The best strategy for implementing an LLM embedding model depends on your specific needs and requirements. However, there are some general guidelines that you can follow:

1. **Choose the right embedding model.** There are many different embedding models available, each with its own strengths and weaknesses. Some popular options include Word2Vec, GloVe, and BERT. Consider the tasks that you want your embedding model to perform and the size of your dataset when choosing a model.
2. **Train your embedding model.** If you are using a pre-trained embedding model, you may not need to train it further. However, if you are using a custom embedding model, or if you want to fine-tune a pre-trained model, you will need to train it on your dataset.
3. **Implement your embedding model.** Once your embedding model is trained, you need to implement it in your application. This typically involves writing code to load the embedding model and to embed your text data.
4. **Evaluate your embedding model.** Once you have implemented your embedding model, you should evaluate its performance on your target tasks. This will help you to identify any areas where the model can be improved.

Sizing your LLM embedding model correctly

The size of your LLM embedding model will depend on the following factors:

- The size of your dataset
- The complexity of the tasks that you want your embedding model to perform
- The available computational resources

If you have a large dataset and/or you want your embedding model to perform complex tasks, you will need a larger model. However, larger models require more computational resources to train and deploy.

To size your LLM embedding model correctly, you can start by using a pre-trained model. If you are not satisfied with the performance of the pre-trained model, you can try training a custom model or fine-tuning the pre-trained model on your dataset.

You can also use a technique called chunking to reduce the computational resources required to train and deploy your embedding model. Chunking involves breaking down your text data into smaller segments and then training or fine-tuning your embedding model on each segment separately.

Once you have trained or fine-tuned your embedding model, you can evaluate its performance on your target tasks. If the model is not performing well, you can try increasing the size of the model or using a different embedding model.

Here are some additional tips for implementing and sizing your LLM embedding model:

- Use a cloud-based service to train and deploy your embedding model. This can save you time and money on computational resources.
- Use a vector database to store your embedding vectors. This will make it easy to query and retrieve your embedding vectors.
- Consider using a transfer learning approach. This involves using a pre-trained embedding model as a starting point for training a custom model on your dataset.
- Experiment with different embedding models and chunking strategies to find the best solution for your needs.

