

Chapter 1. An Introduction to Large Language Models

Humanity is at an inflection point. From 2012 onwards, developments in building AI systems (using deep neural networks) accelerated so that by the end of the decade, they yielded the first software system able to write articles indiscernible from those written by humans. This system was an AI model called Generative Pre-trained Transformer 2, or GPT-2. 2022 marked the release of ChatGPT, which demonstrated how profoundly this technology was poised to revolutionize how we interact with technology and information. Reaching one million active users in five days and then one hundred million active users in two months, the new breed of AI models started out as human-like chatbots but quickly evolved into a monumental shift in our approach to common tasks, like translation, text generation, summarization, and more. It became an invaluable tool for programmers, educators, and researchers.

The success of ChatGPT was unprecedented and popularized more research into the technology behind it, namely large language models (LLMs). Both proprietary and public models were being released at a steady pace, closing in on, and eventually catching up to the performance of ChatGPT. It is not an exaggeration to state that almost all attention was on LLMs.

As a result, 2023 will always be known, at least to us, as the year that drastically changed our field, Language Artificial Intelligence (Language AI), a field characterized by the development of systems capable of understanding and generating human language.

However, LLMs have been around for a while now and smaller models are still relevant to this day. LLMs are much more than just a single model and there are many other techniques and models in the field of language AI that are worth exploring.

In this book, we aim to give readers a solid understanding of the fundamentals of both LLMs and the field of Language AI in general. This chapter serves as the scaffolding for the rest of the book and will introduce concepts and terms that we will use throughout the chapters.

But mostly, we intend to answer the following questions in this chapter:

- What is Language AI?
- What are large language models?
- What are the common use cases and applications of large language models?
- How can we use large language models ourselves?

What Is Language AI?

The term *artificial intelligence* (AI) is often used to describe computer systems dedicated to performing tasks close to human intelligence, such as speech recognition, language translation, and visual perception. It is the intelligence of software as opposed to the intelligence of humans.

Here is a more formal definition by one of the founders of the artificial intelligence discipline:

[Artificial intelligence is] the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.

—John McCarthy, 2007¹

Due to the ever-evolving nature of AI, the term has been used to describe a wide variety of systems, some of which might not truly embody intelligent behavior. For instance, characters in computer games (NPCs [nonplayable characters]) have often been referred to as AI even though many are nothing more than *if-else* statements.

Language AI refers to a subfield of AI that focuses on developing technologies capable of understanding, processing, and generating human language. The term *Language AI* can often be used interchangeably with *natural language processing* (NLP) with the continued success of machine learning methods in tackling language processing problems.

We use the term *Language AI* to encompass technologies that technically might not be LLMs but still have a significant impact on the field, like how retrieval systems can give LLMs superpowers (see [Chapter 8](#)).

Throughout this book, we want to focus on the models that have had a major role in shaping the field of Language AI. This means exploring more than just LLMs in isolation. That, however, brings us to the question: what are large language models? To begin answering this question in this chapter, let's first explore the history of Language AI.

A Recent History of Language AI

The history of Language AI encompasses many developments and models aiming to represent and generate language, as illustrated in [Figure 1-1](#).

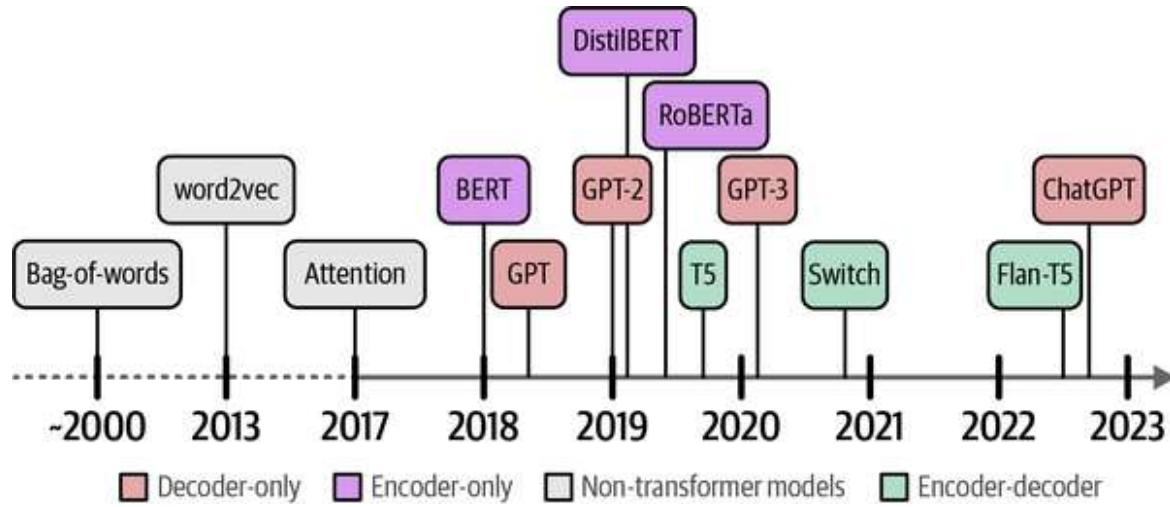


Figure 1-1. A peek into the history of Language AI.

Language, however, is a tricky concept for computers. Text is unstructured in nature and loses its meaning when represented by zeros and ones (individual characters). As a result, throughout the history of Language AI,

there has been a large focus on representing language in a structured manner so that it can more easily be used by computers. Examples of these Language AI tasks are provided in [Figure 1-2](#).

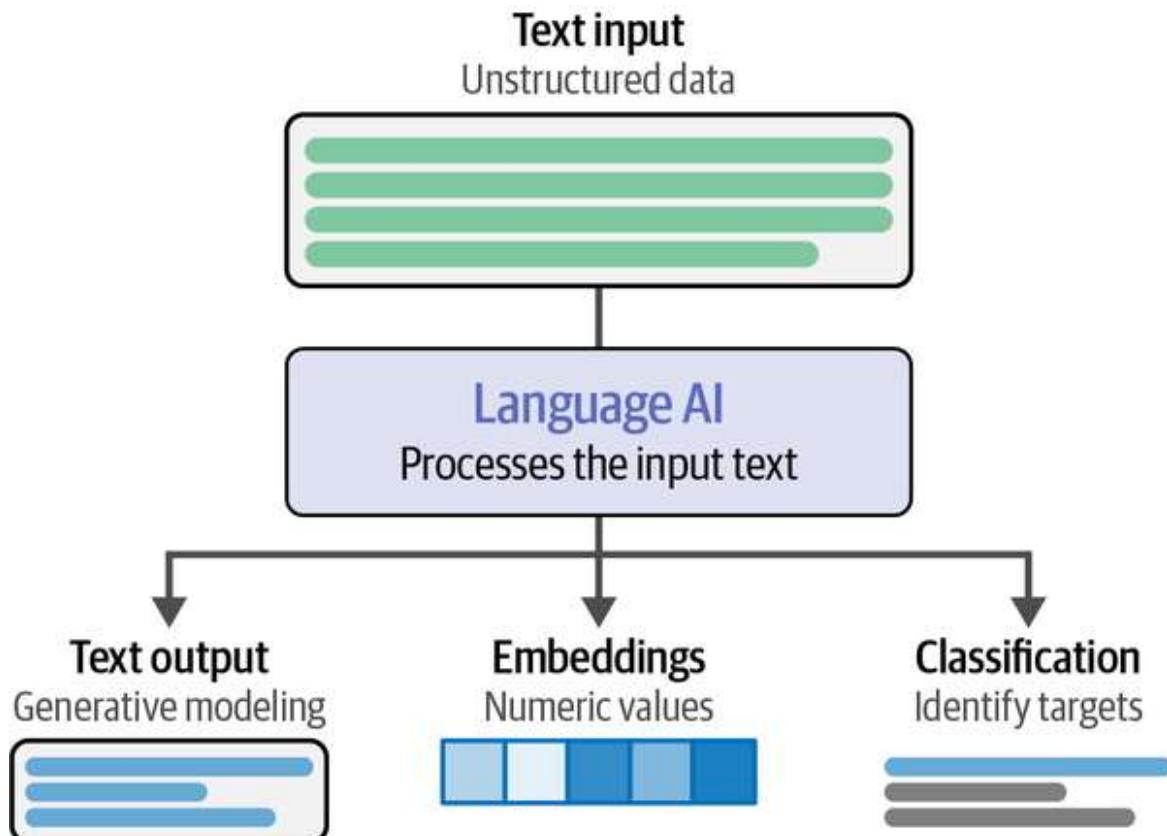


Figure 1-2. Language AI is capable of many tasks by processing textual input.

Representing Language as a Bag-of-Words

Our history of Language AI starts with a technique called bag-of-words, a method for representing unstructured text.² It was first mentioned around the 1950s but became popular around the 2000s.

Bag-of-words works as follows: let's assume that we have two sentences for which we want to create numerical representations. The first step of the bag-of-words model is *tokenization*, the process of splitting up the sentences into individual words or subwords (*tokens*), as illustrated in [Figure 1-3](#).

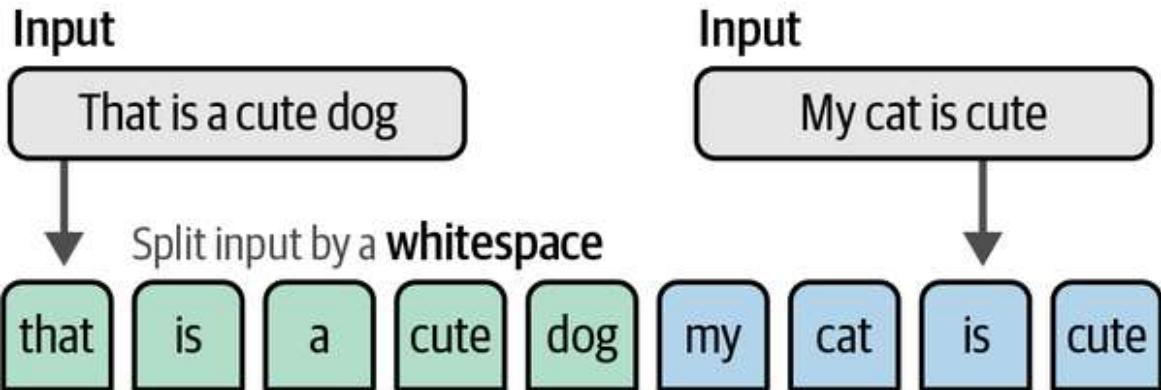


Figure 1-3. Each sentence is split into words (tokens) by splitting on a whitespace.

The most common method for tokenization is by splitting on a whitespace to create individual words. However, this has its disadvantages as some languages, like Mandarin, do not have whitespaces around individual words. In the next chapter, we will go in depth about tokenization and how that technique influences language models. As illustrated in [Figure 1-4](#), after tokenization, we combine all unique words from each sentence to create a vocabulary that we can use to represent the sentences.

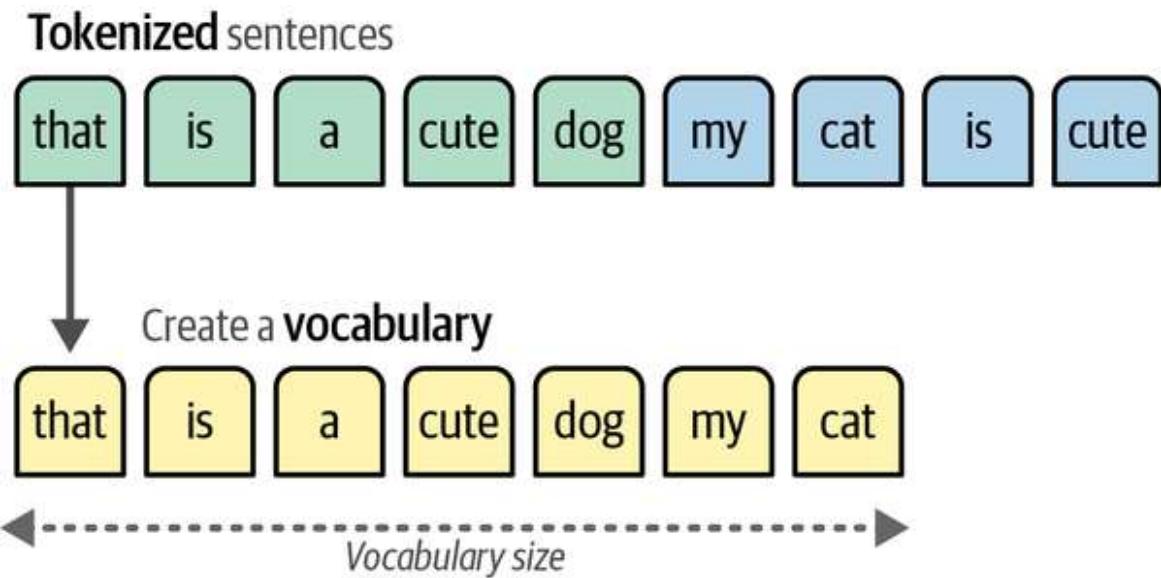


Figure 1-4. A vocabulary is created by retaining all unique words across both sentences.

Using our vocabulary, we simply count how often a word in each sentence appears, quite literally creating a bag of words. As a result, a bag-of-words model aims to create representations of text in the form of numbers, also

called vectors or vector representations, observed in [Figure 1-5](#). Throughout the book, we refer to these kinds of models as *representation models*.

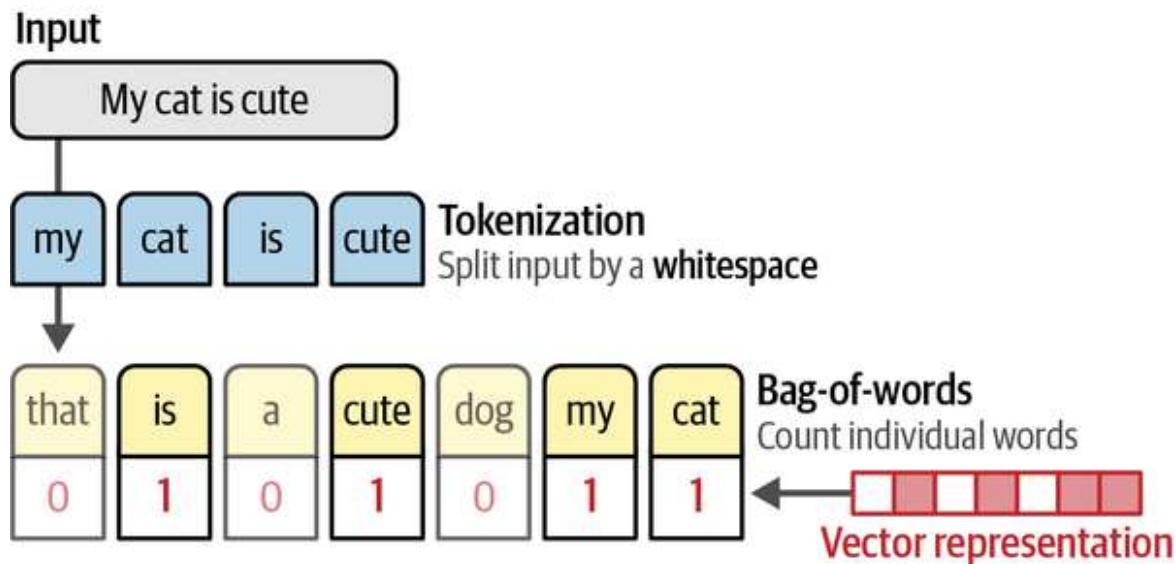


Figure 1-5. A bag-of-words is created by counting individual words. These values are referred to as vector representations.

Although bag-of-words is a classic method, it is by no means completely obsolete. In [Chapter 5](#), we will explore how it can still be used to complement more recent language models.

Better Representations with Dense Vector Embeddings

Bag-of-words, although an elegant approach, has a flaw. It considers language to be nothing more than an almost literal bag of words and ignores the semantic nature, or meaning, of text.

Released in 2013, word2vec was one of the first successful attempts at capturing the meaning of text in *embeddings*.³ Embeddings are vector representations of data that attempt to capture its meaning. To do so, word2vec learns semantic representations of words by training on vast amounts of textual data, like the entirety of Wikipedia.

To generate these semantic representations, word2vec leverages *neural networks*. These networks consist of interconnected layers of nodes that process information. As illustrated in [Figure 1-6](#), neural networks can have

many layers where each connection has a certain weight depending on the input. These weights are often referred to as the *parameters* of the model.

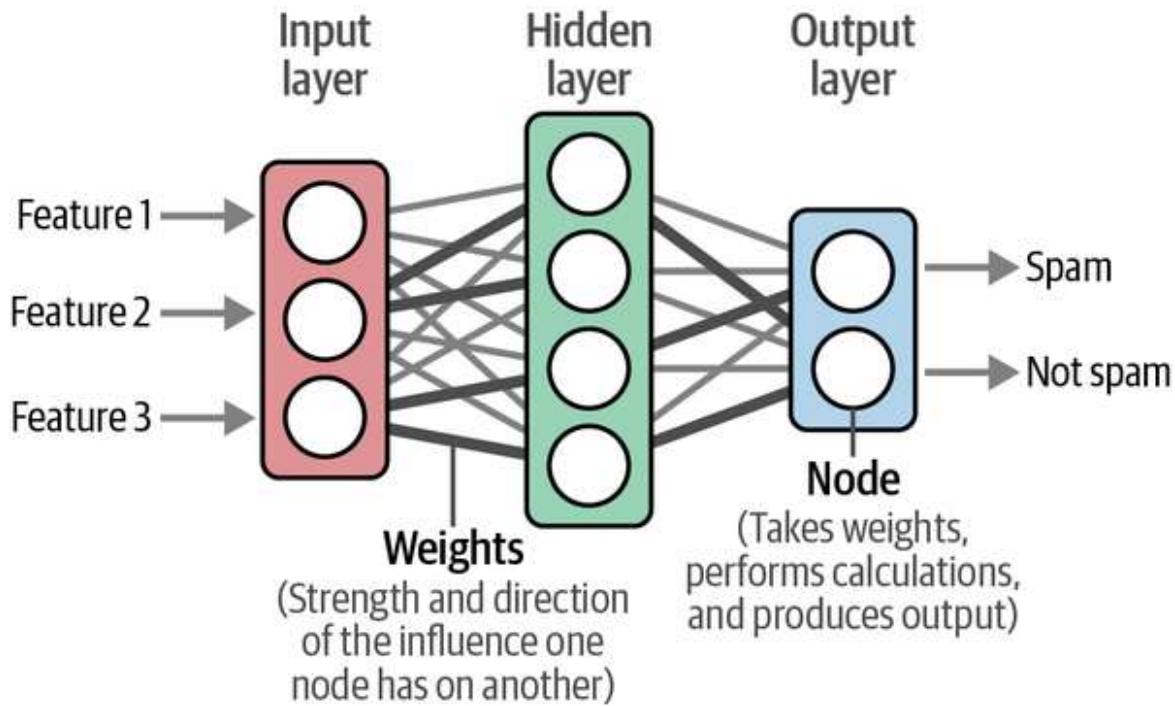


Figure 1-6. A neural network consists of interconnected layers of nodes where each connection is a linear equation.

Using these neural networks, word2vec generates word embeddings by looking at which other words they tend to appear next to in a given sentence. We start by assigning every word in our vocabulary with a vector embedding, say of 50 values for each word initialized with random values. Then in every training step, as illustrated in [Figure 1-7](#), we take pairs of words from the training data and a model attempts to predict whether or not they are likely to be neighbors in a sentence.

During this training process, word2vec learns the relationship between words and distills that information into the embedding. If the two words tend to have the same neighbors, their embeddings will be closer to one another and vice versa. In [Chapter 2](#), we will look closer at word2vec's training procedure.

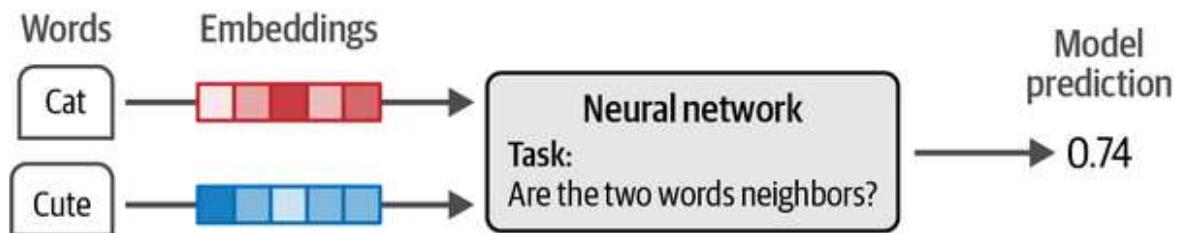


Figure 1-7. A neural network is trained to predict if two words are neighbors. During this process, the embeddings are updated to be in line with the ground truth.

The resulting embeddings capture the meaning of words but what exactly does that mean? To illustrate this phenomenon, let's somewhat oversimplify and imagine we have embeddings of several words, namely “apple” and “baby.” Embeddings attempt to capture meaning by representing the properties of words. For instance, the word “baby” might score high on the properties “newborn” and “human” while the word “apple” scores low on these properties.

As illustrated in [Figure 1-8](#), embeddings can have many properties to represent the meaning of a word. Since the size of embeddings is fixed, their properties are chosen to create a mental representation of the word.

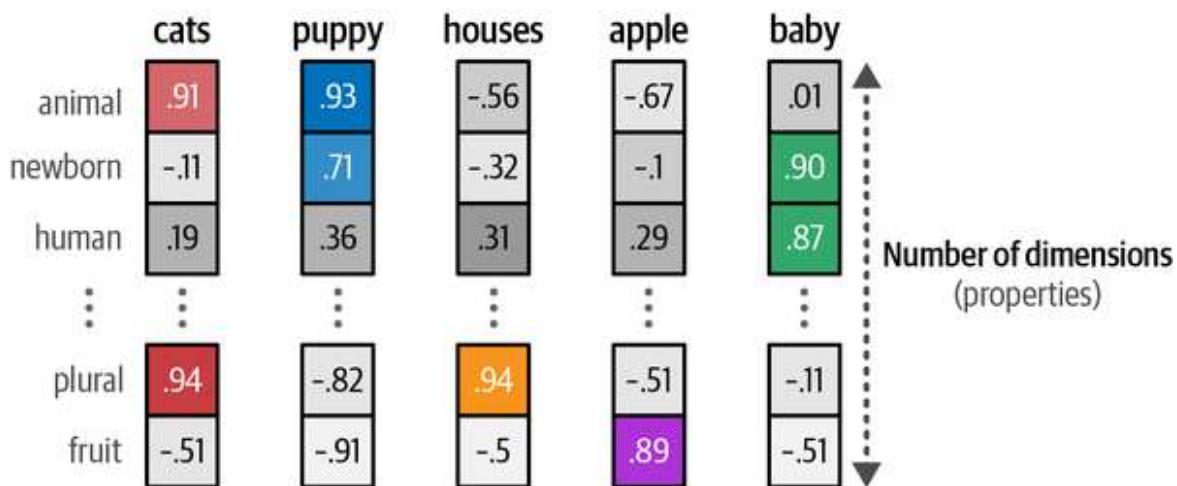


Figure 1-8. The values of embeddings represent properties that are used to represent words. We may oversimplify by imagining that dimensions represent concepts (which they don't), but it helps express the idea.

In practice, these properties are often quite obscure and seldom relate to a single entity or humanly identifiable concept. However, together, these

properties make sense to a computer and serve as a good way to translate human language into computer language.

Embeddings are tremendously helpful as they allow us to measure the semantic similarity between two words. Using various distance metrics, we can judge how close one word is to another. As illustrated in [Figure 1-9](#), if we were to compress these embeddings into a two-dimensional representation, you would notice that words with similar meaning tend to be closer. In [Chapter 5](#), we will explore how to compress these embeddings into n -dimensional space.

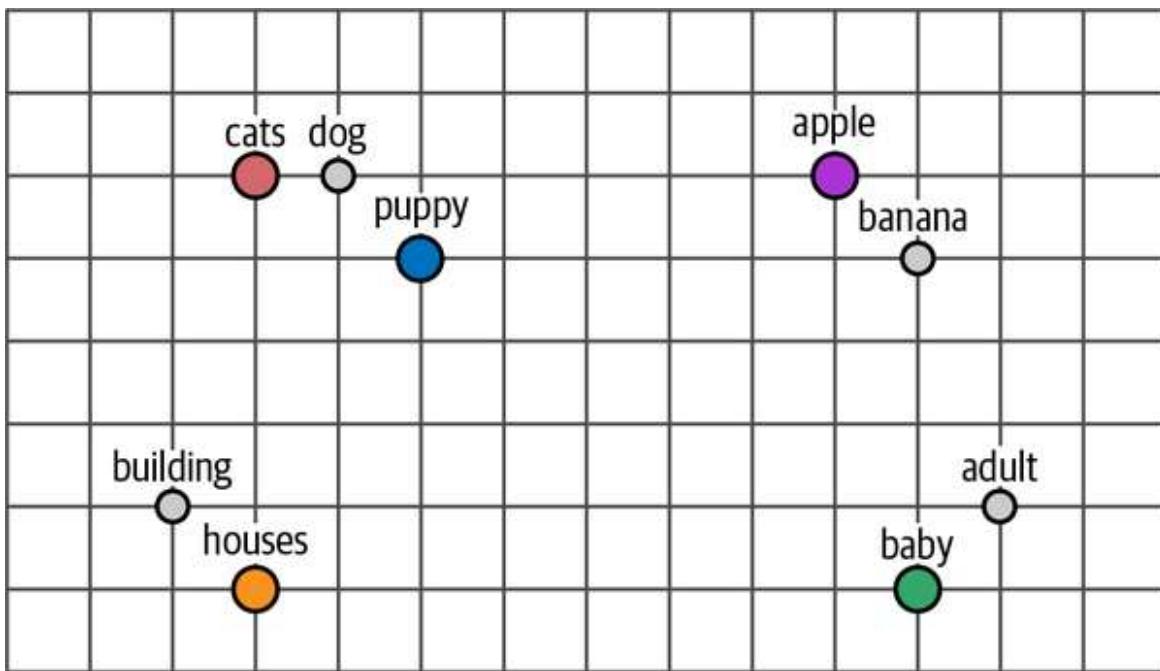


Figure 1-9. Embeddings of words that are similar will be close to each other in dimensional space.

Types of Embeddings

There are many types of embeddings, like word embeddings and sentence embeddings that are used to indicate different levels of abstractions (word versus sentence), as illustrated in [Figure 1-10](#).

Bag-of-words, for instance, creates embeddings at a document level since it represents the entire document. In contrast, word2vec generates embeddings for words only.

Throughout the book, embeddings will take on a central role as they are utilized in many use cases, such as classification (see [Chapter 4](#)), clustering (see [Chapter 5](#)), and semantic search and retrieval-augmented generation (see [Chapter 8](#)). In [Chapter 2](#), we will take our first deep dive into token embeddings.

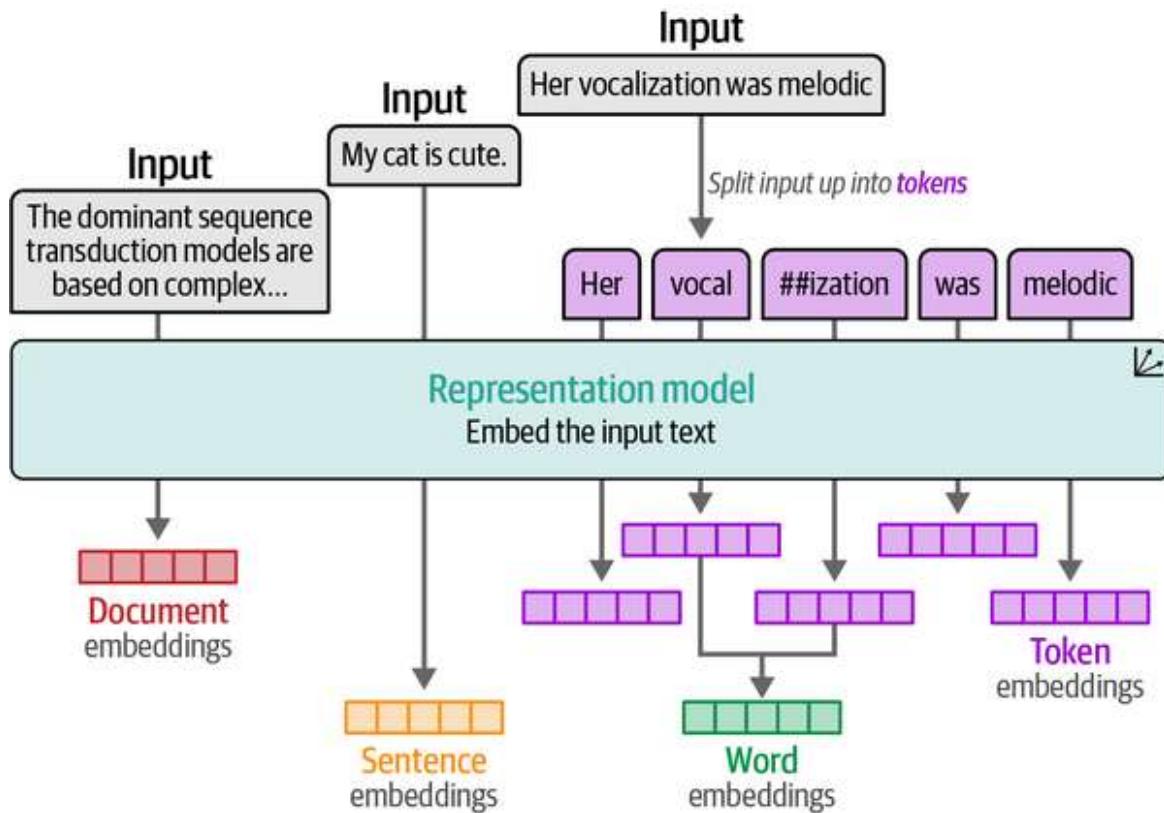


Figure 1-10. Embeddings can be created for different types of input.

Encoding and Decoding Context with Attention

The training process of word2vec creates static, downloadable representations of words. For instance, the word “bank” will always have the same embedding regardless of the context in which it is used. However, “bank” can refer to both a financial bank as well as the bank of a river. Its meaning, and therefore its embeddings, should change depending on the context.

A step in encoding this text was achieved through recurrent neural networks (RNNs). These are variants of neural networks that can model sequences as

an additional input.

To do so, these RNNs are used for two tasks, *encoding* or representing an input sentence and *decoding* or generating an output sentence. [Figure 1-11](#) illustrates this concept by showing how a sentence like “I love llamas” gets translated to the Dutch “Ik hou van lama’s.”

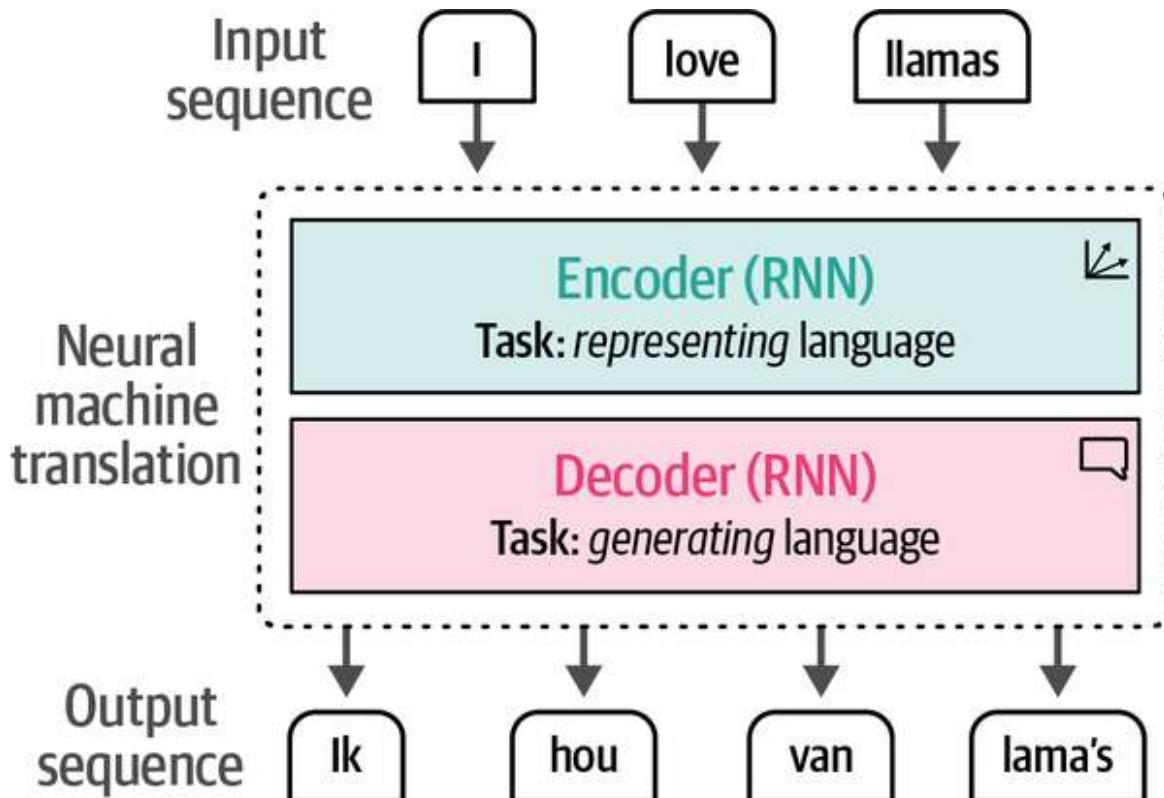


Figure 1-11. Two recurrent neural networks (decoder and encoder) translating an input sequence from English to Dutch.

Each step in this architecture is *autoregressive*. When generating the next word, this architecture needs to consume all previously generated words, as shown in [Figure 1-12](#).

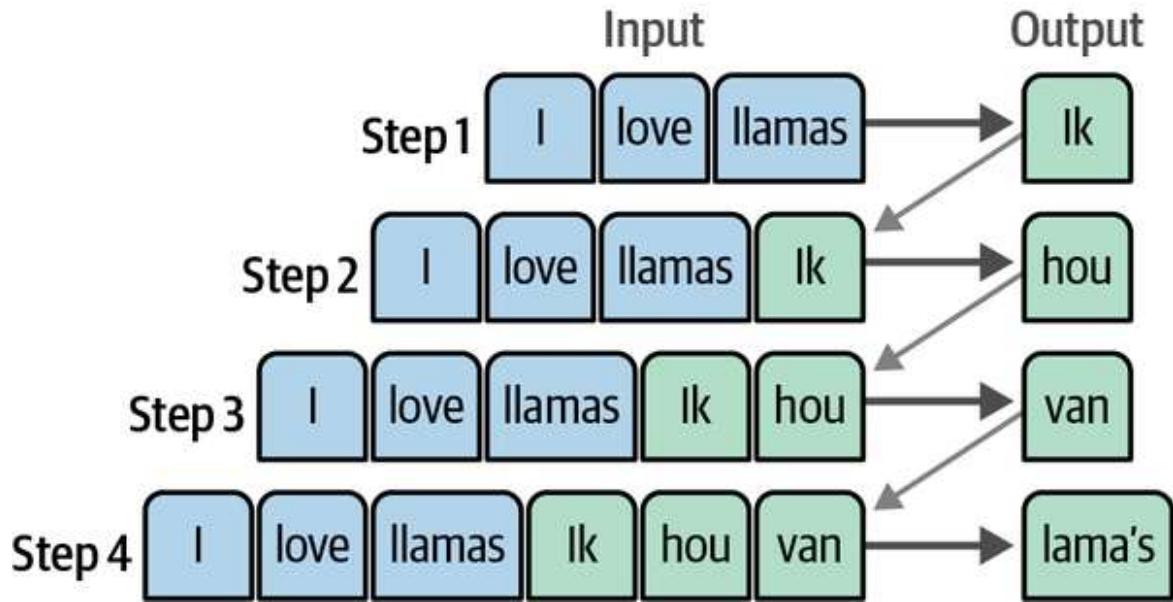


Figure 1-12. Each previous output token is used as input to generate the next token.

The encoding step aims to represent the input as well as possible, generating the context in the form of an embedding, which serves as the input for the decoder. To generate this representation, it takes embeddings as its inputs for words, which means we can use word2vec for the initial representations. In [Figure 1-13](#), we can observe this process. Note how the inputs are processed sequentially, one at a time, as well as the output.

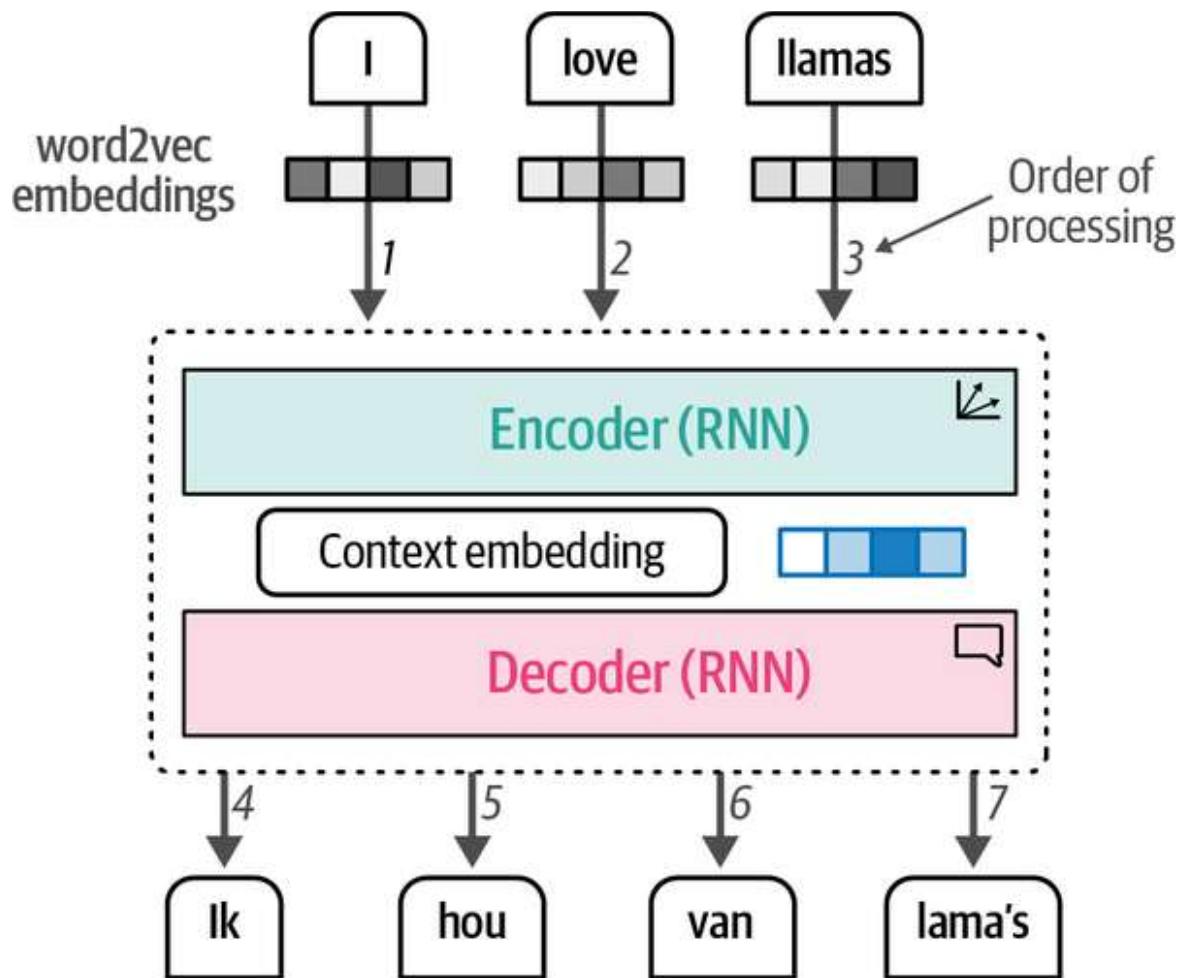


Figure 1-13. Using word2vec embeddings, a context embedding is generated that represents the entire sequence.

This context embedding, however, makes it difficult to deal with longer sentences since it is merely a single embedding representing the entire input. In 2014, a solution called *attention* was introduced that highly improved upon the original architecture.⁴ Attention allows a model to focus on parts of the input sequence that are relevant to one another (“attend” to each other) and amplify their signal, as shown in Figure 1-14. Attention selectively determines which words are most important in a given sentence.

For instance, the output word “lama’s” is Dutch for “llamas,” which is why the attention between both is high. Similarly, the words “lama’s” and “I” have lower attention since they aren’t as related. In Chapter 3, we will go more in depth on the attention mechanism.

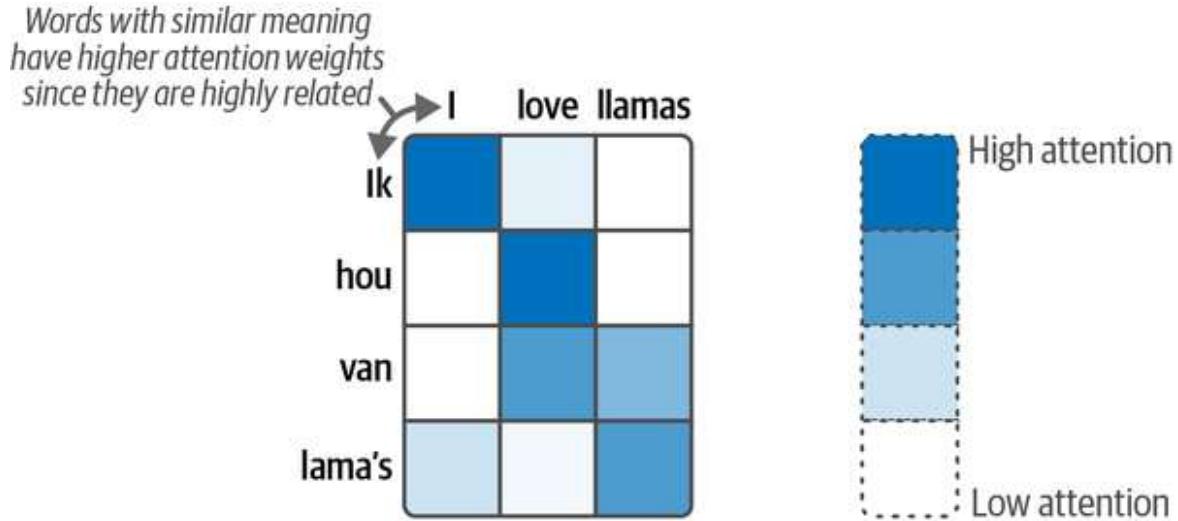


Figure 1-14. Attention allows a model to “attend” to certain parts of sequences that might relate more or less to one another.

By adding these attention mechanisms to the decoder step, the RNN can generate signals for each input word in the sequence related to the potential output. Instead of passing only a context embedding to the decoder, the hidden states of all input words are passed. This process is demonstrated in Figure 1-15.

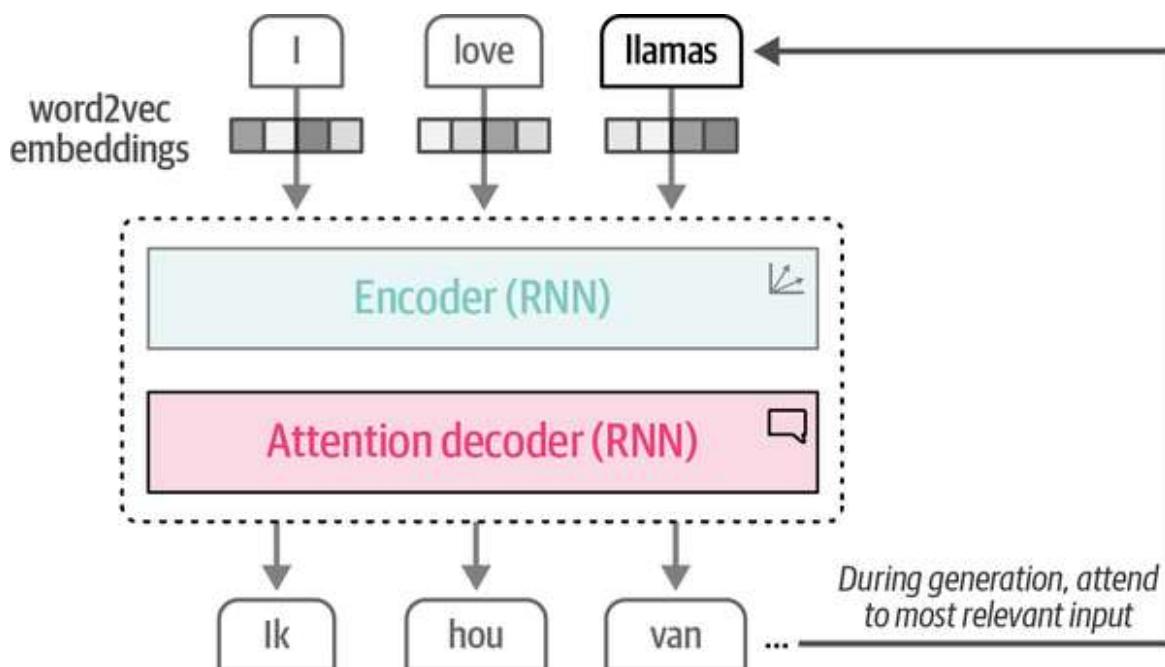


Figure 1-15. After generating the words “Ik,” “hou,” and “van,” the attention mechanism of the decoder enables it to focus on the word “llamas” before it generates the Dutch translation (“lama’s”).

As a result, during the generation of “Ik hou van lama’s,” the RNN keeps track of the words it mostly attends to perform the translation. Compared to word2vec, this architecture allows for representing the sequential nature of text and the context in which it appears by “attending” to the entire sentence. This sequential nature, however, precludes parallelization during training of the model.

Attention Is All You Need

The true power of attention, and what drives the amazing abilities of large language models, was first explored in the well-known “[Attention is all you need](#)” paper released in 2017.⁵ The authors proposed a network architecture called the *Transformer*, which was solely based on the attention mechanism and removed the recurrence network that we saw previously. Compared to the recurrence network, the Transformer could be trained in parallel, which tremendously sped up training.

In the Transformer, encoding and decoder components are stacked on top of each other, as illustrated in [Figure 1-16](#). This architecture remains autoregressive, needing to consume each generated word before creating a new word.

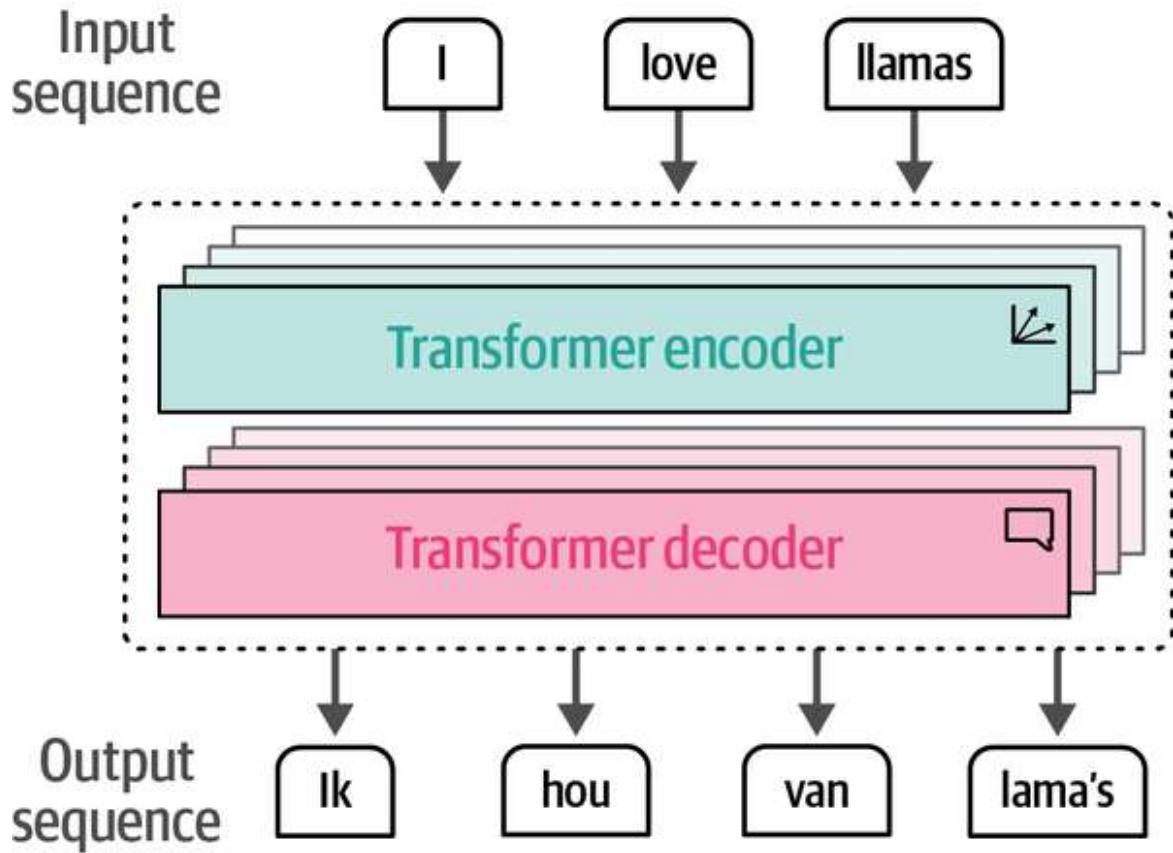


Figure 1-16. The Transformer is a combination of stacked encoder and decoder blocks where the input flows through each encoder and decoder.

Now, both the encoder and decoder blocks would revolve around attention instead of leveraging an RNN with attention features. The encoder block in the Transformer consists of two parts, *self-attention* and a *feedforward neural network*, which are shown in Figure 1-17.

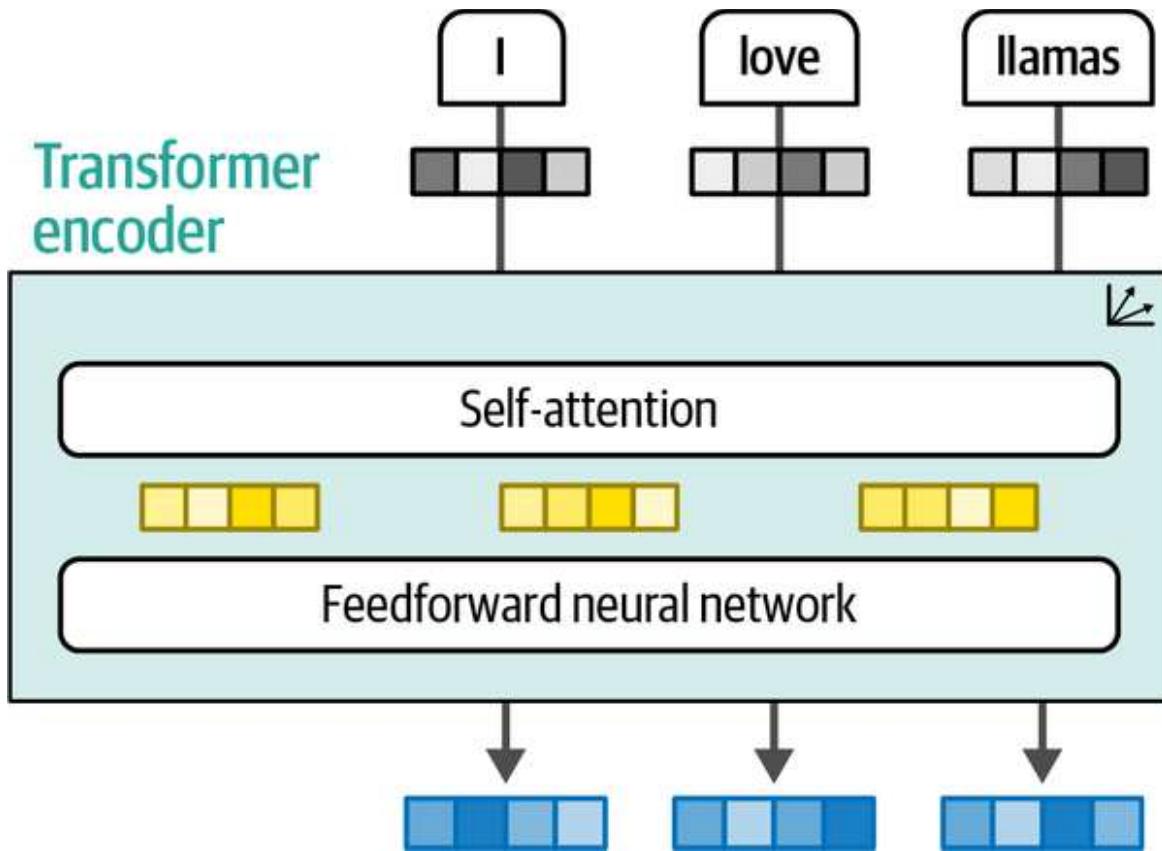


Figure 1-17. An encoder block revolves around self-attention to generate intermediate representations.

Compared to previous methods of attention, self-attention can attend to different positions within a single sequence, thereby more easily and accurately representing the input sequence as illustrated in [Figure 1-18](#). Instead of processing one token at a time, it can be used to look at the entire sequence in one go.

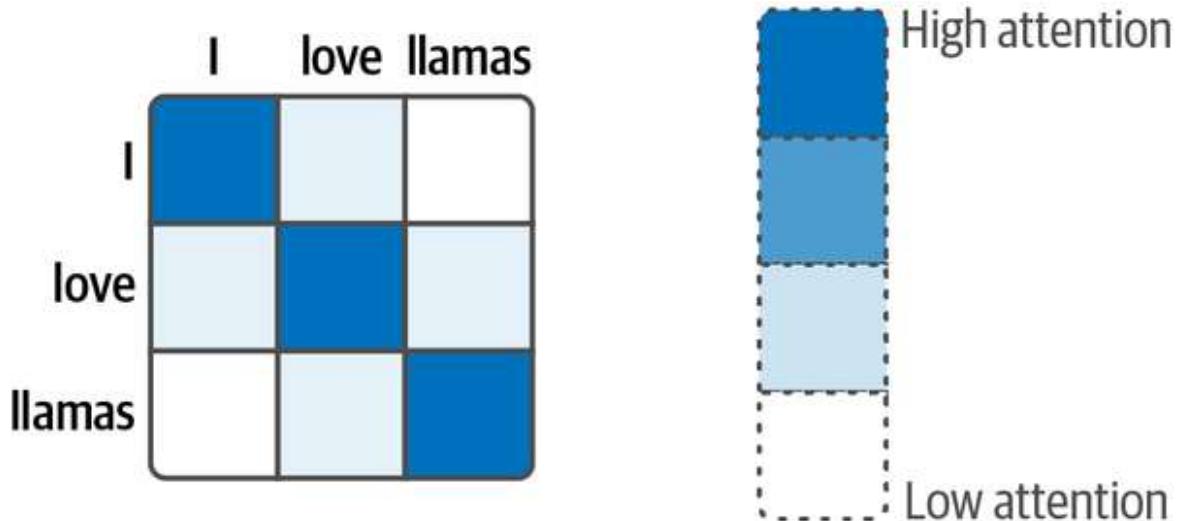


Figure 1-18. Self-attention attends to all parts of the input sequence so that it can “look” both forward and back in a single sequence.

Compared to the encoder, the decoder has an additional layer that pays attention to the output of the encoder (to find the relevant parts of the input). As demonstrated in [Figure 1-19](#), this process is similar to the RNN attention decoder that we discussed previously.

