HANDS-ON LARGE LANGUAGE MODELS

LANGUAGE UNDERSTANDING AND GENERATION

BY JAY ALAMMAR AND MAARTEN GROOTENDORST

Contents

Figure P-1	10
Figure 1-1	11
Figure 1-2	12
Figure 1-3	13
Figure 1-4	14
Figure 1-5	15
Figure 1-6	16
Figure 1-7	17
Figure 1-8	18
Figure 1-9	19
Figure 1-10	20
Figure 1-11	21
Figure 1-12	22
Figure 1-13	23
Figure 1-14	24
Figure 1-15	25
Figure 1-16	26
Figure 1-17	27
Figure 1-18	28
Figure 1-19	29
Figure 1-20	30
Figure 1-21	31
Figure 1-22	32
Figure 1-23	33
Figure 1-24	34
Figure 1-25	35
Figure 1-26	36
Figure 1-27	37
Figure 1-28	38
Figure 1-29	39
Figure 1-30	40
Figure 1-31	41
Figure 1-32	42
Figure 2-1	43
Figure 2-2	44

Figure 2-3	45
Figure 2-4	46
Figure 2-5	47
Figure 2-6	48
Tokenized Text Models	49
Figure 2-7	50
Figure 2-8	51
Figure 2-9	52
Figure 2-10	53
Figure 2-11	54
Figure 2-12	55
Figure 2-13	56
Figure 2-14	57
Figure 2-15	58
Figure 2-16	59
Figure 2-17	60
Figure 3-1	61
Figure 3-2	62
Figure 3-3	63
Figure 3-4	64
Figure 3-5	65
Figure 3-6	66
Figure 3-7	67
Figure 3-8	68
Figure 3-9	69
Figure 3-10	70
Figure 3-11	71
Figure 3-12	72
Figure 3-13	73
Figure 3-14	74
Figure 3-15	75
Figure 3-16	76
Figure 3-17	77
Figure 3-18	78
Figure 3-19	79
Figure 3-20	80
Figure 3-21	81

Figure 3-22	82
Figure 3-23	83
Figure 3-24	84
Figure 3-25	85
Figure 3-26	86
Figure 3-27	87
Figure 3-28	88
Figure 3-29	89
Figure 3-30	90
Figure 3-31	91
Figure 3-32	92
Figure 3-33	93
Figure 4-1	94
Figure 4-2	95
Figure 4-3	96
Figure 4-4	97
Figure 4-5	98
Figure 4-6	99
Figure 4-7	100
Figure 4-8	101
Figure 4-9	102
Figure 4-10	103
Figure 4-11	104
Figure 4-12	105
Figure 4-13	106
Figure 4-14	107
Figure 4-15	108
Figure 4-16	109
Figure 4-17	110
Figure 4-18	111
Figure 4-19	112
Figure 4-20	113
Figure 4-21	114
Figure 4-22	115
Figure 4-23	116
Figure 5-1	117
Figure 5-2	118

Figure 5-3	119
Figure 5-4	120
Figure 5-5	121
Figure 5-6	122
Figure 5-7	123
Figure 5-8	124
Figure 5-9	125
Figure 5-10	126
Figure 5-11	127
Figure 5-12	128
Figure 5-13	129
Figure 5-14	130
Figure 5-15	131
Figure 5-16	132
Figure 5-17	133
Figure 5-18	134
Figure 5-19	135
Figure 5-20	136
Figure 5-21	137
Figure 5-22	138
Figure 5-23	139
Figure 5-24	140
Figure 6-1	141
Figure 6-2	142
Figure 6-3	143
Figure 6-4	144
Figure 6-5	145
Table 6-1	146
Figure 6-6	147
Figure 6-7	148
Figure 6-8	149
Figure 6-9	150
Figure 6-10	151
Figure 6-11	152
Figure 6-12	153
Figure 6-13	154
Figure 6-14	155

Figure 6-15	156
Figure 6-16	157
Figure 6-17	158
Figure 6-18	159
Figure 6-19	160
Figure 6-20	161
Figure 6-21	162
Figure 7-1	163
Figure 7-2	164
Figure 7-3	165
Figure 7-4	166
Figure 7-5	167
Figure 7-6	168
Figure 7-7	169
Figure 7-8	170
Figure 7-9	171
Figure 7-10	172
Figure 7-11	173
Figure 7-12	174
Figure 7-13	175
Table 7-1	176
Figure 7-14	177
Figure 7-15	178
Figure 7-16	179
Figure 7-17	180
Figure 8-1	181
Figure 8-2	182
Figure 8-3	183
Figure 8-4	184
Figure 8-5	185
Figure 8-6	186
Figure 8-7	187
Figure 8-8	188
Figure 8-9	189
Figure 8-10	190
Figure 8-11	191
Figure 8-12	192

Figure 8-13	193
Figure 8-14	194
Figure 8-15	195
Figure 8-16	196
Figure 8-17	197
Figure 8-18	198
Figure 8-19	199
Figure 8-20	200
Figure 8-21	201
Figure 8-22	202
Figure 8-23	203
Figure 8-24	204
Figure 8-25	205
Figure 8-26	206
Figure 9-1	207
Figure 9-2	208
Figure 9-3	209
Figure 9-4	210
Figure 9-5	211
Figure 9-6	212
Figure 9-7	213
Figure 9-8	214
Figure 9-9	215
Figure 9-10	216
Figure 9-11	217
Figure 9-12	218
Figure 9-13	219
Figure 9-14	220
Figure 9-15	221
Figure 9-16	222
Figure 9-17	223
Figure 9-18	224
Figure 9-19	225
Figure 9-20	226
Figure 9-21	227
Figure 10-1	228
Figure 10-2	229

Figure 10-3	230
Figure 10-4	231
Figure 10-5	232
Figure 10-6	233
Figure 10-7	234
Figure 10-8	235
Figure 10-9	236
Figure 10-10	237
Figure 10-11	238
Figure 10-12	239
Figure 10-13	240
Figure 10-14	241
Figure 10-15	242
Figure 11-1	243
Figure 11-2	244
Figure 11-3	245
Figure 11-4	246
Figure 11-5	247
Figure 11-6	248
Figure 11-7	249
Figure 11-8	250
Figure 11-9	251
Figure 11-10	252
Figure 11-11	253
Figure 11-12	254
Figure 11-13	255
Figure 11-14	256
Figure 11-15	257
Figure 11-16	258
Figure 11-17	259
Figure 11-18	260
Figure 11-19	261
Figure 11-20	262
Figure 11-21	263
Figure 12-1	264
Figure 12-2	265
Figure 12-3	266

Figure 12-4	267
Figure 12-5	268
Figure 12-6	269
Figure 12-7	270
Figure 12-8	271
Figure 12-9	272
Figure 12-10	273
Figure 12-11	274
Figure 12-12	275
Figure 12-13	276
Figure 12-14	277
Figure 12-15	278
Figure 12-16	279
Figure 12-17	280
Figure 12-18	281
Figure 12-19	282
Figure 12-20	283
Table 12-1	284
Figure 12-21	285
Figure 12-22	286
Figure 12-23	287
Figure 12-24	288
Figure 12-25	289
Figure 12-26	290
Figure 12-27	291
Figure 12-28	292
Figure 12-29	293
Figure 12-30	294
Figure 12-31	295
Figure 12-32	296
Figure 12-33	297

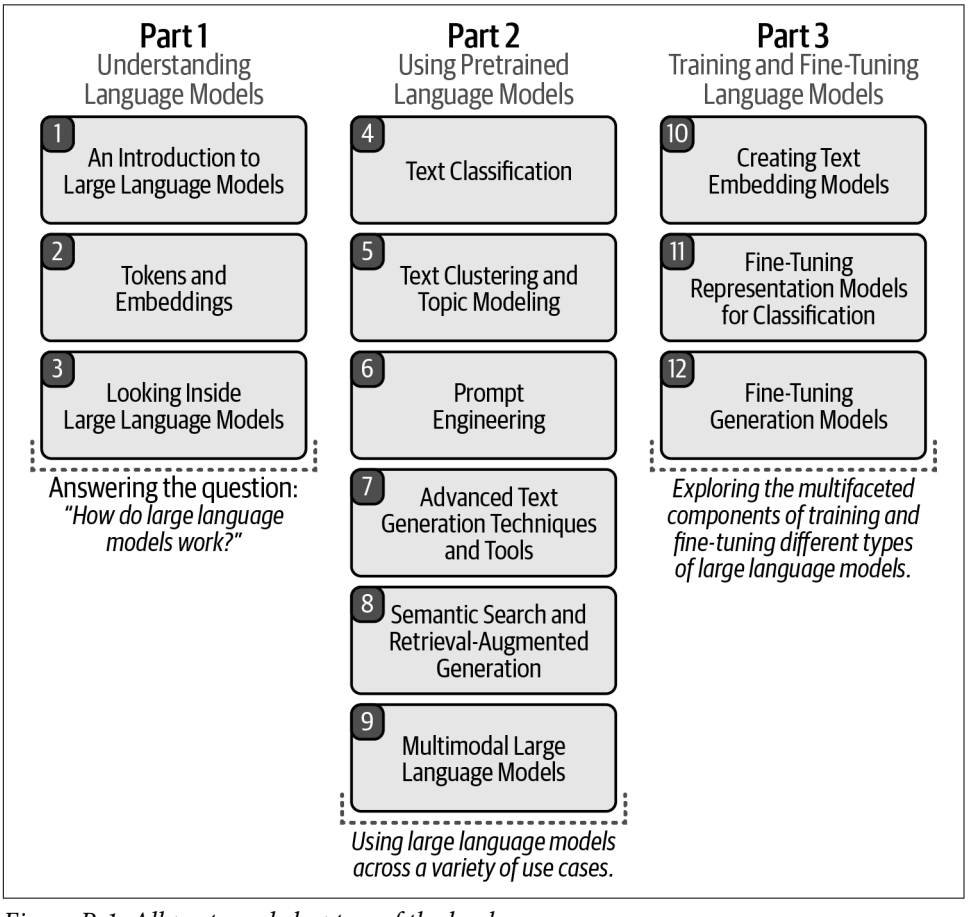

Figure P-1. All parts and chapters of the book.

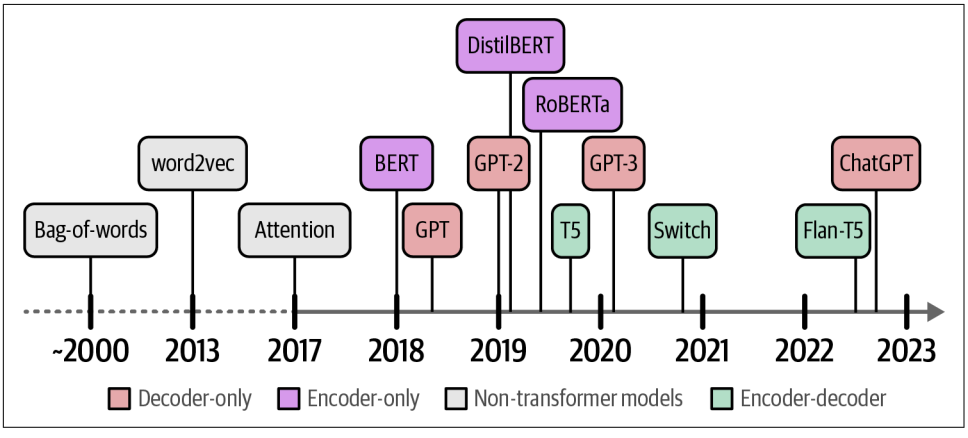

Figure 1-1. A peek into the history of Language AI.

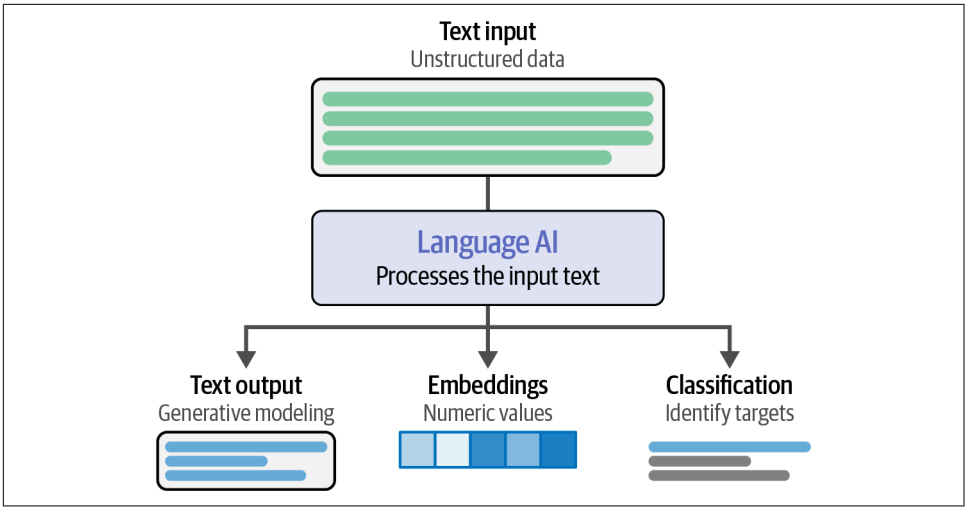

Figure 1-2. Language AI is capable of many tasks by processing textual input.

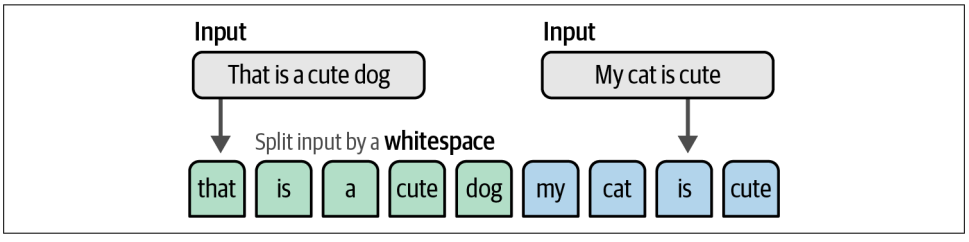

Figure 1-3. Each sentence is split into words (tokens) by splitting on a whitespace.

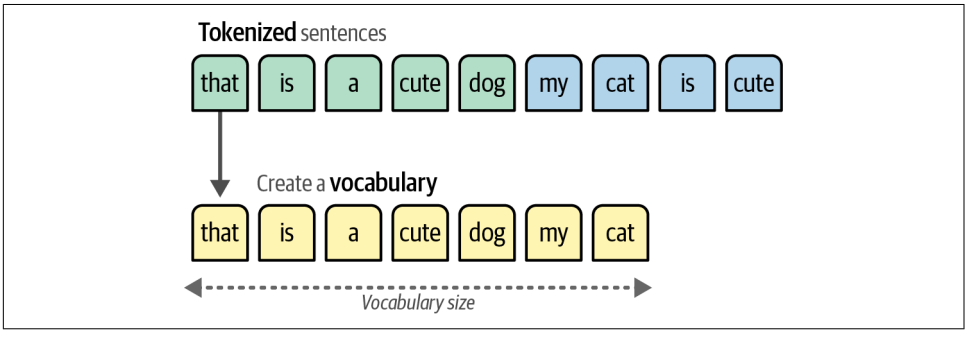

Figure 1-4. A vocabulary is created by retaining all unique words across both sentences.

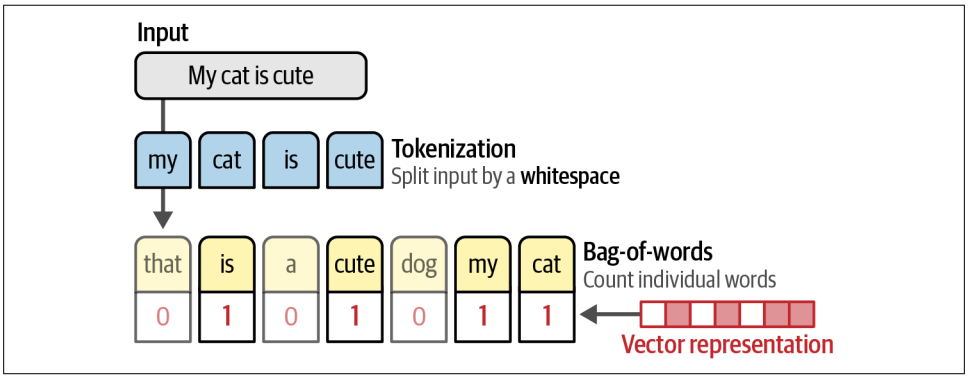

Figure 1-5. A bag-of-words is created by counting individual words. These values are referred to as vector representations.

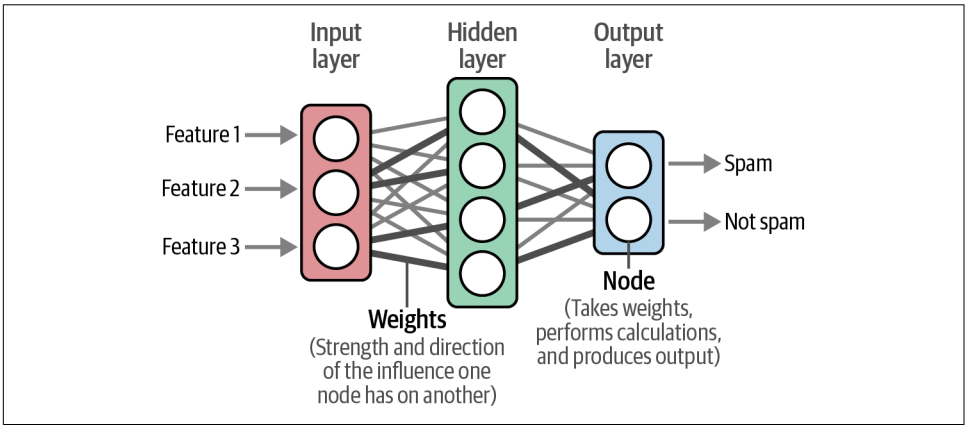

Figure 1-6. A neural network consists of interconnected layers of nodes where each connection is a linear equation.

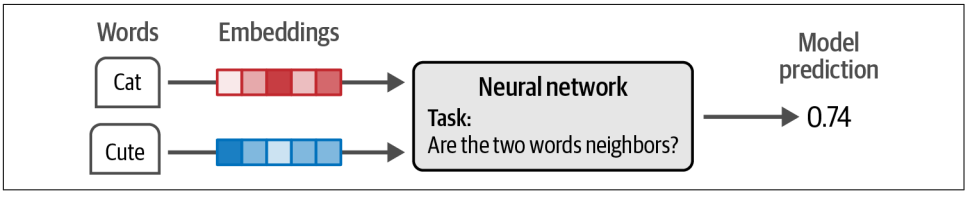

Figure 1-7. A neural network is trained to predict if two words are neighbors. During this process, the embeddings are updated to be in line with the ground truth.

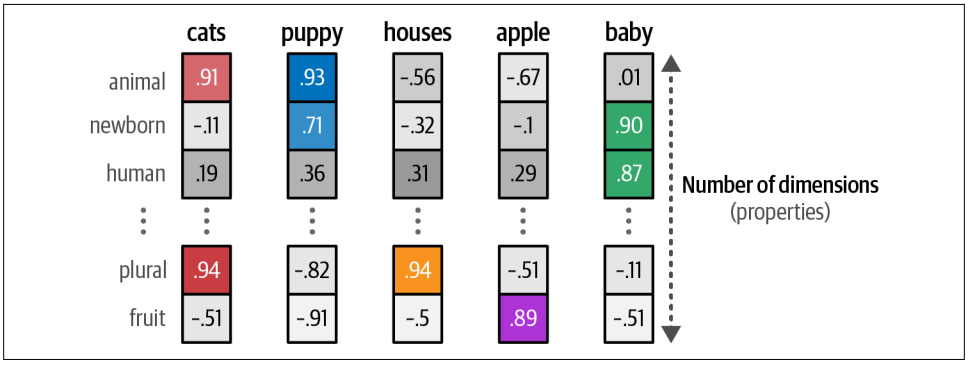

Figure 1-8. The values of embeddings represent properties that are used to represent words. We may oversimplify by imagining that dimensions represent concepts (which they don't), but it helps express the idea.

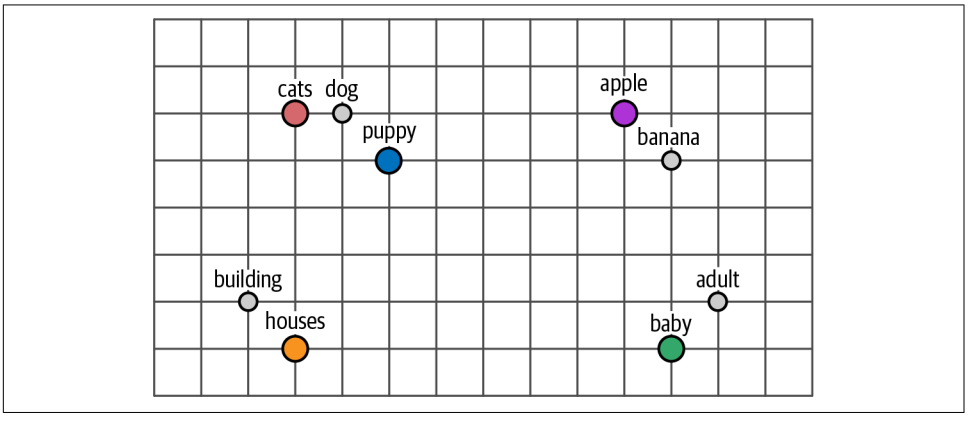

Figure 1-9. Embeddings of words that are similar will be close to each other in dimensional space.

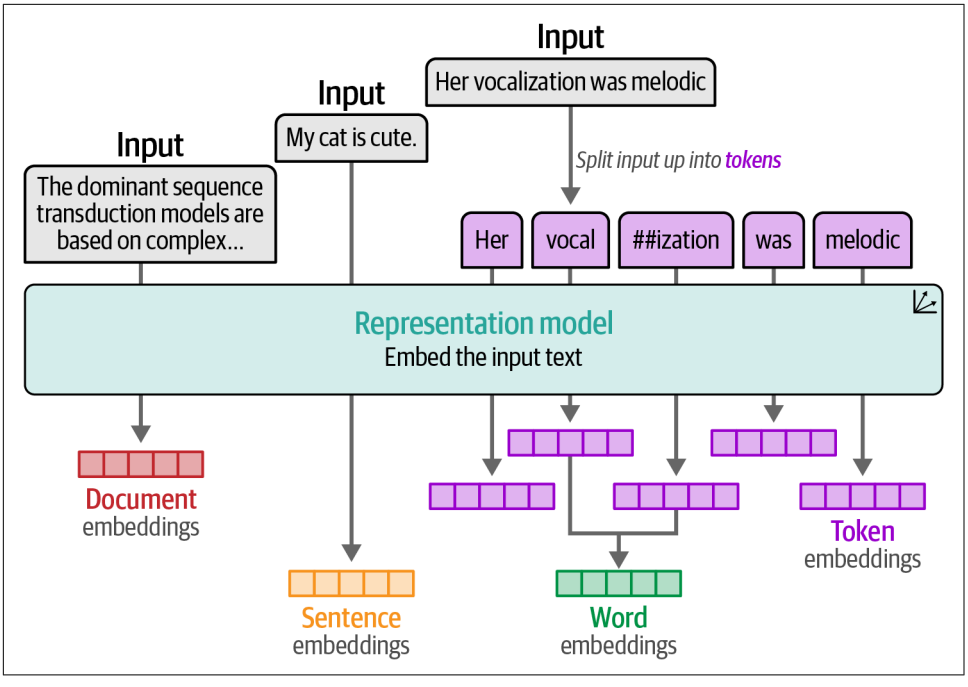

Figure 1-10. Embeddings can be created for different types of input.

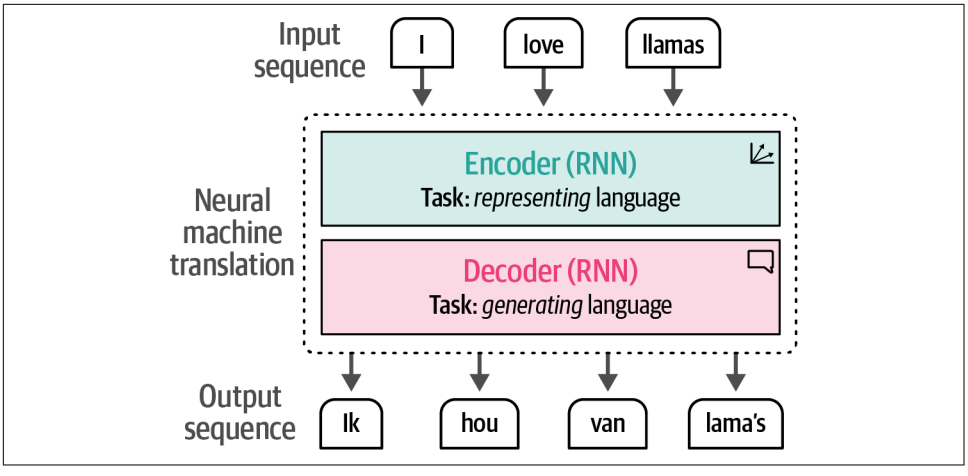

Figure 1-11. Two recurrent neural networks (decoder and encoder) translating an input sequence from English to Dutch.

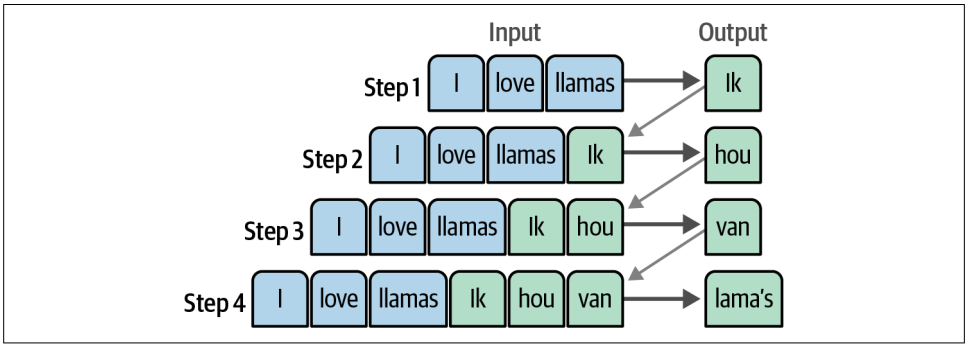

Figure 1-12. Each previous output token is used as input to generate the next token.

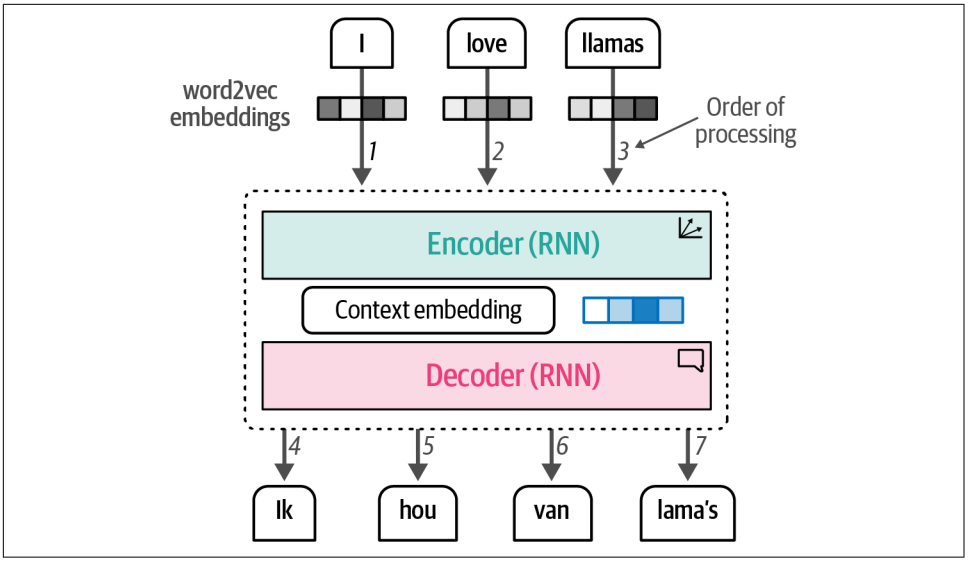

Figure 1-13. Using word2vec embeddings, a context embedding is generated that represents the entire sequence.

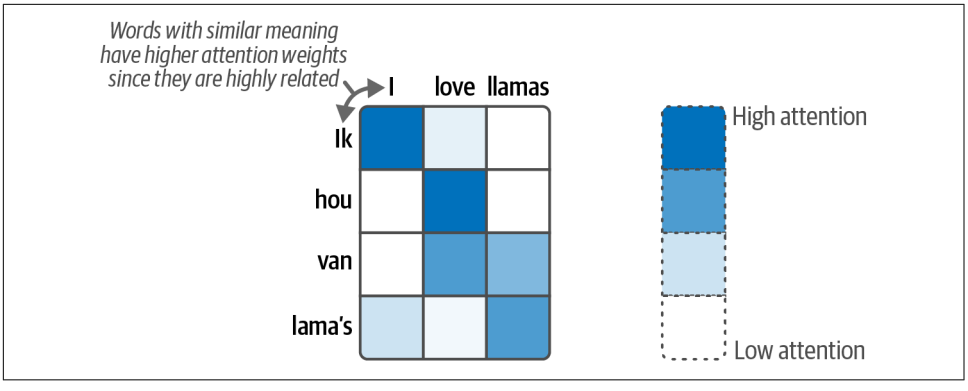

Figure 1-14. Attention allows a model to "attend" to certain parts of sequences that might relate more or less to one another.

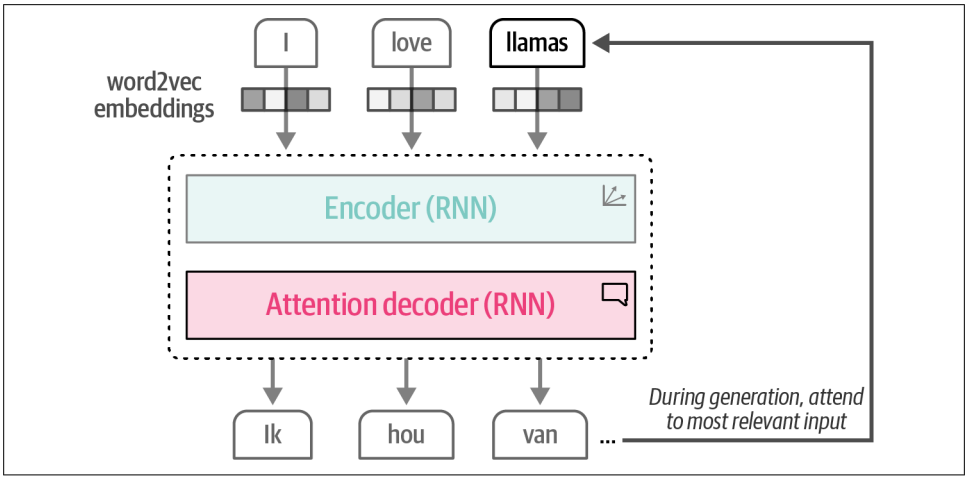

Figure 1-15. After generating the words "Ik," "hou," and "van," the attention mechanism of the decoder enables it to focus on the word "llamas" before it generates the Dutch translation ("lama's").

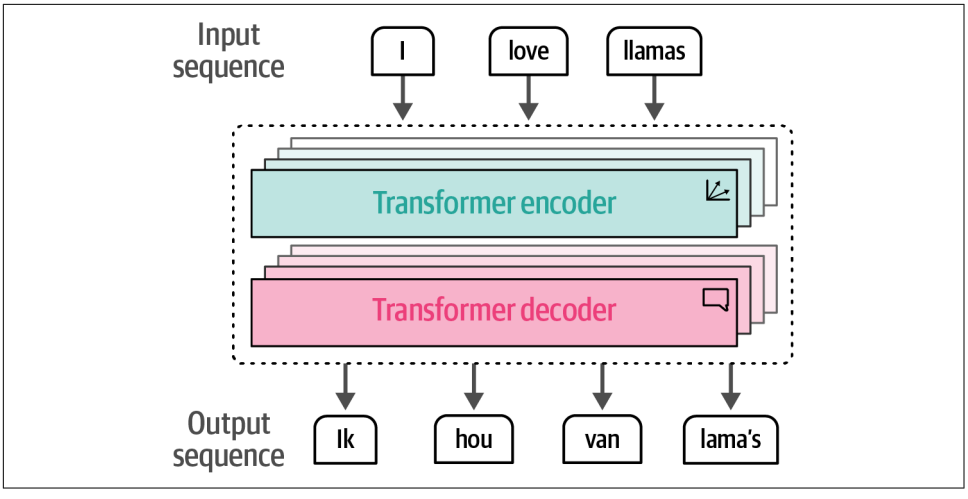

Figure 1-16. The Transformer is a combination of stacked encoder and decoder blocks where the input flows through each encoder and decoder.

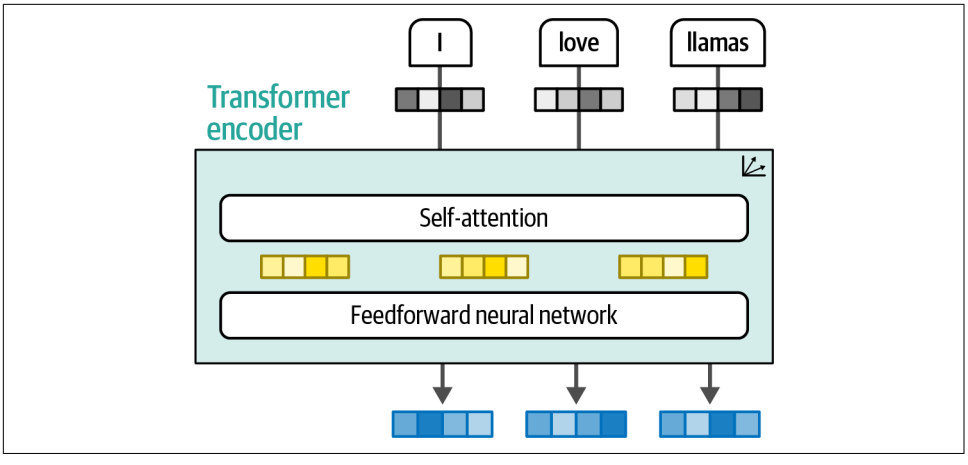

Figure 1-17. An encoder block revolves around self-attention to generate intermediate representations.

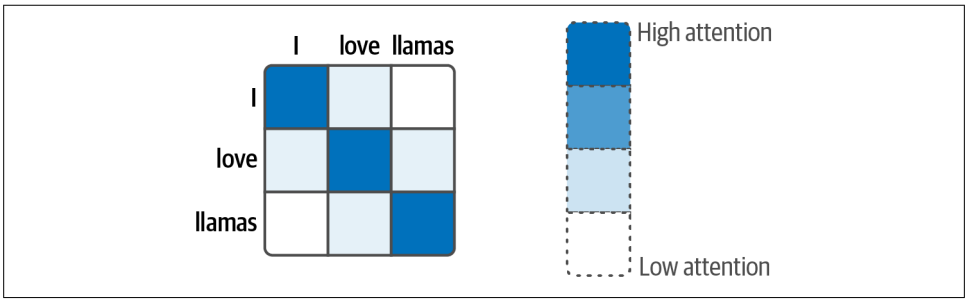

Figure 1-18. Self-attention attends to all parts of the input sequence so that it can "look" both forward and back in a single sequence.

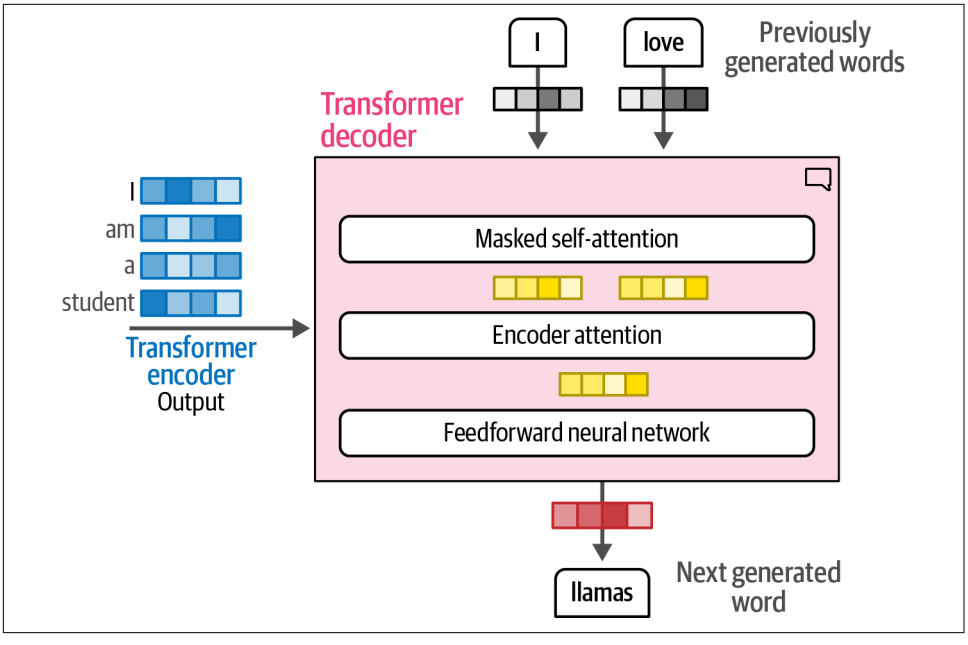

Figure 1-19. The decoder has an additional attention layer that attends to the output of the encoder.

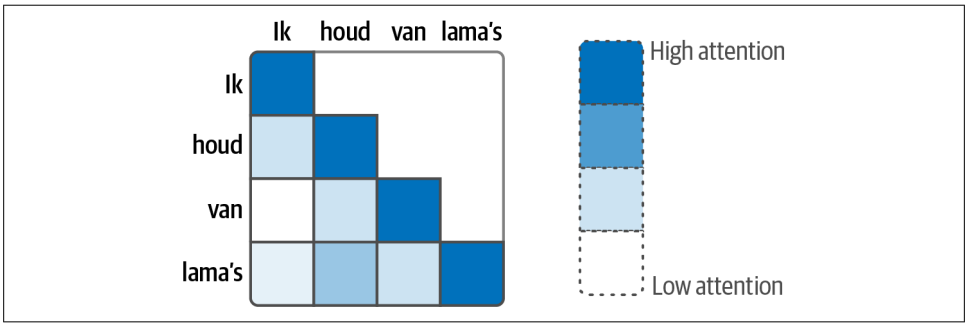

Figure 1-20. Only attend to previous tokens to prevent "looking into the future."

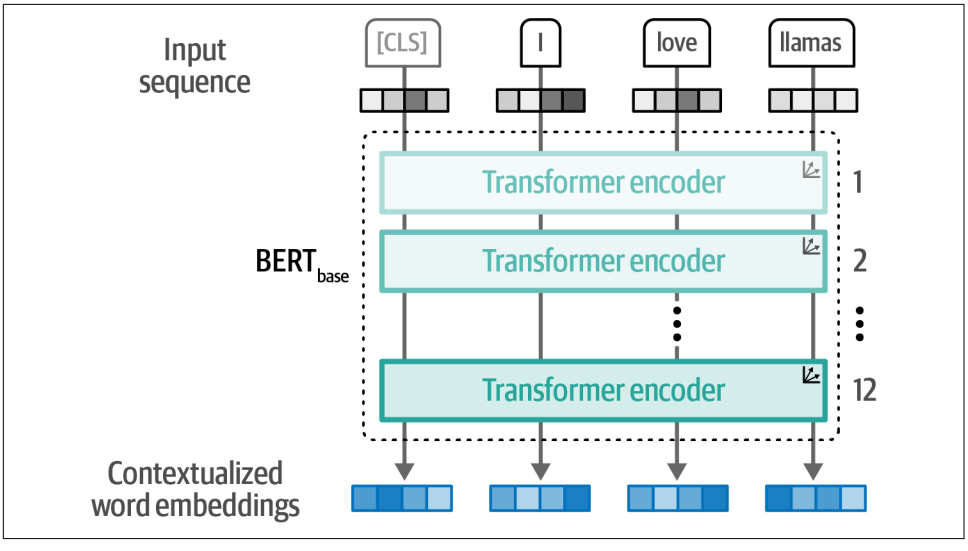

Figure 1-21. The architecture of a BERT base model with 12 encoders.

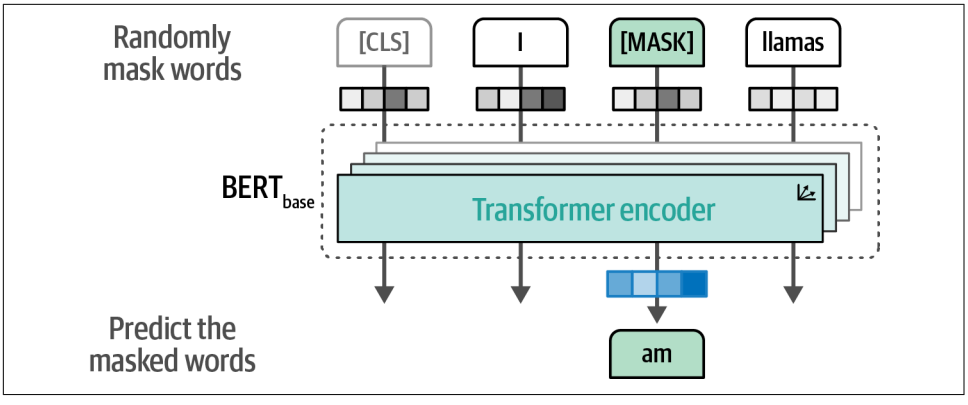

Figure 1-22. Train a BERT model by using masked language modeling.

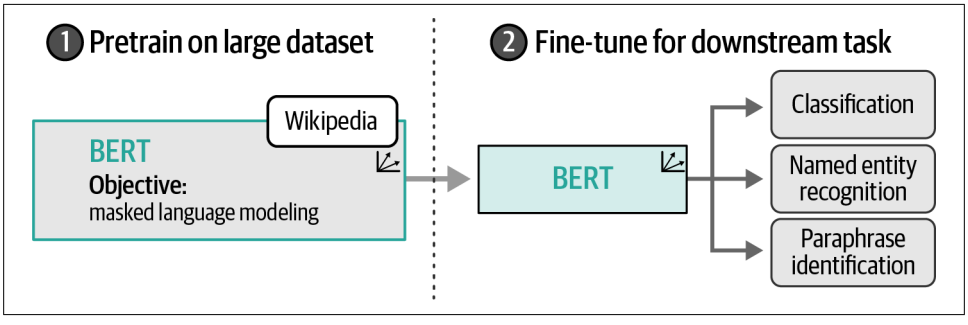

Figure 1-23. After pretraining BERT on masked language model, we fine-tune it for specific tasks.

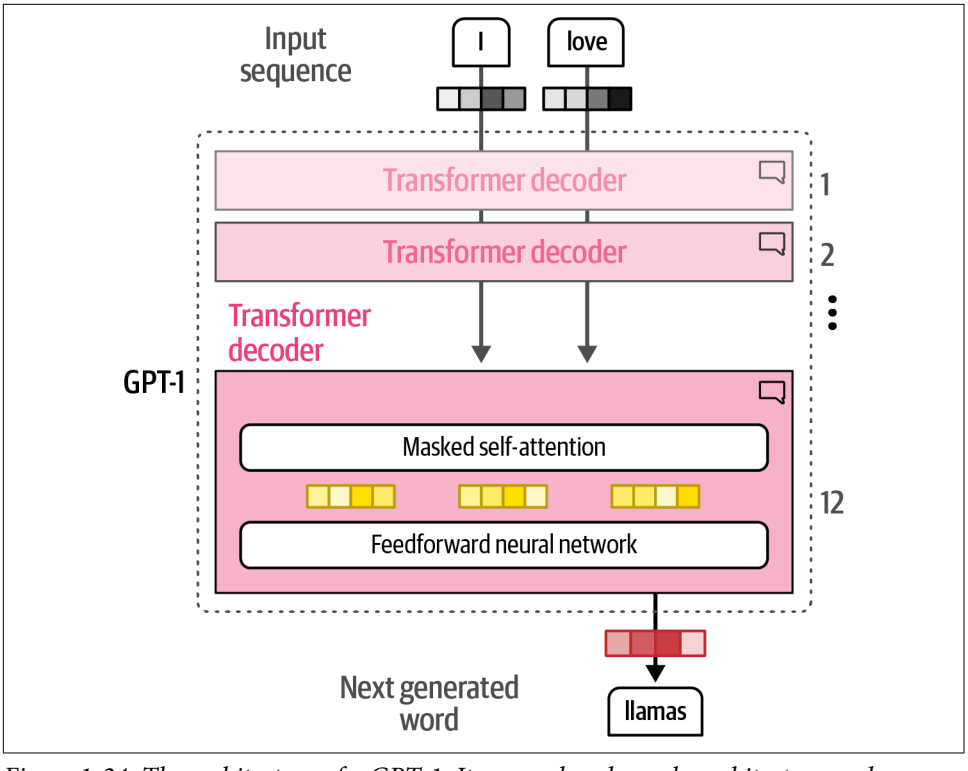

Figure 1-24. The architecture of a GPT-1. It uses a decoder-only architecture and removes the encoder-attention block.

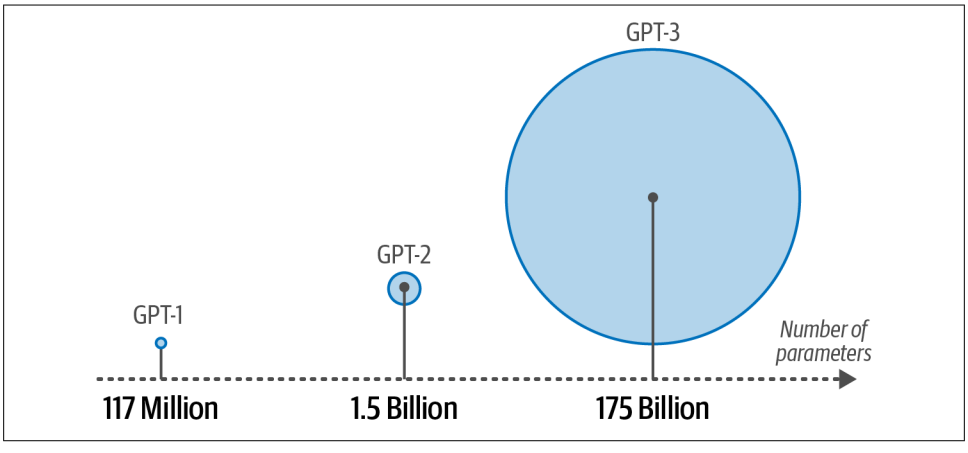

Figure 1-25. GPT models quickly grew in size with each iteration.

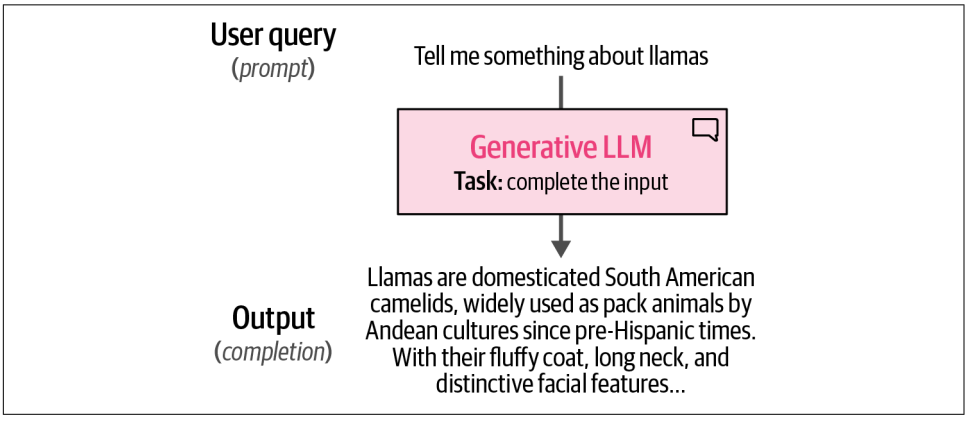

Figure 1-26. Generative LLMs take in some input and try to complete it. With instruct models, this is more than just autocomplete and attempts to answer the question.

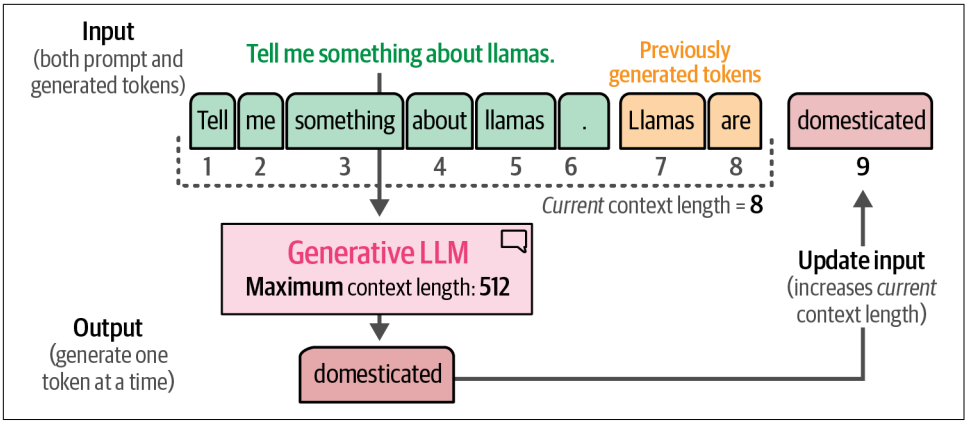

Figure 1-27. The context length is the maximum context an LLM can handle.

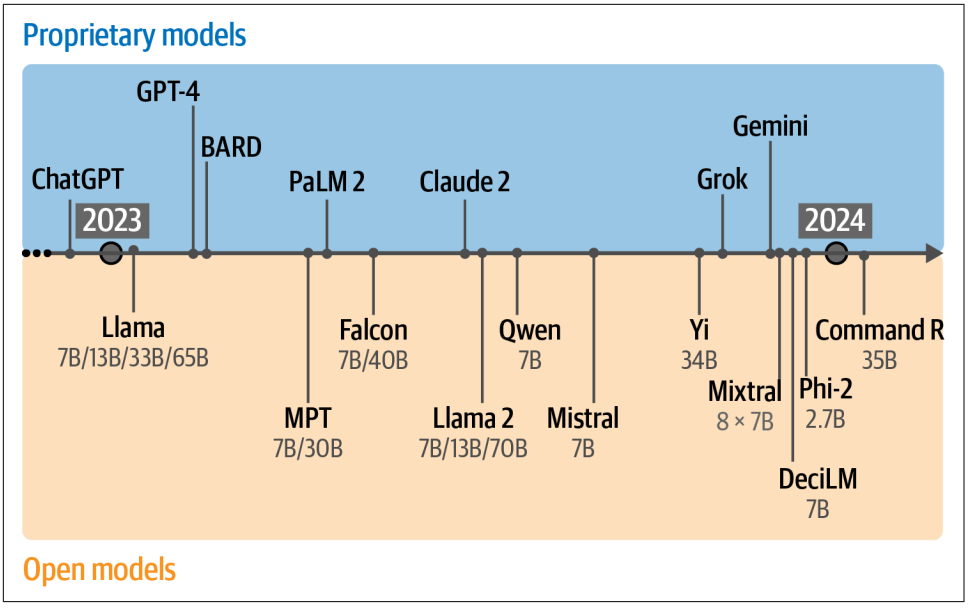

Figure 1-28. A comprehensive view into the Year of Generative AI. Note that many models are still missing from this overview!



Figure 1-29. Traditional machine learning involves a single step: training a model for a specific target task, like classification or regression.

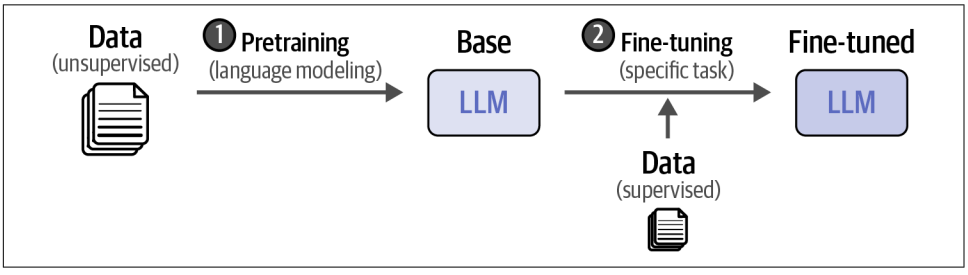

Figure 1-30. Compared to traditional machine learning, LLM training takes a multistep approach.

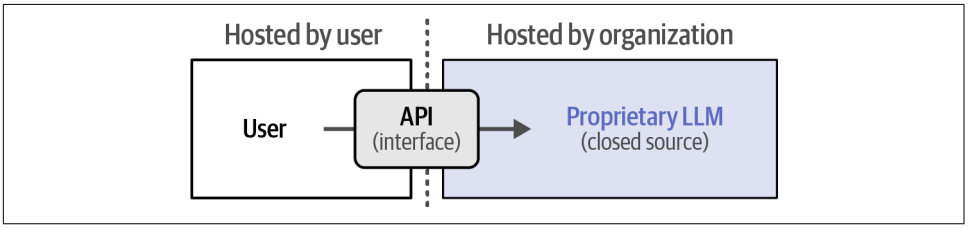

Figure 1-31. Closed source LLMs are accessed by an interface (API). As a result, details of the LLM itself, including its code and architecture are not shared with the user.

