

# 15 Foundation models and emerging search paradigms

## This chapter covers

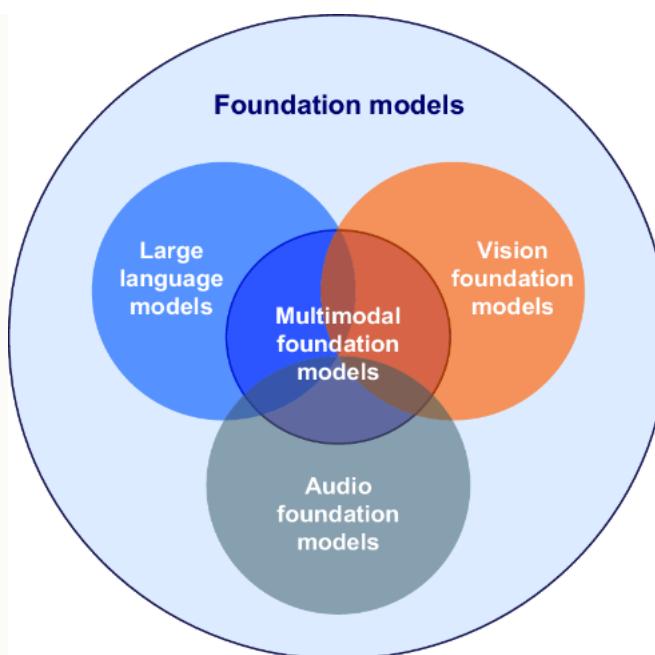
- Retrieval augmented generation (RAG)
- Generative search for results summarization and abstractive question answering
- Integrating foundation models, prompt optimization, and evaluating model quality
- Generating synthetic data for model training
- Implementing multimodal and hybrid search
- The future of AI-powered search

Large language models (LLMs), like the ones we've tested and fine-tuned in the last two chapters, have been front and center in the advances in AI-powered search in recent years. You've already seen some of the key ways search quality can be enhanced by these models, from improving query interpretation and document understanding by mapping content into embeddings for dense vector search, to helping extract answers to questions from within documents.

But what additional advanced approaches are emerging on the horizon? In this chapter, we'll cover some of the more recent advances at the intersection of search and AI. We'll cover how foundation models are being used to extend new capabilities to AI-powered search, like results summarization, abstractive question answering, multimodal search across media types, and even conversational interfaces for search and information retrieval. We'll cover the basics of emerging search paradigms like generative search, retrieval augmented generation (RAG), and new classes of foundation models that are reinventing some of the ways we'll soon approach the frontier of AI-powered search.

## 15.1 Understanding foundation models

A *foundation model* is a model that is pretrained on a large amount of broad data and is designed to be generally effective at a wide variety of tasks. LLMs are a subset of foundation models that are trained on a very large amount of text. Foundation models can also be trained on images, audio, or other sources, or even on multimodal data incorporating many different input types. Figure 15.1 demonstrates common categories of foundation models.



**Figure 15.1 Foundation model types.** LLMs are one of several types of foundation models.

Vision foundation models can be used to map images into embeddings (like how we mapped text to embeddings in chapter 13), which can then be searched to enable image-to-image search.

A multimodal foundation model can be built using both text and images (or other data types), and it can then enable cross-modal search for images based on text queries, or on text based on uploaded images as the query. We'll implement this kind of multimodal text and image search in section 15.3.2. Generative multimodal models, like Stable Diffusion (a text-to-image model), can also be used to generate brand-new images based only on text prompts. Multimodal foundation models that can learn from both images and text are also commonly referred to as *vision language models* (VLMs).

### 15.1.1 What qualifies as a foundation model?

Foundation models are typically trained on a wide variety of data covering many topics so that they are effective at generalized interpretation and prediction across domains. These models are called “foundation” models because they can serve as a base model (or foundation), which can then be more quickly fine-tuned on domain-specific or task-specific training sets to better tackle specific problems.

Foundation models typically meet the following criteria:

1. They are *large*, typically trained on massive amounts of data, often with billions or trillions of parameters.
2. They are *pretrained*, using significant compute capacity to arrive at model weights to be saved and deployed (or fine-tuned) later.
3. They are *generalizable* to many tasks, as opposed to limited to specific tasks.
4. They are *adaptable*, using prompts to pull in additional context from their trained model to adjust their predicted output. This makes the kinds of queries they can accept very flexible.

5. They are *self-supervised*, learning automatically from raw data how to relate and interpret the data and represent it for future use.

We have already worked with several foundation models in previous chapters, including BERT, which is one of the earliest foundation models, and RoBERTa, which we used in chapter 13 to generate embeddings and perform semantic search on those embeddings. Sentence Transformer models such as SBERT (Sentence-BERT) and SRoBERTa (Sentence-RoBERTa) are models that were fine-tuned from the BERT and RoBERTa foundation models to excel at the semantic textual similarity (STS) task. We also fine-tuned the `deepset/roberta-base-squad2` model in chapter 14; it is a model based upon the RoBERTa foundation model that has been fine-tuned for the question-answering task. Technically SBERT, SRoBERTa, and `deepset/roberta-base -squad2` are themselves fine-tuned foundation models, which can be further used as the foundation for more fine-tuning to generate additional models.

The dominant architecture for foundation models is currently the Transformer model, though recurrent neural networks (using architectures like MAMBA) can also be used, and additional architectures will inevitably evolve over time. Most Transformer-based models can be used to generate embedding vectors or predictive output.

The strength of the responses from foundation models reflects the quality of three processes: training, fine-tuning, and prompting.

### 15.1.2 Training vs. fine-tuning vs. prompting

*Training* (or *pretraining*) is the process whereby a massive amount of data (often a large portion of the internet) is used to learn model weights for billions or trillions of parameters in the foundation model's deep neural network. This process can sometimes be very expensive, take months, and cost millions of dollars due to the computing and energy requirements. This process manages to perform lossy compression of much of human knowledge into a neural network from which facts and relationships (words, linguistics, associations, etc.) can be decompressed back out later. Recall from section 13.3 that training a Transformer on text typically follows a self-supervised learning process that optimizes for predicting masked tokens in text sequences to measure overall comprehension of the text (the Cloze test described in section 13.3.1). This training can include any specific datasets that may be beneficial for the model's knowledge base, such as computer code or domain-specific content (financial documents, academic papers, foreign languages, multi-modal content, etc.)

*Fine-tuning* is the process whereby the foundation model can be supplied with additional domain-specific data or instructions. For example, if you want the model to follow instructions or to act like a person or a chatbot, you can train the model with responses to input that reflect those behaviors. There are several approaches to fine-tuning, depending on the task or architecture and compute and budgetary requirements. Some types of fine-tuning will change all the weights of a model, which is helpful if the domain-tuning needs to be highly specific and the task of the original model is the same. More efficient or task-specific approaches may leave all the existing weights intact and add one or more additional

layers to the neural network of the base foundation model. Fine-tuning enables these new sources to extend the capabilities of the original model to new data or patterns with a much smaller training process focused on specific data or goals.

*Prompting* is the process of supplying input to the trained language model to get an output. Prompting is the final “training” step for fine-tuning a model, but it occurs at inference time as opposed to training time. We can supply as much context in the prompt as the model allows, which means that the prompt can be manipulated to use this additional context to affect the output. For example, consider the output to the queries in listings 15.1 and 15.2 when they are sent to OpenAI’s GPT-4 language model.

### **Listing 15.1 Query with no prompt engineering**

Query: "What is a unicorn?"

Response:

A unicorn is a mythical creature that is typically depicted as a horse with a single, spiraled horn projecting from its forehead. Unicorns have been a part of human mythology and folklore for centuries and are often associated with purity, grace, and magic. They are said to possess healing powers and to be able to bring good luck to those who encounter them. While unicorns are not real animals, they continue to be a popular symbol in modern culture, appearing in literature, art, and popular media.

Listing 15.1 provides a good general description of a unicorn. Contrast this with listing 15.2, which uses prompt engineering to give the foundation model a “chatbot” persona named AIPS Chat and then tells the foundation model to respond as if it had a PhD in biology.

## Listing 15.2 Query with additional context from prompt engineering

```
Query: "You are a chatbot named AIPS Chat.  
        You have a Ph.D. in biology.  
        What is a unicorn?"
```

Response:

Hello! I am a chatbot named AIPS Chat, and I have a Ph.D. in biology. While unicorns have been depicted in various cultures throughout history, they are not real animals that exist in the natural world.

From a biological perspective, a unicorn can be described as a mythical creature resembling a horse with a single, spiraled horn on its forehead. However, since unicorns are purely fictional, there is no actual scientific data or research that can be conducted on them.

It is important to note that while many animals possess unique characteristics, such as the horn of a rhinoceros or the tusk of a narwhal, these features are not magical and do not provide any special abilities or powers to the animal possessing them.

In conclusion, while unicorns are a fascinating aspect of human folklore and mythology, they do not have a basis in reality within the field of biology.

The query in listing 15.2, which includes the context of a chatbot having a PhD in biology, uses that additional context to inform its answer. If we had read/write access to the language model, we could fine-tune it with inputs and responses generated by a PhD in biology and by a chatbot. In the listing, we were able to accomplish a similar outcome simply by supplying the model with a prompt to pull in that context from its already-trained model. Fine-tuning will typically generate a better answer to such questions, but prompting is much more flexible, since you can supply any context at inference time, whereas fine-tuning would be more appropriate for general characteristics you want the model to learn and represent in all future interactions.

Providing additional context in the prompt can be key to getting the best output from a foundation model. Since generative foundation models sequentially predict the next token in a sequence, one token at a time, coercing the model to output more relevant context in the response can cause the model to generate more relevant output. A large pretrained model is capable of *few-shot learning* (being able to learn in context without further training by providing two or three examples with the prompt). In listing 15.2, for example, we saw that by adding the “biology” context to the prompt, the answer thereafter included phrases like “in the natural world”, “From a biological perspective”, “actual scientific data or research”, and “within the field of biology”.

The hardest and most expensive part of training is the initial generation of the foundation model. Once it’s trained, the fine-tuning for specific tasks is relatively quick and inexpensive and can often be done with a reasonably small training set. If we compare training these models with human learning, a foundation model may be like a high school student

who generally knows how to read and write and can answer basic questions about math, science, and world history. If we want the student to be able to generate and interpret financial statements (income statements, cash flow statements, and balance sheets), they are going to need to take an accounting course to learn those skills. After 18 or more years of training (life experience and school), however, the student can likely learn sufficient accounting within a few months to generate and interpret financial statements. Similarly, with foundation models, the initial training phase takes the longest and provides the base upon which further knowledge and skills can be much more quickly learned.

## 15.2 Generative search

Most of our journey into building AI-powered search has focused on finding the results, answers, or actions matching the user's intent. In chapter 14, we went so far as to extract specific answers to questions, using our retriever-reader pattern. But what if instead of returning real documents or extracted answers, our search engine could generate new content on the fly in response to queries?

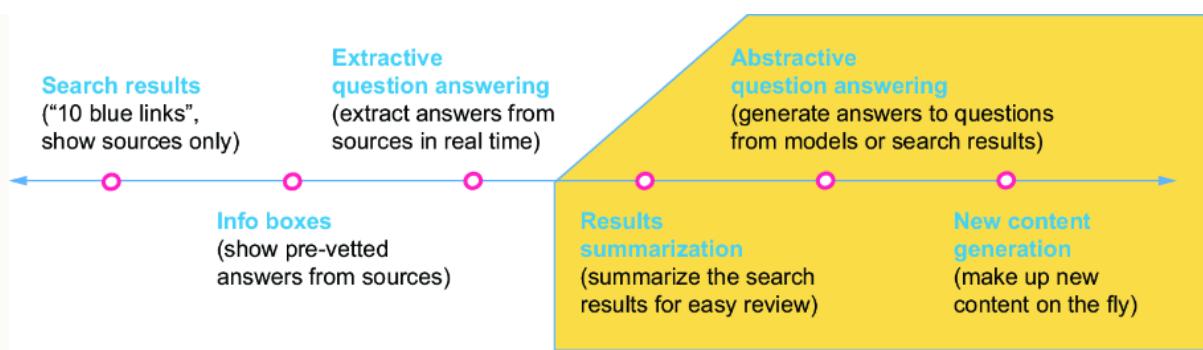
*Generative models* are models that can generate new content based on incoming prompts. Their output can be text content, images generated from text input, audio emulating specific people or sounds, or videos combining both audio and video. Text can be generated to describe an image, or alternatively, an image, audio, or video can be generated to “describe” the text in a different modality.

We are entering a world where someone will be able to enter a search query, and the search engine could return entirely made-up content and images, generated on the fly, in response to the user's query or prompts. While this can be amazing with queries like

What is an optimal agenda for three days in Paris assuming I really like fine dining, historic buildings and museums, but hate crowds, it also introduces severe ethical considerations.

