Get started Open in app

505K Followers · About Follow

HANDS-ON TUTORIALS

How to Build a Semantic Search Engine With Transformers and Faiss

Kostas Stathoulopoulos Nov 9 · 8 min read

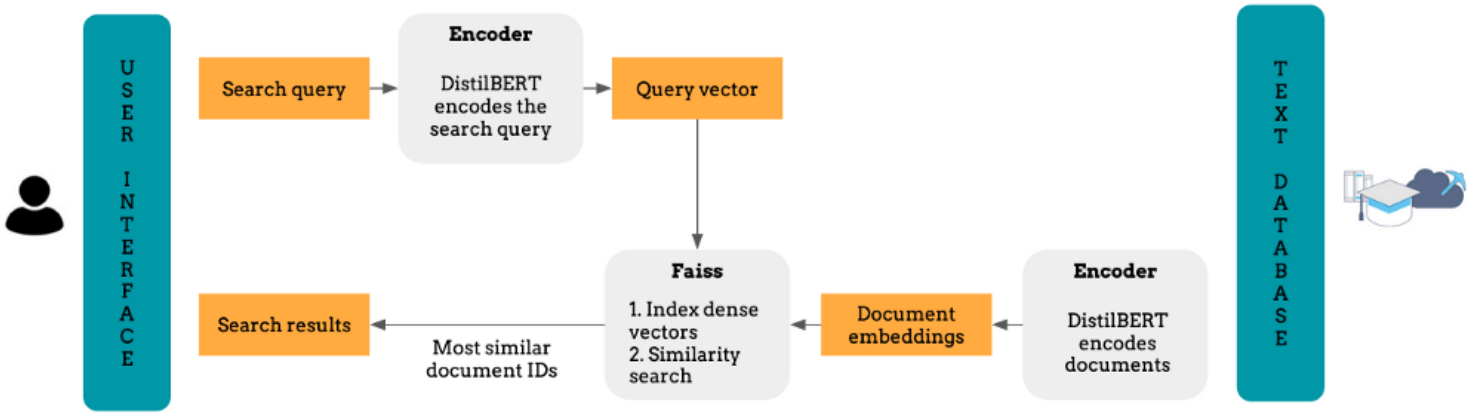

Image by Kostas Stathoulopoulos

Introduction

Have you ever wondered how you can create state-of-the-art sentence embeddings with Transformers and use them in downstream tasks like semantic textual similarity?

In this tutorial, you will learn how to build a vector-based search engine with sentence transformers and Faiss. If you want to jump straight into the code, check out the GitHub

Get started Open in app

In the second part of the tutorial, you will learn how to serve the search engine in a Streamlit application deploy it with Docker and AWS Elastic Beanstalk.

Why build a vector-based search engine?

Keyword-based search engines are easy to use and work well in most scenarios. You ask for *machine learning* papers and they return a bunch of results containing an exact match or a close variation of the query like *machine-learning*. Some of them might even return results containing synonyms of the query or words that appear in a similar context. Others, like Elasticsearch, do all of these things — and even more — in a rapid and scalable way. However, keyword-based search engines usually struggle with:

- Complex queries or with words that have a dual meaning.
- Long queries such as a paper abstract or a paragraph from a blog.
- Users who are unfamiliar with a field's jargon or users who would like to do exploratory search.

Vector-based (also called *semantic*) search engines tackle those pitfalls by finding a numerical representation of text queries using state-of-the-art language models, indexing them in a high-dimensional vector space and measuring how similar a query vector is to the indexed documents.

Indexing, vectorisation and ranking methods

Before diving into the tutorial, I will briefly explain how keyword-based and vector-based search engines (1) **index documents** (ie store them in an easily retrievable form), (2) **vectorise text data** and (3) **measure how relevant a document is to a query**. This will help us highlight the differences between the two systems and understand why vector-based search engines might give more meaningful results for long-text queries.

1. Keyword-based search engines

Let's use an oversimplified version of Elasticsearch as an example. Elasticsearch uses a tokeniser to split a document into tokens (ie meaningful textual units) which are mapped to numerical sequences and used to build an inverted index.

Get started Open in app

basil	Grandma's tomato soup
leaves	African tomato soup, Grandma's tomato soup
salt	African tomato soup, Good ol' tomato soup
tomato	African tomato soup, Good ol' tomato soup, Grandma's tomato soup
water	African tomato soup, Good ol' tomato soup, Grandma's tomato soup
...	...

