Ultimate Guide To Text Similarity With Python - NewsCatcher

Learn the different similarity measures and text embedding techniques. Play around with code examples and develop a general intuition.

In this article, you will learn about different similarity metrics and text embedding techniques. By the end, you'll have a good grasp of when to use what metrics and embedding techniques. You'll also get to play around with them to help establish a general intuition.

You can find the accompanying web app here.

Topics Covered In This Article

Understanding Similarity

Similarity is the distance between two vectors where the vector dimensions represent the features of two objects. In simple terms, similarity is the measure of how different or alike two data objects are. If the distance is small, the objects are said to have a high degree of similarity and vice versa. Generally, it is measured in the range 0 to 1. This score in the range of [0, 1] is called the similarity score.

An important point to remember about similarity is that it's subjective and highly dependent on the domain and use case. For example, two cars can be similar because of simple things like the manufacturing company, color, price range, or technical details like fuel type, wheelbase, horsepower. So, special care should be taken when calculating similarity across features that are unrelated to each other or not relevant to the problem.

As simple as the idea may be, similarity forms the basis of many machine learning techniques.

similarly, K-means clustering utilizes similarity measures to assign data points to appropriate clusters. Even recommendation engines use neighborhood-based collaborative filtering methods which use similarity to identify a user's neighbors.

The use of similarity measures is quite prominent in the field of natural language processing. Everything from information retrieval systems, search engines, paraphrase detection to text classification, automated document linking, spell correction makes use of similarity measures.

Text Similarity

Take a look at the following sentences:

- The bottle is empty.
- There is nothing in the bottle.

As humans, it is very obvious to us that the two sentences mean the same thing despite being written in completely different formats. But how do we make an algorithm come to that same conclusion?

The first part of this problem is representation. How do we represent the text? We could leave the text as it is or convert it into feature vectors using a suitable text embedding technique. Once we have the text representation, we can compute the similarity score using one of the many distance/similarity measures.

Let's dive deeper into the two aspects of the problem, starting with the similarity measures.

Similarity Measures

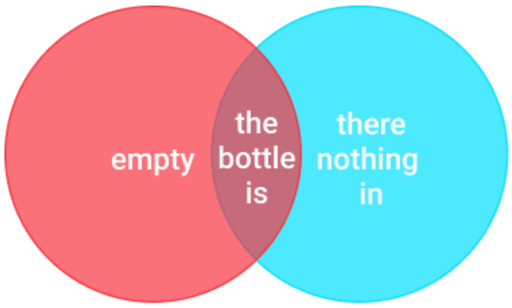
Jaccard Index

defined as the size of the intersection of two sets divided by the size of the union. Let's continue with our previous example:

Sentence 1: The bottle is empty.

Sentence 2: There is nothing in the bottle.

To calculate the similarity using Jaccard similarity, we will first perform text normalization to reduce words their roots/lemmas. There are no words to reduce in the case of our example sentences, so we can move on to the next part. Drawing a Venn diagram of the sentences we get:

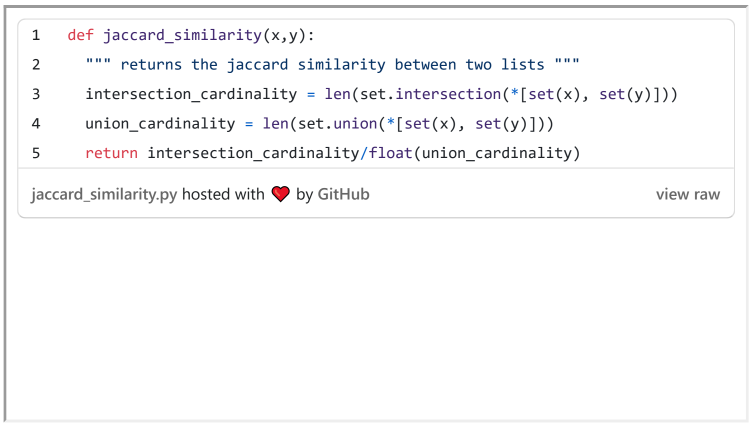


Size of the intersection of the two sets: **3**

Size of the union of the two sets: **1+3+3 = 7**

Using the Jaccard index, we get a similarity score of 3/7 = **0.42**

Python function for Jaccard similarity:



```python
def jaccard_similarity(x,y):
  """ returns the jaccard similarity between two lists """
  intersection_cardinality = len(set.intersection(*[set(x), set(y)]))
  union_cardinality = len(set.union(*[set(x), set(y)]))
  return intersection_cardinality/float(union_cardinality)
```

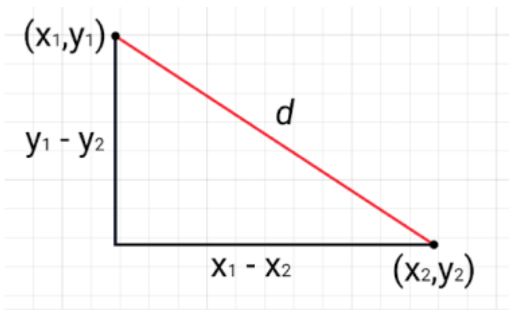
Testing the function for our example sentences

```

# OUPUT
0.42857142857142855
```

test_jaccard.py hosted with ❤ by GitHub view raw

Euclidean Distance

Euclidean distance, or L2 norm, is the most commonly used form of the Minkowski distance. Generally speaking, when people talk about distance, they refer to Euclidean distance. It uses the Pythagoras theorem to calculate the distance between two points as indicated in the figure below:



The larger the distance d between two vectors, the lower the similarity score and vice versa.

Let’s compute the similarity between our example statements using Euclidean distance:

```python
from math import sqrt, pow, exp

def squared_sum(x):
  """ return 3 rounded square rooted value """

  return round(sqrt(sum([a*a for a in x])),3)

def euclidean_distance(x,y):
  """ return euclidean distance between two lists """

  return sqrt(sum(pow(a-b,2) for a, b in zip(x, y)))
```

euclidean_distance.py hosted with ❤ by GitHub view raw

To compute the Euclidean distance we need vectors, so we’ll use spaCy’s in-built Word2Vec model to create text embeddings. (We’ll learn more about this later in the article)

```

# OUTPUT
1.8646982721454675
```

get_euclidean_distance.py hosted with ❤ by GitHub view raw

Okay, so we have the Euclidean distance of 1.86, but what does that mean? See, the problem with using distance is that it’s hard to make sense if there is nothing to compare to. The distances can vary from 0 to infinity, we need to use some way to normalize them to the range of 0 to 1.

Although we have our typical normalization formula that uses mean and standard deviation, it is sensitive to outliers. That means if there are a few extremely large distances, every other distance will become smaller as a consequence of the normalization operation. So the best option here is to use something like the Euler’s constant as follows:

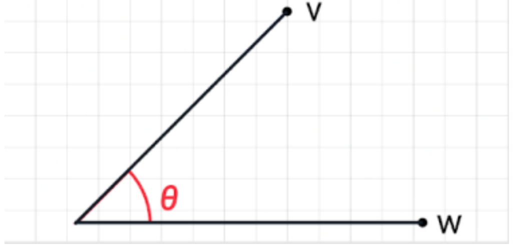
$$\frac{1}{e^d}$$

```python
def distance_to_similarity(distance):
  return 1/exp(distance)

distance_to_similarity(distance)

# OUTPUT
0.8450570465624478
```

distance_to_similarity.py hosted with ❤ by GitHub view raw

Cosine Similarity

Cosine Similarity computes the similarity of two vectors as the cosine of the angle between two vectors. It determines whether two vectors are pointing in roughly the same direction. So if the



It is given as:

$$cos(v, w) = \frac{v.w}{||v|| \times ||w||}$$

Where **||v||** represents the length of the vector **v**, θ denotes the angle between **v** and **w**, and '**.**' denotes the dot product operator.

