13 Semantic search with dense vectors

This chapter covers

- Semantic search using embeddings from LLMs
- An introduction to Transformers, and their effect on text representation and retrieval
- Building autocomplete using Transformer models
- Using ANN search and vector quantization to speed up dense vector retrieval
- Semantic search with bi-encoders and cross-encoders

In this chapter, we'll start our journey into dense vector search, where the hyper-contextual vectors generated by large language models (LLMs) drive significant improvements to the interpretation of queries, documents, and search results. Generative LLMs (like ChatGPT by OpenAI and many other commercial and open source alternatives) are also able to use these vectors to generate new content, including query expansion, search training data, and summarization of search results, as we'll explore further in the coming chapters.

The state of the art in LLMs is changing on a monthly (and sometimes almost daily) basis, but even the best general-purpose models can currently be outperformed on specific tasks by fine-tuning other smaller models for those tasks. Over the next few chapters, we'll discuss the concepts behind LLMs and how to best use them in your search application. We'll introduce Transformers in this chapter and discuss how to use them for semantic search with dense vectors. We'll cover fine-tuning an LLM for question answering in chapter 14 and leveraging LLMs and other foundation models for generative search in chapter 15.

Our story begins with what you learned in section 2.5: that we can represent context as numerical vectors, and we can compare these vectors to see which ones are closer using a similarity metric. In chapter 2, we demonstrated the concept of searching on dense vectors, a technique known as *dense vector search*, but our examples were simple and contrived (searching on made-up food attributes). In this chapter, we'll pose the questions "How can we convert real-world unstructured text into a high dimensional dense vector space that attempts to model the actual meaning of the text representation?" and "How can we use this representation of knowledge for advanced search applications?".

13.1 Representation of meaning through embeddings

We're going to use language translation as an example to understand what we mean by "dense vector" embeddings. Take the following two sentences: "Hello to you!" (English) and "Barev Dzes" (Armenian). These two expressions hold approximately the same meaning: each is a greeting, with some implied sense of formality.

Computationally, to successfully respond to the greeting of "Hello to you!" the machine must both comprehend the meaning of the prompt and also comprehend all the possible ideal responses, in the same vector space. When the answer is decided upon, the machine must then express it to a person by generating the label from the answer's vector representation.

This vector representation of meaning is called an *embedding*. Embeddings are used interchangeably between natural language processing (NLP) tasks and can be further molded to meet specific use cases. We generated embeddings from an LLM (`all -mpnet-base-v2`) in chapter 9, but we glossed over most of the details on how embeddings work. In this chapter, we'll introduce techniques and tools for getting embeddings from text, and we'll use them to significantly enhance query and document interpretation within our search engine.

NATURAL LANGUAGE PROCESSING

NLP is the set of techniques and tools that converts unstructured text into machine-actionable data. The field of NLP is quite large and includes many research areas and types of problems (NLP tasks) to be solved. A comprehensive list of the problem areas is maintained on the NLP-Progress site (**https://nlpprogress.com**).

We will be focusing specifically on applying NLP to information retrieval, an important requirement of AI-powered search.

One important point is worth noting up front: behind the two short English and Armenian greetings we mentioned, there are deep cultural nuances. Each of them carries a rich history, and learning them thus carries the context of those histories. This was the case with the semantic knowledge graphs we explored in chapter 5, but those only used the context of documents within the search engine as their model. Transformers are usually trained on much larger corpuses of text, bringing in significantly more of this nuanced context from external sources.

We can use the human brain as an analogy for how Transformer models learn to represent meaning. How did you, as a baby, child, teen, and beyond, learn the meaning of words? You were told, and you consumed knowledge and its representation. People who taught you already had this knowledge and the power to express it. Aside from someone pointing out a cat and saying "kitty" to you, you also watched movies and videos and then moved on to literature and instructional material. You read books, blogs, periodicals, and letters. Through all of your experiences, you incorporated this knowledge into your brain, creating a dense representation of concepts and how they relate to one another, allowing you to reason about them.

Can we impart to machines the same content from which we obtained this power of language, and then expect them to understand and respond sensibly when queried? Hold on to your hats!

13.2 Search using dense vectors

Understanding when to use dense vectors for search instead of sparse vectors requires understanding how to process and relate text. This section briefly reviews how sparse vector search works in comparison with dense vector search. We will also introduce *nearest-neigh-*

bor search as one type of similarity used for dense vector search, as compared to BM25 (the most common similarity function used for sparse vector search).

VECTOR-BASED NEAREST-NEIGHBOR SEARCH

Also known as KNN (*k*-nearest neighbor), vector-based nearest-neighbor search is the problem space of indexing numerical vectors of a uniform dimensionality into a data structure and searching that data structure with a query vector for the closest `k` - related vectors. We mentioned in chapter 3 that there are many similarity measures for comparing numerical vectors: cosine similarity, dot product, Euclidean distance, and so on. We will use cosine similarity (implemented as dot product over unit-normalized vectors) throughout this chapter for vector similarity comparisons.

13.2.1 A brief refresher on sparse vectors

Sparse vector search is usually implemented using an inverted index. An inverted index is like what you find in the back of any textbook—a list of terms that reference their location in the source content. To efficiently find text, we structure information in an index by processing and normalizing tokens into a dictionary with references to the postings (the document identifiers and positions in which they occur). The resulting data structure is a sparse vector representation that allows for fast lookup of those tokens.

At search time, we tokenize and normalize the query terms and, using the inverted index, match the document hits for retrieval. Then we apply the BM25 formula to score the documents and rank them by similarity, as we covered in section 3.2.

Applying scores for each query term and document feature gives us fast and relevant search, but this model suffers from the "query term dependence" model of relevance, in which the terms (and normalized forms) are retrieved and ranked. The problem is that it uses the presence (and count) of query term strings to search and rank instead of the *meaning* represented by those strings. Therefore, the relevance scores are only useful in a relative sense, to tell you which documents best matched the query, but not to measure if any of the documents were objectively good matches. Dense vector approaches, as we'll see, can supply a more global sense of relevance that is also comparable across queries.

13.2.2 A conceptual dense vector search engine

We want to capture the meaning of content when we process documents, and we want to retrieve and rank based on the meaning and intent of the query when searching. With this goal in mind, we process documents to generate embeddings and then store those embeddings in the search index. At search time, we process queries to get embeddings and use those query embeddings to search the indexed document embeddings. Figure 13.1 shows a simplified diagram of this process, which we'll expand upon in section 13.4.

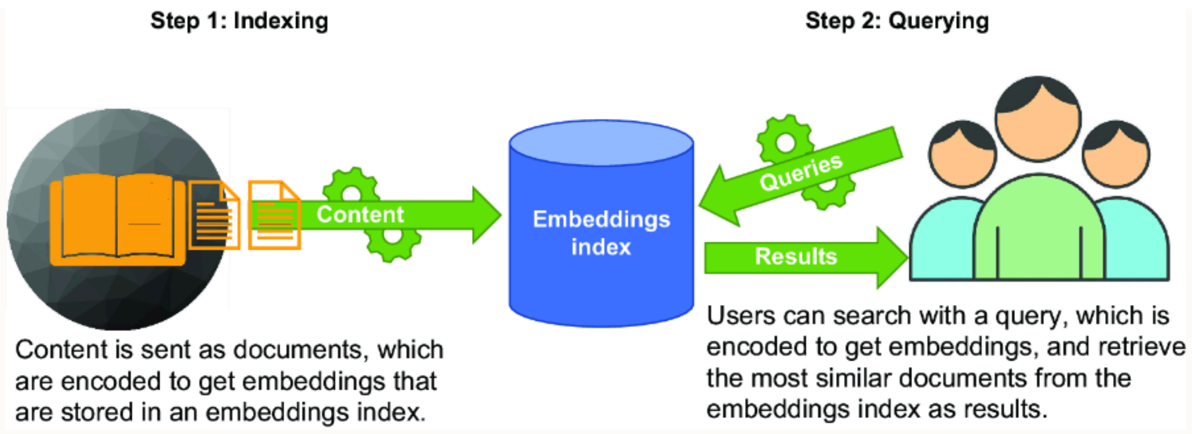

Step 1: Indexing
Step 2: Querying
Content
Embeddings index
Queries
Results
Content is sent as documents, which are encoded to get embeddings that are stored in an embeddings index.
Users can search with a query, which is encoded to get embeddings, and retrieve the most similar documents from the embeddings index as results.

Figure 13.1 Building and searching the embeddings index. The content is processed and added to the index from the left, and a user queries the index to retrieve results.

The embeddings for both documents and queries exist in the same vector space. This is very important. If you map documents to one vector space and queries to another vector space, you'll be matching apples to oranges. The embeddings must belong to the same space for this to work effectively.

But what is an "embedding" exactly, and how do we search one? Well, an embedding is a vector of some set number of dimensions representing information. That information could be a query, a document, a word, a sentence, an image or video, or any other type of information.

EMBEDDING SCOPE AND CHUNKING

One important engineering task when working with embeddings is to figure out the right level of granularity for an embedding. An embedding can be made to represent a single word, sentence, paragraph, or much larger document.

When generating embeddings, it is often useful to break apart larger documents into sections and generate a separate embedding for each section, a process known as *chunking*. You can chunk your content by sentence, paragraph, or other conceptual boundaries, and you can even create overlapping chunks to ensure that the process of splitting the document doesn't destroy relevant context across splits.

If your search engine supports multivalued vector fields, you can index many embeddings into a single document and match based on any of its embeddings. Alternatively, you can index a separate document for each chunk, each with a single embedding, and then store the original document ID as a field to be returned when the indexed chunk document is matched.

It is difficult for very large chunks to be fully represented by an embedding, just as it is difficult for very small chunks to contain the full context needed for an embedding, so figuring out the right chunking granularity for your app can be an important consideration for improving recall.

Since embeddings are represented as vectors, we can use cosine similarity (which was covered in depth in chapters 2–3) or another similar distance measurement to compare two embedding vectors to each other and get a similarity score. This allows us to compare the vector of a query to the vectors of all the documents in the content we want to search. The document vectors that are the most similar to the query vector are referred to as *nearest neighbors*. Figure 13.2 illustrates this with three 2D vectors.

The cosine similarities between the vectors shown in figure 13.2, ordered by highest similarity first, are as follows:

- `cos(b, c) = 0.9762`
- `cos(a, b) = 0.7962`
- `cos(a, c) = 0.6459`

It is clear both visually and mathematically that `b` and `c` are closest to each other, so we say that `b` and `c` are the most similar of the three vectors.

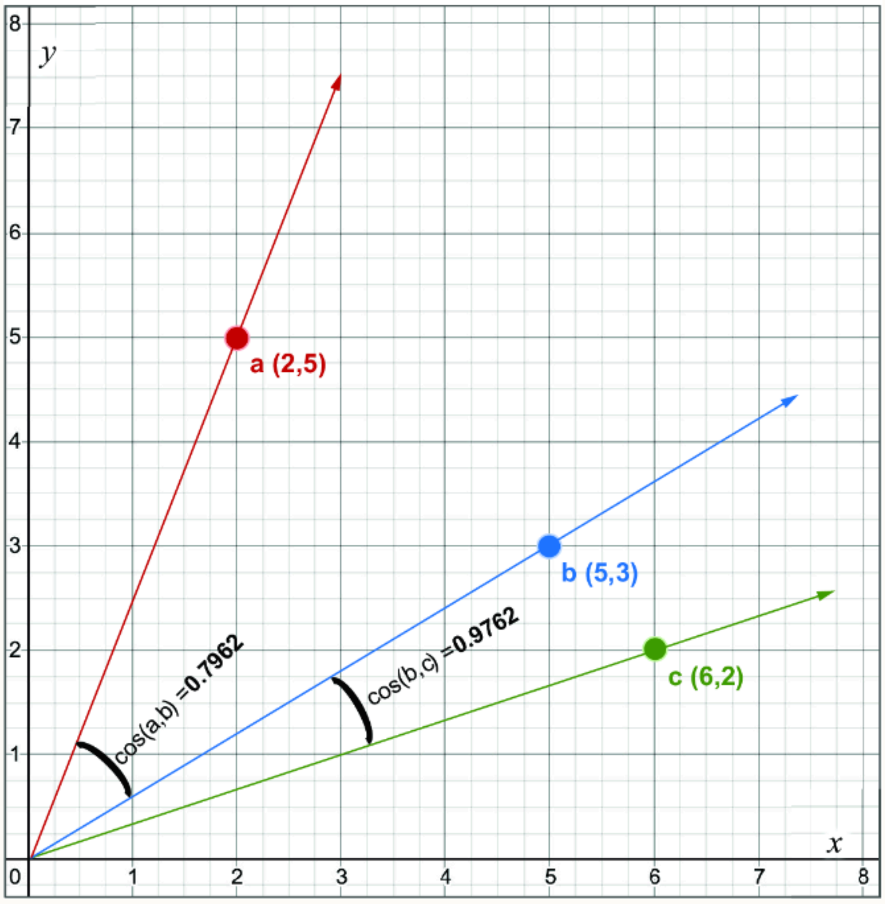


Figure 13.2 Three vectors (a, b, and c) plotted on a Cartesian plane. The similarities between a and b, and between b and c, are illustrated using the cos*q* function.

We can easily apply cosine similarity to vectors of any length. In 3D space, we compare vectors with three features `[x, y, z]`. In a dense vector embedding space, we may use vectors with hundreds or thousands of dimensions. But no matter the number of dimensions, the formula is the same, as shown in figure 13.3.

$$\text{Vector similarity}(\mathbf{a}, \mathbf{b}): \ \cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| \times |\mathbf{b}|}$$

Figure 13.3 Formula for the cosine similarity of two vectors

See section 3.1 for a recap of using this cosine similarity calculation to score the similarity between vectors. There, we walked through examples of calculating both the cosine similarity between vectors and the dot product between vectors. From the formula for cosine similarity (figure 13.3), however, you can see that the cosine is equal to the dot product (`a . b`) divided by the product of the lengths of the vectors (`|a| x |b|`). This means that if we can normalize the features in vectors `a` and `b` so that each of their lengths is `1` (a process called *unit-normalizing*), the cosine similarity and dot product are equal to each other:

```
1 = |a| = |b|
cos(a, b) = (a . b) / |a| x |b|
cos(a, b) = (a . b) / (1 x 1)
cos(a, b) = a . b
```

When a vector is normalized such that its length equals `1`, it is known as a *unit-normalized* vector. But why would we care about normalizing vectors like this? Well, it turns out that calculating a dot product is much more efficient than calculating a cosine similarity, because there is no need to divide the dot product by the magnitudes of each vector (which requires using the Pythagorean theorem to calculate the square root of the sum of squares of each vector's features). Since the cosine calculation is often the most expensive part of the search when scoring a large number of documents, unit-normalizing vectors at index time and performing a dot product of the indexed vectors with a unit-normalized query vector at search time can substantially speed up searches while providing the same result:

```
vector_a = [5.0, 3.0]
vector_b = [6.0, 2.0]

unit_vector_a  #1
  = unit_normalize(vector_a)  #1
  = unit_normalize([5.0, 3.0])  #1
  = [5.0 / sqrt(5.0^2 + 3.0^2), 3.0 / sqrt(5.0^2 + 3.0^2)]  #1
  = [0.8575,  0.5145]  #1
  #1
unit_vector_b  #1
  = unit_normalize(vector_b)  #1
  = unit_normalize([6.0, 2.0])  #1
  = [6.0 / sqrt(6.0^2 + 2.0^2), 2.0 / sqrt(6.0^2 + 2.0^2)]  #1
  = [0.9487, 0.3162]  #1

cos(vector_a, vector_b)  #2
  = cos([5.0, 3.0], [6.0, 2.0])  #2
  = (5.0 x 6.0 + 3.0 x 2.0) /  #2
    (sqrt(5.0^2 + 3.0^2) x sqrt(6.0^2 + 2.0^2))  #2
  = 0.9762  #3

dot_product(unit_vector_a, unit_vector_b)  #4
  = dot_product([0.8575, 0.5145], [0.9487, 0.3162])  #4
  = (0.8575 x 0.9487) + (0.5145 x 0.3162)  #4
  = 0.9762  #3
```

#1 The normalization of vectors a and b to a unit vector. All indexed vectors have this normalization done once, before being indexed.
#2 Full cosine similarity calculation. Notice the denominator performs a square root of the sum of the squares for each vector.
#3 cos(vector_a, vector_b) = dot_product(unit_vector_a, unit_vector_b) = 0.9762
#4 The dot product calculation on unit-normalized vectors. Notice the absence of the denominator and the much simpler sum of multiplied feature weights.

While we are conceptually still performing a cosine similarity (due to the unit-normalized vectors), using the dot product allows us to perform substantially faster calculations at query time. Because this optimization is possible, it is not a good idea performance-wise to perform a full cosine similarity calculation in production. While there are good use-case-specific reasons you would want to perform a cosine versus dot product, such as to ignore or consider the magnitude of vectors (see section 3.1.4 for a refresher), you will virtually always at least *implement* cosine similarity using unit-normalized vectors and a dot product calculation for performance reasons. To reinforce best practices, we'll consistently utilize this pattern in all the remaining code listings when cosine similarity is being implemented.

OPTIMIZING FOR VECTOR SEARCH COMPUTE PERFORMANCE AND COST

Performing vector similarity calculations can be slow and computationally expensive at scale, so it's important to understand how to make the right trade-offs to optimize for performance and cost.

Because dot products are significantly faster to calculate than cosine similarities, we recommend you always implement cosine similarity by indexing unit-normalized vectors and then doing dot product calculations between document vectors and a unit-normalized query vector at search time. In addition, it is often possible to save significant memory and substantially improve search times using other optimization techniques:

- Using approximate nearest neighbors (ANN) approaches to quickly filter to the top-*N* results to rank instead of all documents (covered in section 13.5.3)
- Quantizing (compressing) vectors to reduce the number of bits used to represent each feature in a vector (covered in section 13.7)
- Using Matryoshka Representation Learning (MRL) to only index and/or search on key portions of your embeddings while still maintaining most of your recall (covered in section 13.7)
- Over-requesting a limited number of optimized search results using a cheaper search algorithm or similarity metric, and then reranking the top-*N* results using a more expensive similarity metric (covered in sections 13.5.3 and 13.7)

With the ability to perform dense vector search and nearest-neighbor similarity in place, the next critical step is to figure out a way to generate these mysterious embeddings.

