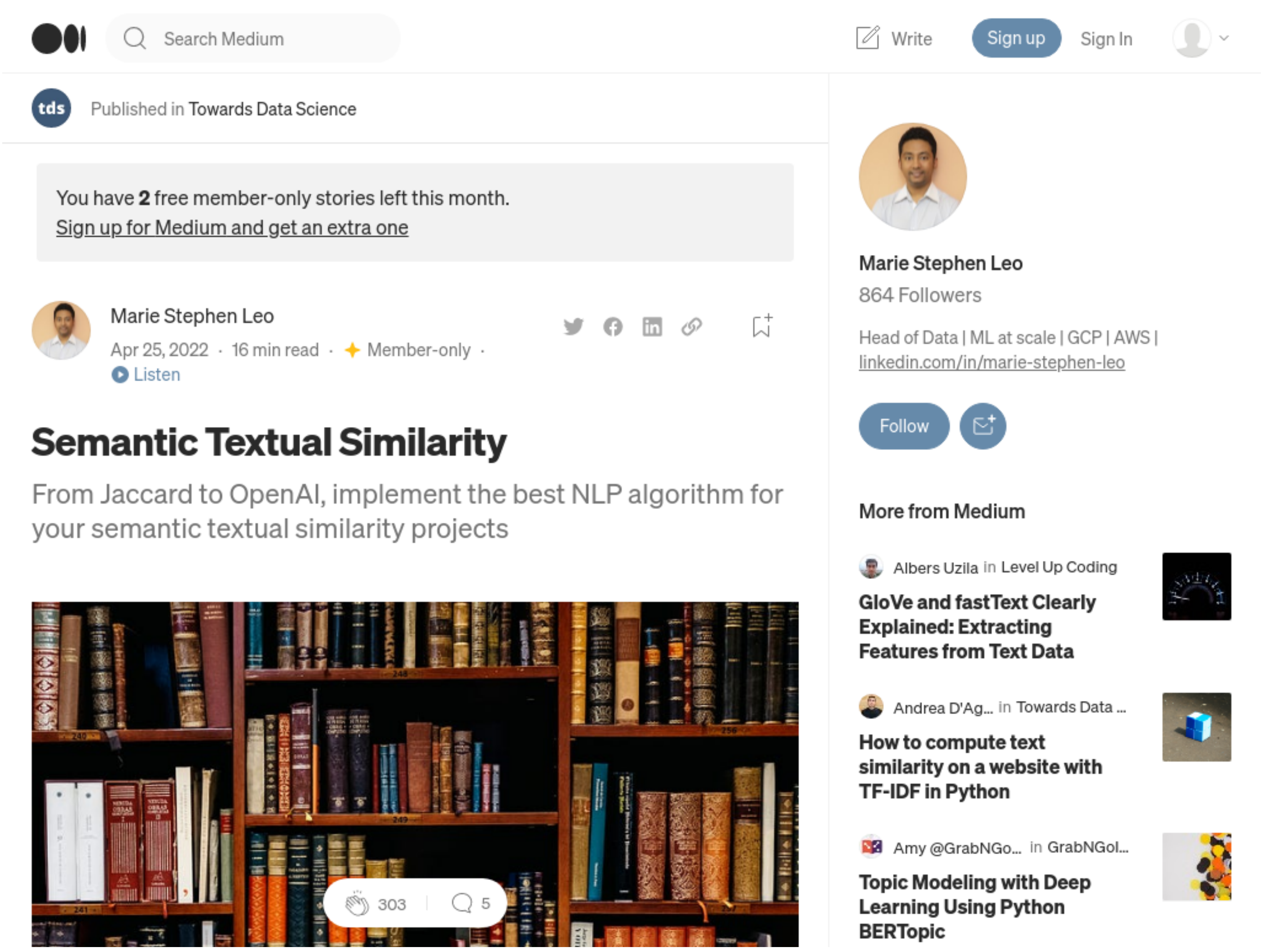
Unknown Title

: 2/3/2023



Published in Towards Data Science

You have **2** free member-only stories left this month.
Sign up for Medium and get an extra one

Marie Stephen Leo
Apr 25, 2022 · 16 min read · ✦ Member-only ·
Listen

Semantic Textual Similarity

From Jaccard to OpenAI, implement the best NLP algorithm for your semantic textual similarity projects

303 5

Marie Stephen Leo
864 Followers

Head of Data | ML at scale | GCP | AWS | linkedin.com/in/marie-stephen-leo

More from Medium

Albers Uzila in Level Up Coding
GloVe and fastText Clearly Explained: Extracting Features from Text Data

Andrea D'Ag... in Towards Data ...
How to compute text similarity on a website with TF-IDF in Python

Amy @GrabNGo... in GrabNGoI...
Topic Modeling with Deep Learning Using Python BERTopic

Semantic Textual Similarity

From Jaccard to OpenAI, implement the best NLP algorithm for your semantic textual similarity projects

Photo by Iñaki del Olmo on Unsplash

🎬 Introduction

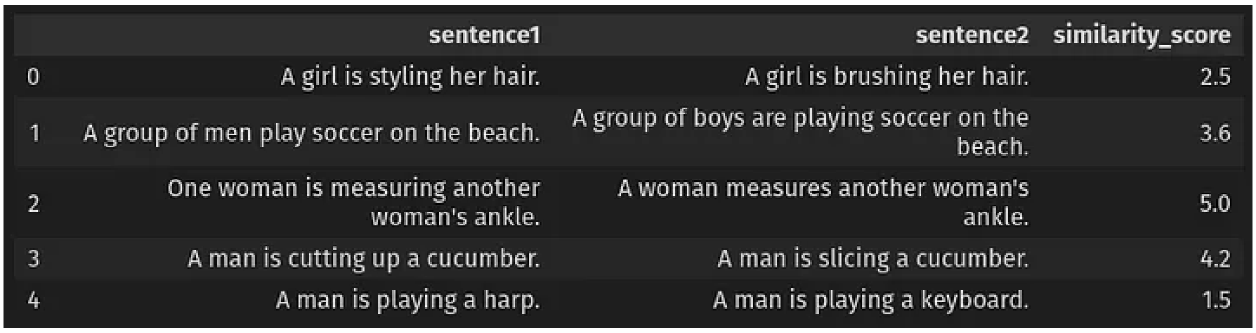
Natural Language Processing (NLP) has tremendous real-world applications in information extraction, natural language understanding, and natural language generation. Comparing the similarity between natural language texts is essential to many information extraction applications such as Google search, Spotify's Podcast search, Home Depot's product search, etc. The semantic textual similarity (STS) problem attempts to compare two texts and decide whether they are similar in meaning. It was a notoriously hard problem due to the nuances of natural language where two texts could be similar despite not having a single word in common!

While this challenge has existed for a long time, recent advancements in NLP have opened up many algorithms to tackle it. Some methods take in two texts as input and directly provide a score on how similar the two texts are. Some other techniques only convert each text into a numeric vector called embedding which we can then compare by mathematically computing the distance between their vectors. To recall,

> 💡 An "embedding" vector is a numeric representation of our natural language texts so that our computers can understand the context and meaning of our text.

