Chapter 4. Text Classification

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mcronin@oreilly.com*.

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.

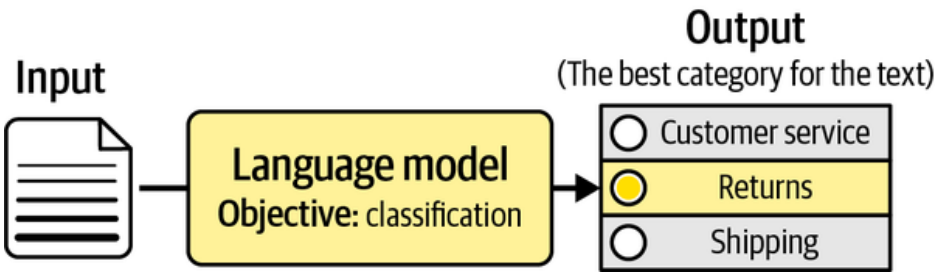

Figure 4-1. Using a language model to classify text.

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:

- The "Text Classification with Representation Models" section demonstrates the flexibility of nongenerative models for classification. We will cover both task-specific models and embedding models.

- The “Text Classification with Generative Models” section 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 pre-trained 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, training such models will be discussed throughout Part III of the book.

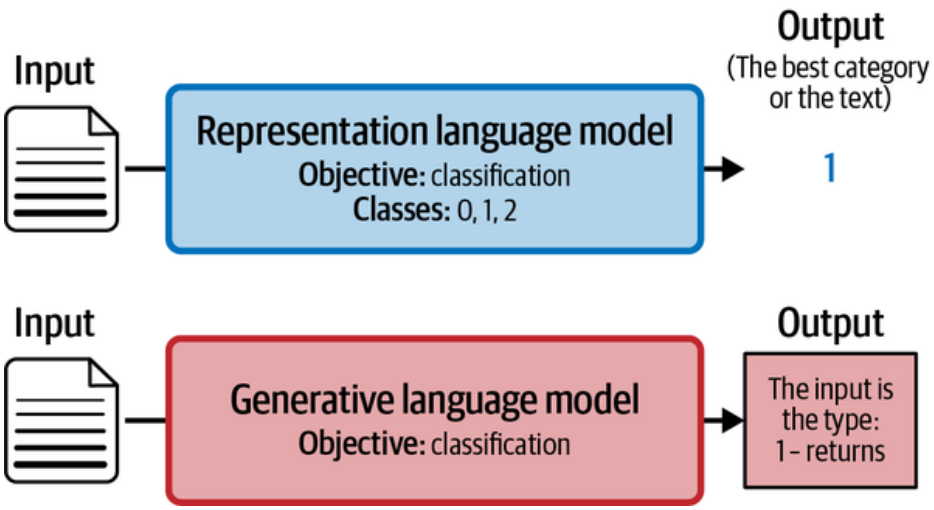

Figure 4-2. The stages of creating and tuning a language model. This chapter will focus on applying models that have already gone through this process.

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

The data we use to explore techniques for classifying text can be found 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("rotten_tomatoes")
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 pre-trained 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.

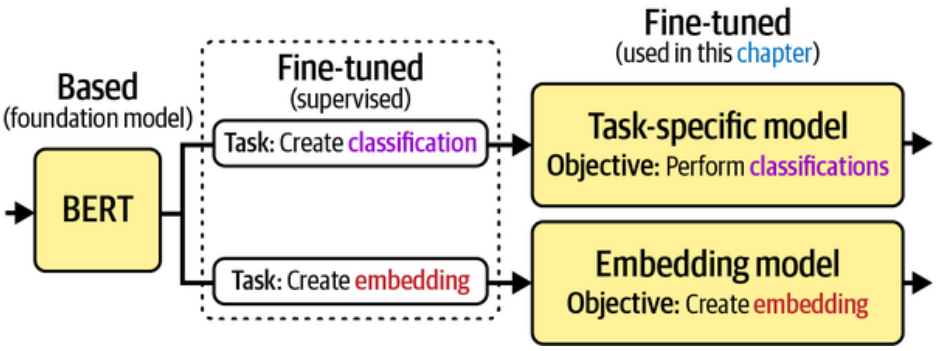

Figure 4-3. A foundation model is fine-tuned for specific tasks, for instance, to perform classification or generate general-purpose embeddings.

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* (non-trainable) and only use their output as shown in Figure 4-4.