Should a search engine be responsible for synthesizing information for its users instead of returning existing content for them to interpret? What if the search engine is politically or commercially biased and is trying to influence users' thinking or behavior? What if the search engine is being used as a tool for censorship or to spread propaganda (such as images or text that were modified on the fly) to trick people into false beliefs or to take dangerous actions?

Some of the more common use cases for generative search are abstractive question answering and results summarization. Whereas traditional search approaches have returned “ten blue links” or predetermined info boxes with information matching certain queries, generative search focuses on creating search responses on the fly based on dynamically analyzing the search results. From left to right, figure 15.2 shows the progression from these traditional search approaches toward generative search.



**Figure 15.2 Spectrum of traditional search approaches (left) to generative search (right)**

Extractive question answering, covered in chapter 14, begins the move toward generative search, in that it analyzes search results to return direct answers to questions instead of just the search results or predetermined answers. It's not entirely "generative" search, however, in that it still only returns answers exactly as presented in the searched documents, without any additional synthesis or new content generation.

As we continue toward the right in figure 15.2, results summarization is the first technique that can be fully considered generative search. Results summarization, which we'll cover in section 15.2.2, is the process of taking the search results and summarizing their content. This can be quite useful for the user, particularly if they are researching a topic and don't have time to read through all the results. Instead of the user needing to click on multiple links and read and analyze each of the results, a summary of the pages (along with citations, if desired) can be returned, saving the user time assessing the content.

Abstractive question answering is very similar to extractive question answering, in that it tries to answer a user's question, but it does so by either analyzing search results (like in results summarization) or simply by relying on the underlying foundation model to generate the answer. Relying on the foundation model is more likely to lead to made-up, or hallucinated, results, so it is often beneficial to use the search results as part of the prompt context for in-context learning. We'll cover this process of retrieval augmented generation in the next section.

Many major search engines and startups have integrated these generative models and interactive chat sessions into their search experiences, which provides the models with access to the internet (or at least a copy of the internet in the form of the search index). This allows the models to have a near-real-time view of the world's information. It means that the models may know detailed public information about the users they are interacting with, which may shape the results, and it also means that anyone can work to inject malicious information or intentions into these models by crafting web pages that relate concepts in misleading ways. If we're not careful, the field of search engine optimization (SEO) may go from trying to boost a website higher in search results to trying to manipulate the AI into providing malicious answers to end users. Applying critical thinking skills and validating the output of these models will be essential for anyone using them, but unfortunately, the models can be so compelling that many people are likely to be fooled into believing misleading output unless significant safeguards are put in place.

These models will continue to evolve to support searchers by interpreting search results. While many people may dream of having an AI-based personal assistant, the ethical considerations of having an AI generate the knowledge you consume daily will need to be handled carefully.

### 15.2.1 Retrieval augmented generation

The workflow of using a search engine or vector database to find relevant documents that can be supplied as context to an LLM is commonly referred to as *retrieval augmented generation* (RAG).

We've mentioned RAG several times before, and it's a big buzzword as this book is being published. There's a very good reason for this: the training of language models is a form of lossy compression of the training data, and the models can't faithfully store the massive amount of data they were trained on without losing fidelity. Their information also dates from the last time they were trained—it is hard for the models to make decisions on evolving information without some continually updated external data source.

No matter how much context a language model can accept in its prompt, it's computationally infeasible or at least cost-prohibitive to rely on an LLM as a source of truth for all the information it was trained on. Thankfully, the entire purpose of search engines is to find relevant context for any incoming query. In fact, this *entire* book is a deep dive into the *retrieval* part of RAG. As we discuss results summarization, abstractive question answering, and generative search in this chapter, we're touching on the *generation* part of RAG.

To perform RAG, most current libraries take documents, split them into one or more sections (chunks), and then index embeddings for each section into a search engine or vector database. Then, at generation time, the application prompting the language model will create queries to find supplementary information needed to execute the next prompt, generate embeddings for those queries, perform a dense vector search to find the highest-scoring sections using vector similarity (usually cosine or dot product), and pass the resulting ranked sections along with the prompt to the generative AI model.

## CHALLENGES WITH CHUNKING

This process of splitting apart documents into sections is known as *chunking*, and it is both necessary and problematic. The problem can be understood as a tension between three constraints:

- *Vector database limitations*—Many vector databases are designed to index single vectors, but summarizing an entire document into a single vector can lose a lot of the specificity and context of the document. It pools the embedding across the entire document into a more vague summary embedding.
- *Loss of context between independent chunks*—Splitting a document into many separate documents to preserve the specificity of each section (chapter, paragraph, sentence, word, or other useful semantic chunk) can cause a loss of context across chunks. If you split a document into 10 chunks and independently generate embeddings, the shared context across those embeddings is lost.

- *Computational complexity of many chunks*—The more chunks you have, the more vectors you need to index, and the more you need to search. This can be computationally expensive and slow, and it can also be difficult to manage the relevance of the results across the chunks when you have to weight many matches on the same initial document relative to fewer, but better, matches on other documents.

At the early stages of implementing RAG for generative AI, many people have focused on chunking documents into multiple separate documents to overcome vector database limitations. Computational complexity can be managed using good ANN (approximate nearest-neighbor) algorithms, but the loss of context across chunks is still problematic in that case, and it's frankly wasteful to create an unbounded number of overlapping chunks instead of using better algorithms to model the embeddings.

As an alternative approach, several search engines and vector databases (such as Vespa and Qdrant) already have support for multivalued vector fields, which makes it possible to create an arbitrary number of chunks and also to create overlapping chunks within one document (so that a single sentence or paragraph can be part of multiple chunks, thereby preserving more context across chunks). Such multivector support will likely become standard in the coming years to support emerging contextualized late interaction approaches like those introduced in the ColBERT family of models.

## THE FUTURE OF RAG

RAG, as a discipline, is still in its infancy, but it's evolving rapidly. Numerous libraries and frameworks have been developed to perform RAG, but they are often based on an overly simple understanding of information retrieval. Current RAG implementations tend to rely entirely on a language model and vector similarity for relevance, ignoring most of the other AI-powered retrieval techniques we've covered in this book. In the context of the dimensions of user intent (figure 1.7), this has three effects:

- The content context is handled well conceptually (assuming the chosen embedding model was trained for query-document retrieval), but not on a specific keyword basis (due to the scope of the vectors summarizing multiple keywords).
- The domain context is handled as well as the language model is fine-tuned.
- The user context is usually ignored entirely.

For the content-understanding dimension of user intent (see section 1.2.5), promising approaches that use contextualized late interaction with embeddings are evolving, and they may eventually overcome the need for chunking altogether. These approaches (such as ColBERT, ColBERTv2, ColPali, and successors) involve generating an embedding per token in the document, but where each token's embedding uses the context of that token within the entire document (or at least a long surrounding context). This prevents the loss of context across chunks and avoids the need to index an unbounded number of chunks into the engine. These kinds of approaches make much more sense for RAG, and retrieval in general, than some of the more naive approaches mentioned in the last subsection. We expect to see similar approaches evolve over the coming years and to significantly improve recall over the current baseline ranking approaches.

The retrieval concepts you've learned in this book, concerning dimensions of user intent, reflected intelligence and signals models, semantic knowledge graphs, learning to rank, and feedback loops (combined with LLM-based vector search), are light years ahead of most current RAG implementations, which only use a subset.

That said, the future of RAG is bright, and it's likely the next few years will both see more sophisticated approaches to tackling current challenges, and see the integration of better retrieval and ranking techniques into the RAG process. RAG techniques are changing and evolving rapidly, and the field is ripe for continued innovation and improvement.

We'll cover one of the most popular generative search approaches using RAG in the coming section: results summarization (with citations).

### 15.2.2 Results summarization using foundation models

In chapter 3, we stated that the search engine is primarily responsible for three things: indexing, matching, and ranking results. Chapter 13 already showed how to enhance indexing (with embeddings), matching (with approximate nearest-neighbor search), and ranking of results (with similarity comparison on embedding vectors). But the way we return results is often just as important as the results themselves.

In chapter 14, we demonstrated how to extract answers out of ranked documents containing those answers, which can be a drastic improvement over returning full documents and forcing users to analyze them individually. But what if the answer to a question *requires* analysis of the documents? What if the desired answer is actually the *result* of an analysis that combines information from multiple documents into a unified, well-sourced answer?

One of the problems with answers generated directly from LLMs is that they are created based on statistical probability distributions from parameters learned by the model. Though the model may have been trained on a massive number of data sources, it does not store those data sources directly. Instead, the model is compressed into a lossy representation of the data. This means that you cannot count on the output from an LLM to accurately reflect the input data—only an approximation of it.

As a result, foundation models are well-known to hallucinate—to generate responses that make up incorrect facts or misrepresent the topics in the response. The context provided in the prompt also heavily influences the answer, to the point where foundation models can sometimes be more a reflection of the user's word choices and bias than a legitimate answer to a question. While this can be useful in creative endeavors, it makes today's foundation models quite unreliable for safely generating the fact-based responses for which search engines are historically relied upon. For users to truly trust the results, they need to be able to verify the sources of the information. Thankfully, instead of directly relying on foundation models for answers to questions, we can combine the search engine with a foundation model, using RAG to create a hybrid output.

Using foundation models like this to summarize search results is a great way to infuse your search engine with better AI-powered responses. Let's walk through an example of

performing search results summarization with citations using a foundation model. We'll use output from OpenAI's GPT-4 model, but you can get similar output from most current open source or permissively licensed LLMs.

The workflow includes two steps:

1. Execute a search. This can involve converting the query to an embedding and performing a dense vector search or using any other technique we've discussed to find the most relevant results.
2. Construct a prompt instructing your foundation model to take the user's query and read and summarize a set of search results returned from the query.

The following listing demonstrates an example prompt combining the search results from step 1 into a prompt from step 2.

### **Listing 15.3 Prompt to foundation model to summarize search results**

```
Web search results:
```

```
[1] "A large language model, or LLM, is a deep learning algorithm that can recognize, summarize, translate, predict and generate text and other content based on knowledge gained from massive datasets."
```

```
URL: https://blogs.nvidia.com/blog/2023/01/26/what-are-large-language-models-used-for/
```

```
[2] A large language model (LLM) is a language model consisting of a neural network with many parameters (typically billions of weights or more), trained on large quantities of unlabeled text using self-supervised learning. LLMs emerged around 2018 and perform well at a wide variety of tasks. This has shifted the focus of natural language processing research away from the previous paradigm of training specialized supervised models for specific tasks.
```

```
URL: https://en.wikipedia.org/wiki/Large\_language\_model
```

```
[3] "In this guide, we'll discuss everything you need to know about Large Language Models (LLMs), including key terms, algorithms, fine-tuning, and more. As ChatGPT has taken the internet by storm crossing 1 million users in its first 5 days, you may be wondering what machine learning algorithm is running under the hood. While ChatGPT uses a specific type of reinforcement learning called "Reinforcement Learning from Human Feedback (RLHF)", at a high level it is an example of a Large Language Model (LLM)."
```

```
URL: https://www.mlq.ai/what-is-a-large-language-model-llm/
```

**Instructions:** Using the provided web search results, write a comprehensive reply to the given query. Make sure to cite results using [[number](URL)] notation after the reference. If the provided search results refer to multiple subjects with the same name, write separate answers for each subject.

**Query:** What is a large language model?

When all of listing 15.3 is passed as a prompt to the language model, we got the following result.

#### **Listing 15.4 Response to the summarization prompt from listing 15.3**

A large language model (LLM) is a deep learning algorithm that can recognize, summarize, translate, predict, and generate text and other content based on knowledge gained from massive datasets  
[[1]](<https://blogs.nvidia.com/blog/2023/01/26/what-are-large-language-models-used-for/>)].  
It consists of a neural network with many parameters (typically billions of weights or more) and is trained on large quantities of unlabeled text using self-supervised learning  
[[2]]([https://en.wikipedia.org/wiki/Large\\_language\\_model](https://en.wikipedia.org/wiki/Large_language_model))]. LLMs emerged around 2018 and have performed well at a wide variety of tasks, shifting the focus of natural language processing research away from the previous paradigm of training specialized supervised models for specific tasks  
[[2]]([https://en.wikipedia.org/wiki/Large\\_language\\_model](https://en.wikipedia.org/wiki/Large_language_model))]. One example of a LLM is ChatGPT, which uses a specific type of reinforcement learning called "Reinforcement Learning from Human Feedback (RLHF)"  
[[3]](<https://www.mlq.ai/what-is-a-large-language-model-l1m/>)].

This output pulls critical details from the search results to answer the original query (What is a large language model?) but does it in a way that cites the original articles. If that answer is too verbose, an additional refinement to the prompt can be added to the end of the instructions: Be concise. This results in the following output instead:

...

