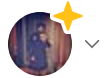




Search Medium

Write



★ Member-only story

Fine-Tuning Your Embedding Model to Maximize Relevance Retrieval in RAG Pipeline

NVIDIA SEC 10-K filing analysis before and after fine-tuning embeddings



Wenqi Glantz · Following

Published in Better Programming · 10 min read · Sep 12



376

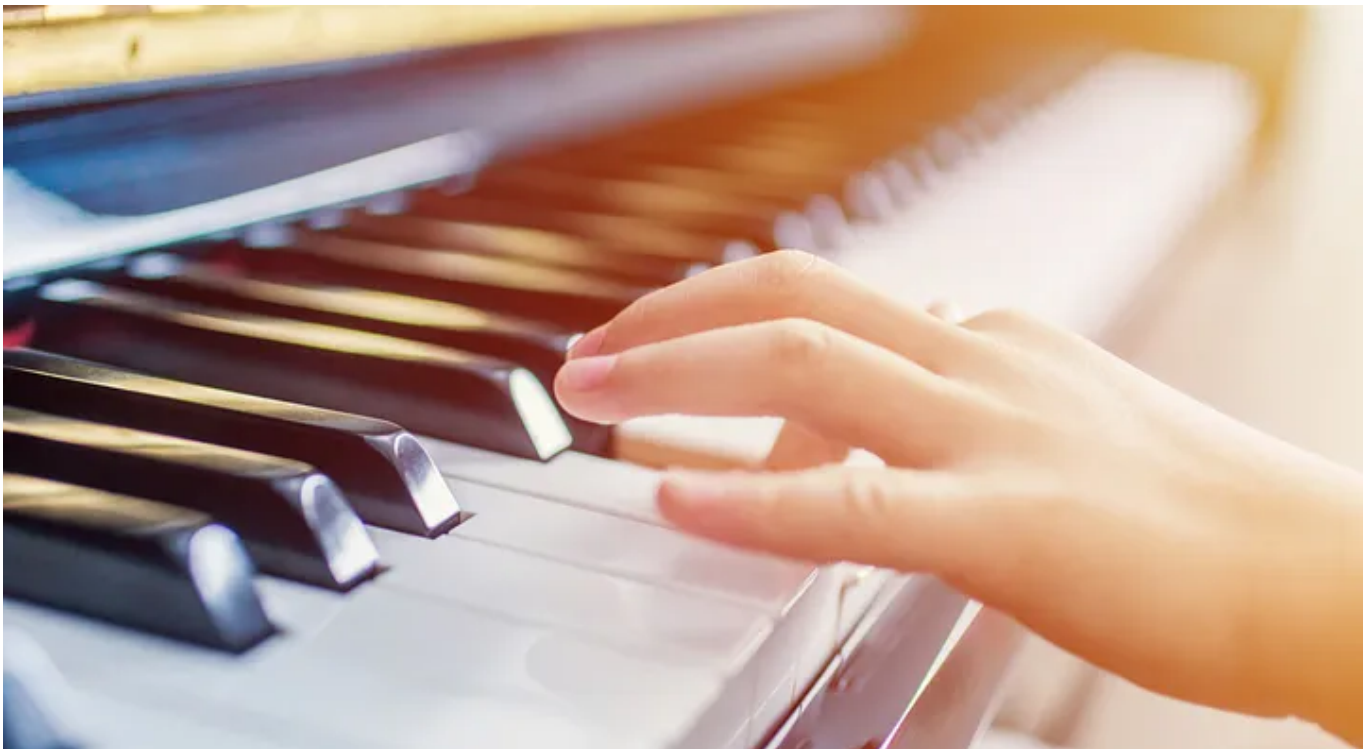


Photo from Canva

Let's continue from our previous article, [Fine-Tuning the GPT-3.5 RAG Pipeline with GPT-4 Training Data](#). This time, let's dive into fine-tuning the other end of the spectrum of our RAG (Retrieval Augmented Generation) pipeline — the embedding model.

By fine-tuning our embedding model, we enhance our system's ability to retrieve the most relevant documents, ensuring that our RAG pipeline performs at its best.

We have been using OpenAI's embedding model `text-embedding-ada-002` for most of our RAG pipelines in our LlamaIndex blog series. However, OpenAI does not offer the feature to fine-tune `text-embedding-ada-002`, so let's explore fine-tuning an open source embedding model in this article.

