Large Language Models with Semantic Search

Notes

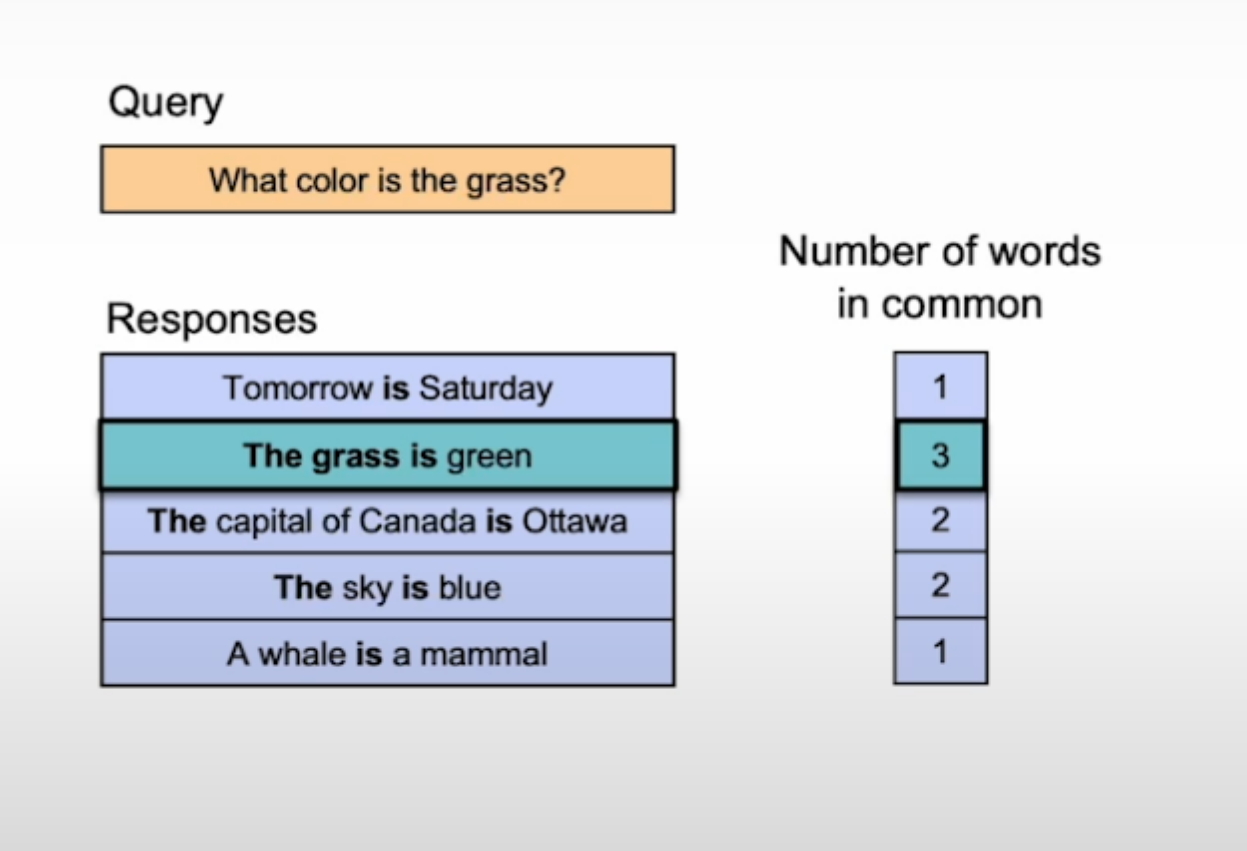
We will learn Lexical search, ReRank, Dense Retrieval, Embeddings, Evaluation methods,

1. Keyword Search:

Most common type of method in the search system. We can do keyword search in the database to get the results.

