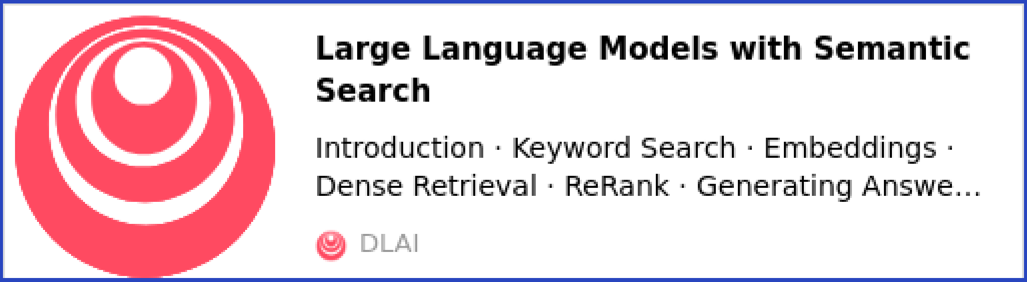
DeepLearningAI

Large Language Models with Semantic Search

Prepared by
Martyna Kopyta

Note

These notes are based on transcripts from the course and contain only theory realted to the topic, so there are no references to the code in this document. You can find the full course in the link provided here:



Large Language Models with Semantic Search

Introduction · Keyword Search · Embeddings · Dense Retrieval · ReRank · Generating Answe...

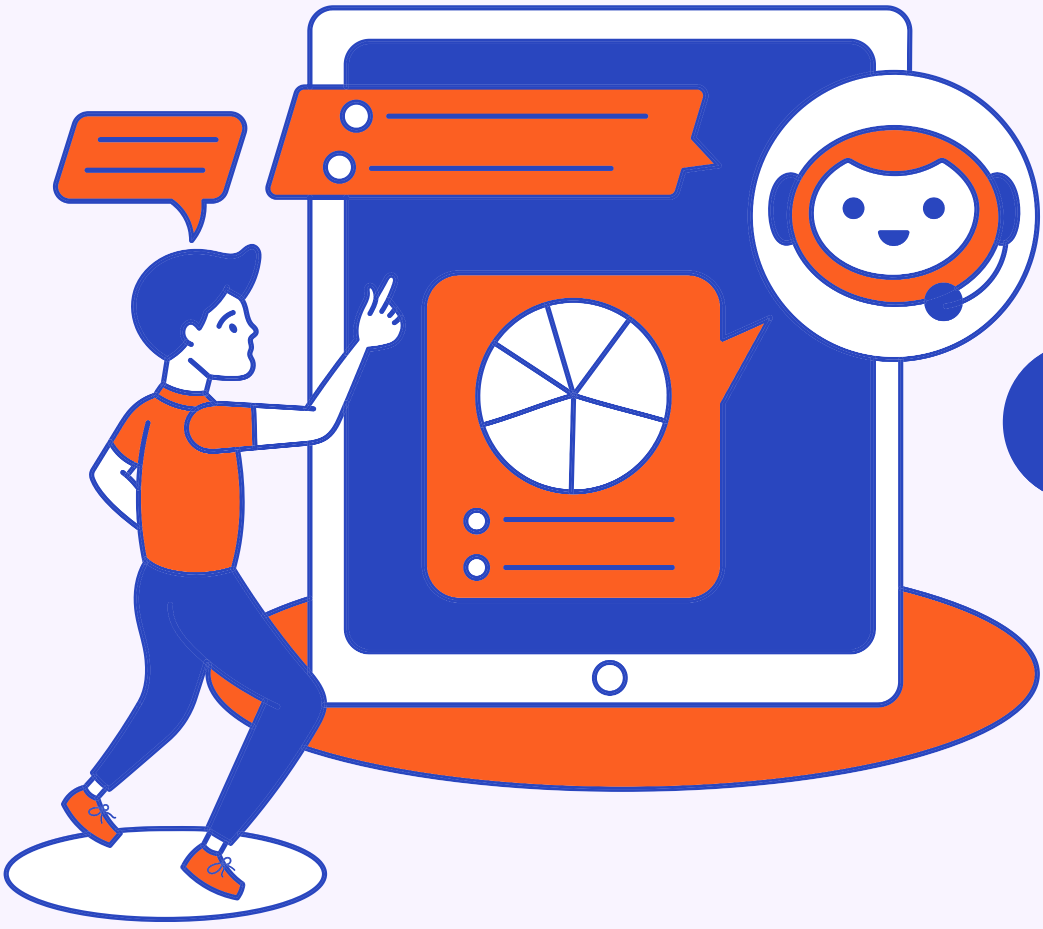
DLAI

Go to page

Introduction

In this course, you'll learn how to incorporate large language models, or LLMs, into information search in your own applications.

For example, let's say you run a website with a lot of articles, picture Wikipedia for the sake of argument, or a website with a lot of e-commerce products. Even before LLMs, it was common to have keyword search to let people maybe search your site. But with LLMs, you can now do much more. First, you can let users ask questions that your system then searches your site or database to answer. Second, the LLM is also making the retrieve results more relevant to the meaning or the semantics of what the user is asking about.



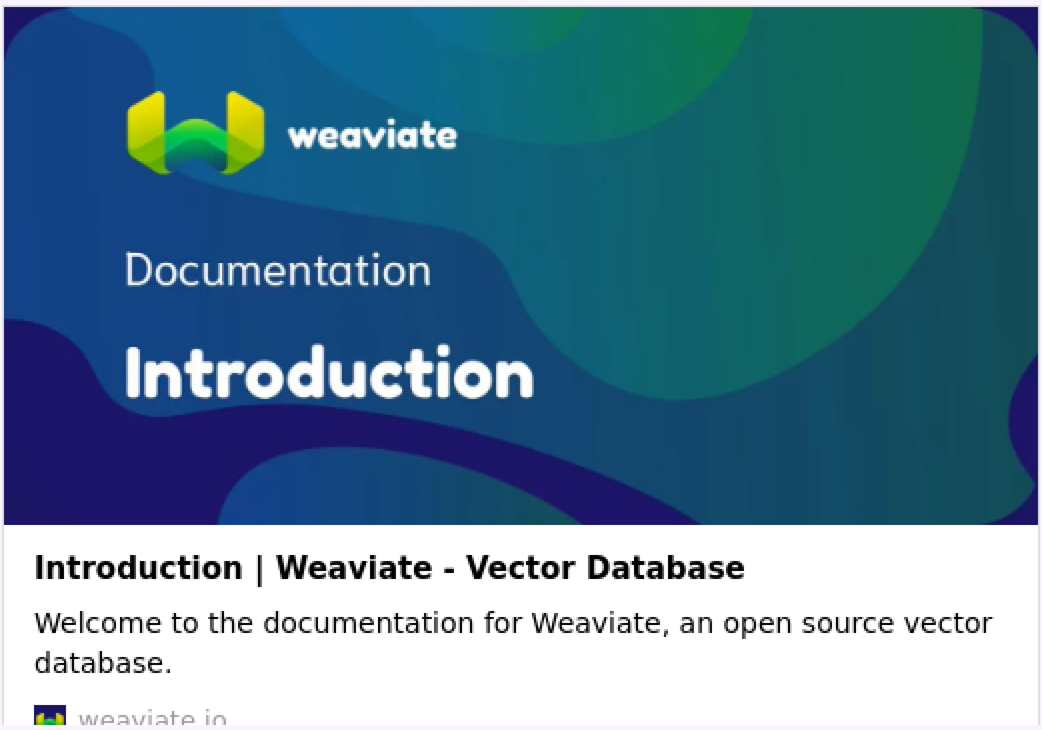
This course consists of the following topics

- First, it shows you how to use basic keyword search, which is also called lexical search, which powered a lot of search systems before LLMs. It consists of finding the documents that has the highest amount of matching words with the query.
- Then you learn how to enhance this type of keyword search with a method called re-rank. As the name suggests, this then ranks the responses by relevance with the query.
- After this, you learn a more advanced method of search, which has vastly improved the results of keyword search, as it tries to use the actual meaning or the actual semantic meaning of the text with which to carry out the search. This method is called dense retrieval. This uses a tool in natural language processing called embeddings, which is a way to associate a vector of numbers with every piece of text.
- Semantic search consists of finding the closest documents to the query in the space of embeddings. Similar to other models, search algorithms need to be properly evaluated. You also learn effective ways to do this.
- Finally, since LLMs can be used to generate answers, you also learn how to plug in the search results into an LLM and have it generate an answer based on them. Dense retrieval with embeddings vastly improves the question answering capabilities of an LLM as it first searches for and retrieves the relevant documents and it creates an answer from this retrieved information.