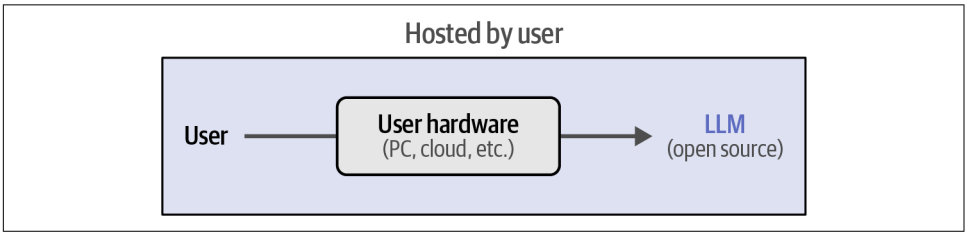

Figure 1-32. Open source LLMs are directly by the user. As a result, details of the LLM itself including its code and architecture are shared with the user.

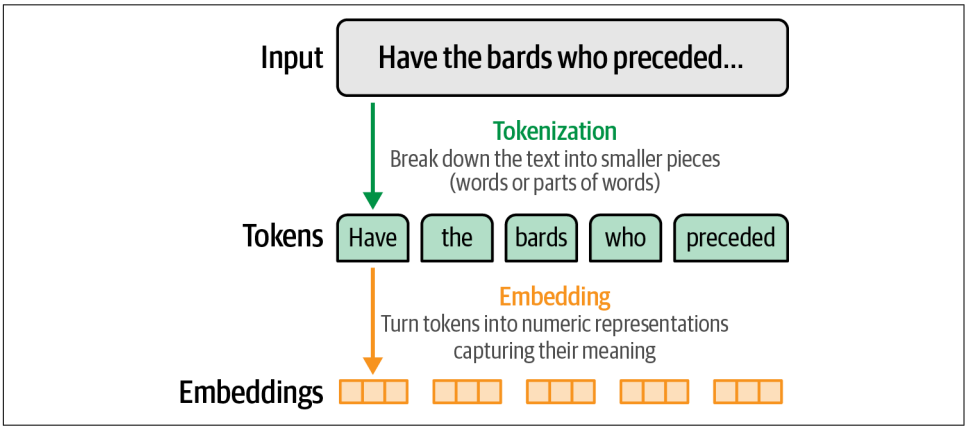

Figure 2-1. Language models deal with text in small chunks called tokens. For the language model to compute language, it needs to turn tokens into numeric representations called embeddings.

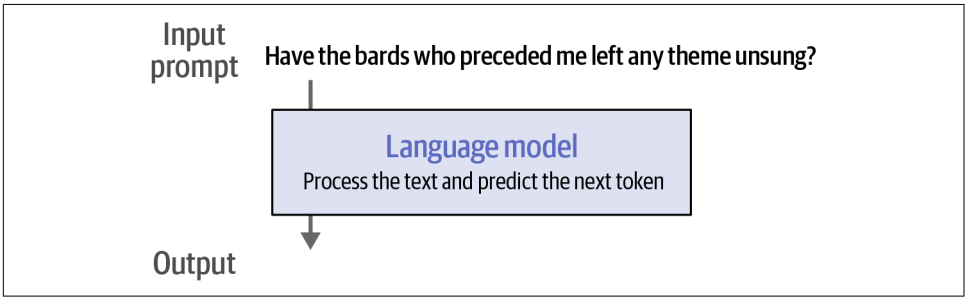

Figure 2-2. High-level view of a language model and its input prompt.

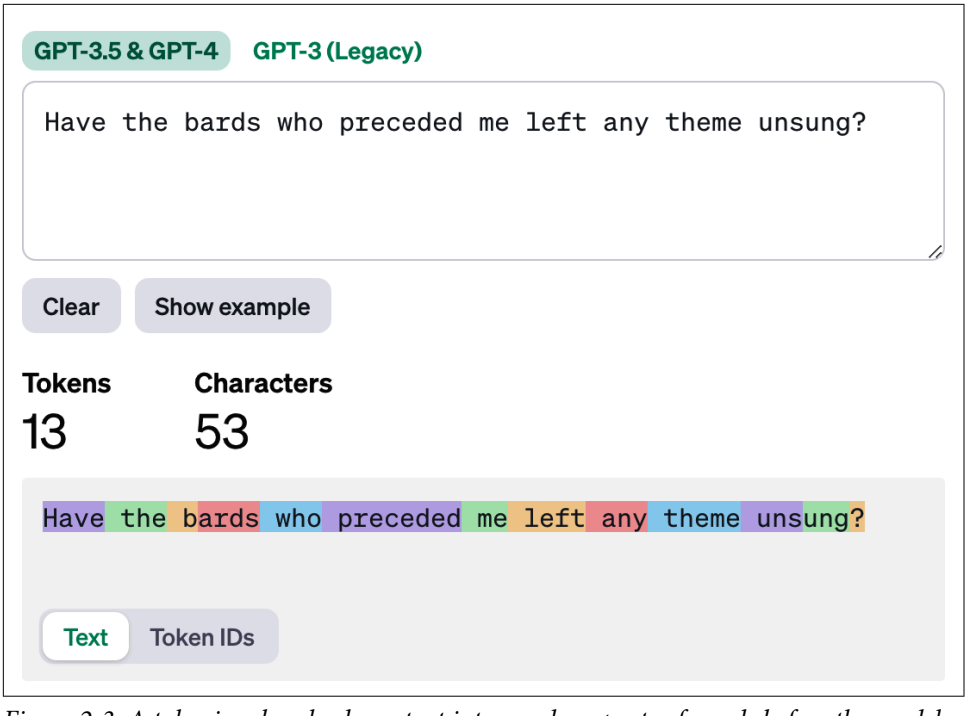

Figure 2-3. A tokenizer breaks down text into words or parts of words before the model processes the text. It does so according to a specific method and training procedure (from https://oreil.ly/ovUWO).

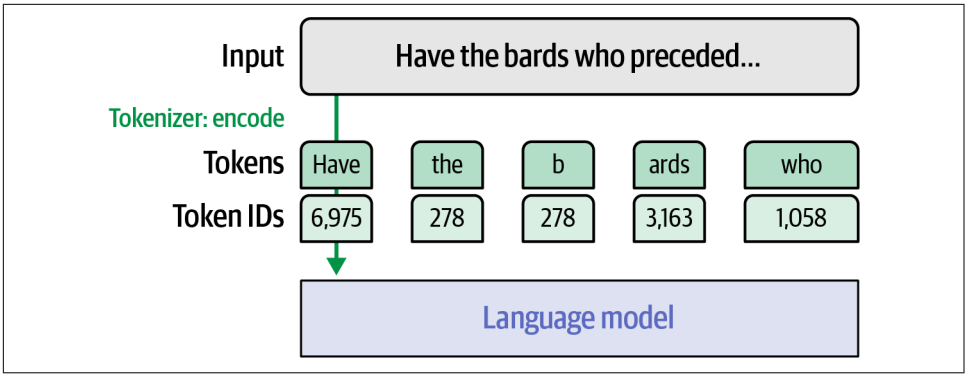

Figure 2-4. A tokenizer processes the input prompt and prepares the actual input into the language model: a list of token IDs. The specific token IDs in the figure are just demonstrative.

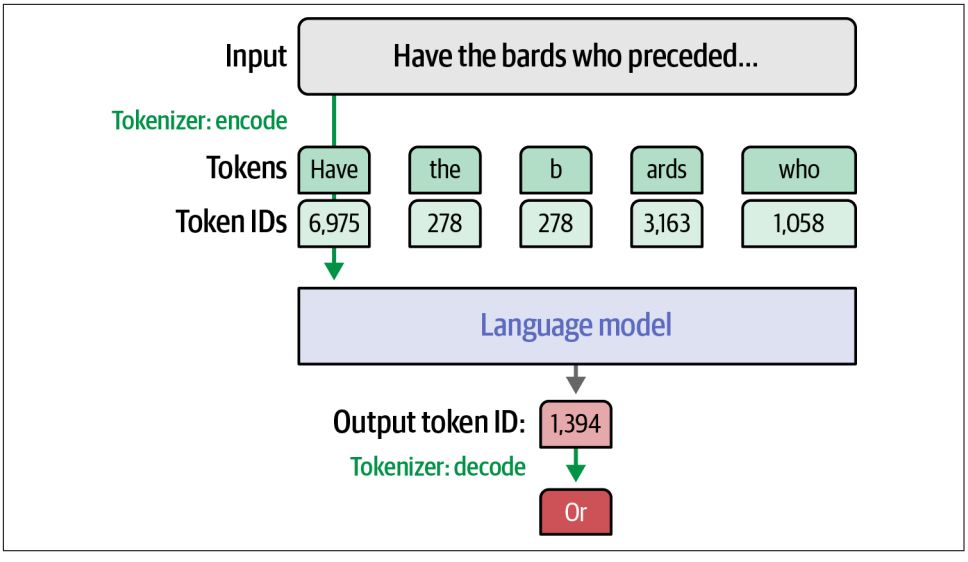

Figure 2-5. Tokenizers are also used to process the output of the model by converting the output token ID into the word or token associated with that ID.

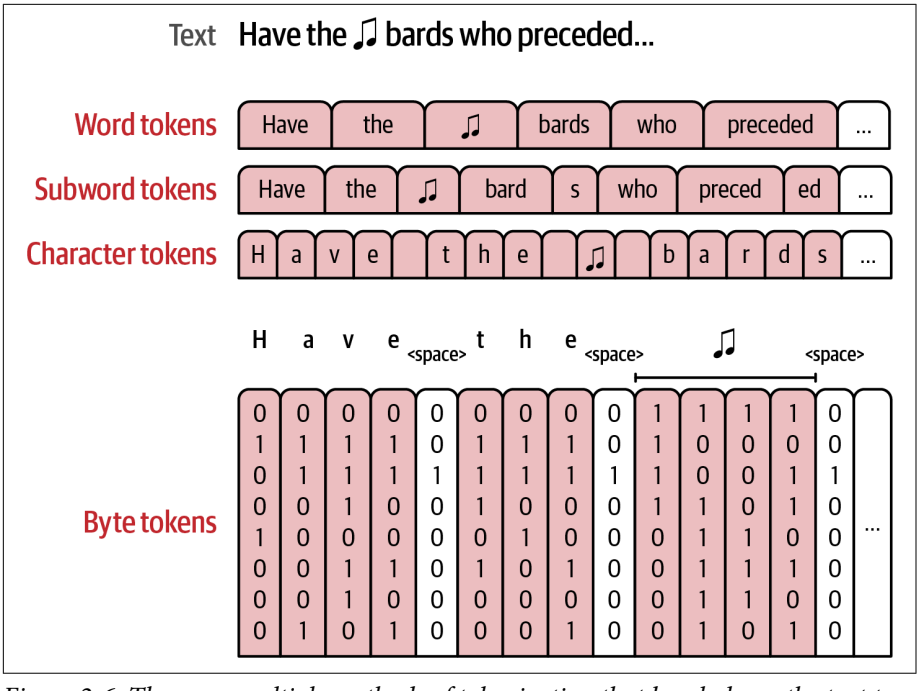

Figure 2-6. There are multiple methods of tokenization that break down the text to different sizes of components (words, subwords, characters, and bytes).

Tokenized Text Models

BERT base model (uncased)	[CLS] english and capital ##ization [UNK] [UNK] show _ token ##s false none eli ##f = = > = else : two tab ##s : " " three tab ##s : " " 12 . 0 * 50 = 600 [SEP]
BERT base model (cased)	[CLS] English and CA ##PI ##TA ##L ##I ##Z ##AT ##ION [UNK] [UNK] show _ token ##s F ##als ##e None el ##if = = > = else : two ta ##bs : " " Three ta ##bs : " " 12 . 0 * 50 = 600 [SEP]
GPT-2	English and CAP ITAL IZ ATION ĠĠĠĠĠĠ show _ t ok ens False None el if == >= else : two tabs :" " Three tabs : " " 12 . 0 * 50 = 600
FLAN-T5	English and CA PI TAL IZ ATION <unk> <unk> show _ to ken s Fal s e None e l if = = > = else : two tab s : " " Three tab s : " " 12. 0 * 50 = 600 </s>
GPT-4	English and CAPITAL IZATION ĠĠĠĠĠĠ show _tokens False None elif == >= else : two tabs :" " Three tabs : " " 12 . 0 * 50 = 600
StarCoder	English and CAPITAL IZATION ĠĠĠĠĠ show _ tokens False None elif == >= else : two tabs :" " Three tabs : " " 1 2 . 0 * 5 0 = 6 0 0
Galactica	English and CAP ITAL IZATION ĠĠĠĠĠĠĠ show _ tokens False None elif == > = else : two t abs : " " Three t abs : " " 1 2 . 0 * 5 0 = 6 0 0
Phi-3 and Llama 2	<s> English and C AP IT AL IZ ATION ĠĠĠĠĠĠĠ show _ to kens False None elif == >= else : two tabs :" " Three tabs : " " 1 2 . 0 * 5 0 = 6 0 0

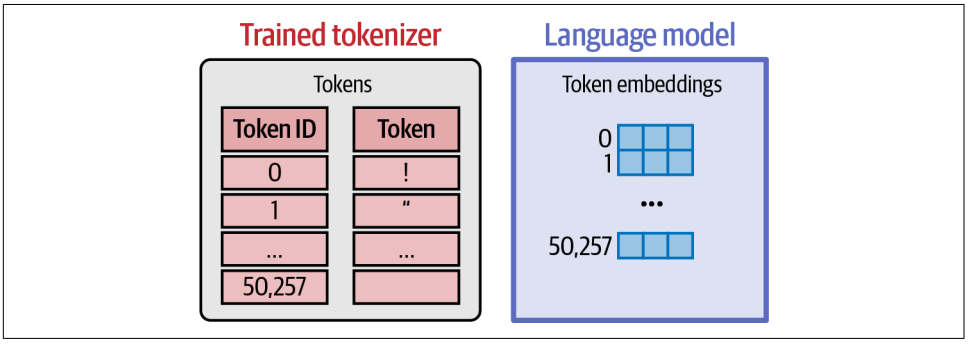

Figure 2-7. A language model holds an embedding vector associated with each token in its tokenizer.

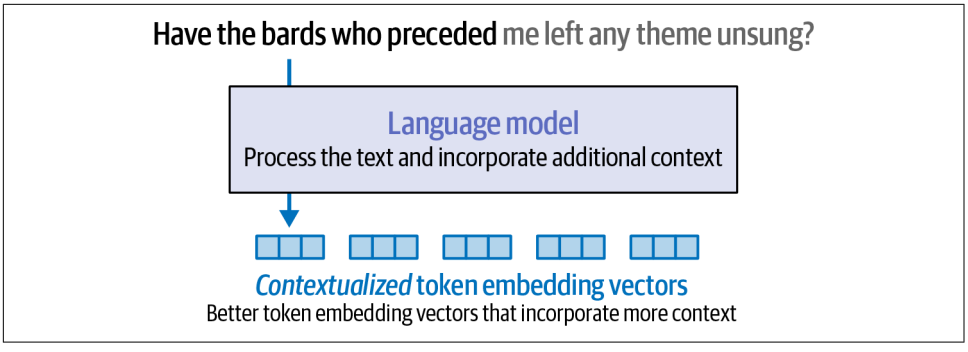

Figure 2-8. Language models produce contextualized token embeddings that improve on raw, static token embeddings.

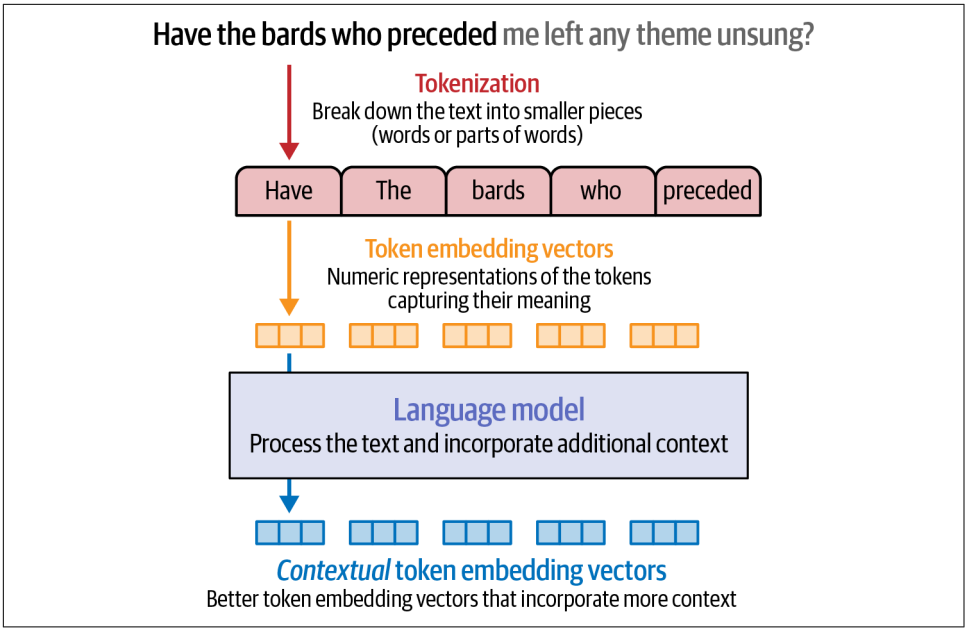

Figure 2-9. A language model operates on raw, static embeddings as its input and produces contextual text embeddings.

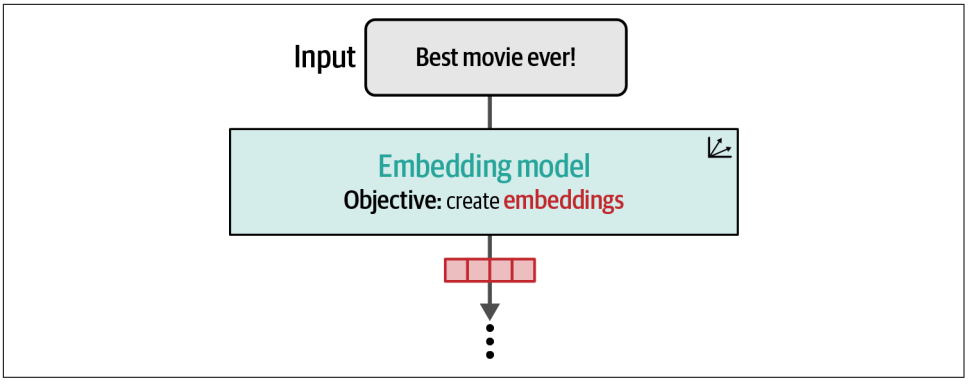

Figure 2-10. In step 1, we use the embedding model to extract the features and convert the input text to embeddings.

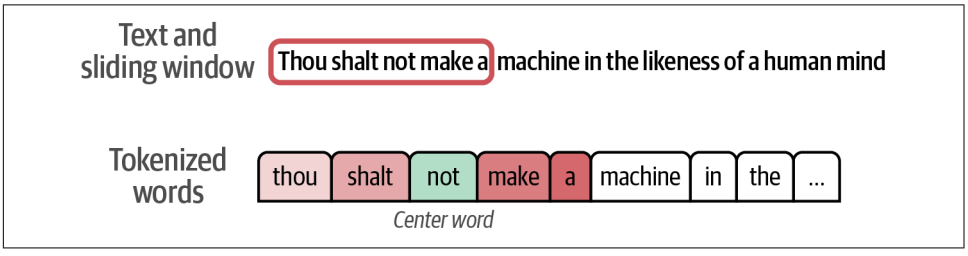

Figure 2-11. A sliding window is used to generate training examples for the word2vec algorithm to later predict if two words are neighbors or not.

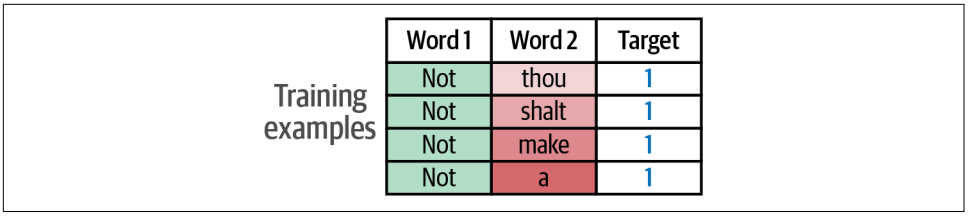

Figure 2-12. Each generated training example shows a pair of neighboring words.

Word 1	Word 2	Target	
not	thou	1	Positive examples
not	shalt	1	
not	make	1	
not	a	1	
thou	apothecary	0	Negative examples
not	sublime	0	
make	def	0	
a	playback	0	

Figure 2-13. We need to present our models with negative examples: words that are not usually neighbors. A better model is able to better distinguish between the positive and negative examples.

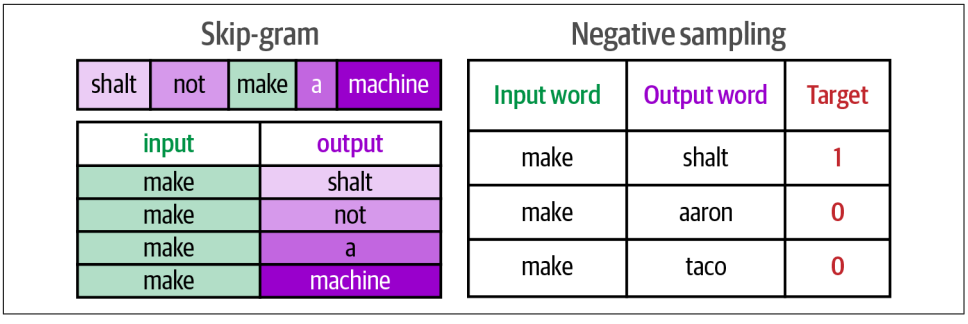

Figure 2-14. Skip-gram and negative sampling are two of the main ideas behind the word2vec algorithm and are useful in many other problems that can be formulated as token sequence problems.

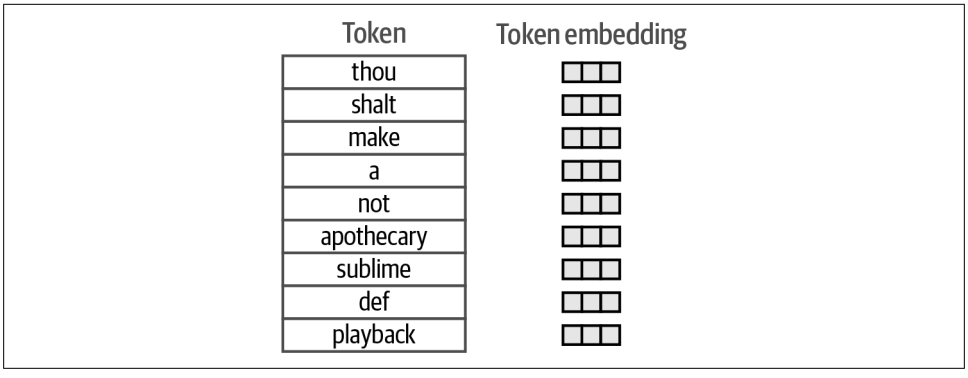

Figure 2-15. A vocabulary of words and their starting, random, uninitialized embedding vectors.

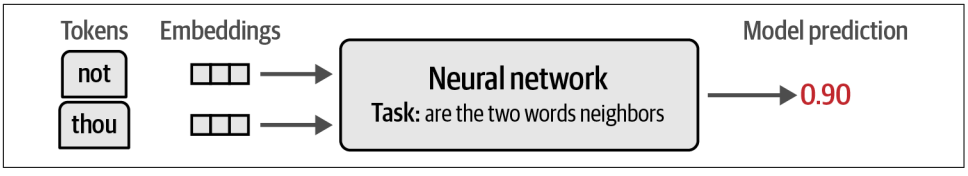

Figure 2-16. A neural network is trained to predict if two words are neighbors. It updates the embeddings in the training process to produce the final, trained embeddings.

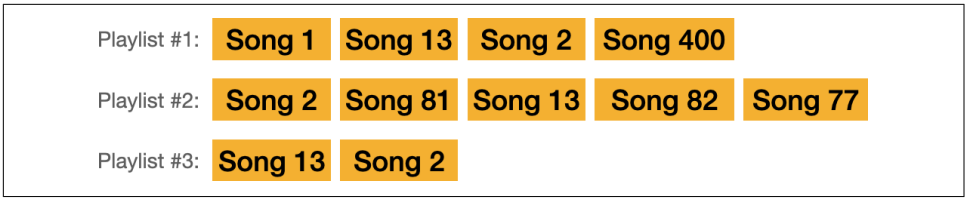

Figure 2-17. For song embeddings that capture song similarity we'll use a dataset made up of a collection of playlists, each containing a list of songs.

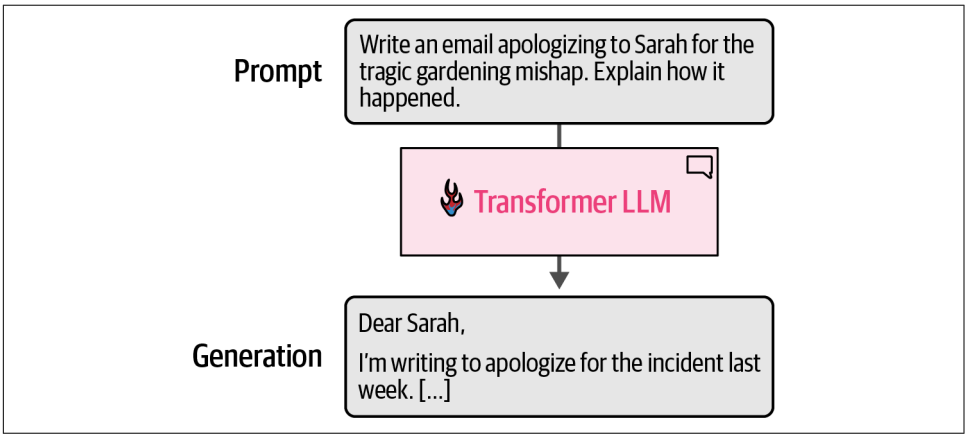

Figure 3-1. At a high level of abstraction, Transformer LLMs take a text prompt and output generated text.

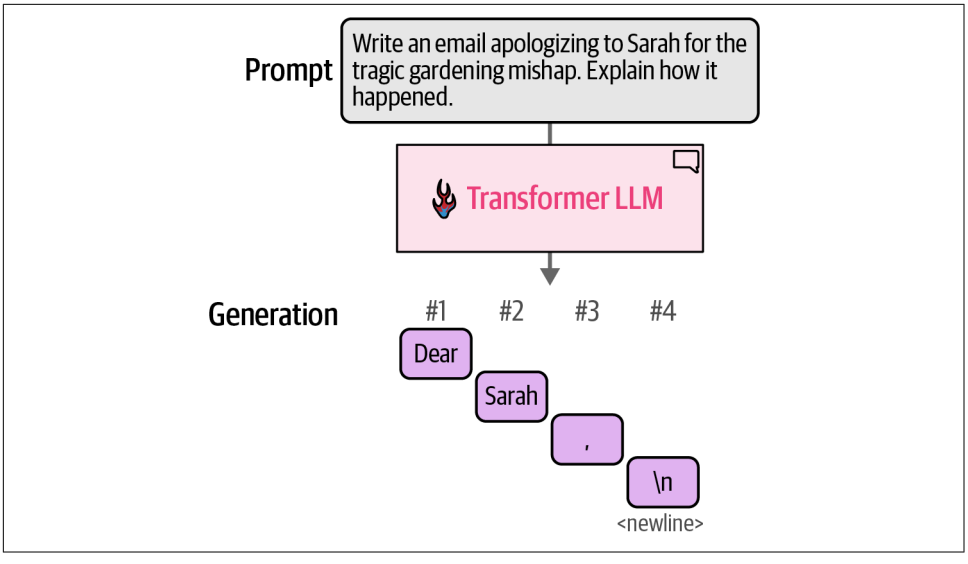

Figure 3-2. Transformer LLMs generate one token at a time, not the entire text at once.

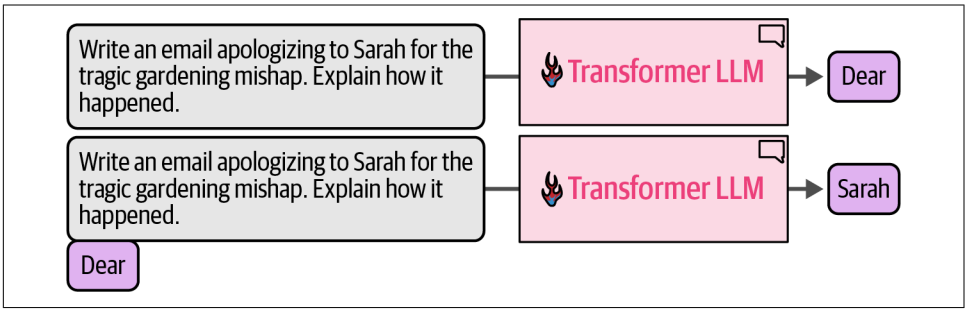

Figure 3-3. An output token is appended to the prompt, then this new text is presented to the model again for another forward pass to generate the next token.

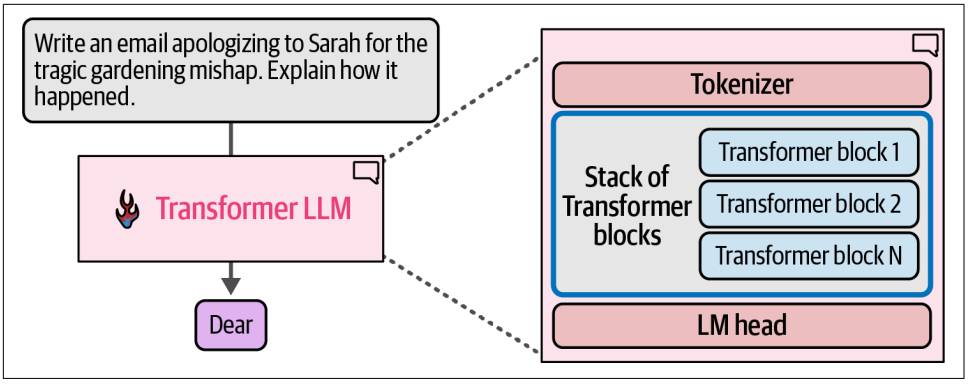

Figure 3-4. A Transformer LLM is made up of a tokenizer, a stack of Transformer blocks, and a language modeling head.

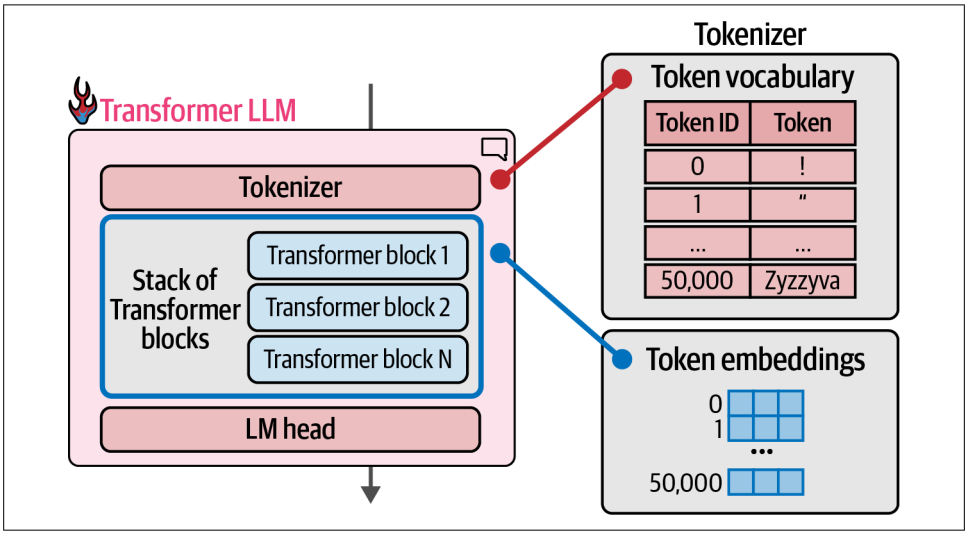

Figure 3-5. The tokenizer has a vocabulary of 50,000 tokens. The model has token embeddings associated with those embeddings.

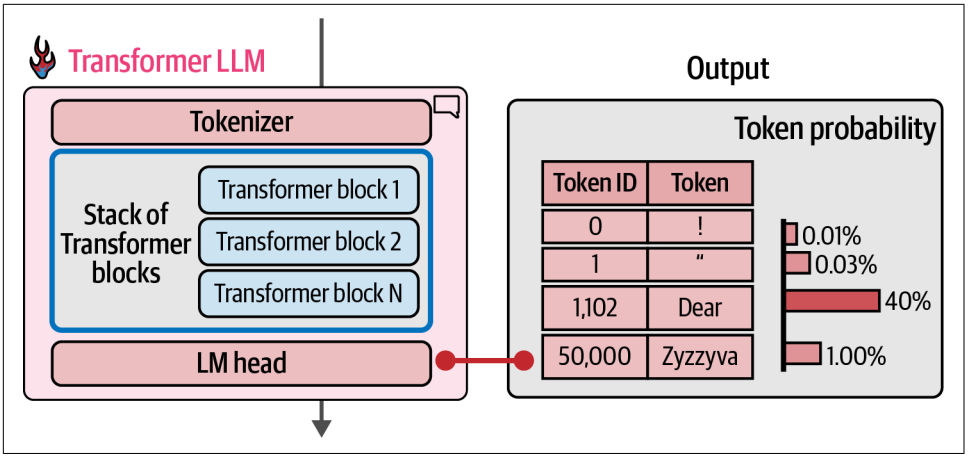

Figure 3-6. At the end of the forward pass, the model predicts a probability score for each token in the vocabulary.

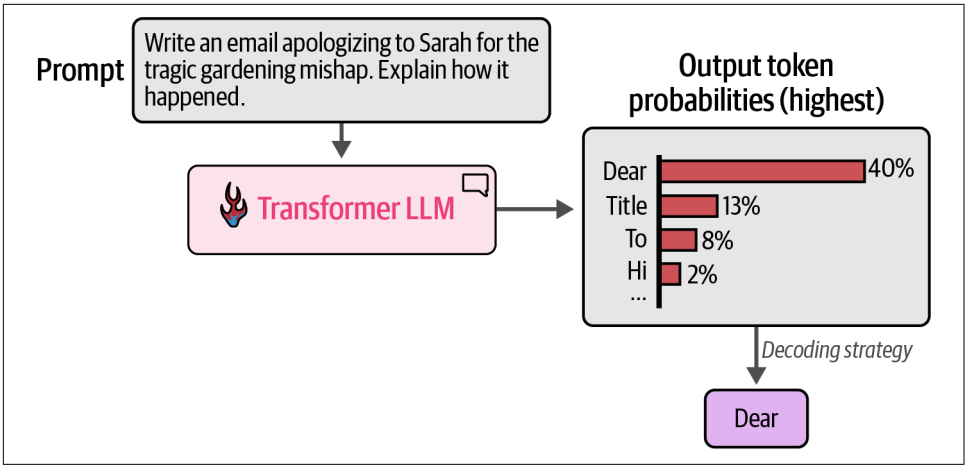

Figure 3-7. The tokens with the highest probability after the model's forward pass. Our decoding strategy decides which of the tokens to output by sampling based on the probabilities.

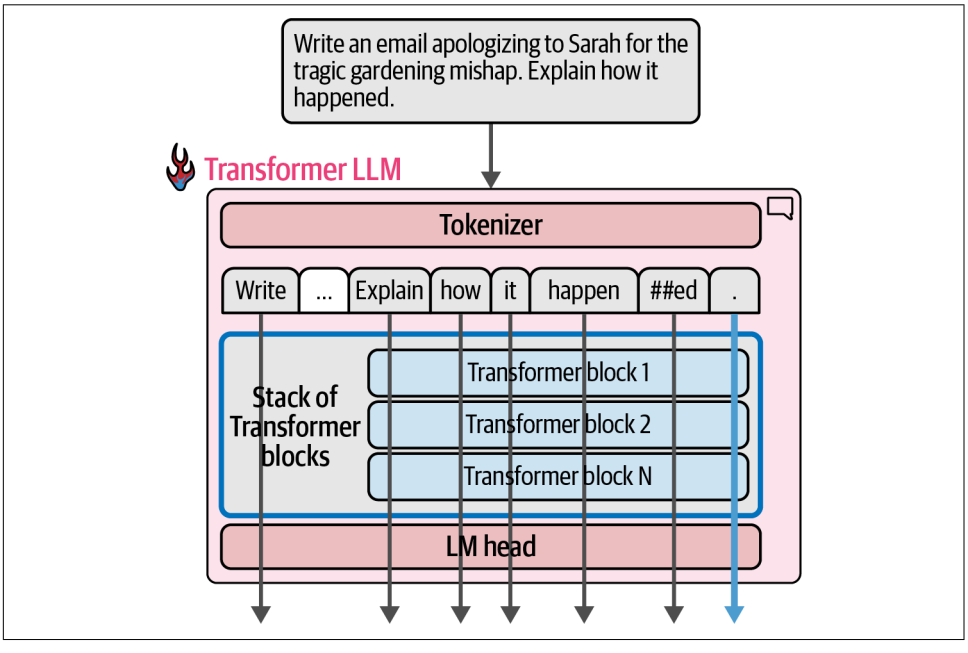

Figure 3-8. Each token is processed through its own stream of computation (with some interaction between them in attention steps, as we'll later see).

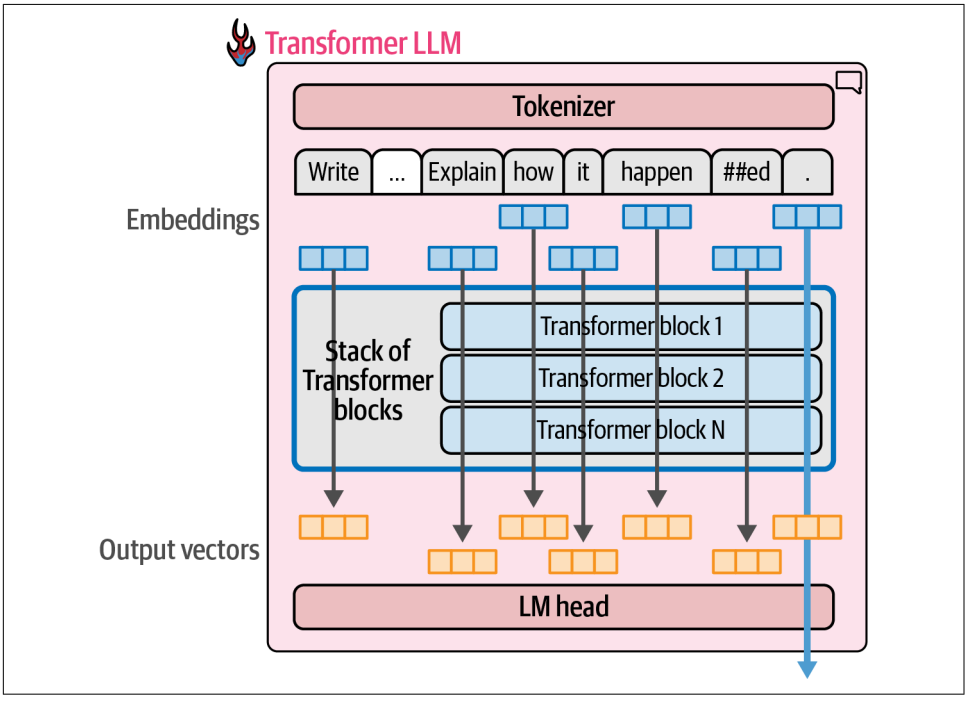

Figure 3-9. Each processing stream takes a vector as input and produces a final resulting vector of the same size (often referred to as the model dimension).

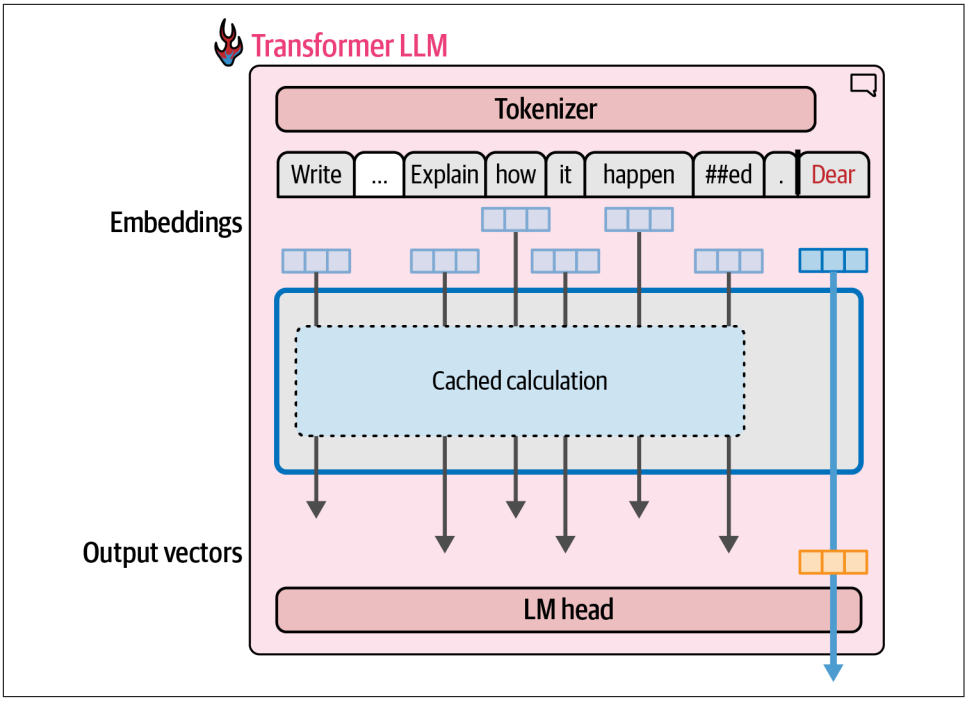

Figure 3-10. When generating text, it's important to cache the computation results of previous tokens instead of repeating the same calculation over and over again.

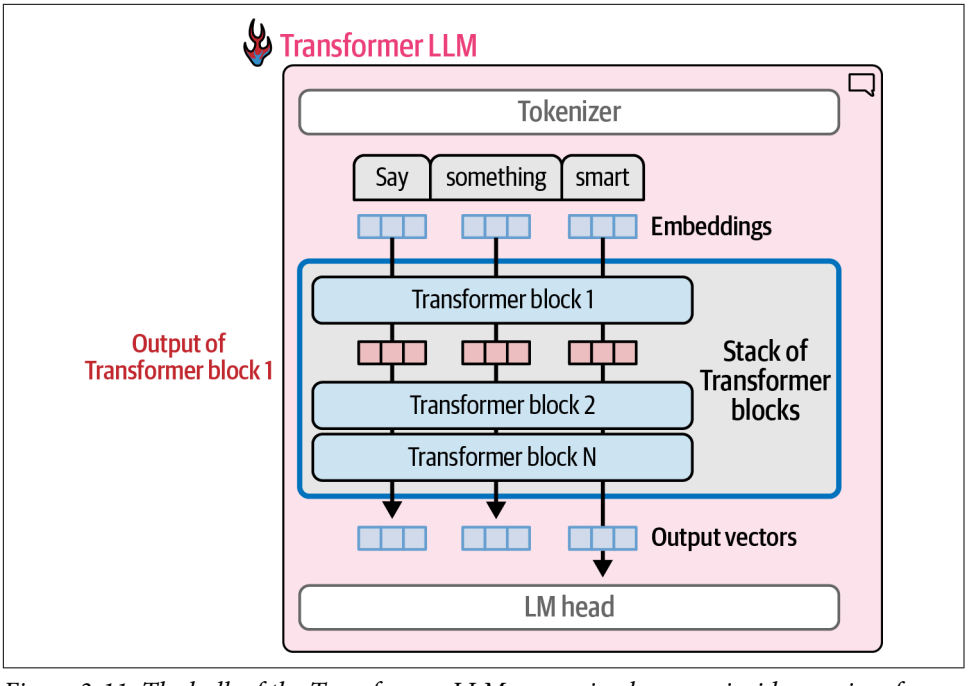

Figure 3-11. The bulk of the Transformer LLM processing happens inside a series of Transformer blocks, each handing the result of its processing as input to the subsequent block.

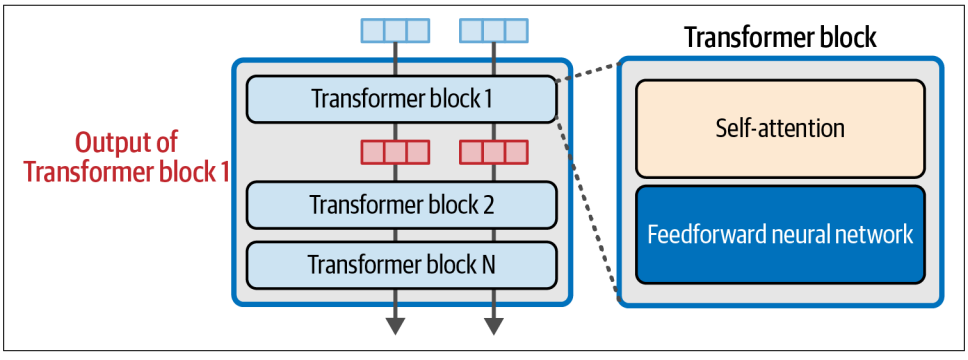

Figure 3-12. A Transformer block is made up of a self-attention layer and a feedforward neural network.

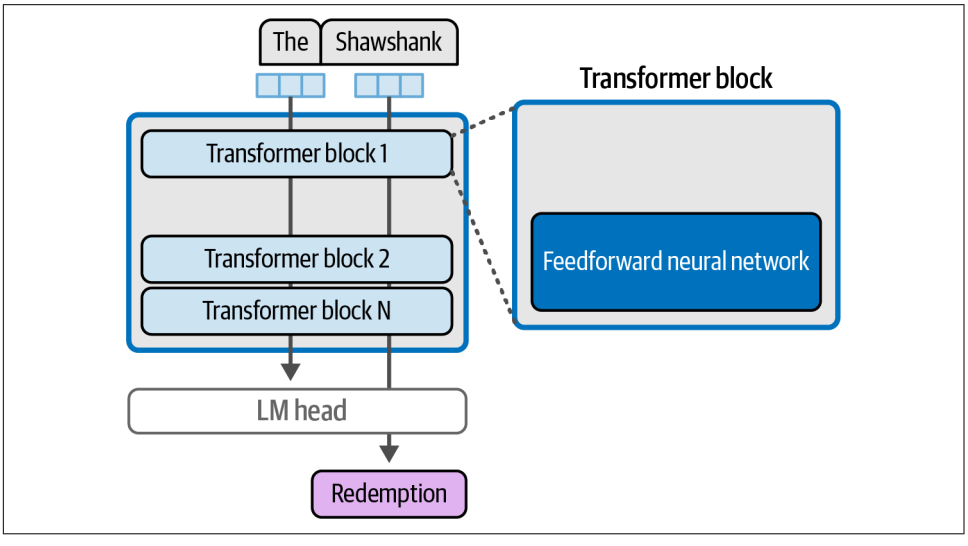

Figure 3-13. The feedforward neural network component of a Transformer block likely does the majority of the model's memorization and interpolation.

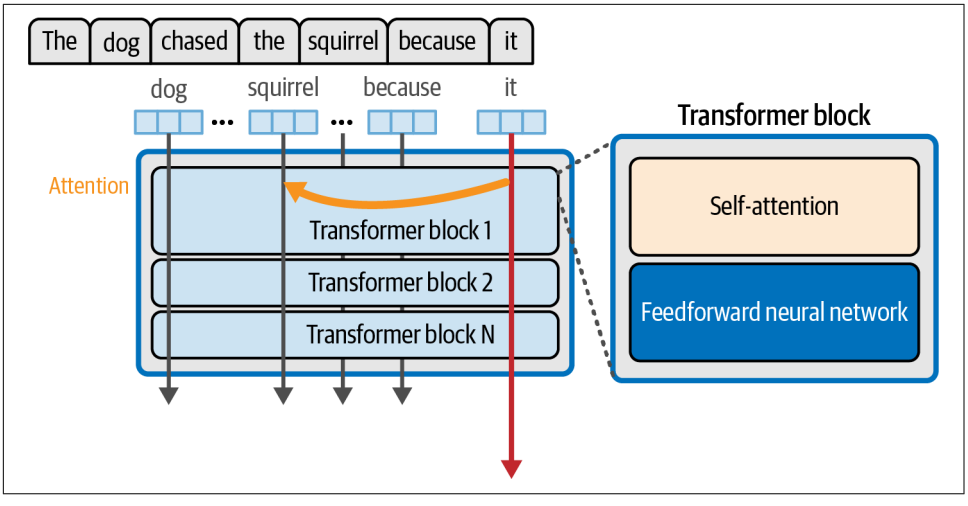

Figure 3-14. The self-attention layer incorporates relevant information from previous positions that help process the current token.

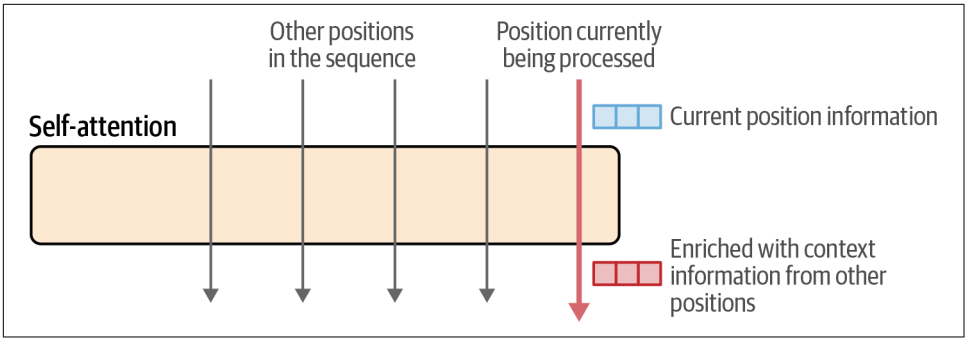

Figure 3-15. A simplified framing of attention: an input sequence and a current position being processed. As we're mainly concerned with this position, the figure shows an input vector and an output vector that incorporates information from the previous elements in the sequence according to the attention mechanism.

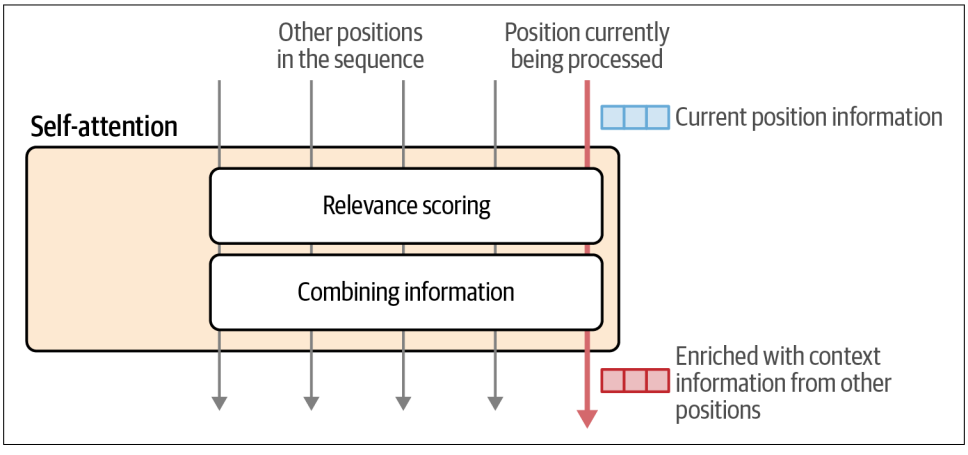

Figure 3-16. Attention is made up of two major steps: relevance scoring for each position, then a step where we combine the information based on those scores.

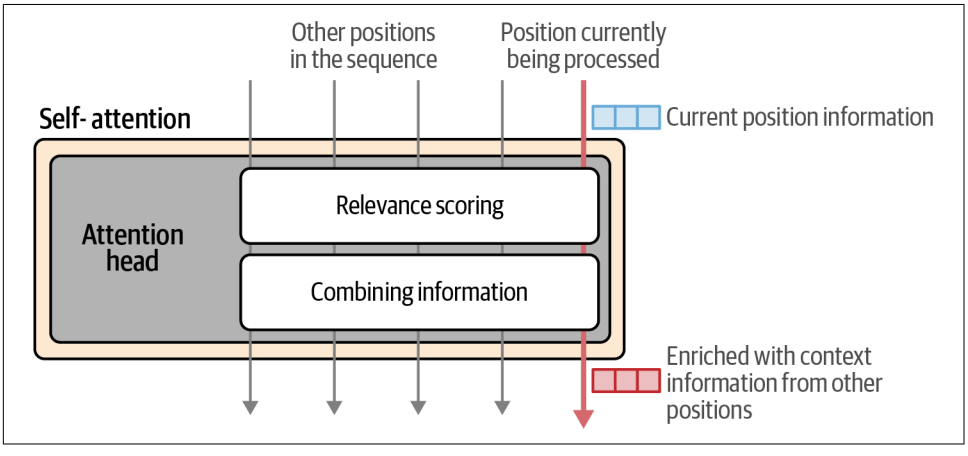

Figure 3-17. We get better LLMs by doing attention multiple times in parallel, increasing the model's capacity to attend to different types of information.