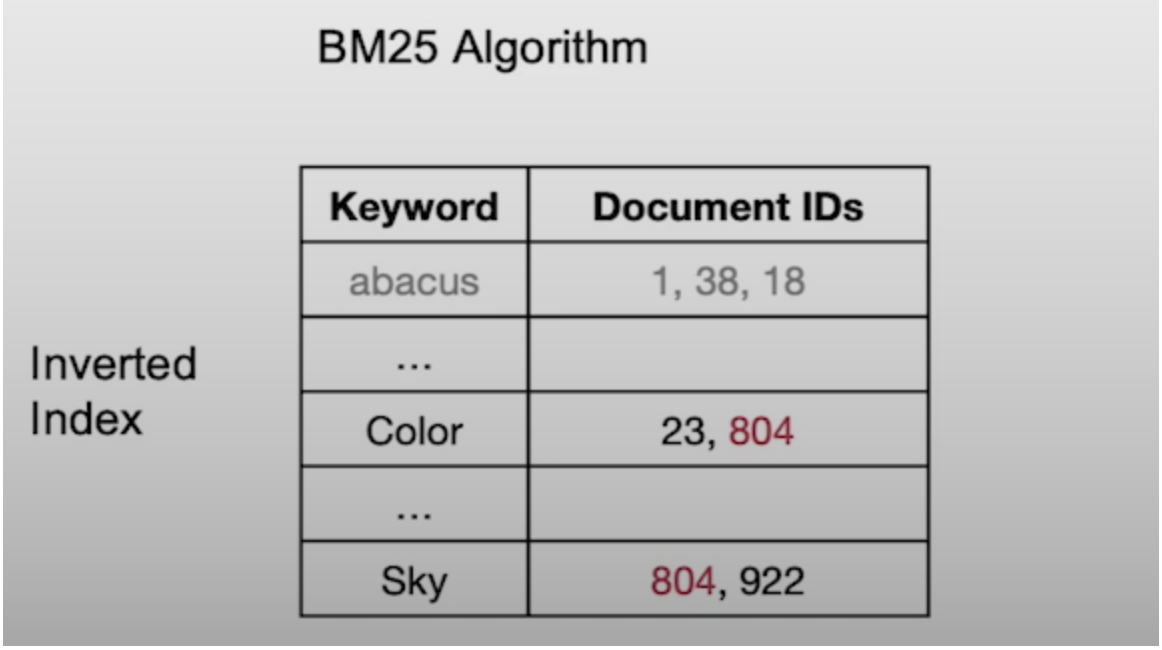
Weaviate is online database that has the keyword search and vector search capabilities.

High level idea of keyword search is that how many words are common between the query (word we are searching) and records/documents present in the database from where we are searching, this is also called as Lexical search.