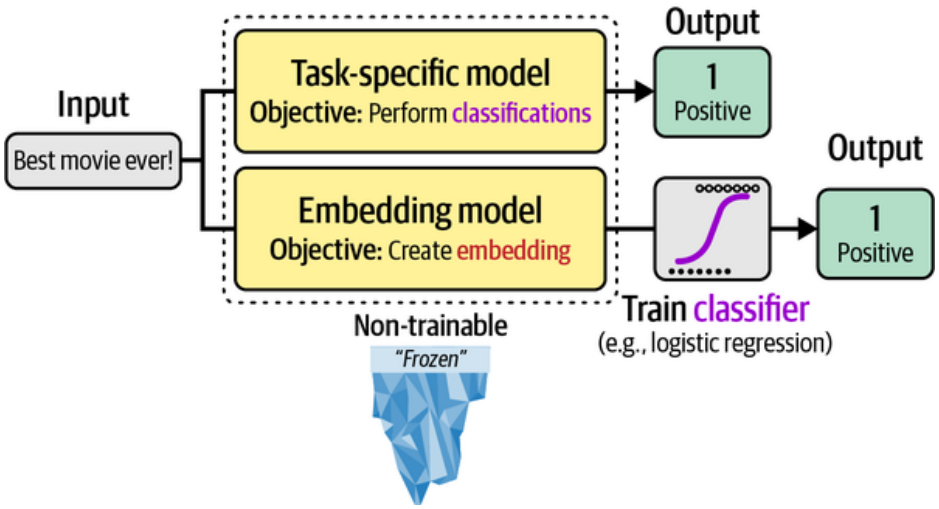

Figure 4-4. Perform classification directly with a task-specific model or indirectly with general-purpose embeddings.

In this section, we will leverage pre-trained 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 50,000 models on the Hugging Face Hub for text classification and more than 7,000 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.

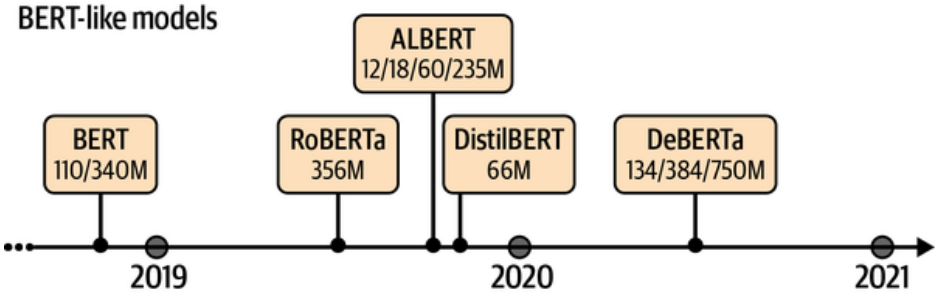

Figure 4-5. A timeline of common BERT-like model releases. These are considered foundation models and are mostly intended to be fine-tuned on a downstream task.

Selecting the right model for the job can be a form of art in itself. Trying thousands of pre-trained 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-mpnet-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.

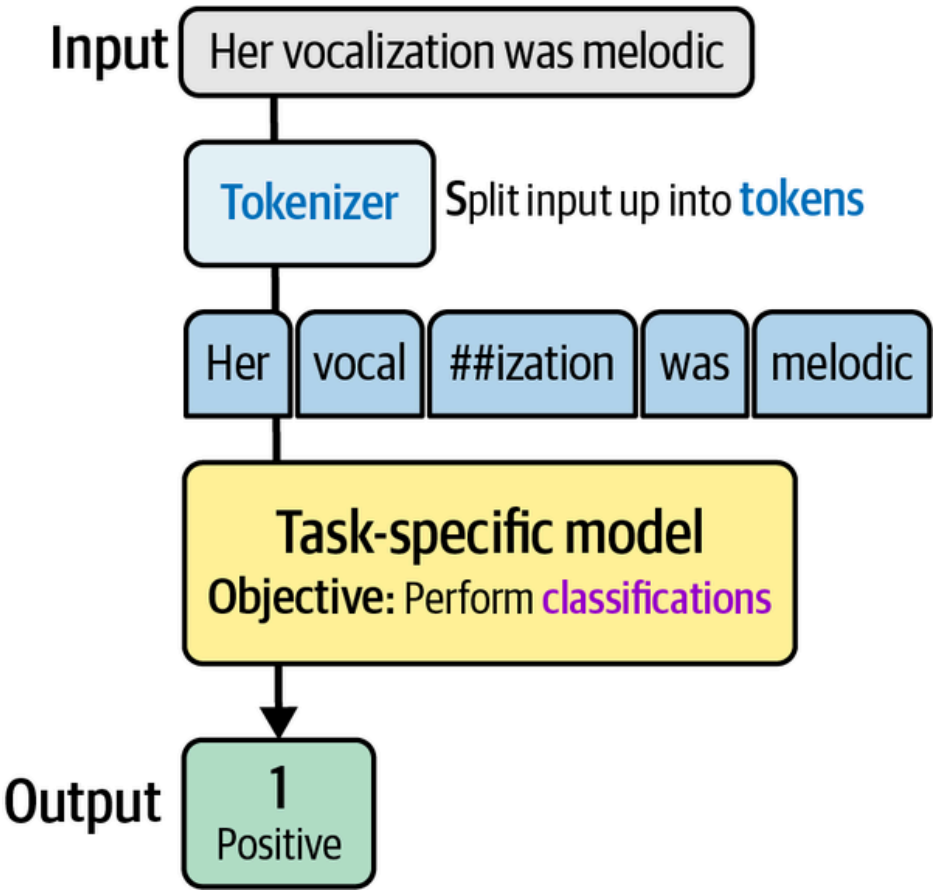

Figure 4-6. An input sentence is first fed to a tokenizer before it can be processed by the task-specific model.

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.