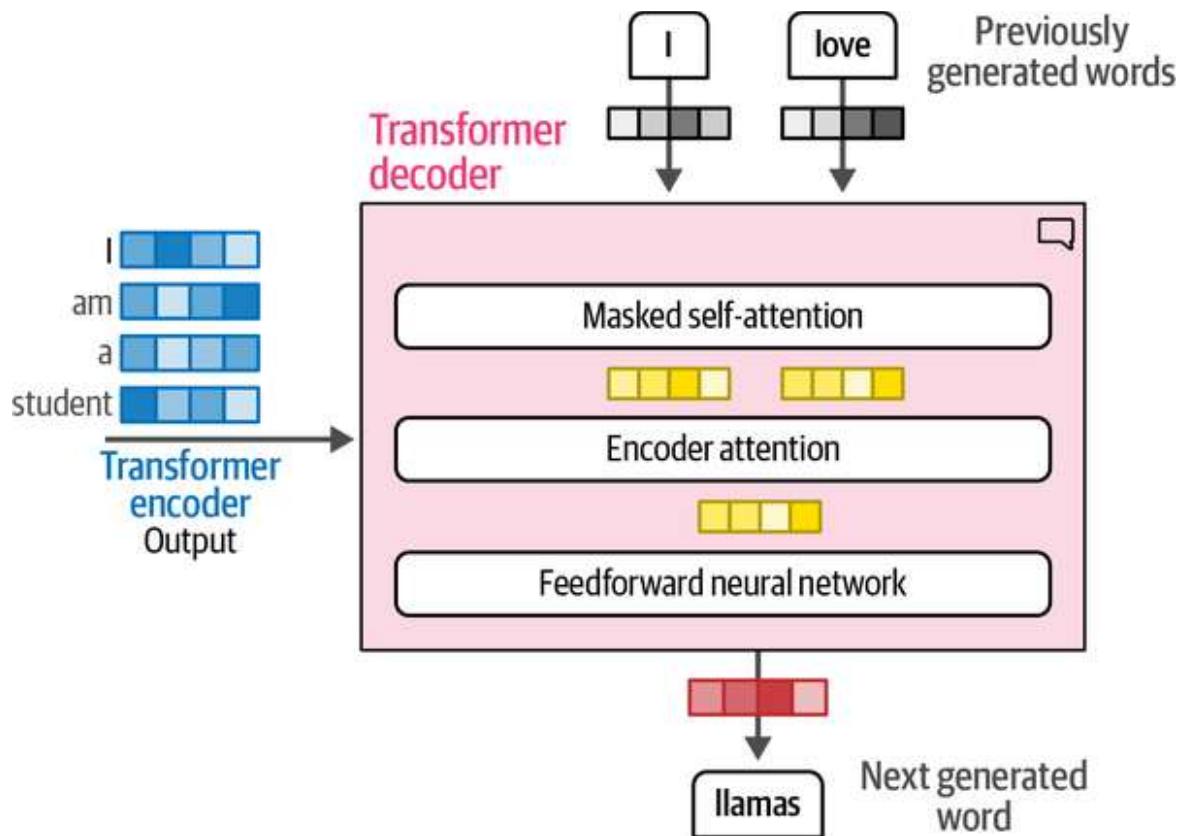


Figure 1-19. The decoder has an additional attention layer that attends to the output of the encoder.

As shown in [Figure 1-20](#), the self-attention layer in the decoder masks future positions so it only attends to earlier positions to prevent leaking information when generating the output.

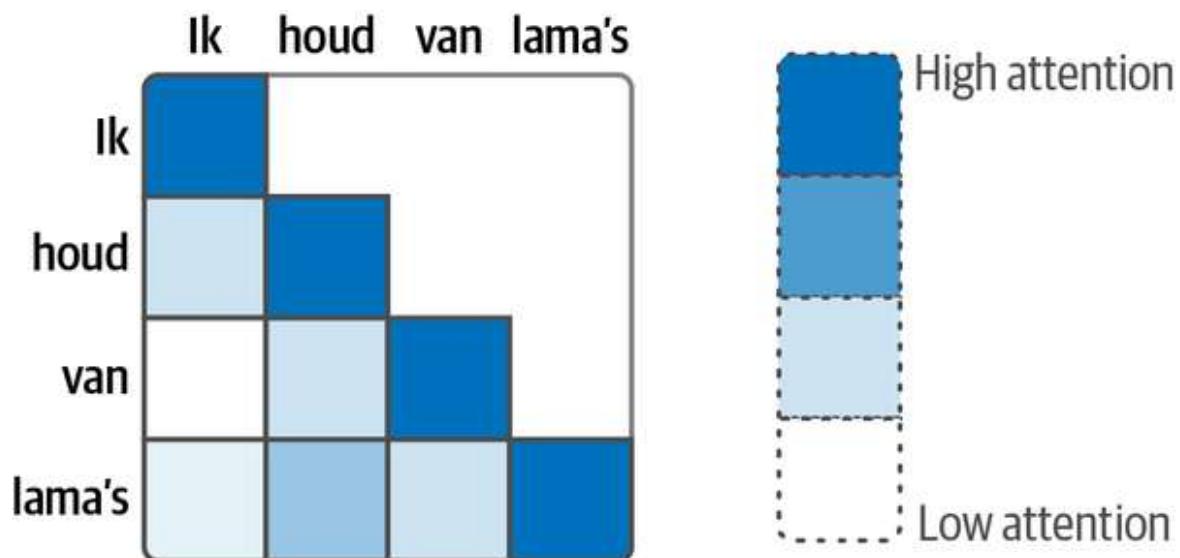


Figure 1-20. Only attend to previous tokens to prevent “looking into the future.”

Together, these building blocks create the Transformer architecture and are the foundation of many impactful models in Language AI, such as BERT and GPT-1, which we cover later in this chapter. Throughout this book, most models that we will use are Transformer-based models.

There is much more to the Transformer architecture than what we explored thus far. In Chapters 2 and 3, we will go through the many reasons why Transformer models work so well, including multi-head attention, positional embeddings, and layer normalization.

Representation Models: Encoder-Only Models

The original Transformer model is an encoder-decoder architecture that serves translation tasks well but cannot easily be used for other tasks, like text classification.

In 2018, a new architecture called Bidirectional Encoder Representations from Transformers (BERT) was introduced that could be leveraged for a wide variety of tasks and would serve as the foundation of Language AI for years to come.⁶ BERT is an encoder-only architecture that focuses on representing language, as illustrated in [Figure 1-21](#). This means that it only uses the encoder and removes the decoder entirely.

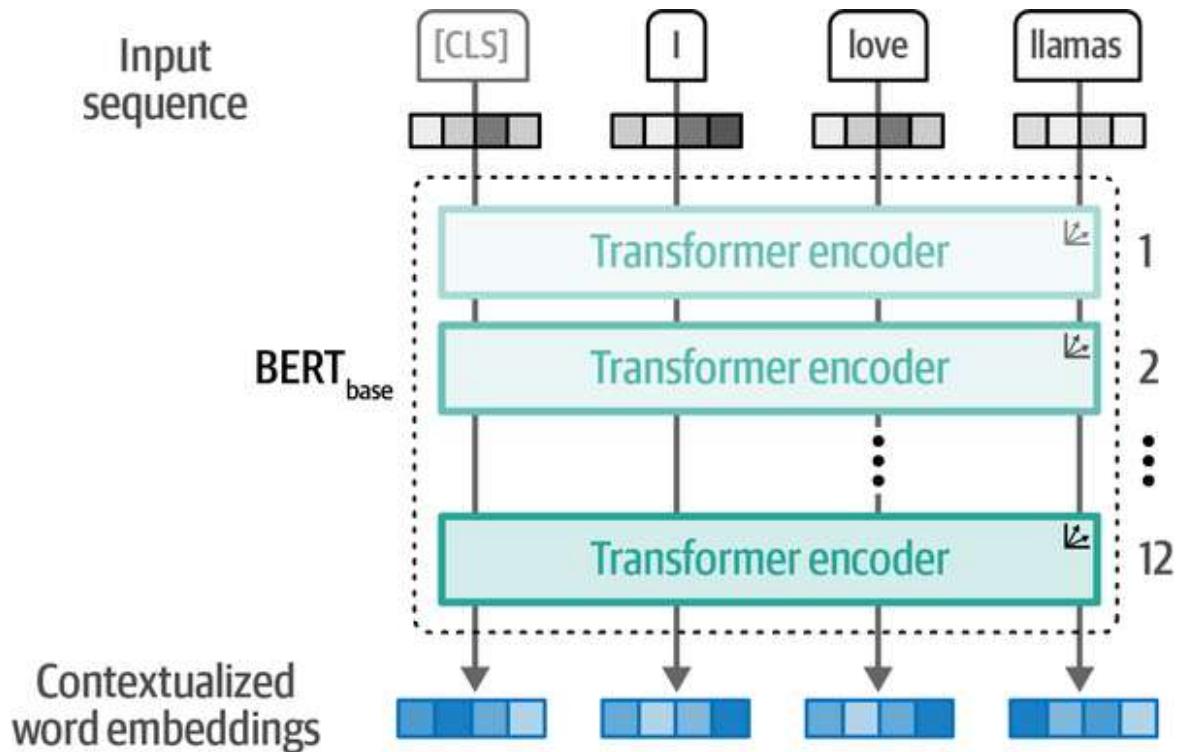


Figure 1-21. The architecture of a BERT base model with 12 encoders.

These encoder blocks are the same as we saw before: self-attention followed by feedforward neural networks. The input contains an additional token, the **[CLS]** or classification token, which is used as the representation for the entire input. Often, we use this **[CLS]** token as the input embedding for fine-tuning the model on specific tasks, like classification.

Training these encoder stacks can be a difficult task that BERT approaches by adopting a technique called *masked language modeling* (see Chapters 2 and 11). As shown in Figure 1-22, this method masks a part of the input for the model to predict. This prediction task is difficult but allows BERT to create more accurate (intermediate) representations of the input.

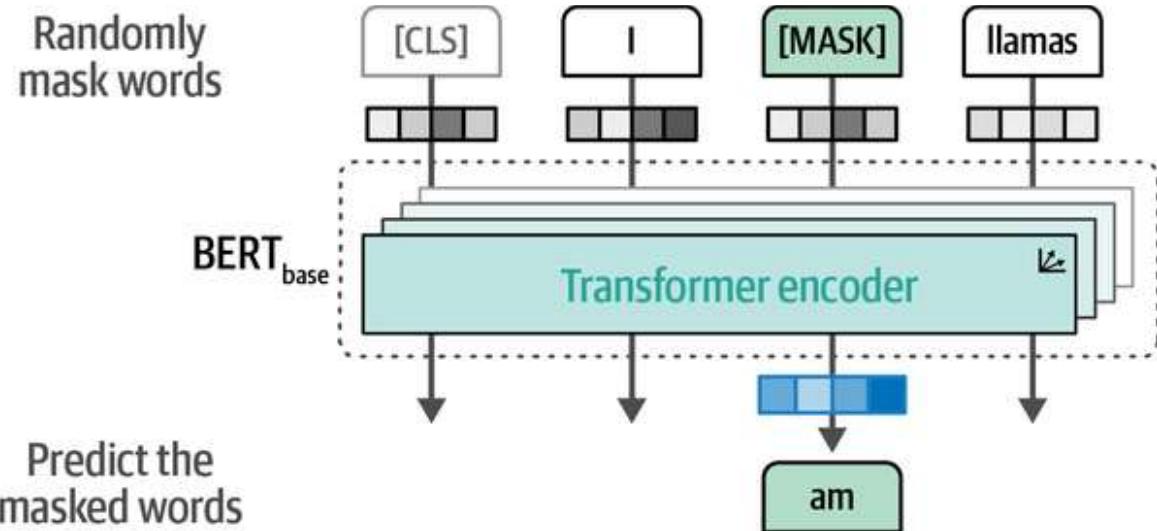


Figure 1-22. Train a BERT model by using masked language modeling.

This architecture and training procedure makes BERT and related architectures incredible at representing contextual language. BERT-like models are commonly used for *transfer learning*, which involves first pretraining it for language modeling and then fine-tuning it for a specific task. For instance, by training BERT on the entirety of Wikipedia, it learns to understand the semantic and contextual nature of text. Then, as shown in [Figure 1-23](#), we can use that *pretrained* model to *fine-tune* it for a specific task, like text classification.

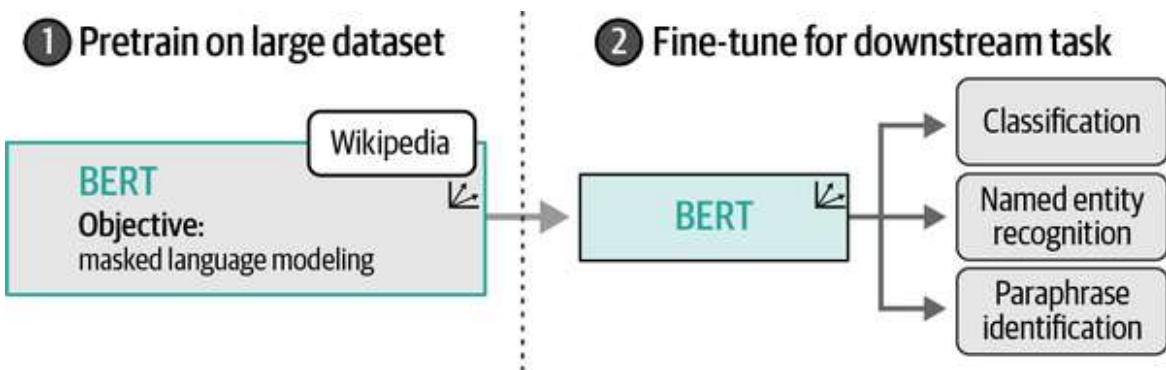


Figure 1-23. After pretraining BERT on masked language model, we fine-tune it for specific tasks.

A huge benefit of pretrained models is that most of the training is already done for us. Fine-tuning on specific tasks is generally less compute-intensive and requires less data. Moreover, BERT-like models generate embeddings at almost every step in their architecture. This also makes

BERT models feature extraction machines without the need to fine-tune them on a specific task.

Encoder-only models, like BERT, will be used in many parts of the book. For years, they have been and are still used for common tasks, including classification tasks (see [Chapter 4](#)), clustering tasks (see [Chapter 5](#)), and semantic search (see [Chapter 8](#)).

Throughout the book, we will refer to encoder-only models as *representation models* to differentiate them from decoder-only, which we refer to as *generative models*. Note that the main distinction does not lie between the underlying architecture and the way these models work.

Representation models mainly focus on representing language, for instance, by creating embeddings, and typically do not generate text. In contrast, generative models focus primarily on generating text and typically are not trained to generate embeddings.

The distinction between representation and generative models and components will also be shown in most images. Representation models are teal with a small vector icon (to indicate its focus on vectors and embeddings) whilst generative models are pink with a small chat icon (to indicate its generative capabilities).

Generative Models: Decoder-Only Models

Similar to the encoder-only architecture of BERT, a decoder-only architecture was proposed in 2018 to target generative tasks.⁷ This architecture was called a Generative Pre-trained Transformer (GPT) for its generative capabilities (it's now known as GPT-1 to distinguish it from later versions). As shown in [Figure 1-24](#), it stacks decoder blocks similar to the encoder-stacked architecture of BERT.

GPT-1 was trained on a corpus of 7,000 books and Common Crawl, a large dataset of web pages. The resulting model consisted of 117 million *parameters*. Each parameter is a numerical value that represents the model's understanding of language.

If everything remains the same, we expect more parameters to greatly influence the capabilities and performance of language models. Keeping this in mind, we saw larger and larger models being released at a steady pace. As illustrated in [Figure 1-25](#), GPT-2 had 1.5 billion parameters⁸ and GPT-3 used 175 billion parameters⁹ quickly followed.

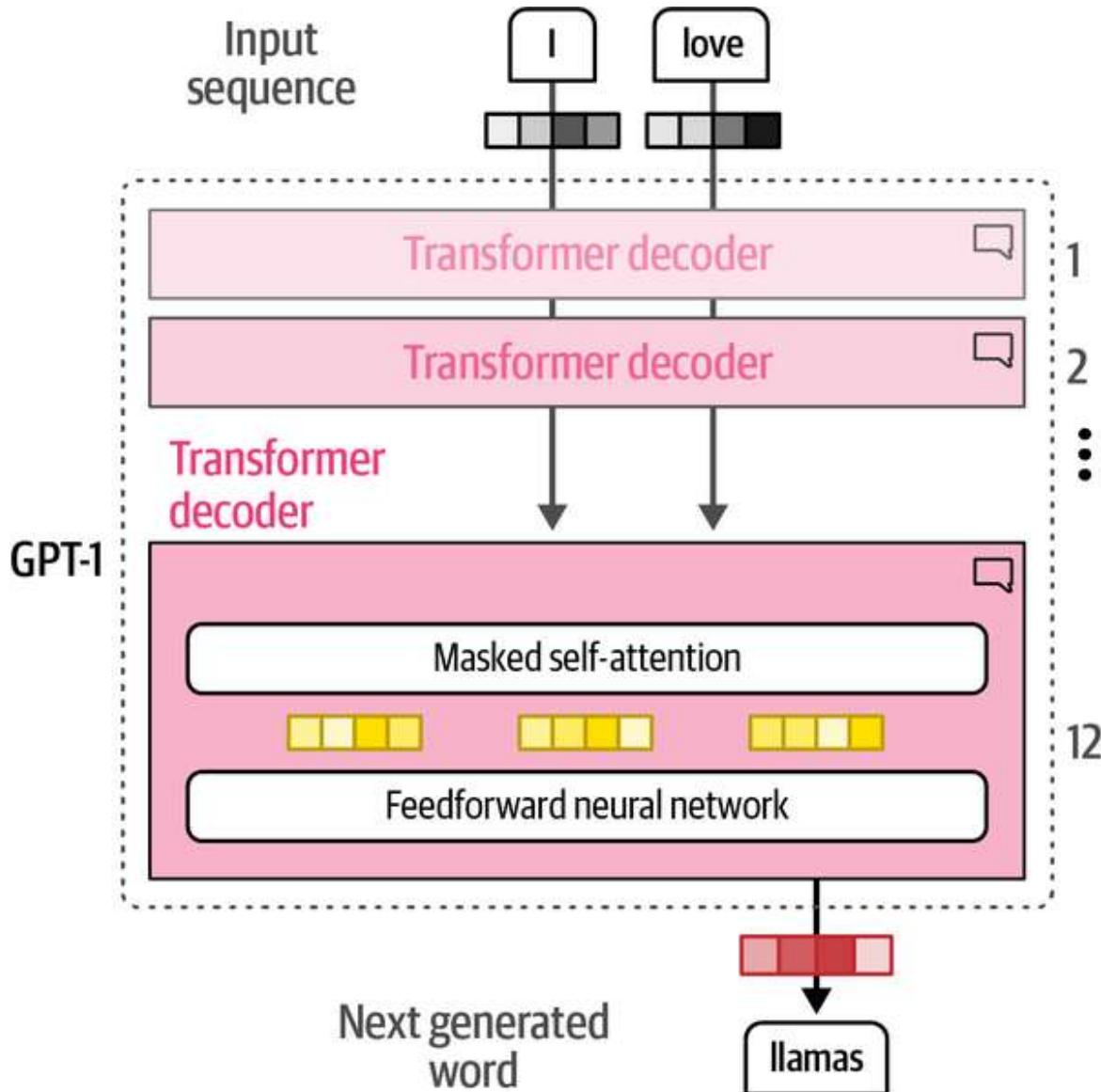


Figure 1-24. The architecture of a GPT-1. It uses a decoder-only architecture and removes the encoder-attention block.

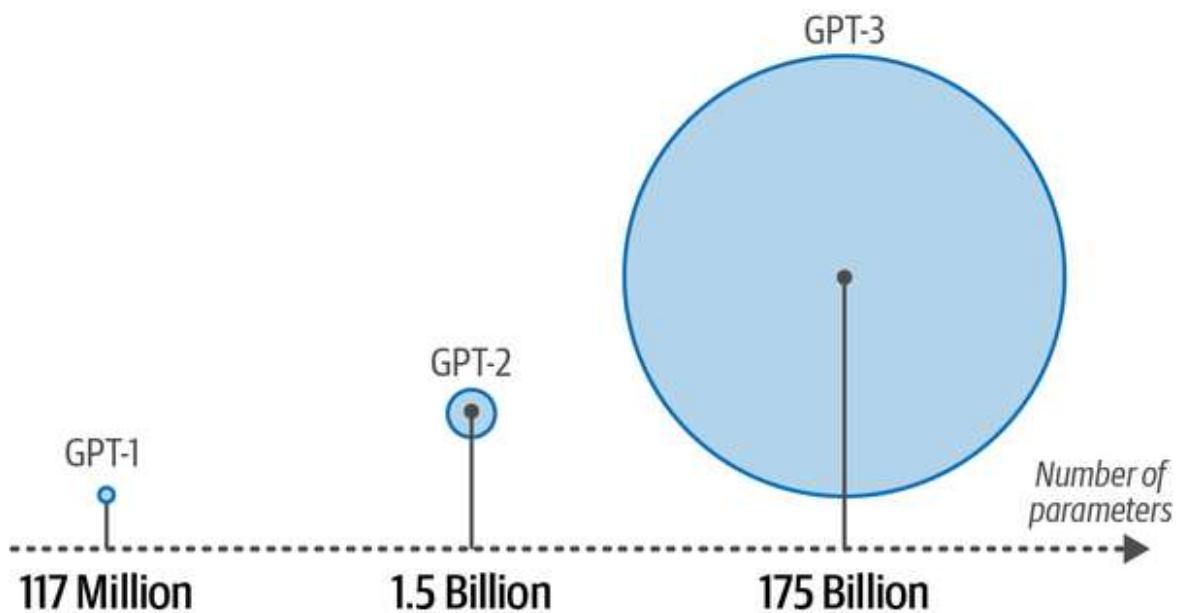


Figure 1-25. GPT models quickly grew in size with each iteration.

These generative decoder-only models, especially the “larger” models, are commonly referred to as *large language models* (LLMs). As we will discuss later in this chapter, the term LLM is not only reserved for generative models (decoder-only) but also representation models (encoder-only).

Generative LLMs, as sequence-to-sequence machines, take in some text and attempt to autocomplete it. Although a handy feature, their true power shone from being trained as a chatbot. Instead of completing a text, what if they could be trained to answer questions? By fine-tuning these models, we can create *instruct* or *chat* models that can follow directions.

As illustrated in [Figure 1-26](#), the resulting model could take in a user query (*prompt*) and output a response that would most likely follow that prompt. As such, you will often hear that generative models are *completion* models.

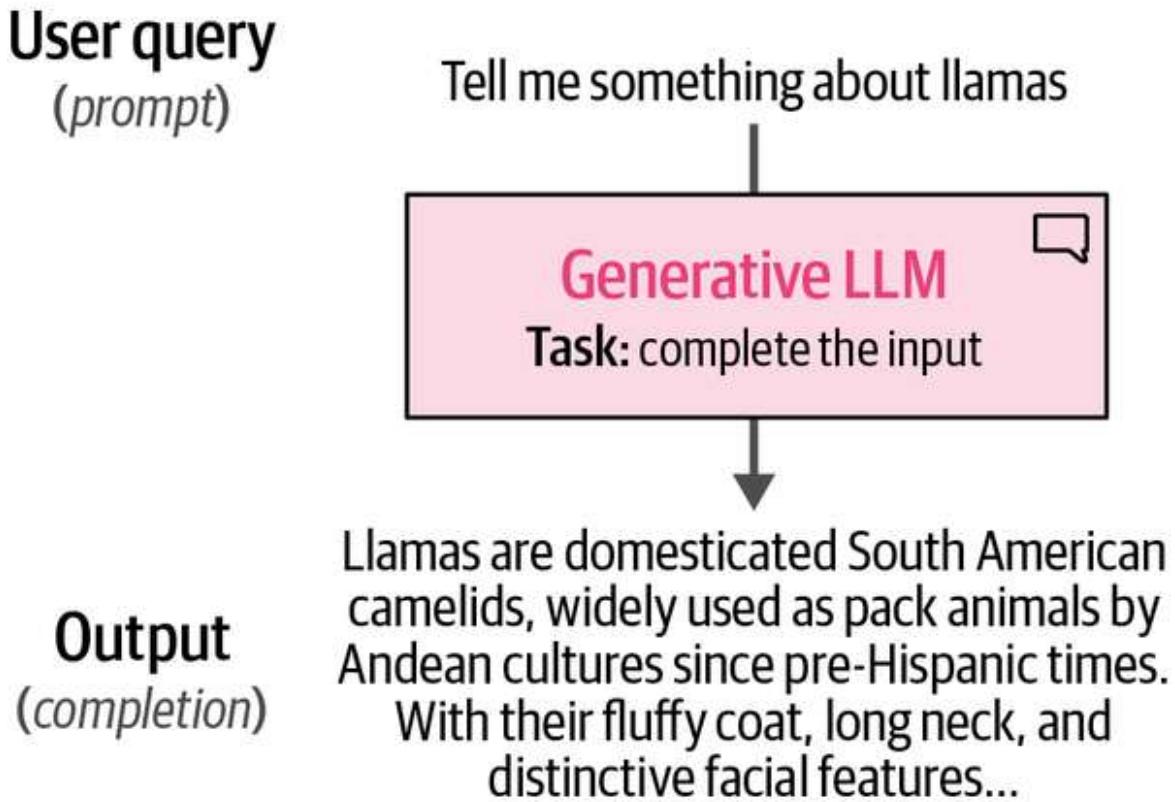


Figure 1-26. Generative LLMs take in some input and try to complete it. With instruct models, this is more than just autocomplete and attempts to answer the question.

A vital part of these completion models is something called the *context length* or *context window*. The context length represents the maximum number of tokens the model can process, as shown in [Figure 1-27](#). A large context window allows entire documents to be passed to the LLM. Note that due to the autoregressive nature of these models, the current context length will increase as new tokens are generated.

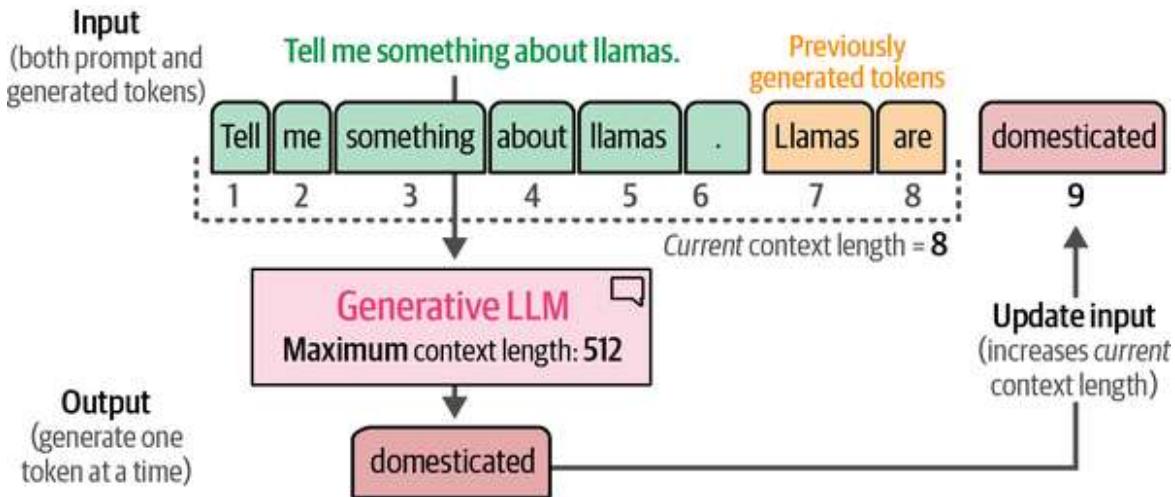


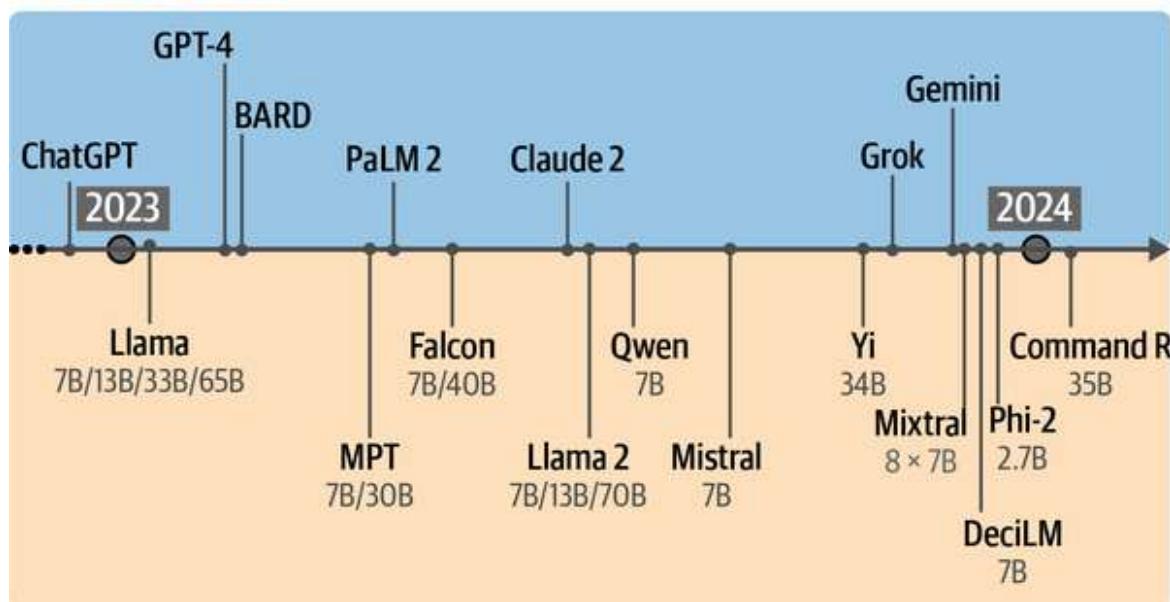
Figure 1-27. The context length is the maximum context an LLM can handle.

The Year of Generative AI

LLMs had a tremendous impact on the field and led some to call 2023 *The Year of Generative AI* with the release, adoption, and media coverage of ChatGPT (GPT-3.5). When we refer to ChatGPT, we are actually talking about the product and not the underlying model. When it was first released, it was powered by the GPT-3.5 LLM and has since then grown to include several more performant variants, such as GPT-4.¹⁰

GPT-3.5 was not the only model that made its impact in the Year of Generative AI. As illustrated in Figure 1-28, both open source and proprietary LLMs have made their way to the people at an incredible pace. These open source base models are often referred to as *foundation models* and can be fine-tuned for specific tasks, like following instructions.

Proprietary models



Open models

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

Apart from the widely popular Transformer architecture, new promising architectures have emerged such as Mamba^{11,12} and RWKV.¹³ These novel architectures attempt to reach Transformer-level performance with additional advantages, like larger context windows or faster inference.

These developments exemplify the evolution of the field and showcase 2023 as a truly hectic year for AI. It took all we had to just keep up with the many developments, both within and outside of Language AI.

As such, this book explores more than just the latest LLMs. We will explore how other models, such as embedding models, encoder-only models, and even bag-of-words can be used to empower LLMs.

The Moving Definition of a “Large Language Model”

In our travels through the recent history of Language AI, we observed that primarily generative decoder-only (Transformer) models are commonly

referred to as *large language models*. Especially if they are considered to be “large.” In practice, this seems like a rather constrained description!

What if we create a model with the same capabilities as GPT-3 but 10 times smaller? Would such a model fall outside the “large” language model categorization?

Similarly, what if we released a model as big as GPT-4 that can perform accurate text classification but does not have any generative capabilities? Would it still qualify as a large “language model” if its primary function is not language generation, even though it still represents text?

The problem with these kinds of definitions is that we exclude capable models. What name we give one model or the other does not change how it behaves.

Since the definition of the term “large language model” tends to evolve with the release of new models, we want to be explicit in what it means for this book. “Large” is arbitrary and what might be considered a large model today could be small tomorrow. There are currently many names for the same thing and to us, “large language models” are also models that do not generate text and can be run on consumer hardware.

As such, aside from covering generative models, this book will also cover models with fewer than 1 billion parameters that do not generate text. We will explore how other models, such as embedding models, representation models, and even bag-of-words can be used to empower LLMs.

The Training Paradigm of Large Language Models

Traditional machine learning generally involves training a model for a specific task, like classification. As shown in [Figure 1-29](#), we consider this to be a one-step process.



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

Creating LLMs, in contrast, typically consists of at least two steps:

Language modeling

The first step, called *pretraining*, takes the majority of computation and training time. An LLM is trained on a vast corpus of internet text allowing the model to learn grammar, context, and language patterns. This broad training phase is not yet directed toward specific tasks or applications beyond predicting the next word. The resulting model is often referred to as a *foundation model* or *base model*. These models generally do not follow instructions.

Fine-tuning

The second step, *fine-tuning* or sometimes *post-training*, involves using the previously trained model and further training it on a narrower task. This allows the LLM to adapt to specific tasks or to exhibit desired behavior. For example, we could fine-tune a base model to perform well on a classification task or to follow instructions. It saves massive amounts of resources because the pretraining phase is quite costly and generally requires data and computing resources that are out of the reach of most people and organizations. For instance, Llama 2 has been trained on a dataset containing 2 trillion tokens.¹⁴ Imagine the compute

necessary to create that model! In [Chapter 12](#), we will go over several methods for fine-tuning foundation models on your dataset.

Any model that goes through the first step, pretraining, we consider a *pretrained model*, which also includes fine-tuned models. This two-step approach of training is visualized in [Figure 1-30](#).

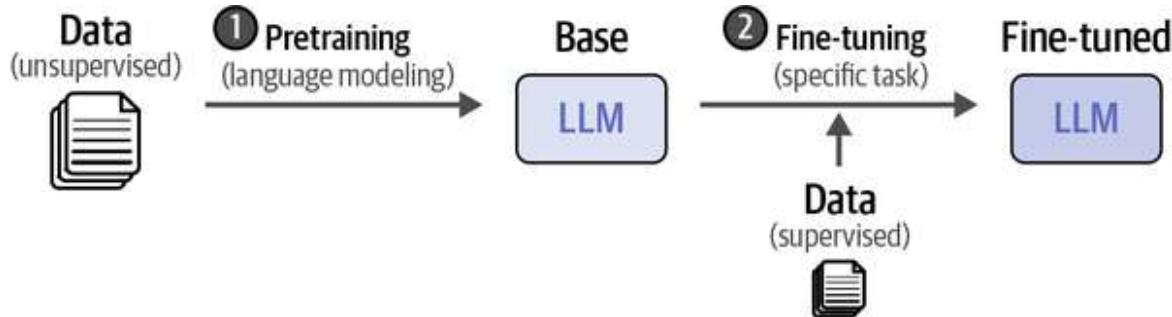


Figure 1-30. Compared to traditional machine learning, LLM training takes a multistep approach.

Additional fine-tuning steps can be added to further align the model with the user's preferences, as we will explore in [Chapter 12](#).

Large Language Model Applications: What Makes Them So Useful?

The nature of LLMs makes them suitable for a wide range of tasks. With text generation and prompting, it almost seems as if your imagination is the limit. To illustrate, let's explore some common tasks and techniques:

Detecting whether a review left by a customer is positive or negative

This is (supervised) classification and can be handled with both encoder- and decoder-only models either with pretrained models (see [Chapter 4](#)) or by fine-tuning models (see [Chapter 11](#)).

Developing a system for finding common topics in ticket issues

This is (unsupervised) classification for which we have no predefined labels. We can leverage encoder-only models to perform the classification itself and decoder-only models for labeling the topics (see [Chapter 5](#)).

Building a system for retrieval and inspection of relevant documents

A major component of language model systems is their ability to add external resources of information. Using semantic search, we can build systems that allow us to easily access and find information for an LLM to use (see [Chapter 8](#)). Improve your system by creating or fine-tuning a custom embedding model (see [Chapter 12](#)).

Constructing an LLM chatbot that can leverage external resources, such as tools and documents

This is a combination of techniques that demonstrates how the true power of LLMs can be found through additional components. Methods such as prompt engineering (see [Chapter 6](#)), retrieval-augmented generation (see [Chapter 8](#)), and fine-tuning an LLM (see [Chapter 12](#)) are all pieces of the LLM puzzle.

Constructing an LLM capable of writing recipes based on a picture showing the products in your fridge

This is a multimodal task where the LLM takes in an image and reasons about what it sees (see [Chapter 9](#)). LLMs are being adapted to other modalities, such as Vision, which opens a wide variety of interesting use cases.

LLM applications are incredibly satisfying to create since they are partially bounded by the things you can imagine. As these models grow more accurate, using them in practice for creative use cases such as role-playing and writing children's books simply becomes more and more fun.

Responsible LLM Development and Usage

The impact of LLMs has been and likely continues to be significant due to their widespread adoption. As we explore the incredible capabilities of LLMs it is important to keep their societal and ethical implications in mind. Several key points to consider:

Bias and fairness

LLMs are trained on large amounts of data that might contain biases.

LLMs might learn from these biases, start to reproduce them, and potentially amplify them. Since the data on which LLMs are trained are seldom shared, it remains unclear what potential biases they might contain unless you try them out.

Transparency and accountability

Due to LLMs' incredible capabilities, it is not always clear when you are talking with a human or an LLM. As such, the usage of LLMs when interacting with humans can have unintended consequences when there is no human in the loop. For instance, LLM-based applications used in the medical field might be regulated as medical devices since they could affect a patient's well-being.

Generating harmful content

An LLM does not necessarily generate ground-truth content and might confidently output incorrect text. Moreover, they can be used to

generate fake news, articles, and other misleading sources of information.

Intellectual property

Is the output of an LLM your intellectual property or that of the LLM's creator? When the output is similar to a phrase in the training data, does the intellectual property belong to the author of that phrase? Without access to the training data it remains unclear when copyrighted material is being used by the LLM.

Regulation

Due to the enormous impact of LLMs, governments are starting to regulate commercial applications. An example is the [European AI Act](#), which regulates the development and deployment of foundation models including LLMs.

As you develop and use LLMs, we want to stress the importance of ethical considerations and urge you to learn more about the safe and responsible use of LLMs and AI systems in general.

Limited Resources Are All You Need

The compute resources that we have referenced several times thus far generally relate to the GPU(s) you have available on your system. A powerful GPU (graphics card) will make both training and using LLMs much more efficient and faster.

In choosing a GPU, an important component is the amount of VRAM (video random-access memory) you have available. This refers to the amount of memory you have available on your GPU. In practice, the more

VRAM you have the better. The reason for this is that some models simply cannot be used at all if you do not have sufficient VRAM.

Because training and fine-tuning LLMs can be an expensive process, GPU-wise, those without a powerful GPU have often been referred to as the GPU-poor. This illustrates the battle for computing resources to train these huge models. To create the Llama 2 family of models, for example, Meta used A100-80 GB GPUs. Assuming renting such a GPU would cost \$1.50/hr, the total costs of creating these models would exceed \$5,000,000!¹⁵

Unfortunately, there is no single rule to determine exactly how much VRAM you need for a specific model. It depends on the model's architecture and size, compression technique, context size, backend for running the model, etc.

This book is for the GPU-poor! We will use models that users can run without the most expensive GPU(s) available or a big budget. To do so, we will make all the code available in Google Colab instances. At the time of writing, a free instance of Google Colab will net you a T4 GPU with 16 GB VRAM, which is the minimum amount of VRAM that we suggest.

Interfacing with Large Language Models

Interfacing with LLMs is a vital component of not only using them but also developing an understanding of their inner workings. Due to the many developments in the field, there has been an abundance of techniques, methods, and packages for communicating with LLMs. Throughout the book, we intend to explore the most common techniques for doing so, including using both proprietary (closed source) and publicly available open models.

Proprietary, Private Models

Closed source LLMs are models that do not have their weights and architecture shared with the public. They are developed by specific

organizations with their underlying code being kept secret. Examples of such models include OpenAI's GPT-4 and Anthropic's Claude. These proprietary models are generally backed by significant commercial support and have been developed and integrated within their services.

You can access these models through an interface that communicates with the LLM, called an API (application programming interface), as illustrated in [Figure 1-31](#). For instance, to use ChatGPT in Python you can use [OpenAI's package](#) to interface with the service without directly accessing it.

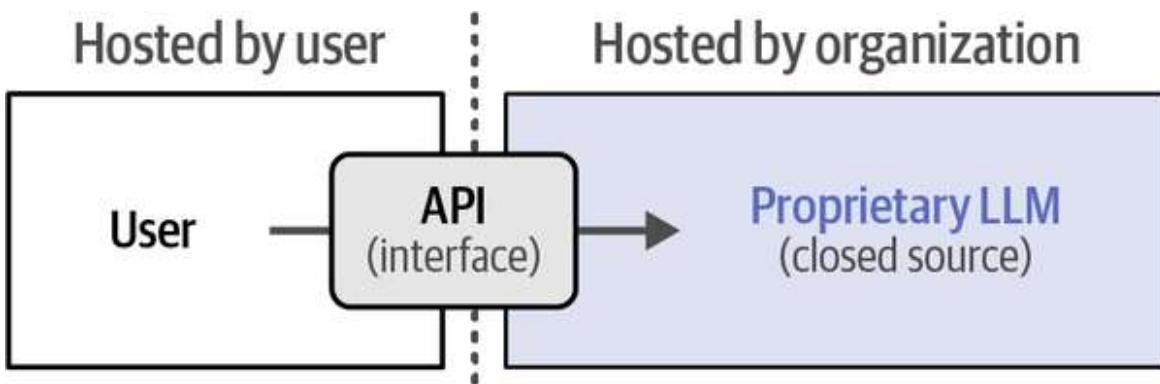


Figure 1-31. Closed source LLMs are accessed by an interface (API). As a result, details of the LLM itself, including its code and architecture are not shared with the user.

A huge benefit of proprietary models is that the user does not need to have a strong GPU to use the LLM. The provider takes care of hosting and running the model and generally has more computing available. There is no expertise necessary concerning hosting and using the model, which lowers the barrier to entry significantly. Moreover, these models tend to be more performant than their open source counterparts due to the significant investment from these organizations.

A downside to this is that it can be a costly service. The provider manages the risk and costs of hosting the LLM, which often translates to a paid service. Moreover, since there is no direct access to the model, there is no method to fine-tune it yourself. Lastly, your data is shared with the provider, which is not desirable in many common use cases, such as sharing patient data.

Open Models

Open LLMs are models that share their weights and architecture with the public to use. They are still developed by specific organizations but often share their code for creating or running the model locally—with varying levels of licensing that may or may not allow commercial usage of the model. Cohere’s Command R, the Mistral models, Microsoft’s Phi, and Meta’s Llama models are all examples of open models.

NOTE

There are ongoing discussions as to what truly represents an open source model. For instance, some publicly shared models have a permissive commercial license, which means that the model cannot be used for commercial purposes. For many, this is not the true definition of open source, which states that using these models should not have any restrictions. Similarly, the data on which a model is trained as well as its source code are seldom shared.

You can download these models and use them on your device as long as you have a powerful GPU that can handle these kinds of models, as shown in [Figure 1-32](#).

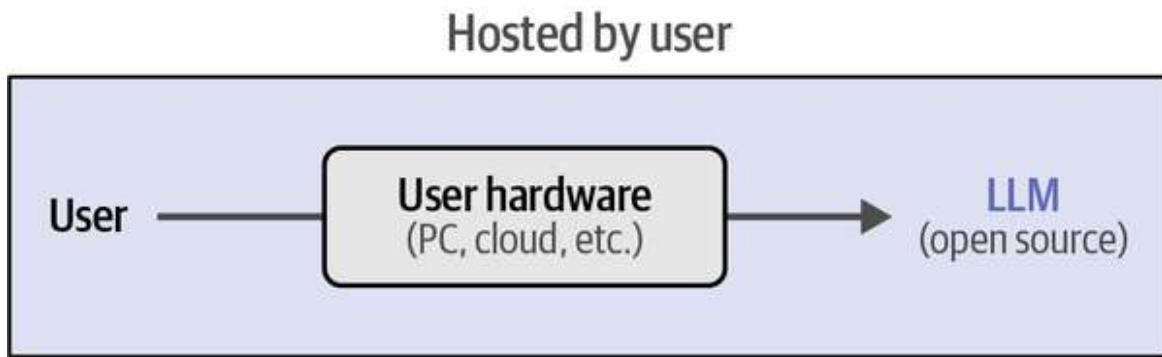


Figure 1-32. Open source LLMs are directly by the user. As a result, details of the LLM itself including its code and architecture are shared with the user.

A major advantage of these local models is that you, the user, have complete control over the model. You can use the model without depending on the API connection, fine-tune it, and run sensitive data through it. You are not dependent on any service and have complete transparency of the

processes that lead to the output of the model. This benefit is enhanced by the large communities that enable these processes, such as [Hugging Face](#), demonstrating the possibilities of collaborative efforts.

A downside is that you need powerful hardware to run these models and even more when training or fine-tuning them. Moreover, it requires specific knowledge to set up and use these models (which we will cover throughout this book).

We generally prefer using open source models wherever we can. The freedom this gives to play around with options, explore the inner workings, and use the model locally arguably provides more benefits than using proprietary LLMs.

Open Source Frameworks

Compared to closed source LLMs, open source LLMs require you to use certain packages to run them. In 2023, many different packages and frameworks were released that, each in their own way, interact with and make use of LLMs. Wading through hundreds upon hundreds of potentially worthwhile frameworks is not the most enjoyable experience.

As a result, you might even miss your favorite framework in this book!

Instead of attempting to cover every LLM framework in existence (there are too many, and they continue to grow in number), we aim to provide you with a solid foundation for leveraging LLMs. The idea is that after reading this book, you can easily pick up most other frameworks as they all work in a very similar manner.

The intuition that we attempt to realize is an important component of this. If you have an intuitive understanding of not only LLMs but also using them in practice with common frameworks, branching out to others should be a straightforward task.

More specifically, we focus on backend packages. These are packages without a GUI (graphical user interface) that are created for efficiently

loading and running any LLM on your device, such as [llama.cpp](#), [LangChain](#), and the core of many frameworks, [Hugging Face Transformers](#).

TIP

We will mostly cover frameworks for interacting with large language models through code. Although it helps you learn the fundamentals of these frameworks, sometimes you just want a ChatGPT-like interface with a local LLM. Fortunately, there are many incredible frameworks that allow for this. A few examples include [text-generation-webui](#), [KoboldCpp](#), and [LM Studio](#).

Generating Your First Text

An important component of using language models is selecting them. The main source for finding and downloading LLMs is the [Hugging Face Hub](#). Hugging Face is the organization behind the well-known Transformers package, which for years has driven the development of language models in general. As the name implies, the package was built on top of the [transformers](#) framework that we discussed in “[A Recent History of Language AI](#)”.

At the time of writing, you will find more than 800,000 models on Hugging Face’s platform for many different purposes, from LLMs and computer vision models to models that work with audio and tabular data. Here, you can find almost any open source LLM.

Although we will explore all kinds of models throughout this book, let’s start our first lines of code with a generative model. The main generative model we use throughout the book is Phi-3-mini, which is a relatively small (3.8 billion parameters) but quite performant model.¹⁶ Due to its small size, the model can be run on devices with less than 8 GB of VRAM. If you perform quantization, a type of compression that we will further discuss in Chapters [7](#) and [12](#), you can use even less than 6 GB of VRAM. Moreover, the model is licensed under the MIT license, which allows the model to be used for commercial purposes without constraints!

Keep in mind that new and improved LLMs are frequently released. To ensure this book remains current, most examples are designed to work with any LLM. We'll also highlight different models in the repository associated with this book for you to try out.

Let's get started! When you use an LLM, two models are loaded:

- The generative model itself
- Its underlying tokenizer

The tokenizer is in charge of splitting the input text into tokens before feeding it to the generative model. You can find the tokenizer and model on the [Hugging Face site](#) and only need the corresponding IDs to be passed. In this case, we use “microsoft/Phi-3-mini-4k-instruct” as the main path to the model.

