2

Semantic Search with LLMs

Introduction

In Chapter 1, we explored the inner workings of language models and the impact that modern LLMs have had on NLP tasks like text classification, generation, and machine translation. Another powerful application of LLMs has also been gaining traction in recent years: semantic search.

Now, you might be thinking that it's time to finally learn the best ways to talk to ChatGPT and GPT-4 to get the optimal results—and we'll start to do that in the next chapter, I promise. In the meantime, I want to show you what else we can build on top of this novel Transformer architecture. While text-to-text generative models like GPT are extremely impressive in their own right, one of the most versatile solutions that AI companies offer is the ability to generate text embeddings based on powerful LLMs.

Text embeddings are a way to represent words or phrases as machine-readable numerical vectors in a multidimensional space, generally based on their contextual meaning. The idea is that if two phrases are similar (we will explore the word "similar" in more detail later on in this chapter), then the vectors that represent those phrases should be close together by some measure (like Euclidean distance), and vice versa. Figure 2.1 shows an example of a simple search algorithm. When a user searches for an item to buy—say, a Magic: The Gathering trading card—they might simply search for "a vintage magic card." The system should then embed this query such that if two text embeddings are near each other, that should indicate the phrases that were used to generate them are similar.

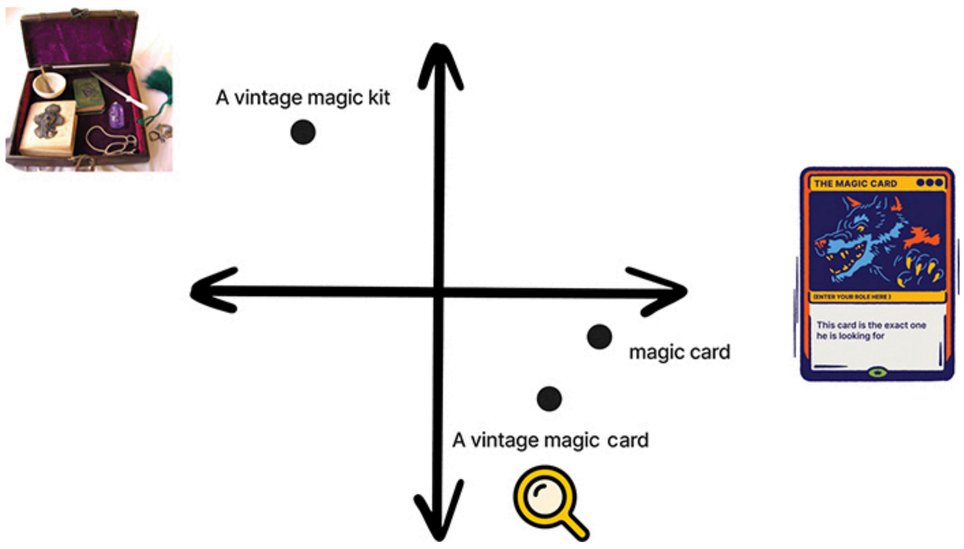


Figure 2.1 Vectors that represent similar phrases should be close together and those that represent dissimilar phrases should be far apart. In this case, if a user wants a trading card, they might ask for “a vintage magic card.” A proper semantic search system should embed the query in such a way that it ends up near relevant results (like “magic card”) and far from nonrelevant items (like “a vintage magic kit”) even if they share certain keywords.

This map from text to vectors can be thought of as a kind of hash with meaning. We can’t really reverse the vectors back to text, though. Rather, they are a representation of the text that has the added benefit of carrying the ability to compare points while in their encoded state.

LLM-enabled text embeddings allow us to capture the semantic value of words and phrases beyond just their surface-level syntax or spelling. We can rely on the pre-training and fine-tuning of LLMs to build virtually unlimited applications on top of them by leveraging this rich source of information about language use.

This chapter introduces the world of semantic search using LLMs to explore how LLMs can be used to create powerful tools for information retrieval and analysis. In Chapter 3, we will build a chatbot on top of GPT-4 that leverages a fully realized semantic search system that we will build in this chapter.

So, without further ado, let’s get into it, shall we?

The Task

A traditional search engine generally takes what you type in and then gives you a bunch of links to websites or items that contain those words or permutations of the characters that you typed in. So, if you typed in "vintage magic the gathering cards" on a marketplace, that search would return items with a title/description containing combinations of those words. That's a pretty standard way to search, but it's not always the best way. For example, I might get vintage magic sets to help me learn how to pull a rabbit out of a hat. Fun, but not what I asked for.

The terms you input into a search engine may not always align with the *exact* words used in the items you want to see. It could be that the words in the query are too general, resulting in a slew of unrelated findings. This issue often extends beyond just differing words in the results; the same words might carry different meanings than what was searched for. This is where semantic search comes into play, as exemplified by the earlier-mentioned Magic: The Gathering cards scenario.

Asymmetric Semantic Search

A **semantic search** system can understand the meaning and context of your search query and match it against the meaning and context of the documents that are available to retrieve. This kind of system can find relevant results in a database without having to rely on exact keyword or *n*-gram matching; instead, it relies on a pre-trained LLM to understand the nuances of the query and the documents (Figure 2.2).

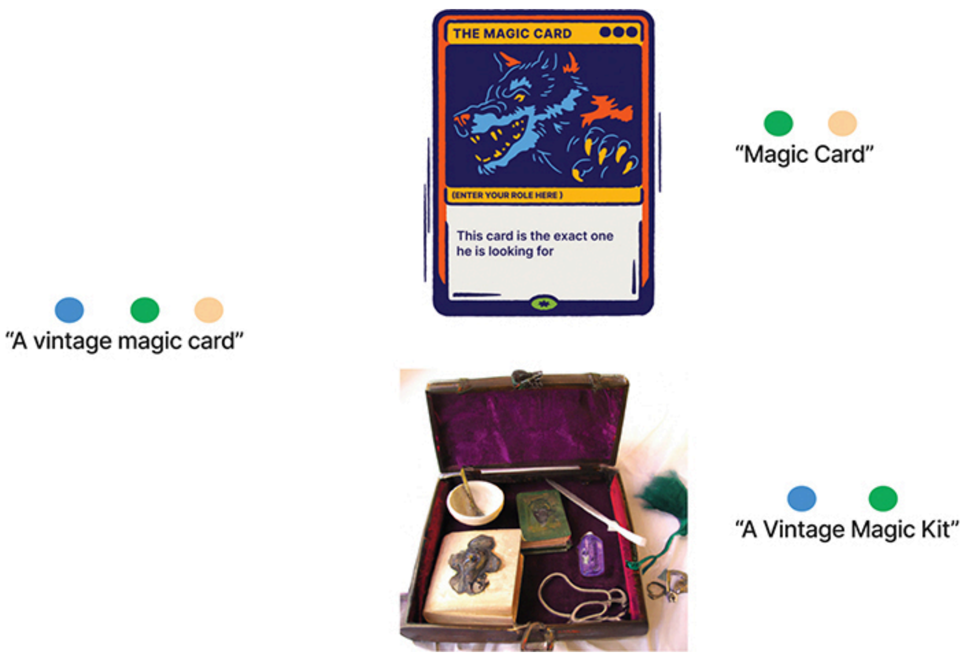


Figure 2.2 A traditional keyword-based search might rank a vintage magic kit with the same weight as the item we actually want, whereas a semantic search system can understand the actual concept we are searching for.

