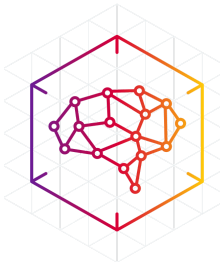
DATASTAX

WHITEPAPER

Vector Search for Generative AI Apps



A guide for developers and architects for using vector search with AI applications.

This document serves as a guide for anyone designing and building generative AI applications combined with proprietary data. We cover the key concepts and considerations that organizations should understand and provide a simple, powerful approach to vastly extending the capabilities of LLMs with vector search.

Generative AI is changing the way we build and use products. OpenAI introduced ChatGPT on November 30, 2022, and in only a few short months, interest in generative AI has taken the world by storm. McKinsey now estimates that generative AI has the potential to be between $2.4 and $4.2 trillion in value to the global economy[1] as of this writing.

At the center of this revolution lies the innovation made possible by Large Language Models (LLMs). LLMs have rapidly become a critical piece of infrastructure and have been adopted by the major cloud providers and beyond. These recent trends include Microsoft's investments in OpenAI[2], the launch of Microsoft Azure OpenAI Service[3], Google's partnership with Anthropic[4], the launch of PaLM 2[5] and Bard[6], a thriving open source community with dozens of models on Huggingface[7], and the introduction of tools and services to build generative AI apps in Amazon Sagemaker[8] and Google's Vertex AI and Generative AI App Builder[9].

Developers are already discussing the new language model stack[10] . Popular new frameworks have emerged such as LangChain[11] and LlamaIndex[12] for building generative AI apps, and vector search has emerged as another crucial, enabling component of the stack for providing long-term memory, real-time access to proprietary data, and customization in generative AI applications.

Here are a few additional facts and survey results indicating the growth of interest in generative AI:

- ChatGPT was the fastest app ever to grow to 100 million Monthly Active Users, in under 3 months[13]
- OpenAI's plugin ecosystem has rapidly grown to over 400 plugins across 21 separate categories since launch[14]
- 92% of developers are using generative AI tools[15]
- 90% of VC-backed companies plan to launch generative AI products[16]
- 65% of executives believe generative AI will have a high or extremely high impact on their organizations[17]
- SEMRush ranked ChatGPT as the 26th most popular Google search term in 2023[18]

In this paper, we'll look into why generative AI is important, the architecture and design patterns that make it work effectively, and how vector search is now an essential component of generative AI architectures.

Generative AI Has Made Building AI-Powered Applications Easier Than Ever

Generative AI and large language models (LLMs) are producing lots of excitement. But for developers, they're even more revolutionary, because they have dramatically simplified how AI applications can be built. Let's take a look at the traditional and generative approaches to understand why.

Building Traditional AI Applications

Before generative AI became widespread, there were many barriers to entry when building AI applications. The major steps in the process were:

1. **Encode the most relevant pieces of your data as vectors.**
   Understanding what the "most relevant pieces" are is a challenging problem, and often involves building separate models just to figure this out instead of making an educated guess. Extracting the relevant pieces from the raw data often proves itself to be a difficult problem. For example, one of Netflix's most important goals is to predict "what you will want to watch when you log in". The vectors that require encoding in this case span many datasets thus increasing complexity.

2. **Train a model using those vectors to accomplish your goal.**
   For common machine learning problems such as image classification, there may be off-the-shelf models that we can "fine-tune" to our particular problem. Google's AutoML Vision[19] was a good example of such a system. We can simply provide the model with an example set of what a dog looks like, and then we have a tailored version of the model that can recognize a dog. In other problems, there are no pre-existing models that we can fine-tune in this way. In these cases, a custom model needs to be trained which can be time-consuming and compute-intensive. Further, modifying existing models for new use cases requires a deep understanding of the underlying mathematics to produce correct results.

3. **Deploy the model and expose it as an API.**
   Deploying a machine model includes making the model service available in the application environment, ensuring it can access the data it needs, and then designing an interface for other applications to access it and request results from its prediction step. The choices for API implementation of this step directly depend on how predictions will be used. Additionally, authentication/authorization will require consideration and implementation.

4. **Run the model in production.**
   Once a model is in production, we often run into new challenges. Some common examples include the speed of the prediction (output) step of the model, model outputs degrading in quality over time, and a lag in critical input data to the model. While there are solutions available for solving each of these problems, significant ongoing attention must be paid to many aspects of the long-term operational life cycle of these types of systems in order to maintain peak performance, deal with changing data or data volumes over time, etc.

Deploying and exposing models is straightforward, although not without challenges. All the other steps in this process often involve bespoke solutions. Further, the team and skill sets required to build and ship this style of AI application can be difficult to assemble and retain. It is common to require a team of 3-4 people to accomplish this. For example, a traditional AI application team might consist of the following roles and accompanying skills:

- Data engineer - responsible for acquiring the raw datasets to be used and then storing and cleaning the data to make it easy for exploration and use. The data engineer needs to be an expert in data modeling, ETL pipelines, and data warehousing.
- Data scientist - responsible for data exploration and feature engineering to turn the data into vectors that will work with the models. A deep knowledge of statistics is often required for the proper selection and training of the AI model itself to avoid problems like bias or overfitting in the solution.
- Decision scientist - expertise in A/B and hypothesis testing. This team member helps ensure that models have the correct outcome for the product. They may also help with the exploratory analysis the data scientist tackles before building an AI model to determine if the insights in the data show causation and not just correlation before investing heavily in building an AI product. In many teams, the data scientist covers this role as well.
- Machine learning engineer - primarily responsible for the last step, running the model in production. This team member ensures the feature engineering and model prediction steps can run at scale, are properly monitored, and return predictions with acceptable response times for the application.

