Building a sentence embedding index with fastText and BM25

Exploring how to build a text search system

David Mezzetti Follow
Jun 16 · 8 min read ★

Photo by Jessica Ruscello on Unsplash

This article covers sentence embeddings and how codequestion built a fastText + BM25 embeddings search. Source code can be found on github.

Natural language processing (NLP) is one of the fastest growing areas in the field of machine learning. Deep innovation is happening on many fronts, leading to users being able to find better data faster. Techniques and algorithms once only realistic for those with massive IT budgets and resources are now able to run on a laptop. NLP is not a new problem though and there are time-tested methods available that perform very well. The simplest solution many times can be the best solution.

This article will explore various methods and go through an evaluation process for building a text search system.

Full-text search engines

Full-text search engines that allow users to enter a search query and find matching results are a reliable solution with a great performance history. In these systems, each document is tokenized usually with a list of common words removed referred to as stop words. Those tokens are stored in an inverted index and each token is weighed based on metrics like token frequency.

Search queries are also tokenized using the same method and the search engine finds the best matching record for the query tokens. There are highly-distributed and very fast solutions that have proved themselves in production for many years, like Elasticsearch. The most common token weighting algorithm today is BM25. It works very well and is still hard to beat.

Embeddings

Word embeddings have rapidly evolved over the last 5–7 years, first starting with Word2vec, followed by GloVe and fastText. These algorithms all typically build 300 dimension vectors. More advanced embeddings are now available that are larger in size (768 + dimensions) and able to understand the context of words within a sentence, with BERT embeddings leading the way. In other words, advancing past simple bag of word methods.

Easy to use, robust libraries are out there that enable developers to take advantage of these advancements. HuggingFace Transformers is an excellent library with a fast-growing set of cutting-edge functionality. Universal sentence embeddings is another important area of development. Having a single dynamic model that can transform text into embeddings for downstream learning tasks is crucial. Sentence Transformers is built on top of Transformers and is an excellent example of a library that can build high-quality sentence embeddings.

Performance considerations

As the march to build the perfect embeddings racers forward, what does that mean for practitioners trying to solve current business problems. Do you always use the latest and greatest advancements? Do you always needs to? There are trade offs with more complexity. When processing millions and billions of data points, processing time vs acceptable functionality is always a conversation that needs to happen. These conversations factor in resources available, time available and what is necessary to satisfy the requirements for a given problem.

Some questions to ask when evaluating a text search solution.

- How fast do I need query responses and indexing to be? How high of accuracy is required?
- What compute resources do I have access to? Do I have access to GPUs?
- How many records vs how much storage do I have? Assuming vectors are using 32 bit (4 byte) floats, a 300 dimensional vector takes 300x4 bytes per record. For a 4096 dimensional vector, it's 4096x4, which requires 13 times more space for the same data.
- Can a full-text search engine satisfy the requirements?

What are we building?

codequestion is an application that allows a user to ask coding questions directly from the terminal. Many developers will have a web browser window open while they develop and run web searches as questions arise. codequestion works to make that process faster so you can focus on development. It also allows users working without direct internet access or reduced internet access the ability to run code question queries. An example of the application in action is below.

codequestion application demo

codequestion is backed by a SQLite database that stores the repository of question-answer responses. A pre-trained model using Stack Exchange data is provided. This database can also be loaded with a custom list of questions.

Crafting a solution

It's always best to start with the simplest solution and work your way up. Can we put a full-text index on the data and call it a day? Adding an external text index wasn't desired as this application is designed to be installed locally on a developer machine. SQLite does provide a text indexing solution called FTS5. That was added easily enough and seemed to work pretty well. But a way to measure results is necessary to really judge how well it works.

Evaluating results

A dataset of 100 queries was constructed to judge how well an index is working. For each query, a web search was manually run and the top Stack Exchange result was stored for the query. Popular queries were selected (i.e. parsing a date in python) and care was taken to not always have direct token matches between the query and answer.

Mean Reciprocal Rank (MRR) is used to measure how the manually annotated top result scores for a search query. The standard FTS5 index scored the following against the 100 query dataset.

MRR = 45.9

Not too bad as a start. Time to increase the complexity to decide if the tradeoffs are worth it.

BM25

SQLite's FTS5 default token weighting scheme is to use tf-idf. It also has support for BM25 and that scored as:

MRR = 49.5

Definitely an improvement. But it still feels like a lot of improvement is available. Lets try sentence embeddings.

fastText sentence embeddings

Standard text token searches are getting caught up with synonyms and non-exact matches. Word embeddings are a great way to find similar results that don't match exactly.

An example of desired functionality is below:

How decode audio file query

The query "linux decode audio file" returned a similar question of "play mp3 or wav file via linux command line". The word decode and audio don't appear in the result returned but decode is similar to play and audio similar to mp3/wav.

The example query above is tokenized to ["linux", "decode", "audio", "file"]. Word embeddings can be retrieved for each token using fastText. To build a sentence embedding, the embeddings can be averaged together to create a single embedding vector.

fastText + BM25

Averaging works surprisingly well. But what if we can have the best of both worlds, using BM25 to weigh how much each token contributes to the sentence embedding? That is the approach codequestion takes. A BM25 index is built over the dataset. Using pre-trained fastText vectors plus BM25, scores as:

MRR = 66.1

Quite an improvement over straight BM25 (66.1 vs 49.5)!

Custom trained fastText embeddings

This level of functionality is an acceptable mix of performance and results. But there is one last thing to try that might improve the score further, a custom trained fastText embeddings model on the questions database itself. Using fastText embeddings trained on the data scores as:

MRR = 76.3

Once again, quite an improvement. We could continue on and try a BERT embeddings model, which would score higher but also take more compute/storage. fastText + BM25 doesn't take order into account, so a query like "python convert UTC to localtime" could match "python convert localtime to UTC". BERT would better handle this use case. But for this project, the embeddings method works well.

Building an index

Embeddings by themselves are just a bunch of numbers. Once the full data repository is transformed to sentence embeddings, there needs to be a way to find similar results for a query.

Cosine similarity is a common method for comparing how similar two vectors are. The simplest approach is brute force comparing the query vector to each result and returning the most similar results. For small datasets, this works well. But as the data grows, it doesn't scale.

Faiss is a vector similarity search library that can help with this. It can quantize vectors to reduce the amount of storage along with supporting approximate nearest neighbor searches that can scale up. Faiss supports highly performant searches that offer a good blend of result accuracy and speed.

Conclusion

This article covered a methodology in evaluating how to build a search system, leading to a sentence embeddings index. The road won't always lead to the same endpoint but having an incremental approach like this should help lead to the best solution for a problem.

Machine Learning · Search · NLP · Python · Towards Data Science