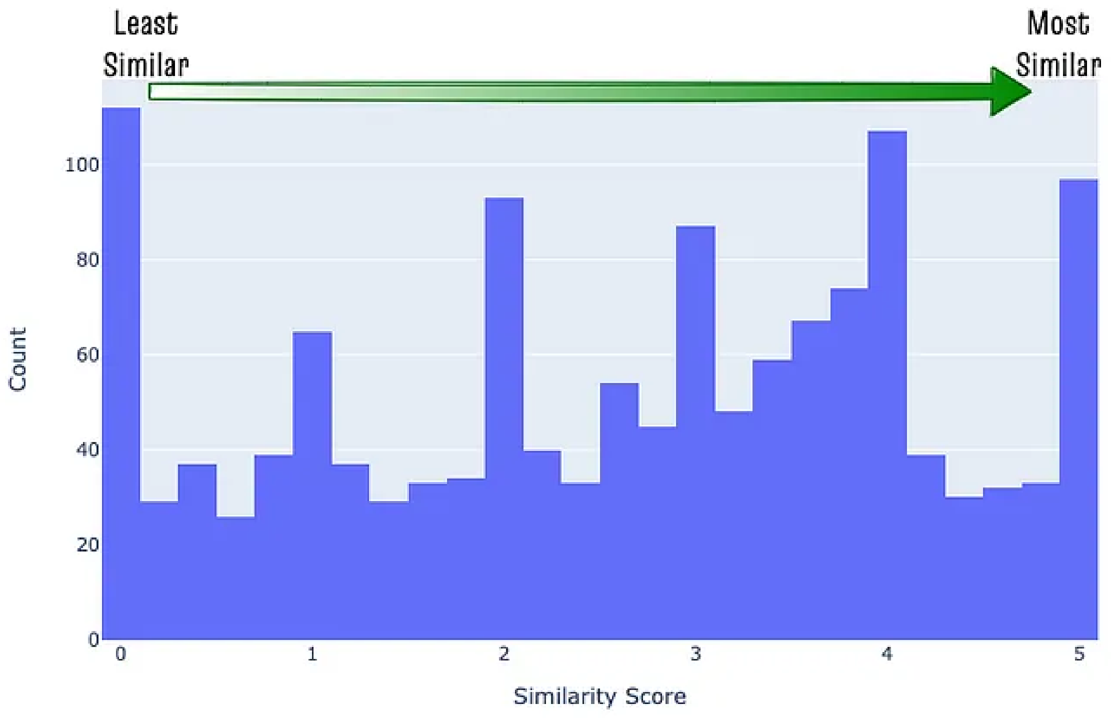
This post will introduce several techniques to tackle the STS problem in various scenarios. My goal is to use pre-trained models as much as possible, which provide excellent results out of the box. At the end of the post, I've included a flowchart to help you decide the best approach to your custom problem. Specifically, in this post, we'll discuss the following. Click on the link to jump to the section:

We'll use the stsb_multi_mt dataset available on Huggingface datasets for this post. The dataset contains pairs of sentences and a positive Real number label indicating each pair's similarity, ranging from 0 (least similar) to 5 (most similar). The authors have licensed the dataset under the Commons Attribution - Share Alike 4.0 International License.



	sentence1	sentence2	similarity_score
0	A girl is styling her hair.	A girl is brushing her hair.	2.5
1	A group of men play soccer on the beach.	A group of boys are playing soccer on the beach.	3.6
2	One woman is measuring another woman's ankle.	A woman measures another woman's ankle.	5.0
3	A man is cutting up a cucumber.	A man is slicing a cucumber.	4.2
4	A man is playing a harp.	A man is playing a keyboard.	1.5

First five rows of the STSB dataset. Image by Author.



STSB test dataset similarity score distribution. Image by Author.

🧩 Setup the environment and download some data

We first need to install all the necessary libraries to test the various embedding strategies. We'll use conda from mini-forge to manage a virtual environment for this project.

Setup the environment and install the necessary libraries

As we'll cover several different algorithms in this post, we need to install the various libraries for all of them to work. Unfortunately, the pyemd library, which is necessary for the WMD algorithm, does not work when pip installed, and so we need to use conda for this project.

```
# Create the necessary directories
mkdir -p semantic_similarity/notebooks semantic_similarity/data/nlp

# Create and activate a conda environment
conda create -n semantic_similarity python=3.8
conda activate semantic_similarity

# Pip install the necessary libraries
pip install -U jupyterlab pandas datasets matplotlib plotly scikit-learn tqdm ipywidgets
pip install -U numpy spacy textdistance fasttext gensim
pip install -U tensorflow tensorflow_hub sentence-transformers openai
conda install pyemd

# Download the Spacy Model
python -m spacy download en_core_web_sm
```

view raw 01_setup.sh hosted with ❤ by GitHub

Code by Author.

Load the STSB dataset

We'll use Huggingface's dataset library to load the STSB dataset into pandas dataframes quickly. The STSB dataset consists of a `train` table and a `test` table. We split the two tables into their respective dataframes `stsb_train` and `stsb_test`.

```
# Imports
from datasets import load_dataset
import pandas as pd
import numpy as np
from tqdm import tqdm
tqdm.pandas()

# Load the English STSB dataset
stsb_dataset = load_dataset('stsb_multi_mt', 'en')
stsb_train = pd.DataFrame(stsb_dataset['train'])
stsb_test = pd.DataFrame(stsb_dataset['test'])

# Check loaded data
print(stsb_train.shape, stsb_test.shape)
stsb_test.head()
```

view raw 02_download_data.py hosted with ❤ by GitHub

Code by Author.

Create some helper functions

Let’s create two helper functions for operations that we’ll repeatedly perform through this post. The first function is to pre-process texts by lemmatizing, lowercasing, and removing numbers and stop words. The second function takes in two columns of text embeddings and returns the row-wise cosine similarity between the two columns.

```
from sklearn.metrics.pairwise import cosine_similarity
import spacy
nlp = spacy.load("en_core_web_sm")

def text_processing(sentence):
"""
Lemmatize, lowercase, remove numbers and stop words

Args:
sentence: The sentence we want to process.

Returns:
A list of processed words
"""
sentence = [token.lemma_.lower()
for token in nlp(sentence)
if token.is_alpha and not token.is_stop]

return sentence


def cos_sim(sentence1_emb, sentence2_emb):
"""
Cosine similarity between two columns of sentence embeddings

Args:
sentence1_emb: sentence1 embedding column
sentence2_emb: sentence2 embedding column

Returns:
The row-wise cosine similarity between the two columns.
For instance is sentence1_emb=[a,b,c] and sentence2_emb=[x,y,z]
Then the result is [cosine_similarity(a,x), cosine_similarity(b,y),
cosine_similarity(c,z)]
"""
cos_sim = cosine_similarity(sentence1_emb, sentence2_emb)
return np.diag(cos_sim)
```

view raw 03_helpers.py hosted with ❤ by GitHub

Code by Author.

Now that we have our environment and data let’s start discussing algorithms!

🧮 Measuring textual similarity with classical non-contextual algorithms

This section will discuss a few techniques to measure similarity between texts using classical non-contextual approaches. In these algorithms, we only use the actual words in similarity calculation without considering the context in which each word appears. As one would expect, these techniques generally have worse performance than more modern contextual approaches.

Jaccard Similarity

The simplest way to compare two texts is to count the number of unique words common to them both. However, if we merely count the number of unique common words, then longer documents would have a higher number of common words. To overcome this bias towards longer documents, in Jaccard similarity, we normalize the number of common unique words to the total number of unique words in both the texts combined.

$$Jaccard\ Similarity = \frac{Number\ of\ common\ unique\ words}{Total\ number\ of\ unique\ words}$$

Jaccard similarity equation. Image by Author.

Jaccard Similarity is one of the several distances that can be trivially calculated in Python using the textdistance library. Note to preprocess the texts to remove stopwords, lower case, and lemmatize them before running Jaccard similarity to ensure that it uses only informative words in the calculation.

```
import textdistance

def jaccard_sim(row):
# Text Processing
sentence1 = text_processing(row['sentence1'])
sentence2 = text_processing(row['sentence2'])

# Jaccard similarity
return textdistance.jaccard.normalized_similarity(sentence1, sentence2)


# Jaccard Similarity
stsb_test['Jaccard_score'] = stsb_test.progress_apply(jaccard_sim, axis=1)
```

view raw 04_jaccard.py hosted with ❤ by GitHub

Code by Author.

We only used words (1-gram) to compute the Jaccard Similarity in the above code. However, the technique can be easily extended to any N-gram as well. Jaccard Similarity using N-grams instead of words (1-gram) is called w-shingling.