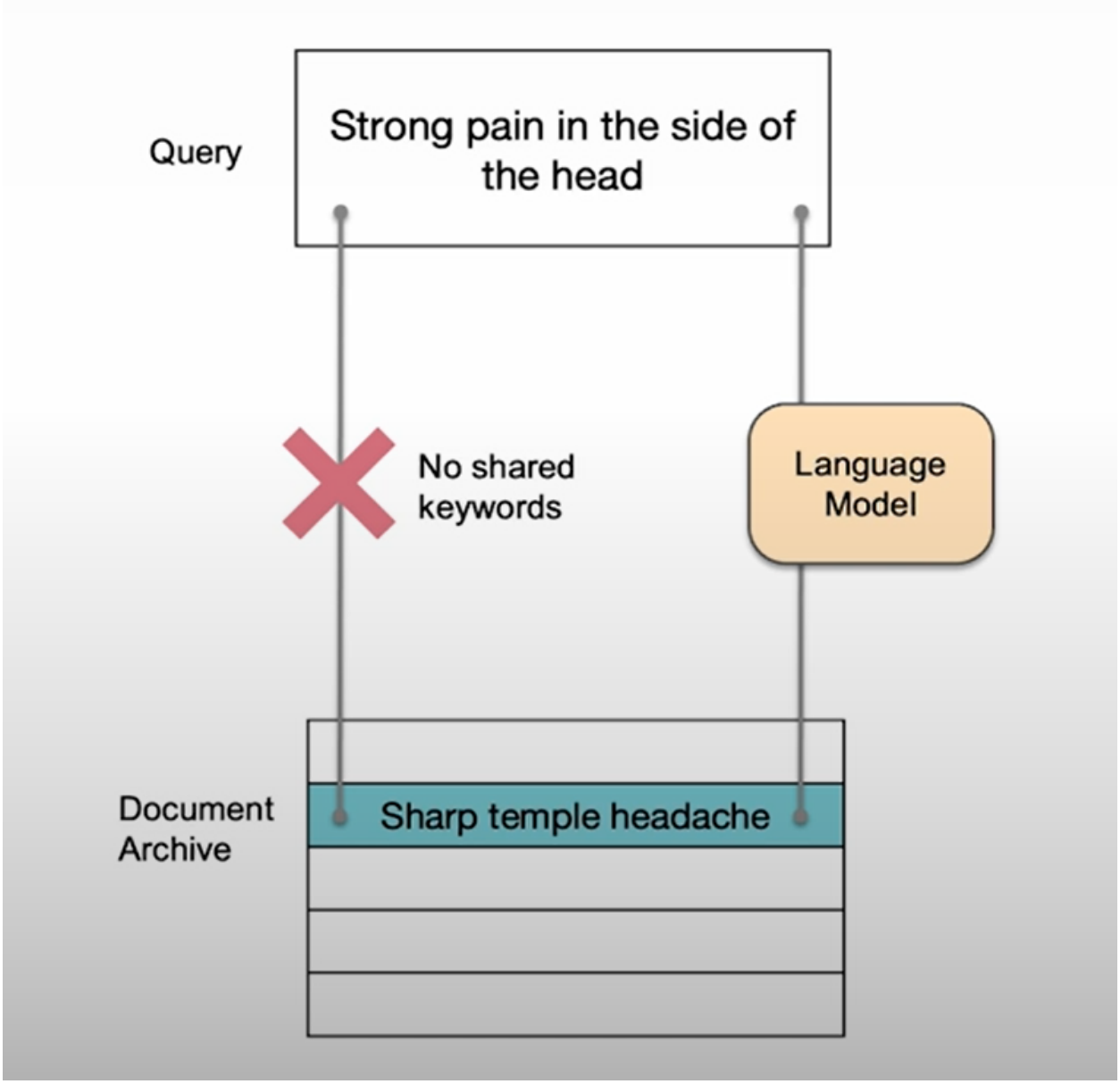
Major components of the keyword search system are:

- Query: Sentence or any question about which we want to retrieve information from database.
- Search System: Search system has multiple stage:
    - Retrieval: Commonly uses the BM25 algorithm to score the documents in the archive versus the query. In this a table called inverted index is prepared that has two columns Keyword and Document IDs. Basically Keyword has word and Document Ids has the ids of the documents in which that specific keyword appeared. This is done to speed up the searching system.
    - Reranking: To include additional signals instead using the text relevance.



Limitations of Keyword Search:

Main limitation is very much highlighted from how keyword search works. If document/record is similar to the query but does not share the keywords then it is irrelevant for this type of system, as it is not understanding the context behind the query and records. That is where LLMs come in and solve this major issue.



Keyword Search using cohere and Weaviate

```
def keyword_search(query,
                   results_lang='en',
                   properties = ["title","url","text"],
                   num_results=3):

    where_filter = {
    "path": ["lang"],
```

```
    "operator": "Equal",
    "valueString": results_lang
    }

    response = (
        client.query.get("Articles", properties

        #we are using bm25 which is algorithm used in keyword se
        .with_bm25(
            query=query
        )
        .with_where(where_filter)
        .with_limit(num_results)
        .do()
        )

    result = response['data']['Get']['Articles']
    return result
```

