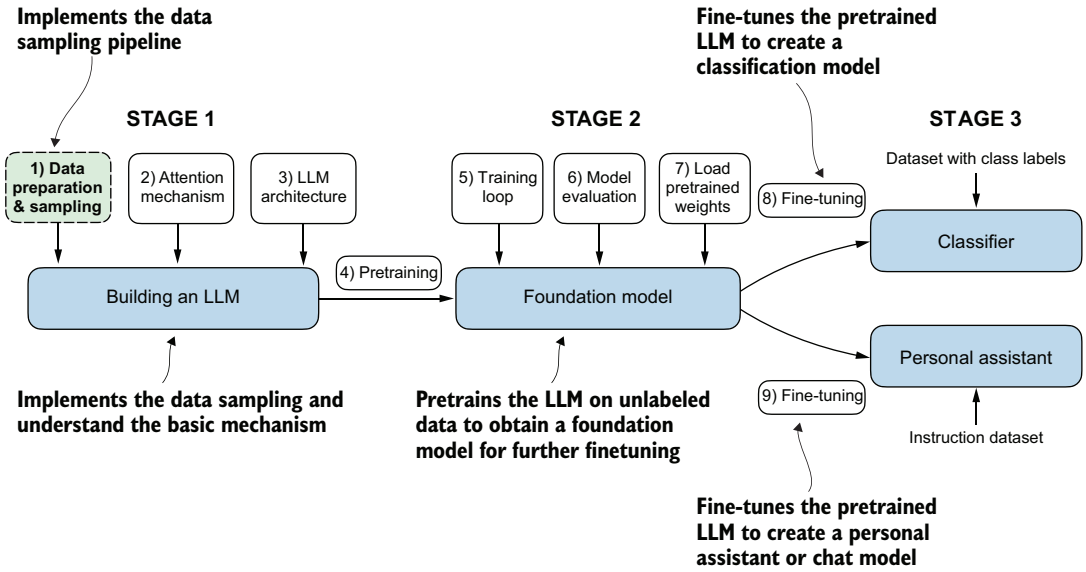

Figure 2.1 The three main stages of coding an LLM. This chapter focuses on step 1 of stage 1: implementing the data sample pipeline.

You'll learn how to prepare input text for training LLMs. This involves splitting text into individual word and subword tokens, which can then be encoded into vector representations for the LLM. You'll also learn about advanced tokenization schemes like byte pair encoding, which is utilized in popular LLMs like GPT. Lastly, we'll implement a sampling and data-loading strategy to produce the input-output pairs necessary for training LLMs.

2.1 Understanding word embeddings

Deep neural network models, including LLMs, cannot process raw text directly. Since text is categorical, it isn't compatible with the mathematical operations used to implement and train neural networks. Therefore, we need a way to represent words as continuous-valued vectors.

> **NOTE** Readers unfamiliar with vectors and tensors in a computational context can learn more in appendix A, section A.2.2.

The concept of converting data into a vector format is often referred to as *embedding*. Using a specific neural network layer or another pretrained neural network model, we can embed different data types—for example, video, audio, and text, as illustrated in figure 2.2. However, it's important to note that different data formats require distinct embedding models. For example, an embedding model designed for text would not be suitable for embedding audio or video data.