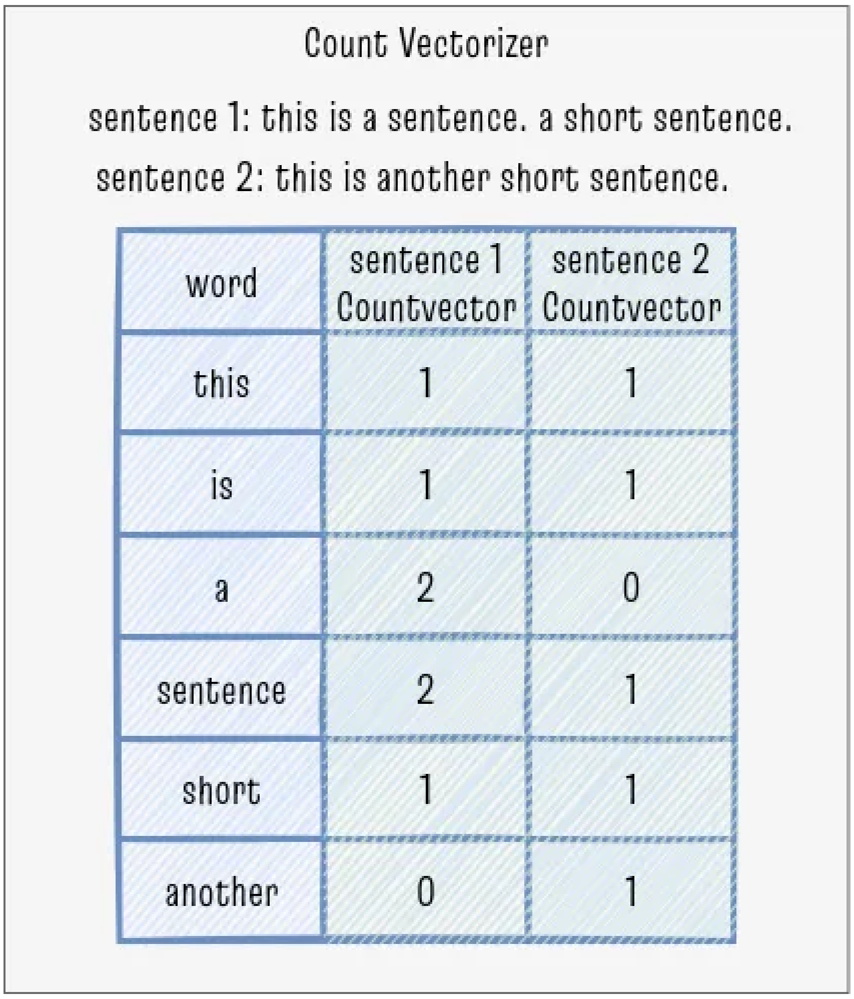
Though Jaccard Similarity and w-shingling are simple methods for measuring text similarity, they perform pretty decently in practice, as shown in the results section at the end of this post!

Bag of Words (BoW)

Bag of Words is a collection of classical methods to extract features from texts and convert them into numeric embedding vectors. We then compare these embedding vectors by computing the cosine similarity between them. There are two popular ways of using the bag of words approach: Count Vectorizer and TFIDF Vectorizer.

Count Vectorizer

This algorithm maps each unique word in the entire text corpus to a unique vector index. The vector values for each document are the number of times each specific word appears in that text. Thus, the vector can consist of integer values, including 0, which indicates that the word does not appear in the text. While Count Vectorizer is simple to understand and implement, its main drawback is that it treats all words equally important irrespective of the actual importance of the word.



Count Vectorizer

sentence 1: this is a sentence. a short sentence.

sentence 2: this is another short sentence.

word	sentence 1 Countvector	sentence 2 Countvector
this	1	1
is	1	1
a	2	0
sentence	2	1
short	1	1
another	0	1

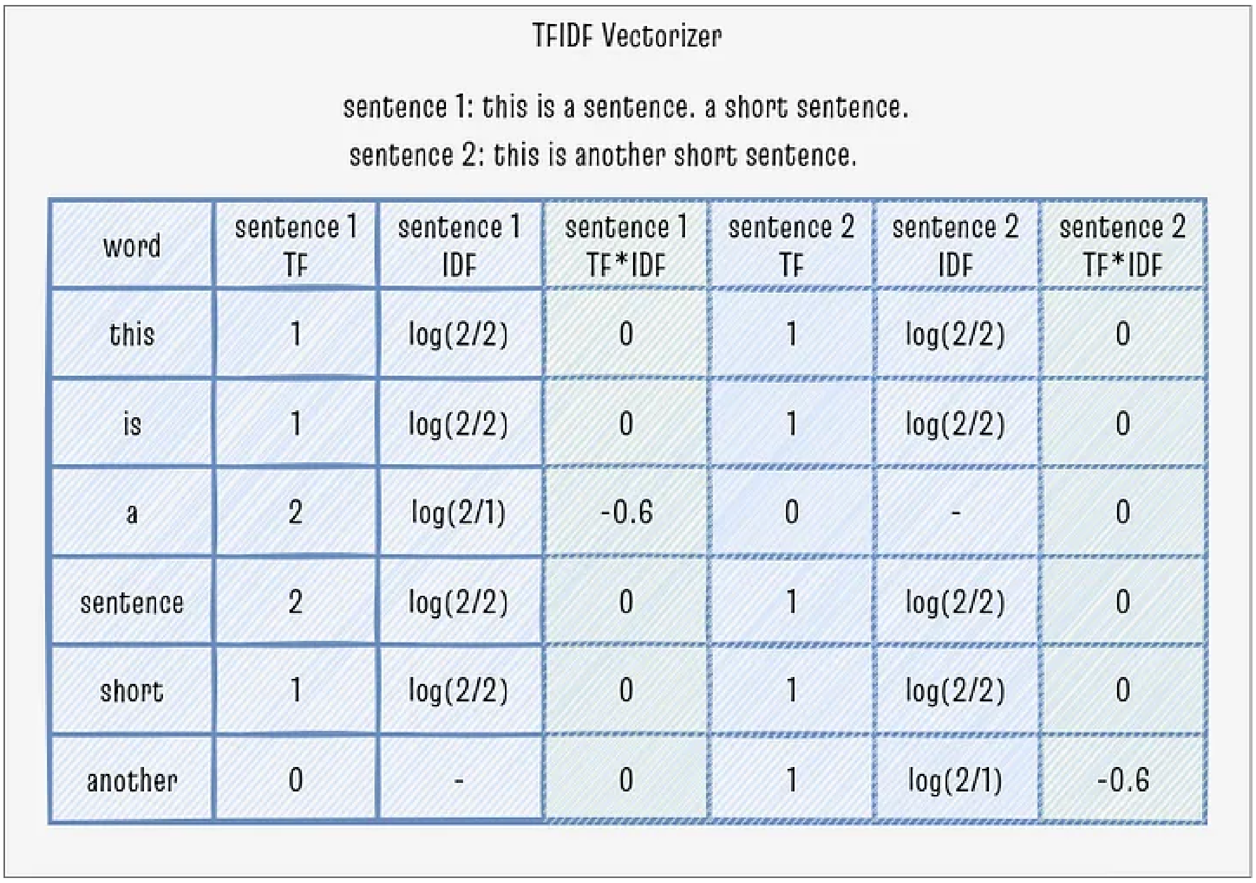
Count Vectorizer example. Image by Author.

TFIDF Vectorizer

To overcome the drawback of the Count Vectorizer, we can use the TFIDF vectorizer. This algorithm also maps each unique word in the entire text corpus to a unique vector index. But instead of a simple count, the values of the vector for each document are the product of two values: Term Frequency (TF) and Inverse Document Frequency (IDF).

1. Term Frequency (TF): Each word's TF is the number of times that word appears in that document and is the same as the value from the count vectorizer. It is a measure of how important that word is to the document.
2. Inverse Document Frequency (IDF): On the other hand, a word's IDF is the log of the inverse of the fraction of documents in which the word appears. It measures how rare the word is in the entire corpus.

Finally, the TFIDF value of each word in each document is the product of the individual TF and IDF scores. The intuition here is that frequent words in one document which are relatively rare across the entire corpus are the crucial words for that document and have a high TFIDF score. Most implementations of TFIDF normalize the values to the document length so that longer documents don't dominate the calculation.