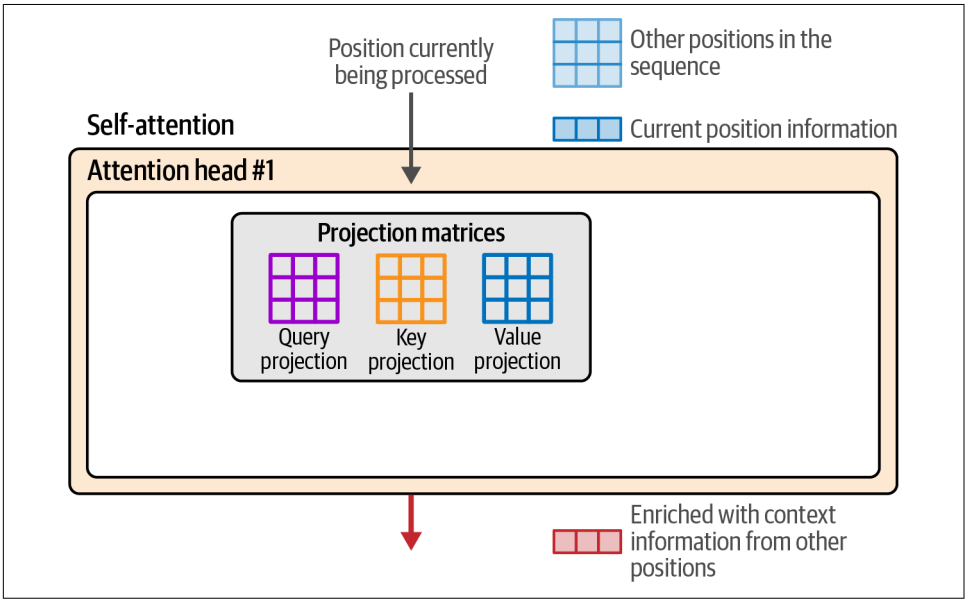

Figure 3-18. Before starting the self-attention calculation, we have the inputs to the layer and projection matrices for queries, keys, and values.

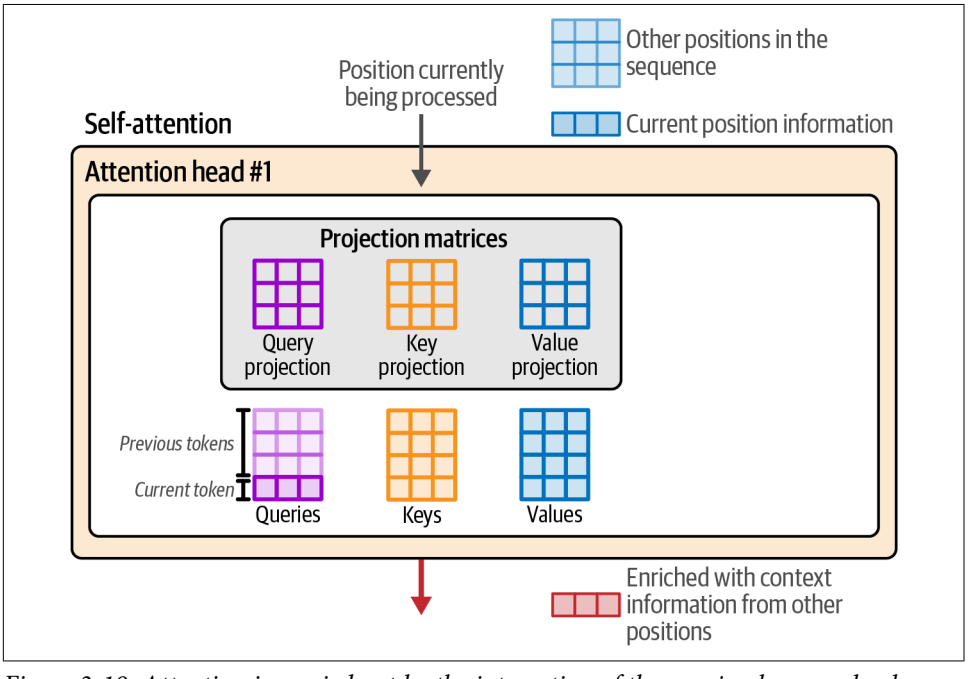

Figure 3-19. Attention is carried out by the interaction of the queries, keys, and values matrices. Those are produced by multiplying the layer's inputs with the projection matrices.

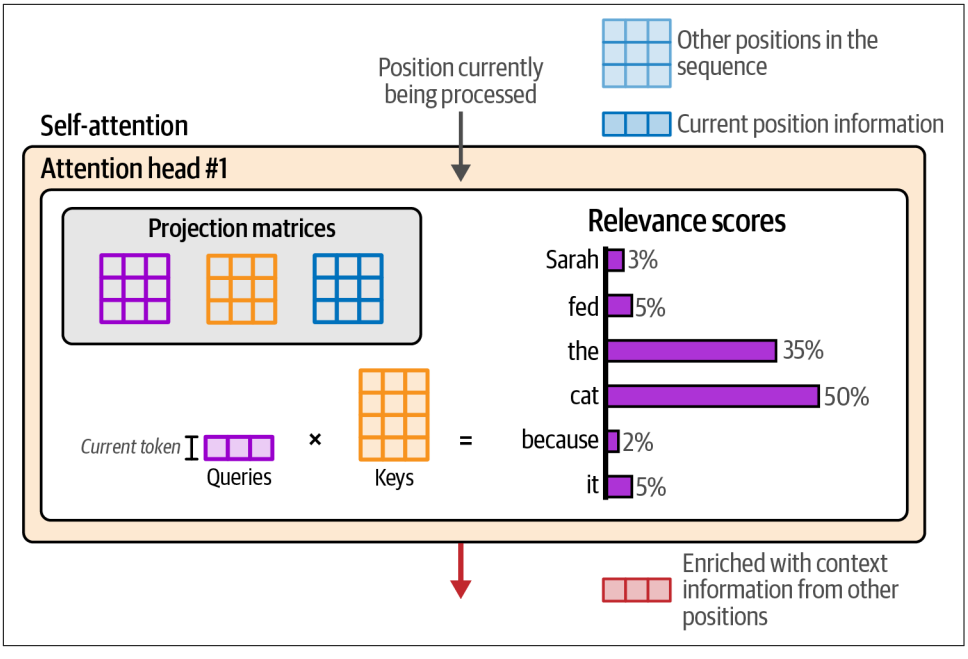

Figure 3-20. Scoring the relevance of previous tokens is accomplished by multiplying the query associated with the current position with the keys matrix.

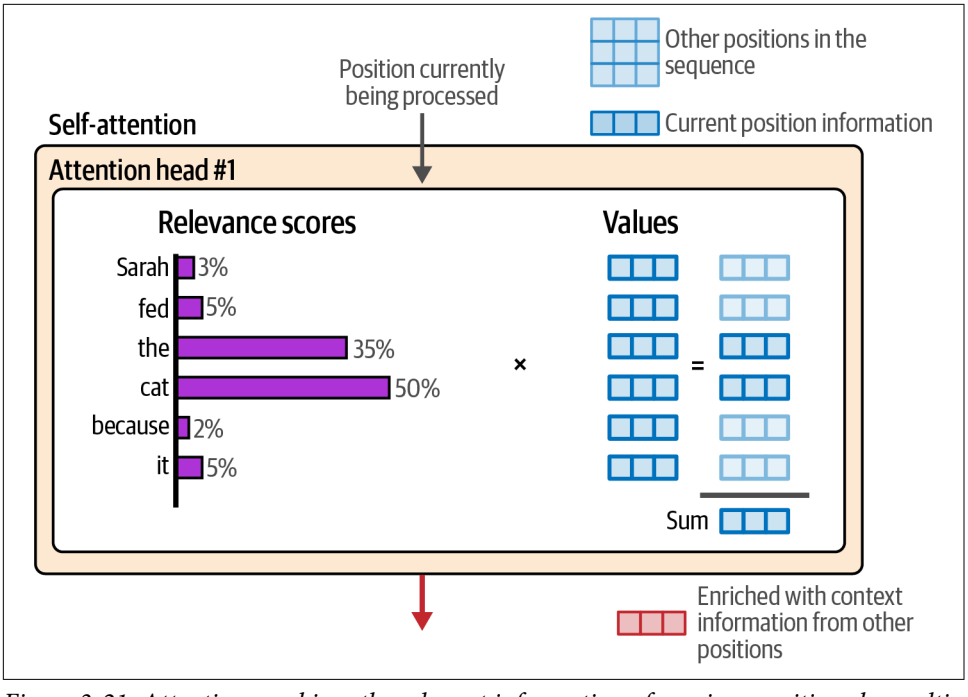

Figure 3-21. Attention combines the relevant information of previous positions by multiplying their relevance scores by their respective value vectors.

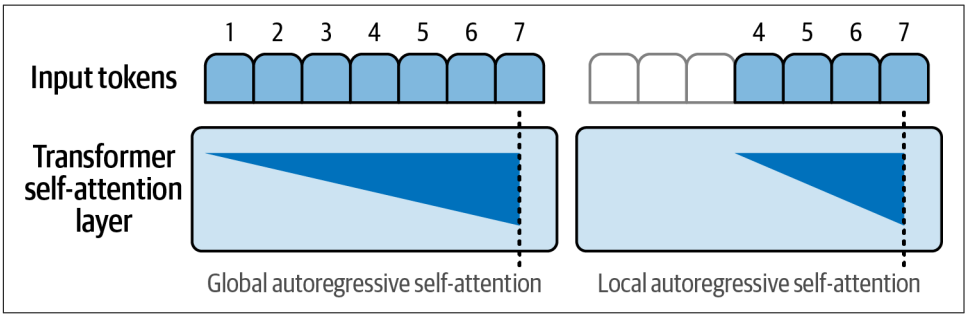

Figure 3-22. Local attention boosts performance by only paying attention to a small number of previous positions.

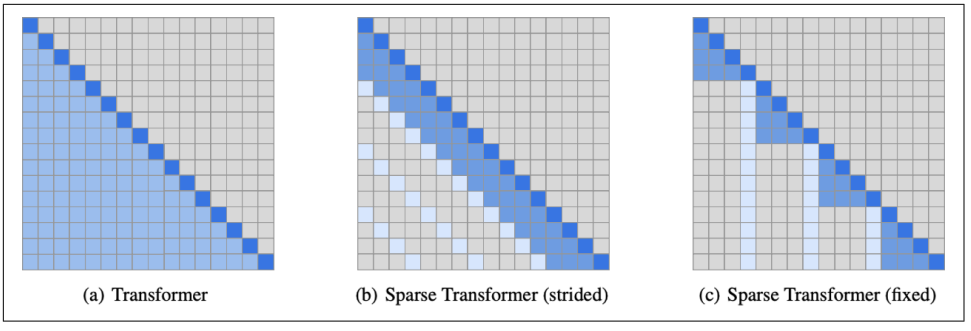

Figure 3-23. Full attention versus sparse attention. Figure 3-24 explains the coloring. (Source: "Generating long sequences with sparse transformers".)

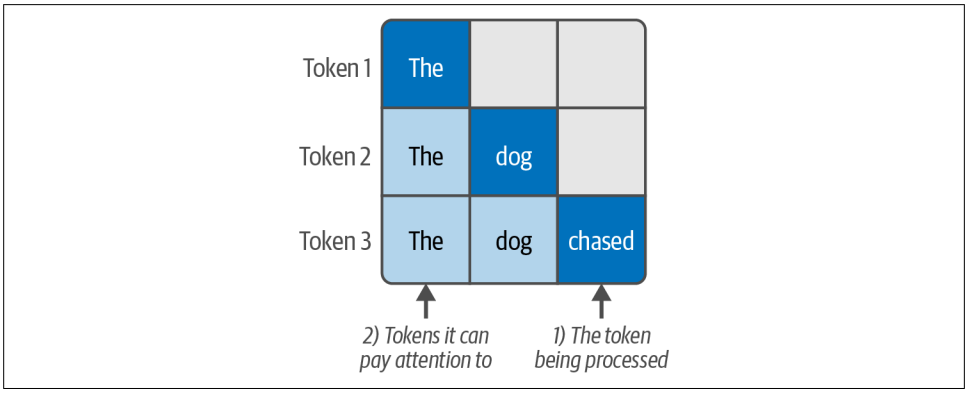

Figure 3-24. Attention figures show which token is being processed, and which previous tokens an attention mechanism allows it to attend to.

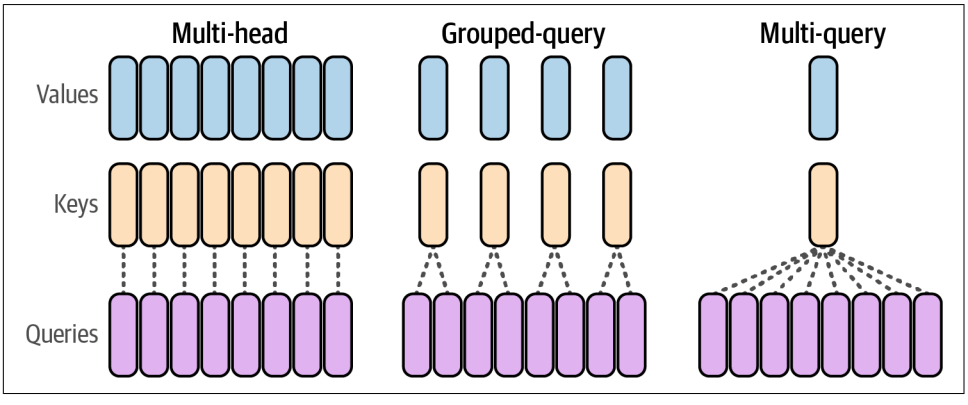

Figure 3-25. A comparison of different kinds of attention: the original multi-head, grouped-query attention, and multi-query attention (source: "Fast transformer decoding: One write-head is all you need").

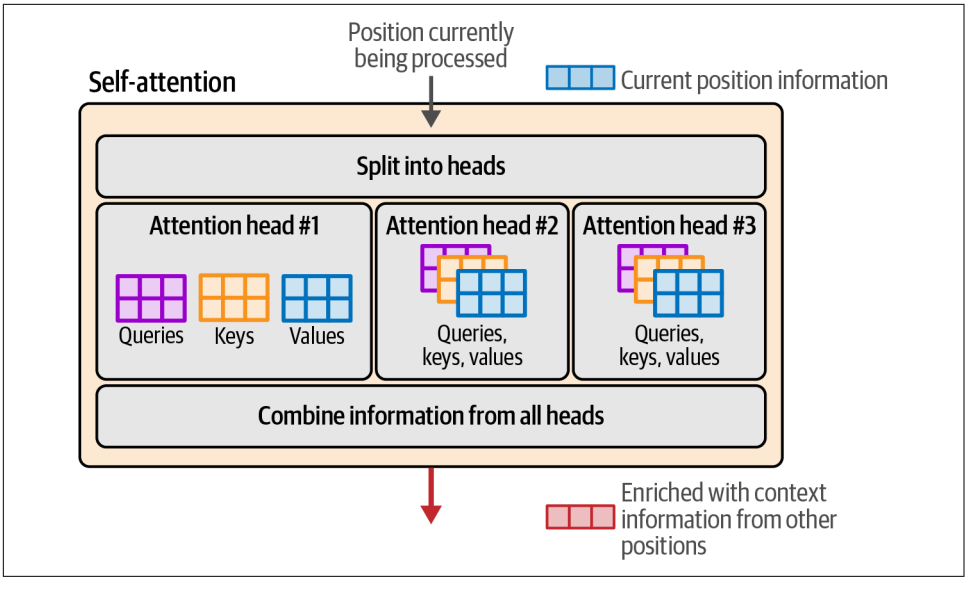

Figure 3-26. Attention is conducted using matrices of queries, keys, and values. In multi-head attention, each head has a distinct version of each of these matrices.

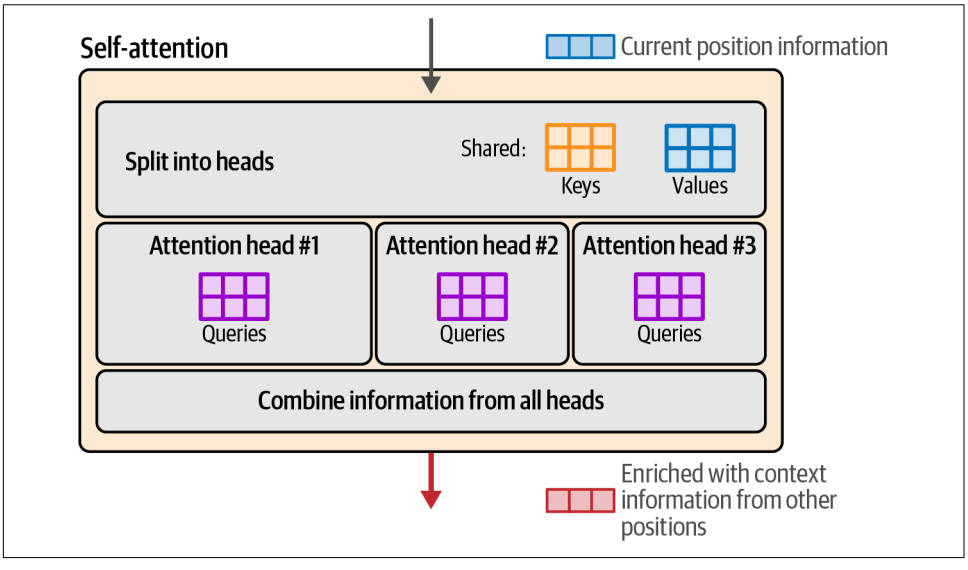

Figure 3-27. Multi-query attention presents a more efficient attention mechanism by sharing the keys and values matrices across all the attention heads.

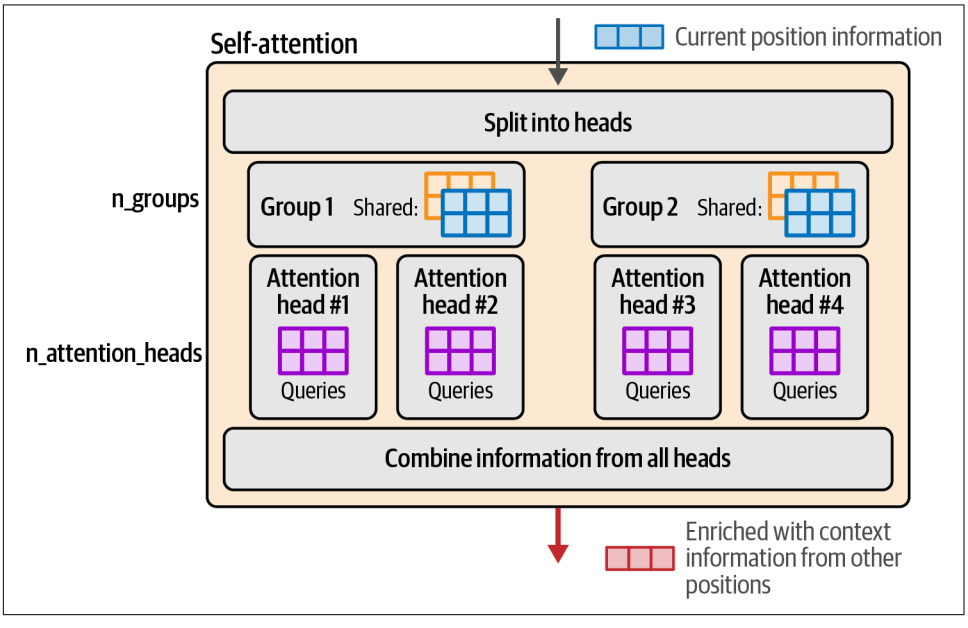

Figure 3-28. Grouped-query attention sacrifices a little bit of the efficiency of multi-query attention in return for a large improvement in quality by allowing multiple groups of shared key/value matrices; each group has its respective set of attention heads.

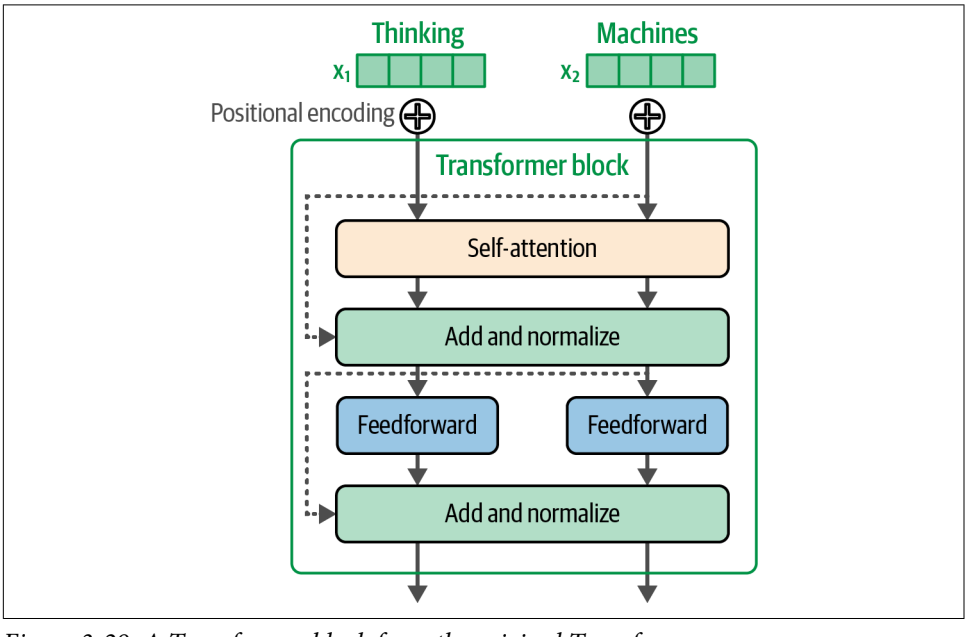

Figure 3-29. A Transformer block from the original Transformer paper.

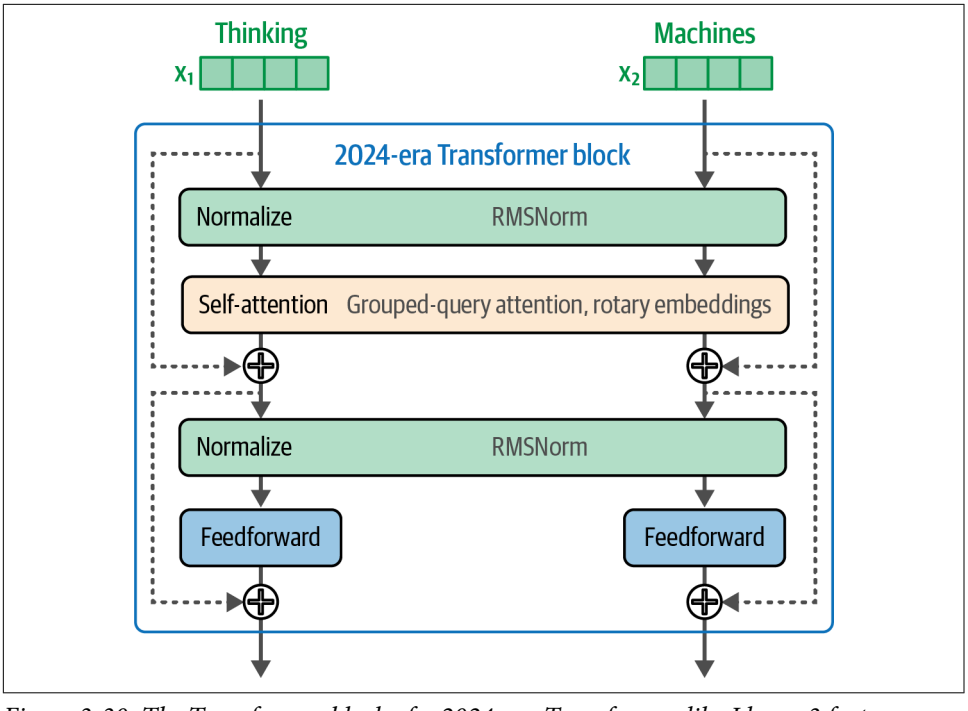

Figure 3-30. The Transformer block of a 2024-era Transformer like Llama 3 features some tweaks like pre-normalization and an attention optimized with grouped-query attention and rotary embeddings.

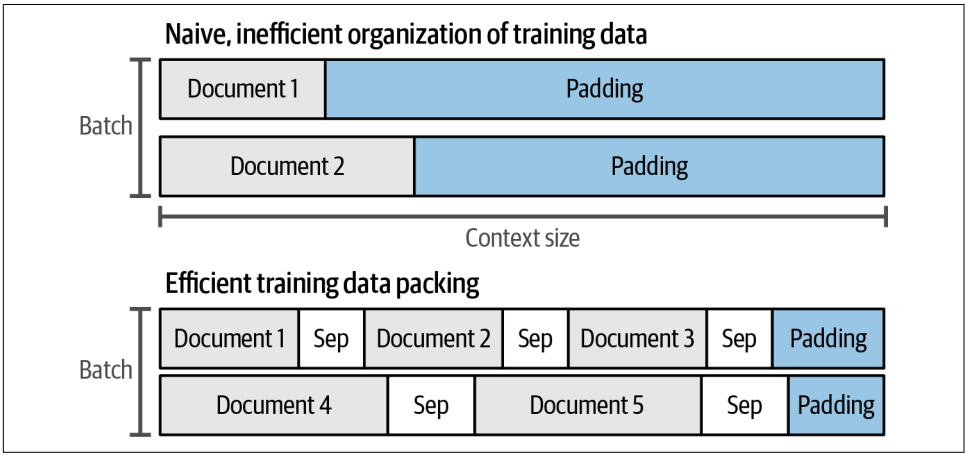

Figure 3-31. Packing is the process of efficiently organizing short training documents into the context. It includes grouping multiple documents in a single context while minimizing the padding at the end of the context.

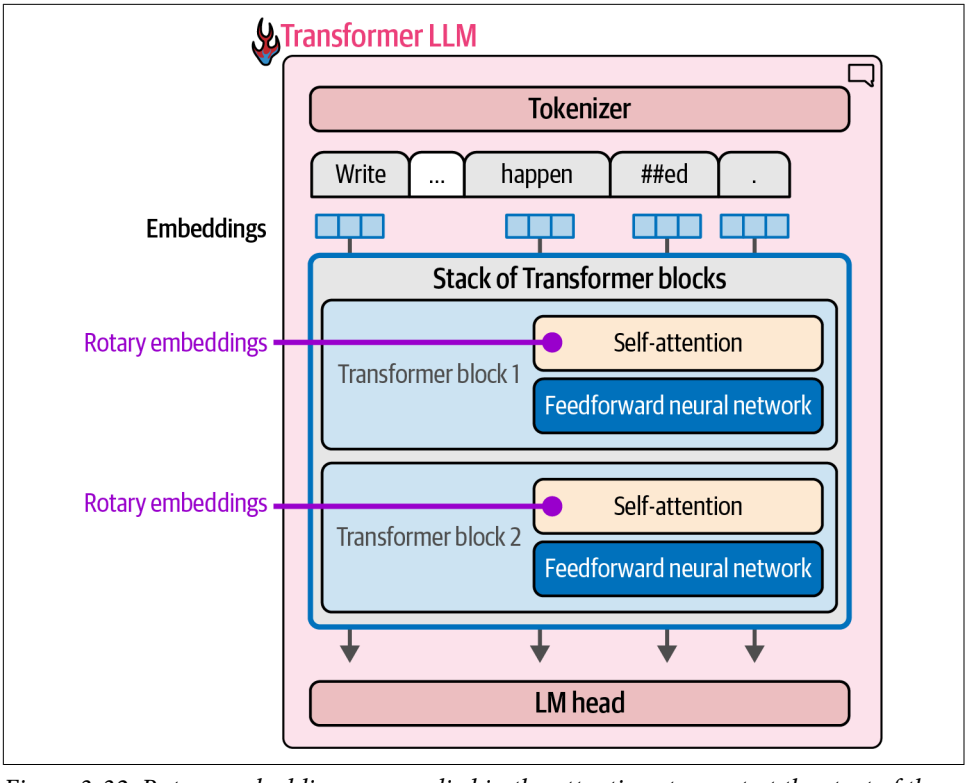

Figure 3-32. Rotary embeddings are applied in the attention step, not at the start of the forward pass.

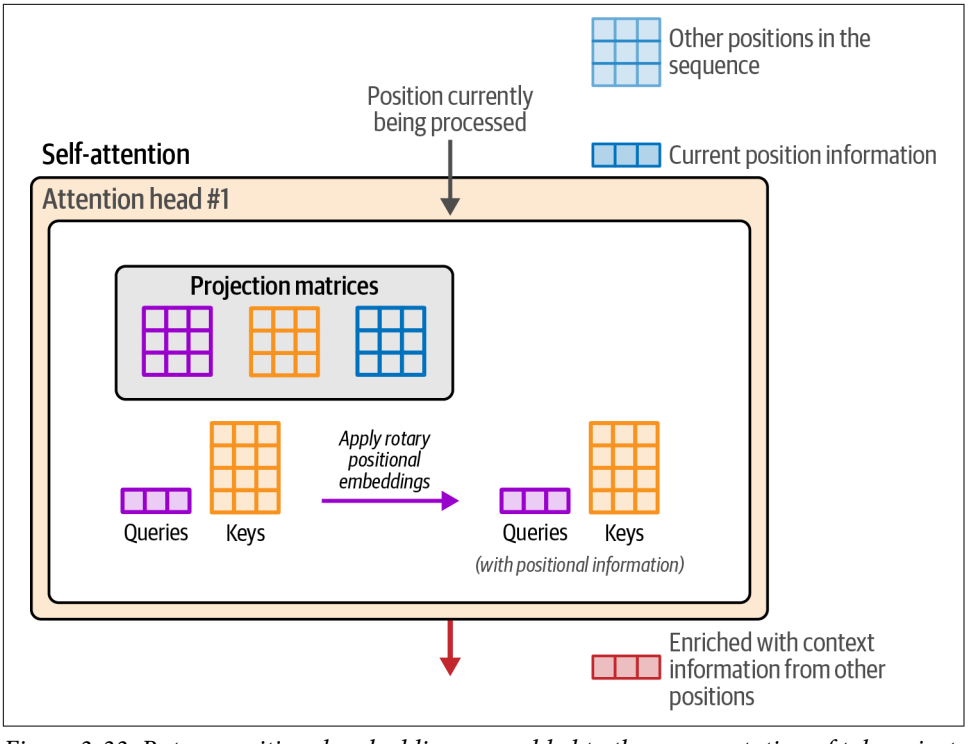

Figure 3-33. Rotary positional embeddings are added to the representation of tokens just before the relevance scoring step in self-attention.

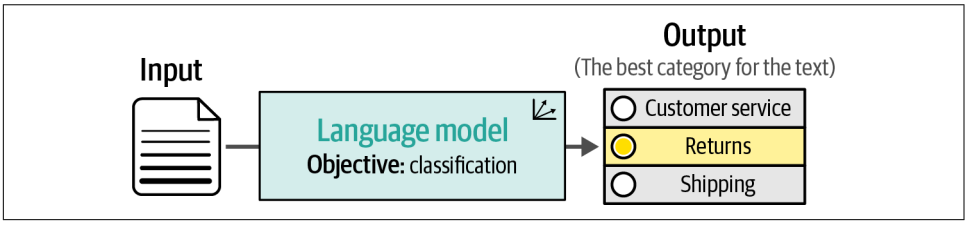

Figure 4-1. Using a language model to classify text.

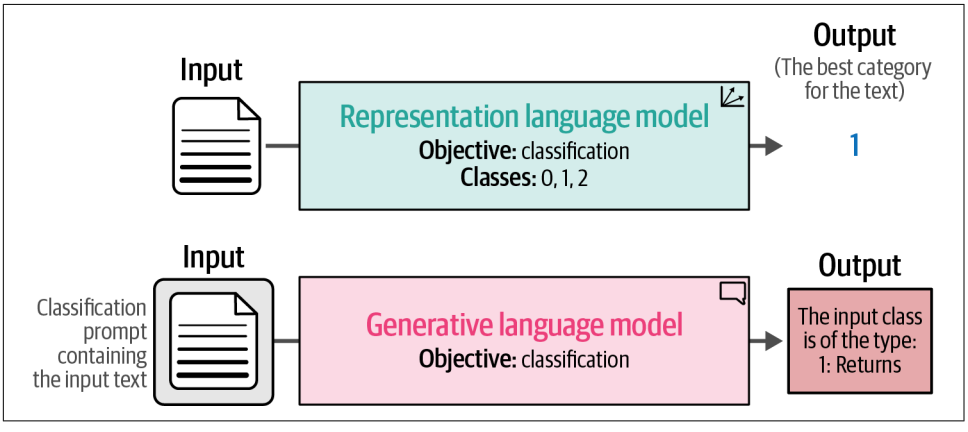

Figure 4-2. Although both representation and generative models can be used for classification, their approaches differ.

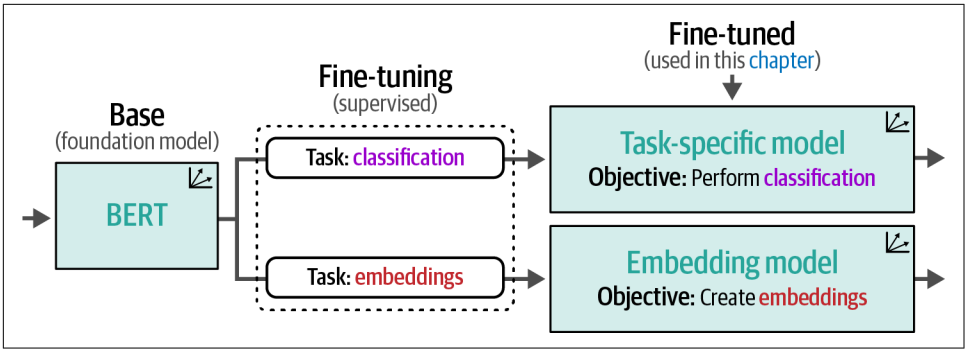

Figure 4-3. A foundation model is fine-tuned for specific tasks; for instance, to perform classification or generate general-purpose embeddings.

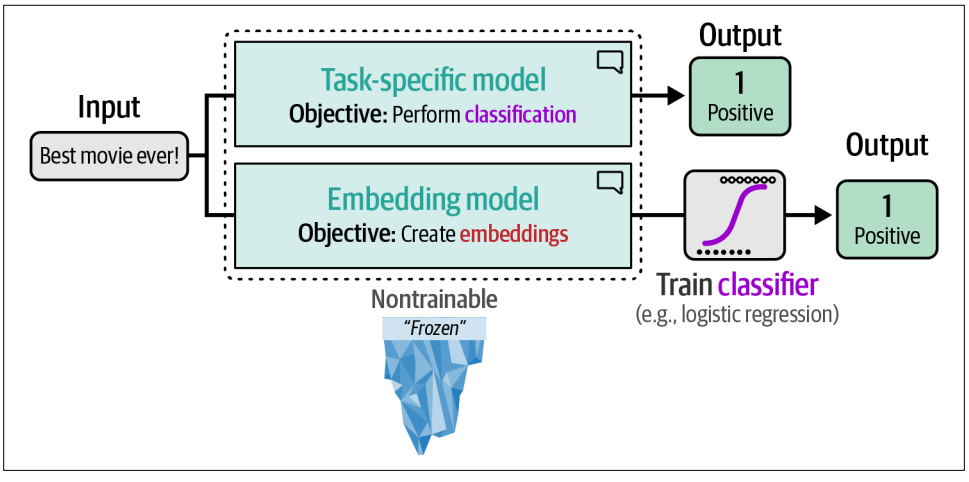

Figure 4-4. Perform classification directly with a task-specific model or indirectly with general-purpose embeddings.

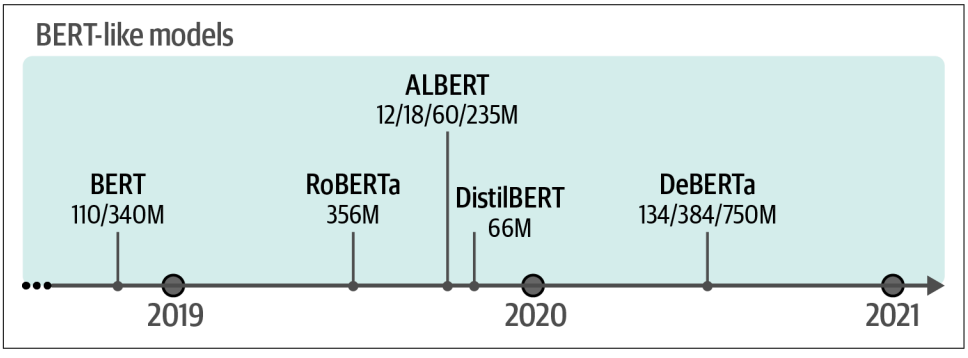

Figure 4-5. A timeline of common BERT-like model releases. These are considered foundation models and are mostly intended to be fine-tuned on a downstream task.

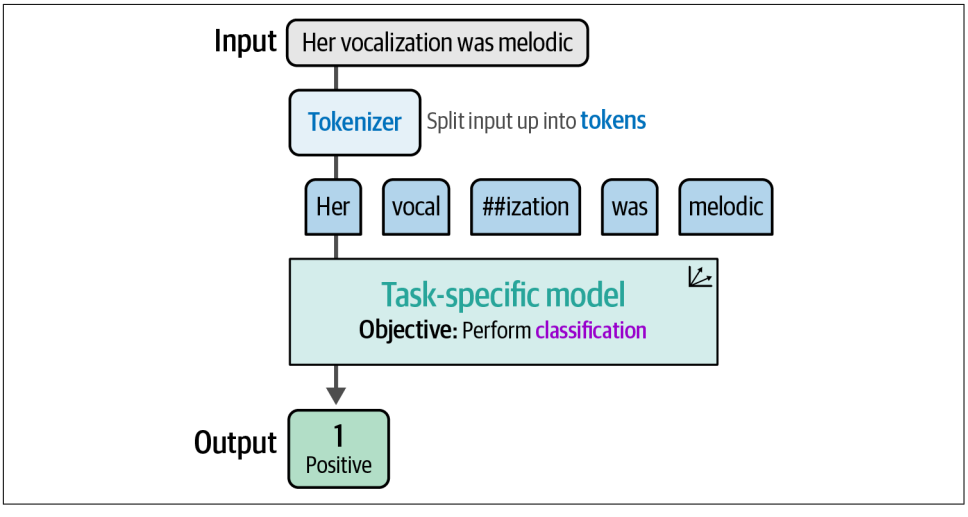

Figure 4-6. An input sentence is first fed to a tokenizer before it can be processed by the task-specific model.

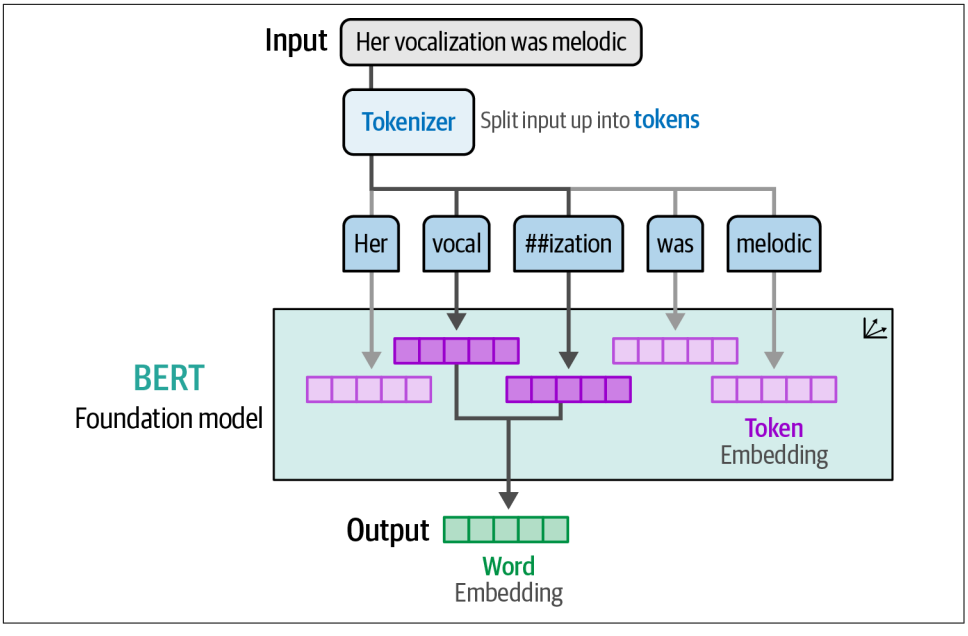

Figure 4-7. By breaking down an unknown word into tokens, word embeddings can still be generated.

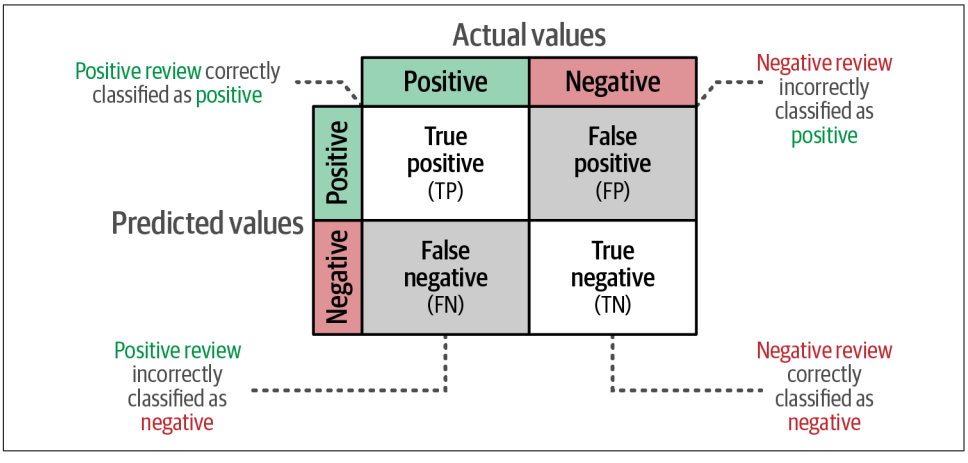

Figure 4-8. The confusion matrix describes four types of predictions we can make.

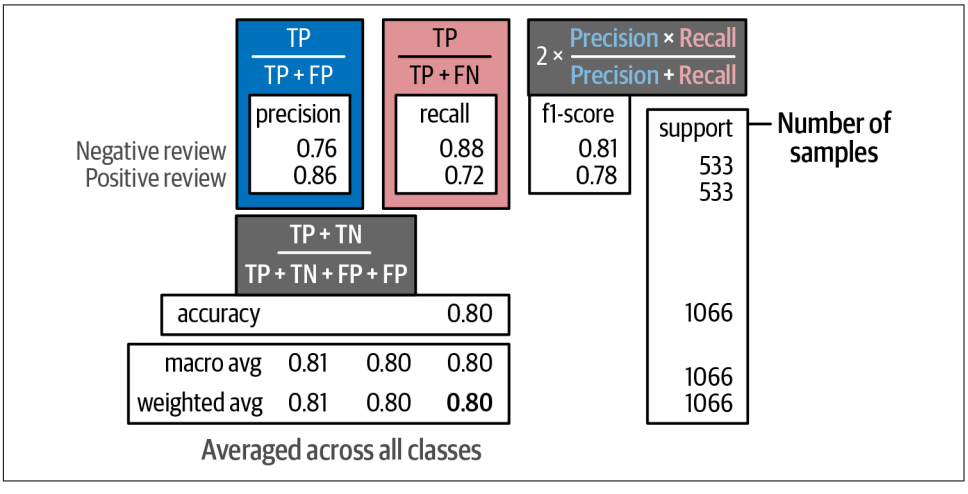

Figure 4-9. The classification report describes several metrics for evaluating a model's performance.

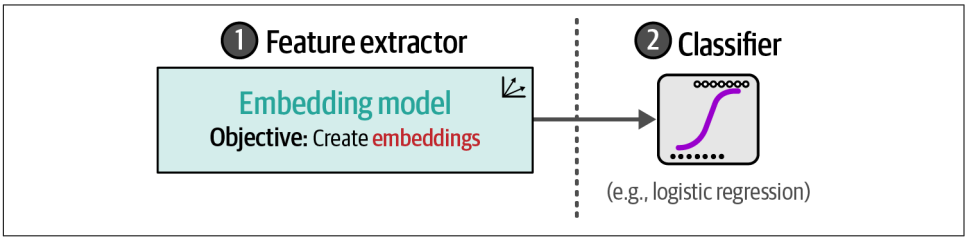

Figure 4-10. The feature extraction step and classification steps are separated.

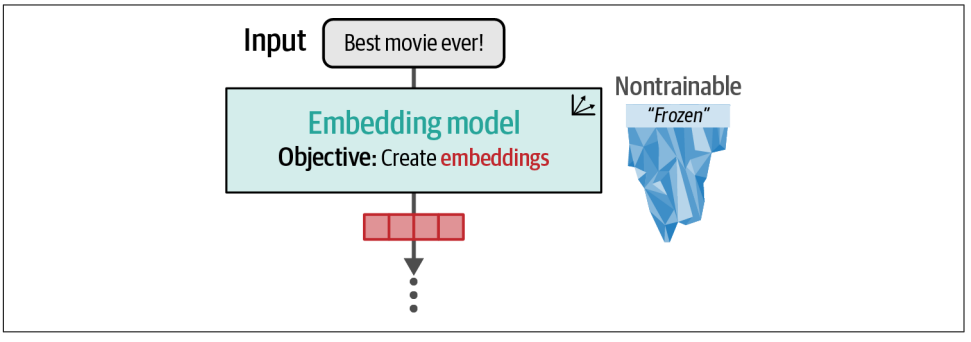

Figure 4-11. In step 1, we use the embedding model to extract the features and convert the input text to embeddings.

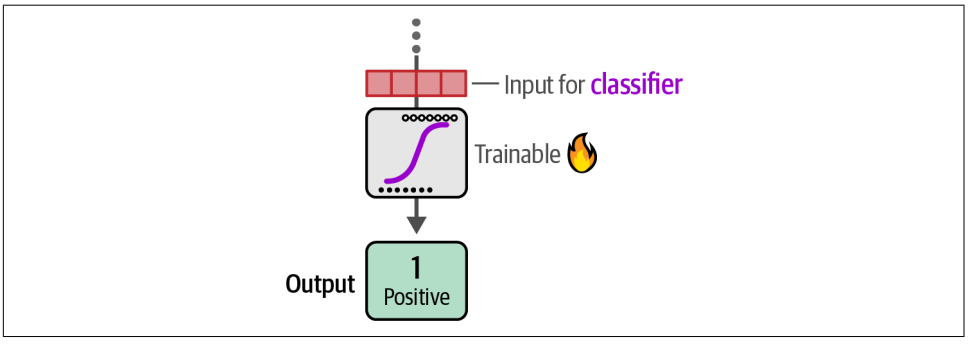

Figure 4-12. Using the embeddings as our features, we train a logistic regression model on our training data.

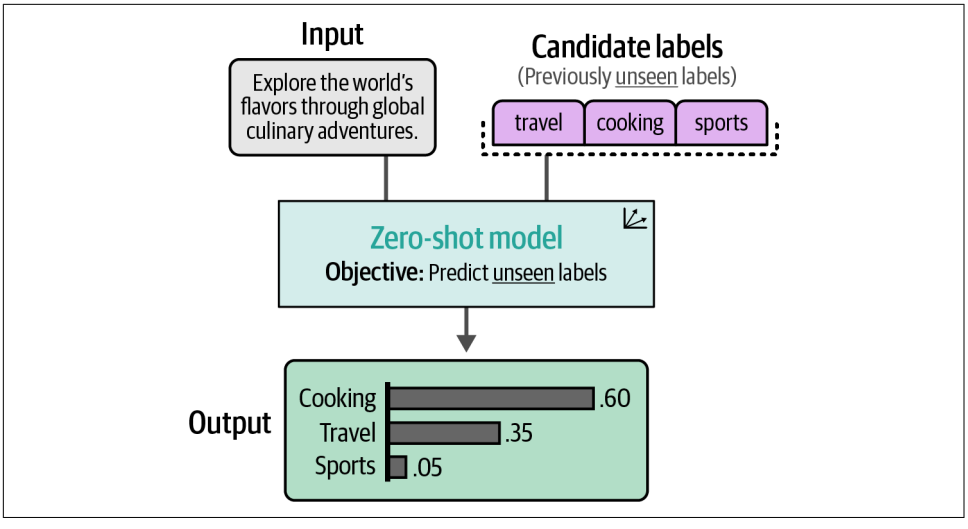

Figure 4-13. In zero-shot classification, we have no labeled data, only the labels themselves. The zero-shot model decides how the input is related to the candidate labels.

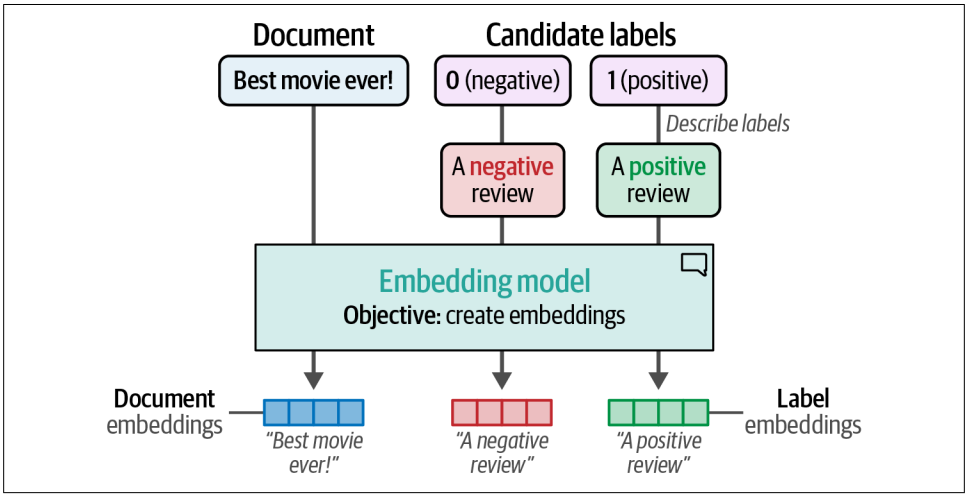

Figure 4-14. To embed the labels, we first need to give them a description, such as "a negative movie review." This can then be embedded through sentence-transformers.

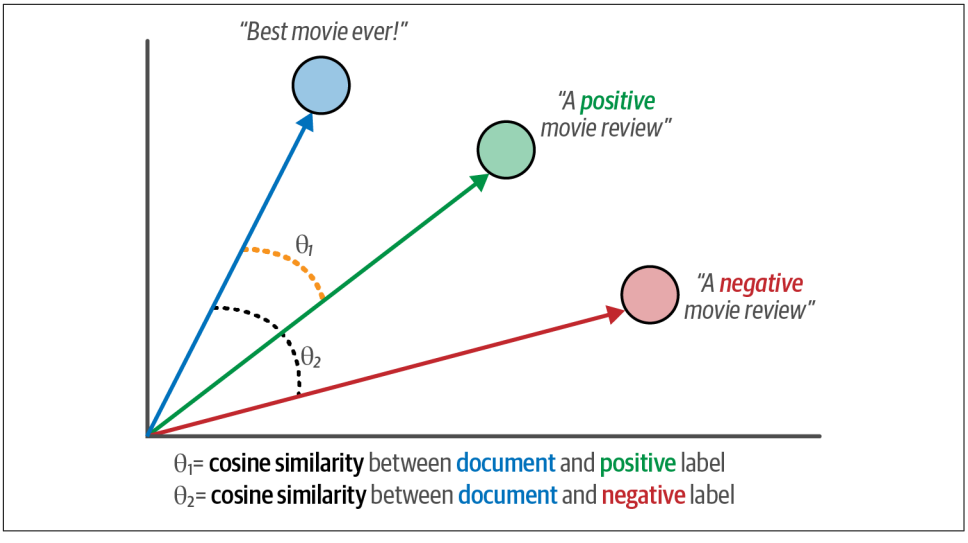

Figure 4-15. The cosine similarity is the angle between two vectors or embeddings. In this example, we calculate the similarity between a document and the two possible labels, positive and negative.

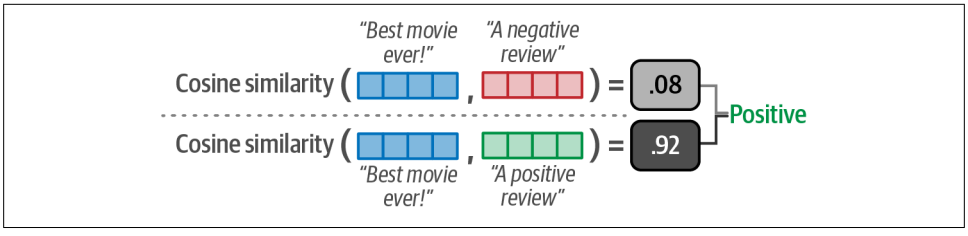

Figure 4-16. After embedding the label descriptions and the documents, we can use cosine similarity for each label document pair.

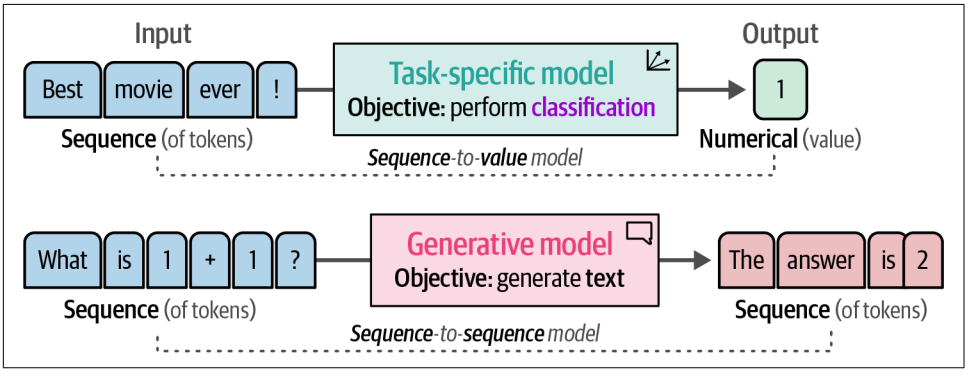

Figure 4-17. A task-specific model generates numerical values from sequences of tokens while a generative model generates sequences of tokens from sequences of tokens.