Selecting an Vector/embedding strategi

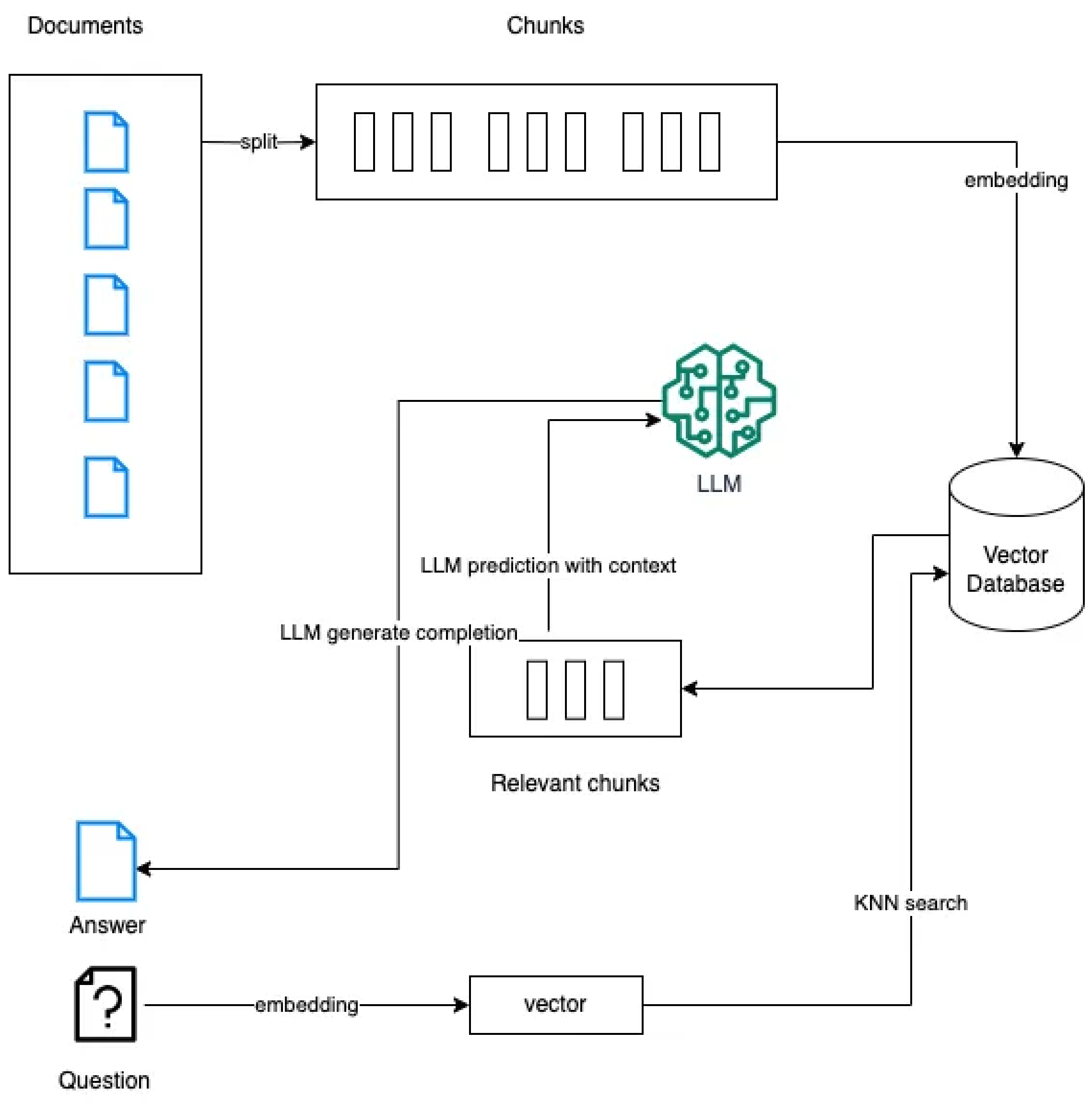
What We Need to Know Before Adopting a Vector Database

To continue with our journey toward applicable Generative AI, I would like to discuss some of the challenges of applying vector databases. These issues are not well discussed, let alone have widely accepted solutions. Vector databases are such a new concept that awareness of these challenges will lead to a more prudent system design.

Table of Contents

Why Vector Database

For people who are familiar with Retrial Augmented Generation (RAG), the purpose of a vector database is easy to understand. As described in the article[1], a large text file needs to be split into chunks to allow LLM to seek the most relevant piece of information. The chunks need to be stored in a vector store. Then, when answering the queries, the RAG will use vector similarity function and KNN to select the chunks that are most relevant to the query.



RAG Architecture

In the case of RAG, the content of the chunk is text, and the way to present the meaning of the chunk is by using a vector—a high-dimensional data space. And a similarity function will be used to choose the most relevant chunks. In most systems, the similarity function being used is the cosine similarity function. People choose the cosine similarity function because it is not sensitive to the scale of the vectors. It only checks the angle between the two vectors. The closest vectors have very small angles. The cosine score will be close to 1. Otherwise, the unrelated vectors formed angle will be diagonal, and the cosine score will be close to 0.

And the payload of a vector database is the key-value pairs, with the vector being the key and the chunk content being the value for the RAG applications. Actually, the content of a vector database doesn’t have to be chunks of text, it can be images, videos, audio, or whatever you care about, and you can generate a vector to represent its meaning.

Imagine we are building a small vector store, and the response time is not a problem; we can simply use any SQL or No-SQL database to store the key-value pairs. We only need to run the cosine similarity function iteratively through the vector store to see which one has the highest score.

If that is the use case, then we don’t have to use a vector database. We can save our time for something more important. But the size of the data in most projects is much larger, and the response time has to be within a certain SLA. In this scenario, we need to use a sophisticated vector database to get optimal speed.