13.3 Getting text embeddings by using a Transformer encoder

In this section, we cover Transformers and how they work to represent meaning. We also discuss how they are used to encode that meaning into embeddings.

13.3.1 What is a Transformer?

Transformers are a class of deep neural network architectures that are optimized to encode meaning as embeddings and also decode the meaning back from embeddings. Text-based Transformers do this by first representing term labels as dense vectors by using their surrounding context in a sentence (the encoding part) and then leveraging an output model to translate the vectors into different text representations (the decoding part).

One beautiful feature of this approach is the separation of concerns between encoding and decoding. We will take advantage of this feature and use the encoding mechanism just to obtain the embeddings, which we can then use as a semantic representation of meaning independent of any decoding steps.

REPRESENTING MEANING

Recall the example English and Armenian greetings from the introduction to section 13.1. Using a specialized Transformer and dataset for English to Armenian language translation, it would be possible to train a model to encode the two phrases "Hello to you!" and "Barev Dzes" into nearly identical dense vectors. These vectors could then be decoded back into text to translate or reconstitute something closer to the original text.

Let's start our journey with Transformers by understanding how Transformer encoder models are trained and what they ultimately learn. To understand the motivations and mechanisms behind Transformers, it is important to know some history of the underlying concepts.

The year is 1953. You find yourself in a classroom with 20 other students, each of you sitting at your own desk. On your desk is a pencil, and a sheet of paper with the sentence `Q: I went to the ________ and bought some vegetables.` You already know what to do, and you write down "store" in the blank. You peek over at a classmate at the desk next to you, and they have written "market". A chime rings, and the answers are tallied. The most common answer is "store", and there are several with "market" and several with "grocer". This is the Cloze test. It is meant to test reading comprehension.

You are transported now to the year 1995. You are sitting in another classroom with students taking another test. This time, your sheet of paper has a very long paragraph. It looks to be about 60 words long and is somewhat complicated:

> *Ours was the marsh country, down by the river, within, as the river wound, twenty miles of the sea. My first most vivid and broad impression of the identity of things seems to me to have been gained on a memorable raw afternoon towards evening. At such a time I found out for certain that this bleak place overgrown with nettles was the churchyard.*

After the paragraph, there is a question listed with a prompt for an answer: `Q: How far away from the sea is the churchyard? A:________`. You write in the blank, "twenty miles". You have just completed one of a dozen questions in the Regents English reading comprehension test. Specifically, this tested your attention.

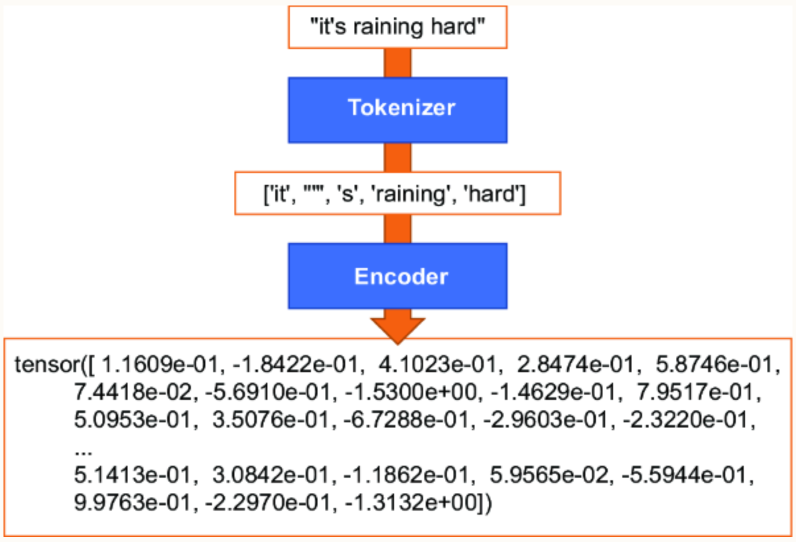
These two tests are foundational to how we measure written language comprehension. To be able to pass these tests, you must read, read, read, and read some more. In fact, by the time most people take these tests in school, they have already practiced reading for about 14 years and have amassed a huge amount of contextual knowledge.

These theories form the basis for LLMs—NLP models trained on lots and lots of text (for example, the entirety of the Common Crawl dataset of the web).

Major breakthroughs in the NLP field culminated in a 2018 paper by researchers at Google (Jacob Devlin et al.) titled *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, which made use of the Cloze test and attention mechanisms within Transformers to reach state-of-the-art performance on many language comprehension benchmarks (**https://arxiv.org/pdf/1810.04805**).

BERT, specifically, performs self-learning by presenting Cloze tests to itself. The training style is “self-supervised”, which means it is supervised learning, framed as an unsupervised task. This is ideal, because it does not require data to be manually labeled beforehand for the initial model pretraining. You can give it any text, and it will make the tests itself. In the training context, the Cloze test is known as *masked language modeling*. The model starts with a more basic embedding (for example, using the well-known word2vec or GloVe libraries for each word in the vocabulary), and it will randomly remove 15% of the tokens in a sentence for the test. The model then optimizes a loss function that will result in a higher Cloze test success rate. Also, during the training process, it uses the surrounding tokens and contexts (the attention). Given a vector in a single training example, the resulting trained output vector is an embedding that contains a deeply learned representation of the word and the surrounding contexts.

We encourage you to learn more about Transformers and BERT, if you are interested, by reading the paper. All you need to understand for now, however, is how to get embeddings from a BERT encoder. The basic concept is shown in figure 13.4.



```
tensor([ 1.1609e-01, -1.8422e-01,  4.1023e-01,  2.8474e-01,  5.8746e-01,
         7.4418e-02, -5.6910e-01, -1.5300e+00, -1.4629e-01,  7.9517e-01,
         5.0953e-01,  3.5076e-01, -6.7288e-01, -2.9603e-01, -2.3220e-01,
         ...
         5.1413e-01,  3.0842e-01, -1.1862e-01,  5.9565e-02, -5.5944e-01,
         9.9763e-01, -2.2970e-01, -1.3132e+00])
```

Figure 13.4 The Transformer encoder

In figure 13.4, we process text by first passing it through a tokenizer. The tokenizer splits text into *word pieces*, which are predefined parts of words that are represented in a vocabulary. This vocabulary is established for the model before it is trained. For example, the term “It’s” will be split into three word pieces during tokenization: `it`, `'`, and `s`. The vocabulary used in the BERT paper included 30,000 word pieces. BERT also uses special word pieces to denote the beginning and end of sentences: `[CLS]` and `[SEP]` respectively. Once tokenized, the token stream is passed into the BERT model for encoding. The encoding process then outputs a *tensor*, which is an array of vectors (one vector for each token).

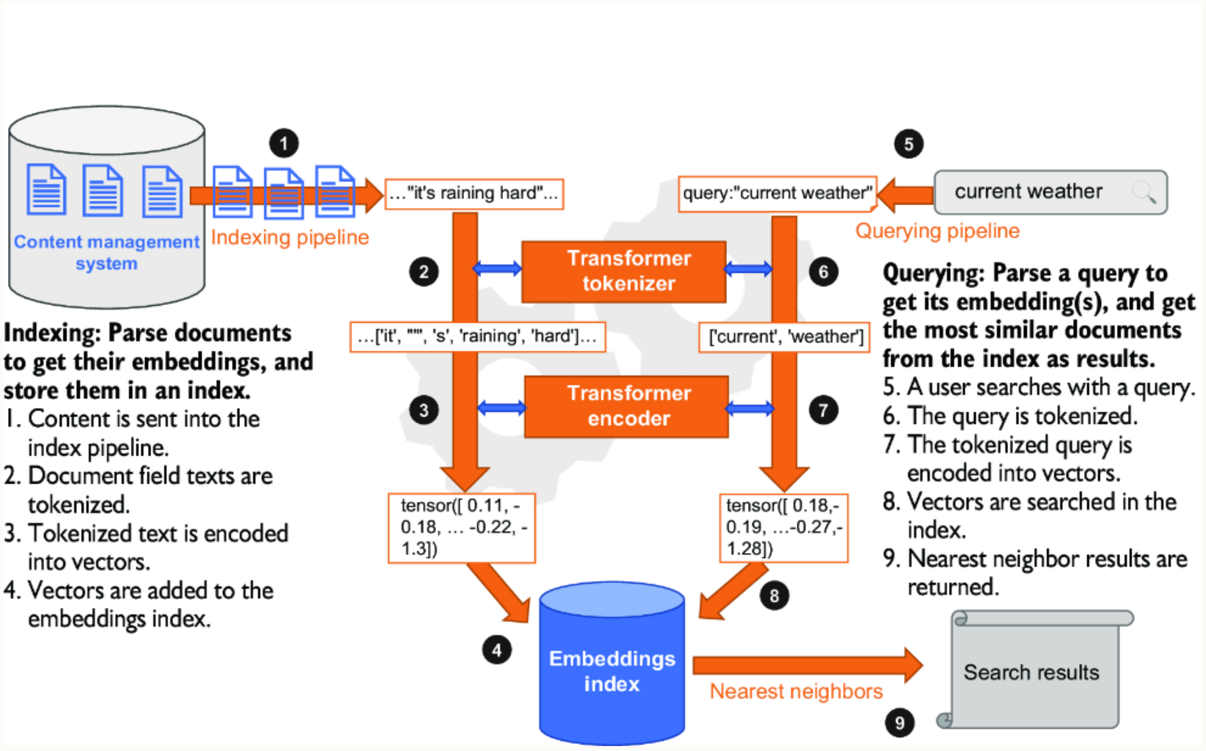
13.3.2 Openly available pretrained Transformer models

While Transformers enable state-of-the-art language models to be built, having the knowledge and resources to build them from scratch can present a large hurdle for many. One very important aspect of working with Transformers is the large community and open toolsets that make it possible for any engineer to quickly get up and running with the technology. All it takes is some knowledge of Python and an internet connection.

The models that are trained by this process from scratch are large and range from hundreds of MBs to hundreds of GBs in size, often needing similar amounts of GPU memory (VRAM) to run them quickly. The training itself also takes a large amount of expensive computing power and time, so being able to use preexisting models as a starting point provides a significant advantage. We'll use this advantage in the next section as we begin applying one of these models to search.

13.4 Applying Transformers to search

In this section, we'll build a highly accurate natural language autocomplete for search, which will recommend more precise and otherwise related keywords based on a prefix of terms. We'll do this by first passing our corpus text through a Transformer to get an index of embeddings. Then we'll use this Transformer at query time to get the query embedding and search the embedding index for the *k*-nearest documents with the most similar embeddings. Figure 13.5 is an architecture diagram demonstrating the steps in this process.



Indexing: Parse documents to get their embeddings, and store them in an index.

1. Content is sent into the index pipeline.
2. Document field texts are tokenized.
3. Tokenized text is encoded into vectors.
4. Vectors are added to the embeddings index.

Querying: Parse a query to get its embedding(s), and get the most similar documents from the index as results.

5. A user searches with a query.
6. The query is tokenized.
7. The tokenized query is encoded into vectors.
8. Vectors are searched in the index.
9. Nearest neighbor results are returned.

Figure 13.5 A conceptual architecture for end-to-end search using Transformer-encoded vectors

We have a content source, a nearest-neighbor index, a way to retrieve vectors from a Transformer, and a similarity formula. We can now build pipelines for all these pieces to process and index content, and then to retrieve and rank documents with a query.

13.4.1 Using the Stack Exchange outdoors dataset

In chapter 5, we introduced several datasets from Stack Exchange. We're choosing to use another one, the Stack Exchange outdoors dataset, here for a very important reason: the vocabulary and contexts in the outdoor question-and-answer domain already have good coverage in the Transformer models we'll be using. Specifically, Wikipedia is used when training many Transformer models, and Wikipedia has a section specifically on outdoors content (https://en.wikipedia.org/wiki/Outdoor).

NOTE The outdoors examples used in chapters 13–15, along with references to Stack Exchange datasets in other chapters, are licensed by Stack Exchange under the CC-by-SA 4.0 License: https://creativecommons.org/licenses/by-sa/4.0.

The following listing walks through creating an outdoors collection and then indexing the outdoors question and answer data.

Listing 13.1 Indexing the outdoors dataset

```
outdoors_collection = engine.create_collection("outdoors")
outdoors_dataframe = load_outdoors_data("data/outdoors/posts.csv")
outdoors_collection.write(outdoors_dataframe)
```

This is the schema for the `outdoors` collection created in listing 13.1:

```
root
 |-- id: integer (nullable = true)
 |-- accepted_answer_id: integer (nullable = true)
 |-- parent_id: integer (nullable = true)
 |-- creation_date: timestamp (nullable = true)
 |-- score: integer (nullable = true)
 |-- view_count: integer (nullable = false)
 |-- body: string (nullable = true)
 |-- owner_user_id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- tags: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- answer_count: integer (nullable = true)
 |-- post_type: string (nullable = true)
 |-- url: string (nullable = true)
```

The indexed dataset contains documents representing both questions and answers, with answers linked to their original questions through the `parent_id` field. Each document contains a `post_type` field to differentiate whether it contains a “question” or an “answer”.

The following listing shows a question post and its related answers.

Listing 13.2 Exploring post data for a question on `climbing knots`

```
[{"id": "18825",
  "accepted_answer_id": 18826, #1
  "body": "If I wanted to learn how to tie certain knots,
  ↪or learn about new knots and what they're used for,
  ↪what are some good resources to look up?",
  "title": "What's a good resource for learning to tie knots for climbing?",
  "post_type": "question"},  #2
 {"id": "24440",  #3
  "parent_id": 18825,  #3
  "body": "Knots and Ropes for Climbers by Duane Raleigh is a fantastic
  ↪illustrated resource tailored specifically to climbers. The ABoK
  ↪is great, but a but beyond the pale of what the average rock...",

 "post_type": "answer"},   #4
 {"id": "18826",  #5
  "parent_id": 18825,  #3
  "body": "Animated Knots By Grog Arguably the best resource online for knot
  ↪tying is Animated Knots by Grog , it's used by virtually every avid
  ↪knot tyer I've known. They have excellent step-by-step animatio...",
  "post_type": "answer"}]  #4
```

#1 Document 18826 is marked as the accepted answer to the question.
#2 This is our question document.
#3 These documents are answers to the question in the first document (id=18825).
#4 These are answer documents that relate to the question.
#5 This is the document marked as the accepted answer to the question (accepted_answer_id=18826).

In the preceding listing, the first document is a question most related to the query `climbing knots`. The question has two answers that are linked back to the parent question through the `parent_id` field on each answer. One of these answers has been chosen as the accepted answer, which is identified by setting the `accepted_answer_id` field (to 18826 in this case) on the question document.

The `body` field of question documents contains elaborations on the question, while the `body` field of an answer contains the full answer. Only question posts have a `title`, which is a summary of the question. Several other fields (such as `view_count`, `answer_count`, and `owner_user_id`) are omitted here but are available in the full dataset as metadata fields, which can help with search relevance using BM25 mixed with other signals.

Now that you're familiarized with the data model, let's take a moment to try out some queries and see what types of questions come back. The following listing searches for questions matching a common query.

Listing 13.3 Running a basic lexical search for `climbing` `knots`

```
def search_questions(query, verbose=False):
  request = {"query": query,
             "query_fields": ["title", "body"],  #1
             "limit": 5,
             "return_fields": ["id", "url", "post_type", "title",
                               "body", "accepted_answer_id", "score"],
             "filters": [("post_type", "question")],
             "order_by": [("score", "desc"), ("title", "asc")]}
  response = outdoors_collection.search(**request)
  display_questions(query, response, verbose)

search_questions("climbing knots")
```

#1 Match the query against the title and body fields.

Response:

```
Query: climbing knots

Ranked Questions:
Question 21855: What are the four climbing knots used by Jim Bridwell?
Question 18825: What's a good resource for learning to tie knots for clim...
Question 18814: How to tie a figure eight on a bight?
Question 9183: Can rock climbers easily transition to canyoning?
Question 22477: Tradeoffs between different stopper knots
```

We can see that these are somewhat relevant titles for this lexical query. But this is just a basic keyword search. Other queries do not perform nearly as well; for example, the query `What is DEET` in the next listing shows very irrelevant results.

Listing 13.4 Basic lexical matching can yield irrelevant results

```
search_questions("What is DEET?")
```

Response:

```
Query What is DEET?:

Ranked Questions:
Question 20403: What is bushcrafting?
Question 20977: What is "catskiing"?
Question 1660: What is Geocaching?
Question 17374: What is a tent skirt and what is its purpose?
Question 913: What is a buff?
```

This shows how traditional lexical search can fail for common natural language use cases. Specifically, the inverted index suffers from the query-term dependency problem. This means that the terms in the query are being matched as strings to terms in the index. This

is why you see strong matches for `what is` in the results in listing 13.4. The *meaning* of the query is not comprehended, so the retrieval can only be based on string matching.
The rest of this chapter will provide the fundamentals needed to use Transformers for natural language search, and in chapter 14 we'll solve the question-answering problem evident in listing 13.4.

13.4.2 Fine-tuning and the Semantic Text Similarity Benchmark

Using a pretrained Transformer model out of the box usually won't yield optimal results for task-specific prompts. This is because the initial training was done in a general language context without any specific use case or domain in mind. In essence, it is "untuned", and using models this way is akin to indexing content in a search engine without tuning for relevance.

To realize the full potential of Transformers, they need to be refined to accomplish a specific task. This is known as *fine-tuning*—the process of taking a pretrained model and training it on more fit-for-purpose data to achieve a specific use case goal. For both autocomplete and semantic search, we are interested in fine-tuning to accomplish text similarity discovery tasks.