The **asymmetric** part of asymmetric semantic search refers to the fact that there is an imbalance between the semantic information (basically the size) of the input query and the documents/information that the search system has to retrieve. Basically, one of them is much shorter than the other. For example, a search system trying to match "magic the gathering cards" to lengthy paragraphs of item descriptions on a marketplace would be considered asymmetric. The four-word search query has much less information than the paragraphs but nonetheless is what we have to compare.

Asymmetric semantic search systems can produce very accurate and relevant search results, even if you don't use exactly the right words in your search. They rely on the learnings of LLMs rather than the user being able to know exactly which needle to search for in the haystack.

I am, of course, vastly oversimplifying the traditional method. There are many ways to make searches more performant without switching to a more complex LLM approach, and pure semantic search systems are not always the answer. They are not simply "the better way to do search." Semantic algorithms have their own deficiencies, including the following:

- They can be overly sensitive to small variations in text, such as differences in capitalization or punctuation.
- They struggle with nuanced concepts, such as sarcasm or irony, that rely on localized cultural knowledge.
- They can be more computationally expensive to implement and maintain than the traditional method, especially when launching a homegrown system with many open-source components.

Semantic search systems can be a valuable tool in certain contexts, so let's jump right into how we will architect our solution.

Solution Overview

The general flow of our asymmetric semantic search system will follow these steps:

- Part I: Ingesting documents (**Figure 2.3**)
  1. Collect documents for embedding (e.g., paragraph descriptions of items)
  2. Create text embeddings to encode semantic information
  3. Store embeddings in a database for later retrieval given a query

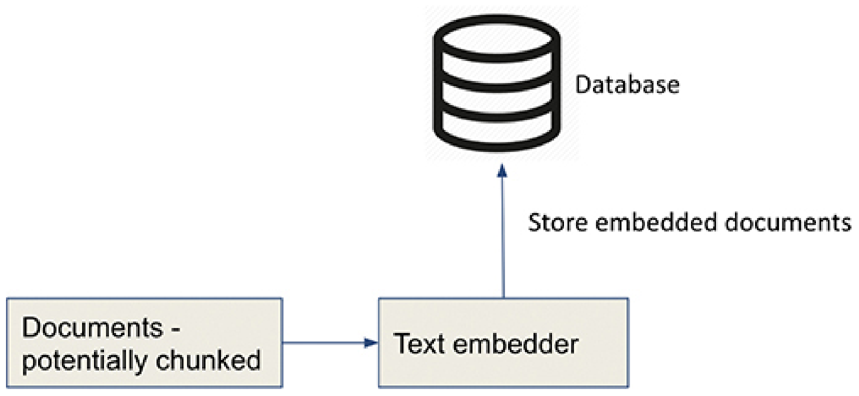


Figure 2.3 Zooming in on Part I, storing documents will consist of doing some preprocessing on our documents, embedding them, and then storing them in some database.

- Part II: Retrieving documents (**Figure 2.4**)
  1. The user has a query that may be preprocessed and cleaned (e.g., a user searching for an item)
  2. Retrieve candidate documents via embedding similarity (e.g., Euclidean distance)
  3. Re-rank the candidate documents if necessary (we will explore this in more detail later on)
  4. Return the final search results to the user

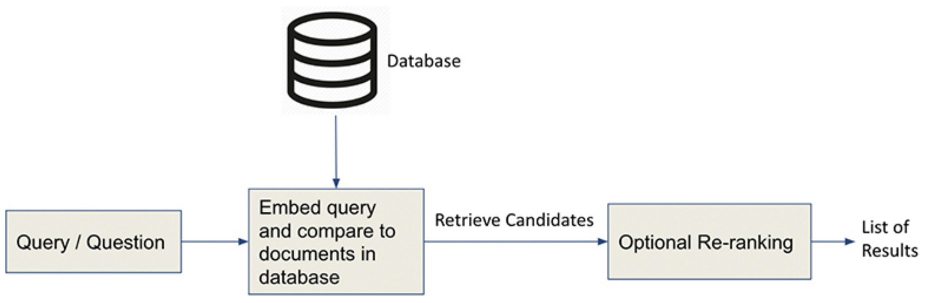


Figure 2.4 Zooming in on Part II, when retrieving documents, we will have to embed our query using the same embedding scheme that we used for the documents, compare them against the previously stored documents, and then return the best (closest) document.

The Components

Let’s go over each of our components in more detail to understand the choices we’re making and which considerations we need to take into account.

Text Embedder

At the heart of any semantic search system is the text embedder. This component takes in a text document, or a single word or phrase, and converts it into a vector. The vector is unique to that text and should capture the contextual meaning of the phrase.

The choice of the text embedder is critical, as it determines the quality of the vector representation of the text. We have many options for how we vectorize with LLMs, both open and closed source. To get off of the ground more quickly, we will use OpenAI's closed-source "Embeddings" product for our purposes here. In a later section, I'll go over some open-source options.

OpenAI's "Embeddings" is a powerful tool that can quickly provide high-quality vectors, but it is a closed-source product, which means we have limited control over its implementation and potential biases. In particular, when using closed-source products, we may not have access to the underlying algorithms, which can make it difficult to troubleshoot any issues that arise.

What Makes Pieces of Text "Similar"

Once we convert our text into vectors, we have to find a mathematical representation of figuring out whether pieces of text are "similar." Cosine similarity is a way to measure how similar two things are. It looks at the angle between two vectors and gives a score based on how close they are in direction. If the vectors point in exactly the same direction, the cosine similarity is 1. If they're perpendicular (90 degrees apart), it's 0. And if they point in opposite directions, it's –1. The size of the vectors doesn't matter; only their orientation does.

Figure 2.5 shows how the cosine similarity comparison would help us retrieve documents given a query.

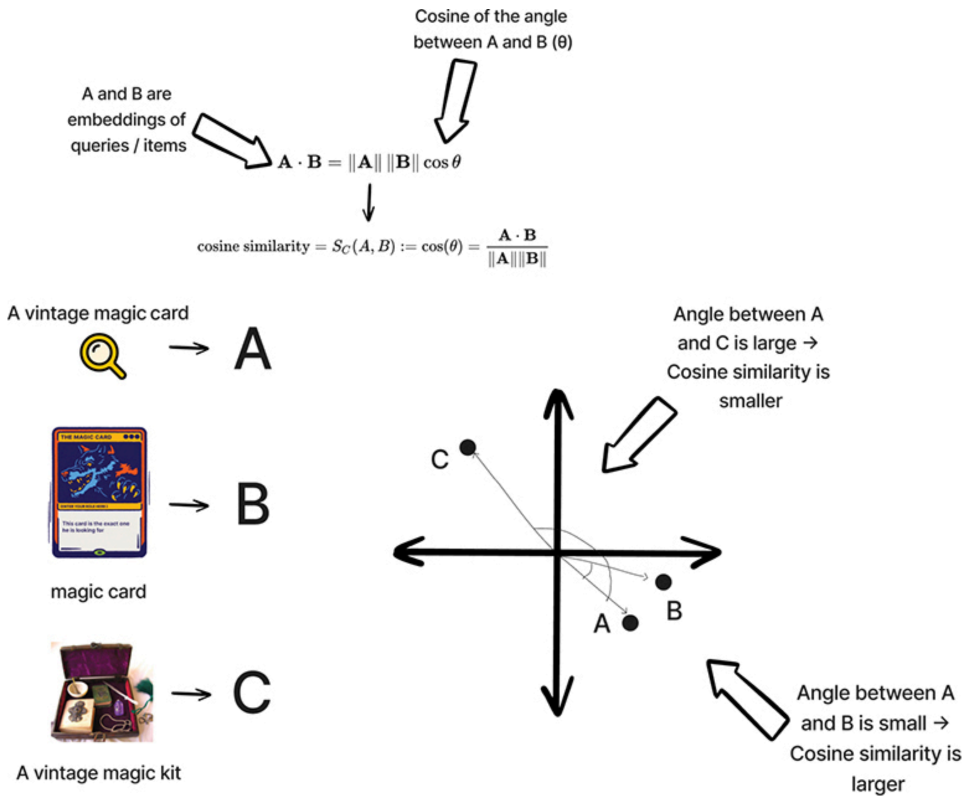

Cosine of the angle between A and B (θ)
A and B are embeddings of queries / items
$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \|\mathbf{B}\| \cos \theta$
cosine similarity $= S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$
A vintage magic card
A
B
magic card
C
A vintage magic kit
Angle between A and C is large → Cosine similarity is smaller
C
A
B
Angle between A and B is small → Cosine similarity is larger