TFIDF Vectorizer

sentence 1: this is a sentence. a short sentence.
sentence 2: this is another short sentence.

word	sentence 1 TF	sentence 1 IDF	sentence 1 TF*IDF	sentence 2 TF	sentence 2 IDF	sentence 2 TF*IDF
this	1	log(2/2)	0	1	log(2/2)	0
is	1	log(2/2)	0	1	log(2/2)	0
a	2	log(2/1)	-0.6	0	-	0
sentence	2	log(2/2)	0	1	log(2/2)	0
short	1	log(2/2)	0	1	log(2/2)	0
another	0	-	0	1	log(2/1)	-0.6

TFIDF Vectorizer example. Image by Author.

Implementing TFIDF in code is simple using the sklearn library, which computes the IDF in a slightly more complicated manner implementing normalization and preventing 0 divisions; you can read more about it here. TFIDF is the only model in this post that we need to train to learn all the unique words in the corpus and their associated IDF values.

```
from sklearn.feature_extraction.text import TfidfVectorizer
model = TfidfVectorizer(lowercase=True, stop_words='english')

# Train the model
```

```
X_train = pd.concat([stsb_train['sentence1'], stsb_train['sentence2']]).unique()
model.fit(X_train)

# Generate Embeddings on Test
sentence1_emb = model.transform(stsb_test['sentence1'])
sentence2_emb = model.transform(stsb_test['sentence2'])

# Cosine Similarity
stsb_test['TFIDF_cosine_score'] = cos_sim(sentence1_emb, sentence2_emb)
```

view raw 05_tfidf.py hosted with ❤ by GitHub

Code by Author.

Though Bag of Words approaches are intuitive and provide us with a vector representation of text, their performance in the real world varies widely. In the STSB task, TFIDF doesn't do as well as Jaccard similarity, as seen in the results section.

Other potential pitfalls of the bag of words approaches are

1. If the number of documents is large, then the vectors generated by this approach will be of very high dimensions as there would be many unique words in the corpus.
2. These vectors would be highly sparse since most words do not appear in most documents.

The following method of measuring textual similarity overcomes these limitations by using pre-trained Word Embeddings.

Word Movers Distance (WMD)

Jaccard Similarity and TFIDF assume that similar texts have many words in common. But, this may not always be the case as even texts without any common non-stop words could be similar, as shown below. One way we can tackle this problem is by using pre-trained word embeddings.

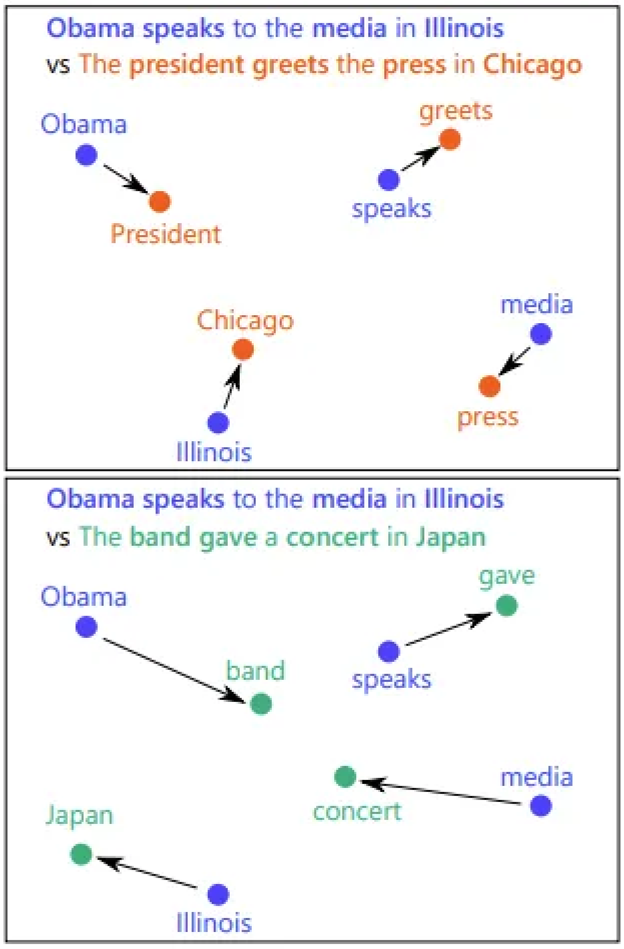
document1: "Obama speaks to the media in Illinois"
document2: "The president greets the press in Chicago"

> 💡 Word embeddings are models that encode words into numeric vectors such that similar words have vectors that are near each other in vector space.

There are several ways to generate word embeddings, the most prominent being Word2Vec, GloVe, and FastText.

Since we need to compare the similarity between texts that contain multiple words, the simplest way to go from individual word embeddings into a single sentence embedding is to calculate the element-wise average of all the word embeddings in that text. However, there is an even better approach to computing the similarity between texts directly from the word embeddings called Word Movers Distance (WMD).

WMD is based on the concept of Earth Movers Distance and is the minimum distance that the word embeddings from one document need to "travel" to reach the word embeddings of the document we are comparing it to. Since each document includes multiple words, the WMD calculation needs to calculate the distances from each word to every other word. It also weights the “travel” by the term frequencies of each word. Thankfully, the gensim library implements this complex computation efficiently using the Fast WMD algorithm. We can easily use it with just a single line of code!


Obama speaks to the media in Illinois
vs The president greets the press in Chicago
Obama
President
greets
speaks
Chicago
Illinois
media
press
Obama speaks to the media in Illinois
vs The band gave a concert in Japan
Obama
band
gave
speaks
Japan
Illinois
concert
media

WMD in action. Both pairs of sentences have no common words. WMD can still find a similar sentence as their words are closer. Image from the Arxiv paper.

Though we can use any word embedding model with WMD, I decide to use the FastText model pre-trained on Wikipedia primarily because FastText uses sub-word information and will never run into Out Of Vocabulary issues that Word2Vec or GloVe might encounter. Take note to preprocess the texts to remove stopwords, lower case, and lemmatize them to ensure that the WMD calculation only uses informative words. Finally, since the WMD is a distance metric while we are looking for a similarity metric, we multiply the WMD value by -1 (Negative WMD) so that more similar texts have numerically larger values.

```
import gensim.downloader as api

# Load the pre-trained model
model = api.load('fasttext-wiki-news-subwords-300')

def word_movers_distance(row):
# Text Processing
sentence1 = text_processing(row['sentence1'])
```

```
sentence2 = text_processing(row['sentence2'])

# Negative Word Movers Distance
return -model.wmdistance(sentence1, sentence2)


# Negative Word Movers Distance
stsb_test['NegWMD_score'] = stsb_test.progress_apply(word_movers_distance, axis=1)
```

view raw 06_wmd.py hosted with ❤ by GitHub

Code by Author.

On the STSB dataset, the Negative WMD score only has a slightly better performance than Jaccard similarity because most sentences in this dataset have many similar words. The performance of NegWMD would be much better than Jaccard on datasets where there are fewer common words between the texts.

One of the limitations of WMD is that the word embeddings used in WMD are non-contextual, where each word gets the same embedding vector irrespective of the context of the rest of the sentence in which it appears. The algorithms in the rest of this post can also use the context to overcome this problem.

🚀 Measuring textual similarity with modern contextual algorithms

This section will discuss several techniques that measure semantic textual similarity, considering the context in which different words appear. These approaches are generally more accurate than the non-contextual approaches.

Universal Sentence Encoder (USE)

In USE, researchers at Google first pre-trained a Transformer-based model on multi-task objectives and then used it for Transfer Learning. To calculate the textual similarity, we first use the pre-trained USE model to compute the contextual word embeddings for each word in the sentence. We then compute the sentence embedding by performing the element-wise sum of all the word vectors and diving by the square root of the length of the sentence to normalize the sentence lengths. Once we have the USE embeddings for each sentence, we can calculate the cosine similarity using the helper function we defined at the beginning of this post. The researchers have open-sourced the pre-trained model on the Tensorflow hub, which we'll use directly.

```
import tensorflow as tf
import tensorflow_hub as hub

# Load the pre-trained model
gpus = tf.config.list_physical_devices('GPU')
for gpu in gpus:
```

```
# Control GPU memory usage
tf.config.experimental.set_memory_growth(gpu, True)

module_url = 'https://tfhub.dev/google/universal-sentence-encoder/4'
model = hub.load(module_url)

# Generate Embeddings
sentence1_emb = model(stsb_test['sentence1']).numpy()
sentence2_emb = model(stsb_test['sentence2']).numpy()

# Cosine Similarity
stsb_test['USE_cosine_score'] = cos_sim(sentence1_emb, sentence2_emb)
```
view raw 07_use.py hosted with ❤ by GitHub

Code by Author.