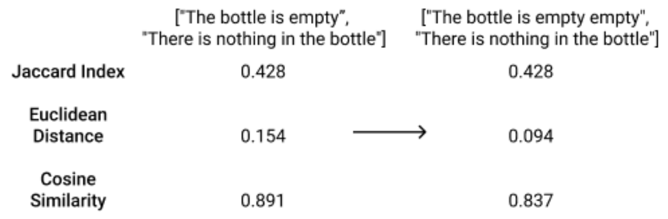
```python
def cos_similarity(x,y):
  """ return cosine similarity between two lists """

  numerator = sum(a*b for a,b in zip(x,y))
  denominator = squared_sum(x)*squared_sum(y)
  return round(numerator/float(denominator),3)

cos_similarity(embeddings[0], embeddings[1])

# OUTPUT
0.891
```

cos_similarity.py hosted with ❤ by GitHub view raw

What Metric To Use?

Jaccard similarity takes into account only the set of unique words for each text document. This makes it the likely candidate for assessing the similarity of documents when repetition is not an issue. A prime example of such an application is comparing product descriptions. For instance, if a term like "HD" or "thermal efficiency" is used multiple times in one description and just once in another, the Euclidean distance and cosine similarity would drop. On the other hand, if the total number of unique words stays the same, the Jaccard similarity will remain unchanged.

example sentence.



	["The bottle is empty", "There is nothing in the bottle"]		["The bottle is empty empty", "There is nothing in the bottle"]
Jaccard Index	0.428		0.428
Euclidean Distance	0.154	$\longrightarrow$	0.094
Cosine Similarity	0.891		0.837

That being said, Jaccard similarity is rarely used when working with text data as it does not work with text embeddings. This means that is limited to assessing the lexical similarity of text, i.e., how similar documents are on a word level.

As far as cosine and Euclidean metrics are concerned, the differentiating factor between the two is that cosine similarity is not affected by the magnitude/length of the feature vectors. Let's say we are creating a topic tagging algorithm. If a word (e.g. senate) occurs more frequently in document 1 than it does in document 2, we could assume that document 1 is more related to the topic of Politics. However, it could also be the case that we are working with news articles of different lengths. Then, the word 'senate' probably occurred more in document 1 simply because it was way longer. As we saw earlier when the word 'empty' was repeated, cosine similarity is less sensitive to a difference in lengths.

In addition to that, Euclidean distance doesn't work well with the sparse vectors of text embeddings. **So cosine similarity is generally preferred over Euclidean distance when working with text data**. The only length-sensitive text similarity use case that comes to mind is plagiarism detection.

Text Embeddings

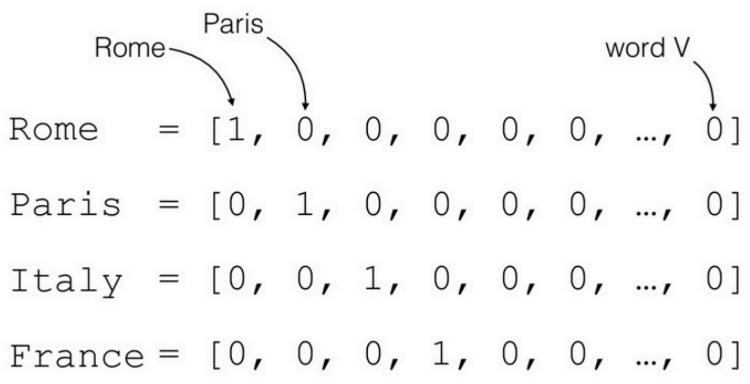
Humans can easily understand and derive meaning from words, but computers don't have encode them into a numeric form, so the computer can apply mathematical formulas and operations to make sense of them. Even beyond the task of text similarity, representing documents in the form of numbers and vectors is an active area of study.

Word Embeddings

Simply put, word embedding is the vector representation of a word. They aim to capture the meaning, context, and semantic relationships of the words. A lot of the word embeddings are created based on the notion of the “distributional hypothesis” introduced by Zellig Harris: *words that are used close to one another typically have the same meaning.*

One-Hot Encoding & Bag-of-Words

The most straightforward way to numerically represent words is through the one-hot encoding method. The idea is simple, create a vector with the size of the total number of unique words in the corpora. Each unique word has a unique feature and will be represented by a 1 with 0s everywhere else.

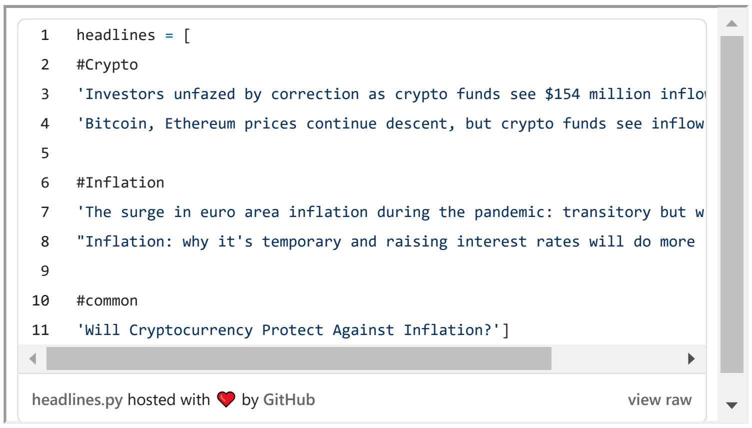


```
Rome   = [1, 0, 0, 0, 0, 0, …, 0]
Paris  = [0, 1, 0, 0, 0, 0, …, 0]
Italy  = [0, 0, 1, 0, 0, 0, …, 0]
France = [0, 0, 0, 1, 0, 0, …, 0]
```

Documents contain large chunks of text with the possibility of repetition. Simply marking the (also called count vectorizing), each word is represented by its count instead of 1. Regardless of that, both these approaches create huge, sparse vectors that capture absolutely no relational information.