An enterprise-ready vector database offers much more than just optimal speed. I’m aware of that for sure. Let’s just keep it simple for the moment.

Different Types of Vector Databases

The field of Vector databases is indeed pretty crowded. There are two types of vector databases:

- SQL databases, No-SQL databases, and full-text search engines extend their functions to support vector data storage and similarity search. For example, Cassandra announced its support for the vector database [2]; Elastic Search is also catching up [3]
- Tailor-made commercial and open-sourced Vector databases. The leading products are:

Milvus
zilliz
Pinecone
vespa
marqo
qdrant
Weaviate
Chroma
Vald
LanceDB

Taylor-mode vector databases

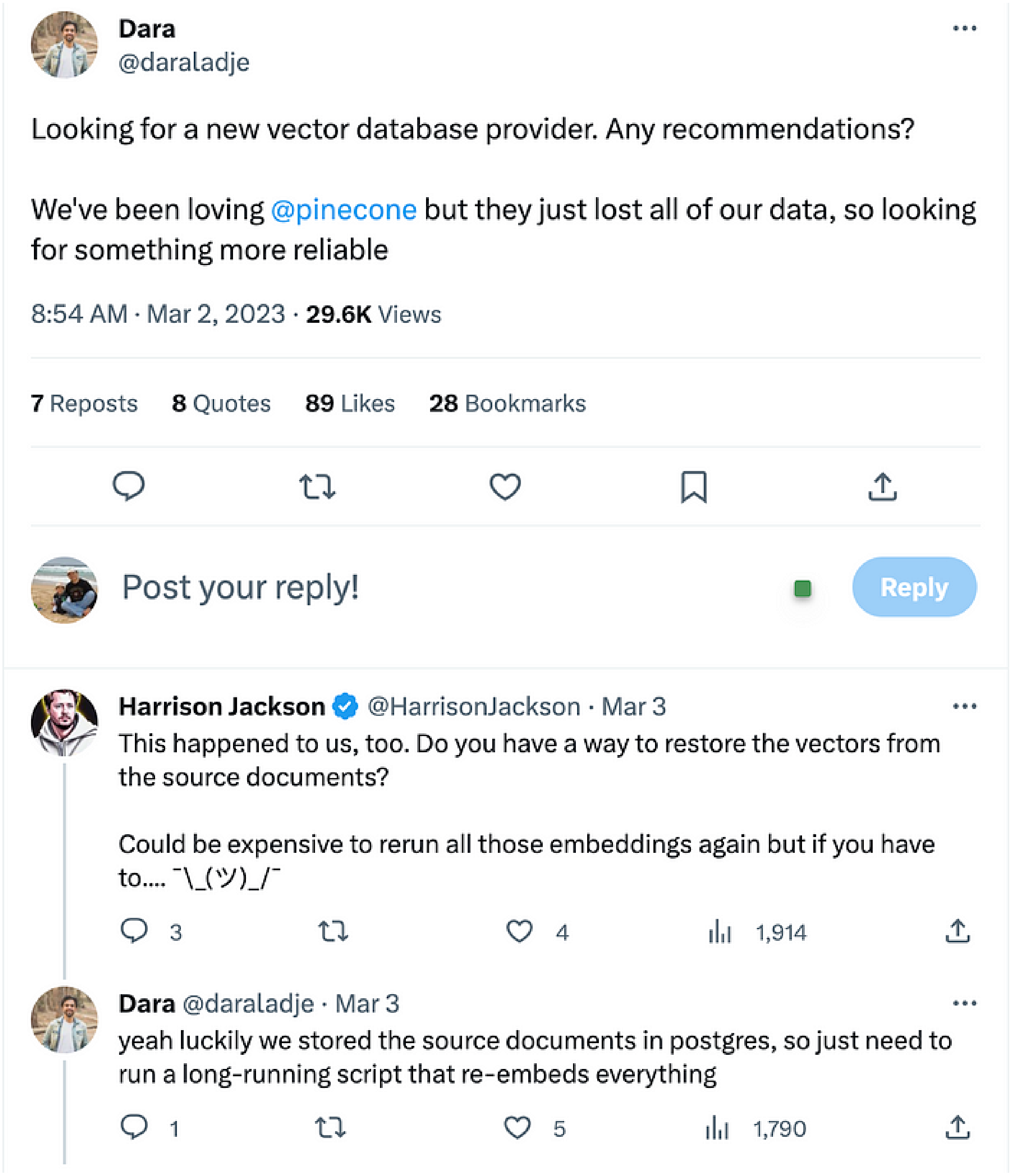
These two types of vector databases both have their pros and cons. While vector-enhanced SQL, No-SQL databases, and full-text search engines are able to support vector search, their performance and function are not as good as those of pure vector databases. Moreover, traditional databases are not designed to handle non-structural data types like images, audio, and video. They are not scaling very well.