Query: What is a large language model? Be concise.

#### **Listing 15.5 Results for listing 15.3's prompt with Be concise.**

A large language model (LLM) is a deep learning algorithm that uses neural networks with billions of parameters, trained on massive unlabeled text datasets for various language tasks [[2]]([https://en.wikipedia.org/wiki/Large\\_language\\_model](https://en.wikipedia.org/wiki/Large_language_model))] [[1]](<https://blogs.nvidia.com/blog/2023/01/26/what-are-large-language-models-used-for/>)].

Many search engines have started relying on foundation models to interpret and cite search results like this. By adding in additional instructions like “avoid being vague, controversial, or off-topic”, “use an unbiased and journalistic tone”, or even “write at a fifth-grade reading level”, you can adjust the quality and tone of the summaries to cater to your search engine’s needs.

### **15.2.3 Data generation using foundation models**

In addition to foundation models adding a synthesis and summarization layer on top of search results, one of the other emerging use cases for the models is the generation of synthetic domain-specific and task-specific training data.

As we think back to the numerous AI-powered search techniques we've already explored, many of them require substantial training data to implement:

- Signals boosting models require user signals data showing which user queries correspond with which documents.
- Click models, useful for automated LTR, require an understanding of which documents users click on and skip over in their search results.
- Semantic knowledge graphs require an index of data to find related terms and phrases.

But what if we fine-tuned a foundation model to “come up with documents about missing topics” or “come up with realistic queries” associated with each document? Or better yet, what if we didn’t need to fine-tune at all, but could instead construct a prompt to generate great queries for documents? Such data could be used to generate synthetic signals to help improve relevance.

Listings 15.6 through 15.9 show how we can use a prompt to find queries and relevance scores by combining an LLM with searches on our Stack Exchange outdoors collection from chapter 14.

For our exercise, we found it better to give several documents to the prompt, as opposed to just one, and even better to return related documents (from the same topic or a results list from a previous query). This is important, because there are niche topics within each set of documents, so having related documents helps to return more fine-grained queries instead of general ones. Additionally, while all the documents may not be perfectly relevant to the original query (or may be noisy), this still gives the LLM a few-shot chance at understanding our corpus, as opposed to basing its context on a single example. The following listing shows the template used with the LLM to generate candidate queries for a list of documents.

### **Listing 15.6 A prompt that generates queries describing documents**

```
What are some queries that should find the following documents? List at least 5 unique queries, where these documents are better than others in an outdoors question and answer dataset. Be concise and only output the list of queries, and a result number in the format [n] for the best result in the resultset. Don't print a relevance summary at the end.
```

```
### Results:  
{resultset}
```

The following listing shows code that can generate the `{resultset}` to be inserted into the prompt from listing 15.6. With the example query `what are minimalist shoes?` we get the `resultset` using the `retriever` function from listing 14.15. The retriever provides answer contexts for a query.

## Listing 15.7 Producing prompt-friendly text search results

```
example_contexts = retriever("What are minimalist shoes?")
resultset = [f"{idx}. {ctx}" for idx, ctx  #1
            in enumerate(list(example_contexts[0:5]["context"]))]  #1
print("\n".join(resultset))
```

#1 We only want the index and context information from the first 5 results returned by the retriever, and we prefix each result with its index of 0 to 4.

Output:

0. Minimalist shoes or "barefoot" shoes are shoes that provide your feet with some form of protection, but get you as close to a barefoot experience as possible...
1. There was actually a project done on the definition of what a minimalist shoe is and the result was "Footwear providing minimal interference with the natural movement of the foot due to its high flexibility, low heel to toe drop, weight and stack height, and the absence of motion control and stability devices". If you are looking for a simpler definition, this is what Wikipedia says, Minimalist shoes are shoes intended to closely approximate barefoot running conditions...
2. One summer job, I needed shoes to walk on a rocky beach, sometimes in the water, for 5 to 10 miles per day all summer. Stretchy neoprene shoes were terrible for this- no traction and no support. So I used regular sneakers. I chose a pair of cross country racing flats... The uppers were extremely well ventilated polyester, so they drained very quickly, and the thin material dried much faster than a padded sandal, and certainly much faster than a regular sneaker of leather or cotton would... The thing to look for is thin fabric that attaches directly to the sole, with no rubber rim that would keep the water from draining...
3. ... It's not unhealthy to wear minimalist footwear, but on what terrain you wear them could be bad for your body in the long run. Human beings were never meant to walk or run exclusively on hard pavement or sidewalks. Nor were we designed to clamber around on sharp rocks at high elevations... If you're running on soft ground and you have the foot strength, then there are plenty of arguments in favour of minimalist shoes being better for you than other shoes, because it brings your posture and gait back to what nature intended it to be. If you're hiking in the mountains on uneven rocky terrain, especially while carrying a heavy bag, then you'd be better off wearing a supportive hiking boot...
4. ... My favourite barefoot shoes are Vibram Five-Fingers , I wear either my Mouris or my KSO's at the beach. Vibram Five-Fingers 'Signa' watersport shoes: The thin sole will be enough to protect your feet from the lava sand...

These five results are then added to the bottom of the prompt from listing 15.6 by replacing the `{resultset}` value with the response from listing 15.7. We take the final, fully substituted prompt, and pass it into our LLM, yielding the following result.

## Listing 15.8 LLM response relating documents to generating queries

```
resultset_text = "\n".join(resultset)
resultset_prompt = summarize_search_prompt.replace("{resultset}",
                                                 resultset_text)
generated_relevance_judgments = get_generative_response(resultset_prompt)
display(generated_relevance_judgments)
```

Output:

1. What is the definition of a minimalist shoe?
2. What are the characteristics of minimalist shoes?
3. Which shoes are best for walking on rocky beaches?
4. Are minimalist shoes suitable for all terrains?
5. What are some recommended barefoot shoe brands?

Results:

1. [1]
2. [0]
3. [2]
4. [3]
5. [4]

Notice the nuance in relevance between the candidate queries and the wording in each of their relevant results. Also note that the ordering of the relevant results is not the same as the ordering of the queries in the output. The context of document 0 is more relevant for the specific query `What are the characteristics of minimalist shoes?` and the context of document 1 is more relevant for the query `What is the definition of a minimalist shoe?`

Note that we just use the pandas index position for the context IDs, instead of the IDs of the documents. In our experience, the IDs can confuse the model by providing irrelevant information. Depending on the LLM you use, this may need to be reverse engineered with some code, like we've done with our existing `example_contexts` from listing 15.7. Also notice how the model changed the order in listing 15.8 (the queries are not in the same order as the judgment results), so we'll need to account for this next when we parse the output.

The following listing shows how to extract the information and make a nice Python dictionary for the queries, documents, and relevant results.

## Listing 15.9 Extracting pairwise judgments from the LLM output

```
def extract_pairwise_judgments(text, contexts):
    query_pattern = re.compile(r"\d+\.\s+(.*)") #1
    result_pattern = re.compile(r"\d+\.\s+[(\d+)\]") #1
    lines = text.split("\n")
    queries = []
    results = []
    for line in lines:
        query_match = query_pattern.match(line)
        result_match = result_pattern.match(line)
        if result_match:
            result_index = int(result_match.group(1))
            results.append(result_index)
        elif query_match:
            query = query_match.group(1)
            queries.append(query)
    output = [{"query": query, "relevant_document": contexts[result]["id"]} for query, result in zip(queries, results)]
    return output
```

#1 Regular expressions are used to see which line belongs to which list. You can experiment with more robust expressions if you run into some strange output from the model.

Next, we pass our output from listing 15.8 into the `extract_pairwise_judgments` from listing 15.9.

## Listing 15.10 Positive relevance judgments generated from the LLM

```
resultset_contexts = example_contexts.to_dict("records") #1
output = extract_pairwise_judgments(
    generated_relevance_judgments,
    resultset_contexts)
display(output)
```

#1 `example_contexts`, from listing 15.7, contains the search results for the query 'What are minimalist shoes?'.  
Response:

```
{"query": "What is the definition of a minimalist shoe?", "relevant_document": 18370}
{"query": "What are the characteristics of minimalist shoes?", "relevant_document": 18376}
 {"query": "Which shoes are best for walking on rocky beaches?", "relevant_document": 18370}
 {"query": "Are minimalist shoes suitable for all terrains?", "relevant_document": 18375}
 {"query": "What are some recommended barefoot shoe brands?", "relevant_document": 13540}
```

Running this process through hundreds of result sets taken from customer queries will produce a large list of relevant query-document pairs. Since we are processing five results

at a time, we'll also be generating five new positive pairwise judgments with every single prompt to the LLM. You can then use these judgments as synthetic signals or as training data to train many of the signals-based models explored in the earlier chapters.

#### 15.2.4 Evaluating generative output

In previous chapters, we covered the importance of using judgment lists to measure the quality of our search algorithms. In chapters 10 and 11, we generated judgment lists manually and then later automatically using click models, for training and measuring the quality of LTR models. In chapter 14, we generated silver sets and golden sets to train and measure the quality of our fine-tuned LLM for extractive question answering.

These are all search use cases for which the query and results pairs are largely deterministic. Measuring the quality of generative model output is much trickier, however. We'll explore some of the generative use cases in this section to see what can be done to overcome those challenges.

Generative output can be subjective. Many generative models produce different outputs given the same prompt when you adjust a temperature parameter. *Temperature* is a value between `0.0` and `1.0` that controls the randomness in the output. The lower the temperature, the more predictable the output; the higher the temperature, the more creative the output. We recommend always setting the temperature to `0.0` for the evaluation and production of your model so you have a higher confidence in its output. However, even a temperature of `0.0` may still produce different responses for the same prompt between runs, depending on the model.

The core methodology for evaluating generative output is straightforward: frame and evaluate tasks that can be measured objectively. Since the output of the prompt can be unpredictable, however, it's not easy to curate a dataset that can be reused to measure different prompts for your exact task. Therefore, we typically rely on established metrics to assess the quality of the model chosen first, before you start your own evaluation on its downstream output.

Various important and common metrics address different language tasks. Each metric usually has a leaderboard, which is a competition-style benchmark that ranks models by performance on the given tasks. These are some common benchmarks:

- *AI2 Reasoning Challenge (ARC)*—A multiple-choice question-answering dataset consisting of 7,787 grade-school science exam questions.
- *HellaSwag*—Common-sense inference tests; for example, a complex reasoning question is given with multiple choice.
- *Massive Multitask Language Understanding (MMLU)*—A test to measure a text model's multitask accuracy. The test covers 57 tasks, including mathematics, history, computer science, law, and more.
- *TruthfulQA*—Measures whether a language model is truthful in generating answers to questions. The benchmark comprises 817 questions that span 38 categories. The authors crafted questions that some humans would answer falsely due to a false belief or misconception.

Figure 15.3 shows an example of a HellaSwag question-and-answer test.

#### Pick the best ending to the context.

How to catch dragonflies. Use a long-handled aerial net with a wide opening. Select an aerial net that is 18 inches (46 cm) in diameter or larger. Look for one with a nice long handle.

- |  |  |  |  |
|--|--|--|--|
| a) Loop 1 piece of ribbon over the handle. Place the hose or hose on your net and tie the string securely. | b) Reach up into the net with your feet. Move your body and head forward when you lift up your feet. | c) If possible, choose a dark-colored net over a light one. Darker nets are more difficult for dragonflies to see, making the net more difficult to avoid. | d) If it's not strong enough for you to handle, use a hand held net with one end shorter than the other. The net should have holes in the bottom of the net. |
|--|--|--|--|

Correct! 😊

**Figure 15.3 HellaSwag example question and multiple choices, with the correct answer highlighted**

Some leaderboards contain multiple metrics, such as the Open LLM Leaderboard in figure 15.4.