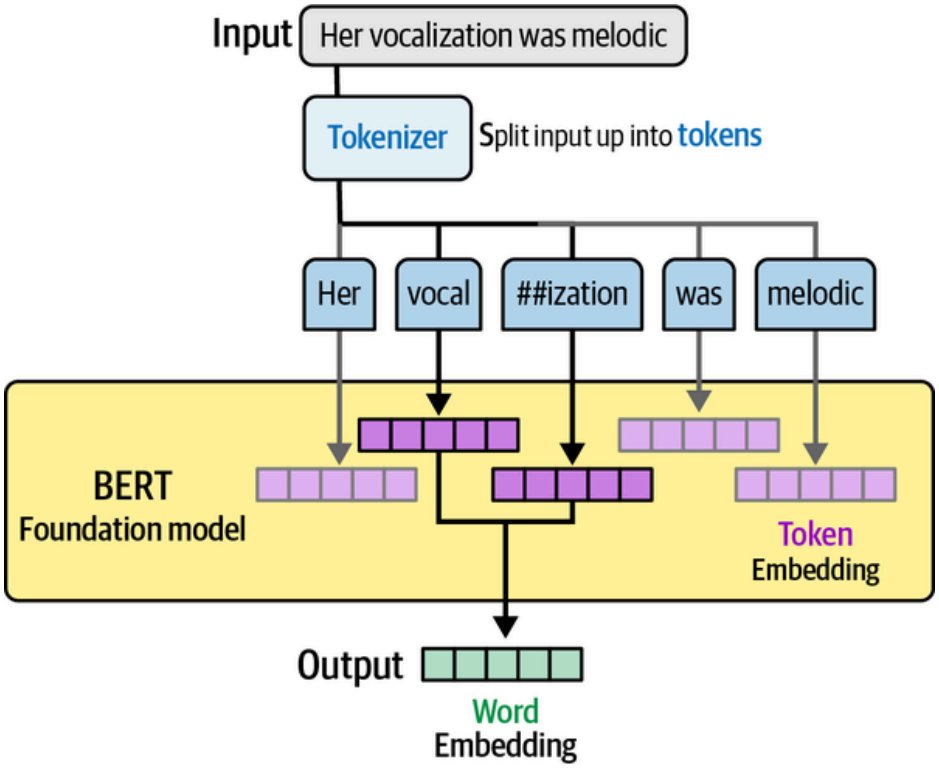

Figure 4-7. By breaking down an unknown word into tokens, word embeddings can still be generated.

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.

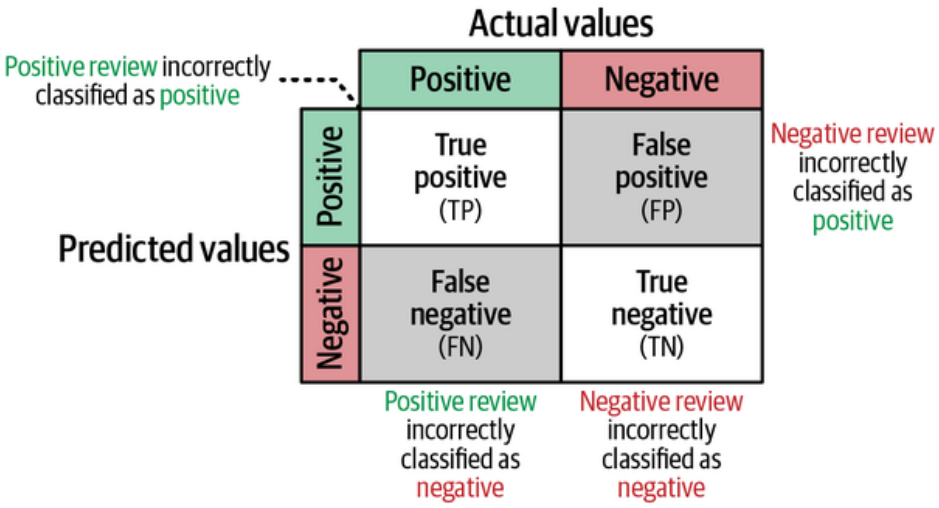

Figure 4-8. The confusion matrix describes four types of predictions we can make.

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 three 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.

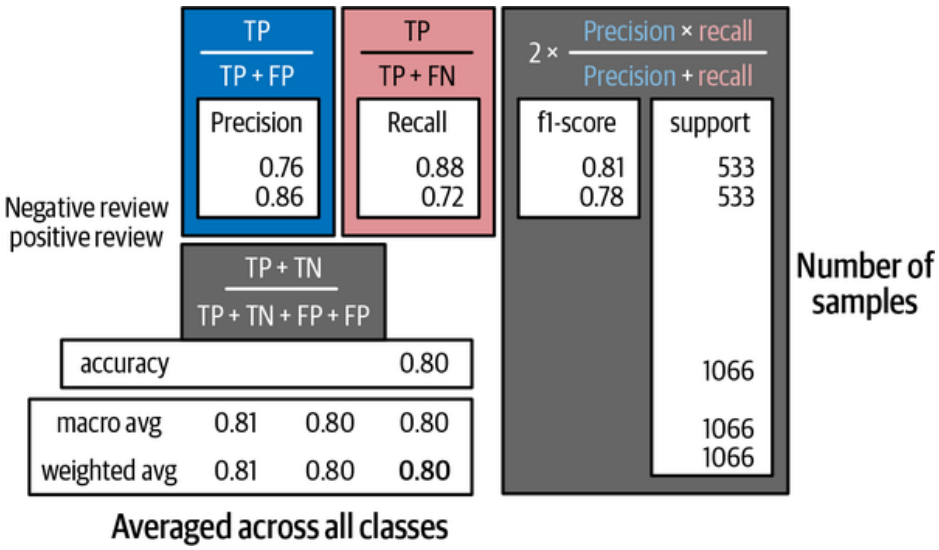

Figure 4-9. The classification report describes several metrics for evaluating a model's performance.