In the following discussion in this article, we will only consider the more typical vector databases.

Challenges of Vector Database Application

Vector Databases are Immature

In contrast to traditional databases, pure vector databases offer comprehensive vector operations and better performance; however, they are not as enterprise-ready as those very mature solutions. To mention some of the problems, nearly all vector databases only support metadata filtering in addition to vector operations. They don't support sophisticated queries like SQL and don't integrate with any other systems. You need to independently verify their access control, resilience plans, transaction control, ACID isolation, and even CRUD support. Lacking awareness of these limitations can cause very costly problems in real business.



Dara
@daraladje

Looking for a new vector database provider. Any recommendations?

We've been loving @pinecone but they just lost all of our data, so looking for something more reliable

8:54 AM · Mar 2, 2023 · **29.6K** Views

7 Reposts **8** Quotes **89** Likes **28** Bookmarks

Harrison Jackson @HarrisonJackson · Mar 3
This happened to us, too. Do you have a way to restore the vectors from the source documents?

Could be expensive to rerun all those embeddings again but if you have to.... ¯_(ツ)_/¯

3 4 1,914

Dara @daraladje · Mar 3
yeah luckily we stored the source documents in postgres, so just need to run a long-running script that re-embeds everything

1 5 1,790

I noticed that Pinecone has some kinds of index backup suggestions. I trust disasters like the above can be remedied if people know the risks and plan ahead. That's the purpose of this article.

Knowing the Tradeoffs

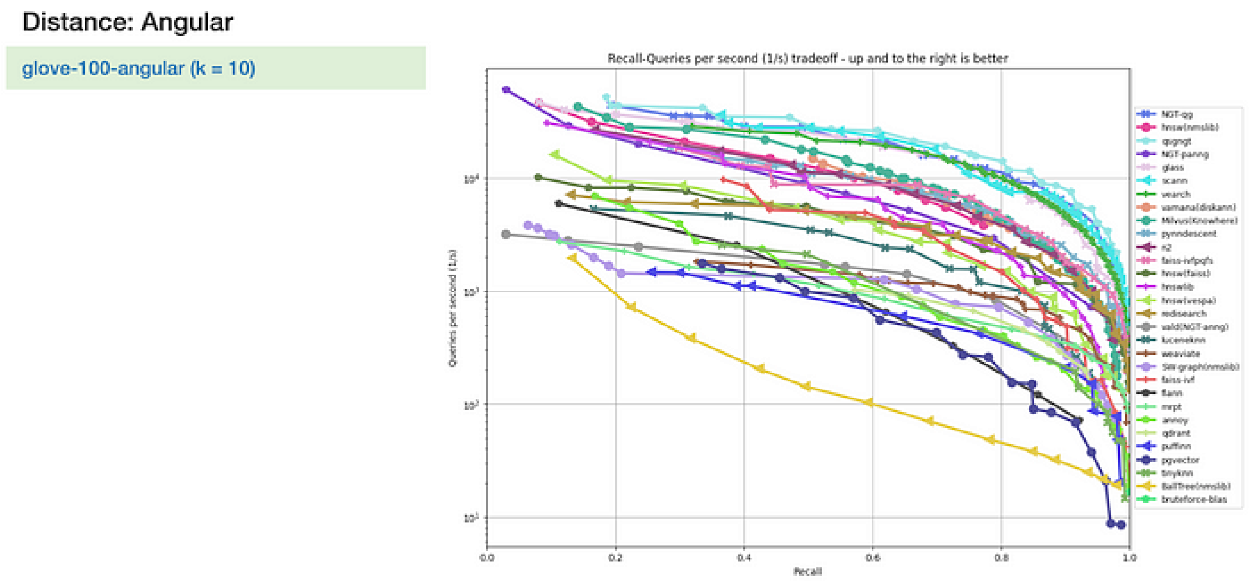
For engineers who are familiar with any traditional database, they may get the impression that a database is just like a car that can hit the ground and run without too much tuning. Even if a database requires some tuning, it is most often the database administrator's responsibility. However, a vector database is not like that. There are a few serious decisions to make in the database design phase instead of the maintenance phase.