Keyword search

Keyword search is the most common method to build search systems.It forms the foundation for search systems, providing a basis for further improvements.Its applications extend to search engines, apps (e.g., Spotify, YouTube, Google Maps), and internal document searches for companies and organizations.

Weaviate is an open-source database with keyword and vector search capabilities that rely on language models that is used throughout this tutorial. The API key we'll be using in the classroom is public, as it is part of a public demo, so the actual key is not a secret and you can use it and its encouraged you use it to access this demo database.



Introduction | Weaviate - Vector Database

Welcome to the documentation for Weaviate, an open source vector database.

weaviate.io

Go to page

A high-level look at how keyword search works is to compare how many words are in common between the query and the documents. So if we compare how many words are in common between the query and the first sentence, they only share the word is. And so that's one word they have in common. And we can see the counts of every sentence in this archive. We can see that the second sentence has the most words in common with the query, and so keyword search might retrieve that as the answer.

How does it work?

Query

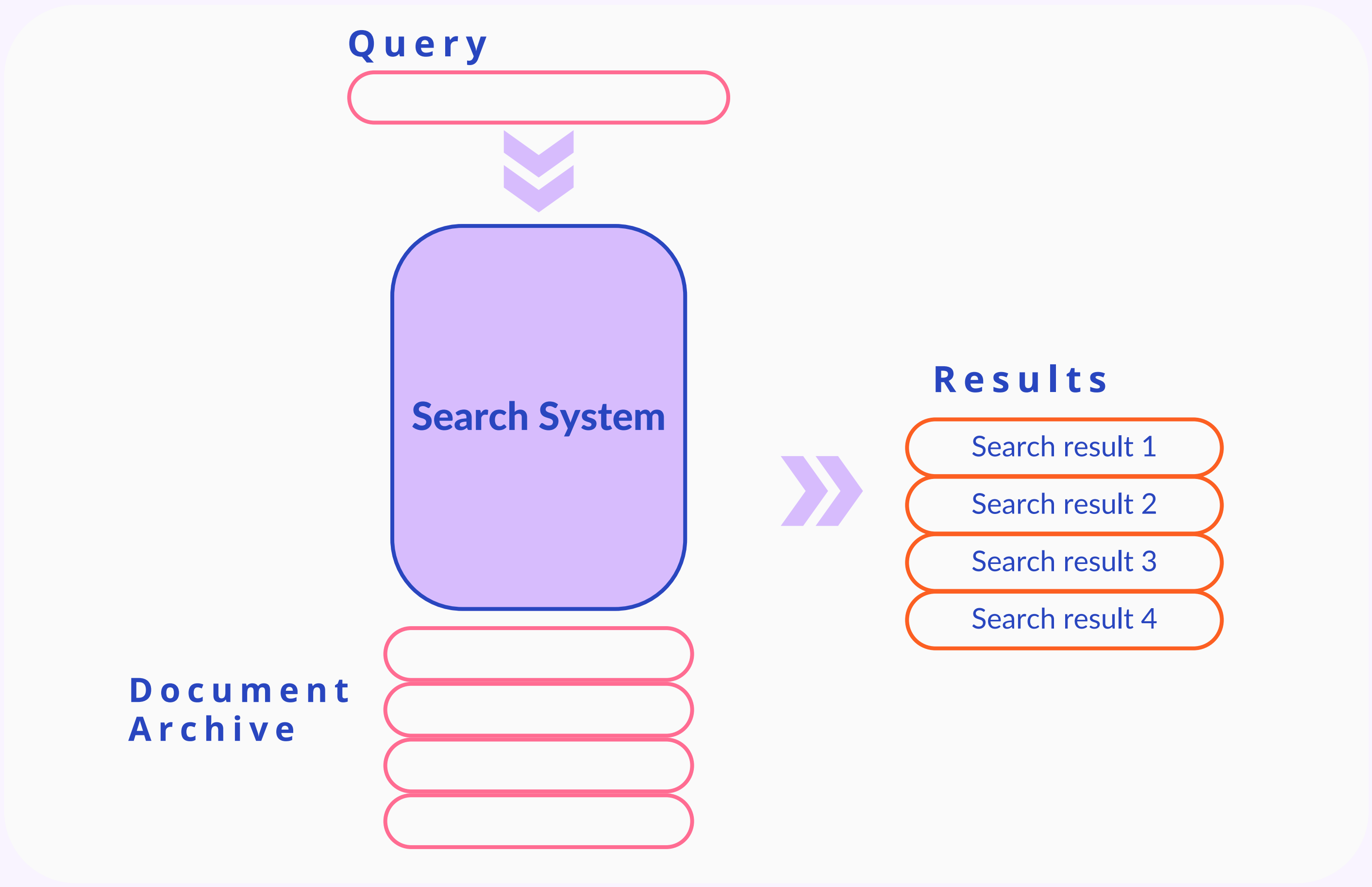
What color is the grass?

Responses	Number of words in common
Tomorrow **is** Saturday	1
The grass is green	3
The capital of Canada **is** Ottawa	2
The sky **is** blue	2
A whale **is** a mammal	1

BM25 algorithm is a commonly used method for keyword search, scoring documents based on shared words with the query.

BM25 only needs one word to be shared for it to score that as somewhat relevant. And the more words the query and the document share, the more it's repeated in the document, the higher the score is. But we can see in general, while these results are returned, this is maybe not the best, most relevant answer to this question or document that is most relevant to this query.

