O'REILLY®

Compliments of aws

Natural Language and Search

Large Language Models (LLMs) for Semantic Search and Generative AI

Jon Handler,
Milind Shyani
& Karen Kilroy

REPORT

Natural Language and Search

Large Language Models (LLMs) for Semantic Search and Generative AI

Jon Handler, Milind Shyani, and Karen Kilroy

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Natural Language and Search
by Jon Handler, Milind Shyani, and Karen Kilroy

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Michelle Smith
Development Editor: Sarah Grey
Production Editor: Katherine Tozer
Copyeditor: nSight, Inc.
Proofreader: Christina Chapman
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

April 2024: First Edition

Revision History for the First Edition
2024-03-22: First Release

This work is part of a collaboration between O'Reilly and AWS. See our statement of editorial independence.

978-1-098-15623-7

[LSI]

Table of Contents

CHAPTER 1
Finding Information in Your Data

Ever since people began recording information, we have needed to find that information. More than 4,000 years ago, in ancient Sumer, bureaucrats were creating catalogs to help organize and locate information. In ancient Greece 2,250 years ago, the scholar Callimachus created the *Pinakes*—a table of authors and works in the Library of Alexandria.[1] Numerous concordances and indices surround ancient texts to help readers find relevant portions.

The story of search and search engines is intimately tied to language. This report is thus largely about language: the means by which people capture information and the tool they use to find information. When you search, whether you're talking to a librarian or typing words into a search box, you're asking a question with words to get something you need. Language is the central element of searching.

Until recently, computers could not simulate human-level comprehension, so finding information was a tedious and error-prone process. The search engine, much like when you use a library card catalog, matches the words you type with words in its catalog of information (its *index*). This is called *lexical* or *keyword* search. This process of locating information is iterative, manual, and often frustrating (Figure 1-1).

1 The Library of Congress, *The Card Catalog: Books, Cards, and Literary Treasures* (San Francisco: Chronicle Books, 2017).

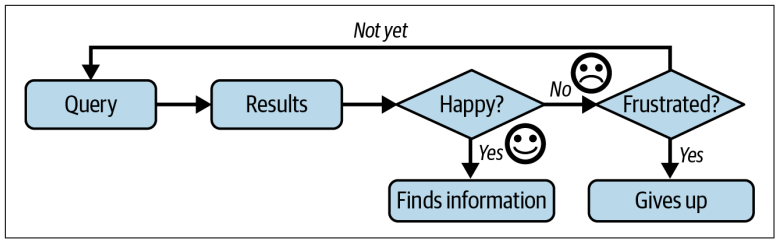

Figure 1-1. The traditional method of iterative searching puts all the work on the searcher

In 2022, decades of research on language in computers culminated in OpenAI releasing ChatGPT, a chatbot that uses a *large language model* (LLM) to generate text responses to text questions. ChatGPT is a *generative pretrained transformer*, which is a class of LLMs that produce text, vectors, images, videos, and more. LLMs capture relationships between words to model their relationships for text, image, and vector generation. ChatGPT and its cousins—such as AI assistants, code generators, and agents—have changed the public's expectations about how we interact with digital tools. ChatGPT's startlingly accurate responses to language prompts have brought excitement and hype around finding and working with information in the digital sphere.

We are on the cusp of a search revolution that will combine the conversational capabilities of chatbots and AI assistants with search engines' ability to match and sift through huge volumes of information quickly. For example, a technique called *retrieval-augmented generation* (RAG) employs a search engine to retrieve relevant information that augments and improves the user's query, enabling an LLM to generate more accurate and relevant answers. (We'll discuss RAG in Chapter 4.) Taken together, LLMs and search engines have shifted information retrieval from user-driven query and refinement to something that looks more conversational (Figure 1-2). It's starting to feel like you are talking to the librarian of Alexandria instead of digging through the library's card catalog!

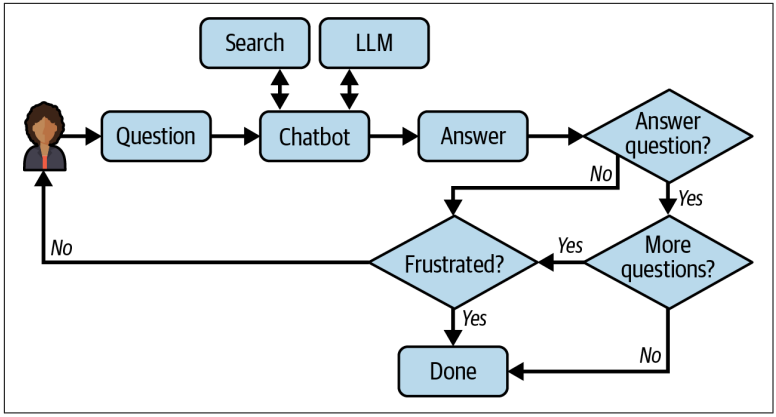

Figure 1-2. RAG supports conversational Q&A, finding answers by using a chatbot and an LLM in tandem

This report will familiarize you with the possibilities that are just opening up in the world of information retrieval. Builders will learn the fundamentals of semantic search and generative AI and where to incorporate them in your application. Product managers will understand the "whys," and the "hows," guiding how to fit LLMs into your product strategy. Executives seeking to capture the power of generative AI (genAI) will learn the art of the possible to shape your teams' direction.

This chapter defines *search engine*, covers their main capabilities, and sets the stage for understanding chatbots and digital assistants. Chapter 2, "Lexical Search" dives into lexical search, covering the core search algorithm and the role of language. Chapter 3, "Vectors: Representing Semantic Information" helps demystify and define vectors and explains why and how vectors are important for processing language. Chapter 4, "Semantic Search" covers semantic search. Chapter 5, "Building with Search" gives you some implementation details and helps you understand how to build ML-driven search. Finally, Chapter 6, "Deploying a Winning Search Strategy" provides hints and example use cases for vector search, strategies for implementing semantic search, and thoughts on governance and ethics.

What's Search and What's a Search Engine?

The purpose of search is to locate information that is relevant to a task at hand. Broadly speaking, whether you're chatting with an AI chatbot or typing words into a search box, you are searching. The technologies that facilitate your path from question to answers begin with search engines. OpenSearch, to name one of the many search engine solutions available, is an open source search and analytics suite integrated with latest genAI and vector search capabilities.

Search websites like *Google.com* are so useful that *search engine* has come to refer not so much to a term of art or a technology, but as a household word, like *Frisbee* or *Band-Aid*—trademarked words that have become so common that they have lost their original brand affinity. In this report, we use *search engine* or *OpenSearch* to refer to the database technology, and *search application* to refer to websites and desktop and mobile apps that employ search in the context of that search application.

As a technology, search engines are closely aligned with and usually described as databases. They fit the high-level characteristics of a database; *databases* store information and enable retrieval of that information. But search engines differ in some important respects from relational databases, NoSQL databases, caches, graph databases, document databases, and the like. At their inception, search engines were developed for low latency and high throughput, trading off transactional behavior and relational representation. The most common architectural pattern is to use a search engine for retrieval and a relational (or other) database for durable primary storage.

Search engines have two core capabilities—indexing and retrieval. To build a search experience, application builders send information to the engine as structured documents. *Document*, in this context, is a term of art referring to a single entity that the engine indexes and that search queries retrieve. The engine indexes the information in the fields of the search documents, providing fast matching and retrieval for text, numbers, dates, geographical coordinates, vectors, and other special types, like Internet Protocol (IP) addresses. We'll discuss how search engines treat text to break it into matchable

terms so the engine can retrieve documents based on matches in large blocks of text.

The search engine's APIs enable flexible information retrieval by supporting Boolean combinations of indexed fields, allowing users to specify complex queries. These queries can match text exactly or match words within larger blocks of text (*free text*). They can also specify numeric ranges like date ranges, integer and floating-point ranges, and much more. Search indices and algorithms enable search engines to provide low-latency, high-throughput responses to API queries.

The common practice for building search-based applications is to use at least two systems—a durable, transactional system like a relational database to serve as the consistent, accurate system of record, and a search engine to search the data in the system of record. The application sends user queries to the search engine and then uses the system of record to retrieve the data for the search results. The application sends updates to the system of record. Backend systems capture and propagate these updates to the search engine. (See Figure 1-3.)