Model	Average	ARC	HellaSwag	MMLU	TruthfulQA	Winogrande	GSM8K	Hub License
abacusai/Smaug-72B-v0.1	80.48	76.02	89.27	77.15	76.67	85.08	78.7	other
ibivibiv/alpaca-dragon-72b-v1	79.3	73.89	88.16	77.4	72.69	86.03	77.63	other
moxeh/MoMo-72B-Lora-1.8.7-DPO	78.55	70.82	85.96	77.13	74.71	84.06	78.62	mit
cloudyu/TomGrc_FusionNet_34Bx2_MoE_v0.1_DPO_f16	77.91	74.06	86.74	76.65	72.24	83.35	74.45	other
HanNayeoniee/LHK_DPO_v1	77.62	74.74	89.3	64.9	79.89	88.32	68.54	mit
cloudyu/TomGrc_FusionNet_34Bx2_MoE_v0.1_full_linear_DPO	77.52	74.06	86.67	76.69	71.32	83.43	72.93	other
zhengr/MixTAO-7Bx2-MoE-v8.1	77.5	73.81	89.22	64.92	78.57	87.37	71.11	apache-2.0
yuncconglong/Truthful_DPO_TomGrc_FusionNet_7Bx2_MoE_13B	77.44	74.91	89.3	64.67	78.02	88.24	69.52	mit
JaeyeonKang/CCK_Asauxa_v1	77.43	73.89	89.07	75.44	71.75	86.35	68.08	cc-by-nc-4.0
fblgit/UNA-SimpleSmaug-34b-v1beta	77.41	74.57	86.74	76.68	70.17	83.82	72.48	apache-2.0
TomGrc/FusionNet_34Bx2_MoE_v0.1	77.38	73.72	86.46	76.72	71.01	83.35	73.01	mit
mittissera/Tess-72B-v1.5b	77.3	71.25	85.53	76.63	71.99	81.45	76.95	other

**Figure 15.4 HuggingFaceH4 Open LLM Leaderboard space (taken March 2024)**

You should use these metrics to decide which model to use for your specific domain and task. For example, if your task is abstractive question answering, consider using a model with the best TruthfulQA metric.

**Warning** Models are licensed, so make sure you only use a model with a license that aligns with your use case. Some models have licenses similar to open source software (like Apache 2.0 or MIT). But be careful, because many models have commercial restrictions (such as the LLaMA models and derivatives or models trained on output from restrictive models like GPT from OpenAI).

Once you have chosen your model, you should take steps to evaluate it on your prompts. Be rigorous, and track responses to your prompts and how they fare. Some purpose-built tools are emerging in the market to do this for you, including automatic prompt optimization approaches, but a simple spreadsheet can suffice. In the following section, we'll construct our own metric to use for a new task.

#### 15.2.5 Constructing your own metric

Let's explore how a generative task can be construed objectively, allowing you to create a dataset and use a metric like precision or recall (both introduced in section 5.4.5) to evaluate a model's accuracy. We'll use a prompt with unstructured data to extract structured information. Since our result will be formatted predictably, we can then measure the accuracy of the response based on a human-labeled result.

For our example, we'll test the named-entity recognition accuracy of a generative model. Specifically, we'll measure whether generative output from an LLM correctly labels entities of type person, organization, or location.

We'll use the following snippet of text from a news article:

```
Walter Silverman of Brighton owned one of the most successful local Carvel franchises, at East Ridge Road and Hudson Avenue in Irondequoit. He started working for Carvel in 1952. This is how it appeared in the late 1970s/early 1980s.  
[Alan Morrell, Democrat and Chronicle, May 29, 2023]
```

If we manually label the entities in this snippet with the tags `<per>` (person), `<org>` (organization), and `<loc>` (location), we'll arrive at the labeled version in the following listing.

### **Listing 15.11 Article tagged with entity labels**

```
<per>Walter Silverman</per> of <loc>Brighton</loc> owned one of the most successful local <org>Carvel</org> franchises, at <loc>East Ridge Road</loc> and <loc>Hudson Avenue</loc> in <loc>Irondequoit</loc>. He started working for <org>Carvel</org> in 1952. This is how it appeared in the late 1970s/early 1980s.
```

This is a format that a generative model should be able to produce, taking text and generating a marked-up version as instructed by a prompt. But this response is only semi-structured with ad hoc markup. We need to process it further, and we can write some Python to extract the entities into a suitable JSON format.

### **Listing 15.12 Extracting entities from generative output**

```
def extract_entities(text):
    entities = []
    pattern = r"<(per|loc|org)>(.*)</(per|loc|org)>"
    matches = re.finditer(pattern, text)
    for match in matches:
        entity = {"label": match.group(1).upper(),
                  "offset": [match.start(), match.end() - 1],
                  "text": match.group(2)}
        entities.append(entity)
    return entities

entities = extract_entities(news_article_labelled)
display(entities)
```

Output:

```
[{"label": "PER", "offset": [0, 26], "text": "Walter Silverman"},  
 {"label": "LOC", "offset": [31, 49], "text": "Brighton"},  
 {"label": "ORG", "offset": [90, 106], "text": "Carvel"},  
 {"label": "LOC", "offset": [123, 148], "text": "East Ridge Road"},  
 {"label": "LOC", "offset": [154, 177], "text": "Hudson Avenue"},  
 {"label": "LOC", "offset": [182, 203], "text": "Irondequoit"},  
 {"label": "ORG", "offset": [229, 245], "text": "Carvel"}]
```

With the entities now represented in a structured form (as JSON), we can use the list to calculate our metric. We can manually label many passages or use the techniques from sections 14.2–14.3 to automate the labeling, using a model to construct a silver set and then correcting the outputs to produce a golden set.

When you have your golden set, you can try a prompt like the following.

### Listing 15.13 Labeled entities

```
For a given passage, please identify and mark the following entities:  
people with the tag '<per>', locations with the tag '<loc>', and  
organizations with the tag '<org>'. Please repeat the passage below  
with the appropriate markup.  
### {text}
```

In the preceding prompt, `{text}` would be replaced by the paragraph for which you want to identify entities. When the model generates the response, it can be passed directly into the `extract_entities` function in listing 15.12.

The output of the extracted entities can then be compared to the golden set to produce the number of true positives, `TP` (correctly identified entities), false positives, `FP` (incorrectly identified text that should not be identified), and false negatives, `FN` (entities that should have been identified but were not).

From these numbers, you can calculate the following:

- Precision =  $(TP / (TP + FP))$
- Recall =  $(TP / (TP + FN))$
- F1 =  $(2 * (Precision * Recall) / (Precision + Recall))$

Many of the metrics listed on leaderboards use `F1` scores.

What generative tasks do you have? Be creative and think about how you can shape your task to be objectively measurable. For example, if you are generating search result summaries, you can perform something similar to the named-entity recognition task—check if the summaries show important spans of text for a given search result set or show result number citations with titles correctly attributed.

## 15.2.6 Algorithmic prompt optimization

Throughout section 15.2, we've covered the use of LLM prompts to generate synthetic data and to summarize and cite search results for RAG. We also covered metrics to quantify the quality of generated data. We discussed this in the context of measuring the *model's* quality, but there's an important point we glossed over: the quality of the *prompt*.

If you recall, we have three different ways to improve LLM output: improve the model during training (pretraining), fine-tune the model, or improve the prompt. Pretraining and fine-tuning are direct, programmatic optimizations of the neural network model, but we've thus far treated prompt creation and improvement (known as *prompt engineering*) as a manual process requiring human intervention.

In reality, the prompt can also be fine-tuned programmatically much like the model. Libraries like DSPy (Declarative Self-improving Language Programs, Pythonically) provide a framework for “programming” language model usage as opposed to manually prompting it. Functionally, this is accomplished by defining the desired output format from the language model (along with training data showing good responses) and letting the library optimize the prompt’s wording to best achieve that output.

DSPy accomplishes this programmatic prompt optimization using four key components:

- *Signatures*—These are simple strings explaining the goal of the training process, as well as the expected input and output. Examples:
  - `question → answer` (question answering)
  - `document → summary` (document summarization)
  - `context, question → answer` (RAG)
- *Modules*—These take in a signature and combine it with a prompting technique, a configured LLM, and a set of parameters to generate a prompt to achieve the desired output.
- *Metrics*—These are measures of how well the output generated by a module matches the desired output.
- *Optimizers*—These are used to train the model by iteratively testing variations to the prompts and parameters to optimize for the specified metric.

Programming the language model usage like this provides four key benefits:

- It allows for the automatic optimization of the prompt, which can be a time-consuming and error-prone process when done manually.
- It allows for the optimization of the prompt to quickly test well-known templates (and new techniques or data over time) and best practices for prompting, as well as to chain multiple prompting stages together into a more sophisticated pipeline.
- It allows you to easily switch out the LLM at any time and “recompile” the DSPy program to re-optimize the prompt for the new model.
- It allows you to chain multiple modules, models, and workflows together into more sophisticated pipelines, and to optimize the prompts for each stage of the pipeline and the pipeline as a whole.

These benefits make your application more robust and easier to maintain, as compared to manual prompt tuning, and they ensure your prompt is always fine-tuned and optimized for your current environment and goals. Once you've coded up the configured DSPy pipeline, you just run the optimizer to *compile* it, at which point all the optimal parameters are learned for each module. Want to then try a new model, technique, or set of training data? Just recompile your DSPy program, and you're ready to go, with your prompts automatically optimized to yield the best results. You can find lots of "getting started" templates for implementing RAG, question answering, summarization, and any number of other LLM-prompt-based tasks in the DSPy documentation at <https://github.com/stanfordnlp/dspy>. If you're building a non-trivial LLM-based application, we highly recommend you opt for such a programmatic approach to prompt optimization.

## 15.3 Multimodal search

Let's now explore one of the most powerful capabilities enabled by foundation-model-based embeddings: multimodal search. Multimodal search engines allow you to search for one content type using another content type (also called *cross-modal* search), or to search multiple content types together. For example, you can search for an image using either text, an image, or a hybrid query combining both text and an image. We'll implement each of these use cases in this section.

Multimodal search is made possible by the fact that vector embeddings can be generated for text, images, audio, video, and other content that can be mapped into overlapping vector spaces. This enables us to search any content type against any other content type without the need for any special indexing or transformation.

Multimodal search engines have the potential to revolutionize the way we search for information, removing barriers that previously prevented us from using all the available context to best understand the incoming queries and rank results.

### 15.3.1 Common modes for multimodal search

While multimodal search entails many different data types being searchable, we'll briefly discuss the currently most common data modalities in this section: natural language, images, audio, and video.

#### NATURAL LANGUAGE SEARCH

In chapters 13 and 14, we implemented semantic search and question-answering using dense vectors. Both techniques are examples of natural language search. A simple query like `shirt without stripes` will confound every e-commerce platform built atop an inverted index, unless special logic is added manually or integrated into a search-focused knowledge graph using *semantic functions* like we implemented in chapter 7.

Today's state-of-the-art LLMs are becoming more sophisticated at handling these queries and are even able to interpret instructions and synthesize information from different sources to generate new responses. Many companies building traditional databases and

NoSQL data stores are aggressively integrating dense vector support as first-class data types to take advantage of all these major advances.

While it is unlikely that these dense-vector-based approaches will fully replace traditional inverted indexes, knowledge graphs, signals boosting, learning to rank, click models, and other search techniques, we will see new hybrid approaches emerge that combine the best of each of these techniques to continue optimizing content and query understanding.

## IMAGE SEARCH

As you saw in chapter 2, images are another form of unstructured data, alongside text.

Traditionally, if you wanted to search for images in an inverted index, you would search through text descriptions of the image or labels applied to the image. With dense vector search, however, searching images for visual or conceptual similarity can be implemented almost as easily as text search. By taking input images, encoding them into embedding vectors with a vision Transformer, and indexing those vectors, it becomes possible to accept another image as a query, encode it into an embedding, and then search with that vector to find visually similar images.

Further, by training a model with images alongside text descriptions of those images, it becomes possible to perform a multimodal search for images either by inputting an image or by text. Rather than having a caption and an unsearchable image, one can now use convolutional neural networks (CNNs) or vision Transformers to encode the image to a dense vector space that co-exists in the same vector space as text. With image representation and text representation in the same vector space, you can search for a more descriptive version of the image than the caption provides, yet still match using text to describe the features in the image.

Given this ability to encode images and text into the same dense vector space, it is also possible to reverse the search and use the image as the query to match any text documents that include similar language to what's occurring in the image, or to combine both an image and text into a hybrid query to find other images best matching a combination of the text and image query modalities. We'll walk through examples of some of these techniques in section 15.3.2.

## AUDIO SEARCH

Searching with audio or for audio has historically been mostly a text-to-speech problem. Audio files would be converted to text and indexed, and audio queries would be converted to text and searched against the text index. Transcribing voice is a very difficult problem. Voice assistants from Google, Apple, Amazon, and Microsoft typically do very well with short queries, but this level of accuracy has historically been difficult to come by for those wanting to integrate open source solutions. Recent breakthroughs with the combinations of speech recognition and language models that can error-correct phonetic misunderstandings are bringing better technology to the market, however.

It's important to keep in mind that many other kinds of important sounds can be present in audio besides just spoken words. If someone searches for `loud train, rushing`

`stream`, or `Irish accent`, these are all qualities they'd hope to find in any returned audio results. Multimodal Transformer models, when trained by text and audio mapping to overlapping vector spaces, now make this possible.

## VIDEO SEARCH

Video is nothing more than the combination of sequential images overlaid and kept in sequence with audio. If audio and images can both be mapped to an overlapping vector space with written text, this means that by indexing each frame of a video (or maybe a few frames per second, depending on the granularity needed), it's possible to create a video search engine that allows searching by text description for any scene in the video, searching by audio clip, or searching by an image to find the most similar video. The deep learning models being generated in the fields of computer vision, natural language processing, and audio processing are all converging, and as long as these different types of media can all be represented in overlapping vector spaces, search engines can now search for them.