That brings us to the Semantic Text Similarity Benchmark (STS-B) training and testing set (**https://ixa2.si.ehu.eus/stswiki/**). This benchmark includes passages that are semantically similar and dissimilar, and they're labeled accordingly. Using this dataset, a model can be fine-tuned to improve the accuracy of nearest-neighbor search between a set of terms and many passages in a corpus, which will be our use case in this chapter.

In chapter 14, we will fine-tune our own question-answering model so you can see how it is done. For our purposes in this chapter, however, we'll use a project that already includes a pretuned model for this task: SBERT.

13.4.3 Introducing the SBERT Transformer library

SBERT, or *Sentence-BERT*, is a technique and Python library based on Transformers that is built on the idea that a BERT model can be fine-tuned in such a way that two semantically similar sentences, and not just tokens, should be represented closer in vector space. Specifically, SBERT *pools* all the BERT embeddings in one sentence to a single vector. (Pooling is a fancy way of saying it combines the values.) Once SBERT pools the values, it trains for similarity between sentences by using a special-purpose neural network that learns to optimize for the STS-B task. For further implementation details, check out the "Sentence-BERT" paper by Nils Reimers and Iryna Gurevych (**https://arxiv.org/abs/1908.10084**).

The upcoming listings will give you an overview of how to use SBERT via the `sentence_transformers` Python library. We'll start by importing the library using a pretrained model named `roberta-base-nli-stsb-mean-tokens`, which is based on an architecture called RoBERTa. It's helpful to think of RoBERTa as an evolved and improved version of BERT, with optimized hyperparameters (configuration settings) and slight modifications to the original techniques.

HYPERPARAMETERS

In machine learning, hyperparameters are any parameter values that can be changed before training and that will alter the learning process and affect the resulting model.

Unfortunately, you often don't know what the hyperparameter values should be set to when you start, so you may have to learn optimized values over time using iteration and measurement.

In the model name, `roberta-base-nli-stsb-mean-tokens`, we can also see some terms that you may not recognize, including "nli" and "mean-tokens". NLI stands for *natural language inference* (a subdomain of NLP used for language prediction), and mean-tokens refers to the whole sentence's tokenization being pooled together as a mean of the numeric values of the token embeddings. Using mean-tokens returns a single 768-dimension embedding for the entire sentence.

The following listing imports the `sentence_transformers` library, loads the model, and displays the full network architecture.

Listing 13.5 Loading the RoBERTa `SentenceTransformer` model

```
from sentence_transformers import SentenceTransformer
transformer = SentenceTransformer("roberta-base-nli-stsb-mean-tokens")
```

Now the PyTorch `transformer` object holds the neural network architecture for the Transformer as well as all the model weights.

With our model loaded, we can retrieve embeddings from text. This is where the fun really begins. We can take sentences and pass them through the neural network architecture using the pretrained model and get the embeddings as a result. We have four sentences that we'll encode and assess in the following listings.

Listing 13.6 demonstrates how to encode multiple phrases into dense vector embeddings.

Listing 13.6 Encoding phrases as dense vector embeddings

```
phrases = ["it's raining hard", "it is wet outside",  #1
           "cars drive fast", "motorcycles are loud"]  #1
embeddings = transformer.encode(phrases, convert_to_tensor=True)  #2
print("Number of embeddings:", len(embeddings))
print("Dimensions per embedding:", len(embeddings[0]))
print("The embedding feature values of \"it's raining hard\":")
print(embeddings[0])
```

#1 The four sentences we want to encode. We will pass all of these to be encoded as a single batch.
#2 Just call transformer.encode, and the abstraction of sentence_transformers does all the heavy lifting for you.

Response:

```
Number of embeddings: 4
Dimensions per embedding: 768
The embedding feature values of "it's raining hard":
tensor( 1.1609e-01, -1.8422e-01,  4.1023e-01,  2.8474e-01,  5.8746e-01,
        7.4418e-02, -5.6910e-01, -1.5300e+00, -1.4629e-01,  7.9517e-01,
        5.0953e-01,  3.5076e-01, -6.7288e-01, -2.9603e-01, -2.3220e-01,
        ...
        5.1413e-01,  3.0842e-01, -1.1862e-01,  5.9565e-02, -5.5944e-01,
        9.9763e-01, -2.2970e-01, -1.3132e+00])
```

In the preceding listing, we take each sentence and pass it to the encoder. This results in a tensor for each sentence. A *tensor* is a generalizable data structure for holding potentially multidimensional values. A scalar (single value), vector (array of scalars), matrix (array of vectors), or even multidimensional matrix (array of matrices, matrix of matrices, etc.) are all examples of tensors of various dimensions. Tensors are produced by Transformer encoders, such as SBERT, when encoding text. For our use case, the tensor in listing 13.6 is an embedding containing 768 dimensions represented as floating point numbers.

With our embeddings, we can now perform cosine similarities (dot product on unit-normalized vectors) to see which phrases are closest neighbors to each other. We'll compare each phrase to every other phrase, and sort them by similarity to see which are most similar. This process is covered step by step in listings 13.7 and 13.8.

We'll use a PyTorch built-in library for calculating the dot product to do these comparisons, which allows us to pass in the embeddings with a single function call. We can then sort the resulting similarities and see which two phrases are most similar to one another and which two are most dissimilar. The following listing calculates the similarities between each of the phrase embeddings.

Listing 13.7 Comparing all the phrases to each other

```
def normalize_embedding(embedding): #1
  normalized = numpy.divide(embedding, numpy.linalg.norm(embedding))
  return list(map(float, normalized))

normalized_embeddings = list(map(normalize_embedding, embeddings))
similarities = sentence_transformers.util.dot_score(normalized_embeddings,
                                                    normalized_embeddings)
print("The shape of the resulting similarities:", similarities.shape)
```

#1 Unit-normalizes embeddings for speed so dot products = cosine similarities

Output:

```
The shape of the resulting similarities: torch.Size([4, 4])
```

We print the shape of the similarities object in listing 13.7 to see how many comparisons we have. The shape is 4 x 4 (`[4, 4]`) because we have 4 phrases, and each phrase has a similarity score to every other phrase and itself. All the similarity scores are between 0.0 (least similar) and 1.0 (most similar). The shape is included here to help show the complexity of comparing many phrases. If there were 100 phrases, the similarities shape would be 100 x 100. If there were 10,000 phrases, the similarities shape would be 10,000 x 10,000. So, as you add phrases to compare, the computational and space costs will increase with complexity n^2, where n is the number of phrases.

With the similarities for our four phrases computed, we sort and print them in the following listing.

Listing 13.8 Sorting by similarities and printing the results

```
def rank_similarities(phrases, similarities, name=None):
  a_phrases = []
  b_phrases = []
  scores = []
  for a in range(len(similarities) - 1):  #1
    for b in range(a + 1, len(similarities)): #2
      a_phrases.append(phrases[a])
      b_phrases.append(phrases[b])
      scores.append(float(similarities[a][b]))  #3

  dataframe = pandas.DataFrame({"score": scores, "phrase a": a_phrases,
                                "phrase b": b_phrases})
  dataframe["idx"] = dataframe.index  #4
  dataframe = dataframe.reindex(columns=["idx", "score",
                                         "phrase a", "phrase b"])

  return dataframe.sort_values(by=["score"],  #5
                               ascending=False,  #5
                               ignore_index=True)  #5

dataframe = rank_similarities(phrases, similarities)
display(HTML(dataframe.to_html(index=False)))
```

#1 Append all the phrase pairs to a dataframe.
#2 We don't duplicate phrases or append a phrase's similarity to itself, as it will always be 1.0.
#3 Get the score for each pair.
#4 Add the index column
#5 Sort the scores (ascending=False for highest scores first).

Response:

```
idx  score     phrase a           phrase b
0    0.669060  it's raining hard  it is wet outside
5    0.590783  cars drive fast    motorcycles are loud
1    0.281166  it's raining hard  cars drive fast
2    0.280800  it's raining hard  motorcycles are loud
4    0.204867  it is wet outside  motorcycles are loud
3    0.138172  it is wet outside  cars drive fast
```

We can now see that the two phrases that are most similar to one another are "it's raining hard" and "it is wet outside". We also see a strong similarity between cars and motorcycles.

The two most dissimilar phrases are “it is wet outside” and “cars drive fast”. It’s very clear from these examples that this semantic encoding process is working—we can associate rain with it being wet outside. The dense vector representations captured the context, and even though the words are different, the meaning is still there. Note the scores: the top two similar comparisons have a score of greater than 0.59, and the next closest comparison has a score of less than 0.29, This is because only the top two comparisons seem to be similar to one another, as we would perceive them in a *natural language understanding* (NLU) task. As intelligent people, we can group rain and wet (“weather”), and we can also group cars and motorcycles (“vehicle”). Also interestingly, cars likely drive slower when it is wet on the ground, so that likely explains the low similarity of the last pair.

13.5 Natural language autocomplete

Now that we know our vector encoding and similarity process is working well, it’s time to put this embedding technique to work in a real search use case—natural language autocomplete!

In this section, we’ll show a practical use for sentence Transformers at search time, with a basic and fast semantic autocomplete implementation. We will apply what we’ve learned thus far to extract concepts from the outdoors dataset. Using spaCy (the Python NLP library we used in chapter 5), we will chunk nouns and verbs to get outdoors concepts. We will put these concepts in a dictionary and process them to get their embeddings. Then we’ll use the dictionary in an approximate nearest-neighbor (ANN) index to query in real time. This will give us the ability to enter a prefix or a term and get the most similar concepts that exist in the dictionary. Finally, we will take those concepts and present them to the user in order of similarity, demonstrating a smart, natural language autocomplete.

Experience and testing show that this works much better than even a well-tuned suggester in most lexical search engines. We will see that it’s much less noisy and also that similar terms that are not spelled the same will be automatically included in the suggestions. This is because we are not comparing keyword strings to one another; rather, we are comparing the embeddings, which represent meaning and context. This is the embodiment of searching for “things, not strings”, as introduced in section 1.2.4.

13.5.1 Getting noun and verb phrases for our nearest-neighbor vocabulary

Using what we learned in chapter 5, we’ll write a simple function to extract *concepts* from the corpus. We won’t include any taxonomical hierarchy, and we won’t be building a full knowledge graph here. We just want a reliable list of frequently used nouns and verbs.

The concepts in our example are the important “things” and “actions” that people usually search for. We also need to understand the dataset, which is best accomplished by spending time reviewing the concepts and how they relate to one another. Understanding the corpus is critical when building any search application, and there’s no exception when using advanced NLP techniques.

The following listing shows a strategy that will provide a decent quality baseline of candidate concepts for our vocabulary while removing significant noise from the autocomplete results.

Listing 13.9 Using the spaCy Matcher to get desired parts of text

```
nlp = spacy.load("en_core_web_sm")  #1
phrases = []  #2
sources = []  #3
matcher = Matcher(nlp.vocab) #4
nountags = ["NN", "NNP", "NNS", "NOUN"]  #5
verbtags = ["VB", "VBD", "VBG", "VBN", ) #6
            "VBP", "VBZ", "VERB"]  #6




matcher.add("noun_phrases", [[{"TAG": {"IN": nountags},  #7
                               "IS_ALPHA": True,  #7
                               "OP": "+"}]])  #7
matcher.add("verb_phrases", [[{"TAG": {"IN": verbtags},
                               "IS_ALPHA": True, "OP": "+",
                               "LEMMA":{"NOT_IN":["be"]}}]])  #8
for doc, _ in tqdm.tqdm(nlp.pipe(yield_tuple(dataframe,  #9
                                   source_field="body",  #9
                                   total=total),)  #9
                                 batch_size=40,  #9
                                 n_threads=4,  #9
                                 as_tuples=True), #9
                        total=total):
  matches = matcher(doc)
  for _, start, end in matches:  #10
    span = doc[start:end]  #10
    phrases.append(normalize(span))  #10
    sources.append(span.text)  #10

concepts = {}
labels = {}
for i, phrase in phrases: #11
  if phrase not in concepts: #11
    concepts[phrase] = 0  #11
    labels[phrase] = sources[i]  #11
  concepts[phrase] += 1  #11
```

#1 Loads the English spaCy NLP model
#2 All the normalized noun/verb phrases ("concepts") in the corpus
#3 The original text labels that were normalized to the concept.
#4 Uses the spaCy Matcher to chunk patterns into concept labels
#5 Part-of-speech tags that match nouns
#6 Part-of-speech tags that match verbs
#7 Adds a noun phrase-matching pattern to the spaCy analysis pipeline
#8 Adds the verb phrase matching pattern. You can add more NOT_IN patterns to exclude other "stop word" verbs.
#9 Processes the body field for each outdoors question, in batches of 40 documents using 4 threads
#10 Gets all the noun and verb phrase matches, and keeps them in the sources and phrases lists
#11 Aggregates the normalized concepts by term frequency

In the preceding listing, we use the spaCy Matcher to detect patterns as part-of-speech tags. We also explicitly remove forms of the verb "to be" from the verb concepts. The verb "to be" is used frequently in many non-useful situations and often clutters the concept sug-

gestions. We could further improve quality by removing other noisy verbs like “have” and “can”, but this is just an example for now. SpaCy’s language pipeline (`nlp.pipe`) is also introduced in this listing. The `pipe` function accepts a batch size and the number of threads to use as parameters, and it then performs streaming processing of the text in parallel batches (and thus more quickly than making individual calls for each document).
With the function in listing 13.9, we can now get the list of concepts. When running this on your machine it may take some time, so be patient. The following listing returns the most prominent concepts and labels from the outdoors collection.

Listing 13.10 Generating the most frequent concepts in the corpus

```
collection = engine.get_collection("outdoors")
concepts, labels = get_concepts(collection, source_field="body",
                                load_from_cache=True)
topcons = {key: value for (key, value)
                      in concepts.items() if value > 5}
print("Total number of labels:", len(labels.keys()))
print("Total number of concepts:", len(concepts.keys()))
print("Concepts with greater than 5 term frequency:", len(topcons.keys()))
print(json.dumps(topcons, indent=2))
```

Response:

```
Total number of labels: 124366
Total number of concepts: 124366
Concepts with greater than 5 term frequency: 12375
{
  "have": 32782,
  "do": 26869,
  "use": 16793,
  ...
  "streamside vegetation": 6,
  "vehicle fluid": 6,
  "birdshot": 6
}
```

Aside from getting the concepts for the outdoors dataset, listing 13.10 filtered the total dataset to `topcons`, which only includes concepts with a frequency greater than `5`. Filtering will limit noise from terms that don’t appear as often in the corpus, such as misspellings and rare terms we don’t want to suggest in an autocomplete scenario.

13.5.2 Getting embeddings

We’re going to perform a complex normalization that will normalize similarly related concepts. But instead of algorithmic normalization (like stemming), we are normalizing to a dense vector space of 768 feature dimensions. Similar to stemming, the purpose of this is to increase *recall* (the percentage of relevant documents successfully returned). But instead of using a stemmer, we’re finding and mapping together closely related concepts. As a reminder, we’re only normalizing noun and verb phrases. Ignoring the other words is similar

to stop-word removal, but that's OK, because we want to suggest similar concepts as concisely as possible. We'll also have a much better representation of the context and meaning of the remaining phrases. Therefore, the surrounding non-noun and non-verb terms are implied.
Now that we have a list of concepts (from the last section), we're going to process them using our loaded `model` (the RoBERTa Sentence Transformer model we loaded in listing 13.5) to retrieve the embeddings. This may take a while if you don't have a GPU, so after we calculate the embeddings the first time, we'll persist them to a "pickle file" (a serialized Python object that can be easily stored and loaded to and from disk). If you ever want to rerun the notebook, you can just load the previously created pickle file and not waste another half hour reprocessing the raw text.

Hyperparameter alert! The `minimum_frequency` term is a hyperparameter, and it's set to be greater than five (`>=6`) in the following listing to minimize noise from rare terms. We'll encounter more hyperparameters in other listings in this chapter and the next, especially when we get into fine-tuning. After you've been through the rest of the listings in this chapter, we encourage you to come back and change the value of `minimum_frequency` and see how it alters the results that are retrieved. You may find a value that's more suitable and more accurate than what we've arrived at here.

Listing 13.11 Retrieving the embeddings of our concept vocabulary

```
def get_embeddings(texts, model, cache_name, ignore_cache=False):
  ...  #1
    embeddings = model.encode(texts)
  ...  #1
  return embeddings

minimum_frequency = 6  #2
phrases = [key for (key, tf) in concepts.items() if tf >= minimum_frequency]
cache_name = "outdoors_embeddings"
embeddings = get_embeddings(phrases, transformer,
                            cache_name, ignore_cache=False)

print(f"Number of embeddings: {len(embeddings)}")
print(f"Dimensions per embedding: {len(embeddings[0])}")
```

#1 Caching code removed for brevity
#2 This is a hyperparameter! We are ignoring terms that occur less than this number of times in the entire corpus. Lowering this threshold may lower precision, and raising it may lower recall.
Response:

```
Number of embeddings: 12375
Dimensions per embedding: 768
```

From listing 13.11, you can see that one embedding was generated from each of our 12,375 concepts. All embeddings have the same dimensionality from the same dense vector space and can therefore be directly compared with one another.

Figure 13.6 demonstrates what these embeddings look like and how they relate to one another when plotted in 3D.

The similarities of some concepts in the figure have been labeled to show neighborhoods of meaning. Concepts related to “wind” and “block” illustrate where they are located relative to each other in the vector space. We used dimensionality reduction to reduce the 768 dimensions for each embedding into 3 dimensions (x, y, z) so they could be easily plotted. *Dimensionality reduction* is a technique for condensing one vector with many features into another vector with fewer features. During this reduction, the relationships in the vector space are maintained as much as possible.

DIMENSIONALITY REDUCTION LOSES CONTEXT

A lot of context is lost when performing dimensionality reduction, so the visualization in figure 13.6 is only presented to give you an intuition of the vector space and concept similarity, not to suggest that reducing to three dimensions is an ideal way to represent the concepts.