These are all highly specialized roles with deep technical skills, and they are all necessary to successfully build and operate traditional AI applications.

Building AI Applications With Generative AI

The reason that everyone is so excited about generative AI with LLMs is that you can often solve a problem without any of the steps above, and in particular without the complex bespoke solutions that were previously required.

With generative AI all you have to do is:

1. Figure out how to get your data into the LLM *(context)*
2. Describe the task you want the AI model to perform in English (*prompt*)
3. Make an API call to an LLM API

This is a dramatically simpler model compared to building traditional AI applications and as such, Generative AI is well within the capabilities of every team and any developer.

> "The hottest new programming language is English"
>
> **- Andrej Karpathy** - Director of AI, Tesla on Twitter[20]

Generative Context

The most important step in building a generative AI application is the first one above, providing context to the LLM. This is referred to as **Retrieval Augmented Generation (RAG)**. Without additional context, LLMs are a mile wide and an inch deep, not much more than interesting toys. RAG gives LLMs the power to solve real business problems.

But providing an LLM with paragraphs or pages of unrelated information won't help. You need to give it *semantically relevant information*. Identifying that information is surprisingly easy using **Vector Search.**

What is Vector Search?

Vector search is a method to find related objects that have similar attributes or characteristics. Common examples of objects that work with vector search include text, physical products, images, and video.

Vector search uses a specific type of machine learning model called **embeddings** to describe the objects and their context. Embeddings are vectors that capture the semantics of objects and we'll discuss them more in the next section. This approach allows vector search to find similar data without needing to know exactly what you are looking for in advance.

Using the English language as an example, the words "happy", "cheerful", and "joyful" all have similar meanings, but in a traditional keyword-based search, documents matching "cheerful" and "joyful" would not be returned for a query of "happy". This is the power of vector search, it understands meaning, allowing users to describe what they are searching for without the need to be exact.

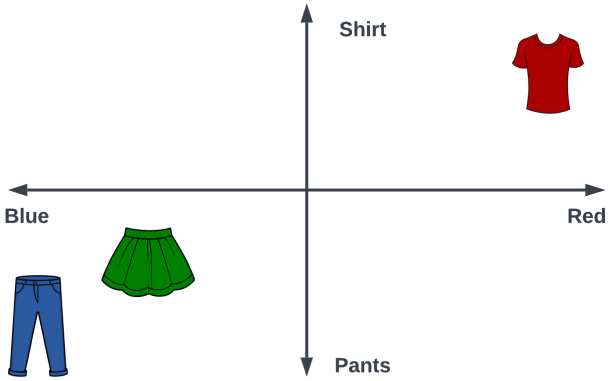
Vector search can also be executed very fast using a set of algorithms called Approximate Nearest Neighbor (ANN) search. These approaches are very efficient and quickly return vectors that are mathematically near each other.

Embeddings 101

Embeddings are a mathematical representation of data that captures the meaning of the text. The process of vectorizing text to turn it into an embedding involves converting the words in the document into a list of numbers. The resulting embeddings then place related content nearby to other similar content in the vector space.

Let's look at an example to make this more clear. In Figure 1 we show a simple embedding space with two dimensions, one for the type of clothes and a second dimension for the attribute of color.

Figure 1. A 2-dimensional embedding space for clothes.

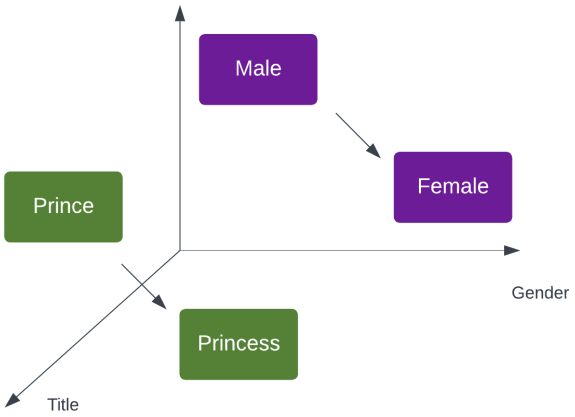


In this example, the vectors have two dimensions, and the entries represent `[clothing type, color]`. The red shirt has a vector [1,1] and the blue pants would have a vector [-1,-1]. Now let's look at the green skirt. A skirt is closer to a pair of pants in function and green is near to blue in the color spectrum, so its embedding vector is near the blue pants and might look like [-0.8,-0.8].

The previous example gives some idea of how to construct an embedding vector space and where those objects live.