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 pre-trained 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 pre-trained task-specific model for sentiment analysis. However, what if we cannot find a model that was pre-trained 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.

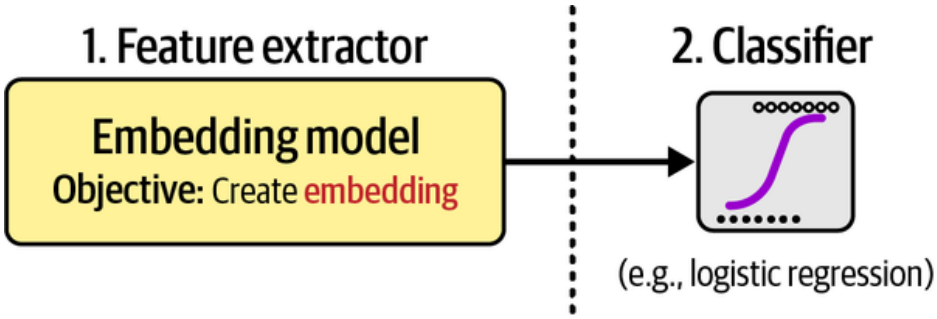

Figure 4-10. The feature extraction step and classification steps are separated.

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.

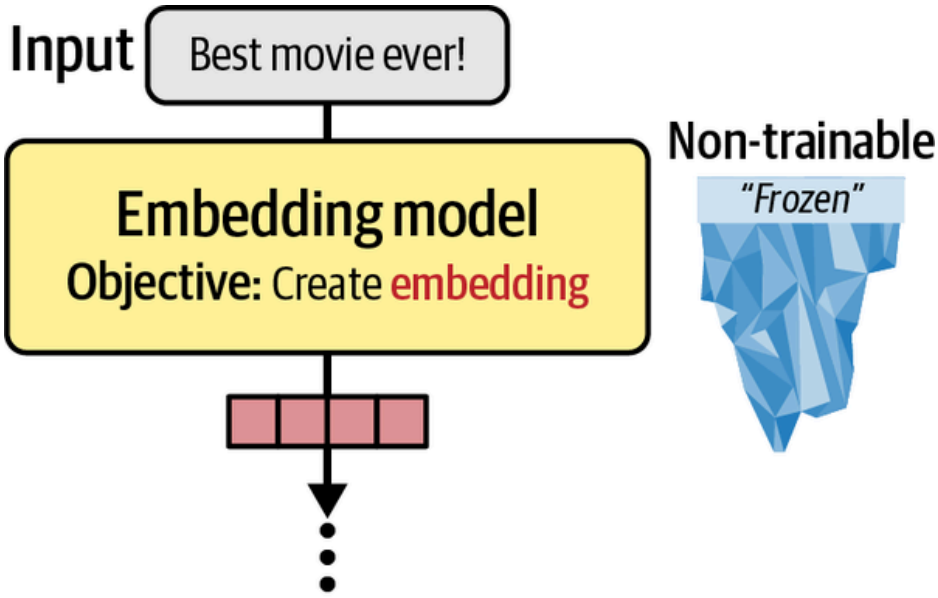

Figure 4-11. In step 1, we use the embedding model to extract the features and convert the input text to embeddings.

We can perform this step with `sentence-transformer`, a popular package for leveraging pre-trained 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.

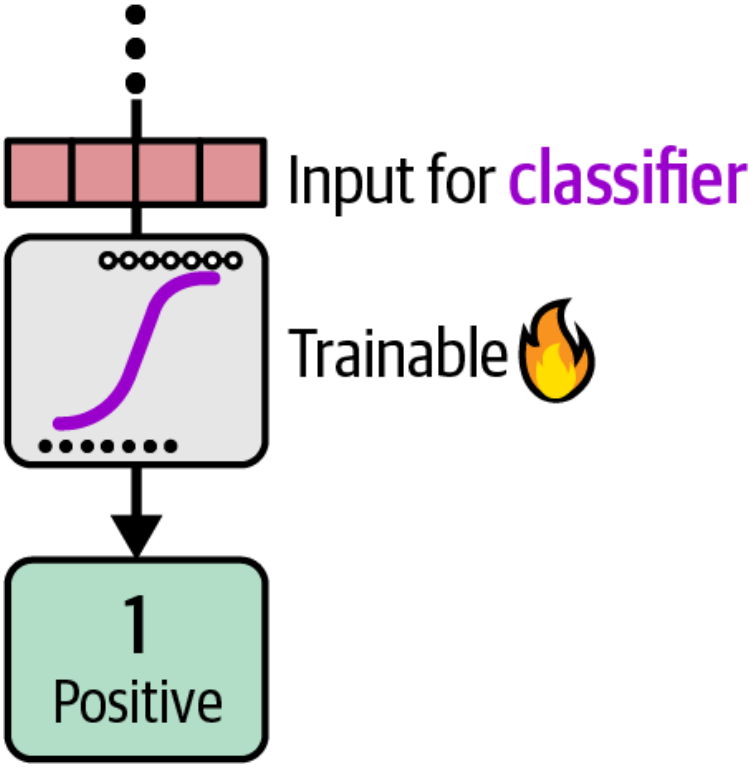

Figure 4-12. Using the embeddings as our features, we train a logistic regression model on our training data.