BAAI/bge-small-en

The current number 1 embedding model on HuggingFace's MTEB (Massive Text Embedding Benchmark) [Leaderboard](#) is `bge-large-en`; it was developed by the Beijing Academy of Artificial Intelligence (BAAI). It is a pretrained transformer model that can be used for various natural language processing tasks, such as text classification, question answering, text generation, etc. The model is trained on a massive dataset of text and code, and it has been fine-tuned on the Massive Text Embedding Benchmark (MTEB).

For this article, we are going to use one of `bge-large-en`'s siblings, `bge-small-en`, a 384-dimensional small-scale model with competitive performance, perfect for running in Google Colab.

Fine-Tune Embedding Model vs. Fine-Tune LLM

From our [last article](#) on fine-tuning `gpt-3.5-turbo`, we gained a solid understanding of the steps involved in fine-tuning an LLM. Compared with LLM fine-tuning, the implementation of fine-tuning `bge-small-en` have some similarities and differences.

Similarities

- Both types of fine-tuning follow the same approach of generating datasets for training and evals, fine-tuning the model, and finally evaluating the performances between the base and fine-tuned models.
- Automate dataset generation for both training and evals with the LLM.

Differences

- Datasets content differs between LLM fine-tuning and embedding model fine-tuning. The datasets for LLM's fine-tuning contain the questions generated by the LLM. During the fine-tuning process, the series of data, including questions, answers, the system prompts, etc., get passed to the model to be fine-tuned in the form of a JSON Line (`jsonl`) file.

However, the datasets for embedding model fine-tuning contain the following three sets:

1. `queries` : a collection of `node_id` mappings and the questions generated by the LLM.
2. `corpus` : a collection of `node_id` mappings and the text in the corresponding node.
3. `relevant_docs` : a collection of cross reference mapping between a query's `node_id` and the corpus' `node_id`. Given a query, this tells the embedding model which text node/corpus to look for.

- Since we are using the open source embedding model, `bge-small-en`, once fine-tuned, it gets downloaded to your local environment. In the case of Google Colab, the fine-tuned model will be downloaded to your notebook's root directory.
- The evaluation method differs between fine-tuning an embedding model and fine-tuning an LLM, where we can use the `ragas` framework to measure faithfulness and answer relevancy. When working with embedding model fine-tuning, however, we cannot measure answer correctness because we can only retrieve the relevant node(s) for our questions. Instead, we use a simple metric called “hit rate,” which means that for each `(query, relevant_doc)` pair, we retrieve top-k documents with the query, and it’s a “hit” if the results contain the `relevant_doc`. This metric can be used for proprietary embeddings, such as OpenAI’s embedding model, and open source embedding models. For open source embeddings, we can also use `InformationRetrievalEvaluator` from `sentence_transformers` for evaluation, as it provides a more comprehensive suite of metrics.

There appears to be a lot involved in fine-tuning an embedding model. Luckily, LlamaIndex has made it super simple to fine-tune embedding models by introducing the following key classes/functions in the recent release version, 0.8.21:

- `SentenceTransformersFinetuneEngine`
- `generate_qa_embedding_pairs`
- `EmbeddingQAFinetuneDataset`

These classes and functions abstract away the detailed integration logic under the hood for us, making it very intuitive for developers to invoke.

High-Level Overview of Fine-Tuning BAAI/bge-small-en

To visualize the main tasks involved in fine-tuning BAAI/bge-small-en, let's look at the following diagram:

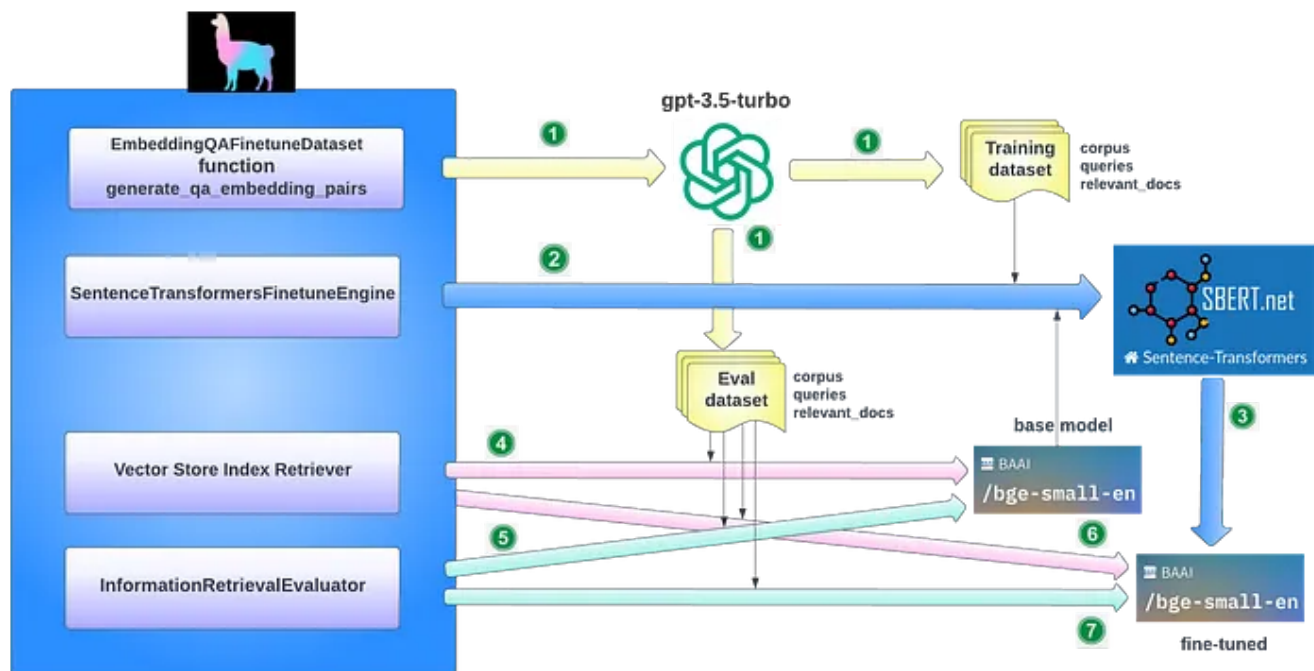


Diagram by author

The main tasks, as indicated by the numeric values in the diagram, include the following:

1. Automate data generation for the evaluation and training datasets by calling `EmbeddingQAFinetuneDataset`'s function `generate_qa_embedding_pairs`.
2. Construct `SentenceTransformersFinetuneEngine` by passing in the base model and the training dataset, then call its `finetune` function to train the

base model.

3. Create the fine-tuned model.
4. Call vector store index retriever to retrieve relevant nodes and assess the hit rate for the base model.
5. Call `InformationRetrievalEvaluator` to evaluate the base model.
6. Call vector store index retriever to retrieve relevant nodes and assess the hit rate for the fine-tuned model.
7. Call `InformationRetrievalEvaluator` to evaluate the fine-tuned model.

We will use the same use case of analyzing NVIDIA's SEC 10-K filing for 2022 as our last article. Based on the [Fine-tune Embeddings guide](#) by LlamaIndex, we will fine-tune the `bge-small-en` model in our use case.

Let's dive in!

Implementation Details

Step 1: Generate datasets

Let's use LLM to auto-generate our datasets for both training and evals.

- Load corpus

In our use case, [NVIDIA's SEC 10-K filing](#) is a 169-page PDF document, so we need to split the document into two portions when generating the datasets — one for the training dataset and the other for the evals dataset.

Using separate datasets for training and evals is considered a good ML practice. The `load_corpus` function can be called to collect the nodes for

either the training dataset (first 90 pages) or the eval dataset (the rest of the pages). Here's a snippet of what `load_corpus` looks like:

```
!curl https://d18rn0p25nwr6d.cloudfront.net/CIK-0001045810/4e9abe7b-fdc7-4cd2-84

def load_corpus(docs, for_training=False, verbose=False):
    parser = SimpleNodeParser.from_defaults()
    if for_training:
        nodes = parser.get_nodes_from_documents(docs[:90], show_progress=verbose)
    else:
        nodes = parser.get_nodes_from_documents(docs[91:], show_progress=verbose)

    if verbose:
        print(f'Parsed {len(nodes)} nodes')

    return nodes

SEC_FILE = ['nvidia-sec-10k-2022.pdf']

print(f"Loading files {SEC_FILE}")

reader = SimpleDirectoryReader(input_files=SEC_FILE)
docs = reader.load_data()
print(f'Loaded {len(docs)} docs')

train_nodes = load_corpus(docs, for_training=True, verbose=True)
val_nodes = load_corpus(docs, for_training=False, verbose=True)
```