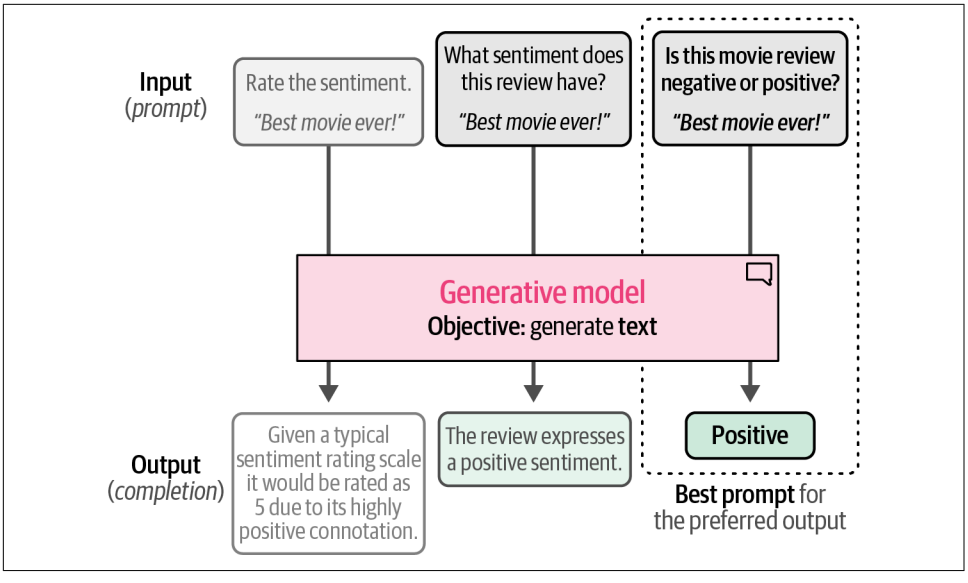

Figure 4-18. Prompt engineering allows prompts to be updated to improve the output generated by the model.

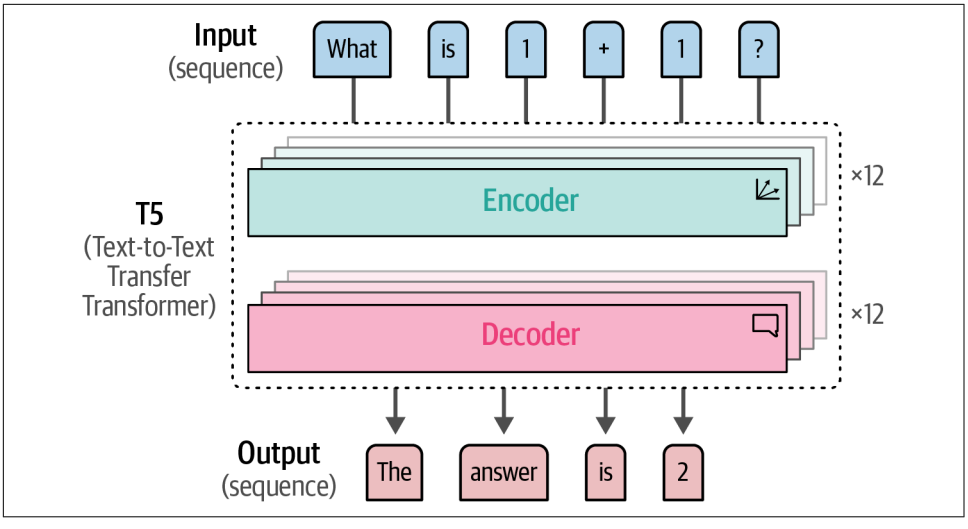

Figure 4-19. The T5 architecture is similar to the original Transformer model, a decoder-encoder architecture.

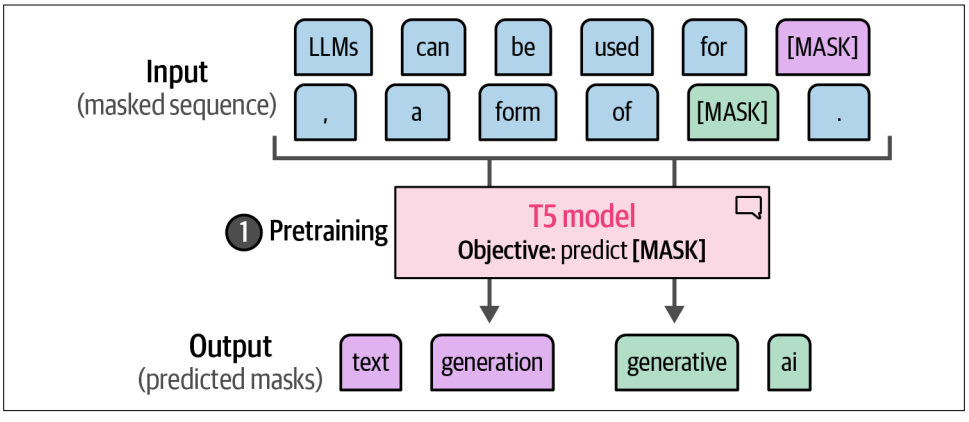

Figure 4-20. In the first step of training, namely pretraining, the T5 model needs to predict masks that could contain multiple tokens.

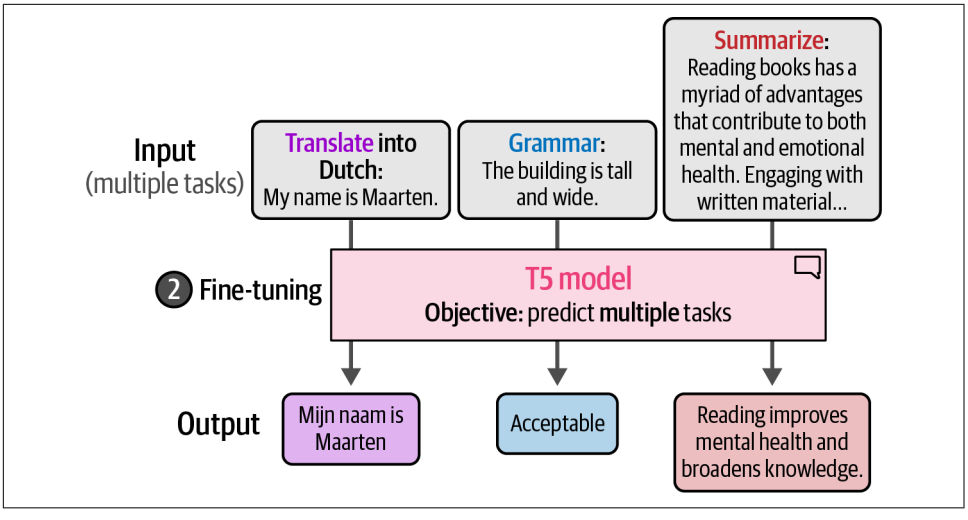

Figure 4-21. By converting specific tasks to textual instructions, the T5 model can be trained on a variety of tasks during fine-tuning.

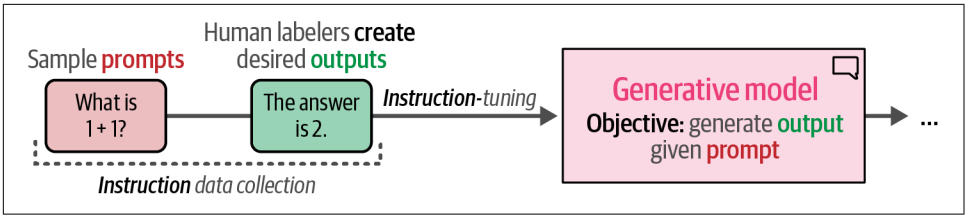

Figure 4-22. Manually labeled data consisting of an instruction (prompt) and output was used to perform fine-tuning (instruction-tuning).

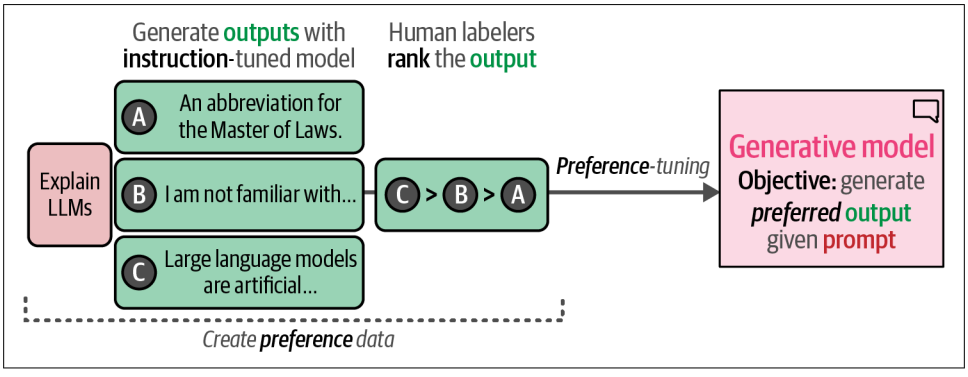

Figure 4-23. Manually ranked preference data was used to generate the final model, ChatGPT.

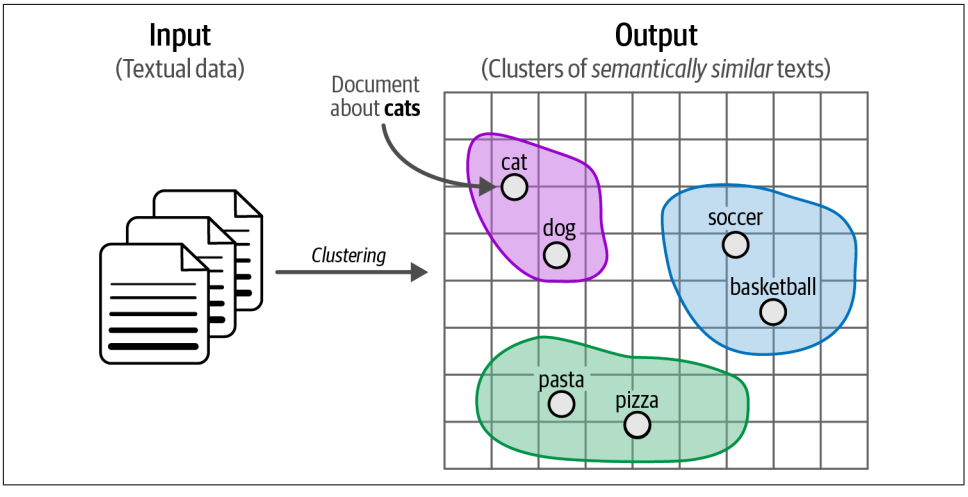

Figure 5-1. Clustering unstructured textual data.

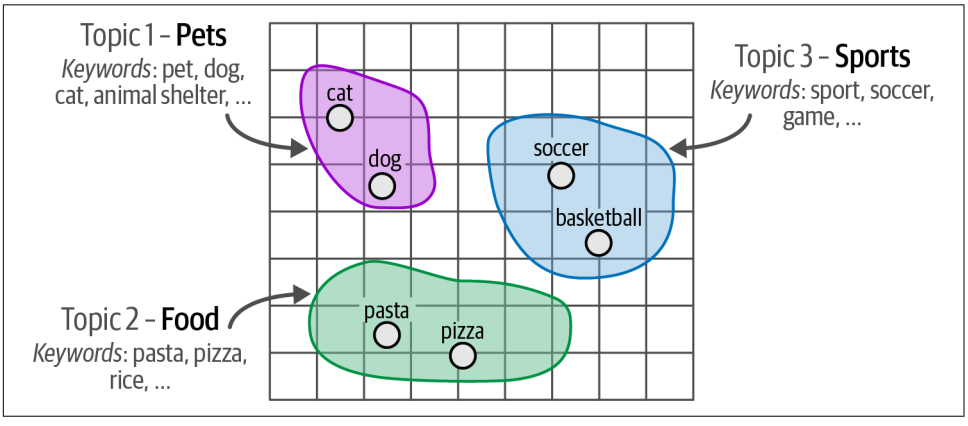

Figure 5-2. Topic modeling is a way to give meaning to clusters of textual documents.

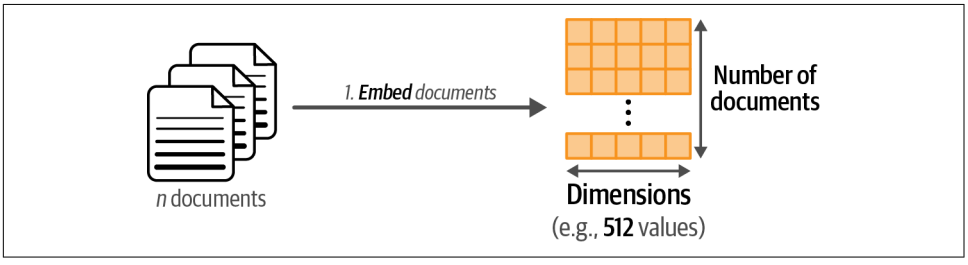

Figure 5-3. Step 1: We convert documents to embeddings using an embedding model.

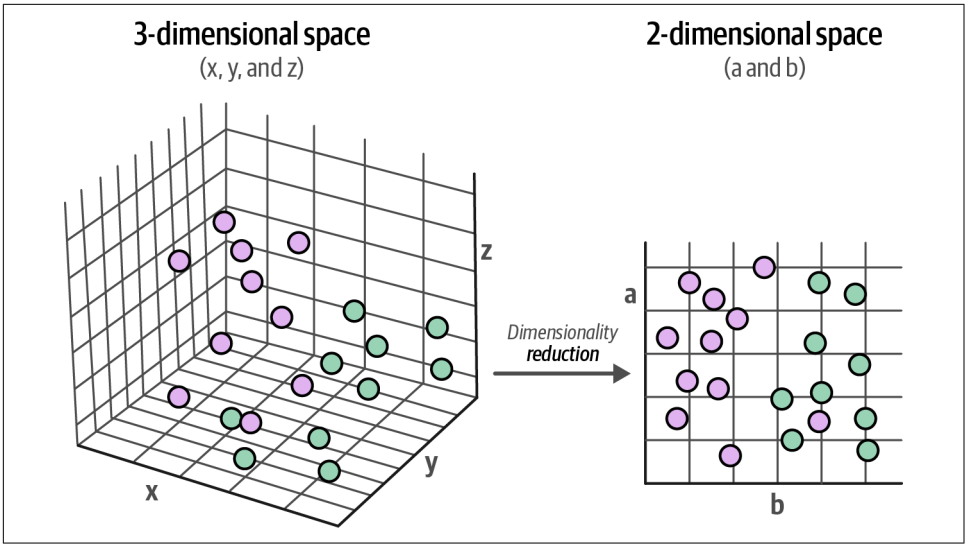

Figure 5-4. Dimensionality reduction allows data in high-dimensional space to be compressed to a lower-dimensional representation.

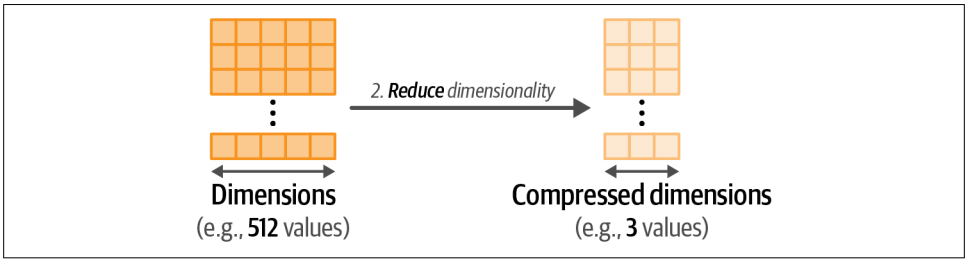

Figure 5-5. Step 2: The embeddings are reduced to a lower-dimensional space using dimensionality reduction.

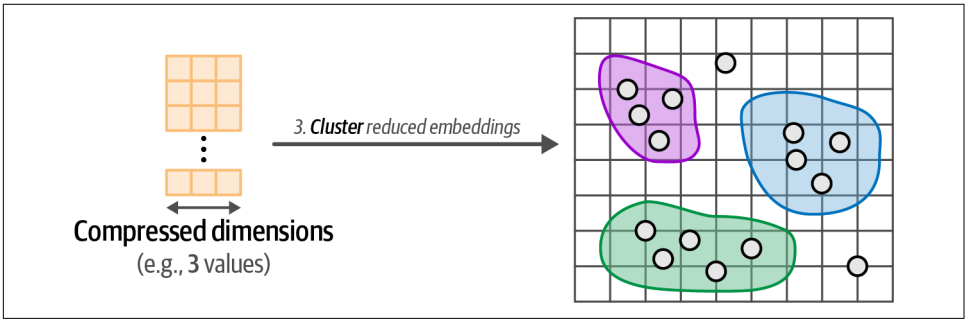

Figure 5-6. Step 3: We cluster the documents using the embeddings with reduced dimensionality.

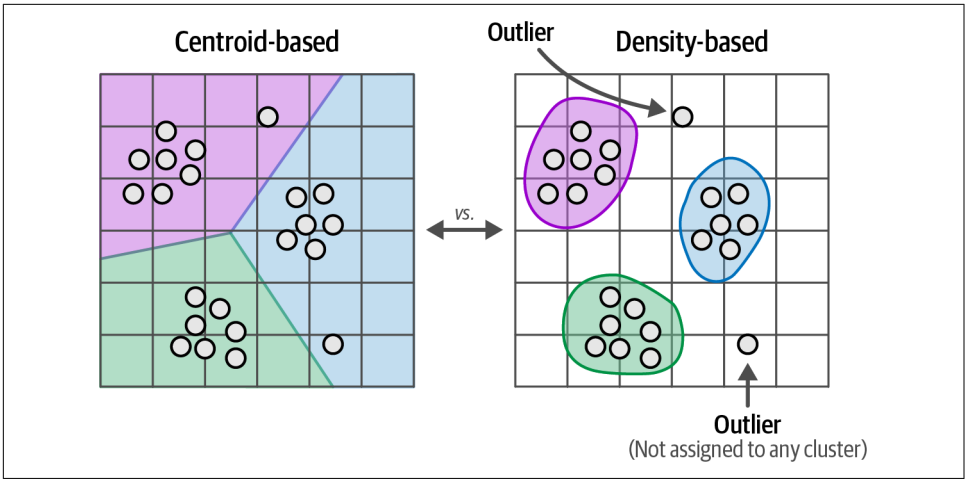

Figure 5-7. The clustering algorithm not only impacts how clusters are generated but also how they are viewed.

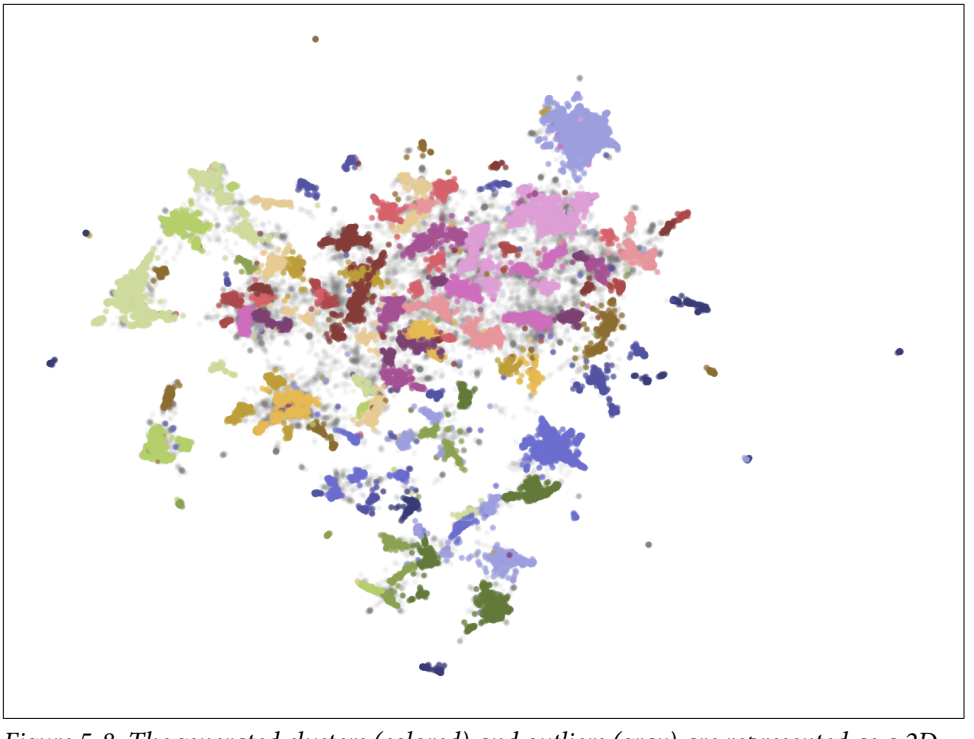

Figure 5-8. The generated clusters (colored) and outliers (gray) are represented as a 2D visualization.

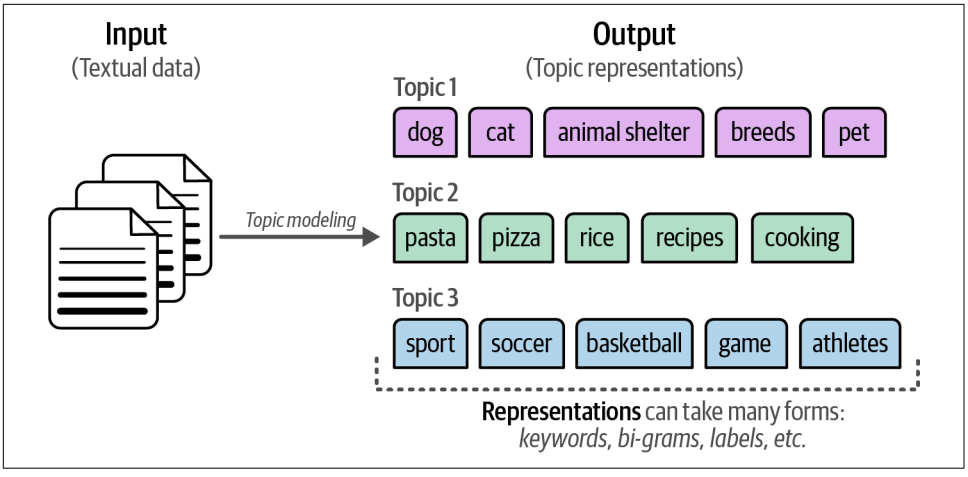

Figure 5-9. Traditionally, topics are represented by a number of keywords but can take other forms.

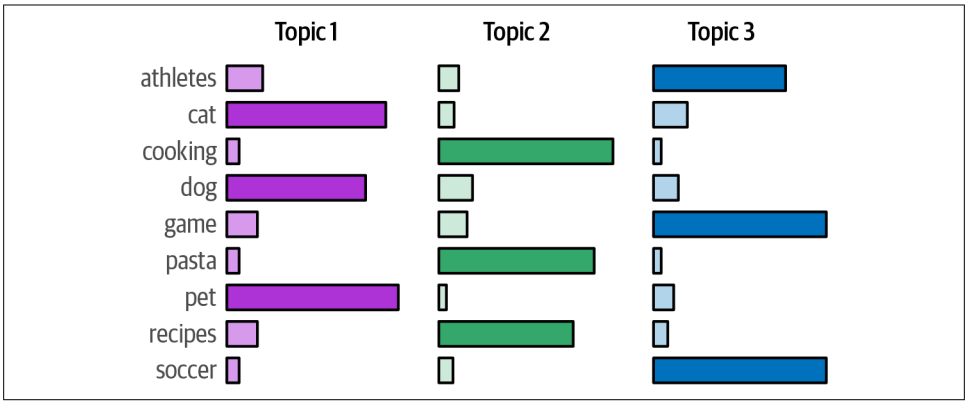

Figure 5-10. Keywords are extracted based on their distribution over a single topic.

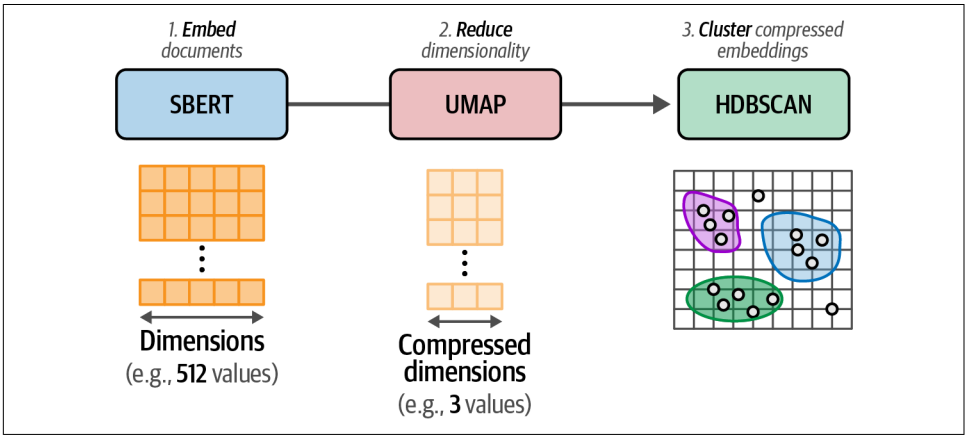

Figure 5-11. The first part of BERTopic's pipeline is to create clusters of semantically similar documents.

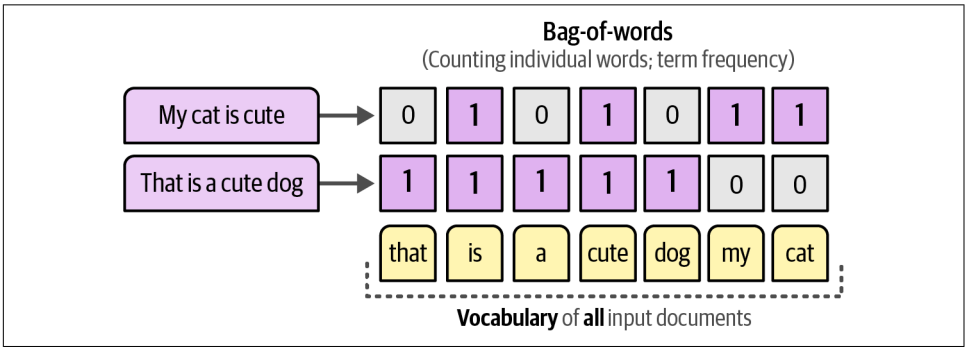

Figure 5-12. A bag-of-words counts the number of times each word appears inside a document.

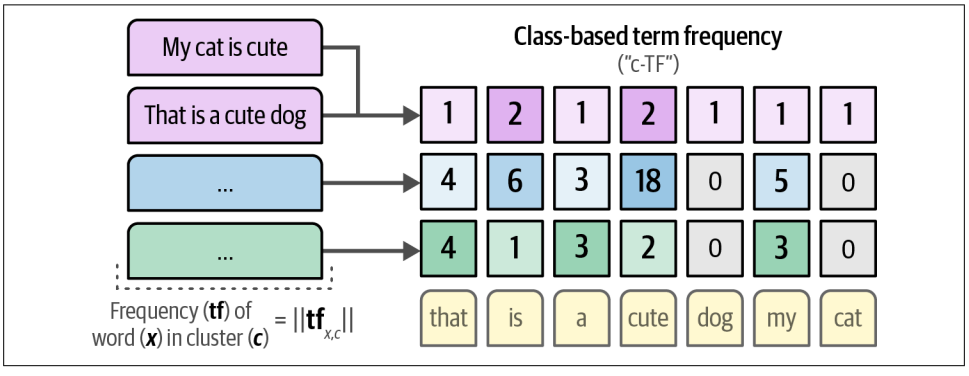

Figure 5-13. Generating c-TF by counting the frequency of words per cluster instead of per document.

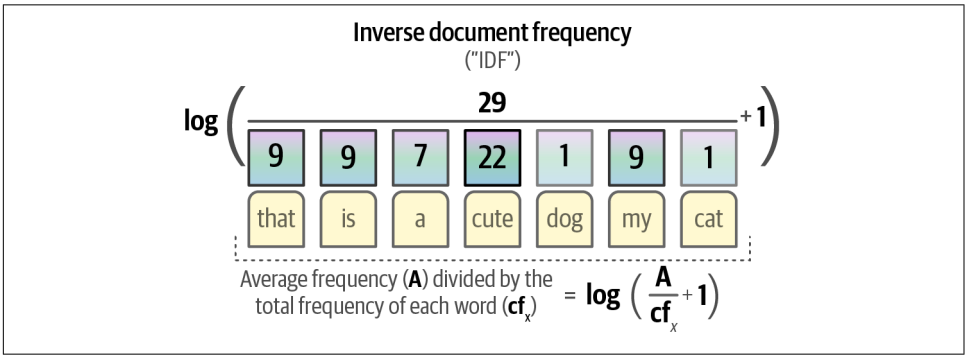

Figure 5-14. Creating a weighting scheme.

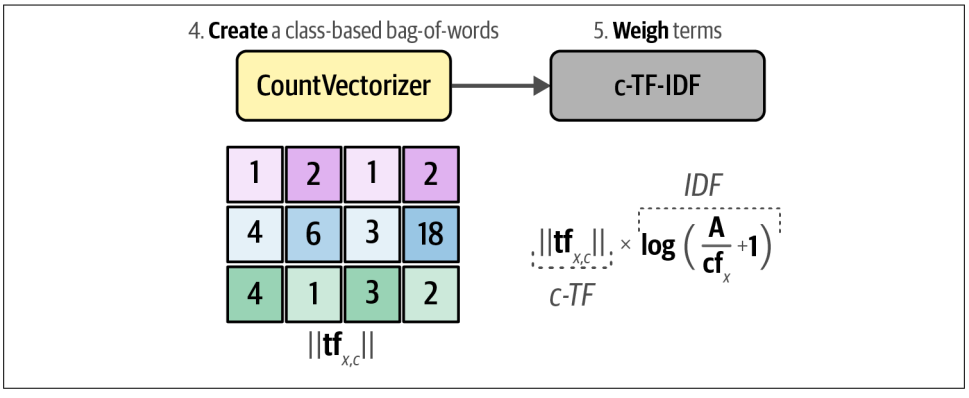

*Figure 5-15. The second part of BERTopic's pipeline is representing the topics: the calculation of the weight of term *x* in a class *c*.*

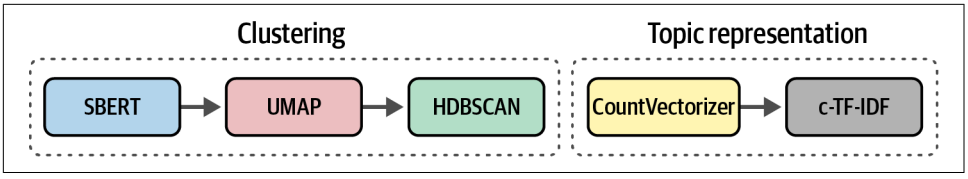

Figure 5-16. The full pipeline of BERTopic, roughly, consists of two steps, clustering and topic representation.

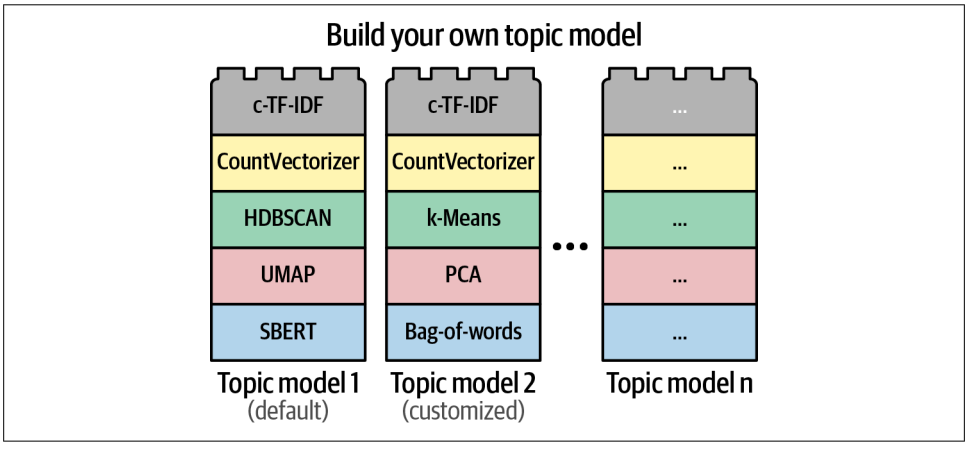

Figure 5-17. The modularity of BERTopic is a key component and allows you to build your own topic model however you want.

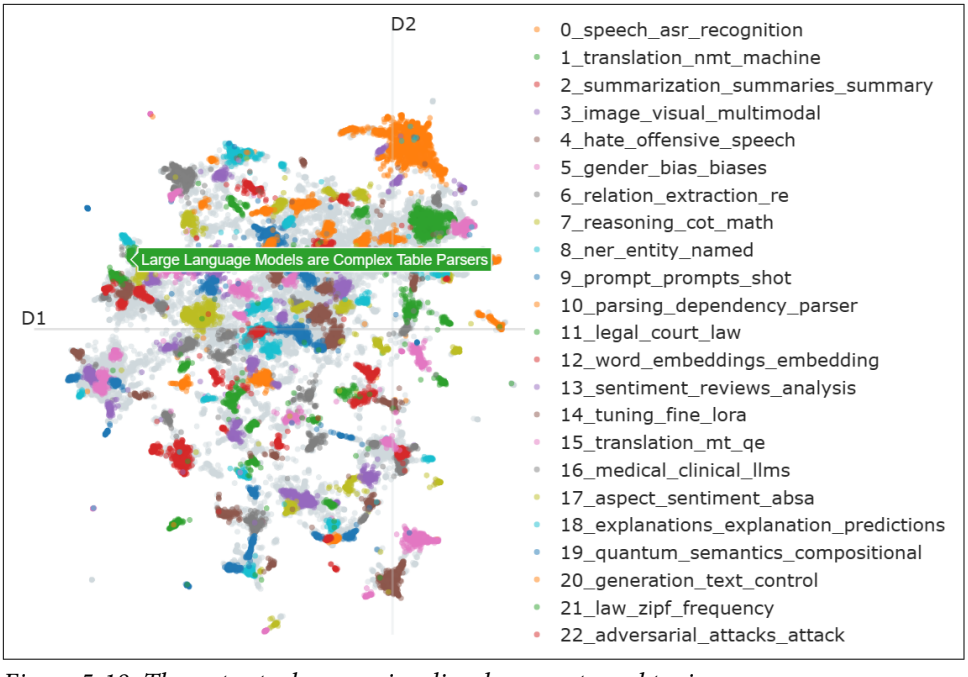

Figure 5-18. The output when we visualize documents and topics.

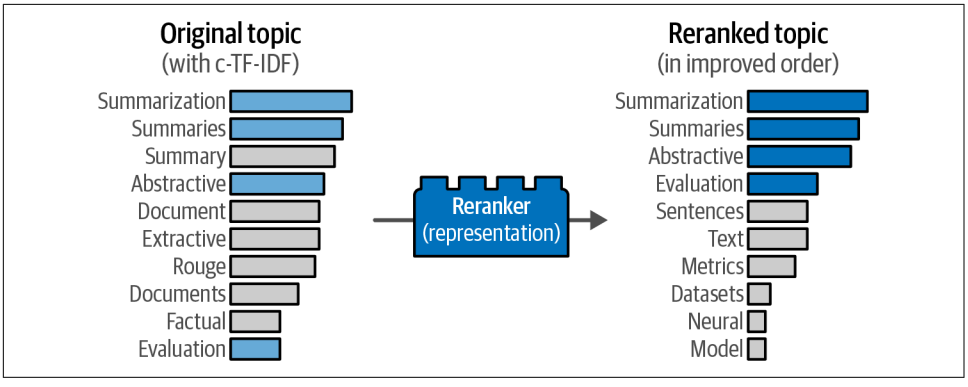

Figure 5-19. Fine-tune the topic representations by reranking the original c-TF-IDF distributions.

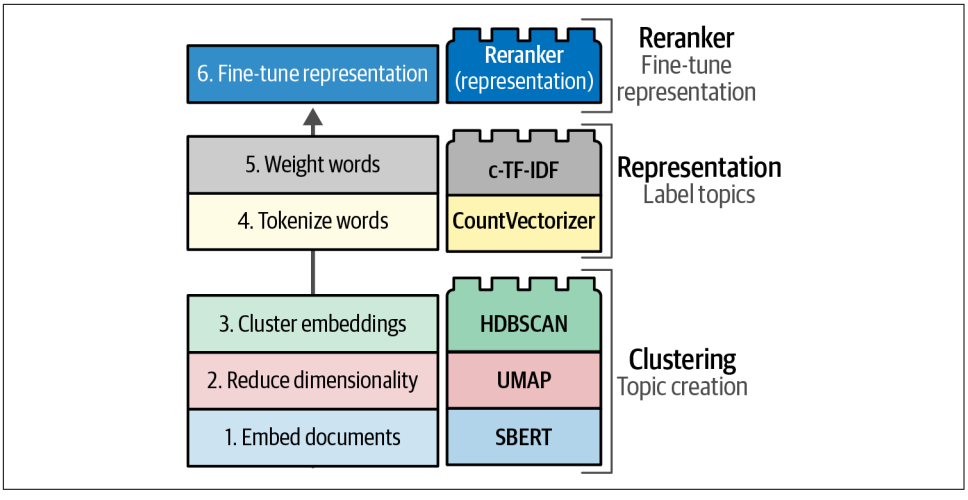

Figure 5-20. The reranker (representation) block sits on top of the c-TF-IDF representation.

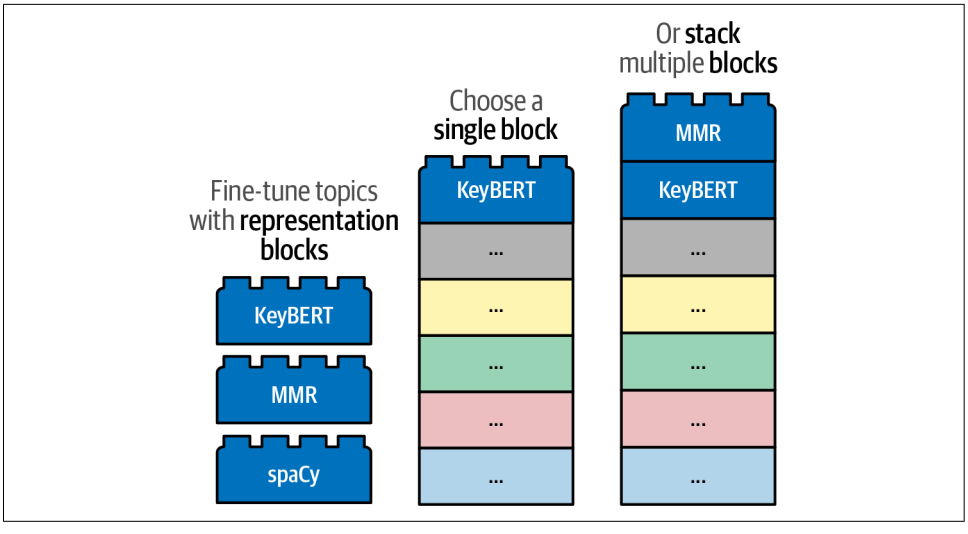

Figure 5-21. After applying the c-TF-IDF weighting, topics can be fine-tuned with a wide variety of representation models, many of which are large language models.

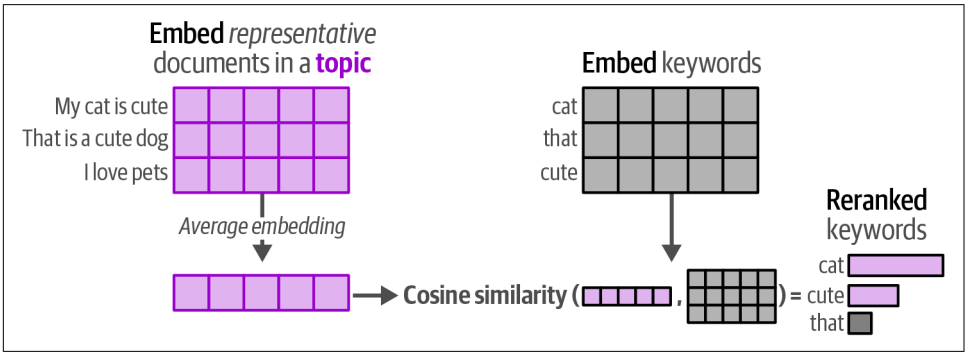

Figure 5-22. KeyBERTInspired representation model procedure.

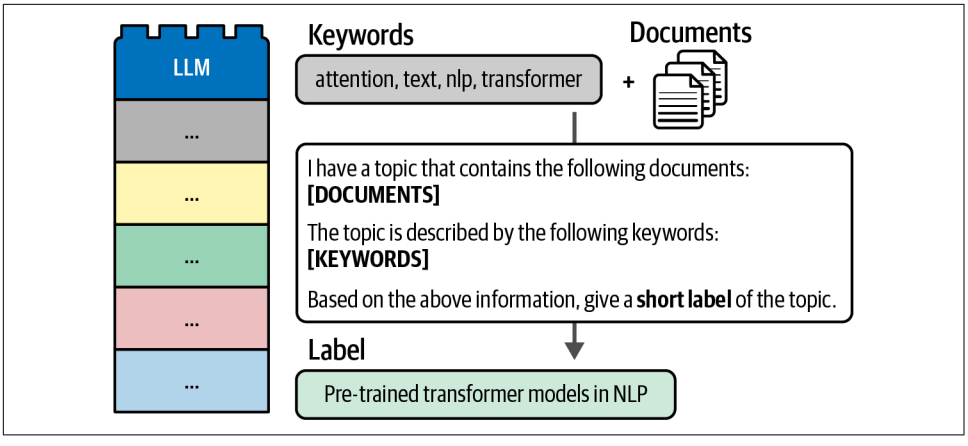

Figure 5-23. Use text generative LLMs and prompt engineering to create labels for topics from keywords and documents related to each topic.

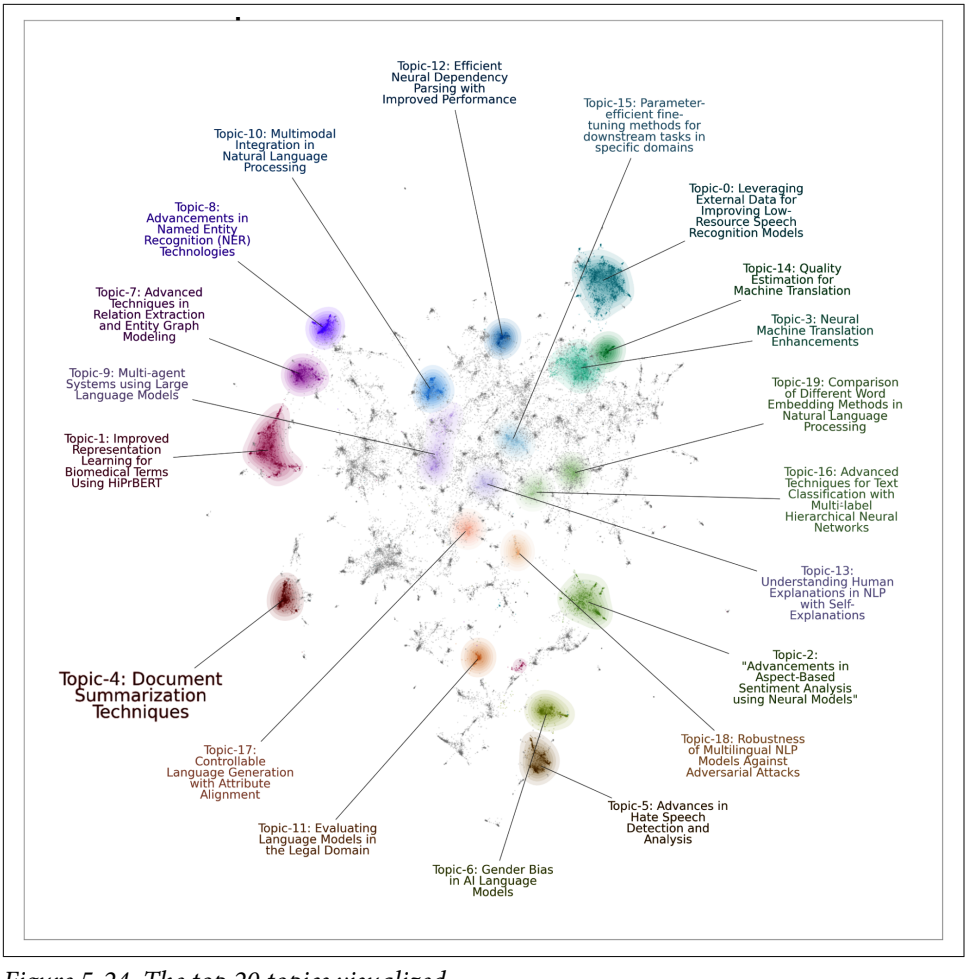

Figure 5-24. The top 20 topics visualized.

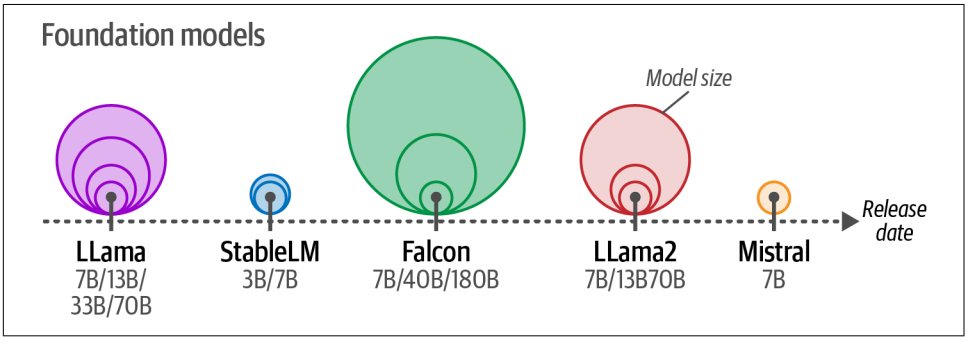

Figure 6-1. Foundation models are often released in several different sizes.

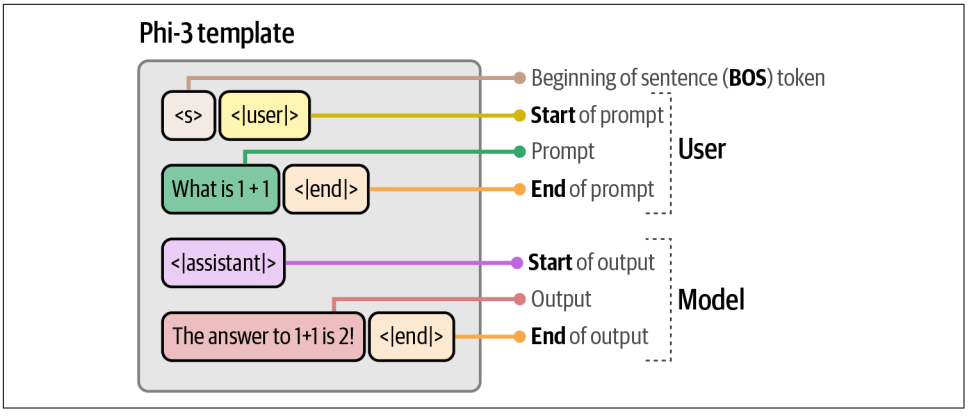

Figure 6-2. The template Phi-3 expects when interacting with the model.

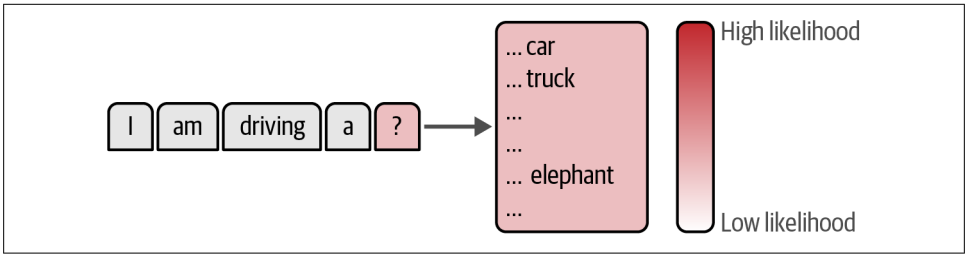

Figure 6-3. The model chooses the next token to generate based on their likelihood scores.

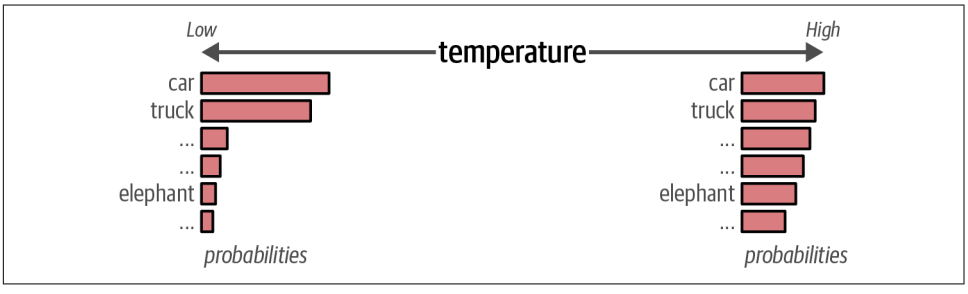

Figure 6-4. A higher `temperature` increases the likelihood that less probable tokens are generated and vice versa.

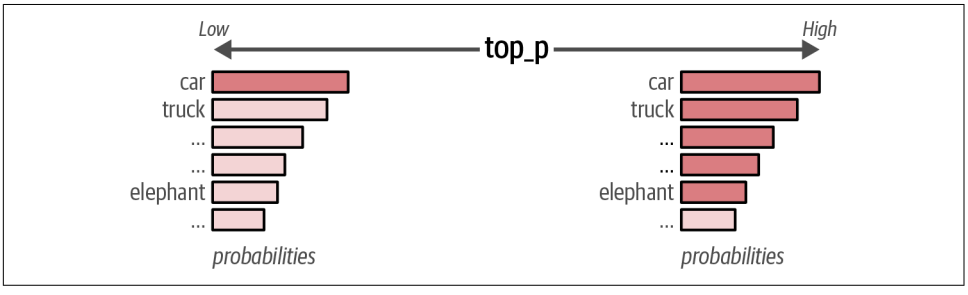

Figure 6-5. A higher `top_p` increases the number of tokens that can be selected to generate and vice versa.

Table 6-1. Use case examples when selecting values for `temperature` and `top_p`.

Example use case	`Temperature`	`top_p`	Description
Brainstorming session	High	High	High randomness with large pool of potential tokens. The results will be highly diverse, often leading to very creative and unexpected results.
Email generation	Low	Low	Deterministic output with high probable predicted tokens. This results in predictable, focused, and conservative outputs.
Creative writing	High	Low	High randomness with a small pool of potential tokens. This combination produces creative outputs but still remains coherent.
Translation	Low	High	Deterministic output with high probable predicted tokens. Produces coherent output with a wider range of vocabulary, leading to outputs with linguistic variety.

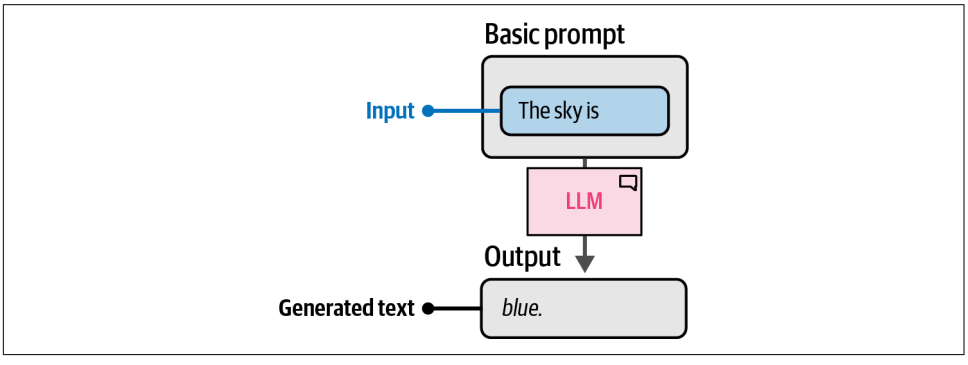

Figure 6-6. A basic example of a prompt. No instruction is given so the LLM will simply try to complete the sentence.

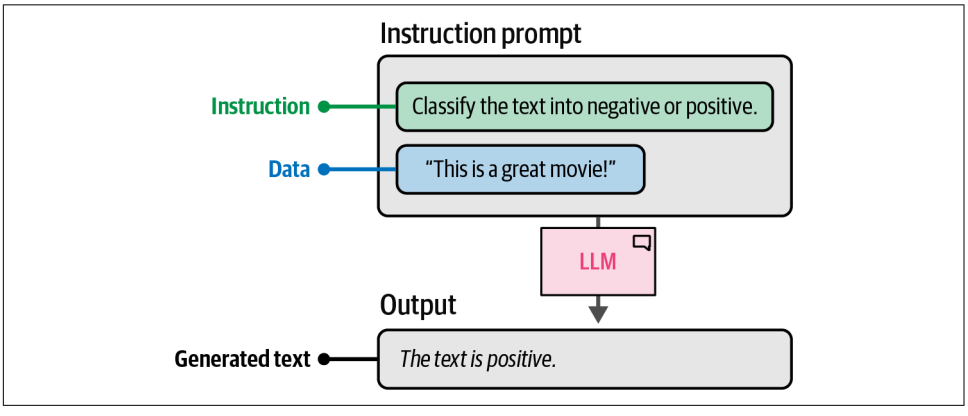

Figure 6-7. Two components of a basic instruction prompt: the instruction itself and the data it refers to.