Implementation

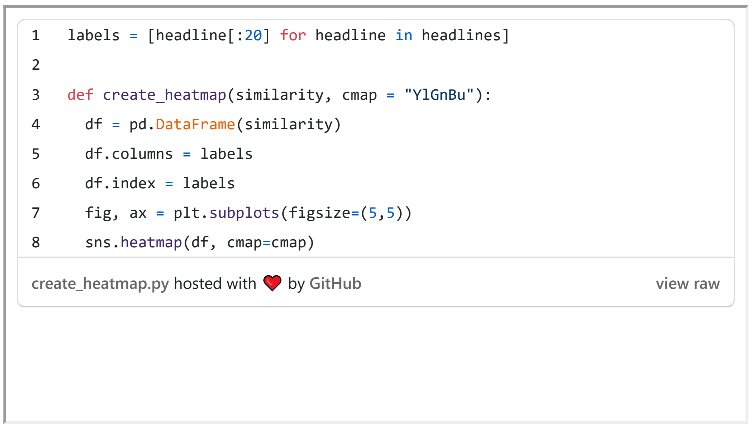
The scikit-learn module implements this method, let's use it to calculate the similarity of the following news headlines:



```python
headlines = [
#Crypto
'Investors unfazed by correction as crypto funds see $154 million inflo
'Bitcoin, Ethereum prices continue descent, but crypto funds see inflow

#Inflation
'The surge in euro area inflation during the pandemic: transitory but w
"Inflation: why it's temporary and raising interest rates will do more

#common
'Will Cryptocurrency Protect Against Inflation?']
```

headlines.py hosted with ❤ by GitHub view raw

To make for better output, let's create a function that creates a heatmap of the similarity scores.



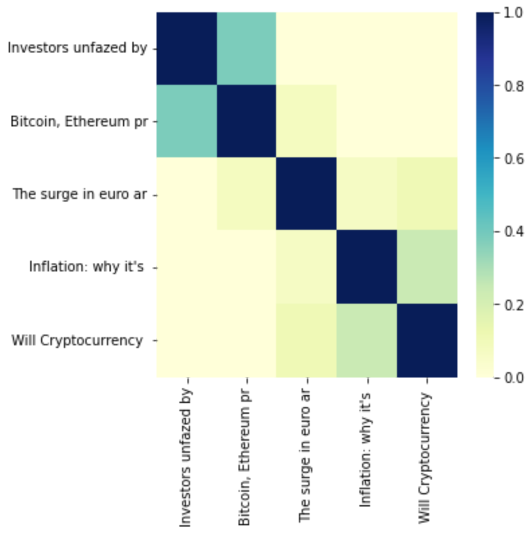
```python
labels = [headline[:20] for headline in headlines]

def create_heatmap(similarity, cmap = "YlGnBu"):
  df = pd.DataFrame(similarity)
  df.columns = labels
  df.index = labels
  fig, ax = plt.subplots(figsize=(5,5))
  sns.heatmap(df, cmap=cmap)
```

create_heatmap.py hosted with ❤ by GitHub view raw

Now that we have our data and helper function, we can test **countvectorizer**

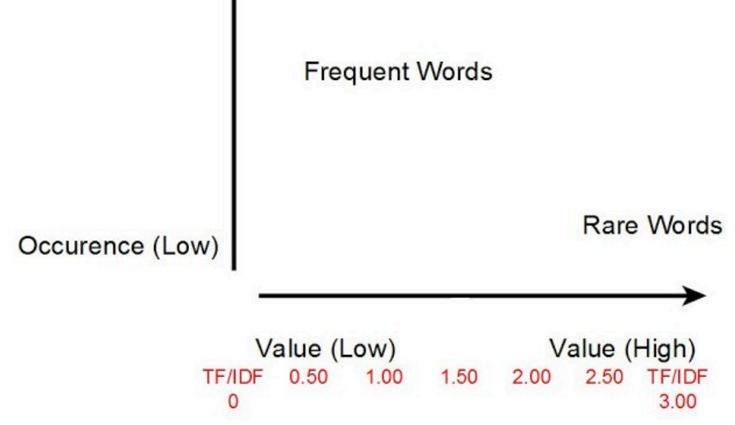
```
X = vectorizer.fit_transform(headlines)
arr = X.toarray()

create_heatmap(cosine_similarity(arr))
```

count_vectorizer.py hosted with ❤ by GitHub   view raw



Term Frequency-Inverse Document Frequency (TF-IDF)

TF-IDF vectors are an extension of the one-hot encoding model. Instead of considering the frequency of words in one document, the frequency of words across the whole corpus is taken into account. The big idea is that words that occur a lot everywhere carry very little meaning or significance. For instance, trivial words like “and”, “or”, “is” don’t carry as much significance as nouns and proper nouns that occur less frequently.

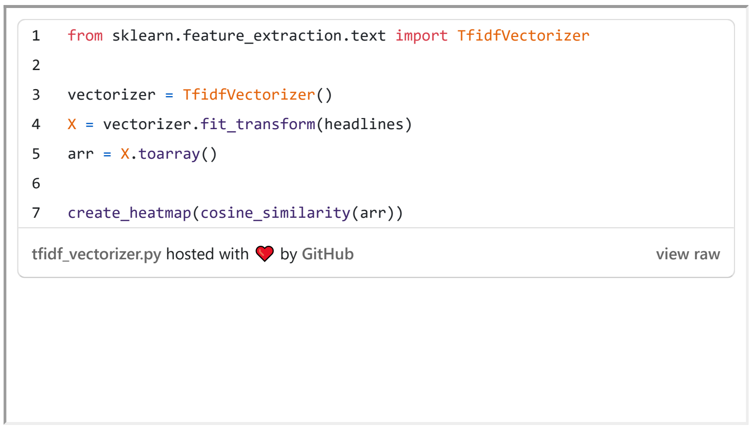


Mathematically, Term Frequency (TF) is the number of times a word appears in a document divided by the total number of words in the document. And Inverse Document Frequency **(IDF) = log(N/n)** where N is the total number of documents and n is the number of documents a term has appeared in. The TF-IDF value for a word is the product of the term frequency and the inverse document frequency.

Although TF-IDF vectors offer a slight improvement over simple count vectorizing, they still have very high dimensionality and don’t capture semantic relationships.

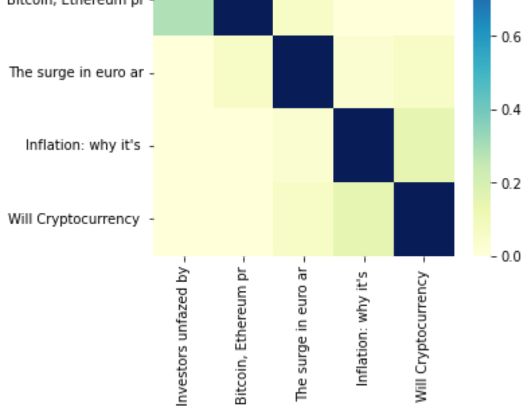
Implementation

Scikit-learn also offers a `**TfidfVectorizer**` class for creating TF-IDF vectors from the text.



```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(headlines)
arr = X.toarray()

create_heatmap(cosine_similarity(arr))
```

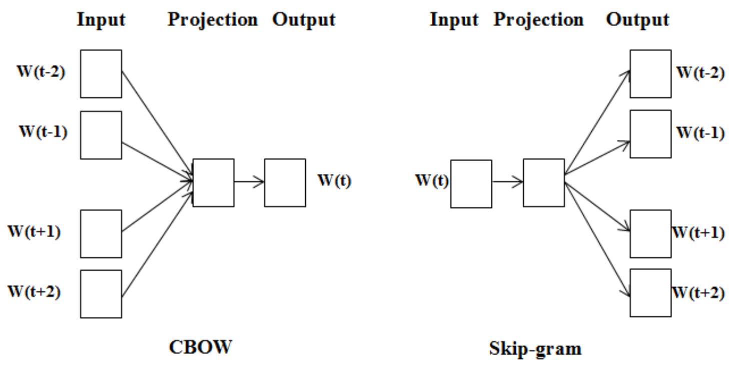
tfidf_vectorizer.py hosted with ❤ by GitHub view raw