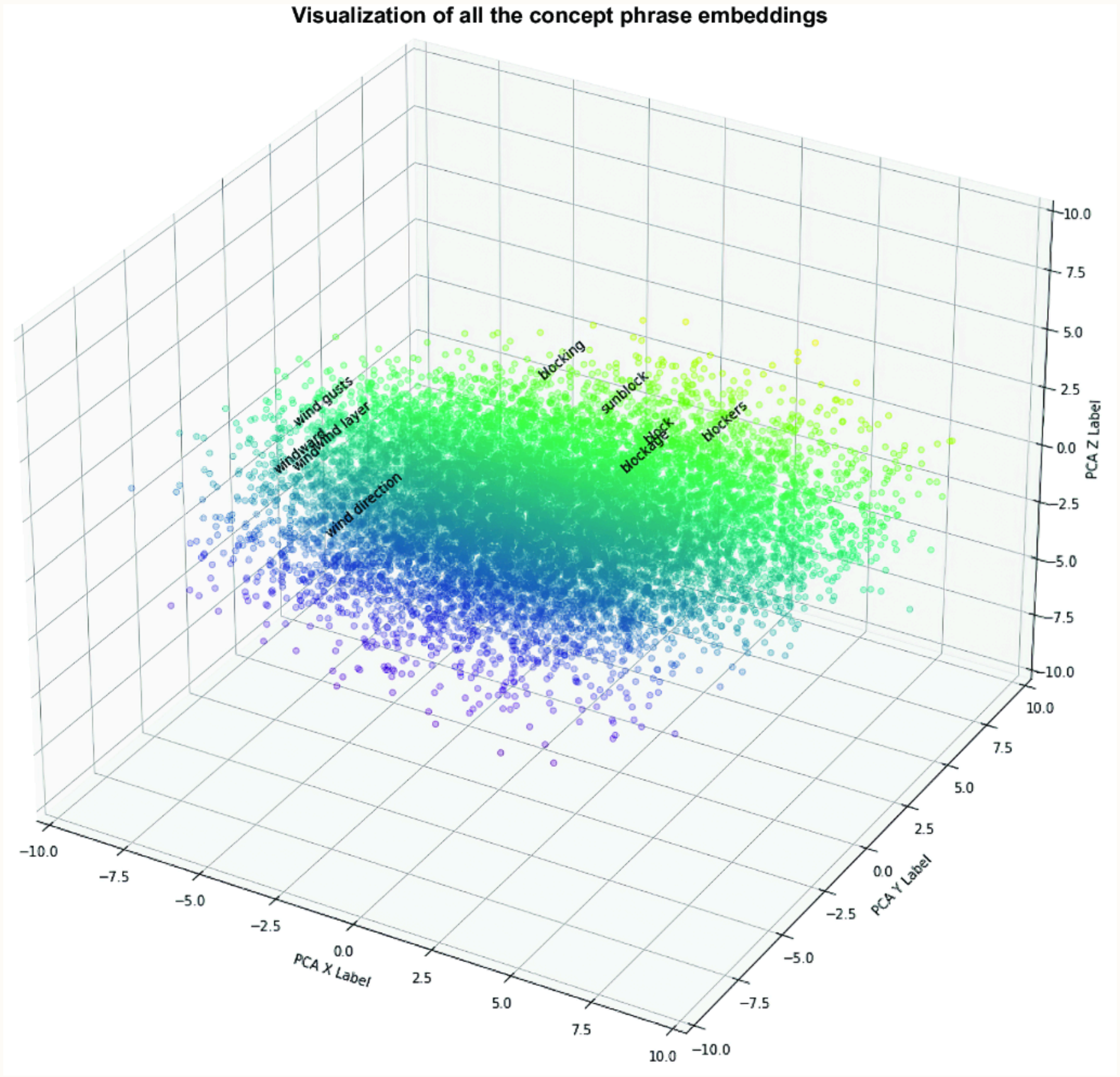


Figure 13.6 The vector space for the concept embeddings mapped to a 3D visualization

With the embeddings calculated from listing 13.11, we can now perform a massive comparison to see which terms are more closely related to one another. We will do this by calculating the cosine similarity—the dot product for each unit-normalized embedding related to every other unit-normalized embedding. Note that we’re limiting the number of embed-

dings that we're comparing in this example, because the number of calculations that are required to be performed grows exponentially as the number of embeddings increases. If you're not sure what we mean, let's do some quick math. Each embedding has a dimensionality of 768 floating point values. Comparing the top `250` embeddings all against each other results in `250` × `250` × `768 = 48,000,000` floating point calculations. Were we to compare the full list of 12,375 embeddings, that would be `12,375` × `12,375` × `768` `=` `117,612,000,000` floating point calculations. Not only would this be slow to process, but it would also take a very large amount of memory.
The following listing performs a brute-force comparison of the top 250 concepts, to assess how similarity scores are distributed.

Listing 13.12 Exploring similarity scores from the head of the vocabulary

```
normalized_embeddings = list(map(normalize_embedding, embeddings))
similarities = sentence_transformers.util.dot_score(  #1
                 normalized_embeddings[0:250],  #1
                 normalized_embeddings[0:250])  #1
comparisons = rank_similarities(phrases, similarities)  #2
display(HTML(comparisons[:10].to_html(index=False)))
```

#1 Finds the pairs with the highest dot product scores
#2 Ranks similarities as defined in listing 13.8

Response:

```
idx     score      phrase a   phrase b
31096   0.928151   protect    protection
13241   0.923570   climbing   climber
18096   0.878894   camp       camping
...
7354    0.782962   climb      climber
1027    0.770643   go         leave
4422    0.768611   keep       stay
```

As you can see in listing 13.12, the `scores` dataframe now holds a sorted list of all phrases compared to one another, with the most similar being "protect" and "protection" with a dot product similarity of `0.928`.

Note that the index of `250` is arbitrary and can be changed to larger values for more data to visualize. Remember what we learned in listing 13.7: using `n` concepts yields a tensor of `[n,` `n]` in shape. This yields a total of `250` × `250` = `62500` similarities for the example in listing 13.12.

The following listing plots the distribution of the top 250 concept comparison similarity scores.

Listing 13.13 The distribution of word similarities

```
from plotnine import *
candidate_synonyms = comparisons[comparisons["score"] > 0.0]
{
  ggplot(comparisons, aes("idx", "score")) +
    geom_violin(color="blue") +
    scale_y_continuous(limits=[-0.4, 1.0],
                       breaks=[-0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1.0])
}
```

The output of listing 13.13 is shown in figure 13.7, and the distribution clarifies the percentage of concepts that are most related to one another. You see that very few comparisons have a similarity score greater than `0.6`, and the vast majority have similarity scores less than that.

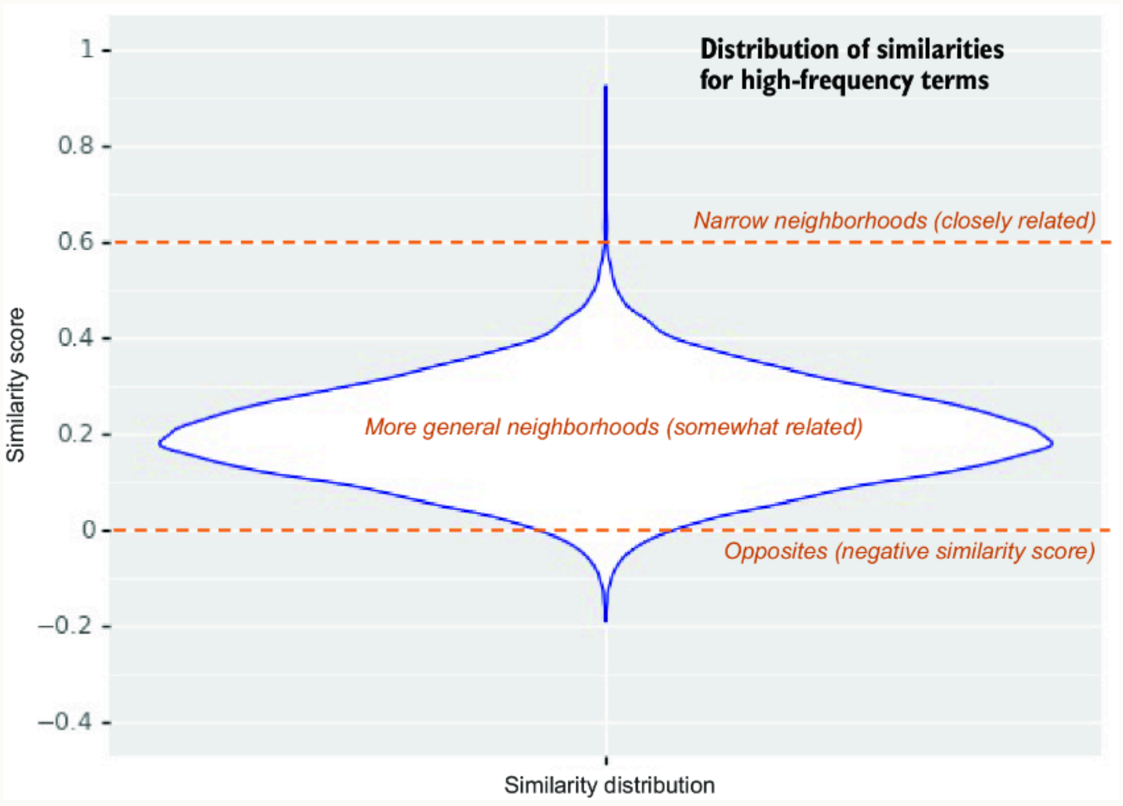


Figure 13.7 Distribution of how the top 250 concepts score with a dot product similarity when compared with each other. Note that very few comparisons result in a score higher than `0.6`, and most scores are less than `0.4` (a very low confidence).

We plotted the distribution of scores so we can assess them and use our intuition for choosing a baseline similarity threshold at query time (used later in listing 13.15). The visualization in figure 13.7 is very promising. Since most concepts are noted as dissimilar, we can reliably choose a high enough number as a threshold of quality suggestions (such as `0.6` in this example). When we're performing an autocomplete during search, we're only interested in seeing the top five to ten suggested terms, so this distribution shows we can do that reliably.

13.5.3 ANN search

Before implementing the working autocomplete, we have one more important problem to solve. The problem is that, at query time, we ideally don't want to compare each search

term to all 12,375 other terms. That would be inefficient and slow due to the dimensionality and computational overhead of using the `sentence_transformers .util.dot_score` function. Even if we were willing to calculate dot product similarities for all our documents, this would just get slower and slower as we scaled to millions of documents, so we ideally would only score documents that have a high chance of being similar.
We can accomplish this goal by performing what is known as an *approximate-nearest-neighbor* (ANN) search. ANN search will efficiently return the most closely related documents when given a vector, without the overhead of calculating embedding similarities across the entire corpus. ANN search is meant to trade some accuracy in exchange for improved logarithmic computational complexity, as well as memory and space efficiencies.

To implement our ANN search, we will be using an index-time strategy to store searchable content vectors ahead of time in a specialized data structure. Think of ANN search like an "inverted index" for dense vector search.

For our purposes, we will use *Hierarchical Navigable Small World* (HNSW) graphs to index and query our dense vectors. We'll also cover other approaches based on clustering and hashing, like product quantization and inverted file indexes (IVF), in section 13.7. HNSW is described in the abstract of its research paper, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," by Yu. A. Malkov and D.A. Yashunin (https://arxiv.org/abs/1603.09320):

> *We present a new approach for the approximate K-nearest neighbor search based on navigable small world graphs with controllable hierarchy (Hierarchical NSW, HNSW)…. Hierarchical NSW incrementally builds a multilayer structure consisting of hierarchical sets of proximity graphs (layers) for nested subsets of the stored elements.*

What this means is that HNSW will cluster similar vectors together as it builds the index. Navigable Small World graphs work by organizing data into neighborhoods and connecting the neighborhoods with probable relationship edges. When a dense vector representation is being indexed, the most appropriate neighborhood and its potential connections are identified and stored in the graph data structure.

DIFFERENT ANN APPROACHES

In this chapter, we use the HNSW algorithm for ANN search. HNSW provides a great balance between recall and query throughput and is currently (as of this writing) among the most popular ANN approaches. However, many other ANN approaches exist, including much simpler techniques like locality-sensitive hashing (LSH). LSH breaks the vector space into hash buckets (representing neighborhoods in the vector space) and encodes (hashes) each dense vector into one of those buckets. While recall is typically much higher for HNSW versus LSH, HNSW depends upon your data to generate the neighborhoods, and the neighborhoods can shift over time to better fit your data. LSH neighborhoods (hashes) are generated in a data-independent way, which can better meet some use cases requiring a priori sharding in distributed systems. We'll also cover other approaches based on clustering and hashing, like product quantization and inverted file indexes (IVF), in section 13.7. It may be worth your while to research different ANN algorithms to find the one that best suits your application.

When an HNSW search is initiated using a dense vector query, it finds the best cluster entry point for the query and searches for the closest neighbors. There are many other optimization techniques that HNSW implements, and we encourage you to read the paper if you want to learn more.

13.5.4 ANN index implementation

For our ANN search implementation, we'll start by using a library called the Non-Metric Space Library (NMSLIB). This library includes a canonical implementation of the *HNSW* algorithm.

We've chosen this library because not only is it fast, but it's also easy to use and requires very little code. Apache Lucene also includes a dense vector field type with native HNSW support, making the algorithm available in Solr, OpenSearch, Elasticsearch, and other Lucene-based engines like MongoDB Atlas Search. An implementation of HNSW is additionally available in other search engines, such as Vespa.ai, Weaviate, Milvus, and more.

NMSLIB is robust, well-tested, and widely used for ANN applications. NMSLIB is also appropriate for showing the simplicity of ANN search without getting into the details of the implementation. There are many other ANN libraries available, and we encourage you to investigate some of them listed on the excellent ANN Benchmarks site: **https://ann-benchmarks.com**.

To begin using NMSLIB, we simply need to import the library, initialize an index, add all of our embeddings to the index as a batch, and then commit. Autocomplete is an ideal use case when building an index in this way, because the vocabulary is rarely updated. Even though NMSLIB and other libraries may suffer from write-time performance in certain situations, this won't affect our read-heavy autocomplete application. From a practical standpoint, we can update our index offline as an evening or weekend job and deploy to production when appropriate.

The following listing creates our HNSW index from all 12,375 embeddings and then performs an example search for concepts similar to the term "bag".

Listing 13.14 ANN search using NMSLIB

```
import nmslib

concepts_index = nmslib.init(method="hnsw",  #1
                             space="negdotprod")  #1
normalized_embeddings = list(map(normalize_embedding, embeddings))
concepts_index.addDataPointBatch(normalized_embeddings)  #2
concepts_index.createIndex(print_progress=True)   #3

ids, _ = concepts_index.knnQuery( #4
           normalized_embeddings[25], k=10)  #4
matches = [labels[phrases[i]].lower() for i in ids] #5
display(matches)
```

#1 Initializes a new index, using an HNSW graph in the negdotprod metric space (distance function is –1 * dot_product)
#2 All the embeddings can be added in a single batch.
#3 Commits the index to memory. This must be done before you can query for nearest neighbors.
#4 Gets the top k nearest neighbors for the term query "bag" (embedding 25) in our embeddings
#5 Looks up the label for each term

Output:

```
['bag', 'bag ratings', 'bag cover', 'bag liner', 'garbage bags', 'wag bags',
 'bag cooking', 'airbag', 'paper bag', 'tea bags']
```

With the index created and committed, we ran a small example comparing the term "bag" and seeing what comes back. Interestingly, all these terms are hyponyms, which reveals another ideal result. We are interested in suggesting more precise terms to the user at autocomplete time. This has a higher likelihood of giving the user the chance to select the term most closely associated with their particular information need.

With our index confirmed working, we can now construct a straightforward query function that accepts any term and returns the top suggestions. SBERT has been trained using a technique that encodes similar terms to similar vector embeddings. Importantly, and in contrast to most lexical autocomplete implementations, this function accepts any query regardless of whether it's already in our dictionary. We first take the query and retrieve embeddings by passing the query through the same SBERT encoder we used to index our documents. With these embeddings, we access the nearest neighbors from the index. If the similarity score is greater than `0.75`, we count it as a match and include that as a suggestion. With this function, we can get suggestions for complete terms, such as "mountain hike", as well as prefixes, such as "dehyd".

Listing 13.15 shows our autocomplete `semantic_suggest` function implementation, which performs an ANN search for concepts. Our query might not be in the dictionary, but we can get the embeddings on demand. We will use the threshold `dist>=0.75` to only return similar terms for which we see a high confidence in similarity.

CHOOSE A GOOD SIMILARITY THRESHOLD

We arrived at the `0.75` threshold for listing 13.15 by looking at the distribution from figure 13.7. This should be further tuned by looking at the quality of results for your actual user queries.

NOTE This function may cause a CPU bottleneck in production, so we recommend measuring throughput at scale and adding hardware accordingly.

Listing 13.15 Encoding a query and returning the *k*-nearest-neighbor concepts

```
def embedding_search(index, query, phrases, k=20,  #1
                     min_similarity=0.75):  #1
  matches = []
  query_embedding = transformer.encode(query) #2
  query_embedding = normalize_embedding(query_embedding)
  ids, distances = index.knnQuery(query_embedding, k=k)
  for i in range(len(ids)):
    similarity = distances[i] * -1  #3
    if similarity >= min_similarity: #4
      matches.append((phrases[ids[i]], similarity))
  if not len(matches):
    matches.append((phrases[ids[1]], distances[1] * -1))  #5
  return matches

def semantic_suggest(prefix, phrases):
  matches = embedding_search(concepts_index, prefix, phrases)
  print_labels(prefix, matches)

semantic_suggest("mountain hike", phrases)
semantic_suggest("dehyd", phrases)
```

#1 We set k=20 for illustration purposes. In a production application, this would likely be set somewhere from 5 to 10.

#2 Gets the embeddings for the query

#3 Converts negative dot product distance into a positive dot product

#4 We're only returning the terms with 0.75 or higher similarity.