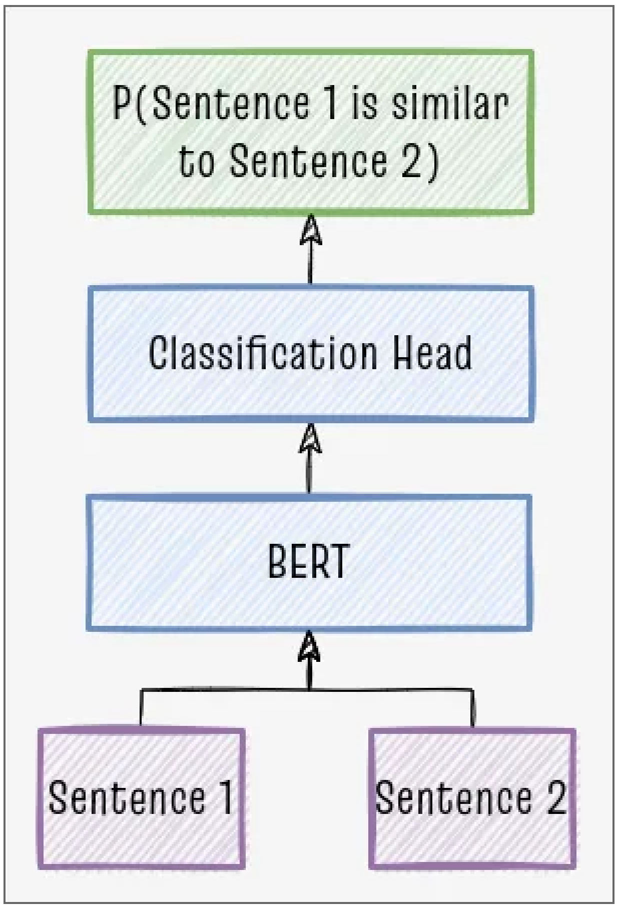
USE achieves a significant 10-point jump on the STSB dataset compared to the NegWMD metric. It shows the potential of using contextual sentence embeddings generated by Transformers in a transfer learning setting. Subsequently, researchers developed even more advanced methods to pre-train Transformer based models using Metric Learning to get even better performance!

Cross Encoder

The advent of the Bidirectional Encoder Representations from Transformers (BERT) model in 2018 ushered in a new era in NLP by beating several benchmarks. Over time, researchers continued to improve over the vanilla BERT model resulting in several notable variants such as RoBERTa, DistilBERT, ALBERT, etc., as discussed in this post.

BERT derives its power from its self-supervised pre-training task called Masked Language Modeling (MLM), where we randomly hide some words and train the model to predict the missing words given the words both before and after the missing word. Training over a massive corpus of text allows BERT to learn the semantic relationships between the various words in the language.

We can use BERT as a Cross Encoder by adding a classification head to the output of the BERT model. The cross-encoder model takes a pair of text documents as input and directly outputs the probability that the two documents are similar. By fine-tuning the pre-trained BERT model on labeled STS datasets, we can achieve state-of-the-art results on STS tasks!



Cross Encoders. Image by Author.

We shall use the sentence_transformers library to efficiently use the various open-source Cross Encoder models trained on SNLI and STS datasets.

```
from sentence_transformers import CrossEncoder

# Load the pre-trained model
model = CrossEncoder('cross-encoder/stsb-roberta-base')

sentence_pairs = []
for sentence1, sentence2 in zip(stsb_test['sentence1'], stsb_test['sentence2']):
sentence_pairs.append([sentence1, sentence2])

stsb_test['SBERT CrossEncoder_score'] = model.predict(sentence_pairs,
show_progress_bar=True)
```

view raw 08_cross_encoder.py hosted with ❤ by GitHub

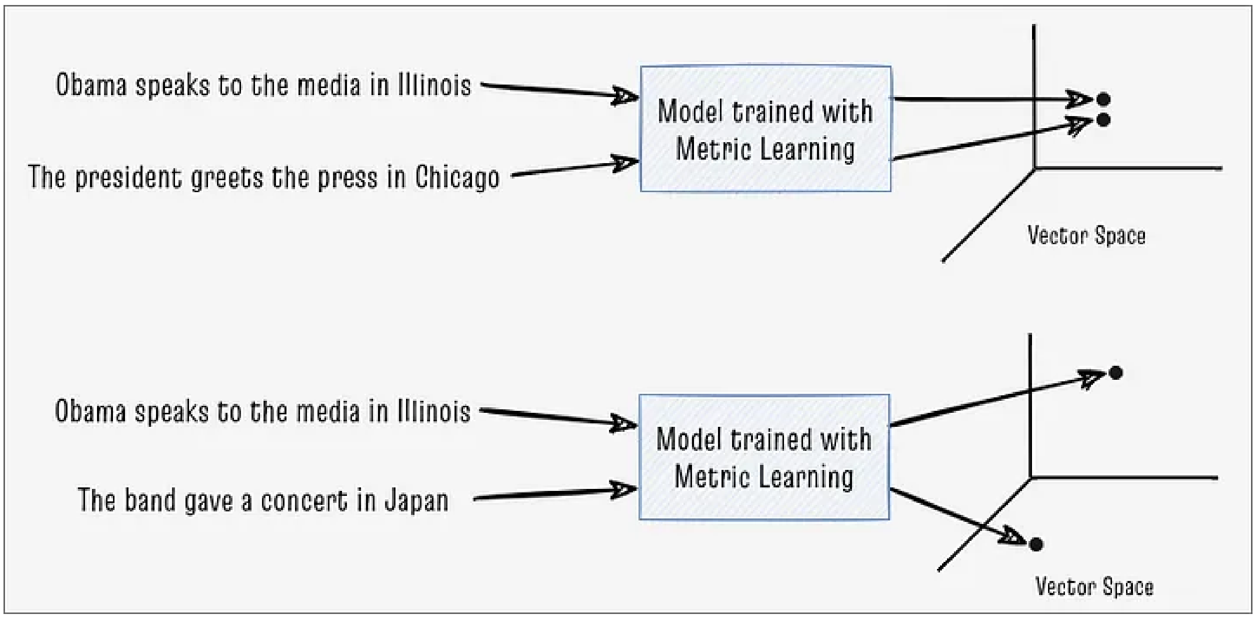
Code by Author.

Cross-encoders do not output any embedding vectors and are thus not very scalable beyond a few thousands of documents. But in general, BERT Cross Encoders provide the best performance in most sentence similarity tasks. In our STSB dataset, the BERT Cross Encoder delivers the best score!

Metric Learning

Metric learning is one of the most promising ways to generate embeddings, especially for similarity search applications. At its very fundamental level, in metric learning,

1. We use a neural network such as a BERT to convert texts to embeddings.
2. We construct these embeddings so that semantically similar texts cluster nearer to each other while dissimilar texts are further apart.


Obama speaks to the media in Illinois
The president greets the press in Chicago
Model trained with Metric Learning
Vector Space
Obama speaks to the media in Illinois
The band gave a concert in Japan
Model trained with Metric Learning
Vector Space

Example of natural language texts in vector space. Image by Author, inspired by Medium Blog Post by Brian Williams.

Training metric learning models requires innovation in how the data is processed and how the model is trained, as described in detail in my previous post. After training the model with this approach, we can find the similarity between two texts by mathematically computing the cosine similarity between their vectors.

SBERT Bi-Encoder

Sentence Transformers (also known as SBERT) are the current state-of-the-art NLP sentence embeddings. It uses BERT and its variants as the base model and is pre-trained utilizing a type of metric learning called contrastive learning. In contrastive learning, the contrastive loss function compares whether two embeddings are similar (0) or dissimilar (1).

The core idea of Sentence Transformers is as follows.