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 light-weight 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.

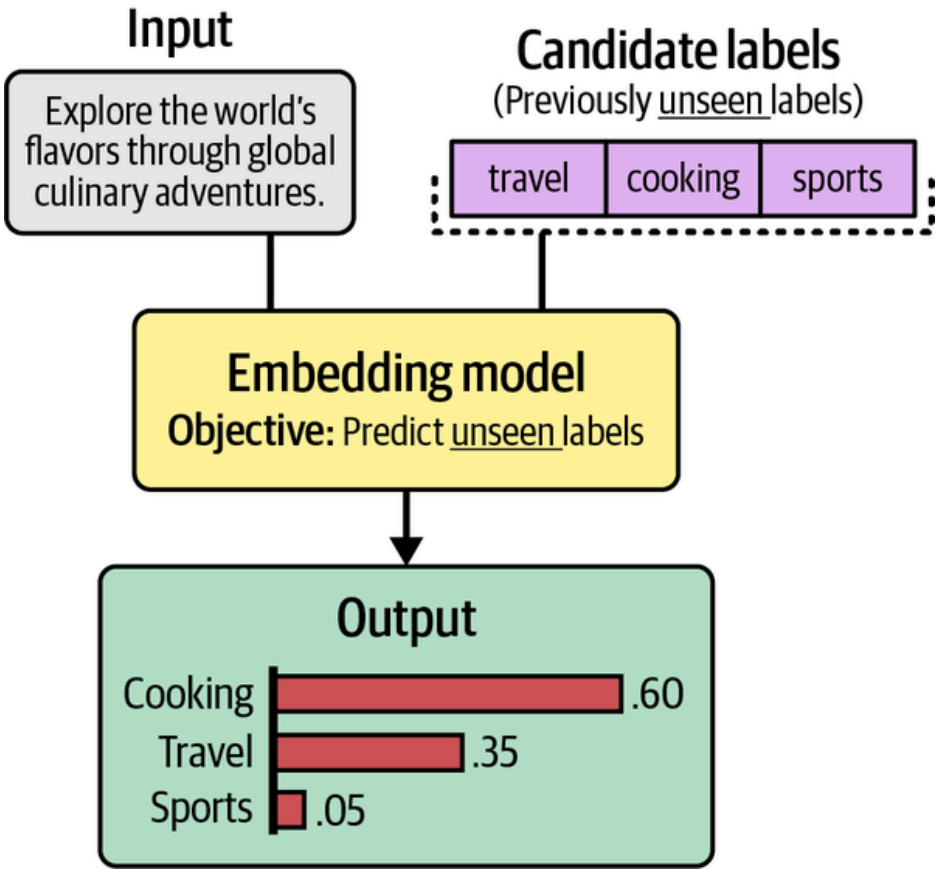

Figure 4-13. In zero-shot classification, we have no labeled data, only the labels themselves. The zero-shot model decides how the input is related to the candidate labels.

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.

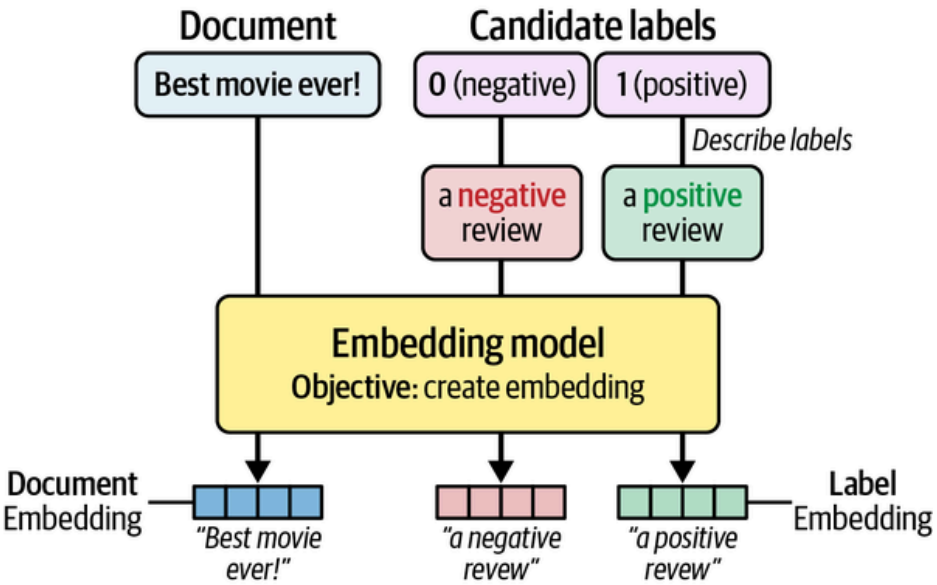

Figure 4-14. To embed the labels, we first need to give them a description, such as "a negative movie review." This description can then be embedded through sentence-transformers.

