Executing the previous code, we find that the data contains "ham" (i.e., not spam) far more frequently than "spam":

```
Label
ham     4825
spam     747
Name: count, dtype: int64
```

For simplicity, and because we prefer a small dataset (which will facilitate faster fine-tuning of the LLM), we choose to undersample the dataset to include 747 instances from each class.

> **NOTE** There are several other methods to handle class imbalances, but these are beyond the scope of this book. Readers interested in exploring methods for dealing with imbalanced data can find additional information in appendix B.

We can use the code in the following listing to undersample and create a balanced dataset.

Listing 6.2 Creating a balanced dataset

```
def create_balanced_dataset(df):
    num_spam = df[df["Label"] == "spam"].shape[0]
    ham_subset = df[df["Label"] == "ham"].sample(
        num_spam, random_state=123
    )
    balanced_df = pd.concat([
        ham_subset, df[df["Label"] == "spam"]
    ])
    return balanced_df

balanced_df = create_balanced_dataset(df)
print(balanced_df["Label"].value_counts())
```

Counts the instances of "spam"

Randomly samples "ham" instances to match the number of "spam" instances

Combines ham subset with "spam"

After executing the previous code to balance the dataset, we can see that we now have equal amounts of spam and non-spam messages:

```
Label
ham     747
spam    747
Name: count, dtype: int64
```

Next, we convert the "string" class labels `"ham"` and `"spam"` into integer class labels 0 and 1, respectively:

```
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})
```

This process is similar to converting text into token IDs. However, instead of using the GPT vocabulary, which consists of more than 50,000 words, we are dealing with just two token IDs: 0 and 1.

Next, we create a `random_split` function to split the dataset into three parts: 70% for training, 10% for validation, and 20% for testing. (These ratios are common in machine learning to train, adjust, and evaluate models.)

Listing 6.3 Splitting the dataset

```
def random_split(df, train_frac, validation_frac):
    df = df.sample(
        frac=1, random_state=123
    ).reset_index(drop=True)
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(len(df) * validation_frac)

    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df

train_df, validation_df, test_df = random_split(
    balanced_df, 0.7, 0.1)
```

Shuffles the entire DataFrame

Calculates split indices

Splits the DataFrame

Test size is implied to be 0.2 as the remainder.

Let's save the dataset as CSV (comma-separated value) files so we can reuse it later:

```
train_df.to_csv("train.csv", index=None)
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Thus far, we have downloaded the dataset, balanced it, and split it into training and evaluation subsets. Now we will set up the PyTorch data loaders that will be used to train the model.

6.3 Creating data loaders

We will develop PyTorch data loaders conceptually similar to those we implemented while working with text data. Previously, we utilized a sliding window technique to generate uniformly sized text chunks, which we then grouped into batches for more efficient model training. Each chunk functioned as an individual training instance. However, we are now working with a spam dataset that contains text messages of varying lengths. To batch these messages as we did with the text chunks, we have two primary options:

- Truncate all messages to the length of the shortest message in the dataset or batch.
- Pad all messages to the length of the longest message in the dataset or batch.

The first option is computationally cheaper, but it may result in significant information loss if shorter messages are much smaller than the average or longest messages,

potentially reducing model performance. So, we opt for the second option, which preserves the entire content of all messages.

To implement batching, where all messages are padded to the length of the longest message in the dataset, we add padding tokens to all shorter messages. For this purpose, we use `"<|endoftext|>"` as a padding token.

However, instead of appending the string `"<|endoftext|>"` to each of the text messages directly, we can add the token ID corresponding to `"<|endoftext|>"` to the encoded text messages, as illustrated in figure 6.6. `50256` is the token ID of the padding token `"<|endoftext|>"`. We can double-check whether the token ID is correct by encoding the `"<|endoftext|>"` using the *GPT-2 tokenizer* from the `tiktoken` package that we used previously:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

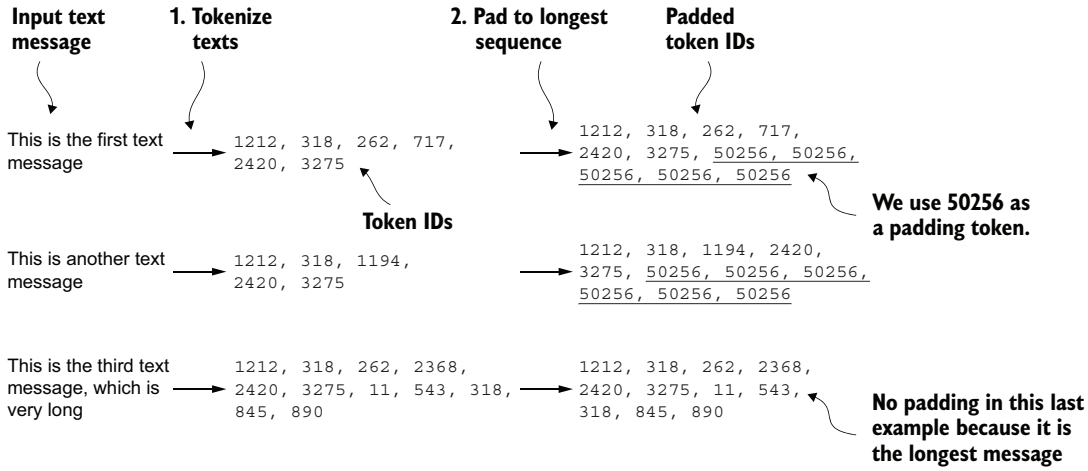

Figure 6.6 The input text preparation process. First, each input text message is converted into a sequence of token IDs. Then, to ensure uniform sequence lengths, shorter sequences are padded with a padding token (in this case, token ID 50256) to match the length of the longest sequence.

Indeed, executing the preceding code returns `[50256]`.

We first need to implement a PyTorch `Dataset`, which specifies how the data is loaded and processed before we can instantiate the data loaders. For this purpose, we define the `SpamDataset` class, which implements the concepts in figure 6.6. This `SpamDataset` class handles several key tasks: it encodes the text messages into token sequences, identifies the longest sequence in the training dataset, and ensures that all other sequences are padded with a *padding token* to match the length of the longest sequence.