**Large Language Models and Vector Databases for News Recommendations**

*Making LLMs into production environments using Sentence Transformers and Qdrant*

[[João Felipe Guedes](https://medium.com/@guedes.joaofelipe?source=post_page-----6f9348fd4030--------------------------------)](https://medium.com/@guedes.joaofelipe?source=post_page-----6f9348fd4030--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----6f9348fd4030--------------------------------)](https://towardsdatascience.com/?source=post_page-----6f9348fd4030--------------------------------)

[João Felipe Guedes](https://medium.com/@guedes.joaofelipe?source=post_page-----6f9348fd4030--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F259f985d397f&operation=register&redirect=https%3A%2F%2Ftowardsdatascience.com%2Flarge-language-models-and-vector-databases-for-news-recommendations-6f9348fd4030&user=Jo%C3%A3o+Felipe+Guedes&userId=259f985d397f&source=post_page-259f985d397f----6f9348fd4030---------------------post_header-----------)

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----6f9348fd4030--------------------------------)

·

8 min read

·

4 days ago

191

2

Large language models (LLMs) generated a global buzz in the machine learning community with recent releases of generative AI tools such as Chat-GPT, Bard, and others alike. One of the core ideas behind these solutions is to compute a numerical representation of unstructured data (such as texts and images) and find similarities between these representations.

However, taking all of these concepts into a production environment has its own set of machine learning engineering challenges:

* How to generate these representations quickly?
* How to store them in a proper database?
* How to quickly compute similarities for production environments?

In this article, I introduce two open-source solutions that aim to solve these questions:

* [**Sentence Transformers**](https://www.sbert.net/)**[1]**: an embedding generation technique based on textual information and;
* [**Qdrant**](https://qdrant.tech/): a vector database capable of storing embeddings and providing an easy interface to query them.

These tools are applied to NPR [2], a News Portal Recommendation dataset (openly available at [Kaggle](https://www.kaggle.com/datasets/joelpl/news-portal-recommendations-npr-by-globo)) which was built to support the academic community to develop recommendation algorithms. By the end of the articles, you'll see how to:

* Generate news embeddings with Sentence Transformers
* Store embeddings with Qdrant
* Query embeddings to recommend news articles

All code for this article is made available on [Github](https://github.com/guedes-joaofelipe/vector-search-api).

**1. Generating embeddings with Sentence Transformers**

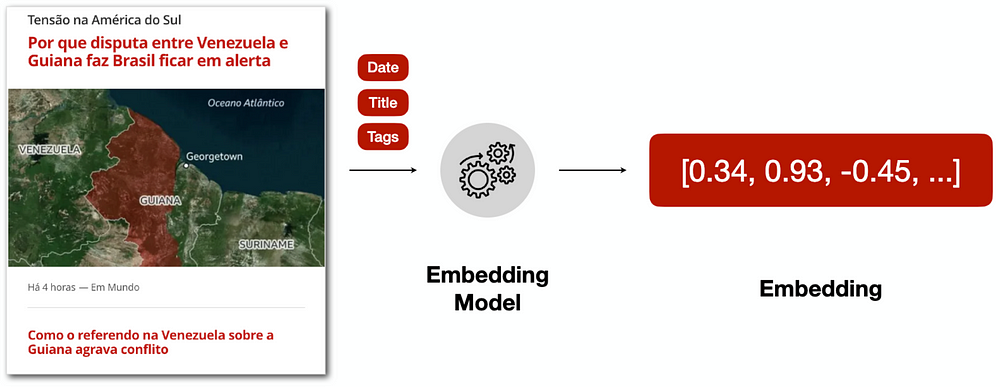
First of all, we need to find a way to convert input data into vectors, which we'll call embeddings (if you want to dig deeper into the embedding concept, I recommend Boykis' article [What Are Embeddings?](https://vickiboykis.com/what_are_embeddings/about.html) [3]).

So let's take a look at what kind of data we can work on with the NPR dataset:

import pandas as pd  
  
df = pd.read\_parquet("articles.parquet")  
df.tail()

**FAILED**

We can see that NPR has some interesting textual data such as the articles' *title* and *body* content. We can use them in an embedding-generation process as the following image:



Embedding generation process (image by author)

So once we define the textual features from our input data, we need to establish an embedding model to generate our numerical representation. Lucky for us, there are websites like [HuggingFace](https://huggingface.co/) where you can look for pre-trained models suitable for specific languages or tasks. In our example, we can use the [*neuralmind/bert-base-portuguese-cased*](https://huggingface.co/neuralmind/bert-base-portuguese-cased) model, which was trained in Brazilian portuguese for the following tasks:

* Named Entity Recognition
* **Sentence Textual Similarity**
* Recognizing Textual Entailment

Code-wise, this is how we translate the embedding-generation process:

from sentence\_transformers import SentenceTransformer  
  
model\_name = "neuralmind/bert-base-portuguese-cased"  
encoder = SentenceTransformer(model\_name\_or\_path=model\_name)  
  
title = """  
 Paraguaios vão às urnas neste domingo (30) para escolher novo presidente  
"""  
  
sentence = title  
  
sentence\_embedding = encoder.encode(sentence)  
print (sentence\_embedding)  
# output: np.array([-0.2875876, 0.0356041, 0.31462672, 0.06252239, ...])