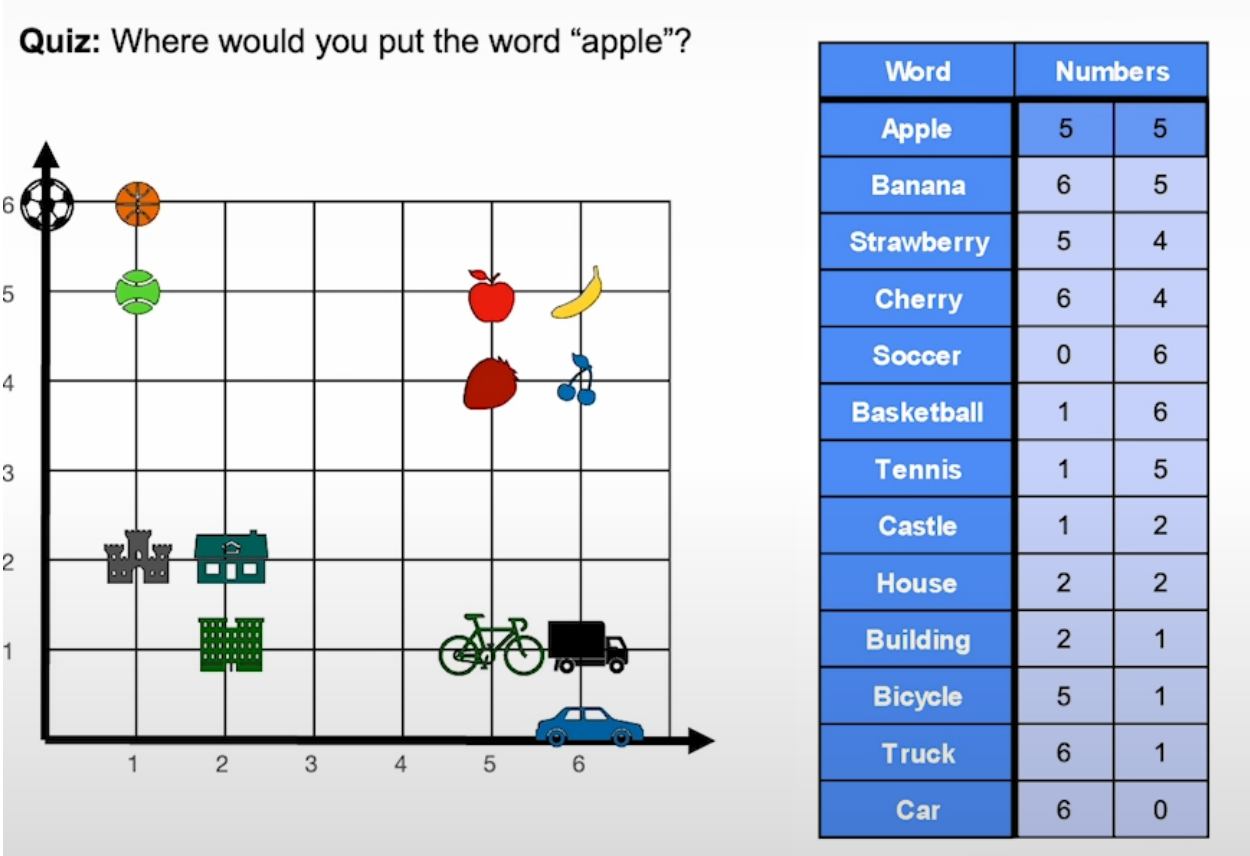
Embeddings:

Embeddings are the numerical representation of the text that computers can more easily process.

Cohere is the library of function that uses Large Language model. Have function like

- Generate
- Embed etc.

Embeddings represent each word with numbers like in below picture each word is represented as two numbers which are the coordinates where that word lie on the plane.



Word	Numbers	
Apple	5	5
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

But in real life embeddings use 100s or 1000s of numbers to represent a word. We can embed sentences or long pieces of the documents.

Large sentences embeddings are represented as a vector called as embedding vector, and sentence that have the similar context have the similar embeddings. See below image where first and all sentence are not same but they have same context so their embeddings are pretty close.

Sentence	Numbers				
Hello, how are you?	0.39	0.49	…	-1.01	-0.72
I'm going to school today	-0.79	-0.05	…	-0.94	2.71
…	…	…	…	…	…
Once upon a time	3.23	-0.23	…	-1.45	0.82
Hi, how's it going?	0.41	0.48	…	-0.98	-0.66

Embeddings of different words and sentences using Cohere embed function:

Words Embeddings

```
#Creating a small dataset of words:

three_words = pd.DataFrame({'text':
  [
      'joy',
      'happiness',
      'potato'
  ]})

#now we will use cohere emved function to make embeddings of the
three_words_emb = co.embed(texts=list(three_words['text']),
                           model='embed-english-v2.0').embedding

word_1 = three_words_emb[0]
word_2 = three_words_emb[1]
```

```
word_3 = three_words_emb[2]

len(word_1)
#-----output
#4096 ---? Each word is represented as 4096 numbers
```

