Chapter 4 - Text Classification

What is Text Classification?

Text classification is assigning labels to text. Common applications:

- Sentiment analysis (positive/negative)
- Intent detection (what does the user want?)
- Entity extraction (finding names, places, etc.)
- Language detection

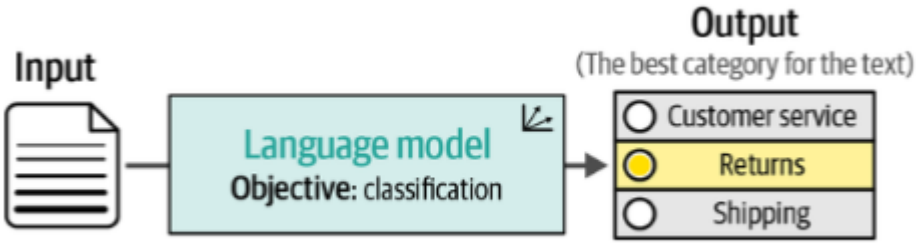

Figure 4-1. Using a language model to classify text.

This chapter covers two main approaches:

1. **Representation models** (BERT-like) - output class directly
2. **Generative models** (GPT-like) - output text that we parse into classes

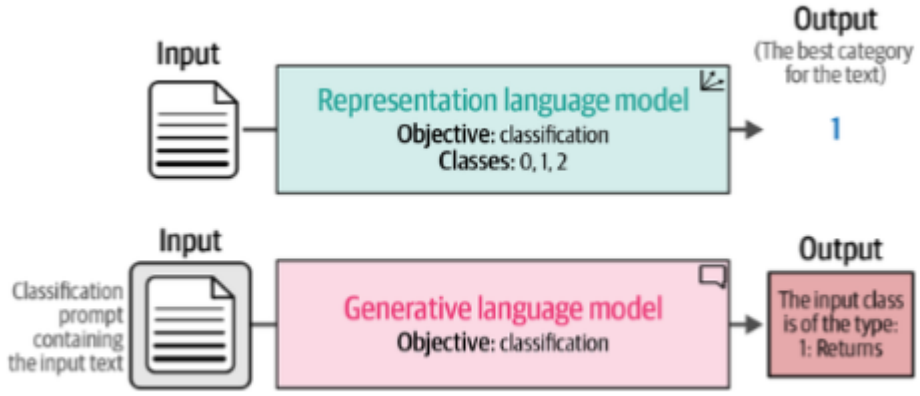

Figure 4-2. Although both representation and generative models can be used for classification, their approaches differ.

Both work. They have different trade-offs.

The Dataset

We'll use the Rotten Tomatoes movie review dataset. It has:

- 5,331 positive reviews
- 5,331 negative reviews
- Binary classification task (positive vs negative)

Loading the data:

```
from datasets import load_dataset

# Load the data
data = load_dataset("rotten_tomatoes")
```

This outputs:

```
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
    })
})
```

As you can see above, the data is split into *train*, *test*, *validation* splits. In this chapter, the *train* split will be used

Example reviews:

```
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]
}
```

The output shows one positive review (label 1) and one negative review (label 0).

Text Classification with Representation Models

Representation models are encoder-only architectures like BERT. They're trained to understand language, not generate it.

Two flavours:

1. **Task-specific models** - fine-tuned for one task (e.g., sentiment analysis)
2. **Embedding models** - generate general-purpose vectors you can use for anything

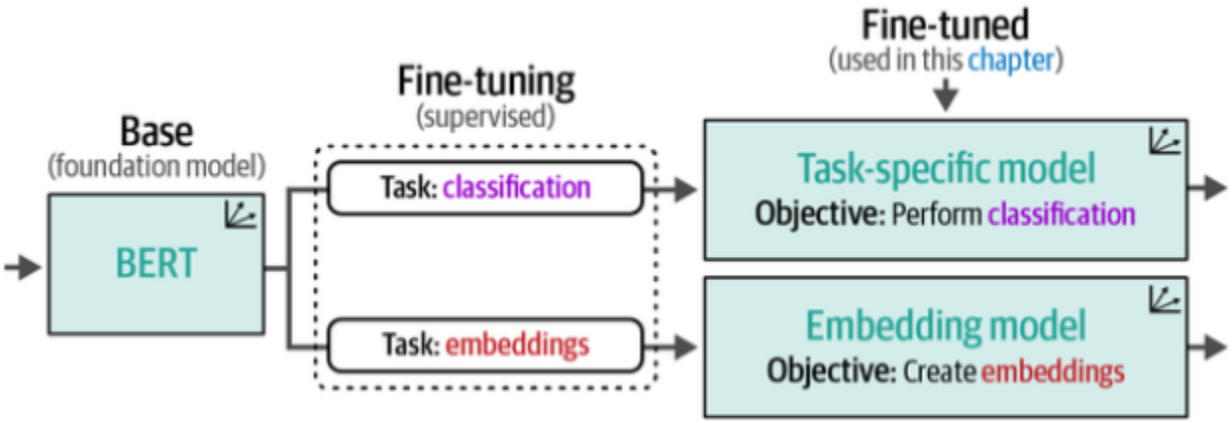

Figure 4-3. A foundation model is fine-tuned for specific tasks; for instance, to perform classification or generate general-purpose embeddings.