Figure 2.5 In an ideal semantic search scenario, the cosine similarity (formula given at the top) gives us a computationally efficient way to compare pieces of text at scale, given that embeddings are tuned to place semantically similar pieces of text near each other (bottom). We start by embedding all items—including the query (bottom left)—and then checking the angle between them. The smaller the angle, the larger the cosine similarity will be (bottom right).

We could also turn to other similarity metrics, such as the dot product or the Euclidean distance. However, OpenAI embeddings have a special property. The magnitudes (lengths) of their vectors are normalized to length 1, which basically means that we benefit mathematically on two fronts:

- Cosine similarity is identical to the dot product.
- Cosine similarity and Euclidean distance will result in the identical rankings.

Having normalized vectors (all having a magnitude of 1) is great because we can use a cheap cosine calculation to see how close two vectors are and, therefore, how close two phrases are semantically via the cosine similarity.

OpenAI's Embedding Engines

Getting embeddings from OpenAI is as simple as writing a few lines of code (Listing 2.1). As mentioned previously, this entire system relies on an embedding mechanism that places semantically similar items near each other so that the cosine similarity is large when the items are actually similar. With these embedders, the LLMs that power the embedding model are hosted on OpenAI servers and we cannot run them locally. Later in this chapter, we will employ some local embedding models to compare speed and performance.

Listing 2.1 **Getting text embeddings from OpenAI**

Click here to view code image

```
# Importing the necessary modules for the script to run
from openai import OpenAI

# Setting the OpenAI API key using the value stored in the environment variable
'OPENAI_API_KEY'
client = OpenAI(
    api_key=os.environ.get("OPENAI_API_KEY")
)

# Setting the engine to be used for text embedding
ENGINE = 'text-embedding-3-large'  # has vector size 3072

# Generating the vector representation of the given text using the specified en
def get_embeddings(texts, engine=ENGINE):
    response = client.embeddings.create(
        input=texts,
        model=engine
    )

    return [d.embedding for d in list(response.data)]


embedded_text = get_embeddings('I love to be vectorized', engine=ENGINE)

# Checking the length of the resulting vector to ensure it is the expected size
len(embeddecd_text[0]) == '3072'
```

OpenAI provides several embedding engine options that can be used for text embedding. Each engine may provide different levels of accuracy and may be optimized for different types of text data. At the time of this book's writing, the engine used in the code block is the most recent and the one OpenAI recommends using.

Additionally, it is possible to pass in multiple pieces of text at once to the `get_embeddings` function, which can generate embeddings for all of them in a single API call. This can be more efficient than calling `get_embedding` multiple times for each individual section of the text. We will see an example of this later on.

Open-Source Embedding Alternatives

While OpenAI and other companies provide powerful text embedding products, several open-source alternatives for text embedding are also available. One popular option is the bi-encoder with BERT, one of the autoencoding LLMs we discussed in Chapter 1. We can find pre-trained bi-encoders in many open-source repositories, including the **Sentence Transformers** library, which provides pre-trained models for a variety of natural language processing tasks to use off the shelf.

A bi-encoder involves training two BERT models: one to encode the input text and the other to encode the output text (Figure 2.6). The two models are trained simultaneously on a large corpus of text data, with the goal of maximizing the similarity between corresponding pairs of input and output text. The resulting embeddings capture the semantic relationship between the input and output text.

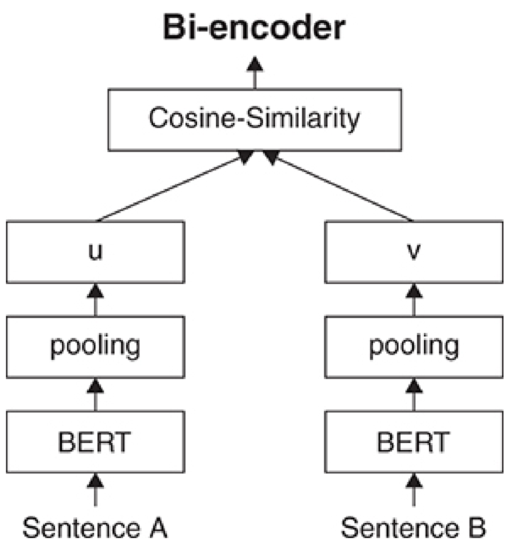


Figure 2.6 A bi-encoder is trained in a unique way, with two clones of a single LLM (in this case, the autoencoding model BERT) being trained in parallel to learn similarities between documents. For example, a bi-encoder can learn to associate questions to paragraphs so they appear near each other in a vector space.

Listing 2.2 is an example of embedding text with a pre-trained bi-encoder with the `sentence_transformer` package.

Listing 2.2 **Getting text embeddings from a pre-trained open-source bi-encoder**

Click here to view code image

```
# Importing the SentenceTransformer library
from sentence_transformers import SentenceTransformer

# Initializing a SentenceTransformer model with the 'multi-qa-mpnet-base-cos-v1
pre-trained model
model = SentenceTransformer(
 'sentence-transformers/all-mpnet-base-v2')

# Defining a list of documents to generate embeddings for
docs = [
 "Around 9 million people live in London",
 "London is known for its financial district"
 ]

# Generate vector embeddings for the documents
doc_emb = model.encode(
 docs, # Our documents (an iterable of strings)
 batch_size=32, # Batch the embeddings by this size
 show_progress_bar=True # Display a progress bar

)

# The shape of the embeddings is (2, 768), indicating a length of 768 and two
embeddings generated
doc_emb.shape # == (2, 768)
```

This code creates an instance of the `SentenceTransformer` class, which is initialized with the pre-trained model `all-mpnet-base-v2`. This model is designed for multitask learning, specifically for tasks such as question-answering and text classification. It was pre-trained using asymmetric data, so we know it can handle both short queries and long documents and be able to compare them well. We use the `encode` function from the `SentenceTransformer` class to generate vector embeddings for the documents, with the resulting embeddings stored in the `doc_emb` variable.

Different algorithms may perform better on different types of text data and will have different vector sizes. The choice of algorithm can have a significant impact on the quality of the resulting embeddings. Additionally, open-source alternatives may require more customization and fine-tuning than closed-source products, but they also provide greater flexibility and control over the embedding process. For more examples of using open-source bi-encoders to embed text, check out the code portion of this book.