Inverted index: Instead of going through each document to check if it contains a query term, the inverted index enables us to look for a term once and retrieve a list of all documents containing the term. Image by author: Morten Ingebrigtsen

In parallel, Elasticsearch represents every indexed document with a high-dimensional, weighted vector, where each distinct index term is a dimension, and their value (or weight) is calculated with TF-IDF.

To find relevant documents and rank them, Elasticsearch combines a Boolean Model (BM) with a Vector Space Model (VSM). BM marks which documents contain a user's query and VSM scores how relevant they are. During search, the query is transformed to vector using the same TF-IDF pipeline and then the VSM score of document `d` for query `q` is the cosine similarity of the weighted query vectors `V(q)` and `V(d)`.

> *This way of measuring similarity is very simplistic and not scalable. The workhorse behind Elasticsearch is Lucene which employs various tricks, from boosting fields to changing how vectors are normalised, to speed up the search and improve its quality.*

Elasticsearch works great in most cases, however, we would like to create a system that pays attention to the words' context too. This brings us to vector-based search engines.

2. Vector-based search engines

Get started Open in app

index.

Creating dense document vectors

In recent years, the NLP community has made strides on that front, with many deep learning models being open-sourced and distributed by packages like Huggingface's Transformers that provides access to state-of-the-art, pretrained models. Using pretrained models has many advantages:

- They usually produce high-quality embeddings as they were trained on large amounts of text data.
- They don't require you to create a custom tokeniser as Transformers come with their own methods.
- It's straightforward to fine-tune a model to your task.

These models produce a fixed-size vector for each **token** in a document. How do we get document-level vectors though? This is usually done by averaging or pooling the word vectors. However, these approaches produce below-average sentence and document embeddings, usually worse than averaging GloVe vectors.

To build our semantic search engine we will use Sentence Transformers that fine-tune BERT-based models to produce semantically meaningful embeddings of long-text sequences.

Building an index and measuring relevance

The most naive way to retrieve relevant documents would be to measure the cosine similarity between the query vector and every document vector in our database and return those with the highest score. Unfortunately, this is very slow in practice.

The preferred approach is to use **Faiss**, a library for efficient similarity search and clustering of dense vectors. Faiss offers a large collection of indexes and composite indexes. Moreover, given a GPU, Faiss scales up to billions of vectors!

Get started Open in app

Tutorial: Building a vector-based search engine with Sentence Transformers and Faiss

In this practical example, we will work with real-world data. I created a dataset of 8,430 academic articles on misinformation, disinformation and fake news published between 2010 and 2020 by querying the Microsoft Academic Graph with Orion.

I retrieved the papers' abstract, title, citations, publication year and ID. I did minimal data cleaning like removing papers without an abstract. Here is how the data look like:

```
df.head(3)
```

	original_title	abstract	year	citations	id
0	When Corrections Fail: The Persistence of Poli...	An extensive literature addresses citizen igno...	2010	901	2132553681
1	A postmodern Pandora's box: anti-vaccination m...	The Internet plays a large role in disseminati...	2010	440	2117485795
2	Spread of (Mis)Information in Social Networks	We provide a model to investigate the tension ...	2010	278	2120015072

Image by Kostas Stathoulopoulos

Importing Python packages and reading the data from S3

Let's import the required packages and read the data. The file is public so you can run the code on Google Colab or locally by accessing the GitHub repo!

```
# Used to import data from S3.
import pandas as pd
import s3fs

# Used to create the dense document vectors.
import torch
from sentence_transformers import SentenceTransformer

# Used to create and store the Faiss index.
import faiss
import numpy as np
import pickle

# Used to do vector searches and display the results.
from vector_engine.utils import vector_search, id2details

```

Get started Open in app

Vectorising documents with Sentence Transformers

Next, let's encode the paper abstracts. Sentence Transformers offers a number of pretrained models some of which can be found in this spreadsheet. Here, we will use the `distilbert-base-nli-stsb-mean-tokens` model which performs great in Semantic Textual Similarity tasks and it's quite faster than BERT as it is considerably smaller.

Here, we will:

1. Instantiate the transformer by passing the model name as a string.
2. Switch to a GPU if it is available.
3. Use the `.encode()` method to vectorise all the paper abstracts.

```
# Instantiate the sentence-level DistilBERT
model = SentenceTransformer('distilbert-base-nli-stsb-mean-tokens')

# Check if CUDA is available ans switch to GPU
if torch.cuda.is_available():
    model = model.to(torch.device("cuda"))
print(model.device)

# Convert abstracts to vectors
embeddings = model.encode(df.abstract.to_list(), show_progress_bar=True)
```

medium_001_embeddings.py hosted with ♡ by GitHub view raw

> *It is recommended to use a GPU when vectorising documents with Transformers. Google Colab offers one for free!* ***If you want to run it on AWS, check out my guide on how to launch a GPU instance on AWS for machine learning.***

Indexing documents with Faiss

Faiss contains algorithms that search in sets of vectors of any size, even ones that do not fit in RAM. To learn more about Faiss, you can read their paper on arXiv or their wiki.