We can use `transformers` to load both the tokenizer and model. Note that we assume you have an NVIDIA GPU (`device_map="cuda"`) but you can choose a different device instead. If you do not have access to a GPU you can use the free Google Colab notebooks we made available in the repository of this book:

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")
```

Running the code will start downloading the model and depending on your internet connection can take a couple of minutes.

Although we now have enough to start generating text, there is a nice trick in transformers that simplifies the process, namely `transformers.pipeline`. It encapsulates the model, tokenizer, and text generation process into a single function:

```
from transformers import pipeline

# Create a pipeline
generator = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=500,
    do_sample=False
)
```

The following parameters are worth mentioning:

`return_full_text`

By setting this to `False`, the prompt will not be returned but merely the output of the model.

`max_new_tokens`

The maximum number of tokens the model will generate. By setting a limit, we prevent long and unwieldy output as some models might continue generating output until they reach their context window.

`do_sample`

Whether the model uses a sampling strategy to choose the next token. By setting this to `False`, the model will always select the next most probable token. In [Chapter 6](#), we explore several sampling parameters that invoke some creativity in the model's output.

To generate our first text, let's instruct the model to tell a joke about chickens. To do so, we format the prompt in a list of dictionaries where each dictionary relates to an entity in the conversation. Our role is that of “user” and we use the “content” key to define our prompt:

```
# The prompt (user input / query)
messages = [
    {"role": "user", "content": "Create a funny joke about
chickens."}
]

# Generate output
output = generator(messages)
print(output[0]["generated_text"])
```

Why don't chickens like to go to the gym? Because they can't crack the egg-sistence of it!

And that is it! The first text generated in this book was a decent joke about chickens.

Summary

In this first chapter of the book, we delved into the revolutionary impact LLMs have had on the Language AI field. It has significantly changed our approach to tasks such as translation, classification, summarization, and more. Through a recent history of Language AI, we explored the fundamentals of several types of LLMs, from a simple bag-of-words representation to more complex representations using neural networks.

We discussed the attention mechanism as a step toward encoding context within models, a vital component of what makes LLMs so capable. We touched on two main categories of models that use this incredible mechanism: representation models (encoder-only) like BERT and generative models (decoder-only) like the GPT family of models. Both categories are considered large language models throughout this book.

Overall, the chapter provided an overview of the landscape of Language AI, including its applications, societal and ethical implications, and the resources needed to run such models. We ended by generating our first text using Phi-3, a model that will be used throughout the book.

In the next two chapters, you will learn about some underlying processes. We start by exploring tokenization and embeddings in [Chapter 2](#), two often underestimated but vital components of the Language AI field. What follows in [Chapter 3](#) is an in-depth look into language models where you will discover the precise methods used for generating text.

- ¹ J. McCarthy (2007). “What is artificial intelligence?” Retrieved from <https://oreil.ly/C7sja> and <https://oreil.ly/n9X8O>.
- ² Fabrizio Sebastiani. “Machine learning in automated text categorization.” *ACM Computing Surveys (CSUR)* 34.1 (2002): 1–47.
- ³ Tomas Mikolov et al. “Efficient estimation of word representations in vector space.” *arXiv preprint arXiv:1301.3781* (2013).
- ⁴ Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate.” *arXiv preprint arXiv:1409.0473* (2014).
- ⁵ Ashish Vaswani et al. “Attention is all you need.” *Advances in Neural Information Processing Systems* 30 (2017).
- ⁶ Jacob Devlin et al. “BERT: Pre-training of deep bidirectional transformers for language understanding.” *arXiv preprint arXiv:1810.04805* (2018).
- ⁷ Alec Radford et al. “Improving language understanding by generative pre-training”, (2018).
- ⁸ Alec Radford et al. “Language models are unsupervised multitask learners.” *OpenAI Blog* 1.8 (2019): 9.
- ⁹ Tom Brown et al. “Language models are few-shot learners.” *Advances in Neural Information Processing Systems* 33 (2020): 1877–1901.
- ¹⁰ OpenAI, “Gpt-4 technical report.” *arXiv preprint arXiv:2303.08774* (2023).
- ¹¹ Albert Gu and Tri Dao. “Mamba: Linear-time sequence modeling with selective state spaces.” *arXiv preprint arXiv:2312.00752* (2023).
- ¹² See “[A Visual Guide to Mamba and State Space Models](#)” for an illustrated and visual guide to Mamba as an alternative to the Transformer architecture.

- ¹³ Bo Peng et al. “RWKV: Reinventing RNNs for the transformer era.” *arXiv preprint arXiv:2305.13048* (2023).
- ¹⁴ Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models.” *arXiv preprint arXiv:2307.09288* (2023).
- ¹⁵ The models were trained for 3,311,616 GPU hours, which refers to the amount of time it takes to train a model on a GPU, multiplied by the number of GPUs available.
- ¹⁶ Marah Abdin et al. “Phi-3 technical report: A highly capable language model locally on your phone.” *arXiv preprint arXiv:2404.14219* (2024).

Chapter 2. Tokens and Embeddings

Tokens and embeddings are two of the central concepts of using large language models (LLMs). As we've seen in the first chapter, they're not only important to understanding the history of Language AI, but we cannot have a clear sense of how LLMs work, how they're built, and where they will go in the future without a good sense of tokens and embeddings, as we can see in [Figure 2-1](#).

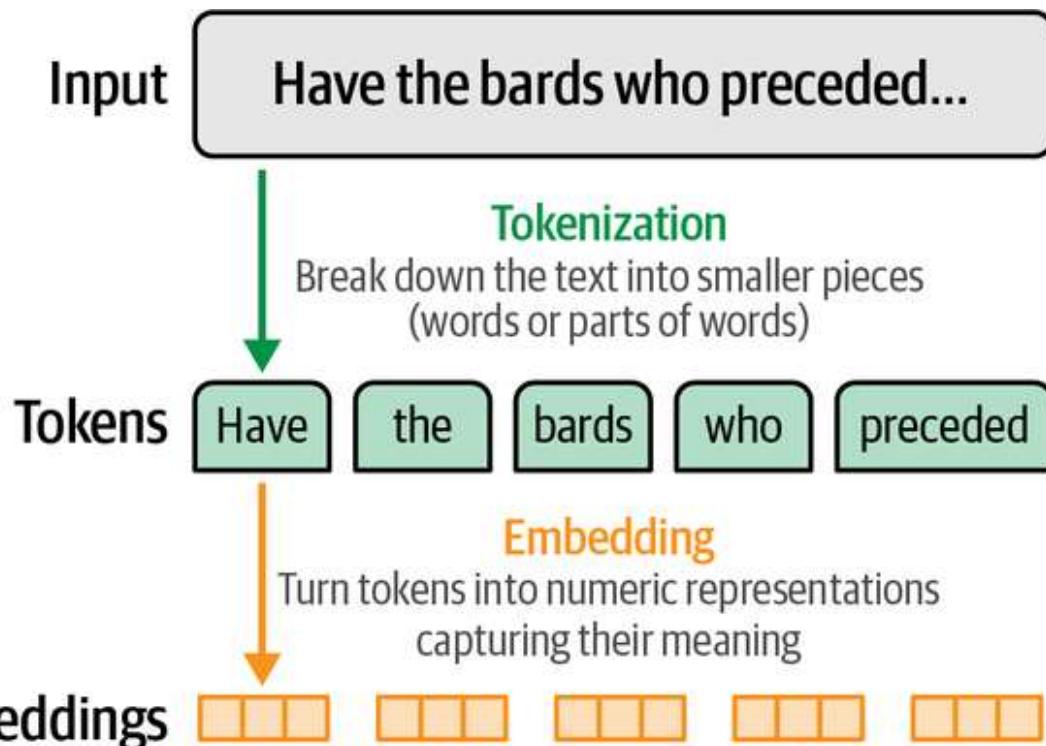


Figure 2-1. Language models deal with text in small chunks called tokens. For the language model to compute language, it needs to turn tokens into numeric representations called embeddings.

In this chapter, we look more closely at what tokens are and the tokenization methods used to power LLMs. We will then dive into the famous word2vec embedding method that preceded modern-day LLMs and see how it's extending the concept of token embeddings to build

commercial recommendation systems that power a lot of the apps you use. Finally, we go from token embeddings into *sentence* or *text* embeddings, where a whole sentence or document can have one vector that represents it—enabling applications like semantic search and topic modeling that we see in Part II of this book.

LLM Tokenization

The way the majority of people interact with language models, at the time of this writing, is through a web playground that presents a chat interface between the user and a language model. You may notice that a model does not produce its output response all at once; it actually generates one token at a time.

But tokens aren't only the output of a model, they're also the way in which the model sees its inputs. A text prompt sent to the model is first broken down into tokens, as we'll now see.

How Tokenizers Prepare the Inputs to the Language Model

Viewed from the outside, generative LLMs take an input prompt and generate a response, as we can see in [Figure 2-2](#).

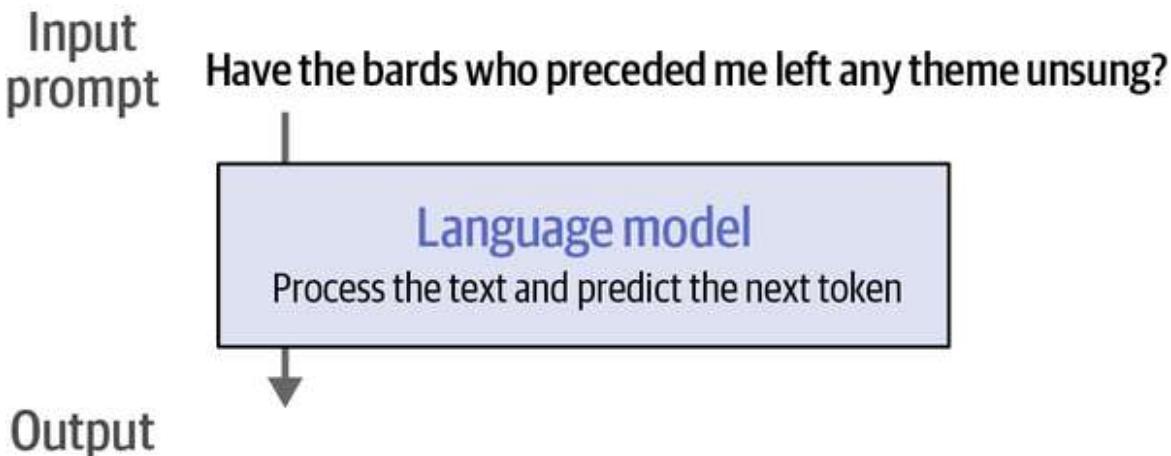


Figure 2-2. High-level view of a language model and its input prompt.

Before the prompt is presented to the language model, however, it first has to go through a tokenizer that breaks it into pieces. You can find an example showing the tokenizer of GPT-4 on the [OpenAI Platform](#). If we feed it the input text, it shows the output in [Figure 2-3](#), where each token is shown in a different color.

GPT-3.5 & GPT-4 GPT-3 (Legacy)

Have the bards who preceded me left any theme unsung?

Tokens 13 **Characters** 53

Have the bards who preceded me left any theme unsung?

Text **Token IDs**

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>).

Let's look at a code example and interact with these tokens ourselves. Here we'll be downloading an LLM and seeing how to tokenize the input before generating text with the LLM.

Downloading and Running an LLM

Let's start by loading our model and its tokenizer as we've done in [Chapter 1](#):

```

from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")

```

We can then proceed to the actual generation. We first declare our prompt, then tokenize it, then pass those tokens to the model, which generates its output. In this case, we're asking the model to only generate 20 new tokens:

```

prompt = "Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.<|assistant|>"

# Tokenize the input prompt
input_ids = tokenizer(prompt,
return_tensors="pt").input_ids.to("cuda")

# Generate the text
generation_output = model.generate(
    input_ids=input_ids,
    max_new_tokens=20
)

# Print the output
print(tokenizer.decode(generation_output[0]))

```

Output:

```

<s> Write an email apologizing to Sarah for the tragic
gardening mishap. Explain how it happened.<|assistant|> Subject:
My Sincere Apologies for the Gardening Mishap

```

```

Dear

```

The text in bold is the 20 tokens generated by the model.

Looking at the code, we can see that the model does not in fact receive the text prompt. Instead, the tokenizers processed the input prompt, and returned the information the model needed in the variable `input_ids`, which the model used as its input.

Let's print `input_ids` to see what it holds inside:

```
tensor([[ 1, 14350, 385, 4876, 27746, 5281, 304, 19235, 363,
278, 25305, 293, 16423, 292, 286, 728, 481, 29889, 12027, 7420,
920, 372, 9559, 29889, 32001]], device='cuda:0')
```

This reveals the inputs that LLMs respond to, a series of integers as shown in [Figure 2-4](#). Each one is the unique ID for a specific token (character, word, or part of a word). These IDs reference a table inside the tokenizer containing all the tokens it knows.

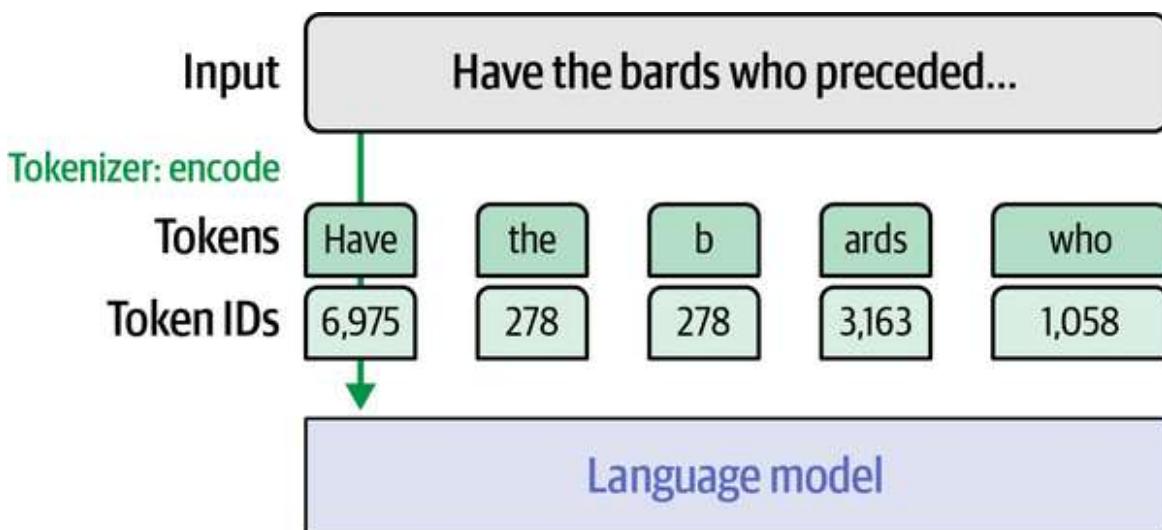


Figure 2-4. A tokenizer processes the input prompt and prepares the actual input into the language model: a list of token IDs. The specific token IDs in the figure are just demonstrative.

If we want to inspect those IDs, we can use the tokenizer's `decode` method to translate the IDs back into text that we can read:

```
for id in input_ids[0]:
    print(tokenizer.decode(id))
```

This prints (each token is on a separate line):

```
<s>  
Write  
an  
email  
apolog  
izing  
to  
Sarah  
for  
the  
trag  
ic  
garden  
ing  
m  
ish  
ap  
. .  
Exp  
lain  
how  
it
```

```
happened
```

```
.
```

```
<|assistant|>
```

This is how the tokenizer broke down our input prompt. Notice the following:

- The first token is ID 1 (<s>), a special token indicating the beginning of the text.
- Some tokens are complete words (e.g., *Write*, *an*, *email*).
- Some tokens are parts of words (e.g., *apolog*, *izing*, *trag*, *ic*).
- Punctuation characters are their own token.

Notice how the space character does not have its own token. Instead, partial tokens (like “izing” and “ic”) have a special hidden character at their beginning that indicates that they’re connected with the token that precedes them in the text. Tokens without that special character are assumed to have a space before them.

On the output side, we can also inspect the tokens generated by the model by printing the `generation_output` variable. This shows the input tokens as well as the output tokens (we’ll highlight the new tokens in bold):

```
tensor([[ 1, 14350, 385, 4876, 27746, 5281, 304, 19235, 363,
278,
25305, 293, 16423, 292, 286, 728, 481, 29889, 12027, 7420,
920, 372, 9559, 29889, 32001, 3323, 622, 29901, 1619, 317,
3742, 406, 6225, 11763, 363, 278, 19906, 292, 341, 728,
481, 13, 13, 29928, 799]], device='cuda:0')
```

This shows us the model generated the token 3323, 'Sub', followed by token 622, 'ject'. Together they formed the word 'Subject'. They were then followed by token 29901, which is the colon ':'...and so on. Just like on the input side, we need the tokenizer on the output side to translate the token ID into the actual text. We do that using the tokenizer's decode method. We can pass it an individual token ID or a list of them:

```
print(tokenizer.decode(3323))
print(tokenizer.decode(622))
print(tokenizer.decode([3323, 622]))
print(tokenizer.decode(29901))
```

This outputs:

```
Sub
ject
Subject
:
```

How Does the Tokenizer Break Down Text?

There are three major factors that dictate how a tokenizer breaks down an input prompt.

First, at model design time, the creator of the model chooses a tokenization method. Popular methods include byte pair encoding (BPE) (widely used by GPT models) and WordPiece (used by BERT). These methods are similar in that they aim to optimize an efficient set of tokens to represent a text dataset, but they arrive at it in different ways.

Second, after choosing the method, we need to make a number of tokenizer design choices like vocabulary size and what special tokens to use. More on this in "[Comparing Trained LLM Tokenizers](#)".

Third, the tokenizer needs to be trained on a specific dataset to establish the best vocabulary it can use to represent that dataset. Even if we set the same methods and parameters, a tokenizer trained on an English text dataset will be different from another trained on a code dataset or a multilingual text dataset.

In addition to being used to process the input text into a language model, tokenizers are used on the output of the language model to turn the resulting token ID into the output word or token associated with it, as [Figure 2-5](#) shows.

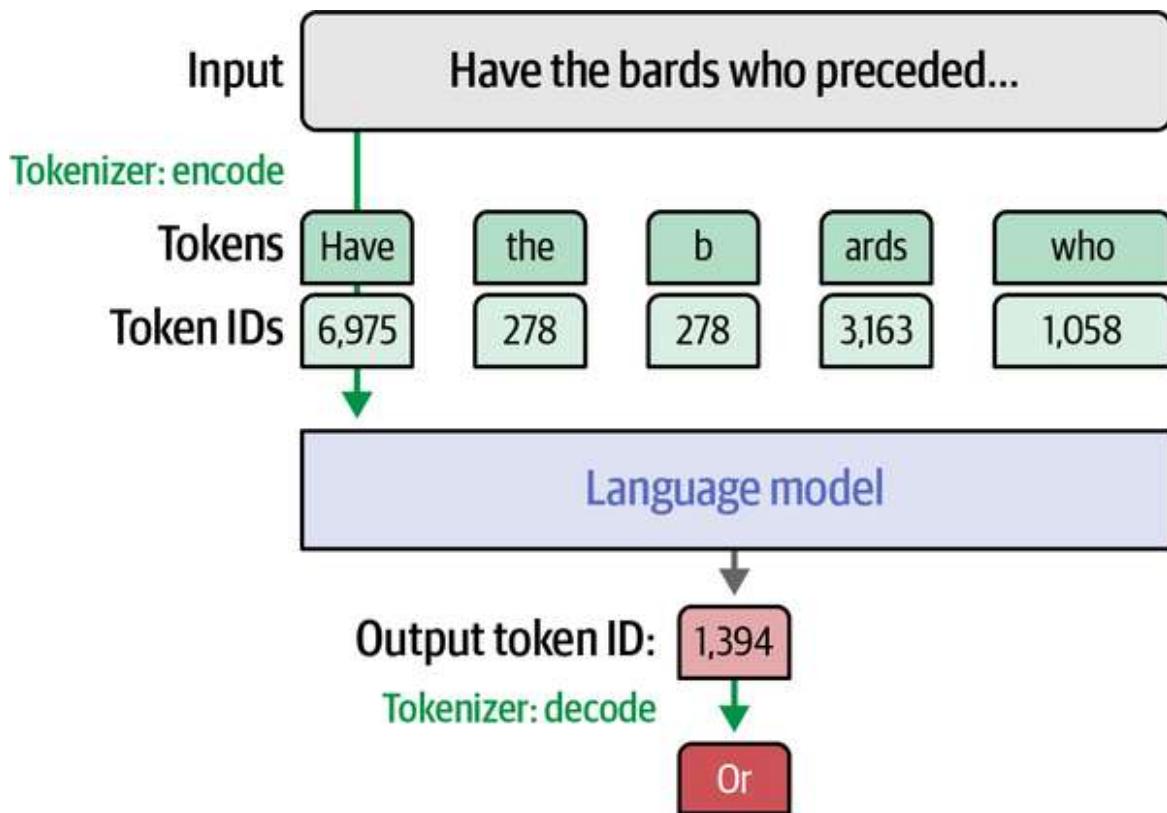


Figure 2-5. Tokenizers are also used to process the output of the model by converting the output token ID into the word or token associated with that ID.

Word Versus Subword Versus Character Versus Byte Tokens

The tokenization scheme we just discussed is called *subword tokenization*. It's the most commonly used tokenization scheme but not the only one. The

four notable ways to tokenize are shown in [Figure 2-6](#). Let's go over them:

Word tokens

This approach was common with earlier methods like word2vec but is being used less and less in NLP. Its usefulness, however, led it to be used outside of NLP for use cases such as recommendation systems, as we'll see later in the chapter.

One challenge with word tokenization is that the tokenizer may be unable to deal with new words that enter the dataset after the tokenizer was trained. This also results in a vocabulary that has a lot of tokens with minimal differences between them (e.g., apology, apologize, apologetic, apologist). This latter challenge is resolved by subword tokenization as it has a token for *apolog*, and then suffix tokens (e.g., *-y*, *-ize*, *-etic*, *-ist*) that are common with many other tokens, resulting in a more expressive vocabulary.

Subword tokens

This method contains full and partial words. In addition to the vocabulary expressivity mentioned earlier, another benefit of the approach is its ability to represent new words by breaking down the new token into smaller characters, which tend to be a part of the vocabulary.

Text Have the ♪ bards who preceded...

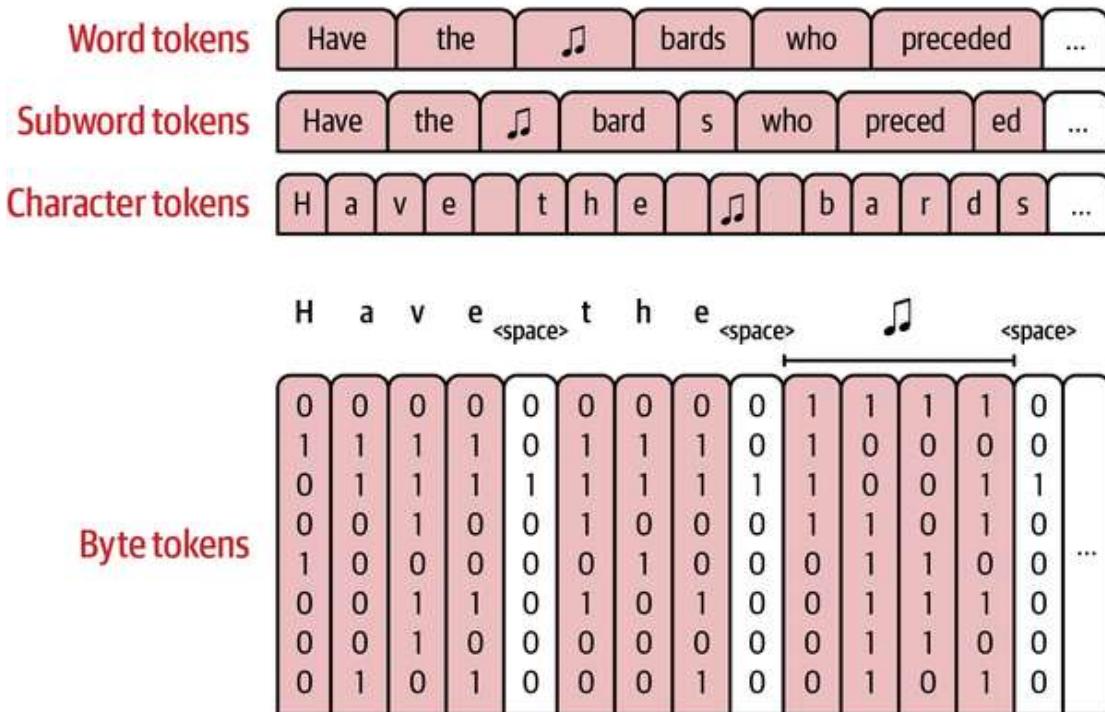


Figure 2-6. There are multiple methods of tokenization that break down the text to different sizes of components (words, subwords, characters, and bytes).

Character tokens

This is another method that can deal successfully with new words because it has the raw letters to fall back on. While that makes the representation easier to tokenize, it makes the modeling more difficult. Where a model with subword tokenization can represent “play” as one token, a model using character-level tokens needs to model the information to spell out “p-l-a-y” in addition to modeling the rest of the sequence.

Subword tokens present an advantage over character tokens in the ability to fit more text within the limited context length of a Transformer model. So with a model with a context length of 1,024, you may be able to fit about three times as much text using subword

tokenization than using character tokens (subword tokens often average three characters per token).

Byte tokens

One additional tokenization method breaks down tokens into the individual bytes that are used to represent unicode characters. Papers like “[CANINE: Pre-training an efficient tokenization-free encoder for language representation](#)” outline methods like this, which are also called “tokenization-free encoding.” Other works like “[ByT5: Towards a token-free future with pre-trained byte-to-byte models](#)” show that this can be a competitive method, especially in multilingual scenarios.

One distinction to highlight here: some subword tokenizers also include bytes as tokens in their vocabulary as the final building block to fall back to when they encounter characters they can’t otherwise represent. The GPT-2 and RoBERTa tokenizers do this, for example. This doesn’t make them tokenization-free byte-level tokenizers, because they don’t use these bytes to represent everything, only a subset, as we’ll see in the next section.

If you want to go deeper into tokenizers, they are discussed in more detail in [Designing Large Language Model Applications](#).

Comparing Trained LLM Tokenizers

We’ve pointed out earlier three major factors that dictate the tokens that appear within a tokenizer: the tokenization method, the parameters and special tokens we use to initialize the tokenizer, and the dataset the tokenizer is trained on. Let’s compare and contrast a number of actual, trained tokenizers to see how these choices change their behavior. This comparison will show us that newer tokenizers have changed their behavior to improve model performance, and we’ll also see how specialized models (like code generation models, for example) often need specialized tokenizers.

We'll use a number of tokenizers to encode the following text:

```
text = """  
English and CAPITALIZATION  
  
show_tokens False None elif == >= else: two tabs:" " Three tabs:  
" "  
12.0*50=600  
""
```

This will allow us to see how each tokenizer deals with a number of different kinds of tokens:

- Capitalization.
- Languages other than English.
- Emojis.
- Programming code with keywords and whitespaces often used for indentation (in languages like Python for example).
- Numbers and digits.
- Special tokens. These are unique tokens that have a role other than representing text. They include tokens that indicate the beginning of the text, or the end of the text (which is the way the model signals to the system that it has completed this generation), or other functions as we'll see.

Let's go from older to newer tokenizers to see how they tokenize this text and what that might say about the language model. We'll tokenize the text, and then print each token with a color background color using this function:

```
colors_list = [
```

```

        '102;194;165', '252;141;98', '141;160;203',
        '231;138;195', '166;216;84', '255;217;47'
    ]

def show_tokens(sentence, tokenizer_name):
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)
    token_ids = tokenizer(sentence).input_ids
    for idx, t in enumerate(token_ids):
        print(
            f'\x1b[0;30;48;2;{colors_list[idx % len(colors_list)]}m' +
            tokenizer.decode(t) +
            '\x1b[0m',
            end=' '
        )

```

BERT base model (uncased) (2018)

[Link to the model on the HuggingFace model hub](#)

Tokenization method: WordPiece, introduced in “[Japanese and Korean voice search](#)”:

Vocabulary size: 30,522

Special tokens:

unk_token [UNK]

An unknown token that the tokenizer has no specific encoding for.

sep_token [SEP]

A separator that enables certain tasks that require giving the model two texts (in these cases, the model is called a cross-encoder). One example is reranking, as we’ll see in [Chapter 8](#).

pad_token [PAD]

A padding token used to pad unused positions in the model’s input (as the model expects a certain length of input, its context-size).

cls_token [CLS]

A special classification token for classification tasks, as we'll see in [Chapter 4](#).

mask_token [MASK]

A masking token used to hide tokens during the training process.

Tokenized text:

```
[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 was released in two major flavors: cased (where the capitalization is kept) and uncased (where all capital letters are first turned into small cap letters). With the uncased (and more popular) version of the BERT tokenizer, we notice the following:

- The newline breaks are gone, which makes the model blind to information encoded in newlines (e.g., a chat log when each turn is in a new line).
- All the text is in lowercase.
- The word “capitalization” is encoded as two subtokens: `capital` `##ization`. The `##` characters are used to indicate this token is a partial token connected to the token that precedes it. This is also a method to indicate where the spaces are, as it is assumed tokens without `##` in front have a space before them.
- The emoji and Chinese characters are gone and replaced with the `[UNK]` special token indicating an “unknown token.”

BERT base model (cased) (2018)

[Link to the model on the HuggingFace model hub](#)

Tokenization method: WordPiece

Vocabulary size: 28,996

Special tokens: Same as the uncased version

Tokenized text:

```
[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]
```

The cased version of the BERT tokenizer differs mainly in including uppercase tokens.

- Notice how “CAPITALIZATION” is now represented as eight tokens: CA ##PI ##TA ##L ##I ##Z ##AT ##ION.
- Both BERT tokenizers wrap the input within a starting [CLS] token and a closing [SEP] token. [CLS] and [SEP] are utility tokens used to wrap the input text and they serve their own purposes. [CLS] stands for classification as it’s a token used at times for sentence classification. [SEP] stands for separator, as it’s used to separate sentences in some applications that require passing two sentences to a model (For example, in [Chapter 8](#), we will use a [SEP] token to separate the text of the query and a candidate result.)

GPT-2 (2019)

[Link to the model on the HuggingFace model hub](#)

Tokenization method: Byte pair encoding (BPE), introduced in “[Neural machine translation of rare words with subword units](#)”.

Vocabulary size: 50,257

Special tokens: <| endoftext |>

```
English and CAP ITAL IZ ATION
```

```
? ? ? ? ? ?
```

```
show tokens False None el if == >= else : two tabs : " "
Three tabs : " "
12 . 0 * 50 = 600
```

With the GPT-2 tokenizer, we notice the following:

- The newline breaks are represented in the tokenizer.
- Capitalization is preserved, and the word “CAPITALIZATION” is represented in four tokens.
- The characters are now represented by multiple tokens each.
While we see these tokens printed as the ☰ character, they actually stand for different tokens. For example, the emoji is broken down into the tokens with token IDs 8582, 236, and 113. The tokenizer is successful in reconstructing the original character from these tokens. We can see that by printing `tokenizer.decode([8582, 236, 113])`, which prints out .
- The two tabs are represented as two tokens (token number 197 in that vocabulary) and the four spaces are represented as three tokens (number 220) with the final space being a part of the token for the closing quote character.
- The two tabs are represented as two tokens (token number 197 in that vocabulary) and the four spaces are represented as three tokens (number 220) with the final space being a part of the token for the closing quote character.

NOTE

What is the significance of whitespace characters? These are important for models to understand or generate code. A model that uses a single token to represent four consecutive whitespace characters is more tuned to a Python code dataset. While a model can live with representing it as four different tokens, it does make the modeling more difficult as the model needs to keep track of the indentation level, which often leads to worse performance. This is an example of where tokenization choices can help the model improve on a certain task.

Flan-T5 (2022)

Tokenization method: Flan-T5 uses a tokenizer implementation called SentencePiece, introduced in “[SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing](#)”, which supports BPE and the *unigram language model* (described in “[Subword regularization: Improving neural network translation models with multiple subword candidates](#)”).

Vocabulary size: 32,100

Special tokens:

- unk_token <unk>
- pad_token <pad>

Tokenized text:

The image shows a horizontal sequence of colored tokens representing the tokenized text. The tokens are color-coded: English, and, C, A, P, I, T, A, L, I, Z, A, T, I, O, N, <unk>, <unk>, s, h, o, w, _, t, a, b, s, t, o, k, e, n, s, F, a, l, s, e, N, o, n, e, e, l, i, f, =, =, >, =, e, l, s, e, :, t, w, o, t, a, b, s, :, " , " , T, h, r, e, e, t, a, b, s, :, " , " , 1, 2, ., 0, *, 5, 0, =, 6, 0, 0, </s>. The colors used include various shades of blue, green, yellow, red, purple, and brown.

The Flan-T5 family of models use the SentencePiece method. We notice the following:

- No newline or whitespace tokens; this would make it challenging for the model to work with code.

- The emoji and Chinese characters are both replaced by the <unk> token, making the model completely blind to them.

GPT-4 (2023)

Tokenization method: BPE

Vocabulary size: A little over 100,000

Special tokens:

- <|endoftext|>
- Fill in the middle tokens. These three tokens enable the LLM to generate a completion given not only the text before it but also considering the text after it. This method is explained in more detail in the paper “[Efficient training of language models to fill in the middle](#)”; its exact details are beyond the scope of this book.
These special tokens are:

- <|fim_prefix|>
- <|fim_middle|>
- <|fim_suffix|>

Tokenized text:

English and CAPITAL IZATION

?????????

show_tokens False None elif == >= else : two tabs :" "

Three tabs : " "

12 . 0 * 50 = 600

The GPT-4 tokenizer behaves similarly to its ancestor, the GPT-2 tokenizer. Some differences are:

- The GPT-4 tokenizer represents the four spaces as a single token. In fact, it has a specific token for every sequence of whitespaces up

to a list of 83 whitespaces.

- The Python keyword `elif` has its own token in GPT-4. Both this and the previous point stem from the model's focus on code in addition to natural language.
- The GPT-4 tokenizer uses fewer tokens to represent most words. Examples here include “CAPITALIZATION” (two tokens versus four) and “tokens” (one token versus three).
- Refer back to what we said about the GPT-2 tokenizer with regards to the `L` tokens.

StarCoder2 (2024)

StarCoder2 is a 15-billion parameter model focused on generating code described in the paper “StarCoder 2 and the stack v2: The next generation”, which continues the work from the original StarCoder described in “StarCoder: May the source be with you!”.

Tokenization method: Byte pair encoding (BPE)

Vocabulary size: 49,152

Example special tokens:

- `<|endoftext|>`
- Fill in the middle tokens:
 - `<fim_prefix>`
 - `<fim_middle>`
 - `<fim_suffix>`
 - `<fim_pad>`
- When representing code, managing the context is important. One file might make a function call to a function that is defined in a different file. So the model needs some way of being able to

identify code that is in different files in the same code repository, while making a distinction between code in different repos. That's why StarCoder2 uses special tokens for the name of the repository and the filename:

- <filename>
- <reponame>
- <gh_stars>

Tokenized text:

English and CAPITAL IZATION

(show tokens False elif == >= else : two tabs : ")

Three tabs : ")

1 2 . 0 * 5 0 = 6 0 0

This is an encoder that focuses on code generation:

- Similar to GPT-4, it encodes the list of whitespaces as a single token.
- A major difference here to everything we've seen so far is that each digit is assigned its own token (so 600 becomes 6 0 0). The hypothesis here is that this would lead to better representation of numbers and mathematics. In GPT-2, for example, the number 870 is represented as a single token. But 871 is represented as two tokens (8 and 71). You can intuitively see how that might be confusing to the model and how it represents numbers.

Galactica

The **Galactica model** described in “**Galactica: A large language model for science**” is focused on scientific knowledge and is trained on many scientific papers, reference materials, and knowledge bases. It pays extra

attention to tokenization that makes it more sensitive to the nuances of the dataset it's representing. For example, it includes special tokens for citations, reasoning, mathematics, amino acid sequences, and DNA sequences.

Tokenization method: Byte pair encoding (BPE)

Vocabulary size: 50,000

Special tokens:

- <s>
- <pad>
- </s>
- <unk>
- References: Citations are wrapped within the two special tokens:
 - [START_REF]
 - [END_REF]
 - One example of usage from the paper is: Recurrent neural networks, long short-term memory [START_REF]Long Short-Term Memory, Hochreiter[END_REF]
- Step-by-step reasoning:
 - <work> is an interesting token that the model uses for chain-of-thought reasoning.

Tokenized text:

English and CAP ITAL IZATION



```

show _ tokens False None elif == > = else : two t abs : " "
Three t abs : " "
1 2 . 0 * 5 0 = 6 0 0

```

The Galactica tokenizer behaves similar to StarCoder2 in that it has code in mind. It also encodes whitespaces in the same way: assigning a single token to sequences of whitespace of different lengths. It differs in that it also does that for tabs, though. So from all the tokenizers we've seen so far, it's the only one that assigns a single token to the string made up of two tabs ('\t\t').

Phi-3 (and Llama 2)

The **Phi-3 model** we look at in this book reuses the tokenizer of **Llama 2** yet adds a number of special tokens.

Tokenization method: Byte pair encoding (BPE)

Vocabulary size: 32,000

Special tokens:

- <|endoftext|>
- Chat tokens: As chat LLMs rose to popularity in 2023, the conversational nature of LLMs started to be a leading use case. Tokenizers have been adapted to this direction by the addition of tokens that indicate the turns in a conversation and the roles of each speaker. These special tokens include:
 - <|user|>
 - <|assistant|>
 - <|system|>

We can now recap our tour by looking at all these examples side by side:

BERT base
model (uncased)

```
[CLS] english and capital ##ization [UNK] [UNK] sh
ow _ token ##s false none eli ##f = > = else : two
tab ##s : " " three tab ##s : " " 12 . 0 * 50 = 600 [SE
P]
```

BERT base
model (cased)

```
[CLS] English and CA ##PI ##TA ##L ##I ##Z ##AT ##I
ON [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
? ? ? ? ? ?
show _ tokens False None el if == > = else : two tab
s : " " 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 tabs s : " "
Three tabs s : " " 12 . 0 * 50 = 600 </s>
```

GPT-4

```
English and CAPITAL IZATION
? ? ? ? ? ?
show _tokens False None el if == > = else : two tabs
: " " Three tabs : " "
12 . 0 * 50 = 600
```

StarCoder

```
English and CAPITAL IZATION
? ? ? ? ?
show _tokens False None el if == > = else : two tabs
: " " Three tabs : " "
```

```
1 2 . 0 * 5 0 = 6 0 0
```

Galactica	English and CAP ITAL IZATION ⌚⌚⌚⌚⌚⌚⌚⌚ show _ tokens False None elif == > = else : two t abs : " " Three tabs : " " 1 2 . 0 * 5 0 = 6 0 0
-----------	--

Phi-3 and Llama 2	<s> English and C AP IT AL IZ ATION ⌚⌚⌚⌚⌚⌚⌚⌚ show _ to kens False None elif == >= else : two tabs : " " Three tabs : " " 1 2 . 0 * 5 0 = 6 0 0
-------------------	---

Tokenizer Properties

The preceding guided tour of trained tokenizers showed a number of ways in which actual tokenizers differ from each other. But what determines their tokenization behavior? There are three major groups of design choices that determine how the tokenizer will break down text: the tokenization method, the initialization parameters, and the domain of the data the tokenizer targets.

Tokenization methods

As we've seen, there are a number of tokenization methods with byte pair encoding (BPE) being the more popular one. Each of these methods outlines an algorithm for how to choose an appropriate set of tokens to represent a dataset. You can find a great overview of all these methods on the Hugging Face [page that summarizes tokenizers](#).

Tokenizer parameters

After choosing a tokenization method, an LLM designer needs to make some decisions about the parameters of the tokenizer. These include:

Vocabulary size

How many tokens to keep in the tokenizer's vocabulary? (30K and 50K are often used as vocabulary size values, but more and more we're seeing larger sizes like 100K.)

Special tokens

What special tokens do we want the model to keep track of? We can add as many of these as we want, especially if we want to build an LLM for special use cases. Common choices include:

- Beginning of text token (e.g., <s>)
- End of text token
- Padding token
- Unknown token
- CLS token
- Masking token

Aside from these, the LLM designer can add tokens that help better model the domain of the problem they're trying to focus on, as we've seen with Galactica's `<work>` and `[START_REF]` tokens.

Capitalization

In languages such as English, how do we want to deal with capitalization? Should we convert everything to lowercase? (Name capitalization often carries useful information, but do we want to waste token vocabulary space on all-caps versions of words?)

The domain of the data

Even if we select the same method and parameters, tokenizer behavior will be different based on the dataset it was trained on (before we even start model training). The tokenization methods mentioned previously work by optimizing the vocabulary to represent a specific dataset. From our guided tour we've seen how that has an impact on datasets like code and multilingual text.

For code, for example, we've seen that a text-focused tokenizer may tokenize the indentation spaces like this (we'll highlight some tokens in color):

```
def add_numbers(a, b):  
    """Add the two numbers `a` and `b`."""  
    return a + b
```

This may be suboptimal for a code-focused model. Code-focused models are often improved by making different tokenization choices:

```
def add_numbers(a, b):  
    """Add the two numbers `a` and `b`."""  
    return a + b
```

These tokenization choices make the model's job easier and thus its performance has a higher probability of improving.

You can find a more detailed tutorial on training tokenizers in the [Tokenizers section of the Hugging Face course](#) and in *Natural Language Processing with Transformers, Revised Edition*.

Token Embeddings

Now that we understand tokenization, we have solved one part of the problem of representing language to a language model. In this sense, language is a sequence of tokens. And if we train a good-enough model on a large-enough set of tokens, it starts to capture the complex patterns that appear in its training dataset:

- If the training data contains a lot of English text, that pattern reveals itself as a model capable of representing and generating the English language.
- If the training data contains factual information (Wikipedia, for example), the model would have the ability to generate some factual information (see the following note).

The next piece of the puzzle is finding the best numerical representation for these tokens that the model can use to calculate and properly model the patterns in the text. These patterns reveal themselves to us as a model's coherence in a specific language, or capability to code, or any of the growing list of capabilities we expect from language models.

As we've seen in [Chapter 1](#), that is what embeddings are. They are the numeric representation space utilized to capture the meanings and patterns in language.

NOTE

Oops: Achieving a good threshold of language coherence and better-than-average factual generation, however, starts to present a new problem. Some users start to trust the model's fact generation ability (e.g., at the beginning of 2023 some language models were being dubbed “[Google killers](#)”). It didn't take long for advanced users to recognize that generation models alone aren't reliable search engines. This led to the rise of retrieval-augmented generation (RAG), which combines search and LLMs. We cover RAG in more detail in [Chapter 8](#).

A Language Model Holds Embeddings for the Vocabulary of Its Tokenizer

After a tokenizer is initialized and trained, it is then used in the training process of its associated language model. This is why a pretrained language model is linked with its tokenizer and can't use a different tokenizer without training.

The language model holds an embedding vector for each token in the tokenizer's vocabulary, as we can see in [Figure 2-7](#). When we download a pretrained language model, a portion of the model is this embeddings matrix holding all of these vectors.

Before the beginning of the training process, these vectors are randomly initialized like the rest of the model's weights, but the training process assigns them the values that enable the useful behavior they're trained to perform.

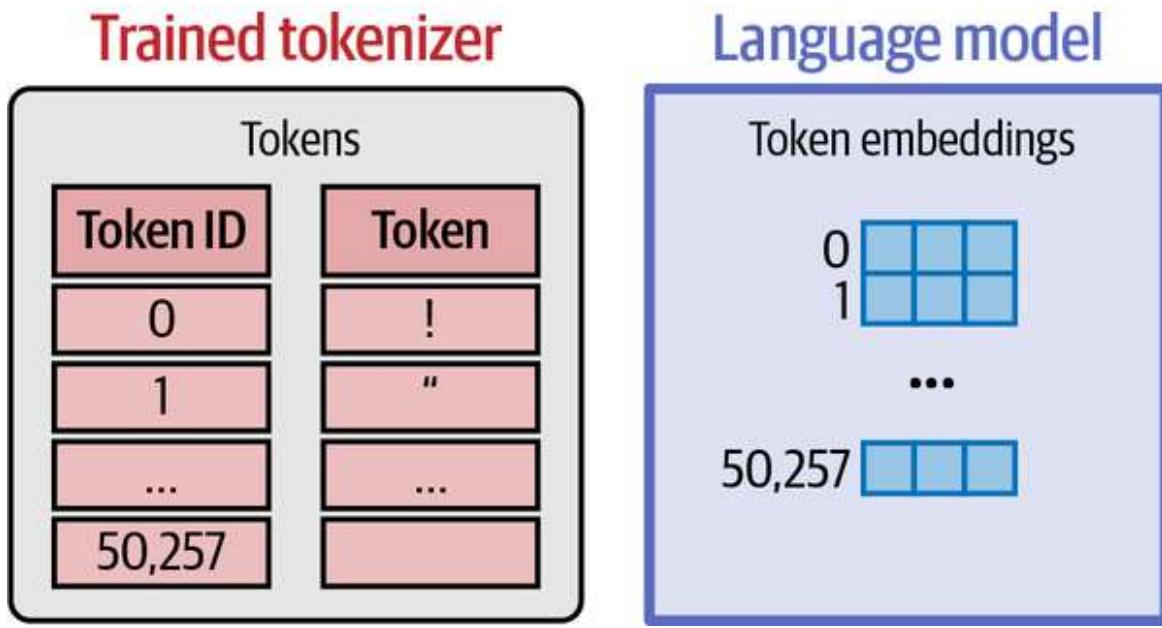


Figure 2-7. A language model holds an embedding vector associated with each token in its tokenizer.

Creating Contextualized Word Embeddings with Language Models

Now that we've covered token embeddings as the input to a language model, let's look at how language models can *create* better token embeddings. This is one of the primary ways to use language models for text representation. This empowers applications like named-entity recognition or extractive text summarization (which summarizes a long text by highlighting the most important parts of it, instead of generating new text as a summary).

Instead of representing each token or word with a static vector, language models create contextualized word embeddings (shown in [Figure 2-8](#)) that represent a word with a different token based on its context. These vectors can then be used by other systems for a variety of tasks. In addition to the text applications we mentioned in the previous paragraph, these contextualized vectors, for example, are what powers AI image generation systems like DALL·E, Midjourney, and Stable Diffusion, for example.

Have the bards who preceded me left any theme unsung?

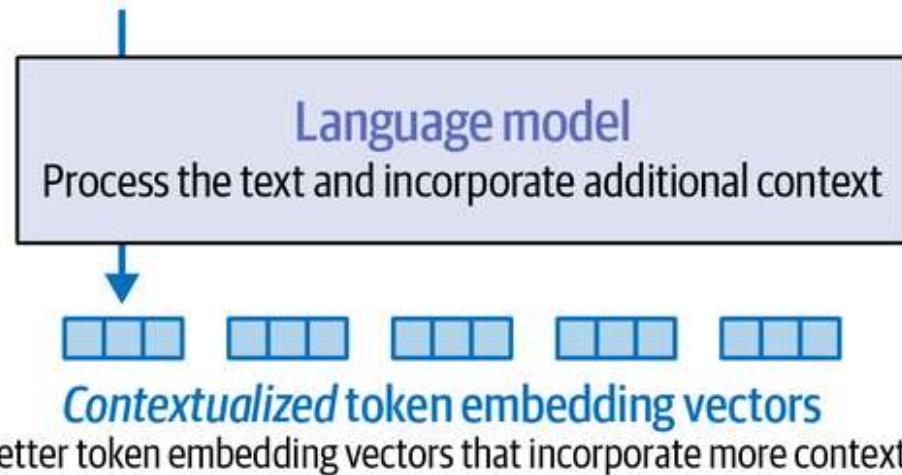


Figure 2-8. Language models produce contextualized token embeddings that improve on raw, static token embeddings.

Let's look at how we can generate contextualized word embeddings; the majority of this code should be familiar to you by now:

```
from transformers import AutoModel, AutoTokenizer

# Load a tokenizer
tokenizer = AutoTokenizer.from_pretrained("microsoft/deberta-base")

# Load a language model
model = AutoModel.from_pretrained("microsoft/deberta-v3-xsmall")

# Tokenize the sentence
tokens = tokenizer('Hello world', return_tensors='pt')

# Process the tokens
output = model(**tokens)[0]
```

The model we're using here is called DeBERTa v3, which at the time of writing is one of the best-performing language models for token embeddings while being small and highly efficient. It is described in the paper [“DeBERTaV3: Improving DeBERTa using ELECTRA-style pre-training gradient-disentangled embedding sharing”](#).

This code downloads a pretrained tokenizer and model, then uses them to process the string “Hello world”. The output of the model is then saved in the output variable. Let’s inspect that variable by first printing its dimensions (we expect it to be a multidimensional array):

```
output.shape
```

This prints out:

```
torch.Size([1, 4, 384])
```

Skipping the first dimension, we can read this as four tokens, each one embedded in a vector of 384 values. The first dimension is the batch dimension used in cases (like training) when we want to send multiple input sentences to the model at the same time (they’re processed at the same time, which speeds up the process).

But what are these four vectors? Did the tokenizer break the two words into four tokens, or is something else happening here? We can use what we’ve learned about tokenizers to inspect them:

```
for token in tokens['input_ids'][0]:  
    print(tokenizer.decode(token))
```

This prints out:

```
[CLS]  
Hello  
world  
[SEP]
```

This particular tokenizer and model operate by adding the [CLS] and [SEP] tokens to the beginning and end of a string.

Our language model has now processed the text input. The result of its output is the following:

```
tensor([[  
    [-3.3060, -0.0507, -0.1098, ..., -0.1704, -0.1618, 0.6932],  
    [ 0.8918, 0.0740, -0.1583, ..., 0.1869, 1.4760, 0.0751],  
    [ 0.0871, 0.6364, -0.3050, ..., 0.4729, -0.1829, 1.0157],  
    [-3.1624, -0.1436, -0.0941, ..., -0.0290, -0.1265, 0.7954]  
]], grad_fn=<NativeLayerNormBackward0>)
```

This is the raw output of a language model. The applications of large language models build on top of outputs like this.

We recap the input tokenization and resulting outputs of a language model in [Figure 2-9](#). Technically, the switch from token IDs into raw embeddings is the first step that occurs inside a language model.

Have the bards who preceded me left any theme unsung?

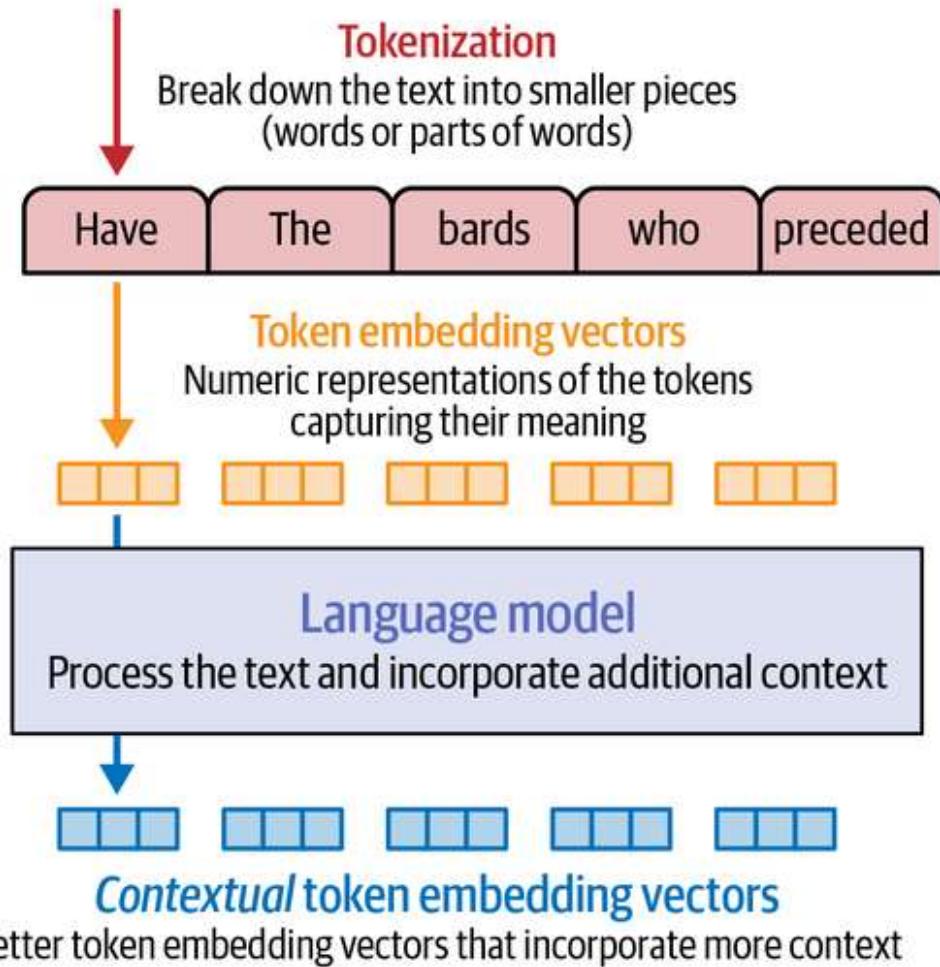


Figure 2-9. A language model operates on raw, static embeddings as its input and produces contextual text embeddings.

A visual like this is essential for the next chapter when we start to look at how Transformer-based LLMs work.

Text Embeddings (for Sentences and Whole Documents)

While token embeddings are key to how LLMs operate, a number of LLM applications require operating on entire sentences, paragraphs, or even text documents. This has led to special language models that produce text

embeddings—a single vector that represents a piece of text longer than just one token.

We can think of text embedding models as taking a piece of text and ultimately producing a single vector that represents that text and captures its meaning in some useful form. [Figure 2-10](#) shows that process.

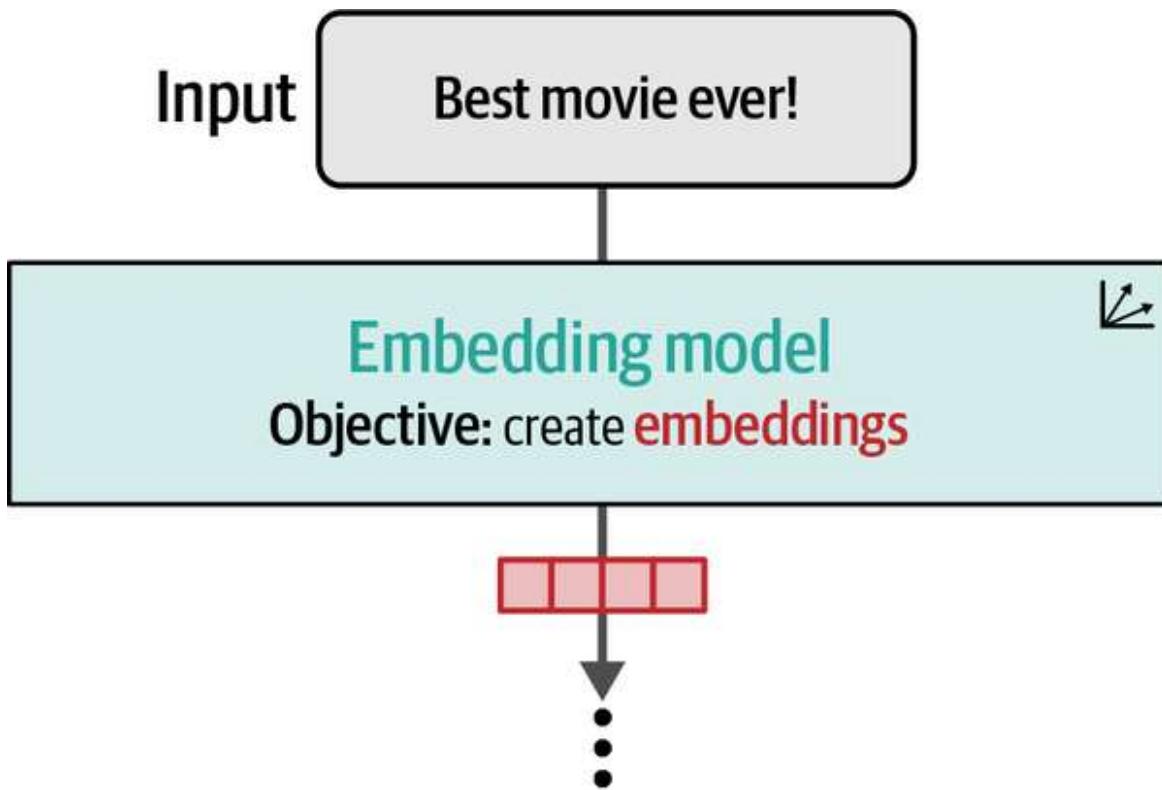


Figure 2-10. In step 1, we use the embedding model to extract the features and convert the input text to embeddings.

There are multiple ways of producing a *text* embedding vector. One of the most common ways is to average the values of all the *token* embeddings produced by the model. Yet high-quality text embedding models tend to be trained specifically for text embedding tasks.

We can produce text embeddings with [sentence-transformers](#), a popular package for leveraging pretrained embedding models.¹ The package, like `transformers` in the previous chapter, can be used to load publicly available models. To illustrate creating embeddings, we use the [all-](#)

`mpnet-base-v2` model. Note that in [Chapter 4](#), we will further explore how you can choose an embedding model for your task.

```
from sentence_transformers import SentenceTransformer

# Load model
model = SentenceTransformer("sentence-transformers/all-mpnet-
base-v2")

# Convert text to text embeddings
vector = model.encode("Best movie ever!")
```

The number of values, or the dimensions, of the embedding vector depends on the underlying embedding model. Let's explore that for our model:

```
vector.shape
```

```
(768, )
```

This sentence is now encoded in this one vector with a dimension of 768 numerical values. In Part II of this book, once we start looking at applications, we'll start to see the immense usefulness of these text embeddings vectors in powering everything from categorization to semantic search to RAG.

Word Embeddings Beyond LLMs

Embeddings are useful even outside of text and language generation. Embeddings, or assigning meaningful vector representations to objects, turns out to be useful in many domains, including recommender engines and robotics. In this section, we'll look at how to use pretrained word2vec embeddings and touch on how the method creates word embeddings. Seeing how word2vec is trained will prime you to learn about contrastive training in [Chapter 10](#). Then in the following section, we'll see how those embeddings can be used for recommendation systems.

Using pretrained Word Embeddings

Let's look at how we can download pretrained word embeddings (like word2vec or GloVe) using the **Gensim library**:

```
import gensim.downloader as api

# Download embeddings (66MB, glove, trained on wikipedia, vector
size: 50)
# Other options include "word2vec-google-news-300"
# More options at https://github.com/RaRe-Technologies/gensim-
# data
model = api.load("glove-wiki-gigaword-50")
```

Here, we've downloaded the embeddings of a large number of words trained on Wikipedia. We can then explore the embedding space by seeing the nearest neighbors of a specific word, "king" for example:

```
model.most_similar([model['king']], topn=11)
```

This outputs:

```
[('king', 1.0000001192092896),
 ('prince', 0.8236179351806641),
 ('queen', 0.7839043140411377),
 ('ii', 0.7746230363845825),
 ('emperor', 0.7736247777938843),
 ('son', 0.766719400882721),
 ('uncle', 0.7627150416374207),
 ('kingdom', 0.7542161345481873),
 ('throne', 0.7539914846420288),
 ('brother', 0.7492411136627197),
 ('ruler', 0.7434253692626953)]
```

The Word2vec Algorithm and Contrastive Training

The word2vec algorithm described in the paper "**Efficient estimation of word representations in vector space**" is described in detail in [The](#)

Illustrated Word2vec. The central ideas are condensed here as we build on them when discussing one method for creating embeddings for recommendation engines in the following section.

Just like LLMs, word2vec is trained on examples generated from text. Let's say, for example, we have the text "Thou shalt not make a machine in the likeness of a human mind" from the *Dune* novels by Frank Herbert. The algorithm uses a sliding window to generate training examples. We can, for example, have a window size two, meaning that we consider two neighbors on each side of a central word.

The embeddings are generated from a classification task. This task is used to train a neural network to predict if words commonly appear in the same context or not (*context* here means in many sentences in the training dataset we're modeling). We can think of this as a neural network that takes two words and outputs 1 if they tend to appear in the same context, and 0 if they do not.

In the first position for the sliding window, we can generate four training examples, as we can see in [Figure 2-11](#).

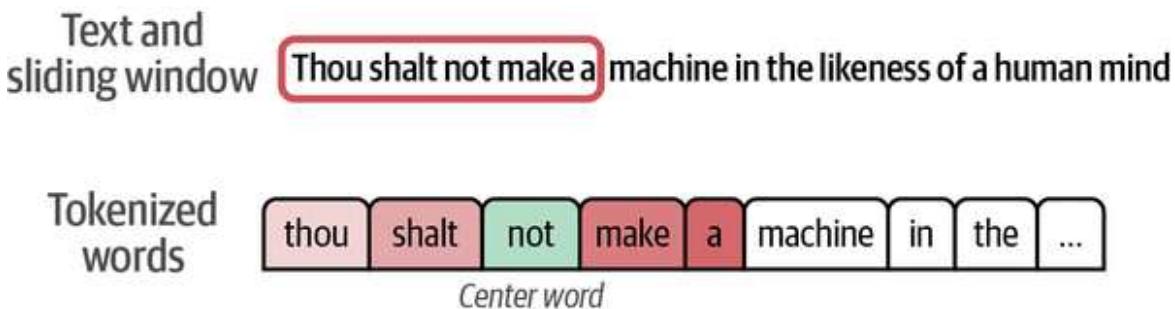


Figure 2-11. A sliding window is used to generate training examples for the word2vec algorithm to later predict if two words are neighbors or not.

In each of the produced training examples, the word in the center is used as one input, and each of its neighbors is a distinct second input in each training example. We expect the final trained model to be able to classify this neighbor relationship and output 1 if the two input words it receives are indeed neighbors. These training examples are visualized in [Figure 2-12](#).

Training examples

Word 1	Word 2	Target
Not	thou	1
Not	shalt	1
Not	make	1
Not	a	1

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

If, however, we have a dataset of only a target value of 1, then a model can cheat and ace it by outputting 1 all the time. To get around this, we need to enrich our training dataset with examples of words that are not typically neighbors. These are called negative examples and are shown in Figure 2-13.

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

Positive examples

Negative examples

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.

It turns out that we don't have to be too scientific in how we choose the negative examples. A lot of useful models result from the simple ability to detect positive examples from randomly generated examples (inspired by an

important idea called *noise-contrastive estimation* and described in “[Noise-contrastive estimation: A new estimation principle for unnormalized statistical models](#)”). So in this case, we get random words and add them to the dataset and indicate that they are not neighbors (and thus the model should output 0 when it sees them).

With this, we’ve seen two of the main concepts of word2vec ([Figure 2-14](#)): skip-gram, the method of selecting neighboring words, and negative sampling, adding negative examples by random sampling from the dataset.

The diagram illustrates the Skip-gram and Negative sampling components of the word2vec algorithm.

Skip-gram:

shalt	not	make	a	machine
input		output		
make		shalt		
make		not		
make		a		
make		machine		

Negative sampling:

Input word	Output word	Target
make	shalt	1
make	aaron	0
make	taco	0

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.

We can generate millions and even billions of training examples like this from running text. Before proceeding to train a neural network on this dataset, we need to make a couple of tokenization decisions, which, just like we’ve seen with LLM tokenizers, include how to deal with capitalization and punctuation and how many tokens we want in our vocabulary.

We then create an embedding vector for each token, and randomly initialize them, as can be seen in [Figure 2-15](#). In practice, this is a matrix of dimensions `vocab_size x embedding_dimensions`.

Token	Token embedding
thou	
shalt	
make	
a	
not	
apothecary	
sublime	
def	
playback	

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

A model is then trained on each example to take in two embedding vectors and predict if they're related or not. We can see what this looks like in [Figure 2-16](#).