#5 No neighbors found! Returns just the original term

Response:

```
Results for: mountain hike

1.000 | mountain hike
0.975 | mountain hiking
0.847 | mountain trail
0.787 | mountain guide
0.779 | mountain terrain
0.775 | mountain climbing
0.768 | mountain ridge
0.754 | winter hike

Results for: "dehyd"

0.941 | dehydrate
0.931 | dehydration
0.852 | rehydration
...
0.812 | hydrate
0.788 | hydration pack
0.776 | hydration system
```

We’ve done it! We can now efficiently serve up a semantic autocomplete based on Transformer embeddings and approximate nearest-neighbor search.

Overall, the quality of results for many queries with this model is quite impressive. But beware, it’s extremely important to use a labeled dataset to measure success before deploying a solution like this to real customers. We’ll demonstrate this process of using labeled data to measure and improve relevance when implementing question-answering in chapter 14.

13.6 Semantic search with LLM embeddings

Using what we’ve learned so far, we’ll now take dense vector search to the next level: we are going to query the document embeddings with the query embeddings as a recall step at search time.

We specifically started with autocomplete as our first implementation, because it was helpful to understand the basics of language similarity. It is essential to develop a strong intuition about why things are similar or dissimilar in a vector space. Otherwise, you will endlessly chase recall problems when using embeddings. To build that intuition we started with matching and scoring basic concepts only a few words in length each.

With that understanding, we will now move on to comparing entire sentences. We’re going to perform a semantic search for titles. Remember that we’re searching on the Stack Exchange outdoors dataset, so the document titles are really the summaries of the questions being asked by the contributors. As a bonus, we can use the same implementation from the last section to search for question titles that are similar to one another.

This function will mostly be a repeat of the encoding and similarity functions from the previous section. The code in this section is even shorter, since we don’t need to extract concepts.

Here are the steps we'll follow:

1. Get the embeddings for all the titles in the outdoors dataset.
2. Create an NMSLIB index with the embeddings.
3. Get the embeddings for a query.
4. Search the NMSLIB index.
5. Show the nearest-neighbor titles.

13.6.1 Getting titles and their embeddings

Our NMSLIB index will be made up of title embeddings. We're using the exact same function from the previous autocomplete example, but instead of transforming concepts, we're now transforming the titles of all the questions the outdoors community asked. The following listing shows the process of encoding the titles into embeddings.

Listing 13.16 Encoding the titles into embeddings

```
outdoors_dataframe = load_dataframe("data/outdoors/posts.csv")
titles = outdoors_dataframe.rdd.map(lambda x: x.title).collect()  #1
titles = list(filter(None, titles))  #1
embeddings = get_embeddings(titles, cache_name)  #2

print(f"Number of embeddings: {len(embeddings)}")
print(f"Dimensions per embedding: {len(embeddings[0])}")
```

#1 Gets the titles for every question in the outdoors corpus
#2 Gets the embeddings for the titles (this takes a little while on the first run, until it is cached)

Response:

```
Number of embeddings: 5331
Dimensions per embedding: 768
```

We have encoded 5,331 titles into embeddings, and figure 13.8 plots the title embedding similarity distribution.

Compare figure 13.8 to the concept similarity distributions from figure 13.7. Note the slightly different shape and score distributions, due to the difference between titles and concepts. Figure 13.7 has a longer "needle" on top. This is because titles are more specific, and therefore will relate differently than broader noun and verb phrases.

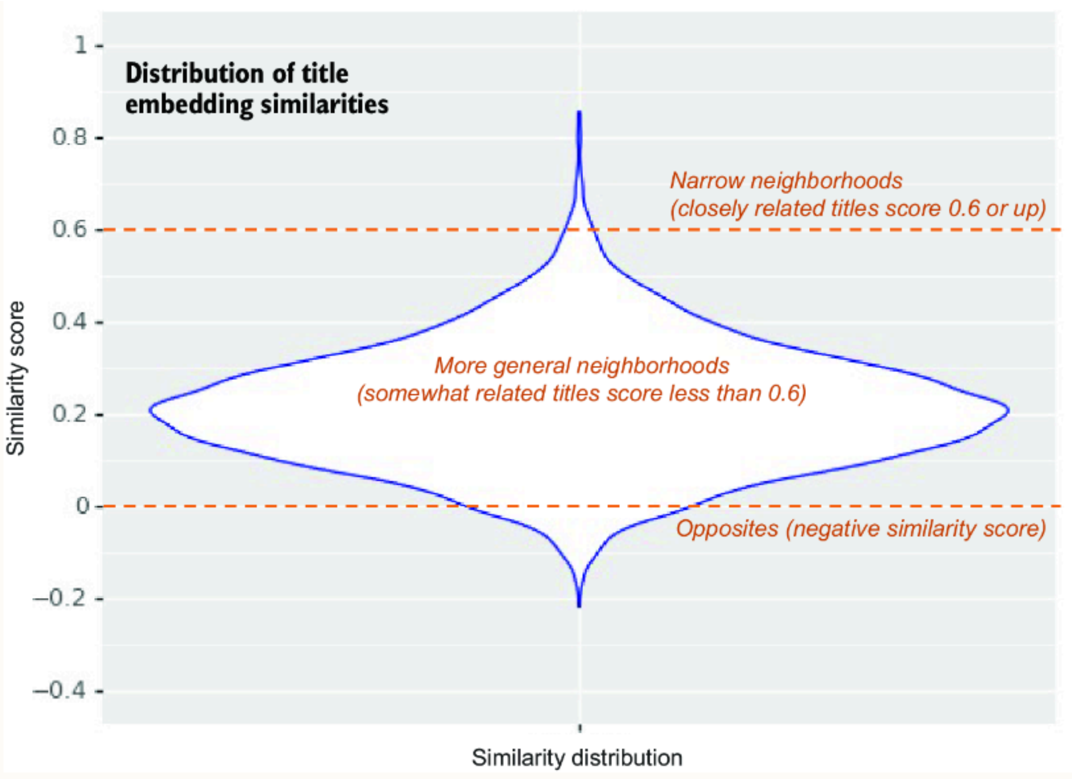


Figure 13.8 The distribution of similarity scores when comparing all title embeddings to each another

13.6.2 Creating and searching the nearest-neighbor index

Now that we have generated the embeddings for all the question titles in the corpus, we can easily create the nearest-neighbor index.

Listing 13.17 Creating the ANN title embeddings index

```
import nmslib
titles_index = nmslib.init(method="hnsw", space="negdotprod")
normalized_embeddings = list(map(normalize_embedding, embeddings))
titles_index.addDataPointBatch(normalized_embeddings)
titles_index.createIndex(print_progress=True)
```

With our newly created index, searching is easy! Listing 13.18 shows the new `semantic_search` function, which implements ANN search for question titles given a query. This is very similar to the `semantic_suggest` from listing 13.15 that we implemented for auto-complete—the main difference is that here the underlying embedding index is comprised of `title` content instead of concepts extracted from the `body` content.

Listing 13.18 Performing a semantic search for titles

```
def semantic_search(query, phrases):
  results = embedding_search(titles_index, query, phrases,  #1
                             k=5, min_similarity=0.6)  #1
  print_labels(query, results)

semantic_search("mountain hike", titles)
```

#1 embedding_search from listing 13.15

Response:

```
Results for: mountain hike

0.723 | How is elevation gain and change measured for hiking trails?
0.715 | How do I Plan a Hiking Trip to Rocky Mountain National Park, CO
0.698 | Hints for hiking the west highland way
0.694 | New Hampshire A.T. Section Hike in May? Logistics and Trail Condi...
0.678 | Long distance hiking trail markings in North America or parts the...
```

Now let’s take a moment and reflect on these results. Are they all relevant? Yes—they are all absolutely questions related to the query `mountain hike`. But, and this is very important, are they the *most* relevant documents? We don’t know! The reason we don’t know is that `mountain hike` does not provide much context at all. So, while the titles are all semantically similar to the query, we don’t have enough information to know if they are the documents we should surface for the user.

That said, it is clear that this embedding-based approach to search brings interesting new capabilities to our matching and ranking toolbox, providing the ability to conceptually relate results. Whether those results are better or not depends on the context.

Thus far, we’ve implemented dense vector search in this chapter by hand, relying on the NMSLIB library to do the heavy lifting, but otherwise showing you how to build a dense vector index with ANN (HNSW) support and query it. We’ve done this intentionally to help you understand the inner workings of dense vector search. In your production system, however, you’re likely going to use your search engine’s built-in support for dense vector search. In the next listing, we switch over to using our `collection` interface to implement the same semantic search functionality using your configured search engine or vector database.

Listing 13.19 Performing vector search with the configured search engine

```
def display_results(query, search_results):
  print_labels(query, [(d["title"], d["score"])
                       for d in search_results])

def index_outdoor_title_embeddings(): #1
  create_view_from_collection(engine.get_collection("outdoors"),
                              "outdoors")
  outdoors_dataframe = spark.sql("""SELECT id, title FROM outdoors
                                  WHERE title IS NOT NULL""")
  ids = outdoors_dataframe.rdd.map(lambda x: x.id).collect()
  titles = outdoors_dataframe.rdd.map(lambda x: x.title).collect()
  embeddings = list(
    map(normalize_embedding,  #2
        get_embeddings(titles, cache_name)))  #2
  embeddings_dataframe = spark.createDataFrame(
    zip(ids, titles, embeddings),
    schema=["id", "title", "title_embedding"])

  collection = engine.create_collection("outdoors_with_embeddings")
  collection.write(embeddings_dataframe)
  return collection

def semantic_search_with_engine(collection, query, limit=10):  #3
  query_vector = transformer.encode(query) #4
  query_vector = normalize_embedding(query_vector)  #4
  request = {"query": query_vector,
             "query_fields": ["title_embedding"],
             "return_fields": ["title", "score", "title_embedding"],
             "quantization_size": "FLOAT32",  #5
             "limit": limit}
  response = collection.search(**request)
  return response["docs"]

embeddings_collection = index_outdoor_title_embeddings()

query = "mountain hike"
search_results = semantic_search_with_engine(embeddings_collection, query)
display_results(query, search_results)
```

#1 Builds a new collection with documents containing their title and embeddings
#2 Calculates normalized embeddings for all documents
#3 Returns documents by searching with a query against title_embedding
#4 The string query is encoded and normalized then built into a vector search request.
#5 The quantization size of the embeddings, which in this case is 32 bits

Response:

```
0.723 | How is elevation gain and change measured for hiking trails?
0.715 | How do I Plan a Hiking Trip to Rocky Mountain National Park, CO
0.698 | Hints for hiking the west highland way
0.694 | New Hampshire A.T. Section Hike in May? Logistics and Trail Condi...
0.678 | Long distance hiking trail markings in North America or parts the...
```

Every search engine or vector database has its own unique APIs for implementing keyword search, vector search, hybrid keyword and vector search, and other capabilities. The `semantic_search_with_engine` function in listing 13.19 demonstrates using an engine-agnostic interface to query your configured search engine, though you may find it more powerful to perform certain operations using your engine's APIs directly for more advanced use cases.

We used NMSLIB earlier in this chapter to help you better understand the inner workings of dense vector search. Unless you're doing something very custom, though, you likely will want to use your search engine's built-in, scalable support for dense vector search as opposed to implementing it by hand locally with a library like NMSLIB, FAISS (which we'll introduce later in the chapter), or NumPy. You'll note that listing 13.19 returns the exact same results from your engine as the NMSLIB implementation returned in listing 13.18.

RERANKING RESULTS FOUND WITH DENSE VECTOR SIMILARITY

In listing 13.18, we chose the default `min_similarity` threshold to require a similarity score of `0.6` or greater. Examine the title similarity distributions in figure 13.8—would you change this number to be different than `0.6`?

You could set `min_similarity` to a value lower than `0.6` to potentially increase recall and change `k` to be a value higher than `5` as a rerank window size (for example, `250`). Then, using this larger result set, you could perform a rerank using dot product similarity. Using what you learned in chapters 10–12, you could also incorporate the dense vector similarity as a feature (potentially among many) in a more sophisticated learning-to-rank model.

Similarity scoring of embeddings is one of many features in a mature AI-powered search stack. This similarity will be used alongside personalization, learning to rank, and knowledge graphs for a robust search experience. Nearest-neighbor-based dense vector search is rapidly growing in popularity and is likely to supplant Boolean matching with BM25 ranking at some point as the most common retrieval and ranking technique for searching unstructured text. The two approaches—dense vector search and lexical search—are complementary, however, and hybrid approaches combining the two usually work even better.

13.7 Quantization and representation learning for more efficient vector search

In the last section, we introduced two concepts commonly used to speed up dense vector search: ANN search and reranking. Conceptually, ANN is a way to reduce the number of vector similarity calculations necessary at query time by efficiently locating and filtering to the top vectors that are the most likely to be similar to the query vector. Because ANN search is an approximation of the best results, it's common to rerank those top potential results using more precise (and computationally expensive) vector similarity calculations to get recall and relevance back on par with a non-ANN-optimized search.

The computation time and amount of memory required to represent and perform similarity calculations on vectors is directly related to the size of the vectors being searched. In this section, we'll introduce a few additional techniques for improving the efficiency of vector search:

- Scalar quantization
- Binary quantization
- Product quantization
- Matryoshka Representation Learning

Quantization is a technique used to reduce the memory footprint and computational complexity of numeric data representations, such as embedding vectors. In the context of embeddings, quantization involves compressing them by reducing the number of bits used to represent the features of the vector. Embeddings are typically represented as floating point numbers (floats), which are 32 bits (or 4 bytes) in size by default. If a typical vector embedding has 1,024 dimensions, this translates into 1,024 x 4 bytes, or 4,096 bytes (4 KB) per vector. If you have a large number of vectors to store and search, this can quickly add up to a significant amount of memory and computational overhead.

Quantizing embeddings allows you to trade off some ideally small amount of recall for significant improvements in storage efficiency and query speed, which is crucial for large-scale search systems. For example, you can reduce the memory usage of a vector of 32-bit floats (Float32) by 75% by converting each feature to an 8-bit integer (Int8). Compressing the individual numerical values (scalars) of each dimension like this is known as *scalar quantization*. This can often be done without significantly affecting recall, and it can be especially useful when you have a large number of vectors to store and search. You can even quantize each feature down to a single bit—a technique known as *binary quantization*—and still maintain a relatively high level of recall if it's combined with a reranking step using a higher-precision vector on the top-*N* results. Figure 13.9 visually demonstrates the concepts of scalar quantization and binary quantization using a vector containing an image of the cover of this book (assume the dimensions of the vector represent pixels in the image).

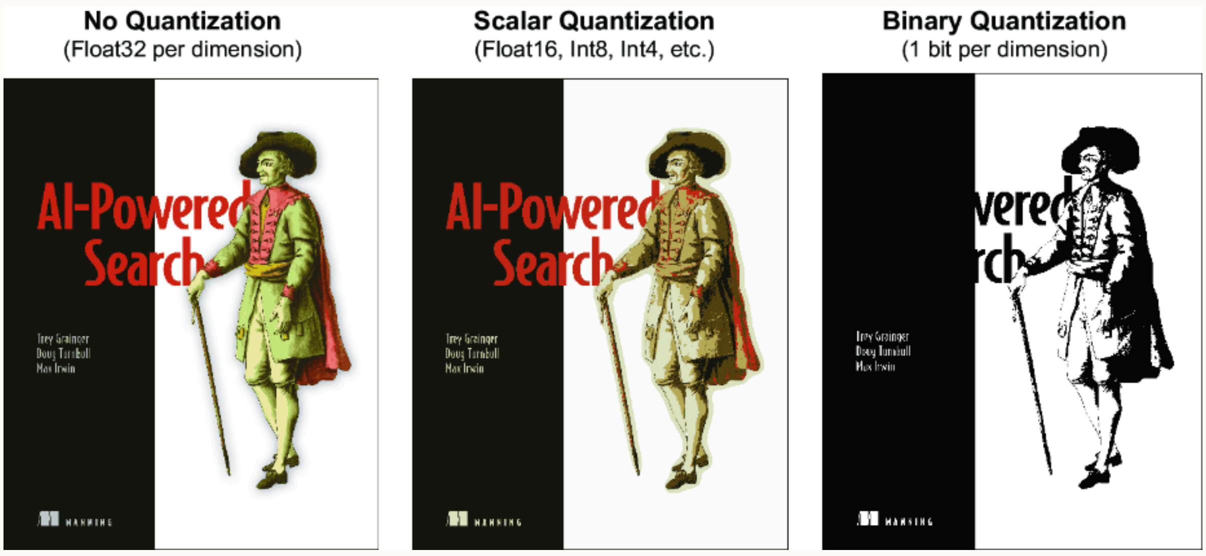


Figure 13.9 Quantizing data: full precision, reduced scalar precision, and binary precision

In the figure, you can see the original image (no quantization), a scalar quantized image (using a reduced color palette that maps to similar ranges but with less colors/precision), and a binary quantized image (where each pixel is either black or white). You'll notice that

the scalar quantized image still retains most of the important detail from the original image, and the binary quantized version is still clearly recognizable, albeit losing some important data (some of the characters in the title and the colors).
In this section, we'll cover scalar quantization, binary quantization, and a third type of quantization called product quantization. We'll also introduce a multi-tiered embedding approach known as Matryoshka Representation Learning, which can be used to dynamically switch the precision levels of already-generated embeddings without the need for additional quantization or retraining.

13.7.1 Scalar quantization

Scalar quantization is the simplest form of quantization, where each value in the embedding vector is independently mapped to a lower-precision representation. Consider the following two vectors:

```
[ -1.2345679,   2.2345679, 100.45679 ]  #4 bytes = 32 bits
[ -1.234,       2.234,     100.44 ]     #2 bytes = 16 bits
```

The first vector is a 32-bit float representation, and the second is a 16-bit float representation, each rounded to the maximum reliable precision. The second vector requires 50% less memory (2 bytes versus 4 bytes), all while still representing approximately the same values, just with less precision.

This reduced precision is a simple example of scalar quantization, taking higher-precision values and mapping them into lower-precision representations requiring less memory to store and less computation to process.