Get started Open in app

a few 10s to 100s.

> *Faiss uses only 32-bit floating point matrices. This means we will have to change the data type of the input before building the index.*

Here, we will use the `IndexFlatL2` index that performs a brute-force L2 distance search. It works well with our dataset, however, it can be very slow with a large dataset as it scales linearly with the number of indexed vectors. Faiss offers fast indexes too!

To create an index with the abstract vectors, we will:

1. Change the data type of the abstract vectors to `float32`.
2. Build an index and pass it the dimension of the vectors it will operate on.
3. Pass the index to `IndexIDMap`, an object that enables us to provide a custom list of IDs for the indexed vectors.
4. Add the abstract vectors and their ID mapping to the index. In our case, we will map vectors to their paper IDs from Microsoft Academic Graph.

To test the index works as expected, we can query it with an indexed vector and retrieve its most similar documents as well as their distance. The first result should be our query!

```
# Step 1: Change data type
embeddings = np.array([embedding for embedding in embeddings]).astype("float32")

# Step 2: Instantiate the index
index = faiss.IndexFlatL2(embeddings.shape[1])

# Step 3: Pass the index to IndexIDMap
index = faiss.IndexIDMap(index)

# Step 4: Add vectors and their IDs
index.add_with_ids(embeddings, df.id.values)

# Retrieve the 10 nearest neighbours
D, I = index.search(np.array([embeddings[5415]]), k=10)
```

Get started Open in app

```
# Paper abstract
df.iloc[5415, 1]
```

```
"We address the diffusion of information about the COVID-19 with a massive data analysis on Twitter, Instagram, YouTube, Reddit and Gab. We analyze engagement and interest in the COVID-19 topic and provide a differential assessment on the evolution of the discourse on a global scale for each platform and their users. We fit information spreading with epidemic models characterizing the basic reproduction number [Formula: see text] for each social media platform. Moreover, we identify information spreading from questionable sources, finding different volumes of misinformation in each platform. However, information from both reliable and questionable sources do not present different spreading patterns. Finally, we provide platform-dependent numerical estimates of rumors' amplification."
```

```
# Retrieve the 10 nearest neighbours
D, I = index.search(np.array([embeddings[5415]]), k=10)
print(f'L2 distance: {D.flatten().tolist()}\n\nMAG paper IDs: {I.flatten().tolist()}')
```

```
L2 distance: [0.0, 1.267284631729126, 62.72160339355469, 63.670326232910156, 64.58393859863281, 67.47344970703125, 67.96402740478516, 69.47564697265625, 72.563354492187
5, 74.62230682373047]

MAG paper IDs: [3092618151, 3011345566, 3012936764, 3055557295, 3011186656, 3044429417, 3092128270, 3024620668, 3047284882, 3048848247]
```

```
# Fetch the paper titles based on their index
id2details(df, I, 'original_title')
```

```
[['The COVID-19 social media infodemic.'],
 ['The COVID-19 Social Media Infodemic'],
 ['Understanding the perception of COVID-19 policies by mining a multilanguage Twitter dataset'],
 ['Covid-19 infodemic reveals new tipping point epidemiology and a revised R formula.'],
 ['Coronavirus Goes Viral: Quantifying the COVID-19 Misinformation Epidemic on Twitter'],
 ['Effects of misinformation on COVID-19 individual responses and recommendations for resilience of disastrous consequences of misinformation'],
 ['Analysis of online misinformation during the peak of the COVID-19 pandemics in Italy'],
 ['Quantifying COVID-19 Content in the Online Health Opinion War Using Machine Learning'],
 ['Global Infodemiology of COVID-19: Analysis of Google Web Searches and Instagram Hashtags.'],
 ['COVID-19-Related Infodemic and Its Impact on Public Health: A Global Social Media Analysis.']]
```

Since we queried Faiss with an indexed vector, the first result must be the query and the distance must be equal to zero! Image by Kostas Stathoulopoulos