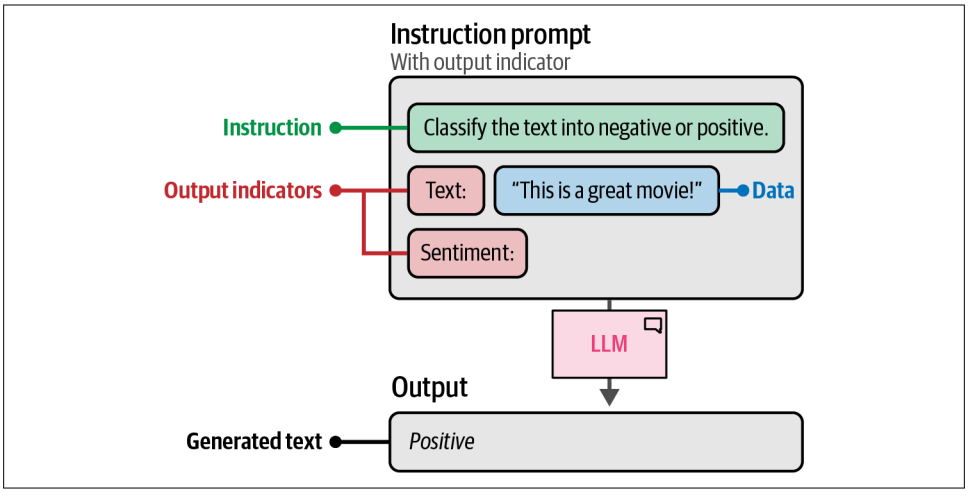

Figure 6-8. Extending the prompt with an output indicator that allows for a specific output.

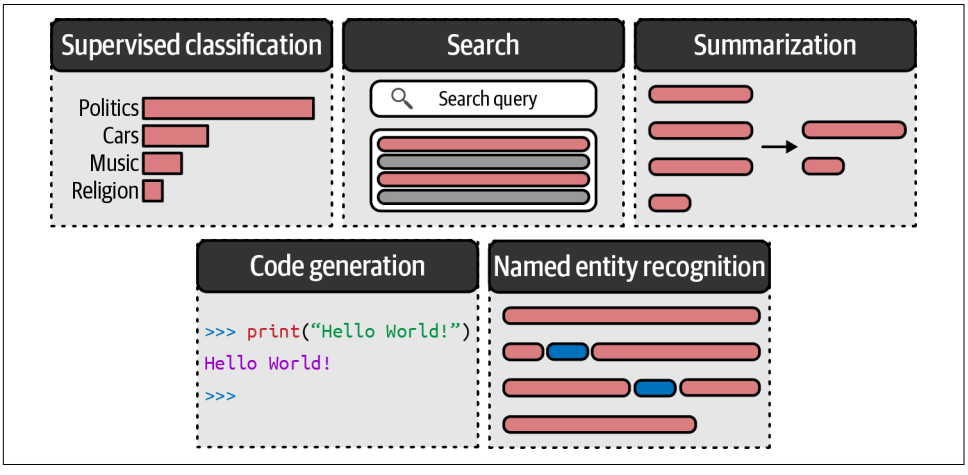

Figure 6-9. Use cases for instruction-based prompting.

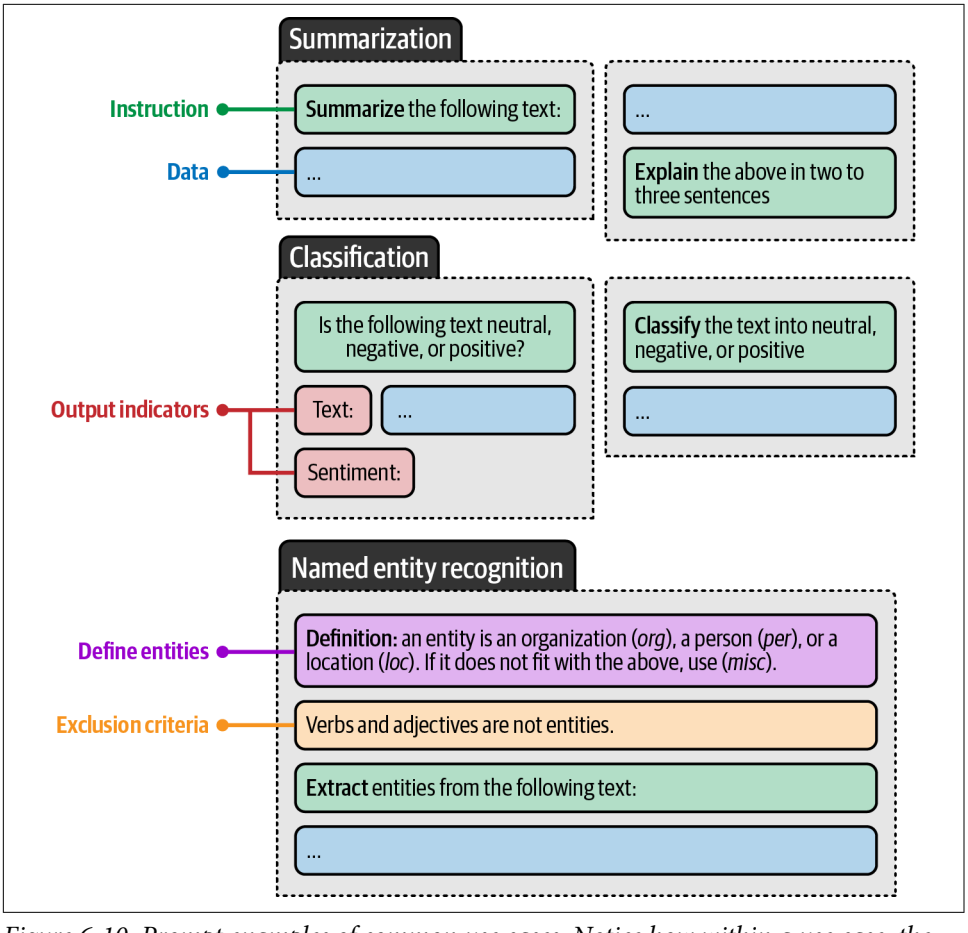

Figure 6-10. Prompt examples of common use cases. Notice how within a use case, the structure and location of the instruction can be changed.

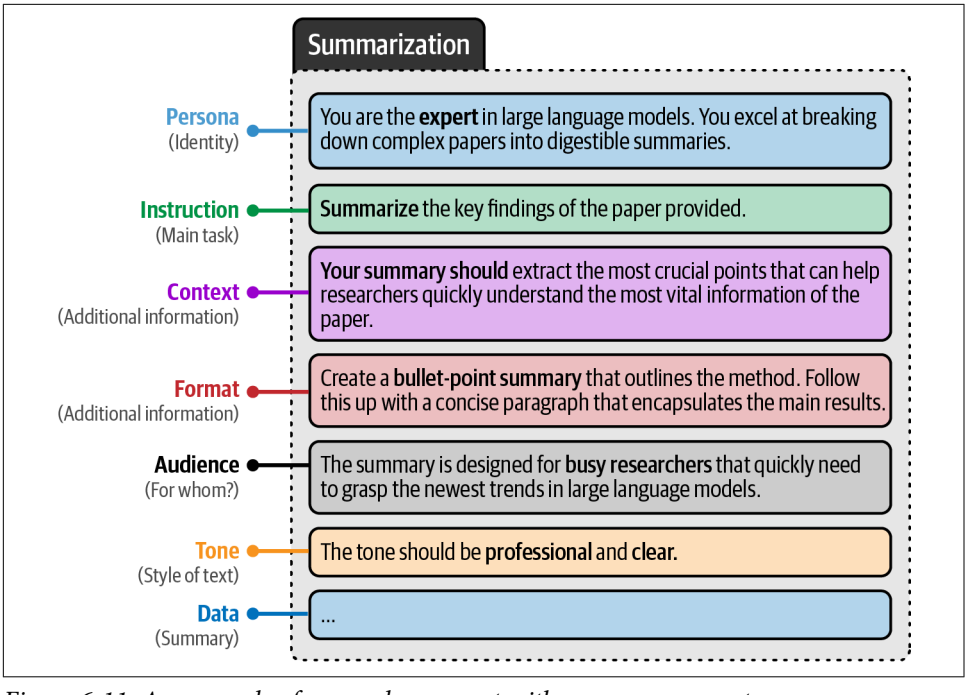

Figure 6-11. An example of a complex prompt with many components.

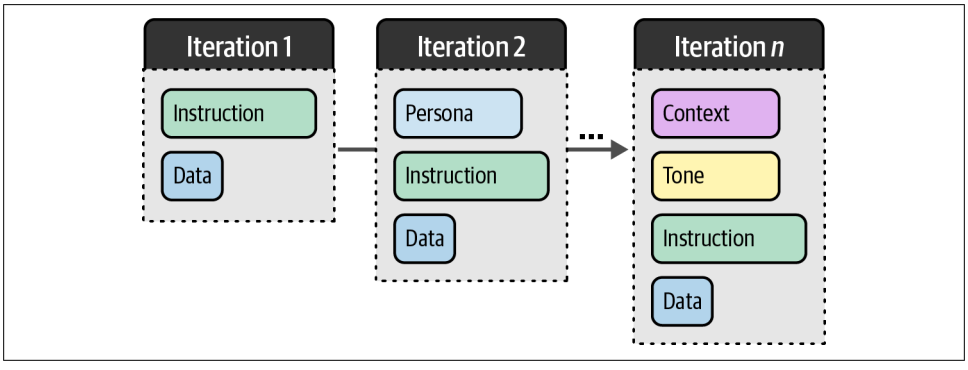

Figure 6-12. Iterating over modular components is a vital part of prompt engineering.

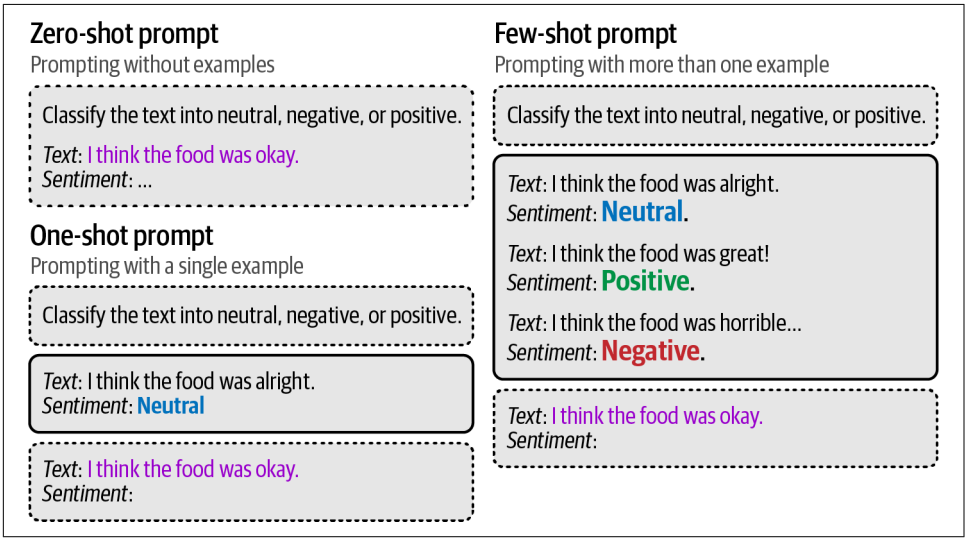

Figure 6-13. An example of a complex prompt with many components.

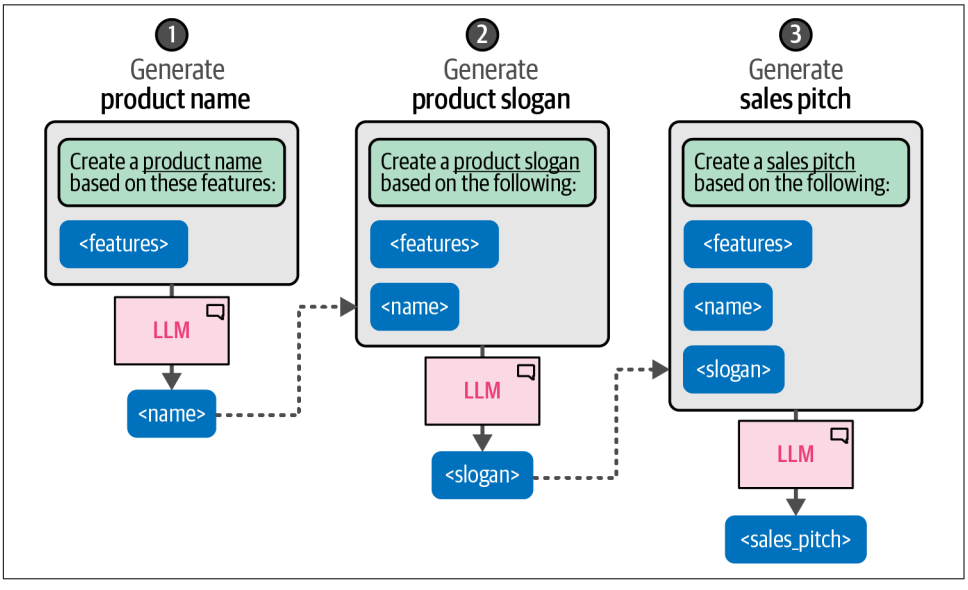

Figure 6-14. Using a description of a product's features, chain prompts to create a suitable name, slogan, and sales pitch.

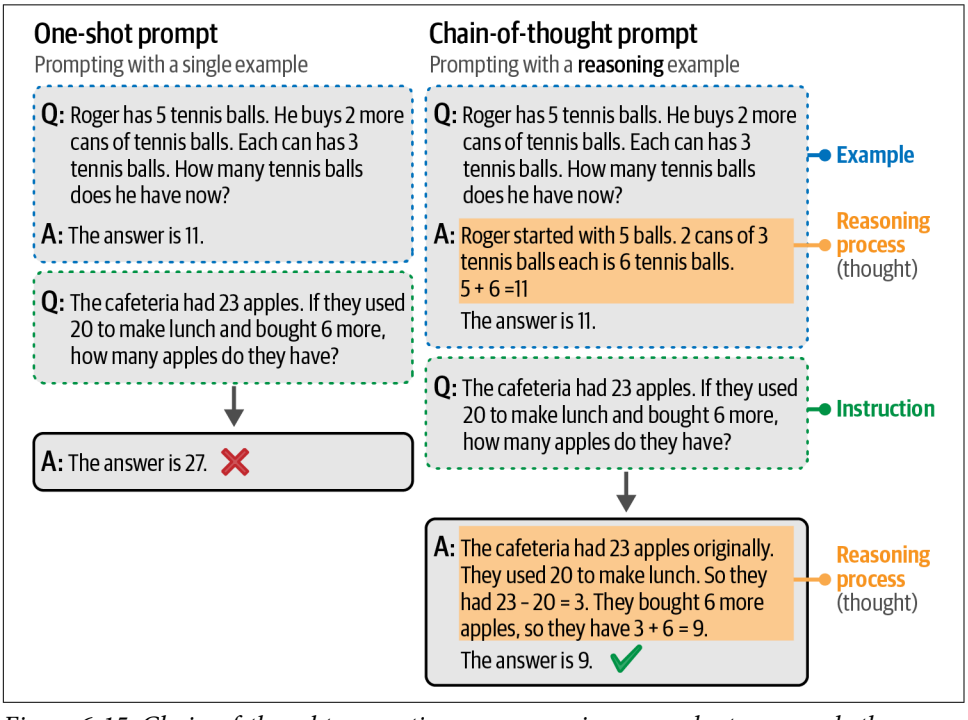

Figure 6-15. Chain-of-thought prompting uses reasoning examples to persuade the generative model to use reasoning in its answer.

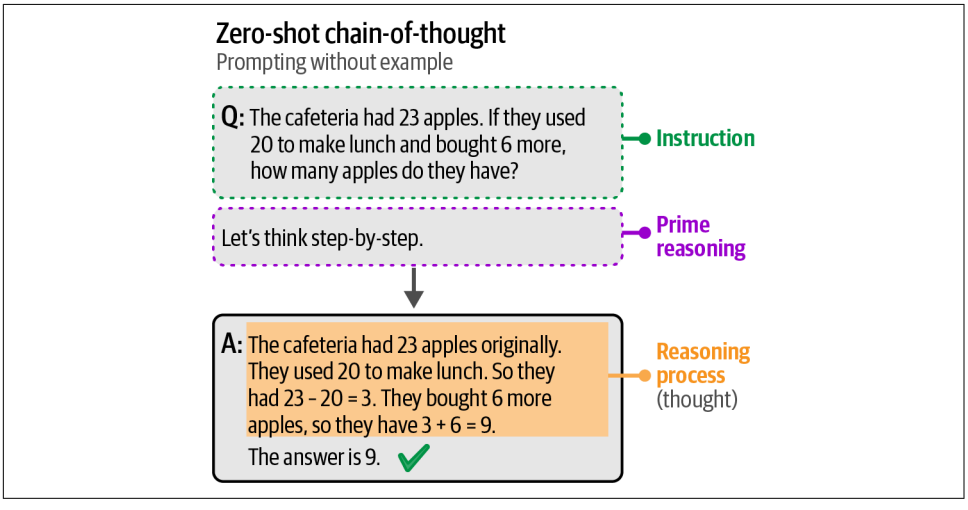

Figure 6-16. Chain-of-thought prompting without using examples. Instead, it uses the phrase "Let's think step-by-step" to prime reasoning in its answer.

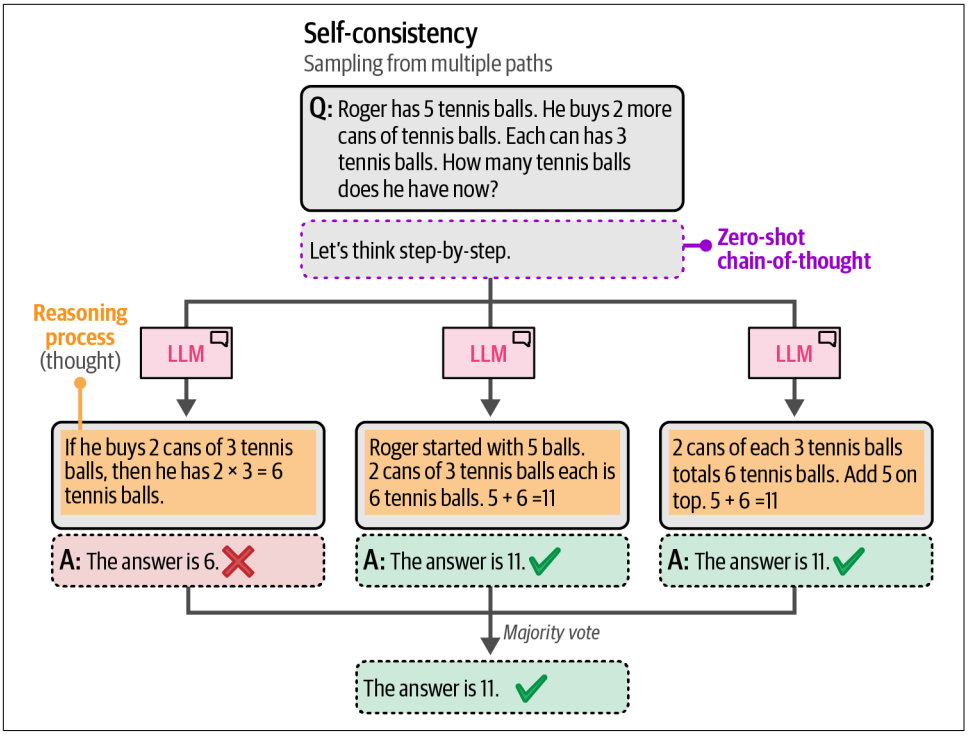

Figure 6-17. By sampling from multiple reasoning paths, we can use majority voting to extract the most likely answer.

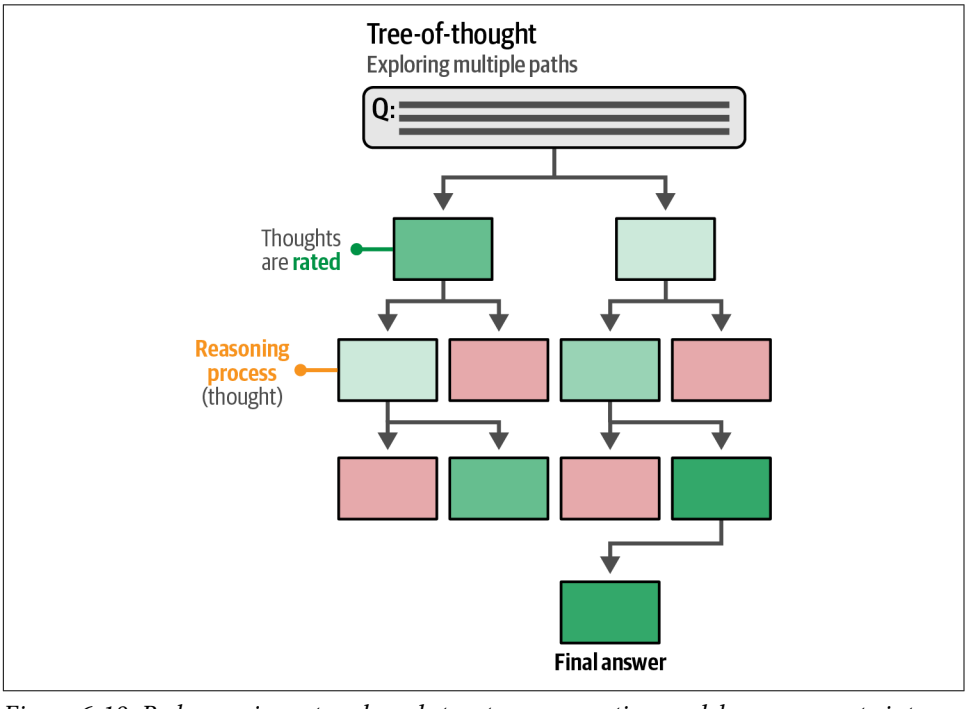

Figure 6-18. By leveraging a tree-based structure, generative models can generate intermediate thoughts to be rated. The most promising thoughts are kept and the lowest are pruned.

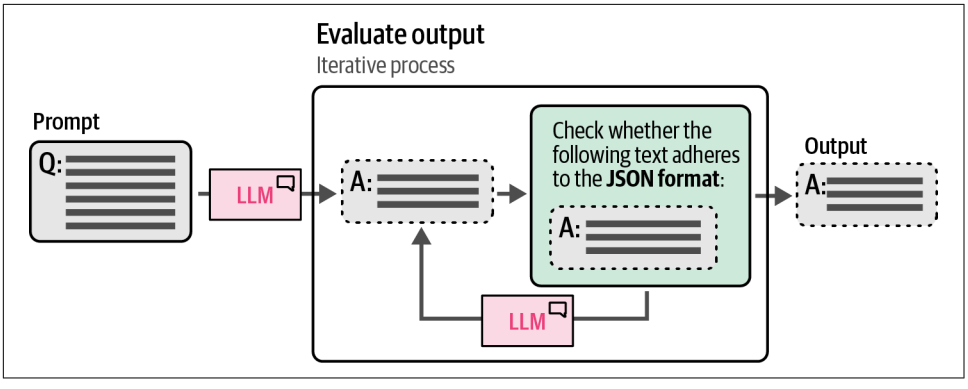

Figure 6-19. Use an LLM to check whether the output correctly follows our rules.

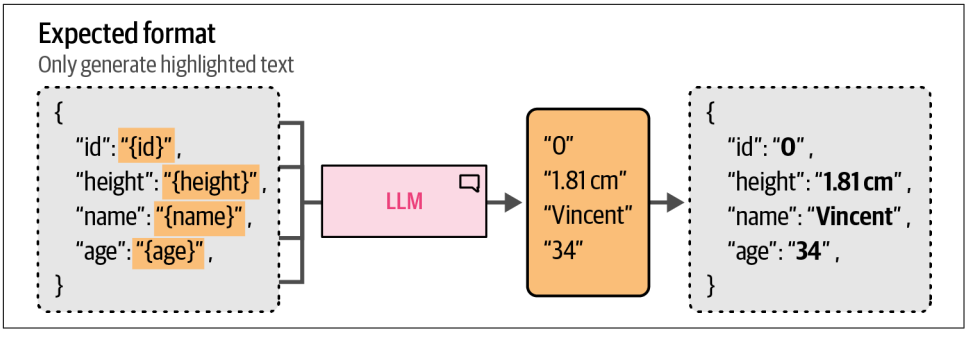

Figure 6-20. Use an LLM to generate only the pieces of information we do not know beforehand.

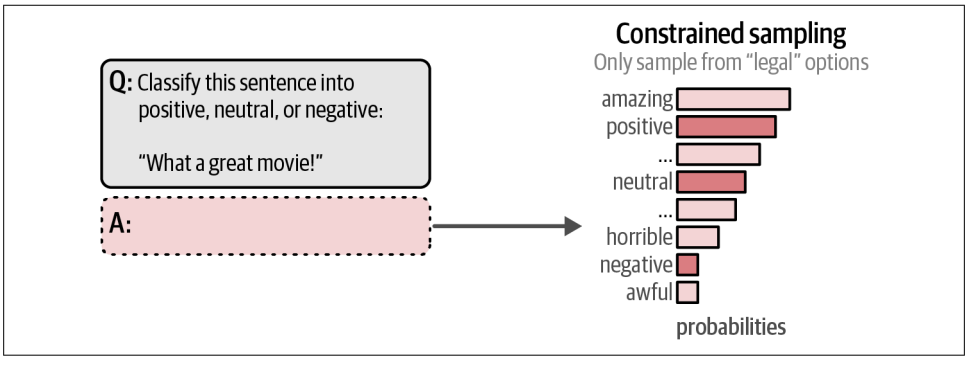

Figure 6-21. Constrain the token selection to only three possible tokens: "positive," "neutral," and "negative."

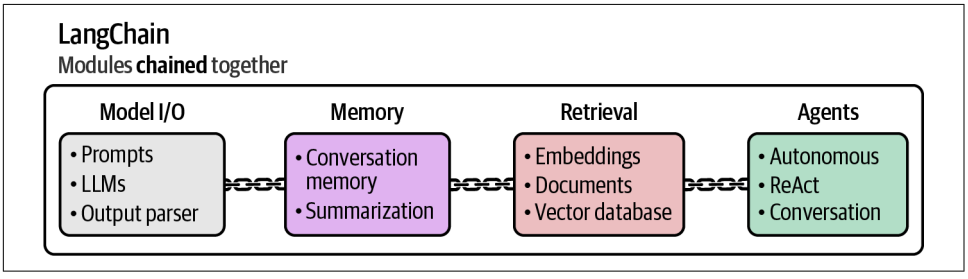

Figure 7-1. LangChain is a complete framework for using LLMs. It has modular components that can be chained together to allow for complex LLM systems.

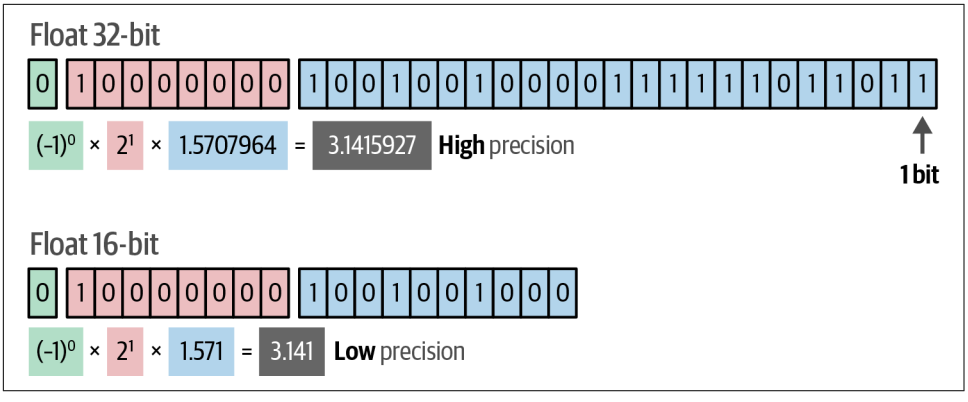

Figure 7-2. Attempting to represent pi with float 32-bit and float 16-bit representations. Notice the lowered accuracy when we halve the number of bits.

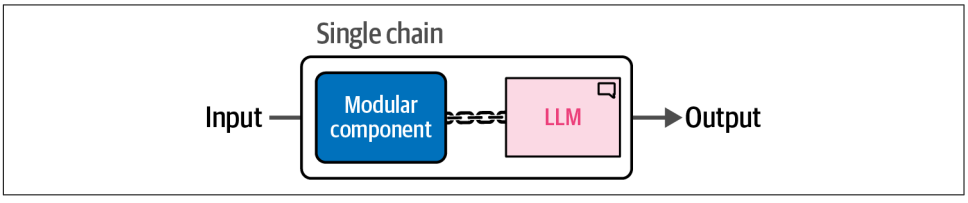

Figure 7-3. A single chain connects some modular component, like a prompt template or external memory, to the LLM.

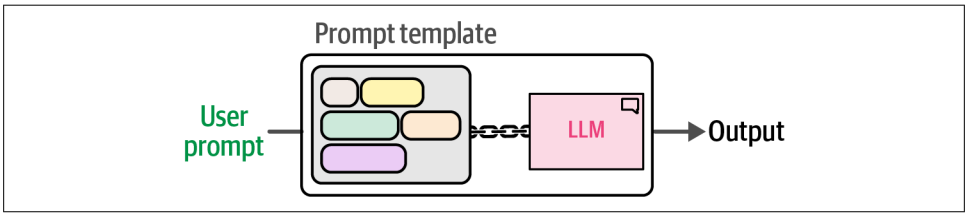

Figure 7-4. By chaining a prompt template with an LLM, we only need to define the input prompts. The template will be constructed for you.

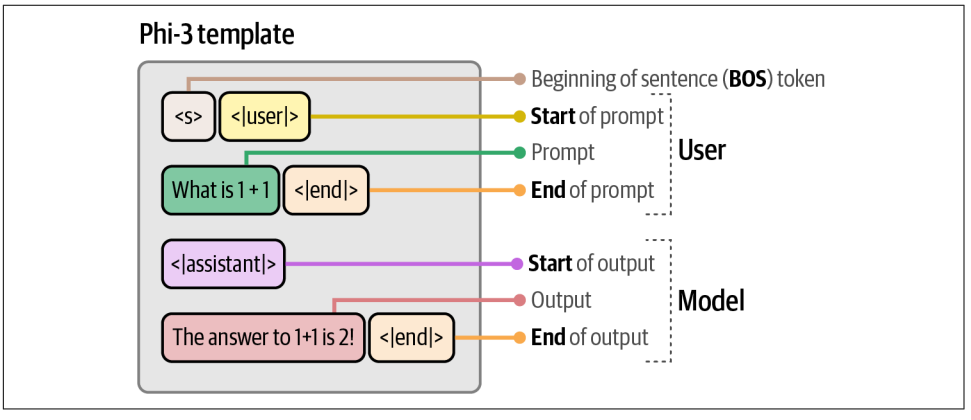

Figure 7-5. The prompt template Phi-3 expects.

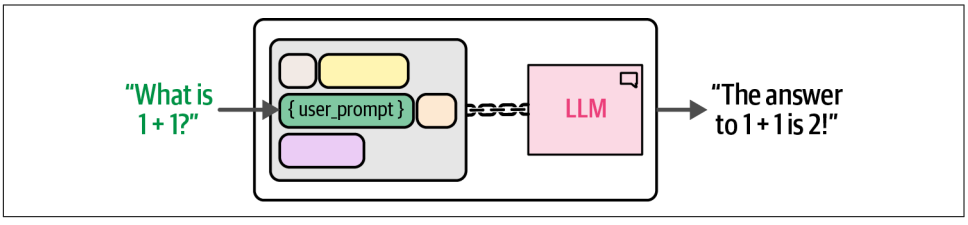

Figure 7-6. An example of a single chain using Phi-3's template.

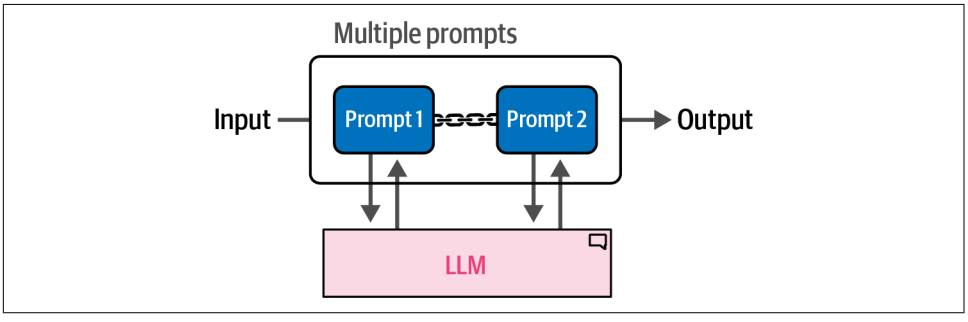

Figure 7-7. With sequential chains, the output of a prompt is used as the input for the next prompt.

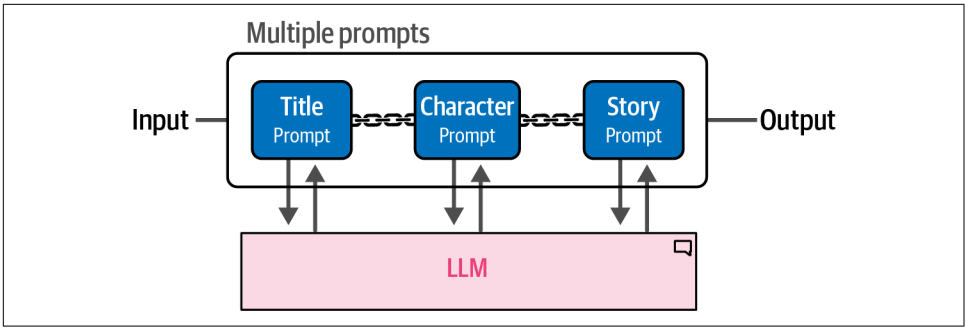

Figure 7-8. The output of the title prompt is used as the input of the character prompt. To generate the story, the output of all previous prompts is used.

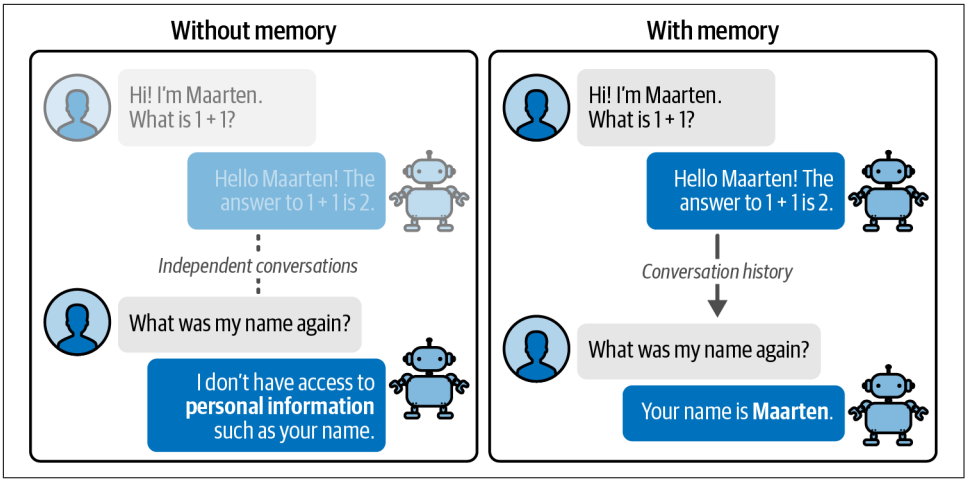

Figure 7-9. An example of a conversation between an LLM with memory and without memory.

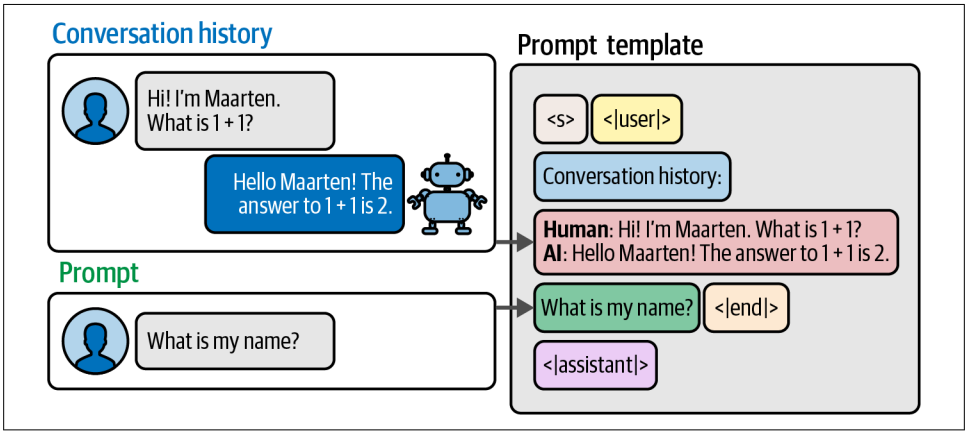

Figure 7-10. We can remind an LLM of what previously happened by simply appending the entire conversation history to the input prompt.

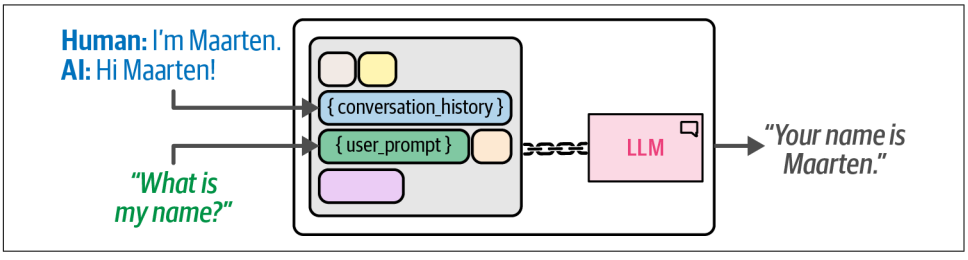

Figure 7-11. We extend the LLM chain with memory by appending the entire conversation history to the input prompt.

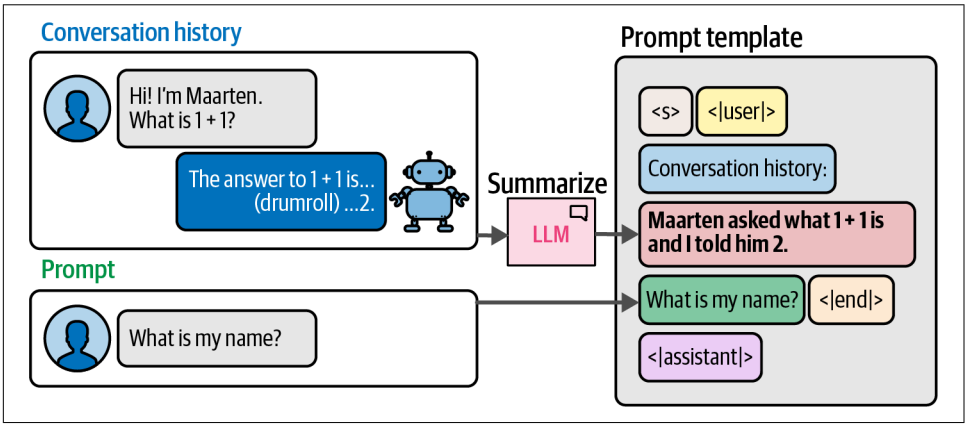

Figure 7-12. Instead of passing the conversation history directly to the prompt, we use another LLM to summarize it first.

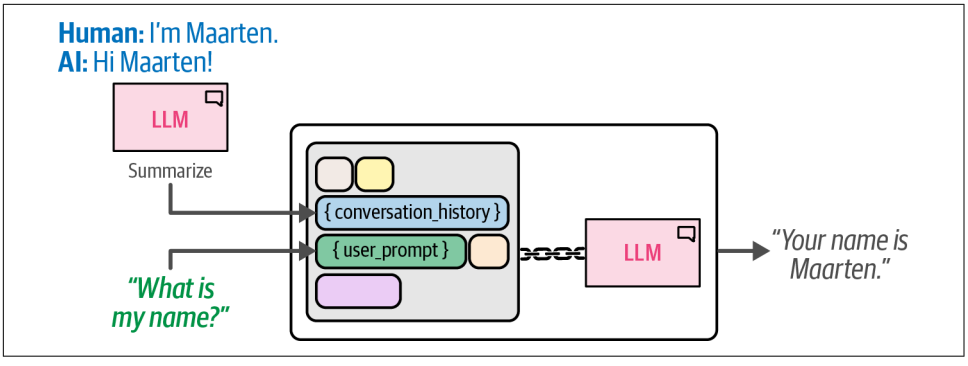

Figure 7-13. We extend the LLM chain with memory by summarizing the entire conversation history before giving it to the input prompt.

Table 7-1. The pros and cons of different memory types.

Memory type	Pros	Cons
Conversation Buffer	• Easiest implementation • Ensures no information loss within context window	• Slower generation speed as more tokens are needed • Only suitable for large-context LLMs • Larger chat histories make information retrieval difficult
Windowed Conversation Buffer	• Large-context LLMs are not needed unless chat history is large • No information loss over the last *k* interactions	• Only captures the last *k* interactions • No compression of the last *k* interactions
Conversation Summary	• Captures the full history • Enables long conversations • Reduces tokens needed to capture full history	• An additional call is necessary for each interaction • Quality is reliant on the LLM's summarization capabilities

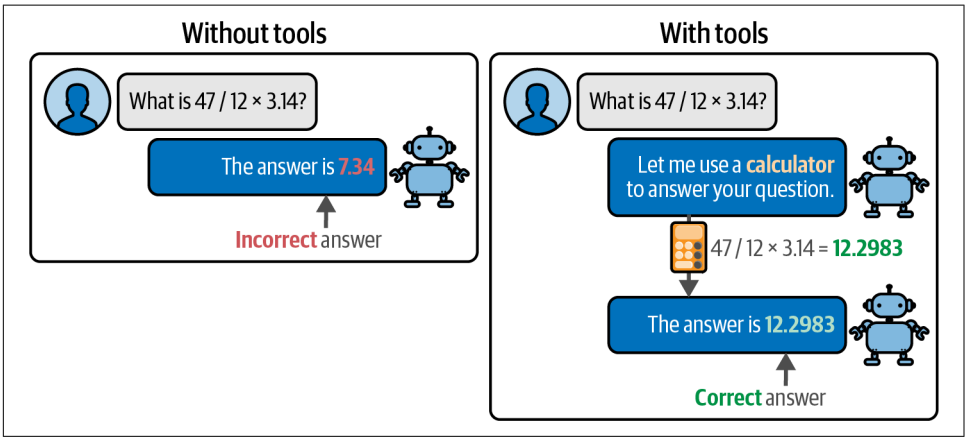

Figure 7-14. Giving LLMs the ability to choose which tools they use for a particular problem results in more complex and accurate behavior.

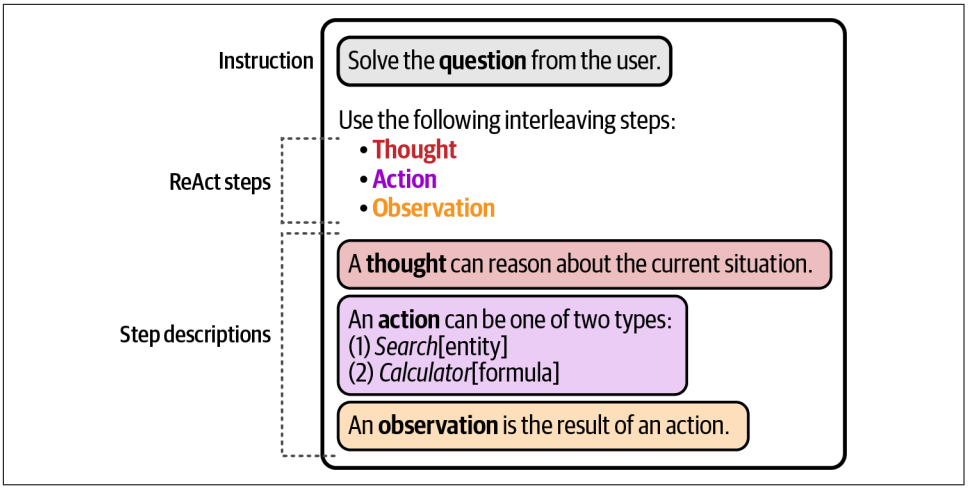

Figure 7-15. An example of a ReAct prompt template.

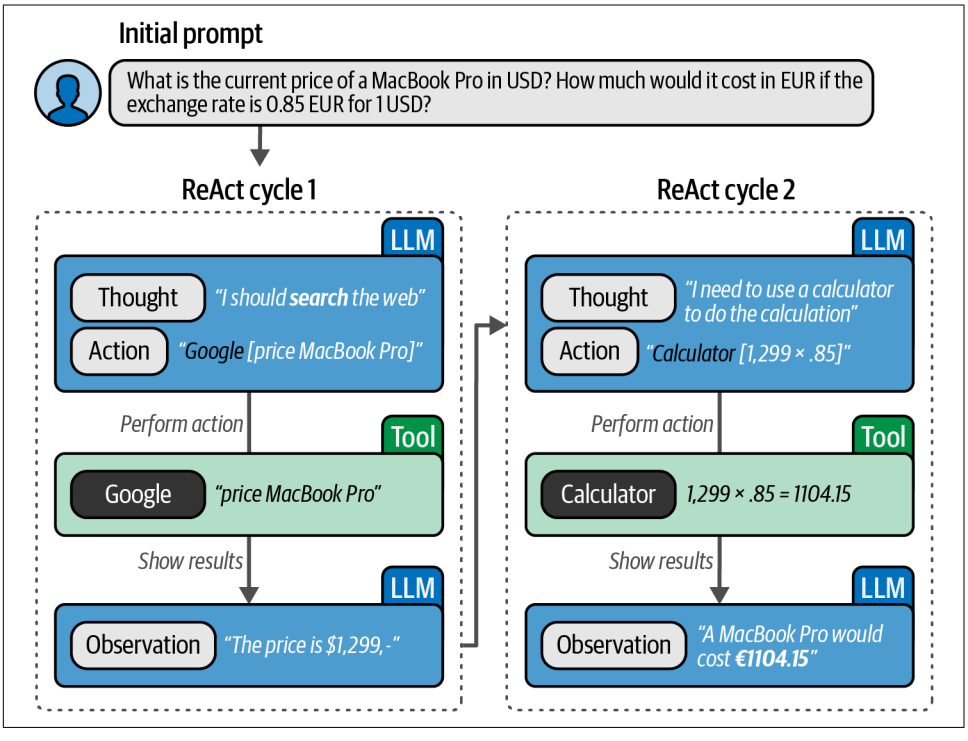

Figure 7-16. An example of two cycles in a ReAct pipeline.

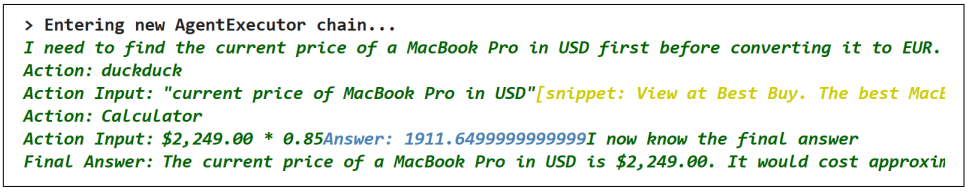

Figure 7-17. An example of the ReAct process in LangChain.

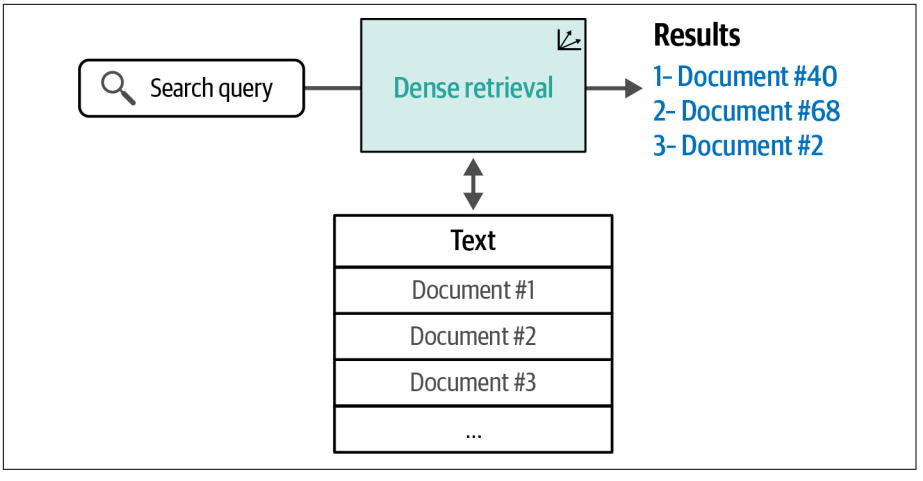

Figure 8-1. Dense retrieval is one of the key types of semantic search, relying on the similarity of text embeddings to retrieve relevant results.

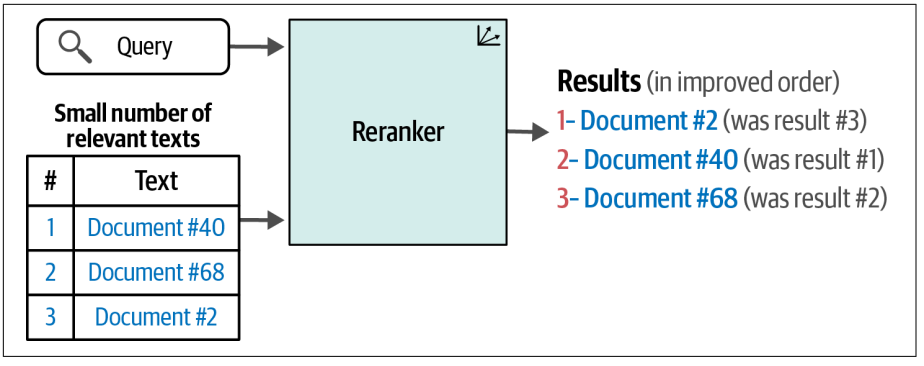

Figure 8-2. Rerankers, the second key type of semantic search, take a search query and a collection of results, and reorder them by relevance, often resulting in vastly improved results.

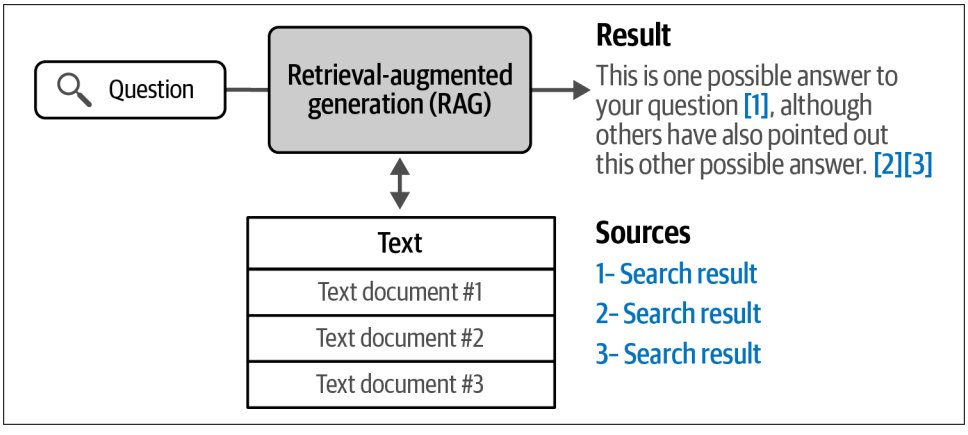

Figure 8-3. A RAG system formulates an answer to a question and (preferably) cites its information sources.

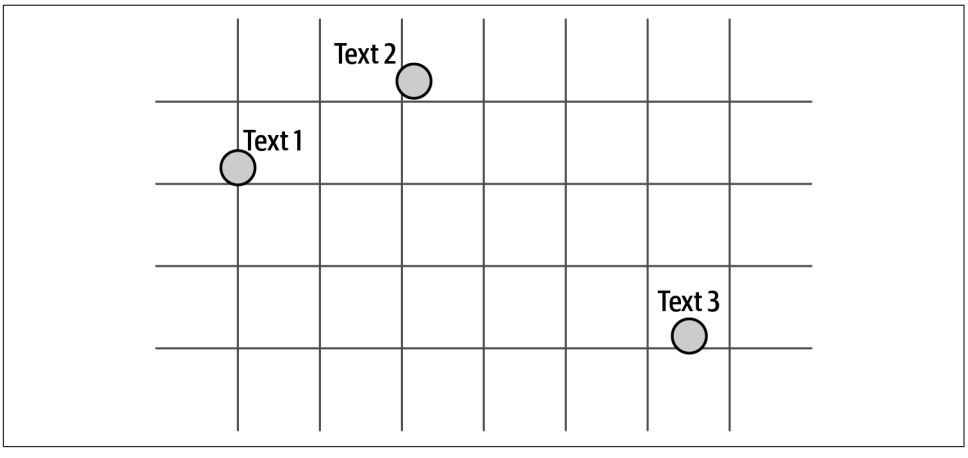

Figure 8-4. The intuition of embeddings: each text is a point and texts with similar meaning are close to each other.

