Chapter 5. Vector Databases with FAISS and Pinecone

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

This chapter introduces the concept of embeddings and vector databases, discussing how they can be used to provide relevant context in prompts.

A *vector database* is a tool most commonly used for storing text data in a way that enables querying based on similarity or semantic meaning. This technology is used to decrease hallucinations (where the AI model makes something up) by referencing data the model isn't trained on, significantly improving the accuracy and quality of the LLMs response. Use cases for vector databases also include reading documents, recommending similar products, or remembering past conversations.

Vectors are lists of numbers representing text (or images), which you might think of as coordinates for a location. The vector for the word `mouse` using OpenAI's `text-embedding-ada-002` model, is a list of 1,536 numbers, each representing the value for a feature the embedding model learned in training:

```
[-0.011904156766831875,
 -0.0323905423283577,
 0.001950666424818337,
...]
```

When these models are trained, texts that appear together in the training data will be pushed closer together in values, and texts that are unrelated will be pushed further away. Imagine we trained a simple model with only two parameters, `Cartoon` and `Hygiene`, that must describe the entire world, but only in terms of these two variables. Starting from the word `mouse`, increasing the value for the parameter `Cartoon` we would travel towards the most famous cartoon mouse, `mickey mouse`, as is shown in Figure 5-1. Decreasing the value for the `Hygiene` parameter would take us towards `rat`, because rats are rodents similar to mice, but are associated with plague and disease (i.e. being unhygenic).

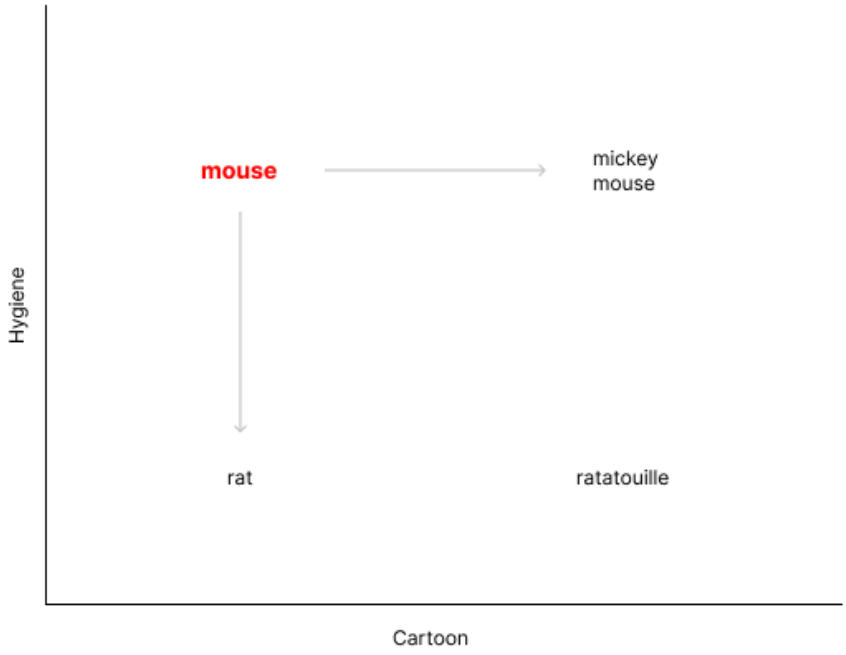

Figure 5-1. 2D Vector Distances

Each location on the graph can be found by two numbers on the X and Y axes, which represent the features of the model `Cartoon` and `Hygiene`. In reality, vectors can have thousands of parameters, because more parameters allows the model to capture a wider range of similarities and differences. Hygiene is not the only difference between mice and rats, and Mickey Mouse isn't just a cartoon mouse. These features are learned from the data in a way that makes them hard for humans to interpret, and we would need a graph with thousands of axes to display a location in *latent space* (the abstract multi-dimensional space formed by the model's parameters). Often there is no human-understandable explanation of what a feature means. However, we can create a simplified two-dimensional projection of the distances between vectors, as has been done in Figure 5-2.

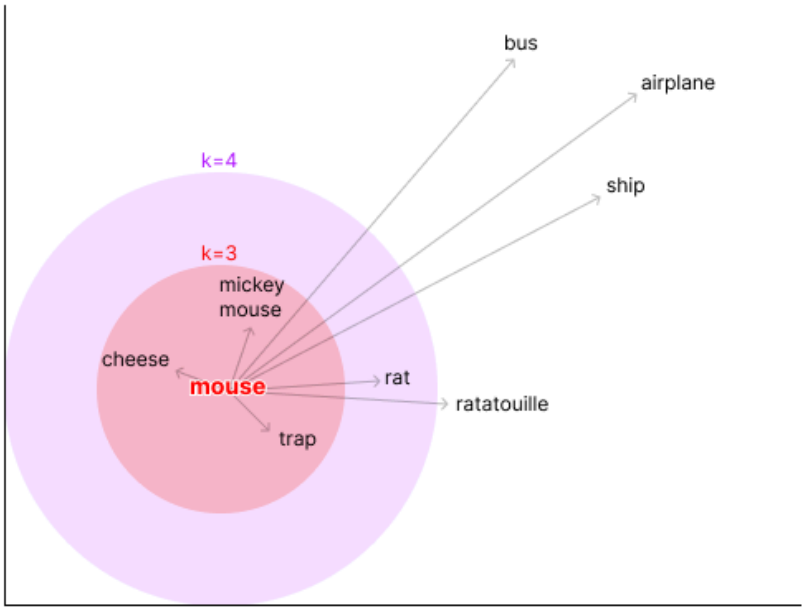

Figure 5-2. Multi-Dimension Vector Distances

To conduct a vector search, you first get the vector (or location) of what you want to look up, and find the `k` closest records in the database. In this case the word `mouse` is closest to `mickey mouse`, `cheese`, and `trap` where `k=3` (return the 3 nearest records). The word `rat` is excluded if `k=3`, but would be included if `k=4` as it is the next closest vector. The word `airplane` in this example is far away because it is rarely associated with the word `mouse` in the training data. The word `ship` is still co-located near the other forms of transport, but is closer to `mouse` and `rat` because they are often found on ships, as per the training data.

A vector database stores the text records with their vector representation as the key. This is unlike other types of database, where you might find records based on an ID, relation, or where the text contains a string. For example, if you queried a relational database based on the text in Figure 5-2 to find records where text contains `mouse`, you'd return the record `mickey mouse` but nothing else, as no other record contains that exact phrase. With vectors search you could also return the records `cheese` and `trap`, because they are closely associated, even though they aren't an exact match for your query.

The ability to query based on similarity is extremely useful, and vector search powers a lot of AI functionality. For example:

- *Document reading*: Find related sections of text to read in order to provide a more accurate answer.
- *Recommendation systems*: iscover similar products or items in order to suggest them to a user.

- *Long-term memory*: Look up relevant snippets of conversation history so a chatbot remembers past interactions.

AI models are able to handle these tasks at small scale, so long as your documents, product list, or conversation memory fits within the token limits of the model you're using. However, at scale you quite quickly run into token limits and excess cost from passing too many tokens in each prompt. OpenAI's `gpt-4-1106-preview` was released in November 2023 with an enormous 128,000 token context window, but it costs 10 times more per token than `gpt-3.5-turbo`, which has 88% fewer tokens and was released a year earlier. The more efficient approach is to look up only the most relevant records to pass into the prompt at runtime, in order to provide the most relevant context to form a response. This practice is typically referred to as RAG.

Retrieval Augmented Generation (RAG)

Vector databases are a key component of RAG, which typically involves searching by similarity to the query, retrieving the most relevant documents, and inserting them into the prompt as context. This lets you stay within what fits in the current context window, while avoiding spending money on wasted tokens by inserting irrelevant text documents in the context.

Retrieval can also be done using traditional database searches or web browsing, and in many cases a vector search by semantic similarity is not necessary. RAG is typically used to solve hallucinations in open-ended scenarios, like a user talking to a chatbot which is prone to making things up when asked about something not in its training data. Vector search can insert documents that are semantically similar to the user query into the prompt, greatly decreasing the chances the chatbot will hallucinate.