But what if we wanted to compress our vectors even further to a single byte (or even a few bits)—can we still pull this off and preserve most of our recall? The answer is yes, and we commonly accomplish this by mapping the range of float values into an Int8, as shown in figure 13.10.

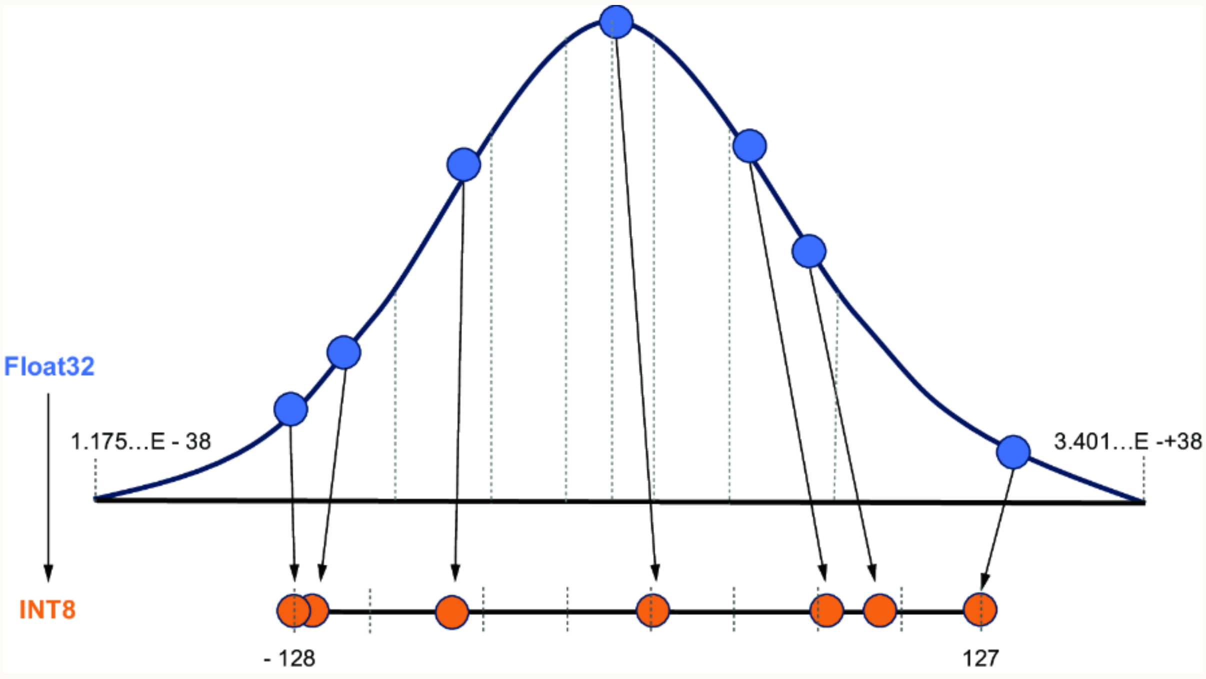


Figure 13.10 Scalar quantization from Float32 to Int8

In the figure, the curve at the top represents the distribution of values in the floating point range. Since we are quantizing from 32 bits to 8 bits, we map the range of the Float32 values into the smaller range of –128 to 127 (or 0 to 255 if using an unsigned integer). Depending on the quantization algorithm being used, attempts are often made to utilize the new ranges as fully as possible by *clamping* the values (limiting the range to the minimum and maximum), as well as by utilizing the density of values in the original vector to map more evenly into the new, quantized range.

Let's implement a scalar quantization example to see what effect this optimization yields on index size, recall, and search speed. Multiple libraries exist for performing scalar quantization. You can use the `sentence_transformers.quantization` module, or your search engine or language model may have its own quantization implementations built in. We are going to utilize the FAISS library for our quantized indexes, and a combination for the `sentence_transformers` library and FAISS for each of our quantization examples. FAISS (Facebook AI Similarity Search) is an open source library designed for efficient similarity search and clustering of dense vectors. It's similar to the NMSLIB library we used earlier in the chapter for semantic search, but it has some additional features, including built-in support for quantization. FAISS is widely used in production systems for dense vector search, and it's a great choice for implementing quantized indexes.

We'll look at an example later of how to run a search using the `collection` abstraction for your chosen search engine, but since not every engine has support for all quantization modes, and since each engine has different overhead and performance characteristics, we'll use FAISS for our benchmarking.

Let's now build a FAISS index with full-precision Float32 embeddings. We'll then use this as our baseline versus various quantized embedding indexes to compare index sizes, search speeds, and recall rates. Listing 13.20 shows the code to create the full-precision Float32 embeddings and index them into a FAISS index.

Listing 13.20 Indexing full-precision embeddings using FAISS

```
from sentence_transformers.quantization import quantize_embeddings

model = SentenceTransformer(
          "mixedbread-ai/mxbai-embed-large-v1",  #1
          similarity_fn_name=SimilarityFunction.DOT_PRODUCT,
          truncate_dim=1024)  #2

def index_full_precision_embeddings(doc_embeddings, name):
  index = faiss.IndexFlatIP(doc_embeddings.shape[1])  #3
  index.add(doc_embeddings)  #4
  faiss.write_index(index, name)  #5
  return index

def get_outdoors_embeddings(model):
  outdoors_dataframe = load_dataframe("data/outdoors/posts.csv")
  post_texts = [post["title"] + " " + post["body"]
                for post in outdoors_dataframe.collect()]
  return numpy.array(
    get_embeddings(post_texts, model, "outdoors_mrl_normed"))

doc_embeddings = get_outdoors_embeddings(model)  #6
full_index = index_full_precision_embeddings(  #7
               doc_embeddings, "full_embeddings")  #7
```

#1 This model creates embeddings supporting all upcoming optimization techniques.
#2 The original embeddings will have 1,024 dimensions.
#3 IndexFlatIP is a simple, unoptimized index supporting different embedding formats.
#4 Adds documents to the index
#5 Writes the index to disk
#6 Generates embeddings for the outdoors dataset
#7 Creates a full-precision (Float32) FAISS index

In this listing, we calculate embeddings for the outdoors dataset using the `mixedbread-ai/mxbai-embed-large-v1` model, which produces high-quality embeddings that work well with all our quantization techniques and also supports Matryoshka Representation Learning, which we'll explore later in the chapter. We then index those embeddings to a full-precision (Float32) FAISS index, which we'll use soon as our baseline for benchmarking the performance of various quantization techniques.

For our benchmarks, we'll also need to encode some test queries into embeddings, which is shown in listing 13.21.

Listing 13.21 Generating query embeddings and full-index benchmarks

```
def get_test_queries():
  return ["tent poles", "hiking trails", "mountain forests",
          "white water", "best waterfalls", "mountain biking",
          "snowboarding slopes", "bungee jumping", "public parks"]

queries = get_test_queries()  #1
query_embeddings = model.encode(queries,  #2
                     convert_to_numpy=True,  #2
                     normalize_embeddings=True)  #2

full_results = time_and_execute_search( #3
                 full_index, "full_embeddings",  #3
                 query_embeddings, k=25)  #3
display_statistics(full_results)  #4
```

#1 Gets test queries for benchmarking
#2 Generates embeddings for each query
#3 Generates search time, index size, and recall statistics for the full-precision (Float32) index
#4 Displays the benchmarking stats

Output:

```
full_embeddings search took: 7.621 ms
full_embeddings index size: 75.6 MB
Recall: 1.0
```

In the preceding listing, we define a list of queries we'll use to benchmark our full-precision index against various vector search optimization strategies. We then call a `time_and_execute_search` function (omitted for brevity) on the full-precision index from listing 13.20 and then pass the results to a `display_statistics` function (also omitted for brevity), which displays the search time, index size, and recall statistics.

This provides a baseline for comparison with our upcoming quantized (or otherwise optimized) indexes. Listing 13.22 shows the implementation of two additional functions we'll use to compare results from other indexing strategies: an `evaluate_search` function and an `evaluate_rerank_search` function.

Listing 13.22 Functions to benchmark optimized search approaches

```
def evaluate_search(full_index, optimized_index,  #1
                    optimized_index_name,  #1
                    query_embeddings,  #1
                    optimized_query_embeddings,  #1
                    k=25, display=True, log=False):  #1
  full_results = time_and_execute_search(  #2
                   full_index, "full_embeddings",  #2
                   query_embeddings, k=k)  #2
  optimized_results = time_and_execute_search(  #2
                        optimized_index,  #2
                        optimized_index_name,  #2
                        optimized_query_embeddings, k=k)  #2
  if display:
    display_statistics(optimized_results, full_results)
  return optimized_results, full_results

def evaluate_rerank_search(full_index, optimized_index,  #3
                           query_embeddings,  #3
                           optimized_embeddings,  #3
                           k=50, limit=25):  #3
  results, full_results = evaluate_search(
                            full_index,
                            optimized_index, None,
                            query_embeddings,
                            optimized_embeddings,
                            display=False, k=k)

  doc_embeddings = get_outdoors_embeddings(model)  #4
  rescore_scores, rescore_ids = [], []
  for i in range(len(results["results"])):
    embedding_ids = results["faiss_ids"][i]
    top_k_embeddings = [doc_embeddings[id]  #5
                        for id in embedding_ids]  #5
    query_embedding = query_embeddings[i] #5
    scores = query_embedding @ \ #5
             numpy.array(top_k_embeddings).T  #5
    indices = scores.argsort()[::-1][:limit]  #6
    top_k_indices = embedding_ids[indices]  #6
    top_k_scores = scores[indices]  #6
    rescore_scores.append(top_k_scores)  #6
    rescore_ids.append(top_k_indices)  #6

  results = generate_search_results(rescore_scores, rescore_ids)  #7
  recall = calculate_recall(full_results["results"], results)  #8
  print(f"Reranked recall: {recall}")
```

#1 Brings back the top 25 results from each index and compares them

#2 Calculates query speed, index size, and recall for optimized versus full-precision index

#3 Same as evaluate_search, but over-requests to k=50 results (by default) and reranks those using full-precision embeddings

#4 Generates embeddings for each doc in the outdoors dataset

#5 Performs a dot product between the query embedding and the top-k embeddings

#6 Sorts the results by dot product score to rerank them

#7 Helper function (omitted) that combines IDs and scores with other document fields to return

#8 Calculates the recall of results after reranking versus the full-precision index

The `evaluate_search` function internally calls the `time_and_execute_search` function on both the full-precision index and the quantized index, and it passes the results to a `display_statistics` function to compare and display the search time, index size, and recall statistics.

The `evaluate_rerank_search` function then calculates recall again after reranking the top-*N* results from the quantized index using full-precision embeddings. While quantization can drastically reduce memory and search times, we'll also see that it reduces recall, meaning that some of the results that *should* be returned are not. But by over-requesting and reranking just the top-*N* results using full-precision embeddings (typically loaded from disk after the quantized search, not kept in memory with the index), we can recapture most of that lost recall.

We'll show both the quantized recall and the reranked quantized recall in each subsequent listing to demonstrate the trade-offs between full-precision searches, quantized searches, and reranked quantized searches. For our first quantization example, let's implement Int8 scalar quantization.

Listing 13.23 Creating an Int8 quantized embeddings index using FAISS

```
def index_int8_embeddings(doc_embeddings, name):
  int8_embeddings = quantize_embeddings(  #1
                    doc_embeddings, precision="int8")  #1
  print("Int8 embeddings shape:", int8_embeddings.shape)
  index = faiss.IndexFlatIP(int8_embeddings.shape[1]) #2
  index.add(int8_embeddings)  #3
  faiss.write_index(index, name) #4
  return index

int8_index_name = "int8_embeddings"
int8_index = index_int8_embeddings(doc_embeddings, int8_index_name)

quantized_queries = quantize_embeddings(  #5
  query_embeddings,  #5
  calibration_embeddings=doc_embeddings,  #5
  precision="int8")  #5

evaluate_search(full_index, int8_index,  #6
                int8_index_name, query_embeddings,  #6
                quantized_queries)  #6
evaluate_rerank_search(full_index, int8_index,  #7
          query_embeddings, quantized_queries)  #7
```

#1 Quantizes the doc embeddings to Int8 precision

#2 Creates an index configured to expect the shape of the embeddings

#3 Adds the quantized embeddings to the index

#4 Saves the index to disk so we can measure its size

#5 Quantizes the query embeddings to Int8 precision

#6 Performs benchmarks for search time, index size, and recall

#7 Performs benchmarks again allowing reranking of top results with full-precision embeddings

Output:

```
Int8 embeddings shape: (18456, 1024)
int8_embeddings search took: 9.070 ms (38.65% improvement)
int8_embeddings index size: 18.91 MB (74.99% improvement)
Recall: 0.9289
Reranked recall: 1.0
```

In listing 13.23's output, we see that the Int8 quantized index is 75% smaller than the full-precision index, which makes sense given that we've reduced from Float32 to Int8 precision, which is an ~75% reduction from 32 bits to 8 bits. Likewise, we see that the total search time improved due to lower precision numbers being more efficient to process (at least on some systems). Note that search speed will vary *considerably* on all these benchmarks from system to system and run to run, but the index size benchmark should always be the same. Also note that we have a fairly small number of documents in our index, and query speed improvements from quantization are likely to become more pronounced as the number of documents increases.

The most important numbers, however, are the recall numbers. The Int8 quantized search maintained a 92.89% recall rate. This means that we achieved an ~75% reduction in index size and a significant improvement in search speed with only 7.11% of the top *N*=25 results missing in the quantized search. Just as the middle image of this book's cover in figure 13.9 maintained the vast majority of the important details of the original image, we were likewise able to retain high fidelity to our quantized embeddings in listing 13.23 when using Int8 quantization.

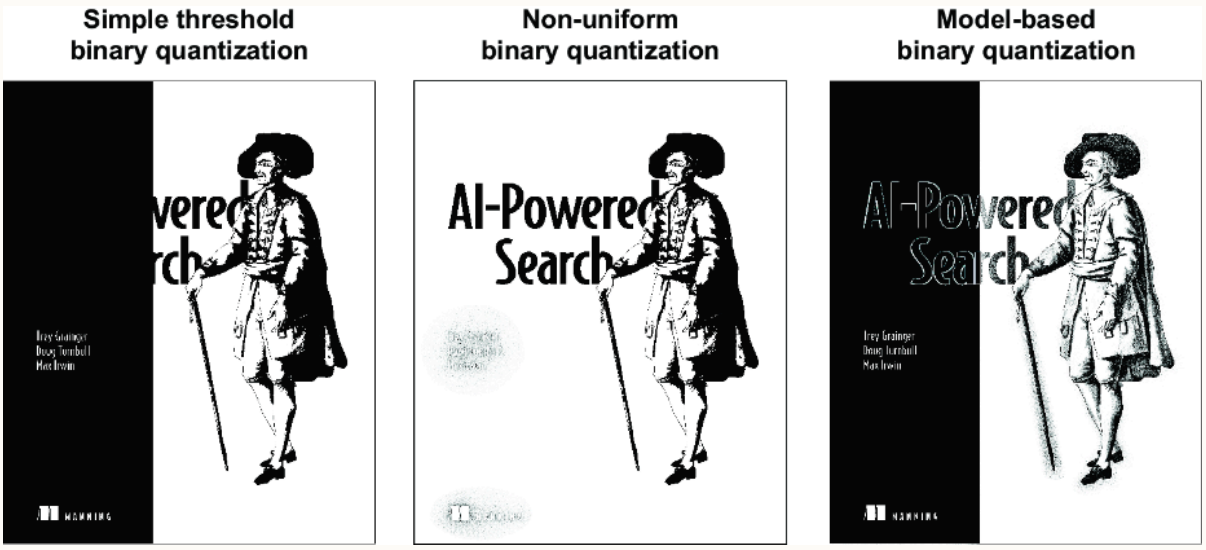
The `Reranked recall` value of 1.0 further indicates that by just requesting the top *N*=50 results and reranking them using the original full-precision embeddings, we get back to 100% recall. This is a common pattern when using quantization in dense vector search: perform an initial search that over-requests results using a quantized index (for a significant memory and speed improvement) and then rerank the top *N* results using higher-precision embeddings (usually pulled from disk so they don't affect your index and memory requirements) to recapture the lost recall and ranking precision. While these improvements are impressive, we could continue compressing even further, using 4 bits (Int4) or less. In the next section, we'll see what happens when we compress each dimension all the way down to a single bit!

13.7.2 Binary quantization

Binary quantization is an extreme form of quantization where each value in the embedding vector is represented by a single bit, reducing the values to `0` or `1`. This method is akin to converting an image to black and white only (just one shade of black, not grayscale), like in the example on the right in figure 13.9.

Quantizing every feature into a single bit can be done using a simple threshold to assign a bit of `0` for any feature less than or equal to `0` and a value of `1` for any feature greater than `1.0`. This works well if feature values across embeddings have a uniform distribution between positive and negative values. If feature values are not uniformly distributed across documents, however, it can be helpful to use the median value for each feature, or some similar threshold for assigning `0` versus `1` as the binary quantized value, in order to more evenly distribute the values.

Figure 13.11 illustrates the result of binary quantization, where only the most essential information is preserved, akin to the black-and-white version of the cover of this book. The far left image demonstrates using a simple threshold, the middle image represents using a non-uniform threshold that assigns values based on the relative distribution of values for each feature in the original embeddings, and the far right image shows an optimal learned binary representation directly from the Transformer encoder model. This last "model-based" binary quantization looks the best because the choice of values takes into account the context of the entire image when generating the binary representation, not just the individual features. This allows for a much more meaningful and accurate representation of the original embedding to be encoded into the available binary quantized feature values based on how the model has been trained to understand the image as a whole. We won't demonstrate an example of doing model-based binary quantization, since we're trying to benchmark recall for the same embeddings at different levels of quantization, but it's worth keeping in mind that if your model supports model-based binary quantization, it will generally provide better results than performing binary quantization on features after the model has already encoded and returned them at a higher level of precision.