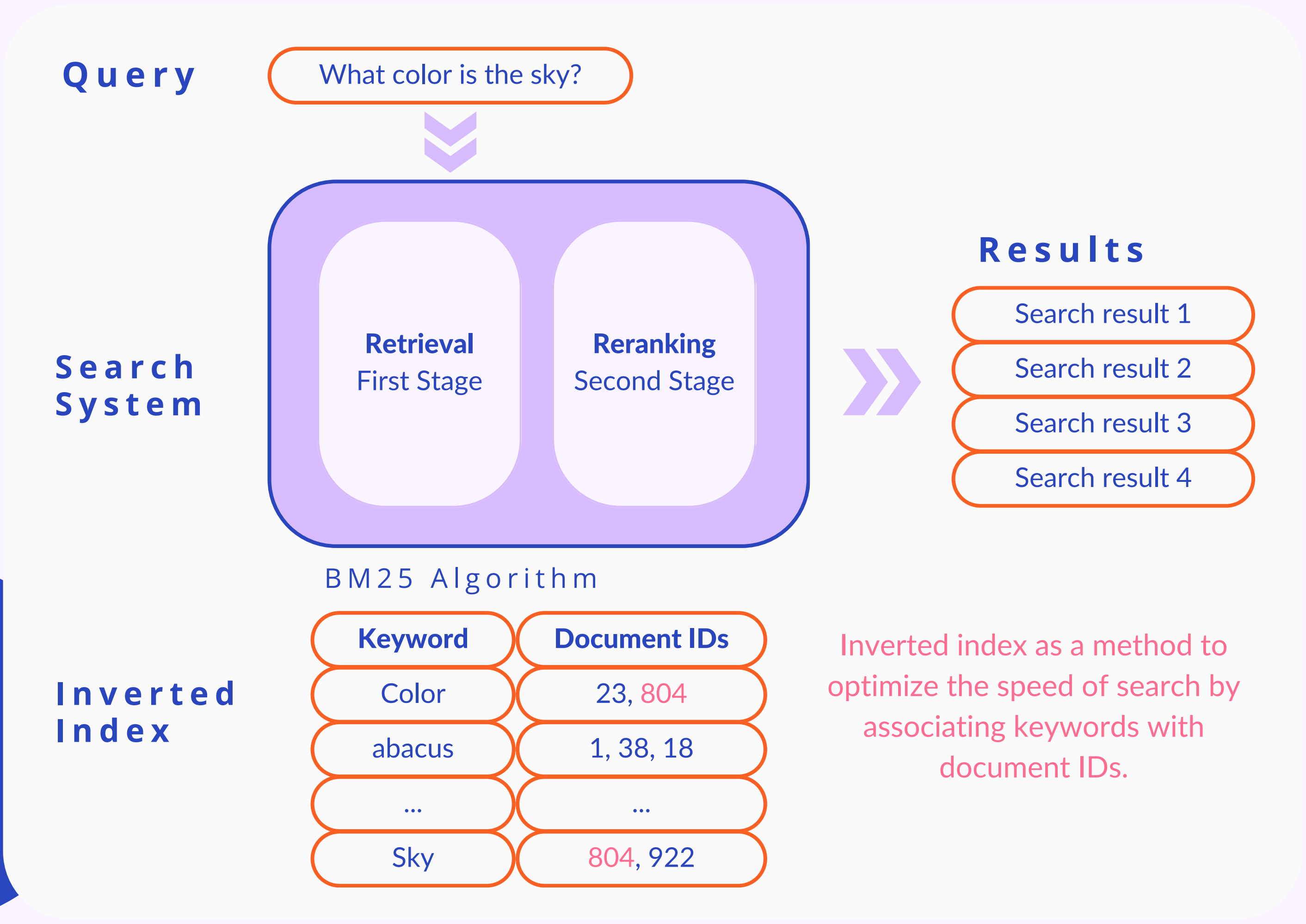
Search at high level



We'll see how language models help with this. Let's look back at search at a high level. The major components are the query, the search system, the search system has access to a document archive that it processed beforehand, and then in response to the query the system gives us a list of results ordered by the most relevant to the query.

If we look a little bit more closely, we can think of search systems as having multiple stages. The first stage is often a retrieval or a search stage, and there's another stage after it called re-ranking.

Keyword Search Internals



Keyword	Document IDs
Color	23, 804
abacus	1, 38, 18
...	...
Sky	804, 922

Inverted index as a method to optimize the speed of search by associating keywords with document IDs.

Re-ranking is often needed because we want to involve or include additional signals rather than just text relevance. The first stage, the retrieval, commonly uses the BM25 algorithm to score the documents in the archive versus the query. The implementation of the first stage retrieval often contains this idea of an inverted index. Notice that this table (on the next schema) is a little bit different than the table we showed you before of the documents.

The inverted index is this kind of table that has kind of these two columns. One is the keyword, and then next to the keyword is the documents that this keyword is present in. This is done to optimize the speed of the search. When you enter a query into a search engine, you expect the results in a few milliseconds. This is how it's done.

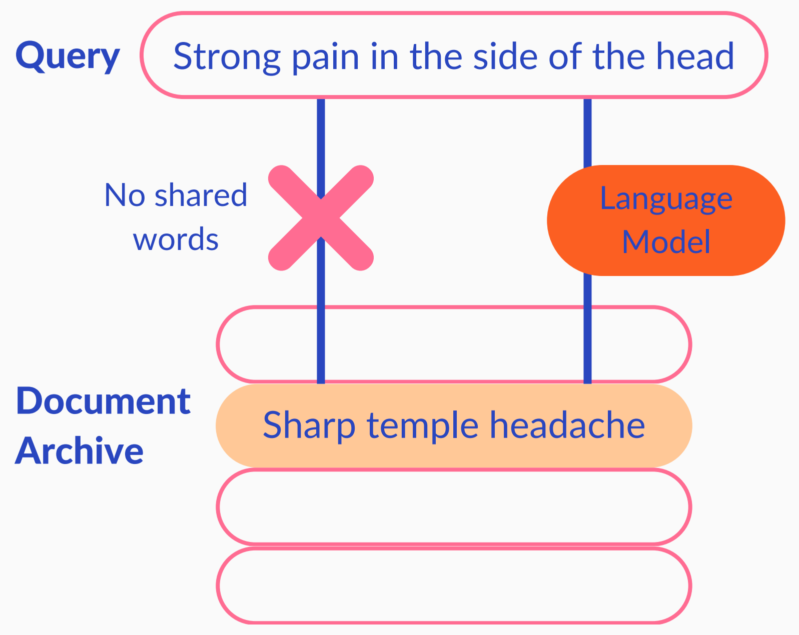
In practice, in addition to the document ID, the frequency of how many times this keyword appears is also added to this call. With this, you now have a good high-level overview of keyword search. Now for this query, what color is the sky, let’s look at the inverted index.

Inverted Index

Keyword	Document IDs
Color	23, 804
abacus	1, 38, 18
...	...
Sky	804, 922

The word color has the document 804, and the word sky also has the document 804. So 804 will be highly rated from the results that are retrieved in the first stage. From our understanding of keyword search, we can see some of the limitations.

Limitation of keyword matching

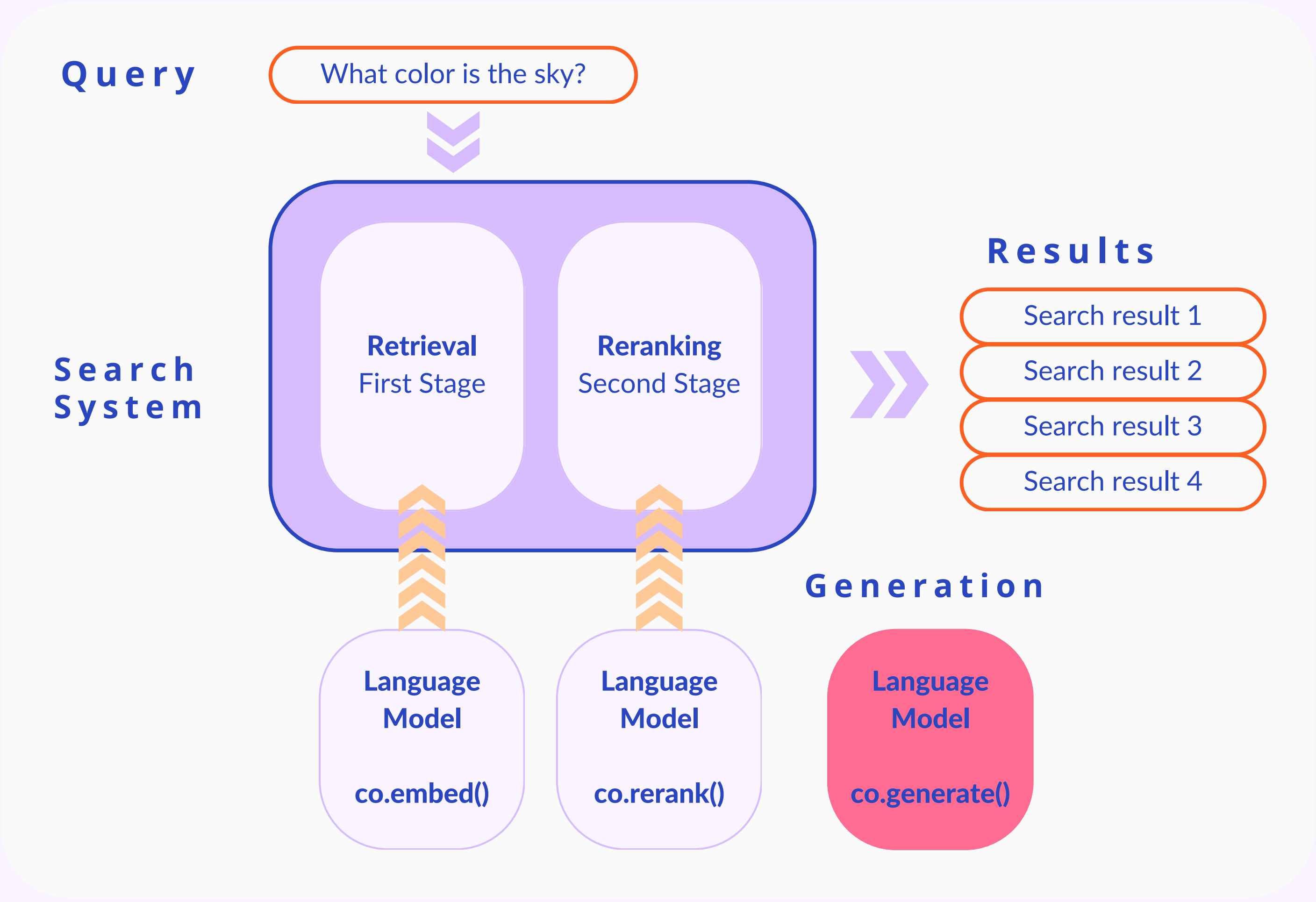


When documents use different keywords to convey similar meanings keyword matching may not work as intended. Language models can be a solution to overcome these limitations.