So given an example input data, we can concatenate the *title* and *tags* content into a single text and pass it to an encoder to generate the text embedding.

We can apply the same process for all other articles in the NPR dataset:

def generate\_item\_sentence(item: pd.Series, text\_columns=["title"]) -> str:  
 return ' '.join([item[column] for column in text\_columns])  
  
df["sentence"] = df.apply(generate\_item\_sentence, axis=1)  
df["sentence\_embedding"] = df["sentence"].apply(encoder.encode)

***Note****: bear in mind that this process might take a bit longer depending on your machine's processing power.*

Once we have the embeddings for all news articles, let's define a strategy to store them.

**2. Storing embeddings**

Since generating embeddings might be an expensive process, we can use a vector database to store these embeddings and execute queries based on diverse strategies.

There are several vector database software to achieve this task, but I'll use **Qdrant** for this article, which is an open-source solution with APIs available for popular programming languages like *Python*, *Go,* and *Typescript*. For a better comparison between these vector databases, check this [article](https://www.datacamp.com/blog/the-top-5-vector-databases) [4].

**Setting Qdrant**

To deal with all Qdrant operations, we need to create a client object that points out to a vector database. Qdrant lets you create a free tier service to test remote connection to a database but, for the sake of simplicity, I'll create and persist the database locally:

from qdrant\_client import QdrantClient  
client = QdrantClient(path="./qdrant\_data")

Once this connection is stablished, we can create a collection in the database that will store the news articles embeddings:

from qdrant\_client import models  
from qdrant\_client.http.models import Distance, VectorParams  
  
client.create\_collection(  
 collection\_name = "news-articles",  
 vectors\_config = models.VectorParams(  
 size = encoder.get\_sentence\_embedding\_dimension(),  
 distance = models.Distance.COSINE,  
 ),  
)  
  
print (client.get\_collections())  
# output: CollectionsResponse(collections=[CollectionDescription(name='news-articles')])

Notice that vector configuration parameters are used to create the collection. These parameters tell Qdrant some properties from the vectors, like their *size* and the *distance* metric to be used when comparing vectors (I'll use the cosine similarity but you can also use other strategies like the [inner product or Euclidean distance](https://qdrant.tech/documentation/concepts/search/#metrics)).

**Generating Vectors Points**

Prior to finally populating the database, we need to create proper objects to be uploaded. In Qdrant, vectors can be stored using a [*PointStruct*](https://qdrant.tech/documentation/concepts/points/) class, which you can use to define the following properties:

* ***id***: the vector's ID (in the NPR case, is the *newsId*)
* ***vector***: a 1-dimensional array representing the vector (generated by the embedding model)
* ***payload***: a dictionary containing any other relevant metadata that can later be used to query vectors in a collection (in the NPR case, the article's *title*, *body,* and *tags*)

from qdrant\_client.http.models import PointStruct  
  
metadata\_columns = df.drop(["newsId", "sentence", "sentence\_embedding"], axis=1).columns  
  
def create\_vector\_point(item:pd.Series) -> PointStruct:  
 """Turn vectors into PointStruct"""  
 return PointStruct(  
 id = item["newsId"],  
 vector = item["sentence\_embedding"].tolist(),  
 payload = {  
 field: item[field]  
 for field in metadata\_columns  
 if (str(item[field]) not in ['None', 'nan'])  
 }  
 )  
  
points = df.apply(create\_vector\_point, axis=1).tolist()

**Uploading Vectors**

Finally, after all items are turned into point structures, we can upload them in chunks to the database:

CHUNK\_SIZE = 500  
n\_chunks = np.ceil(len(points)/CHUNK\_SIZE)  
  
for i, points\_chunk in enumerate(np.array\_split(points, n\_chunks)):  
 client.upsert(  
 collection\_name="news-articles",  
 wait=True,  
 points=points\_chunk.tolist()  
 )

**3. Querying Vectors**

Now that collections are finally populated with vectors, we can start querying the database. There are many ways we can input information to query the database, but I think there are 2 very useful inputs we can use:

* An input text
* An input vector ID

**3.1 Querying vectors with an input vector**

Let's say we built this vector database to be used in a search engine. In this case, we expect the user's input to be an input text and we have to return the most relevant items.

Since all operations in a vector database are done with….VECTORS, we first need to transform the user's input text into a vector so we can find similar items based on that input. Recall that we used Sentence Transformers to encode textual data into embeddings, so we can use the very same encoder to generate a numerical representation for the user's input text.

Since the NPR contains news articles, let's say the user typed *"Donald Trump"* to learn about US elections:

query\_text = "Donald Trump"  
query\_vector = encoder.encode(query\_text).tolist()  
print (query\_vector)  
# output: [-0.048, -0.120, 0.695, ...]