Let's look now at an example that illustrates how arithmetic operations on embeddings behave. In this example, the vector space may have a very large dimension (the vector may be composed of many numbers). In that vector space, we have enough information to capture the concept of a person's title as well as their gender. See Figure 2 below for an example.

Figure 2. Example of vector arithmetic that preserves semantic meaning.



Because the embeddings are vectors, we can perform some arithmetic on the vectors and those operations should preserve a concept of what the vectors mean in English.

For example, if we take `Vector(Prince) – Vector(Male) + Vector(Female)` the resulting vector will be extremely close in distance to `Vector(Princess)`.

Similarly, the following two vectors should be very close in distance to each other: `Vector(Prince) – Vector(Male)` and `Vector(Princess) – Vector(Female)`. In this case, both these vectors capture the semantic concept of royalty that is a direct descendant of a still-ruling king or queen.

The Retrieval Augmented Generation (RAG) Pattern

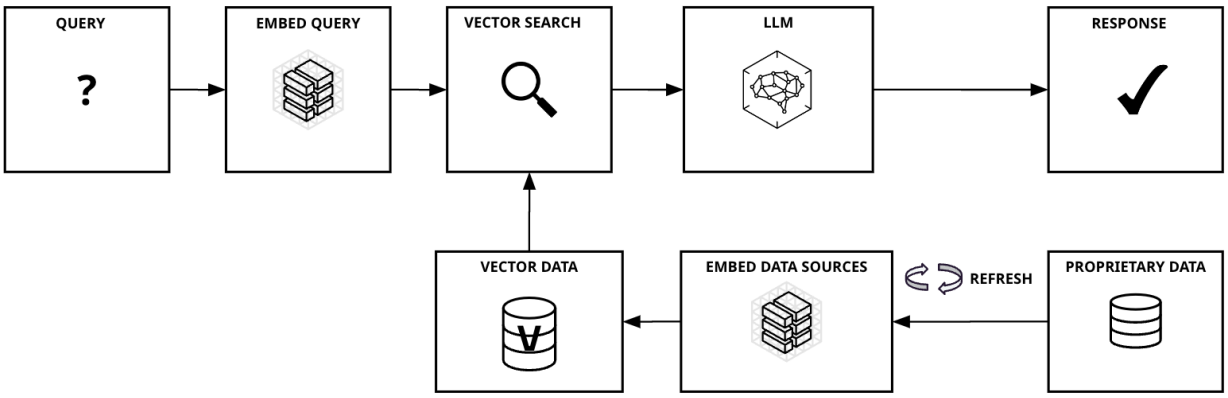
LLMs have some limitations that may prevent generated responses from being useful in your particular application. This generally occurs because the knowledge stored in the LLM is limited to its training data, which does not include recent data or proprietary data that is not generally available on the internet.

Retrieval Augmented Generation (RAG) allows LLMs to integrate specific documents that are relevant to solving the task that the user has asked them to perform.

Figure 3 illustrates how a RAG application works. First, a query is submitted to the application. In generative AI apps the query may be a search engine type query with some keywords such as “george washington history”, an actual question “when was george washington born”, or a set of instructions “write an essay about the life of george washington”. We embed the query with the same embedding model used to embed our proprietary data sources and then we perform a vector search using the query embedding to find the nearest matches. Efficient similarity search is the crucial capability that vector search enables.

The app then takes the text of the top matching embeddings and inserts those into the prompt sent to the LLM to produce a result for the task the query was instructing the LLM to do.

Figure 3. Basic RAG flow with LLMs for an application.



This approach has multiple benefits in generating responses:

- **Reduction of Hallucinations**
  Hallucinations occur when LLMs produce factually incorrect responses to a query. The RAG design pattern tells the LLM to focus on the documents retrieved from the vector search in generating its response. By using a smaller set of documents rather than the complete set of training data in the LLM, the response will more accurately solve the query.

- **Leveraging Proprietary Data**
  The information and documents most relevant to your application have been directly provided to the LLM to tailor the response to your product and use case.

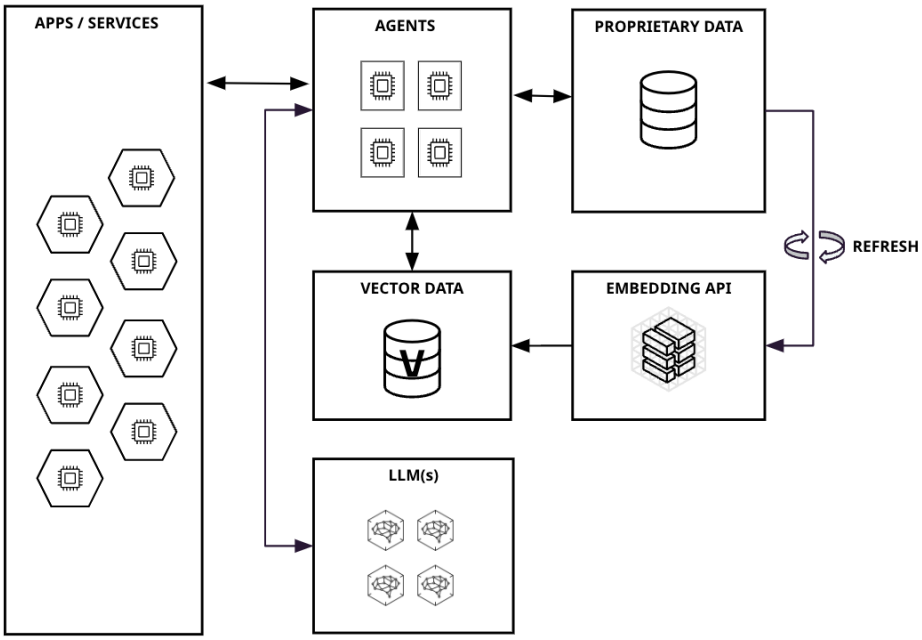
- **Quick "Training" of the LLM**
  The RAG pattern is easy to implement and maintain as a way to tune the model to your particular application. Refreshing the embedding data as new documents are added, existing documents are modified, or new data sources become available allows you to keep the model up-to-date and accurate in its responses.