### 15.3.2 Implementing multimodal search

In this section, we'll implement one of the most popular forms of multimodal search: text-to-image search. We'll do this by using embeddings that were jointly trained on image and text data pairs from the internet. This results in the latent text features learned being in the same vector space as the latent image features learned. This also means that in addition to performing text-to-image search, we can use the same embedding model to perform an image-to-image visual search based on the pixels in any incoming image. We'll also show an example of combining modalities in a hybrid query, searching for results that best match both a text query *and* an image at the same time.

We'll use the CLIP model, which is a multimodal model that can understand images and text in the same vector space. CLIP is a Transformer-based model developed by OpenAI that was trained on a large dataset of images and their associated text captions. The model was trained to predict which caption goes with which image, and vice versa. This means that the model has learned to map images and text into the same vector space so that similar images and related text embeddings are located close together.

We will return to The Movie Database (TMDB) dataset we used in chapter 10 for implementing our multimodal search examples. In this case, however, instead of searching on the text of movies, we'll search on images from movies.

In the following listing, we define the functionality to calculate normalized embeddings and build a movie collection. The collection is composed of the movie's image embeddings and movie metadata, including titles, image source URLs, and the URL for the movie's TMDB page.

## Listing 15.14 Indexing cached movie embeddings

```
def normalize_embedding(embedding): #1
    return numpy.divide(embedding, #1
        numpy.linalg.norm(embedding, axis=0)).tolist() #1

def read(cache_name): #2
    cache_file_name = f"data/tmdb/{cache_name}.pickle" #2
    with open(cache_file_name, "rb") as fd: #2
        return pickle.load(fd) #2

def tmdb_with_embeddings_dataframe(): #2
    movies = read("movies_with_image_embeddings") #2
    embeddings = movies["image_embeddings"]
    normalized_embeddings = [normalize_embedding(e) #1
        for e in embeddings] #1
    movies_dataframe = spark.createDataFrame(
        zip(movies["movie_ids"], movies["titles"],
            movies["image_ids"], normalized_embeddings),
        schema=["movie_id", "title", "image_id", "image_embedding"])
    return movies_dataframe

embeddings_dataframe = tmdb_with_embeddings_dataframe() #3
embeddings_collection = engine.create_collection( #3
    "tmdb_with_embeddings") #3
embeddings_collection.write(embeddings_dataframe) #3
```

#1 Normalize movie embeddings at index time so we can use the more efficient dot product (versus cosine) calculation at query time.

#2 We pregenerated and cached the movie embeddings so you don't have to download and process all the images.

#3 Builds a movie collection containing image embeddings

Because the CLIP model is a multimodal model jointly trained on text and images, the embeddings we generate from either text or images can be used to search the same image embeddings.

With the index now built using normalized embeddings, all that's left to do is to execute a vector search against the collection. The following listing shows the key functions needed to search for images using text queries, image queries, or hybrid text and image queries.

## Listing 15.15 Multimodal text/image vector search using CLIP

```
device = "cuda" if torch.cuda.is_available() else "cpu" #1
model, preprocess = clip.load("ViT-B/32", device=device) #1

def movie_search(query_embedding, limit=8): #2
    collection = engine.get_collection("tmdb_with_embeddings")
    request = {"query_vector": query_embedding, #3
                "query_field": "image_embedding", #3
                "return_fields": ["movie_id", "title", #3
                                  "image_id", "score"], #3
                "limit": limit, #3
                "quantization_size": "FLOAT32"} #4
    return collection.search(**request)

def encode_text(text, normalize=True): #5
    text = clip.tokenize([text]).to(device) #5
    text_features = model.encode_text(text) #5
    embedding = text_features.tolist()[0] #5
    if normalize: #5
        embedding = normalize_embedding(embedding) #5
    return embedding #5

def encode_image(image_file, normalize=True): #6
    image = load_image(image_file) #6
    inputs = preprocess(image).unsqueeze(0).to(device) #6
    embedding = model.encode_image(inputs).tolist()[0] #6
    if normalize: #6
        embedding = normalize_embedding(embedding) #6
    return embedding #6

def encode_text_and_image(text_query, image_file): #7
    text_embedding = encode_text(text_query, False)
    image_embedding = encode_image(image_file, False)
    return numpy.average((normalize_embedding( #8
        [text_embedding, image_embedding])), axis=0).tolist() #8
```

```
#1 Loads the pretrained CLIP model and image preprocessor
#2 Executes a vector search for a query embedding
#3 Constructs a search request with a query embedding to search against the indexed image embeddings
#4 The data type used for each embedding feature value, in this case a 32-bit float.
#5 Computes the normalized embeddings for text
#6 Computes the normalized embeddings for an image
#7 Computes and combines normalized embeddings for an image and text
#8 Averages the text and image vectors to create a multimodal query
```

The `movie_search` function follows a process similar to one we used in chapter 13: take a query vector and execute a vector search against a collection with embeddings. Our `encode_text` and `encode_image` functions calculate normalized embeddings based on text or an image. The `encode_text_and_image` function is a hybrid of the two, where we generate embeddings from both text and images, unit-normalize them, and pool them together by averaging them.

With the core multimodal embedding calculations in place, we can now implement a simple search interface to display the top results for an incoming text, image, or text and image query.

#### Listing 15.16 Multimodal vector search on text and image embeddings

```
def search_and_display(text_query="", image_query=None):
    if image_query:
        if text_query:
            query_embedding = encode_text_and_image(text_query, image_query)
        else:
            query_embedding = encode_image(image_query)
    else:
        query_embedding = encode_text(text_query)
    display_results(movie_search(query_embedding), show_fields=False)
```

The `search_and_display` function takes in either a text query, an image query, or both, and then retrieves the embeddings and executes the search. The function then displays the top results in a simple grid. Figure 15.5 shows an example output for the query `singing in the rain`:

```
search_and_display(text_query="singing in the rain")
```

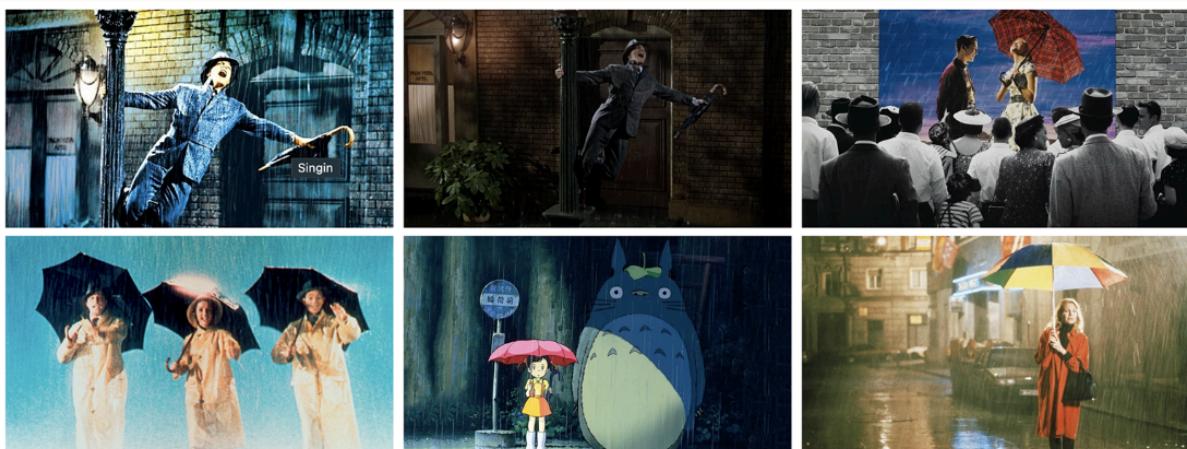
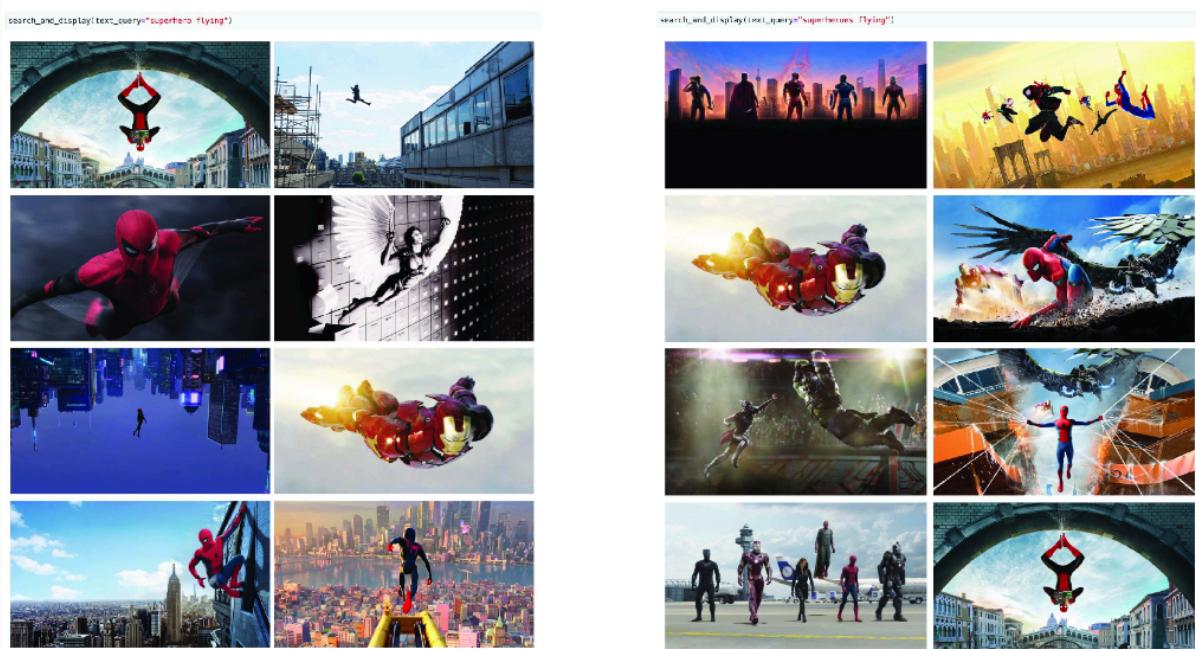


Figure 15.5 Text query for `singing in the rain`

Three of the first four images are from the movie *Singin' in the Rain*, all of the images show someone in the rain or with an umbrella, and many are either from musicals or show people actively singing. These results demonstrate the power of performing image search using embeddings that were trained on both text and images: we now have the ability to search for pictures whose pixels contain the meaning expressed by the words!

To demonstrate some of the nuances of mapping text to meaning in images, let's run two variations of the same query: `superhero flying` versus `superheroes flying`. In traditional keyword search, we'd usually throw away the plural, but in the context of searching on embeddings, and particularly in the context of multimodal search, let's see how this slight difference affects the results. As you can see in figure 15.6, our search results changed from images of a single superhero flying (or at least up high off the ground) to pictures of mostly groups of superheroes containing similar actions.

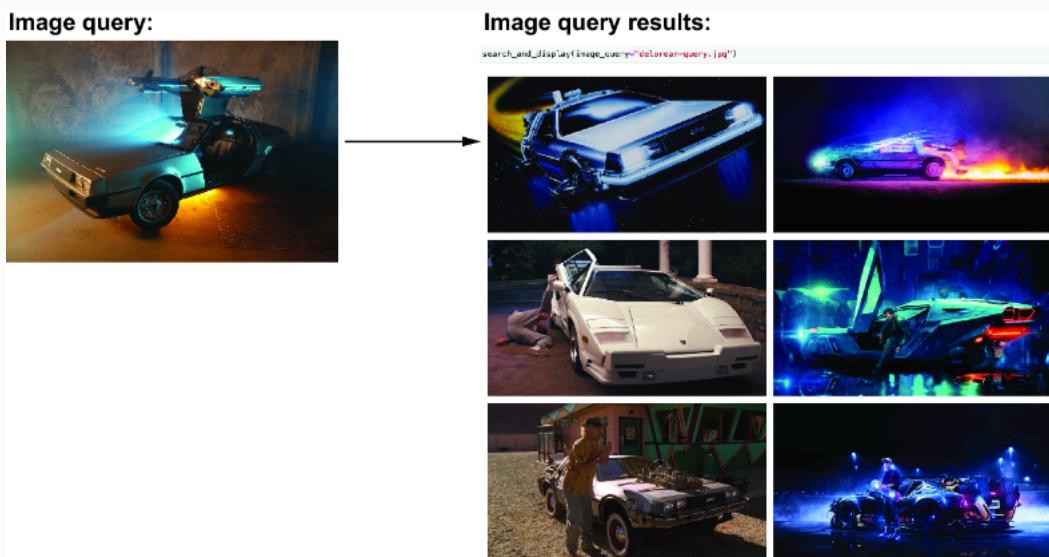
"superhero flying" vs. "superheroes flying"



**Figure 15.6** Nuanced difference between `superhero flying` vs. `superheroes flying` queries

This cross-modal (text-to-image) search is impressive, but let's also test out image-to-image search to demonstrate how reusable these multimodal embeddings are. Figure 15.7 shows the results of an image search using a picture of a DeLorean (the car famously converted to a time machine in the movie *Back to the Future*).

```
search_and_display(image_query="chapters/ch15/delorean-query.jpg")
```



**Figure 15.7** Image-to-image search for a car that looks similar to a DeLorean

As you would expect, most of the cars are actually the famous DeLorean from *Back to the Future*. The other cars have similar aesthetic features: similar shapes, similarly opening doors. Not only have we matched the features of the car well, but many of the results also reflect the glowing lighting from the image query.