Sentences Embeddings

```
sentences = pd.DataFrame({'text':
  [
   'Where is the world cup?',
   'The world cup is in Qatar',
   'What color is the sky?',
   'The sky is blue',
   'Where does the bear live?',
   'The bear lives in the the woods',
   'What is an apple?',
   'An apple is a fruit',
  ]})

emb = co.embed(texts=list(sentences['text']),
               model='embed-english-v2.0').embeddings
#each sentence is also represented with 4096 numbers
```

Plotting Embeddings

This is the code that we can use to plot the embeddings:

```
import umap
import altair as alt

from numba.core.errors import NumbaDeprecationWarning, NumbaPend
import warnings

warnings.simplefilter('ignore', category=NumbaDeprecationWarning
```

```
warnings.simplefilter('ignore', category=NumbaPendingDeprecatio


def umap_plot(text, emb):

    cols = list(text.columns)
    # UMAP reduces the dimensions from 1024 to 2 dimensions tha
    reducer = umap.UMAP(n_neighbors=2)
    umap_embeds = reducer.fit_transform(emb)
    # Prepare the data to plot and interactive visualization
    # using Altair
    #df_explore = pd.DataFrame(data={'text': qa['text']})
    #print(df_explore)

    #df_explore = pd.DataFrame(data={'text': qa_df[0]})
    df_explore = text.copy()
    df_explore['x'] = umap_embeds[:,0]
    df_explore['y'] = umap_embeds[:,1]

    # Plot
    chart = alt.Chart(df_explore).mark_circle(size=60).encode(
        x=#'x',
        alt.X('x',
            scale=alt.Scale(zero=False)
        ),
        y=
        alt.Y('y',
            scale=alt.Scale(zero=False)
        ),
        tooltip=cols
        #tooltip=['text']
    ).properties(
        width=700,
        height=400
    )
    return chart
```

```
def umap_plot_big(text, emb):

    cols = list(text.columns)
    # UMAP reduces the dimensions from 1024 to 2 dimensions that
    reducer = umap.UMAP(n_neighbors=100)
    umap_embeds = reducer.fit_transform(emb)
    # Prepare the data to plot and interactive visualization
    # using Altair
    #df_explore = pd.DataFrame(data={'text': qa['text']})
    #print(df_explore)

    #df_explore = pd.DataFrame(data={'text': qa_df[0]})
    df_explore = text.copy()
    df_explore['x'] = umap_embeds[:,0]
    df_explore['y'] = umap_embeds[:,1]

    # Plot
    chart = alt.Chart(df_explore).mark_circle(size=60).encode(
        x=#'x',
        alt.X('x',
            scale=alt.Scale(zero=False)
        ),
        y=
        alt.Y('y',
            scale=alt.Scale(zero=False)
        ),
        tooltip=cols
        #tooltip=['text']
    ).properties(
        width=700,
        height=400
    )
    return chart

def umap_plot_old(sentences, emb):
```

```
    # UMAP reduces the dimensions from 1024 to 2 dimensions that
    reducer = umap.UMAP(n_neighbors=2)
    umap_embeds = reducer.fit_transform(emb)
    # Prepare the data to plot and interactive visualization
    # using Altair
    #df_explore = pd.DataFrame(data={'text': qa['text']})
    #print(df_explore)

    #df_explore = pd.DataFrame(data={'text': qa_df[0]})
    df_explore = sentences
    df_explore['x'] = umap_embeds[:,0]
    df_explore['y'] = umap_embeds[:,1]

    # Plot
    chart = alt.Chart(df_explore).mark_circle(size=60).encode(
        x=#'x',
        alt.X('x',
            scale=alt.Scale(zero=False)
        ),
        y=
        alt.Y('y',
            scale=alt.Scale(zero=False)
        ),
        tooltip=['text']
    ).properties(
        width=700,
        height=400
    )
    return chart
```

Lets use above functions:

```
chart = umap_plot(sentences, emb)
```

To plot embeddings of the large articles:

```
articles = wiki_articles[['title', 'text']]
embeds = np.array([d for d in wiki_articles['emb']])
chart = umap_plot_big(articles, embeds)
chart.interactive()
```

Semantic Search

Semantic search is the search by meaning, where meaning/context of the two sentences are considered while giving results. There are two ways of doing semantic search:

- Dense Retrieval
- ReRank

Dense Retrieval:

As we know similar sentence embeddings have pretty similar embeddings and they are near in the plane, when we plot embeddings.