For example, if your author Mike told a chatbot "My name is Mike", then three messages later asked "What is my name?", it can easily recall the right answer. The message containing Mike's name is still within the context window of the chat. However, if it was 3,000 messages ago, the text of those messages may be too large to fit inside the context window. Without this important context, it might hallucinate a name, or refuse to answer for lack of information. A keyword search might help, but could return too many irrelevant documents or fail to recall the right context in which the information was captured in the past. There may be many times Mike mentioned the word `name`, in different formats, and for different reasons. By passing the question to the vector database, it can return the top three similar messages from the chat that match what the user asked:

```
## Context
Most relevant previous user messages:
1. "My name is Mike".
2. "My dog's name is Hercules".
3. "My coworker's name is James".

## Instructions
Please answer the user message using the context above.
User message: What is my name?
AI message:
```

It's impossible to pass all 3,000 past messages into the prompt for most models, and for a tradtional search the AI model would have to formulate the right search query, which can be unreliable. Using the RAG pattern, you would pass the current user message to a vector search function, and return the most relevant three records as context, which the chatbot can then use to respond correctly.

GIVE DIRECTION

Rather than inserting static knowledge into the prompt, vector search allows you to dynamically insert the most relevant knowledge into the prompt.

Here's how the process works for production applications using RAG:

1. Break documents into chunks of text.
2. Index chunks in a vector database.
3. Search by vector for similar records.
4. Insert records into the prompt as context.

In this instance the documents would be all the 3,000 past user messages to serve as the chatbot's memory, but it could also be sections of a PDF document we uploaded to give the chatbot the ability to read, or a list of all the relevant products you sell to enable the chatbot to make a recommendation. The ability of our vector search to find the most similar texts is wholly dependent on the AI model used to generate the vectors, referred to as embeddings when you're dealing with semantic or contextual information.

Introducing Embeddings

The word *embeddings* typically refers to the vector representation of the text returned from a pre-trained AI model. At the time of writing, the standard model for generating embeddings is OpenAI's `text-`

`embedding-ada-002`, although embedding models have been available long before the advent of generative AI.

Although it is helpful to visualize vector spaces as a two-dimensional chart, as is done in Figure 5-2, in reality the embeddings returned from `text-embedding-ada-002` are in 1,536 dimensions, which is difficult to depict graphically. More dimensions allows the model to capture deeper semantic meaning and relationships. For example, while a 2D space might be able to separate cats from dogs, a 300D space could capture information about the differences between breeds, sizes, colors, and other intricate details. The following code shows how to retrieve embeddings from the OpenAI API. The code for the following examples is included in the GitHub repository for the book:

Input:

```
from openai import OpenAI
client = OpenAI()

# Function to get the vector embedding for a given text
def get_vector_embeddings(text):
    response = client.embeddings.create(
        input=text,
        model="text-embedding-ada-002"
    )
    embeddings = [r.embedding for r in response.data]
    return embeddings[0]

get_vector_embeddings("Your text string goes here")
```

Output:

```
[
-0.006929283495992422,
-0.005536422007530928,
...
-4.547132266452536e-05,
-0.024047505110502243
]
```

This code uses the OpenAI API to create an embedding for a given input text using a specific embedding model:

1. `from openai import OpenAI` imports the OpenAI library and `client = OpenAI()` sets up the client. It retrieves your OpenAI API key from an environment variable `OPENAI_API_KEY`, in order

to charge the cost of the embeddings to your account. You need to set this in your environment (usually in a *.env* file), which can be obtained by creating an account and visiting *https://platform.openai.com/account/api-keys*.

2. `response = client.embeddings.create(...)`: This line calls the `create` method of the `Embedding` class from the `client` from the OpenAI library. The method takes two arguments:
    - `input`: This is where you provide the text string for which you want to generate an embedding.
    - `model`: This specifies the embedding model you want to use. In this case, it is `text-embedding-ada-002`, which is a model within the OpenAI API.
3. `embeddings = [r.embedding for r in response.data]`: After the API call, the `response` object contains the generated embeddings in JSON format. This line extracts the actual numerical embedding from the response, by iterating through a list of embeddings in `response.data`.

After executing this code, the `embeddings` variable will hold the numerical representation (embedding) of the input text, which can then be used in various *natural language processing* (NLP) tasks or machine learning models. This process of retrieving or generating embeddings is sometimes referred to as *document loading*.

The term *loading* in this context refers to the act of computing or retrieving the numerical (vector) representations of text from a model and storing them in a variable for later use. This is distinct from the concept of *chunking*, which typically refers to breaking down a text into smaller, manageable pieces or chunks to facilitate processing. These two techniques are regularly used in conjunction with each other, as it's often useful to break large documents up into pages or paragraphs in order to facillitate more accurate matching and to only pass the most relevant tokens into the prompt.

There is a cost associated with retrieving embeddings from OpenAI, but it is relatively inexpensive at $0.0004 per 1,000 tokens at the time of writing. For instance, the King James version of the Bible, which comprises around 800,000 words or approximately 4,000,000 tokens, would cost about $1.60 to retrieve all the embeddings for the entire document.

Paying for embeddings from OpenAI is not your only option. There are also open-source models you can use, for example, the Sentence Transformers library provided by Hugging Face, which has 384 dimensions:

Input:

```
import requests
import os

model_id = "sentence-transformers/all-MiniLM-L6-v2"
hf_token = os.getenv("HF_TOKEN")

api_url = "https://api-inference.huggingface.co/"
api_url += f"pipeline/feature-extraction/{model_id}"
headers = {"Authorization": f"Bearer {hf_token}"}

def query(texts):
    response = requests.post(api_url, headers=headers,
    json={"inputs": texts,
    "options":{"wait_for_model":True}})
    return response.json()

texts = ["mickey mouse",
        "cheese",
        "trap",
        "rat",
        "ratatouille"
        "bus",
        "airplane",
        "ship"]

output = query(texts)
output
```

Output:

```
[[-0.03875632584095001, 0.04480459913611412,
0.016051070764660835, -0.01789097487926483,
-0.03518553078174591, -0.013002964667975903,
0.14877274632453918, 0.048807501792907715,
0.011848390102386475, -0.044042471796274185,
...
-0.026688814163208008, -0.0359361357986927,
-0.03237859532237053, 0.008156519383192062,
-0.10299170762300491, 0.0790356695652008,
-0.008071334101259708, 0.11919838190078735,
0.0005506130401045084, -0.03497892618179321]]
```

This code uses the Hugging Face API to obtain embeddings for a list of text inputs using a pre-trained model. The model used here is the `sentence-transformers/all-MiniLM-L6-v2`, which is a smaller version of BERT, an open-source NLP model introduced by Google in 2017

(based on the transformer model), which is optimized for sentence-level tasks. Here's how it works step-by-step:

1. `model_id` is assigned the identifier of the pre-trained model, `sentence-transformers/all-MiniLM-L6-v2`.
2. `hf_token = os.getenv("HF_TOKEN")` retrieves the API key for the Hugging Face API token from your environment. You need to set this in your environment with your own token, which can be obtained by creating an account and visiting *https://hf.co/settings/tokens*.
3. The `requests` library is imported to make HTTP requests to the API.
4. `api_url` is assigned the URL for the Hugging Face API, with the model ID included in the URL.
5. `headers` is a dictionary containing the authorization header with your Hugging Face API token.
6. The `query()` function is defined, which takes a list of text inputs and sends a POST request to the Hugging Face API with the appropriate headers and JSON payload containing the inputs and an option to wait for the model to become available. The function then returns the JSON response from the API.
7. `texts` is a list of strings from your database.
8. `output` is assigned the result of calling the `query()` function with the `texts` list.
9. The `output` variable is printed, which will display the feature embeddings for the input texts.

When you run this code, the script will send text to the Hugging Face API, and the API will return embeddings for each string of text sent.

If you pass the same text into an embedding model you'll get the exact same vector back every time. However vectors are not usually comparable across models (or versions of models) due to differences in training. The embeddings you get from OpenAI are different from those you get from `BERT`, or `SpaCY` (a natural language processing library).