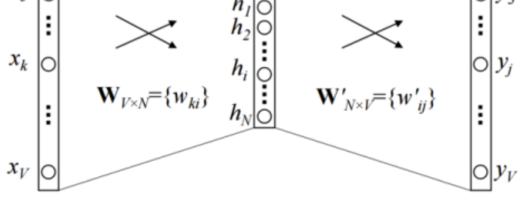
Word2Vec

Word2Vec is a predictive method for forming word embeddings. Unlike the previous methods that need to be "trained" on the working corpus, Word2Vec is a pre-trained two-layer neural network. It takes as input the text corpus and outputs a set of feature vectors that represent words in that corpus. It uses one of two neural network-based methods:

- Continuous Bag Of Words (CBOW)
- Skip-Gram



Continuous Bag Of Words takes the context of each word as the input and tries to predict the word corresponding to the context. Here, context simply means the surrounding words.

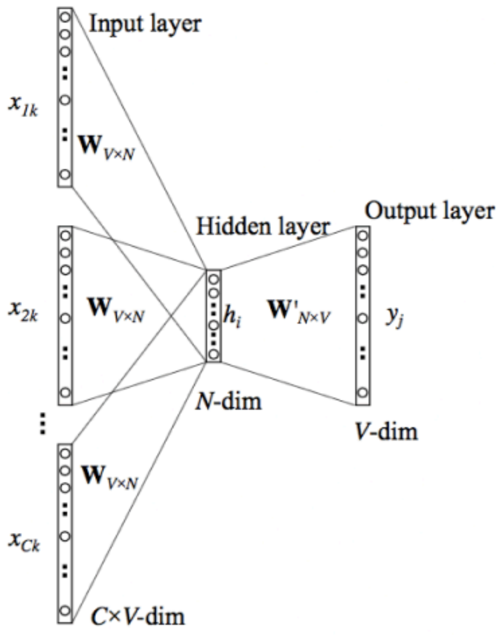


x_k
x_V
h_1
h_2
h_i
h_N
$\mathbf{W}_{V\times N}=\{w_{ki}\}$
$\mathbf{W}'_{N\times V}=\{w'_{ij}\}$
y_j
y_V

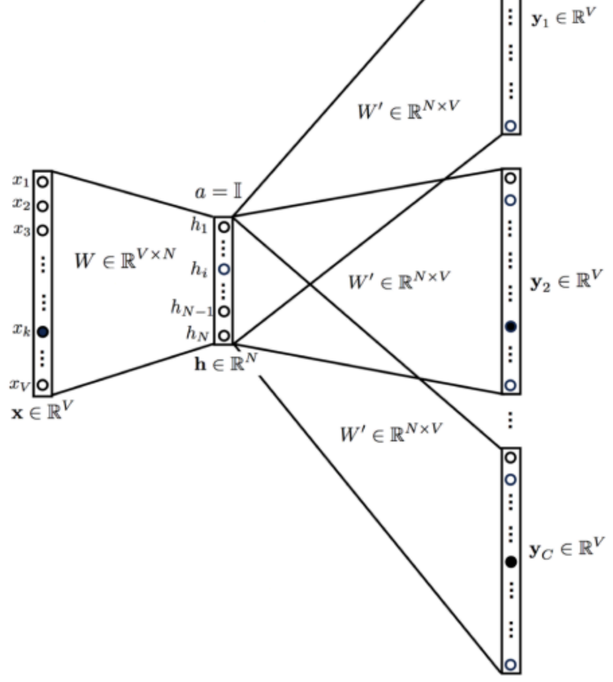
For example, consider the greeting: “Have a nice day”

Let’s say we use the word ‘nice’ as the input to the neural network and we are trying to predict the word ‘day’. We will use the one-hot encoding of the input word ‘nice’, then measure and optimize for the output error of the target word ‘day’. In this process of trying to predict the target word, this shallow network learns its vector representation.

Just like how this model used a single context word to predict the target, it can be extended to use multiple context words to do the same:



So CBOW generates word representations based on the context words, but there’s another way to do the same. We can use the target word, i.e., the word we want to generate the representation for, to predict the context.


$\mathbf{y}_1 \in \mathbb{R}^V$
$W' \in \mathbb{R}^{N\times V}$
x_1
x_2
x_3
x_k
x_V
$W \in \mathbb{R}^{V\times N}$
$a = \mathbb{I}$
h_1
h_i
h_{N-1}
h_N
$\mathbf{h} \in \mathbb{R}^N$
$W' \in \mathbb{R}^{N\times V}$
$\mathbf{y}_2 \in \mathbb{R}^V$
$\mathbf{x} \in \mathbb{R}^V$
$W' \in \mathbb{R}^{N\times V}$
$\mathbf{y}_C \in \mathbb{R}^V$

That's what **Skip-gram** does. In the process of predicting the context words, the Skip-gram model learns the vector representation of the target word. representations are generated using the context words.

When To Use What?

Intuitively, the CBOW task is much simpler as it is using multiple inputs to predict one target while Skip-gram relies on one-word inputs. This is reflected in the faster convergence time of CBOW, in the original paper the authors wrote that CBOW took hours to train, while Skip-gram took 3 days.

Continuing on the same train of thought, CBOW is better at learning syntactic relationships between words while skip-gram is better at understanding the semantic relationships. In practical terms, this means that for a word like 'dog', CBOW would return morphologically similar words like plurals like 'dogs'. On the other hand, Skip-gram would consider morphologically different but semantically similar words like 'cat' or 'hamster'.

words. Because even if some words appear more times during the training they considered one at a time. CBOW is prone to overfit frequent words because they can appear several times in the same set of context words. This characteristic also allows Skip-gram to be more efficient in terms amount of documents required to achieve good performance.

TLDR: Skip-gram works better when working with a small amount of data, focuses on semantic similarity of words, and represents rare words well. On the other hand, CBOW is faster, focuses more on the morphological similarity of words, and needs more data to achieve similar performance.

Implementation

Word2Vec is used in spaCy to create word vectors, one important thing to note: In order to create word vectors we need larger spaCy models. For example, For example, the medium or large English model, but not the small one. So if we want to use vectors, we will go with a model that ends in '**md**' or '**lg**'. More details about this can be found here.