So, let's say we have this query, strong pain in the side of the head. If we search a document archive that has this other document that answers it exactly, but it uses different keywords, so it doesn't use them exactly, it says sharp temple headache, keyword search is not going to be able to retrieve this document.

This is an area where language models can help, because they're not comparing keywords simply.

Language Models Can Improve both Search Stages



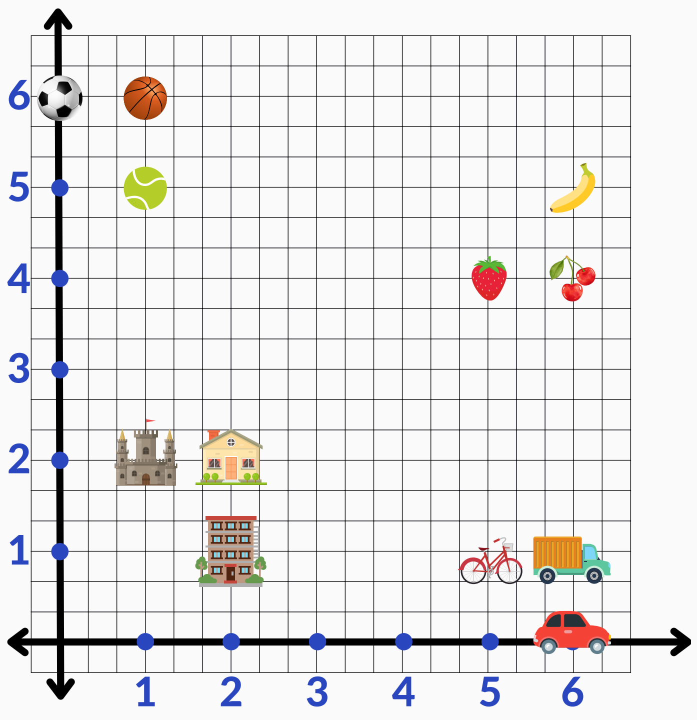
LLMs can look at the general meaning and they're able to retrieve a document like this for a query like this. Language models can improve both search stages, we'll look at how to do that. We'll look at how language models can improve the retrieval or first stage using embeddings, which are the topic of the next lesson. And then we'll look at how re-ranking works and how it can improve the second stage. And at the end of this course, we'll look at how large language models can generate responses as informed by a search step that happens beforehand.

Embeddings

Embeddings are **numerical representations of text that computers can more easily process**. This makes them one of the most important components of large language models.

Embeddings

Given the locations of these words, where would you put the word apple?



Word	Numbers	
Apple	?	?
Banana	6	5
Strawberry	5	4
Cherry	6	4
Soccer	0	6
Basketball	1	6
Tennis	1	5
Castle	1	2
House	2	2
Building	2	1
Bicycle	5	1
Truck	6	1
Car	6	0

As you can see in this embedding, similar words are grouped together. So in the top left you have sports, in the bottom left you have houses and buildings and castles, in the bottom right you have vehicles like bikes and cars, and in the top right you have fruits. So the apple would go among the fruits. Then the coordinates for Apple here are 5'5 because we are associating each word in the table in the right to two numbers, the horizontal and the vertical coordinate. **This is an embedding.**

Now the embedding from the exmple on the previous page, sends each word to two numbers. In general, embeddings would send words to a lot more numbers and we would have all the possible words. Embeddings that we use in practice could send text to hundreds of different numbers or even thousands. If some words/sentences/... are similar, the embedding sends them to numbers that are really close to each other.

Word	Numbers			
A	-0,82	-0,32	...	0,23
Aardvark	0,42	1,28	...	-0,06
...	...	...	...	...
Zygote	-0,74	-1,02	...	1,35

$\longleftrightarrow$ **Hundreds**

Dense Retrieval

We have a query (next page), “What is the capital of Canada”, and we have five possible responses or sentences in our archive. We can plot these just like we did with the embeddings, and we can see that sentences that are similar in meaning will be close to each other in the plot. So, if we plot all of these five, we can see that the sentences about the capitals of Canada and France are close to each other, and then the sentences about the colors are close to each other at the top right there.

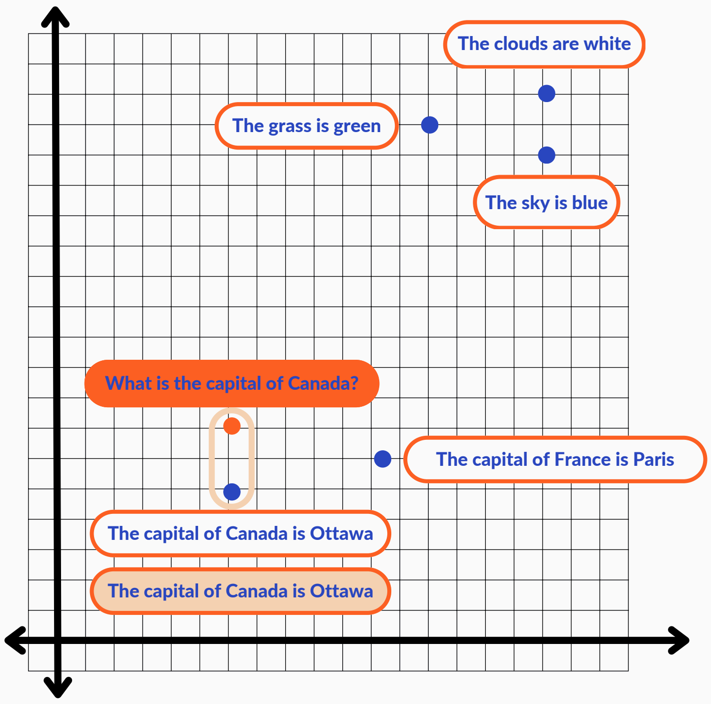
Dense Retrieval

Query

What is the capital of Canada?

Responses

- The capital of Canada is Ottawa
- The capital of France is Paris
- The sky is blue
- The clouds are white
- The grass is green



The clouds are white
The grass is green
The sky is blue
What is the capital of Canada?
The capital of France is Paris
The capital of Canada is Ottawa
The capital of Canada is Ottawa