Searching with user queries

Let's try to find relevant academic articles for a new, unseen search query. In this example, I will query our index with the first paragraph of the *Can WhatsApp benefit from debunked fact-checked stories to reduce misinformation?* article that was published on HKS Misinformation Review.

```
user_query = """
WhatsApp was alleged to have been widely used to spread misinformation and propaganda
during the 2018 elections in Brazil and the 2019 elections in India. Due to the
private encrypted nature of the messages on WhatsApp, it is hard to track the dissemination
of misinformation at scale. In this work, using public WhatsApp data from Brazil and India, we
observe that misinformation has been largely shared on WhatsApp public groups even after they
were already fact-checked by popular fact-checking agencies. This represents a significant portion
of misinformation spread in both Brazil and India in the groups analyzed. We posit that such
misinformation content could be prevented if WhatsApp had a means to flag already fact-checked
content. To this end, we propose an architecture that could be implemented by WhatsApp to counter
such misinformation. Our proposal respects the current end-to-end encryption architecture on WhatsApp,
thus protecting users' privacy while providing an approach to detect the misinformation that benefits
from fact-checking efforts.
"""
```

Image by Kostas Stathoulopoulos

To retrieve academic articles for a new query, we would have to:

Get started Open in app

2. Change its data type to `float32`.

3. Search the index with the encoded query.

For convenience, I have wrapped up these steps in the `vector_search()` function.

```python
import numpy as np

def vector_search(query, model, index, num_results=10):
    """Tranforms query to vector using a pretrained, sentence-level
    DistilBERT model and finds similar vectors using FAISS.

    Args:
        query (str): User query that should be more than a sentence long.
        model (sentence_transformers.SentenceTransformer.SentenceTransformer)
        index (`numpy.ndarray`): FAISS index that needs to be deserialized.
        num_results (int): Number of results to return.

    Returns:
        D (:obj:`numpy.array` of `float`): Distance between results and query.
        I (:obj:`numpy.array` of `int`): Paper ID of the results.

    """
    vector = model.encode(list(query))
    D, I = index.search(np.array(vector).astype("float32"), k=num_results)
    return D, I


def id2details(df, I, column):
    """Returns the paper titles based on the paper index."""
    return [list(df[df.id == idx][column]) for idx in I[0]]

# Querying the index
D, I = vector_search([user_query], model, index, num_results=10)
```

medium_001_search.py hosted with ♡ by GitHub view raw

The article discusses misinformation, fact-checking, WhatsApp and elections in Brazil and India. We would expect our vector-based search engine to return results on these

Get started Open in app

```
# Fetching the paper titles based on their index
id2details(df, I, 'original_title')
```

```
[['Can WhatsApp Benefit from Debunked Fact-Checked Stories to Reduce Misinformation?'],
 ['A Dataset of Fact-Checked Images Shared on WhatsApp During the Brazilian and Indian Elections'],
 ['A Dataset of Fact-Checked Images Shared on WhatsApp During the Brazilian and Indian Elections'],
 ['A System for Monitoring Public Political Groups in WhatsApp'],
 ['Politics of Fake News: How WhatsApp Became a Potent Propaganda Tool in India'],
 ['Characterizing Attention Cascades in WhatsApp Groups'],
 ['OS IMPACTOS JURÍDICOS E SOCIAIS DAS FAKE NEWS EM TERRITÓRIO BRASILEIRO'],
 ['Fake News and Social Media: Indian Perspective'],
 ['Images and Misinformation in Political Groups: Evidence from WhatsApp in India'],
 ['Can WhatsApp Counter Misinformation by Limiting Message Forwarding']]
```

Image by Kostas Stathoulopoulos

Conclusion

In this tutorial, we built a vector-based search engine using Sentence Transformers and Faiss. Our index works well but it's fairly simple. We could improve the quality of the embeddings by using a domain-specific transformer like SciBERT which has been trained on papers from the corpus of semanticscholar.org. We could also remove duplicates before returning the results and experiment with other indexes.

For those working with Elasticsearch, Open Distro introduced an approximate k-NN similarity search feature which is also part of AWS Elasticsearch service. In another blog, I will dive into that too!

Finally, you can find the code on GitHub and try it out with Google Colab.

References

[1] Thakur, N., Reimers, N., Daxenberger, J. and Gurevych, I., 2020. Augmented SBERT: Data Augmentation Method for Improving Bi-Encoders for Pairwise Sentence Scoring Tasks. *arXiv preprint arXiv:2010.08240*.

[2] Johnson, J., Douze, M. and Jégou, H., 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*.

Sign up for The Daily Pick