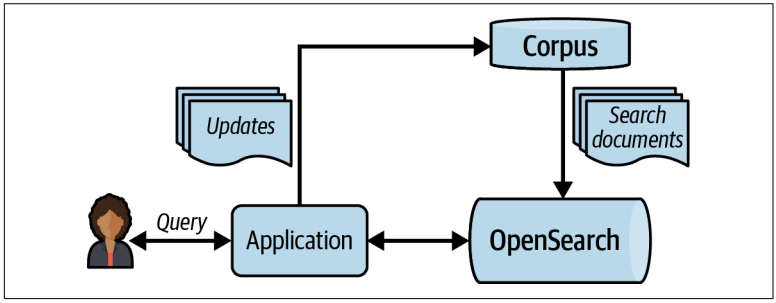

Figure 1-3. Search data flow: when you query, the application sends a request to OpenSearch, which holds search documents pulled from a corpus

When you visit *Google.com* or *Amazon.com* and type some words into the search bar, the website responds with a list of search results. Ideally, your desired product or website will be the first result at the top of the list. Search engines, unlike other databases, always sort their results and are optimized to return the most relevant results rather than *all* of the data that matches a particular query. *Relevance* is a measure of how useful a search result is in performing the user's intended task. For example, for ecommerce sites like Amazon, the

relevance of top results is closely related to whether the user buys the product.

Modern search is no longer purely lexical; search now employs machine learning (ML) and other strategies to make it easier to return relevant results (see also Figure 5-1). Behavioral tracking, query rewriting, and personalization feed an individual searcher's *user behavior signals*, like clicks and purchases, back to the search engine to improve that searcher's queries. For example, Learning to Rank is an open source plug-in that can use this data in open source search engines like OpenSearch. User behavior signals are also used to rewrite the query sent to the search engine. Search applications also use other signals and contextual information like the searcher's location, device type, feature engaged, etc.

Search ecosystems gather information about relationships between a user's queries and purchases to augment their queries with additional terms or boosts. They can also use ML data to personalize customers' search experiences: for example, speeding the customer's time-to-purchase or time-to-click by altering their queries and rankings based on brand affinity, or by relating that customer's segment to particular brands or categories of results.

Beyond Free-Text Search

Free-text search is used to search *unstructured information*—that is, blocks of text. A search engine can also search information with structure. Consider an ecommerce website: the products in its catalog carry brand, category, pricing, rating, and other information, stored either as fixed text or as numbers. A search engine can match text and numbers exactly, as it does with words for free-text search. Query offloading and curated datasets are two examples of more structured search workloads.

Query Offloading

Query offloading involves running queries on a replica of your production database that is hosted in a search engine. This lets you take advantage of search engines' high-throughput, low-latency query capabilities, sorting, and aggregations to perform database-like query processing without greatly increasing the load on the source database itself. Query offloading opens the door to extremely high scalability. Search engines can scale to handle 100,000 queries

per second or more while maintaining latencies that are in the milliseconds.

So why not make a search engine your primary data store? Because, to provide high scalability, high throughput, and low latency, search engines trade off on consistency, relational data structures, transactional behavior, and to some extent data durability. In most primary data stores, all of these characteristics are highly desirable or even necessary. Further, most search engines are distributed systems, deployed in a cluster and relying on intracluster communication during query processing to produce results. Large result sets (10 megabytes or larger) can paralyze the cluster with internal communication. In contrast to some other database systems, search engines are designed for retrieving small, sorted result sets—subsets of all of a query's results, rather than the full result set.

Curated Datasets

Raw data is the data in your organization that is not curated. It can include all types of structured, unstructured, and semistructured data, such as images, audio files, text, databases, PDFs, backups, archives, JSON files, and XML files. Some of these data sources are obvious, but others are less so, like recordings of meetings.

Curated datasets are organized and enriched datasets that often use a *data catalog* to list the data. You can send the metadata from your raw data to OpenSearch to provide the data catalog and enable search to help your internal users find content they need. You may even send the contents of your raw data, along with its metadata, to enable search in the catalog and contents.

Chatbots

The search process is evolving to include natural language interaction as a prominent way that people find information. Chat applications like Slack, WhatsApp, and WeChat provide ways for people to talk to one another in short-form messages. AI-based chat applications replace the person at the other end of the conversation with an LLM-backed text generator. You can read more about chatbots and RAG in Chapter 4.

Conclusion

In this chapter, we covered the broad sweep of how people search. Language is the central tool that people use to store and retrieve information. As search moves from a manual iterative process to an automated, natural language–driven process, builders are expanding their tool sets, based on advances in AI and ML for natural language, to support finding and acting on information. Even as the once-futuristic world of talking to our tools emerges, searching by language remains an important capability. In the next chapter, you'll learn how search engines work with language to retrieve relevant results.

CHAPTER 2
Lexical Search

It's important to understand how lexical search works, because it predates and anticipates how semantic search works. Lexical search for unstructured text works directly with language by decomposing blocks of text into words and matching those words to text from a query. You might think of it as the "assembly code" of searching.

Consider a website that advises its users on which games to play. The builders of the site use game descriptions as the corpus of documents for their search engine. They might include the following two blocks of text, which discuss the dice game craps and the card game blackjack, in their respective documents:

Text sample A (craps) from Wikipedia

> The shooter must shoot toward the farther back wall and is generally required to hit the farther back wall with both dice. Casinos may allow a few warnings before enforcing the dice to hit the back wall and are generally lenient if at least one die hits the back wall.

Text sample B (blackjack) from Wikipedia

> The dealer deals from their left ("first base") to their far right ("third base"). Each box gets an initial hand of two cards visible to the people playing on it. The dealer's hand gets its first card face-up and, in "hole card" games, immediately gets a second card face-down.

For the search `play dice games`, text sample A seems like a better match than text sample B. Most search engines solve this problem in three phases—matching, merging, and ranking.

Matching

Search engines analyze the text to break it into single words, then bring each word to a normalized form. OpenSearch, for example, contains 34 language-aware analyzers that refine terms to make them match in a more intuitive way:

Segmenting
: When the engine *segments* terms, it removes punctuation and lowercases terms, so that `Run.` matches `run`.

Stemming
: Language-specific *stemming* rules remove common inflections: this means `Run` matches `run`, `runs`, and `running`.

Stop-word filtering
: A *stop words* filter removes common terms like articles: `a`, `an`, `the`, and the like. These terms appear in almost every document, so they have low value for matching.

Synonym matching
: Finally, it adds *synonyms* so that terms can match across common groupings. For example, the engine might use `begin` and `initiate` as synonyms for `start`.

This report focuses on the English language analyzer. Searching across multiple languages is a broad topic touching index design, language analysis, and index, cluster, and document tenancy.

To OpenSearch, the transformed text samples look like this:

Text sample A (craps), analyzed
: shooter must shoot toward farther back wall gener requir hit farther back wall both dice casino mai allow few warn befor enforc dice hit back wall gener lenient least on die hit back wall

Text sample B (blackjack), analyzed
: dealer deal from left first base far right third base each box get initi hand two card visibl peopl plai dealer hand get it first card face up hole card game immedi get second card face down

The output looks odd. What's going on here? Each term that you see is an actual, matchable term from OpenSearch's English analyzer (segmentation). You can see that stop words are removed. We haven't applied synonyms, so there are no additional terms. Finally, the English analyzer employs a stemmer that transforms, for example, `generally` → `gener`, and `required` → `requir`.

Of course, textual queries undergo a similar analysis. The query `play dice games` is rendered `plai dice game`.

To match a query, search engines use an *inverted index*. An index in a book maps words to the numbers of pages where those words occur; to find something, you go to the index, look up a term, and go to the page number(s) indicated until you find what you are looking for. Similarly, an inverted index in a search engine maps terms onto the document IDs for documents that contain those terms. The search engine looks up the terms in the index, then combines the sets of document IDs (*posting lists*) to determine the match.

Figure 2-1 shows some of the terms that might be in our example corpus. On the left, you see the *terms index*; on the right are the *posting lists* for each entry. During matching, the engine looks up the term in the terms index. To match `play dice games`, it looks up the analyzed terms—`plai`, `dice`, and `game`—to get the posting lists [41, 42, 43], [3, 12, 15, 19, 38, 86], and [14, 33, 42, 75].