Remember, in LlamaIndex, nodes and pages don't match exactly. For a 169-page doc, the result shows it parsed 97 nodes for the training dataset and 91 for the evals dataset. The two datasets have a close enough number of nodes. Let's continue.

```

Loading files ['nvidia-sec-10k-2022.pdf']
Loaded 169 docs
[nltk_data] Downloading package punkt to /tmp/llama_index...
[nltk_data] Unzipping tokenizers/punkt.zip.
Parsing documents into nodes: 100% ██████████ 90/90 [00:00<00:00, 395.52it/s]
Parsed 97 nodes
Parsing documents into nodes: 100% ██████████ 78/78 [00:00<00:00, 185.80it/s]
Parsed 91 nodes

```

- Generate synthetic queries and datasets

Now, let's generate our datasets for training and evaluation. Notice we are not passing an LLM (`gpt-3.5-turbo-0613`) here, only the OpenAI API key. This is because LlamaIndex's default LLM is `gpt-3.5-turbo-0613` ; it defaults to it if no LLM is defined as long as the OpenAI API key is provided.

`generate_qa_embedding_pairs` is a convenient function to generate a dataset. Based on the nodes returned from the `load_corpus` function above, it generates questions for each node (defaults to two questions per node and can be customized), then constructs the dataset with all three sets of data: `queries` , `corpus` , and `relevant_docs` (mapping between `queries` and `corpus` by their corresponding `node_id`).

```

from llama_index.finetuning import (
    generate_qa_embedding_pairs,
    EmbeddingQAFinetuneDataset,
)
from llama_index.llms import OpenAI

os.environ["OPENAI_API_KEY"] = "sk-#####"
openai.api_key = os.environ["OPENAI_API_KEY"]

train_dataset = generate_qa_embedding_pairs(train_nodes)
val_dataset = generate_qa_embedding_pairs(val_nodes)

train_dataset.save_json("train_dataset.json")
val_dataset.save_json("val_dataset.json")

```



```
train_dataset = EmbeddingQAFinetuneDataset.from_json("train_dataset.json")
val_dataset = EmbeddingQAFinetuneDataset.from_json("val_dataset.json")
```

Here is what a sample training dataset looks like. Notice `queries` and `corpus` are collapsed in the screenshot as there are over a hundred data pairs in each:



Step 2: Fine-Tune Embedding Model

`SentenceTransformersFinetuneEngine` is designed for this task. Under the hood, it carries out multiple subtasks:

- Load a pretrained model by constructing a `SentenceTransformer`, passing in `BAAI/bge-small-en` model id.
- Define Dataloader. It loads our training dataset, parses it into `queries`, `corpus`, and `relevant_docs`. It then loops through the queries, maps the `node_id` from `relevant_docs` with the text node from `corpus`, constructs `InputExample`, the list of which in turn gets passed into creating a `DataLoader`.

- Define loss. It uses `sentence_transformers MultipleNegativesRankingLoss` to train embeddings for retrieval setups.
- Define evaluator. It sets up an evaluator with an evals dataset to monitor how well the embedding model performs during training.
- Run training. It plugs in the data loader, loss function, and evaluator defined above to run training.

LlamaIndex encapsulates all these detailed subtasks of fine-tuning an embedding model in one `SentenceTransformersFinetuneEngine`, and all we need to do is to call its `finetune` function. Below, you can see the code snippet showing the beauty and brilliance of LlamaIndex!!

```
from llama_index.finetuning import SentenceTransformersFinetuneEngine

finetune_engine = SentenceTransformersFinetuneEngine(
    train_dataset,
    model_id="BAAI/bge-small-en",
    model_output_path="test_model",
    val_dataset=val_dataset,
)

finetune_engine.finetune()

embed_model = finetune_engine.get_finetuned_model()
```

Step 3: Evaluate the fine-tuned model

As briefly mentioned above, we use two different evaluation approaches:

- Hit rate: simple top-k retrieval for each `query / relevant_doc` pair. If the results contain the `relevant_doc`, it's a "hit." This can be used for proprietary embeddings, such as OpenAI's embedding model and open

source embedding models. See the `evaluate` function in the code snippet below.