Document Chunking

Once we have our text embedding engine set up, we need to consider the challenge of embedding large documents. It is often not practical to embed entire documents as a single vector, particularly when we're dealing with long documents such as books or research papers. One solution to this problem is to use document chunking, which involves dividing a large document into smaller, more manageable chunks for embedding.

Max Token Window Chunking

One approach to document chunking is max token window chunking. One of the easiest methods to implement, it involves splitting the document into chunks of a given maximum size. For example, if we set a token window to be 500, we would expect each chunk to be a bit less than 500 tokens. Creating chunks that are all roughly the same size will also help make our system more consistent.

One common concern with this method is that we might accidentally cut off some important text between chunks, splitting up the context. To mitigate this problem, we can set overlapping windows with a specified amount of tokens to overlap so that tokens are shared between chunks. Of course, this introduces a sense of redundancy, but that's often okay in service of higher accuracy and latency.

Let's see an example of overlapping window chunking with some sample text (Listing 2.3). We'll begin by ingesting a large document. How about a recent book I wrote that has more than 400 pages?

Listing 2.3 **Ingesting an entire textbook**

Click here to view code image

```
# Use the PyPDF2 library to read a PDF file
import PyPDF2

# Open the PDF file in read-binary mode
with open('../data/pds2.pdf', 'rb') as file:

 # Create a PDF reader object
 reader = PyPDF2.PdfReader(file)

 # Initialize an empty string to hold the text
 principles_of_ds = ‘’

 # Loop through each page in the PDF file
 for page in tqdm(reader.pages):
 # Extract the text from the page
 text = page.extract_text()

 # Find the starting point of the text we want to extract
 # In this case, we are extracting text starting from the string ' ]'
 principles_of_ds += '\n\n' + text[text.find(' ]')+2:]

# Strip any leading or trailing whitespace from the resulting string
principles_of_ds = principles_of_ds.strip()
```

Now let's chunk this document by getting chunks of at most a certain token size (Listing 2.4).

Listing 2.4 **Chunking the textbook with and without overlap**

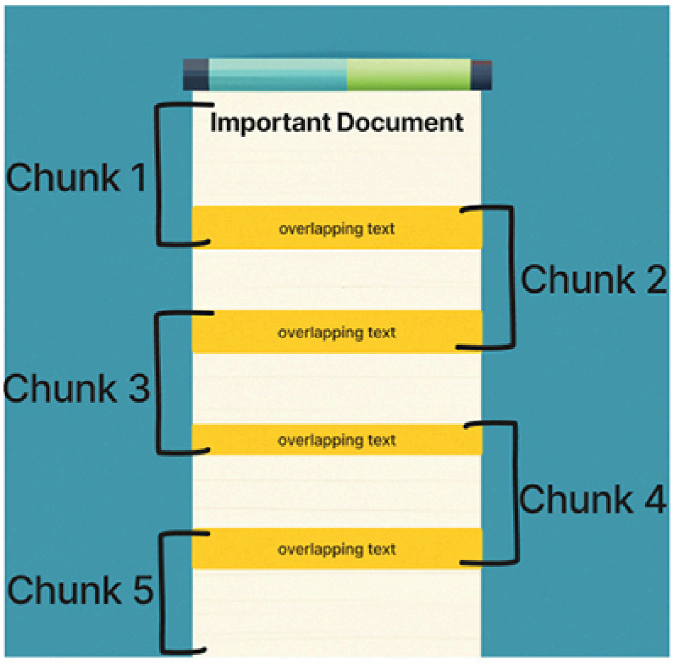
Click here to view code image

```
# Function to split the text into chunks of a maximum number of tokens.
Inspired by OpenAI
def overlapping_chunks(text, max_tokens = 500, overlapping_factor = 5):
 '''
 max_tokens: tokens we want per chunk
 overlapping_factor: number of sentences to start each chunk with that overlaps
the previous chunk
 '''

 # Split the text using punctuation
 sentences = re.split(r'[.?!]', text)

 # Get the number of tokens for each sentence
 n_tokens = [len(tokenizer.encode(" " + sentence)) for sentence in sentences]

 chunks, tokens_so_far, chunk = [], 0, []

 # Loop through the sentences and tokens joined together in a tuple
 for sentence, token in zip(sentences, n_tokens):

 # If the number of tokens so far plus the number of tokens in the current sent
greater
 # than the max number of tokens, then add the chunk to the list of chunks and
 # the chunk and tokens so far
 if tokens_so_far + token > max_tokens:
 chunks.append(". ".join(chunk) + ".")
 if overlapping_factor > 0:
 chunk = chunk[-overlapping_factor:]
 tokens_so_far = sum([len(tokenizer.encode(c)) for c in chunk])
 else:
 chunk = []
 tokens_so_far = 0
 # If the number of tokens in the current sentence is greater than the max numb
 # tokens, go to the next sentence
 if token > max_tokens:
 continue

 # Otherwise, add the sentence to the chunk and add the number of tokens to the
 chunk.append(sentence)
 tokens_so_far += token + 1

 return chunks

split = overlapping_chunks(principles_of_ds, overlapping_factor=0)
avg_length = sum([len(tokenizer.encode(t)) for t in split]) / len(split)
print(f'non-overlapping chunking approach has {len(split)} documents with avera
length {avg_length:.1f} tokens')
non-overlapping chunking approach has 286 documents with average length 474.1
tokens

# with 5 overlapping sentences per chunk
split = overlapping_chunks(principles_of_ds, overlapping_factor=5)
avg_length = sum([len(tokenizer.encode(t)) for t in split]) / len(split)
print(f'overlapping chunking approach has {len(split)} documents with average
{avg_length:.1f} tokens')
overlapping chunking approach has 391 documents with average length 485.4
tokens
```

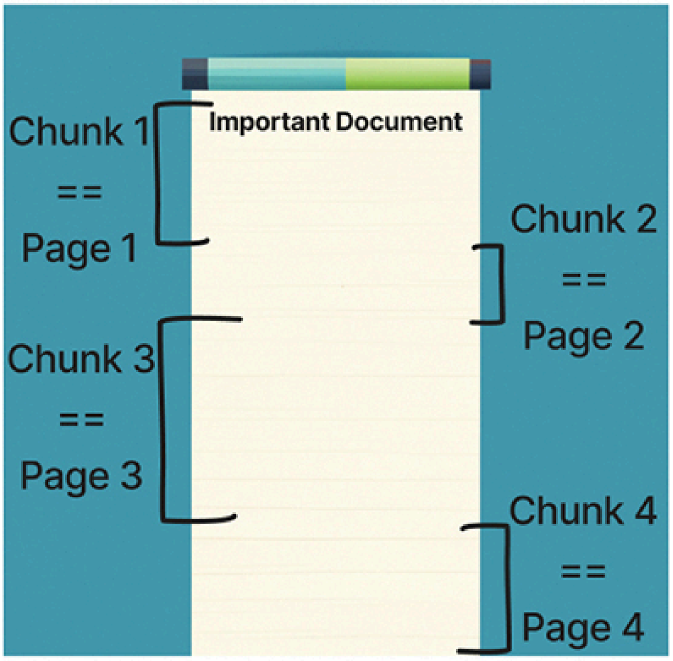
With overlap, we see an increase in the number of document chunks, but they are all approximately the same size. The higher the overlapping factor, the more redundancy we introduce into the system. The max token window method does not take into account the natural structure of the document, however, and it may result in information being split up between chunks or chunks with overlapping information, confusing the retrieval system.