To take our DeLorean search one step further, let's try a multimodal search combining our last two query examples. Let's perform a multimodal query for both the previous DeLorean image (image modality) and the text query `superhero` (text modality). The output of this query is shown in figure 15.8.

```
search_and_display(text_query="superhero",
                   image_query="chapters/ch15/delorean-query.jpg")
```

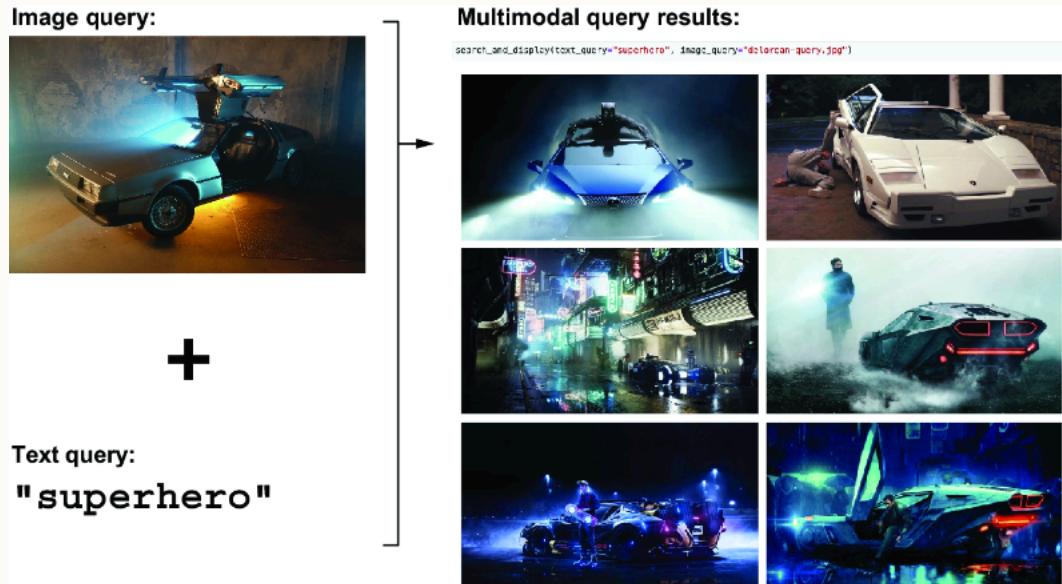


Figure 15.8 Multimodal search for an image and text query

Interesting! The first image is of a superhero (Black Panther) on top of a sporty car, with lights glowing from the car, similar to the original picture. Most of the results now show the heroes or protagonists of their movies, with each of the results containing sporty cars and most images containing illuminating lighting effects picked up from the image query. This is a great example of how multimodal embeddings can be used to infer nuanced intent and to enable querying in new ways across diverse types of datasets.

The implications of multimodal search across text and images are vast. In e-commerce search, someone can upload a picture of a part they need and immediately find it without knowing the name, or upload a picture of an outfit or furniture that they like and find similar styles. Someone can describe a medical problem and immediately find images reflecting the symptoms or body systems involved. From chapter 9, you may remember our work to learn latent features of items and users from user behavior signals. This behavior is yet another modality, like images, that can be learned and cross-trained with text and other modalities to increase the intelligence of your search engine. As we'll discuss further in section 15.5, many additional data modalities like this will continue to become commonly searchable in the future, helping to make search engines even more powerful at understanding and finding relevant content and answers.

## 15.4 Other emerging AI-powered search paradigms

Throughout this book, you've learned techniques in most key AI-powered search categories, such as signals processing and crowdsourced relevance, knowledge graphs, personalized search, learning to rank, and semantic search using Transformer-based embeddings.

The introduction of foundation models and the ability to represent language as dense vectors has revolutionized how we think about and build search engines in recent years. Whereas search used to be primarily text-based, now it is possible to search on anything

that can be encoded into dense vectors. This includes searching on images, audio, video, concepts, or any combination of these or other modalities.

Additionally, the way that users interface with search continues to evolve in new and groundbreaking ways. This includes the ability to ask questions, receive summarized answers that are a combination of multiple search results, generate new content explaining search results, and generate images, prose, code, instructions, or other content. Chatbot interfaces, combined with context tracking throughout conversations, have enabled iterative refinement of search tasks, where users can provide real-time feedback on responses and enable AI to act as an agent to search, synthesize, and refine the relevance of search results based upon live user feedback.

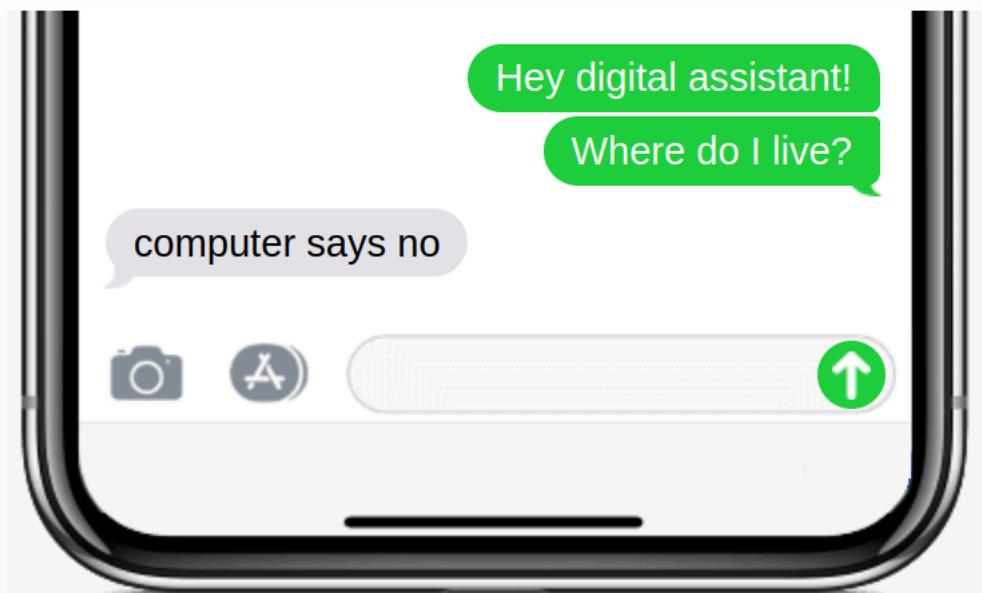
In this final section of our AI-powered search journey, we'll touch on a few of the trends we're seeing evolve and how they are likely to shape the future of search.

#### 15.4.1 Conversational and contextual search

We've spent significant time covering contextual search—understanding the content context, the domain context, and the user context when interpreting queries. As search engines become more conversational, the personal context begins to take even more priority. For example,

```
User: "Please, take me home."  
Phone: "I don't know where you live."
```

Despite someone having their address bookmarked under the name “Home” and their phone tracking their locations daily (including where they sleep at night), some highly used digital assistants are still unable to automatically bring in that context, like in figure 15.9, and instead require your home location to be explicitly configured in the settings. These shortcomings are likely to disappear in the coming years as personal assistants build much more precise personalization models from all available contexts.



**Figure 15.9** Digital assistants will need to pull in many different data sources to build out robust personalization models

In addition to these personalization contexts, there's an additional context that is becoming increasingly important: conversational context. Here's an example of a chatbot (virtual assistant) that lacks awareness of the current conversational context:

```
User: "Take me to the plumbing store."  
Phone: "Here is a list of stores that I found. [shares list]"  
  
User: "Go to the one in East Rochester."  
Phone: [Provides directions to the geographical center of East Rochester,  
not the plumbing store it suggested in the last exchange]
```

Good chatbots integrate a strong search index to power their ability to answer questions and return information, but many still lack short-term and long-term memory of the previous conversational context. They may search each query independently, without considering the previous queries and responses. Depending on the application and type of query, some previous conversational context makes sense to remember indefinitely as part of a personalization model, such as someone's home address, family member names, or frequented places (doctor, dentist, school, etc.). In many cases, though, short-term memory just during the current conversation is sufficient. As search experiences become more conversational, short-term and long-term memory of the conversational context becomes more important.

Foundation models trained for chat contexts, such as ChatGPT by OpenAI, have brought about drastic improvements to the conversational capabilities of chatbots. Although they are pretrained models that don't update in real time, additional context can be injected into their prompts, making it entirely possible to capture and add personalized contexts into future prompts to improve their ability to cater to each user.

#### 15.4.2 Agent-based search

Just as we discussed the importance of pipelines in chapter 7 for best interpreting and representing the meaning of user queries, pipelines are also an integral component for the kind of multistep problem-solving that is emerging with chatbots and foundation model interactions.

With agent-based search, a search engine or other AI interface is given a prompt and can then generate new prompts or tasks, chaining them together to achieve a desired outcome. Solving a user's request may require multiple steps, like generating new tasks ("now go find the top websites on this topic" or "explain if your last answer was correct"), reasoning about data ("now combine lists that were pulled from each website", or "summarize the results"), and other functional steps ("now return the search results").

A significant portion of web traffic in the future will originate from AI-based agents searching for information to use in their problem-solving. The degree to which web search engines act as the data-serving layer for these agents, since web search engines typically have a cached copy of much of the web already, remains to be seen, but search engines are natural launch points for these AI-based agents. Major search engines like Bing, as well as

several startups, have already rolled out multistep searches involving some level of iterative task-following to compose better-researched responses. This is likely to be a growing trend for some time to come.

## 15.5 Hybrid search

In chapters 2 and 3, we introduced two different search paradigms: keyword-based lexical search (based on sparse vector representations, like an inverted index and typically ranked using BM25) and dense vector search (based on dense vector representations and typically ranked using cosine, dot product, or a similar vector-based similarity calculation). In this section, we'll demonstrate how to combine results across these different paradigms.

We've mostly treated lexical and vector search as orthogonal search approaches, but in practice, you'll often get the best results by using both approaches as part of a hybrid search. *Hybrid search* is the process of combining results from multiple search paradigms to deliver the most relevant results. Hybrid search usually involves combining lexical and vector search results, though the term isn't limited to just these two approaches.

There are a few ways to implement hybrid search. Some engines have direct support for combining lexical query syntax and vector search syntax as part of the same query, effectively allowing you to swap out specific keywords or filters within your query with vectors and combine vector similarity scores, BM25 scores, and function scores in arbitrarily complex ways. Some engines don't support running vector and lexical search as part of the same query, but instead support hybrid search *fusion* algorithms that enable you to merge results from separate lexical and vector queries in a thoughtful way. In other cases, you may be using a vector database that is separate from your lexical search engine, in which case you may end up using a similar fusion algorithm to combine those results outside of the engine.

### 15.5.1 Reciprocal rank fusion

In this section, we'll demonstrate a popular hybrid search algorithm for merging two sets of search results, called *reciprocal rank fusion* (RRF). RRF combines two or more sets of search results by ranking the documents based on their relative ranks across the different result sets. The algorithm is simple: for each document, it sums the reciprocal of the ranks of the document in each result set and then sorts the documents based on this sum. This algorithm is particularly effective when the two sets of search results are complementary, as it will rank documents higher that are already ranked higher in either set, but highest when they are ranked in both sets. The following listing implements the RRF algorithm.

## Listing 15.17 RRF for combining multiple sets of search results

```
def reciprocal_rank_fusion(search_results, k=None):
    if k is None: k = 60
    scores = {}
    for ranked_docs in search_results: #1
        for rank, doc in enumerate(ranked_docs, 1): #2
            scores[doc["id"]] = (scores.get(doc["id"], 0) + #3
                                  (1.0 / (k + rank))) #3
    sorted_scores = dict(sorted(scores.items(), #4
                                key=lambda item: item[1], reverse=True)) #4
    return sorted_scores #4
```

#1 `search_results` is a list of sets of ranked documents associated with different searches (lexical search, vector search, etc.).

#2 `ranked_docs` is a list of documents for a specific search.