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.

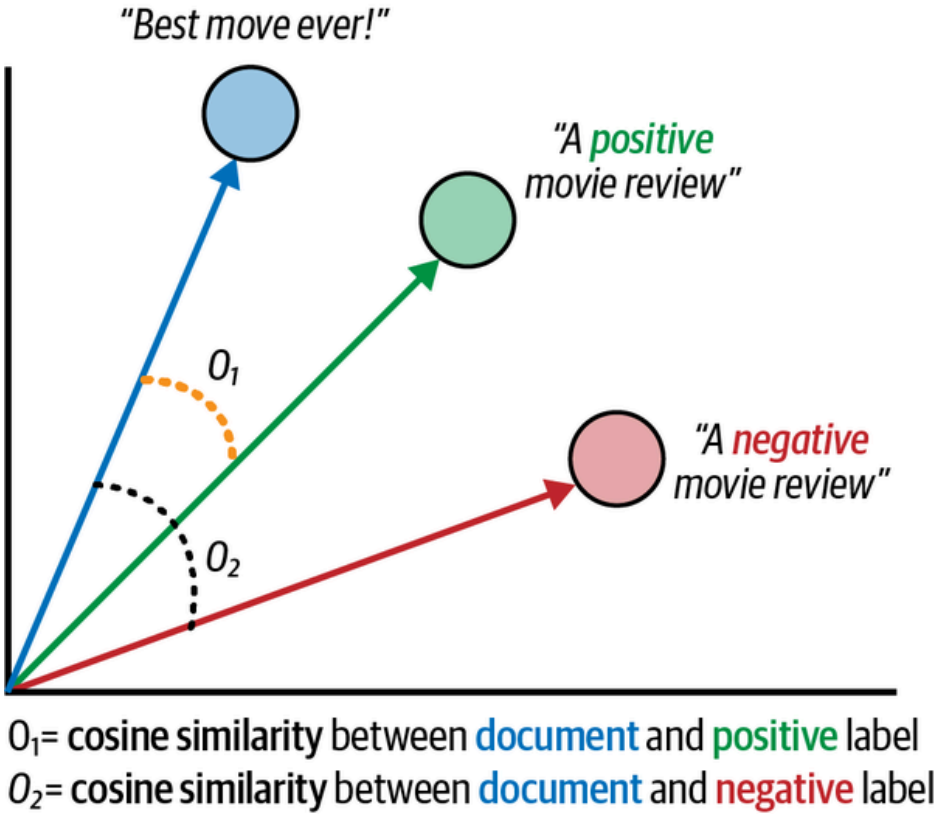

Figure 4-15. The cosine similarity is the angle between two vectors or embeddings. In this example, we calculate the similarity between a document and the two possible labels, positive and negative.

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.

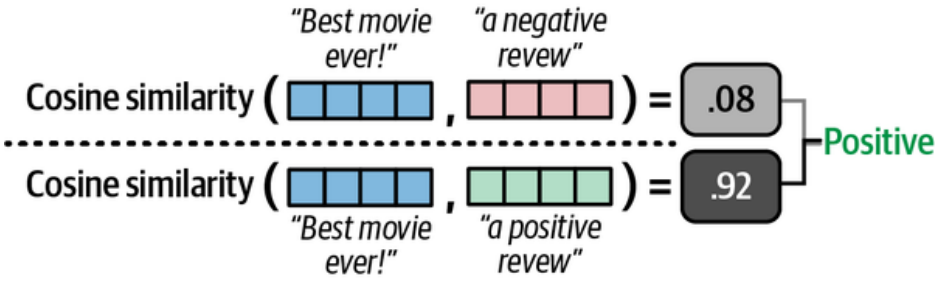

Figure 4-16. After embedding the label descriptions and the documents, we can use cosine similarity for each label document pair.

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 towards 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. P

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.

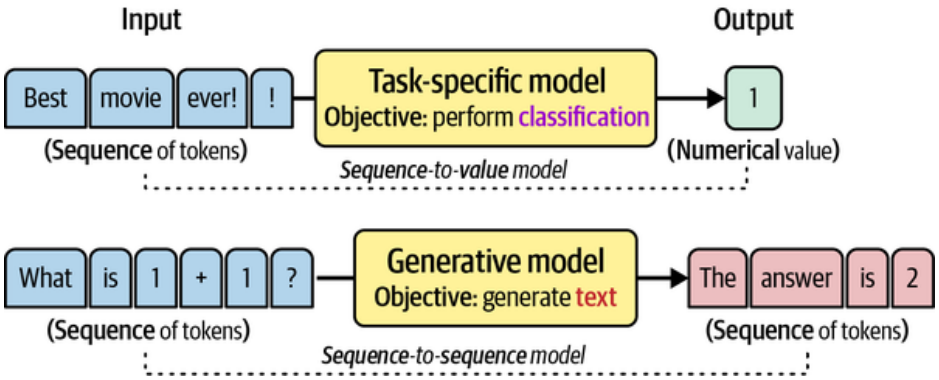

Figure 4-17. A task-specific model generates numerical values from sequences of tokens while a generative model generates sequences of tokens from sequences of tokens.