Install spaCy and download one of the larger models:

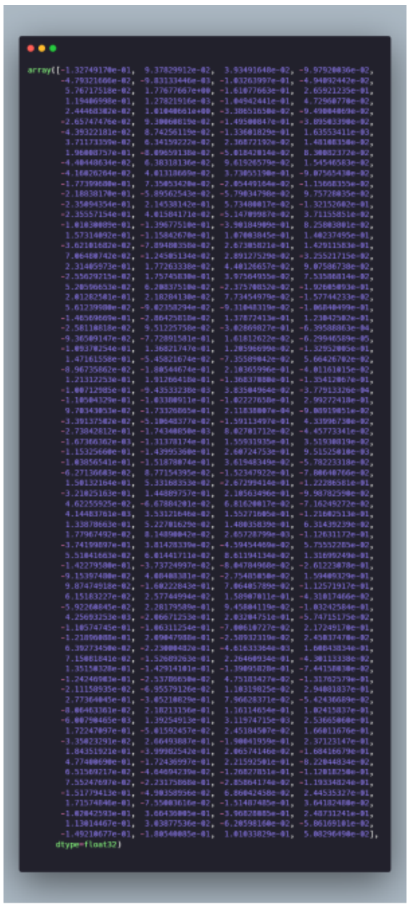
```
python -m spacy download en_core_web_md
```

Create a pipeline object and use it to create to the Docs for the headlines:

We can look up the embedding vector for the `**Doc**` or individual tokens using the `**.vector**` attribute. Let's see what the headline embeddings look like.

```
print(docs[0].vector)
```

word_vector_example.py hosted with ❤ by GitHub view raw

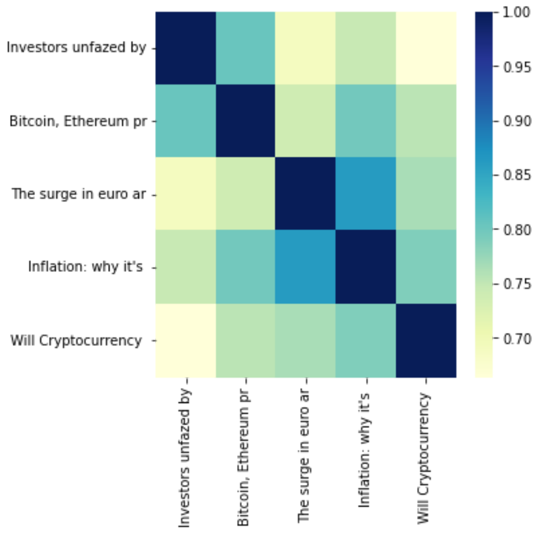


```
array([-1.32749170e-01,  9.37829912e-02,  3.93491640e-02, -9.97920036e-02,
       -4.79321066e-02, -9.83133446e-03, -1.03263997e-01, -4.94092442e-02,
        5.76717518e-02,  1.77677667e+00, -1.61077663e-01,  2.65921235e-01,
        1.19406998e-01,  1.27821916e-03, -1.04942441e-01,  4.72960770e-02,
        2.44468302e-02,  1.01040661e+00, -3.38651650e-02, -9.49004069e-02,
       -2.65747476e-02,  9.30060819e-02, -1.49500847e-01, -3.09503390e-02,
       -4.39322181e-02,  8.74256119e-02, -1.33601829e-01,  1.63553411e-03,
        3.71173359e-02,  6.34159222e-02,  2.36872192e-02,  1.48108350e-02,
        1.96008757e-01, -8.09659138e-02, -5.01842014e-02,  8.30082372e-02,
       -4.40448634e-02,  6.38318136e-02,  9.61926579e-02,  1.54546583e-02,
       -4.16026264e-02,  4.01318669e-02,  3.73055190e-01, -9.07565430e-02,
       -1.77399680e-01,  7.35053420e-02, -2.05449164e-02, -1.15668355e-02,
       -2.18838170e-01, -5.89562543e-02, -5.79034798e-02,  9.75720035e-02,
       -2.35094354e-01,  2.14538142e-01,  5.73400017e-02, -1.32152602e-01,
       -2.35557154e-01,  4.01584171e-02, -5.14709987e-02,  3.71155851e-02,
       -1.01830089e-01, -1.39677510e-01, -3.90184909e-01,  8.25803801e-02,
        1.57314092e-01, -1.15842670e-01,  1.07003845e-01,  1.40237495e-01,
       -3.62101682e-02, -7.89480358e-02,  2.67305821e-01,  1.42911583e-01,
        7.06480742e-02, -1.24505134e-02,  2.89127529e-02, -3.25521715e-02,
        2.31405973e-01,  1.77263338e-02,  4.40126657e-02,  9.07586738e-02,
       -2.55629215e-02,  1.75745830e-01,  3.97564955e-02,  7.53588814e-02,
        5.20596653e-02,  6.20837510e-02, -2.37570852e-02, -1.92605093e-01,
        2.01282501e-01,  2.18204130e-02,  7.73454979e-02, -1.57744233e-02,
        5.61239980e-02, -9.02358294e-02, -9.31048319e-02, -1.06840499e-01,
       -1.46569669e-01, -2.86425810e-02,  1.37872413e-01,  1.23042502e-01,
       -2.58110818e-02,  9.51225758e-02, -3.02869827e-01, -6.39588863e-04,
       -9.36509147e-02, -7.72891581e-01,  1.61812622e-02, -6.29946589e-05,
       -1.09370254e-01,  1.36821747e-01,  1.20596699e-02, -1.32952005e-01,
        1.47161558e-01, -5.45821674e-02, -7.35589042e-02,  5.66426702e-02,
       -8.96735862e-02, -1.80544674e-01,  2.10365996e-01, -4.01161015e-02,
        1.21312253e-01,  1.91266418e-01, -1.36837880e-01, -1.35412067e-01,
       -1.00712985e-01, -9.43533238e-03,  3.83504964e-02, -3.77913326e-04,
       -1.10504329e-01, -1.03380911e-01, -1.02227658e-01,  2.99272418e-01,
        9.70343053e-02, -1.73326865e-01,  2.11838007e-04, -9.00919851e-02,
       -3.39137502e-02, -5.10648377e-02, -1.59113497e-01,  4.33996730e-02,
       -2.73842812e-01, -1.74340850e-03,  8.02701712e-02, -4.45773341e-02,
       -1.67366362e-03, -1.31370174e-01,  1.55931935e-01,  3.51930819e-02,
       -1.15325660e-01, -1.43995360e-01,  2.60724753e-01,  9.51525010e-03,
       -1.03856541e-01, -1.51878074e-01,  3.61948349e-02, -5.78223318e-02,
       -6.27136603e-02,  8.77154395e-02, -1.52347922e-01, -7.80640766e-02,
        1.50132164e-01,  5.33168353e-02, -2.67299414e-01, -1.22286581e-01,
       -3.21025163e-01,  1.44889757e-01,  2.10563496e-01, -9.98782590e-02,
        4.62255925e-02, -6.67884201e-02,  6.81620017e-02, -7.16249272e-02,
        4.14483761e-03,  3.53121646e-02,  1.55271605e-01, -1.21602513e-01,
        1.33878663e-01,  5.22701629e-02,  1.48035839e-01,  6.31439239e-02,
        1.77967492e-02,  8.14890042e-02,  2.65728799e-03, -1.12631172e-01,
       -3.74199897e-01,  3.81428339e-02, -4.59454489e-02,  5.75552285e-02,
        5.51041663e-02,  6.01441711e-02,  8.61194134e-02,  1.31699249e-01,
       -1.42279580e-01, -3.73724997e-02, -8.04784968e-02, -2.61223078e-01,
       -9.15397400e-02,  4.00408381e-02, -2.75485850e-02,  1.59409329e-01,
        9.87474918e-02, -1.60222843e-01,  7.06405789e-02, -1.12571917e-01,
        6.15183227e-02,  2.57744994e-02,  1.58907011e-01, -4.31017466e-02,
       -5.92260845e-02,  2.28179509e-01,  9.45804119e-02, -1.03242584e-01,
        4.25693253e-03, -2.06671253e-01,  2.03204751e-01, -5.74715175e-02,
       -1.10574745e-01, -1.06311254e-01, -7.00610727e-02,  2.17249170e-01,
       -1.21896088e-01,  2.09047988e-01, -2.50932319e-02,  2.45037470e-02,
        6.39273450e-02, -2.23000482e-01, -4.61633364e-03,  1.60043834e-01,
        7.15081841e-02, -1.52689263e-01,  2.26460934e-01, -4.30113338e-02,
        1.35150328e-01, -1.42914101e-01, -1.39095820e-01, -7.44150030e-02,
       -1.24246903e-01, -2.53700650e-02,  4.75183427e-02, -1.31762579e-01,
       -2.11158935e-02, -6.95579126e-02,  1.10319825e-02,  2.94081837e-01,
        2.77364045e-01, -3.05210829e-01,  7.96628371e-02, -5.42436689e-02,
       -8.06463361e-02,  2.18213156e-01,  1.16114654e-01,  1.02415837e-01,
       -6.00790465e-03,  1.39254913e-01,  3.11974715e-03,  2.53665060e-01,
        1.72247097e-01, -5.01592457e-02,  2.45184507e-02,  1.66011676e-01,
       -3.35023291e-02,  2.66493887e-01, -1.90041959e-01,  2.37123147e-01,
        1.84351921e-01, -3.99982542e-01,  2.06574146e-02, -1.68416679e-01,
        4.77400690e-01, -1.72436997e-01,  2.21592501e-01, -8.22044834e-02,
        6.51589217e-02, -4.04094239e-02, -1.26827851e-01, -1.12018250e-01,
        7.55247697e-02, -2.23175080e-01, -2.85864174e-02, -1.19334824e-01,
       -1.51779413e-01, -4.90358956e-02,  6.86042458e-02,  2.44535327e-01,
        1.71574846e-01, -7.55003616e-02, -1.51487485e-01,  3.64182480e-02,
       -1.02042593e-01,  3.66430005e-01, -3.96828885e-01,  2.48731241e-01,
        1.13014467e-01,  3.03877536e-02, -6.20598160e-02, -5.06169101e-02,
       -1.49210677e-01, -1.80540085e-01,  1.01033829e-01,  5.00296490e-02],
      dtype=float32)
```