- `InformationRetrievalEvaluator` : A more comprehensive suite of metrics for evaluating open source embeddings. See the `evaluate_st` function in the code snippet below.

```
from llama_index.embeddings import OpenAIEmbedding
from llama_index import ServiceContext, VectorStoreIndex
from llama_index.schema import TextNode
from tqdm.notebook import tqdm
import pandas as pd

# function for hit rate evals
def evaluate(
    dataset,
    embed_model,
    top_k=5,
    verbose=False,
):
    corpus = dataset.corpus
    queries = dataset.queries
    relevant_docs = dataset.relevant_docs

    service_context = ServiceContext.from_defaults(embed_model=embed_model)
    nodes = [TextNode(id=id_, text=text) for id_, text in corpus.items()]
    index = VectorStoreIndex(nodes, service_context=service_context, show_progress_bar=True)
    retriever = index.as_retriever(similarity_top_k=top_k)

    eval_results = []
    for query_id, query in tqdm(queries.items()):
        retrieved_nodes = retriever.retrieve(query)
        retrieved_ids = [node.node_id for node in retrieved_nodes]
        expected_id = relevant_docs[query_id][0]
        is_hit = expected_id in retrieved_ids # assume 1 relevant doc

        eval_result = {
            "is_hit": is_hit,
            "retrieved": retrieved_ids,
            "expected": expected_id,
            "query": query_id,
        }
        eval_results.append(eval_result)
```

```
return eval_results

from sentence_transformers.evaluation import InformationRetrievalEvaluator
from sentence_transformers import SentenceTransformer

def evaluate_st(
    dataset,
    model_id,
    name,
):
    corpus = dataset.corpus
    queries = dataset.queries
    relevant_docs = dataset.relevant_docs

    evaluator = InformationRetrievalEvaluator(queries, corpus, relevant_docs, name)
    model = SentenceTransformer(model_id)
    return evaluator(model, output_path="results/")
```

- Evals for OpenAI

Now, let's evaluate OpenAI's embedding model `text-embedding-ada-002`.
Here's the code:

```
ada = OpenAIEmbedding()
ada_val_results = evaluate(val_dataset, ada)

df_ada = pd.DataFrame(ada_val_results)

hit_rate_ada = df_ada['is_hit'].mean()
```

And the result...

```
✓ 0s hit_rate_ada = df_ada['is_hit'].mean()  
hit_rate_ada  
0.9065934065934066
```

- Evals for BAAI/bge-small-en

```
bge = "local:BAAI/bge-small-en"  
bge_val_results = evaluate(val_dataset, bge)  
  
df_bge = pd.DataFrame(bge_val_results)  
  
hit_rate_bge = df_bge['is_hit'].mean()  
  
evaluate_st(val_dataset, "BAAI/bge-small-en", name='bge')
```

And the result...

```
✓ 0s [22] hit_rate_bge = df_bge['is_hit'].mean()  
hit_rate_bge  
0.8516483516483516  
  
✓ 1m [23] evaluate_st(val_dataset, "BAAI/bge-small-en", name='bge')
```

- Evals for the fine-tuned model

```
finetuned = "local:test_model"  
val_results_finetuned = evaluate(val_dataset, finetuned)
```

```
df_finnetuned = pd.DataFrame(val_results_finnetuned)

hit_rate_finnetuned = df_finnetuned['is_hit'].mean()

evaluate_st(val_dataset, "test_model", name='finnetuned')
```

Look at the result:



The screenshot shows two lines of Jupyter Notebook output. The first line, labeled [26], shows the calculation of the hit rate for the fine-tuned model, resulting in 0.8626373626373627. The second line, labeled [27], shows the evaluation of the fine-tuned model, resulting in 0.7315810324641175. Both results are highlighted with red boxes.