The secret of the vector database's high efficiency is that it uses sophisticated data structures and algorithms to make the query faster. The process of building the data structure is called indexing. During the indexing, the developers have to make their first tradeoff: do we want to bias toward faster indexing or better query speed? Then they need to decide on the second question: how to balance accuracy and speed?

For instance, Pinecone's IMI index (Inverted Multi-Index, a variant of ANN) creates storage overhead and is computationally intensive. It is primarily designed for static or semi-static datasets, and can be challenging if vectors are frequently added, modified, or removed. Milvus uses indexes called Product Quantization and Hierarchical Navigable Small World (HNSW), which are approximate techniques that trade off search accuracy for efficiency. Moreover, its indexing requires configuring various parameters. Incorrectly setting up parameters may impact the quality of search results or introduce inefficiencies.

HNSW is a very popular indexing technology because it offers very high query performance. The drawback is that it requires building up a graph of all the vector nodes before hand. For a major vector database with ~1 billion vectors, you can expect a whole day or even longer to complete the initial indexing.

To get a good impression of how the tradeoffs impact performance and make an educated decision about selecting the right algorithm, it's good to check the ANN Benchmarks website [6]. This is one of the charts in the benchmarks:



ANN benchmark

Takeaway from the chart: higher accuracy leads to a quickly dropped performance; higher dimensions cause slower speed. For applications that desperately seek high accuracy, this is really a painful decision point.

Be Cost-Aware

Assuming we are building a RAG application with a modest 10 million pages, each page has 3000 characters, or 800 tokens, and will be processed as a chunk. Thus, we have 10 million embeddings. We chose the OpenAI embedding model text-embedding-ada-002 as the embedding model, and the embedding vector will have 1536 dimensions. Disregard all metadata; every chunk will occupy 1536 x 4 + 3000 = 9.144 KB. The total storage requirement is 91GB.

We use four Pinecone P1.x8 pods. It would cost about $3,300 per month for the vector database. There is also a one-time cost of $3,300 for embedding generation, and an additional $3,300 every time the embedding model is changed. If the application receives a lot of queries per month, there will be an additional recurring expense for the query embedding service and response generation.

Please notice that this is a relatively small dataset, and the runtime settings are not performant. With these settings, the database's capacity is only 30 queries per second. It takes 4 days to initialise the 10 million vectors at this speed. If we are not happy with this speed, we will need more replicas. The cost will be much higher than in this calculation. Enterprises with larger corpora can expect to pay 10 to 100 times more for a RAG application.