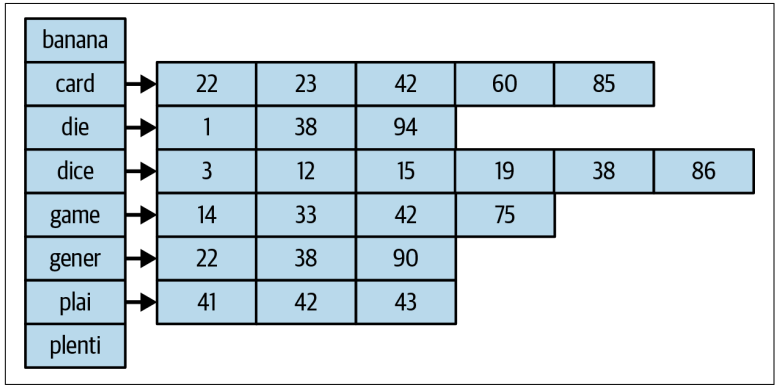

Figure 2-1. An inverted index, showing the terms index on the left and the posting lists on the right

Merging

The second phase of search is *merging* the posting lists to get a single set of matches. Search engines use set math to compute the match. If the application wants to match all terms in the query, it will use an `AND` operator for the query, and the engine will compute the intersection of the posting lists—in this case, `[ ]`, or an empty set. In most cases, the application will use an `OR` operator—a *union*—to retrieve `[3, 12, 14, 15, 19, 33, 38, 41, 42, 43, 75, 86]`. It relies on scoring and sorting to bring the most relevant documents to the top.

Ranking

The final phase of search is *ranking*. A search engine for lexical search employs a ranking algorithm called Term Frequency-Inverse Document Frequency (TF-IDF), which works based on the overall statistics of the text. Rare terms get high scores; common terms get low scores. A document's score for a particular query is the sum of the scores for the terms that match, multiplied by the number of times they occur. During the ranking phase, the engine sorts all matching documents by score to produce a final search result.

We'll imagine that `plai` and `game` are common for this document set, and that `dice` is rarer, with a higher value. The term `dice` occurs twice for craps and not at all for blackjack, so craps will receive a higher score and rank higher in the results than blackjack.

Conclusion

In this chapter you learned how search engines break up blocks of text and store them for matching in an efficient, language-aware way. Lexical search is about extracting as much meaning as possible from raw words to match and rank them by intent and meaning. But while this symbol-based text processing and retrieval goes a long way, it still doesn't directly get at the meaning of the symbols. For example, compare the meaning or sense of "a couch" and "a cozy place to curl up by the fire." The word "couch" doesn't match any of the words "a cozy place to curl up by the fire," but we feel the concepts match. To move beyond lexical search, ML had to supply tools, in the form of machine-learned vectors. That is where we turn our attention in the next chapter.

CHAPTER 3

Vectors: Representing Semantic Information

Language comes so naturally to humans that its complexity is hard to understand. We go from concept and meaning to spoken or written word (and back), mostly unconsciously. If computers were humans, they could easily communicate in natural language. AI researchers have studied symbolic natural language processing (NLP) in computers for decades with mixed results. The advent of modern machine learning and the age of big data has revolutionized NLP and brought a paradigm shift in our approach, enabling us to code language as high-dimensional vectors.[1] In this chapter, you will learn how ML systems train, employ, and create vectors to work with natural language.

Vector Basics

Computers and ML models only understand numbers. To work with the information contained in natural language, they need that information in number form. *Vectors* are that number form.

Vectors for semantic search (called *embeddings*) represent natural language as a set of values across many dimensions. When people train ML models for use in search engines, the goal is to produce a

1 There's a fascinating, long-running debate about the nature of consciousness and intelligence that spans the disciplines of philosophy, psychology, and computer science called *philosophy of mind*.

model that generates vectors that are close together for text that has similar meaning and far apart for text that has different meanings.

A vector (when centered at the origin) is a value for each of the axes in an *n*-dimensional space. Figure 3-1 shows the vector (4,6) in two dimensions—*X* and *Y*. You visualize this vector by drawing the line from the origin to the (*X*,*Y*) point. We call this a *two-dimensional* vector.

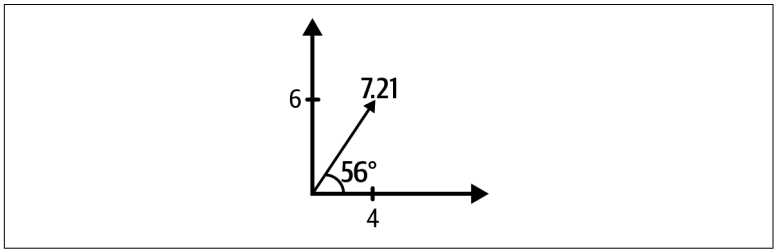

Figure 3-1. A two-dimensional vector

A 768-dimension vector has 768 values along orthogonal axes, just like the 2-dimensional vector in Figure 3-1 has 2 values along the two orthogonal *X* and *Y* axes. Of course, you can't easily draw or visualize 768 orthogonal dimensions!

To see how we encode language as vectors, let's start with the naïve stance that you could assign just one number to each word: `aard vark` = 1, `abacus` = 2, `apple` = 3, and so on. But, while 1 + 2 = 3, it doesn't make any sense to say that `aardvark` + `abacus` = `apple`.

Expanding from a single number, you could use two numbers, like the (latitude, longitude) coordinates used in GPS:

`aardvark` = (1,0)

`abacus` = (0,1)

`apple` = (0,2)

`banana` = (1,2)

Adding `aardvark` and `abacus` no longer results in `apple`.

Unfortunately, there's still a problem. Adding `aardvark` and `apple` gives (1,2)—which gives you `banana`. Adding more dimensions helps reduce nonsensical implications, but how many more dimensions is enough?

Dimensionality

One answer is to use as many dimensions as there are words in the vocabulary. Let `aardvark` = (1,0,0,0,...), `abacus` = (0,1,0,0,...), `apple` = (0,0,1,0,...), and so on. With this encoding, no two words can be added to give another word, since no word's representation can have more than one 1. This is often called *sparse*, *keyword*, or *one-hot encoding*, and the corresponding vectors are known as *sparse* or *one-hot vectors*.

Sparse vectors are quite useful in search and other natural language tasks, and until quite recently they were considered to be the state of the art. But sparse vectors can't combine in a meaningful way (no word has more than one "1"). Take the words `red` and `fruit`. Adding their vectors should not produce the vector corresponding to `aardvark` or `banana`. But it would make sense if it equaled the word `apple`, since an apple is indeed a red fruit.

To facilitate such natural language understanding and semantics, you need to use a vector with fewer dimensions than there are words. These vectors, known as *dense vectors*, are ubiquitous in modern ML models such as LLMs. Modern LLMs produce vectors with dimensions ranging from a few hundreds to a few thousands.

How LLMs Learn Vectors

ML uses an algorithm called *backpropagation* in neural networks to discover the vectors that encode words. Let's start with a simple example. A line in two dimensions has two parameters (the slope and the intercept). Given a set of data points in two dimensions, you can train a neural network (of two parameters) to draw a line that passes through the specified points. This is done by slowly and steadily changing the parameters (using backpropagation) until the points lie on the line. If the data is, say, in the shape of a circle and does not lie on a straight line, the neural network will never be able to find a line that passes through all of them. But in such a scenario, increasing the model's number of parameters would enable it to draw complex curves (and not just straight lines), and thus fit all the data points.

This is precisely why most LLMs have billions of parameters: natural language data is extremely complex. Their vectors live in very high-dimensional spaces and do not align in a straight line!

LLMs used for search are trained in a similar way, but they produce *n*-dimensional vectors (instead of curves or straight lines). LLM training via backpropagation uses a *loss function*, a function that measures how well the model fits the data points. For our example of learning a line, the loss function would be the distance from the point to the line. For language, the loss function depends on the task: for instance, the loss function for learning vectors that encode language for search is different than the loss function used for generating text.

For our use case, we want the models to learn good vectors that set us up for good search results. But what *are* "good" vectors and how do models learn them? Alternatively, what loss function should we use to teach the model "good" vectors?

You know that search LLMs convert every piece of text (word, sentence, or document) into a vector. A good model should produce vectors that satisfy the condition that vectors near each other correspond to similar texts, and vectors far away from each other correspond to dissimilar texts (Figure 3-2). The precise distance between two relevant vectors becomes the loss function. The model is trained to reduce the loss function—that is, the distance between the relevant vectors. It is similar for irrelevant vectors, except that the loss function now has a minus sign, so reducing the loss function is equivalent to increasing the distance. Typically, the distance between vectors is measured in terms of either the angle between them or the length of the straight line that connects them.