The result is a 300-dimensional vector of the first headline. We can use these vectors to calculate calculates the cosine similarity.

```python
similarity = []
  for i in range(len(docs)):
    row = []
    for j in range(len(docs)):
      row.append(docs[i].similarity(docs[j]))
    similarity.append(row)
create_heatmap(similarity)
```

word2vec_cosine_similarity.py hosted with ❤ by GitHub view raw



Need For Contextual Embeddings

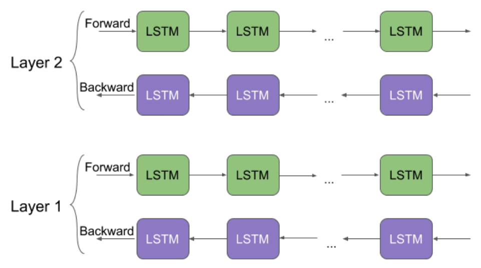
So far all the text embeddings we have covered learn a global word embedding. They first build a vocabulary of all the unique words in the document, then learn similar representations for words that appear together frequently. The issue with such word representations is that the words' contextual meaning is ignored. In practice, this approach to word representation does not address polysemy or the coexistence of many possible meanings for a given word or phrase. For instance, consider the following statement :

“After stealing gold from the bank vault, the bank robber was seen fishing on the river bank.”

'bank' has two different meanings in the sentence and needs to have two different representations in the embedding space. Contextual embedding methods like BERT and ELMo learn sequence-level semantics by considering the sequence of all words in the document. As a result, these techniques learn different representations for polysemous words like 'bank' in the above example, based on their context.

ELMo

ELMo computes the embeddings from the internal states of a two-layer bidirectional Language Model (LM), thus the name "ELMo": **E**mbeddings from **L**anguage **Mo**dels. It assigns each word a representation that is a function of the entire corpus of sentences. ELMo embeddings are a function of all of the internal layers of the biLM. Different layers encode different kinds of information for the same word. For example, the lower levels work well for Part-Of-Speech tagging, while the higher levels are better at dealing with polysemous words.



Concatenating the activations of all layers allows ELMo to combine a wide range of word representations that perform better on downstream tasks. In addition to that, ELMo works on the character level instead of words. This enables it to take advantage of sub-word

This means that the way ELMo is used is quite different compared to traditional embedding methods. Instead of having a dictionary of words and their corresponding vectors, ELMo creates embeddings on the fly.

Implementation

There are many implementations of ELMo, we'll be trying out the **simpe-elmo** module. We'll also need to download a pre-trained model to create the embeddings

```
wget http://vectors.nlpl.eu/repository/20/209.zip
```

download_elmo hosted with ❤ by GitHub	view raw

Now, let's create an ElmoModel instance and load the pre-trained model we just downloaded.

```python
from simple_elmo import ElmoModel
model = ElmoModel()
model.load("/content/209.zip")

sentence = "After stealing gold from the bank vault, the bank robber was se
```

create_elmo_model.py hosted with ❤ by GitHub	view raw

Create the text embedding

```
Tensor shape: (1, 92, 1024)
```

elmo_vectors.py hosted with ❤ by GitHub view raw

The Tensor's second dimension of 92 corresponds to the 92 characters in the sentence. To get the word embeddings we can average the embeddings of the characters for each word. Our main concern is ELMo's ability to extract contextual information so let's focus only on the three instances of the word "bank":

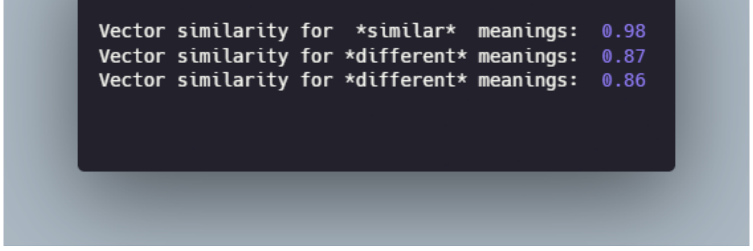
```python
vault = np.sum(elmo_vectors[0][29:33], axis = 0)/4
robber = np.sum(elmo_vectors[0][45:49], axis = 0)/4
river = np.sum(elmo_vectors[0][87:91], axis = 0)/4
```

elmo_word_vector.py hosted with ❤ by GitHub view raw

We can now evaluate how similar the three instances are:

```python
diff_bank_1 = cosine_similarity(vault, river)
diff_bank_2 = cosine_similarity(river, robber)
same_bank = cosine_similarity(vault, robber)

print('Vector similarity for  *similar*  meanings:  %.2f' % same_bank)
print('Vector similarity for *different* meanings:  %.2f' % diff_bank_1)
print('Vector similarity for *different* meanings:  %.2f' % diff_bank_2)
```

test_elmo_word_vectors.py hosted with ❤ by GitHub view raw



```
Vector similarity for  *similar*  meanings:  0.98
Vector similarity for *different* meanings:  0.87
Vector similarity for *different* meanings:  0.86
```

Sentence Embeddings

So far we have discussed how word embeddings represent the meaning of the words in a text document. But sometimes we need to go a step further and encode the meaning of the whole sentence to readily understand the context in which the words are used. This sentence representation is important for many downstream tasks. It enables us to understand the meaning of the sentence without calculating the individual embeddings of the words. It also allows us to make comparisons on the sentence level.