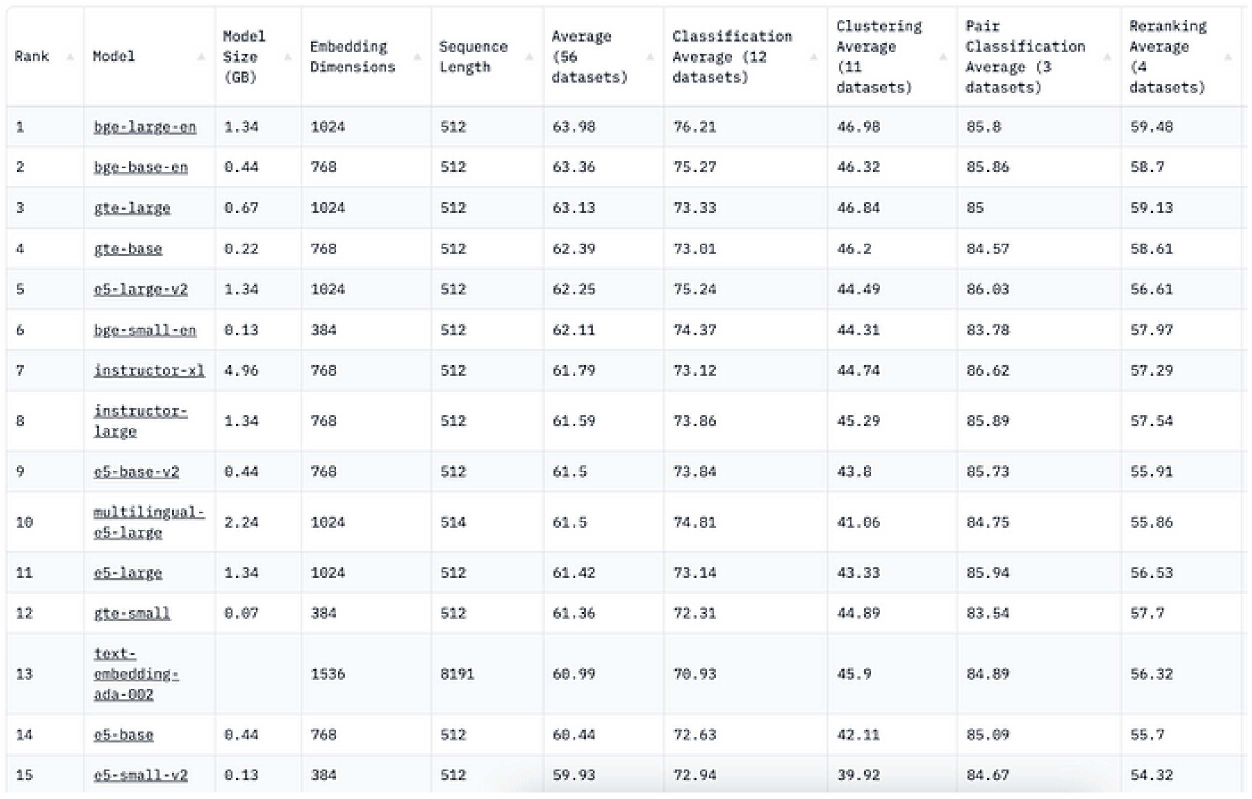
Tight-Coupling with the Embedding Models

All vector databases are focused on how to efficiently store and retrieve vectorised embeddings. None of them offers a solution for managing the embedding process, e.g., the process of generating the vectors. As described in the RAG architecture, embedding models is required in two processes:

- indexing, or populating the vector database with the payload key-value pairs or embedding vector and the chunk
- querying, or generating the embedding for the query to be used in the similarity search.

In both processes, the embedding model must be the same. In the whole life cycle of the vector database, all the embedding vectors must be generated with the same model. The models are not only producing the same number of dimensions, but they must be exactly the same model.