Figure 3.1 outlines a complete architecture for running applications or services using RAG. In this architecture, we introduce one additional concept, an **agent**. Agents are automated reasoning and decision engines. They are the component of your code or application that takes the input or query from a user and decides what should be done.

In the simple RAG example from Figure 3, the agent encompasses the steps for embedding the user query, running the vector search, and creating the prompt for the LLM including the results from the search.

In more complex applications, agents may break down tasks into a series of simpler sub-tasks, decide if external tools need to be used to solve the problem, or store and maintain memory for the application.

Figure 3.1 Retrieval Augmented Generation: Architecture Context View



Adopting a vector-enabled database for storing the embeddings and executing the vector search provides the benefits of scaling the amount of data available to very large datasets while maintaining fast document retrieval. Further, it enables highly efficient similarity search which is a fundamental requirement for any generative experience.

Vector Database Adoption

A recent survey by Sequoia Capital[10] found that **88% of companies** working with generative AI believe that a document retrieval mechanism is a crucial part of their infrastructure going forward to enable generative AI applications.

Implementing a RAG AI Application

Let's look at the steps in the RAG pattern (Fig. 3) and the architecture context diagram (Fig. 3.1) in more detail to build a simple generative AI app that uses retrieval augmented generation.

There are two separate workflows needed to build a RAG system:

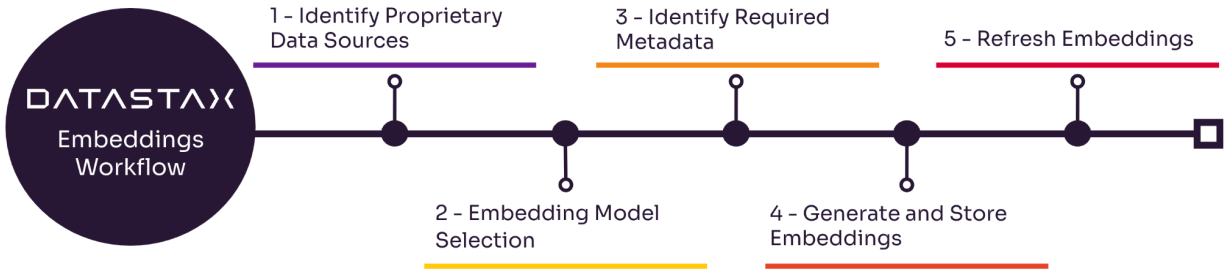
- Generate the knowledge base
- Run the generation process

We'll look at each one of these separately.

Generate the Knowledge Base

First, let's look at building out the knowledge base of embeddings that our RAG application will access. There are several steps to this workflow outlined below in Figure 4.

Figure 4. Workflow for turning proprietary data into a vector store of embeddings.



1. **Identify Proprietary Data Sources**

   This step is dependent on the application you are building. When looking at datasets that you may want to use, some things to consider are: which unique information does this data source provide? Are there publicly available datasets we wish to embed?

   You may wish to include publicly available datasets because they are more up-to-date than the datasets the LLM was originally trained on. As an example, you could embed recent news articles or new developer documentation.

2. **Embedding Model Selection**

   When selecting an embedding model, we do not need to use the same model for embeddings and text generation, or even select the LLM model and embedding models from the same vendor or open-source model provider. It is important, however, that the same embedding model is used for embedding both the query and the dataset for retrieval.

   The Massive Text Embedding Benchmark (MTEB)[21] is a good source to get ideas about embedding model performance when making a choice. Note that if the benchmark is being recomputed the leaderboard will be unavailable until it completes, check back later.

   For getting started quickly, the text-embedding-ada-002[22] model from OpenAI performs well and is accessible with an API call.

   For open source options that you can host yourself, the instructor models[23] (available in 3 sizes

where the 2 smallest can generally be run on a laptop) have good performance and also allow you to tailor the embeddings to your specific use case.

3. Identify Required Metadata

There are a few reasons we will want metadata associated with each vector.

First, after retrieving the top matches via vector search, we'll need to add the text of those embeddings to the prompt we pass to the LLM.