The main difference with embeddings generated by modern transformer models is that the vectors are contextual rather than static, meaning the word *bank* would have different embeddings in the context of a *river bank* versus *financial bank*. The embeddings you get from OpenAI Ada 002 and HuggingFace Sentence Transformers are examples of dense vectors, where each number in the array is almost always non-zero (i.e. they contain semantic information). There are also sparse vectors, which normally have a large number of dimensions (e.g. 100,000+) with many of the dimensions having a value of zero. This allows capturing specific impor-

tant features (each feature can have its own dimension) which tends to be important for performance in keyword-based search applications. Most AI applications use dense vectors for retrieval, although hybrid search (both dense and sparse vectors) is rising in popularity, as both similarity and keyword search can be useful in combination.

The accuracy of the vectors is wholly reliant on the accuracy of the model you use to generate the embeddings. Whatever biases or knowledge gaps the underlying models have will also be an issue for vector search. For example, the `text-embedding-ada-002` model is currently only trained up to August 2020, and therefore is unaware of any new words or new cultural associations that formed after that cut-off date. This can cause a problem for use cases that need more recent context, or niche domain knowledge not available in the training data, which may necessitate training a custom model.

In some instances it might make sense to train your own embedding model. For instance, you might do this if the text used has a domain-specific vocabulary where specific words have a meaning separate from the generally accepted meaning of the word. One example might be tracing the language used by toxic groups on social media like Q-Anon, who evolve the language they use in posts to bypass moderation actions.

Training your own embeddings can be done with tools like `Word2Vec`, a method to represent words in a vector space, enabling you to capture the semantic meanings of words. More advanced models may be used, like `GloVe` (Global Vectors for Word Representation), which is used by SpaCy for its embeddings, which are trained on the CommonCrawl dataset, an open-source snapshot of the web. The library `gensim` offers a simple process for training your own custom embeddings using the open source algorithm `Word2Vec`:

Input:

```
from gensim.models import Word2Vec

# Sample data: list of sentences, where each sentence is
# a list of words.
# In a real-world scenario, you'd load and preprocess your
# own corpus.
sentences = [
    ["the", "cake", "is", "a", "lie"],
    ["if", "you", "hear", "a", "turret", "sing", "you're",
    "probably", "too", "close"],
    ["why", "search", "for", "the", "end", "of", "a",
    "rainbow", "when", "the", "cake", "is", "a", "lie?"],
    # ...
```

```
    ["there's", "no", "cake", "in", "space,", "just", "ask",
    "wheatley"],
    ["completing", "tests", "for", "cake", "is", "the",
    "sweetest", "lie"],
    ["I", "swapped", "the", "cake", "recipe", "with", "a",
    "neurotoxin", "formula,", "hope", "that's", "fine"],
] + [
    ["the", "cake", "is", "a", "lie"],
    ["the", "cake", "is", "definitely", "a", "lie"],
    ["everyone", "knows", "that", "cake", "equals", "lie"],
    # ...
] * 10  # repeat several times to emphasize


# Train the Word2Vec model
model = Word2Vec(sentences, vector_size=100, window=5,
min_count=1, workers=4, seed=36)

# Save the model
model.save("custom_word2vec_model.model")

# To load the model later
# loaded_model = Word2Vec.load(
# "custom_word2vec_model.model")

# Get vector for a word
vector = model.wv['cake']

# Find most similar words
similar_words = model.wv.most_similar("cake", topn=5)
print("Top 5 most similar words to 'cake': ", similar_words)

# Directly query the similarity between "cake" and "lie"
cake_lie_similarity = model.wv.similarity("cake", "lie")
print("Similarity between 'cake' and 'lie': ",
cake_lie_similarity)
```

Output:

```
Top 5 most similar words to 'cake':  [('lie',
0.23420444130897522), ('test', 0.23205122351646423),
('tests', 0.17178669571876526), ('GLaDOS',
0.1536172330379486), ('got', 0.14605288207530975)]
Similarity between 'cake' and 'lie':  0.23420444
```

This code creates a `Word2Vec` model using the `Gensim` library and then using the model to determine words that are similar to a given word. Let's break it down:

1. The variable `sentences` contains a list of sentences, where each sentence is a list of words. This is the data on which the Word2Vec model will be trained. In a real application, instead of such hardcoded sentences, you'd often load a large corpus of text and preprocess it to obtain such a list of tokenized sentences.
2. An instance of the `Word2Vec` class is created to represent the model. While initializing this instance, several parameters are provided:
    - `sentences` : This is the training data.
    - `vector_size=100` : This defines the size of the word vectors. So each word will be represented as a 100-dimensional vector.
    - `window=5` : This represents the maximum distance between the current and predicted word within a sentence.
    - `min_count=1` : This ensures that even words that appear only once in the dataset will have vectors created for them.
    - `workers=4` : Number of CPU cores to use during training. It speeds up training on multi-core machines.
    - `seed=36` : This is set for reproduceability, so that the random processes in training deliver the same result each time (not guaranteed with multiple workers).
3. After training, the model is saved to a file named `custom_word2vec_model.model` using the `save` method. This allows you to reuse the trained model later without needing to train it again.
4. There is a commented-out line that shows how to load the model back from the saved file. This is useful when you want to load a pretrained model in a different script or session.
5. The variable `vector` is assigned the vector representation of the word *cake*. This vector can be used for various purposes, like similarity calculations, arithmetic operations, etc.
6. The `most_similar` method is used to find words that are most similar to the provided vector (in this case, the vector for *cake*). The method returns the top five (`topn=5`) most similar words.
7. The `similarity` method queries the similarity between *cake* and *lie* direction, showing a small positive value.

The dataset is small and heavily repetitive, which might not provide a diverse context to properly learn the relationship between the words. Normally, Word2Vec benefits from larger and more diverse corpora and typically won't get good results until you're into the tens of millions of words. In the example we set a seed value to cherrypick one instance where *lie* came back in the top five results, but if you remove that seed you'll find it rarely discovers the association successfully.

For smaller document sizes a simpler technique *TF-IDF* (Term Frequency-Inverse Document Frequency)is recommended, a statistical measure used

to evaluate the importance of a word in a document relative to a collection of documents. The TF-IDF value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the wider corpus, which helps to adjust for the fact that some words are generally more common than others.

To compute the similarity between *cake* and *lie* using TF-IDF, you can use the open-source scientific library `scikit-learn`, and compute the *cosine similarity* (a measure of distance between two vectors). Words that are frequently co-located in sentences will have high cosine similarity (approaching 1), whereas words that appear infrequently will show a low value (or 0, if not co-located at all). This method is robust to even small documents like our toy example:

Input:

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Convert sentences to a list of strings for TfidfVectorizer
document_list = [' '.join(s) for s in sentences]

# Compute TF-IDF representation
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(document_list)

# Extract the position of the words "cake" and "lie" in
# the feature matrix
cake_idx = vectorizer.vocabulary_['cake']
lie_idx = vectorizer.vocabulary_['lie']

# Extract and reshape the vector for 'cake'
cakevec = tfidf_matrix[:, cake_idx].toarray().reshape(1, -1)

# Compute the cosine similarities
similar_words = cosine_similarity(cakevec, tfidf_matrix.T).flatten()

# Get the indices of the top 6 most similar words
# (including 'cake')
top_indices = np.argsort(similar_words)[-6:-1][::-1]

# Retrieve and print the top 5 most similar words to
# 'cake' (excluding 'cake' itself)
names = []
for idx in top_indices:
    names.append(vectorizer.get_feature_names_out()[idx])
print("Top 5 most similar words to 'cake': ", names)

# Compute cosine similarity between "cake" and "lie"
```

```
similarity = cosine_similarity(np.asarray(tfidf_matrix[:,
    cake_idx].todense()), np.asarray(tfidf_matrix[:, lie_idx].todense()))
# The result will be a matrix; we can take the average or
# max similarity value
avg_similarity = similarity.mean()
print("Similarity between 'cake' and 'lie'", avg_similarity)

# Show the similarity between "cake" and "elephant"
elephant_idx = vectorizer.vocabulary_['sing']
similarity = cosine_similarity(np.asarray(tfidf_matrix[:,
    cake_idx].todense()), np.asarray(tfidf_matrix[:,
    elephant_idx].todense()))
avg_similarity = similarity.mean()
print("Similarity between 'cake' and 'sing'",
    avg_similarity)
```

Output:

```
Top 5 most similar words to 'cake':  ['lie', 'the', 'is',
'you', 'definitely']
Similarity between 'cake' and 'lie' 0.8926458157227388
Similarity between 'cake' and 'sing' 0.010626735901461177
```

Let's break down this code step by step:

1. The `sentences` variable is re-used from the previous example. The code converts these lists of words into full sentences (strings) using a list comprehension, resulting in `document_list`.
2. An instance of `TfidfVectorizer` is created. The `fit_transform` method of the vectorizer is then used to convert the `document_list` into a matrix of TF-IDF features, which is stored in `tfidf_matrix`.
3. The code extracts the position (or index) of the words *cake* and *lie* in the feature matrix using the `vocabulary_` attribute of the vectorizer.
4. The TF-IDF vector corresponding to the word *cake* is extracted from the matrix and reshaped.
5. The cosine similarity between the vector for *cake* and all other vectors in the TF-IDF matrix is computed. This results in a list of similarity scores.
    - The indices of the top six most similar words (including *cake*) are identified.
    - Using these indices, the top five words (excluding *cake*) with the highest similarity to *cake* are retrieved and printed.
6. The cosine similarity between the TF-IDF vectors of the words *cake* and *lie* is computed. Since the result is a matrix, the code computes

the mean similarity value across all values in this matrix, and then prints the average similarity.
7. Now we compute the similarity between *cake* and *sing*. The average similarity value is calculated and printed to show that the two words are not commonly co-located (close to zero).

As well as the embedding model used, the strategy for what you embed is also important, because there is a tradeoff between context and similarity. If you embed a large block of text, say an entire book, the vector you get back will be the average of the locations of the tokens that make up the full text. As you increase the size of the chunk, there is a regression to the mean where it approaches the average of all the vectors, and no longer contains much semantic information.

Smaller chunks of text will be more specific in terms of location in vector space, and as such might be more useful when you need close similarity. For example, isolating smaller sections of text from a novel may better separate comedic from tragic moments in the story, whereas embedding a whole page or chapter may mix both together. However, making the chunks of text too small might also cause them to lose meaning if the text is cut off in the middle of a sentence or paragraph. Much of the art of working with vector databases is in the way you load the document and break it into chunks.

Document Loading

One common use case of AI is to be able to search across documents based on similarity to the text of the user query. For example, you may have a series of PDFs representing your employee handbook, and you want to return the correct snippet of text from those PDFs that relates to an employee question. The way you load documents into your vector database will be dictated by the structure of your documents, how many examples you want to return from each query, as well as the number of tokens you can afford in each prompt.

For example, `gpt-4-0613` has an 8,192 token limit, which needs to be shared between the prompt template, the examples inserted into the prompt, and the completion the model provides in response. Setting aside around 2,000 words or approximately 3,000 tokens for the prompt and response, you could pull the five most similar chunks of 1,000 tokens of text each into the prompt as context. However, if you naively split the document into 1,000 token chunks, you will run into a problem. The arbitrary place where each split takes place might be in the middle of a paragraph or sentence, so you risk losing the meaning of what's being conveyed.

LangChain has a series of text splitters, including the commonly used recursive character text splitter. It tries to split on line breaks then spaces until the chunks are small enough. This keeps all paragraphs (and then sentences, and then words) together much as possible, to retain semantic groupings inherent in the structure of the text:

Input:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=100, # 100 tokens
    chunk_overlap=20, # 20 tokens of overlap
    )

text = """
Welcome to the "Unicorn Enterprises: Where Magic Happens"
Employee Handbook! We're thrilled to have you join our team
of dreamers, doers, and unicorn enthusiasts. At Unicorn
Enterprises, we believe that work should be as enchanting as
it is productive. This handbook is your ticket to the
magical world of our company, where we'll outline the
principles, policies, and practices that guide us on this
extraordinary journey. So, fasten your seatbelts and get
ready to embark on an adventure like no other!

...

As we conclude this handbook, remember that at Unicorn
Enterprises, the pursuit of excellence is a never-ending
quest. Our company's success depends on your passion,
creativity, and commitment to making the impossible
possible. We encourage you to always embrace the magic
within and outside of work, and to share your ideas and
innovations to keep our enchanted journey going. Thank you
for being a part of our mystical family, and together, we'll
continue to create a world where magic and business thrive
hand in hand!
"""

chunks = text_splitter.split_text(text=text)
print(chunks[0:3])
```

Output:

```
['Welcome to the "Unicorn Enterprises: Where Magic Happens"
Employee Handbook! We\'re thrilled to have you join our team
of dreamers, doers, and unicorn enthusiasts.',
"We're thrilled to have you join our team of dreamers,
```

```
doers, and unicorn enthusiasts. At Unicorn Enterprises, we
believe that work should be as enchanting as it is
productive.",
 ...
"Our company's success depends on your passion, creativity,
and commitment to making the impossible possible. We
encourage you to always embrace the magic within and outside
of work, and to share your ideas and innovations to keep our
enchanted journey going.",
"We encourage you to always embrace the magic within and
outside of work, and to share your ideas and innovations to
keep our enchanted journey going. Thank you for being a part
of our mystical family, and together, we'll continue to
create a world where magic and business thrive hand in
hand!"]
```

Here's how this code works step by step:

1. *Creating a text splitter instance*: An instance of `RecursiveCharacterTextSplitter` is created using the `from_tiktoken_encoder` method. This method is specifically designed to handle the splitting of text based on token counts.
    - *Chunk size*: The `chunk_size` parameter, set to 100, ensures that each chunk of text will contain approximately 100 tokens. This is a way of controlling the size of each text segment.
    - *Chunk overlap*: The `chunk_overlap` parameter, set to 20, specifies that there will be an overlap of 20 tokens between consecutive chunks. This overlap ensures that the context is not lost between chunks, which is crucial for understanding and processing the text accurately.
2. *Preparing the text*: The variable `text` contains a multi-paragraph string, representing the content to be split into chunks.
3. *Splitting the text*: The `split_text` method of the `text_splitter` instance is used to split the `text` into chunks based on the previously defined `chunk_size` and `chunk_overlap`. This method processes the text and returns a list of text chunks.
4. *Outputting the chunks*: The code prints the first three chunks of the split text to demonstrate how the text has been divided. This output is helpful for verifying that the text has been split as expected, adhering to the specified chunk size and overlap.

SPECIFY FORMAT

The relevance of the chunk of text provided to the prompt will depend heavily on your chunking strategy. Shorter chunks of text without overlap may not contain the right answer, whereas longer chunks of text with too much overlap may return too many irrelevant results and confuse the LLM, or cost you too many tokens.

Memory Retrieval with FAISS

Now that you have your documents processed into chunks, you need to store them in a vector database. It is common practice to store vectors in a database so that you do not need to re-compute them, as there is typically some cost and latency associated to doing so. If you don't change your embedding model, the vectors won't change, so you do not typically need to update them once stored. You can use an open-source library to store and query your vectors called FAISS, a library developed by Facebook AI that provides efficient similarity search and clustering of dense vectors. First install FAISS in the terminal with `pip install faiss-cpu`. The code for this example is included in the GitHub repository for the book:

Input:

```
import numpy as np
import faiss

#  The get_vector_embeddings function is defined in a preceding example
emb = [get_vector_embeddings(chunk) for chunk in chunks]
vectors = np.array(emb)

# Create a FAISS index
index = faiss.IndexFlatL2(vectors.shape[1])
index.add(vectors)

# Function to perform a vector search
def vector_search(query_text, k=1):
    query_vector = get_vector_embeddings(query_text)
    distances, indices = index.search(
        np.array([query_vector]), k)
    return [(chunks[i], float(dist)) for dist,
        i in zip(distances[0], indices[0])]

# Example search
user_query = "do we get free unicorn rides?"
search_results = vector_search(user_query)
print(f"Search results for {user_query}:", search_results)
```

Output:

```
Search results for do we get free unicorn rides?: [("You'll
enjoy a treasure chest of perks, including unlimited unicorn
rides, a bottomless cauldron of coffee and potions, and
access to our company library filled with spellbinding
books. We also offer competitive health and dental plans,
ensuring your physical well-being is as robust as your
magical spirit.\n\n**5: Continuous Learning and
Growth**\n\nAt Unicorn Enterprises, we believe in continuous
learning and growth.", 0.3289167582988739)]
```

Here is an explanation of the preceding code:

1. Import the Facebook AI Similarity Search (FAISS) library with `import faiss`.
2. `vectors = np.array([get_vector_embeddings(chunk) for chunk in chunks])` applies `get_vector_embeddings` to each element in `chunks`, which returns a vector representation (embedding) of each element. These vectors are then used to create a numpy array, which is stored in the variable `vectors`.
3. The line `index = faiss.IndexFlatL2(vectors.shape[1])` creates a FAISS index for efficient similarity search. The argument `vectors.shape[1]` is the dimension of the vectors that will be added to the index. This kind of index (`IndexFlatL2`) performs brute-force L2 distance search, which looks for the closest items to a particular item in a collection by measuring the straight-line distance between them, checking each item in the collection one by one.
4. Then you add the array of vectors to the created FAISS index with `index.add(vectors)`.
5. `def vector_search(query_text, k=1):` defines a new function named `vector_search` that accepts two parameters: `query_text` and `k` (with a default value of 1). The function will retrieve the embeddings for the `query_text`, then use that to search the index for the `k` closest vectors.
6. Inside the `vector_search` function, `query_vector = get_vector_embeddings(query_text)` generates a vector embedding for the query text using the `get_vector_embeddings` function.
7. The `distances, indices = index.search(np.array([query_vector]), k)` line performs a search in the FAISS index. It looks for the `k` closest vectors to `query_vector`. The method returns two arrays: `distances` (the squared L2 distances to the query vector) and `indices` (the indices of the closest vectors in the database).

8. `return [(chunks[i], float(dist)) for dist, i in zip(distances[0], indices[0])]` returns a list of tuples. Each tuple contains a chunk (retrieved using the indices found in the search) and the corresponding distance from the query vector. Note that the distance is converted to a float before returning.
9. Finally you perform a vector search for the string containing the user query: `search_results = vector_search(user_query)`. The result (the closest chunk and its distance) is stored in the variable `search_results`.

Once the vector search is complete, the results can be injected into the prompt to provide useful context. It's also important to set the system message so that the model is focused on answering based on the context provided rather than making an answer up. The RAG technique as demonstrated here is widely used in AI to help protect against hallucination:

Input:

```
# Function to perform a vector search and then ask # GPT-3.5-turbo a question
def search_and_chat(user_query, k=1):
  # Perform the vector search
  search_results = vector_search(user_query, k)
  print(f"Search results: {search_results}\n\n")

  prompt_with_context = f"""Context:{search_results}\
  Answer the question: {user_query}"""

  # Create a list of messages for the chat
  messages = [
      {"role": "system", "content": """Please answer the
      questions provided by the user. Use only the context
      provided to you to respond to the user, if you don't
      know the answer say \"I don't know\"."""},
      {"role": "user", "content": prompt_with_context},
  ]

  # Get the model's response
  response = client.chat.completions.create(
    model="gpt-3.5-turbo", messages=messages)

  # Print the assistant's reply
  print(f"""Response:
  {response.choices[0].message.content}""")

# Example search and chat
search_and_chat("What is Unicorn Enterprises' mission?")
```

Output:

```
Search results: [("As we conclude this handbook, remember that at Unicorn\nEnterpr
the pursuit of excellence is a never-ending\nquest. Our company's success depends
passion,\ncreativity, and commitment to making the impossible\npossible. We encour
to always embrace the magic\nwithin and outside of work, and to share your ideas
and\ninnovations to keep our enchanted journey going. Thank you", 0.26446571946144

Response:
Unicorn Enterprises' mission is to pursue excellence in their
work by encouraging their employees to embrace the magic within and outside of wor
their ideas and innovations, and make the impossible possible.
```

Here is a step-by-step explanation of what the function does:

1. Using a function named `vector_search`, the program performs a vector search with `user_query` as the search string and `k` as the number of search results to return. The results are stored in `search_results`.
2. The search results are then printed to the console.
3. A `prompt_with_context` is created by concatenating the `search_results` and `user_query`. The goal is to provide the model with context from the search results and a question to answer.
4. A list of messages is created. The first message is a system message that instructs the model to answer questions provided by the user using only the given context. If the model doesn't know the answer, it's advised to respond with *I don't know*. The second message is a user message containing the `prompt_with_context`.
5. The `openai.ChatCompletion.create()` function is used to get the model's response. It's provided with the model name (`gpt-3.5-turbo`) and the list of messages.
6. At the end of the code, the `search_and_chat()` function is called with the question as the `user_query`.

PROVIDE EXAMPLES

Without testing the writing style, it would be hard to guess which prompting strategy would win. Now you can be confident this is the correct approach.

Although our code is working end to end now, it doesn't make sense to be collecting embeddings and creating a vector database with every query. Even if you're using an open-source model for embeddings, there will be a cost in terms of compute and latency. You can save the FAISS index to a file using the `faiss.write_index` function:

```
# Save the index to a file
faiss.write_index(index, "data/my_index_file.index")
```

This will create a file called *my_index_file.index* in your current directory, which contains the serialized index. You can load this index back into memory later with `faiss.read_index`:

```
# Load the index from a file
index = faiss.read_index("data/my_index_file.index")
```

This way, you can persist your index across different sessions, or even share it between different machines or environments. Just make sure to handle these files carefully, as they can be quite large for big indexes.

If you have more than one saved vector database it's also possible to merge them together. This can be useful when serializing the loading of documents or making batch updates to you records.

You can merge two FAISS indices using the `faiss.IndexFlatL2` index's `add` method:

```
# Assuming index1 and index2 are two IndexFlatL2 indices
index1.add(index2.reconstruct_n(0, index2.ntotal))
```

In this code, `reconstruct_n(0, index2.ntotal)` is used to fetch all vectors from `index2`, and then `index1.add()` is used to add those vectors to `index1`, effectively merging the two indices.

This should work because `faiss.IndexFlatL2` supports the `reconstruct` method to retrieve vectors. However, please note that this process will not move any IDs associated with the vectors from `index2` to `index1`. After merging, the vectors from `index2` will have new IDs in `index1`.

If you need to preserve vector IDs, you'll need to manage this externally by keeping a separate mapping from vector IDs to your data items. Then, when you merge the indices, you also merge these mappings.

TIP

Be aware that this method may not work for all types of indices, especially for those which do not support the `reconstruct` method like `IndexIVFFlat` or if the two indices have different configurations. In those cases, it may be better to keep the original vectors used to build each index, and then merge and rebuild the index.

RAG with LangChain

As one of the most popular frameworks for AI engineering, LangChain has a wide coverage of RAG techniques. Other frameworks like LlamaIndex focus specifically on RAG, and are worth exploring for sophisticated use cases. As you are familiar with LangChain from Chapter 4, we'll continue in this framework for the examples in this chapter. After manually performing RAG based on a desired context, let's create a similar example using LCEL on four small text documents with FAISS:

```
from langchain_community.vectorstores.faiss import FAISS
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI, OpenAIEmbeddings

# 1. Create the documents:
documents = [
    "James Phoenix worked at JustUnderstandingData.",
    "James phoenix currently is 31 years old.",
    """Data engineering is the designing and building systems for collecting,
    storing, and analysing data at scale.""",
]

# 2. Create a vectorstore:
vectorstore = FAISS.from_texts(texts=documents, embedding=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()

# 3. Create a prompt:
template = """Answer the question based only on the following context:
---
Context: {context}
---
Question: {question}
"""
prompt = ChatPromptTemplate.from_template(template)

# 4. Create a chat model:
model = ChatOpenAI()
```

1. The code begins by importing necessary modules from the Langchain library and defines a list of text documents to be processed.
2. It utilizes `FAISS`, a library for efficient similarity search, to create a vector store from the text documents. This involves converting the texts into vector embeddings using OpenAI's embedding model.
3. A prompt template for handling questions and a `ChatOpenAI` model are initialized for generating responses. Additionally, the prompt template enforces that the LLM only replys using the context provided from the retriever.