Simple threshold binary quantization
Non-uniform binary quantization
Model-based binary quantization
AI-Powered Search
Trey Grainger
Doug Turnbull
Max Irwin
MANNING
Figure 13.11 Different binary quantization techniques

Let's now implement binary quantization using FAISS and benchmark the results.

Listing 13.24 Indexing and benchmarking binary quantized embeddings

```
def index_binary_embeddings(doc_embeddings,
                            binary_index_name):
  binary_embeddings = quantize_embeddings( #1
    doc_embeddings,  #1
    precision="binary").astype(numpy.uint8)  #1
  print("Binary embeddings shape:", binary_embeddings.shape)
  index = faiss.IndexBinaryFlat(  #2
    binary_embeddings.shape[1] * 8)  #2
  index.add(binary_embeddings)  #3
  faiss.write_index_binary(index, binary_index_name)  #4
  return index

binary_index_name = "binary_embeddings"
binary_index = index_binary_embeddings(
  doc_embeddings, binary_index_name)

quantized_queries = quantize_embeddings(  #5
  query_embeddings,  #5
  calibration_embeddings=doc_embeddings, #6
  precision="binary").astype(numpy.uint8)  #7

evaluate_search(full_index, binary_index,  #8
  binary_index_name,  #8
  query_embeddings, quantized_queries)  #8
evaluate_rerank_search(full_index, binary_index,  #8
  query_embeddings, quantized_queries)  #8
```

#1 Quantizes the doc embeddings to binary (1 bit per dimension)
#2 Creates the binary embeddings index
#3 Adds all the doc embeddings to the index
#4 Writes the index to disk
#5 Quantizes the query embeddings to binary
#6 The doc embeddings are provided to the quantizer to calibrate the best thresholds for assigning 0 or 1.
#7 Saves every 8 dimensions as 1 byte, encoded as unsigned Int8
#8 Performs benchmarks with and without reranking versus the full-precision index

Output:

```
Binary embeddings shape: (18456, 128)
binary_embeddings search took: 1.232 ms (83.38% improvement)
binary_embeddings index size: 2.36 MB (96.87% improvement)
Recall: 0.6044
Reranked recall: 1.0
```

While it should come as no surprise that binary quantization reduces the index size and memory requirements by 96.87% (from 32 bits to 1 bit), it is mind-boggling that we were able to retain 60.44% of the recall with just a single bit per embedding feature. While 60% recall isn't necessarily sufficient for many search use cases, note that when we over-requested by a factor of 2 (`N=50` to find the top 25) and reranked, we were able to get recall back to 100% for our test queries. Across a larger dataset and number of queries, you won't likely be able to maintain 100% recall, but even approaching full recall with just a single bit per feature for your initial search is an impressive feat.

With the ability to achieve such extreme compression while still getting back to nearly 100% recall by over-requesting and reranking, binary quantization pushes the limits of what's possible with quantization. You might think that a single bit per dimension would be the limit for how far we can quantize our embeddings, but in the next section we'll introduce a technique that allows us to compress even further—product quantization.

13.7.3 Product quantization

While scalar quantization and binary quantization focus on reducing the precision of the individual features of an embedding, product quantization (PQ) instead focuses on quantizing the entire embedding down to a more memory-efficient representation. It allows for even deeper compression than binary quantization, making PQ particularly beneficial for large-scale search applications with a high number of embedding dimensions.

Imagine you have a long sentence and you split it into shorter phrases; it's easier to handle and process each phrase independently. Similarly, PQ divides the vector space into smaller regions (subvectors) and quantizes each subvector individually. PQ then clusters each of the subvector regions to find a set of cluster centroids and finally assigns one centroid ID to each subvector for each document. This list of centroid IDs (one per subvector) for each document is called a PQ code, and it is the quantized representation of the document's embedding. Figure 13.12 demonstrates the PQ process.

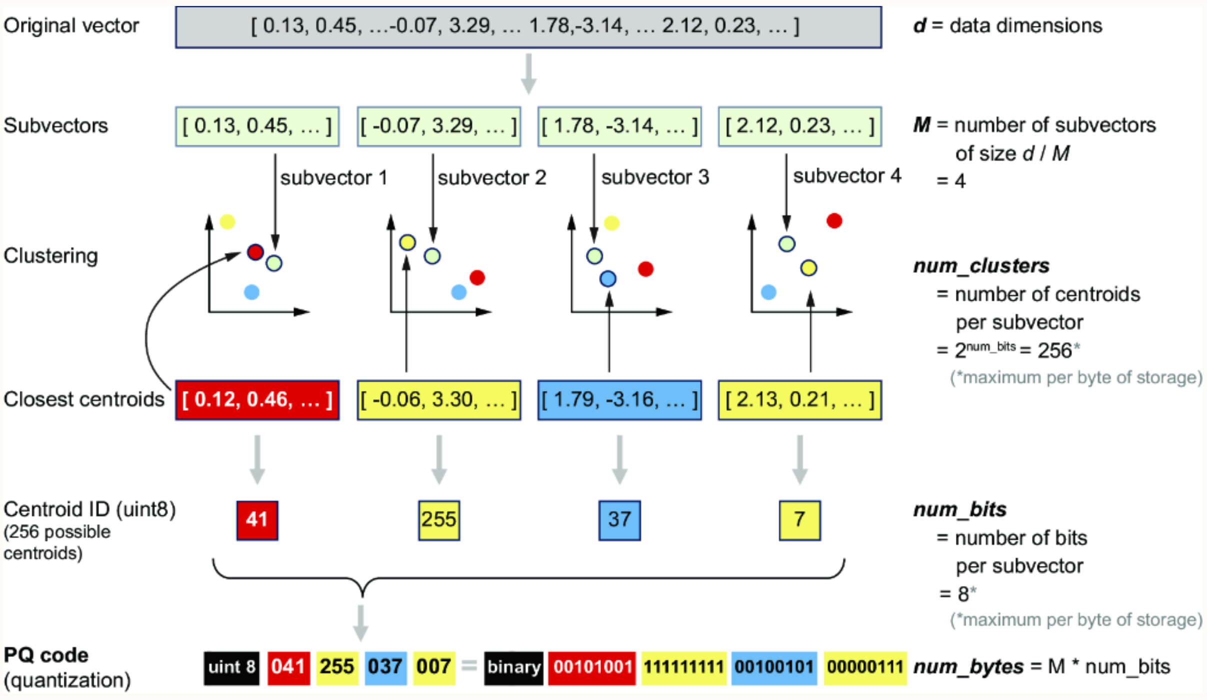


Figure 13.12 The product quantization process

The quantization process starts with the top layer of the figure and progresses sequentially down. First, the original vector space of `d` dimensions is divided into `M` subvector spaces. For each of the `M` subvector spaces, the corresponding subvector is partitioned from each original embedding. Clustering (usually using a *k*-means algorithm) is then performed on all the subvectors within each subspace to create a list of `num_clusters` clusters identified by their centroid vector within the subspace. Each centroid is then assigned an ID, which is recorded in a *codebook* containing a mapping of each centroid ID to its full subvectors. Finally, the centroid IDs for every subspace are concatenated together per document to create the *PQ code*, which is the official quantized representation of each document. The documents' PQ codes are then indexed, and at query time the query embedding is divided into the same `M` subspaces, and the distance between the query subvector in that subspace and each centroid in the subspace is calculated and cached in a lookup table. Since each document's PQ code maps to a specific centroid in each subspace, we can get the approximate distance between the query subvector and document subvector by looking it up in the cached lookup table. The vector similarity scores between the query and each document can then be calculated and ranked by the smallest distance across the combined subspaces, using a metric like inner product or Euclidean distance. Calculating the Euclidean distance, for example, just requires taking the square root of the sum of the squared distances from each subvector.

Libraries like FAISS provide efficient implementations of PQ out of the box. Listing 13.25 demonstrates the process of building an index using PQ.

Listing 13.25 Building and benchmarking a product quantization index

```
def index_pq_embeddings(doc_embeddings, index_name, num_subvectors=16):
  dimensions = doc_embeddings.shape[1]  #1
  M = num_subvectors   #2
  num_bits = 8  #3
  index = faiss.IndexPQ(dimensions, M, num_bits)  #4
  index.train(doc_embeddings) #5
  index.add(doc_embeddings) #6
  faiss.write_index(index, index_name)  #7
  return index

pq_index_name = "pq_embeddings"
pq_index = index_pq_embeddings(doc_embeddings,
                               pq_index_name)

evaluate_search(full_index, pq_index, pq_index_name,  #8
                query_embeddings, query_embeddings)  #8
evaluate_rerank_search(full_index, pq_index,  #8
                       query_embeddings,  #8
                       query_embeddings)  #8
```

#1 The original embedding is 1024 dimensions.
#2 Divides the embedding into M=16 subvectors (of 64 dimensions each)
#3 8 bits = 256 maximum cluster centroids per subvector.
#4 Creates the PQ index
#5 Generates the cluster centroids using k-means clustering
#6 Adds all the doc_embeddings to the index
#7 Saves the index to disk so we can measure the size
#8 Runs the benchmarks versus the full-precision index

Output:

```
pq_embeddings search took: 2.092 ms (75.22% improvement)
pq_embeddings index size: 1.34 MB (98.22% improvement)
Recall: 0.3333
Reranked recall: 0.6800
```

You'll notice immediately that the recall for the PQ index is significantly lower than the scalar and binary quantization types we benchmarked at 33.33% recall without reranking and 68% recall after reranking. You'll also notice, however, that the index size is 98.22% smaller than the original full-precision index. Obviously, we've made an intentional trade-off here to hyper-optimize the index size in exchange for recall. Unlike the scalar and binary quantization techniques, however, PQ provides levers for adjusting that trade-off by increasing either the number of subvectors (`M_subvectors`) or the `num_bits` to store more precision in the index and improve recall. For example, if you were to set `M_subvectors` to 64, you would get the following:

```
pq_embeddings search took: 4.061 ms (43.99% improvement)
pq_embeddings index size: 2.23 MB (97.05% improvement)
Recall: 0.5778
Reranked recall: 0.9911
```

These results are now more on par with the binary quantization results. The key takeaway, then, is that the primary benefit of PQ is in the flexibility to control the trade-off between index size and recall, particularly when you need to significantly compress your embeddings.

We've now explored several quantization approaches to reduce the size of the indexed embeddings and speed up search times while maintaining a very high level of recall relative to the compression level. In the next subsection, we'll explore a different approach to tackling the compression versus recall trade-off by introducing embeddings that actually encode multiple levels of precision inside the original embedding representation.

13.7.4 Matryoshka Representation Learning

Matryoshka Representation Learning (MRL) is a novel approach to vector performance optimization that learns a hierarchical representation of a vector space, where multiple levels of precision are encoded by different ranges of dimensions within the vector space. This allows for flexible-length representations where shorter segments of an embedding can approximate the meaning of the full embedding, just with reduced precision. MRL is named after the Russian nesting dolls (Matryoshka dolls), signaling the "layers" of precision to be discovered inside of other layers.

As a conceptual explanation of how MRL works, imagine that someone who's never seen the animated Disney movie *The Lion King* asked you to describe it. If you're familiar with the movie, you might start with a very high-level summary and then expand on your description if the person wants more details. For example, you could imagine the following possible responses:

1. It's a Disney animated movie about a lion.
2. It's a Disney animated movie about a lion cub who grows up to become king.
3. It's a Disney animated movie about a lion cub named Simba who grows up to become king after his father is killed by his uncle.
4. It's a Disney animated movie about a lion cub named Simba who must discover his place in the circle of life after running away from his kingdom as a boy and returning as an

adult to reclaim his throne from his uncle who killed his father.

Notice that these all give the main idea of the story at various levels of granularity. If it was really important that you provide the most accurate description, you'd go with the one with the most detail, but in reality, it can be more effective in some cases to start with the higher-level description and only provide additional levels of detail as needed.

This concept of hierarchical representations that reveal more about the subject at different hierarchical levels of granularity is the key idea behind MRL. This technique allows vectors to have progressive accuracy, meaning that the more of the vector you use, the more precise the representation becomes. To illustrate this, let's consider figure 13.13, which shows the cover of this book at various pixelation levels.

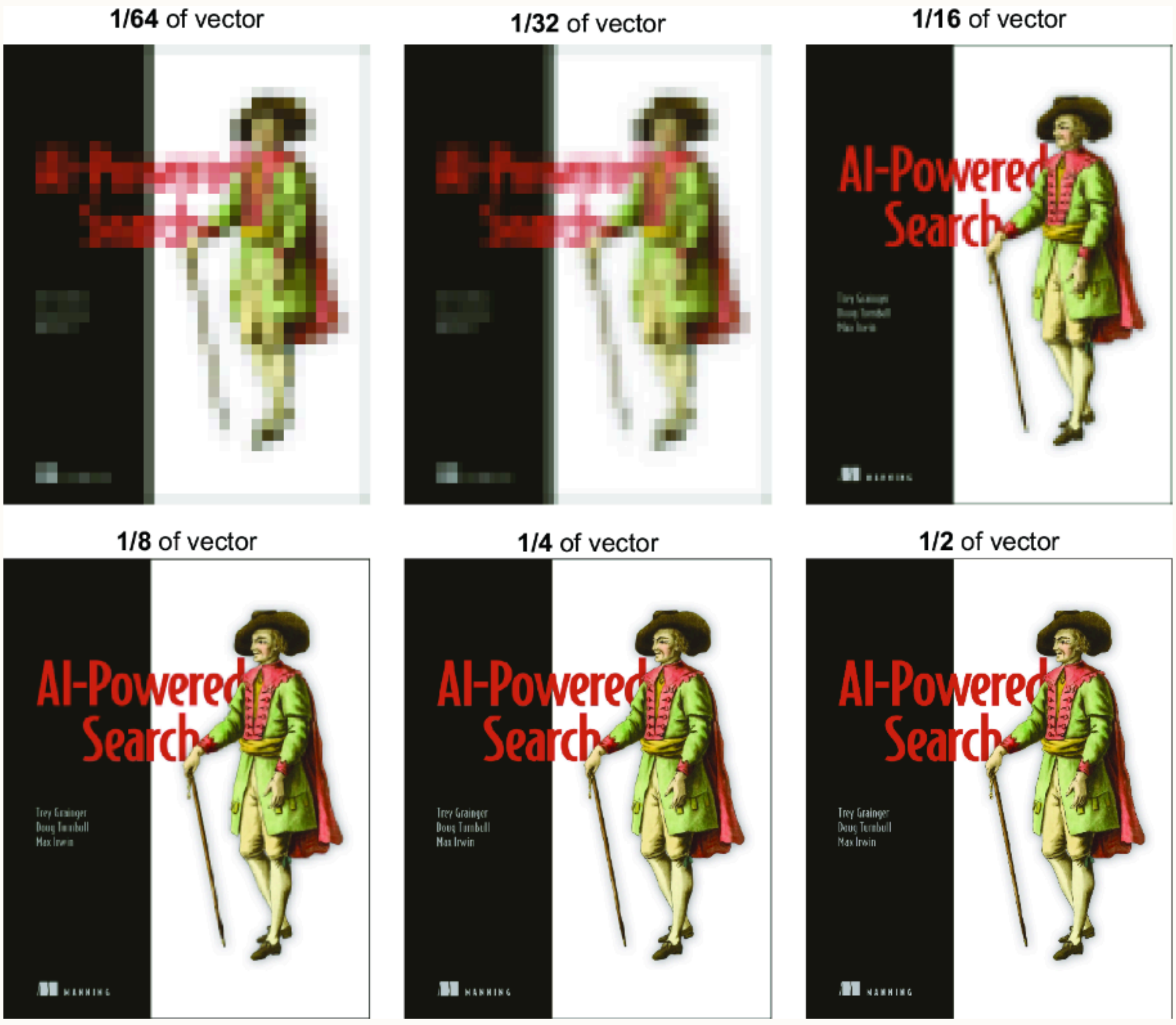


Figure 13.13 Hierarchical representation levels balancing accuracy and compression

As you can see, the image at the top left is highly pixelated, representing a very coarse approximation of the original cover. Moving to the right, and then to the next row, each subsequent image includes twice as many dimensions, eventually leading to a clear and detailed representation. These dimensions are *refinements* on the previous dimensions, however; they are not entirely new information. In other words, it's possible to use a full vector, only the first half of the vector, only the first quarter of the vector, and so on, and still get an approximation of the original vector, just with less precision. This is similar to how MRL embeddings work.

Listing 13.26 demonstrates building and benchmarking a FAISS index using MRL embeddings. Note that because the lower-precision representations are achieved by simply cutting off later dimensions of the original embedding, no special indexing strategy is necessary for MRL embeddings. You simply need to reduce the number of dimensions of the dense vector field you are indexing and searching to the number of dimensions for the MRL embeddings you choose to use (cutting them in half each time and discarding the latter half).

Listing 13.26 Benchmarking MRL embeddings at different thresholds

```
def get_mrl_embeddings(embeddings, num_dimensions):
  mrl_embeddings = numpy.array(  #1
    list(map(lambda e: e[:num_dimensions], embeddings)))  #1
  return mrl_embeddings

def index_mrl_embeddings(doc_embeddings, num_dimensions, mrl_index_name):
  mrl_doc_embeddings = get_mrl_embeddings(doc_embeddings, num_dimensions)
  print(f"{mrl_index_name} embeddings shape:", mrl_doc_embeddings.shape)
  mrl_index = index_full_precision_embeddings(  #2
    mrl_doc_embeddings, mrl_index_name)  #2
  return mrl_index

print(f"Original embeddings shape:", doc_embeddings.shape)
original_dimensions = doc_embeddings.shape[1]  #3

for num_dimensions in [original_dimensions//2,  #4
                       original_dimensions//4,  #5
                       original_dimensions//8]:  #6

  mrl_index_name = f"mrl_embeddings_{num_dimensions}"
  mrl_index = index_mrl_embeddings(doc_embeddings,
                                   num_dimensions,
                                   mrl_index_name)
  mrl_queries = get_mrl_embeddings(query_embeddings,
                                   num_dimensions)

  evaluate_search(full_index, mrl_index, mrl_index_name,  #7
                  query_embeddings, mrl_queries) #7
  evaluate_rerank_search(full_index,   #8
    mrl_index, query_embeddings, mrl_queries)  #8
```

#1 Uses only the top num_dimensions dimensions

#2 An MRL index is a standard index, just with a reduced number of dimensions.