In this chapter, we keep both models frozen (nontrainable) and use their output as shown in the figure below.

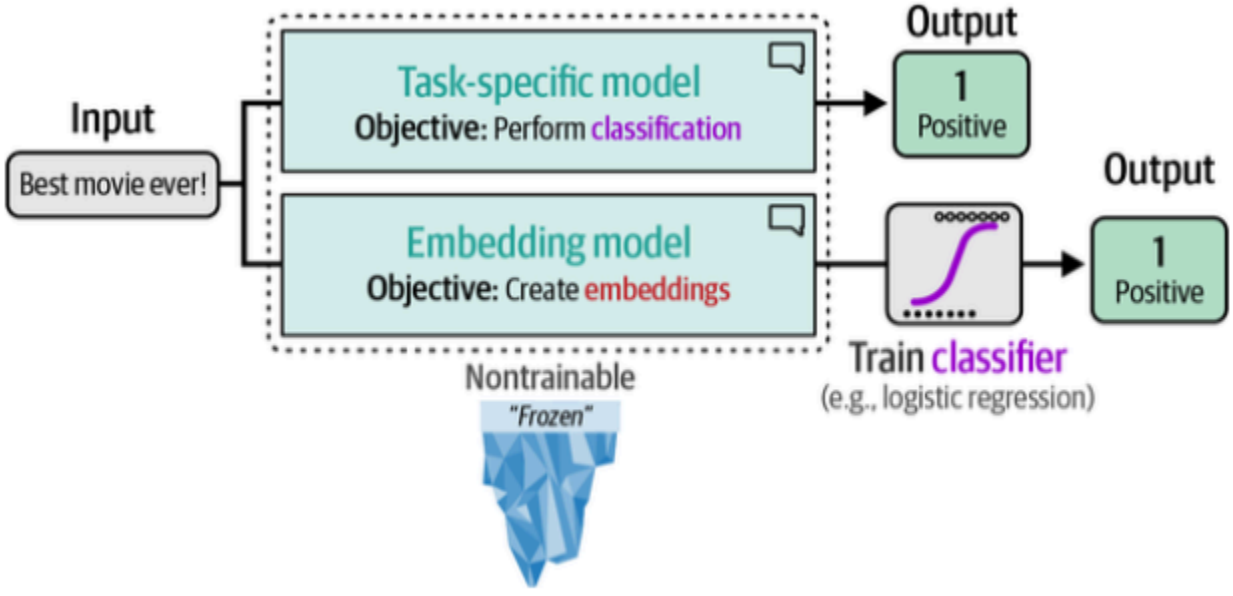

Figure 4-4. Perform classification directly with a task-specific model or indirectly with general-purpose embeddings.

Model Selection

Hugging Face Hub has more than 60,000 models for text classification over 8,000 models for generating embeddings as of the time of writing of the book. When choosing a model, it is important to take into account the use case to know which architecture, size, and language compatibility to consider.

As discussed in Chapter 1, BERT is a well-known ***encoder-only*** architecture. It is a popular choice for creating task-specific and embedding models.

Good baseline models to try:

- `bert-base-uncased` (110M params)
- `roberta-base` (125M params)
- `distilbert-base-uncased` (66M params - faster)
- `microsoft/deberta-base` (134M params)
- `bert-tiny`
- `ALBERT base v2`

BERT variants over time:

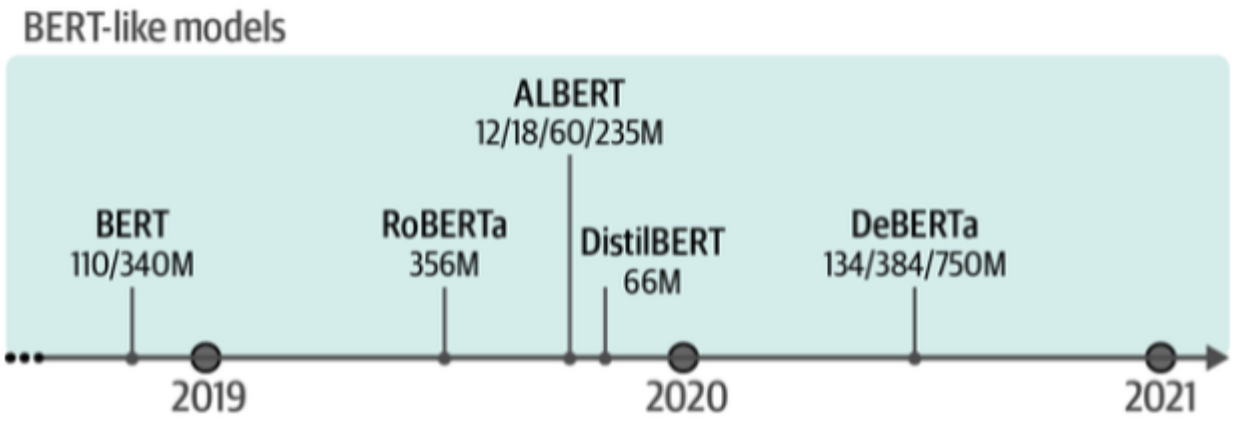

Figure 4-5. A timeline of common BERT-like model releases. These are considered foundation models and are mostly intended to be fine-tuned on a downstream task.