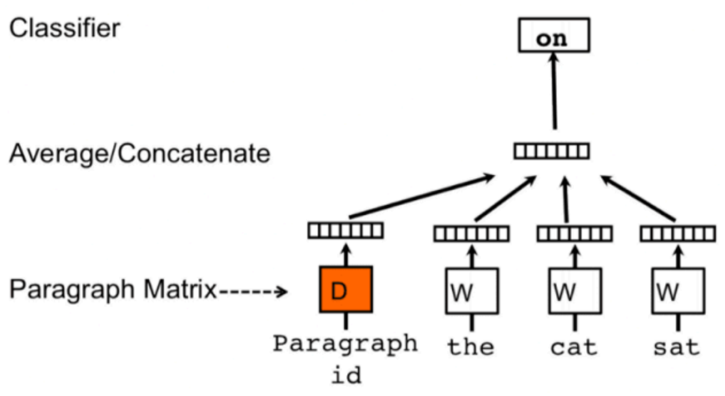
Using simple mathematical manipulation, it is possible to adapt sentence embeddings for tasks such as semantic search, clustering, intent detection, paraphrase detection. In addition to that, cross-lingual sentence embedding models can be used for parallel text mining or translation pair detection. For example, the TAUS Data Marketplace uses a data cleaning technique that uses sentence embeddings to compute the semantic similarity between parallel segments of text in different languages to assess translation quality.

A straightforward approach for creating sentence embeddings is to use a word embedding model to encode all words of the given sentence and take the average of all the related to word order and other aspects of overall sentence semantics.

Doc2Vec

The Doc2Vec model (or Paragraph Vector) extends the idea of the Word2Vec algorithm. The algorithm follows the assumption that a word's meaning is given by the words that appear close by. Similar to Word2Vec, Doc2Vec has two variants.

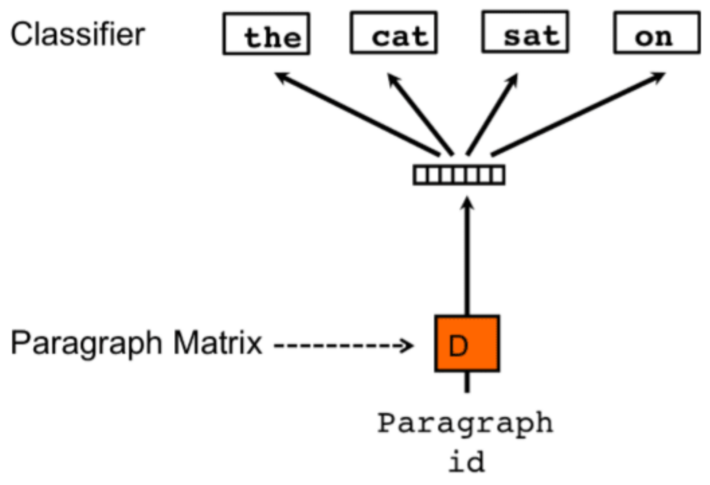
Distributed Memory Model(DM)



Each word and sentence of the training corpus are one-hot encoded and stored in matrices D and W, respectively. The training process involves passing a sliding window over the sentence, trying to predict the next word based on the previous words and the sentence vector (or Paragraph Matrix in the figure above). This prediction of the next word is done by concatenating the sentence and word vectors and passing the result into a softmax layer. The sentence vectors change with sentences, while the word vectors remain the same. Both are updated during training.

The inference process also involves the same sliding window approach. The difference is that all the vectors of the models are fixed except the sentence vector. After all the predictions of the vector.

Distributed Bag-Of-Words (DBOW) Model



The DBOW model ignores the word order and has a simpler architecture. Each sentence in the training corpus is converted into a one-hot representation. During training, a random sentence is selected from the corpus, and from the sentence, a random number of words. The model tries to predict these words using only the sentence ID, and the sentence vector is updated(Paragraph ID and Paragraph Matrix in the figure).

During inference, a new sentence ID is trained with random words from the sentence. The sentence vector is updated in each step, and the resulting sentence vector is the embedding for that sentence.

What Variant To Use?

The DM model takes into account the word order, the DBOW model doesn't. Also, the DBOW model doesn't use word vectors so the semantics of the words are not preserved. As a result, it's harder for it to detect similarities between words. And because of its simpler architecture, the DBOW model requires more training to obtain accurate an embedding, which is higher than with the DBOW model.

What approach produces better Sentence Embeddings? In the original paper, the authors state that the DM model is "consistently better than" DBOW. However, later studies showed that the DBOW approach is better for most tasks. For that reason, the implementation in the **Gensim** of Doc2Vec uses the DBOW approach as the default algorithm.

Implementation

Install Gensim, get the "**text8**" dataset to train the Doc2Vec model.

```python
import gensim
import gensim.downloader as api
dataset = api.load("text8")
data = [i for i in dataset]
```

get_text8.py hosted with ❤ by GitHub view raw

Tag the text data, then use it to build the model vocabulary and train the model.

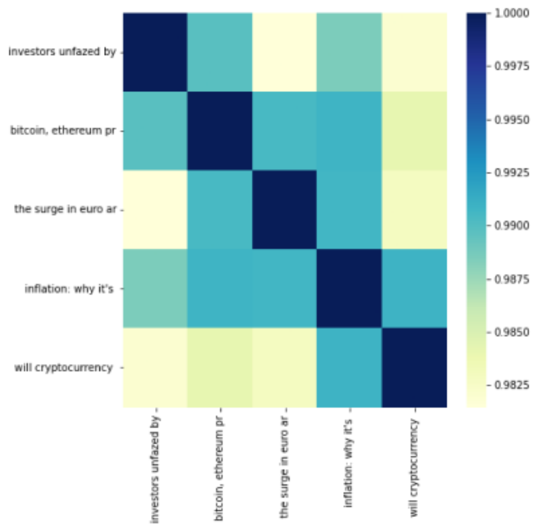
```python
def tagged_document(list_of_list_of_words):
   for i, list_of_words in enumerate(list_of_list_of_words):
      yield gensim.models.doc2vec.TaggedDocument(list_of_words, [i])

training_data = list(tagged_document(data))
model = gensim.models.doc2vec.Doc2Vec(vector_size=40, min_count=2, epochs=3

model.build_vocab(training_data)
model.train(training_data, total_examples=model.corpus_count, epochs=model.
```

build_train_doc2vec.py hosted with ❤ by GitHub view raw

Use the model to get the sentence embeddings of the headlines and calculate the cosine similarity between them.

```python
        row = []
        for j in range(len(sentences)):
            row.append(cosine_similarity(vectors[i],vectors[j])[0][0])
        similarity.append(row)

create_heatmap(similarity)
```