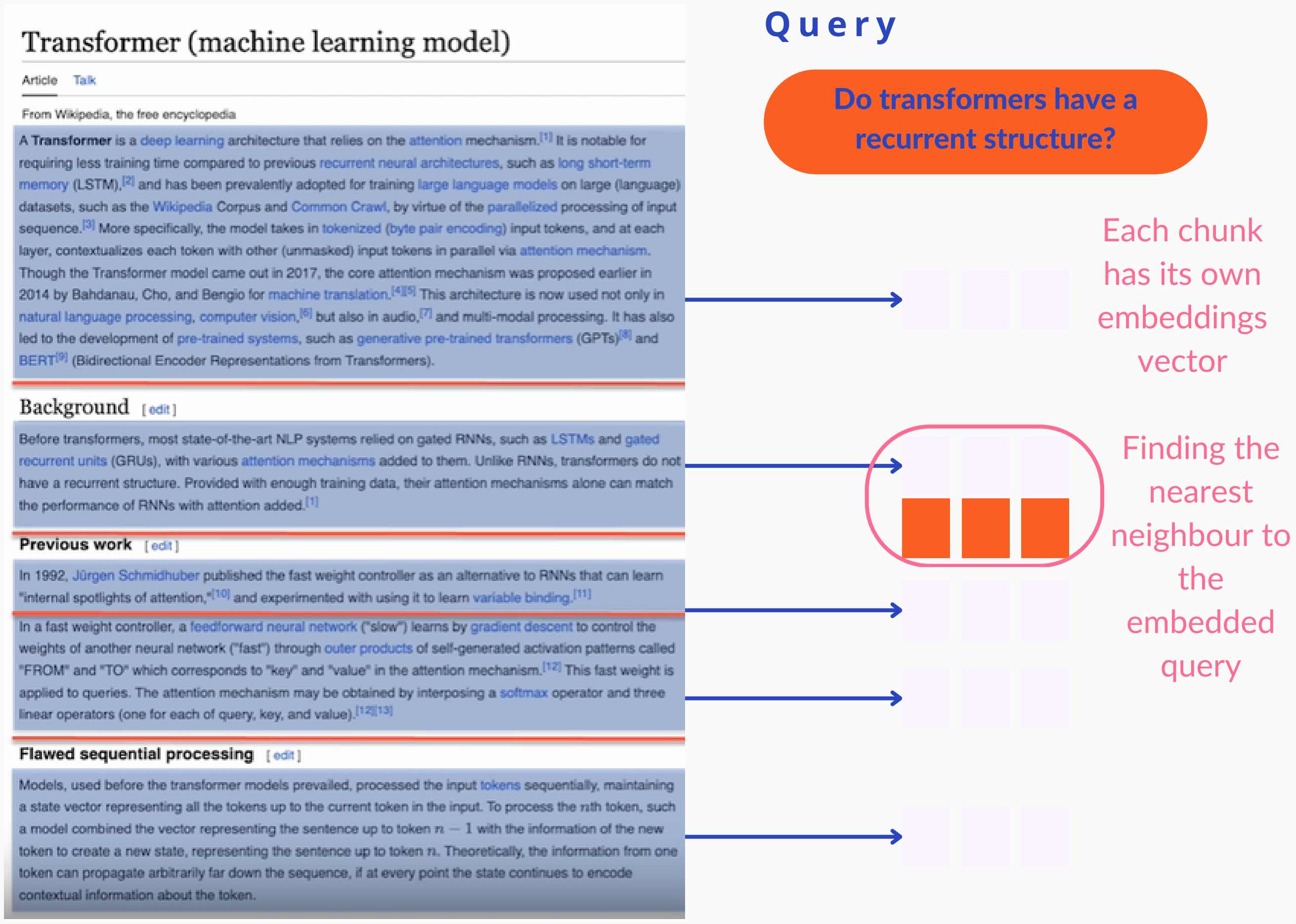
Now, where would this query be if we project it into the same embedding space? If we use an embeddings model that is optimized for search, it will be closest to the answer of that query. And so, when we ask what is the capital of Canada, it will be closest to this point, this sentence that says the capital of Canada is Ottawa. And that is how we use what we learned in embeddings, these properties of similarity and distance into search. And **that is dense retrieval**, one of the two main ways of doing semantic search, the other one being re-rankers.

DenseRetrieval really is able to **capture the intent of the query and return the best result** for it. Another thing to showcase here where Dense Retrieval really helps with are multilingual casees. So, this is how you can support search in multiple languages

In orderd to use dense retrieval, we can use an existing data base or create our own. Let’s see how we can build our own vector database.

We copied over some text from Wikipedia and came into one of the more important questions around semantic search. **We need to break this text down into chunks**. But how big are these chunks? Do we split it into sentences? Or do we split it into paragraphs? Or what is the best way to chunk it? And this is a question that is commonly asked, **The answer is different for every task**. A couple of the common ways is to split, let's say, at every dot.

Chunking



Query
Do transformers have a recurrent structure?
Each chunk has its own embeddings vector
Finding the nearest neighbour to the embedded query

Transformer (machine learning model)

Article Talk

From Wikipedia, the free encyclopedia

A **Transformer** is a deep learning architecture that relies on the attention mechanism.[1] It is notable for requiring less training time compared to previous recurrent neural architectures, such as long short-term memory (LSTM),[2] and has been prevalently adopted for training large language models on large (language) datasets, such as the Wikipedia Corpus and Common Crawl, by virtue of the parallelized processing of input sequence.[3] More specifically, the model takes in tokenized (byte pair encoding) input tokens, and at each layer, contextualizes each token with other (unmasked) input tokens in parallel via attention mechanism. Though the Transformer model came out in 2017, the core attention mechanism was proposed earlier in 2014 by Bahdanau, Cho, and Bengio for machine translation.[4][5] This architecture is now used not only in natural language processing, computer vision,[6] but also in audio,[7] and multi-modal processing. It has also led to the development of pre-trained systems, such as generative pre-trained transformers (GPTs)[8] and BERT[9] (Bidirectional Encoder Representations from Transformers).

Background [edit]

Before transformers, most state-of-the-art NLP systems relied on gated RNNs, such as LSTMs and gated recurrent units (GRUs), with various attention mechanisms added to them. Unlike RNNs, transformers do not have a recurrent structure. Provided with enough training data, their attention mechanisms alone can match the performance of RNNs with attention added.[1]

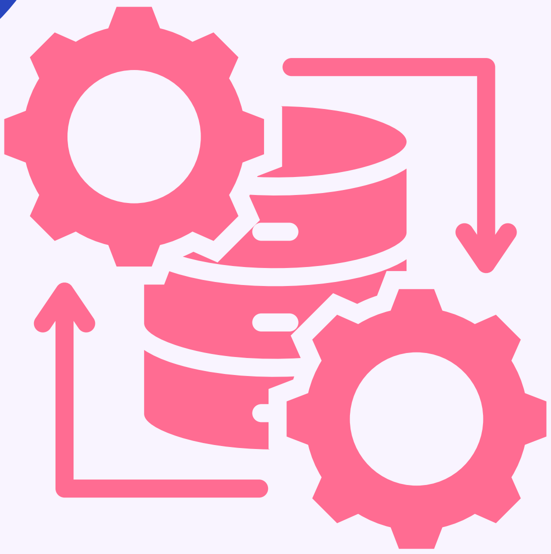
Previous work [edit]

In 1992, Jürgen Schmidhuber published the fast weight controller as an alternative to RNNs that can learn "internal spotlights of attention,"[10] and experimented with using it to learn variable binding.[11]

In a fast weight controller, a feedforward neural network ("slow") learns by gradient descent to control the weights of another neural network ("fast") through outer products of self-generated activation patterns called "FROM" and "TO" which corresponds to "key" and "value" in the attention mechanism.[12] This fast weight is applied to queries. The attention mechanism may be obtained by interposing a softmax operator and three linear operators (one for each of query, key, and value).[12][13]

Flawed sequential processing [edit]

Models, used before the transformer models prevailed, processed the input tokens sequentially, maintaining a state vector representing all the tokens up to the current token in the input. To process the nth token, such a model combined the vector representing the sentence up to token $n-1$ with the information of the new token to create a new state, representing the sentence up to token n. Theoretically, the information from one token can propagate arbitrarily far down the sequence, if at every point the state continues to encode contextual information about the token.



In real life, if you're dealing with a lot of noisy text, you might need to process it a little bit beforehand. And then, you might also be needing to use libraries that actually split at various sentences in more complicated or advanced ways than just splitting at the period.

To **add the context to the chunks**, before we calculate embeddings, one thing we can do is copy the title and then just append it to the beginning of each of these chunks. We know that somethig about the context of chunks. This is a heuristic we've done here for Wikipedia, but then with every database or data set that you have to break or chunk, you need to think about these design choices

The main library we use to find the approximate nearest neighbour is **Annoy**, which is a nearest neighbor library. It's kind of similar to Weaviate, but it's a little simpler. We'll talk about the difference between nearest neighbor libraries and vector databases

Dense retrieval really works by finding the nearest neighbors to a search query. Now, to make it optimized for speed, we actually search for approximate nearest neighbors. So, finding the exact nearest neighbor is taxing in computation, but we have a lot of really highly optimized algorithms that can get you the approximate nearest neighbor, and that's why they're called ANN or approximate nearest neighbors. We talked about the Annoy library, but there are a bunch of other ones that are also open source.