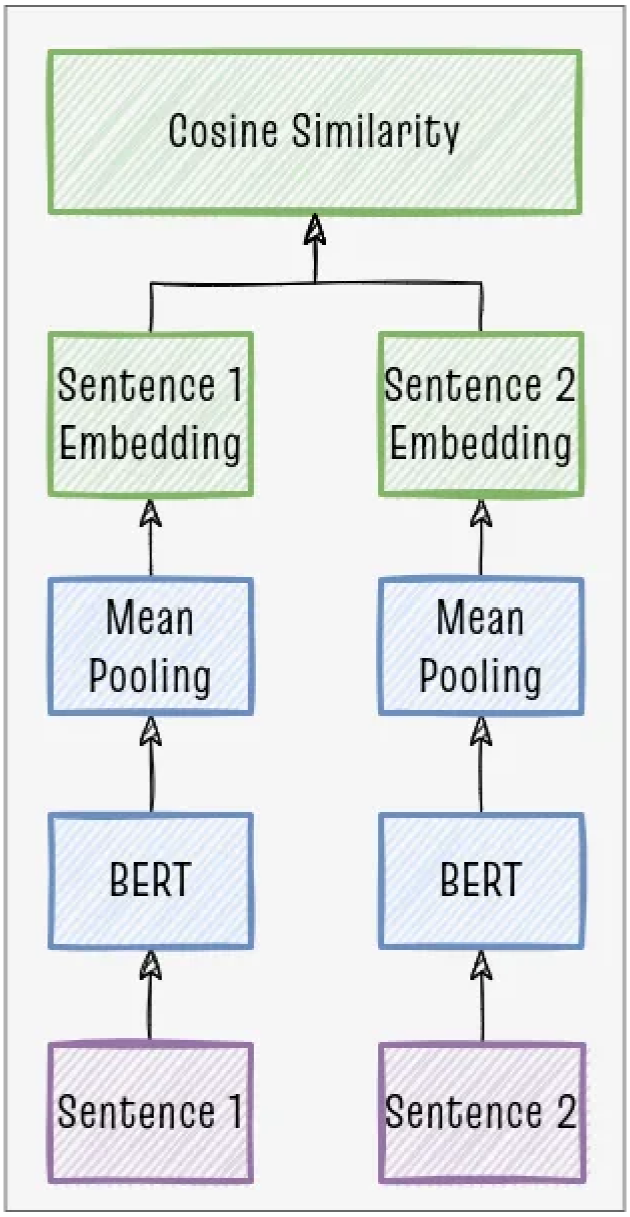
1. Use the labeled SNLI dataset or the STS datasets as training data. These datasets contain several thousand pairs of sentences labeled as either similar or dissimilar.
2. For each text in the training dataset, compute the contextual word embeddings of that text using any pre-trained BERT model as an encoder.
3. Compute the element-wise average of all the token embeddings to obtain a single fixed dimension sentence embedding for the entire text. This operation is called Mean Pooling.
4. Train the model using a Siamese Network architecture with contrastive loss. The model's objective is to move the embeddings for similar texts closer together so that the distance between them is

near 0. Conversely, the model aims to move the embeddings from dissimilar texts further away from each other such that the distance between them is large.

5. After we complete training the model, we can compare any two texts by computing the cosine similarity between the embeddings of those two texts.

A Bi-Encoder Sentence Transformer model takes in one text at a time as input and outputs a fixed dimension embedding vector as the output. We can then compare any two documents by computing the cosine similarity between the embeddings of those two documents.

Though the Bi-Encoder Sentence Transformer has slightly lower performance than the Cross Encoder on our STSB dataset, Bi-Encoders shine when scaling to billions or even trillions of documents by combining them with vector search databases such as Milvus!



Bi-Encoders. Image by Author.

We shall use the sentence_transformers library to efficiently use the various open-source SBERT Bi-Encoder models trained on SNLI and STS datasets.

```
from sentence_transformers import SentenceTransformer
```

```

# Load the pre-trained model
model = SentenceTransformer('stsb-mpnet-base-v2')

# Generate Embeddings
sentence1_emb = model.encode(stsb_test['sentence1'], show_progress_bar=True)
sentence2_emb = model.encode(stsb_test['sentence2'], show_progress_bar=True)

# Cosine Similarity
stsb_test['SBERT BiEncoder_cosine_score'] = cos_sim(sentence1_emb,
sentence2_emb)
```

view raw 09_bi_encoder.py hosted with ❤ by GitHub

Code by Author.

Despite the strong performance on the STSB dataset, Sentence Transformers are, unfortunately, fully supervised models and require a large labeled corpus of sentence pairs to train. Thus adopting Sentence Transformers to new domains is a time-consuming and expensive process of collecting massive, high-quality labeled data. Fortunately, some cutting-edge research in semi-supervised and self-supervised learning shows promising results!

SimCSE

In my previous post on Computer Vision embeddings, I introduced SimCLR, a self-supervised algorithm for learning image embeddings using contrastive loss. In this post, let's discuss SimCSE, the NLP equivalent of SimCLR.
SimCSE stands for Simple Contrastive Learning of Sentence Embeddings. We can train it either as a supervised model if labeled data is available or in a completely unsupervised fashion!

The core idea of SimCSE is as follows.

1. Given a text document, compute the embeddings of that text using any pre-trained BERT model as an encoder and take the embeddings of the [CLS] token.
2. Create two noisy versions of the same text embedding by applying two different Dropout masks on the original embedding. These two noisy embeddings generated from the same input text are considered a "positive" pair, and the model expects them to have a cosine distance of 0. Experiments in the SimCSE paper found that a Dropout rate of 0.1 was optimal for the STSB dataset.
3. We consider the embeddings from all the other texts in the batch as "negatives." The model expects the "negatives" to have a cosine distance of 1 to the target text embeddings from the previous step. The loss function then updates the parameters of the encoder model such that the embeddings move closer to our expectations.
4. Supervised SimCSE has one additional step where we use a Natural Language Inference (NLI) labeled dataset to obtain "positive" pairs from texts that are labeled "entailment" and "negative" pairs from texts that are labeled as "contradiction."

The below image explains the whole process conceptually.

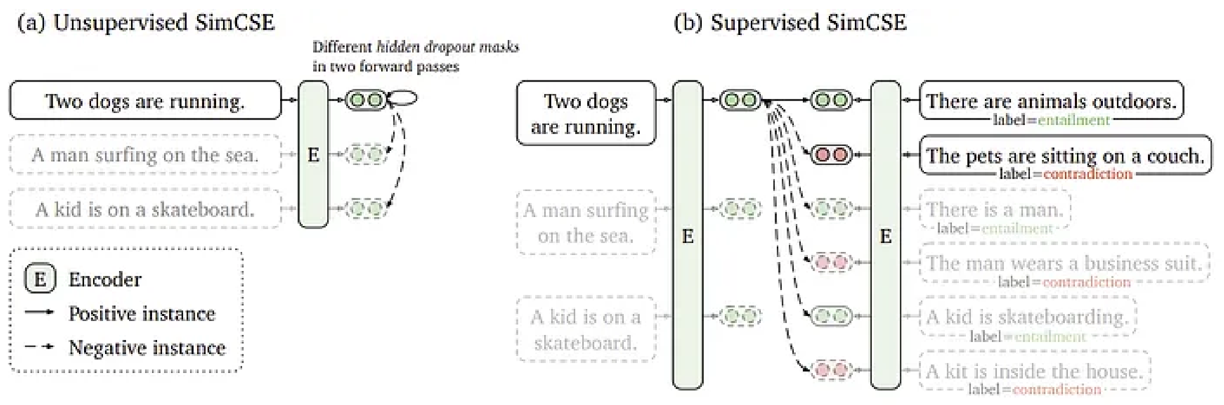


Figure 1: (a) Unsupervised SimCSE predicts the input sentence itself from in-batch negatives, with different hidden dropout masks applied. (b) Supervised SimCSE leverages the NLI datasets and takes the entailment (premise-hypothesis) pairs as positives, and contradiction pairs as well as other in-batch instances as negatives.

Unsupervised and Supervised SimCSE. Image from the arxiv paper.