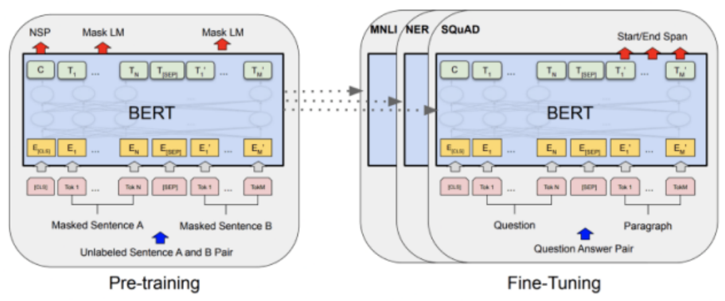
test_doc2vec_vectors.py hosted with ❤ by GitHub view raw



Sentence Transformers

Much like ELMo, there are a few bidirectional LSTM based sentence encoders (InferSent, etc) but LSTMs have certain problems. Firstly, they use a hidden vector to represent the memory at the current state of the input. But for long input sequences such as sentences, a single vector isn't enough to provide all the information needed to predict the next state correctly. This bottleneck of the size of the hidden vector makes LSTM based methods more susceptible to mistakes, as in practice it can only hold information from a limited number of steps back. The Attention mechanism in transformers doesn't have this bottleneck issue as it has access to all the previous hidden states for making predictions.

Another problem with LSTMs is the time to train. As the output is always dependent on the previous input, the training is done sequentially. This makes parallelization harder and results in mechanism in a neural network allowing for lower training time.



Transformer-based general language understanding models perform much better than all their predecessors. When BERT was introduced, it achieved state-of-the-art results in a wide range of tasks such as question answering or language inference with minor tweaks in its architecture. That being said, it has a massive computational overhead. For example, finding the most similar pair of sentences in a collection of 10,000 requires about 50 million inference computations (~65 hours). The structure of BERT makes it unsuitable for semantic similarity search as well as for unsupervised tasks like clustering.

Sentence-BERT (SBERT) is a modified BERT network that uses siamese and triplet network structures to derive semantically meaningful sentence embeddings. This reduces the effort for finding the most similar pair from 65 hours with BERT / RoBERTa to about 5 seconds with SBERT, while maintaining the accuracy from BERT.

Implementation

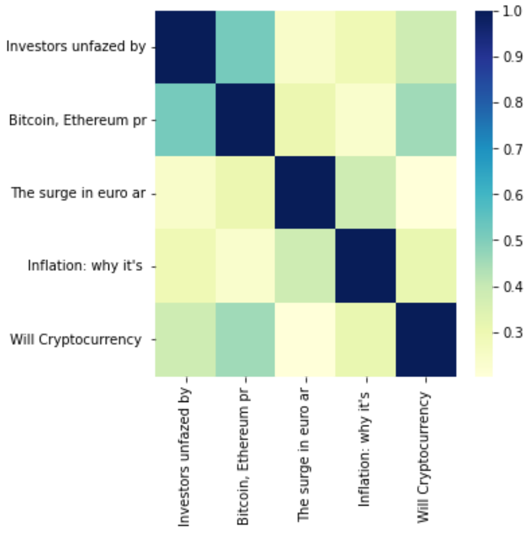
We'll try out the RoBERTa based models implemented in the **sentence-transformer** module.

Download the '**stsb-roberta-large**' model.

Use it to calculate the headline embeddings and their pairwise similarity.

```python
embeddings = model.encode(sentences, convert_to_tensor=True)

  similarity = []
  for i in range(len(sentences)):
    row = []
    for j in range(len(sentences)):
      row.append(util.pytorch_cos_sim(embeddings[i], embeddings[j]).item()
    similarity.append(row)

create_heatmap(similarity)
```

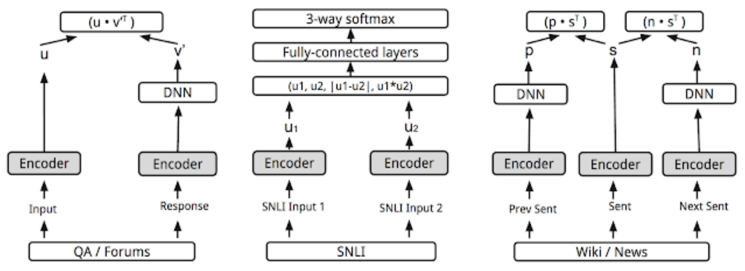
test_sent_transformer_vectors.py hosted with ❤ by GitHub view raw



Universal Sentence Encoder

Google's [Universal Sentence Encoder(USE)](#) leverages a one-to-many multi-tasking learning framework to learn a universal sentence embedding by switching between several tasks. The 6 tasks, skip-thoughts prediction of the next/previous sentence, neural machine translation, constituency parsing, and natural while preserving the performance on a variety of transfer tasks.

One version of the Universal Sentence Encoder model uses a deep average network (DAN) encoder, while another version uses a Transformer.



The more complicated Transformer architecture performs better than the simpler DAN model on a variety of sentiment and similarity classification tasks. The compute time for the Transformer version increases noticeably as sentence length increases. On the other hand, the compute time for the DAN model stays nearly constant as sentence length increases.

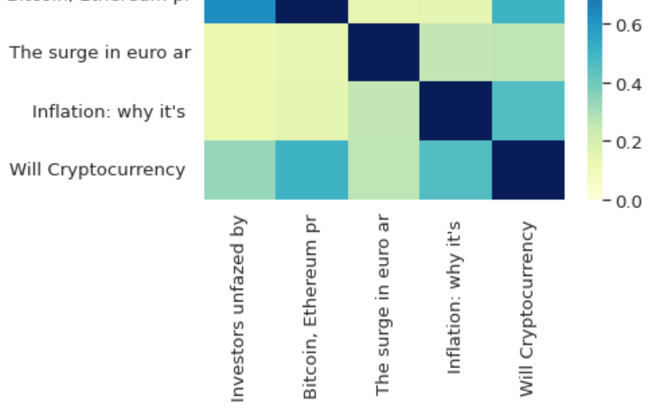
Implementation

```python
import tensorflow as tf
import tensorflow_hub as hub
module_url = "https://tfhub.dev/google/universal-sentence-encoder/4"
model = hub.load(module_url)
```

get_USE_model.py hosted with ❤ by GitHub view raw

```python
embeddings = model(text)
similarity = cosine_similarity(embeddings)
create_heatmap(similarity)
```

test_USE_vectors.py hosted with ❤ by GitHub view raw



What Embedding To Use?

As a general rule of thumb, traditional embeddings techniques like Word2Vec and Doc2Vec offer good results when the task only requires the global meaning of the text. This is reflected in their outperformance of state-of-the-art deep learning techniques on tasks such as semantic text similarity or paraphrase detection. On the other hand, when the task needs somethings more specific than just the global meaning, take for example sentiment analysis or sequence labeling, more complex contextual methods perform better.

So always start with a simple and fast method as the baseline before moving on to more complex methods if required.

Read more related blogs

Ready for Custom News Solutions?

Drop your email and find out how our API delivers precisely what your business needs.

Enter your email

Get Started