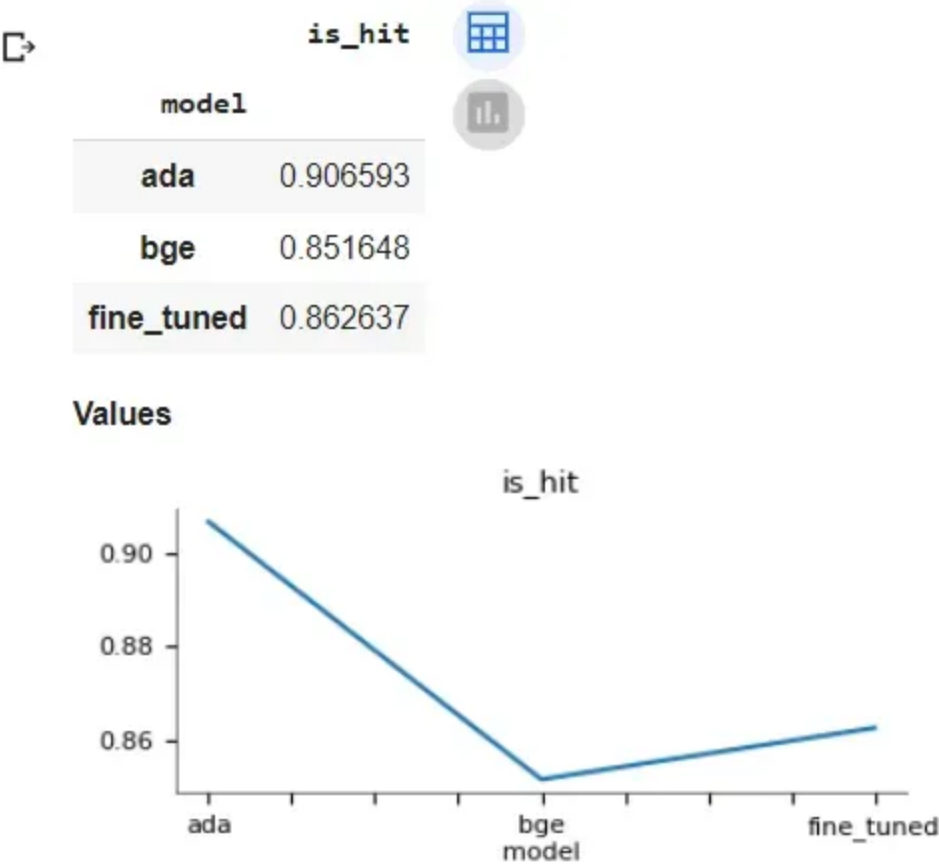
```
✓ [26] hit_rate_finnetuned = df_finnetuned['is_hit'].mean()
0s hit_rate_finnetuned
0.8626373626373627

✓ [27] evaluate_st(val_dataset, "test_model", name='finnetuned')
0.7315810324641175
```

- Summary of results

Putting the eval results side by side, let's take a closer look.

Hit rate: our fine-tuned model has a 1.29% performance increase over its base model `bge-small-en`. Compared with OpenAI's embedding model, our fine-tuned model only has a 4.85% performance loss.



InformationRetrievalEvaluator results: the fine-tuned model has a 5.81% performance boost over its base model. And the fine-tuned model has better numbers for each of those 30+ metrics columns than the base model.

model	epoch	steps	cos_sim_Accuracy@1	cos_sim_Accuracy@3	cos_sim_Accuracy@5	cos_sim_Accuracy@10	cos_sim_Precision@1	cos_sim_Recall@1	cos_sim_Precision@3	cos_sim_Recall@3	...	dot_score_Recall@1	dot_score_Precision@3	dot_score_Recall@3	dot_score_Precision@5	dot_score_Recall@5	dot_score_Precision@10	dot_score_Recall	
	bge	-1	-1	0.576923	0.758242	0.835165	0.901099	0.576923	0.576923	0.252747	0.758242	—	0.582418	0.254579	0.763736	0.164835	0.824176	0.090110	0.901
	fine_tuned	-1	-1	0.631868	0.780220	0.862637	0.939560	0.631868	0.631868	0.260073	0.780220	—	0.631868	0.258242	0.774725	0.174725	0.873626	0.093407	0.934

2 rows × 32 columns

Fine-Tune A Linear Adapter for Any Embedding Model

Developing at the speed of light, LlamaIndex released a new linear adapter for any embedding model in its latest release. A new fine-tuned engine, EmbeddingAdapterFinetuneEngine , was introduced.

When should you use this linear adapter? From [the guide](#) and this [article](#), we learn we can fine-tune a linear adapter on top of embeddings produced from *any* model (`sentence_transformers` , OpenAI, and more). This allows us to transform our embedding representations into a new latent space optimized for retrieval over our specific data and queries. This can lead to small increases in retrieval performance that, in turn, translate to better-performing RAG pipelines.