You'll need to create an LCEL chain that will contain the `"context"` and `"question"` keys:

```
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

By adding a retriever to `"context"` it will automatically fetch four documents which are converted into a string value. Combined with the `"question"` key, these are then used to format the prompt. The LLM generates a response which is then parsed into a string value by `StrOutputParser()`.

You'll invoke the chain and pass in your question that gets assigned to `"question"` and manually test three different queries:

```
chain.invoke("What is data engineering?")
# 'Data engineering is the process of designing and building systems for
# collecting, storing, and analyzing data at scale.'

chain.invoke("Who is James Phoenix?")
# 'Based on the given context, James Phoenix is a 31-year-old individual who
# worked at JustUnderstandingData.'

chain.invoke("What is the president of the US?")
# I don't know
```

Notice how the LLM only appropriately answered the first two queries because it didn't have any relevant context contained within the vector database to answer the third query!

The Langchain implementation uses significantly less code, is easy to read, and allows you to rapidly implement retrieval augmented generation.

Hosted Vector Databases with Pinecone

There are a number of hosted vector database providers emerging to support AI use cases, including Chroma, Weaviate, and Pinecone. Hosts of other types of databases are also offering vector search functionality, such as Supabase with the pgvector addon. Examples in this book use Pinecone, as it is the current leader at the time of writing, but usage patterns are relatively consistent across providers and concepts should be transferrable.

Hosted vector databases offer several advantages over open-source local vector stores:

- *Maintainance*: With a hosted vector database, you don't need to worry about setting up, managing, and maintaining the database yourself. This can save significant time and resources, especially for businesses that may not have dedicated DevOps or database management teams.
- *Scalability*: Hosted vector databases are designed to scale with your needs. As your data grows, the database can automatically scale to handle the increased load, ensuring that your applications continue to perform efficiently.
- *Reliability*: Managed services typically offer high availability with service level agreements, as well as automatic backups and disaster recovery features. This can provide peace of mind and save you from potential data loss.
- *Performance*: Hosted vector databases often have optimized infrastructure and algorithms that can provide better performance than self-managed, open-source solutions. This can be particularly important for applications that rely on real-time or near-real-time vector search capabilities.
- *Support*: With a hosted service, you typically get access to support from the company providing the service. This can be very helpful if you experience issues or need help optimizing your use of the database.
- *Security*: Managed services often have robust security measures in place to protect your data, including things like encryption, access control, and monitoring. Major hosted providers are more likely to have the necessary compliance certificates and be in compliance with privacy legislation in regions like the EU.

Of course, this extra functionality comes at a cost, as well as a risk of overspending. As is the case with using Amazon Web Services, Microsoft Azure, or Google Cloud, stories of developers accidentally spending thousands of dollars through incorrect configuration or mistakes in code abound. There is also some risk of vendor lock-in, because although each vendor has similar functionality, they differ in certain areas, and as such it's not quite straightforward to migrate between them. The other major consideration is privacy, because sharing data with a third-party comes with security risks and potential legal implications.

The steps for working with a hosted vector database remain the same as when you set up your open-source FAISS vector store. First you chunk your documents and retrieve vectors, then you index your document chunks in the vector database, allowing you to retrieve similar records to our query, in order to insert into the prompt as context. First let's create an index in Pinecone, a popular commercial vector database vendor. Then log into Pinecone and retrieve your API key (visit API Keys in the side menu, and click "create API Key"). The code for this example is provided in the GitHub repository for the book:

Input:

```
from pinecone import Pinecone, ServerlessSpec
import os

# Initialize connection (get API key at app.pinecone.io):
os.environ["PINECONE_API_KEY"] = "insert-your-api-key-here"

index_name = "employee-handbook"
environment = "us-west-2"
pc = Pinecone()  # This reads the PINECONE_API_KEY env var

# Check if index already exists:
# (it shouldn't if this is first time)
if index_name not in pc.list_indexes().names():
    # if does not exist, create index
    pc.create_index(
        index_name,
        dimension=1536,  # Using the same vector dimensions as text-embedding-ada-
        metric="cosine",
        spec=ServerlessSpec(cloud="aws", region=environment),
    )

# Connect to index:
index = pc.Index(index_name)

# View index stats:
index.describe_index_stats()
```

Output:

```
{'dimension': 1536,
 'index_fullness': 0.0,
 'namespaces': {},
 'total_vector_count': 0}
```

Let's go through this code step by step:

1. *Importing libraries*: The script begins with importing the necessary modules. `from pinecone import Pinecone, ServerlessSpec` and `import os` is used for accessing and setting environment variables.
2. *Setting up the Pinecone API key*: The Pinecone API key, which is crucial for authentication, is set as an environment variable using `os.environ["PINECONE_API_KEY"] = "insert-your-api-key-here"`. It's important to replace `"insert-your-api-key-here"` with your actual Pinecone API key.
3. *Defining index name and environment*: The variables `index_name` and `environment` are set up. `index_name` is given the value `"employee-handbook"`, which is the name of the index to be created or accessed in the Pinecone database. The `environment` variable is assigned `"us-west-2"`, indicating the server's location.
4. *Initializing Pinecone connection*: The connection to Pinecone is initialized with the `Pinecone()` constructor. This constructor automatically reads the `PINECONE_API_KEY` from the environment variable.
5. *Checking for existing index*: The script checks whether an index with the name `index_name` already exists in the Pinecone database. This is done through `pc.list_indexes().names()` functions, which returns a list of all existing index names.
6. *Creating the index*: If the index doesn't exist, it is created using the `pc.create_index()` function. This function is invoked with several parameters that configure the new index:
   - `index_name`: Specifies the name of the index.
   - `dimension=1536`: Sets the dimensionality of the vectors to be stored in the index.
   - `metric='cosine'`: Determines that the cosine similarity metric will be used for vector comparisons.
7. *Connecting to the index*: After verifying or creating the index, the script connects to it using `pc.Index(index_name)`. This connection is necessary for subsequent operations like inserting or querying data.
8. *Index statistics*: The script concludes with calling `index.describe_index_stats()`, which retrieves and displays

various statistics about the index, such as its dimensionality and the total count of vectors stored.

Next you need to store your vectors in the newly created index, by looping through all the text chunks and vectors, and upserting them as records in Pinecone. The database operation `upsert` is a combination of *update* and *insert*, and it either updates an existing record or inserts a new record if the record does not already exist (refer to this Jupyter Notebook for the chunks variable):

Input:

```
from tqdm import tqdm # For printing a progress bar
from time import sleep

# How many embeddings you create and insert at once
batch_size = 10
retry_limit = 5  # maximum number of retries

for i in tqdm(range(0, len(chunks), batch_size)):
    # Find end of batch
    i_end = min(len(chunks), i+batch_size)
    meta_batch = chunks[i:i_end]
    # Get ids
    ids_batch = [str(j) for j in range(i, i_end)]
    # Get texts to encode
    texts = [x for x in meta_batch]
    # Create embeddings
    # (try-except added to avoid RateLimitError)
    done = False
    try:
        # Retrieve embeddings for the whole batch at once
        embeds = []
        for text in texts:
            embedding = get_vector_embeddings(text)
            embeds.append(embedding)
        done = True
    except:
        retry_count = 0
        while not done and retry_count < retry_limit:
            try:
                for text in texts:
                    embedding = get_vector_embeddings(text)
                    embeds.append(embedding)
                done = True
            except:
                sleep(5)
                retry_count += 1

    if not done:
```

```
            print(f"""Failed to get embeddings after
            {retry_limit} retries.""")
            continue

        # Cleanup metadata
        meta_batch = [{
            'batch': i,
            'text': x
        } for x in meta_batch]
        to_upsert = list(zip(ids_batch, embeds, meta_batch))