Once the input query vector is computed, we can search for the closest vectors in the collection and define what sort of output we want from those vectors, like their *newsId*, *title,* and *topics:*

from qdrant\_client.models import Filter  
from qdrant\_client.http import models  
  
client.search(  
 collection\_name="news-articles",  
 query\_vector=query\_vector,  
 with\_payload=["newsId", "title", "topics"],  
 query\_filter=None  
)

***Note****: by default, Qdrant uses Approximate Nearest Neighbors to scan for embeddings quickly, but*[*you can also do a full scan and bring the exact nearest neighbors*](https://qdrant.tech/documentation/concepts/search/#search-api)*— just bear in mind this is a much more expensive operation.*

After running this operation, here are the generated output titles (translated into english for better comprehension):

* **Input Sentence**: *Donald Trump*
* **Output 1***:* *Paraguayans go to the polls this Sunday (30) to choose a new president*
* **Output 2*:*** *Voters say Biden and Trump should not run in 2024, Reuters/Ipsos poll shows*
* **Output 3*:*** *Writer accuses Trump of sexually abusing her in the 1990s*
* **Output 4*:*** *Mike Pence, former vice president of Donald Trump, gives testimony in court that could complicate the former president*

It seems that besides bringing news related to Trump himself, the embedding model also managed to represent topics related to presidential elections. Notice that in the first output, there is no direct reference to the input term “*Donald Trump*” other than the presidential election.

Also, I left out a *query\_filter* parameters. This is a very useful tool if you want to specify that the output must satisfy some given condition. For instance, in a news portal, it is frequently important to filter only the most recent articles (say from the past 7 days onwards). Therefore, you could query for news articles that satisfy a minimum publication timestamp.

***Note:****in the news recommendation context, there are multiple concerning aspects to consider like fairness and diversity. This is an open topic of discussion but, should you be interested in this area, take a look at the articles from the*[*NORMalize Workshop*](https://sites.google.com/view/normalizeworkshop/home)*.*

**3.2 Querying vectors with an input vector ID**

Lastly, we can ask the vector database to "recommend" items that are closer to some desired vector IDs but far from undesired vector IDs. The desired and undesired IDs are called *positive* and *negative* examples, respectively, and they are thought of as seeds for the recommendation.

For instance, let's say we have the following positive ID:

seed\_id = '8bc22460-532c-449b-ad71-28dd86790ca2'  
# title (translated): 'Learn why Joe Biden launched his bid for re-election this Tuesday'

We can then ask for items similar to this example:

client.recommend(  
 collection\_name="news-articles",  
 positive=[seed\_id],  
 negative=None,  
 with\_payload=["newsId", "title", "topics"]  
)

After running this operation, here are the translated output titles :

* **Input item:** *Learn why Joe Biden launched his bid for re-election this Tuesday*
* **Output 1:** *Biden announces he will run for re-election*
* **Output 2:** *USA: the 4 reasons that led Biden to run for re-election*
* **Output 3:***Voters say Biden and Trump should not run in 2024, Reuters/Ipsos poll shows*
* **Output 4:** *Biden’s advisor’s gaffe that raised doubts about a possible second government after the election*

**Conclusion**

This article demonstrates how to combine LLMs and vector databases to serve recommendations. In particular, Sentence Transformers were used to generate numerical representations (embeddings) from textual news articles in the NPR dataset. Once these embeddings are computed, they can populate a vector database such as Qdrant which facilitates querying vectors based on several strategies.

A whole lot of improvements can be made after the examples in this article, such as:

* testing other embedding models
* testing other distance metrics
* testing other vector databases
* using compile-based programming languages like Go for better performance
* creating an API to serve recommendations

In other words, many ideas may come up to improve the machine learning engineering of recommendations with LLMs. So, if you feel like sharing your ideas about these improvements, don't hesitate to send me a message [here](https://www.linkedin.com/in/joao-felipe-guedes/) :)

**About Me**

I am a senior data scientist at [Globo](https://www.globo.com/), a Brazilian media-tech company. Working at the company's recommendation team, I am surrounded by an amazing and talented team who put a lot of effort to deliver personalized content to millions of users through digital products like [G1](https://g1.globo.com/), [GE](https://ge.globo.com/), [Globoplay](https://globoplay.globo.com/), and many others. This article wouldn't be possible without their indispensable knowledge.

**References**

[1] N. reimers and I. Gurevych, [Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks](http://arxiv.org/abs/1908.10084) (2019), Association for Computational Linguistics.

[2] J. Pinho, J. Silva and L. Figueiredo, [NPR: a News Portal Recommendations dataset](https://sites.google.com/view/normalizeworkshop/contributions?authuser=0) (2023), ACM Conference on Recommender Systems

[3] V. Boykis, [What are embeddings?](https://vickiboykis.com/what_are_embeddings/), personal blog

[4] M. Ali, [The Top 5 Vector Databases](https://www.datacamp.com/blog/the-top-5-vector-databases)(2023), DataCamp blog