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 were to 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 towards 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*.

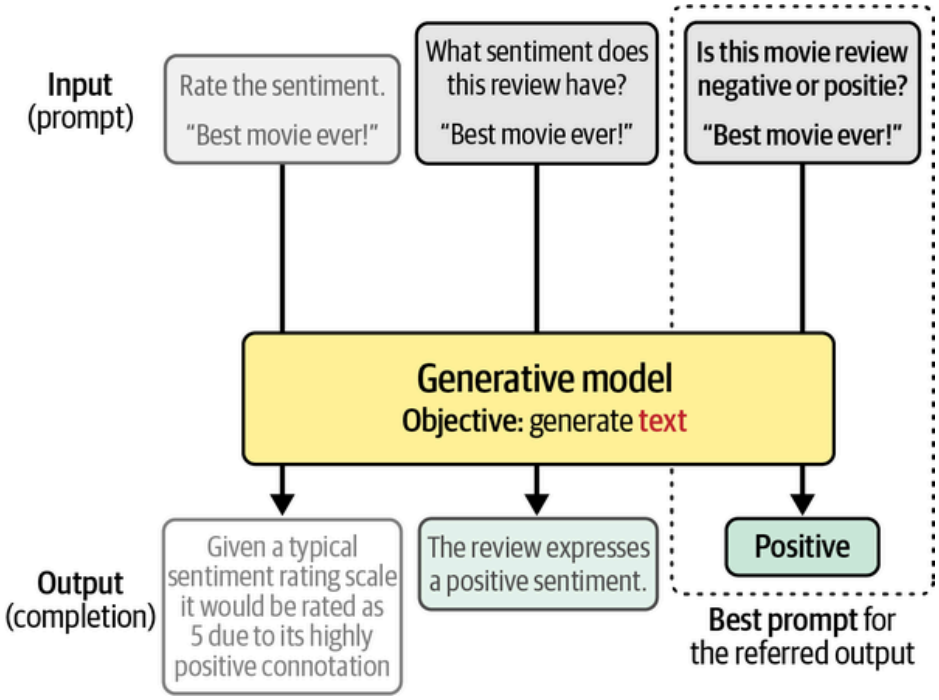

Figure 4-18. Prompt engineering allows prompts to be updated to improve the output generated by the model.

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, their architecture is similar to the original Transformer where 12 decoders and 12 encoders are stacked together.[7]

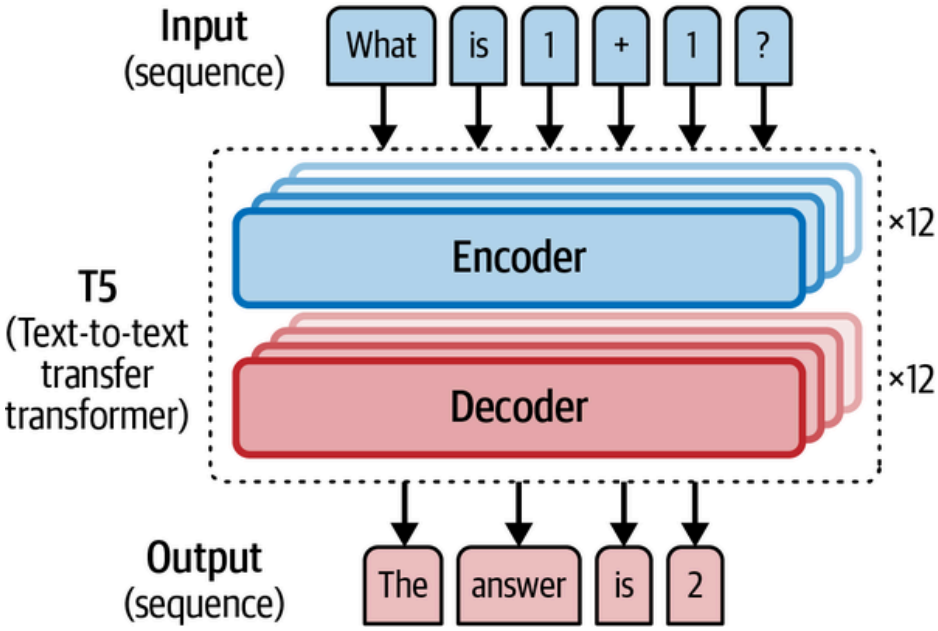

Figure 4-19. The T5 architecture is similar to the original Transformer model, a decoder-encoder architecture.

With this architecture, these models were first pre-trained 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 pre-training.