So in dense retrieval most closed sentences to our query are returned as a result.

Dense Retrieval using Weaviate:

```
def dense_retrieval(query,
                    results_lang='en',
                    properties = ["text", "title", "url", "views
                    num_results=5):

        #our query text
    nearText = {"concepts": [query]}

    # To filter by language
```

```
    where_filter = {
    "path": ["lang"],
    "operator": "Equal",
    "valueString": results_lang
    }
    response = (
        client.query
        .get("Articles", properties)

        #we are doing dense retrieval so we want nearest senten
        #we will pass our query
        .with_near_text(nearText)
        .with_where(where_filter)
        .with_limit(num_results)
        .do()
    )

    result = response['data']['Get']['Articles']

    return result
```

If we compare results of Dense retrieval with the results of Keyword search, Dense retrieval returns more accurate results. Moreover it supports multiple languages such that we can ask query in one language mean while our database is in other language, It will properly return the results.

Building Dense retrieval for Semantic Search from scratch:

- Text:

  First of all we need text that we can embed and store in vector search database.

  ```
  text = """
  Interstellar is a 2014 epic science fiction film co-written,
  ```

```
It stars Matthew McConaughey, Anne Hathaway, Jessica Chastain
Set in a dystopian future where humanity is struggling to sur

Brothers Christopher and Jonathan Nolan wrote the screenplay,
Caltech theoretical physicist and 2017 Nobel laureate in Phys
Cinematographer Hoyte van Hoytema shot it on 35 mm movie film
Principal photography began in late 2013 and took place in Al
Interstellar uses extensive practical and miniature effects a

Interstellar premiered on October 26, 2014, in Los Angeles.
In the United States, it was first released on film stock, ex
The film had a worldwide gross over $677 million (and $773 mi
It received acclaim for its performances, direction, screenpl
It has also received praise from many astronomers for its sci
Interstellar was nominated for five awards at the 87th Academ
```

- Chunking:

  Make chunks of the whole text on the basis of sentences (split at .) or on the basis of paragraphs (split on the /n).

```
# Split into a list of sentences
texts = text.split('.')

# Clean up to remove empty spaces and new lines
texts = np.array([t.strip(' \n') for t in texts])
```

- Adding Context to each chunk:

  We are adding this title of the document in start of each chunk so that we can have context of that in each chunk (will give more accurate results). For example if chunk says It makes $2 million. then it be embedded and located away from sentences having pure information of movie.

- Get the embeddings:

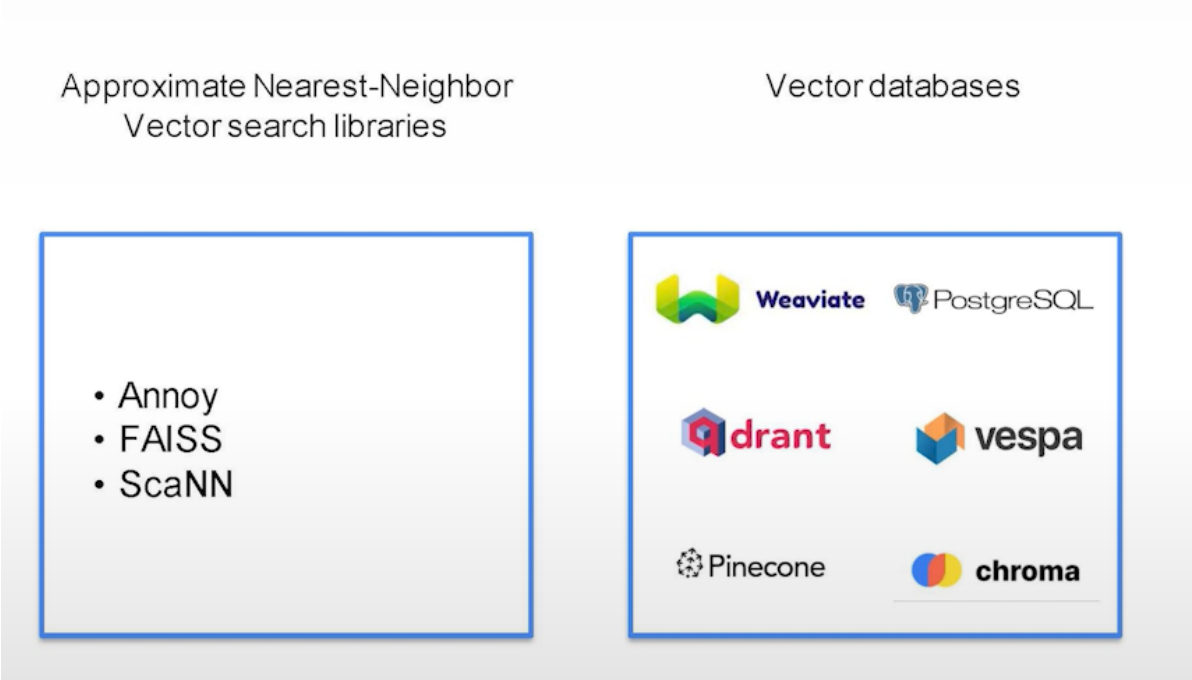
```
response = co.embed(
    texts=texts.tolist()
).embeddings

embeds = np.array(response)
embeds.shape
```

- Setting up Nearest Neighbor Library:

  Basically we have approximate nearest neighbor libraries like Annoy, FAISS, SCANN that are open source, they are easy to set up and only stores vectors.

  On the other hand we have vector databases like Pinecone, Weaviate, chroma etc. It is little bit more advance in a way:

  - You can add new data easily in vector databases, on the other hand you to reindex the whole data from start using ANN libraries.
  - Vector database stores text and vectors both.
  - Allow more advance queries.



```
search_index = AnnoyIndex(embeds.shape[1], 'angular')
# Add all the vectors to the search index
for i in range(len(embeds)):
    search_index.add_item(i, embeds[i])

search_index.build(10) # 10 trees
search_index.save('test.ann')
```

- Search:

```
pd.set_option('display.max_colwidth', None)

def search(query):

  # Get the query's embedding
  query_embed = co.embed(texts=[query]).embeddings

  # Retrieve the nearest neighbors
  similar_item_ids = search_index.get_nns_by_vector(query_embed
                                                    3,
                                                  include_distar
  # Format the results
  results = pd.DataFrame(data={'texts': texts[similar_item_ids[0
                               'distance': similar_item_ids[1]})

  print(texts[similar_item_ids[0]])

  return results


query = "How much did the film make?"
search(query)
```

```
['Interstellar (film) Interstellar (film) The film had a worldwide gross over $677 million (and
$773 million with subsequent re-releases), making it the tenth-highest grossing film of 2014'
 'Interstellar (film) Interstellar (film) Interstellar premiered on October 26, 2014, in Los An
geles'
 'Interstellar (film) Interstellar (film) In the United States, it was first released on film s
tock, expanding to venues using digital projectors']
```

	texts	distance
0	Interstellar (film) Interstellar (film) The film had a worldwide gross over $677 million (and $773 million with subsequent re-releases), making it the tenth-highest grossing film of 2014	1.010105
1	Interstellar (film) Interstellar (film) Interstellar premiered on October 26, 2014, in Los Angeles	1.154889
2	Interstellar (film) Interstellar (film) In the United States, it was first released on film stock, expanding to venues using digital projectors	1.168121

ReRank:

It is a second method of doing the Semantic Search.

In this Large Language models sort the results from best to worst on the basis of similarity with the query.

The main disadvantage of Dense retrieval is its own feature which is returning the nearest sentence. It becomes disadvantage when similar sentence is stored near to query but it is not the actual answer of the query.

ReRank basically give a score for 0 to 1 to each result on the basis of its relevance with the query.

ReRank is trained using two things:

- A lot of queries with their correct answers.
- A lot of queries with wrong answers. That answer can be close and similar to query but not correct.