        # Upsert to Pinecone
        index.upsert(vectors=to_upsert)
```

Output:

```
100% 13/13 [00:53<00:00, 3.34s/it]
```

Let's break this code down:

1. Import necessary libraries `tqdm` and `time`. The library`tqdm` displays progress bars, and `time` provide the `sleep()` function, which is used in this script for retry logic.
2. Set the variable `batch_size` to 10 (normally set to 100 for real workloads), representing how many items will be processed at once in the upcoming loop. Also set the `retry_limit` to make sure we stop after 5 tries.
3. The `tqdm(range(0, len(chunks), batch_size))` part is a loop that will run from 0 to the length of `chunks` (defined previously), with a step of `batch_size`. `chunks` a list of text to be processed. `tqdm` is used here to display a progress bar for this loop.
4. The `i_end` variable is calculated to be the smaller of the length of `chunks` or `i + batch_size`. This is used to prevent an Index Error if `i + batch_size` exceeds the length of `chunks`.
5. `meta_batch` is a subset of `chunks` for the current batch. This is created by slicing the `chunks` list from index `i` to `i_end`.
6. `ids_batch` is a list of string representations of the range `i` to `i_end`. These are IDs that are used to identify each item in `meta_batch`.
7. Texts list is just the text from `meta_batch`, ready for processing for embeddings.
8. Try to get the embeddings by calling `get_vector_embeddings()` with the `texts` as the argument. The result is stored in the variable `embeds`. This is done inside a `try-except` block to handle any ex-

ceptions that might be raised by this function, such as a rate limit error.

9. If an exception is raised, the script enters a while loop where it will sleep for 5 seconds, then try again to retrieve the embeddings. It will continue this until successful or the number of retries is reached, at which point it sets `done = True` to exit the while loop.
10. Modify the `meta_batch` to be a list of dictionaries. Each dictionary has two keys: `batch`, which is set to the current batch number `i`, and `text`, which is set to the corresponding item in `meta_batch`. This is where you could additional metadata for filtering queries later, such as the page, title or chapter.
11. Create `to_upsert` list by using the `zip` function to combine `ids_batch`, `embeds`, and `meta_batch` into tuples, and then turning that into a list. Each tuple contains the ID, the corresponding embedding, and the corresponding metadata for each item in the batch.
12. The last line of the loop calls a method `upsert` on `index`, a Pinecone (a vector database service) index. The `vectors=to_upsert` argument passes the `to_upsert` list as the data to be inserted or updated in the index. If a vector with a given ID already exists in the index, it will be updated; if it doesn't exist, it will be inserted.

Once the records are stored in Pinecone, you can query them as you need, just like when you had saved your vectors locally with FAISS. Embeddings remain the same so long as you're using the same embedding model to retrieve vectors for your query, so you do not need to update your database unless you have additional records or metadata to add:

Input:

```
# Retrieve from Pinecone
user_query = "do we get free unicorn rides?"

def pinecone_vector_search(user_query, k):
    xq = get_vector_embeddings(user_query)
    res = index.query(vector=xq, top_k=k, include_metadata=True)
    return res

pinecone_vector_search(user_query, k=1)
```

Output:

```
{'matches':
    [{'id': '15',
    'metadata': {'batch': 10.0,
    'text': "You'll enjoy a treasure chest of perks, "
```

```
              'including unlimited unicorn rides, a '
              'bottomless cauldron of coffee and potions, '
              'and access to our company library filled '
              'with spellbinding books. We also offer '
              'competitive health and dental plans, '
              'ensuring your physical well-being is as '
              'robust as your magical spirit.\n'
              '\n'
              '**5: Continuous Learning and Growth**\n'
              '\n'
              'At Unicorn Enterprises, we believe in '
              'continuous learning and growth.'},
    'score': 0.835591,
    'values': []},],
 'namespace': ''}
```

This script performs a nearest neighbors search using Pinecone's API to identify the most similar vectors to a given input vector in a high-dimensional space. Here's a step-by-step breakdown:

1. The function `pinecone_vector_search` is defined with two parameters: `user_query` and `k`. `user_query` is the input text from the user, ready to be converted into a vector, and `k` indicates the number of closest vectors you want to retrieve.
2. Within the function, `xq` is defined by calling another function `get_vector_embeddings(user_query)`. This function (defined previously) is responsible for transforming the `user_query` into a vector representation.
3. The next line performs a query on an object named `index`, a Pinecone index object, using the `query` method. The `query` method takes three parameters:
    - The first parameter is `vector=xq`, the vector representation of our `user_query`.
    - The second parameter `top_k=k` specifies that you want to return only the `k` closest vectors in the Pinecone index.
    - The third parameter `include_metadata=True` specifies that you want to include metadata (such as IDs or other associated data) with the returned results.
    - If you wanted to filter the results by metadata, for example specifying the batch (or any other metadata you upserted), you could do this here by adding the a fourth parameter: `filter={"batch": 1}`.
4. The results of the `query` method are assigned to `res` and then returned by the function.
5. Finally, the function `pinecone_vector_search` is called with arguments `user_query` and `k`, returning the response from Pinecone.

You have now effectively emulated the job FAISS was doing, returning the exact same record of lorem ipsum with a similarity search by vector. If you replace `vector_search(user_query, k)` with `pinecone_vector_search(user_query, k)` in the `search_and_chat` function (from the previous example), the chatbot will run the same, except that the vectors will be stored in a hosted Pinecone database instead of locally using FAISS.

When you upserted the records into Pinecone, you passed the batch number as metadata. Pinecone supports the following formats of metadata:

- String
- Number (integer or floating-point, gets converted to a 64-bit floating point)
- Booleans (true, false)
- List of strings

The metadata strategy for storing records can be just as important as the chunking strategy, as you can use metadata to filter queries. For example, if you wanted to only search for similarity limited to a specific batch number, you could add a filter to the `index.query`:

```
res = index.query(xq, filter={
        "batch": {"$eq": 1}
    }, top_k=1, include_metadata=True)
```

This can be useful for limiting the scope of where you are searching for similarity. For example, it would allow you to store past conversations for all chatbots in the same vector database, and then query only for past conversations related to a specific chatbot id when querying to add context to that chatbot's prompt. Other common uses of metadata filters include searching for more recent timestamps, for specific page numbers of documents, or products over a certain price.

NOTE

Note that more metadata storage is likely to increase storage costs, as is storing large chunks that are infrequently referenced. Understanding how vector databases work should give you license to experiment with different chunking and metadata strategies, and see what works for your use cases.

Self Querying

Retrieval can get quite sophisticated, and you're not limited to a basic retriever that fetches documents from a vector database based purely on semantic relevance. For example, consider using metadata from within a user's query. By recognizing and extracting such filters your retriever can autonomously generate a new query to execute against the vector database, as in the structure depicted in Figure 5-3.

NOTE

This approach also generalizes to NoSQL, SQL or any common database and it is not solely limited to vector databases.

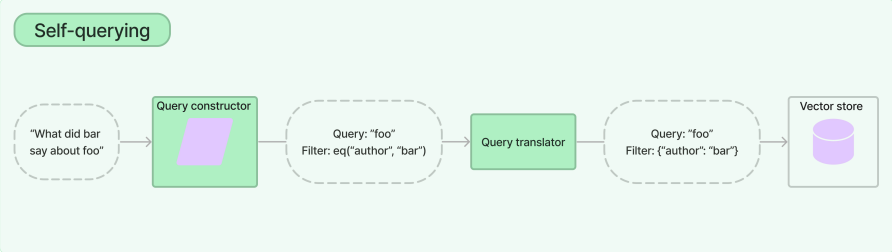

Figure 5-3. A self querying retriever architecture

Self querying yields several significant benefits:

- *Schema definition*: You can establish a schema reflecting anticipated user descriptions, enabling a structured understanding of the information sought by users.
- *Dual-layer retrieval*: The retriever performs a two-tier operation. First, it gauges the semantic similarity between the user's input and the database's contents. Simultaneously, it discerns and applies filters based on the metadata of the stored documents or rows, ensuring an even more precise and relevant retrieval.

This method maximizes the retriever's potential in serving user-specific requests.

Install `lark` on the terminal with: `pip install lark`

In the subsequent code, essential modules such as `langchain`, `lark`, `getpass`, and `chroma` are imported. For a streamlined experience, potential warnings are suppressed:

```
from langchain_core.documents import Document
from langchain_community.vectorstores.chroma import Chroma
from langchain_openai import OpenAIEmbeddings
import lark
import getpass
import os
import warnings

# Disabling warnings:
warnings.filterwarnings("ignore")
```

In the upcoming section, you'll craft a list named `docs`, filling it with detailed instances of the `Document` class. Each `Document` lets you capture rich details of a book. Within the metadata dictionary, you'll store valuable information such as the title, author, and genre. You'll also include data like the ISBN, publisher, and a concise summary to give you a snapshot of each story. The `"rating"` offers a hint of its popularity. By setting up your data this way, you're laying the groundwork for a systematic and insightful exploration of a diverse library:

```
docs = [
    Document(
        page_content="A tale about a young wizard and his \
            journey in a magical school.",
        metadata={
            "title": "Harry Potter and the Philosopher's Stone",
            "author": "J.K. Rowling",
            "year_published": 1997,
            "genre": "Fiction",
            "isbn": "978-0747532699",
            "publisher": "Bloomsbury",
            "language": "English",
            "page_count": 223,
            "summary": "The first book in the Harry Potter \
            series where Harry discovers his magical \
            heritage.",
            "rating": 4.8,
        },
    ),
    # ... More documents ...
]
```

You'll import `ChatOpenAI`, `SelfQueryRetriever` and `OpenAIEmbeddings`. Following this, you'll a new vector database using the `Chroma.from_documents(..)` method.

Next, the `AttributeInfo` class is used to structure metadata for each book. Through this class, you'll systematically specify the attribute's name, description, and type. By curating a list of `AttributeInfo` entries, the self-query retriever can perform metadata filtering:

```
from langchain_openai.chat_models import ChatOpenAI
from langchain.retrievers.self_query.base \
    import SelfQueryRetriever
from langchain.chains.query_constructor.base \
    import AttributeInfo

# Create the embeddings and vectorstore:
embeddings = OpenAIEmbeddings()
vectorstore = Chroma.from_documents(docs, OpenAIEmbeddings())

# Basic Info
basic_info = [
    AttributeInfo(name="title", description="The title of the book",
    type="string"),
    AttributeInfo(name="author", description="The author of the book",
    type="string"),
    AttributeInfo(
        name="year_published",
        description="The year the book was published",
        type="integer",
    ),
]

# Detailed Info
detailed_info = [
    AttributeInfo(
        name="genre", description="The genre of the book",
        type="string or list[string]"
    ),
    AttributeInfo(
        name="isbn",
        description="The International Standard Book Number for the book",
        type="string",
    ),
    AttributeInfo(
        name="publisher",
        description="The publishing house that published the book",
        type="string",
    ),
    AttributeInfo(
        name="language",
        description="The primary language the book is written in",
        type="string",
    ),
    AttributeInfo(
        name="page_count", description="Number of pages in the book",
```

```
            type="integer"
        ),
    ]

    # Analysis
    analysis = [
        AttributeInfo(
            name="summary",
            description="A brief summary or description of the book",
            type="string",
        ),
        AttributeInfo(
            name="rating",
            description="""An average rating for the book (from reviews), ranging
            from 1-5""",
            type="float",
        ),
    ]

    # Combining all lists into metadata_field_info
    metadata_field_info = basic_info + detailed_info + analysis
```

Let's run this through step-by-step:

1. Import `ChatOpenAI`, `SelfQueryRetriever`, and `AttributeInfo` from the LangChain modules for chat model integration, self-querying, and defining metadata attributes.
2. Create an `OpenAIEmbeddings` instance for handling OpenAI model embeddings.
3. A `Chroma` vector database is created from the documents.
4. Define three lists (`basic_info`, `detailed_info`, `analysis`), each containing `AttributeInfo` objects for different types of book metadata.
5. Combine these lists into a single list `metadata_field_info` for comprehensive book metadata management.

Now, set up a `ChatOpenAI` model and assign a `document_content_description` to specify what content type you're working with. The `SelfQueryRetriever` then using this along with your LLM to fetch relevant documents from your `vectorstore`. With a simple query, such as asking for sci-fi books, the `invoke` method scans through the dataset and returns a list of `Document` objects.

Each `Document` encapsulates valuable metadata about the book, like its genre, author, and a brief summary, transforming the results into organized, rich data for your application:

```
document_content_description = "Brief summary of a movie"
llm = ChatOpenAI(temperature=0)
retriever = SelfQueryRetriever.from_llm(
    llm, vectorstore, document_content_description, metadata_field_info
)

# Looking for sci-fi books
retriever.invoke("What are some sci-fi books?")
# [Document(page_content='''A futuristic society where firemen burn books to
# maintain order.''', metadata={'author': 'Ray Bradbury', 'genre': '...
# More documents..., truncated for brevity
```

EVALUATE QUALITY

By setting the temperature to zero, you instruct the model to prioritize generating consistent metadata filtering outputs, rather than being more creative and therefore inconsistent. These metadata filters are then leveraged against the vector database to retrieve relevant documents.

When you wish to fetch books from a specific author, you're directing the `retriever` to pinpoint books authored by `J.K. Rowling`. The `Comparison` function with the `EQ` (equals) comparator ensures the retrieved documents have their `'author'` attribute precisely matching `'J.K. Rowling'`:

```
# Querying for a book by J.K. Rowling:
retriever.invoke(
    '''I want some books that are published by the
    author J.K.Rowling'''
)
# query=' ' filter=Comparison(comparator=<Comparator.EQ:
# 'eq'>, attribute='author', value='J.K. Rowling')
# limit=None
# Documents [] ommited to save space
```

Initializing the `SelfQueryRetriever` with an added `enable_limit` flag set to `True` allows you to dictate the number of results returned. Then, you craft a query to obtain precisely `2 Fantasy` books. By using the `Comparison` function with the `EQ` (equals) comparator on the `'genre'` attribute, the retriever zeroes in on `'Fantasy'` titles. The `limit` parameter ensures you get only two results, optimizing your output for precision and brevity:

```
retriever = SelfQueryRetriever.from_llm(
    llm,
    vectorstore,
```

```
    document_content_description,
    metadata_field_info,
    enable_limit=True,
)

retriever.get_relevant_documents(
    query="Return 2 Fantasy books",
)
# query=' ' filter=Comparison(
#    comparator=<Comparator.EQ: 'eq'>, attribute='genre',
#   value='Fantasy') limit=2
# Documents [] ommited to save space
```

Alternative Retrieval Mechanisms

When it comes to retrieval implementations, various intriguing methods each demonstrate their distinct approaches, advantages, and limitations.

- *MultiQueryRetriever*: The MultiQueryRetriever aims to overcome the limitations of distance-based retrieval by generating multiple queries from different perspectives for a given user input query. This leads to the generation of a larger set of potentially relevant documents, offering broader insights. However, challenges may arise if the different queries produce contradicting results or overlap.
- *Contextual compression*: The Contextual Compression Retriever handles long documents by compressing irrelevant parts, ensuring relevance to context. The challenge with this method is the expertise needed to determine the relevance and importance of information.
- *Ensemble retriever*: The Ensemble Retriever uses a list of retrievers and combines their results. It's essentially a "hybrid" search methodology that leverages the strengths of various algorithms. However, the Ensemble Retriever implies more computational workload due to the use of multiple retrieval algorithms, potentially affecting retrieval speed.
- *Parent document retriever*: The Parent Document Retriever ensures the maintenance of rich document backgrounds by retrieving original source documents from which smaller chunks are derived. But it might increase computational requirements due to the retrieval of larger parent documents.
- *Time-weighted vector store*: The Time Weighted VectorStore Retriever incorporates *time decay* into document retrieval. Despite its advantages, the time decay factor might cause overlooking of relevant older documents, risking the loss of historical context.

The key to effective retrieval is understanding the trade-offs and selecting the method, or combination of methods, that best address your specific use case. Vector search adds additional cost and latency to your application, so ensure in testing you find that the additional context is worth it. For heavy workloads, paying the upfront cost of fine-tuning a custom model may be beneficial compared to the ongoing additional cost of prompts plus embeddings plus vector storage. In other scenarios, providing static examples of correct work in the prompt may work fine. However, when you need to pull in context to a prompt dynamically, based on similarity rather than a direct keyword search, there's no real substitute for RAG using a vector database.

Summary

In this chapter, you have learned about the power of vector databases for storing and querying text based on similarity. These databases use vectors, which are lists of numbers representing text, to determine the distance between texts. By searching for the most similar records, vector databases can retrieve relevant information to provide context in your prompts, helping AI models stay within token limits and avoid unnecessary costs. You have also discovered that the accuracy of vectors depends on the underlying model, and have seen examples where they may fail.

Furthermore, you have explored the process of index documents in a vector database, searching for similar records using vectors, and inserting records into prompts as context, called RAG. In this chapter you went through code examples for retrieving embeddings from both the OpenAI API and open-source models like the Sentence Transformers library. You also learned the cost and benefits associated with retrieving embeddings from OpenAI relative to alternative options.

In the next chapter on autonomous agents, you will delve into the world of AI agents that can make decisions and take actions on their own. You will learn about the different types of autonomous agents, their capabilities, and how they can be trained to perform specific tasks. Additionally, you will explore the challenges and unreliability issues associated with agents.