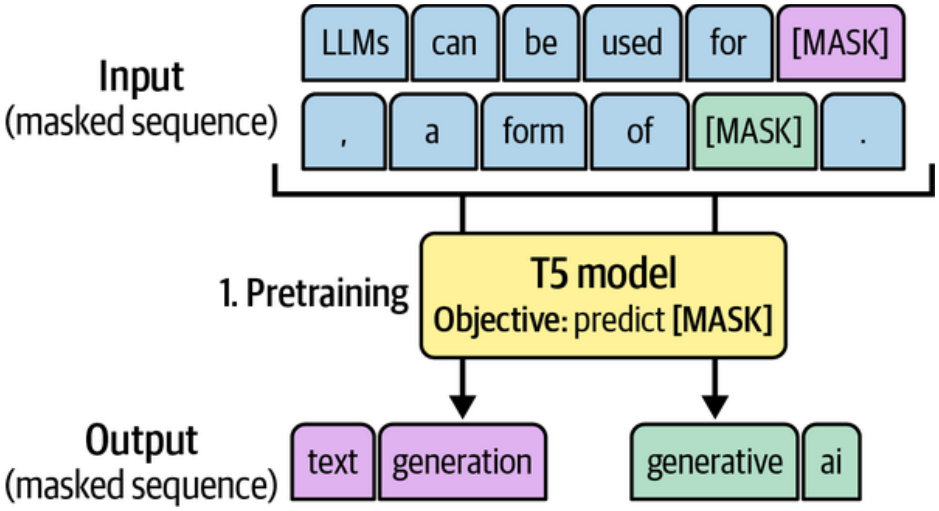

Figure 4-20. In the first step of training, namely pre-training, the T5 model needs to predict masks that could contain multiple tokens.

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.

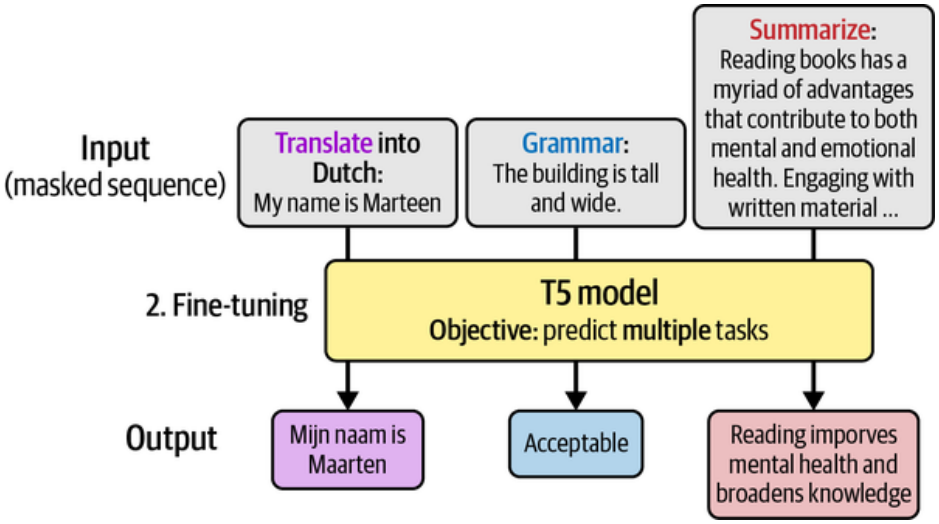

Figure 4-21. By converting specific tasks to textual instructions, the T5 model can be trained on a variety of tasks during fine-tuning.

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 pre-trained 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 their model.

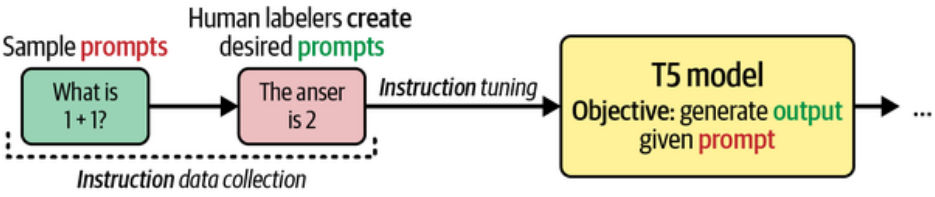

Figure 4-22. Manually labeled data consisting of an instruction (prompt) and output was used to perform fine-tuning (instruction-tuning).

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 their final model, ChatGPT.

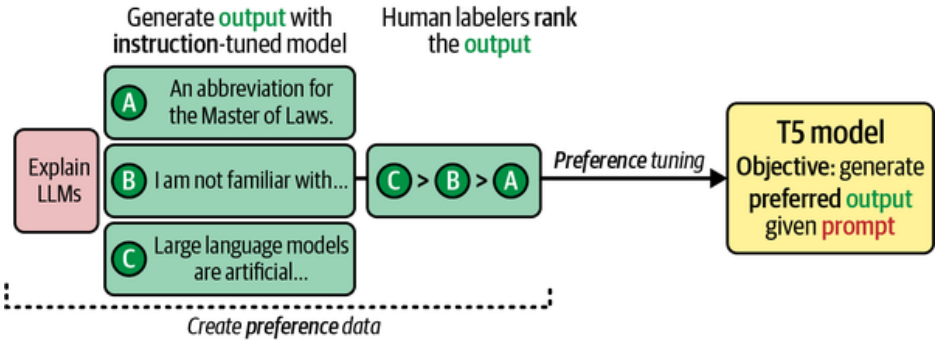

Figure 4-23. Manually ranked preference data was used to generate the final model, ChatGPT.

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://openai.com/* and create an API key here: *https://platform.openai.com/api-keys*. 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

[DOCUMENT]

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

# 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 pre-trained on a large dataset specifically for sentiment analysis and showed us that pre-trained 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).