#3 A document's score increases by being in multiple `ranked_docs` lists and by being higher in each list.

#4 Return the docs, sorted from highest RRF score to lowest.

In this function, an arbitrary list (`search_results`) of sets of ranked documents from each individual search is passed in. For example, you may have two items in `search_results`:

- The ranked docs from a lexical search
- The ranked docs from a vector search

The RRF score for each document is then calculated by summing the reciprocal ranks of the document ( $1 / (k + \text{rank})$ ) from each set of search results (`ranked_docs`). The final RRF scores are then sorted and returned.

The `k` parameter is a constant that can be increased to prevent an outlier ranking highly in one search result from carrying too much weight. The higher the `k`, the more weight is given to the docs that appear in multiple ranked document lists, as opposed to docs that rank higher in any given list of ranked documents. The `k` parameter is often set to `60` by default, based on research showing that this value works well in practice (<https://plg.uwaterloo.ca/~gvcormac/cormacksigir09-rrf.pdf>).

We've integrated the RRF logic into the `collection.hybrid_search` function, which we'll invoke in listing 15.19. First, however, let's see what the initial results look like for a lexical search versus the corresponding vector search for a sample phrase query: "singin' in the rain".

## Listing 15.18 Running lexical and vector searches independently

```
over_request_limit = 15
base_query = {"return_fields": ["id", "title", "id", "image_id",
                                 "movie_id", "score", "image_embedding"],
              "limit": over_request_limit,
              "order_by": [("score", "desc"), ("title", "asc")]}

def lexical_search_request(query_text):
    return {"query": query_text,
            "query_fields": ["title", "overview"],
            "default_operator": "OR",
            **base_query}

def vector_search_request(query_embedding):
    return {"query": query_embedding,
            "query_fields": ["image_embedding"],
            "quantization_size": "FLOAT32",
            **base_query}

def display_lexical_search_results(query_text):
    collection = engine.get_collection("tmdb_lexical_plus_embeddings")
    lexical_request = lexical_search_request(query_text)
    lexical_search_results = collection.search(**lexical_request)

    display_results(lexical_search_results, display_header= \
                    get_display_header(lexical_request=lexical_request))

def display_vector_search_results(query_text):
    collection = engine.get_collection("tmdb_lexical_plus_embeddings")
    query_embedding = encode_text(query_text)
    vector_request = vector_search_request(query_embedding)
    vector_search_results = collection.search(**vector_request)

    display_results(vector_search_results, display_header= \
                    get_display_header(vector_request=vector_request))

query = ''' + "singin' in the rain" + '''
display_lexical_search_results(query)
display_vector_search_results(query)
```

This listing takes the lexical phrase query `"singin' in the rain"` and encodes it as an embedding using the same `encode_text` function from listing 15.15. It then executes a lexical search and a vector search against the `tmdb_lexical_plus_embeddings` collection, which contains both the text fields needed for a lexical search and an image embedding for some of the movies in the TMDB dataset. The results from the lexical search and vector search are shown in figures 15.10 and 15.11, respectively.

# Lexical Query: "singin' in the rain"

## Singin' in the Rain ( score: 6.5101776 )



Figure 15.10 A single lexical search result for the phrase query "singin' in the rain"

The lexical search, in this case, only returns one result: the movie *Singin' in the Rain*. Since the user's query was a phrase (quoted) query for the exact name of the title, it matched perfectly and found only the specific item the user was likely looking for. Figure 15.11 shows the corresponding vector search results for the same query.



Figure 15.11 Vector search results for the query "singin' in the rain"

You'll notice that the vector search results generally show images containing "rain" or "singing" or matching similar concepts, like umbrellas, weather, and musicals. This is because our vector search is on image embeddings, so the results are based on the visual content of the images, not the text content of the movies. While these images conceptually match the meaning of the words in the query better than the lexical search results, they don't contain the exact title match that the lexical search results do, so the ideal document for the movie *Singin' in the Rain* is missing from the top few results.

By combining these two sets of results using RRF, we can get the best of both worlds: the exact title match from the lexical search and the conceptually relevant images from the vector search. In the following listing, we demonstrate how to combine these two sets of search results by invoking `collection.hybrid_search`.

## Listing 15.19 Hybrid search function

```
def display_hybrid_search_results(text_query, limit=8):
    lexical_search = lexical_search_from_text_query(text_query)
    vector_search = vector_search_from_embedding(encode_text(text_query))
    hybrid_search_results = collection.hybrid_search(
        [lexical_request, vector_request], limit=10,
        algorithm="rrf", algorithm_params={"k": 60})
    display_header = get_display_header(lexical_search, vector_search)
    display_results(hybrid_search_results, display_header)

display_hybrid_search_results(query)
```

We pass in an array of searches to execute—in this case, our `lexical_search` and `vector_search` from listing 15.18. The `collection.hybrid_search` function internally defaults to the RRF algorithm with `k=60` set, so if you'd like to pass in these parameters explicitly or change them, the full syntax is

```
collection.hybrid_search([lexical_search, vector_search], limit=10,
                           algorithm="rrf", algorithm_params={"k": 60})
```

The call executes both searches and combines the results using RRF, with the output shown in figure 15.12.

Hybrid Results:

```
--Lexical Query: "singin' in the rain"
--Vector Query: [0.048921154115761124, 0.024482925930009714, ...]
```

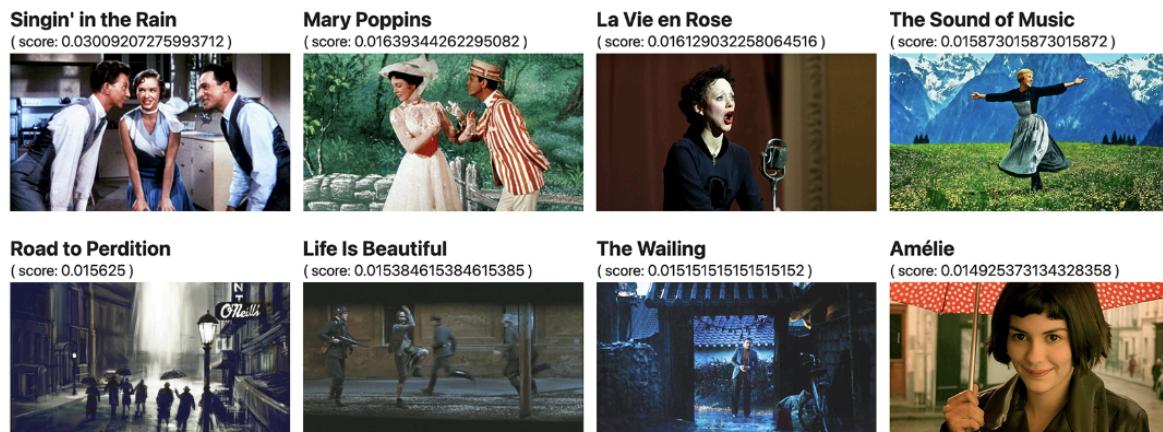


Figure 15.12 Hybrid search results for lexical and vector searches using RRF for the query "singin' in the rain"

These results demonstrate how hybrid search can often provide the best of both worlds: the exact keyword matches from the lexical search and the conceptually relevant results from the vector search. Vector search often struggles with matching exact names, specific keywords, and product names or IDs. Likewise, lexical search misses conceptually relevant results for the query text. In this case, the top result is the exact match the user was likely looking for (previously missing from the vector search results), but it has now been supplemented with other results conceptually similar to the user's query (previously missing from the lexical search).

Of course, you can pass in more than two searches to the hybrid search function if you want to. For example, you may want to add an additional vector field containing an embedding for the `title` or `overview` so that instead of just lexical search on text content, and vector search on images, you also have semantic search on the text content. Using RRF with these three sets of search results would allow you to combine the best of all three search approaches into a single set of results.

In addition to overcoming the respective functional limitations of keyword and vector search, hybrid search often yields better overall search ranking. This is because algorithms like RRF are designed to rank documents highest when they're found in multiple sets of search results. When the same items are returned as relevant from different searches, the fusion algorithm can use that agreement between the sets of results to boost those items' scores. This is particularly useful when one or more sets of search results have irrelevant documents, as the agreement between the two result sets can help filter out the noise, and surface the most relevant results. Listing 15.20 demonstrates this concept using the query `the hobbit`.

#### **Listing 15.20 Lexical, vector, and hybrid searches for `the hobbit`**

```
query = "the hobbit"
display_lexical_search_results(query)
display_vector_search_results(query)
display_hybrid_search_results(query)
```

Figures 15.13 through 15.15 show the results from listing 15.20.

In figure 15.13, you'll see five relevant results in the top six results (*The Hobbit* is part of the *Lord of the Rings* franchise). The text “The Hobbit” appears in the title of three of the first four results. There is one clearly bad result in the fifth position of the lexical search results, and all results are bad after the sixth document, mostly just matching on keywords like “the” and “of” in the `title` and `overview` fields.

Figure 15.14 shows the vector search results for the same query. These also show five relevant results related to *The Lord of the Rings*, but one of the relevant results from the lexical search is missing, and an additional relevant movie has been returned in the vector search results.

Lexical Query: the hobbit

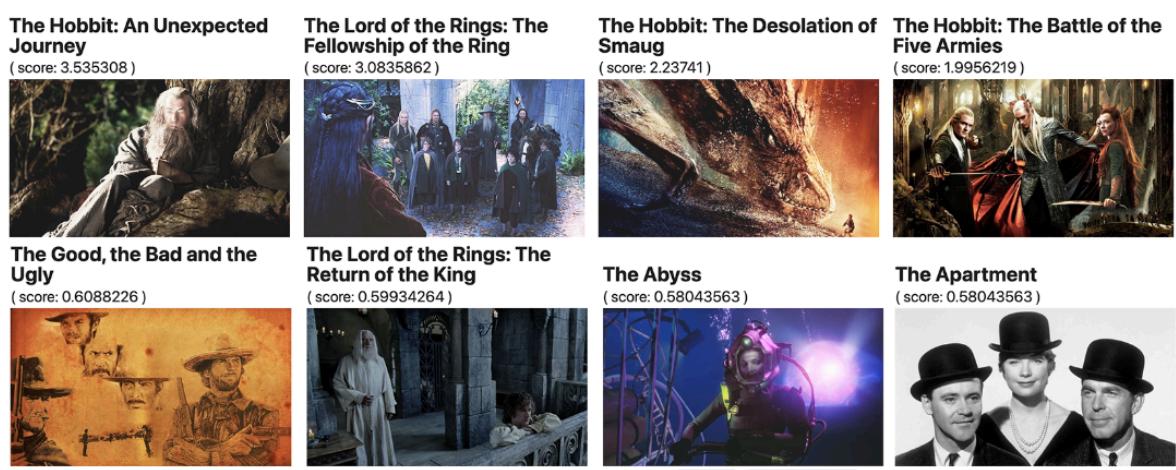


Figure 15.13 Lexical search results for the query the hobbit

Vector Query: [-0.016278795212303375, -0.014356400471339553, ...]

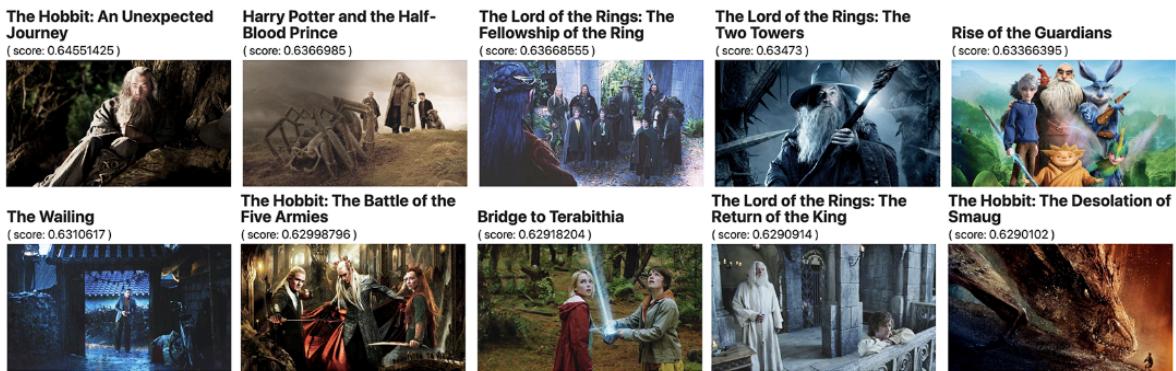


Figure 15.14 Vector search results for the query the hobbit

Many of the remaining results are conceptually related to the query, showing mostly movies with fantasy landscapes and magic. Given the heavy overlap of good results between the lexical and vector search results, as well as the lack of overlap between the less relevant results, we should expect the hybrid search results to be quite good.

Indeed, in figure 15.15, we see that the first five results are all relevant and that six of the first seven results are related to *The Lord of the Rings*. We've moved from two different search results lists that were each missing a different document and showing some irrelevant results to a final list of results.