Embeddings are generally done on shorter sections of text than the full document, called chunks. Depending on the use case and information about the underlying dataset, we may also want to store the full text of the document with each chunk. This decision requires some knowledge about the underlying dataset. If the documents are very long and discuss many topics, we should only use the text from the chunk as this is more likely to be related to the query. If we know the longer document is all on a specific topic, there may be additional details in the full document that would be beneficial to the LLM.

Additional metadata that we may need can include a unique identifier that can be recomputed when we check to see if the embeddings need to be updated due to a change in the source document.

Adding metadata that we can filter on to select subsets of our datasets that we wish to perform the vector search over can also be useful. A simple example is having different data sources available, such as "product documentation" and "internal company wiki". There may be cases where the "internal company wiki" contains data that we do not want to share with certain users, so we may want to exclude that source from the search results.

4. Generate and Store Embeddings

Now that we have a model selected for embeddings, we need to generate the vectors.

The first step in this process is *chunking* the text. This is the process of deciding how to break the text up into smaller-length strings. There are two main reasons for doing this.

The first is that embedding models have limits to how much text they can accept as input. In the case of text-ada-002, that limit is 8,192 tokens. In English, 1 token is approximately 4 characters. Other models may have shorter string lengths they can accept.

Beyond the model limitations, there are also considerations about both your data source and application. Is your app returning entire documents to the user? You may want to chunk as large as possible. Does your app depend on a set of facts to power a question-and-answer

system? Shorter embeddings that are around a few sentences in length are more likely to encompass a single fact or idea and will be a better fit for your use case.

Once you've settled on a length for your chunks, you now need to come up with a chunking strategy. There are many ways to do this.

The simplest approach is to break the text up into even-length chunks. Allowing for some overlap of chunks can help with making sure that the concepts in the document are properly captured in case the chunks break a word or sentence in the middle of explaining a key fact.

More advanced strategies often involve chunking based on document or language structure. Examples might include chunking by document section, paragraphs, or in a way that preserves sentence boundaries.

Once you have created the set of chunks for your document, you then add the associated metadata defined in the prior step and store the vector for each chunk along with the metadata in a data store that allows for fast vector search via ANN. AstraDB is a great option for working with large sets of vector data while also enabling additional operations on the metadata stored with the vectors.

5. Refresh Embeddings

Over time the proprietary data sources will change. In order to avoid hallucinations, it is important that embeddings are updated as soon as new source data is available.

Changes in the data source may occur because a new document is available or because there were updates to a piece of content that was previously encoded.

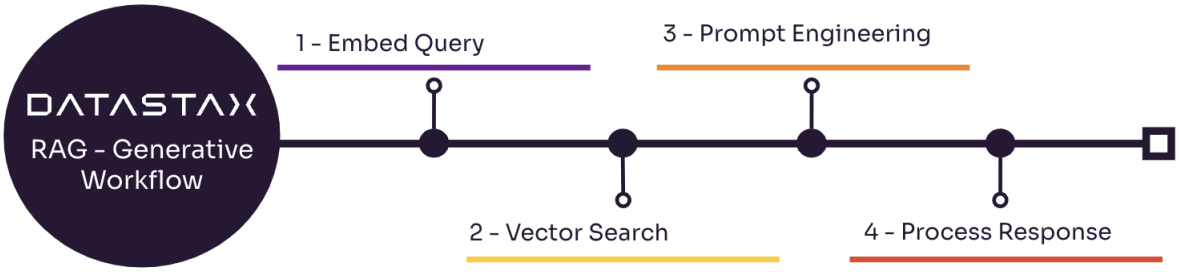
When updating the embeddings for an existing document, remember that for each object in the proprietary data, there may be multiple embeddings. For text data sources this will depend on the length of the full text and the chunking strategy selected. When updating embeddings be careful to replace all embeddings for the document when doing an update.

It is important to remember that not all embedding models are deterministic, meaning that if you encode the same text multiple times you may get different results. The resulting embeddings will always be near each other, but it is not a good idea to rely on the raw values of the embedding vector when replacing existing embeddings with new ones and instead use a unique identifier.

Running Generative Applications

Now let's look more carefully at how you generate the response from the LLM using the datasets we encoded in the previous section.

Figure 5. Workflow for creating text generations using RAG.



1. **Embed Query**

   The text generation in a RAG application starts with a query. The query may come directly from the end user of the application or could be generated by an LLM or some other procedural piece of code.

   We then encode the query by generating an embedding for the text of the query.

   It is important that the embedding model used at this point is the same one that was used to embed the data sets for the vector search, otherwise, the search will not work correctly.

2. **Vector Search**

   Now we need to retrieve the closest embeddings from our vector data store. We first do a nearest neighbor search that will return nearby vectors to the query. Often this search is done by asking for the "top K" matches. Using the top 3 matches is very common, but may be adjusted depending on both the length of the chunks you used in the embeddings, the context window (size of the input allowed to the LLM), and your use case.

   Once these vectors are returned, we might request that the vector search return the score or measure of similarity between the returned vectors and the query vector. If this score is below a threshold, we could optionally remove the returned documents from inclusion in the prompt in later steps due to the matched documents being of low quality. The exact threshold to use is application, data set, and similarity metric dependent. A good spot to start experimenting is with values between 0.5 and 0.8 for a cutoff, assuming the scores are normalized to a

maximum of 1.