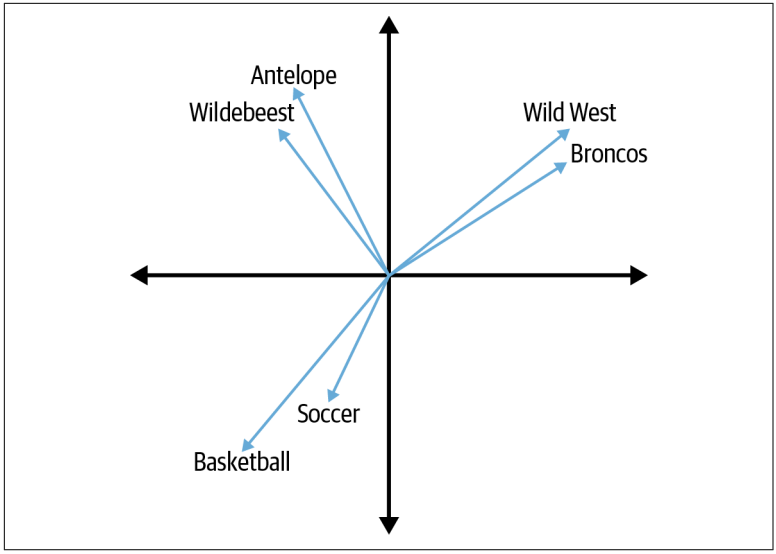

Figure 3-2. Good models produce vectors with small distances between related words and large distances between unrelated words

Steps 3 and 4 (see Figure 3-3) guarantee that vectors close to each other are related and vectors that are far apart are unrelated. Thus, if you encode your query with the same LLM and find neighboring vectors, the LLM will retrieve relevant results. Just finding the distances between these vectors can help us build better search engines.

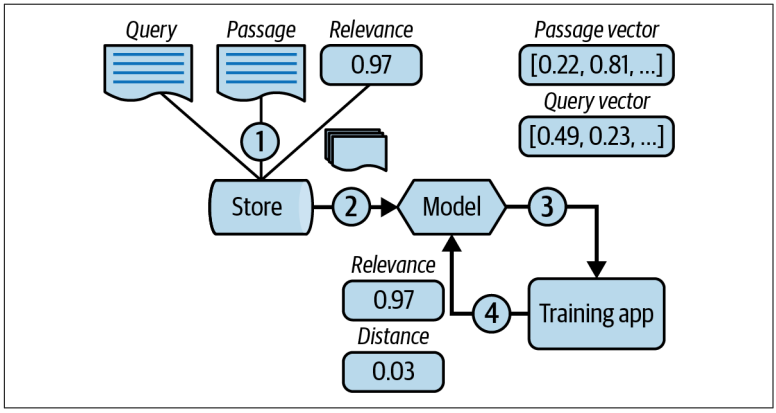

Figure 3-3. Training an LLM

If you start with a model with random initial parameters, the vectors it produces probably won't satisfy the condition that distance ≈ meaning ≈ relevance (that is, closer vectors are more relevant to each other). The loss function captures that requirement. You can use the loss function and the backpropagation algorithm to *train* an ML model, as shown in Figure 3-3. The training steps, in simplified form, are:

1. Collect a dataset of (query, passage) texts, labelled with a relevance value.
2. Feed the texts and relevance values to the model.
3. Use the ML model to create a query vector and a passage vector.
4. Train the ML model to change this distance (decrease the loss function) using backpropagation.

Figure 3-4 shows the vectors that an ML model created for several texts before and after its neural network training. Most public models are *pretrained* (trained on lots of data for a given task), so, often, all you need to do is download the model and run it.

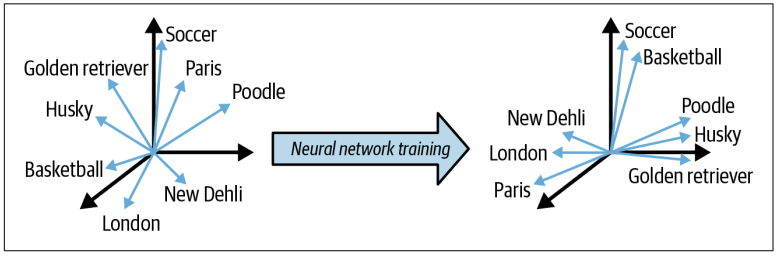

Figure 3-4. LLMs for search are trained with a loss function that produces good vectors

Types of LLMs

Most modern ML search applications use a particular kind of ML model, deployed in an architecture called a *transformer*. Transformers primarily come in two different types: encoder-only and decoder-only.

The *decoder-only* transformer is the architecture behind very large language models, such as OpenAI's GPT-3.5 and GPT-4, Anthropic's Claude, Google's Gemini, and Amazon's Titan. These models' input and output is most often in the form of text. That text is represented internally, always in the form of vectors, but these vectors aren't really used for anything directly.

Encoder-only transformers, such as Bidirectional Encoder Representations from Transformers (BERT), typically take input in the form of text (such as a word, sentence, document, or even an entire book) and output a vector that represents the input text. You can use this vector to do all sorts of tasks, including search.

Foundation Models

Foundation models are LLMs that are trained on a large amount of data (like a corpus consisting of all the text on the internet). Services like Amazon Bedrock provide these pretrained models to make it easier to get started. But because they are trained on such a broad corpus, they are not specific to a particular user domain (such as health care). To improve its vector embeddings and thus customize it to perform better for more specific use cases, you fine-tune the foundation model by training it with your specific data (see "Fine-Tuning: Beyond Pretrained Models" on page 21).

Multimodal Models

With the right preprocessing, you can train LLMs on diverse kinds of data, including images, video, and sound. Models such as Titan Multimodal and Gemini are trained on both text and image data. Titan can respond to text prompts with image-based embeddings, while Gemini can generate images or text from its prompts. Models like OpenAI's Sora can generate video from text prompts.

Multilingual Models

Some models work with more than one language. Multilingual LLMs such as GPT-4 or Claude provide capabilities for generating text across languages. Cohere's Embed v3 can create vector embeddings for multilingual text.

The Limitations of Dense Vector Search

So far, we have discussed the strengths of dense vector search and how it can be used to encode semantic similarity. But lexical similarity is important too!

When you have a document corpus that uses very specialized and specific words, like a corpus of product documents that contain lots of alphanumeric strings, you need to be able to match those strings exactly. If a user searches for a product by its serial number, `AX7D`, they want to retrieve documents that match `AX7D` *exactly*. Because the query has no semantic information, a semantic embedding would actually harm the results. A dense vector encoding of `AX7D` will lie close to possibly unrelated items in the vector space. Semantic search will retrieve some of these unrelated neighbors, but they won't match the searcher's intent, which was to retrieve exactly and only the product with serial number `AX7D`.

Hybrid Search

The same is true when the query is a highly specific word in a specialized domain, like `ADHD` in psychology, or `OB/GYN` in medicine. In such cases, a simple keyword search will yield better results. Since a typical document corpus contains a lot of natural language but also domain-specific, highly specialized words, the right search solution would need to *combine* dense vector search with keyword search: a *hybrid search*.

Recall that dense vector search retrieves the top k documents along with their scores, and keyword search scored by TF/IDF retrieves the top k documents with their respective keyword scores. To get the final result, the engine combines these scores. It can do this using a few strategies. One strategy is to interleave the lexical and semantic results, where the final ranked list—comprising, for instance, 10 items—is obtained by alternating between the top five results of each individual retriever. Another strategy, *score normalization*, normalizes the TF/IDF scores and dense vector scores to lie between 0 and 1 before combining them. OpenSearch has conducted extensive research to benchmark the performance of different normalization and combination techniques.

OpenSearch also supports efficient filtering for vector search that can reduce the universe of possible matches by lexically filtering by

an attribute value, like brand or product category. This elicits more relevant results and solves cases where your customers are searching for `AX7D`.

Sparse Vectors and SPLADE

Earlier in this chapter, we introduced sparse vectors—vectors with the same number of dimensions as there are terms, encoding each term as a single "1" value in a single dimension. We mentioned a shortcoming of sparse vectors, that they don't generalize so that, for example, "red" + "fruit" = "apple." Sparse Lexical and Expansion Model (SPLADE) produces a *sparse vector* but uses a training objective to make sure that *some* of the entries in the vector are larger than zero. Allowing some nonzero values keeps relevant vectors close to each other and preserves generalization to closely related concepts. At the same time, the nearly-one-hot encoding maintains a tighter relationship between the source term and the vector than a dense vector does. Using sparse vectors, you get the benefit of term-based matching with the generalization of dense vector matching.