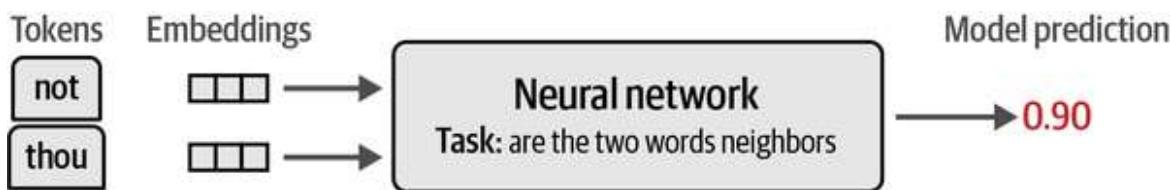


Figure 2-16. A neural network is trained to predict if two words are neighbors. It updates the embeddings in the training process to produce the final, trained embeddings.

Based on whether its prediction was correct or not, the typical machine learning training step updates the embeddings so that the next time the model is presented with those two vectors, it has a better chance of being

more correct. And by the end of the training process, we have better embeddings for all the tokens in our vocabulary.

This idea of a model that takes two vectors and predicts if they have a certain relation is one of the most powerful ideas in machine learning, and time after time has proven to work very well with language models. This is why we're dedicating [Chapter 10](#) to this concept and how it optimizes language models for specific tasks (like sentence embeddings and retrieval).

The same idea is also central to bridging modalities like text and images, which is key to AI image generation models, as we'll see in [Chapter 9](#) on multimodal models. In that formulation, a model is presented with an image and a caption, and it should predict whether that caption describes the image or not.

Embeddings for Recommendation Systems

As we've mentioned, the concept of embeddings is useful in so many other domains. In industry, it's widely used for recommendation systems, for example.

Recommend Songs by Embeddings

In this section we'll use the word2vec algorithm to embed songs using human-made music playlists. Imagine if we treated each song as we would a word or token, and we treated each playlist like a sentence. These embeddings can then be used to recommend similar songs that often appear together in playlists.

The [dataset](#) we'll use was collected by Shuo Chen from Cornell University. It contains playlists from hundreds of radio stations around the US.

[Figure 2-17](#) demonstrates this dataset.

Playlist #1:	Song 1	Song 13	Song 2	Song 400
Playlist #2:	Song 2	Song 81	Song 13	Song 82
Playlist #3:	Song 13	Song 2		

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.

Let's demonstrate the end product before we look at how it's built. So let's give it a few songs and see what it recommends in response.

Let's start by giving it Michael Jackson's "Billie Jean," the song with ID 3822:

```
# We will define and explore this function in detail below
print_recommendations(3822)
```

id	Title	artist
4181	Kiss	Prince & The Revolution
12749	Wanna Be Startin' Somethin'	Michael Jackson
1506	The Way You Make Me Feel	Michael Jackson
3396	Holiday	Madonna
500	Don't Stop 'Til You Get Enough	Michael Jackson

That looks reasonable. Madonna, Prince, and other Michael Jackson songs are the nearest neighbors.

Let's step away from pop and into rap, and see the neighbors of 2Pac's "California Love":

```
print_recommendations(842)
```

id	Title	artist
413	If I Ruled the World (Imagine That) (w\ Nas V Laury Hill)	Nas
196	I'll Be Missing You	Puff Daddy & The Family
330	Hate It or Love It (w\ 50 Cent)	The Game
211	Hypnotize	The Notorious B.I.G.
5788	Drop It Like It's Hot (w\ Pharrell)	Snoop Dogg

Another quite reasonable list! Now that we know it works, let's see how to build such a system.

Training a Song Embedding Model

We'll start by loading the dataset containing the song playlists as well as each song's metadata, such as its title and artist:

```

import pandas as pd
from urllib import request

# Get the playlist dataset file
data = request.urlopen('https://storage.googleapis.com/maps-premium/dataset/yes_complete/train.txt')

# Parse the playlist dataset file. Skip the first two lines as
# they only contain metadata
lines = data.read().decode("utf-8").split('\n')[2:]

# Remove playlists with only one song
playlists = [s.rstrip().split() for s in lines if len(s.split()) > 1]

# Load song metadata
songs_file =
request.urlopen('https://storage.googleapis.com/maps-premium/dataset/yes_complete/song_hash.txt')
songs_file = songs_file.read().decode("utf-8").split('\n')
songs = [s.rstrip().split('\t') for s in songs_file]
songs_df = pd.DataFrame(data=songs, columns = ['id', 'title', 'artist'])
songs_df = songs_df.set_index('id')

```

Now that we've saved them, let's inspect the `playlists` list. Each element inside it is a playlist containing a list of song IDs:

```

print( 'Playlist #1:\n ', playlists[0], '\n')
print( 'Playlist #2:\n ', playlists[1])

Playlist #1: ['0', '1', '2', '3', '4', '5', ..., '43']
Playlist #2: ['78', '79', '80', '3', '62', ..., '210']

```

Let's train the model:

```

from gensim.models import Word2Vec

# Train our Word2Vec model
model = Word2Vec(
    playlists, vector_size=32, window=20, negative=50,

```

```
    min_count=1, workers=4
)
```

That takes a minute or two to train and results in embeddings being calculated for each song that we have. Now we can use those embeddings to find similar songs exactly as we did earlier with words:

```
song_id = 2172

# Ask the model for songs similar to song #2172
model.wv.most_similar(positive=str(song_id))
```

This outputs:

```
[('2976', 0.9977465271949768),
 ('3167', 0.9977430701255798),
 ('3094', 0.9975950717926025),
 ('2640', 0.9966474175453186),
 ('2849', 0.9963167905807495)]
```

That is the list of the songs whose embeddings are most similar to song 2172.

In this case, the song is:

```
print(songs_df.iloc[2172])

title Fade To Black
artist Metallica
Name: 2172 , dtype: object
```

This results in recommendations that are all in the same heavy metal and hard rock genre:

```
import numpy as np
```

```

def print_recommendations(song_id):
    similar_songs = np.array(
        model.wv.most_similar(positive=str(song_id), topn=5)
    )[:, 0]
    return songs_df.iloc[similar_songs]

# Extract recommendations
print_recommendations(2172)

```

id	Title	artist
11473	Little Guitars	Van Halen
3167	Unchained	Van Halen
5586	The Last in Line	Dio
5634	Mr. Brownstone	Guns N' Roses
3094	Breaking the Law	Judas Priest

Summary

In this chapter, we have covered LLM tokens, tokenizers, and useful approaches to using token embeddings. This prepares us to start looking closer at language models in the next chapter, and also opens the door to learn about how embeddings are used beyond language models.

We explored how tokenizers are the first step in processing input to an LLM, transforming raw textual input into token IDs. Common tokenization schemes include breaking text down into words, subword tokens, characters, or bytes, depending on the specific requirements of a given application.

A tour of real-world pretrained tokenizers (from BERT to GPT-2, GPT-4, and other models) showed us areas where some tokenizers are better (e.g., preserving information like capitalization, newlines, or tokens in other languages) and other areas where tokenizers are just different from each other (e.g., how they break down certain words).

Three of the major tokenizer design decisions are the tokenizer algorithm (e.g., BPE, WordPiece, SentencePiece), tokenization parameters (including vocabulary size, special tokens, capitalization, treatment of capitalization and different languages), and the dataset the tokenizer is trained on.

Language models are also creators of high-quality contextualized token embeddings that improve on raw static embeddings. Those contextualized token embeddings are what's used for tasks including named-entity recognition (NER), extractive text summarization, and text classification. In addition to producing token embeddings, language models can produce text embeddings that cover entire sentences or even documents. This empowers plenty of applications that will be shown in Part II of this book covering language model applications

Before LLMs, word embedding methods like word2vec, GloVe, and fastText were popular. In language processing, this has largely been replaced with contextualized word embeddings produced by language models. The word2vec algorithm relies on two main ideas: skip-gram and negative sampling. It also uses contrastive training similar to the type we'll see in [Chapter 10](#).

Embeddings are useful for creating and improving recommender systems as we discussed in the music recommender we built from curated song playlists.

In the next chapter, we will take a deep dive into the process after tokenization: how does an LLM process these tokens and generate text? We will look at some of the main intuitions of how LLMs that use the Transformer architecture work.

-
- ¹ Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence embeddings using Siamese BERT-networks.” *arXiv preprint arXiv:1908.10084* (2019).

Chapter 3. Looking Inside Large Language Models

Now that we have a sense of tokenization and embeddings, we're ready to dive deeper into the language model and see how it works. In this chapter, we'll look at some of the main intuitions of how Transformer language models work. Our focus will be on text generation models so we get a deeper sense for generative LLMs in particular.

We'll be looking at both the concepts and some code examples that demonstrate them. Let's start by loading a language model and getting it ready for generation by declaring a pipeline. In your first read, feel free to skip the code and focus on grasping the concepts involved. Then in a second read, the code will get you to start applying these concepts.

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer,
pipeline

# Load model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")

model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)

# Create a pipeline
generator = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
```

```
    max_new_tokens=50,  
    do_sample=False,  
)
```

An Overview of Transformer Models

Let's begin our exploration with a high-level overview of the model, and then we'll see how later work has improved upon the Transformer model since its introduction in 2017.

The Inputs and Outputs of a Trained Transformer LLM

The most common picture of understanding the behavior of a Transformer LLM is to think of it as a software system that takes in text and generates text in response. Once a large enough text-in-text-out model is trained on a large enough high-quality dataset, it becomes able to generate impressive and useful outputs. [Figure 3-1](#) shows one such model used to author an email.



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

The model does not generate the text all in one operation; it actually generates one token at a time. **Figure 3-2** shows four steps of token generation in response to the input prompt. Each token generation step is one forward pass through the model (that's machine-learning speak for the inputs going into the neural network and flowing through the computations it needs to produce an output on the other end of the computation graph).

Prompt

Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.



Generation

#1

#2

#3

#4

Dear

Sarah

,

\n

<newline>

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

After each token generation, we tweak the input prompt for the next generation step by appending the output token to the end of the input prompt. We can see this in [Figure 3-3](#).

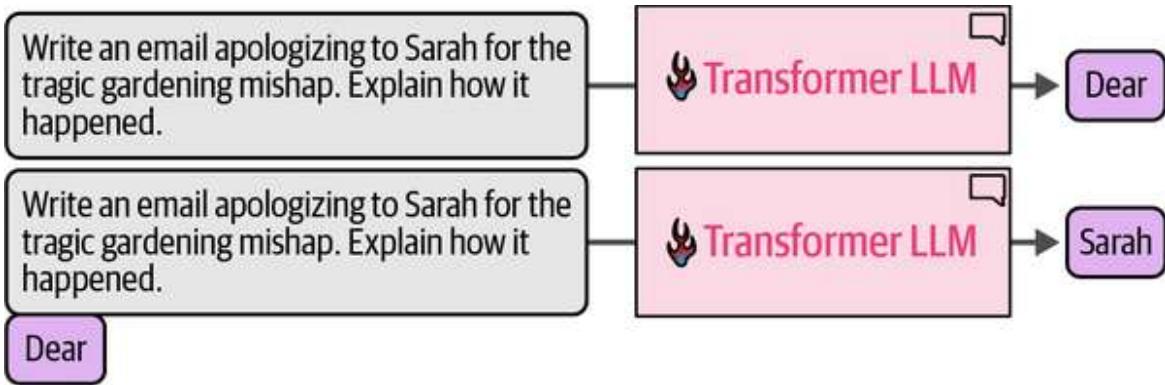


Figure 3-3. An output token is appended to the prompt, then this new text is presented to the model again for another forward pass to generate the next token.

This gives us a more accurate picture of the model as it is simply predicting the next token based on an input prompt. Software around the neural network basically runs it in a loop to sequentially expand the generated text until completion.

There's a specific word used in machine learning to describe models that consume their earlier predictions to make later predictions (e.g., the model's first generated token is used to generate the second token). They're called *autoregressive* models. That is why you'll hear text generation LLMs being called autoregressive models. This is often used to differentiate text generation models from text representation models like BERT, which are not autoregressive.

This autoregressive, token-by-token generation is what happens under the hood when we generate text with the LLM like we see here:

```

prompt = "Write an email apologizing to Sarah for the tragic
gardening mishap. Explain how it happened."

output = generator(prompt)

print(output[0]['generated_text'])

```

This generates the text:

Solution 1:

```
Subject: My Sincere Apologies for the Gardening Mishap

Dear Sarah,

I hope this message finds you well. I am writing to express my
deep
```

We can see the model begin to write the email starting with the subject. It stopped abruptly because it reached the token limit we established by setting `max_new_tokens` to 50 tokens. If we increase that, it will continue until concluding the email.

The Components of the Forward Pass

In addition to the loop, two key internal components are the tokenizer and the language modeling head (LM head). [Figure 3-4](#) shows where these components lie in the system. We saw in the previous chapter how tokenizers break down the text into a sequence of token IDs that then become the input to the model.

The tokenizer is followed by the neural network: a stack of Transformer blocks that do all of the processing. That stack is then followed by the LM head, which translates the output of the stack into probability scores for what the most likely next token is.

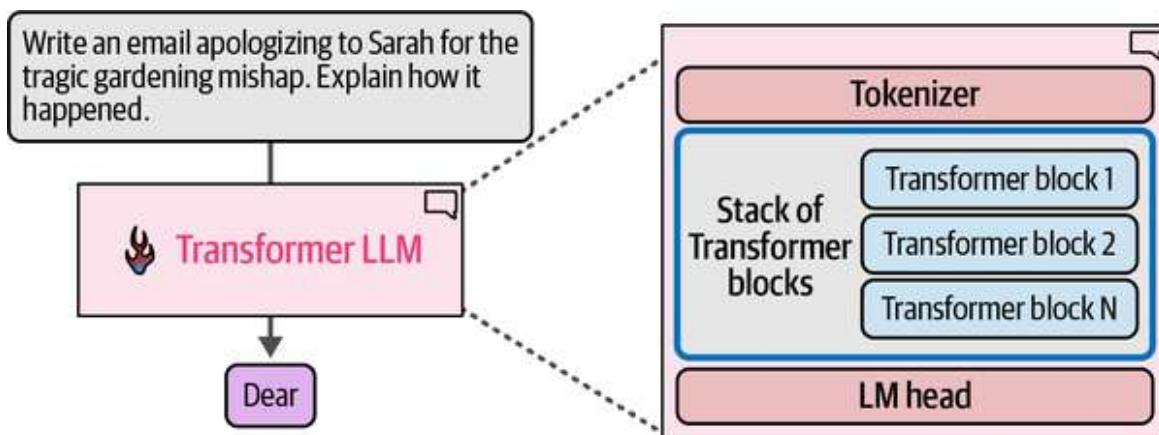


Figure 3-4. A Transformer LLM is made up of a tokenizer, a stack of Transformer blocks, and a language modeling head.

Recall from [Chapter 2](#) that the tokenizer contains a table of tokens—the tokenizer’s *vocabulary*. The model has a vector representation associated with each of these tokens in the vocabulary (token embeddings). [Figure 3-5](#) shows both the vocabulary and associated token embeddings for a model with a vocabulary of 50,000 tokens.

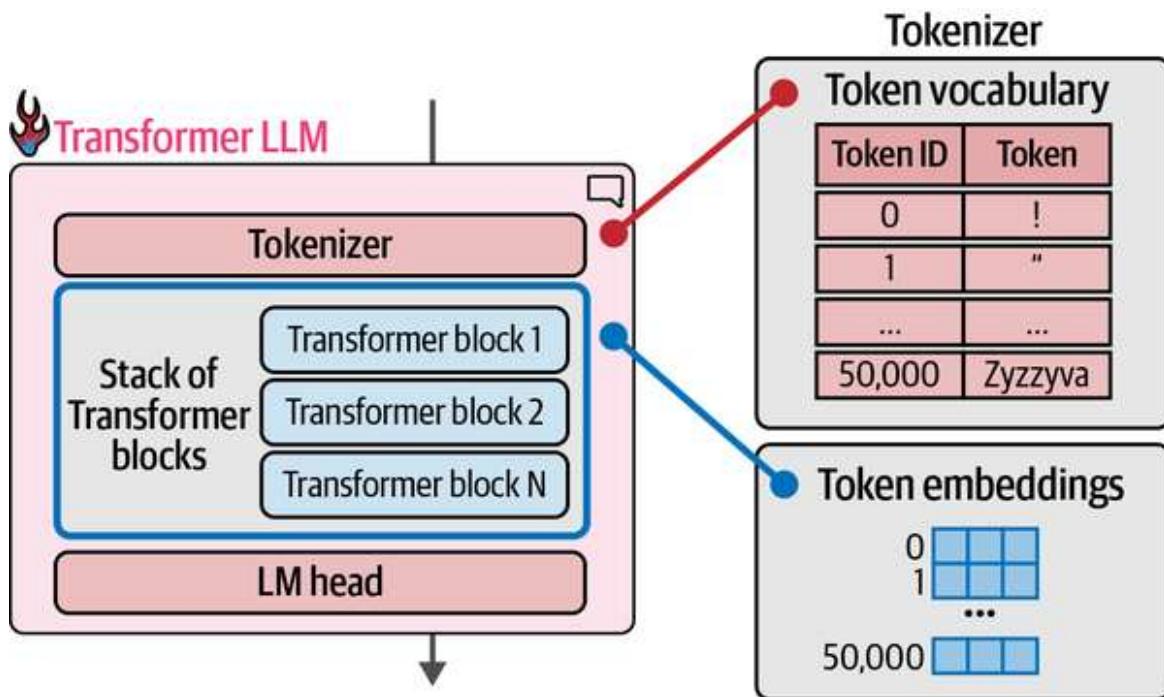


Figure 3-5. The tokenizer has a vocabulary of 50,000 tokens. The model has token embeddings associated with those embeddings.

The flow of the computation follows the direction of the arrow from top to bottom. For each generated token, the process flows once through each of the Transformer blocks in the stack in order, then to the LM head, which finally outputs the probability distribution for the next token, seen in [Figure 3-6](#).

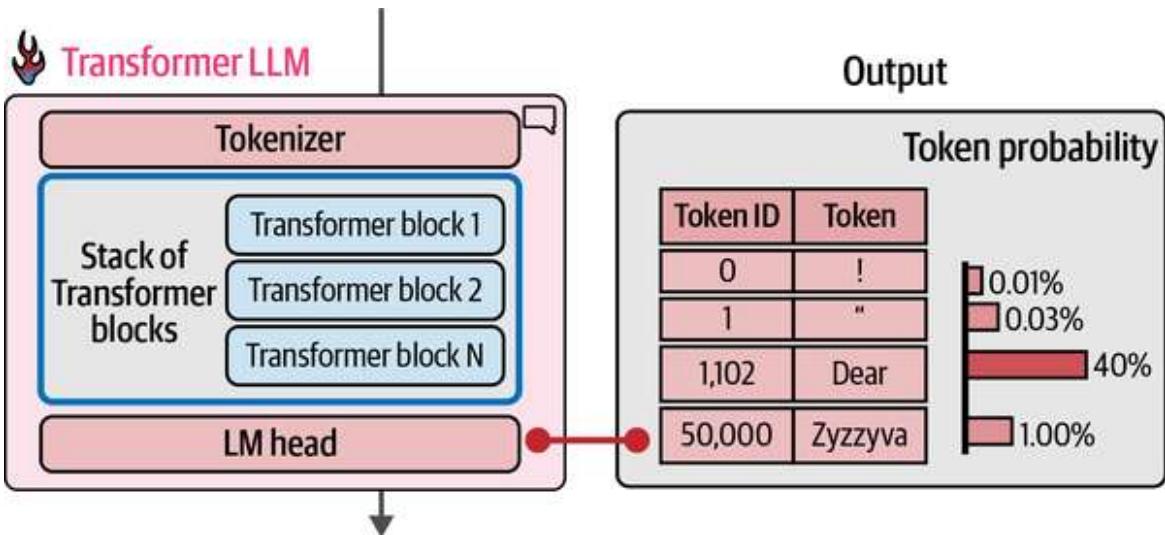


Figure 3-6. At the end of the forward pass, the model predicts a probability score for each token in the vocabulary.

The LM head is a simple neural network layer itself. It is one of multiple possible “heads” to attach to a stack of Transformer blocks to build different kinds of systems. Other kinds of Transformer heads include sequence classification heads and token classification heads.

We can display the order of the layers by simply printing out the model variable. For this model, we have:

```

Phi3ForCausalLM(
    (model): Phi3Model(
        (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
        (embed_dropout): Dropout(p=0.0, inplace=False)
        (layers): ModuleList(
            (0-31): 32 x Phi3DecoderLayer(
                (self_attn): Phi3Attention(
                    (o_proj): Linear(in_features=3072, out_features=3072,
bias=False)
                    (qkv_proj): Linear(in_features=3072,
out_features=9216, bias=False)
                    (rotary_emb): Phi3RotaryEmbedding()
                )
                (mlp): Phi3MLP(
                    (gate_up_proj): Linear(in_features=3072,
out_features=16384, bias=False)
                    (down_proj): Linear(in_features=8192,

```

```

        out_features=3072, bias=False)
            (activation_fn): SiLU()
        )
        (input_layernorm): Phi3RMSNorm()
        (resid_attn_dropout): Dropout(p=0.0, inplace=False)
        (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
        (post_attention_layernorm): Phi3RMSNorm()
    )
)
(norm): Phi3RMSNorm()
)
(lm_head): Linear(in_features=3072, out_features=32064,
bias=False)
)

```

Looking at this structure, we can notice the following highlights:

- This shows us the various nested layers of the model. The majority of the model is labeled `model`, followed by `lm_head`.
- Inside the `Phi3Model` model, we see the embeddings matrix `embed_tokens` and its dimensions. It has 32,064 tokens each with a vector size of 3,072.
- Skipping the dropout layer for now, we can see the next major component is the stack of Transformer decoder layers. It contains 32 blocks of type `Phi3DecoderLayer`.
- Each of these Transformer blocks includes an attention layer and a feedforward neural network (also known as an `mlp` or multilevel perceptron). We'll cover these in more detail later in the chapter.
- Finally, we see the `lm_head` taking a vector of size 3,072 and outputting a vector equivalent to the number of tokens the model knows. That output is the probability score for each token that helps us select the output token.

Choosing a Single Token from the Probability

Distribution (Sampling/Decoding)

At the end of processing, the output of the model is a probability score for each token in the vocabulary, as we saw previously in [Figure 3-6](#). The method of choosing a single token from the probability distribution is called the *decoding strategy*. [Figure 3-7](#) shows how this leads to picking the token “Dear” in one example.

The easiest decoding strategy would be to always pick the token with the highest probability score. In practice, this doesn’t tend to lead to the best outputs for most use cases. A better approach is to add some randomness and sometimes choose the second or third highest probability token. The idea here is to basically *sample* from the probability distribution based on the probability score, as the statisticians would say.

What this means for the example in [Figure 3-7](#) is that if the token “Dear” has a 40% probability of being the next token, then it has a 40% chance of being picked (instead of greedy search, which would pick it directly for having the highest score). So with this method, all the other tokens have a chance of being picked according to their score.

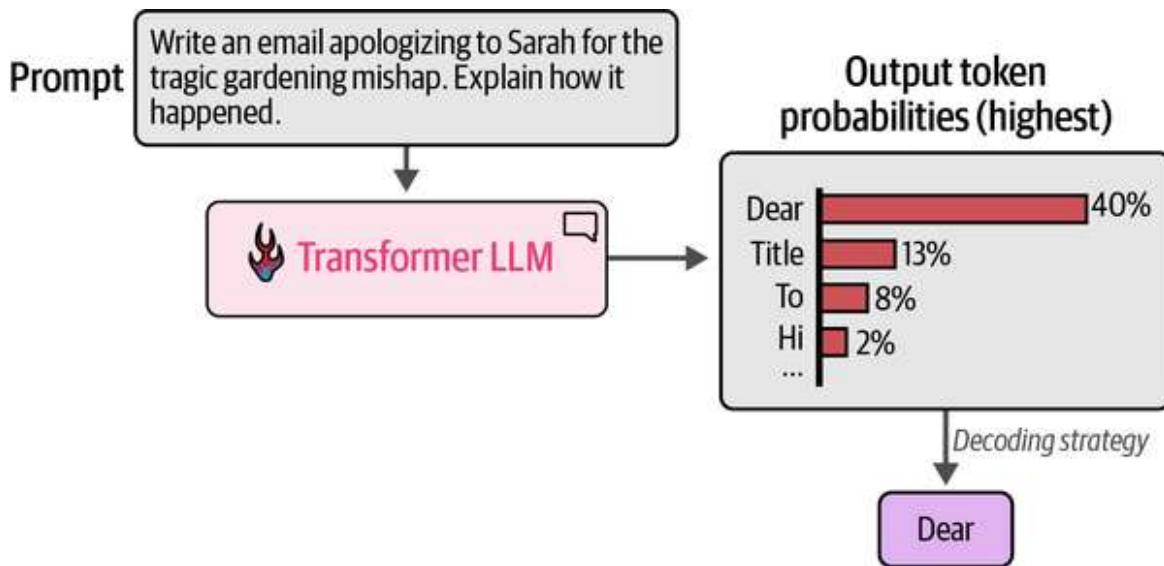


Figure 3-7. The tokens with the highest probability after the model’s forward pass. Our decoding strategy decides which of the tokens to output by sampling based on the probabilities.

Choosing the highest scoring token every time is called *greedy decoding*. It’s what happens if you set the temperature parameter to zero in an LLM.

We cover the concept of temperature in [Chapter 6](#).

Let's look more closely at the code that demonstrates this process. In this code block, we pass the input tokens through the model, and then `lm_head`:

```
prompt = "The capital of France is"

# Tokenize the input prompt
input_ids = tokenizer(prompt, return_tensors="pt").input_ids

# Tokenize the input prompt
input_ids = input_ids.to("cuda")

# Get the output of the model before the lm_head
model_output = model.model(input_ids)

# Get the output of the lm_head
lm_head_output = model.lm_head(model_output[0])
```

Now, `lm_head_output` is of the shape [1, 6, 32064]. We can access the token probability scores for the last generated token using `lm_head_output[0, -1]`, which uses the index 0 across the batch dimension; the index `-1` gets us the last token in the sequence. This is now a list of probability scores for all 32,064 tokens. We can get the top scoring token ID, and then decode it to arrive at the text of the generated output token:

```
token_id = lm_head_output[0, -1].argmax(-1)
tokenizer.decode(token_id)
```

In this case this turns out to be:

```
Paris
```

Parallel Token Processing and Context Size

One of the most compelling features of Transformers is that they lend themselves better to parallel computing than previous neural network architectures in language processing. In text generation, we get a first glance at this when looking at how each token is processed. We know from the previous chapter that the tokenizer will break down the text into tokens. Each of these input tokens then flows through its own computation path (that's a good first intuition, at least). We can see these individual processing tracks or streams in [Figure 3-8](#).

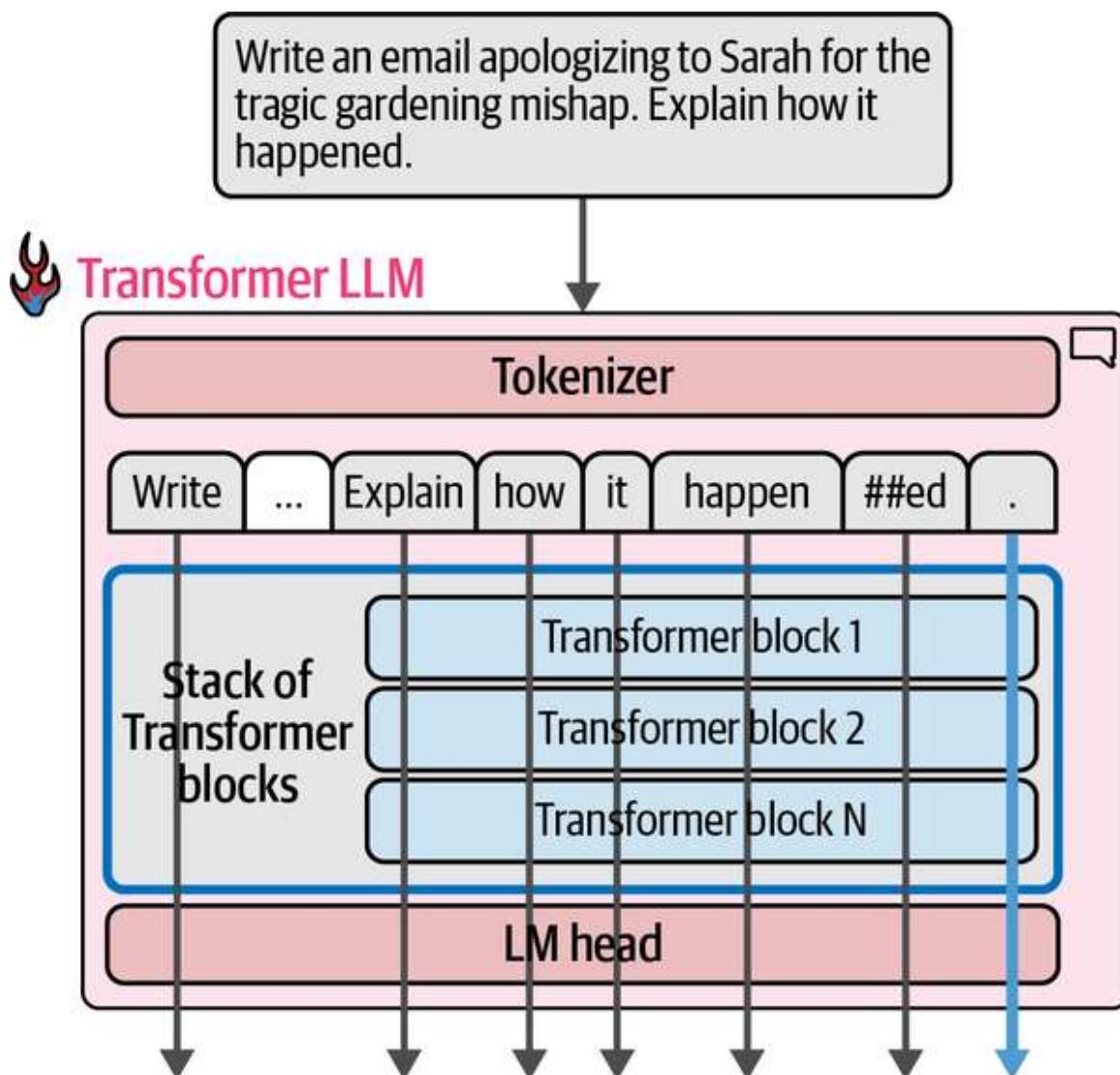


Figure 3-8. Each token is processed through its own stream of computation (with some interaction between them in attention steps, as we'll later see).

Current Transformer models have a limit for how many tokens they can process at once. That limit is called the model's context length. A model with 4K context length can only process 4K tokens and would only have 4K of these streams.

Each of the token streams starts with an input vector (the embedding vector and some positional information; we'll discuss positional embeddings later in the chapter). At the end of the stream, another vector emerges as the result of the model's processing, as shown in [Figure 3-9](#).

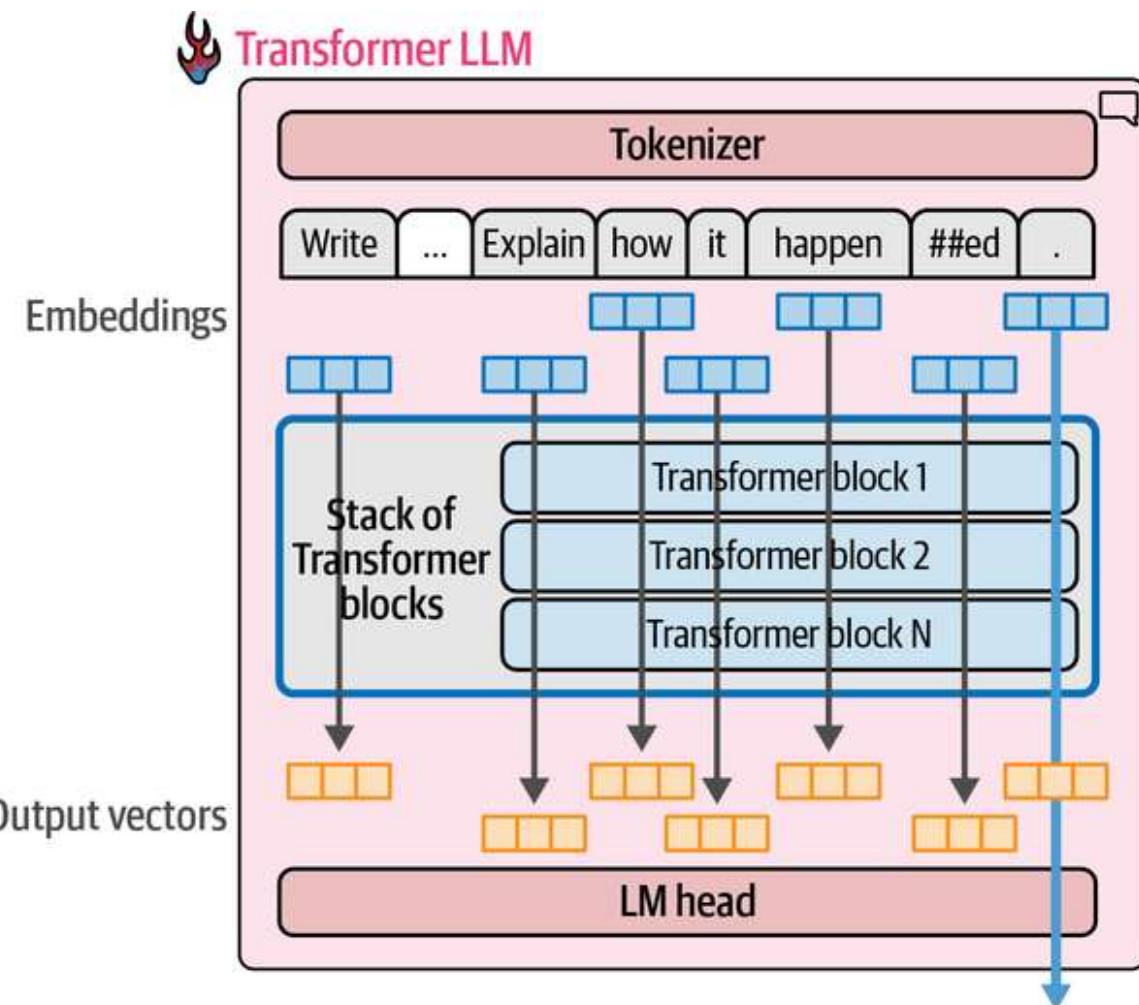


Figure 3-9. Each processing stream takes a vector as input and produces a final resulting vector of the same size (often referred to as the model dimension).

For text generation, only the output result of the last stream is used to predict the next token. That output vector is the only input into the LM head as it calculates the probabilities of the next token.

You may wonder why we go through the trouble of calculating all the token streams if we’re discarding the outputs of all but the last token. The answer is that the calculations of the previous streams are required and used in calculating the final stream. Yes, we’re not using their final output vector, but we use earlier outputs (in each Transformer block) in the Transformer block’s attention mechanism.

If you’re following along with the code examples, recall that the output of `lm_head` was of the shape [1, 6, 32064]. That was because the input to it was of the shape [1, 6, 3072], which is a batch of one input string, containing six tokens, each of them represented by a vector of size 3,072 corresponding to the output vectors after the stack of Transformer blocks.

We can access these matrices and view their dimensions by printing:

```
model_output[0].shape
```

This outputs:

```
torch.Size([1, 6, 3072])
```

Similarly, we can print the output of the LM head:

```
lm_head_output.shape
```

This outputs:

```
torch.Size([1, 6, 32064])
```

Speeding Up Generation by Caching Keys and Values

Recall that when generating the second token, we simply append the output token to the input and do another forward pass through the model. If we

give the model the ability to cache the results of the previous calculation (especially some of the specific vectors in the attention mechanism), we no longer need to repeat the calculations of the previous streams. This time the only needed calculation is for the last stream. This is an optimization technique called the **keys and values (kv) cache** and it provides a significant speedup of the generation process. Keys and values are some of the central components of the attention mechanism, as we'll see later in this chapter.

Figure 3-10 shows how when generating the second token, only one processing stream is active as we cache the results of the previous streams.

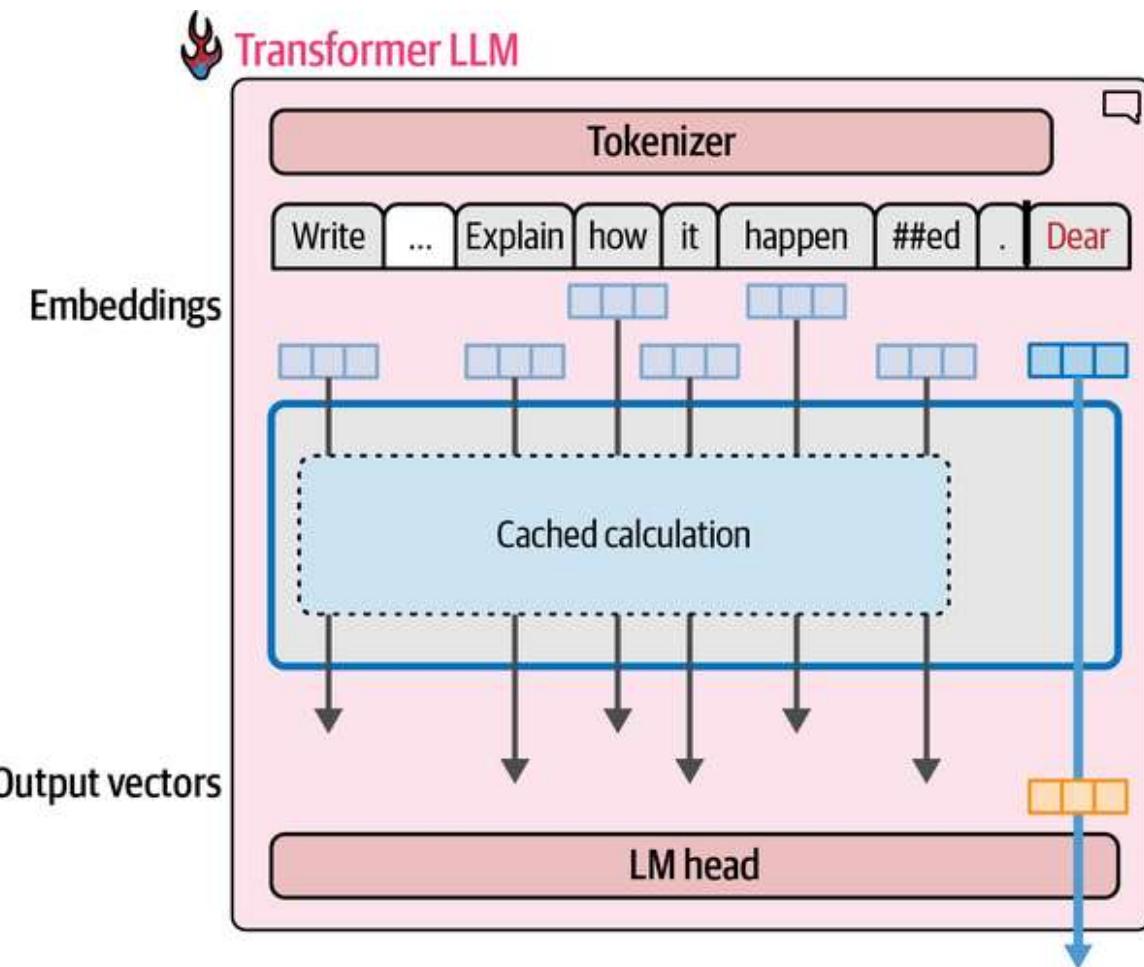


Figure 3-10. When generating text, it's important to cache the computation results of previous tokens instead of repeating the same calculation over and over again.

In Hugging Face Transformers, cache is enabled by default. We can disable it by setting `use_cache` to `False`. We can see the difference in speed by

asking for a long generation, and timing the generation with and without caching:

```
prompt = "Write a very long email apologizing to Sarah for the  
tragic gardening mishap. Explain how it happened."  
# Tokenize the input prompt  
input_ids = tokenizer(prompt, return_tensors="pt").input_ids  
input_ids = input_ids.to("cuda")
```

Then we time how long it takes to generate 100 tokens with caching. We can use the `%%timeit` magic command in Jupyter or Colab to time how long the execution takes (it runs the command several times and gets the average):

```
%%timeit -n 1  
# Generate the text  
generation_output = model.generate(  
    input_ids=input_ids,  
    max_new_tokens=100,  
    use_cache=True  
)
```

On a Colab with a T4 GPU, this comes to 4.5 seconds. How long would that take if we disable the cache, however?