We can manually determine a good cutoff for excluding matches by having the vector search return a large number of matches, say 100 to 1,000 along with their scores, and then manually review the text that was used to generate the embedding. We are looking for a point where the returned document text is not relevant to the query.

After we have a set of vectors that matched well with our query, we then need to pull the associated text that was stored as metadata with those vectors to use in the prompt engineering step.

3. Prompt Engineering

Prompt engineering is the design of the instructions that are given to the LLM. Some LLMs take only a single prompt. Others, particularly chat models, may consist of a few different inputs you need to define. We'll discuss each section of the prompt separately below.

The first is context. This tells the model what it is doing, such as "Explain this code" or "you are a helpful customer service bot that answers questions".

You may then provide some examples. This is called few-shot learning and helps the LLM understand what you want it to do. For chat LLMs, you generally have to provide both sides of the conversation in the examples. This involves writing out text examples for both the user and the AI responses.

For RAG applications, you then need to insert the text of the retrieved embeddings that you want to include in the instructions to the LLM. The pattern for this adds an instruction of the form:

> "Use the articles below to complete the task.
>
> **Article:**
> <insert text from the 1st matched vector>
>
> **Article:**
> <insert text from the 2nd matched vector> "

You can continue adding as many articles as you wish to include from the vector search. Note that you can also be more specific in describing the task to be completed as well. For a question/answer bot, you might modify this to, "use facts from the documents below to answer

the question".

You can also provide instructions to the LLM to reason about whether the documents provided by the vector search are relevant to the task you've asked it to undertake, which may improve results.

For example, in a question and answer system, you can modify the initial instruction in the following way: "Use facts from the documents below to answer the question. If the documents do not help you answer the question, please say 'I don't know'".

Finally, you need to make sure there is a clear instruction or task you've asked the LLM to perform.

4. **Process Response**

   After providing the complete instructions including the insertion of text from the vector search process, you'll receive the final text output from the LLM. In most cases, this is where post-processing will take place such as the incorporation of AI safety analysis, quality control parsing, and any additional filtering or adjustment of an LLM response that is required by your use case. Many organizations also use this stage as an opportunity to store a copy of the generated response for collective review at a later time. Once the post-processing of an LLM response is complete, a final response is provided back to the user and the RAG process is complete.

LLM Selection

We'll need to choose an LLM to use for the text generations. If you want to get started quickly with minimal setup, the OpenAI models are a good starting point. They have libraries available[24] for Python, Node.js, as well as the Azure OpenAI libraries. There are also community libraries available for many other languages.

For OpenAI models, gpt-4 is the most powerful and will produce the best results, however, it is in limited beta and may not be available on all accounts. Their gpt-3.5-turbo model is also quite powerful and much cheaper. This model will be a good starting point for many projects if you want to control costs while building.

Both models have 2 different variations with different size context windows. A larger context window allows you to provide more data in the input of the prompt, such as longer documents or more examples of matched documents retrieved from the vector search step. The larger context window models are more expensive per token, and the larger input will also result in more tokens billed, so the cost will go up significantly if you are really taking advantage of the larger context window. You'll need to determine if that extra input space is needed for your application.

If you are a Google Cloud user, their PaLM2 models[5] for text and chat are available in Vertex AI[25]. For open-source options, take a look at the Open LLM Leaderboard[7] on Hugging Face.

Use Cases

Semantic search

Semantic search is often described as a search with meaning or understanding. This is in contrast to lexical or keyword search that tries to exactly match words or phrases.

The power of semantic search can be seen with some simple examples. The phrases “website theme”, “website template”, and “website design”, all refer to the visual presentation of a website, but in a traditional lexical search if we looked for “themes” and a particular product called them “templates”, your search would not return a match. A semantic search for images would place a falcon near a hawk because they are both birds, but these would be far away from an image of a house.

There are many types of problems we can solve with semantic search. A few examples include:

- **Search Engines**
  Perhaps a little obvious, but search engines built with semantic search are very powerful and fast.

  In this case, you simply perform the vector search and return the top K nearest matches, where K is the number of matched documents you wish to return. The nearest vectors are generally found via Approximate Nearest Neighbor (ANN) search.

  To get potentially better results you can also rerank the search results based on other data you have available, such as the traffic to the documents, click-through rates, or other metrics.

- **Summarization and Extraction**
  Instead of presenting users with a long list of documents that they need to read through individually, one can use the output of the vector search as an input to an LLM to summarize or use an LLM to extract a specific piece of information.

- **Recommendations**
  Recommender systems are powerful personalization tools that enable apps to increase engagement with their products by ensuring users are being presented with the content they are most interested in. They become even more powerful and wide-ranging when combined with the power and versatility of generative AI-augmented with your proprietary data.

  There are many ways to implement recommendations, but a very basic approach that uses vector search is:

- Generate item embeddings. These may be articles, videos, photos, or products. Item embeddings capture properties of the content or products that often include attribute properties such as genre or topic for content, and attributes such as color, size, or style for physical products. These could also be configuration parameters, combination rules, common combinations, etc. for products, services, or offerings.
- Generate user embeddings. Each user can be represented with an embedding vector that describes their preferences.
- A set of recommendations can be generated by using vector search to find the item embeddings that are most similar to the user embeddings.