ANN vector search vs Vector databases

Approximate Nearest-Neighbor Vector search libraries

- Annoy
- FAISS
- ScaNN

- Easy to set up
- Store vectors only

Vector databases

Weaviate PostgreSQL

Qdrant vespa

Pinecone chroma

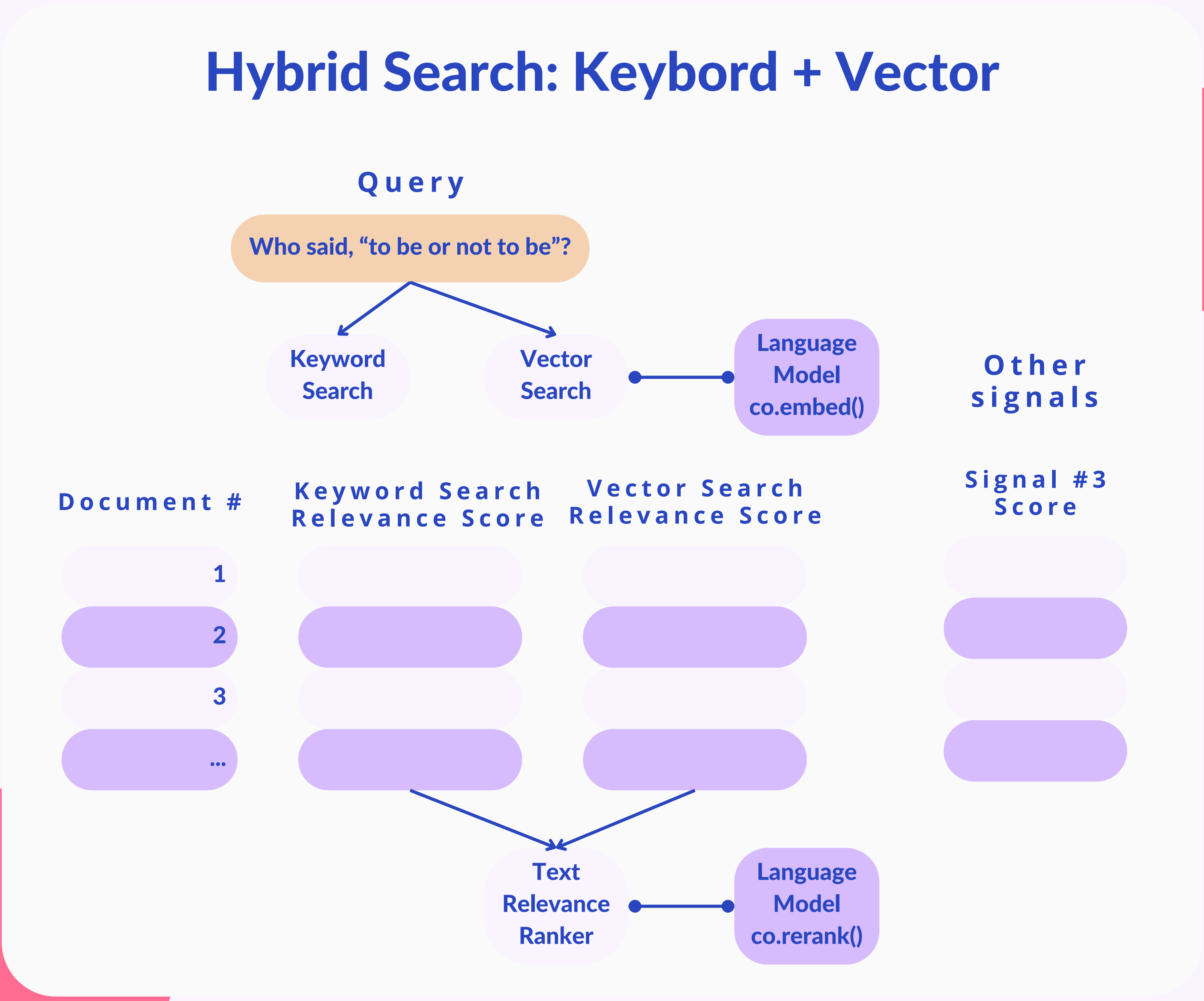
- Store vectors and text
- Easier to update (add new records)
- Allow filtering and more advanced queries

On the other hand, the vector databases, there's a lot more variety there and options. Weaviate is one of them. There are a couple of other ones that are also online services like Pinecone and a couple more open-source ones. Now, a common question is to wonder what is the difference between these two.

The ANN vector search library is usually simpler to set up. So, you saw how Annoy setting it up and installing it is maybe a little easier than the other choices. These vector search libraries also tend to only store the vectors, so they don't store the text. With the Weaviate example, you saw that we gave it a search query. It returned to us the text, so it manages that. And it stores the text, and it is able to retrieve the text that we want. So, it does a little bit more. It's a little bit more feature-rich.

Another major difference between these libraries and the vector databases is that the databases are easier to update. So, if you want to add new records or modify records, if you're using a library you need to rebuild your index, while if you're using a database, the database handles that for you. Vector databases also allow us to filter and do more advanced queries, kind of like what we saw by filtering by the language. These are all useful things that vector databases allow us to do.

In the real world you don't need to really replace keyword search completely with vector search. They complement each other.


Hybrid Search: Keybord + Vector
Query
Who said, "to be or not to be"?
Keyword Search
Vector Search
Language Model co.embed()
Other signals
Document #
Keyword Search Relevance Score
Vector Search Relevance Score
Signal #3 Score
1
2
3
...
Text Relevance Ranker
Language Model co.rerank()

You can use them both together in the same pipeline as **hybrid search**. What that entails is that when you get a query you do keyword search and vector search on it in the same time and you can see how a language model is going to power the vector search component.

Both these search components would then give a score, assign a score to each document in the text archive and then we can aggregate these scores and present the best results.

Do you see this other signals on the right? This is where you can inject **other signals** into the search results. Google, for example, has its famous PageRank algorithm that assigns an authority score to various web pages and websites based on how many other pages link to them. So, you can treat that as an additional signal.

Search engines have tens, hundreds, sometimes even thousands of signals that feed into the decision of how they order the search results. This is just how you can encompass text relevance with other signals by this aggregation step.

If you want to learn more about the history of dense retrieval in the last few years, this is a really good resource - a book and a paper called: Pre-trained Transformers for Text Ranking BERT and Beyond.

ReRank

Rerank is a way for a large language model to sort search results from best to worst based on the relevance they have with respect to the query.

The query says, what is the capital of Canada? And let's say that the possible responses are these five. The capital of Canada is Ottawa, which is correct. Toronto is in Canada, which is correct, but irrelevant to the question. The capital of France is Paris, which is also correct, but not the answer to the question. Then, a wrong sentence. The capital of Canada is Sydney, which is not correct. And then, a sentence says the capital of Ontario is Toronto, which is true, but also not answering the questions.

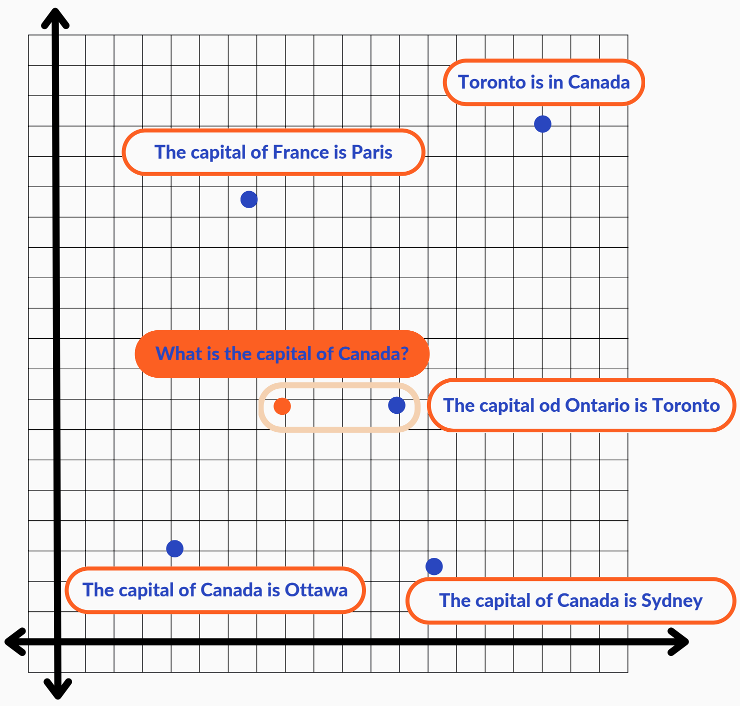
Dense Retrieval is also not perfect

Query

What is the capital of Canada?