```
%%timeit -n 1  
# Generate the text  
generation_output = model.generate(  
    input_ids=input_ids,  
    max_new_tokens=100,  
    use_cache=False  
)
```

This comes out to 21.8 seconds. A dramatic difference. In fact, from a user experience standpoint, even the four-second generation time tends to be a long time to wait for a user that's staring at a screen and waiting for an output from the model. This is one reason why LLM APIs stream the output

tokens as the model generates them instead of waiting for the entire generation to be completed.

Inside the Transformer Block

We can now talk about where the vast majority of processing happens: the Transformer blocks. As [Figure 3-11](#) shows, Transformer LLMs are composed of a series Transformer blocks (often in the range of six in the original Transformer paper, to over a hundred in many large LLMs). Each block processes its inputs, then passes the results of its processing to the next block.

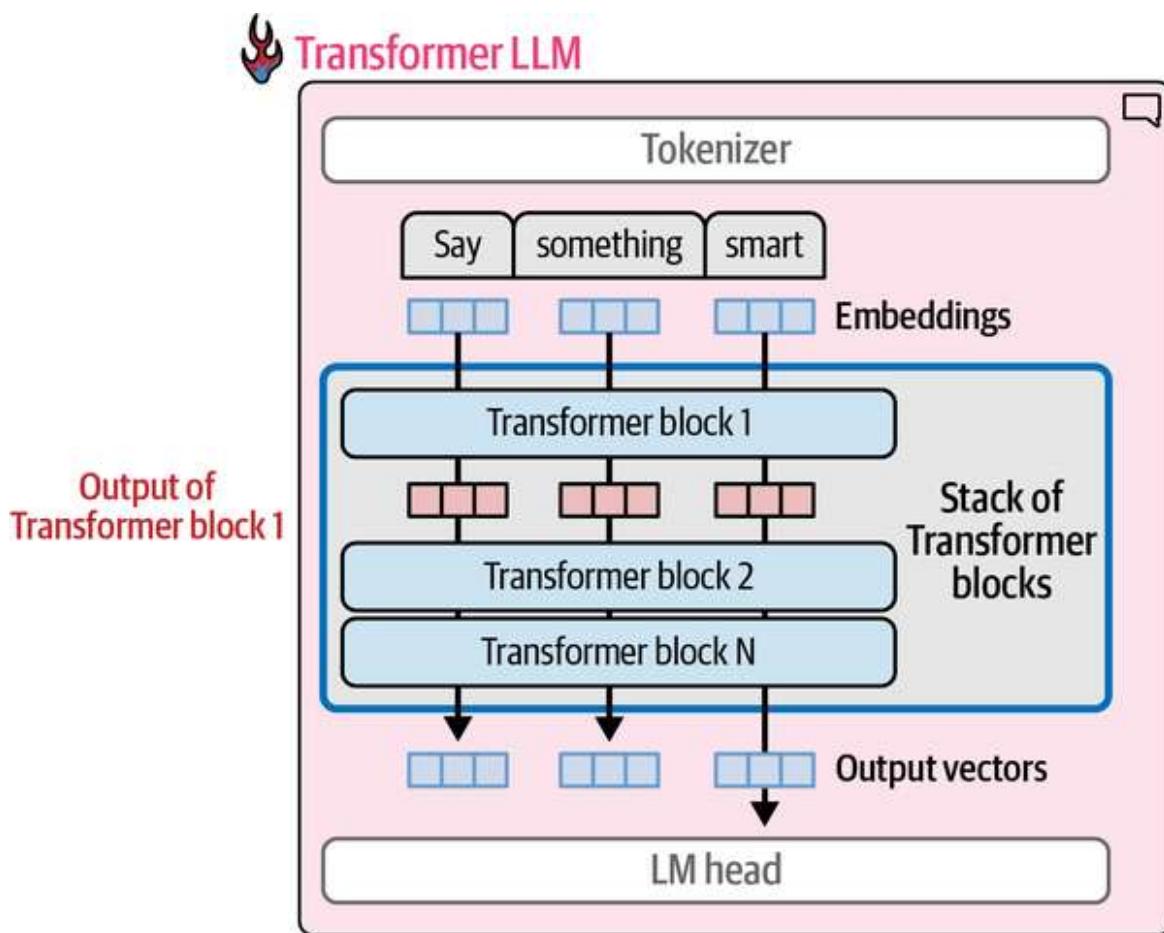


Figure 3-11. The bulk of the Transformer LLM processing happens inside a series of Transformer blocks, each handing the result of its processing as input to the subsequent block.

A Transformer block ([Figure 3-12](#)) is made up of two successive components:

1. *The attention layer* is mainly concerned with incorporating relevant information from other input tokens and positions
2. *The feedforward layer* houses the majority of the model's processing capacity

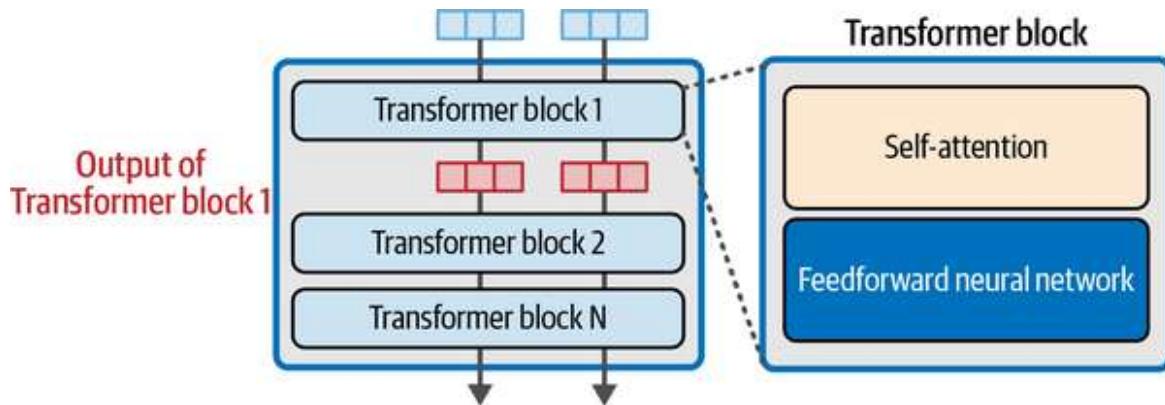


Figure 3-12. A Transformer block is made up of a self-attention layer and a feedforward neural network.

The feedforward neural network at a glance

A simple example giving the intuition of the feedforward neural network would be if we pass the simple input “The Shawshank” to a language model, with the expectation that it will generate “Redemption” as the most probable next word (in reference to the film from 1994).

The feedforward neural network (collectively in all the model layers) is the source of this information, as [Figure 3-13](#) shows. When the model was successfully trained to model a massive text archive (which included many mentions of “The Shawshank Redemption”), it learned and stored the information (and behaviors) that make it succeed at this task.

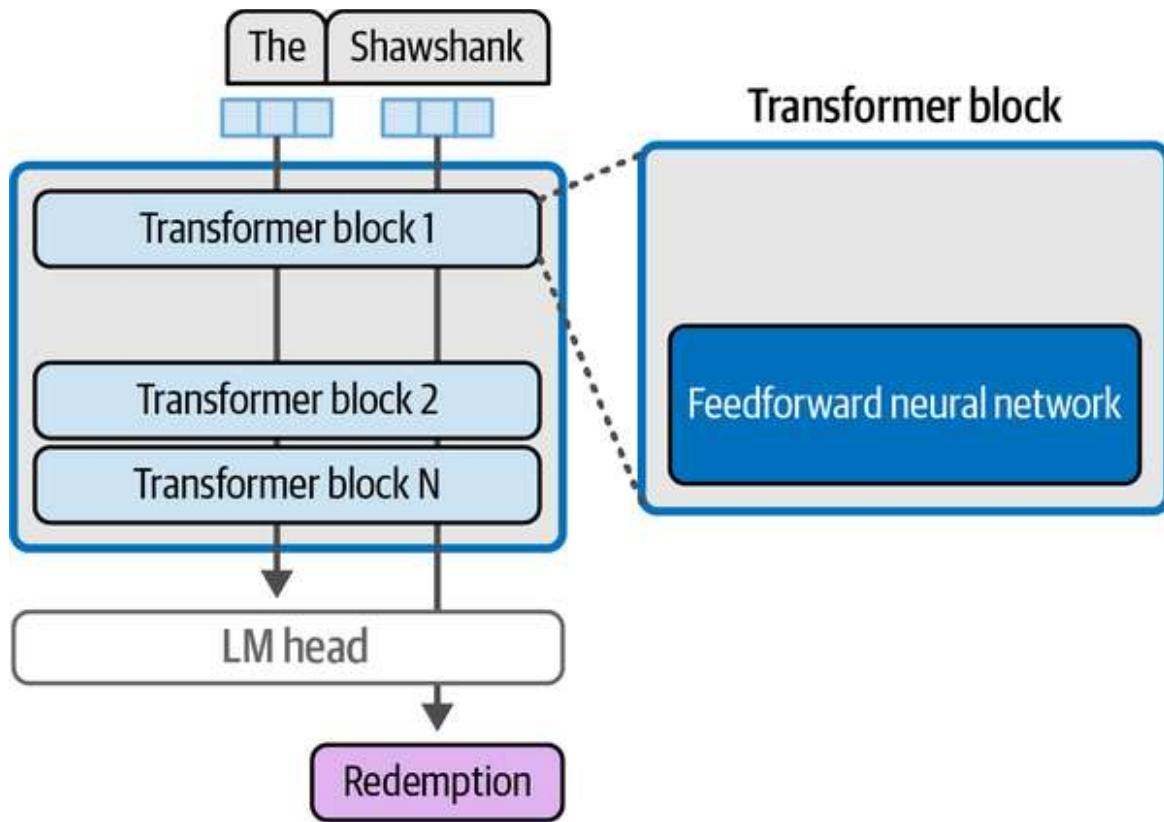


Figure 3-13. The feedforward neural network component of a Transformer block likely does the majority of the model’s memorization and interpolation.

For an LLM to be successfully trained, it needs to memorize a lot of information. But it is not simply a large database. Memorization is only one ingredient in the recipe of impressive text generation. The model is able to use this same machinery to interpolate between data points and more complex patterns to be able to generalize—which means doing well on inputs it hadn’t seen in the past and were not in its training dataset.

NOTE

When you use a modern commercial LLM, the outputs you get are not the ones mentioned earlier in the strict meaning of a “language model.” Passing “The Shawshank” to a chat LLM like GPT-4 produces an output:

"The Shawshank Redemption" is a 1994 film directed by Frank Darabont and is based on the novella "Rita Hayworth and Shawshank Redemption" written by Stephen King. ...etc.

This is because raw language models (like GPT-3) are difficult for people to properly utilize. This is why the language model is then trained on instruction-tuning and human preference and feedback fine-tuning to match people’s expectations of what the model should output.

The attention layer at a glance

Context is vital in order to properly model language. Simple memorization and interpolation based on the previous token can only take us so far. We know that because this was one of the leading approaches to build language models before neural networks (see Chapter 3, “N-gram Language Models” of *Speech and Language Processing* by Daniel Jurafsky and James H. Martin).

Attention is a mechanism that helps the model incorporate context as it’s processing a specific token. Think of the following prompt:

“The dog chased the squirrel because it”

For the model to predict what comes after “it,” it needs to know what “it” refers to. Does it refer to the dog or the squirrel?

In a trained Transformer LLM, the attention mechanism makes that determination. Attention adds information from the context into the representation of the “it” token. We can see a simple version of that in [Figure 3-14](#).

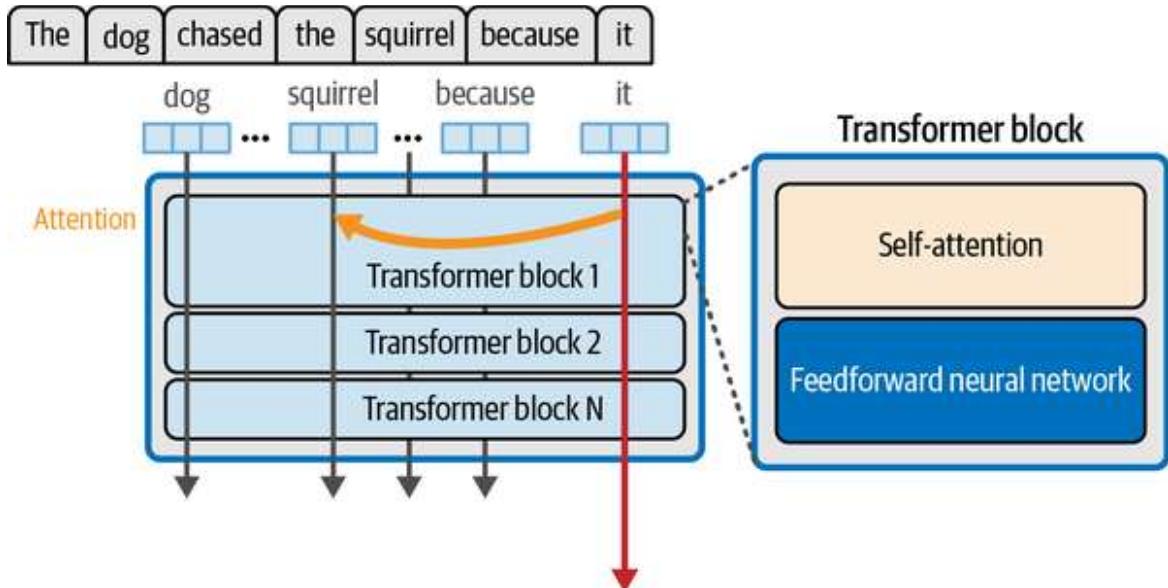


Figure 3-14. The self-attention layer incorporates relevant information from previous positions that help process the current token.

The model does that based on the patterns seen and learned from the training dataset. Perhaps previous sentences also give more clues, like, for example, referring to the dog as “she” thus making it clear that “it” refers to the squirrel.

Attention is all you need

It is worth diving deeper into the attention mechanism. The most stripped-down version of the mechanism is shown in [Figure 3-15](#). It shows multiple token positions going into the attention layer; the final one is the one being currently processed (the pink arrow). The attention mechanism operates on the input vector at that position. It incorporates relevant information from the context into the vector it produces as the output for that position.

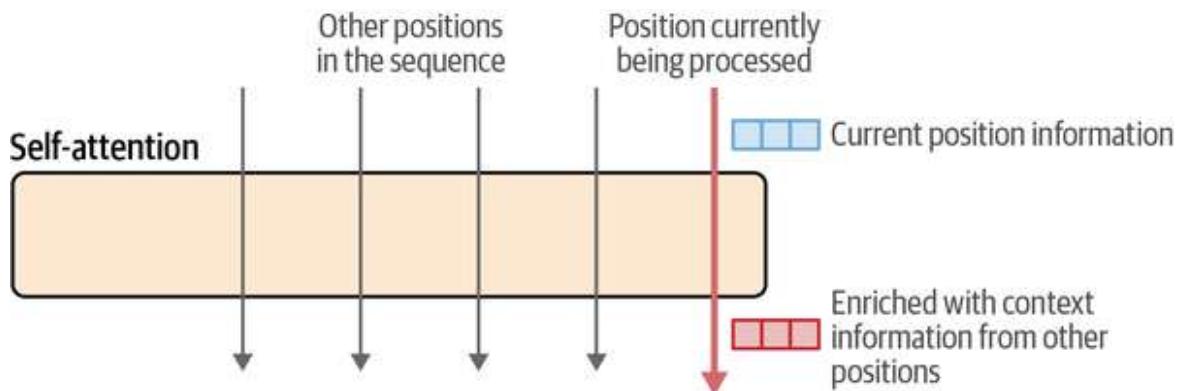


Figure 3-15. A simplified framing of attention: an input sequence and a current position being processed. As we're mainly concerned with this position, the figure shows an input vector and an output vector that incorporates information from the previous elements in the sequence according to the attention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

Two main steps are involved in the attention mechanism:

1. A way to score how relevant each of the previous input tokens are to the current token being processed (in the pink arrow).
2. Using those scores, we combine the information from the various positions into a single output vector.

Figure 3-16 shows these two steps.

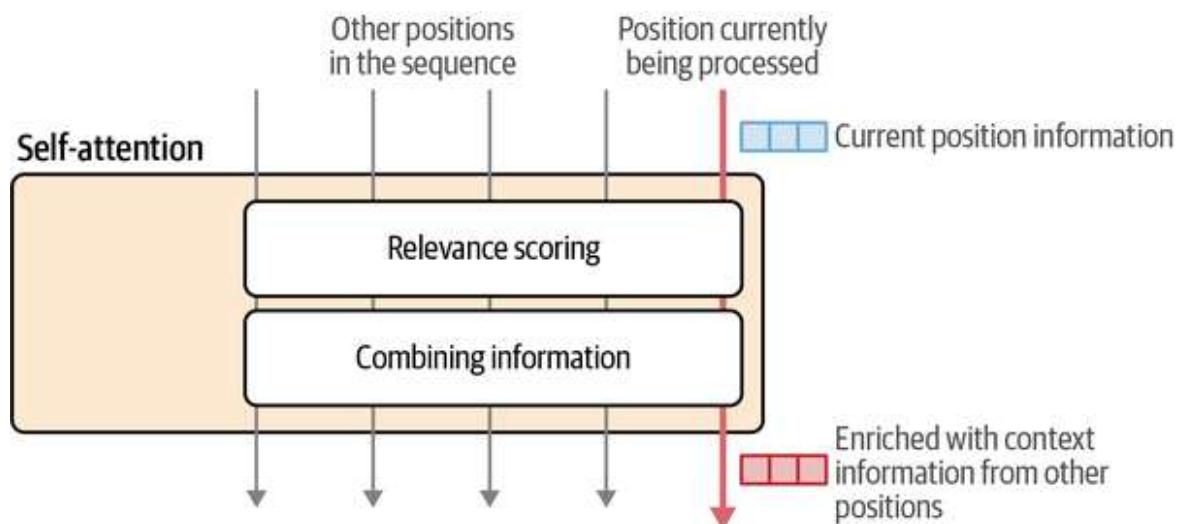


Figure 3-16. Attention is made up of two major steps: relevance scoring for each position, then a step where we combine the information based on those scores.

To give the Transformer more extensive attention capability, the attention mechanism is duplicated and executed multiple times in parallel. Each of these parallel applications of attention is conducted into an *attention head*. This increases the model's capacity to model complex patterns in the input sequence that require paying attention to different patterns at once.

Figure 3-17 shows the intuition of how attention heads run in parallel with a preceding step of splitting information and a later step of combining the results of all the heads.

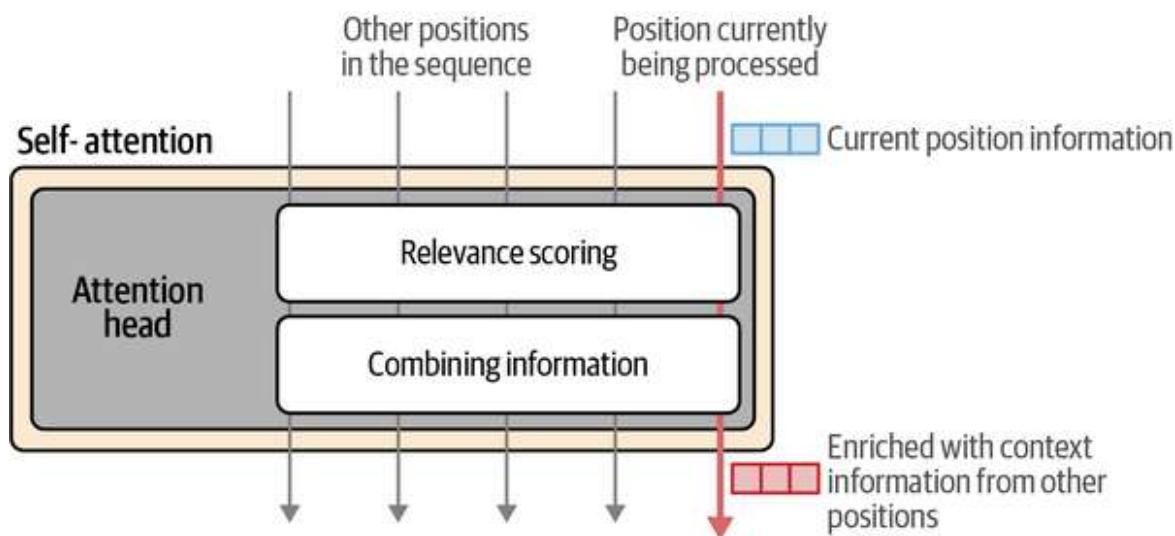


Figure 3-17. We get better LLMs by doing attention multiple times in parallel, increasing the model's capacity to attend to different types of information.

How attention is calculated

Let's look at how attention is calculated inside a single attention head. Before we start the calculation, let's observe the following as the starting position:

- The attention layer (of a generative LLM) is processing attention for a single position.
- The inputs to the layer are:
 - The vector representation of the current position or token
 - The vector representations of the previous tokens

- The goal is to produce a new representation of the current position that incorporates relevant information from the previous tokens:
 - For example, if we’re processing the last position in the sentence “Sarah fed the cat because it,” we want “it” to represent the cat—so attention bakes in “cat information” from the cat token.
- The training process produces three projection matrices that produce the components that interact in this calculation:
 - A query projection matrix
 - A key projection matrix
 - A value projection matrix

Figure 3-18 shows the starting position for all of these components before the attention calculations start. For simplicity, let’s look at only one attention head because the other heads have identical calculations but with their individual projection matrices.

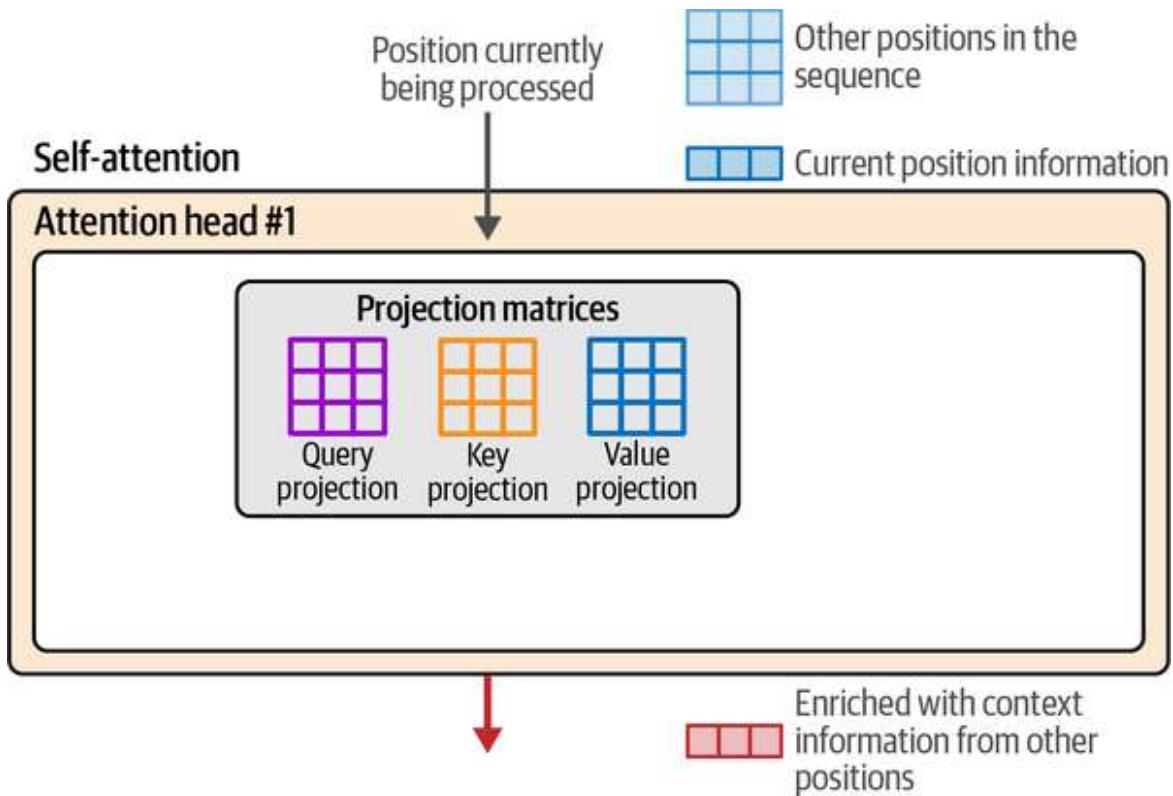


Figure 3-18. Before starting the self-attention calculation, we have the inputs to the layer and projection matrices for queries, keys, and values.

Attention starts by multiplying the inputs by the projection matrices to create three new matrices. These are called the queries, keys, and values matrices. These matrices contain the information of the input tokens projected to three different spaces that help carry out the two steps of attention:

1. Relevance scoring
2. Combining information

Figure 3-19 shows these three new matrices, and how the bottom row of all three matrices is associated with the current position while the rows above it are associated with the previous positions.

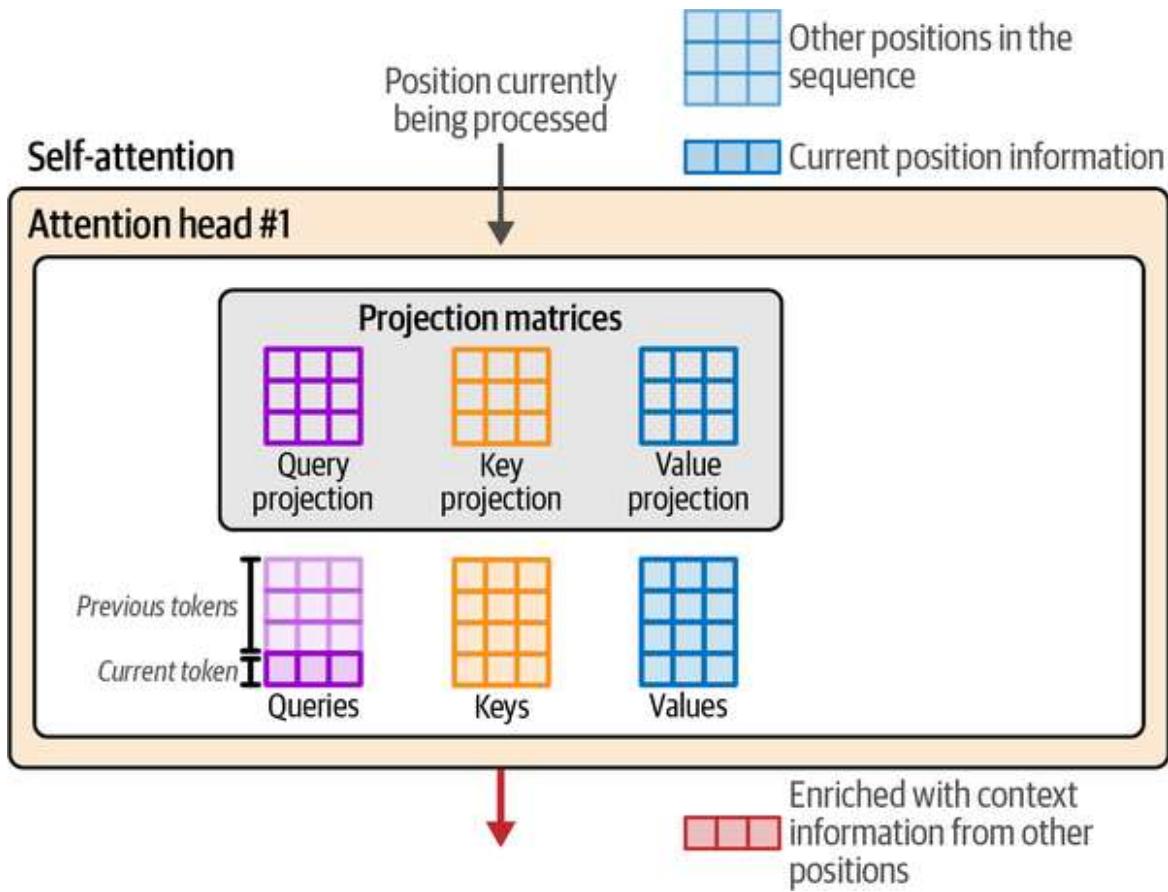


Figure 3-19. Attention is carried out by the interaction of the queries, keys, and values matrices. Those are produced by multiplying the layer's inputs with the projection matrices.

Self-attention: Relevance scoring

In a generative Transformer, we're generating one token at a time. This means we're processing one position at a time. So the attention mechanism here is only concerned with this one position, and how information from other positions can be pulled in to inform this position.

The relevance scoring step of attention is conducted by multiplying the query vector of the current position with the keys matrix. This produces a score stating how relevant each previous token is. Passing that by a softmax operation normalizes these scores so they sum up to 1. **Figure 3-20** shows the relevance score resulting from this calculation.

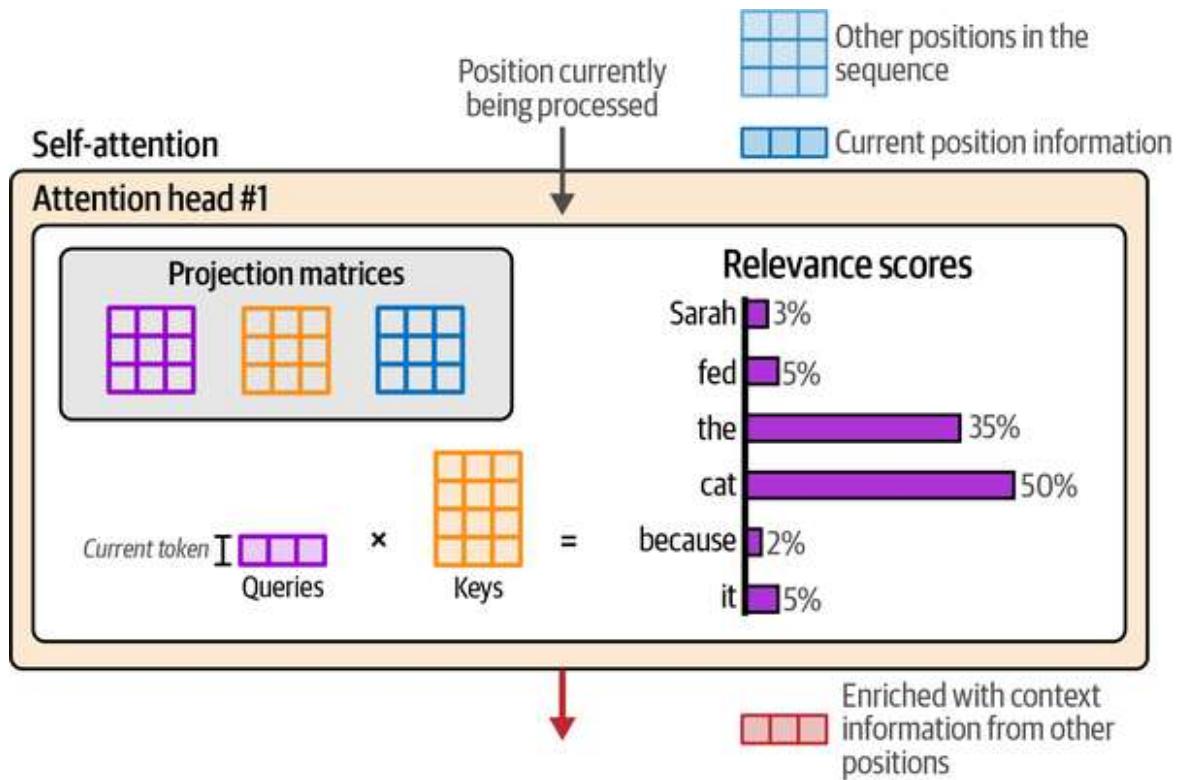


Figure 3-20. Scoring the relevance of previous tokens is accomplished by multiplying the query associated with the current position with the keys matrix.

Self-attention: Combining information

Now that we have the relevance scores, we multiply the value vector associated with each token by that token's score. Summing up those resulting vectors produces the output of this attention step, as we see in Figure 3-21.

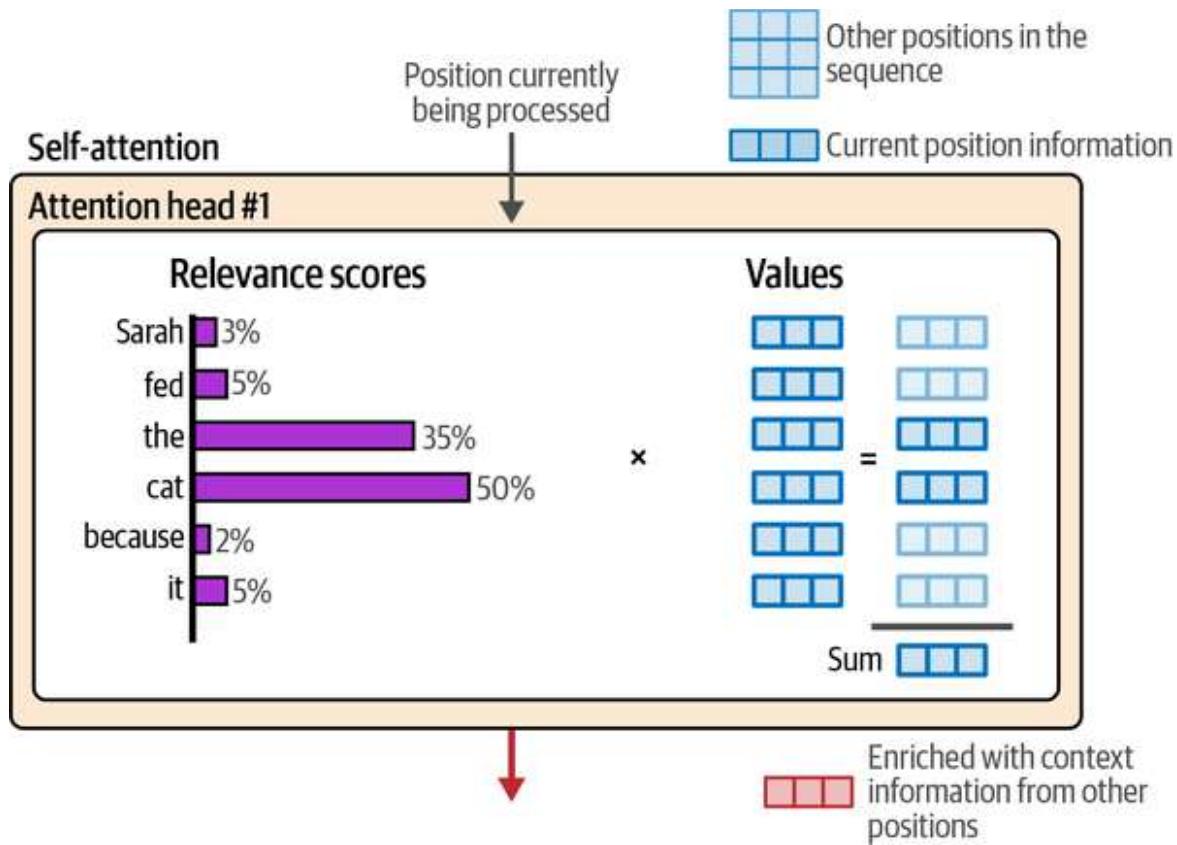


Figure 3-21. Attention combines the relevant information of previous positions by multiplying their relevance scores by their respective value vectors.

Recent Improvements to the Transformer Architecture

Since the release of the Transformer architecture, much work has been done to improve it and create better models. This spans training on larger datasets and optimizations for the training process and learning rates to use, but it also extends to the architecture itself. At the time of writing, a lot of the ideas of the original Transformer stand unchanged. There are a few architectural ideas that have proved to be valuable. They contribute to the performance of more recent Transformer models like Llama 2. In this final section of the chapter, we go over a number of the important recent developments of the Transformer architecture.

More Efficient Attention

The area that gets the most focus from the research community is the attention layer of the Transformer. This is because the attention calculation is the most computationally expensive part of the process.

Local/sparse attention

As Transformers started getting larger, ideas like sparse attention (“[Generating long sequences with sparse transformers](#)”) and sliding window attention (“[Longformer: The long-document transformer](#)”) provided improvements for the efficiency of the attention calculation. Sparse attention limits the context of previous tokens that the model can attend to, as we can see in [Figure 3-22](#).

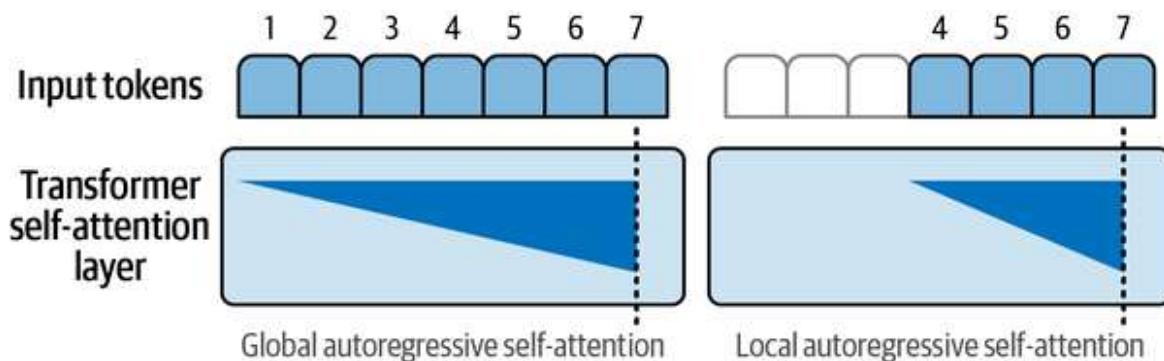


Figure 3-22. Local attention boosts performance by only paying attention to a small number of previous positions.

One model that incorporates such a mechanism is GPT-3. But it does not use that for all the Transformer blocks—the quality of the generation would vastly degrade if the model could only see a small number of previous tokens. The GPT-3 architecture interweaved full-attention and efficient-attention Transformer blocks. So the Transformer blocks alternate between full attention (e.g., blocks 1 and 3) and sparse attention (e.g., blocks 2 and 4).

To demonstrate different kinds of attention, review [Figure 3-23](#), which shows how different attention mechanisms work. Each figure shows which previous tokens (light blue) can be attended to when processing the current token (in dark blue).

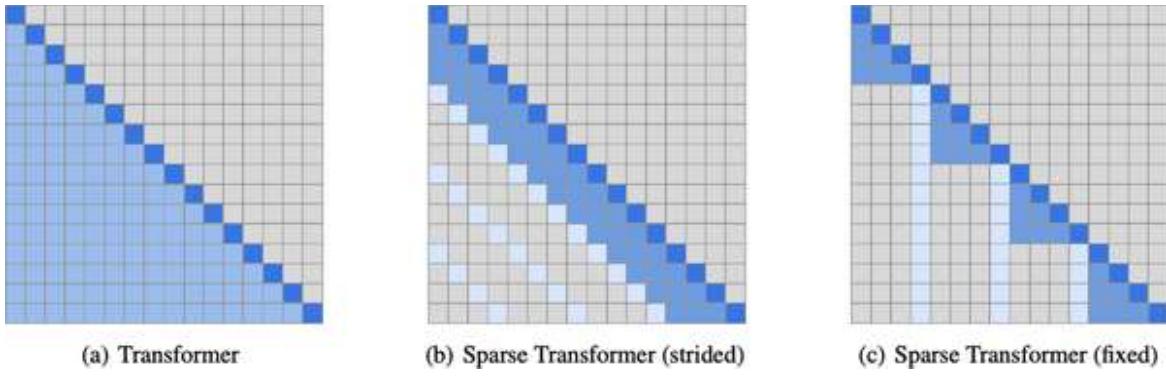


Figure 3-23. Full attention versus sparse attention. Figure 3-24 explains the coloring. (Source: “Generating long sequences with sparse transformers”.)

Each row corresponds to a token being processed. The color coding indicates which tokens the model is able to pay attention to while it's processing the token in the dark blue cell. [Figure 3-24](#) describes this with more clarity.

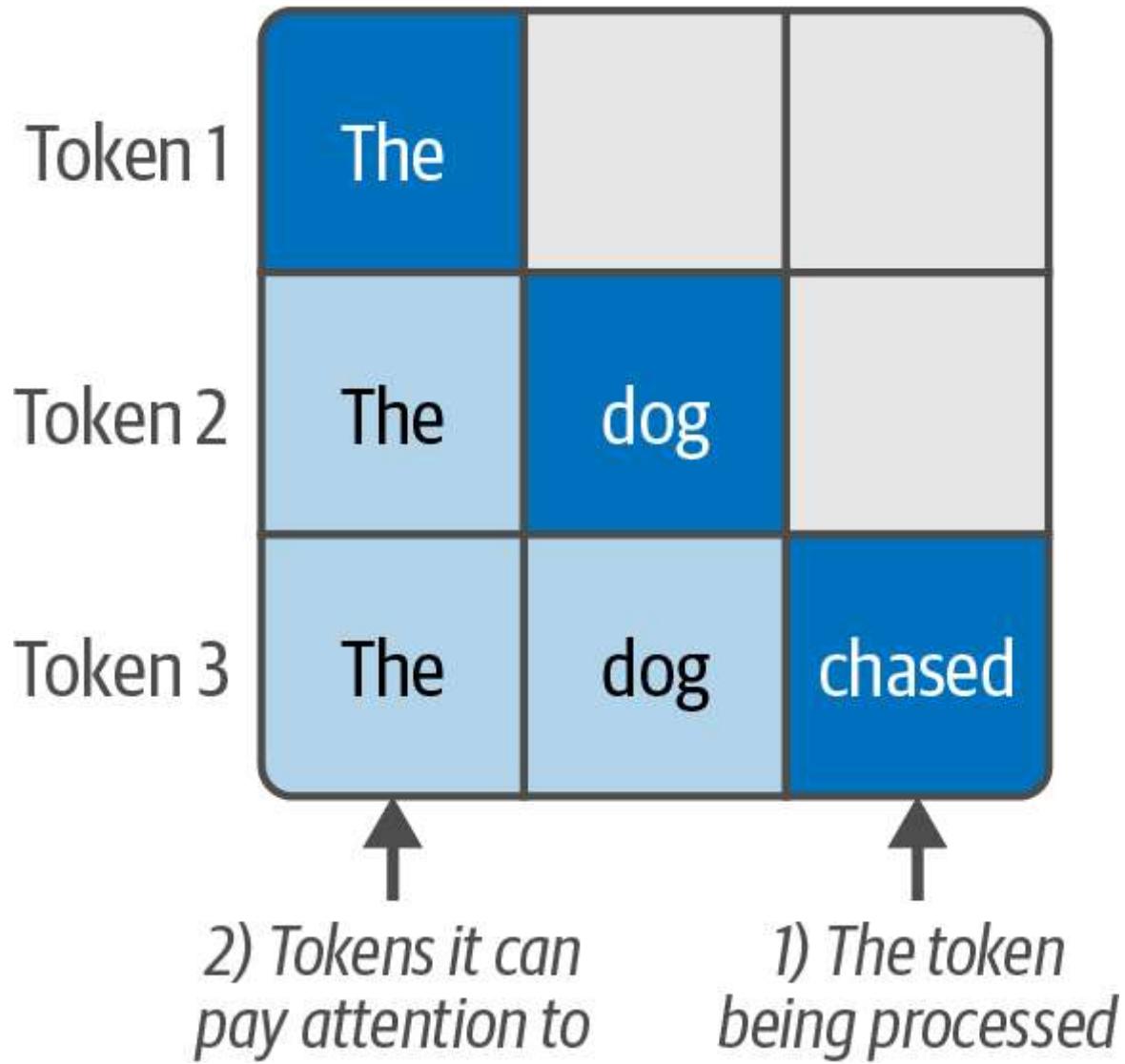


Figure 3-24. Attention figures show which token is being processed, and which previous tokens an attention mechanism allows it to attend to.

This figure also shows the autoregressive nature of decoder Transformer blocks (which make up most text generation models); they can only pay attention to previous tokens. Contrast this to BERT, which can pay attention to both sides (hence the B in BERT stands for bidirectional).

Multi-query and grouped-query attention

A more recent efficient attention tweak to the Transformer is grouped-query attention (“[GQA: Training generalized multi-query transformer models from multi-head checkpoints](#)”), which is used by models like Llama 2 and

3. Figure 3-25 shows these different types of attention, and the next section continues to explain them.

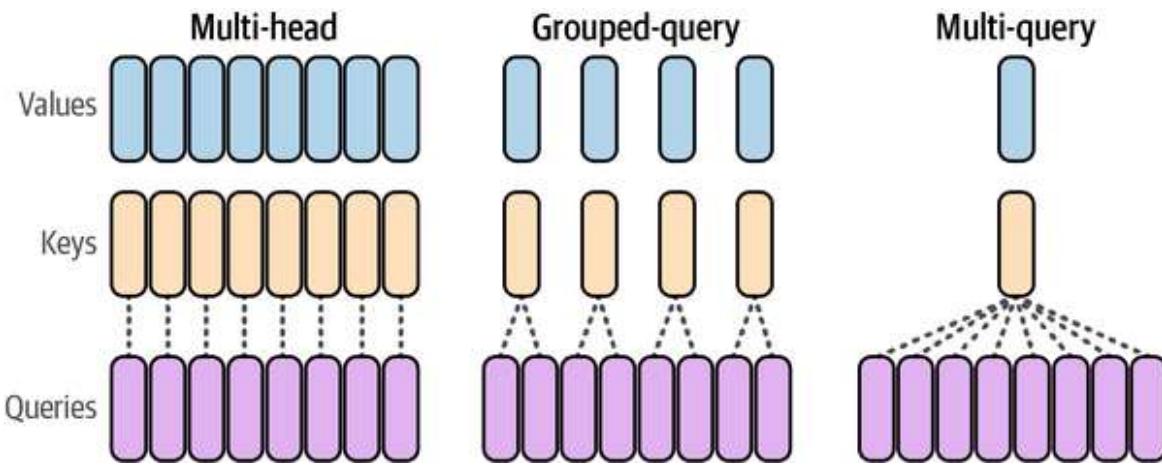


Figure 3-25. A comparison of different kinds of attention: the original multi-head, grouped-query attention, and multi-query attention (source: “Fast transformer decoding: One write-head is all you need”).

Grouped-query attention builds on multi-query attention (“Fast transformer decoding: One write-head is all you need”). These methods improve inference scalability of larger models by reducing the size of the matrices involved.

Optimizing attention: From multi-head to multi-query to grouped query

Earlier in the chapter we showed how the Transformer paper described multi-headed attention. [The Illustrated Transformer](#) discusses in detail how the queries, keys, and values matrices are used to conduct the attention operation. Figure 3-26 shows how each “attention head” has its own distinct query, key, and value matrices calculated for a given input.

The way that multi-query attention optimizes this is to share the keys and values matrices between all the heads. So the only unique matrices for each head would be the queries matrices, as we can see in Figure 3-27.

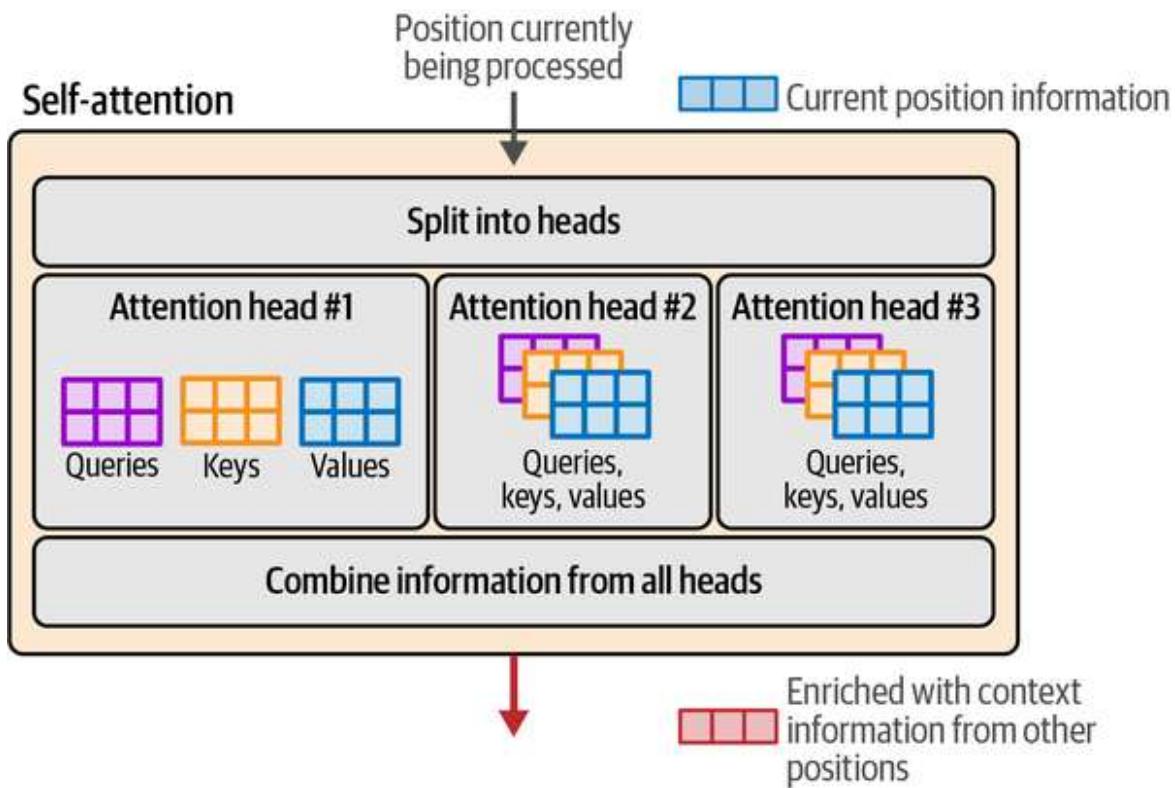


Figure 3-26. Attention is conducted using matrices of queries, keys, and values. In multi-head attention, each head has a distinct version of each of these matrices.

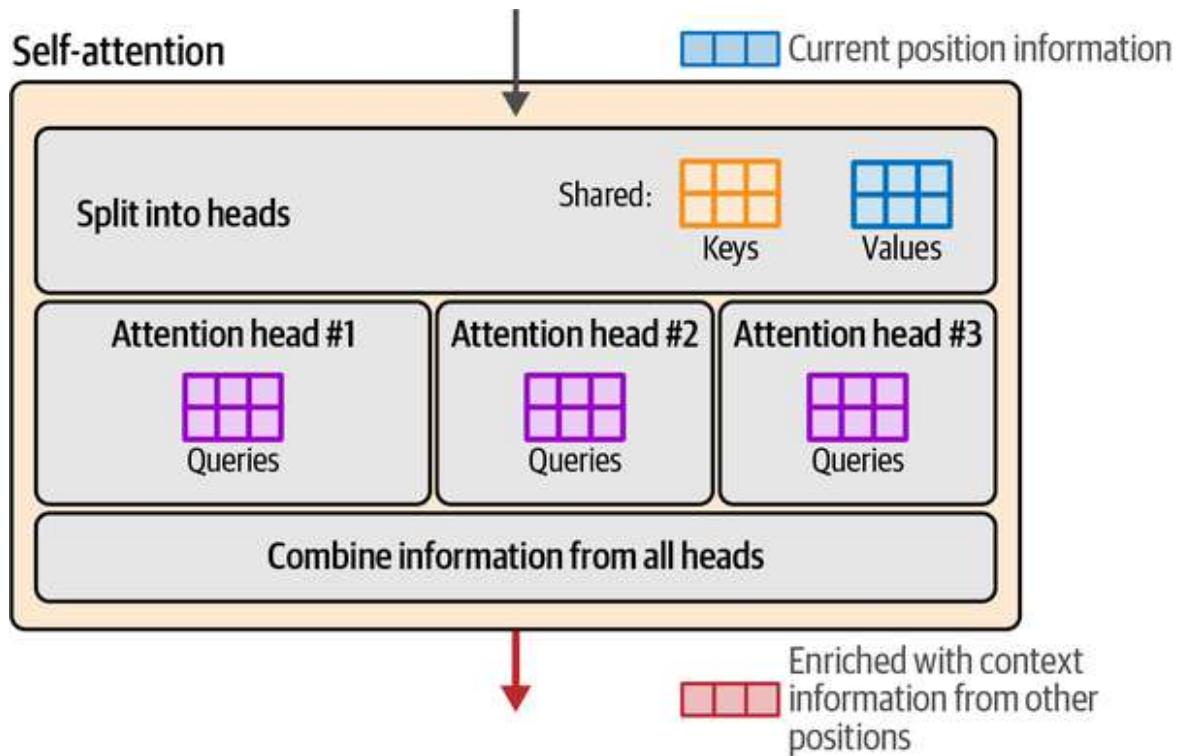


Figure 3-27. Multi-query attention presents a more efficient attention mechanism by sharing the keys and values matrices across all the attention heads.

As model sizes grow, however, this optimization can be too punishing and we can afford to use a little more memory to improve the quality of the models. This is where grouped-query attention comes in. Instead of cutting the number of keys and values matrices to one of each, it allows us to use more (but less than the number of heads). [Figure 3-28](#) shows these groups and how each group of attention heads shares keys and values matrices.

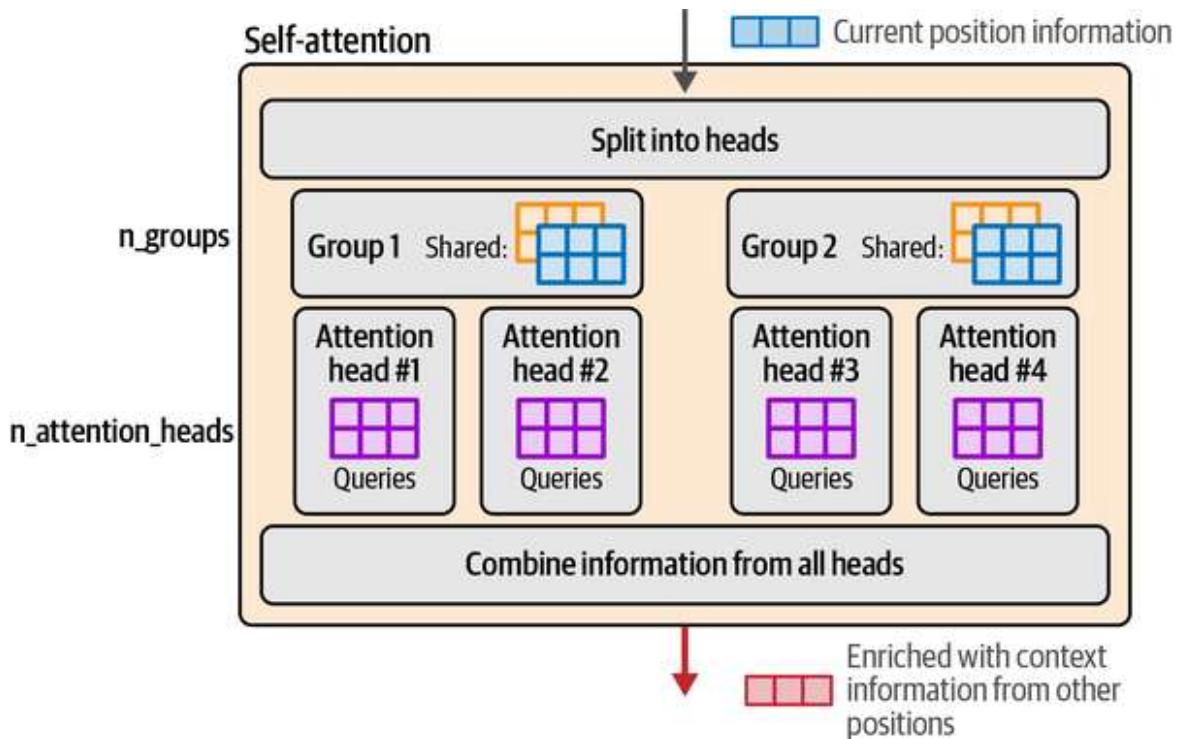


Figure 3-28. Grouped-query attention sacrifices a little bit of the efficiency of multi-query attention in return for a large improvement in quality by allowing multiple groups of shared key/value matrices; each group has its respective set of attention heads.

Flash Attention

Flash Attention is a popular method and implementation that provides significant speedups for both training and inference of Transformer LLMs on GPUs. It speeds up the attention calculation by optimizing what values are loaded and moved between a GPU's shared memory (SRAM) and high bandwidth memory (HBM). It is described in detail in the papers “[FlashAttention: Fast and memory-efficient exact attention with IO-awareness](#)” and the subsequent “[FlashAttention-2: Faster attention with better parallelism and work partitioning](#)”.

The Transformer Block

Recall that the two major components of a Transformer block are an attention layer and a feedforward neural network. A more detailed view of the block would also reveal the residual connections and layer-normalization operations that we can see in [Figure 3-29](#).

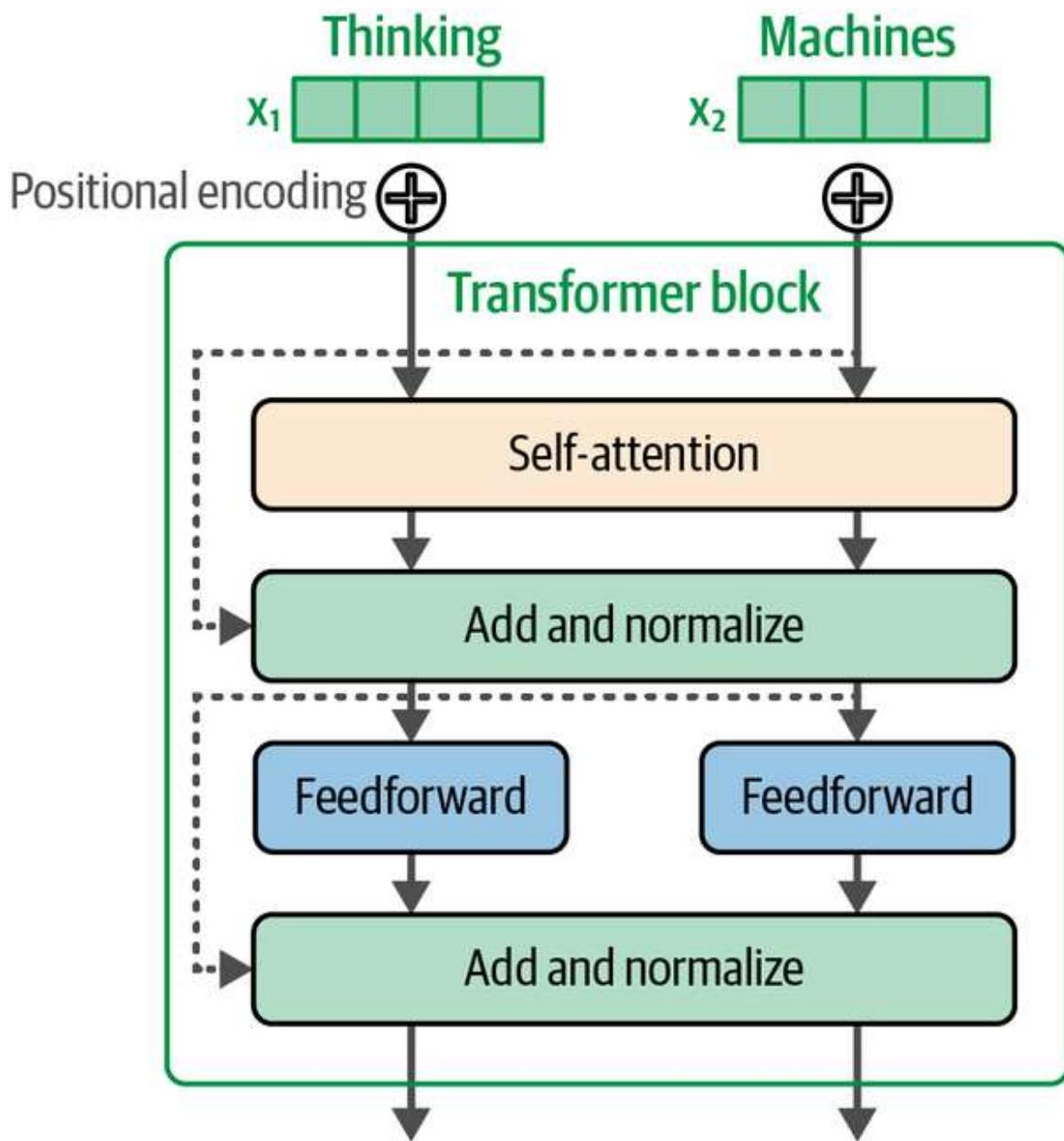


Figure 3-29. A Transformer block from the original Transformer paper.

The latest Transformer models at the time of this writing still retain the major components, yet make a number of tweaks as we can see in [Figure 3-30](#).

One of the differences we see in this version of the Transformer block is that normalization happens prior to attention and the feedforward layers. This has been reported to reduce the required training time (read: “[On layer normalization in the Transformer architecture](#)”). Another improvement in

normalization here is using RMSNorm, which is simpler and more efficient than the LayerNorm used in the original Transformer (read: “[Root mean square layer normalization](#)”). Lastly, instead of the original Transformer’s ReLU activation function, newer variants like SwiGLU (described in “[GLU Variants Improve Transformer](#)”) are now more common.

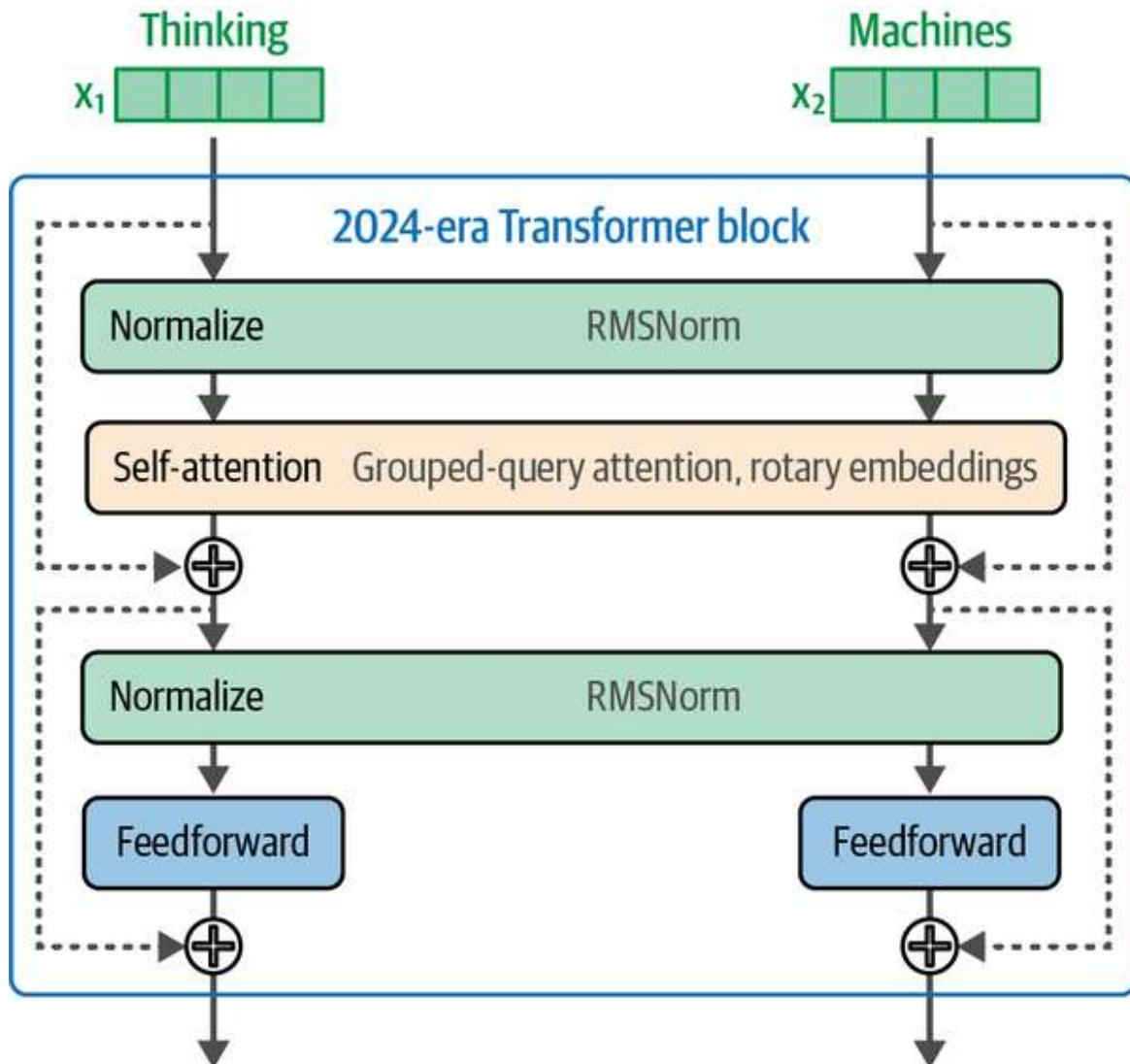


Figure 3-30. The Transformer block of a 2024-era Transformer like Llama 3 features some tweaks like pre-normalization and an attention optimized with grouped-query attention and rotary embeddings.

Positional Embeddings (RoPE)

Positional embeddings have been a key component since the original Transformer. They enable the model to keep track of the order of tokens/words in a sequence/sentence, which is an indispensable source of information in language. From the many positional encoding schemes proposed in the past years, rotary positional embeddings (or “RoPE,” introduced in [“RoFormer: Enhanced Transformer with rotary position embedding”](#)) is especially important to point out.

The original Transformer paper and some of the early variants had absolute positional embeddings that, in essence, marked the first token as position 1, the second as position 2...etc. These could either be static methods (where the positional vectors are generated using geometric functions) or learned (where the model training assigns them their values during the learning process). Some challenges arise from such methods when we scale up models, which requires us to find ways to improve their efficiency.

For example, one challenge in efficiently training models with large context is that a lot of documents in the training set are much shorter than that context. It would be inefficient to allocate the entire, say, 4K context to a short 10-word sentence. So during model training, documents are packed together into each context in the training batch, as [Figure 3-31](#) shows.

Naive, inefficient organization of training data



Efficient training data packing

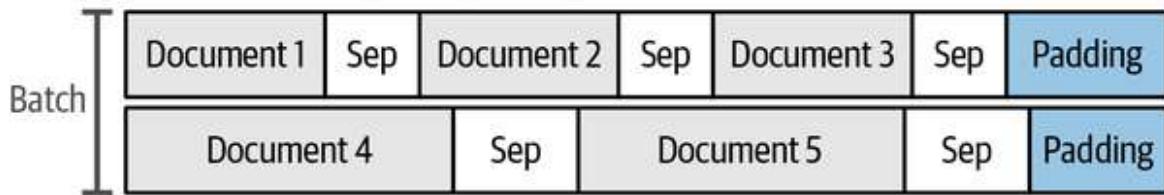


Figure 3-31. Packing is the process of efficiently organizing short training documents into the context. It includes grouping multiple documents in a single context while minimizing the padding at the end of the context.

Learn more about packing by reading “[Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance](#)” and watching the great visuals in “[Introducing packed BERT for 2X training speed-up in natural language processing](#)”.

Positional embedding methods have to adapt to this and other practical considerations. If Document 50, for example, starts at position 50, then we’d be misinforming the model if we tell it that that first token is number 50 and that would affect its performance (because it would assume there’s previous context while in reality the earlier tokens belong to a different and unrelated document the model should ignore).

Instead of the static, absolute embeddings that are added in the beginning of the forward pass, rotary embeddings are a method to encode positional information in a way that captures absolute and relative token position information. It is based on the idea of rotating vectors in their embeddings space. In the forward pass, they are added in the attention step, as [Figure 3-32](#) shows.

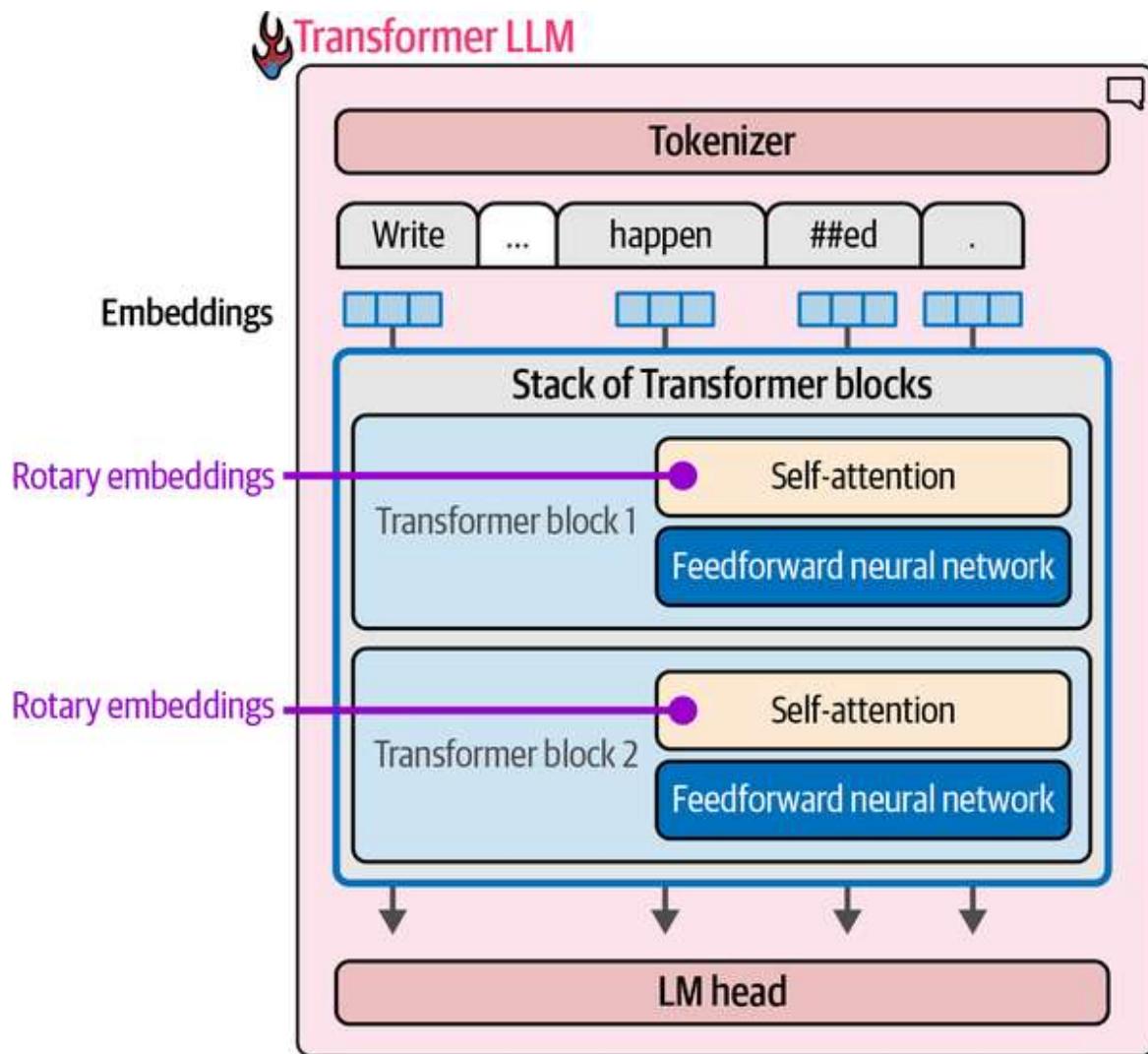


Figure 3-32. Rotary embeddings are applied in the attention step, not at the start of the forward pass.

During the attention process, the positional information is mixed in specifically to the queries and keys matrices just before we multiply them for relevance scoring, as we can see in [Figure 3-33](#).

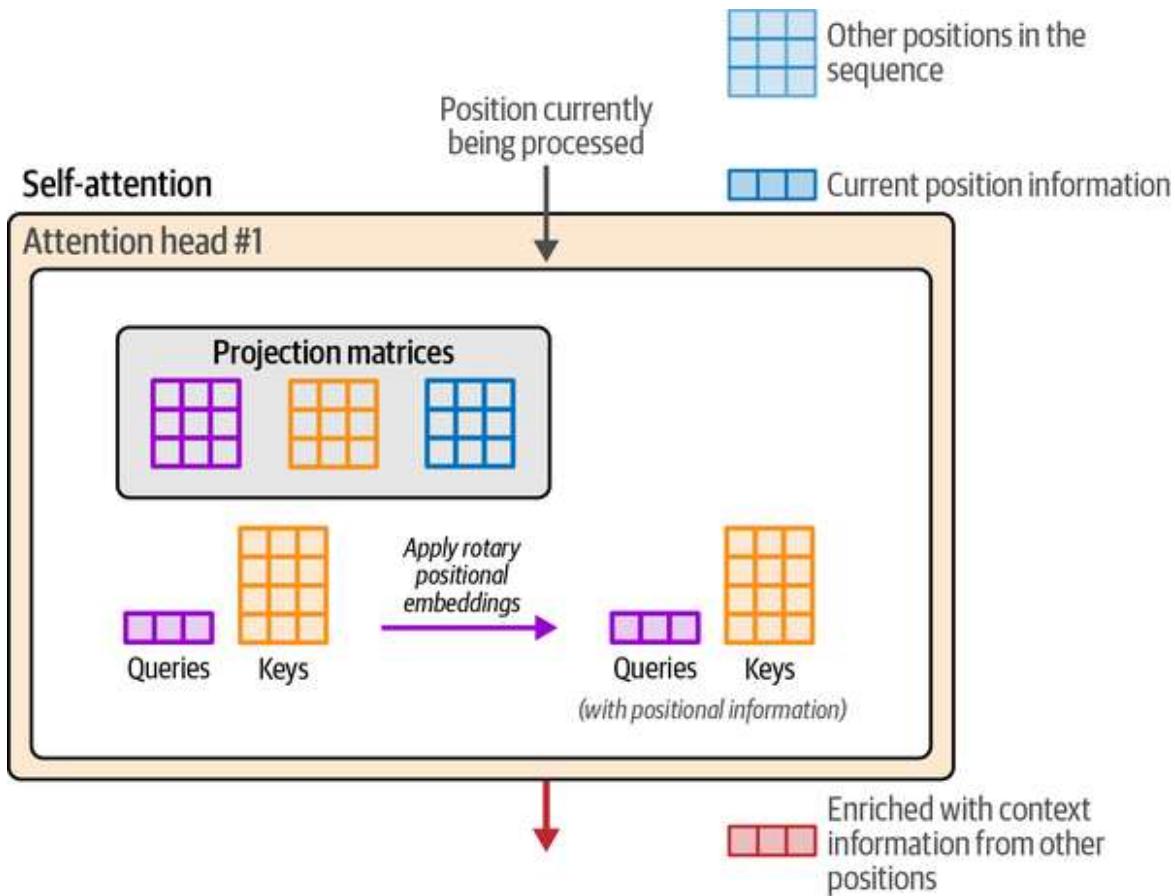


Figure 3-33. Rotary positional embeddings are added to the representation of tokens just before the relevance scoring step in self-attention.

Other Architectural Experiments and Improvements

Many tweaks of the Transformer are proposed and researched on a continuous basis. “[A Survey of Transformers](#)” highlights a few of the main directions. Transformer architectures are also constantly adapted to domains beyond LLMs. Computer vision is an area where a lot of Transformer architecture research is happening (see: “[Transformers in vision: A survey](#)” and “[A survey on vision transformer](#)”). Other domains include robotics (see “[Open X-Embodiment: Robotic learning datasets and RT-X models](#)”) and time series (see “[Transformers in time series: A survey](#)”).

Summary

In this chapter we discussed the main intuitions of Transformers and recent developments that enable the latest Transformer LLMs. We went over many new concepts, so let's break down the key concepts that we discussed in this chapter:

- A Transformer LLM generates *one token at a time*.
- That output token is *appended to the prompt*, then this updated prompt is presented to the model again for another forward pass to generate the next token.
- The *three major components* of the Transformer LLM are the tokenizer, a stack of Transformer blocks, and a language modeling head.
- The tokenizer contains the *token vocabulary* for the model. The model has *token embeddings* associated with those tokens. Breaking the text into tokens and then using the embeddings of these tokens is the first step in the token generation process.
- The forward pass flows through all the stages once, *one by one*.
- Near the end of the process, the LM head scores the *probabilities of the next possible token*. Decoding strategies inform which actual token to pick as the output for this generation step (sometimes it's the most probable next token, but not always).
- One reason the Transformer excels is its ability to process tokens in parallel. Each of the input tokens flow into their *individual tracks or streams of processing*. The number of streams is the model's "context size" and this represents the max number of tokens the model can operate on.
- Because Transformer LLMs loop to generate the text one token at a time, it's a good idea to *cache* the processing results of each step so

we don't duplicate the processing effort (these results are stored as various matrices within the layers).

- The majority of processing happens within *Transformer blocks*. These are made up of two components. One of them is the *feedforward neural network*, which is able to store information and make predictions and interpolations from data it was trained on.
- The second major component of a Transformer block is the *attention* layer. Attention incorporates contextual information to allow the model to better capture the nuance of language.
- Attention happens in two major steps: (1) scoring relevance and (2) combining information.
- A Transformer attention layer conducts several attention operations in parallel, each occurring inside an *attention head*, and their outputs are aggregated to make up the output of the attention layer.
- Attention can be accelerated via sharing the keys and values matrices between all heads, or groups of heads (*grouped-query attention*).
- Methods like *Flash Attention* speed up the attention calculation by optimizing how the operation is done on the different memory systems of a GPU.

Transformers continue to see new developments and proposed tweaks to improve them in different scenarios, including language models and other domains and applications.

In Part II of the book, we will cover some of these practical applications of LLMs. In [Chapter 4](#), we start with text classification, a common task in Language AI. This next chapter serves as an introduction to applying both generative and representation models.

Part II. Using Pretrained Language Models

OceanofPDF.com

Chapter 4. Text Classification

A common task in natural language processing is classification. The goal of the task is to train a model to assign a label or class to some input text (see [Figure 4-1](#)). Classifying text is used across the world for a wide range of applications, from sentiment analysis and intent detection to extracting entities and detecting language. The impact of language models, both representative and generative, on classification cannot be understated.

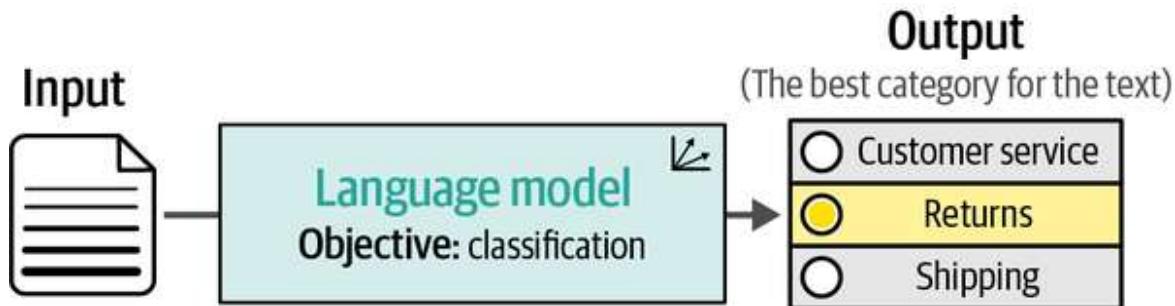


Figure 4-1. Using a language model to classify text.

In this chapter, we will discuss several ways to use language models for classifying text. It will serve as an accessible introduction to using language models that already have been trained. Due to the broad field of text classification, we will discuss several techniques and use them to explore the field of language models:

- “[Text Classification with Representation Models](#)” demonstrates the flexibility of nongenerative models for classification. We will cover both task-specific models and embedding models.
- “[Text Classification with Generative Models](#)” is an introduction to generative language models as most of them can be used for classification. We will cover both an open source as well as a closed source language model.

In this chapter, we will focus on leveraging pretrained language models, models that already have been trained on large amounts of data that can be

used for classifying text. As illustrated in [Figure 4-2](#), we will examine both representation and language models and explore their differences.

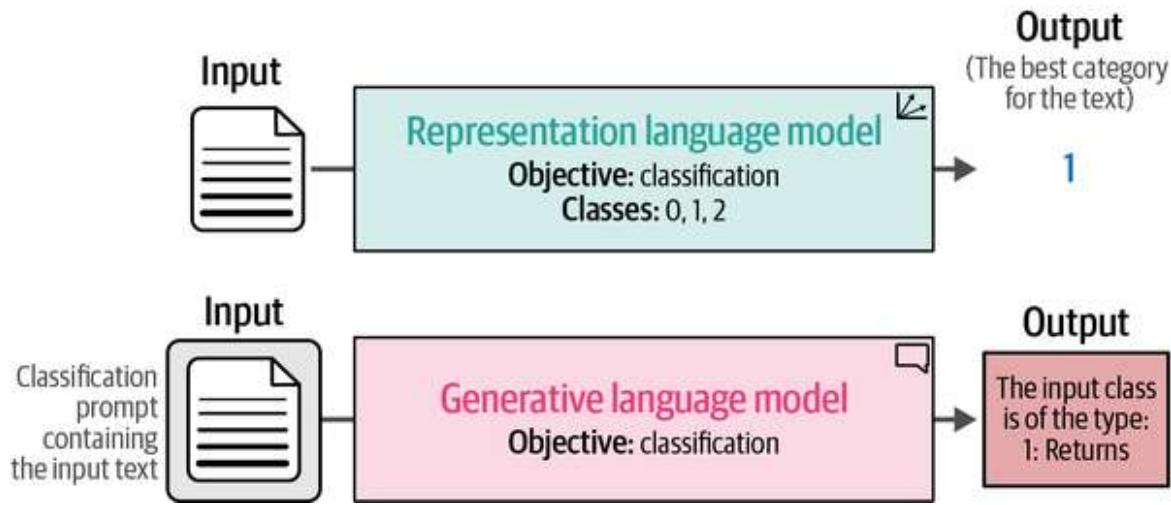


Figure 4-2. Although both representation and generative models can be used for classification, their approaches differ.

This chapter serves as an introduction to a variety of language models, both generative and nongenerative. We will encounter common packages for loading and using these models.

TIP

Although this book focuses on LLMs, it is highly advised to compare these examples against classic, but strong baselines such as representing text with TF-IDF and training a logistic regression classifier on top of that.

The Sentiment of Movie Reviews

You can find the data we use to explore techniques for classifying text on the Hugging Face Hub, a platform for hosting models but also [data](#). We will use the well-known “[rotten_tomatoes](#)” [dataset](#) to train and evaluate our models.¹ It contains 5,331 positive and 5,331 negative movie reviews from Rotten Tomatoes.

To load this data, we make use of the `datasets` package, which will be used throughout the book:

```
from datasets import load_dataset

# Load our data
data = load_dataset("rottentomatoes")
data

DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 8530
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 1066
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 1066
    })
})
```

The data is split up into *train*, *test*, and *validation* splits. Throughout this chapter, we will use the *train* split when we train a model and the *test* split for validating the results. Note that the additional *validation* split can be used to further validate generalization if you used the *train* and *test* splits to perform hyperparameter tuning.

Let's take a look at some examples in our *train* split:

```
data["train"][0, -1]

{'text': ['the rock is destined to be the 21st century\'s new "conan" and that he\'s going to make a splash even greater than arnold schwarzenegger , jean-claud van damme or steven segal .',
          'things really get weird , though not particularly scary : the movie is all portent and no content .'],
 'label': [1, 0]}
```

These short reviews are either labeled as positive (1) or negative (0). This means that we will focus on binary sentiment classification.

Text Classification with Representation Models

Classification with pretrained representation models generally comes in two flavors, either using a task-specific model or an embedding model. As we explored in the previous chapter, these models are created by fine-tuning a foundation model, like BERT, on a specific downstream task as illustrated in Figure 4-3.

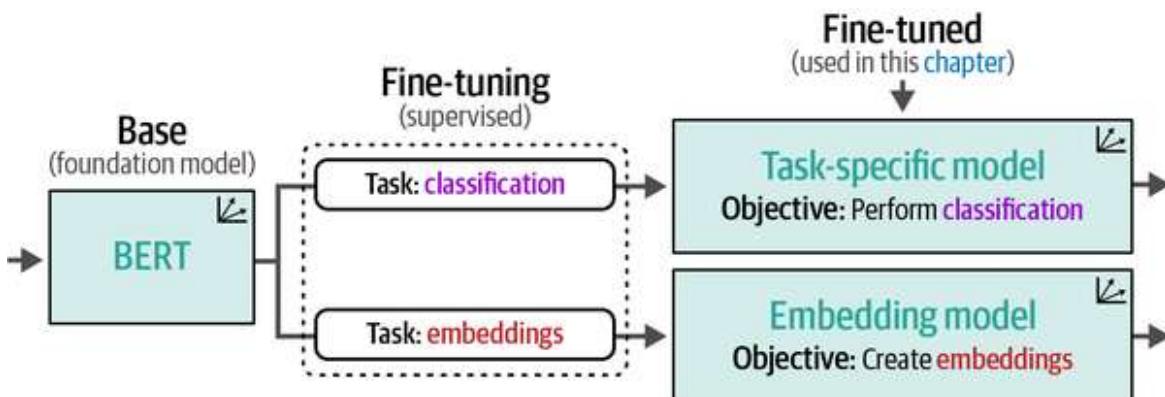


Figure 4-3. A foundation model is fine-tuned for specific tasks; for instance, to perform classification or generate general-purpose embeddings.

A task-specific model is a representation model, such as BERT, trained for a specific task, like sentiment analysis. As we explored in [Chapter 1](#), an embedding model generates general-purpose embeddings that can be used for a variety of tasks not limited to classification, like semantic search (see [Chapter 8](#)).

The process of fine-tuning a BERT model for classification is covered in [Chapter 11](#) while creating an embedding model is covered in [Chapter 10](#). In this chapter, we keep both models *frozen* (nontrainable) and only use their output as shown in [Figure 4-4](#).

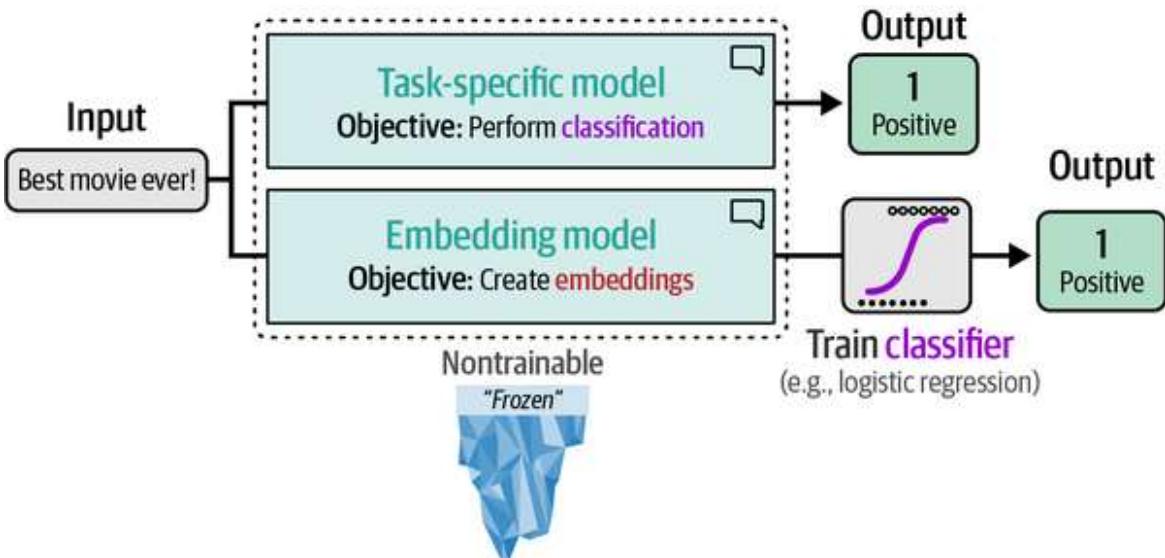


Figure 4-4. Perform classification directly with a task-specific model or indirectly with general-purpose embeddings.

We will leverage pretrained models that others have already fine-tuned for us and explore how they can be used to classify our selected movie reviews.

Model Selection

Choosing the right models is not as straightforward as you might think with over 60,000 models on the [Hugging Face Hub for text classification](#) and more than 8,000 models that [generate embeddings](#) at the moment of writing. Moreover, it's crucial to select a model that fits your use case and consider its language compatibility, the underlying architecture, size, and performance.

Let's start with the underlying architecture. As we explored in [Chapter 1](#), BERT, a well-known encoder-only architecture, is a popular choice for creating task-specific and embedding models. While generative models, like the GPT family, are incredible models, encoder-only models similarly excel in task-specific use cases and tend to be significantly smaller in size.

Over the years, many variations of BERT have been developed, including RoBERTa,² DistilBERT,³ ALBERT,⁴ and DeBERTa,⁵ each trained in

various contexts. You can find an overview of some well-known BERT-like models in [Figure 4-5](#).

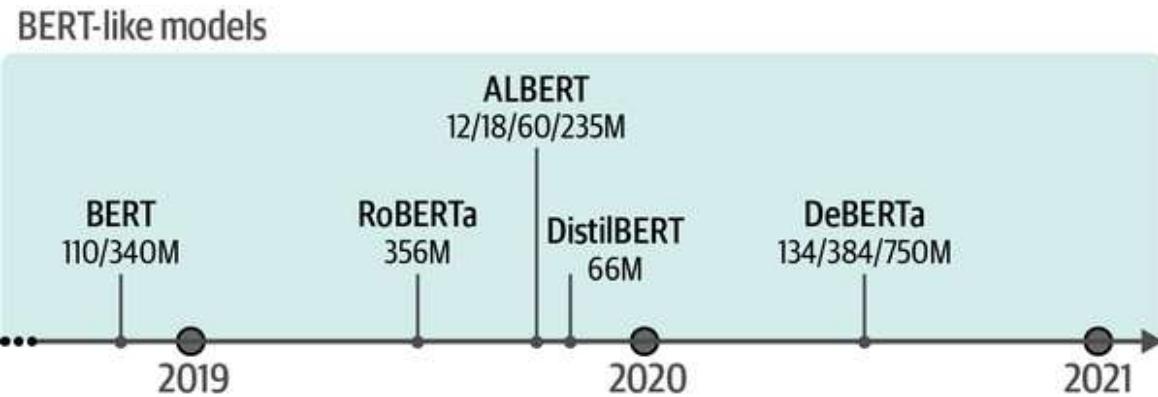


Figure 4-5. A timeline of common BERT-like model releases. These are considered foundation models and are mostly intended to be fine-tuned on a downstream task.

Selecting the right model for the job can be a form of art in itself. Trying thousands of pretrained models that can be found on Hugging Face's Hub is not feasible so we need to be efficient with the models that we choose. Having said that, several models are great starting points and give you an idea of the base performance of these kinds of models. Consider them solid baselines:

- BERT base model (uncased)
- RoBERTa base model
- DistilBERT base model (uncased)
- DeBERTa base model
- bert-tiny
- ALBERT base v2

For the task-specific model, we are choosing the [Twitter-RoBERTa-base for Sentiment Analysis](#) model. This is a RoBERTa model fine-tuned on tweets for sentiment analysis. Although this was not trained specifically for movie reviews, it is interesting to explore how this model generalizes.

When selecting models to generate embeddings from, [the MTEB leaderboard](#) is a great place to start. It contains open and closed source models benchmarked across several tasks. Make sure to not only take performance into account. The importance of inference speed should not be underestimated in real-life solutions. As such, we will use [sentence-transformers/all-mnlp-base-v2](#) as the embedding throughout this section. It is a small but performant model.

Using a Task-Specific Model

Now that we have selected our task-specific representation model, let's start by loading our model:

```
from transformers import pipeline

# Path to our HF model
model_path = "cardiffnlp/twitter-roberta-base-sentiment-latest"

# Load model into pipeline
pipe = pipeline(
    model=model_path,
    tokenizer=model_path,
    return_all_scores=True,
    device="cuda:0"
)
```

As we load our model, we also load the *tokenizer*, which is responsible for converting input text into individual tokens, as illustrated in [Figure 4-6](#). Although that parameter is not needed as it is loaded automatically, it illustrates what is happening under the hood.

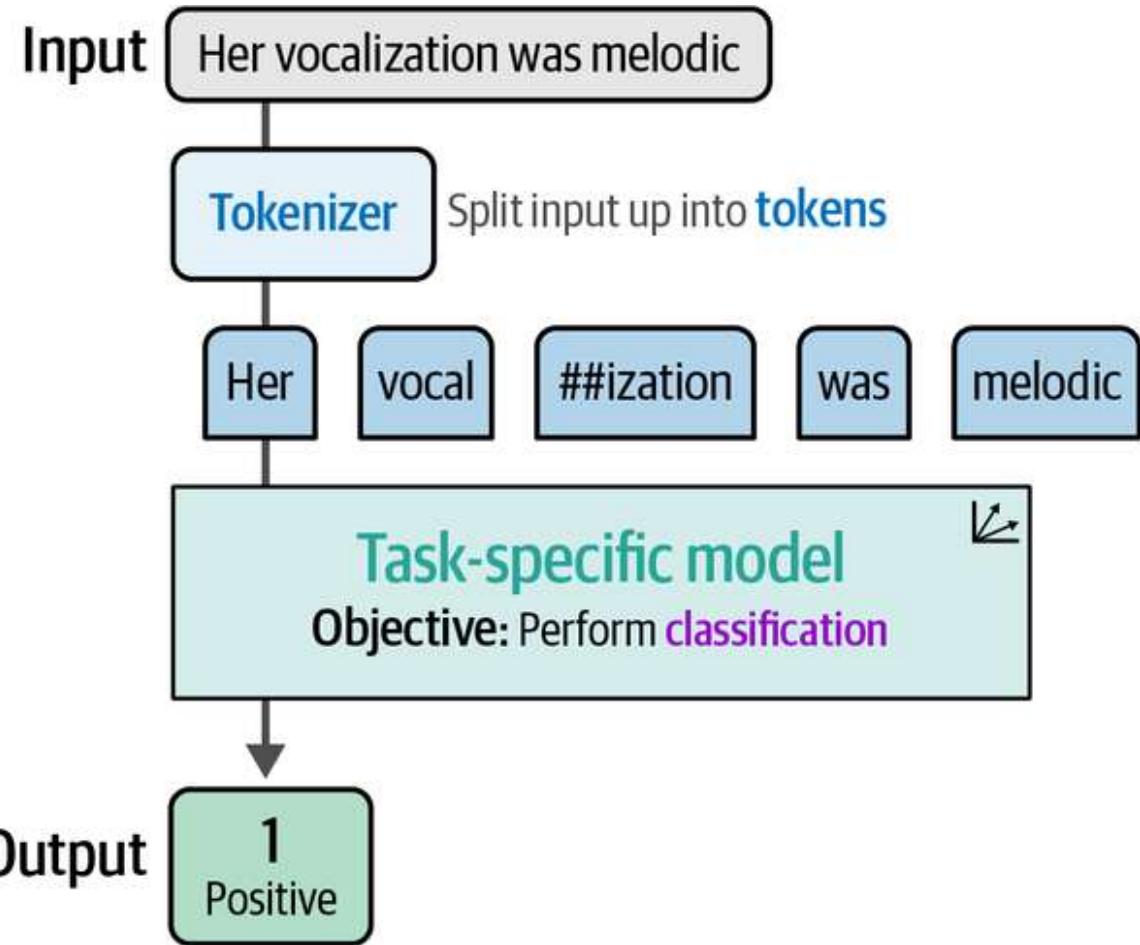


Figure 4-6. An input sentence is first fed to a tokenizer before it can be processed by the task-specific model.

These tokens are at the core of most language models, as explored in depth in [Chapter 2](#). A major benefit of these tokens is that they can be combined to generate representations even if they were not in the training data, as shown in [Figure 4-7](#).

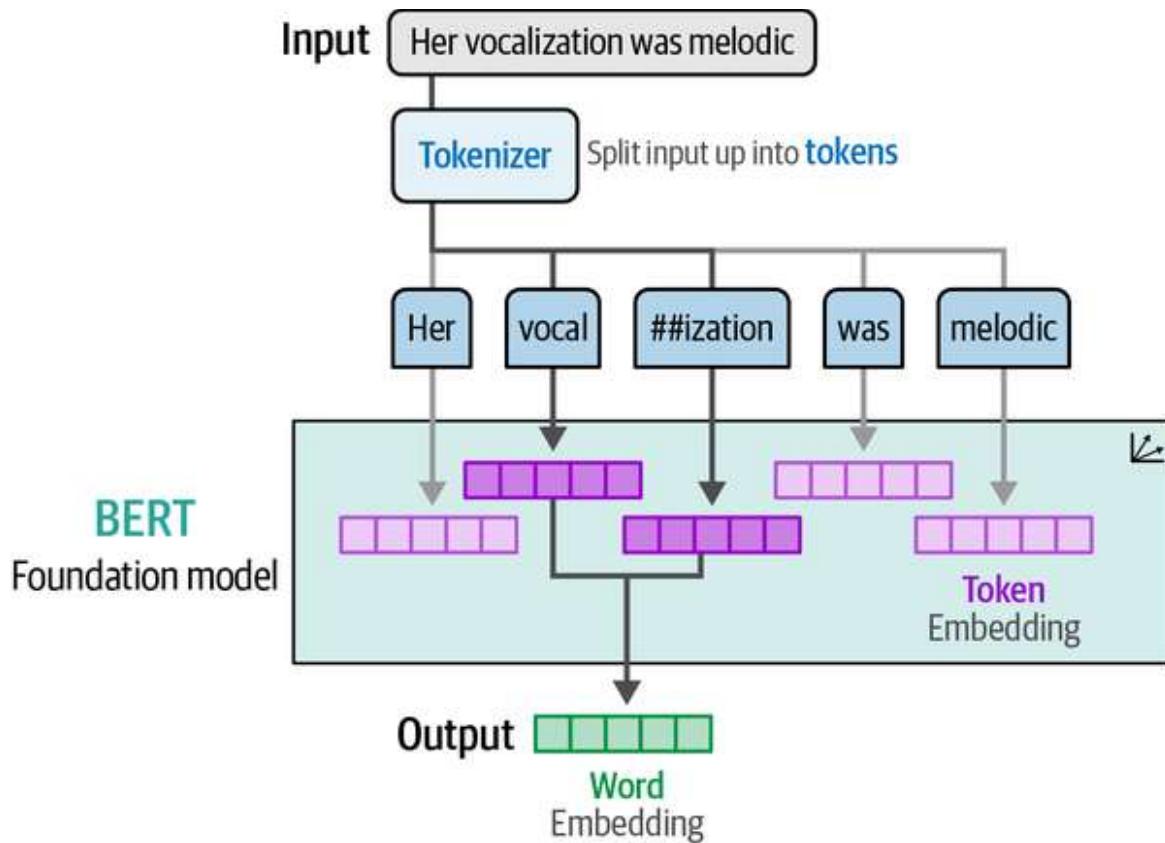


Figure 4-7. By breaking down an unknown word into tokens, word embeddings can still be generated.

After loading all the necessary components, we can go ahead and use our model on the test split of our data:

```
import numpy as np
from tqdm import tqdm
from transformers.pipelines.pt_utils import KeyDataset

# Run inference
y_pred = []
for output in tqdm(pipe(KeyDataset(data["test"], "text")),
total=len(data["test"])):
    negative_score = output[0]["score"]
    positive_score = output[2]["score"]
    assignment = np.argmax([negative_score, positive_score])
    y_pred.append(assignment)
```

Now that we have generated our predictions, all that is left is evaluation. We create a small function that we can easily use throughout this chapter:

```

from sklearn.metrics import classification_report

def evaluate_performance(y_true, y_pred):
    """Create and print the classification report"""
    performance = classification_report(
        y_true, y_pred,
        target_names=["Negative Review", "Positive Review"]
    )
    print(performance)

```

Next, let's create our classification report:

```
evaluate_performance(data["test"]["label"], y_pred)
```

	precision	recall	f1-score	support
Negative Review	0.76	0.88	0.81	533
Positive Review	0.86	0.72	0.78	533
accuracy			0.80	1066
macro avg	0.81	0.80	0.80	1066
weighted avg	0.81	0.80	0.80	1066

To read the resulting classification report, let's first start by exploring how we can identify correct and incorrect predictions. There are four combinations depending on whether we predict something correctly (True) versus incorrectly (False) and whether we predict the correct class (Positive) versus incorrect class (Negative). We can illustrate these combinations as a matrix, commonly referred to as a *confusion matrix*, in Figure 4-8.

		Actual values	
		Positive	Negative
Predicted values	Positive	True positive (TP)	False positive (FP)
	Negative	False negative (FN)	True negative (TN)

Annotations around the matrix:

- Top-left cell (Positive Predicted, Positive Actual): "Positive review correctly classified as positive"
- Top-right cell (Positive Predicted, Negative Actual): "Negative review incorrectly classified as positive"
- Bottom-left cell (Negative Predicted, Positive Actual): "Positive review incorrectly classified as negative"
- Bottom-right cell (Negative Predicted, Negative Actual): "Negative review correctly classified as negative"

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

Using the confusion matrix, we can derive several formulas to describe the quality of the model. In the previously generated classification report we can see four such methods, namely *precision*, *recall*, *accuracy*, and the *F1 score*:

- *Precision* measures how many of the items found are relevant, which indicates the accuracy of the relevant results.
- *Recall* refers to how many relevant classes were found, which indicates its ability to find all relevant results.
- *Accuracy* refers to how many correct predictions the model makes out of all predictions, which indicates the overall correctness of the model.
- The *F1 score* balances both precision and recall to create a model's overall performance.

These four metrics are illustrated in Figure 4-9, which describes them using the aforementioned classification report.

	TP TP + FP	TP TP + FN	$2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$	
Negative review				
Positive review	precision 0.76 0.86	recall 0.88 0.72	f1-score 0.81 0.78	
				Number of samples
				533 533
				1066
				1066
				1066
	$\frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$			
	accuracy 0.80			
	macro avg 0.81	0.80	0.80	
	weighted avg 0.81	0.80	0.80	
Averaged across all classes				

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

We will consider the weighted average of the F1 score throughout the examples in this book to make sure each class is treated equally. Our pretrained BERT model gives us an F1 score of 0.80 (we are reading this from the *weighted avg* row and the *f1-score* column), which is great for a model not trained specifically on our domain data!

To improve the performance of our selected model, we could do a few different things including selecting a model trained on our domain data, movie reviews in this case, like [DistilBERT base uncased finetuned SST-2](#). We could also shift our focus to another flavor of representation models, namely embedding models.

Classification Tasks That Leverage Embeddings

In the previous example, we used a pretrained task-specific model for sentiment analysis. However, what if we cannot find a model that was pretrained for this specific task? Do we need to fine-tune a representation model ourselves? The answer is no!

There might be times when you want to fine-tune the model yourself if you have sufficient computing available (see [Chapter 11](#)). However, not everyone has access to extensive computing. This is where general-purpose embedding models come in.

Supervised Classification

Unlike the previous example, we can perform part of the training process ourselves by approaching it from a more classical perspective. Instead of directly using the representation model for classification, we will use an embedding model for generating features. Those features can then be fed into a classifier, thereby creating a two-step approach as shown in [Figure 4-10](#).

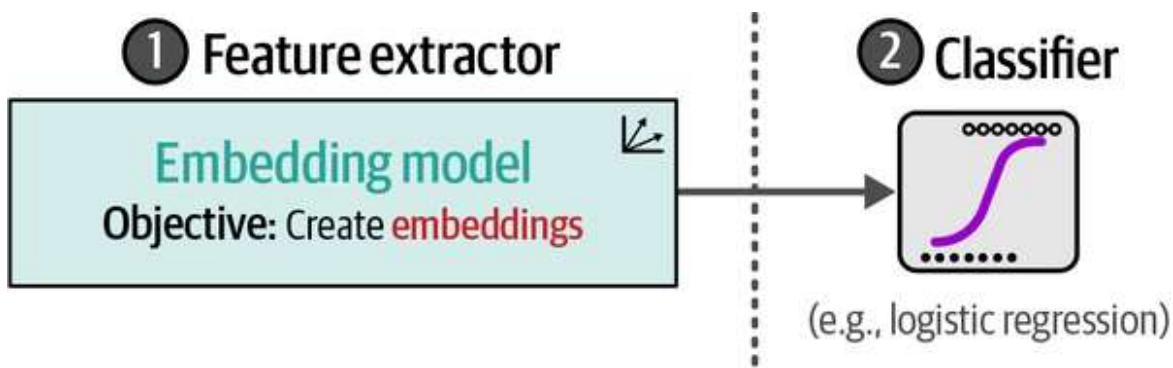


Figure 4-10. The feature extraction step and classification steps are separated.

A major benefit of this separation is that we do not need to fine-tune our embedding model, which can be costly. In contrast, we can train a classifier, like a logistic regression, on the CPU instead.

In the first step, we convert our textual input to embeddings using the embedding model as shown in [Figure 4-11](#). Note that this model is similarly kept *frozen* and is not updated during the training process.

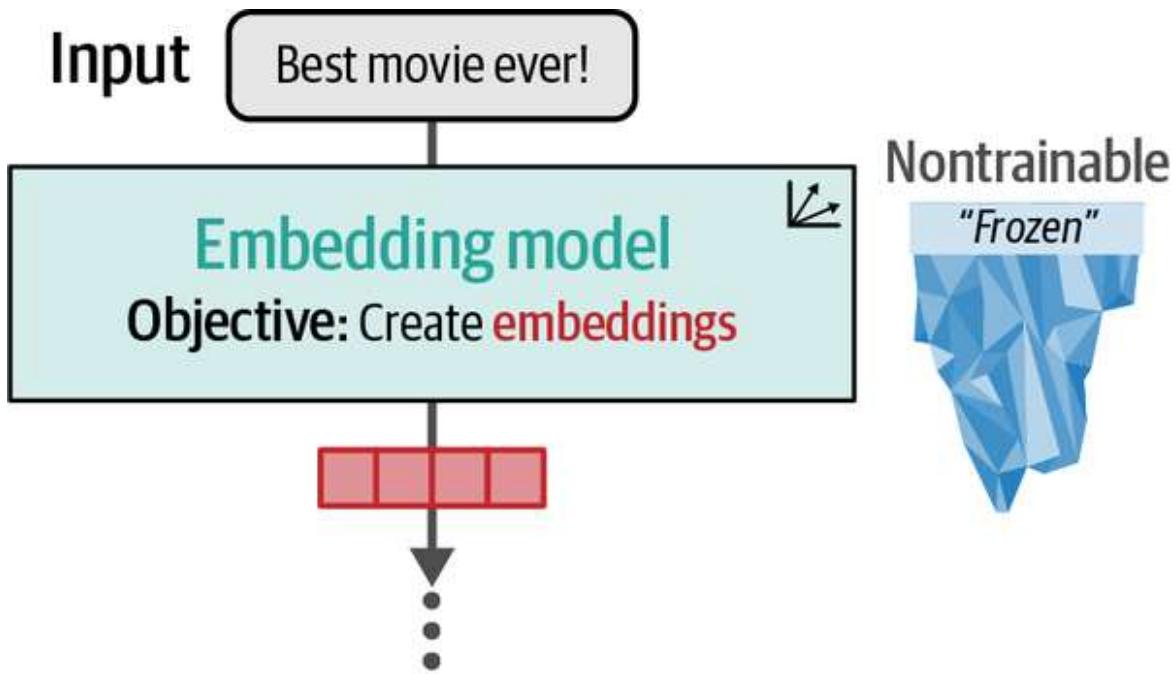


Figure 4-11. In step 1, we use the embedding model to extract the features and convert the input text to embeddings.

We can perform this step with `sentence-transformer`, a popular package for leveraging pretrained embedding models.⁶ Creating the embeddings is straightforward:

```
from sentence_transformers import SentenceTransformer

# Load model
model = SentenceTransformer("sentence-transformers/all-mpnet-base-v2")

# Convert text to embeddings
train_embeddings = model.encode(data["train"]["text"],
                                show_progress_bar=True)
test_embeddings = model.encode(data["test"]["text"],
                                show_progress_bar=True)
```

As we covered in [Chapter 1](#), these embeddings are numerical representations of the input text. The number of values, or dimension, of the embedding depends on the underlying embedding model. Let's explore that for our model:

```
train_embeddings.shape
```

```
(8530, 768)
```

This shows that each of our 8,530 input documents has an embedding dimension of 768 and therefore each embedding contains 768 numerical values.

In the second step, these embeddings serve as the input features to the classifier illustrated in [Figure 4-12](#). The classifier is trainable and not limited to logistic regression and can take on any form as long as it performs classification.

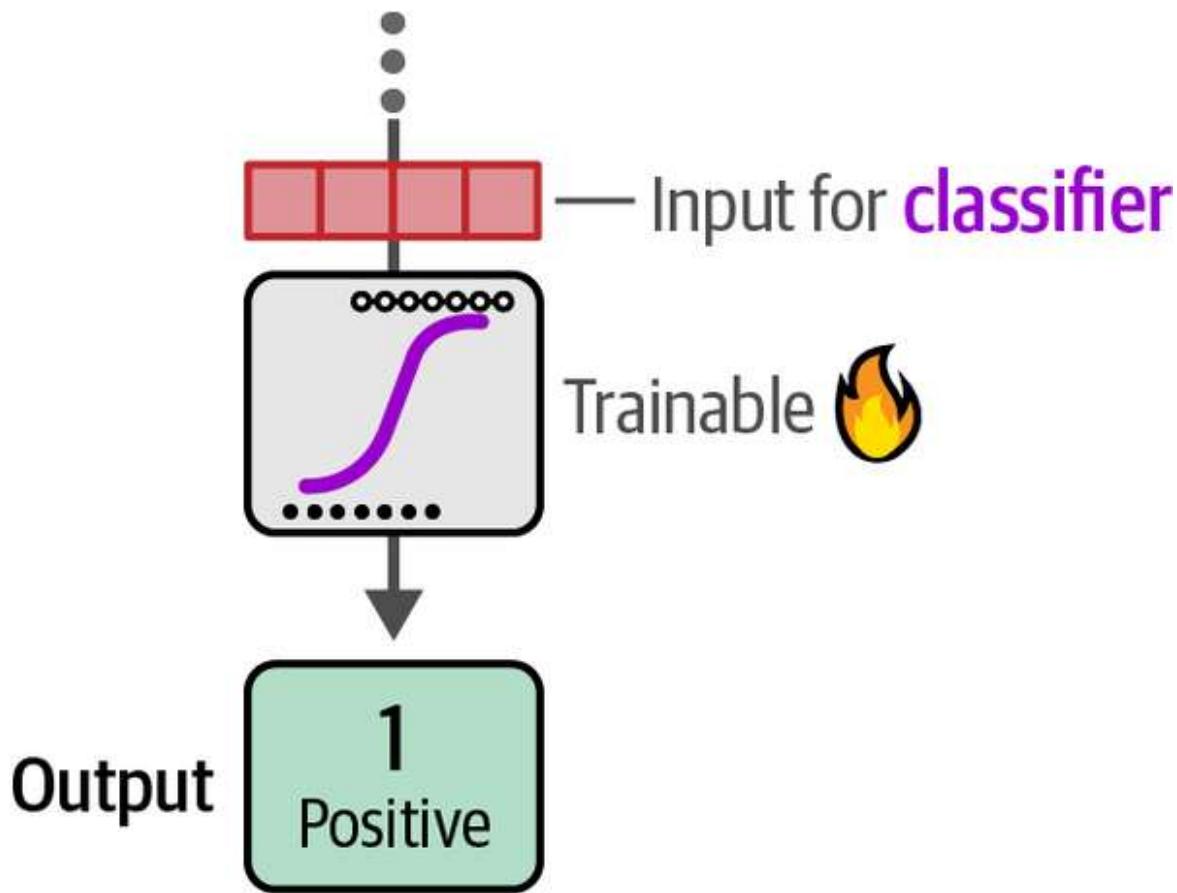


Figure 4-12. Using the embeddings as our features, we train a logistic regression model on our training data.

We will keep this step straightforward and use a logistic regression as the classifier. To train it, we only need to use the generated embeddings

together with our labels:

```
from sklearn.linear_model import LogisticRegression

# Train a logistic regression on our train embeddings
clf = LogisticRegression(random_state=42)
clf.fit(train_embeddings, data["train"]["label"])
```

Next, let's evaluate our model:

```
# Predict previously unseen instances
y_pred = clf.predict(test_embeddings)
evaluate_performance(data["test"]["label"], y_pred)
```

	precision	recall	f1-score	support
Negative Review	0.85	0.86	0.85	533
Positive Review	0.86	0.85	0.85	533
accuracy			0.85	1066
macro avg	0.85	0.85	0.85	1066
weighted avg	0.85	0.85	0.85	1066

By training a classifier on top of our embeddings, we managed to get an F1 score of 0.85! This demonstrates the possibilities of training a lightweight classifier while keeping the underlying embedding model frozen.

TIP

In this example, we used `sentence-transformers` to extract our embeddings, which benefits from a GPU to speed up inference. However, we can remove this GPU dependency by using an external API to create the embeddings. Popular choices for generating embeddings are Cohere's and OpenAI's offerings. As a result, this would allow the pipeline to run entirely on the CPU.

What If We Do Not Have Labeled Data?

In our previous example, we had labeled data that we could leverage, but this might not always be the case in practice. Getting labeled data is a resource-intensive task that can require significant human labor. Moreover, is it actually worthwhile to collect these labels?

To test this, we can perform zero-shot classification, where we have no labeled data to explore whether the task seems feasible. Although we know the definition of the labels (their names), we do not have labeled data to support them. Zero-shot classification attempts to predict the labels of input text even though it was not trained on them, as shown in [Figure 4-13](#).

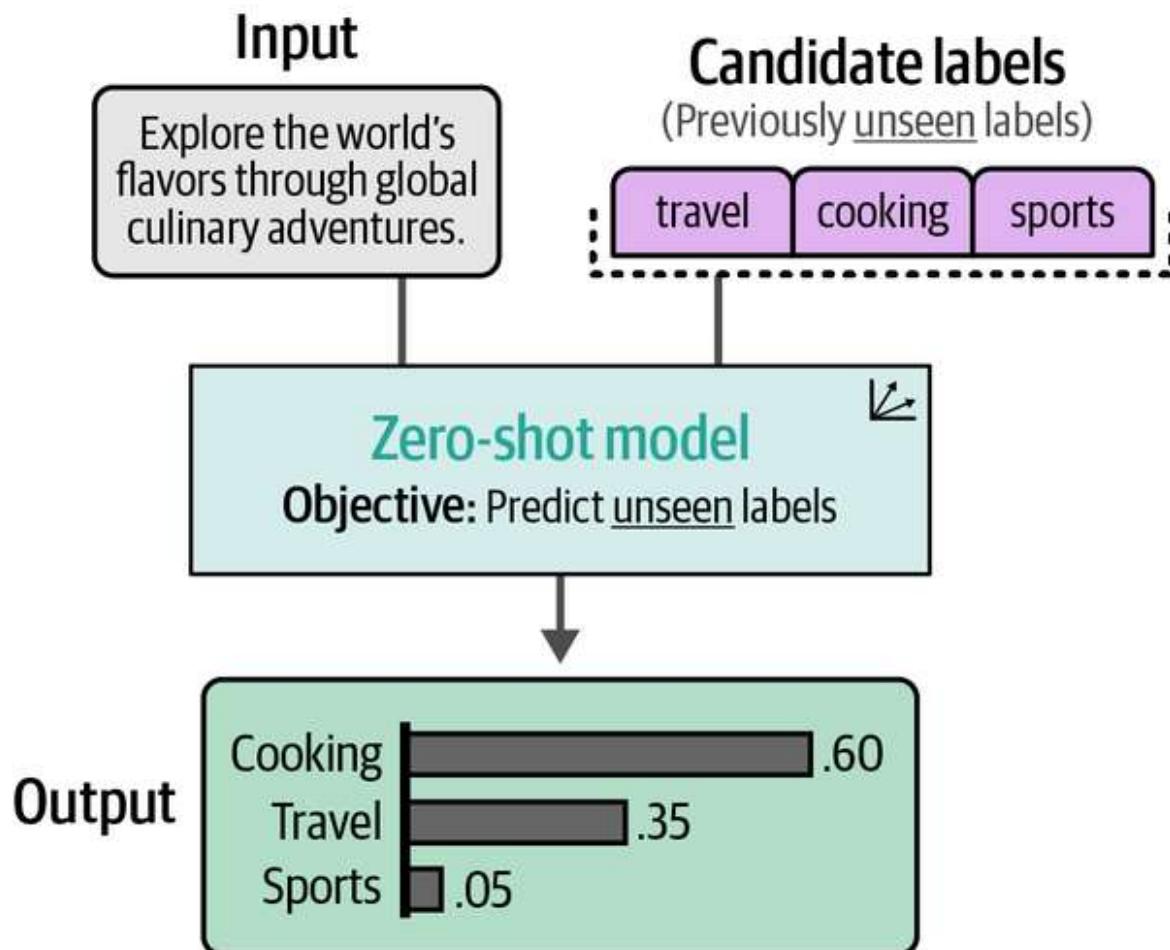


Figure 4-13. In zero-shot classification, we have no labeled data, only the labels themselves. The zero-shot model decides how the input is related to the candidate labels.

To perform zero-shot classification with embeddings, there is a neat trick that we can use. We can describe our labels based on what they should represent. For example, a negative label for movie reviews can be described as “This is a negative movie review.” By describing and embedding the labels and documents, we have data that we can work with. This process, as illustrated in [Figure 4-14](#), allows us to generate our own target labels without the need to actually have any labeled data.

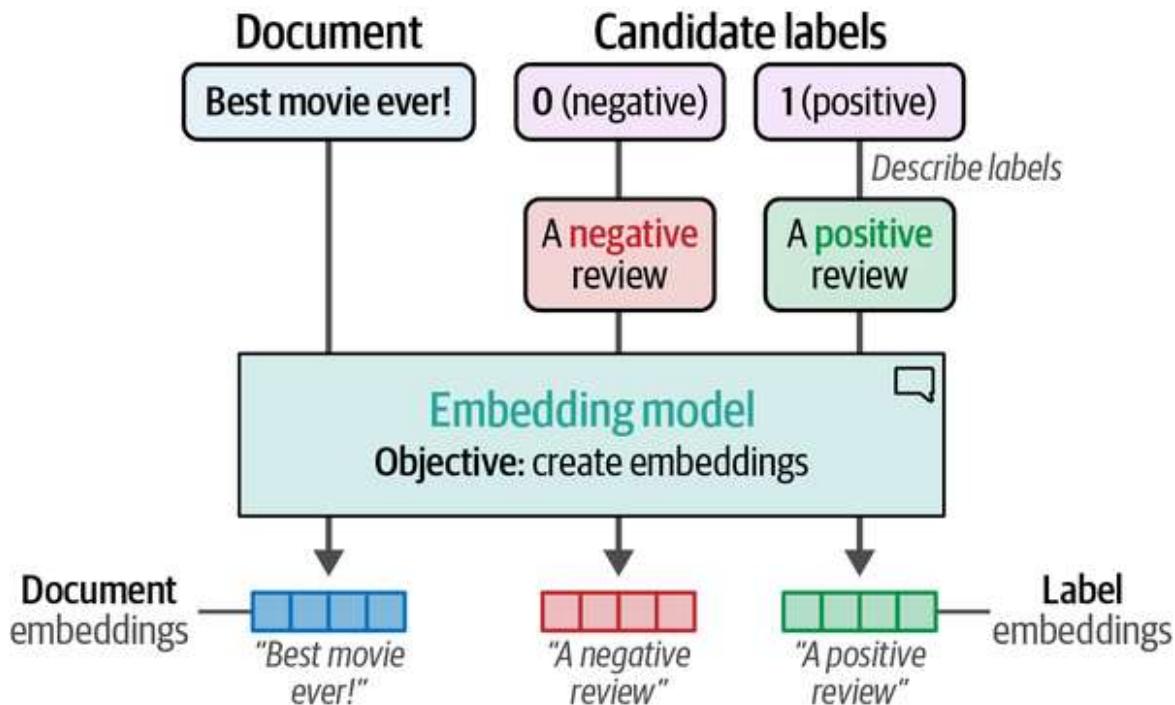


Figure 4-14. To embed the labels, we first need to give them a description, such as “a negative movie review.” This can then be embedded through sentence-transformers.

We can create these label embeddings using the `.encode` function as we did earlier:

```
# Create embeddings for our labels
label_embeddings = model.encode(["A negative review", "A positive review"])
```

To assign labels to documents, we can apply cosine similarity to the document label pairs. This is the cosine of the angle between vectors, which

is calculated through the dot product of the embeddings and divided by the product of their lengths, as illustrated in [Figure 4-15](#).

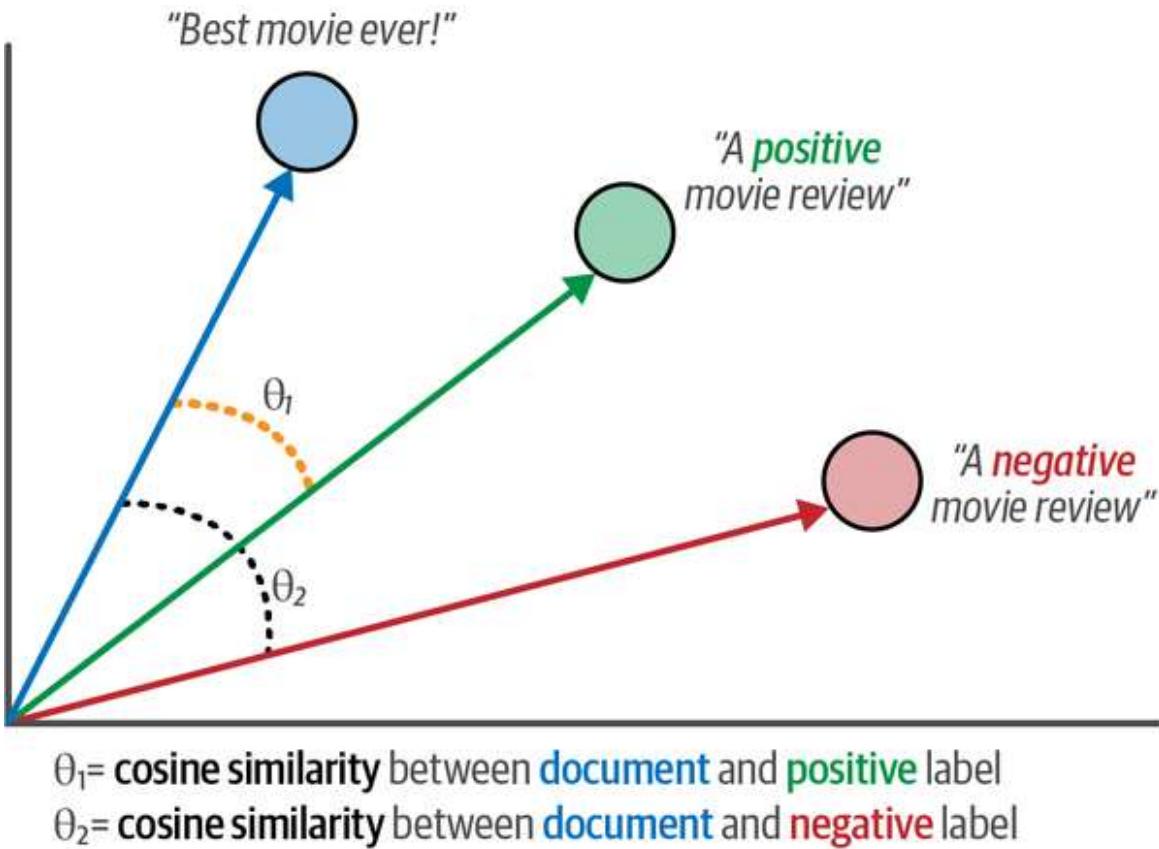


Figure 4-15. The cosine similarity is the angle between two vectors or embeddings. In this example, we calculate the similarity between a document and the two possible labels, positive and negative.

We can use cosine similarity to check how similar a given document is to the description of the candidate labels. The label with the highest similarity to the document is chosen as illustrated in [Figure 4-16](#).

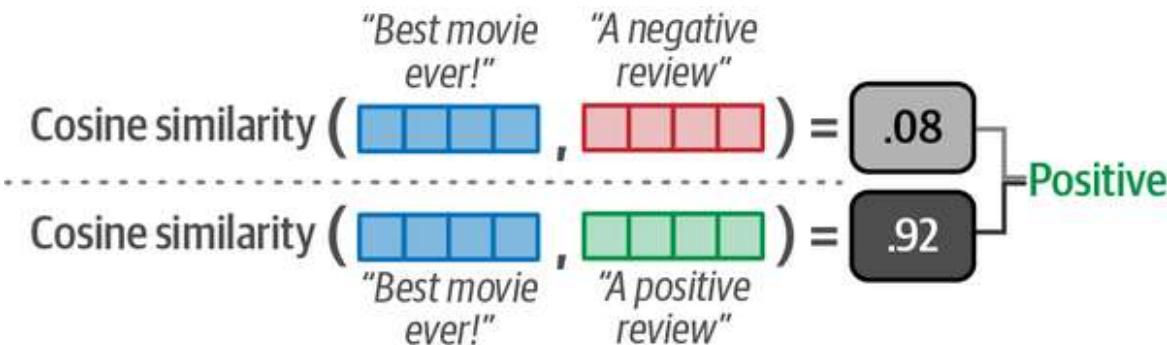


Figure 4-16. After embedding the label descriptions and the documents, we can use cosine similarity for each label-document pair.

To perform cosine similarity on the embeddings, we only need to compare the document embeddings with the label embeddings and get the best matching pairs:

```
from sklearn.metrics.pairwise import cosine_similarity

# Find the best matching label for each document
sim_matrix = cosine_similarity(test_embeddings, label_embeddings)
y_pred = np.argmax(sim_matrix, axis=1)
```

And that is it! We only needed to come up with names for our labels to perform our classification tasks. Let's see how well this method works:

```
evaluate_performance(data["test"]["label"], y_pred)
```

	precision	recall	f1-score	support
Negative Review	0.78	0.77	0.78	533
Positive Review	0.77	0.79	0.78	533
accuracy			0.78	1066
macro avg	0.78	0.78	0.78	1066
weighted avg	0.78	0.78	0.78	1066

NOTE

If you are familiar with **zero-shot classification** with Transformer-based models, you might wonder why we choose to illustrate this with embeddings instead. Although natural language inference models are amazing for zero-shot classification, the example here demonstrates the flexibility of embeddings for a variety of tasks. As you will see throughout the book, embeddings can be found in most Language AI use cases and are often an underestimated but incredibly vital component.

An F1 score of 0.78 is quite impressive considering we did not use any labeled data at all! This just shows how versatile and useful embeddings are, especially if you are a bit creative with how they are used.

TIP

Let's put that creativity to the test. We decided upon "A negative/positive review" as the name of our labels but that can be improved. Instead, we can make them a bit more concrete and specific toward our data by using "A very negative/positive movie review" instead. This way, the embedding will capture that it is a movie review and will focus a bit more on the extremes of the two labels. Try it out and explore how it affects the results.

Text Classification with Generative Models

Classification with generative language models, such as OpenAI's GPT models, works a bit differently from what we have done thus far. These models take as input some text and generative text and are thereby aptly named sequence-to-sequence models. This is in stark contrast to our task-specific model, which outputs a class instead, as illustrated in [Figure 4-17](#).

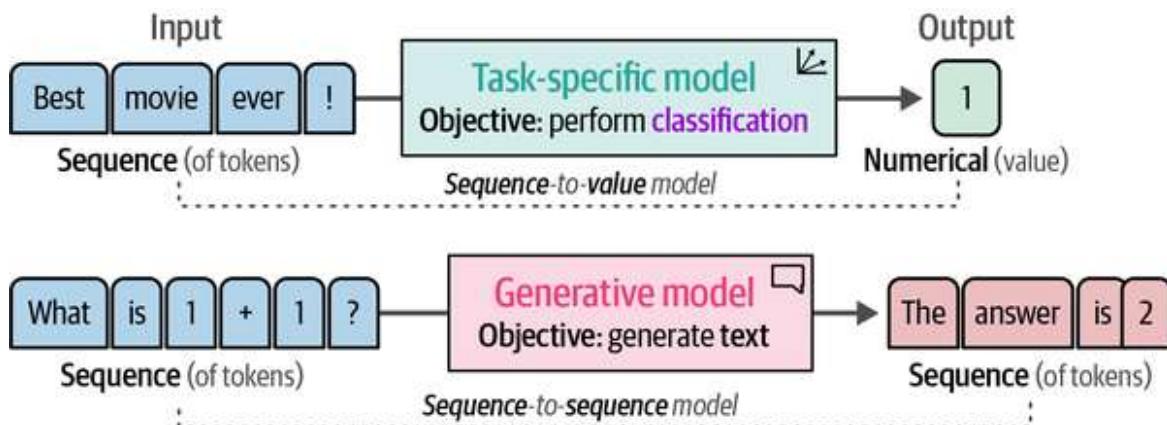


Figure 4-17. A task-specific model generates numerical values from sequences of tokens while a generative model generates sequences of tokens from sequences of tokens.

These generative models are generally trained on a wide variety of tasks and usually do not perform your use case out of the box. For instance, if we give a generative model a movie review without any context, it has no idea what to do with it.

Instead, we need to help it understand the context and guide it toward the answers that we are looking for. As demonstrated in [Figure 4-18](#), this guiding process is done mainly through the instruction, or *prompt*, that you

give such a model. Iteratively improving your prompt to get your preferred output is called *prompt engineering*.

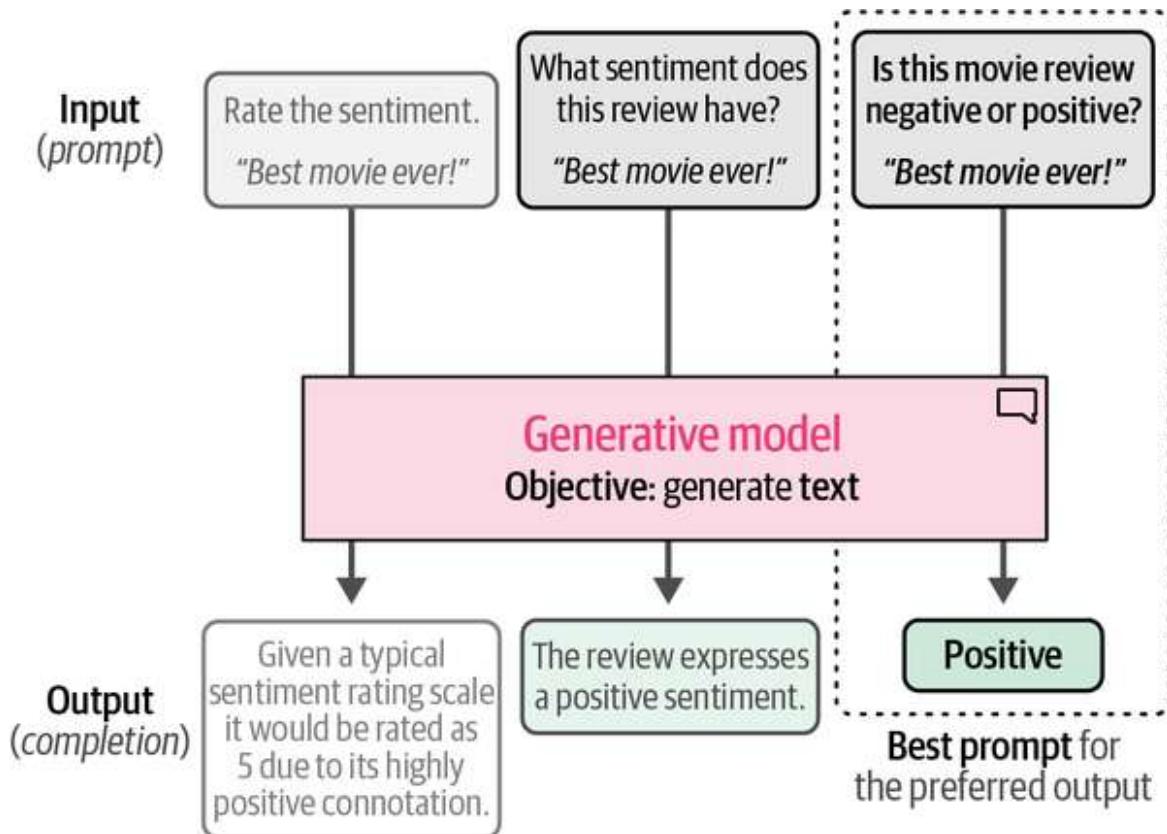


Figure 4-18. Prompt engineering allows prompts to be updated to improve the output generated by the model.

In this section, we will demonstrate how we can leverage different types of generative models to perform classification without our Rotten Tomatoes dataset.

Using the Text-to-Text Transfer Transformer

Throughout this book, we will explore mostly encoder-only (representation) models like BERT and decoder-only (generative) models like ChatGPT. However, as discussed in [Chapter 1](#), the original Transformer architecture actually consists of an encoder-decoder architecture. Like the decoder-only models, these encoder-decoder models are sequence-to-sequence models and generally fall in the category of generative models.

An interesting family of models that leverage this architecture is the Text-to-Text Transfer Transformer or T5 model. Illustrated in [Figure 4-19](#), its architecture is similar to the original Transformer where 12 decoders and 12 encoders are stacked together.⁷

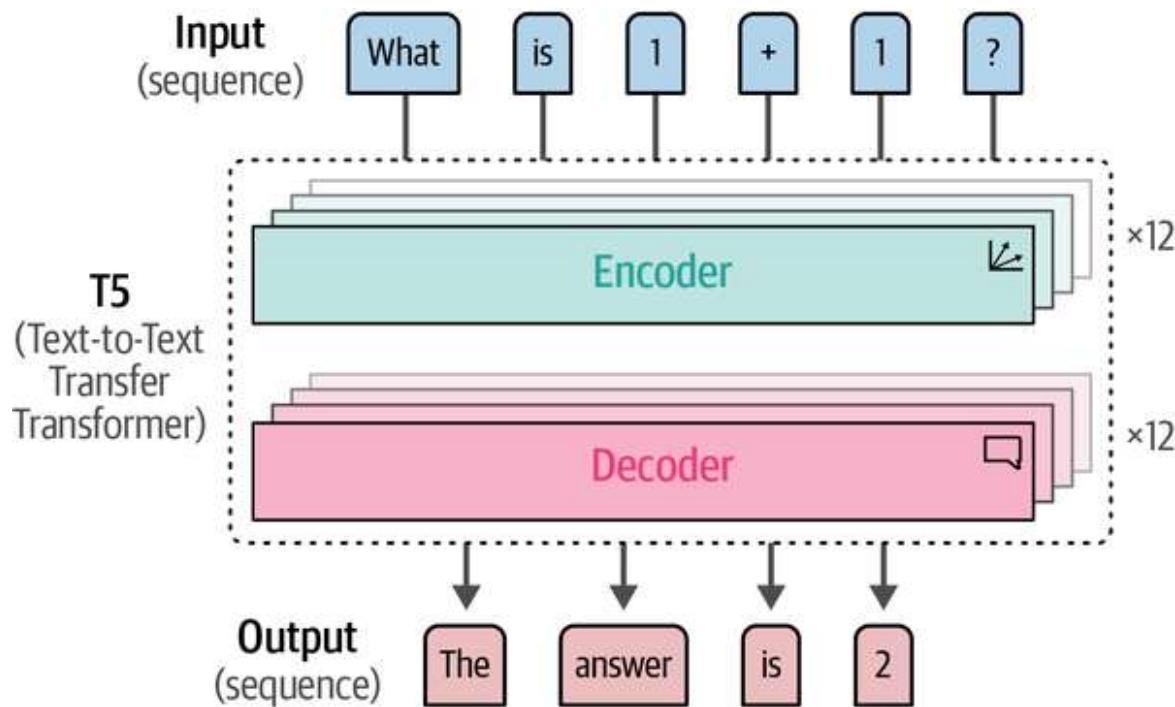


Figure 4-19. The T5 architecture is similar to the original Transformer model, a decoder-encoder architecture.

With this architecture, these models were first pretrained using masked language modeling. In the first step of training, illustrated in [Figure 4-20](#), instead of masking individual tokens, sets of tokens (or *token spans*) were masked during pretraining.

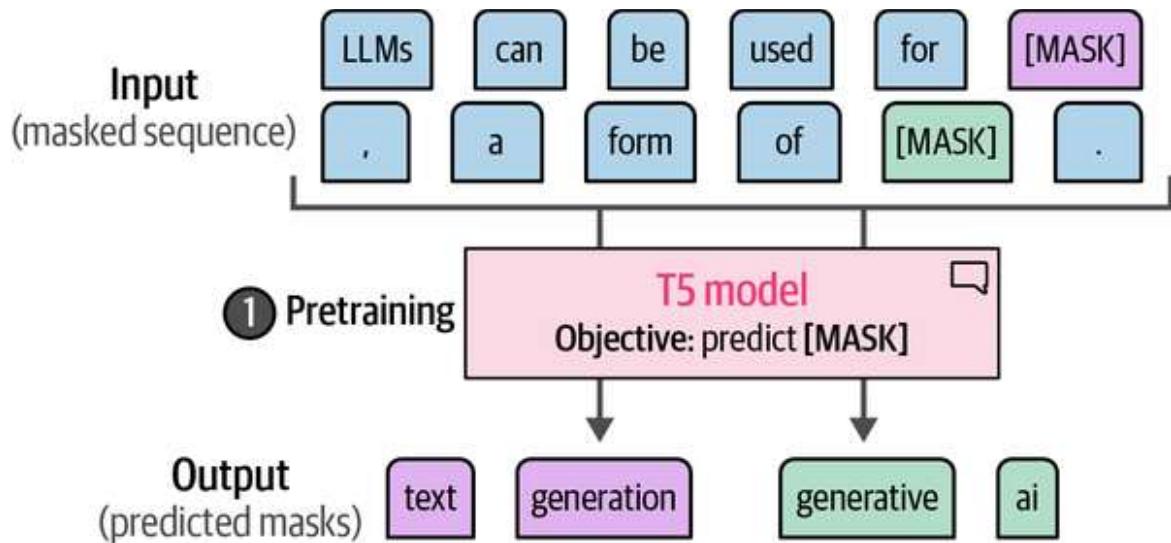


Figure 4-20. In the first step of training, namely pretraining, the T5 model needs to predict masks that could contain multiple tokens.

The second step of training, namely fine-tuning the base model, is where the real magic happens. Instead of fine-tuning the model for one specific task, each task is converted to a sequence-to-sequence task and trained simultaneously. As illustrated in Figure 4-21, this allows the model to be trained on a wide variety of tasks.

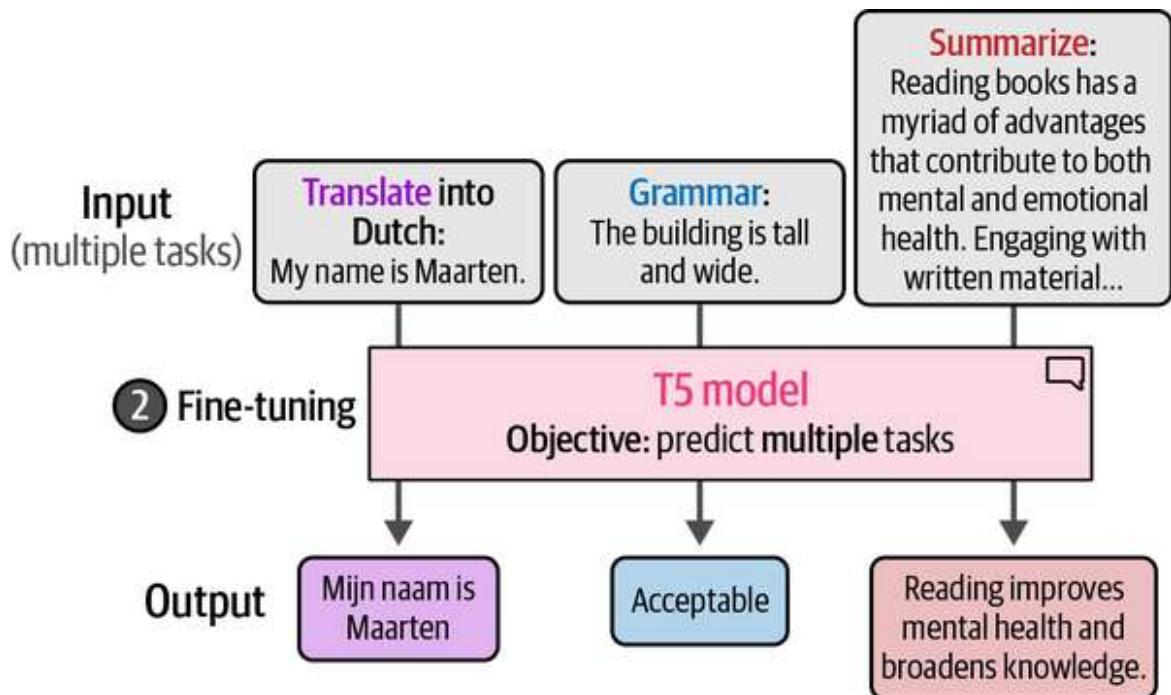


Figure 4-21. By converting specific tasks to textual instructions, the T5 model can be trained on a variety of tasks during fine-tuning.

This method of fine-tuning was extended in the paper “[Scaling instruction-finetuned language models](#)”, which introduced more than a thousand tasks during fine-tuning that more closely follow instructions as we know them from GPT models.⁸ This resulted in the Flan-T5 family of models that benefit from this large variety of tasks.

To use this pretrained Flan-T5 model for classification, we will start by loading it through the "text2text-generation" task, which is generally reserved for these encoder-decoder models:

```
# Load our model
pipe = pipeline(
    "text2text-generation",
    model="google/flan-t5-small",
    device="cuda:0"
)
```

The Flan-T5 model comes in various sizes (flan-t5-small/base/large/xl/xxl) and we will use the smallest to speed things up a bit. However, feel free to play around with larger models to see if you can improve the results.

Compared to our task-specific model, we cannot just give the model some text and hope it will output the sentiment. Instead, we will have to instruct the model to do so.

Thus, we prefix each document with the prompt “Is the following sentence positive or negative?”:

```
# Prepare our data
prompt = "Is the following sentence positive or negative? "
data = data.map(lambda example: {"t5": prompt + example['text']})
data

DatasetDict({
    train: Dataset({
        features: ['text', 'label', 't5'],
        num_rows: 8530
    })
    validation: Dataset({
```

```

        features: ['text', 'label', 't5'],
        num_rows: 1066
    })
test: Dataset({
    features: ['text', 'label', 't5'],
    num_rows: 1066
})
})

```

After creating our updated data, we can run the pipeline similar to the task-specific example:

```

# Run inference
y_pred = []
for output in tqdm(pipe(KeyDataset(data["test"]), "t5")),
total=len(data["test"])):
    text = output[0]["generated_text"]
    y_pred.append(0 if text == "negative" else 1)

```

Since this model generates text, we did need to convert the textual output to numerical values. The output word “negative” was mapped to 0 whereas “positive” was mapped to 1.

These numerical values now allow us to test the quality of the model in the same way we have done before:

```
evaluate_performance(data["test"]["label"], y_pred)
```

	precision	recall	f1-score	support
Negative Review	0.83	0.85	0.84	533
Positive Review	0.85	0.83	0.84	533
accuracy			0.84	1066
macro avg	0.84	0.84	0.84	1066
weighted avg	0.84	0.84	0.84	1066

With an F1 score of 0.84, it is clear this Flan-T5 model is an amazing first look into the capabilities of generative models.

ChatGPT for Classification

Although we focus throughout the book on open source models, another major component of the Language AI field is closed sourced models; in particular, ChatGPT.

Although the underlying architecture of the original ChatGPT model (GPT-3.5) is not shared, we can assume from its name that it is based on the decoder-only architecture that we have seen in the GPT models thus far.

Fortunately, OpenAI shared [an overview of the training procedure](#) that involved an important component, namely preference tuning. As illustrated in [Figure 4-22](#), OpenAI first manually created the desired output to an input prompt (instruction data) and used that data to create a first variant of its model.

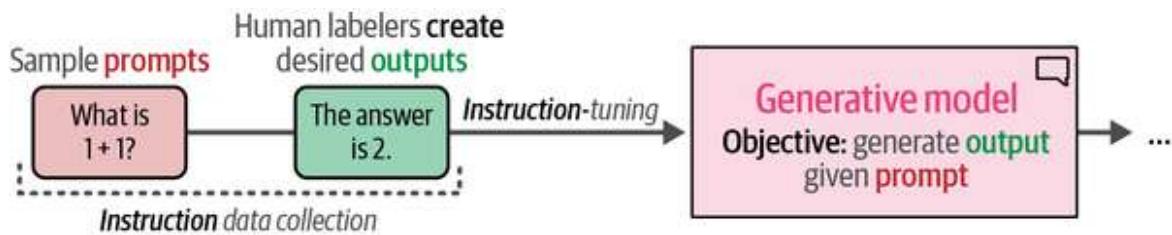


Figure 4-22. Manually labeled data consisting of an instruction (prompt) and output was used to perform fine-tuning (instruction-tuning).

OpenAI used the resulting model to generate multiple outputs that were manually ranked from best to worst. As shown in [Figure 4-23](#), this ranking demonstrates a preference for certain outputs (preference data) and was used to create its final model, ChatGPT.

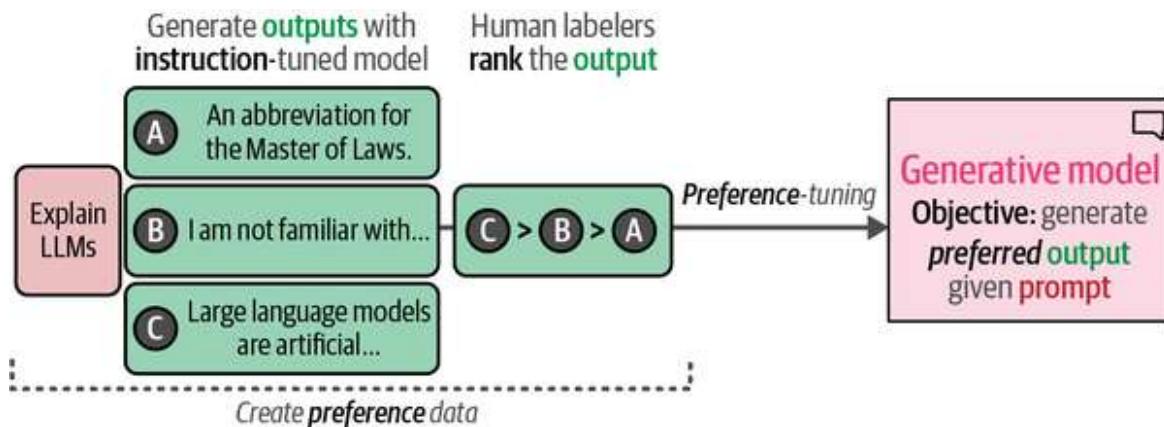


Figure 4-23. Manually ranked preference data was used to generate the final model, ChatGPT.

A major benefit of using preference data over instruction data is the nuance it represents. By demonstrating the difference between a good and better output the generative model learns to generate text that resembles human preference. In [Chapter 12](#), we will explore how these fine-tuning and preference-tuning methodologies work and how you can perform them yourself.

The process of using a closed sourced model is quite different from the open sourced examples we have seen thus far. Instead of loading the model, we can access the model through OpenAI's API.

Before we go into the classification example, you will first need to create a free account on <https://oreil.ly/AEXvA> and create an API key here: <https://oreil.ly/lrTXl>. After doing so, you can use your API to communicate with OpenAI's servers.

We can use this key to create a client:

```
import openai

# Create client
client = openai.OpenAI(api_key="YOUR_KEY_HERE")
```

Using this client, we create the `chatgpt_generation` function, which allows us to generate some text based on a specific prompt, input document, and the selected model:

```

def chatgpt_generation(prompt, document, model="gpt-3.5-turbo-0125"):
    """Generate an output based on a prompt and an input document."""
    messages=[
        {
            "role": "system",
            "content": "You are a helpful assistant."
        },
        {
            "role": "user",
            "content": prompt.replace("[DOCUMENT]", document)
        }
    ]
    chat_completion = client.chat.completions.create(
        messages=messages,
        model=model,
        temperature=0
    )
    return chat_completion.choices[0].message.content

```

Next, we will need to create a template to ask the model to perform the classification:

```

# Define a prompt template as a base
prompt = """Predict whether the following document is a positive or negative movie review:

[DOCUMENT]

If it is positive return 1 and if it is negative return 0. Do not give any other answers.

"""

# Predict the target using GPT
document = "unpretentious , charming , quirky , original"
chatgpt_generation(prompt, document)

```

This template is merely an example and can be changed however you want. For now, we kept it as simple as possible to illustrate how to use such a template.

Before you use this over a potentially large dataset, it is important to always keep track of your usage. External APIs such as OpenAI's offering can quickly become costly if you perform many requests. At the time of writing, running our test dataset using the “gpt-3.5-turbo-0125” model costs 3 cents, which is covered by the free account, but this might change in the future.

TIP

When dealing with external APIs, you might run into rate limit errors. These appear when you call the API too often as some APIs might limit the rate with which you can use it per minute or hour.

To prevent these errors, we can implement several methods for retrying the request, including something referred to as *exponential backoff*. It performs a short sleep each time we hit a rate limit error and then retries the unsuccessful request. Whenever it is unsuccessful again, the sleep length is increased until the request is successful or we hit a maximum number of retries.

To use it with OpenAI, there is [a great guide](#) that can help you get started.

Next, we can run this for all reviews in the test dataset to get its predictions. You can skip this if you want to save your (free) credits for other tasks.

```
# You can skip this if you want to save your (free) credits
predictions = [
    chatgpt_generation(prompt, doc) for doc in tqdm(data["test"])
    ["text"])
]
```

Like the previous example, we need to convert the output from strings to integers to evaluate its performance:

```
# Extract predictions
y_pred = [int(pred) for pred in predictions]
```

```
# Evaluate performance
evaluate_performance(data["test"]["label"], y_pred)
```

	precision	recall	f1-score	support
Negative Review	0.87	0.97	0.92	533
Positive Review	0.96	0.86	0.91	533
accuracy			0.91	1066
macro avg	0.92	0.91	0.91	1066
weighted avg	0.92	0.91	0.91	1066

The F1 score of 0.91 already gives a glimpse into the performance of the model that brought generative AI to the masses. However, since we do not know what data the model was trained on, we cannot easily use these kinds of metrics for evaluating the model. For all we know, it might have actually been trained on our dataset!

In [Chapter 12](#), we will explore how we can evaluate both open source and closed source models on more generalized tasks.

Summary

In this chapter, we discussed many different techniques for performing a wide variety of classification tasks, from fine-tuning your entire model to no tuning at all! Classifying textual data is not as straightforward as it may seem on the surface and there is an incredible amount of creative techniques for doing so.

In this chapter, we explored text classification using both generative and representation language models. Our goal was to assign a label or class to input text for the classification of a review's sentiment.

We explored two types of representation models, a task-specific model and an embedding model. The task-specific model was pretrained on a large dataset specifically for sentiment analysis and showed us that pretrained models are a great technique for classifying documents. The embedding

model was used to generate multipurpose embeddings that we used as the input to train a classifier.

Similarly, we explored two types of generative models, an open source encoder-decoder model (Flan-T5) and a closed source decoder-only model (GPT-3.5). We used these generative models in text classification without requiring specific (additional) training on domain data or labeled datasets.

In the next chapter, we will continue with classification but focus instead on unsupervised classification. What can we do if we have textual data without any labels? What information can we extract? We will focus on clustering our data as well as naming the clusters with topic modeling techniques.

-
- ¹ Bo Pang and Lillian Lee. “Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales.” *arXiv preprint cs/0506075* (2005).
 - ² Yinhan Liuet et al. “RoBERTa: A robustly optimized BERT pretraining approach.” *arXiv preprint arXiv:1907.11692* (2019).
 - ³ Victor Sanh et al. “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.” *arXiv preprint arXiv:1910.01108* (2019).
 - ⁴ Zhenzhong Lan et al. “ALBERT: A lite BERT for self-supervised learning of language representations.” *arXiv preprint arXiv:1909.11942* (2019).
 - ⁵ Pengcheng He et al. “DeBERTa: Decoding-enhanced BERT with disentangled attention.” *arXiv preprint arXiv:2006.03654* (2020).
 - ⁶ Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence embeddings using Siamese BERT-networks.” *arXiv preprint arXiv:1908.10084* (2019).
 - ⁷ Colin Raffel et al. “Exploring the limits of transfer learning with a unified text-to-text transformer.” *The Journal of Machine Learning Research* 21.1 (2020): 5485–5551.
 - ⁸ Hyung Won Chung et al. “Scaling instruction-finetuned language models.” *arXiv preprint arXiv:2210.11416* (2022).

Chapter 5. Text Clustering and Topic Modeling

Although supervised techniques, such as classification, have reigned supreme over the last few years in the industry, the potential of unsupervised techniques such as text clustering cannot be understated.

Text clustering aims to group similar texts based on their semantic content, meaning, and relationships. As illustrated in [Figure 5-1](#), the resulting clusters of semantically similar documents not only facilitate efficient categorization of large volumes of unstructured text but also allow for quick exploratory data analysis.

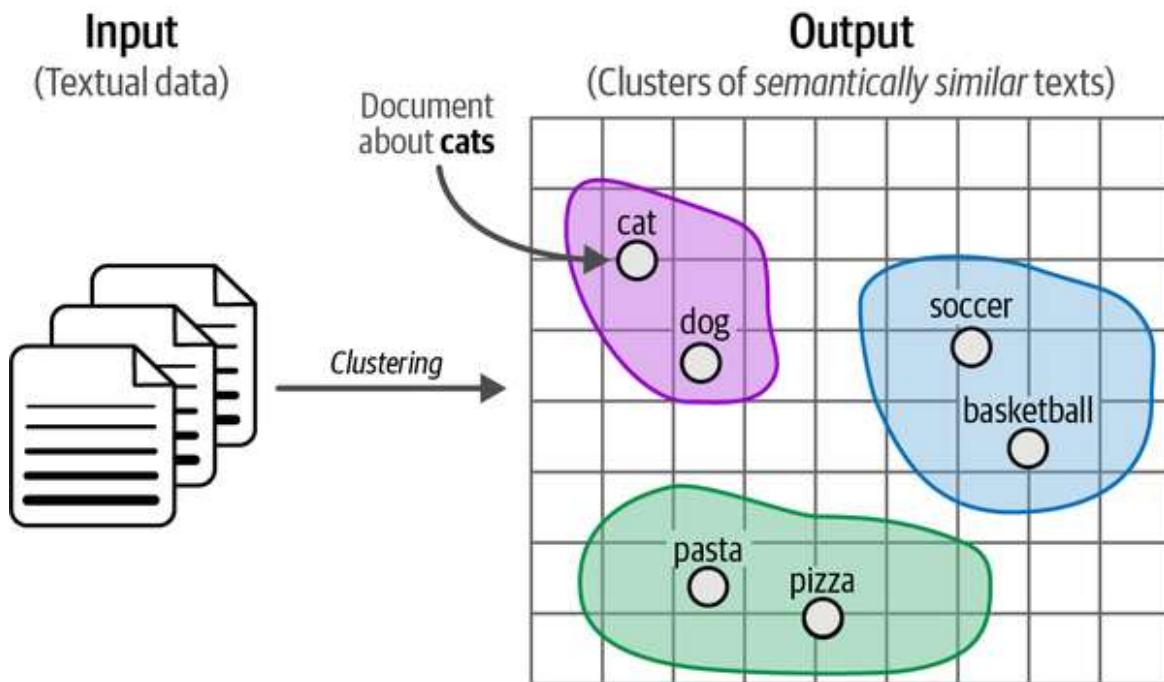


Figure 5-1. Clustering unstructured textual data.

The recent evolution of language models, which enable contextual and semantic representations of text, has enhanced the effectiveness of text clustering. Language is more than a bag of words, and recent language models have proved to be quite capable of capturing that notion. Text

clustering, unbound by supervision, allows for creative solutions and diverse applications, such as finding outliers, speedup labeling, and finding incorrectly labeled data.

Text clustering has also found itself in the realm of topic modeling, where we want to discover (abstract) topics that appear in large collections of textual data. As shown in [Figure 5-2](#), we generally describe a topic using keywords or keyphrases and, ideally, have a single overarching label.

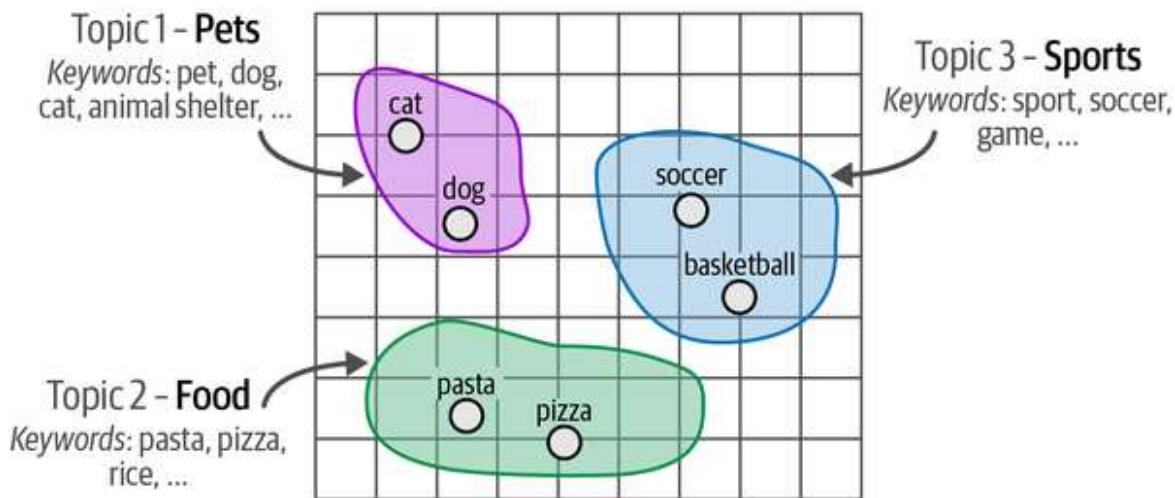


Figure 5-2. Topic modeling is a way to give meaning to clusters of textual documents.

In this chapter, we will first explore how to perform clustering with embedding models and then transition to a text-clustering-inspired method of topic modeling, namely BERTopic.

Text clustering and topic modeling have an important role in this book as they explore creative ways to combine a variety of different language models. We will explore how combining encoder-only (embeddings), decoder-only (generative), and even classical methods (bag-of-words) can result in amazing new techniques and pipelines.

ArXiv's Articles: Computation and Language

Throughout this chapter, we will be running clustering and topic modeling algorithms on ArXiv articles. [ArXiv](#) is an open-access platform for scholarly articles, mostly in the fields of computer science, mathematics,

and physics. We will explore articles in the field of Computation and Language to keep with the theme of this book. The **dataset** contains 44,949 abstracts between 1991 and 2024 from ArXiv's cs.CL (Computation and Language) section.

We load the data and create separate variables for the abstracts, titles, and years of each article:

```
# Load data from Hugging Face
from datasets import load_dataset
dataset = load_dataset("maartengr/arxiv_nlp")["train"]

# Extract metadata
abstracts = dataset["Abstracts"]
titles = dataset["Titles"]
```

A Common Pipeline for Text Clustering

Text clustering allows for discovering patterns in data that you may or may not be familiar with. It allows for getting an intuitive understanding of the task, for example, a classification task, but also of its complexity. As a result, text clustering can become more than just a quick method for exploratory data analysis.

Although there are many methods for text clustering, from graph-based neural networks to centroid-based clustering techniques, a common pipeline that has gained popularity involves three steps and algorithms:

1. Convert the input documents to embeddings with an *embedding model*.
2. Reduce the dimensionality of embeddings with a *dimensionality reduction model*.
3. Find groups of semantically similar documents with a *cluster model*.

Embedding Documents

The first step is to convert our textual data to embeddings, as illustrated in [Figure 5-3](#). Recall from previous chapters that embeddings are numerical representations of text that attempt to capture its meaning.

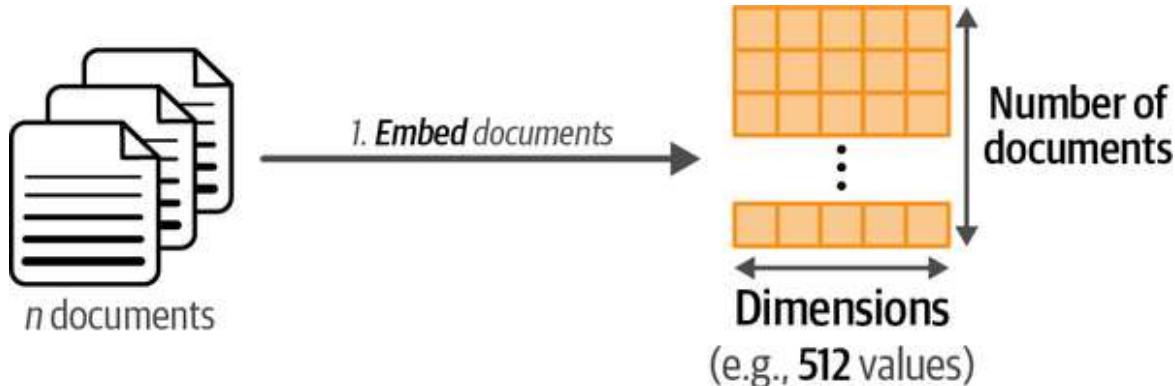


Figure 5-3. Step 1: We convert documents to embeddings using an embedding model.

Choosing embedding models optimized for semantic similarity tasks is especially important for clustering as we attempt to find groups of semantically similar documents. Fortunately, most embedding models at the time of writing focus on just that, semantic similarity.

As we did in the previous chapter, we will use [the MTEB leaderboard](#) to select an embedding model. We will need an embedding model that has a decent score on clustering tasks but also is small enough to run quickly. Instead of using the “sentence-transformers/all-mpnet-base-v2” model we used in the previous chapter, we use the “[thenlper/gte-small](#)” model instead. It is a more recent model that outperforms the previous model on clustering tasks and due to its small size is even faster for inference. However, feel free to play around with newer models that have been released since!

```
from sentence_transformers import SentenceTransformer

# Create an embedding for each abstract
embedding_model = SentenceTransformer("thenlper/gte-small")
embeddings = embedding_model.encode(abstracts,
show_progress_bar=True)
```

Let's check how many values each document embedding contains:

```
# Check the dimensions of the resulting embeddings  
embeddings.shape
```

```
(44949, 384)
```

Each embedding has 384 values that together represent the semantic representation of the document. You can view these embeddings as the features that we want to cluster.

Reducing the Dimensionality of Embeddings

Before we cluster the embeddings, we will first need to take their high dimensionality into account. As the number of dimensions increases, there is an exponential growth in the number of possible values within each dimension. Finding all subspaces within each dimension becomes increasingly complex.

As a result, high-dimensional data can be troublesome for many clustering techniques as it gets more difficult to identify meaningful clusters. Instead, we can make use of dimensionality reduction. As illustrated in [Figure 5-4](#), this technique allows us to reduce the size of the dimensional space and represent the same data with fewer dimensions. Dimensionality reduction techniques aim to preserve the global structure of high-dimensional data by finding low-dimensional representations.

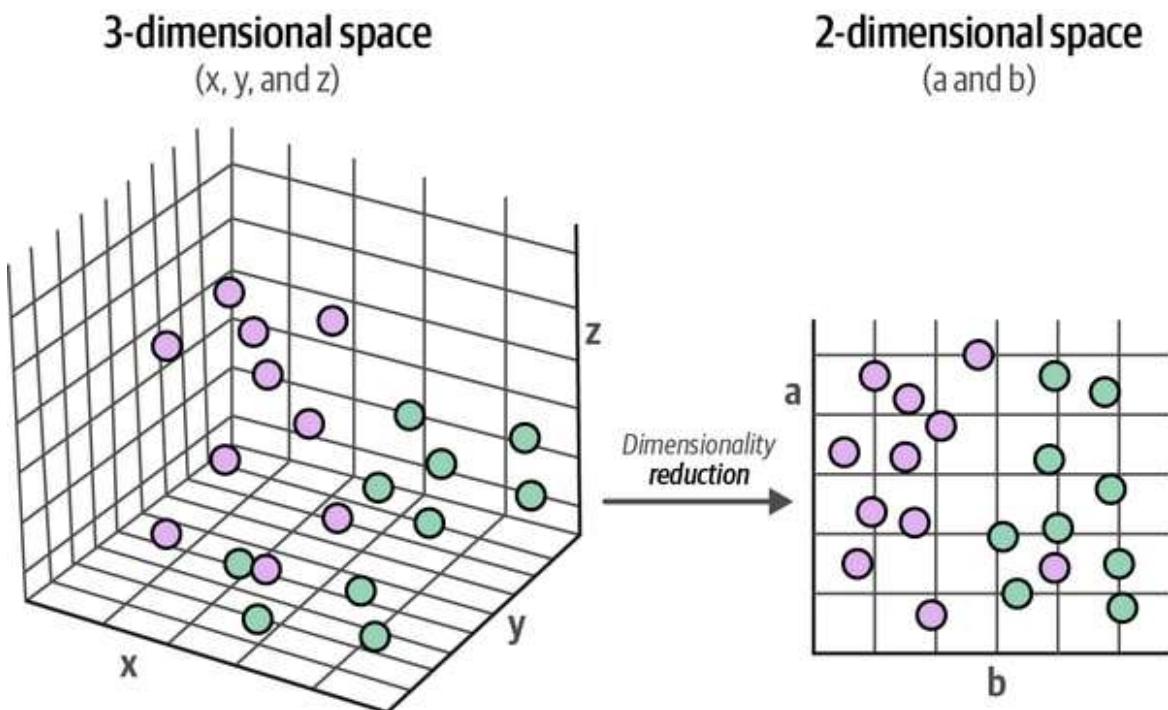


Figure 5-4. Dimensionality reduction allows data in high-dimensional space to be compressed to a lower-dimensional representation.

Note that this is a compression technique and that the underlying algorithm is not arbitrarily removing dimensions. To help the cluster model create meaningful clusters, the second step in our clustering pipeline is therefore dimensionality reduction, as shown in [Figure 5-5](#).

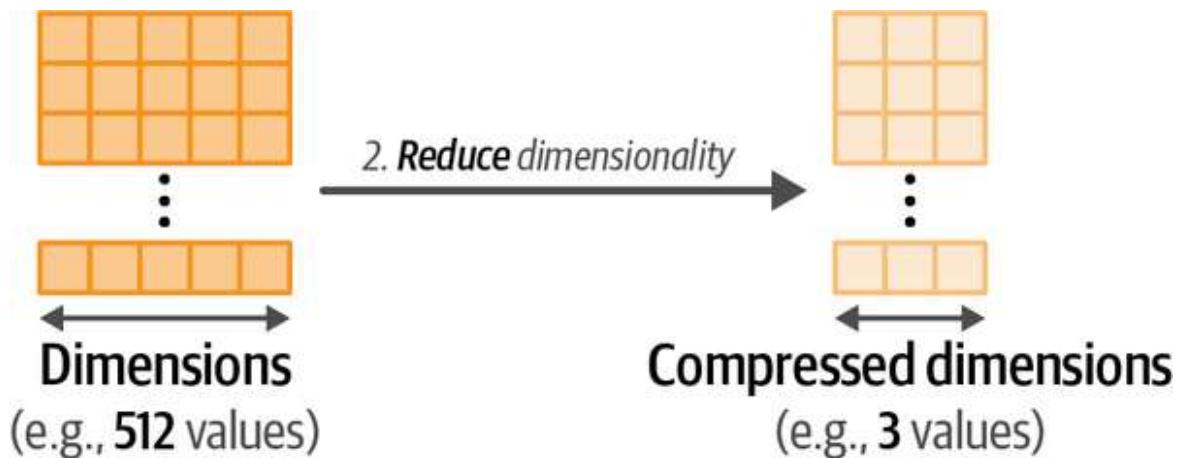


Figure 5-5. Step 2: The embeddings are reduced to a lower-dimensional space using dimensionality reduction.

Well-known methods for dimensionality reduction are Principal Component Analysis (PCA)¹ and Uniform Manifold Approximation and Projection (UMAP).² For this pipeline, we are going with UMAP as it tends to handle nonlinear relationships and structures a bit better than PCA.

NOTE

Dimensionality reduction techniques, however, are not flawless. They do not perfectly capture high-dimensional data in a lower-dimensional representation. Information will always be lost with this procedure. There is a balance between reducing dimensionality and keeping as much information as possible.

To perform dimensionality reduction, we need to instantiate our UMAP class and pass the generated embeddings to it:

```
from umap import UMAP

# We reduce the input embeddings from 384 dimensions to 5
# dimensions
umap_model = UMAP(
    n_components=5, min_dist=0.0, metric='cosine',
    random_state=42
)
reduced_embeddings = umap_model.fit_transform(embeddings)
```

We can use the `n_components` parameter to decide the shape of the lower-dimensional space, namely 5 dimensions. Generally, values between 5 and 10 work well to capture high-dimensional global structures.

The `min_dist` parameter is the minimum distance between embedded points. We are setting this to 0 as that generally results in tighter clusters. We set `metric` to '`'cosine'`' as Euclidean-based methods have issues dealing with high-dimensional data.

Note that setting a `random_state` in UMAP will make the results reproducible across sessions but will disable parallelism and therefore slow

down training.

Cluster the Reduced Embeddings

The third step is to cluster the reduced embeddings, as illustrated in [Figure 5-6](#).

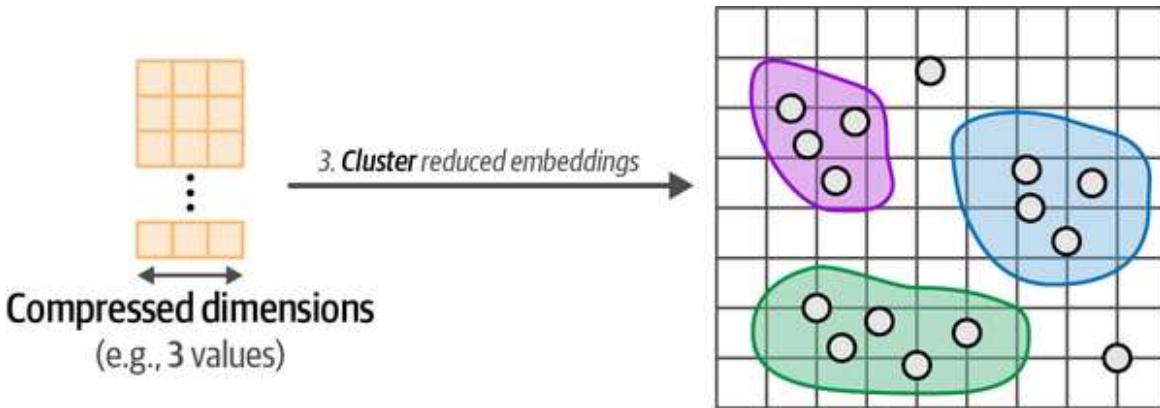


Figure 5-6. Step 3: We cluster the documents using the embeddings with reduced dimensionality.

Although a common choice is a centroid-based algorithm like k-means, which requires a set of clusters to be generated, we do not know the number of clusters beforehand. Instead, a density-based algorithm freely calculates the number of clusters and does not force all data points to be part of a cluster, as illustrated in [Figure 5-7](#).

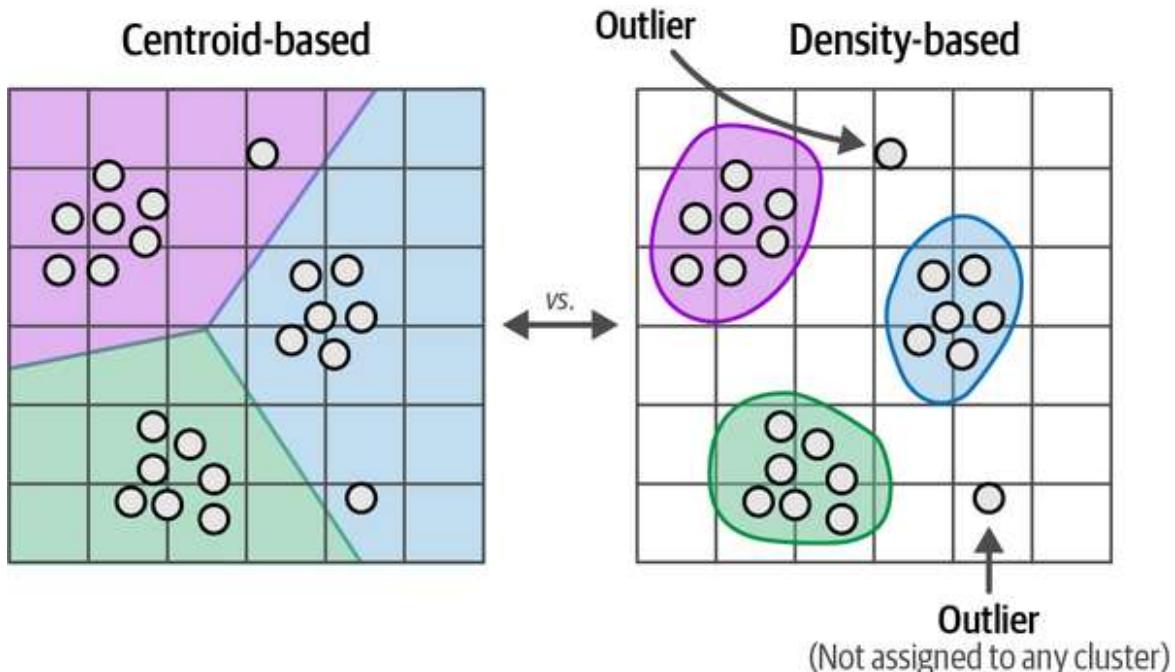


Figure 5-7. The clustering algorithm not only impacts how clusters are generated but also how they are viewed.

A common density-based model is Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN).³ HDBSCAN is a hierarchical variation of a clustering algorithm called DBSCAN that allows for dense (micro)-clusters to be found without having to explicitly specify the number of clusters.⁴ As a density-based method, HDBSCAN can also detect *outliers* in the data, which are data points that do not belong to any cluster. These outliers will not be assigned or forced to belong to any cluster. In other words, they are ignored. Since ArXiv articles might contain some niche papers, using a model that detects outliers could be helpful.

As with the previous packages, using HDBSCAN is straightforward. We only need to instantiate the model and pass our reduced embeddings to it:

```
from hdbscan import HDBSCAN

# We fit the model and extract the clusters
hdbscan_model = HDBSCAN(
    min_cluster_size=50, metric="euclidean",
    cluster_selection_method="eom"
).fit(reduced_embeddings)
```

```
clusters = hdbscan_model.labels_

# How many clusters did we generate?
len(set(clusters))
```

```
156
```

With HDBSCAN, we generated 156 clusters in our dataset. To create more clusters, we will need to reduce the value of `min_cluster_size` as it represents the minimum size that a cluster can take.

Inspecting the Clusters

Now that we have generated our clusters, we can inspect each cluster manually and explore the assigned documents to get an understanding of its content. For example, let us take a few random documents from cluster 0:

```
import numpy as np

# Print first three documents in cluster 0
cluster = 0
for index in np.where(clusters==cluster)[0][:3]:
    print(abstracts[index][:300] + "... \n")
```

This works aims to design a statistical machine translation from English text to American Sign Language (ASL). The system is based on Moses tool with some modifications and the results are synthesized through a 3D avatar for interpretation. First, we translate the input text to gloss, a written fo...

Researches on signed languages still strongly dissociate linguistic issues related on phonological and phonetic aspects, and gesture studies for recognition and synthesis purposes. This paper focuses on the imbrication of motion and meaning for the analysis, synthesis and evaluation of sign lang...