SimCSE models are Bi-Encoder Sentence Transformer models trained using the SimCSE approach. Thus, we can directly reuse all the code from the Bi-Encoder Sentence Transformer model but change the pre-trained model to the SimCSE models.

```
########## Supervised ##########
# Load the pre-trained model
model = SentenceTransformer('princeton-nlp/sup-simcse-roberta-large')

# Generate Embeddings
sentence1_emb = model.encode(stsb_test['sentence1'], show_progress_bar=True)
sentence2_emb = model.encode(stsb_test['sentence2'], show_progress_bar=True)

# Cosine Similarity
stsb_test['SimCSE Supervised_cosine_score'] = cos_sim(sentence1_emb, sentence2_emb)


########## Un-Supervised ##########
# Load the pre-trained model
model = SentenceTransformer('princeton-nlp/unsup-simcse-roberta-large')

# Generate Embeddings
sentence1_emb = model.encode(stsb_test['sentence1'], show_progress_bar=True)
sentence2_emb = model.encode(stsb_test['sentence2'], show_progress_bar=True)

# Cosine Similarity
stsb_test['SimCSE Unsupervised_cosine_score'] = cos_sim(sentence1_emb, sentence2_emb)
```

view raw 10_simcse.py hosted with ❤ by GitHub

Code by Author.

From the results on the STSB dataset, we see that the un-supervised SimCSE model has a significant performance degradation compared to the supervised SimCSE and other supervised Sentence

Transformer models. However, despite being trained completely un-supervised just using Dropout to create “positive” pairs, unsupervised SimCSE could comfortably beat other methods such as WMD and USE. Thus, unsupervised SimCSE would be the go-to method in domains where sufficient labeled data is unavailable or expensive to collect.

OpenAI

One of the significant limitations of all the BERT-based models, such as Sentence Transformers and SimCSE, is that they can only encode texts up to 512 tokens long. This limitation is because the BERT family of models has a 512 token input limit. Also, since BERT’s sub-word tokenizer might split each word into multiple tokens, the texts that can be converted to embeddings using these techniques need to have lesser than 512 words. It might pose a problem if you need to compare the similarity between longer documents. The non-BERT-based models do not face this limitation, but their performance is worse than the BERT-based models, so we prefer to avoid them if a better alternative is available.

The final method to generate state-of-the-art embeddings is to use a paid hosted service such as OpenAI’s embeddings endpoint. It supports texts up to 2048 tokens, and thus it is perfect for longer text documents that are longer than the 512 token limitations of BERT. However, the OpenAI endpoints are expensive, larger in dimensions (12288 dimensions vs. 768 for the BERT-based models), and suffer a performance penalty compared to the best in class free open-sourced Sentence Transformer models.

To give you an idea of how expensive it is, I spent around **USD20** to generate the OpenAI Davinci embeddings on this small STSB dataset, even after ensuring I only generate the embeddings once per unique text! Scaling this embedding generation to an enormous corpus would be too expensive even for a large organization. It might be cheaper to collect labeled data and train your model in-house. Hence, I believe this technique has limited uses in the real world, but I still include it in this article for completion.

```
import openai
import os
import pickle
openai.api_key = 'update_your_openai_API_key_here'

if os.path.exists('../data/nlp/davinci_emb.pkl'):
print('Loading Davinci Embeddings')
with open('../data/nlp/davinci_emb.pkl', 'rb') as f:
davinci_emb = pickle.load(f)
else:
print('Querying Davinci Embeddings')
davinci_emb = {}
engine='text-similarity-davinci-001'

unique_sentences = list(set(stsb_test['sentence1'].values.tolist() +
stsb_test['sentence2'].values.tolist()))
for sentence in tqdm(unique_sentences):
if sentence not in davinci_emb.keys():
davinci_emb[sentence] = openai.Embedding.create(input = [sentence],
engine=engine)['data'][0]['embedding']
# Save embeddings to file
with open('../data/nlp/davinci_emb.pkl', 'wb') as f:
pickle.dump(davinci_emb, f)

# Generate Embeddings
sentence1_emb = [davinci_emb[sentence] for sentence in stsb_test['sentence1']]
sentence2_emb = [davinci_emb[sentence] for sentence in stsb_test['sentence2']]

# Cosine Similarity
stsb_test['OpenAI Davinci_cosine_score'] = cos_sim(sentence1_emb,
sentence2_emb)
```

view raw 11_openai.py hosted with ❤ by GitHub

Code by Author.

🏅 Results & Conclusion

Finally, let's compare the results of the various text similarity methods I've covered in this post. Many papers on Semantic Textual Similarity use the Spearman Rank Correlation Coefficient to measure the performance of the models as it is not sensitive to outliers, non-linear relationships, or non-normally distributed data as described in this paper.

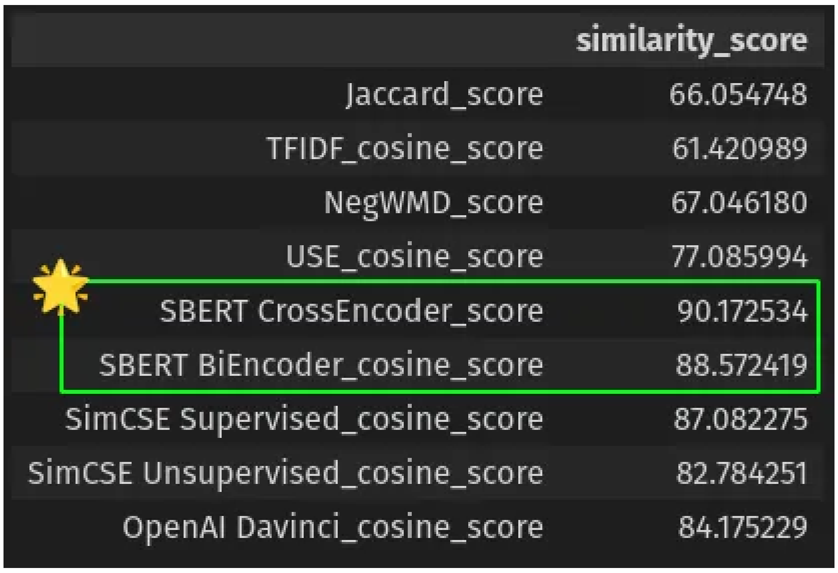
Thus, we shall calculate the Spearman Rank Correlation between the similarity scores from each method to the actual `similarity_score` label provided by the STSB dataset. The calculation is relatively straightforward using the inbuilt `corr` method in `pandas` as shown below.

```
score_cols = [col for col in stsb_test.columns if '_score' in col]

# Spearman Rank Correlation
spearman_rank_corr = stsb_test[score_cols].corr(method='spearman').iloc[1:,
0:1]*100
spearman_rank_corr.head(10)
```

view raw 12_spearman_rank_correlation.py hosted with ❤ by GitHub

Code by Author.

The Spearman Rank Correlation scores below show that SBERT Cross Encoder has the best performance, followed closely by SBERT Bi-Encoder. The unsupervised SimCSE's performance is quite promising as it is much better than the other methods like Jaccard, TFIDF, WMD, and USE. Finally, OpenAI Davinci shows good performance, but its cost outweighs most benefits of accepting texts longer than 512 tokens.