We can use ReRank with both Keyword search where we can pass the huge number of results taken from keyword search to ReRank function and we can get the most relevant answer.

```
def rerank_responses(query, responses, num_responses=10):
    reranked_responses = co.rerank(
        model = 'rerank-english-v2.0',
        query = query,
        documents = responses,
```

```
        top_n = num_responses,
        )
    return reranked_responses

texts = [result.get('text') for result in results]
reranked_text = rerank_responses(query_1, texts)
```

we can use this ReRank with Dense search as we have used it with keyword search, in which we can get results from the dense search and pass those list of results to ReRank function to get highly relevant result.

We can test each search system by preparing the dataset having queries and correct answer and then using Mean Average precision, Mean Reciprocal Rank to evaluate the search systems.

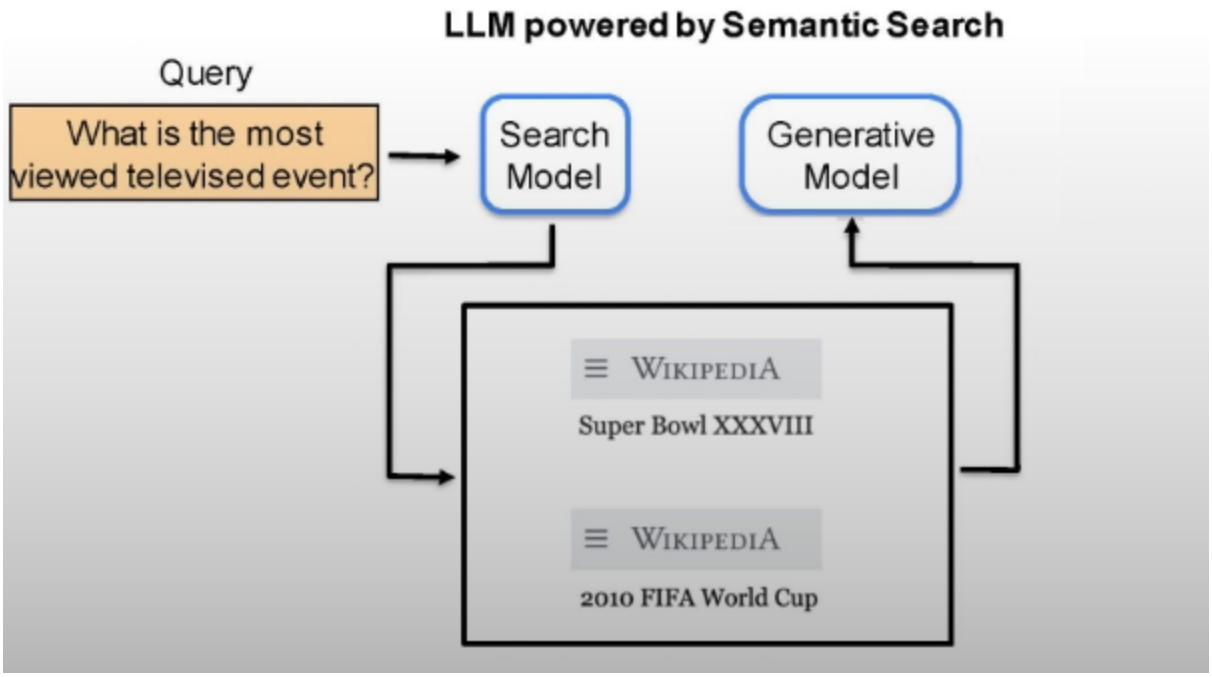
Generating Answers using LLMs:

This is the last step after receiving the search results. Where instead of getting the search results we try to get a answer. Useful for chatting with book, document, articles.

This searched answer will act as a context for the LLM to answer correctly and precisely.

So here are the steps:

- Search query using dense search, keyword, ReRank.
- Get results and embed the results in the prompt.
- Pass the prompt to LLM and get the final answer.


LLM powered by Semantic Search
Query
What is the most viewed televised event?
Search Model
Generative Model
WIKIPEDIA
Super Bowl XXXVIII
WIKIPEDIA
2010 FIFA World Cup