set of class labels, such as "spam" and "not spam." Examples of classification tasks extend beyond LLMs and email filtering: they include identifying different species of plants from images; categorizing news articles into topics like sports, politics, and technology; and distinguishing between benign and malignant tumors in medical imaging.

The key point is that a classification fine-tuned model is restricted to predicting classes it has encountered during its training. For instance, it can determine whether something is "spam" or "not spam," as illustrated in figure 6.3, but it can't say anything else about the input text.

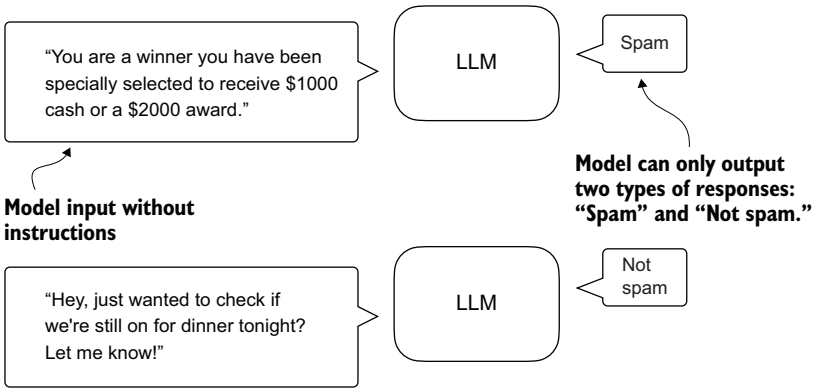

Figure 6.3 A text classification scenario using an LLM. A model fine-tuned for spam classification does not require further instruction alongside the input. In contrast to an instruction fine-tuned model, it can only respond with "spam" or "not spam."

In contrast to the classification fine-tuned model depicted in figure 6.3, an instruction fine-tuned model typically can undertake a broader range of tasks. We can view a classification fine-tuned model as highly specialized, and generally, it is easier to develop a specialized model than a generalist model that works well across various tasks.

Choosing the right approach

Instruction fine-tuning improves a model's ability to understand and generate responses based on specific user instructions. Instruction fine-tuning is best suited for models that need to handle a variety of tasks based on complex user instructions, improving flexibility and interaction quality. Classification fine-tuning is ideal for projects requiring precise categorization of data into predefined classes, such as sentiment analysis or spam detection.

While instruction fine-tuning is more versatile, it demands larger datasets and greater computational resources to develop models proficient in various tasks. In contrast, classification fine-tuning requires less data and compute power, but its use is confined to the specific classes on which the model has been trained.

6.2 Preparing the dataset

We will modify and classification fine-tune the GPT model we previously implemented and pretrained. We begin by downloading and preparing the dataset, as highlighted in figure 6.4. To provide an intuitive and useful example of classification fine-tuning, we will work with a text message dataset that consists of spam and non-spam messages.

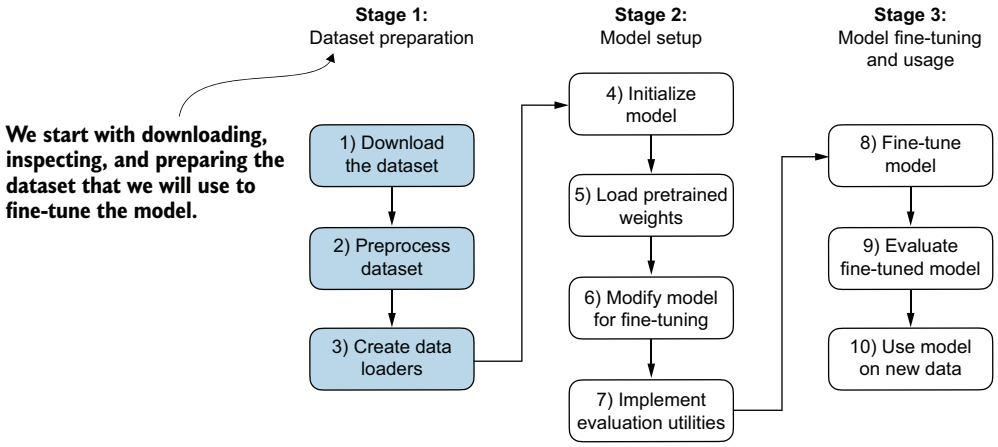

Figure 6.4 The three-stage process for classification fine-tuning an LLM. Stage 1 involves dataset preparation. Stage 2 focuses on model setup. Stage 3 covers fine-tuning and evaluating the model.

> **NOTE** Text messages typically sent via phone, not email. However, the same steps also apply to email classification, and interested readers can find links to email spam classification datasets in appendix B.

The first step is to download the dataset.

Listing 6.1 Downloading and unzipping the dataset

```
import urllib.request
import zipfile
import os
from pathlib import Path

url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"


def download_and_unzip_spam_data(
        url, zip_path, extracted_path, data_file_path):
    if data_file_path.exists():
```

```
        print(f"{data_file_path} already exists. Skipping download "
              "and extraction."
        )
        return

    with urllib.request.urlopen(url) as response:
        with open(zip_path, "wb") as out_file:
            out_file.write(response.read())

    with zipfile.ZipFile(zip_path, "r") as zip_ref:
        zip_ref.extractall(extracted_path)

    original_file_path = Path(extracted_path) / "SMSSpamCollection"
    os.rename(original_file_path, data_file_path)
    print(f"File downloaded and saved as {data_file_path}")

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)
```

Downloads the file

Unzips the file

Adds a .tsv file extension

After executing the preceding code, the dataset is saved as a tab-separated text file, `SMSSpamCollection.tsv`, in the `sms_spam_collection` folder. We can load it into a pandas `DataFrame` as follows:

```
import pandas as pd
df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
df
```

Renders the data frame in a Jupyter notebook. Alternatively, use print(df).

Figure 6.5 shows the resulting data frame of the spam dataset.

	Label	Text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...	...	...
5571	ham	Rofl. Its true to its name

5572 rows × 2 columns

Figure 6.5 Preview of the `SMSSpamCollection` dataset in a pandas `DataFrame`, showing class labels ("ham" or "spam") and corresponding text messages. The dataset consists of 5,572 rows (text messages and labels).

Let's examine the class label distribution:

```
print(df["Label"].value_counts())
```