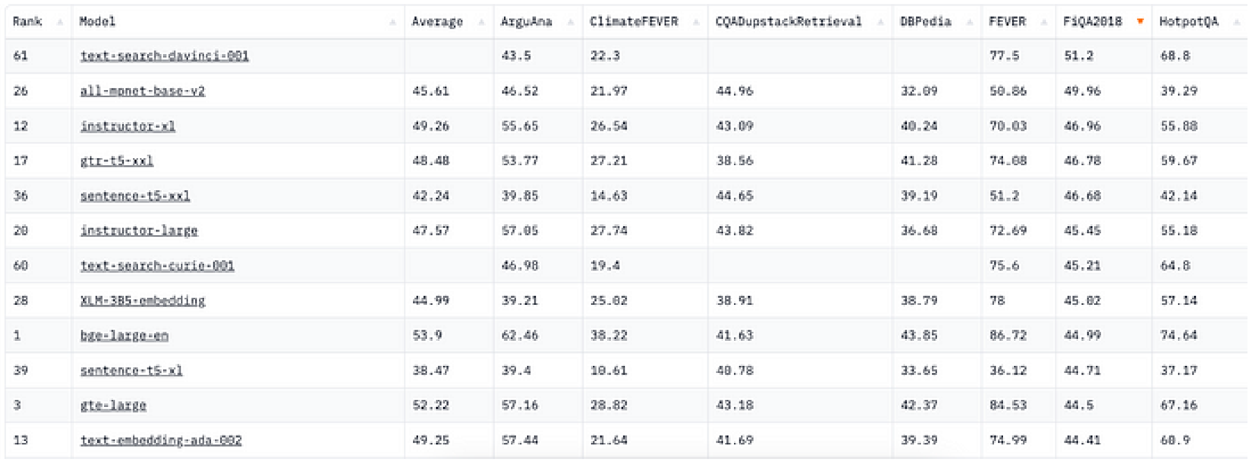
In that case, we have to investigate the stability of the embedding models. Just three weeks ago, the top model on the MTEB benchmark was E5-large-v2, and the text-embedding-ada-002 ranked 7th. Since then, the BGE and GTE model families have surpassed the E5-large-v2. The once-cutting-edge E5-large-v2 is only ranked 5th, while the text-embedding-ada-002 has fallen to 13th.



Rank	Model	Model Size (GB)	Embedding Dimensions	Sequence Length	Average (56 datasets)	Classification Average (12 datasets)	Clustering Average (11 datasets)	Pair Classification Average (3 datasets)	Reranking Average (4 datasets)
1	bge-large-en	1.34	1024	512	63.98	76.21	46.98	85.8	59.48
2	bge-base-en	0.44	768	512	63.36	75.27	46.32	85.86	58.7
3	gte-large	0.67	1024	512	63.13	73.33	46.84	85	59.13
4	gte-base	0.22	768	512	62.39	73.01	46.2	84.57	58.61
5	e5-large-v2	1.34	1024	512	62.25	75.24	44.49	86.03	56.61
6	bge-small-en	0.13	384	512	62.11	74.37	44.31	83.78	57.97
7	instructor-xl	4.96	768	512	61.79	73.12	44.74	86.62	57.29
8	instructor-large	1.34	768	512	61.59	73.86	45.29	85.89	57.54
9	e5-base-v2	0.44	768	512	61.5	73.84	43.8	85.73	55.91
10	multilingual-e5-large	2.24	1024	514	61.5	74.81	41.06	84.75	55.86
11	e5-large	1.34	1024	512	61.42	73.14	43.33	85.94	56.53
12	gte-small	0.07	384	512	61.36	72.31	44.89	83.54	57.7
13	text-embedding-ada-002		1536	8191	60.99	70.93	45.9	84.89	56.32
14	e5-base	0.44	768	512	60.44	72.63	42.11	85.09	55.7
15	e5-small-v2	0.13	384	512	59.93	72.94	39.92	84.67	54.32

MTEB Leaderboard as of 14/08/2023

The LLM is an incredibly fast-evolving field. It is very bold to bet that a model does not require updates during the whole lifecycle of a vector database system. If we further check the Retrieval Ranking tab of the MTEB leaderboard:



Rank	Model	Average	ArguAna	ClimateFEVER	CQADupstackRetrieval	DBPedia	FEVER	FiQA2018	HotpotQA
61	text-search-davinci-001		43.5	22.3			77.5	51.2	68.8
26	all-mpnet-base-v2	45.61	46.52	21.97	44.96	32.09	50.86	49.96	39.29
12	instructor-xl	49.26	55.65	26.54	43.09	40.24	70.03	46.96	55.88
17	gtr-t5-xxl	48.48	53.77	27.21	38.56	41.28	74.08	46.78	59.67
36	sentence-t5-xxl	42.24	39.85	14.63	44.65	39.19	51.2	46.68	42.14
20	instructor-large	47.57	57.05	27.74	43.82	36.68	72.69	45.45	55.18
60	text-search-curie-001		46.98	19.4			75.6	45.21	64.8
28	XLM-385-embedding	44.99	39.21	25.02	38.91	38.79	78	45.02	57.14
1	bge-large-en	53.9	62.46	38.22	41.63	43.85	86.72	44.99	74.64
39	sentence-t5-xl	38.47	39.4	10.61	40.78	33.65	36.12	44.71	37.17
3	gte-large	52.22	57.16	28.82	43.18	42.37	84.53	44.5	67.16
13	text-embedding-ada-002	49.25	57.44	21.64	41.69	39.39	74.99	44.41	60.9