In situations where you don't want to re-embed your documents with fine-tuning embeddings, this linear adapter is handy as we do not need to re-embed our documents! Simply transform the query instead.

The implementation of fine-tuning a linear adapter on top of your current embeddings is very similar to the steps we walked through in fine-tuning an embedding model. The difference lies in the fine-tune engine. When fine-tuning a linear adapter, we should use `EmbeddingAdapterFinetuneEngine` instead of `SentenceTransformersFinetuneEngine` , which is for fine-tuning an embedding model.

Can fine-tuning an embedding model be combined with fine-tuning a linear adapter? Yes!

- If you have a use case to fine-tune your embedding model, just started your project, and no docs have been embedded yet, `SentenceTransformersFinetuneEngine` is the perfect option for you to fine-tune your embedding model.
- It is also a good idea to call `EmbeddingAdapterFinetuneEngine` to boost the performance of the fine-tuned embedding model on a regular basis.

Summary

We explored the steps involved in fine-tuning an embedding model for a RAG pipeline in this article. We used an open source `sentence_transformers` model `BAAI/bge-small-en` as our base embedding model, walked through the steps on how to generate datasets for training and evaluation, how to fine-tune it, and how to evaluate the performance difference between the base and fine-tuned models.

The evaluation results showed about a 1–6% performance gain for fine-tuned embedding model over the base model, and the fine-tuned model has only 4.85% performance loss compared to OpenAI's embedding model. This performance boost may vary depending on the quality and quantity of your datasets.

We also briefly explored LlamaIndex's latest release of fine-tuning a linear adapter for any embedding model, which boosts performance and avoids re-embedding documents in your RAG pipeline.

The complete source code can be found in [my GitHub repo](#) and [my Colab notebook](#).

Happy coding!

References

- [Finetune Embeddings](#)
- [Fine-Tuning Embeddings for RAG with Synthetic Data](#)
- [BAAI/bge-large-en](#)
- [Fine-Tuning a Linear Adapter for Any Embedding Model](#)
- [finetune-embedding](#)

Fine Tuning

Llamaindex

Baai

Embedding

Eval

More from the list: "NLP"

Curated by Himanshu Birla



Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

★ · 11 min read · Sep 4, 2021



Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

★ · 6 min read · Sep 3, 2021



Jon Gi... in

The Word2vec

★ · 15 min read



[View list](#)



Written by Wenqi Glantz

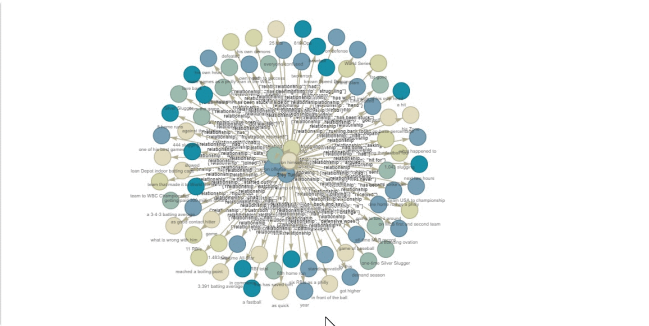
6.3K Followers · Writer for Better Programming

Following



Mom, wife, software architect with a passion for technology and crafting quality products

More from Wenqi Glantz and Better Programming



 Wenqi Glantz in Better Programming

7 Query Strategies for Navigating Knowledge Graphs With...

Exploring NebulaGraph RAG Pipeline with the Philadelphia Phillies

★ · 17 min read · 4 days ago


 501

 4







 Vinita in Better Programming

How To Disagree With Someone More Powerful Than You

Disagreements can save time and energy by preventing critical mistakes and course...