Finding Custom Delimiters

To help aid our chunking method, we could search for custom natural delimiters like page breaks in a PDF or newlines between paragraphs. For a given document, we would identify natural whitespace within the text and use it to create more meaningful units of text that will end up in document chunks that eventually get embedded (Figure 2.7).



Max Token Window Method with Overlap



Natural Whitespace Chunking with No Overlap

Figure 2.7 Max token chunking and natural whitespace chunking can be done with or without overlap. The natural whitespace chunking tends to end up with non-uniform chunk sizes.

Let's look for common types of whitespace in the textbook (Listing 2.5).

Listing 2.5 **Chunking the textbook with natural whitespace**

Click here to view code image

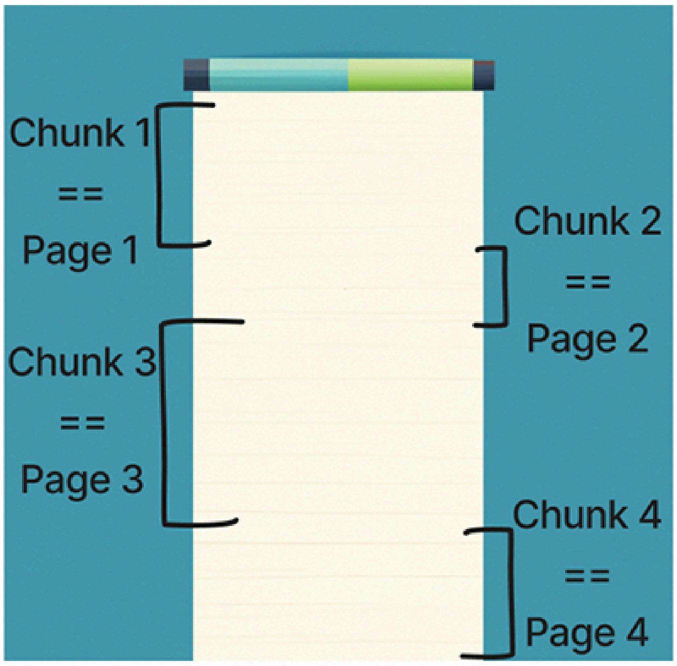
```
# Importing the Counter and re libraries
from collections import Counter
import re

# Find all occurrences of one or more spaces in 'principles_of_ds'
matches = re.findall(r'[\s]{1,}', principles_of_ds)

# The 5 most frequent spaces that occur in the document
most_common_spaces = Counter(matches).most_common(5)

# Print the most common spaces and their frequencies
print(most_common_spaces)

[(' ', 82259),
 ('\n', 9220),
 (' ', 1592),
 ('\n\n', 333),
 ('\n ', 250)]
```

The most common double whitespace is two newline characters in a row, which is actually how I earlier distinguished between pages. That makes sense because the most natural whitespace in a book is by page. In other cases, we may have found natural whitespace between paragraphs as well. This method is very hands-on and requires a good amount of familiarity with and knowledge of the source documents.

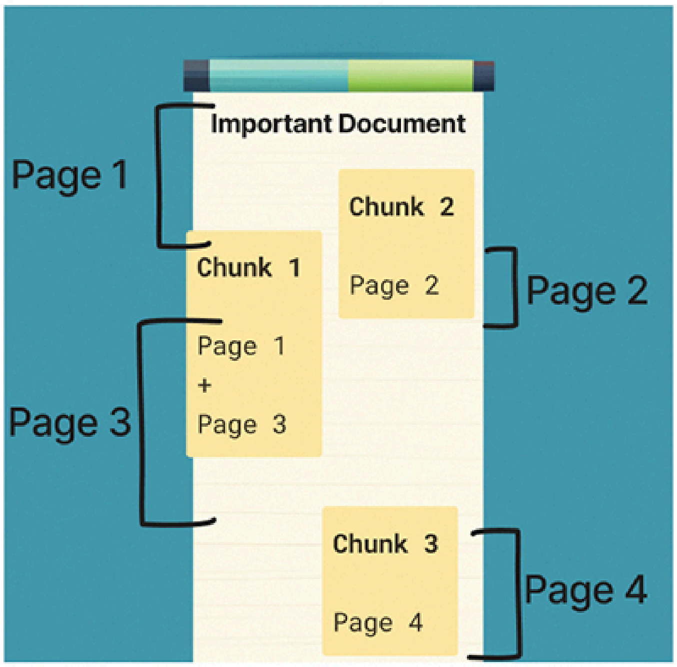
We can also turn to more machine learning to get slightly more creative with how we architect document chunks.

Using Clustering to Create Semantic Documents

Another approach to document chunking is to use clustering to create semantic documents. This approach involves creating new documents by combining small chunks of information that are semantically similar (Figure 2.8). It requires some creativity, as any modifications to the document chunks will alter the resulting vector. We could use an instance of agglomerative clustering from scikit-learn, for example, where similar sentences or paragraphs are grouped together to form new documents.



Natural Whitespace
Chunking with No Overlap



Grouping Natural Chunks by
Semantic Similarity

Figure 2.8 We can group any kinds of document chunks together by using some separate semantic clustering system (shown on the right) to create brand-new documents with chunks of information in them that are similar to each other.

Let’s try to cluster together those chunks we found from the textbook in our last section (Listing 2.6).

Listing 2.6 **Clustering pages of the document by semantic similarity**

Click here to view code image

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Assume you have a list of text embeddings called 'embeddings'
# First, compute the cosine similarity matrix between all pairs of embeddings
cosine_sim_matrix = cosine_similarity(embeddings)

# Instantiate the AgglomerativeClustering model
agg_clustering = AgglomerativeClustering(
 n_clusters=None, # The algorithm will determine the optimal number of cluster
on the data
 distance_threshold=0.1, # Clusters will be formed until all pairwise distance
between clusters are greater than 0.1
 affinity='precomputed', # We are providing a precomputed distance matrix
(1 - similarity matrix) as input
 linkage='complete' # Form clusters by iteratively merging the smallest cluste
on the maximum distance between their components
)

# Fit the model to the cosine distance matrix (1 - similarity matrix)
agg_clustering.fit(1 - cosine_sim_matrix)

# Get the cluster labels for each embedding
cluster_labels = agg_clustering.labels_

# Print the number of embeddings in each cluster
unique_labels, counts = np.unique(cluster_labels, return_counts=True)
for label, count in zip(unique_labels, counts):
 print(f'Cluster {label}: {count} embeddings')