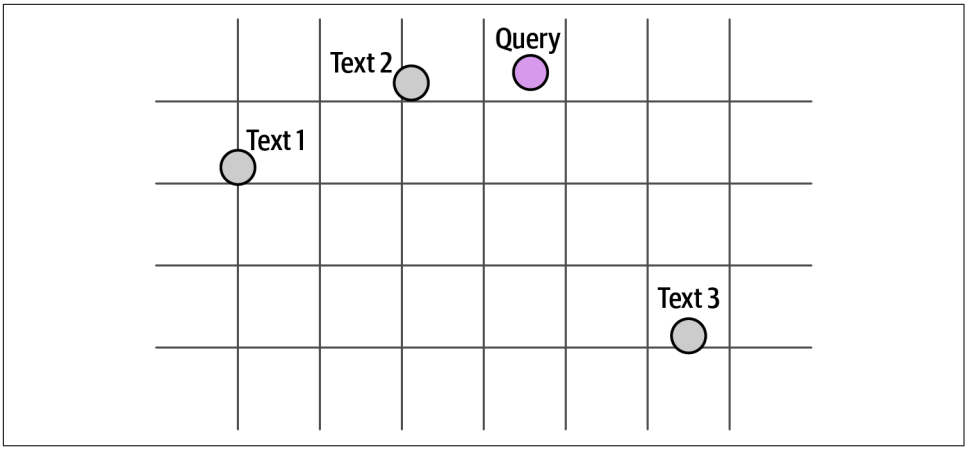

Figure 8-5. Dense retrieval relies on the property that search queries will be close to their relevant results.

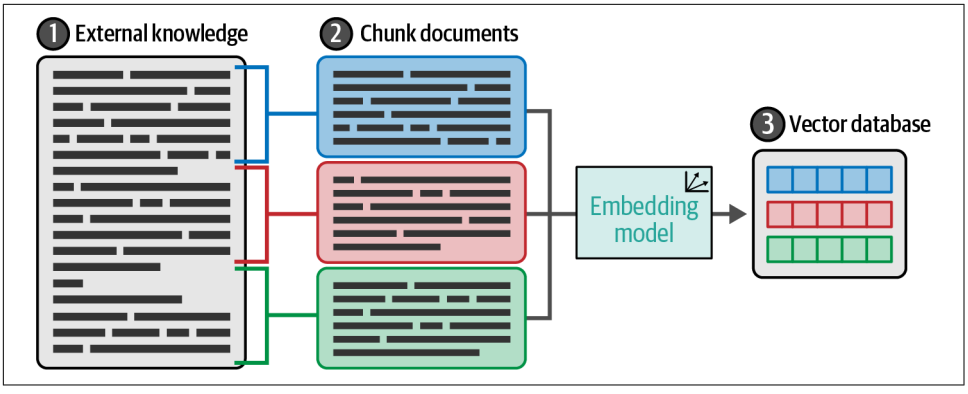

Figure 8-6. Convert some external knowledge base to a vector database. We can then query this vector database for information about the knowledge base.

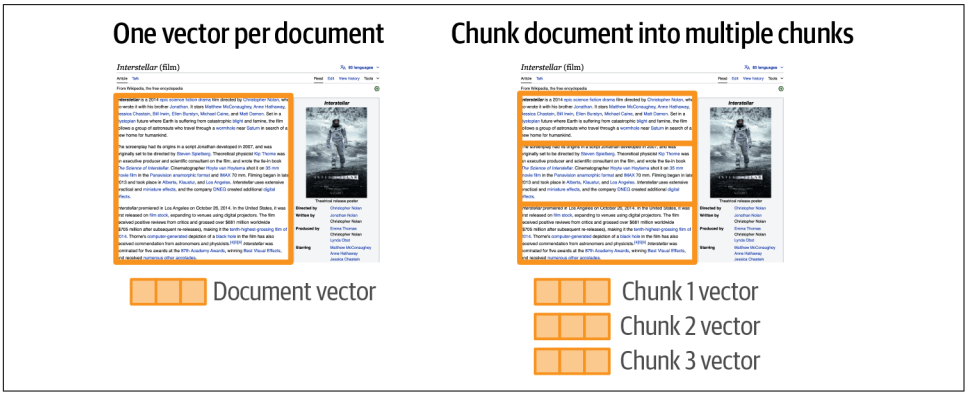

Figure 8-7. It's possible to create one vector representing an entire document, but it's better for longer documents to be split into smaller chunks that get their own embeddings.

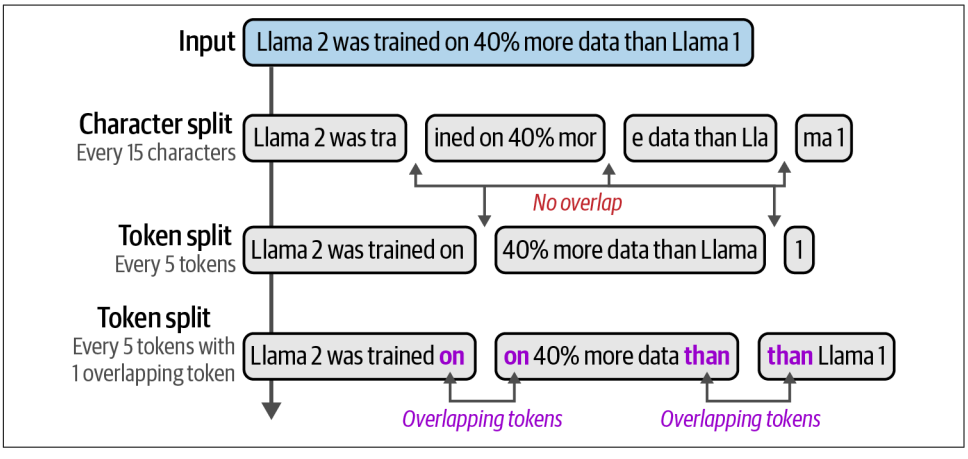

Figure 8-8. Several chunking methods and their effects on the input text. Overlapping chunks can be important to prevent the absence of context.

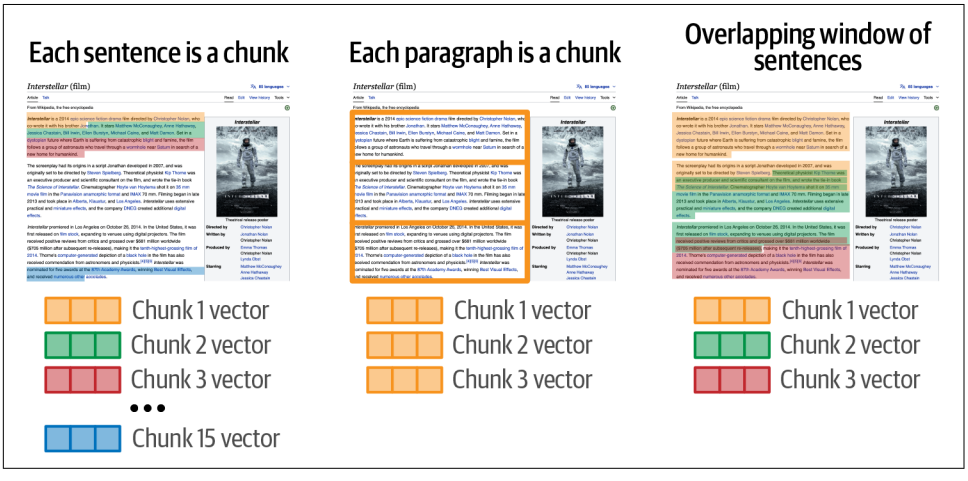

Figure 8-9. A number of possible options for chunking a document for embedding.

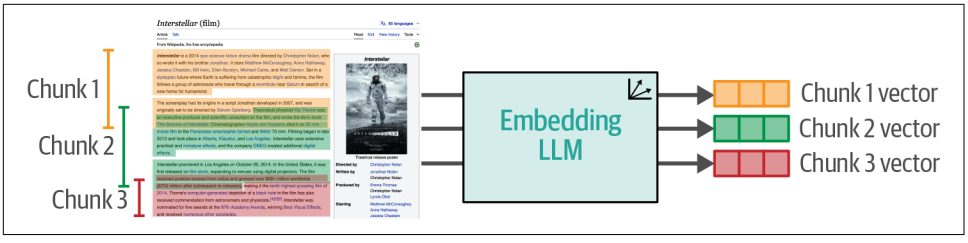

Figure 8-10. Chunking the text into overlapping segments is one strategy to retain more of the context around different segments.

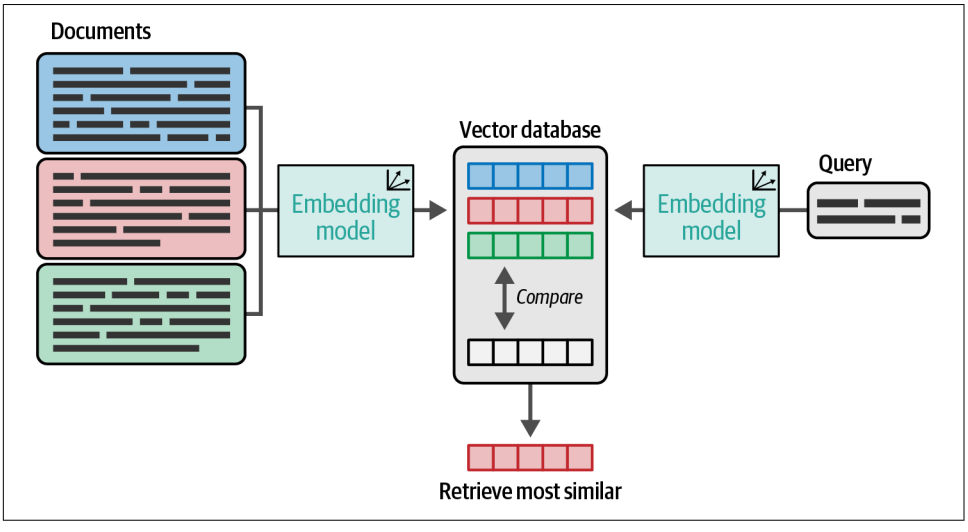

Figure 8-11. As we saw in Chapter 3, we can compare embeddings to quickly find the most similar documents to a query.

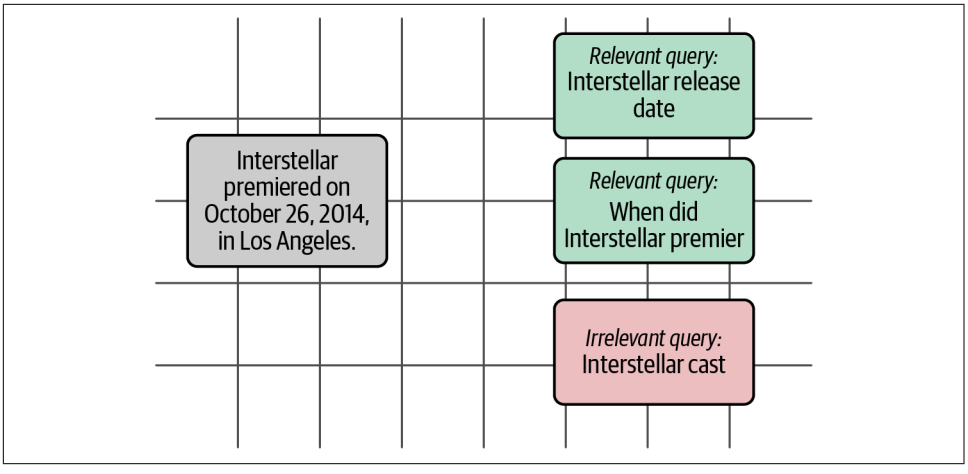

Figure 8-12. Before fine-tuning, the embeddings of both relevant and irrelevant queries may be close to a particular document.

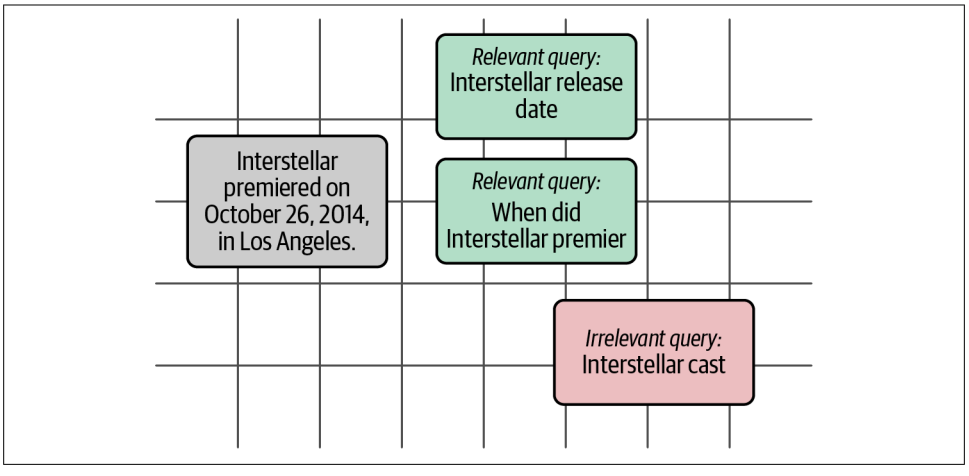

Figure 8-13. After the fine-tuning process, the text embedding model becomes better at this search task by incorporating how we define relevance on our dataset using the examples we provided of relevant and irrelevant documents.

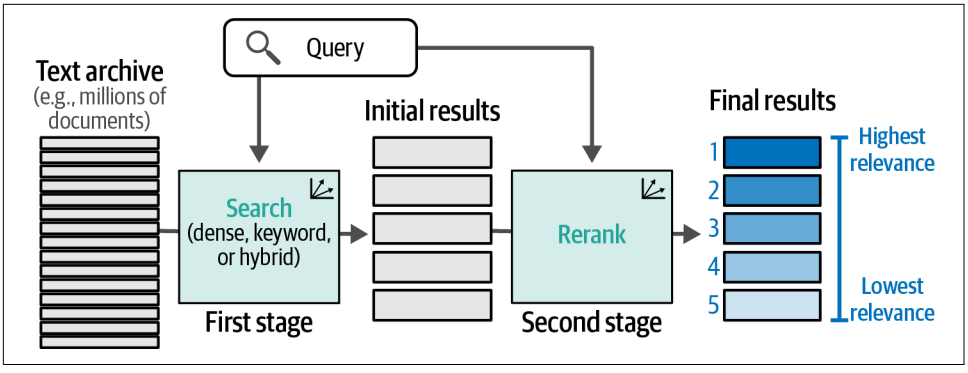

Figure 8-14. LLM rerankers operate as part of a search pipeline with the goal of reordering a number of shortlisted search results by relevance.

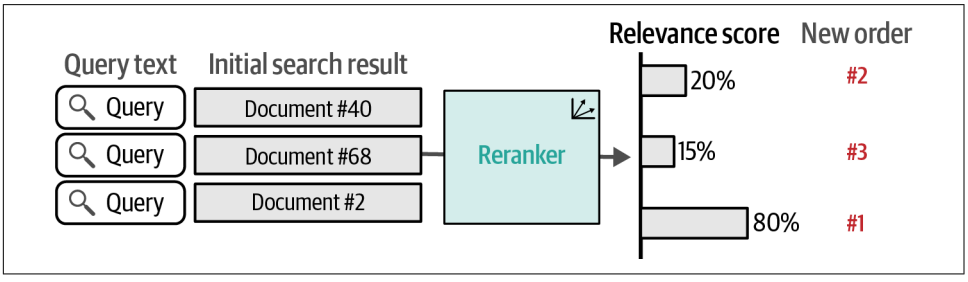

Figure 8-15. A reranker assigns a relevance score to each document by looking at the document and the query at the same time.

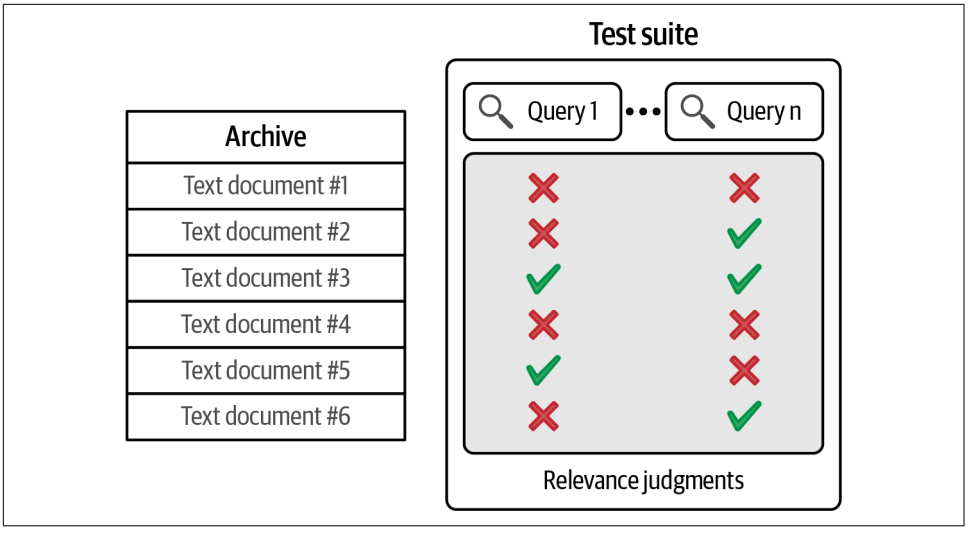

Figure 8-16. To evaluate search systems, we need a test suite including queries and relevance judgments indicating which documents in our archive are relevant for each query.

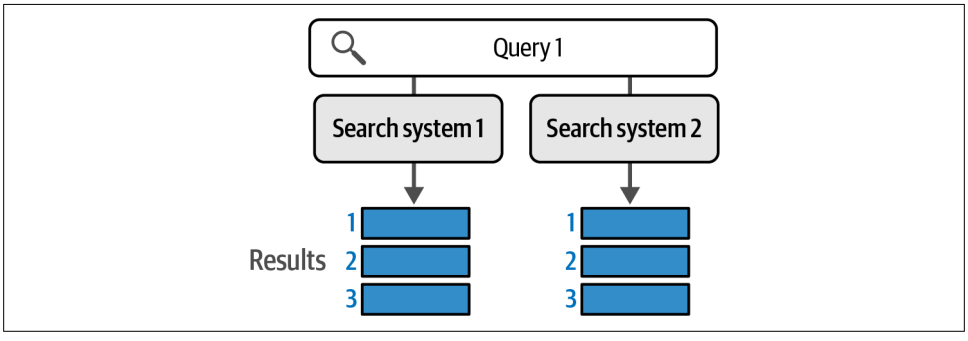

Figure 8-17. To compare two search systems, we pass the same query from our test suite to both systems and look at their top results.

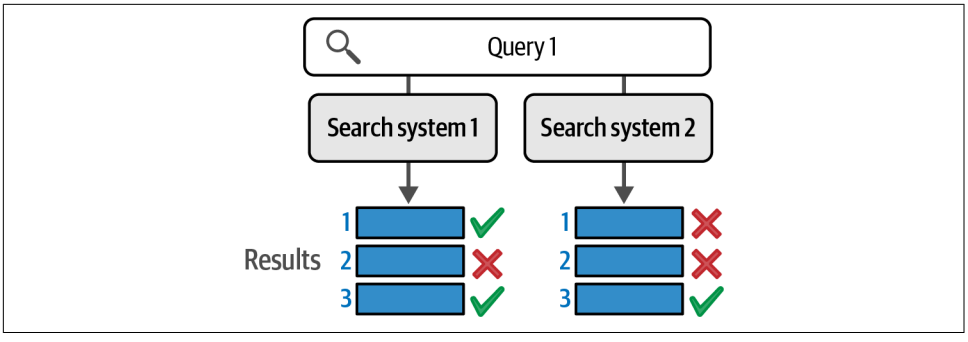

Figure 8-18. Looking at the relevance judgments from our test suite, we can see that system 1 did a better job than system 2.

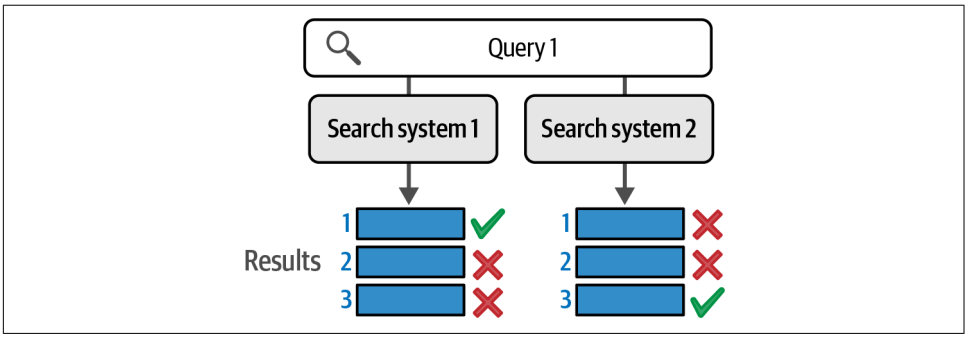

Figure 8-19. We need a scoring system that rewards system 1 for assigning a high position to a relevant result—even though both systems retrieved only one relevant result in their top three results.

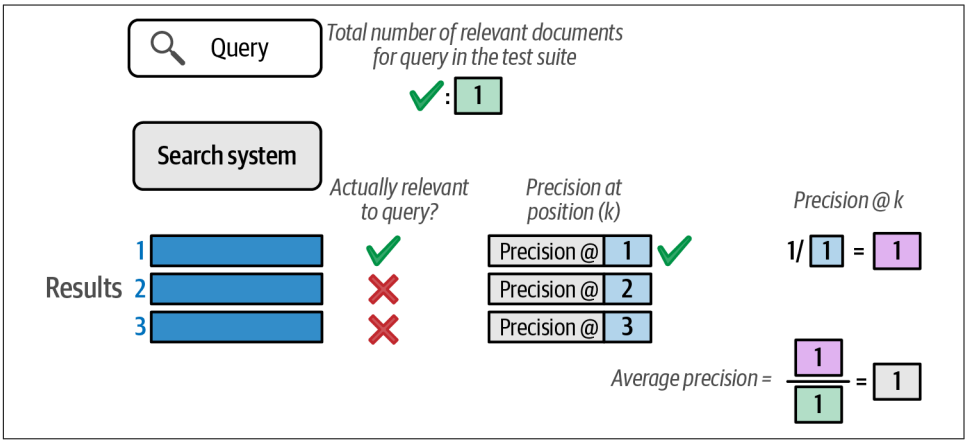

Figure 8-20. To calculate mean average precision, we start by calculating precision at each position, starting with position 1.

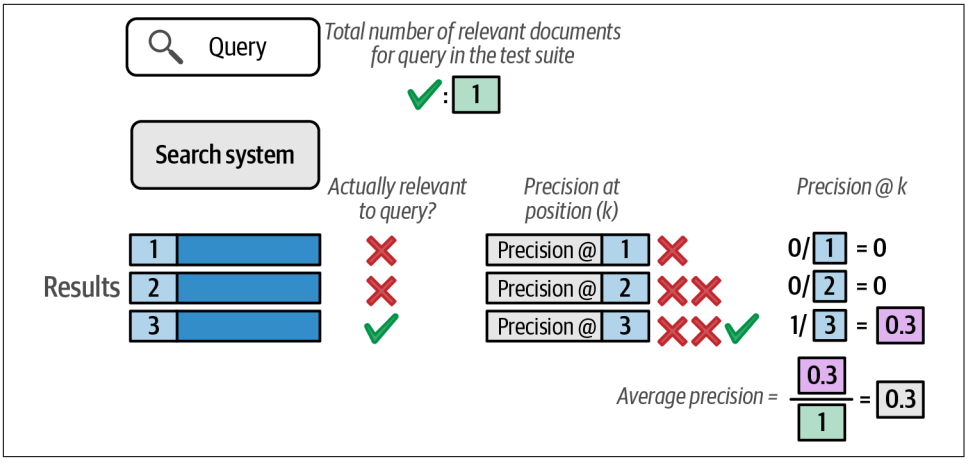

Figure 8-21. If the system places nonrelevant documents ahead of a relevant document, its precision score is penalized.

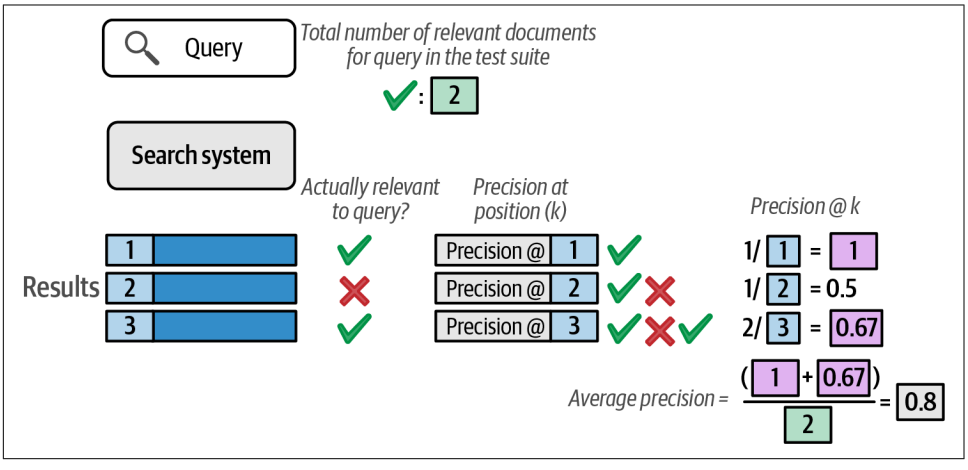

Figure 8-22. Average precision of a document with multiple relevant documents considers the precision at k results of all the relevant documents.

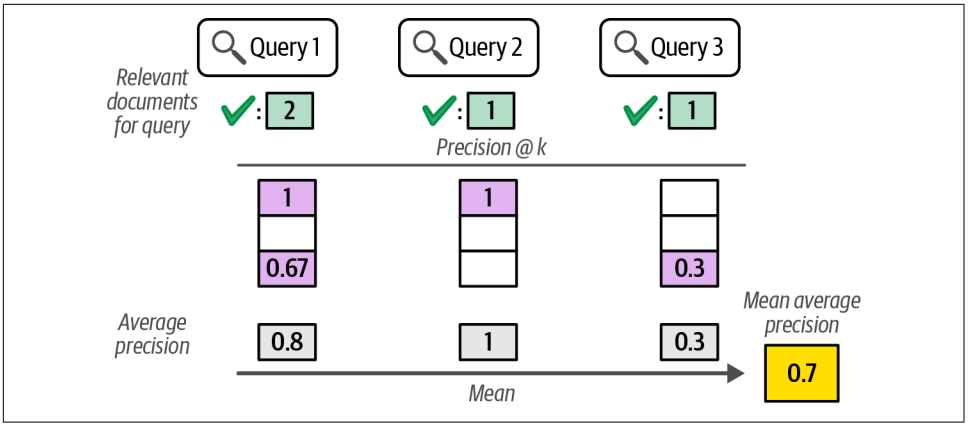

Figure 8-23. The mean average precision takes into consideration the average precision score of a system for every query in the test suite. By averaging them, it produces a single metric that we can use to compare a search system against another.

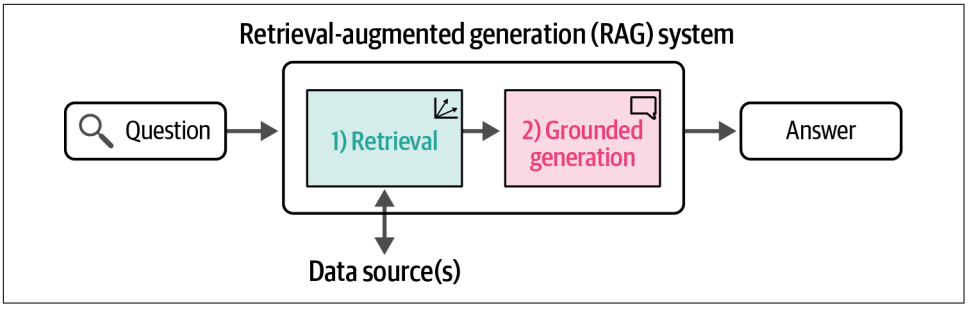

Figure 8-24. A basic RAG pipeline is made up of a search step followed by a grounded generation step where the LLM is prompted with the question and the information retrieved from the search step.

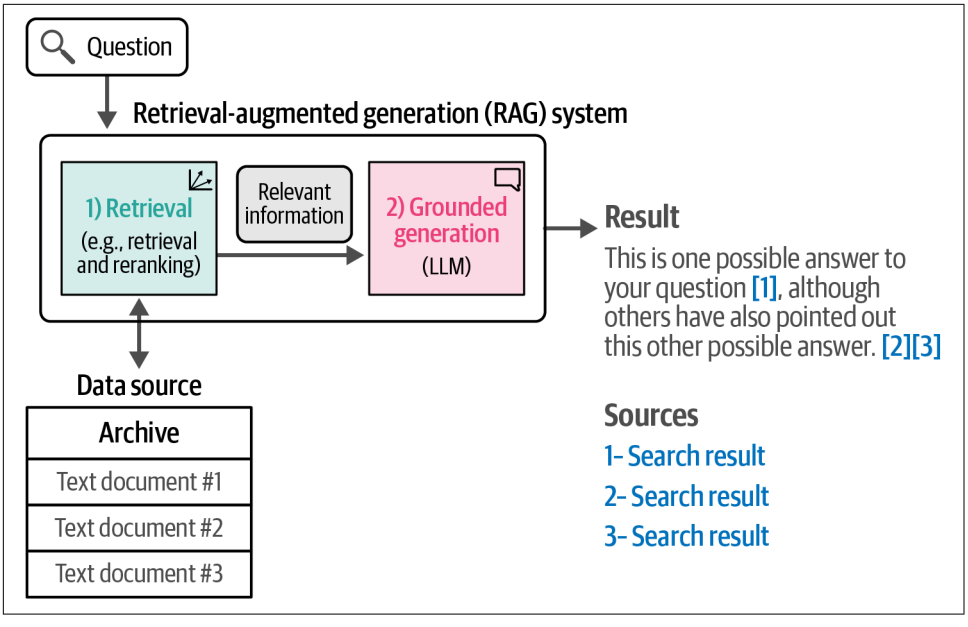

Figure 8-25. Generative search formulates answers and summaries at the end of a search pipeline while citing its sources (returned by the previous steps in the search system).

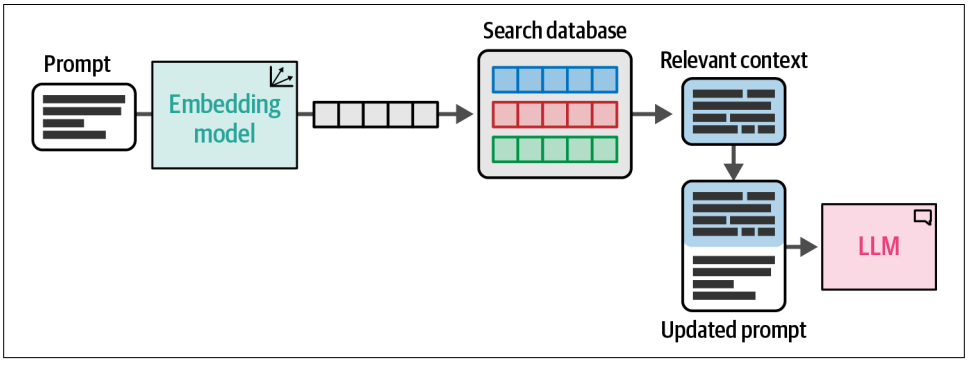

Figure 8-26. Find the most relevant information to an input prompt by comparing the similarities between embeddings. The most relevant information is added to the prompt before giving it to the LLM.

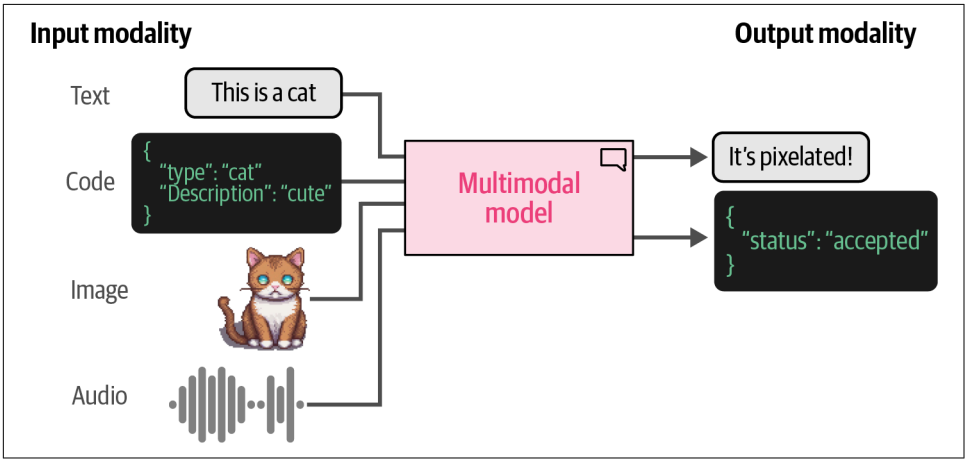

Figure 9-1. Models that are able to deal with different types (or modalities) of data, such as images, audio, video, or sensors, are said to be multimodal. It's possible for a model to accept a modality as input yet not be able to generate in that modality.

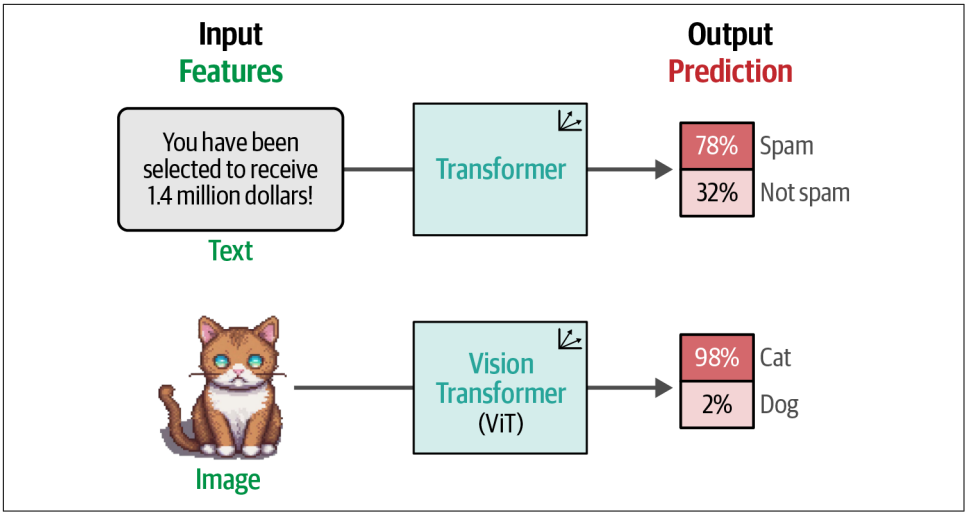

Figure 9-2. Both the original Transformer as well as the Vision Transformer take unstructured data, convert it to numerical representations, and finally use that for tasks like classification.

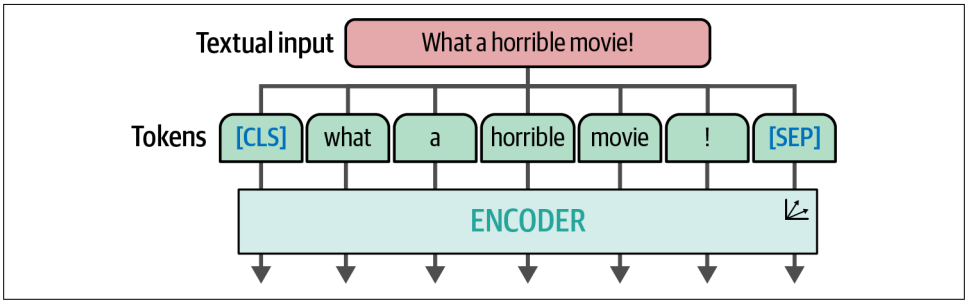

Figure 9-3. Text is passed to one or multiple encoders by first tokenizing it using a tokenizer.

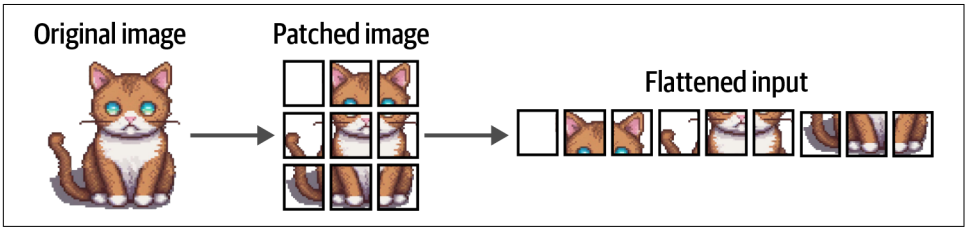

Figure 9-4. The "tokenization" process for image input. It converts an image into patches of subimages.

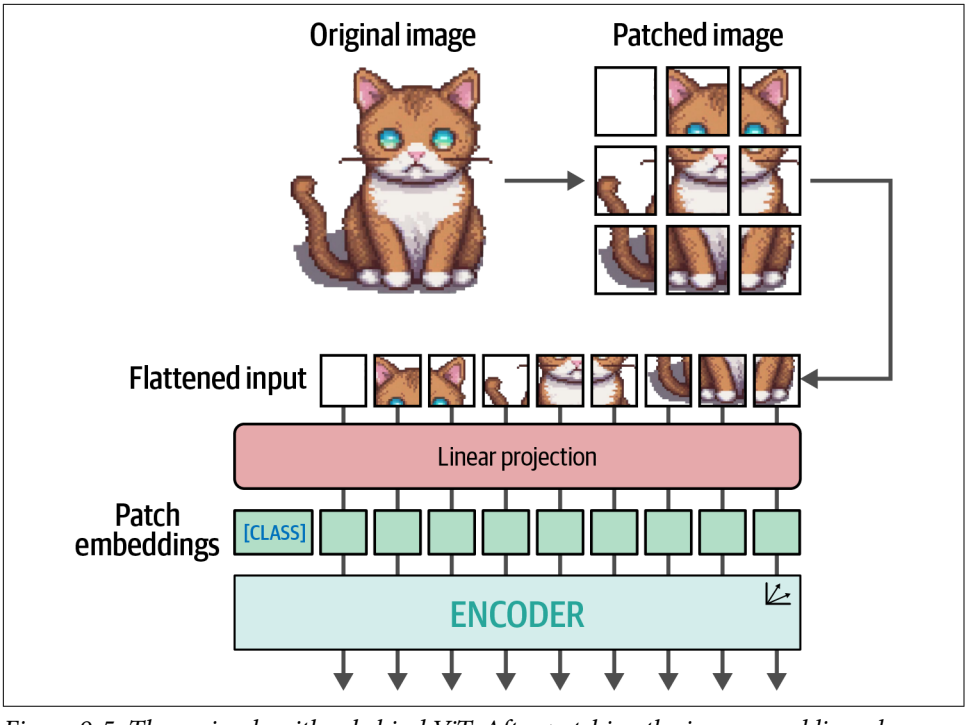

Figure 9-5. The main algorithm behind ViT. After patching the images and linearly projecting them, the patch embeddings are passed to the encoder and treated as if they were textual tokens.

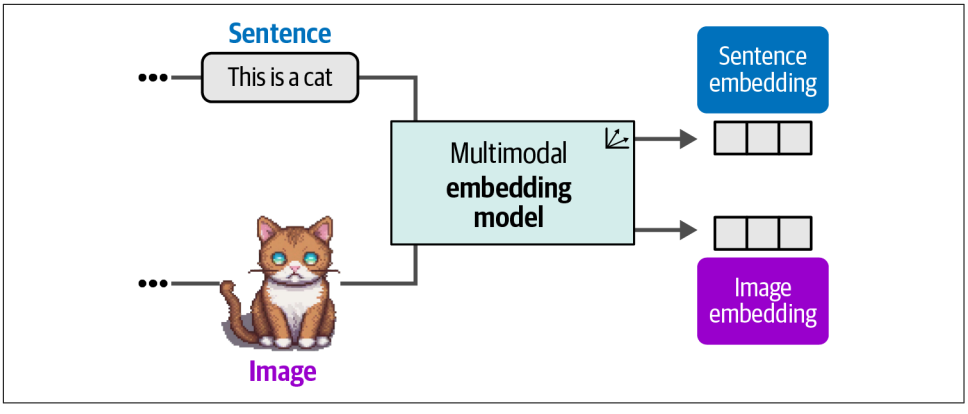

Figure 9-6. Multimodal embedding models can create embeddings for multiple modalities in the same vector space.

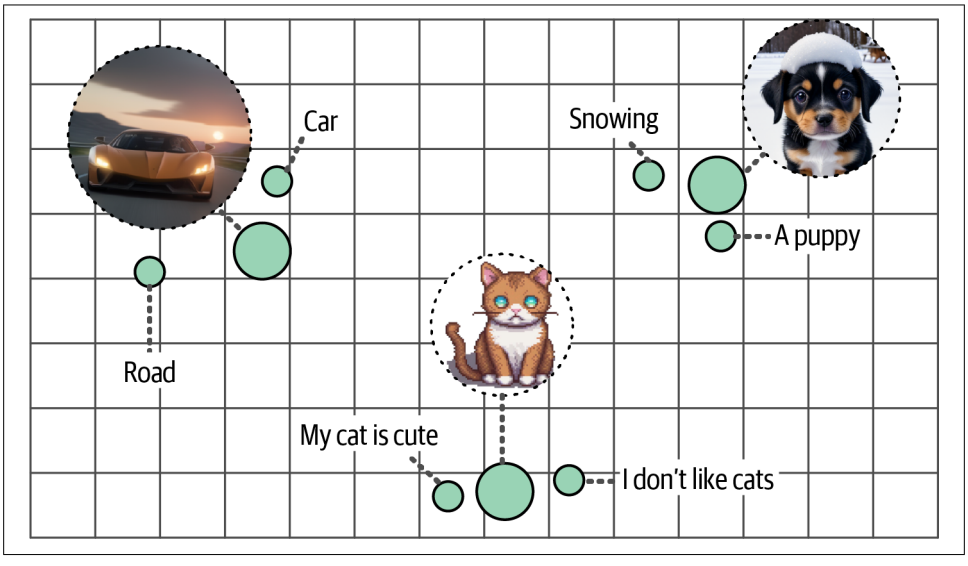

Figure 9-7. Despite having coming from different modalities, embeddings with similar meaning will be close to each other in vector space.

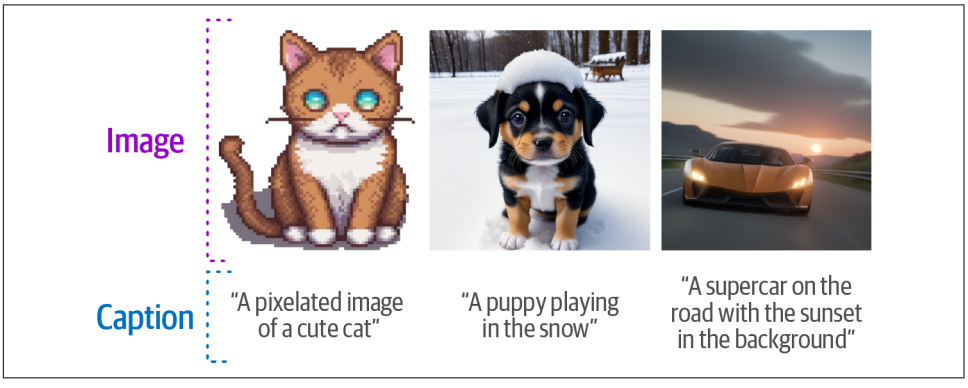

Figure 9-8. The type of data that is needed to train a multimodal embedding model.

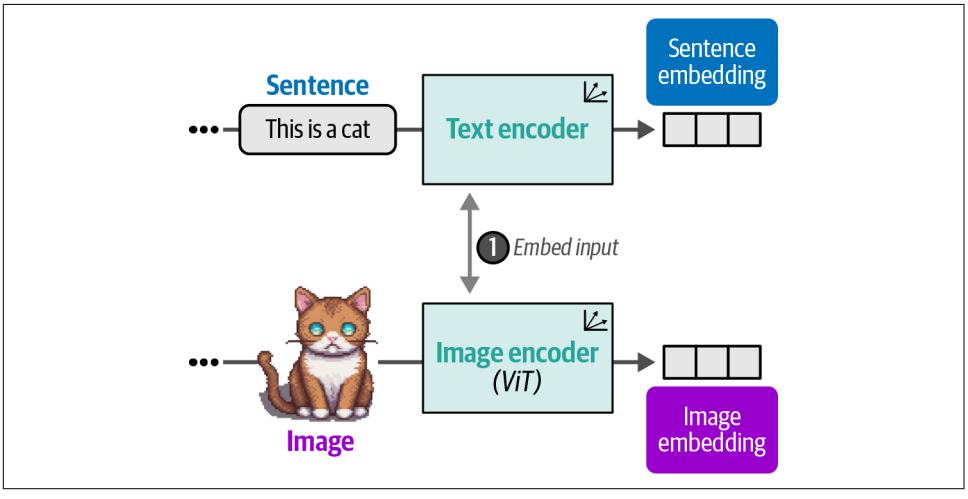

Figure 9-9. In the first step of training CLIP, both images and text are embedded using an image and text encoder, respectively.

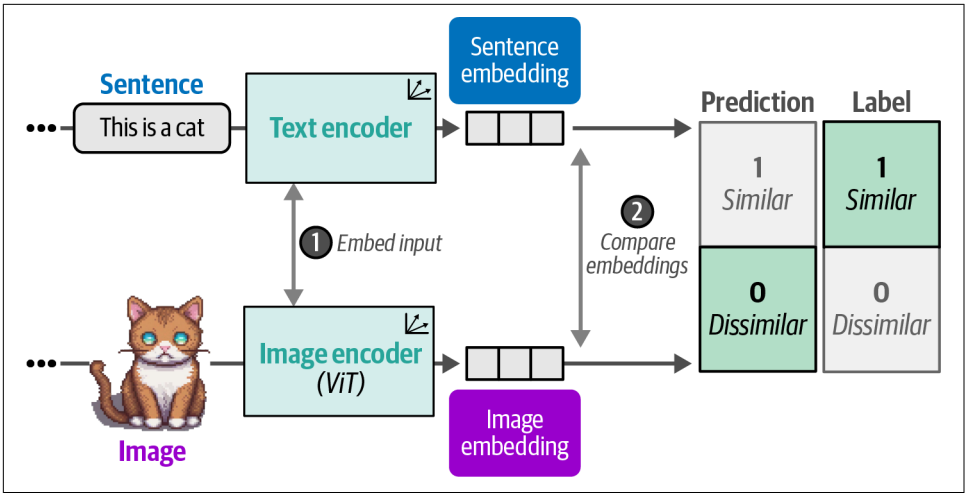

Figure 9-10. In the second step of training CLIP, the similarity between the sentence and image embedding is calculated using cosine similarity.

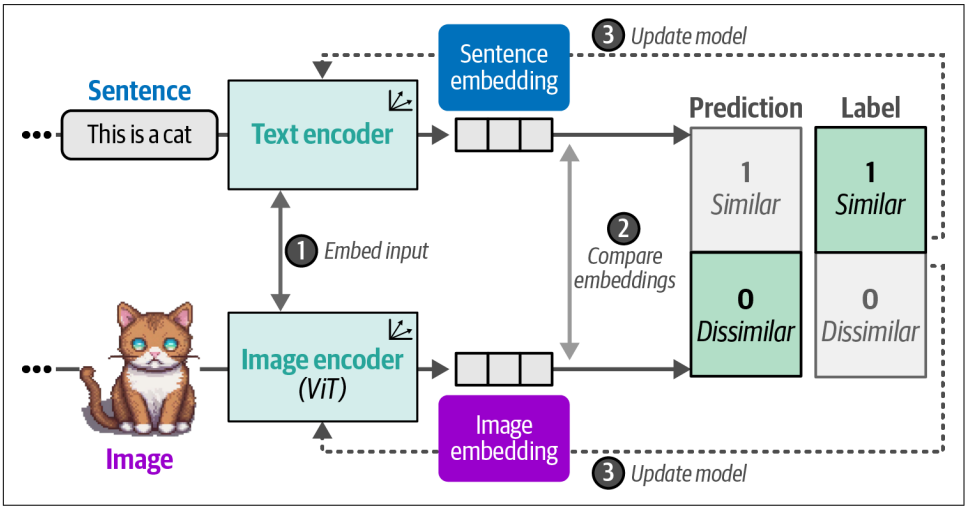

Figure 9-11. In the third step of training CLIP, the text and image encoders are updated to match what the intended similarity should be. This updates the embeddings such that they are closer in vector space if the inputs are similar.



Figure 9-12. An AI-generated image of a puppy playing in the snow.



Figure 9-13. The preprocessed input image by CLIP.

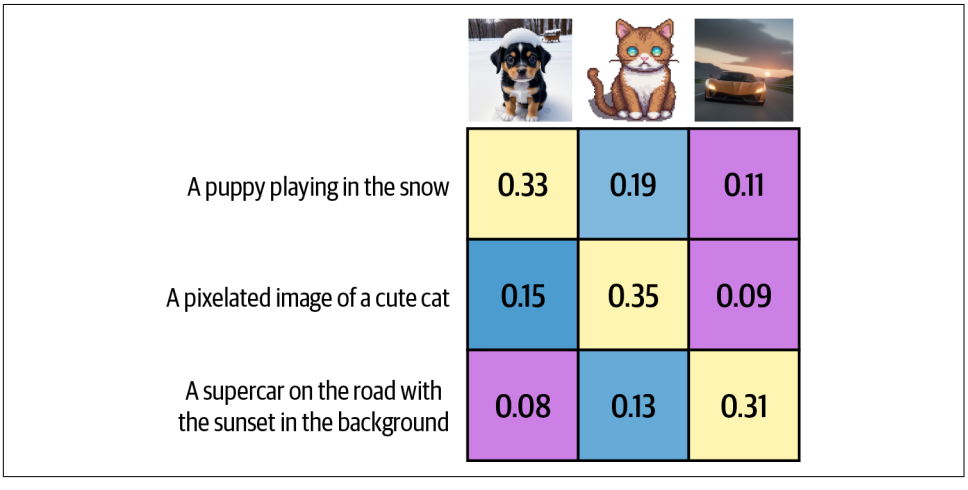

Figure 9-14. The similarity matrix between three images and three captions.

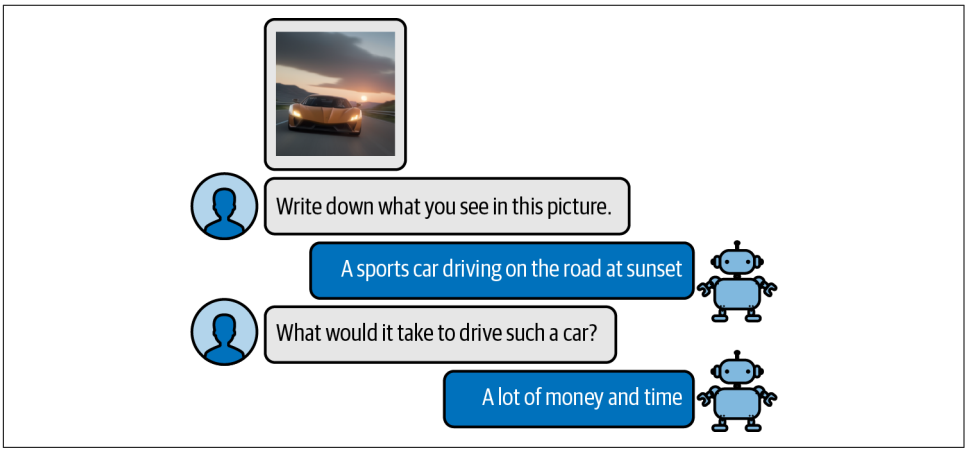

Figure 9-15. An example of a multimodal text generation model (BLIP-2) that can reason about input images.

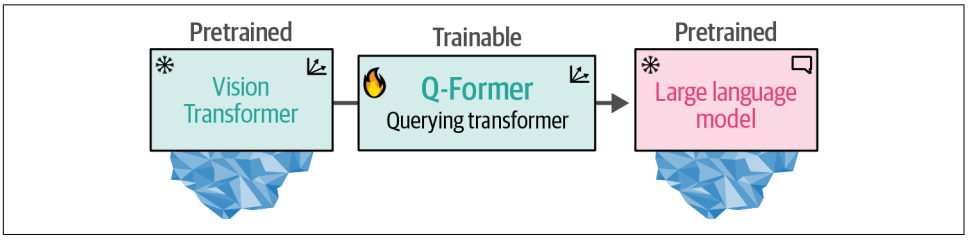

Figure 9-16. The Querying Transformer is the bridge between vision (ViT) and text (LLM) that is the only trainable component of the pipeline.

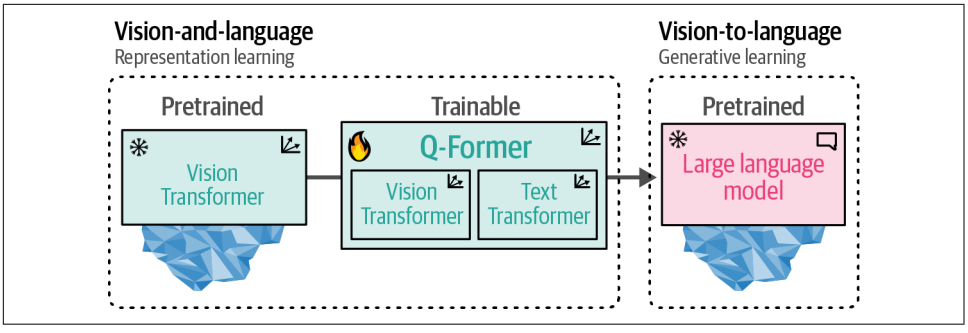

Figure 9-17. In step 1, representation learning is applied to learn representations for vision and language simultaneously. In step 2, these representations are converted to soft visual prompts to feed the LLM.