Fine-Tuning: Beyond Pretrained Models

So far, we have described ML models that you can download from a public repository. These models are trained on lots of data and often perform decently on general-purpose questions. But if your queries rely on a corpus and the model has not been trained on similar data, it will not perform well. Fine-tuning allows the model to specialize for particular tasks, improving its performance on those tasks.

In the fine-tuning phase, you start with a pretrained model and collect (`query, document`) pairs from the target domain. For example, if you're fine-tuning for medicine, you might use a training pair like the one in Figure 3-5.

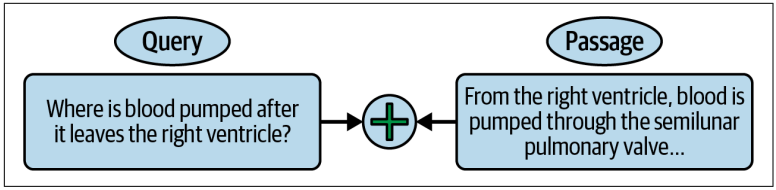

Figure 3-5. A healthcare question-and-answer pair with a positive correlation

You would ask the model to map the corresponding vectors close together. Similarly, you would also take irrelevant pairs, like the one in Figure 3-6, and ask the model to map their corresponding vectors far away from each other (for example, revisit Figure 3-3). This process forces the model to readjust its parameters based on new data.

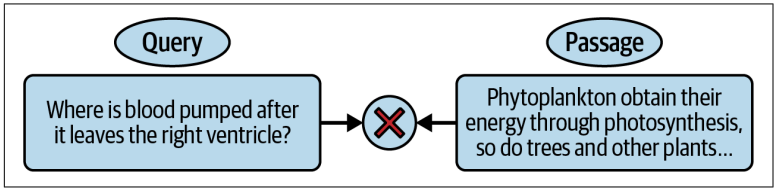

Figure 3-6. A healthcare question-and-answer pair with a negative correlation

Conclusion

In this chapter, you learned about the foundation of how search engines retrieve semantically relevant results: vectors. Vectors also underlie the LLMs that power chatbots, AI assistants, and AI agents. In the next chapter, you'll learn how to use vectors for semantic search.

CHAPTER 4
Semantic Search

As you've seen, lexical (or keyword) search looks at the terms of a query and matches those terms to terms in its index. Lexical search is powerful in its own right, especially when users have a good idea of exactly what they want to find. If you know that you want to buy `Zinus Ricardo Sofa Couch with tufted cushions in Lyon blue`, a site like *Amazon.com* can easily retrieve that couch for you based on the lexical match between that query and the product's title and description.

But what if you've never heard of that particular couch? Maybe you just know that your décor favors cool colors, you have a fireplace, and you want to spend some quiet nights by that fireplace reading books. You might search Amazon for `a cozy place to curl up by the fire`, but the results will be disappointing, since none of the terms `cozy`, `curl up`, `place`, and `fire` appear in the title or description for the Zinus couch.

While the terms are not there, what the couch offers and what you mean when you type `a cozy place to curl up by the fire` match closely. Semantically, couches are related to coziness, fireplaces are related to coziness, and couches are related to curling up. The *words* don't match, but their *meanings* do.

Vectors generated by LLMs encode semantic information from blocks of text. In semantic search, the search application uses an LLM to create a vector for each document in the corpus (Figure 4-1). When a user runs a search, the application encodes the text of that search query with the same LLM to produce a vector

in the same space. It then uses a distance function to produce a score for the query/document pairs and ranks the results by that score. In some cases, it uses efficient filtering, hybrid search, or score normalization, in addition to the distance score. Figures 4-1 and 4-2 show this process.

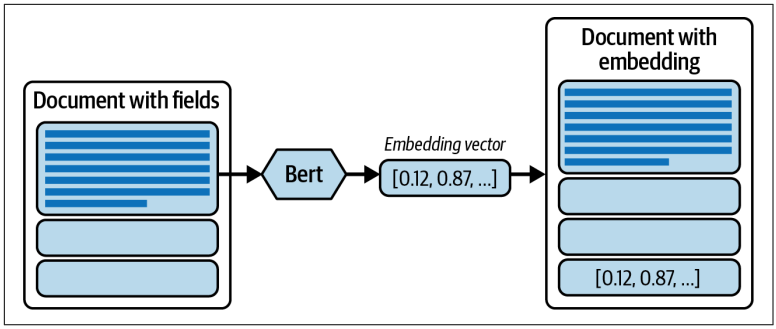

Figure 4-1. Using a BERT model to encode a document's fields

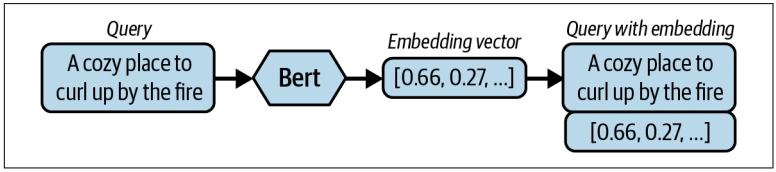

Figure 4-2. Using a BERT model to encode a user's query

You can see the difference in Figure 4-3, a screenshot from the OpenSearch Playground. In this example, the query is `sailboat shoes`. On the left are the results from a lexical search. On the right are results from a semantic search by OpenSearch's Neural plug-in. The lexical matches contain *one* pair of boat shoes, but they also contain many items that simply mention `sailboat` or `shoes` in their descriptions. The semantic matches, by contrast, are for shoes that a person might actually use on a sailboat.

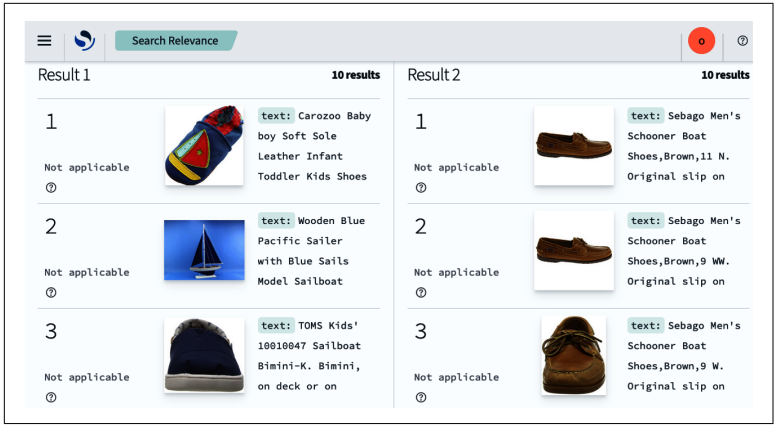

Figure 4-3. Comparing a BM25-ranked set of results (left) and a kNN-ranked set of results (right) for the query "sailboat shoes" side by side

Exact and Approximate Nearest-Neighbor Search

Semantic searching by nearest neighbors gives high-fidelity results, especially when augmented by hybrid search methods. In *exact k-Nearest Neighbors* (kNN), the brute-force method of semantic search, the engine tries *every* combination of query/document vectors and computes their distances. This is fine when the corpus is small, but it becomes computationally expensive and leads to high latency as the number of documents grows into the millions.

Approximate kNN (or ANN) solves this by reducing the number of vectors the engine compares. There are two main algorithms for ANN: Hierarchical Navigable Small World (HNSW) and Inverted File (IVF). During indexing HNSW creates layers of graphs at different granularities, in which nodes are vectors and edges lead to other neighborhoods of close vectors as the layers get deeper. During a search, the HNSW algorithm walks the graph at the coarsest layer until it finds a match close enough to enable it to go to the next layer. It deepens progressively until it reaches the bottom layer, where it collects highly local neighbors.

The IVF algorithm uses clustering techniques during indexing to create clusters of related points. It then builds a small table of identifiers for the clusters, with a representative vector for each. During search, it finds the nearest cluster of vectors and then scores against that small cluster. A method called *product quantization* can further reduce the storage needed by downsampling the vectors within the vector cluster.

Approximate nearest-neighbor algorithms (HNSW and IVF) trade accuracy for faster retrieval (and, in the case of IVF, storage). HNSW gives better accuracy than IVF, but its storage needs can be expensive. Be sure to evaluate search results carefully so that you can make the right trade-offs.