Responses

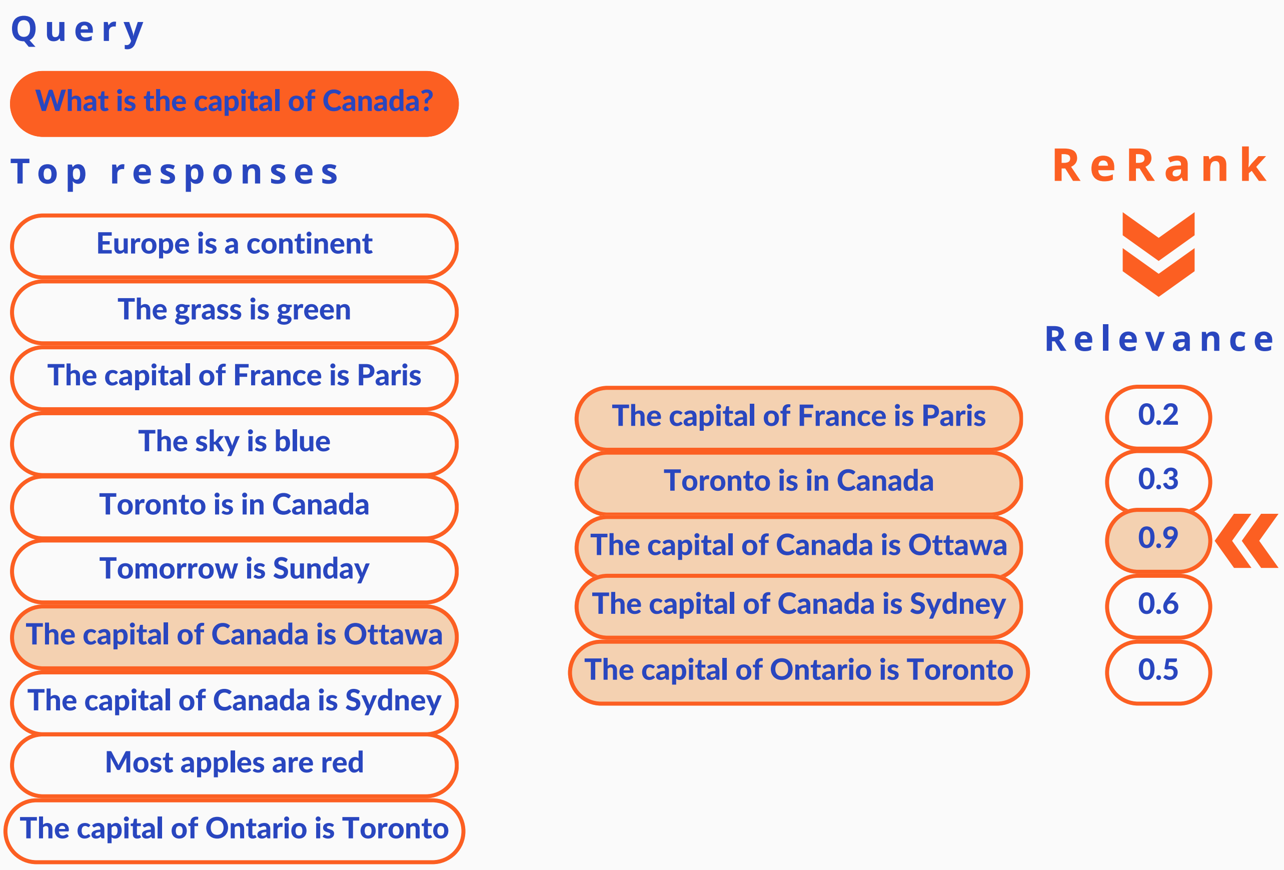
- The capital of Canada is Ottawa
- Toronto is in Canada
- The capital of France is Paris
- The capital of Canada is Sydney
- The capital od Ontario is Toronto


Toronto is in Canada
The capital of France is Paris
What is the capital of Canada?
The capital od Ontario is Toronto
The capital of Canada is Ottawa
The capital of Canada is Sydney

What happens when we do dense retrieval here?

Remember that the way dense retrieval works is it puts the query inside the embedding and then it returns the closest of the responses which in this case is the capital of Ontario is Toronto. It looks at similarities, so it returns the response that is the most similar to the question. This may not be the correct answer, this may not even be a true statement, it's just a sentence that happens to be close to the question semantically. So, therefore, **dense retrieval has the potential to return things that are not necessarily the answer**. How do we fix this? Well, this is where Rerank comes into play.

Solution: ReRank


Query
What is the capital of Canada?
Top responses
Europe is a continent
The grass is green
The capital of France is Paris
The sky is blue
Toronto is in Canada
Tomorrow is Sunday
The capital of Canada is Ottawa
The capital of Canada is Sydney
Most apples are red
The capital of Ontario is Toronto
ReRank
Relevance
The capital of France is Paris
0.2
Toronto is in Canada
0.3
The capital of Canada is Ottawa
0.9
The capital of Canada is Sydney
0.6
The capital of Ontario is Toronto
0.5

Let's say that the query is what is the capital of Canada, and we have 10 possible. As you can see some of these are relevant to the question and some of them are not. When we use dense retrieval, it gives us the top five, let's say, the five responses that are the most similar to the query.

But we don't know which one is the response. We just have five sentences that are pretty close to the query. **Rerank assigns to each query-response pair a relevant score that tells you how relevant the answer is with respect to the query**. It could also be a document. So, how relevant the document is with respect to the query. As you can see, the highest relevance here was 0.9, which corresponds to the capital of Canada is Ottawa, which is the correct answer.

You may be wondering **how Rerank gets trained**. The way train re-rank is by giving it a lot of good pairs, where the query and the response are very relevant, or when the query and document are very relevant, and training it to give those high relevance scores. Then also giving it a big set of wrong query responses - query responses where the response does not correspond to the query. It may be close, but it may not correspond to it or a document that may not correspond to the query.

ReRank is trained on

Many queries with correct answers

What is the capital of Canada?	The capital of Canada is Ottawa
What is the capital of France?	The capital of France is Paris
...	...
What color is the sky?	The sky is blue

Many queries with wrong answers

What is the capital of Canada?	Toronto is in Canada
What is the capital of France?	The capital of France is Cassis
...	...
What color is the sky?	The sky is red

If you train a model to give high scores to the good query response pairs and low scores to the bad query response pairs then you have the re-rank model that assigns a relevance and the relevance is high when you have a query and a response that are very related.

Now, that we have all these search systems you may be wondering **how to evaluate** them. There are several ways to evaluate them, and some of them are mean average.

Evaluationg Search Systems

- Mean Average Precision (MAP)
- Mean Reciprocal Rank (MRR)
- Normalized Discounted Cumulative Gain (NDCG)

Precision or MAP, Mean Reciprocal Rank or MRR, and Normalized Discounted Cumulative Gain or NDCG.

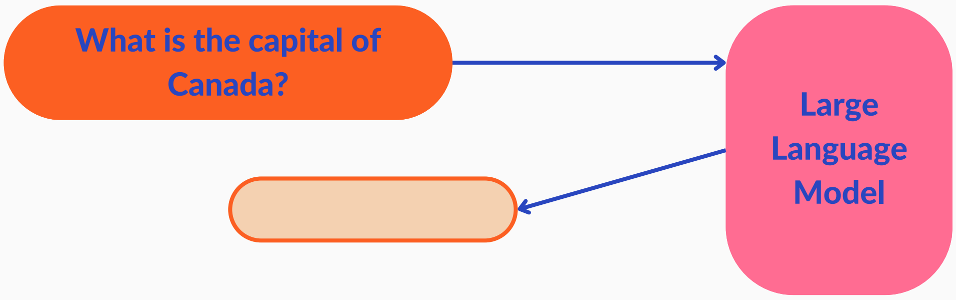
Now, how would you make a test set for evaluating these models? Well, a good test set would be one containing queries, and correct responses, and then, you can compare these correct responses with the responses that the model gives you in a very similar way as you would find the accuracy or precision, or recall of a classification model.

Generating Answers

Now, we'll add a generation step using an LLM at the end of the search pipeline. This way, we can get an answer instead of search results, for example.

This is a cool method to build apps where a user can chat with a document or a book, or, as we'll see in this lesson, an article.