MTEB Retrieval Ranking Leader board as of 14/08/2023

We will notice that the ranking varies from task to task. For the FIQA2018 task, the leading model is text-search-davinci-001, which has a low overall ranking of just 61st. OpenAI has announced that text-embedding-ada-002 will take its place. This task-related performance difference reveals the necessity of building project-related performance metrics and project-fine-tuned embedding models[4, 5].

The problem is that the data will have to be re-indexed whenever the model changes. As discussed earlier, re-indexing is a time-consuming, costly, and service-disrupting process.

The ever-evolving embedding models and the fine-tuned models leave the developers in a dilemma: how can we maintain the models without disturbing the data too much? If you update your LLM by fine-tuning, upgrading the model, or even increasing your dimensionality, you need to re-index and pay the full cost again.

Conclusion

Compared to traditional pure software systems, machine learning systems are always more complicated to develop and maintain. The challenge of applying vector databases is not special. Personally, I think today's vector databases still have a lot of room to improve. But here, I would suggest practitioners address these challenges through engineering strategies.

First of all, we need to do more research on the technical options. This includes comparing more products and doing more experiments to testify to a wide range of functional and non-functional solutions. That means that after an impressive MVP presentation, we need to hold back a little bit before launching a major project.

Second, communication. A successful ML project requires all stakeholders to get involved. However, Machine Learning is still not a well-known concept in most companies. So, be sure to communicate with all different parties. For example, we need to discuss this in detail with the product owners to find the right balance between accuracy and latency, and we need to collaborate with the maintenance team to know how to set up the gaol of the product's stability.

Last but not least, we need to develop a strategy for managing the embedding models. We need to keep records of the models and their mapping to the vector data. We also need a strategy for updating the database with changing models.

This article is by no means trying to intimidate people away from applying vector databases. Instead, I would encourage ML engineers to get familiar with this new technology, including its beauty and complexity.

Svar • Legg til lenke som vedlegg • Slett

Kennet Dahl Kusk i går kl. 13.22

RAG and finetuning are two different approaches to improving the performance of large language models (LLMs) on specific tasks.

RAG stands for Retriever-Augmented Generator. It is a hybrid approach that combines the power of retriever systems and generative models. Retriever systems are able to retrieve relevant information from large knowledge bases, while generative models are able to generate text. RAG systems work by first retrieving relevant information from a knowledge base, and then using that information to generate a response.

Finetuning is a process of retraining a pre-trained LLM on a new dataset of task-specific data. This helps the LLM to learn the patterns and relationships that are relevant to the new task. Finetuning can be used to improve the performance of LLMs on a wide range of tasks, such as machine translation, question answering, and summarization.

Here is a table comparing RAG and finetuning:

Characteristic	RAG	Finetuning
Approach	Hybrid	Retraining
Knowledge base	Required	Not required
Labeled data	Less required	More required
Computational resources	Less required	More required
Performance	Good on a wide range of tasks	Best on complex tasks
Transparency	More transparent	Less transparent

Which approach to choose depends on the specific task and the available resources:

- **RAG** is a good choice for tasks where labeled data is scarce or expensive to obtain, or where the knowledge base is important for generating accurate and informative responses. For example, RAG is often used for tasks such as question answering and chatbot development.
- **Finetuning** is a good choice for tasks where complex patterns and relationships need to be learned, and where there is enough labeled data available. For example, finetuning is often used for tasks such as machine translation and text summarization