Cluster 0: 2 embeddings
Cluster 1: 3 embeddings
Cluster 2: 4 embeddings
...
```

This approach tends to yield chunks that are more cohesive semantically but suffer from pieces of content being out of context with the surrounding text. It works well when the chunks you start with are known to not necessarily relate to each other—that is, when chunks are more independent of one another.

Use Entire Documents Without Chunking

Alternatively, it is possible to use entire documents without chunking. This approach is probably the easiest option overall but has drawbacks when the document is far too long, and we hit a context window limit when we embed the text. We also might fall victim to the document being filled with extraneous disparate context points, and the resulting embeddings may be trying to encode too much and suffer in quality. These drawbacks compound for very large (multi-page) documents.

It is important to consider the trade-offs between chunking and using entire documents when selecting an approach for document embedding (Table 2.1). Once we decide how we want to chunk our documents, we need a home for the embeddings we create. Locally, we can rely on matrix operations for quick retrieval. However, we are building for the cloud here, so let's look at our database options.

Table 2.1 **Outlining Different Document Chunking Methods with Pros and Cons**

Type of Chunking	Description	Pros	Cons
Max token window chunking with no overlap	The document is split into fixed-size windows, with each window representing a separate document chunk.	Simple and easy to implement.	May cut off context in between chunks, resulting in loss of information.
Max token window chunking with overlap	The document is split into fixed-size overlapping windows.	Simple and easy to implement.	May result in redundant information across different chunks.
Chunking on natural delimiters	Natural whitespace in the document is used to determine the boundaries of each chunk.	Can result in more meaningful chunks that correspond to natural breaks in	May be time-consuming to find the right delimiters.

Type of Chunking	Description	Pros	Cons
		the document.	
Clustering to create semantic documents	Similar document chunks are combined to form larger semantic documents.	Can create more meaningful documents that capture the overall meaning of the document.	Requires more computational resources and may be more complex to implement.
Use entire documents without chunking	The entire document is treated as a single chunk.	Simple and easy to implement.	May suffer from a context window for embedding, resulting in extraneous context that affects the quality of the embedding.

Vector Databases

A **vector database** is a data storage system that is specifically designed to both store and retrieve vectors quickly. This type of database is useful for storing the embeddings generated by an LLM that encode and store the semantic meaning of our documents or chunks of documents. By storing embeddings in a vector database, we can efficiently perform nearest-neighbor searches to retrieve similar pieces of text based on their semantic meaning.

Pinecone is a vector database that is designed for small to medium-sized datasets (usually ideal for fewer than 1 million entries). It is easy to get started with Pinecone for free, but it also has a pricing plan that provides additional features and increased scalability. Pinecone is optimized for fast vector search and retrieval, making it a great choice for applications that require low-latency search, such as recommendation systems, search engines, and chatbots.

Several open-source alternatives to Pinecone can be used to build a vector database for LLM embeddings. One such alternative is Pgvector, a

PostgreSQL extension that adds support for vector data types and provides fast vector operations. Another option is Weaviate, a cloud-native, open-source vector database that is designed for machine learning applications. Weaviate provides support for semantic search and can be integrated with other machine learning tools such as TensorFlow and PyTorch. ANNOY is an open-source library for approximate nearest-neighbor searching that is optimized for large-scale datasets. It can be used to build a custom vector database that is tailored to specific use cases.

Re-ranking the Retrieved Results

After retrieving potential results from a vector database given a query using a similarity comparison (e.g., cosine similarity), it is often useful to re-rank them to ensure that the most relevant results are presented to the user (**Figure 2.9**). One way to re-rank results is by using a cross-encoder, a type of Transformer model that takes pairs of input sequences and predicts a score indicating how relevant the second sequence is to the first. By using a cross-encoder to re-rank search results, we can take into account the entire query context rather than just individual keywords. Of course, this will add some overhead and worsen our latency, but it could also help improve performance. In a later section, we'll compare and contrast using versus not using a cross-encoder to see how these approaches measure up.

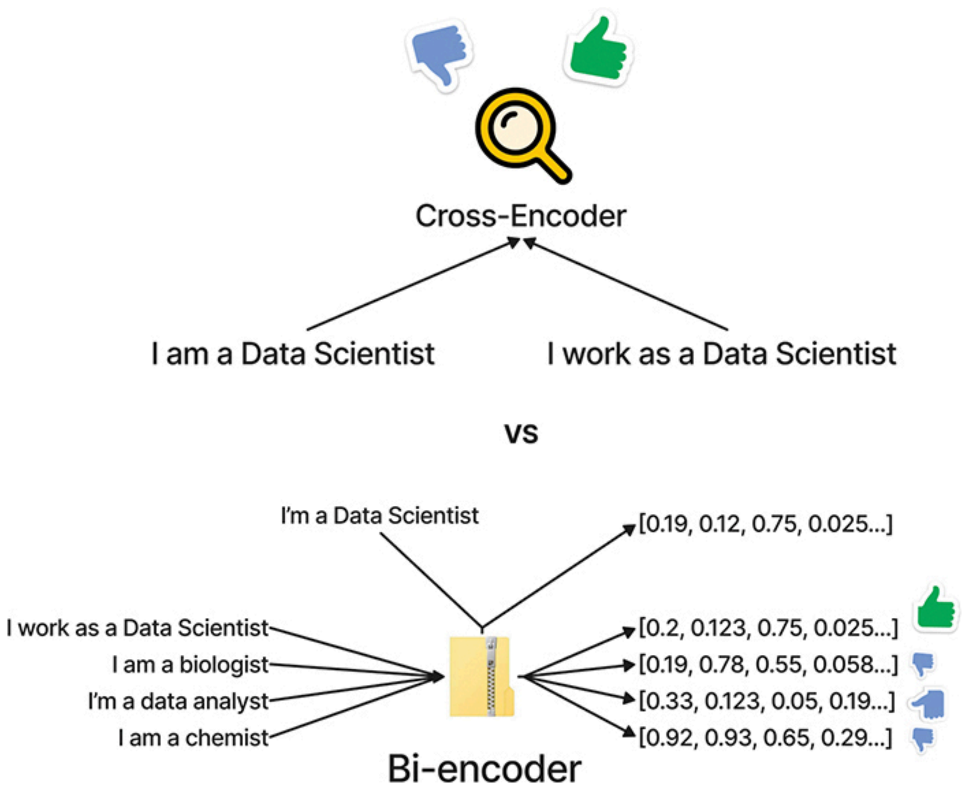


Figure 2.9 A cross-encoder takes in two pieces of text and outputs a similarity score without returning a vectorized format of the text. A bi-encoder embeds a bunch of pieces of text into vectors up front and then retrieves them later in real time given a query (e.g., looking up "I'm a Data Scientist").

One popular source of cross-encoder models is the Sentence Transformers library, which is where we found our bi-encoders earlier. We can also fine-tune a pre-trained cross-encoder model on our task-specific dataset to improve the relevance of the search results and provide more accurate recommendations.

Another option for re-ranking search results is by using a traditional retrieval model like BM25 (BM stands for "best-matching"), which ranks results by the frequency of query terms in the document and considers term proximity and inverse document frequency. BM25, which was originally developed in the 1970s, does not consider the entire query context, but it can still be a useful way to re-rank search results and improve the overall relevance of the results. A great implementation of BM25 can be found here: **https://pypi.org/project/rank-bm25**.

API

We now need a place to put all of these components so that users can access the documents in a fast, secure, and easy way. To do this, let's create an API.

FastAPI

FastAPI is a web framework for building APIs with Python relatively quickly. It is designed to be both fast and easy to set up, making it an excellent choice for our semantic search API. FastAPI uses the Pydantic data validation library to validate request and response data, and it is considered to be one of the most high-performance web frameworks in Python.

Setting up a FastAPI project is straightforward and requires minimal configuration. FastAPI provides automatic documentation generation with the OpenAPI standard, which makes it easy to build API documentation and client libraries. **Listing 2.7** is a skeleton of what that file would look like.

Listing 2.7 **FastAPI skeleton code**

Click here to view code image

```
import hashlib
import os
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

openai.api_key = os.environ.get('OPENAI_API_KEY', '')
pinecone_key = os.environ.get('PINECONE_KEY', '')