#3 1024 dimensions

#4 512 dimensions

#5 256 dimensions

#6 128 dimensions

#7 Benchmark MRL search

#8 Benchmark MRL search + reranking

Output:

```
Original embeddings shape: (18456, 1024)

mrl_embeddings_512 embeddings shape: (18456, 512)
mrl_embeddings_512 search took: 3.586 ms (49.15% improvement)
mrl_embeddings_512 index size: 37.8 MB (50.0% improvement)
Recall: 0.7022
Reranked recall: 1.0

mrl_embeddings_256 embeddings shape: (18456, 256)
mrl_embeddings_256 search took: 1.845 ms (73.45% improvement)
mrl_embeddings_256 index size: 18.9 MB (75.0% improvement)
Recall: 0.4756
Reranked recall: 0.9689

mrl_embeddings_128 embeddings shape: (18456, 128)
mrl_embeddings_128 search took: 1.061 ms (84.35% improvement)
mrl_embeddings_128 index size: 9.45 MB (87.5% improvement)
Recall: 0.2489
Reranked recall: 0.64
```

In the output, we see that 70.22% of the recall was encoded into the first 50% of the features of the embedding, 47.56% was encoded into the first 25% of the features, and 24.89% was encoded into the first 12.5% of the features. While this level of recall relative to the compression may not be as impressive as some of the earlier quantization approaches, the fact that it is built into the embedding representation to be used (or not used) whenever desired with no special indexing requirements provides some useful flexibility. Additionally, as we'll see in the next subsection, MRL can also be combined with most of the other approaches.

13.7.5 Combining multiple vector search optimization approaches

In this chapter, we've discussed multiple methods to improve the efficiency of vector search:

- ANN search to filter the number of documents that must be scored to those likely to score well
- Quantization techniques to reduce the memory requirements and processing time for embeddings
- MRL to reduce the number of dimensions required to find an initial set of search results
- Reranking to improve the recall and ranking of the top-*N* results after applying the other techniques more aggressively

In practice, you can often combine several of these techniques to achieve your desired level of performance. Listing 13.27 shows a final example combining each of these approaches in a single implementation: ANN using an inverted file index (IVF) for performance, binary quantization for performance and compression, MRL at 1/2 the original dimensionality for performance and compression, and reranking 2 times the number of results to improve recall and relevance ranking.

Listing 13.27 Combining ANN, quantization, MRL, and reranking

```
def index_binary_ivf_mrl_embeddings(reduced_mrl_doc_embeddings,
                                    binary_index_name):
  binary_embeddings = quantize_embeddings( #1
    reduced_mrl_doc_embeddings,  #1
    calibration_embeddings=reduced_mrl_doc_embeddings,  #1
    precision="binary").astype(numpy.uint8)  #1

  dimensions = reduced_mrl_doc_embeddings.shape[1]  #2
  quantizer = faiss.IndexBinaryFlat(dimensions)  #2


  num_clusters = 256  #3
  index = faiss.IndexBinaryIVF(  #3
            quantizer, dimensions, num_clusters)  #3
  index.nprobe = 4  #3

  index.train(binary_embeddings) #4
  index.add(binary_embeddings)  #4
  faiss.write_index_binary(index, binary_index_name)  #4
  return index

mrl_dimensions = doc_embeddings.shape[1] // 2  #5
reduced_mrl_doc_embeddings =  get_mrl_embeddings(  #5
  doc_embeddings, mrl_dimensions)  #5

binary_ivf_mrl_index_name = "binary_ivf_mrl_embeddings"
binary_ivf_mrl_index = index_binary_ivf_mrl_embeddings(
  reduced_mrl_doc_embeddings, mrl_dimensions,
  binary_ivf_mrl_index_name)

mrl_queries = get_mrl_embeddings(query_embeddings,  #6
                                 mrl_dimensions)  #6
quantized_queries = quantize_embeddings(mrl_queries,  #7
  calibration_embeddings=reduced_mrl_doc_embeddings,  #7
  precision="binary").astype(numpy.uint8)  #7

evaluate_search(full_index, binary_ivf_mrl_index,  #8
  binary_ivf_mrl_index_name,  #8
  query_embeddings, quantized_queries)  #8
evaluate_rerank_search(  #9
  full_index, binary_ivf_mrl_index,  #9
  query_embeddings, quantized_queries)  #9
```

#1 Binary quantization: applies quantization to the doc embeddings
#2 Configuration so the index knows how the doc embeddings have been quantized
#3 ANN: uses a binary-quantized IVF index for ANN search
#4 Trains, adds documents, and saves the combined index to disk
#5 MRL: gets reduced-dimension doc embeddings
#6 MRL: gets reduced-dimension query embeddings
#7 Binary quantization: applies quantization to the query embeddings
#8 Benchmarks the binary ANN, binary quantization, and MRL embeddings
#9 Benchmarks again with reranking using full-precision embeddings

Output:

```
binary_ivf_mrl_embeddings search took: 0.064 ms (99.09% improvement)
binary_ivf_mrl_embeddings index size: 1.35 MB (98.22% improvement)
Recall: 0.3511
Reranked recall: 0.7244
```

As you can see from the listing, the different embedding optimization approaches can be combined in complementary ways to achieve your desired balance between query speed, index compression and memory usage, and final relevance ranking. In this case, by combining ANN, MRL, binary quantization, and reranking, we were able to achieve by far the fastest search time (~99% faster than full-precision search) and by far the smallest index size (~98% reduction), while still maintaining over 72% recall after reranking.

USING QUANTIZATION IN A SUPPORTED ENGINE

We've created quantized indexes using FAISS in order to ensure all the demonstrated ANN and quantization approaches can be easily reproduced for index size, search speed, and recall, independent of your configured default `engine`. Different search engines have varying levels of support for quantization approaches, with some performing quantization inside the engine and some requiring you to quantize outside of the engine and configure an appropriate quantization format (`FLOAT32`, `INT8`, `BINARY`, and so on). The `search` method on our `collection` interface implements support for scalar quantization and binary quantization using the latter approach by accepting a `quantization_size` parameter (previously demonstrated in listing 11.19), and MRL should be supported by any engine with vector search capabilities by truncating MRL embeddings prior to indexing and searching. Reranking is also supported by adding a `rerank_query` section to your search request. For example:

```
{
   'query': [0, ..., 1],
   'query_fields': ['binary_embedding'],
   'quantization_size': 'BINARY',
   'order_by': [('score', 'desc')],
   'limit': 25,
     'rerank_query': {
       'query': [-0.01628, ..., 0.02110],
       'query_fields': ['full_embedding'],
       'quantization_size': 'FLOAT32',
       'order_by': [('score', 'desc')],
       'rerank_count': 50
     }
 }
```

This example implements an initial binary quantized query (to whatever degree your engine supports it) and returns the top 25 results after reranking the top 50 results using full-precision (`Float32`) embeddings.

Combining various ANN, scalar quantization, binary quantization, product quantization, and MRL techniques can significantly improve the efficiency of your vector search system.

Many other techniques for quantization exist, and more are being developed all the time, but this overview should give you a great starting point if you're looking to optimize your vector search usage. By experimenting with these techniques and combining them in different ways, you can optimize your search system to achieve the desired balance between speed, memory usage, and recall, especially when applying reranking as a final step. But reranking based on embeddings isn't always the best way to get the most relevant final search results. In the next section, we'll explore an often better way to perform the final reranking of top results: using a cross-encoder.

13.8 Cross-encoders vs. bi-encoders

In this chapter, we've built semantic search using a Transformer-based encoder to generate embeddings. Our strategy involved finding the similarity between a query and document by separately encoding both of them into embeddings, and then using a cosine similarity to generate a relevance score based on the similarity of the two embeddings. An encoder like this that separately encodes each input into an embedding, so that those embeddings can be compared, is known as a *bi-encoder*.

In contrast with a bi-encoder, a *cross-encoder* encodes pairs of inputs together and returns a similarity score. Assuming the inputs are a query and a document, the cross-encoder will concatenate and then encode the query and document together, returning a similarity score measuring how well the document answers the query.

Functionally, the cross-encoder is still generating a similarity score between the two inputs, but it can capture the shared context between the query and document in a way that a bi-encoder cannot. For example, a bi-encoder might place the query `mountain hike` near the document containing the text "first time snow hiking" because they are both related to a similar concept—hiking. But the cross-encoder would pass both the query and document to the encoder (a Transformer), which could then identify through its attention mechanism that while both inputs are about hiking, the document is about beginner snow hiking (which likely wouldn't involve mountains the first time) instead of specifically being about the query of `mountain hike`. By using the context of the query to interpret the document, the cross-encoder can therefore reach a more nuanced interpretation of how well a document matches a query than a bi-encoder, which only interprets the query and document independently.

Figure 13.14 visualizes the bi-encoder and cross-encoder architectures.

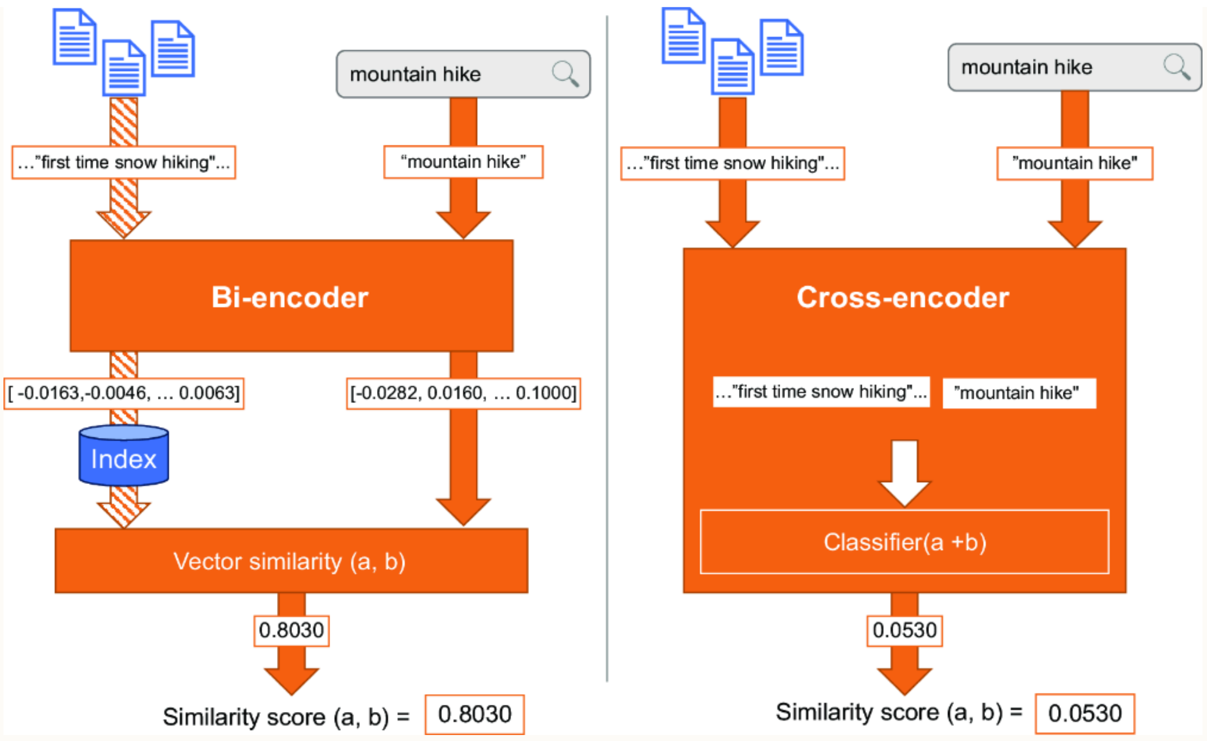


Figure 13.14 Bi-encoders vs. cross-encoders. Bi-encoders process query and document inputs separately, whereas cross-encoders process them together into a similarity score.

In the figure, note the following key characteristics:

- Bi-encoders encode queries and documents separately into embeddings that can then be compared using a similarity function (like cosine).
- Cross-encoders encode queries together to assign a similarity score.
- Solid-colored arrows indicate data processing that must occur at query time, whereas striped arrows indicate work that can be performed at index time and cached. Note that bi-encoders only need to encode the query once at query time, whereas cross-encoders must encode the query along with every document that needs a similarity score at query time.

Cross-encoders are more computationally expensive than bi-encoders at query time, but they are also usually more accurate than bi-encoders because they can capture the relevant interactions between the query and the document contexts. For this reason, cross-encoders are often used to rerank a small subset of the top results from a much faster bi-encoder-based vector search or lexical search to provide a more accurate similarity score for the top query and document pairs.

Just like the learning-to-rank models we built in chapters 10–12, cross-encoders are another form of *ranking classifier*—a model that classifies inputs into probable similarity scores. Listing 13.28 demonstrates how to invoke a cross-encoder to rerank search results. We'll use the same initial query from back in listing 13.19, but we'll over-request the number of search results (`limit=50`), so we'll provide more options for the cross-encoder to rerank.

Listing 13.28 Reranking search results with a cross-encoder

```
from sentence_transformers import CrossEncoder
cross_encoder = \  #1
  CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")  #1

query = "mountain hike"
search_results = semantic_search_with_engine(  #2
  embeddings_collection, query, limit=50)  #2
pairs_to_score = [[query, doc["title"]]  #3
                  for doc in search_results]  #3
cross_scores = cross_encoder.predict(pairs_to_score,  #4
                 activation_fct=torch.nn.Sigmoid())  #5
reranked_results = rerank(search_results, cross_scores) #6
display_results(query, reranked_results[:10])
```

#1 Cross-encoder model trained on a similar dataset of questions and answers.
#2 Generates a pair of query + document title to score for each document
#3 Over-requests 50 results to supply sufficient candidates to rerank
#4 Invokes the cross-encoder to score each pair
#5 Optional activation function to normalize between 0 and 1
#6 Updates the relevance ranking based on the cross-encoder scores

Response:

```
0.578 | What constitutes mountain exposure when hiking or scrambling?
0.337 | ... hiking trails... in... Rocky Mountain National Park...
0.317 | Where in the US can I find green mountains to hike...?
0.213 | Appropriate terms... hiking, trekking, mountaineering...
0.104 | Camping on top of a mountain
0.102 | ... Plan a Hiking Trip to Rocky Mountain National Park, CO
0.093 | What considerations... for... a hiking ascent of Mount...
0.073 | Are there any easy hiking daytrips up mountains...
0.053 | First time snow hiking
0.049 | Advice for first Grand Canyon Hike for Eastern Hikers
```

Though the scores are not cosine similarities (and thus not directly comparable with the bi-encoder scores), the quality of the results seems generally improved after applying the bi-encoder. Notice that more documents are now related to mountain hiking as opposed to hiking in general.

Taking this chapter as a whole, one common pattern for integrating cross-encoders with bi-encoders is as follows:

1. Perform an initial search using a combination of ANN, quantization, and representation learning techniques (super fast).
2. Rerank a medium-sized number of results (hundreds or thousands) using *N* higher-precision vectors loaded from disk for only the top *N* results (fast, since the number results is reduced).
3. Take the top page or two of results, and rerank them with a cross-encoder to get the optimal ranking for the top results (slow, but only for a small number of results).

You can, of course, use any learning-to-rank model for your final ranking step. Cross-encoders are one of the most-deployed kinds of learning-to-rank model (focused solely on document content), as they don't require explicitly modeled features and instead have learned to model language and content features automatically from a deep learning training process. You can (and should) certainly fine-tune your cross-encoder model, leveraging the techniques from chapters 10 and 11, but many people just use off-the-shelf pretrained cross-encoder models without fine-tuning them, because they tend to generalize well when dealing with general text content.

With what you've learned in this chapter, you should now be able to do the following with your own content:

- Assess and choose an existing fine-tuned Transformer encoder model that matches your use case.
- Encode important text from your documents, and add them to an embeddings index for ANN search.
- Build an autocomplete pipeline to accept plain text queries, and quickly return the most closely related concepts.
- Add a powerful high-recall semantic search to your product.
- Optimize the performance of your dense vector search system by combining ANN, quantization, MRL, and reranking techniques.
- Rank results with a bi-encoder, and rerank those search results with a cross-encoder to improve search relevance ranking.

The technology underlying dense vector search still has room for improvement, because ranking on embeddings can be calculation-intensive, and ANN approaches have nontrivial scaling trade-offs. Sparse term vectors leveraging an inverted index are still much more efficient and simpler to scale. But tremendous forward progress continues to be made toward productionizing these dense vector search techniques, and for good reason. Not only does searching on vectors enable better semantic search on text, but it also enables cutting-edge approaches to question answering, results summarization, image search, and other advanced search use cases like retrieval augmented generation (RAG), all of which we'll cover in the following chapters.

Summary

- Dense vector search ranks relevant documents by comparing the distance between embedding vectors, such as from large language models (LLMs).
- Transformers enable LLMs to encode the meaning of content (queries, documents, sentences, etc.) into vectors and also to decode the meaning out of encoded vectors.
- Semantic search and other use search cases like semantic autocomplete can be implemented using embeddings.
- Approximate nearest-neighbor (ANN) search is a technique to speed up dense vector retrieval by filtering to documents containing similar vectors prior to performing expensive similarity calculations between the query and each document's vectors.
- Dense vector search can be heavily optimized for search speed, memory usage, and recall of the best results by combining techniques like ANN search, quantization,

Matryoshka Representation Learning (MRL) embeddings, and over-requesting and reranking of results.

- Bi-encoders generate separate embeddings for queries and documents and support high-volume matching and ranking, whereas cross-encoders require much more computation at query time and are thus best used to rerank a smaller number of top results from a bi-encoder or lexical search.

Previous chapter
‹ Part 4 The search frontier

Next chapter
14 Question answering with a fine-tuned large language model ›