```
Modern computational linguistic software cannot produce
important aspects of
sign language translation. Using some researches we deduce that
the majority of
automatic sign language translation systems ignore many aspects
when they
generate animation; therefore the interpretation lost the truth
inf...
```

From these documents, it seems that this cluster contains documents mostly about translation from and to sign language, interesting!

We can take this one step further and attempt to visualize our results instead of going through all documents manually. To do so, we will need to reduce our document embeddings to two dimensions, as that allows us to plot the documents on an x/y plane:

```
import pandas as pd

# Reduce 384-dimensional embeddings to two dimensions for easier
# visualization
reduced_embeddings = UMAP(
    n_components=2, min_dist=0.0, metric="cosine",
    random_state=42
).fit_transform(embeddings)

# Create dataframe
df = pd.DataFrame(reduced_embeddings, columns=["x", "y"])
df["title"] = titles
df["cluster"] = [str(c) for c in clusters]

# Select outliers and non-outliers (clusters)
to_plot = df.loc[df.cluster != "-1", :]
outliers = df.loc[df.cluster == "-1", :]
```

We also created a dataframe for our clusters (`clusters_df`) and for the outliers (`outliers_df`) separately since we generally want to focus on the clusters and highlight those.

NOTE

Using any dimensionality reduction technique for visualization purposes creates information loss. It is merely an approximation of what our original embeddings look like. Although it is informative, it might push clusters together and drive them further apart than they actually are. Human evaluation, inspecting the clusters ourselves, is therefore a key component of cluster analysis!

To generate a static plot, we will use the well-known plotting library, **matplotlib**:

```
import matplotlib.pyplot as plt

# Plot outliers and non-outliers separately
plt.scatter(outliers_df.x, outliers_df.y, alpha=0.05, s=2,
c="grey")
plt.scatter(
    clusters_df.x, clusters_df.y,
c=clusters_df.cluster.astype(int),
    alpha=0.6, s=2, cmap="tab20b"
)
plt.axis("off")
```

As we can see in [Figure 5-8](#), it tends to capture major clusters quite well. Note how clusters of points are colored in the same color, indicating that HDBSCAN put them in a group together. Since we have a large number of clusters, the plotting library cycles the colors between clusters, so don't think that all green points are one cluster, for example.



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

This is visually appealing but does not yet allow us to see what is happening inside the clusters. Instead, we can extend this visualization by going from text clustering to topic modeling.

From Text Clustering to Topic Modeling

Text clustering is a powerful tool for finding structure among large collections of documents. In our previous example, we could manually inspect each cluster and identify them based on their collection of documents. For instance, we explored a cluster that contained documents about sign language. We could say that the *topic* of that cluster is “sign language.”

This idea of finding themes or latent topics in a collection of textual data is often referred to as *topic modeling*. Traditionally, it involves finding a set of

keywords or phrases that best represent and capture the meaning of the topic, as we illustrate in [Figure 5-9](#).

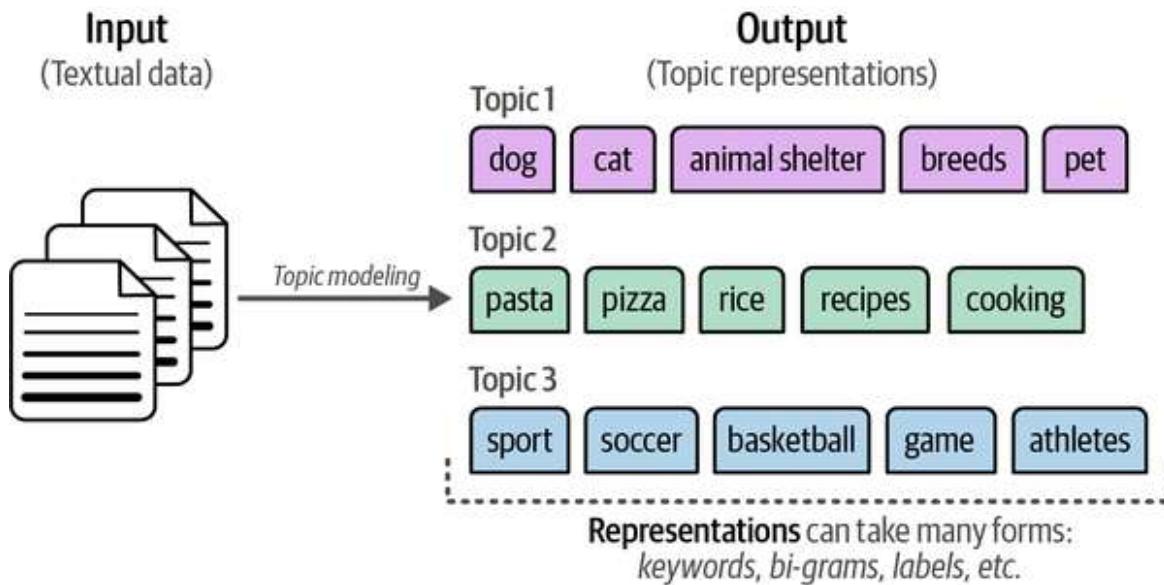


Figure 5-9. Traditionally, topics are represented by a number of keywords but can take other forms.

Instead of labeling a topic as “sign language,” these techniques use keywords such as “sign,” “language,” and “translation” to describe the topic. As such, this does not give a single label to a topic and instead requires the user to understand the meaning of the topic through those keywords.

Classic approaches, like latent Dirichlet allocation, assume that each topic is characterized by a probability distribution of words in a corpus’s vocabulary.⁵ [Figure 5-10](#) demonstrates how each word in a vocabulary is scored against its relevance to each topic.

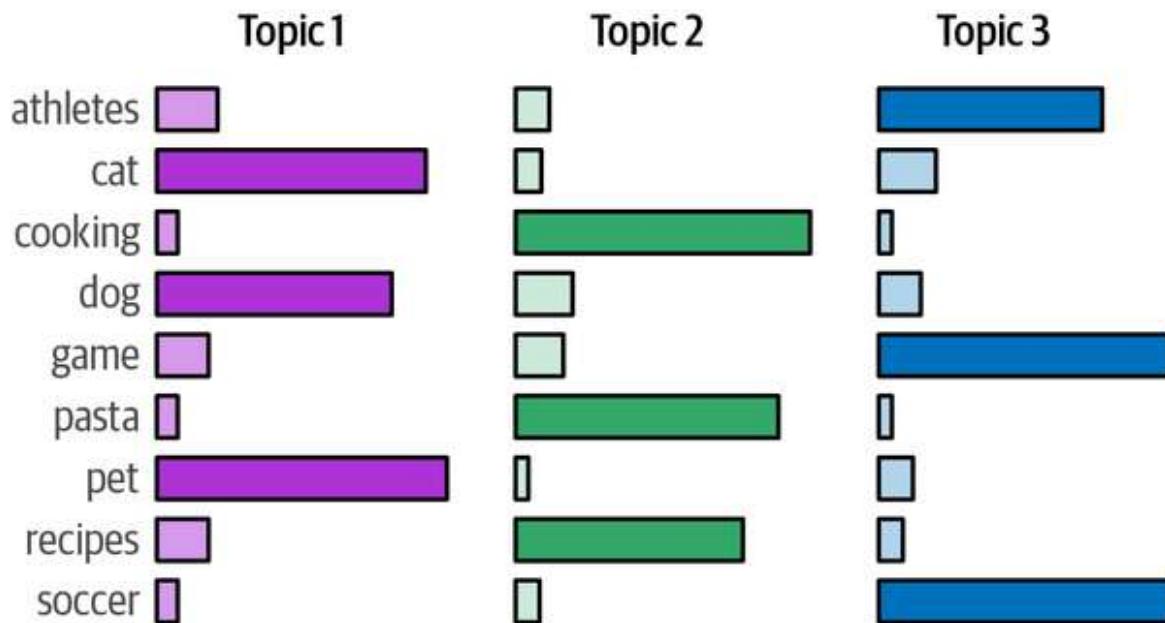


Figure 5-10. Keywords are extracted based on their distribution over a single topic.

These approaches generally use a bag-of-words technique for the main features of the textual data, which does not take the context nor the meaning of words and phrases into account. In contrast, our text clustering example does take both into account as it relies on Transformer-based embeddings that are optimized for semantic similarity and contextual meaning through attention.

In this section, we will extend text clustering into the realm of topic modeling through a highly modular text clustering and topic modeling framework, namely BERTopic.

BERTopic: A Modular Topic Modeling Framework

BERTopic is a topic modeling technique that leverages clusters of semantically similar texts to extract various types of topic representations.⁶ The underlying algorithm can be thought of in two steps.

First, as illustrated in [Figure 5-11](#), we follow the same procedure as we did before in our text clustering example. We embed documents, reduce their dimensionality, and finally cluster the reduced embedding to create groups of semantically similar documents.

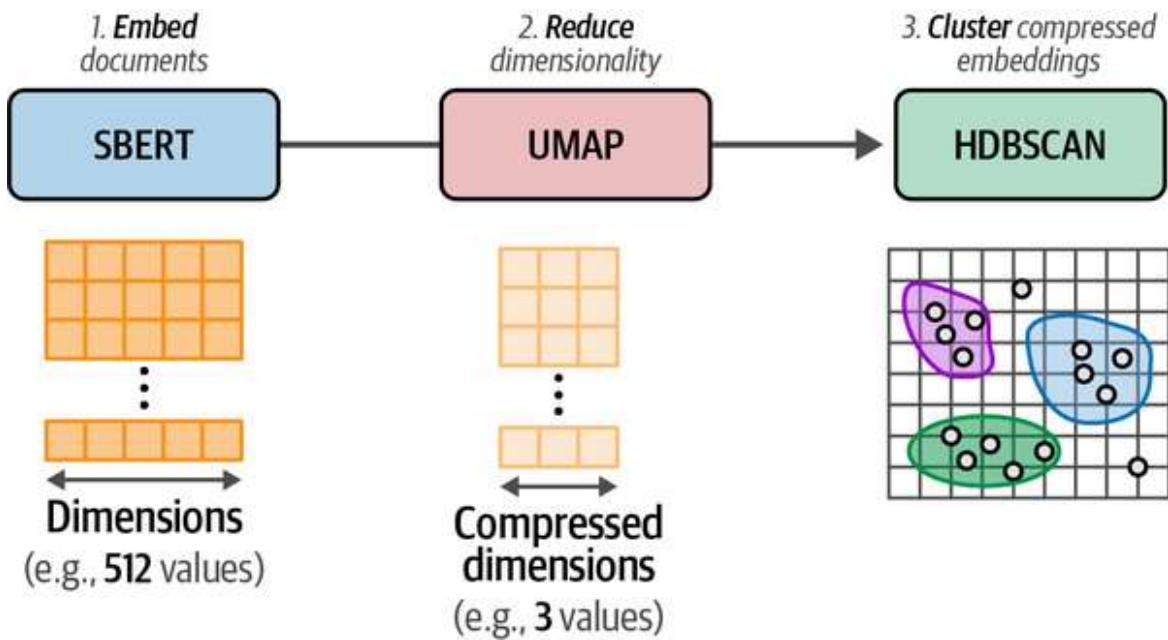


Figure 5-11. The first part of BERTopic’s pipeline is to create clusters of semantically similar documents.

Second, it models a distribution over words in the corpus’s vocabulary by leveraging a classic method, namely bag-of-words. The bag-of-words, as we discussed briefly in [Chapter 1](#) and illustrate in [Figure 5-12](#), does exactly what its name implies, counting the number of times each word appears in a document. The resulting representation could be used to extract the most frequent words inside a document.

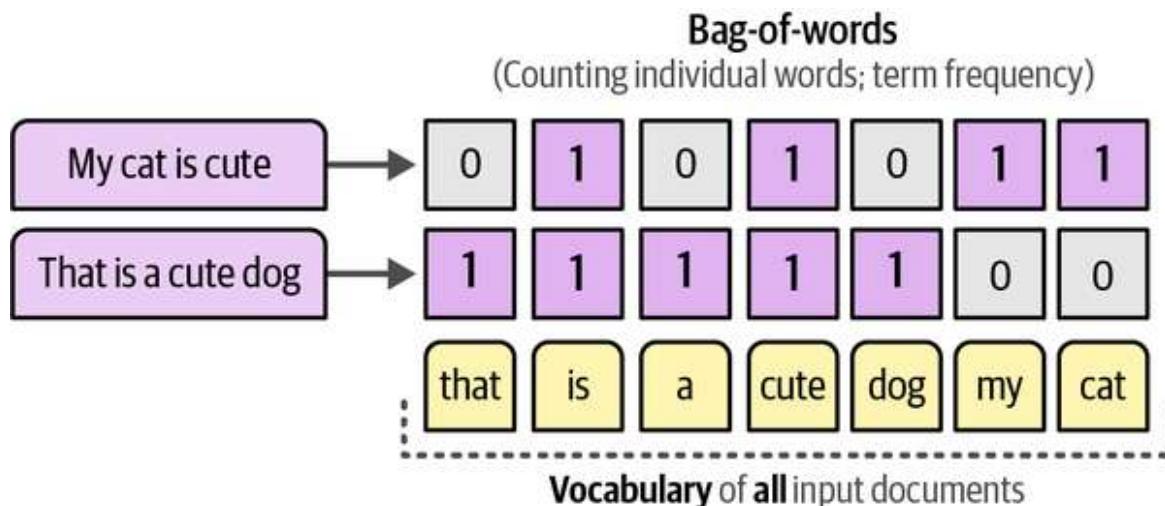


Figure 5-12. A bag-of-words counts the number of times each word appears inside a document.

There are two caveats, however. First, this is a representation on a document level and we are interested in a cluster-level perspective. To address this, the frequency of words is calculated within the entire cluster instead of only the document, as illustrated in [Figure 5-13](#).

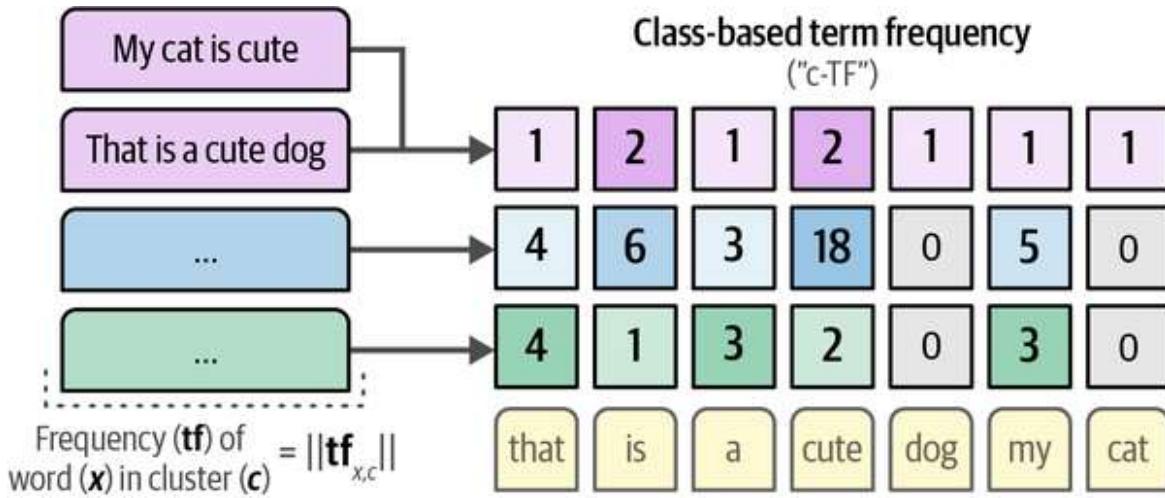


Figure 5-13. Generating c-TF by counting the frequency of words per cluster instead of per document.

Second, stop words like “the” and “I” tend to appear often in documents and provide little meaning to the actual documents. BERTopic uses a class-based variant of term frequency-inverse document frequency (c-TF-IDF) to put more weight on words that are more meaningful to a cluster and put less weight on words that are used across all clusters.

Each word in the bag-of-words, the c-TF in c-TF-IDF, is multiplied by the IDF value of each word. As shown in [Figure 5-14](#), the IDF value is calculated by taking the logarithm of the average frequency of all words across all clusters divided by the total frequency of each word.

Inverse document frequency ("IDF")

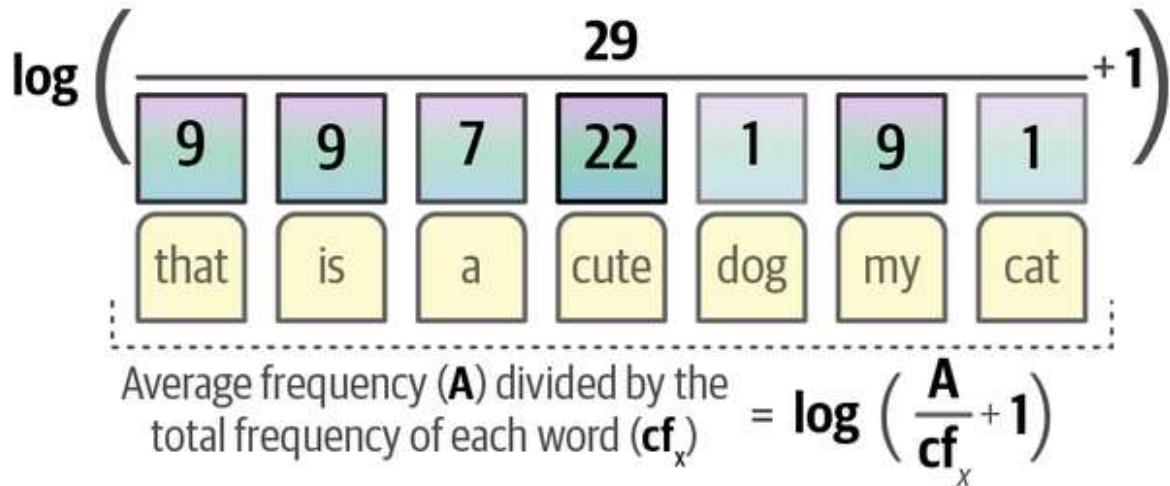


Figure 5-14. Creating a weighting scheme.

The result is a weight ("IDF") for each word that we can multiply with their frequency ("c-TF") to get the weighted values ("c-TF-IDF").

This second part of the procedure, as shown in [Figure 5-15](#), allows for generating a distribution over words as we have seen before. We can use scikit-learn's `CountVectorizer` to generate the bag-of-words (or term frequency) representation. Here, each cluster is considered a topic that has a specific ranking of the corpus's vocabulary.

4. **Create** a class-based bag-of-words 5. **Weigh** terms

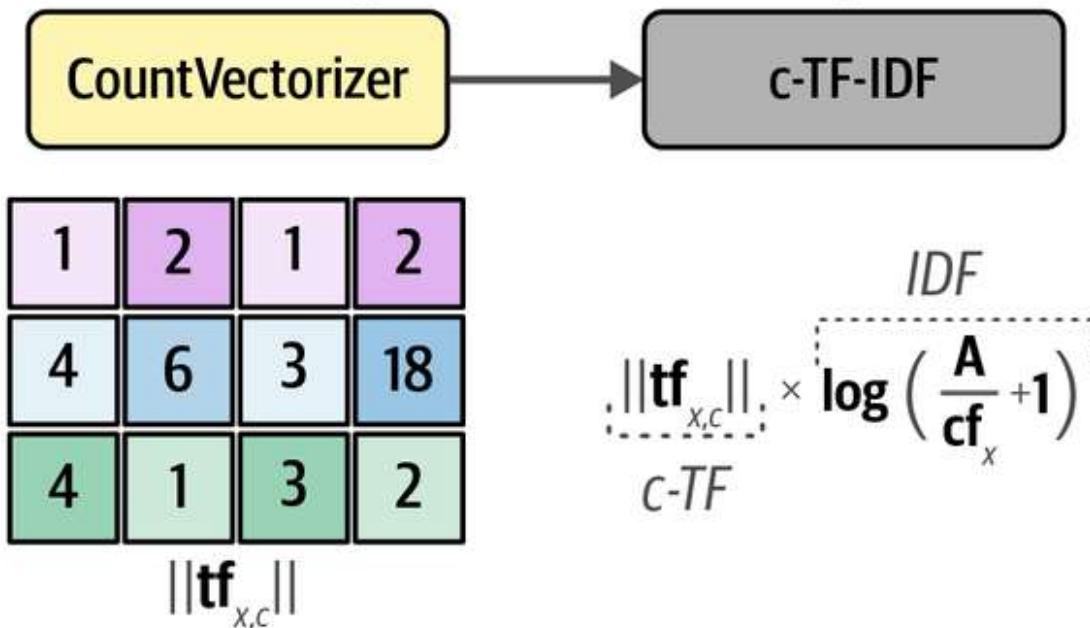


Figure 5-15. The second part of BERTopic's pipeline is representing the topics: the calculation of the weight of term x in a class c .

Putting the two steps together, clustering and representing topics, results in the full pipeline of BERTopic, as illustrated in Figure 5-16. With this pipeline, we can cluster semantically similar documents and from the clusters generate topics represented by several keywords. The higher a word's weight in a topic, the more representative it is of that topic.



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

A major advantage of this pipeline is that the two steps, clustering and topic representation, are largely independent of one another. For instance, with c-TF-IDF, we are not dependent on the models used in clustering the documents. This allows for significant modularity throughout every component of the pipeline. And as we will explore later in this chapter, it is a great starting point to fine-tune the topic representations.

As illustrated in [Figure 5-17](#), although sentence-transformers is used as the default embedding model, we can swap it with any other embedding technique. The same applies to all other steps. If you do not want outliers generated with HDBSCAN, you can use k-means instead.

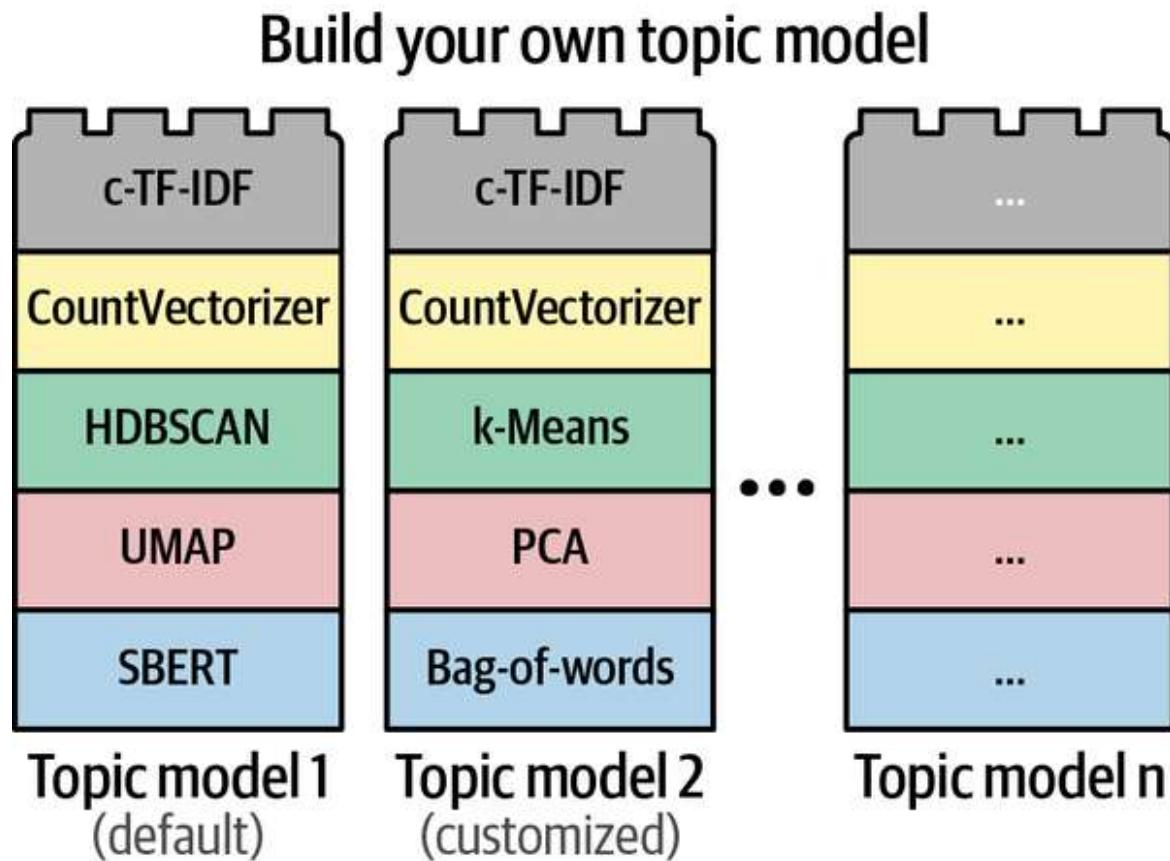


Figure 5-17. The modularity of BERTopic is a key component and allows you to build your own topic model however you want.

You can think of this modularity as building with Lego blocks; each part of the pipeline is completely replaceable with another, similar algorithm. Through this modularity, newly released models can be integrated within its architecture. As the field of Language AI grows, so does BERTopic!

THE MODULARITY OF BERTOPIC

The modularity of BERTopic has another advantage: it allows it to be used and adapted to different use cases using the same base model. For instance, BERTopic supports a wide variety of algorithmic variants:

- Guided topic modeling
- (Semi-)supervised topic modeling
- Hierarchical topic modeling
- Dynamic topic modeling
- Multimodal topic modeling
- Multi-aspect topic modeling
- Online and incremental topic modeling
- Zero-shot topic modeling
- Etc.

The modularity and algorithmic flexibility are the foundation of the author's aim to make BERTopic the one-stop-shop for topic modeling. You can find a full overview of its capabilities in [the documentation](#) or the [repository](#).

To run BERTopic with our ArXiv dataset, we can use our previously defined models and embeddings (although it is not mandatory):

```
from bertopic import BERTopic

# Train our model with our previously defined models
topic_model = BERTopic(
    embedding_model=embedding_model,
    umap_model=umap_model,
    hdbscan_model=hdbscan_model,
```

```
    verbose=True  
).fit(abstracts, embeddings)
```

Let us start by exploring the topics that were created. The `get_topic_info()` method is useful to get a quick description of the topics that we found:


```
topic_model.get_topic_info()
```

Topic	Count	Name
-1	14520	-1_the_of_and_to
0	2290	0_speech_asr_recognition_end
1	1403	1_medical_clinical_biomedical_pa
2	1156	2_sentiment_aspect_analysis_revie
3	986	3_translation_nmt_machine_neural
...
150	54	150_coherence_discourse_paragraph
151	54	151_prompt_prompts_optimization

Topic	Count	Name
152	53	152_sentence_sts_embeddings_sim
153	53	153_counseling_mental_health_the
154	50	154_backdoor_attacks_attack_trigg

Each of these topics is represented by several keywords, which are concatenated with a “_” in the **Name** column. This **Name** column allows us to quickly get a feeling of what the topic is about as it shows the four keywords that best represent it.

NOTE

You might also have noticed that the very first topic is labeled -1. That topic contains all documents that could not be fitted within a topic and are considered outliers. This is a result of the clustering algorithm, HDBSCAN, which does not force all points to be clustered. To remove outliers, we could either use a non-outlier algorithm like k-means or use BERTopic’s `reduce_outliers()` function to reassign the outliers to topics.

We can inspect individual topics and explore which keywords best represent them with the `get_topic` function. For example, topic 0 contains the

following keywords:

```
topic_model.get_topic(0)
[('speech', 0.028177697715245358),
 ('asr', 0.018971184497453525),
 ('recognition', 0.013457745472471012),
 ('end', 0.00980445092749381),
 ('acoustic', 0.009452082794507863),
 ('speaker', 0.0068822647060204885),
 ('audio', 0.006807649923681604),
 ('the', 0.0063343444687017645),
 ('error', 0.006320144717019838),
 ('automatic', 0.006290216996043161)]
```

For example, topic 0 contains the keywords “speech,” “asr,” and “recognition.” Based on these keywords, it seems that the topic is about automatic speech recognition (ASR).

We can use the `find_topics()` function to search for specific topics based on a search term. Let’s search for a topic about topic modeling:

```
topic_model.find_topics("topic modeling")
([22, -1, 1, 47, 32],
 [0.95456535, 0.91173744, 0.9074769, 0.9067007, 0.90510106])
```

This returns that topic 22 has a relatively high similarity (0.95) with our search term. If we then inspect the topic, we can see that it is indeed a topic about topic modeling:

```
topic_model.get_topic(22)
[('topic', 0.06634619076655907),
 ('topics', 0.035308535091932707),
 ('lda', 0.016386314730705634),
 ('latent', 0.013372311924864435),
 ('document', 0.012973600191120576),
```

```
('documents', 0.012383715497143821),  
('modeling', 0.011978375291037142),  
('dirichlet', 0.010078277589545706),  
('word', 0.008505619415413312),  
('allocation', 0.007930890698168108)]
```

Although we know that this topic is about topic modeling, let's see if the BERTopic abstract is also assigned to this topic:

```
topic_model.topics_[titles.index("BERTopic: Neural topic modeling  
with a class-based TF-IDF procedure")]
```

22

It is! These functionalities allow us to quickly find the topics that we are interested in.

TIP

The modularity of BERTopic gives you a lot of choices, which can be overwhelming. For that purpose, the author created a [best practices guide](#) that goes through common practices to speed up training, improve representations, and more.

To make exploration of the topics a bit easier, we can look back at our text clustering example. There, we created a static visualization to see the general structure of the created topic. With BERTopic, we can create an interactive variant that allows us to quickly explore which topics exist and which documents they contain.

Doing so requires us to use the two-dimensional embeddings, `reduced_embeddings`, that we created with UMAP. Moreover, when we hover over documents, we will show the title instead of the abstract to quickly get an understanding of the documents in a topic:

```
# Visualize topics and documents
```

```

fig = topic_model.visualize_documents(
    titles,
    reduced_embeddings=reduced_embeddings,
    width=1200,
    hide_annotations=True
)

# Update fonts of legend for easier visualization
fig.update_layout(font=dict(size=16))

```

As we can see in [Figure 5-18](#), this interactive plot quickly gives us a sense of the created topics. You can zoom in to view individual documents or double-click a topic on the righthand side to only view it.

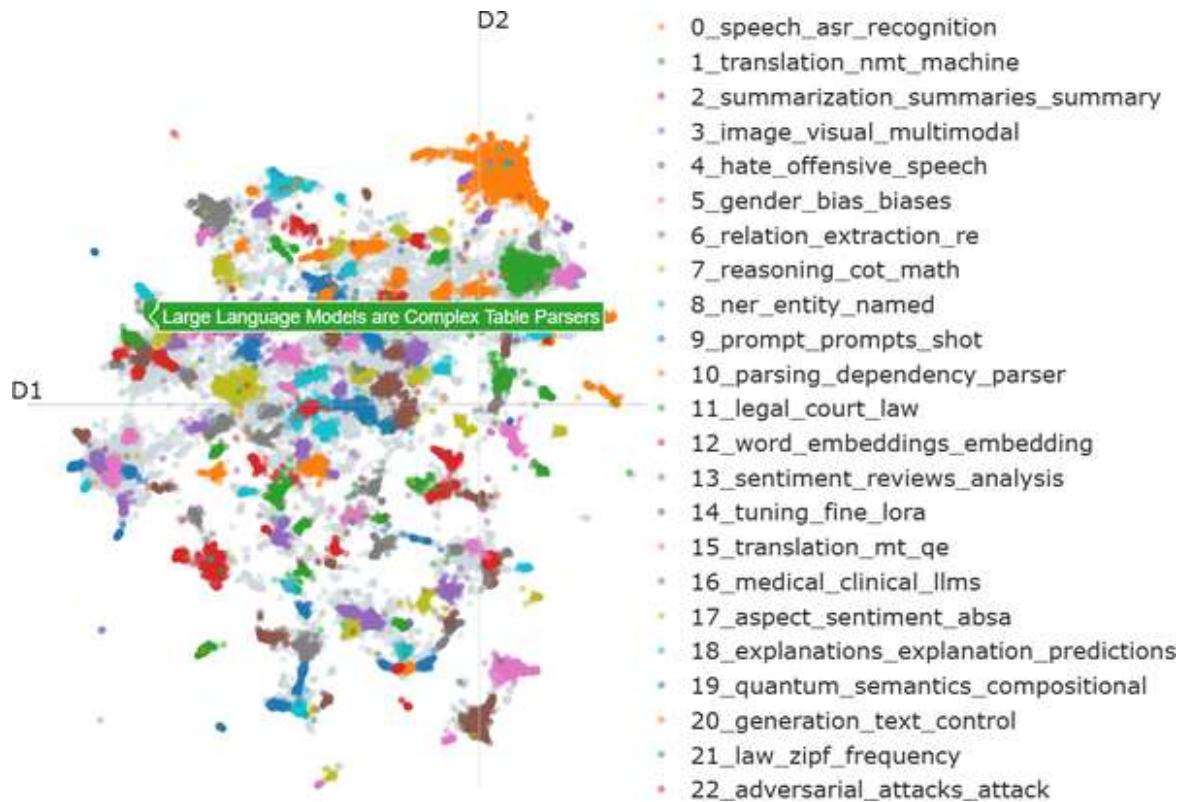


Figure 5-18. The output when we visualize documents and topics.

There is a wide range of visualization options in BERTopic. There are three that are worthwhile to explore to get an idea of the relationships between topics:

```
# Visualize barchart with ranked keywords
```

```

topic_model.visualize_barchart()

# Visualize relationships between topics
topic_model.visualize_heatmap(n_clusters=30)

# Visualize the potential hierarchical structure of topics
topic_model.visualize_hierarchy()

```

Adding a Special Lego Block

The pipeline in BERTopic that we have explored thus far, albeit fast and modular, has a disadvantage: it still represents a topic through a bag-of-words without taking into account semantic structures.

The solution is to leverage the strength of the bag-of-words representation, which is its speed to generate a meaningful representation. We can use this first meaningful representation and tweak it using more powerful but slower techniques, like embedding models. As shown in [Figure 5-19](#), we can rerank the initial distribution of words to improve the resulting representation. Note that this idea of reranking an initial set of results is a main staple in neural search, a subject that we cover in [Chapter 8](#).

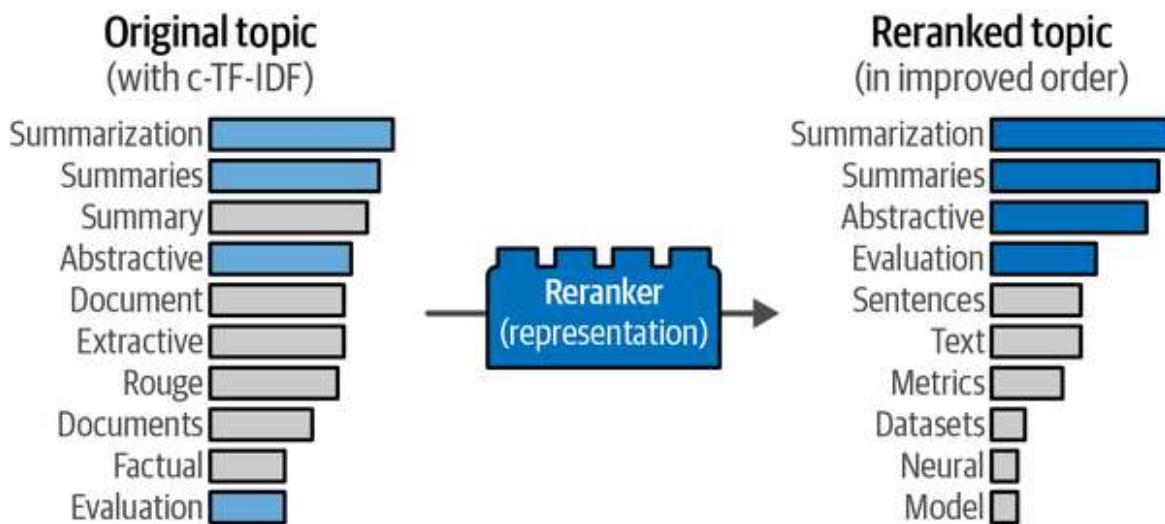


Figure 5-19. Fine-tune the topic representations by reranking the original c-TF-IDF distributions.

As a result, we can design a new Lego block, as shown in [Figure 5-20](#), that takes in this first topic representation and spits out an improved representation.

In BERTopic, such reranker models are referred to as *representation models*. A major benefit of this approach is that the optimization of topic representations only needs to be done as many times as we have topics. For instance, if we have millions of documents and a hundred topics, the representation block only needs to be applied once for every topic instead of for every document.

As shown in [Figure 5-21](#), a wide variety of representation blocks have been designed for BERTopic that allows you to fine-tune the representations. The representation block can even be stacked multiple times to fine-tune representations using different methodologies.

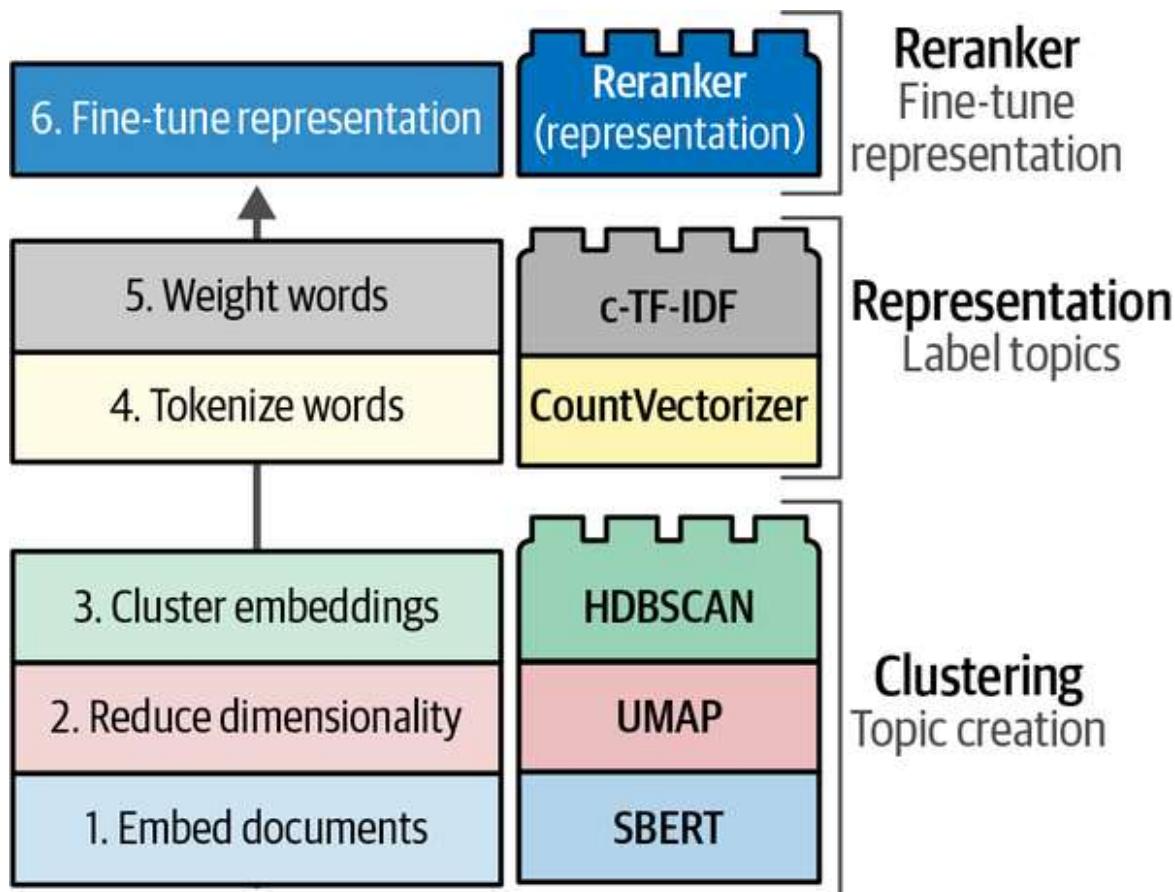


Figure 5-20. The reranker (representation) block sits on top of the c-TF-IDF representation.

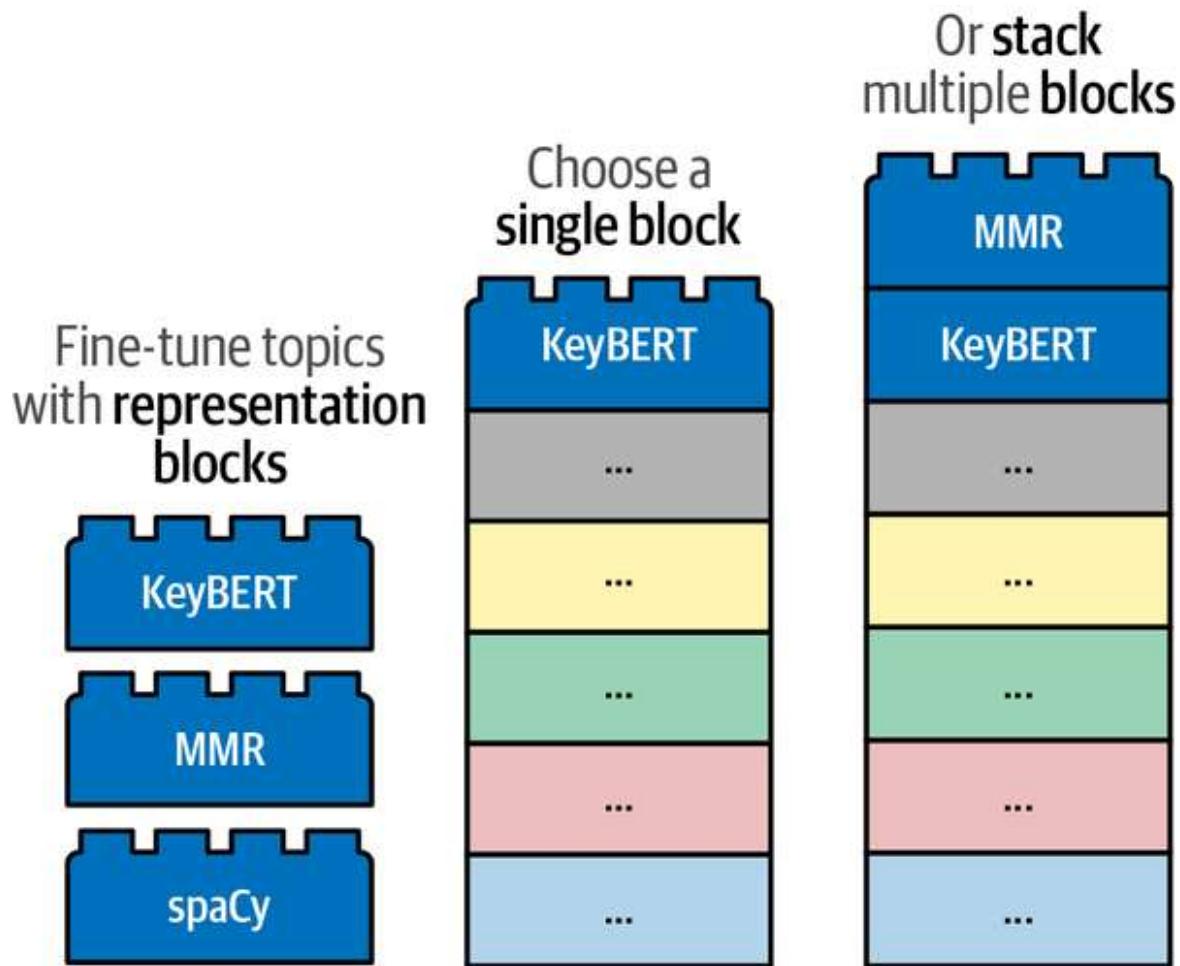


Figure 5-21. After applying the c-TF-IDF weighting, topics can be fine-tuned with a wide variety of representation models, many of which are large language models.

Before we explore how we can use these representation blocks, we first need to do two things. First, we are going to save our original topic representations so that it will be much easier to compare with and without representation models:

```
# Save original representations
from copy import deepcopy
original_topics = deepcopy(topic_model.topic_representations_)
```

Second, let's create a short wrapper that we can use to quickly visualize the differences in topic words to compare with and without representation models:

```

def topic_differences(model, original_topics, nr_topics=5):
    """Show the differences in topic representations between two
models """
    df = pd.DataFrame(columns=["Topic", "Original", "Updated"])
    for topic in range(nr_topics):

        # Extract top 5 words per topic per model
        og_words = " | ".join(list(zip(*original_topics[topic])))
[0][:5])
        new_words = " | ".join(list(zip(*model.get_topic(topic))))
[0][:5])
        df.loc[len(df)] = [topic, og_words, new_words]

    return df

```

KeyBERTInspired

The first representation block that we are going to explore is KeyBERTInspired. KeyBERTInspired is, as you might have guessed, a method inspired by the keyword extraction package, [KeyBERT](#).⁷ KeyBERT extracts keywords from texts by comparing word and document embeddings through cosine similarity.

BERTopic uses a similar approach. KeyBERTInspired uses c-TF-IDF to extract the most representative documents per topic by calculating the similarity between a document's c-TF-IDF values and those of the topic they correspond to. As shown in [Figure 5-22](#), the average document embedding per topic is calculated and compared to the embeddings of candidate keywords to rerank the keywords.

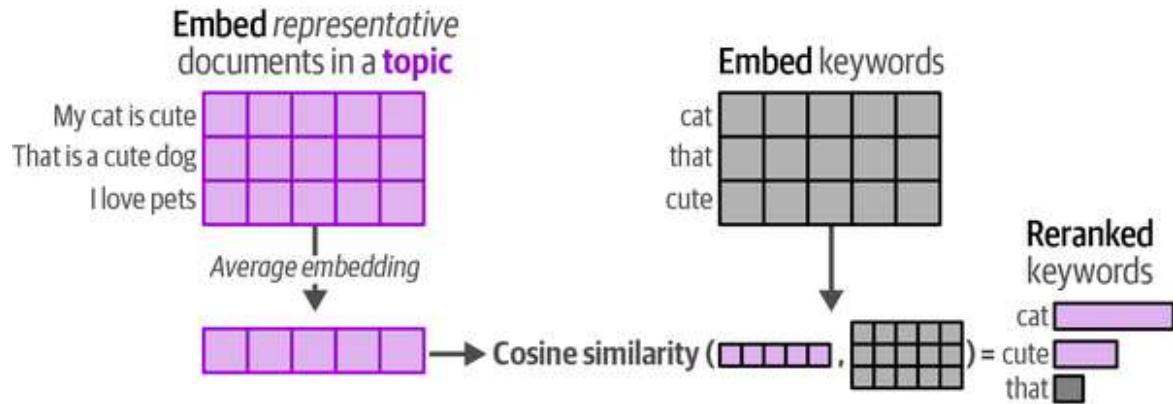


Figure 5-22. KeyBERTInspired representation model procedure.

Due to the modular nature of BERTopic, we can update our initial topic representations with KeyBERTInspired without needing to perform the dimensionality reduction and clustering steps:

```
from bertopic.representation import KeyBERTInspired

# Update our topic representations using KeyBERTInspired
representation_model = KeyBERTInspired()
topic_model.update_topics(abstracts,
representation_model=representation_model)
```

```
# Show topic differences
topic_differences(topic_model, original_topics)
```

Topic	Original	Updated
0	speech asr recognition end acoustic	speech encoder phonetic language trans...
1	medical clinical biomedical patient he...	nlp ehr clinical biomedical language
2	sentiment aspect analysis reviews opinion	aspect sentiment aspects sentiments cl...
3	translation nmt machine neural bleu	translation translating translate transl...
4	summarization summaries summary abstract...	summarization summarizers summaries summ...

The updated model shows that the topics are easier to read compared to the original model. It also demonstrates the downside of using embedding-based techniques. Words in the original model, like *nmt* (topic 3), which stands for neural machine translation, are removed as the model could not properly represent the entity. For domain experts, these abbreviations are highly informative.

Maximal marginal relevance

With c-TF-IDF and the previously shown KeyBERTInspired techniques, we still have significant redundancy in the resulting topic representations. For instance, having both the words “summaries” and “summary” in a topic representation introduces redundancy as they are quite similar.

We can use maximal marginal relevance (MMR) to diversify our topic representations. The algorithm attempts to find a set of keywords that are diverse from one another but still relate to the documents they are compared to. It does so by embedding a set of candidate keywords and iteratively calculating the next best keyword to add. Doing so requires setting a diversity parameter, which indicates how diverse keywords need to be.

In BERTopic, we use MMR to go from a set of initial keywords, let's say 30, to a smaller but more diverse set of keywords, let's say 10. It filters out redundant words and only keeps words that contribute something new to the topic representation.

Doing so is rather straightforward:

```
from bertopic.representation import MaximalMarginalRelevance

# Update our topic representations to MaximalMarginalRelevance
representation_model = MaximalMarginalRelevance(diversity=0.2)
topic_model.update_topics(abstracts,
representation_model=representation_model)
```

```
# Show topic differences
topic_differences(topic_model, original_topics)
```

Topic	Original	Updated
0	speech asr recognition end acoustic	speech asr error model training
1	medical clinical biomedical patient he...	clinical biomedical patient healthcare ...
2	sentiment aspect analysis reviews opinion	sentiment analysis reviews absa polarity
3	translation nmt machine neural bleu	translation nmt bleu parallel multilin...
4	summarization summaries summary abstract...	summarization document extractive rouge ...

The resulting topics demonstrate more diversity in their representations. For instance, topic 4 only shows one “summary”-like word and instead adds other words that might contribute more to the overall representation.

TIP

Both KeyBERTInspired and MMR are amazing techniques for improving the first set of topic representations. KeyBERTInspired especially tends to remove nearly all stop words since it focuses on the semantic relationships between words and documents.

The Text Generation Lego Block

The representation block in BERTopic has been acting as a reranking block in our previous examples. However, as we already explored in the previous chapter, generative models have great potential for a wide variety of tasks.

We can use generative models in BERTopic quite efficiently by following a part of the reranking procedure. Instead of using a generative model to identify the topic of all documents, of which there can potentially be millions, we will use the model to generate a label for our topic. As illustrated in [Figure 5-23](#), instead of generating or reranking keywords, we ask the model to generate a short label based on keywords that were previously generated and a small set of representative documents.

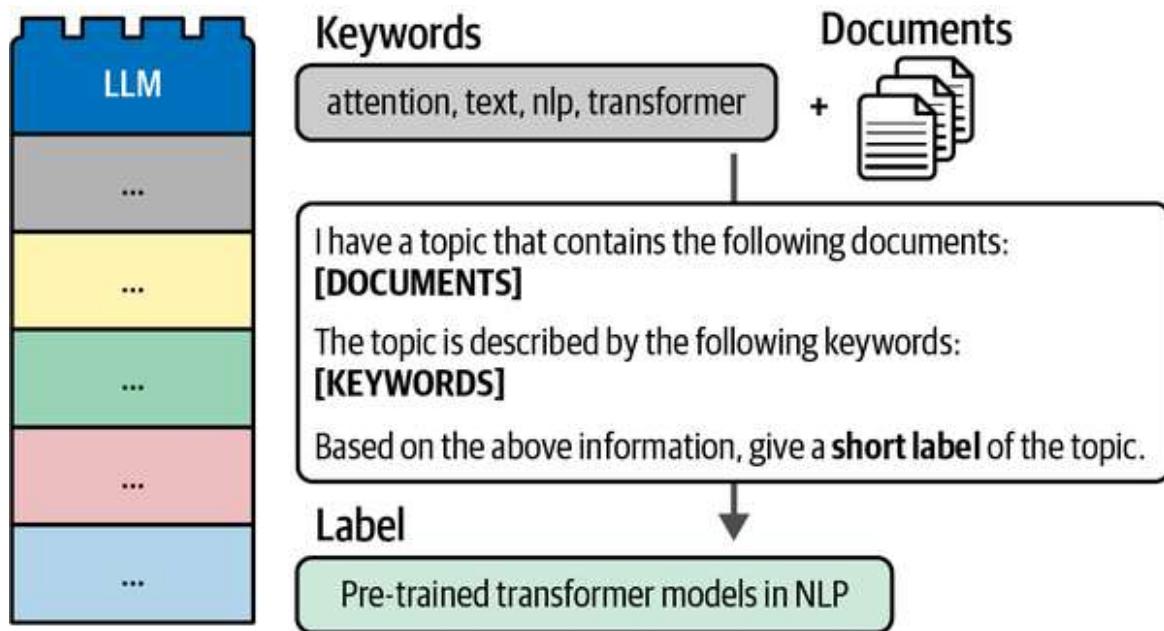


Figure 5-23. Use text generative LLMs and prompt engineering to create labels for topics from keywords and documents related to each topic.

There are two components to the illustrated prompt. First, the documents that are inserted using the [DOCUMENTS] tag are a small subset of documents, typically four, that best represent the topic. The documents with the highest cosine similarity of their c-TF-IDF values with those of the topic are selected. Second, the keywords that make up a topic are also passed to the prompt and referenced using the [KEYWORDS] tag. The

keywords could be generated by c-TF-IDF or any of the other representations we discussed thus far.

As a result, we only need to use the generative model once for every topic, of which there could be potentially hundreds, instead of once for each document, of which there could potentially be millions. There are many generative models that we can choose from, both open source and proprietary. Let's start with a model that we have explored in the previous chapter, the Flan-T5 model.

We create a prompt that works well with the model and use it in BERTopic through the `representation_model` parameter:

```
from transformers import pipeline
from bertopic.representation import TextGeneration

prompt = """I have a topic that contains the following documents:
[DOCUMENTS]

The topic is described by the following keywords: '[KEYWORDS]'.

Based on the documents and keywords, what is this topic about?"""

# Update our topic representations using Flan-T5
generator = pipeline("text2text-generation", model="google/flan-t5-small")
representation_model = TextGeneration(
    generator, prompt=prompt, doc_length=50,
    tokenizer="whitespace"
)
topic_model.update_topics(abstracts,
representation_model=representation_model)
```

```
# Show topic differences
topic_differences(topic_model, original_topics)
```

Topic	Original	Updated
0	speech asr recognition end acoustic	Speech-to-description
1	medical clinical biomedical patient he...	Science/Tech
2	sentiment aspect analysis reviews opinion	Review
3	translation nmt machine neural bleu	Attention-based neural machine translation
4	summarization summaries summary abstract...	Summarization

Some of these labels, like “Summarization” seem to be logical when comparing them to the original representations. Others, however, like “Science/Tech,” seem quite broad and do not do the original topic justice. Let’s explore instead how OpenAI’s GPT-3.5 would perform considering the model is not only larger but expected to have more linguistic capabilities:

```
import openai
from bertopic.representation import OpenAI
```

```

prompt = """
I have a topic that contains the following documents:
[DOCUMENTS]

The topic is described by the following keywords: [KEYWORDS]

Based on the information above, extract a short topic label in
the following format:
topic: <short topic label>
"""

# Update our topic representations using GPT-3.5
client = openai.OpenAI(api_key="YOUR_KEY_HERE")
representation_model = OpenAI(
    client, model="gpt-3.5-turbo", exponential_backoff=True,
    chat=True, prompt=prompt
)
topic_model.update_topics(abstracts,
representation_model=representation_model)

# Show topic differences
topic_differences(topic_model, original_topics)

```

Topic	Original	Updated
0	speech asr recognition end acoustic	Leveraging External Data for Improving Low-Res...
1	medical clinical biomedical patient he...	Improved Representation Learning for Biomedica...
2	sentiment aspect analysis reviews opinion	Advancements in Aspect-Based Sentiment Analys...
3	translation nmt machine neural bleu	Neural Machine Translation

Topic	Original	Updated
	Enhancements	
4	summarization summaries summary abstract...	Document Summarization Techniques

The resulting labels are quite impressive! We are not even using GPT-4 and the resulting labels seem to be more informative than our previous example. Note that BERTopic is not confined to only using OpenAI's offering but has local backends as well.

TIP

Although it seems like we do not need the keywords anymore, they are still representative of the input documents. No model is perfect and it is generally advised to generate multiple topic representations. **BERTopic allows for all topics to be represented by different representations.** You could, for example, use KeyBERTInspired, MMR, and GPT-3.5 side by side to get different perspectives on the same topic.

With these GPT-3.5 generated labels, we can create beautiful illustrations using the **datamapplot** package (Figure 5-24):

```
# Visualize topics and documents
fig = topic_model.visualize_document_datamap(
    titles,
    topics=list(range(20)),
    reduced_embeddings=reduced_embeddings,
    width=1200,
    label_font_size=11,
    label_wrap_width=20,
    use_medoids=True,
)
```

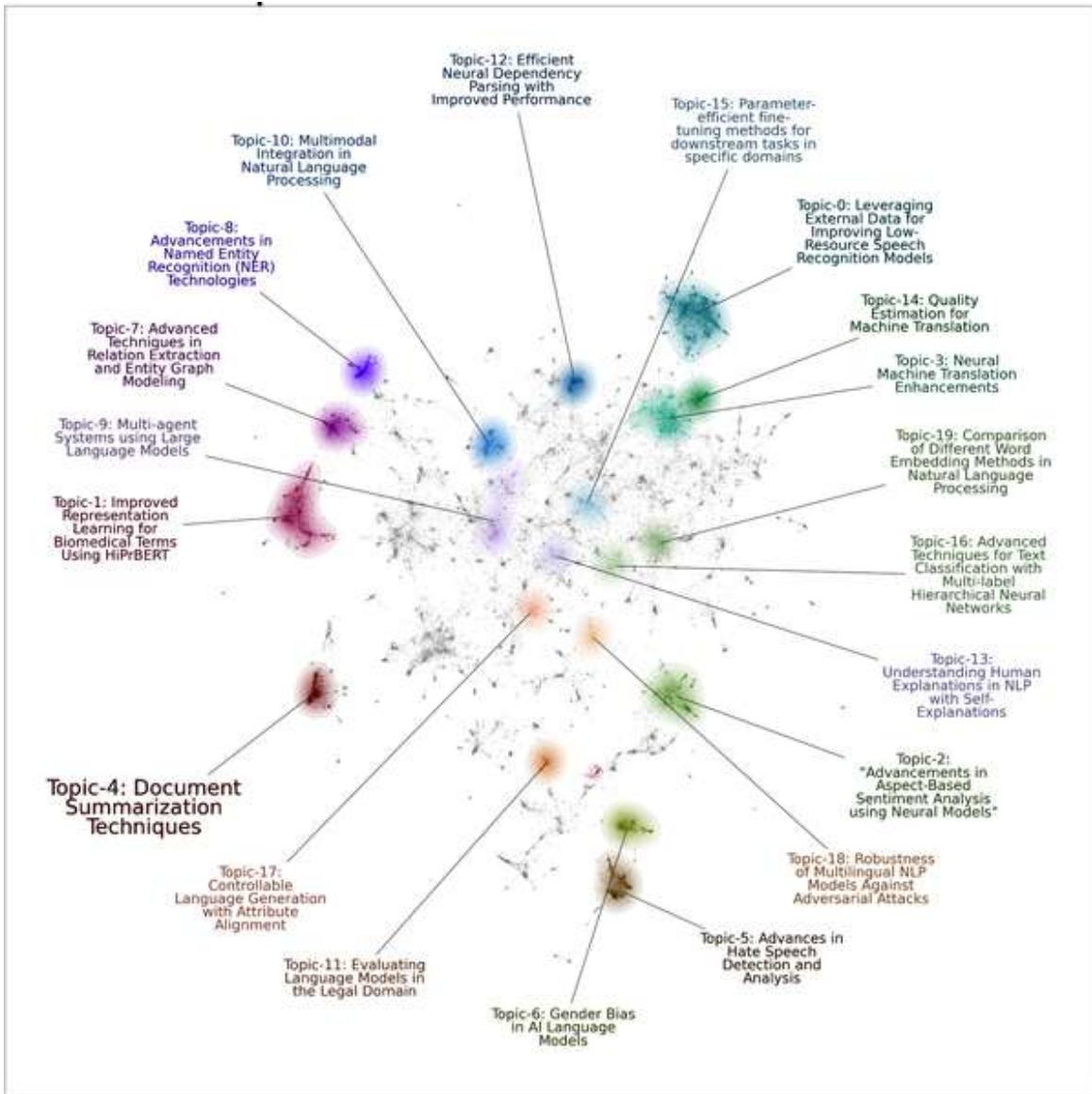


Figure 5-24. The top 20 topics visualized.

Summary

In this chapter, we explored how LLMs, both generative and representative, can be used in the domain of unsupervised learning. Despite supervised methods like classification being prevalent in recent years, unsupervised approaches such as text clustering hold immense potential due to their ability to group texts based on semantic content without prior labeling.

We covered a common pipeline for clustering textual documents that starts with converting input text into numerical representations, which we call embeddings. Then, dimensionality reduction is applied to these embeddings to simplify high-dimensional data for better clustering outcomes. Finally, a clustering algorithm on the dimensionality-reduced embeddings is applied to cluster the input text. Manually inspecting the clusters helped us understand which documents they contained and how to interpret these clusters.

To transition away from this manual inspection, we explored how BERTopic extends this text clustering pipeline with a method for automatically representing the clusters. This methodology is often referred to as topic modeling, which attempts to uncover themes within large amounts of documents. BERTopic generates these topic representations through a bag-of-words approach enhanced with c-TF-IDF, which weighs words based on their cluster relevance and frequency across all clusters.

A major benefit of BERTopic is its modular nature. In BERTopic, you can choose any model in the pipeline, which allows for additional representations of topics that create multiple perspectives of the same topic. We explored maximal marginal relevance and KeyBERTInspired as methodologies to fine-tune the topic representations generated with c-TF-IDF. Additionally, we used the same generative LLMs as in the previous chapter (Flan-T5 and GPT-3.5) to further improve the interpretability of topics by generating highly interpretable labels.

In the next chapter, we shift focus and explore a common method for improving the output of generative models, namely prompt engineering.

¹ Harold Hotelling. “Analysis of a complex of statistical variables into principal components.” *Journal of Educational Psychology* 24.6 (1933): 417.

² Leland McInnes, John Healy, and James Melville. “UMAP: Uniform Manifold Approximation and Projection for dimension reduction.” *arXiv preprint arXiv:1802.03426* (2018).

- ³ Leland McInnes, John Healy, and Steve Astels. “hdbscan: Hierarchical density based clustering.” *J. Open Source Softw.* 2.11 (2017): 205.
- ⁴ Martin Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise.” *KDD’96*, Aug. 1996: 226–231.
- ⁵ David M. Blei, Andrew Y. Ng, and Michael I. Jordan. “Latent Dirichlet allocation.” *Journal of Machine Learning Research* 3. Jan (2003): 993–1022.
- ⁶ Maarten Grootendorst. “BERTopic: Neural topic modeling with a class-based TF-IDF procedure.” *arXiv preprint arXiv:2203.05794* (2022).
- ⁷ Maarten Grootendorst. “KeyBERT: Minimal keyword extraction with BERT.” (2020).