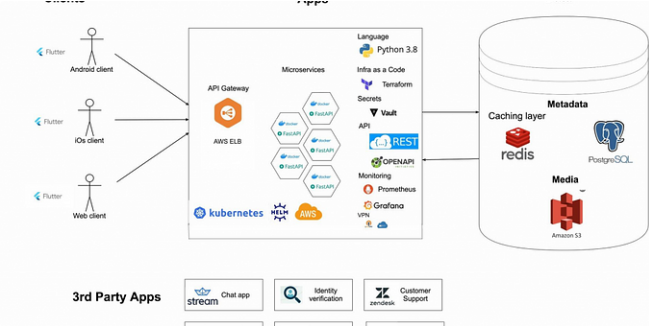
★ · 9 min read · Sep 19


 1.9K

 30








 Dmitry Kruglov in Better Programming

The Architecture of a Modern Startup

Hype wave, pragmatic evidence vs the need to move fast

16 min read · Nov 7, 2022



 Wenqi Glantz in Better Programming

Fine-Tuning GPT-3.5 RAG Pipeline with GPT-4 Training Data

NVIDIA SEC 10-K filing analysis before and after fine-tuning

★ · 11 min read · Sep 4

6.1K

59

238

2

See all from Wenqi Glantz

See all from Better Programming

Recommended from Medium



Han HELOIR, Ph.D. in Artificial Corner

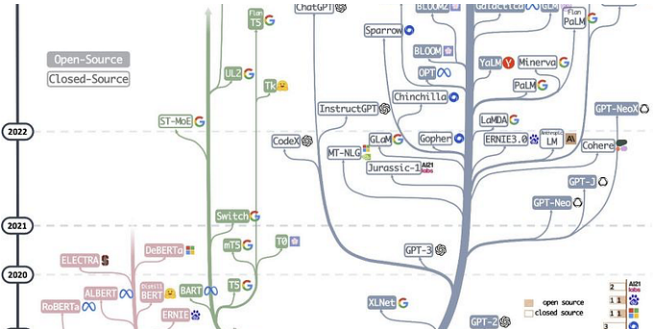
MongoDB and Langchain Magic:
Your Beginner’s Guide to Setting...

Introduction:

★ · 7 min read · Sep 12

1.4K

12



Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone’s...

15 min read · Sep 14

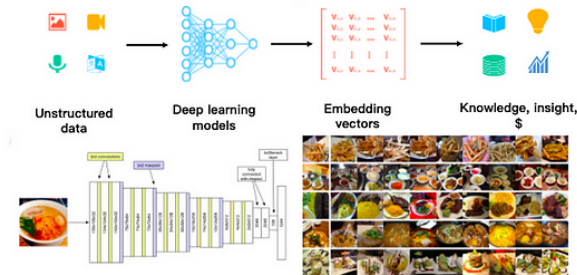
372

Lists



Natural Language Processing

669 stories · 283 saves

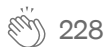


Jayita Bhattacharyya in GoPenAI

Primer on Vector Databases and Retrieval-Augmented Generation...

Vector Databases Generation (RAG)
Langchain Pinecone HuggingFace Large...

9 min read · Aug 16



228



1



Ryan Nguyen in Towards AI

So, You Want To Improve Your RAG Pipeline

Ways to go from prototype to production with LlamaIndex

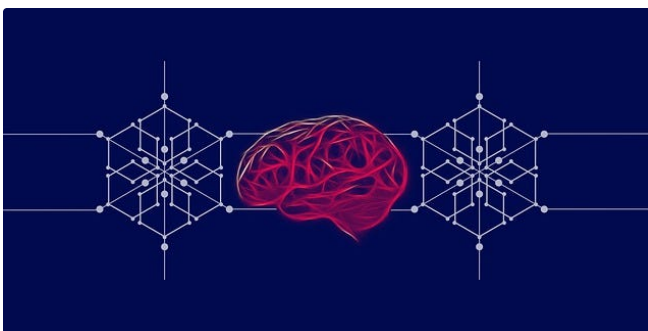
🌟 · 9 min read · Sep 27



176



2

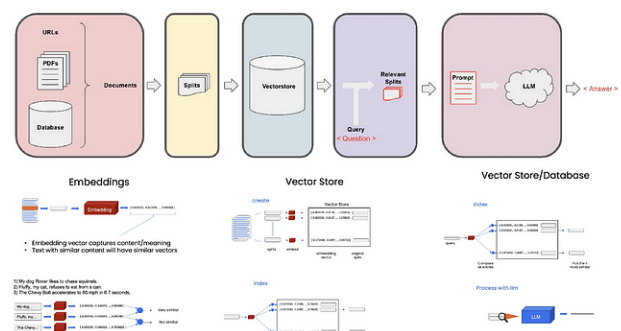


ai geek (wishes)

Best Practices for Deploying Large Language Models (LLMs) in...

Large Language Models (LLMs) have revolutionized the field of natural language...

10 min read · Jun 26



TeeTracker

Chat with your PDF (Streamlit Demo)

Conversation with specific files

4 min read · Sep 15



See more recommendations