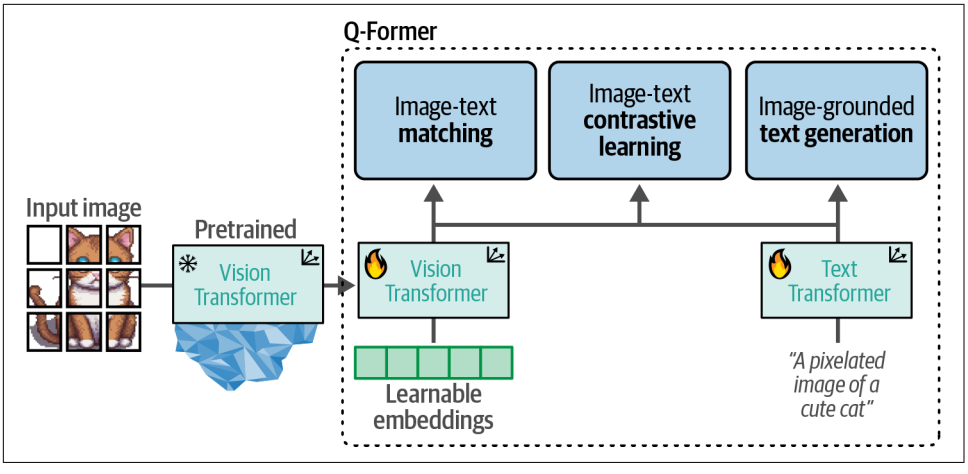

Figure 9-18. In step 1, the output of the frozen ViT is used together with its caption and trained on three contrastive-like tasks to learn visual-text representations.

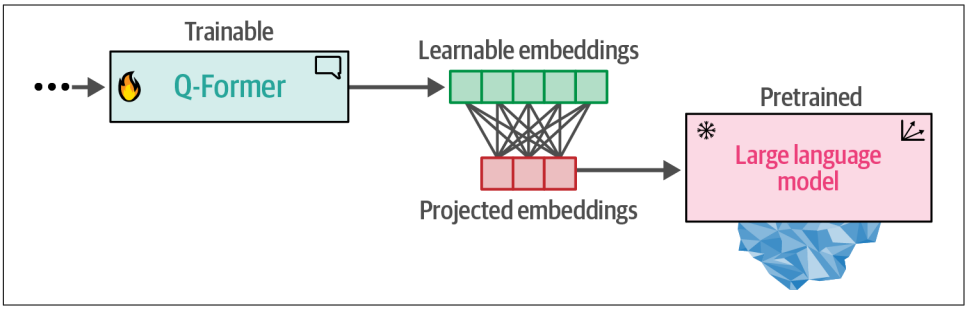

Figure 9-19. In step 2, the learned embeddings from the Q-Former are passed to the LLM through a projection layer. The projected embeddings serve as a soft visual prompt.

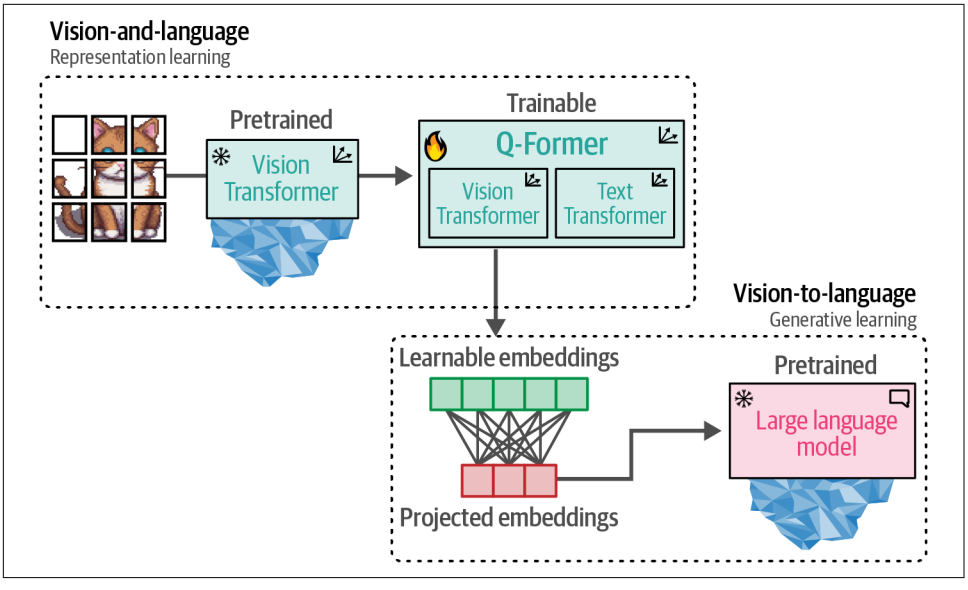

Figure 9-20. The full BLIP-2 procedure.

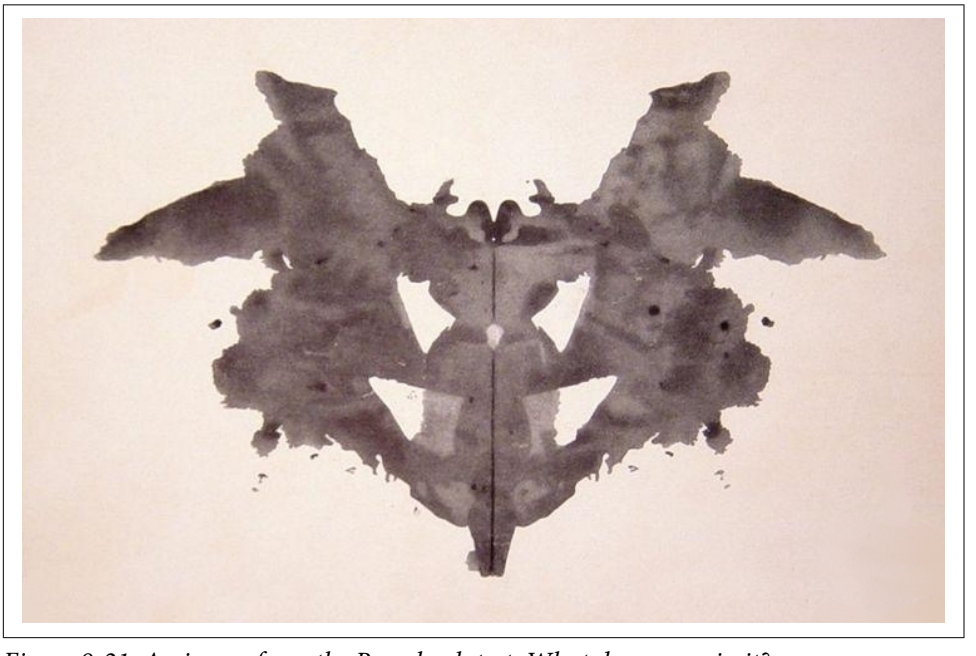

Figure 9-21. An image from the Rorschach test. What do you see in it?

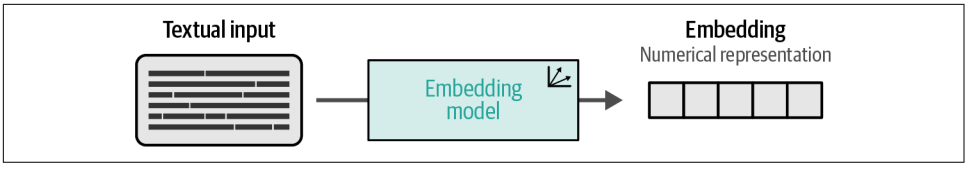

Figure 10-1. We use an embedding model to convert textual input, such as documents, sentences, and phrases, to numerical representations, called embeddings.

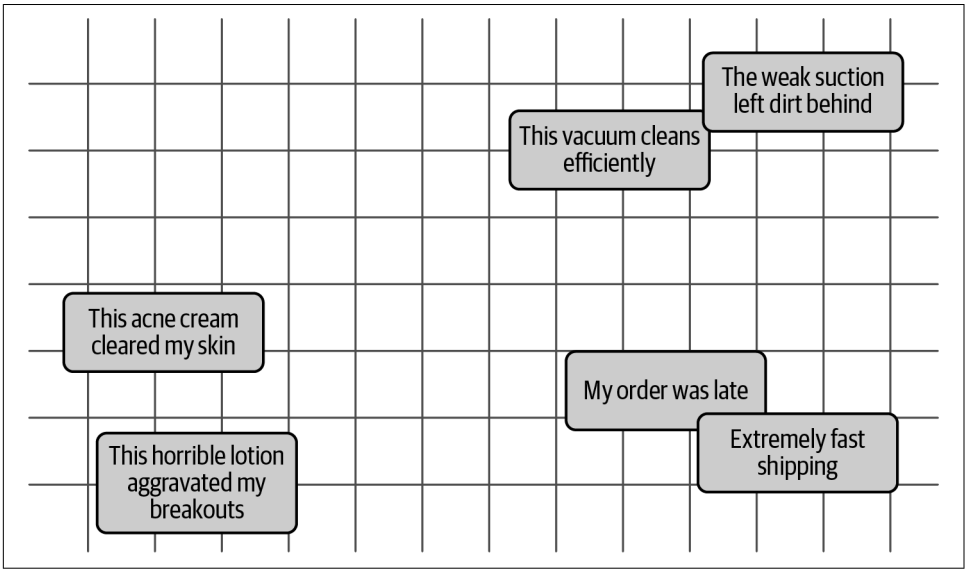

Figure 10-2. The idea of semantic similarity is that we expect textual data with similar meanings to be closer to each other in n-dimensional space (two dimensions are illustrated here).

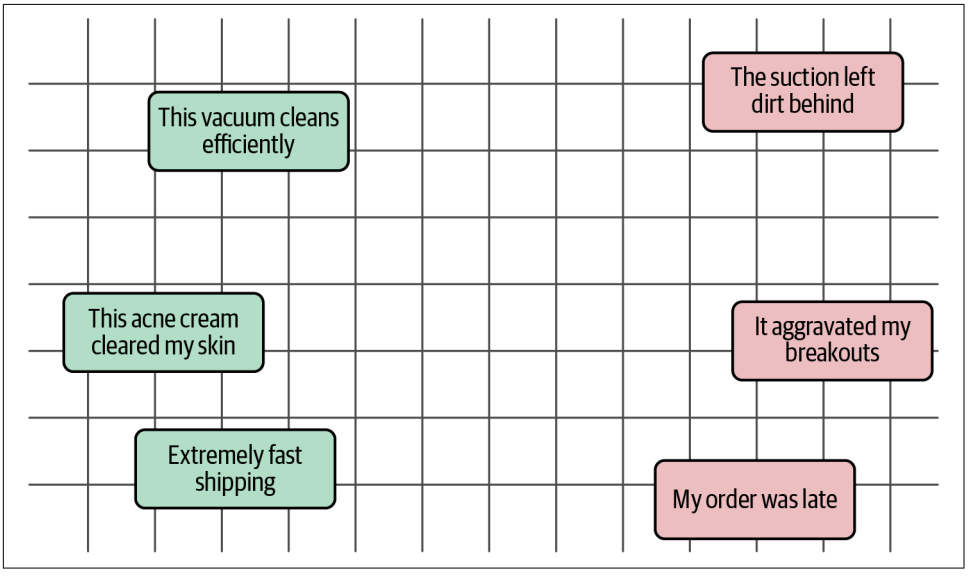

Figure 10-3. In addition to semantic similarity, an embedding model can be trained to focus on sentiment similarity. In this figure, negative reviews (red) are close to one another and dissimilar to positive reviews (green).

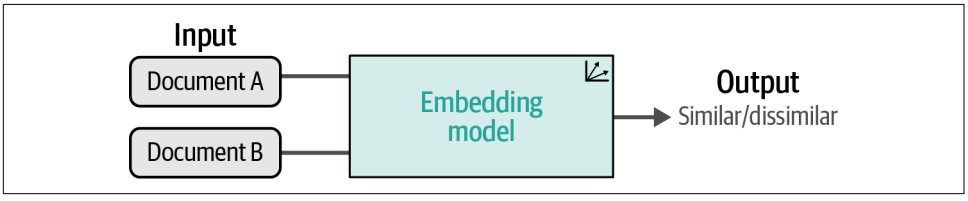

Figure 10-4. Contrastive learning aims to teach an embedding model whether documents are similar or dissimilar. It does so by presenting groups of documents to a model that are similar or dissimilar to a certain degree.

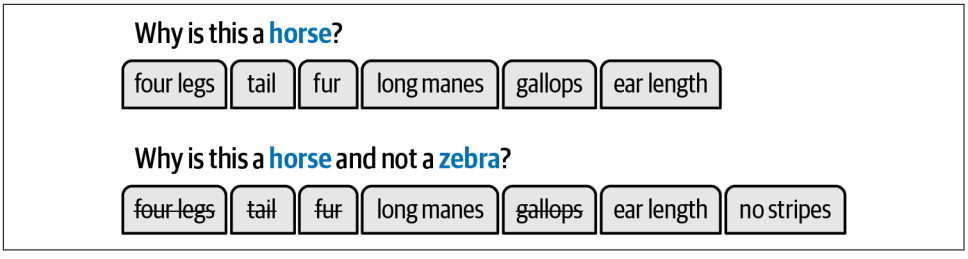

Figure 10-5. When we feed an embedding model different contrasts (degrees of similarity), it starts to learn what makes things different from one another and thereby the distinctive characteristics of concepts.

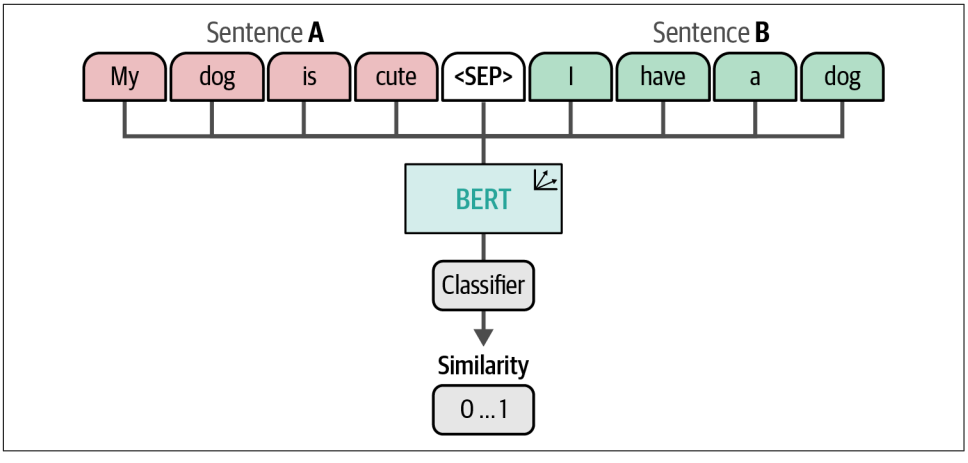

Figure 10-6. The architecture of a cross-encoder. Both sentences are concatenated, separated with a <SEP> token, and fed to the model simultaneously.

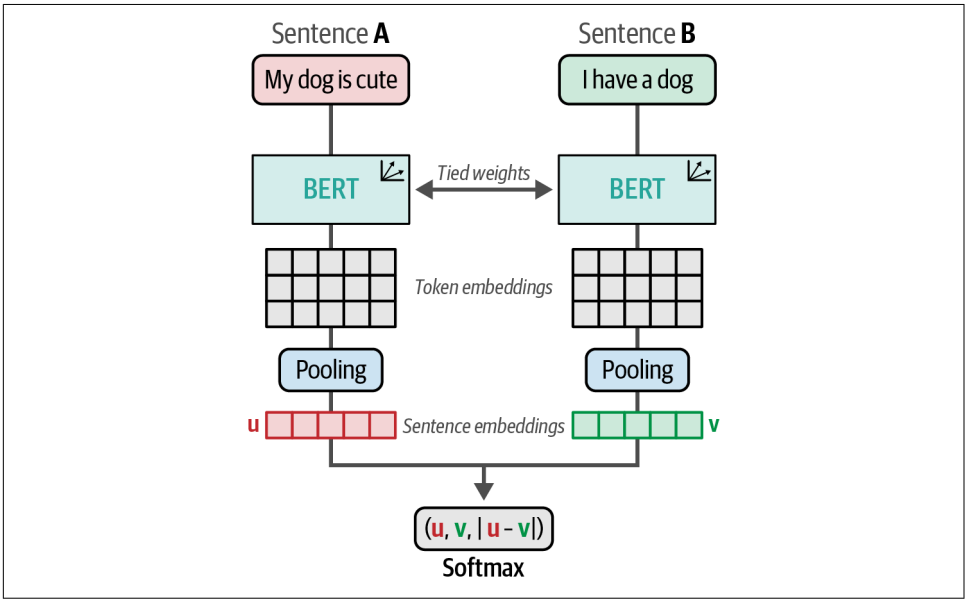

Figure 10-7. The architecture of the original `sentence-transformers` model, which leverages a Siamese network, also called a bi-encoder.

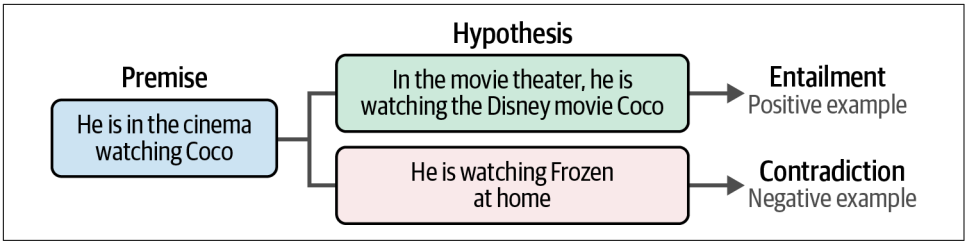

Figure 10-8. We can leverage the structure of NLI datasets to generate negative examples (contradiction) and positive examples (entailments) for contrastive learning.

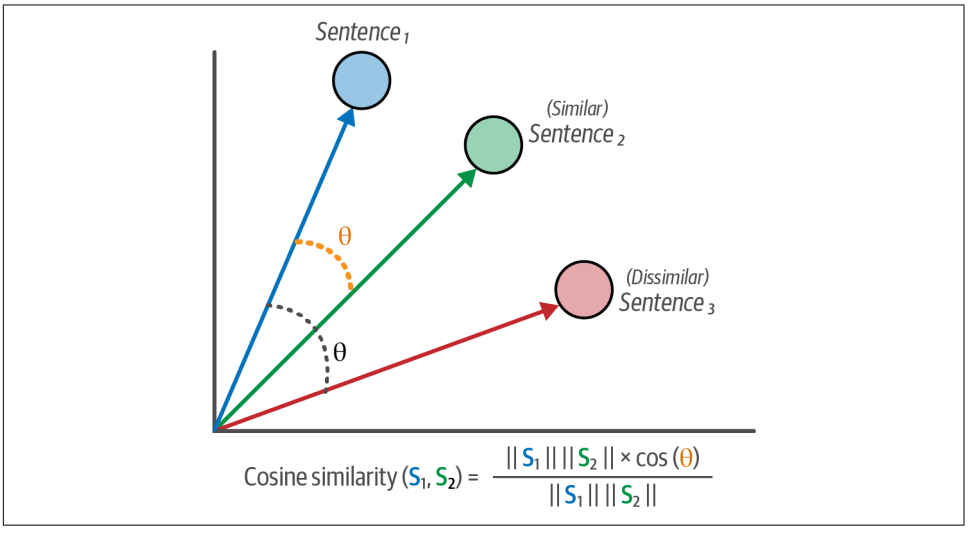

Figure 10-9. Cosine similarity loss aims to minimize the cosine distance between semantically similar sentences and to maximize the distance between semantically dissimilar sentences.

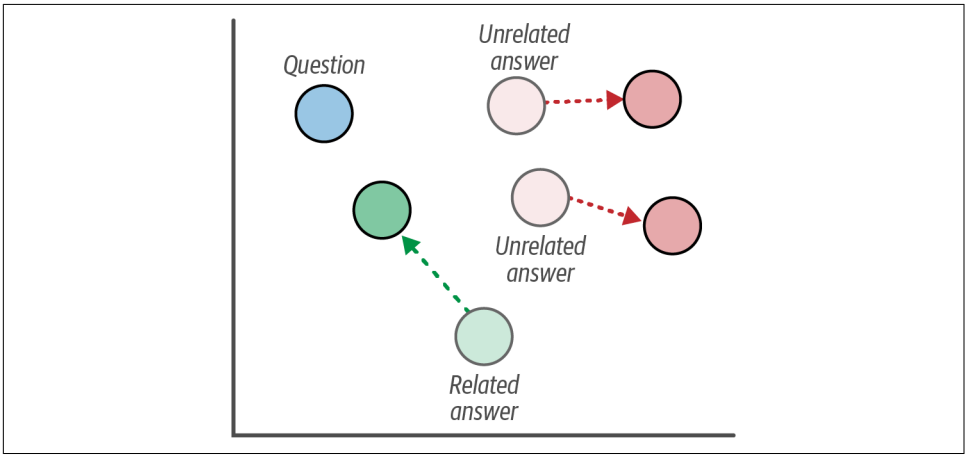

Figure 10-10. Multiple negatives ranking loss aims to minimize the distance between related pairs of text, such as questions and answers, and maximize the distance between unrelated pairs, such as questions and unrelated answers.

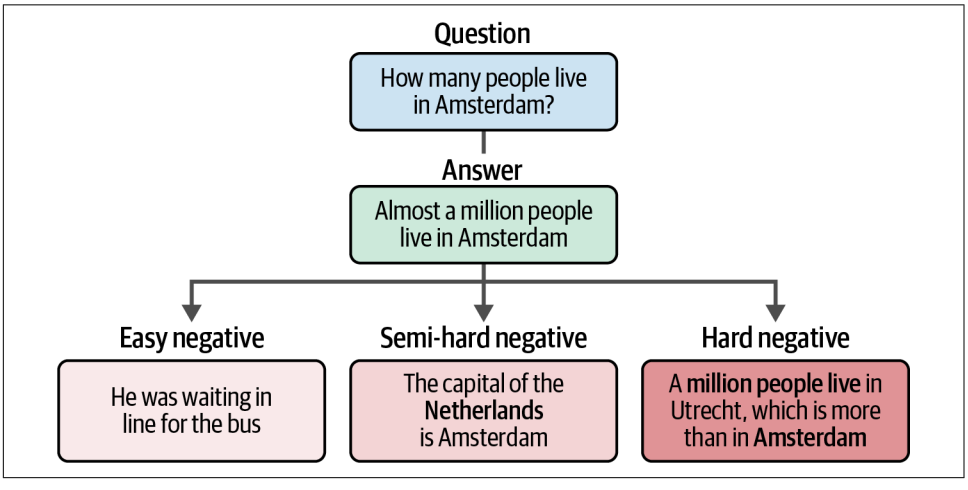

Figure 10-11. An easy negative is typically unrelated to both the question and answer. A semi-hard negative has some similarities to the topic of the question and answer but is somewhat unrelated. A hard negative is very similar to the question but is generally the wrong answer.

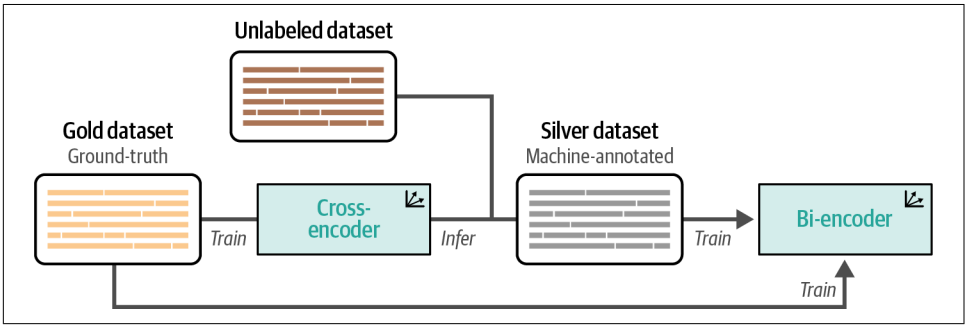

Figure 10-12. Augmented SBERT works through training a cross-encoder on a small gold dataset, then using that to label an unlabeled dataset to generate a larger silver dataset. Finally, both the gold and silver datasets are used to train the bi-encoder.

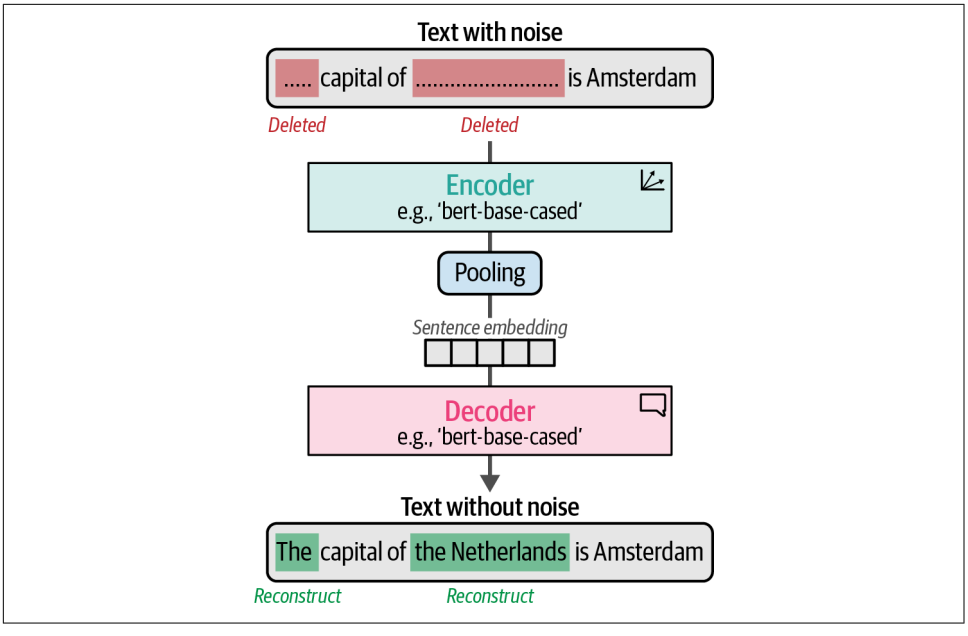

Figure 10-13. TSDAE randomly removes words from an input sentence that is passed through an encoder to generate a sentence embedding. From this sentence embedding, the original sentence is reconstructed.

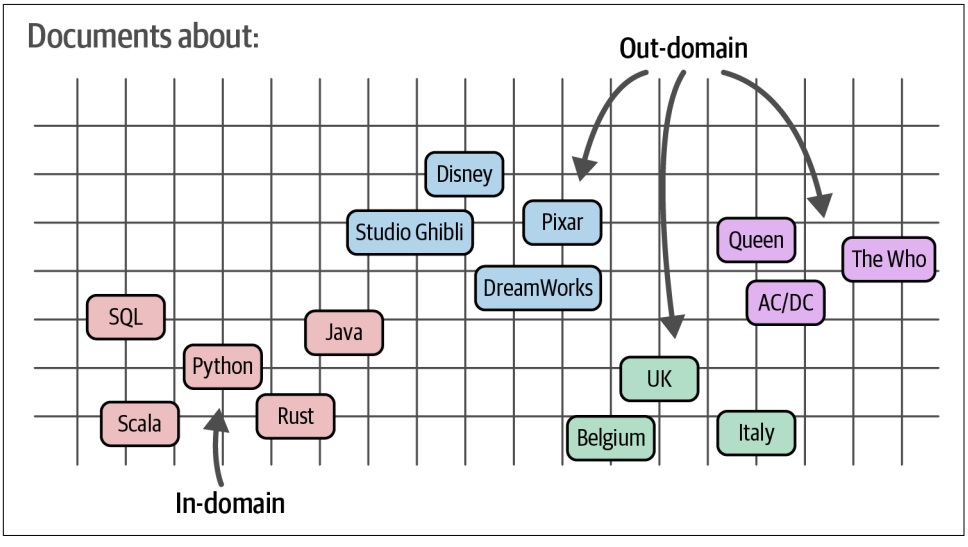

Figure 10-14. In domain adaptation, the aim is to create and generalize an embedding model from one domain to another.

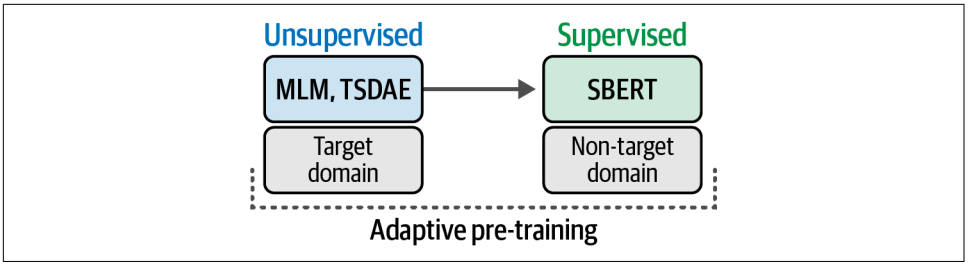

Figure 10-15. Domain adaptation can be performed with adaptive pretraining and adaptive fine-tuning.

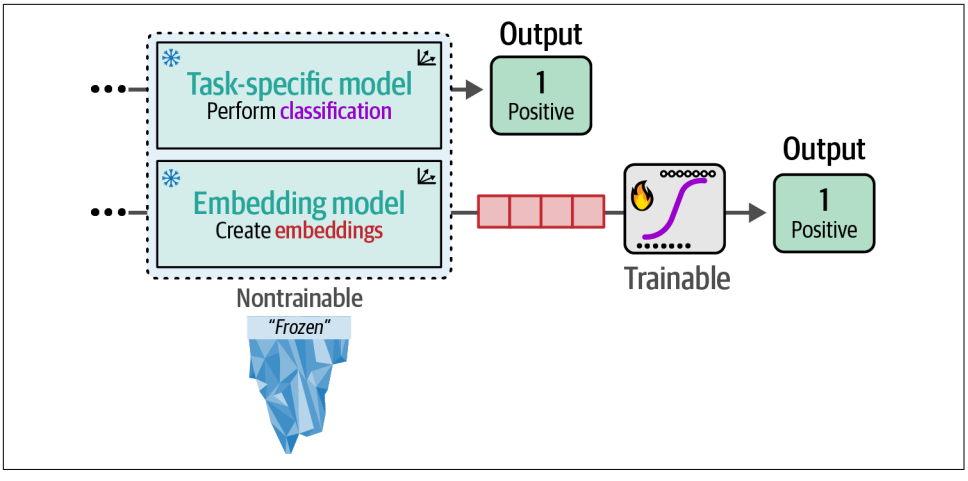

Figure 11-1. In Chapter 4, we used pretrained models to perform classification without updating their weight. These models were kept "frozen."

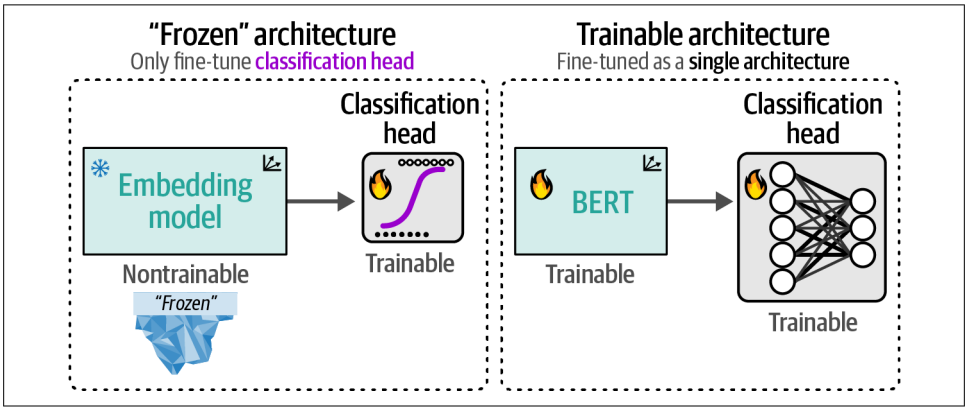

Figure 11-2. Compared to the "frozen" architecture, we instead train both the pretrained BERT model and the classification head. A backward pass will start at the classification head and go through BERT.

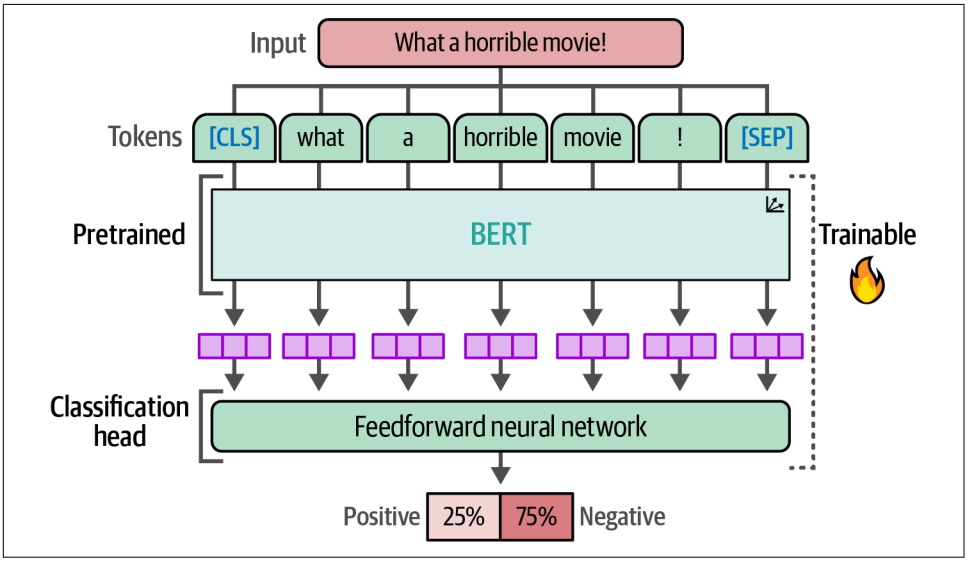

Figure 11-3. The architecture of a task-specific model. It contains a pretrained representation model (e.g., BERT) with an additional classification head for the specific task.

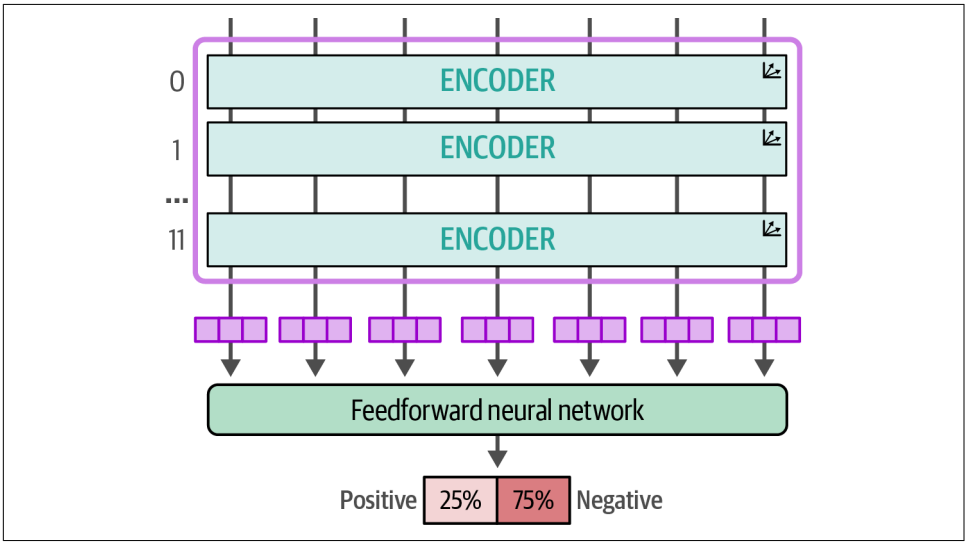

Figure 11-4. The basic architecture of BERT with the additional classification head.

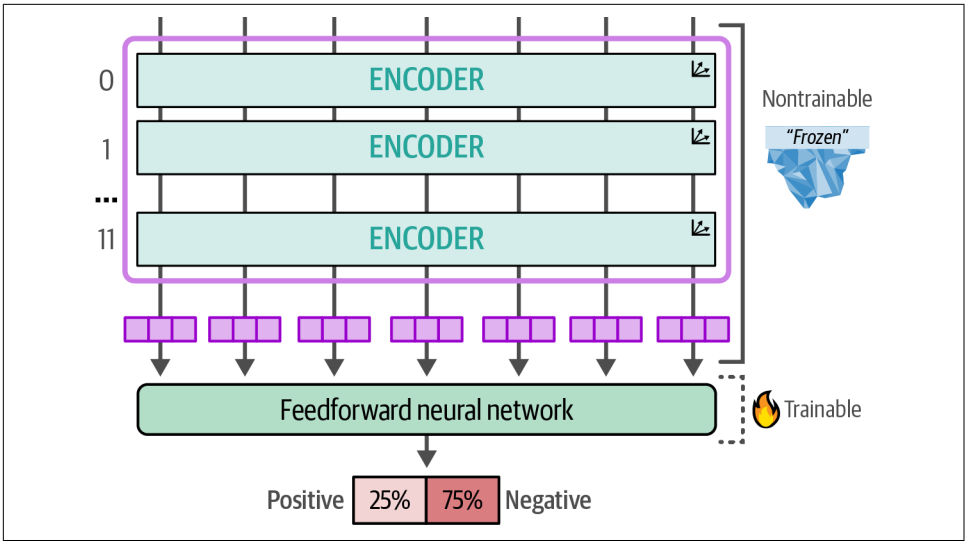

Figure 11-5. We fully freeze all encoder blocks and embedding layers such that the BERT model does not learn new representations during fine-tuning.

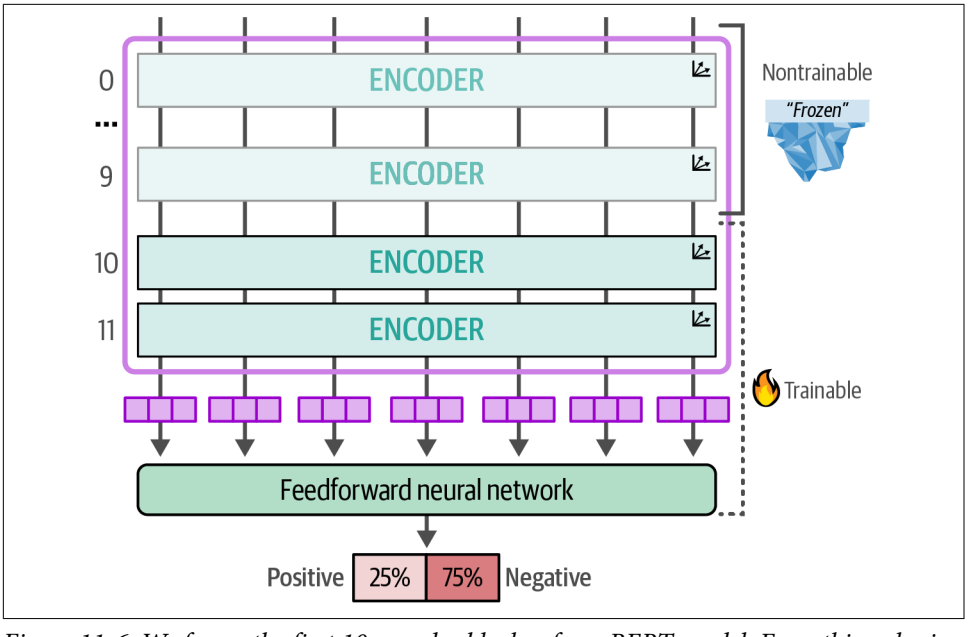

Figure 11-6. We freeze the first 10 encoder blocks of our BERT model. Everything else is trainable and will be fine-tuned.

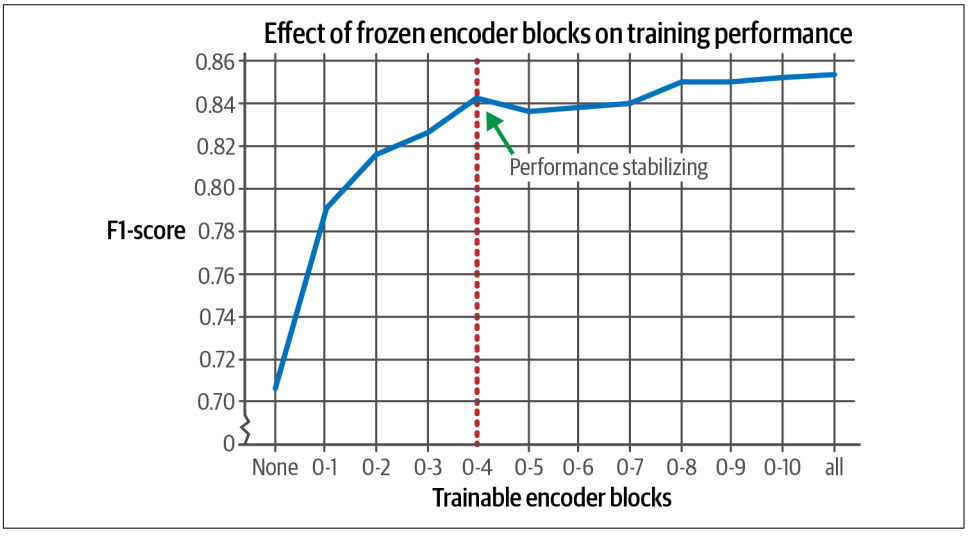

Figure 11-7. The effect of freezing certain encoder blocks on the performance of the model. Training more blocks leads to improved performance but stabilizes early on.

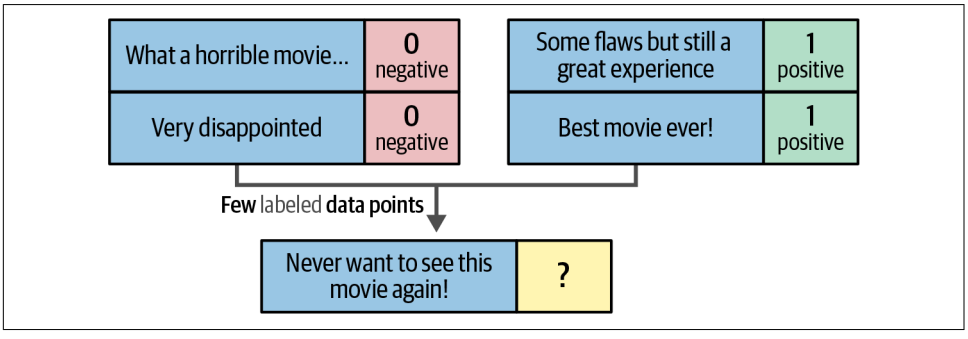

Figure 11-8. In few-shot classification, we only use a few labeled data points to learn from.

Text	Class
I write my code in Python	Programming languages
I should practice SQL	Programming languages
My dog is a labrador	Pets
I have a Siamese cat	Pets

Figure 11-9. Data in two classes: text about programming languages and text about pets.

Text 1	Text 2	Pair type
I write my code in Python	I should practice SQL	Positive
My dog is a labrador	I have a Siamese cat	Positive
I write my code in Python	My dog is a labrador	Negative
I have a Siamese cat	I should practice SQL	Negative

Figure 11-10. Step 1: sampling training data. We assume sentences within a class are similar and create positive pairs while sentences in different classes become negative pairs.

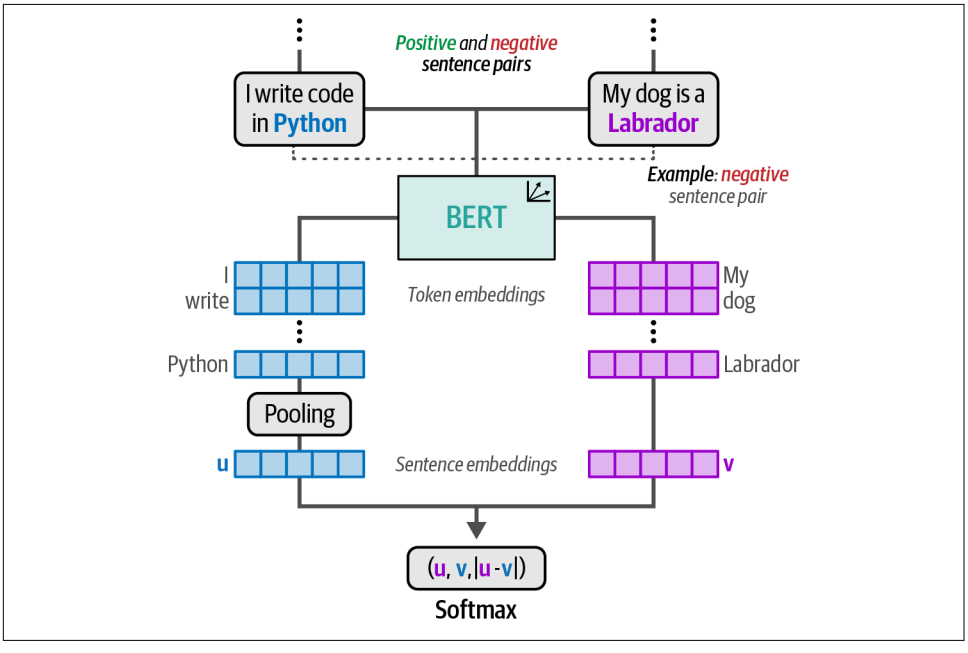

Figure 11-11. Step 2: Fine-tuning a `SentenceTransformers` model. Using contrastive learning, embeddings are learned from positive and negative sentence pairs.

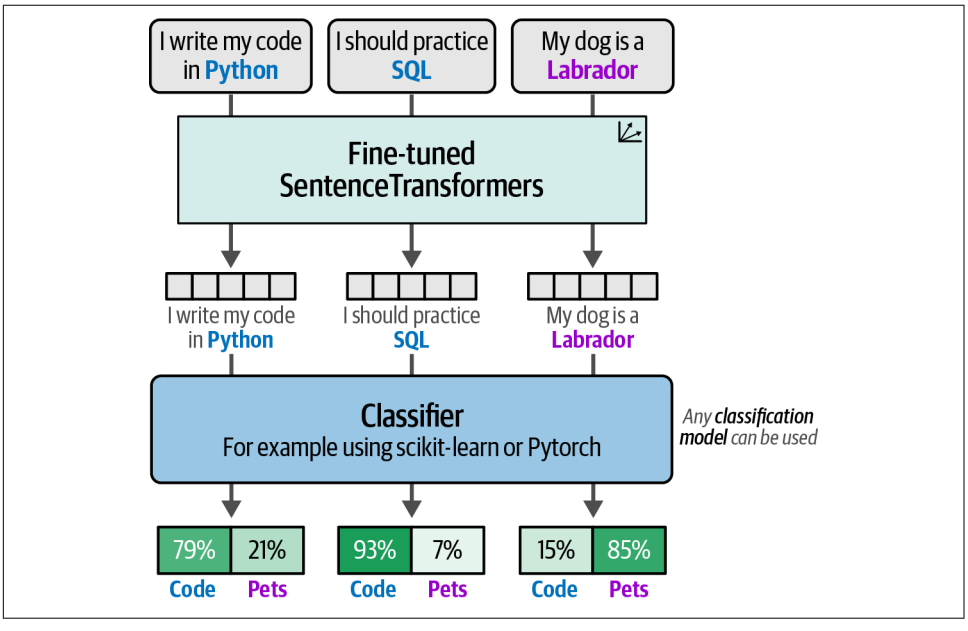

Figure 11-12. Step 3: Training a classifier. The classifier can be any scikit-learn model or a classification head.

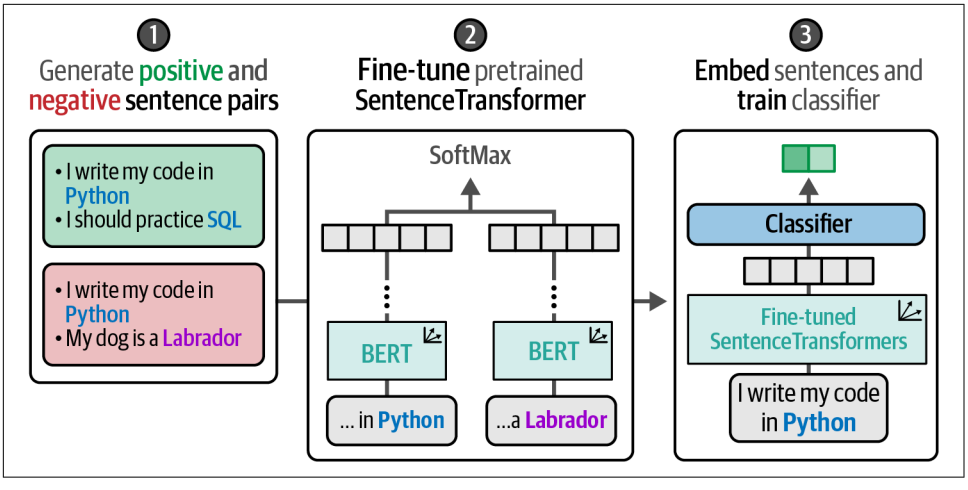

Figure 11-13. The three main steps of SetFit.

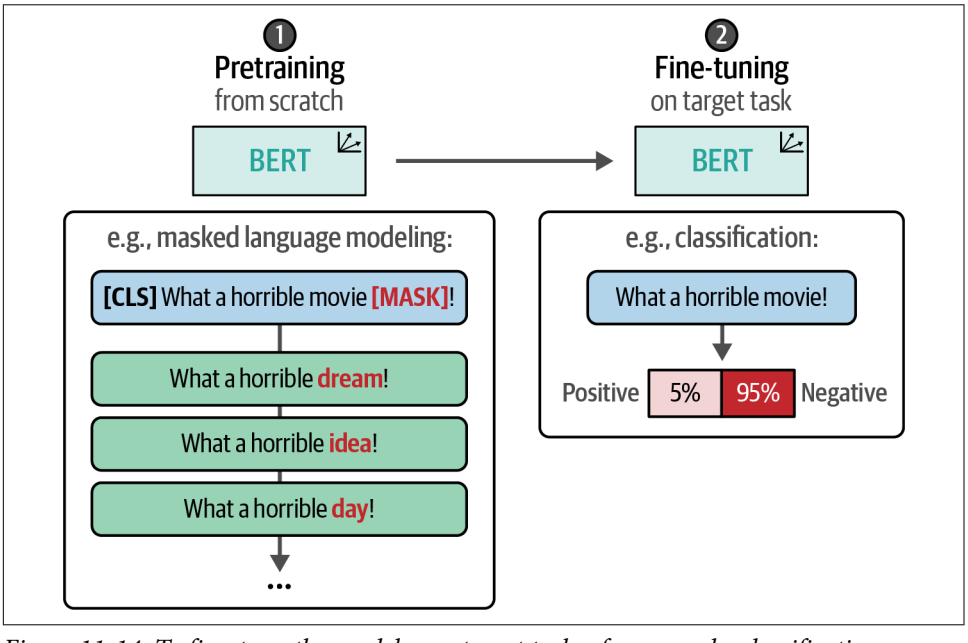

Figure 11-14. To fine-tune the model on a target task—for example, classification—we either start with pretraining a BERT model or use a pretrained one.

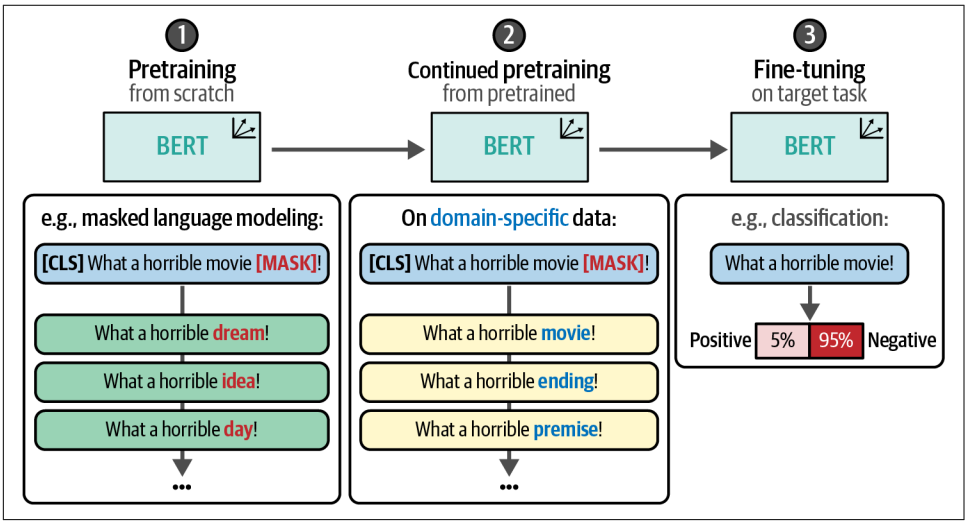

Figure 11-15. Instead of a two-step approach, we can add another step that continues to pretrain the pretrained model before fine-tuning it on the target task. Notice how the masks were filled with abstract concepts in 1 while they were filled with movie-specific concepts in 2.