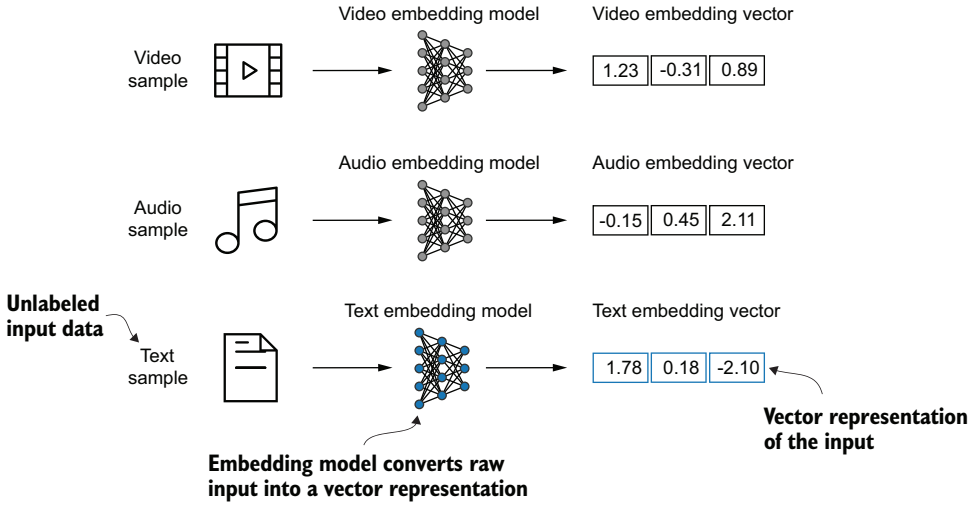

Figure 2.2 Deep learning models cannot process data formats like video, audio, and text in their raw form. Thus, we use an embedding model to transform this raw data into a dense vector representation that deep learning architectures can easily understand and process. Specifically, this figure illustrates the process of converting raw data into a three-dimensional numerical vector.

At its core, an embedding is a mapping from discrete objects, such as words, images, or even entire documents, to points in a continuous vector space—the primary purpose of embeddings is to convert nonnumeric data into a format that neural networks can process.

While word embeddings are the most common form of text embedding, there are also embeddings for sentences, paragraphs, or whole documents. Sentence or paragraph embeddings are popular choices for *retrieval-augmented generation*. Retrieval-augmented generation combines generation (like producing text) with retrieval (like searching an external knowledge base) to pull relevant information when generating text, which is a technique that is beyond the scope of this book. Since our goal is to train GPT-like LLMs, which learn to generate text one word at a time, we will focus on word embeddings.

Several algorithms and frameworks have been developed to generate word embeddings. One of the earlier and most popular examples is the *Word2Vec* approach. Word2Vec trained neural network architecture to generate word embeddings by predicting the context of a word given the target word or vice versa. The main idea behind Word2Vec is that words that appear in similar contexts tend to have similar meanings. Consequently, when projected into two-dimensional word embeddings for visualization purposes, similar terms are clustered together, as shown in figure 2.3.

Word embeddings can have varying dimensions, from one to thousands. A higher dimensionality might capture more nuanced relationships but at the cost of computational efficiency.

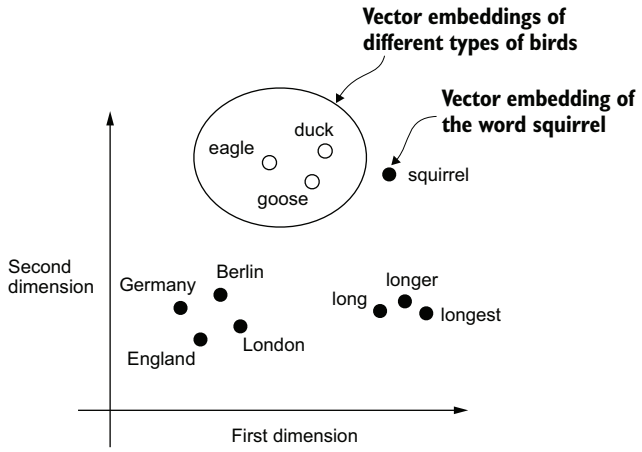

Figure 2.3 If word embeddings are two-dimensional, we can plot them in a two-dimensional scatterplot for visualization purposes as shown here. When using word embedding techniques, such as Word2Vec, words corresponding to similar concepts often appear close to each other in the embedding space. For instance, different types of birds appear closer to each other in the embedding space than in countries and cities.

While we can use pretrained models such as Word2Vec to generate embeddings for machine learning models, LLMs commonly produce their own embeddings that are part of the input layer and are updated during training. The advantage of optimizing the embeddings as part of the LLM training instead of using Word2Vec is that the embeddings are optimized to the specific task and data at hand. We will implement such embedding layers later in this chapter. (LLMs can also create contextualized output embeddings, as we discuss in chapter 3.)

Unfortunately, high-dimensional embeddings present a challenge for visualization because our sensory perception and common graphical representations are inherently limited to three dimensions or fewer, which is why figure 2.3 shows two-dimensional embeddings in a two-dimensional scatterplot. However, when working with LLMs, we typically use embeddings with a much higher dimensionality. For both GPT-2 and GPT-3, the embedding size (often referred to as the dimensionality of the model's hidden states) varies based on the specific model variant and size. It is a tradeoff between performance and efficiency. The smallest GPT-2 models (117M and 125M parameters) use an embedding size of 768 dimensions to provide concrete examples. The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

Next, we will walk through the required steps for preparing the embeddings used by an LLM, which include splitting text into words, converting words into tokens, and turning tokens into embedding vectors.