Chat and Customer Support

Chat and Customer Support apps use generative AI to create human-like responses to user questions. Examples of apps that leverage this pattern include use cases such as customer support, virtual tutors, and personal assistants.

It is important to note that this same approach can be used to expose a wide range of additional features and capabilities, such as recommendations based on parameters provided by a user within the query or within elements of their conversational dialog. For example, based on a set of requirements or constraints shared by a user for identifying suitable nursing homes for an aging relative, a RAG-based experience would be able to narrow down potential candidates and provide them in a list to a user for further examination if augmented with the proper embedding data.

These types of experiences are all possible because, unlike traditional Chat and Customer Support systems that rely on predefined answers or structured databases, generative AI models can understand the context and semantics of questions and generate human-like responses in real-time.

Taking this paradigm one step further, it is very simple to use the same underlying data platform to store generative response history from these types of experiences and convert them to embeddings for future use. This becomes far more powerful than storing simple chatbot history as it enables teams with the ability to construct fully searchable, long-term memories of all customer interactions, each of which can be referenced in the context of future conversations.

Chat and Customer Support systems can be implemented with LLMs using the RAG design pattern we covered earlier. At its core, semantic search is performed to find documents that help answer the question that was asked, which is very similar to the semantic search use case above. However, Chat and Customer Support systems have an additional step which is to instruct the LLM to use the results from the semantic search to then write a solution to the query. This combination of capabilities makes these types of experiences far more versatile and useful for customers with non-trivial requests.

LLMs may also be used to generate question-and-answer pairs which can help improve the accuracy of the answers provided by the application. In some cases, this may be accomplished by prompting the LLM to produce pairs of questions and answers from existing documents, expanding existing question and answer pairs in other directions, or generating synthetic data containing more sets of question and answer pairs that cover topic areas not covered in the existing knowledge base.

Classification

Classification problems arise in many use cases. We see them arise frequently in image analysis. Is there a car in this photo? Does this image contain a stoplight?

In text analysis we may see problems with applying a topic taxonomy to a document, to determine if the content is about sports, politics, or entertainment.

Let's look at a few different approaches to classification with LLMs and the benefits and drawbacks of each method. We'll start from the easiest in terms of implementation and work our way up to more complex approaches.

Few Shot Learning for Classification

LLMs can be used directly to classify text using a few shot learning prompt that provides examples of the different labels for the classification.

Let's look at a simple example of classifying positive and negative sentiment in the OpenAI playground using gpt-3.5-turbo to illustrate how this can be done.

We'll use the system in the playground to define the task and provide the few shot learning examples:

System Prompt for Sentiment Classification

You are classifying documents for positive and negative sentiment.

Document:
This movie was horrible.
Sentiment:
Negative.

Document:
I loved the movie I saw this weekend, it was so exciting!
Sentiment:
Positive.

Document:
Worth every penny! It was soooo good!
Sentiment:
Positive.

In the user prompt, we can now provide a new document and ask the LLM to tell us the sentiment.

User Prompt for Sentiment Classification

Document:
I hated that movie, it was so boring.
Sentiment:

The reply from the LLM, in this case, is **Assistant: Negative.**

Fine-Tuning an LLM for Classification

Fine-tuning is an option supported by some LLMs that allows you to provide a large set of input/output pairs that explain to the model what you want it to do. To tailor the LLM's behavior this usually requires providing at least a few hundred examples. Generally, putting so many examples directly into the prompt is not feasible either due to limits on the amount of text you can input or the cost associated with the large number of tokens in the prompt.

Let's consider a case where we might need to fine-tune versus using a few shot prompt. Consider a classification problem that labels whether a specific Stripe product was discussed in a text. Stripe has products such as "Payments", "Checkouts", and "Billing" which also have common vernacular uses that an LLM may have trouble distinguishing.

Consider the phrase "My stripe payments failed."

The user might be saying "I set up stripe payments and it didn't work" or they might be trying to articulate "I tried to pay my stripe invoice and it failed."

In this simple example, it's hard to say if we should classify the original phrase as a discussion about the product "Stripe Payments" or not.

Providing a much larger set of examples, with several positive instances, where the document is discussing the Stripe payments product along with several negative examples where the word payments occurred near the word Stripe, but the word "payments" was being used in its vernacular form and not as a description of their product will help the LLM distinguish between these two cases.

Results of this approach:

- **Pro** - improved accuracy of the classifier
- **Con** - need to maintain a set of at least a few hundred training examples
- **Con** - fine-tuned models from cloud vendors are more expensive than standard LLMs

Classification With Embeddings

A third approach to classification is to use embeddings to train a traditional classifier model, such as logistic regression, random forest, or support vector machine.

The process for this approach requires building a set of training data where you have many embeddings for each label in your classifier. You can then use a standard library such as Scikit-Learn or LightGBM to train the classifier.

Since this approach is based on embeddings as well as a large training dataset, we expect that it would perform very well at complex classification problems for text data.