Semantic Search for Retrieval-Augmented Generation

When you prompt a chatbot, the LLM for that chatbot, in anthropomorphic terms, "makes up" a response based on the structure and content of its training data. It can make up perfectly right-sounding answers that are wrong—*hallucinations*. Here is an example of a hallucination generated by ChatGPT 3.5 in response to the prompt "Does the Turing test mention gender":

> No, the original formulation of the Turing test, as proposed by Alan Turing in his 1950 paper "Computing Machinery and Intelligence," does not explicitly mention or involve gender as a specific criterion.

However, "Computing Machinery and Intelligence" does indeed *mention* gender: "The object of the game for the interrogator is to determine which of the other two is the man and which is the woman… Will the interrogator decide wrongly as often when the game is played like this as he does when the game is played between a man and a woman?"[1]

Prompt engineering is the process of structuring a prompt to elicit better text from the LLM. Prompt engineering helps reduce hallucinations by adding additional context that helps constrain the LLM's output. Prompts can contain declaratory information ("you are a

1 Alan M. Turing, "Computing Machinery and Intelligence," *Mind* 49 (1950): 433–60. This is perhaps a subtle point. However, "No" is not the correct answer.

student in middle school"), procedural information ("to take the average of 12 and 24, add the numbers and divide by 2"), and other factual information ("the average of 34 and 48 is 41").

RAG is a technique that feeds additional data to the LLM as part of the prompt to help avoid hallucinations. In the previous example, the application might search for "does the Turing test mention gender" and retrieve Turing's "Computing Machinery and Intelligence." It can then add the text of the article to the prompt to influence the generated text, removing the hallucination.

Chat applications use many different data stores for RAG, including relational databases, data lakes, and, of course, a search engine. These data stores serve as a *knowledge base*—a pool of information from which the LLM can retrieve relevant results for the user's query. The application augments the LLM prompt with accurate data from these sources to drive better results.

For example, the Amazon OpenSearch Service builders might want to provide a chatbot that can help its users with questions about the service. As step 0 in Figure 4-4 shows, they could index relevant information from the documentation, the service's website, code from OpenSearch's GitHub repositories, and any other relevant sources. In step 1, users send questions to a chatbot. The bot requests (step 2) and retrieves (step 3) information from OpenSearch. The bot then sends an *augmented prompt* to an LLM (step 4) and returns its response to the user.

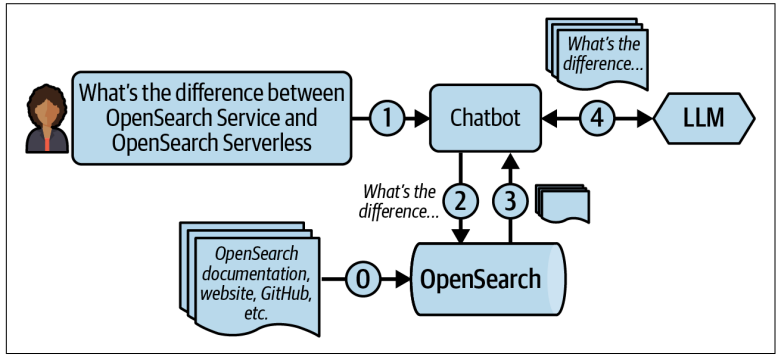

Figure 4-4. RAG data flow

Conclusion

Semantic search uses LLMs to capture the meaning of text and the meaning of a user's query text to perform vector matching and find documents whose meaning is close to the query. In the next chapter, you'll learn how to build with OpenSearch to put semantic search to work.

CHAPTER 5
Building with Search

In this chapter, we'll outline how you can put these concepts into practice. For the search technology, we'll discuss OpenSearch and Amazon OpenSearch Service, but you could similarly employ any commercial or open source solution.

OpenSearch, like most search engines, is a distributed system, deployed on clusters of instances in different roles. When you use it, you run the OpenSearch process on these nodes, and OpenSearch discovers the other nodes to form a cluster. OpenSearch provides automatic data replication for better durability and to parallelize the workload across compute, memory, and storage resources. You can run OpenSearch on your own or use a hosted version, such as Amazon OpenSearch Service. Amazon OpenSearch Service takes care of undifferentiated tasks like deploying hardware, installing and upgrading OpenSearch software, and monitoring for and repairing defects in your cluster.

Amazon OpenSearch Service has included a vector engine since 2021, when the service made a kNN plug-in available that provides the data structures and algorithms to support storing and searching vectors. OpenSearch's Neural plug-in simplifies the process of generating and attaching vector embeddings to documents and creating the embeddings for user queries.

ML-Powered Search

As an example, consider an ecommerce website, built to bring products to customers who want to buy those products. Search engines have played a primary role in retrieving relevant products, and the architecture of search applications reflects this past.

Figure 5-1 shows some of the components and data flows that support an ecommerce application. The band of systems at the top processes and prepares the catalog of products. In the center band, you can see the query-processing chain. Along the bottom is where user behavior is captured and processed. The feedback loop from user behavior to catalog enrichment is a key part of continuously improving the application's ability to retrieve products that customers want to buy.

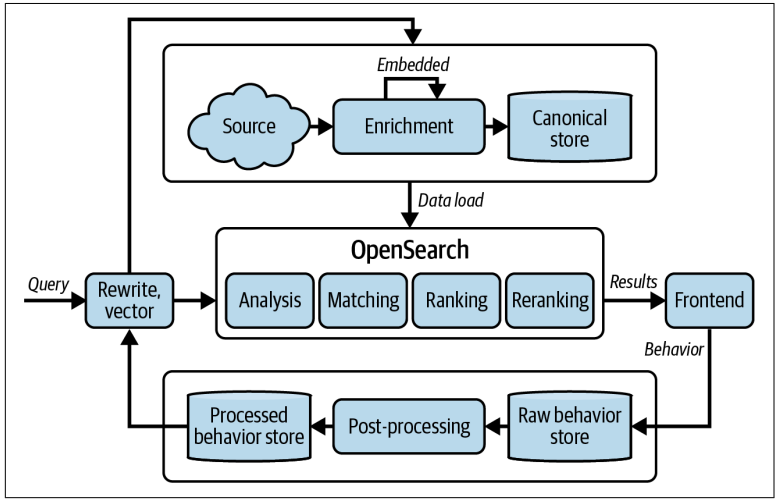

Figure 5-1. Search applications employ many components with varied purposes to bring relevant products to customers for purchase

As we've highlighted throughout, this type of application can be augmented with GPTs that summarize results or converse with customers. With RAG, the customer from our Chapter 4 example could chat with the search application about a cozy place to curl up by the fire and find couches, recliners, and the like. If the model powering the application is a good one, its embeddings will capture the semantics of concepts like cozy, comfortable, sit, and warm. OpenSearch will retrieve products that match these concepts and

feed them forward to the language model, which will summarize them and generate text.[1]

Apart from RAG-driven chat, the application can look at a customer's previous purchases to improve OpenSearch's ranking and output, using the behavioral loop shown in Figure 5-1. The Learning to Rank plug-in for OpenSearch lets you use ML on captured user behavior to rerank OpenSearch's results. The application can also employ customization, adding known user preferences and past user behavior as weights and modifications to its queries.

Setting Up Your Model

When you bring LLMs into your application to support semantic search, you must decide which ML model you will use to create vectors for your documents and your queries. You can find a collection of models in the Hugging Face Sentence Transformers repository or choose one from the list in the OpenSearch ml-commons library's supported models. If you're using Amazon OpenSearch Service, you can take advantage of our prebuilt integrations to connect with third-party model-hosting services like Amazon SageMaker, and Amazon Bedrock. When choosing a model, some factors to consider include:

Model size
: Larger models with higher vector dimensionality generally produce better results. Some newer, smaller models are almost matching GPT's results, so stay tuned.

Training data
: Models are more accurate when their test and training data distributions are similar. For instance, if a model was trained for search on an ecommerce corpus, it will not yield the best search results when used on a corpus of scientific papers.

Latency/accuracy trade-offs
: Larger models lead to better results, but they require more computation, which leads to higher latencies and expenses.

1 At the time of writing, the OpenSearch project has an experimental feature supporting conversational search that encompasses much of the RAG processing needed to generate conversational results.

Modality

In this report we have been primarily concerned with models trained on text. Some multimodal models provide better accuracy through training with video, image, or even sound data.

Once you have decided on your model, you can deploy it with the ml-commons APIs or use Amazon OpenSearch Service's one-click connector interface (Figure 5-2) to connect to a third-party model host.