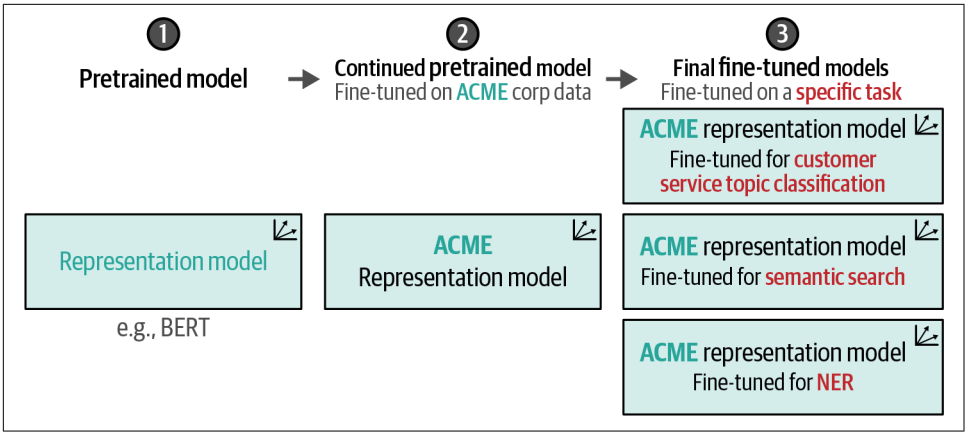

Figure 11-16. The three-step approach illustrated for specific use cases.

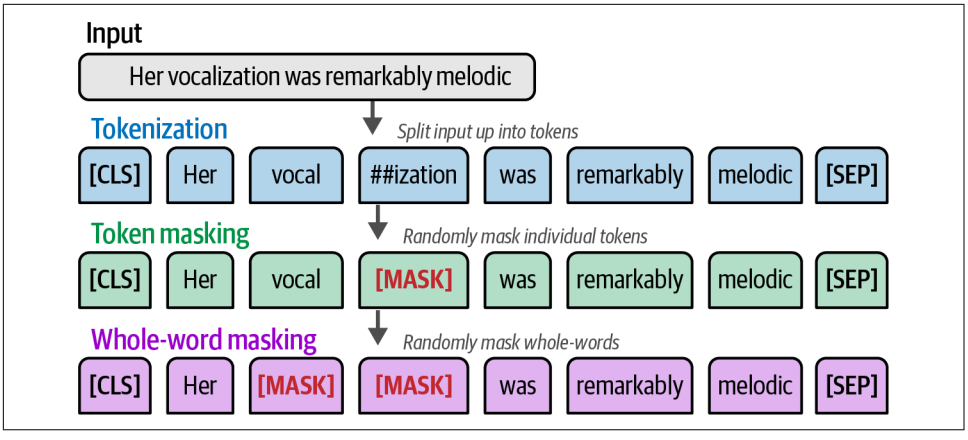

Figure 11-17. Different methods for randomly masking tokens.

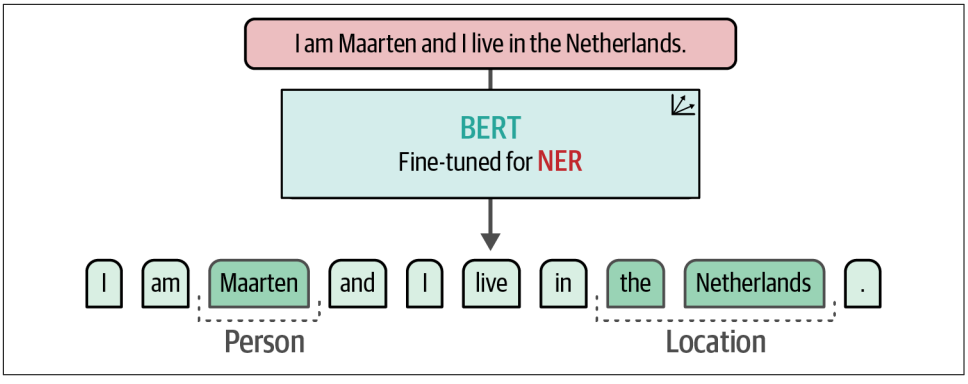

Figure 11-18. Fine-tuning a BERT model for NER allows for the detection of named entities, such as people or locations.

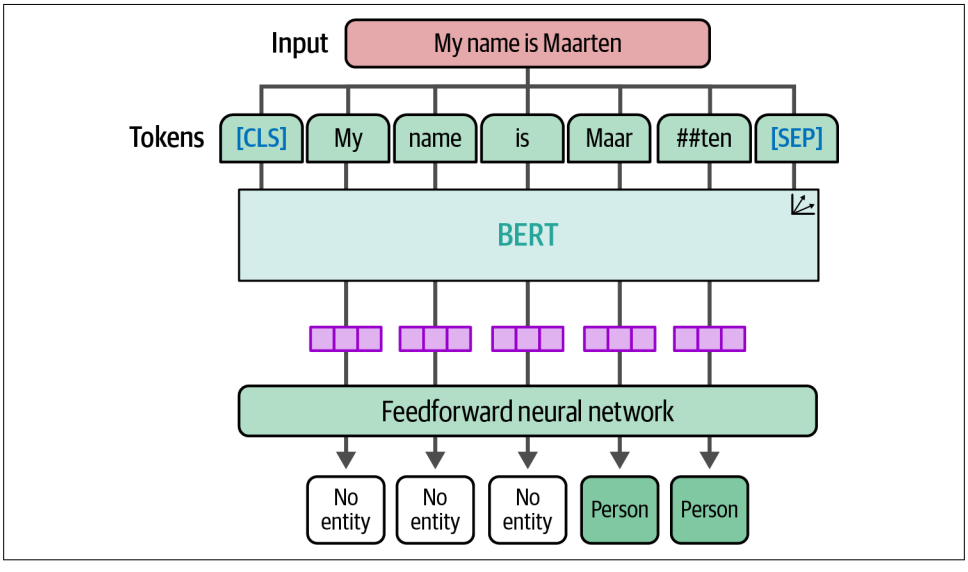

Figure 11-19. During the fine-tuning process of a BERT model, individual tokens are classified instead of words or entire documents.

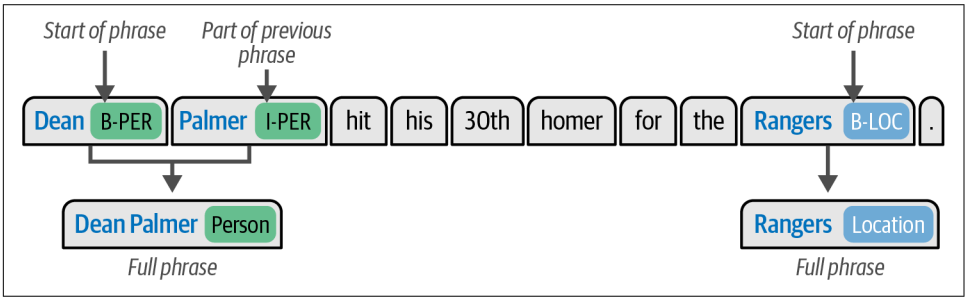

Figure 11-20. By indicating the start and end of the phrase with the same entity, we can recognize entities of entire phrases.

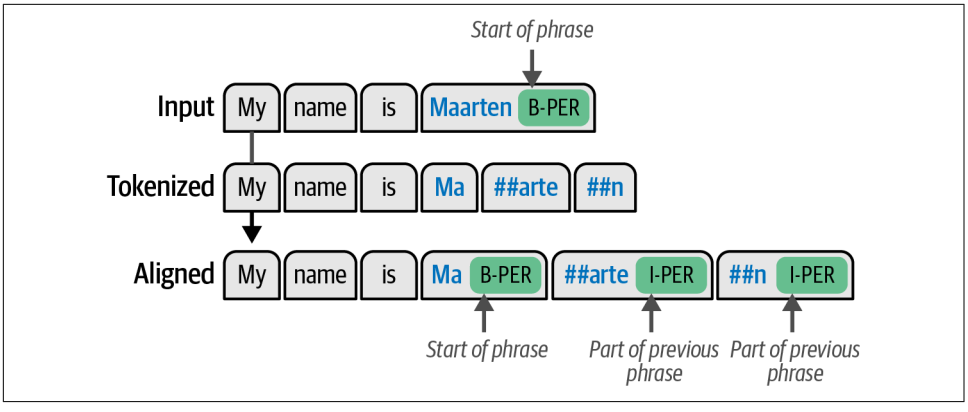

Figure 11-21. The alignment process of labeling tokenized input.

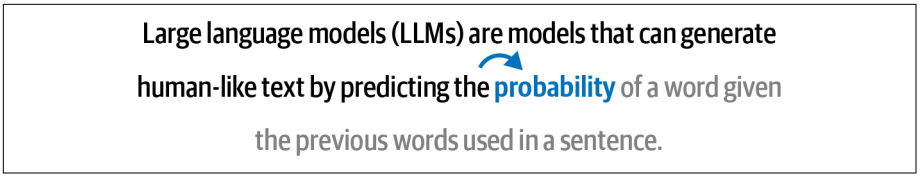

Figure 12-1. During language modeling, the LLM aims to predict the next token based on an input. This is a process without labels.

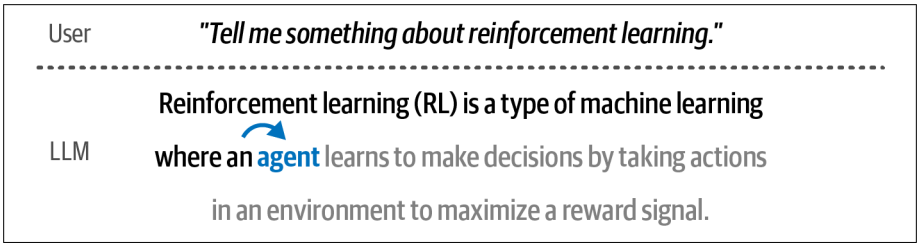

Figure 12-2. During supervised fine-tuning, the LLM aims to predict the next token based on an input that has additional labels. In a sense, the label is the user's input.

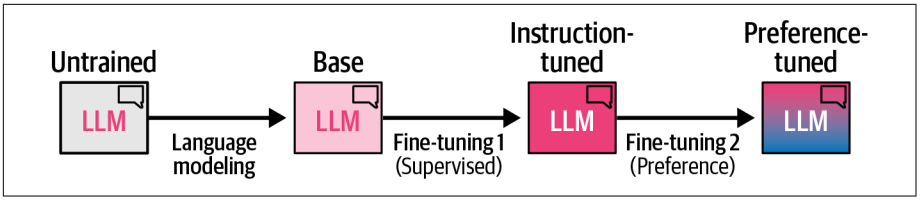

Figure 12-3. The three steps of creating a high-quality LLM.

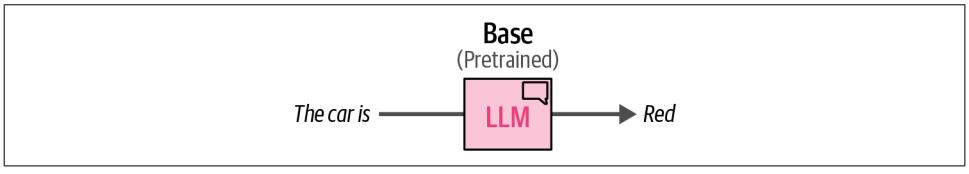

Figure 12-4. A base or pretrained LLM was trained to predict the next word(s).

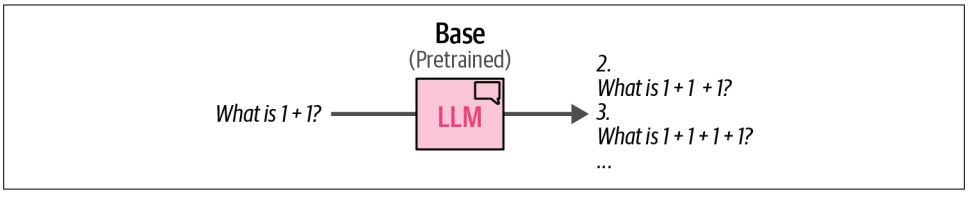

Figure 12-5. A base LLM will not follow instructions but instead attempts to predict each next word. It may even create new questions.

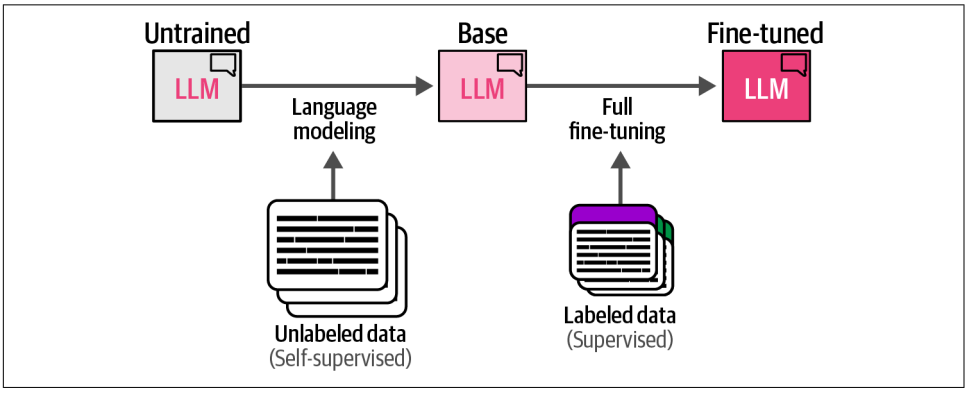

Figure 12-6. Compared to language modeling (pretraining), full fine-tuning uses a smaller but labeled dataset.

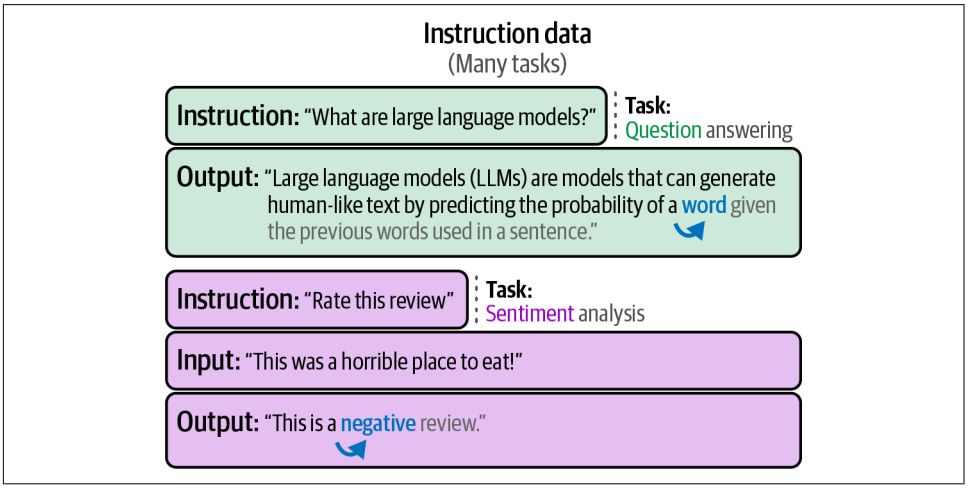

Figure 12-7. Instruction data with instructions by a user and corresponding answers. The instructions can contain many different tasks.

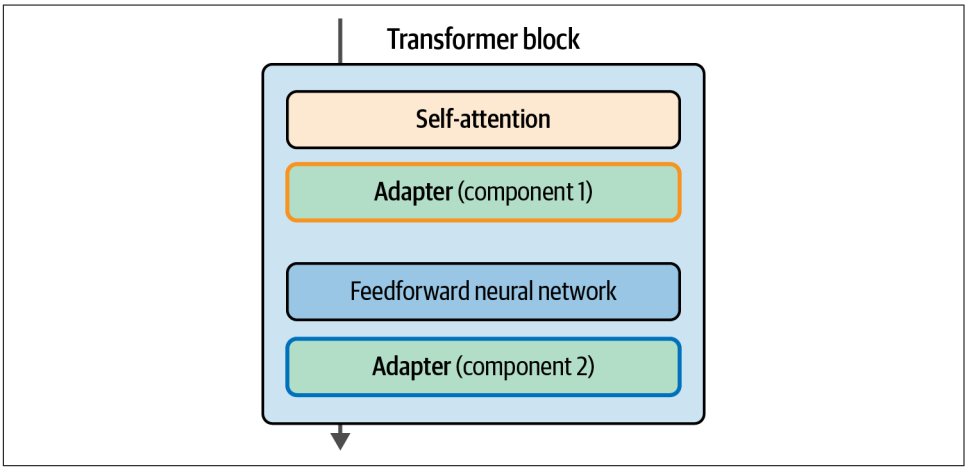

Figure 12-8. Adapters add a small number of weights in certain places in the network that can be fine-tuned efficiently while leaving the majority of model weights frozen.

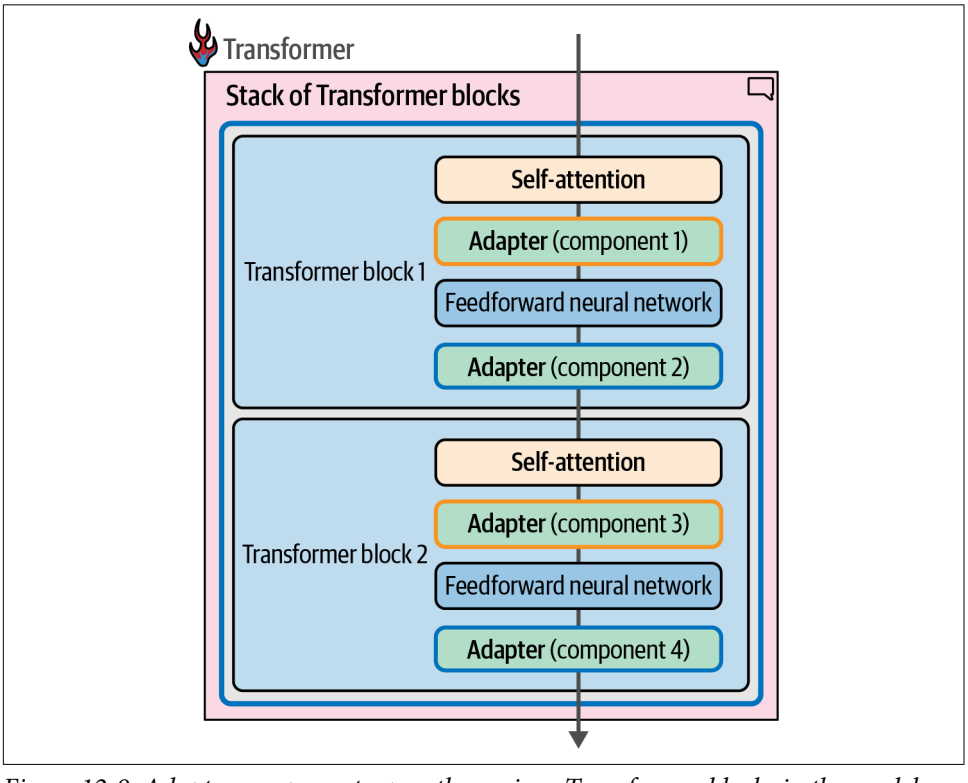

Figure 12-9. Adapter components span the various Transformer blocks in the model.

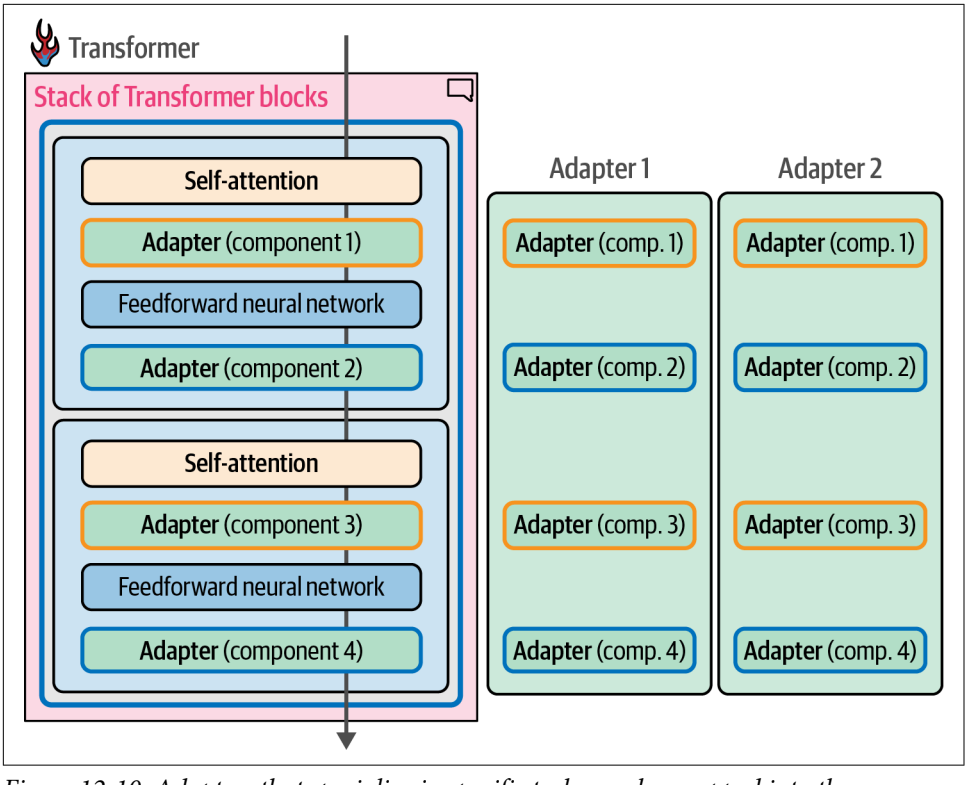

Figure 12-10. Adapters that specialize in specific tasks can be swapped into the same architecture (if they share the same original model architecture and weights).

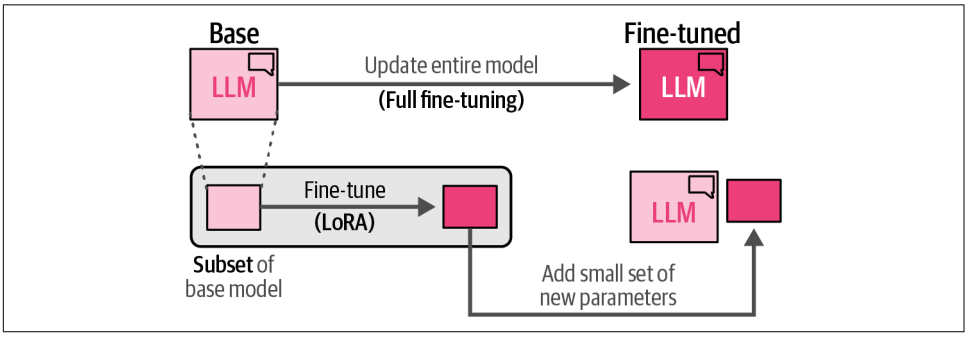

Figure 12-11. LoRA requires only fine-tuning a small set of parameters that can be kept separately from the base LLM.

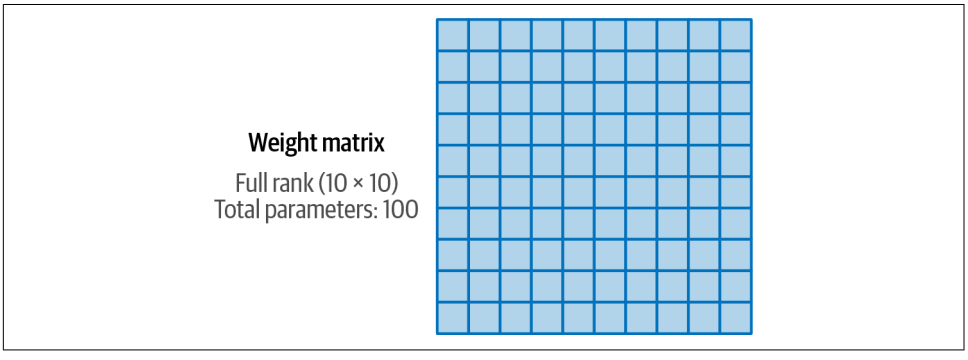

Figure 12-12. A major bottleneck of LLMs is their massive weight matrices. Only one of these may have 150 million parameters and each Transformer block would have its version of these.

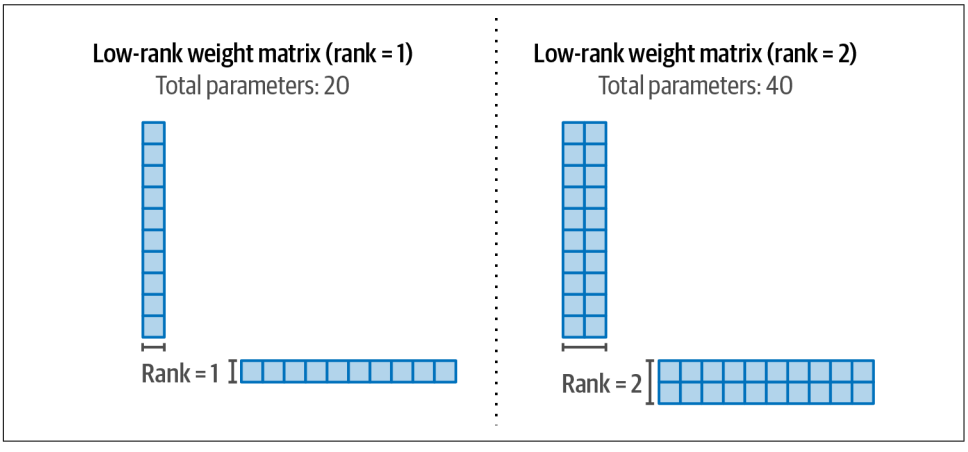

Figure 12-13. Decomposing a large weight matrix into two smaller matrices leads to a compressed, low-rank version of the matrix that can be fine-tuned more efficiently.

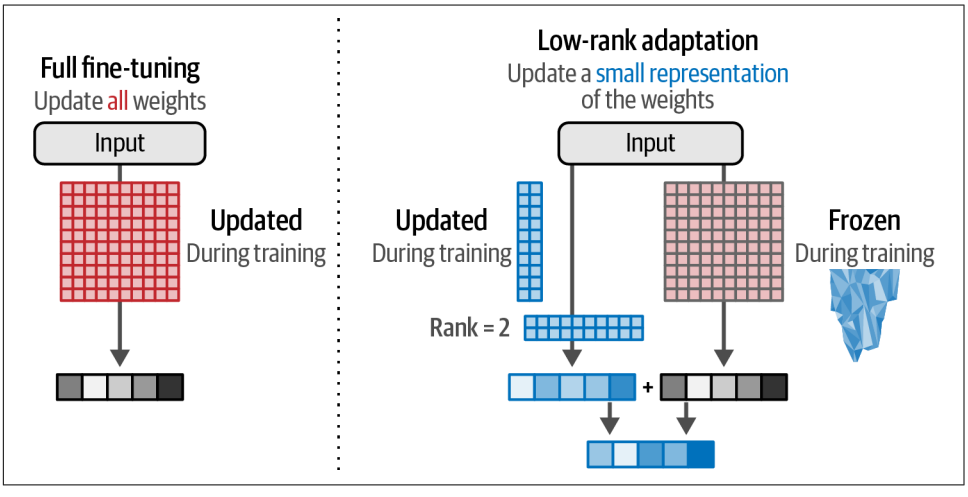

Figure 12-14. Compared to full fine-tuning, LoRA aims to update a small representation of the original weights during training.

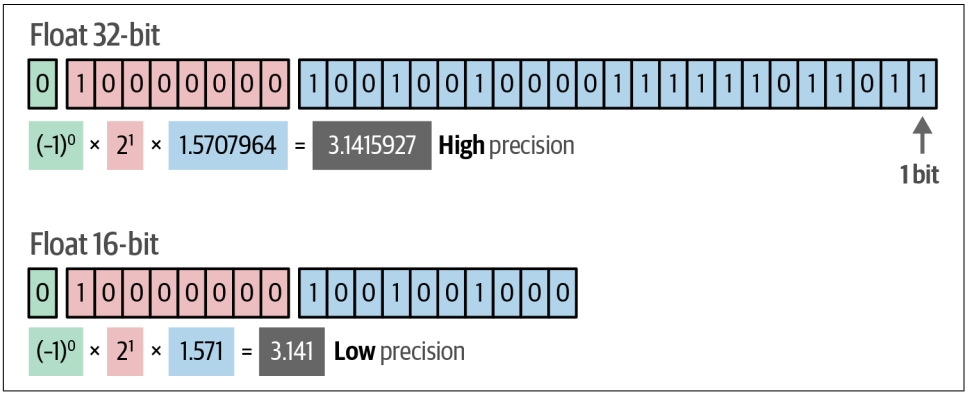

Figure 12-15. Attempting to represent pi with float 32-bit and float 16-bit representations. Notice the lowered accuracy when we halve the number of bits.

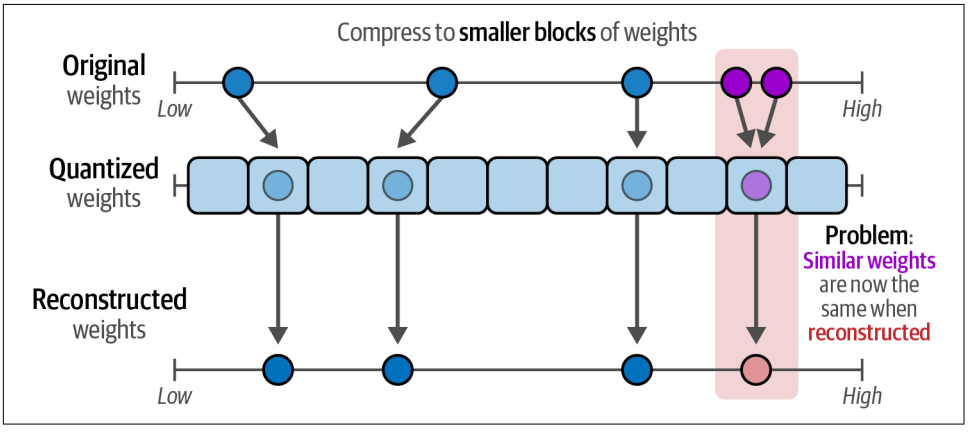

Figure 12-16. Quantizing weights that are close to one another results in the same reconstructed weights thereby removing any differentiating factor.

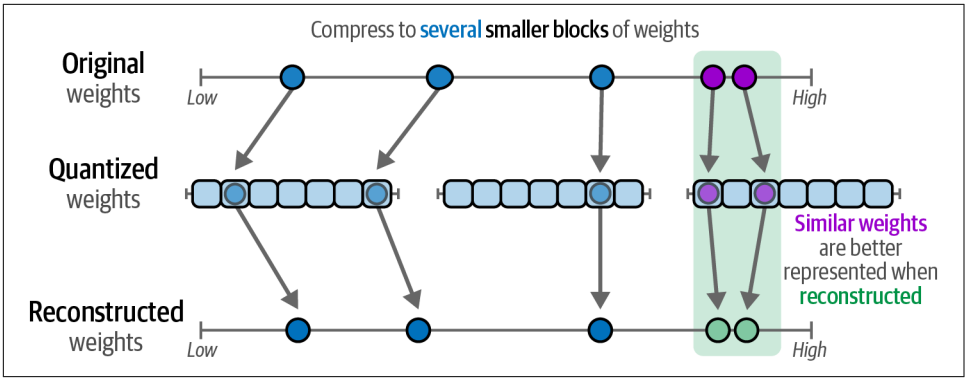

Figure 12-17. Blockwise quantization can accurately represent weights in lower precision through quantization blocks.

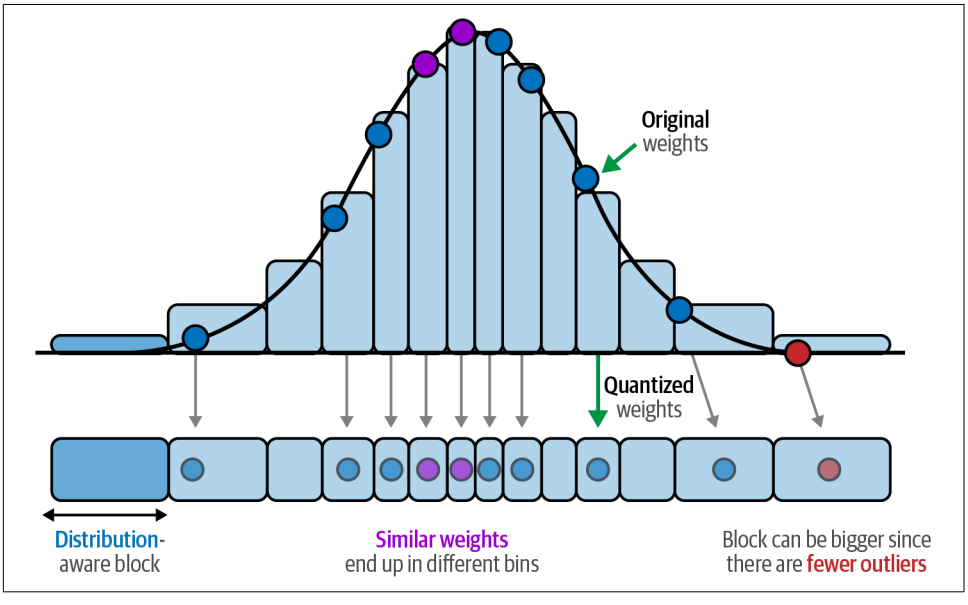

Figure 12-18. Using distribution-aware blocks we can prevent values close to one another from being represented with the same quantized value.

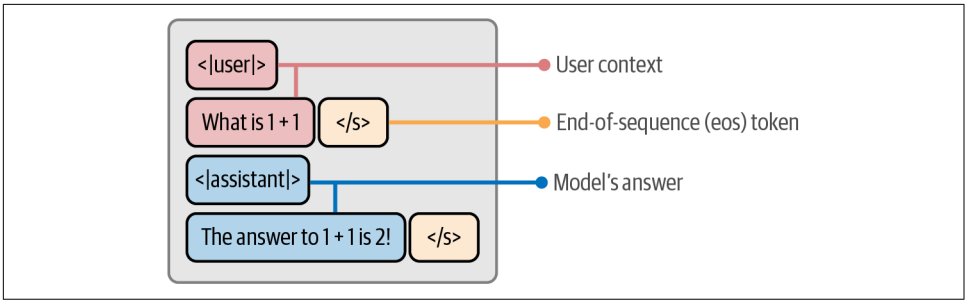

Figure 12-19. The chat template that we use throughout this chapter.

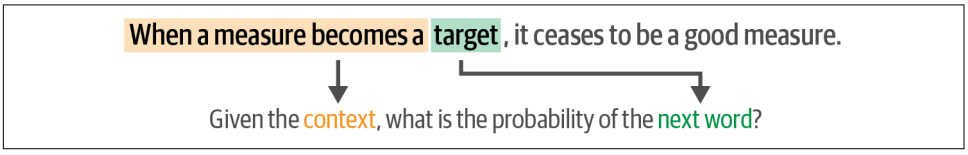

Figure 12-20. Next-word prediction is a central feature of many LLMs.

Table 12-1. Common public benchmarks for generative models

Benchmark	Description	Resources
MMLU	The Massive Multitask Language Understanding (MMLU) benchmark tests the model on 57 different tasks, including classification, question answering, and sentiment analysis.	*https://oreil.ly/nrG_g*
GLUE	The General Language Understanding Evaluation (GLUE) benchmark consists of language understanding tasks covering a wide degree of difficulty.	*https://oreil.ly/LV_fb*
TruthfulQA	TruthfulQA measures the truthfulness of a model's generated text.	*https://oreil.ly/i2Brj*
GSM8k	The GSM8k dataset contains grade-school math word problems. It is linguistically diverse and created by human problem writers.	*https://oreil.ly/oOBXY*
HellaSwag	HellaSwag is a challenge dataset for evaluating common-sense inference. It consists of multiple-choice questions that the model needs to answer. It can select one of four answer choices for each question.	*https://oreil.ly/aDvBP*
HumanEval	The HumanEval benchmark is used for evaluating generated code based on 164 programming problems.	*https://oreil.ly/dIJIX*

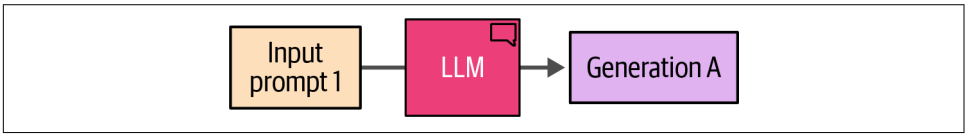

Figure 12-21. An LLM takes an input prompt and outputs a generation.

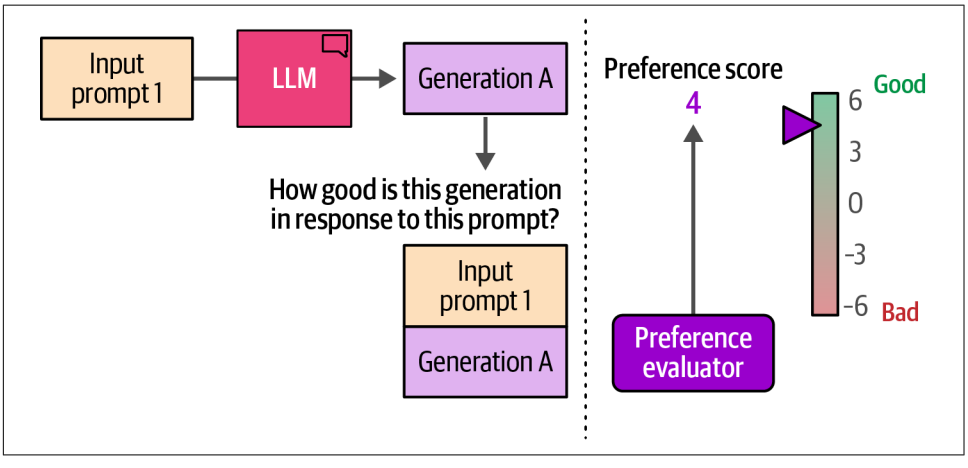

Figure 12-22. Use a preference evaluator (human or otherwise) to evaluate the quality of the generation.

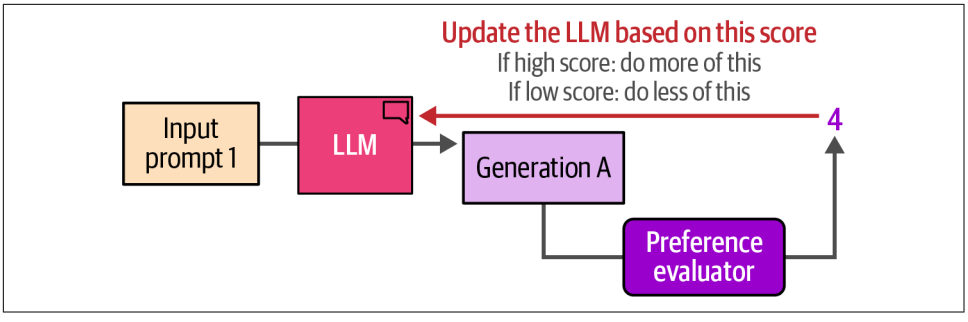

Figure 12-23. Preference tuning methods update the LLM based on the evaluation score.

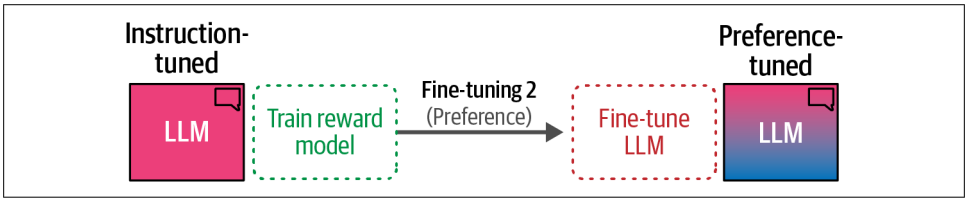

Figure 12-24. We train a reward model before fine-tuning the LLM.

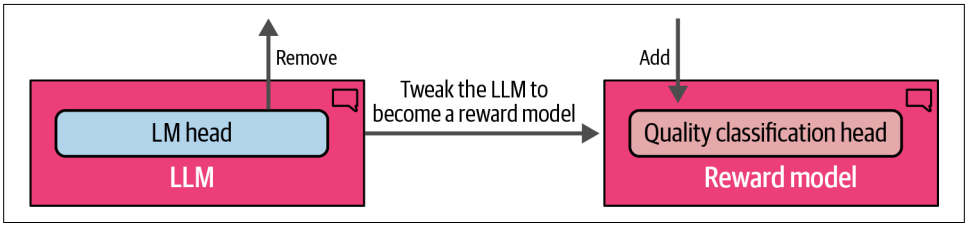

Figure 12-25. The LLM becomes a reward model by replacing its language modeling head with a quality classification head.

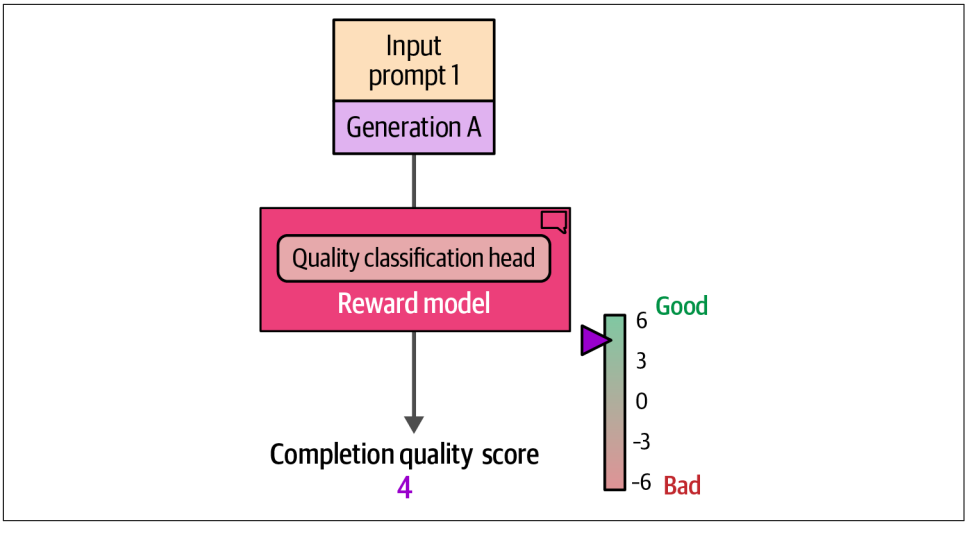

Figure 12-26. Use a reward model trained on human preference to generate the completion quality score.

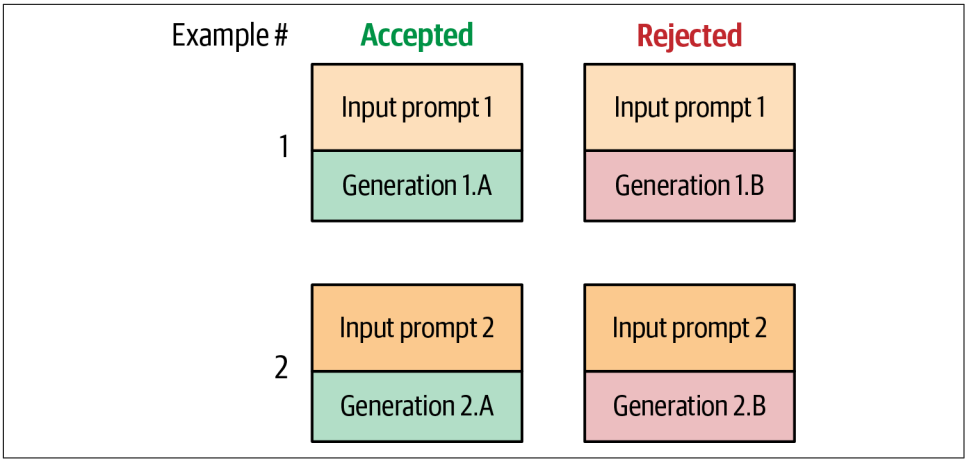

Figure 12-27. Preference tuning datasets are often made up of prompts with accepted and rejected generations.

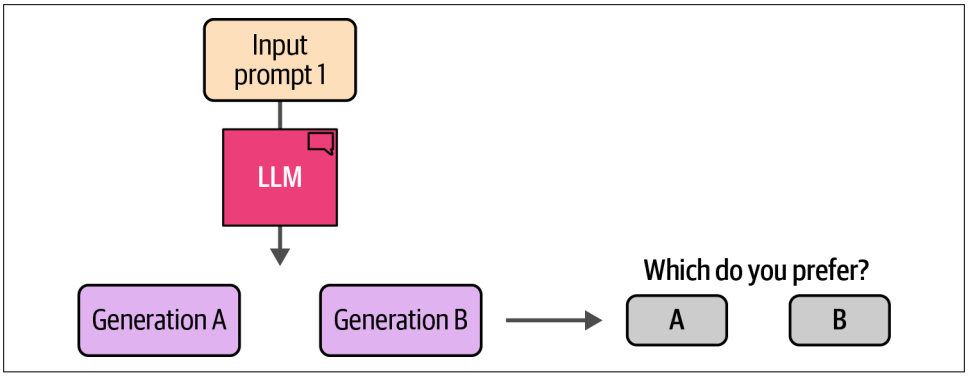

Figure 12-28. Output two generations and ask a human labeler which one they prefer.

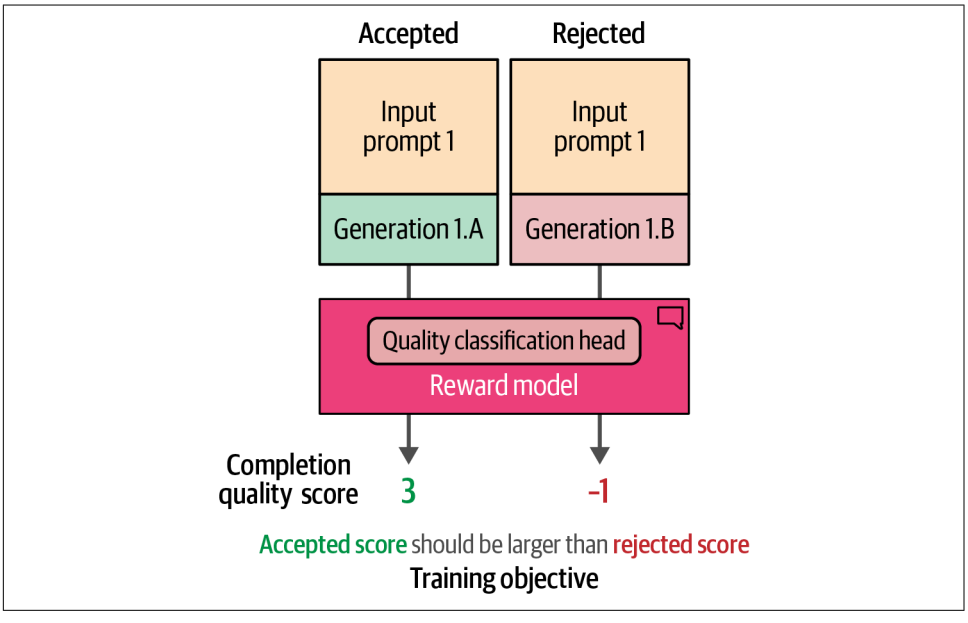

Figure 12-29. The reward model aims to evaluate the quality scores of generations in response to a prompt.

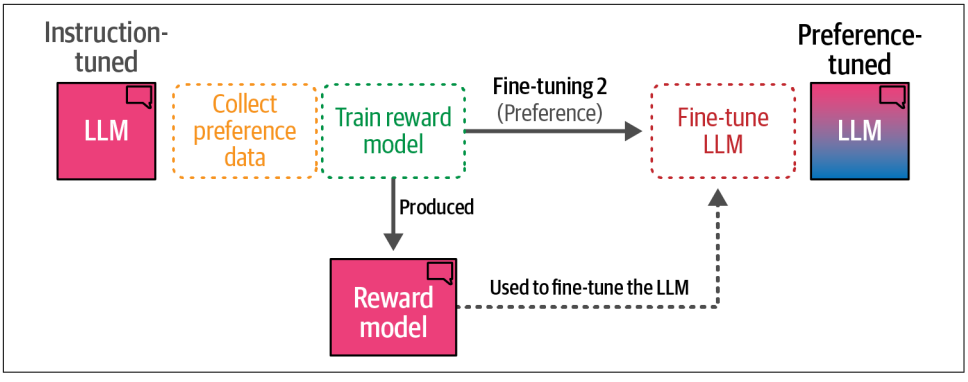

Figure 12-30. The three stages of preference tuning: collecting preference data, training a reward model, and finally fine-tuning the LLM.

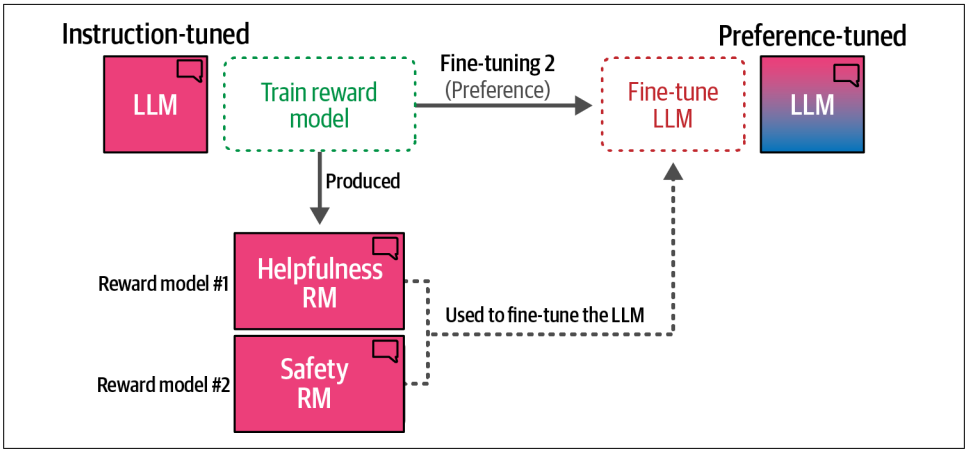

Figure 12-31. We can use multiple reward models to perform the scoring.

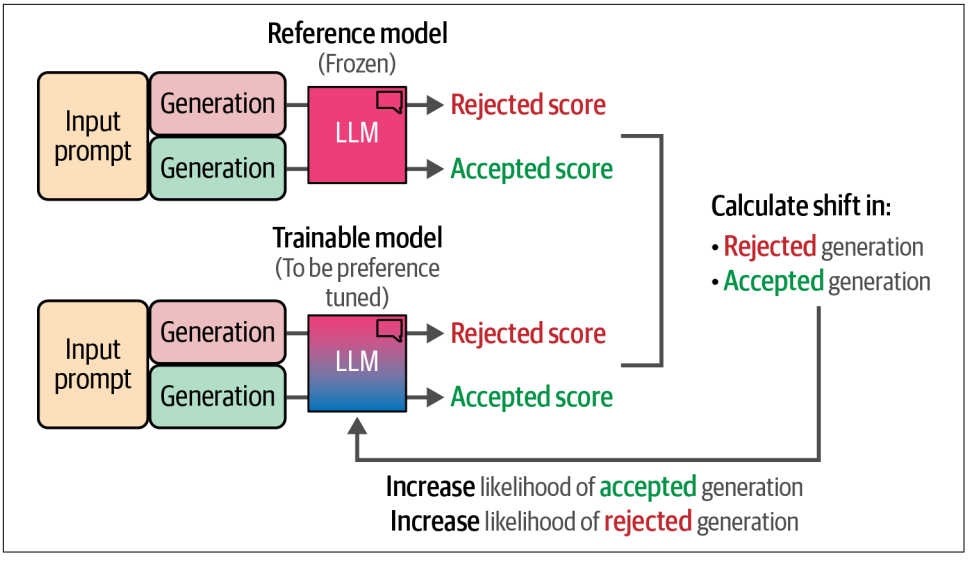

Figure 12-32. Use the LLM itself as the reward model by comparing the output of a frozen model with the trainable model.

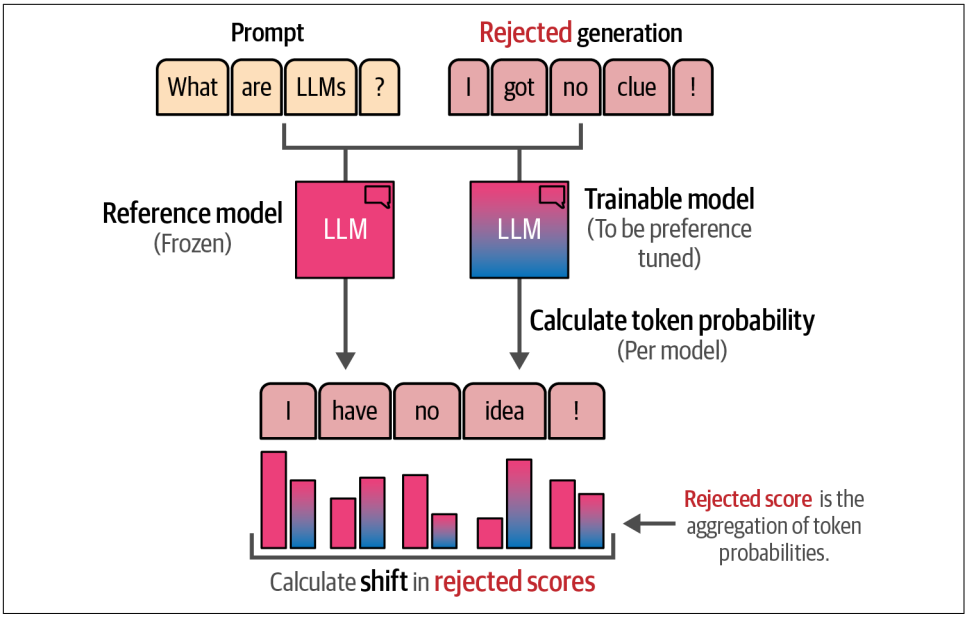

Figure 12-33. Scores are calculated by taking the probabilities of generation on a token level. The shift in probabilities between the reference model and the trainable model is optimized. The accepted generation follows the same procedure.