Results of this approach:

- **Pro** - high accuracy for the classifier (though not necessarily outperforming the LLM approaches)
- **Pro** - cost per applied label is very low. LLMs from vendors charge by token processed. If you have long documents, a very large set of documents that need to be classified, or both, the cost of running an LLM classifier may be very expensive.
- **Con** - the time and effort to build and maintain the training data set are very high

LLM Caching

Calling a LLM is frequently expensive and slow. In many scenarios - e.g. enterprise search - a majority of questions or search queries are semantically the same, but often have slightly different ways of asking for the same question. We can use Vector Search to provide a semantic-level understanding of the queries and look up a result form the cache accordingly.

A simple implementation of LLM caching takes the following steps:

1. For each query submitted to the LLM, generate an embedding.
2. Store the response of the LLM to the query.
3. Store both embeddings in a vector datastore.
4. When future calls to the LLM are made, use vector search to see if a similar query has previously been run and cached. Return the cached result.

This technique can greatly decrease the latency and cost of running LLMs in production applications.

However, one downside is you have to be very careful to ensure private information is not stored in the cached responses and then returned to the end user.

Anomaly Detection With Embeddings

Anomaly detection refers to the process of identifying patterns or data points that deviate significantly from the expected or normal behavior within a given dataset. It involves detecting outliers, unusual events, or rare occurrences that do not conform to typical patterns.

A simple process using vector search for density-based anomaly detection is to:

- Create a set of embeddings for your application data.
- For a new document or image, generate the embedding for the new content.
- Find the nearest neighbor to the new embedding.
- Check if the distance is above a threshold indicating a far distance from normal data.
- Label the new embedding as an anomaly.

References

1. “Economic potential of generative AI.” McKinsey, 14 June 2023, https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/the-economic-potential-of-generative-ai-the-next-productivity-frontier#key-insights
2. “Microsoft and OpenAI extend partnership - The Official Microsoft Blog.” The Official Microsoft Blog, 23 January 2023, https://blogs.microsoft.com/blog/2023/01/23/microsoftandopenaiextendpartnership/
3. Hawk, Jessica. “Build next-generation, AI-powered applications on Microsoft Azure | Azure Blog.” Microsoft Azure, 23 May 2023, https://azure.microsoft.com/en-us/blog/build-next-generation-ai-powered-applications-on-microsoft-azure/
4. “Anthropic Partners with Google Cloud.” Anthropic, 3 February 2023, https://www.anthropic.com/index/anthropic-partners-with-google-cloud
5. “Google AI PaLM 2 – Google AI.” Google AI, https://ai.google/discover/palm2/
6. Try Bard, an AI experiment by Google, https://bard.google.com/
7. “Open LLM Leaderboard - a Hugging Face Space by HuggingFaceH4.” Hugging Face, https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard
8. Colmer, Paul. “Get started with generative AI on AWS using Amazon SageMaker JumpStart | Amazon Web Services.” Amazon AWS, 4 May 2023, https://aws.amazon.com/blogs/machine-learning/get-started-with-generative-ai-on-aws-using-amazon-sagemaker-jumpstart/
9. “Generative AI.” Google Cloud, https://cloud.google.com/ai/generative-ai
10. Fradin, Michelle, and Lauren Reeder. “The New Language Model Stack.” Sequoia Capital, 14 June 2023, https://www.sequoiacap.com/article/llm-stack-perspective/
11. https://langchain-langchain.vercel.app/docs/get_started/introduction.html
12. https://gpt-index.readthedocs.io/en/latest/
13. Wodecki, Ben. “UBS: ChatGPT is the Fastest Growing App of All Time.” AI Business, https://aibusiness.com/nlp/ubs-chatgpt-is-the-fastest-growing-app-of-all-time
14. https://roihacks.com/
15. “Survey reveals AI’s impact on the developer experience.” The GitHub Blog, 13 June 2023, https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/
16. “90% of VC-Backed Companies Plan to Launch Generative AI in their Products, 64% this Year.” Productboard, 6 June 2023, https://www.productboard.com/blog/generative-ai-and-products/
17. “KPMG Generative AI Survey.” KPMG U.S., https://info.kpmg.us/news-perspectives/technology-innovation/kpmg-generative-ai-2023.html
18. “Most Searched Thing on Google: Top Google Searches in 2023.” Semrush, 13 June 2023, https://www.semrush.com/blog/most-searched-keywords-google/
19. “AutoML Vision documentation.” Google Cloud, https://cloud.google.com/vision/automl/docs
20. Karpathy, Andrej. “Andrej Karpathy on Twitter: "The hottest new programming language is English."” Twitter, 24 January 2023, https://twitter.com/karpathy/status/1617979122625712128

21. “MTEB Leaderboard - a Hugging Face Space by mteb.” Hugging Face, https://huggingface.co/spaces/mteb/leaderboard
22. https://platform.openai.com/docs/guides/embeddings/types-of-embedding-models
23. Instructor Text Embedding, https://instructor-embedding.github.io/
24. “Libraries - OpenAI API.” Platform OpenAI, https://platform.openai.com/docs/libraries
25. “Vertex AI.” Google Cloud, https://cloud.google.com/vertex-ai