Hybrid Results:

--Lexical Query: the hobbit

--Vector Query: [-0.016278795212303375, -0.014356400471339553, ...]



Figure 15.15 Hybrid search results for the lexical and vector search results using RRF for the query the hobbit

RRF has allowed us to surface missing results from both lists and push down irrelevant results only appearing in one list, truly emphasizing the best qualities of each search paradigm (lexical versus vector) while overcoming each of their weaknesses.

### 15.5.2 Other hybrid search algorithms

While RRF is a popular and effective algorithm for combining search results, there are, of course, many other algorithms that can be used for similar purposes. One popular algorithm is *relative score fusion* (RSF), which is similar to RRF but uses the relative scores of documents in each search result set to combine the results. Since relevance scores are often not comparable across different ranking algorithms, the scores per query modality are typically scaled to the same range (often 0.0 to 1.0) based on the minimum and maximum scores from each modality. The relative scores are then combined using a weighted average, with the weights often set based on the relative performance of each modality on a validation set.

Another common way to combine search results is to use one modality for an initial match and then rerank the results using another modality. Many engines support both lexical search and vector search as separate operations but allow you to run one query version (such as a lexical query) and then rerank the resulting documents using the other version (such as a vector query).

For example, if you want to ensure you match on specific keywords but also boost results that are conceptually similar to the query, you might run a lexical search first and then rerank the results using a vector search. The following listing demonstrates this concept.

#### Listing 15.21 Hybrid lexical search with vector search reranking

```
def lexical_vector_rerank(text_query, limit=10):
    lexical_request = lexical_search_request(text_query)
    vector_request = vector_search_request(encode_text(text_query))
    hybrid_search_results = collection.hybrid_search(
        [lexical_request, vector_request],
        algorithm="lexical_vector_rerank", limit=limit)
    header = get_display_header(lexical_request, vector_request)
    display_results(hybrid_search_results, display_header=header)

lexical_vector_rerank("the hobbit")
```

Internally, this translates into a normal lexical search, but with the results then re-sorted using the score of a vector search for the same (encoded) query. Syntactically, these are the key parts of the search request generated by listing 15.21:

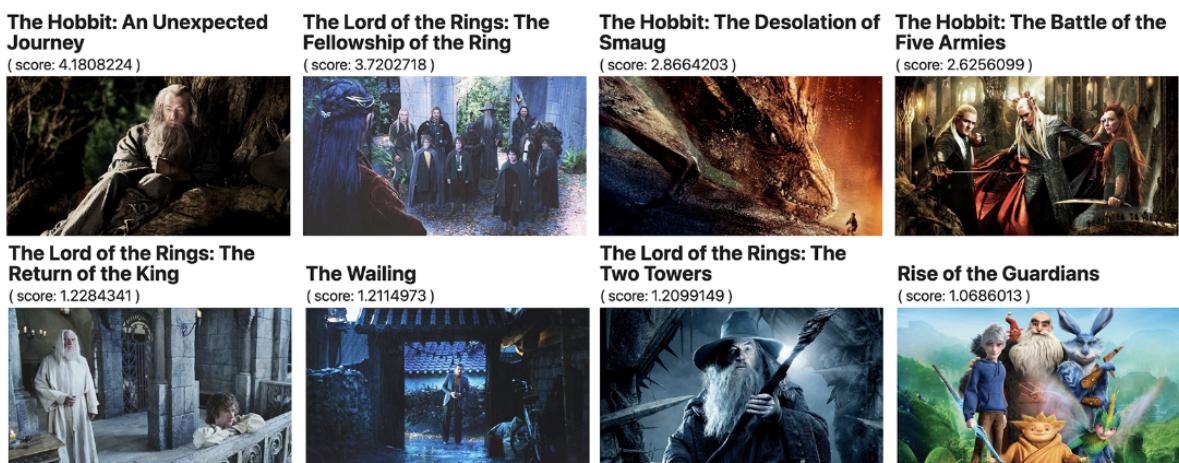
```
{'query': 'the hobbit',
'query_fields': ['title', 'overview'],
'default_operator': 'OR',
...
'order_by': [('score', 'desc'), ('title', 'asc')],
'rerank_query': {
    'query': [-0.016278795212303375, ..., -0.02110762217111629],
    'query_fields': ['image_embedding'],
    'quantization_size': 'FLOAT32',
    ...
    'order_by': [('score', 'desc'), ('title', 'asc')], ...
}
}
```

The output of this lexical search with vector search reranking is shown in figure 15.16. In this particular case, the results look very similar to the RRF results. Note that, unlike in figure 15.10 where the default operator for the lexical query was `AND`, resulting in a single exact lexical match, here the default operator is `OR`, so that more results are returned for the vector search to rerank. If you'd like to increase precision and only return more precise lexical matches, you can set the default operator to `AND` or add a `min_match` threshold in the lexical search request.

#### Hybrid Results:

--Lexical Query: the hobbit

--Vector Query: [-0.016278795212303375, -0.014356400471339553, ... ]



**Figure 15.16** Hybrid search by vector reranking of a lexical query

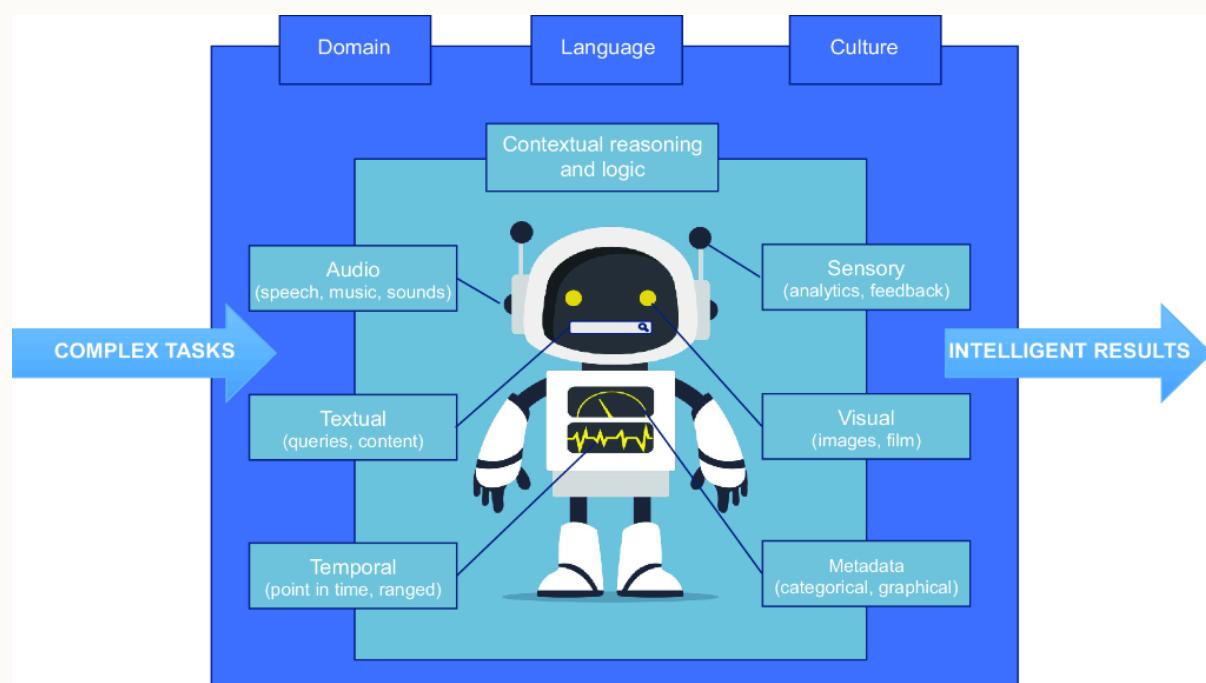
Functionally, the lexical search is entirely responsible for which documents return, while the vector search is entirely responsible for the order of the results. This may be preferred if you want to emphasize lexical keyword matching but with a more semantically relevant ranking. On the other hand, a fusion approach like RRF or RSF will provide a more blended approach, ensuring that the best results from each modality are surfaced. There are many other ways to combine search results, and the best approach will depend on the specific use case and the relative strengths and weaknesses of each search modality.

## 15.6 Convergence of contextual technologies

Just as recommendations, chatbots, and question-answering systems are all types of AI-powered information retrieval, many other technologies are also beginning to converge with search engines. Vector databases are emerging that function like search engines for

multimodal data, and many different data sources and data modalities are being pulled together as additional contexts for optimal matching, ranking, and reasoning over data. Generative models are enabling a base-level understanding sufficient to generate new written and artistic works.

But most of these technologies are still implemented piecemeal into production systems. As researchers continue working toward building artificial general intelligence, intelligent robots, and smarter search systems, we're likely to see a continued convergence of technologies integrating new and different contexts to build a more comprehensive understanding of content, domains, and users. Figure 15.17 demonstrates what this convergence of technologies is likely to look like over the coming years.



**Figure 15.17 The convergence of contextual technologies to deliver more intelligent results**

In the figure, we see the textual, audio, and video modalities we've already discussed, as well as a metadata modality that may provide additional context. We see a sensory modality, which might be available for a physical network of sensor-equipped devices or a robot with direct interactive access to the physical world. We also see the temporal modality, as the timing of both the queries and the data being searched can influence the relevance of the results.

Just as LLMs learn both the domain and the language structure of text (see chapter 13), additional cultural contexts and customs can also be learned by foundation models based on the geography and the observed behaviors from real-world interactions.

Much of this may seem *very* far away from traditional search, or even AI-powered search today—and it is! However, the goal of search is to best meet a user's information need, and as more overlapping technologies emerge to solve this problem from different angles, we see search being a critical base layer with which many of these technologies will integrate.

Many AI practitioners already recognize the importance of using RAG to find and supply real-time context to generative AI models, and data security, data accuracy, and model size considerations make it almost certain that search engines will be a critical backbone for

knowledge-based generative AI systems well into the future. Large amounts of data coming from many sources will need to be indexed and available for real-time retrieval and use by generative AI models.

Having these technologies converge into end-to-end systems will enable solving these AI-powered search problems better than any of these systems could in isolation. Whether this gets labeled “robotics”, “AI”, “general intelligence”, “AI-powered search”, or something else entirely is yet to be seen. But many of the AI-powered search techniques you’ve learned have been, and will continue to be, central to the development of these next-generation contextual reasoning engines.

## 15.7 All the above, please!

Throughout this book, we’ve done a deep dive through the core concepts of building AI-powered search. We’ve covered both the theory behind how modern AI-powered search relevance works, and we’ve walked through code examples demonstrating each topic with real-world use cases.

The techniques and algorithms we’ve looked at won’t be applicable to all use cases, but usually by combining multiple approaches, you will deliver a more effective and relevant search experience.

While the field of AI-powered search is evolving quickly, particularly due to the rapid innovation happening with generative foundation models and dense vector approaches to search, the core principles remain the same. The job of a search engine is to understand user intent and to return the content that best fulfills each user’s information needs. This requires a proper understanding of the content context, the user context, and the domain context for each interaction. In addition to learning language models and knowledge graphs from your content, your search engine’s capabilities will be supercharged if it can learn from your users implicitly, through their interactions (user signals) with the search engine.

You’ve learned how key AI-powered search algorithms work and how to implement and automate these algorithms into a self-learning search engine. Whether you need to automate the building of signals boosting models, learning to rank models, click models, collaborative filtering models, knowledge graph learning, or fine-tuning of deep-learning-based Transformer models for enhanced semantic search and question answering, you now have the knowledge and skills needed to implement a world-class AI-powered search engine. We look forward to seeing what you build!

## Summary

- Foundation models serve as base models that are trained and can later be fine-tuned for specific domains or tasks.
- Prompt engineering allows you to inject additional context and data into each request, providing a way to perform real-time fine-tuning of the content that will be returned for the request. LLM-based applications should ideally be programmed to autogenerate and optimize prompts, as opposed to requiring manual prompt verification and adjust-

ments, so they can automatically handle model and environment changes over time, while still performing optimally.

- Search results summarization and training data generation are two key areas in which foundation models can help drive relevance improvements in search engines.
- Jointly training embedding models on multiple data types enables powerful multimodal search capabilities (text-to-image, image-to-image, hybrid text-plus-image-to-image, and so on) extending user expressiveness and the search engine's ability to interpret user intent.
- AI-powered search is rapidly evolving with the rise of large language models, dense vector search, multimodal search, conversational and contextual search, generative search, and emerging hybrid search approaches.
- There are many techniques for implementing AI-powered search, and the best systems in the future will be those that can effectively apply multiple relevant approaches within hybrid search systems.

Previous chapter

< [14 Question answering with a fine-tuned large language model](#)

Next chapter

[appendix A Running the code examples](#) >