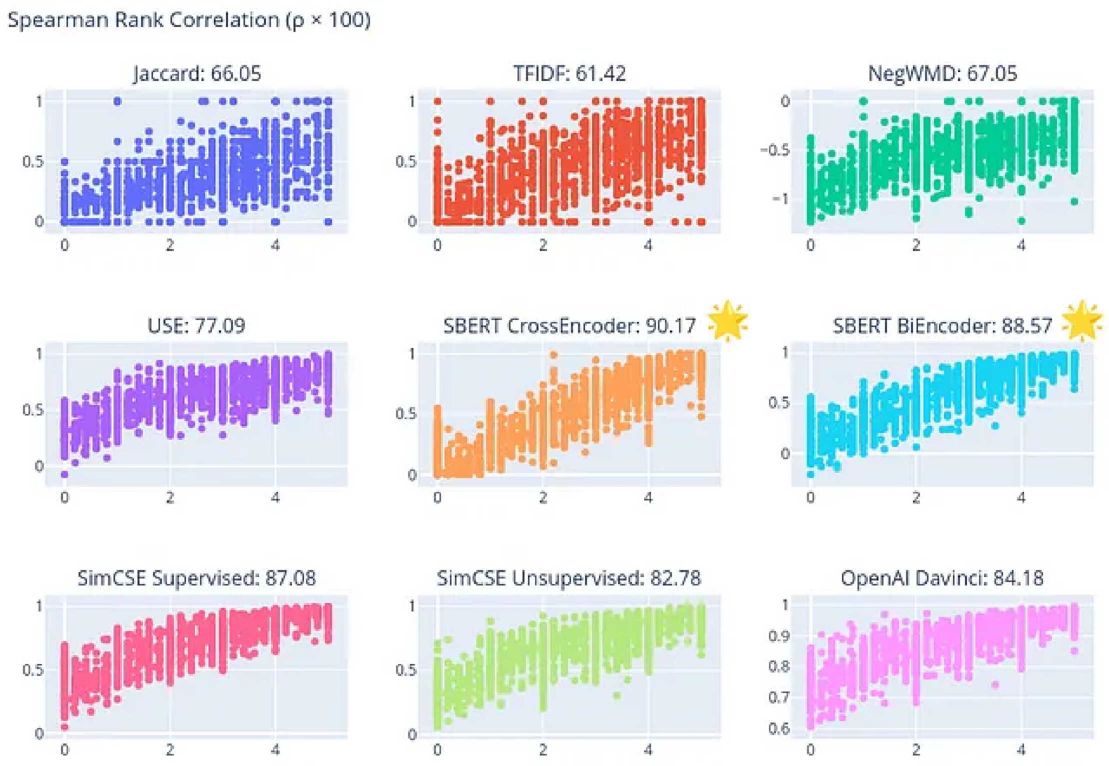
	similarity_score
Jaccard_score	66.054748
TFIDF_cosine_score	61.420989
NegWMD_score	67.046180
USE_cosine_score	77.085994
SBERT CrossEncoder_score	90.172534
SBERT BiEncoder_cosine_score	88.572419
SimCSE Supervised_cosine_score	87.082275
SimCSE Unsupervised_cosine_score	82.784251
OpenAI Davinci_cosine_score	84.175229

Spearman Rank Correlation performance of various algorithms on the STSB dataset. Image by Author.

We plot the correlations between the actual similarity_score and the predicted similarity from the various algorithms. Visually, SBERT and SimCSE's correlations look pretty strong!

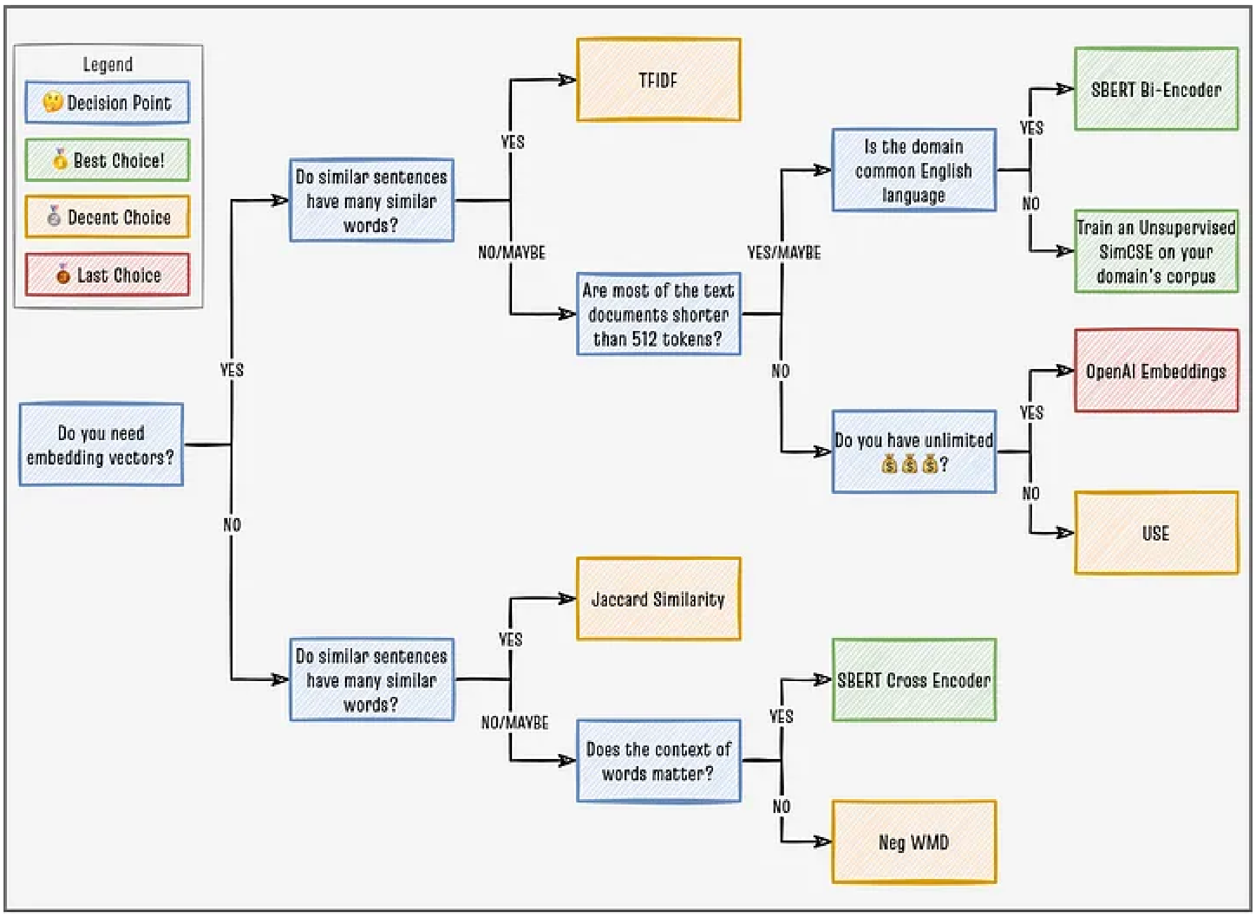
```
from plotly.subplots import make_subplots
import plotly.graph_objects as go

nrows = 4
ncols = 3
plot_array = np.arange(0, nrows*ncols).reshape(nrows, ncols)

subplot_titles = [f'{row.Index.split("_")[0]}: {row.similarity_score:.2f}' for row in spearman_rank_corr.itertuples()]
fig = make_subplots(rows=nrows, cols=ncols, subplot_titles=subplot_titles)

for index, score in enumerate(spearman_rank_corr.index):
row, col = np.argwhere(plot_array == index)[0]

fig.add_trace(
go.Scatter(
x=stsb_test[score_cols[0]],
y=stsb_test[score],
mode='markers',
),
row=row+1, col=col+1
)


fig.update_layout(height=700, width=1000, title_text='Spearman Rank Correlation (ρ × 100)', showlegend=False)
fig.show()
```

view raw 13_plot.py hosted with ❤ by GitHub
Code by Author.



Spearman Rank Correlation Coefficient and actual vs. predicted similarity score for the various methods introduced in this post. Image by Author.

I developed the below flowchart to help with choosing a method for your own Semantic Textual Similarity task. I hope it enables you to pick the best technique for your use cases! Thank you for reading.


Legend
Decision Point
Best Choice!
Decent Choice
Last Choice
Do you need embedding vectors?
YES
NO
Do similar sentences have many similar words?
YES
NO/MAYBE
TFIDF
Are most of the text documents shorter than 512 tokens?
YES/MAYBE
NO
Is the domain common English language
YES
NO
SBERT Bi-Encoder
Train an Unsupervised SimCSE on your domain's corpus
Do you have unlimited 💰💰💰?
YES
NO
OpenAI Embeddings
USE
Do similar sentences have many similar words?
YES
NO/MAYBE
Jaccard Similarity
Does the context of words matter?
YES
NO
SBERT Cross Encoder
Neg WMD

Flowchart to choose a semantic textual similarity algorithm. Image by Author.