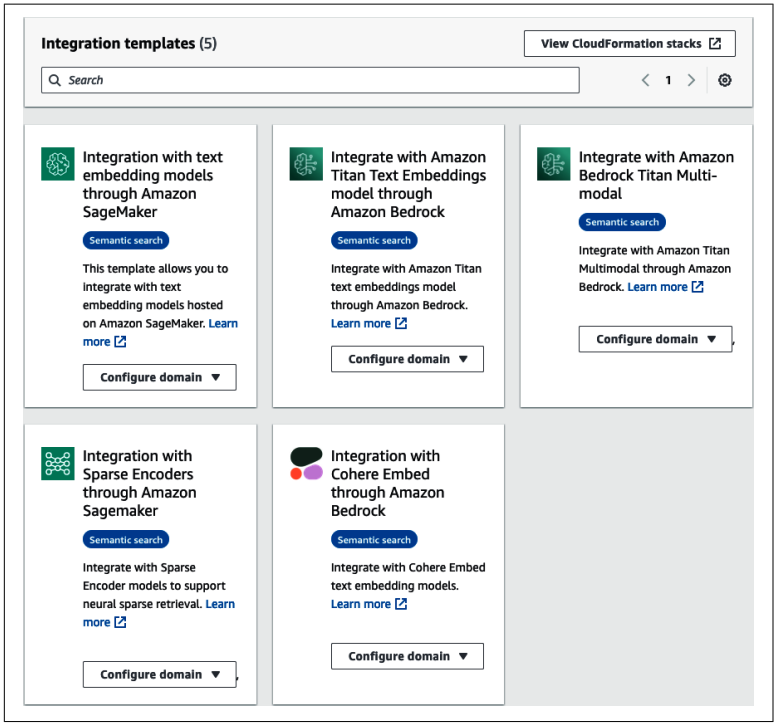

Figure 5-2. As of this writing, you can use Amazon OpenSearch Service's console on the Amazon Web Services (AWS) console to create connectors to third-party model hosts. The console experience employs AWS CloudFormation templates to deploy infrastructure that supports connecting to these systems.

The integration templates create a connector and, within OpenSearch, a *model*. The OpenSearch model is a placeholder for the Neural plug-in to connect to the hosting service. You use its ID to refer to the connector.

Finally, you need to create an index that is ready to use OpenSearch's kNN plug-in. All indices in OpenSearch have an associated schema called a *mapping*. You specify index settings, like `knn`, as well as field definitions that control how those fields are analyzed.

Ingestion

Once you have created an index, it's time to ingest documents and turn them into vectors. For this, you'll use an OpenSearch *ingest pipeline*—an OpenSearch construct that lets you mutate the documents during ingestion—and the Neural plug-in to convert natural language prompts to embeddings. An ingest processor is an OpenSearch construct that defines a transform to apply to documents as they go through the ingest pipeline. When you define an ingest processor that uses the Neural plug-in, you specify which fields to use as source text and in which fields to store the embeddings. The Neural plug-in calls your model to create the embeddings (Figure 5-3).

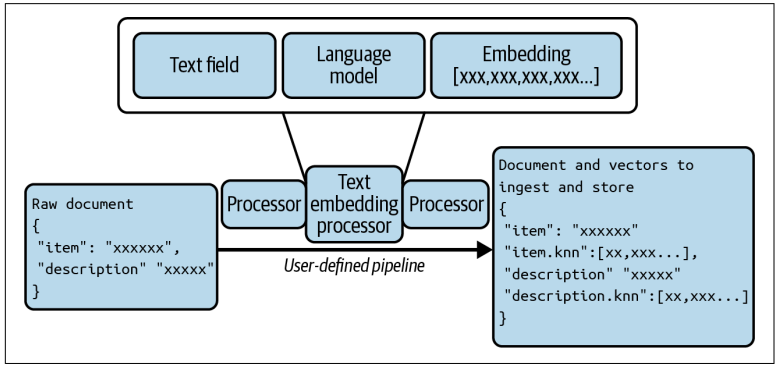

Figure 5-3. Ingesting and embedding item and description fields from a product catalog through a user-defined pipeline in OpenSearch

Most search models can only read the first 512 tokens of a document, so you may need to chunk the documents into several pieces and create a vector for each chunk before sending them to OpenSearch.

Retrieval

Now you're ready to search! At runtime, the Neural plug-in converts every incoming query into a vector and uses the kNN plug-in to find its nearest neighbors in the high-dimensional space and retrieve relevant results.

Conclusion

This chapter illustrated how to implement semantic search with an example for an ecommerce site. In the next chapter, you'll learn how enterprises have used this strategy to deploy effective semantic search applications.

CHAPTER 6

Deploying a Winning Search Strategy

This chapter discusses important considerations for using ML to drive your search strategy. We'll begin by examining four real organizations that have successfully rethought their search strategies to include ML-driven search: Walmart, CoStar, Syte, and Novartis. We'll discuss critical factors such as scalability, continuous improvements and updates, and governance. We'll finish by discussing how to make sure your search engine uses responsible AI techniques, including basic safety and security considerations for AI models.

Success Stories

These success stories explain how four organizations have used highly scalable ML-driven search engine strategies to boost their customers' experiences. Walmart is augmenting its staff with ML-driven search in its stores and online. CoStar is now achieving 30-millisecond search results with a modern data plane for millions of concurrent users. Syte is increasing retail conversions with visual discovery, and Novartis is improving search recommendations by applying groundbreaking NLP and ML techniques. These stories speak to the breadth and depth of how people are building ML-driven solutions.

Walmart: AI Assistants

The world's largest private employer, Walmart, is augmenting its staff with ML-driven search that assists customers online and in physical stores. These AI knowledge workers relieve staff from many tedious tasks related to stocking and finding items, leading to higher staffing levels and lower turnover.

Walmart's in-store search strategy starts with shelving products as they are unloaded from delivery trucks. An AI assistant helps staff members scan the boxes and instructs them on exactly where to place each product, based on customer behavior and sales data.

At night, while floor-scrubbing robots rove Walmart stores around the world, they also scan and photograph every shelf and product, calculating inventory (even on messy shelves). The robots feed that data back into the system to inform restocking and product placement. The AI inventory scanners even calculate the likelihood that a product has been pushed to the back of a shelf, behind other products.

CoStar: Database Offloading

CoStar is a global provider of commercial real estate information and news. It needed to scale its site up to support hundreds of millions of real estate listings, millions of daily users, and thousands of commercial real estate companies and brokers—all while keeping search results fast and accurate. CoStar's existing search solution was slowing it down and hitting the limits of scaling its solution.

CoStar uses Amazon OpenSearch Service to provide real estate listing search across multiple fields like the number of bedrooms, the number of bathrooms, the square footage, or the price. Amazon OpenSearch Service gave CoStar's customers a richer set of search results and brought CoStar some impressive benefits: a 30-millisecond average time to return search results, a 4-second average time to update listings, the capacity to support 50 million concurrent users, a 75% reduction in cost of their solution, and a fully modernized architecture that can continue to grow with the business.

Syte: Multimodal Search

Syte, a search provider for ecommerce websites, needed to optimize its *visual discovery* feature, which allows end users to upload an image (such as a photo of an item of clothing they like) as a search input instead of typing in a search phrase. The catalog then matches that photo with available items.

Syte also uses *product tagging*, which makes products more searchable by associating related words and phrases with catalog pages. The company wanted to automate this feature with AI. To build this system, Syte implemented ML-driven search using Amazon OpenSearch Service and Amazon SageMaker, a cloud service that allows AWS customers to build and deploy highly scalable ML and NLP models. This resulted in an average 200% increase in traffic and a 42% reduction in cost per transaction for the ecommerce websites. As an added benefit, Syte improved its system's scalability without needing to hire new people.

Novartis: Semantic Search

Novartis, a global healthcare company and pharmaceutical manufacturer, has a product catalog containing more than two million items. Its challenge was to make search results faster and more relevant.

The toughest issues to get past were related to accounting for price, properties, and vendors in the search results. Free text input relied heavily on keyword matching, without context, and the discovery journey stopped after the first search results were presented to the customer.

Novartis deployed semantic search within Amazon OpenSearch Service, using OpenSearch's vector engine to support rich search and recommendations across items within its catalog.

To make your own strategy as successful as these, the remainder of this report looks at some additional considerations for creating ML-driven search models.

Digital Assistants and Agents

Along with such interactive advances as two-way chatbot conversations, LLM-based workflow applications go much deeper than just finding and presenting the best results. They can actually perform tasks much like a human assistant would.