Search can help LLMs in multiple ways

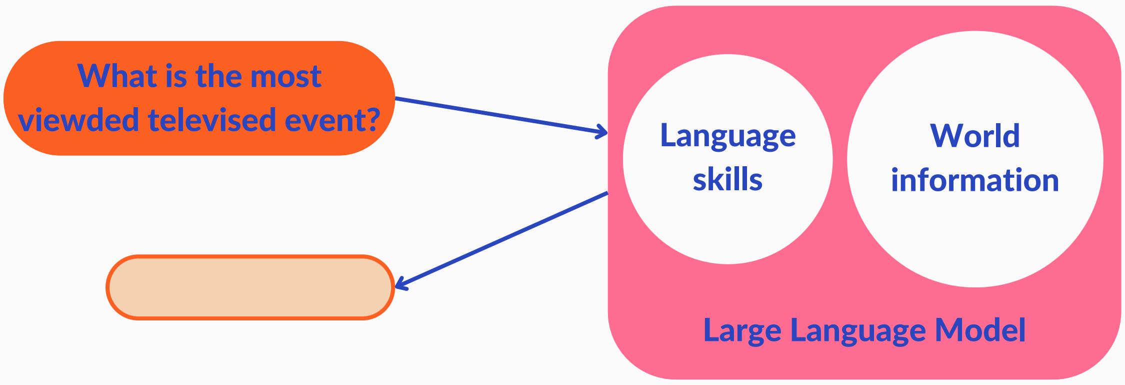


Retrieval can help LLMs with:

- Factual information
- Private information
- Updated information

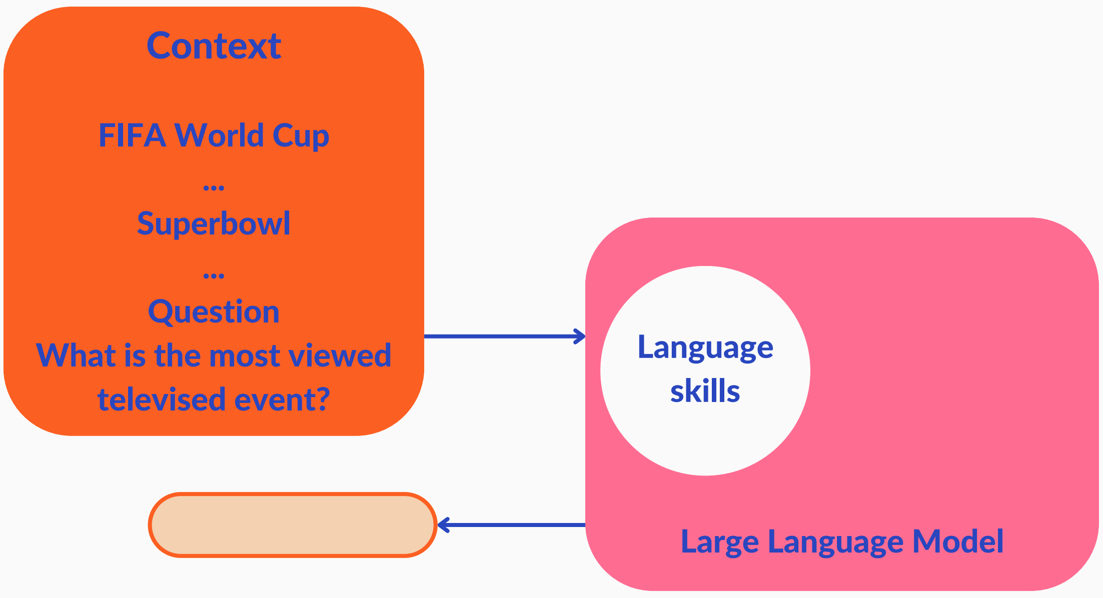
We can ask a large-language model a question, and they're able to answer many questions. But sometimes we want them to answer from a specific document or archive. **This is where you can add a search component before the generation step to improve those generations.** When you rely on a large-language model's direct answer, you're relying on the world information it has stored inside of it.

Where is the information stored?



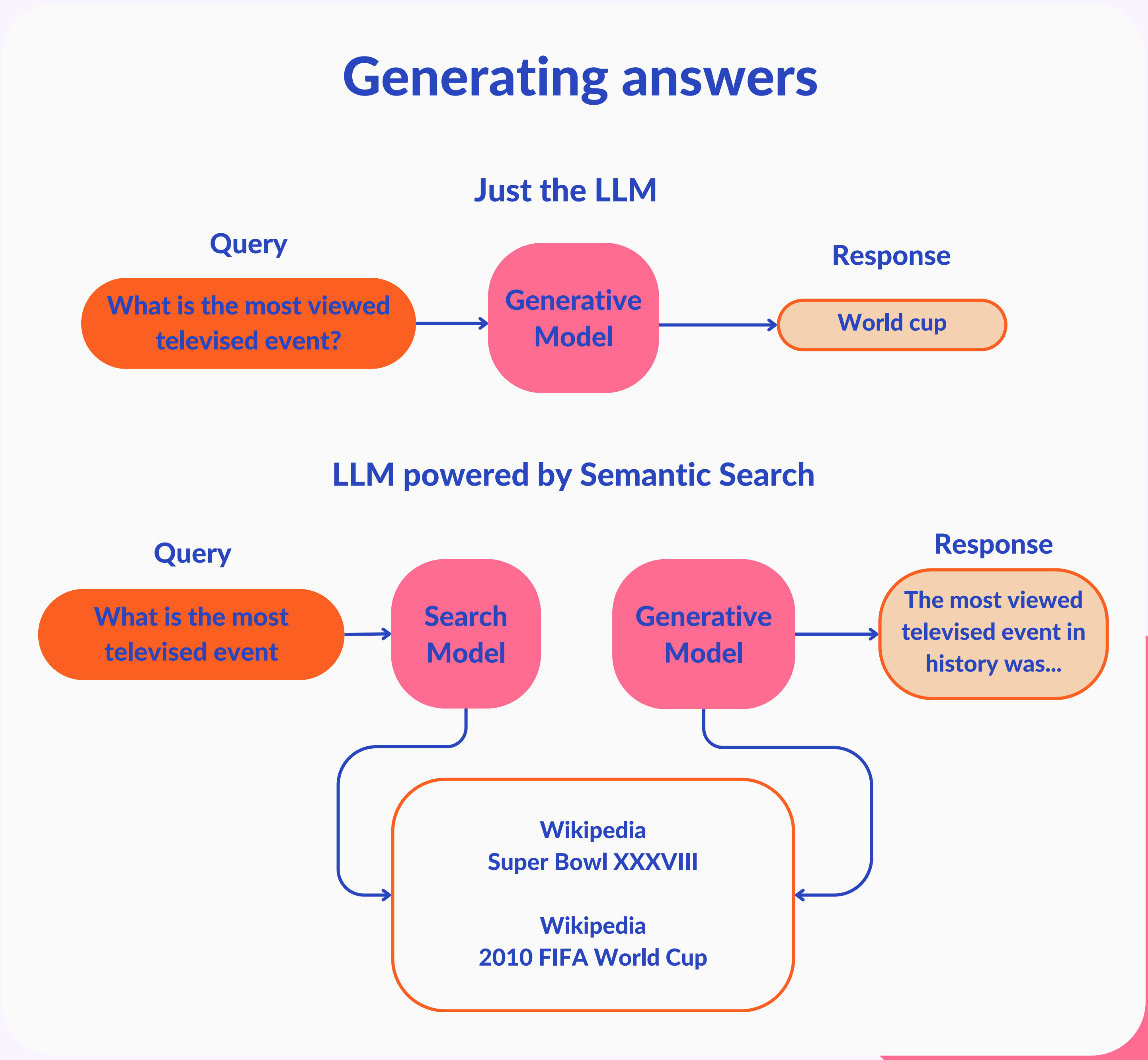
But you can provide it with context using a search step beforehand, for example. When you provide it to the context in the prompt, that leads to better generations for cases when you want to anchor the model to a specific domain or article or document or our text archive in general. This also improves factual generations.

Search can add some context



So, in a lot of cases where you want facts to be retrieved from the model, and you augment it with a context like this, that improves the probability of the model being more factual in its generation.

The difference between the two steps is, instead of just asking the question to a generative model and seeing the result that it prints out, we can first present the question to a search system, exactly like the ones we built earlier in this course. Then we retrieve some of these results, we pose them in the prompt to the generative model in addition to the question and then we get that response that was informed by the context.



Course finished

Check a more extensive course in LLMs at Cohere called LLM University. In this course, you can learn a lot more topics regarding LLMs.

Welcome to LLM University!

LLM University (LLMU) offers in-depth, practical NLP and LLM training. Ideal for all skill levels. Learn, build, and deploy Language AI with Cohere.

Cohere AI

Go to page