# Create an index in Pinecone with the necessary properties

def my_hash(s):
 # Return the MD5 hash of the input string as a hexadecimal string
 return hashlib.md5(s.encode()).hexdigest()


class DocumentInputRequest(BaseModel):
 # Define input to /document/ingest

class DocumentInputResponse(BaseModel):
 # Define output from /document/ingest

class DocumentRetrieveRequest(BaseModel):
 # Define input to /document/retrieve

class DocumentRetrieveResponse(BaseModel):
 # Define output from /document/retrieve


# API route to ingest documents
@app.post("/document/ingest", response_model=DocumentInputResponse)
async def document_ingest(request: DocumentInputRequest):
 # Parse request data and chunk it
 # Create embeddings and metadata for each chunk
 # Upsert embeddings and metadata to Pinecone
 # Return number of upserted chunks
 return DocumentInputResponse(chunks_count=num_chunks)


# API route to retrieve documents
@app.post("/document/retrieve", response_model=DocumentRetrieveResponse)
async def document_retrieve(request: DocumentRetrieveRequest):
 # Parse request data and query Pinecone for matching embeddings
 # Sort results based on re-ranking strategy, if any
 # Return a list of document responses
 return DocumentRetrieveResponse(documents=documents)
```

```
  if __name__ == "__main__":
    uvicorn.run("api:app", host="0.0.0.0", port=8000, reload=True)
```

For the full file, be sure to check out the code repository for this book.

Putting It All Together

We now have a solution for all of our components. Let's look at where we are in our solution. Items in bold are new from the last time we outlined this solution.

- Part I: Ingesting documents
  1. Collect documents for embedding—**Chunk any document to make it more manageable**
  2. Create text embeddings to encode semantic information—**OpenAI's Embeddings**
  3. Store embeddings in a database for later retrieval given a query—**Pinecone**
- Part II: Retrieving documents
  1. The user has a query that may be preprocessed and cleaned—**FastAPI**
  2. Retrieve candidate documents—**OpenAI's Embeddings + Pinecone**
  3. Re-rank the candidate documents if necessary—**Cross-encoder**
  4. Return the final search results—**FastAPI**

With all these moving parts, let's take a look at our final system architecture in **Figure 2.10**.

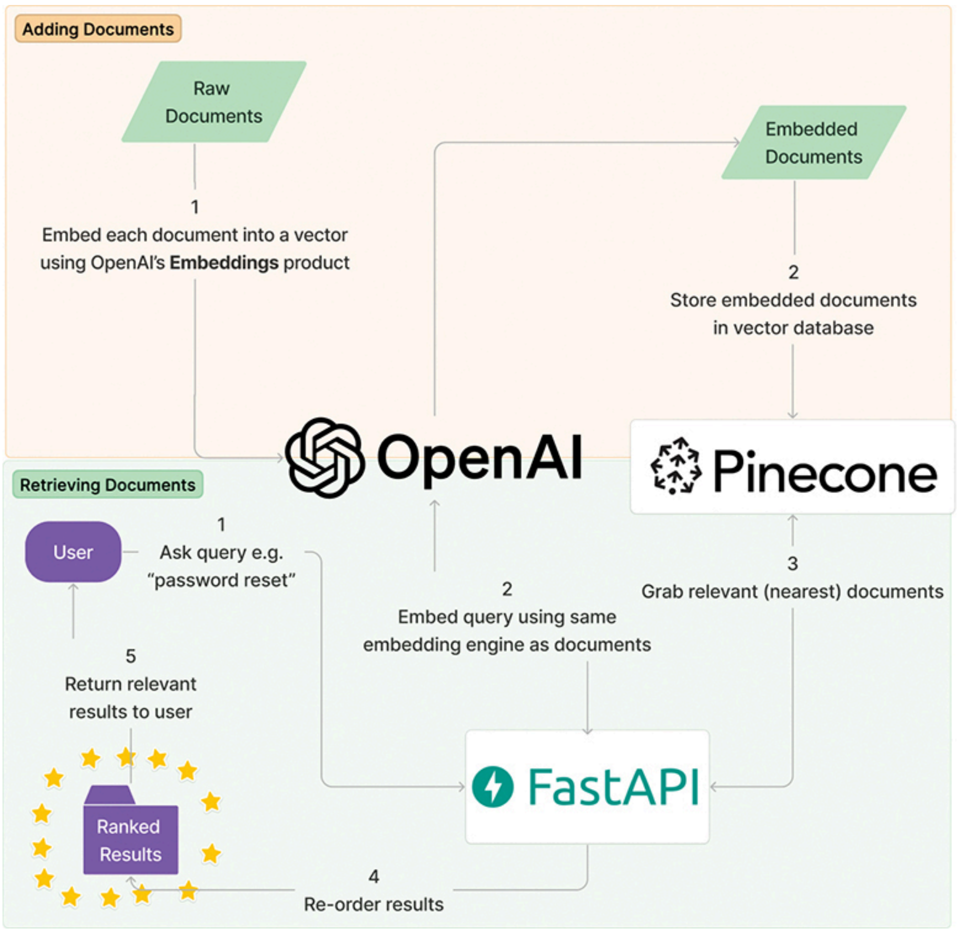

Adding Documents
Raw Documents
1
Embed each document into a vector using OpenAI's **Embeddings** product
Embedded Documents
2
Store embedded documents in vector database
OpenAI
Pinecone
Retrieving Documents
User
1
Ask query e.g. "password reset"
2
Embed query using same embedding engine as documents
3
Grab relevant (nearest) documents
FastAPI
5
Return relevant results to user
Ranked Results
4
Re-order results

Figure 2.10 Our complete semantic search architecture using two closed-source systems (OpenAI and Pinecone) and an open-source API framework (FastAPI).

We now have a complete end-to-end solution for our semantic search. Let's see how well the system performs against a validation set.

Performance

I've outlined a solution to the problem of semantic search, but I also want to talk about how to test how these different components work together. For this purpose, let's use a well-known benchmark to run the tests against: the XTREME benchmark—a multitask question-answering dataset for yes/no questions containing about 12,000 English examples. This dataset contains (question, passage) pairs that indicate, for a given question, whether that passage would be the best passage to answer the question. Listing 2.8 shows a code snippet for loading up the dataset.

Listing 2.8 **Getting text embeddings from OpenAI**

Click here to view code image

```
from datasets import load_dataset

dataset = load_dataset("xtreme", "MLQA.en.en")

# rename test -> train and val -> test (as we will use it in later in this chap
dataset['train'] = dataset['test']
dataset['test'] = dataset['validation']

print(f"Context: {dataset['train'][0]['context']}")
print(f"Question: {dataset['train'][0]['question']}")
print(f"Answers: {dataset['train'][0]['answers'][ 'text']}")