For example, a customer might type, "Please order packing peanuts for 100 of our large boxes, get it to me by next Wednesday, and don't spend over $25." The application can interact with the LLM and the customer to fill this request. Now, instead of spending time doing these tasks, the customer is free to do something else and can just approve each of the actions in the steps the LLM generates.

If the customer ordered packing peanuts in the traditional way (pictured in Figure 6-1), they would need to:

- Search for items to purchase.
- Sign in to their own order system.
- Create and complete a purchase requisition.
- Get sign-offs on the requisition, tracking it all the way.
- Create a purchase order (PO) from the requisition (or request this from someone else).
- Send the PO to the vendor to make the purchase.

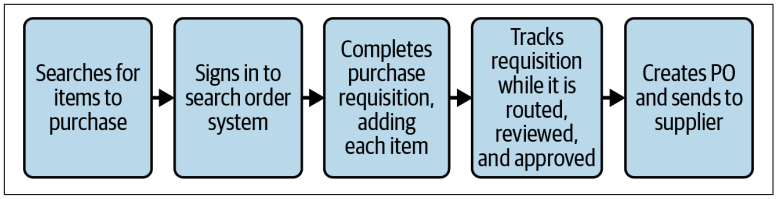

Figure 6-1. How a customer completes a traditional purchase. All work steps are manual and time-consuming.

Contrast that with what you see in Figure 6-2: an automated work process, fully driven by a chat interface. The customer issues a prompt stating what they want to buy, how much they can spend, and when they need the item. So long as the proper APIs and API credentials are in place, the workflow can be fully automated.

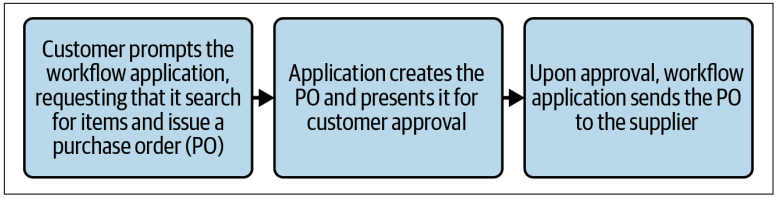

Figure 6-2. How work is automatically performed and presented for review from a chatbot interface. All steps are quick for the customer because the LLM-powered application does most of the work, interfacing directly with the APIs of the systems it needs to use.

Ethics and Responsible AI

Ethics and responsibility in AI are hot topics, and for good reason. They're very important in making sure your model functions as you want it to over the long term. This section provides a very brief overview of some of the issues in building with and using LLMs as signposts toward implementing responsible AI that meets the standards of your stakeholders.

Bias and Data Leakage

The internet encompasses a wide range of viewpoints on almost every topic. Foundation models trained on general internet content benefit from the vastness of this amount of text, but they also pick up biases that reflect predominant viewpoints. For some topics, the darker, seamier sides of the internet contain viewpoints that are biased and objectionable, and the models' results can reflect biases like racism and sexism in some pretty alarming ways. There is a growing body of research on removing bias in models trained on general internet content so that the text they generate is inclusive and promotes a positive and diverse community.

Data leakage is another problem. LLMs generate text based on their training data. When you train or fine-tune your model with your own data, that data remains coded in the model. If you train with confidential data, your model might generate text that includes this information. This could include personally identifying information (PII) like an individual's medical, financial, or demographic data, or proprietary data from your enterprise, like secret methods or processes, organizational information, and salaries.

Partly for these reasons, you need to think about authentication and access control. If any of your systems are powered by LLMs trained on such information, only people authorized to access confidential information should have access to those systems. Of course, you can intervene earlier in the process and remove confidential information from the training data.

Access control is an expanding area, with challenges driven by the opacity of LLMs, and possible solutions involving semantic access control—is this user authorized to ask this question, based on the semantics of the question? Is this user authorized to receive this answer, based on the semantics of the answer? As of this writing in 2024, search engines only provide access controls that operate on an entity-data relationship.

Traceability, Provenance, and the "Opaque Box"

In Chapter 3 we described how LLMs encode the correlations between words and concepts in their inputs with billions of parameters. You can't point at a single parameter in the model and say, "This one is responsible for the output." As of early 2024, there's no process that can map parameters to rules that people can understand. This makes every LLM an *opaque box*: it performs a task, but with no visibility into why or how. At present, what you *can* do is to provide traceability and provenance for the content you use to train or fine-tune your models, and for the output of those models. You can break down your AI's supply chain and add ways to establish origin and provenance of the data. Using a visible layer of proof that shows the intent, origin, and components of the model, as shown in Figure 6-3, will help make your AI supply chain more transparent.

Showing authenticity through user-viewable content *provenance*, or a step-by-step history of that piece of content, is important in establishing customer trust. One solution is to display a simple, easy-to-read *trust logo*: an insignia indicating that the content meets the standards of a reputable body, such as the Content Authenticity Initiative, a community of technology organizations led by Adobe that is focused on content provenance. Trust logos can be served by the frontend of AI-generated content, allowing users to find provenance at the level that gives them comfort. Being able to prove provenance is also a key factor in defending legal risk issues involving AI.

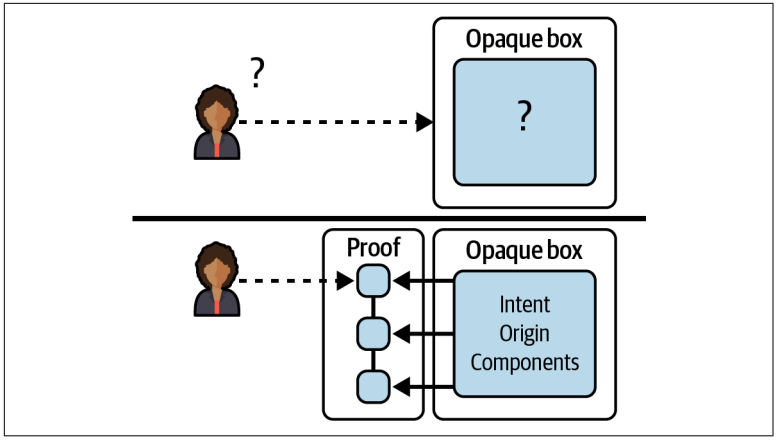

Figure 6-3. Top: customer wondering why they should trust the results of the model. Bottom: the same customer resolving their concern by checking for proof of the intent, origin, and components that comprise the model.

Conclusion

In this report, you've learned how search is built to support both lexical and semantic querying of information. You've learned the core structures and algorithms for lexical search, the initial basis of search engines. You've learned to understand vectors, how vectors play a central role in enabling search with natural language, and how that can extend to other modalities. You've learned how to implement semantic search, with vector embeddings and approximate nearest neighbor search algorithms. Finally, you read about some of the companies that are employing semantic search to great success.

The process of searching begins with a goal and ends with the means to satisfy that goal. Historically, searching for information has been very different when talking with a person and when using technology. The latest advances in AI and ML have narrowed the gap, making it possible for people to use language to interact with technology.

About the Authors

Jon Handler is a Senior Principal Solutions Architect at Amazon Web Services based in Palo Alto, CA. Jon works closely with OpenSearch and Amazon OpenSearch Service, providing help and guidance to a broad range of customers who have search and log analytics workloads that they want to move to the AWS Cloud. Prior to joining AWS, Jon's career as a software developer included four years of coding a large-scale, ecommerce search engine. Jon holds a Bachelor of Arts from the University of Pennsylvania, and a Master of Science and a Ph.D. in computer science and artificial intelligence from Northwestern University.

Milind Shyani is an applied scientist at Amazon Web Services working on large language models, information retrieval and machine learning algorithms. He is a theoretical physicist by training and received his Ph.D. from Stanford University.

Karen Kilroy is a life-long technologist with heart, as well as a full-stack software engineer, speaker, and author living in Northwest Arkansas. As CEO of Kilroy Blockchain, Karen has invented several products, including FLO, CASEY, Kilroy Blockchain PaaS, CARNAK, and RILEY, an AI mobile app that won the IBM Watson Build award for North America in 2017. Karen was selected as a recipients of a National Science Foundation research grant in 2018 and focused their studies on autonomous vehicles. Karen was also selected as a 2022 recipient of the Life Works Here award by the Northwest Arkansas council. Karen is studying agriculture and music at the University of Arkansas, is a four-time IBM Champion, and is a professional dragon boat coach.