Context: 'In 1994, five unnamed civilian contractors and the…'
Question: 'Who analyzed the biopsies?'
Answers: ['Rutgers University biochemists']
```

Table 2.2 outlines a few trials that I ran and coded for this experiment. I used combinations of embedders, re-ranking solutions, and some fine-tuning of the cross-encoder to see how well the system performed as indicated by the "Top Result Accuracy" column.

Note

I know we haven't discussed the details of fine-tuning LLMs yet. We will see our first full fine-tuning example in Chapter 5. For now, I just want to sneak in these results to give a motivating example of when fine-tuning tends to work in our favor. Tasks in a specific domain (e.g., medical, financial, legal) are prime candidates for fine-tuning as we are moving away from "foundational" knowledge in most off-the-shelf LLMs and toward a task-specific implementation. Even without fine-tuning, our pre-trained cross-encoder boosted our performance, but with fine-tuning, we saw even more performance squeezed out of our model.

Table 2.2 **Performance Results from Various Combinations Against a Subset of the XTREME Benchmark**

Embedder (CS = closed source, OS = open source)	Re-ranking Method	Top Result Accuracy	Notes
OpenAI (CS)	None	0.754	Easiest to run by far

Embedder (CS = closed source, OS = open source)	Re-ranking Method	Top Result Accuracy	Notes
OpenAI (CS)	`ms-marco-MiniLM-L-12-v2` (OS; no fine-tuning)	0.833	A decent accuracy boost from the pre-trained cross-encoder
OpenAI (CS)	`ms-marco-MiniLM-L-12-v2` (OS; with fine-tuning)	**0.849**	A slight boost in accuracy post fine-tuning the cross-encoder
`all-mpnet-base-v2` (OS)	None	0.502	Vastly underperforms OpenAI's embedding engine on this dataset
`all-mpnet-base-v2` (OS)	`ms-marco-MiniLM-L-12-v2` (OS; with fine-tuning)	0.619	An accuracy boost but not enough to catch up to OpenAI

For each known pair of (question, passage) in our XTREME validation set, we test if the system's top result is the intended passage. If we are not using a cross-encoder, the top result is simply the passage with the highest cosine similarity to the query given the embedding engine. If we are using a cross-encoder, I retrieved *50* results from the vector database and re-ranked them using the cross-encoder and used its final ranking as opposed to the embedding engine's ranking.

A reminder: The full code base can be found on our GitHub. This chapter would double in size if we included all of the code here! Here are the key takeaways from the experiment:

- A combination of closed-source and open-source models won the day: OpenAI for embedding and an open-source cross-encoder for re-ranking.
- Our open-source embedder did not perform as well as OpenAI on this specific dataset.

- Fine-tuning our cross-encoder yielded marginally better results over using the off-the-shelf model.

Some experiments I didn't try include the following:

- Fine-tuning the cross-encoder for more epochs and spending more time finding optimal learning parameters (e.g., weight decay, learning rate scheduler).
- Using other OpenAI embedding engines. (To be fair, I used the most expensive and most powerful one—according to OpenAI.)
- Fine-tuning an open-source bi-encoder on the training set. We will see an example of this in a later chapter while building a recommendation engine.

Table 2.2 shows only the results for looking for the top result accuracy, whereas Figure 2.11 provides a broader representation of our experiments by relaxing the requirements to look for the right document in the top 1, 3, 5, 10, 25, and 50 results. This process is called **recall** in semantic search: That is, are we able to "recall" the right document if we look for it in a list of retrieved results? In cases where we expect a model to create a short list to be reviewed by a human, it can be useful to see results in a more relaxed environment. In this case, our open-source embedder, which performed poorly in terms of the top result, is much closer to OpenAI's performance at the top 5 or top 10 result category.

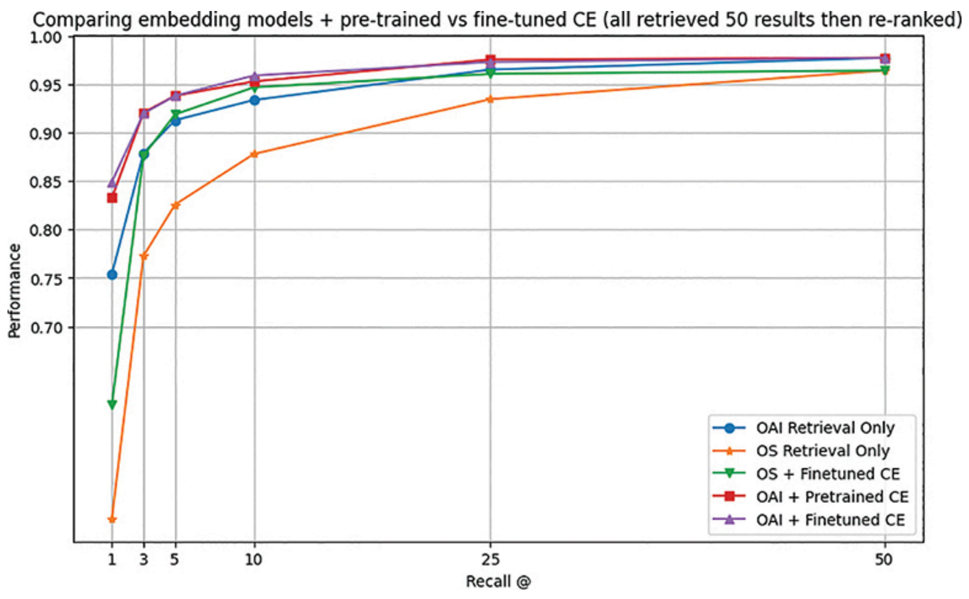


Figure 2.11 Measuring our ability to find the right document (i.e., recall) across five experiments in semantic search. Our open-source embedder underperforms by a bit if the list is short (1–3 examples) but starts to reach parity around 10–25 examples.

Note that the models I used for the cross-encoder and the bi-encoder were both specifically pre-trained on data in a way similar to asymmetric semantic search. This is important because we want the embedder to produce vectors for both short queries and long documents, and to place them near each other when they are related. Also note that it will not always be the case that the open-source embedder underperforms a closed-source model. Thus, we should compare models' performance on a test set on a per-test-set basis. In the first edition of this book, we used a different benchmark (BoolQ); in that edition, the open-source embedder performed slightly better than the OpenAI model!

Let's assume we want to keep things simple to get our project off the ground, so we'll use only the OpenAI embedder and do no re-ranking (row 1) in our application. We should now consider the costs associated with using FastAPI, Pinecone, and OpenAI for text embeddings.

The Cost of Closed-Source Components

We have a few components in play, and not all of them are free. Fortunately, FastAPI is an open-source framework and does not require any licensing fees. Our cost with FastAPI is that associated with hosting—which could be on a free tier depending on which service we use. I like Render, which has a free tier but also offers pricing starting at $7/month for 100% uptime. At the time of writing, Pinecone offers a free tier with a limit of 100,000 embeddings and up to 3 indexes; beyond that level, charges are based on the number of embeddings and indexes used. Pinecone's standard plan charges $49/month for up to 1 million embeddings and 10 indexes.

Let's assume that OpenAI charges $0.00013 per every 1000 tokens for the embedding engine we used (this was true as of May 2024 for text-embedding-3-large, the embedding we used in our example). If we assume an average of 500 tokens per document (roughly more than a page's worth of English writing), the cost per document would be $0.000065. For example, if we wanted to embed 1 million documents, it would cost approximately $65.

If we want to build a system with 1 million embeddings, and we expect to update the index once a month with totally fresh embeddings, the total cost per month would be:

Pinecone cost = $49

OpenAI cost = $65

FastAPI hosting cost = $7

Total cost = $49 + $65 + $7 = **$121/month**

These costs can quickly add up as the system scales. It may be worth exploring open-source alternatives or other strategies to reduce costs—such as using open-source bi-encoders for embedding or Pgvector as your vector database.

Summary

With all these components accounted for, our pennies added up, and alternatives available at every step of the way, I'll leave you to it. Enjoy setting up your new semantic search system, and be sure to check out the complete code for this—including a fully working FastAPI app with instructions on how to deploy it—on the book's code repository. You can experiment to your heart's content to make this solution work as well as possible for your domain-specific data.

Stay tuned for our next chapter, where we will build on this API with a chatbot based on GPT-4 and our retrieval system.