For embeddings:
Check the MTEB leaderboard for rankings. We'll use `sentence-transformers/all-mpnet-base-v2` - small but performs well.

For this chapter:

- Task-specific: `cardiffnlp/twitter-roberta-base-sentiment-latest`
- Embeddings: `sentence-transformers/all-mpnet-base-v2`

Using a Task-Specific Model

Task-specific models are the simplest approach. Load model, pass text, get prediction.

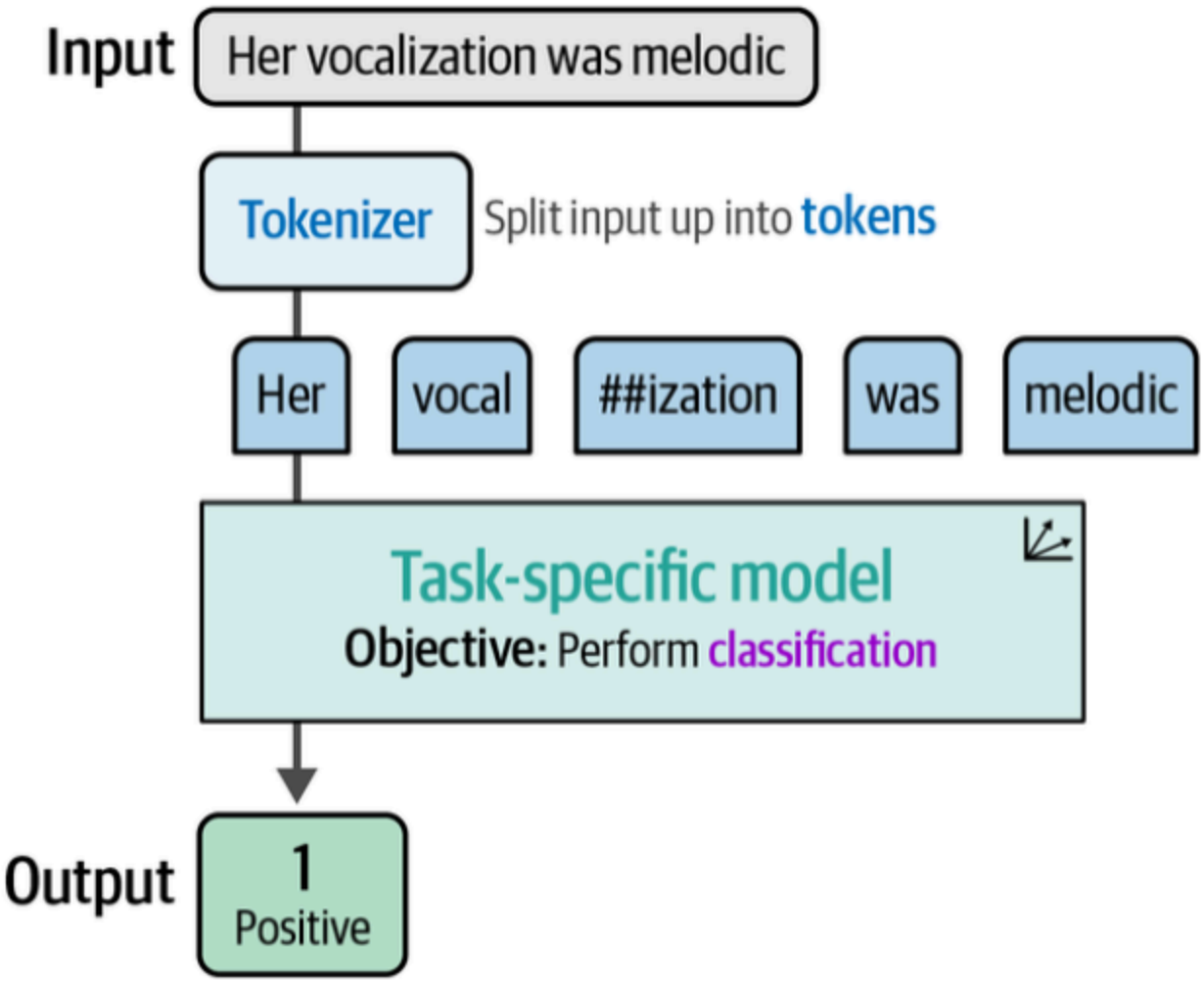

Figure 4-6. An input sentence is first fed to a tokenizer before it can be processed by the task-specific model.

Loading the 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 shown above, the *tokenizer* is loaded with the model. From Chapter 1, we know that a tokenizer is responsible for converting input text to *tokens*.

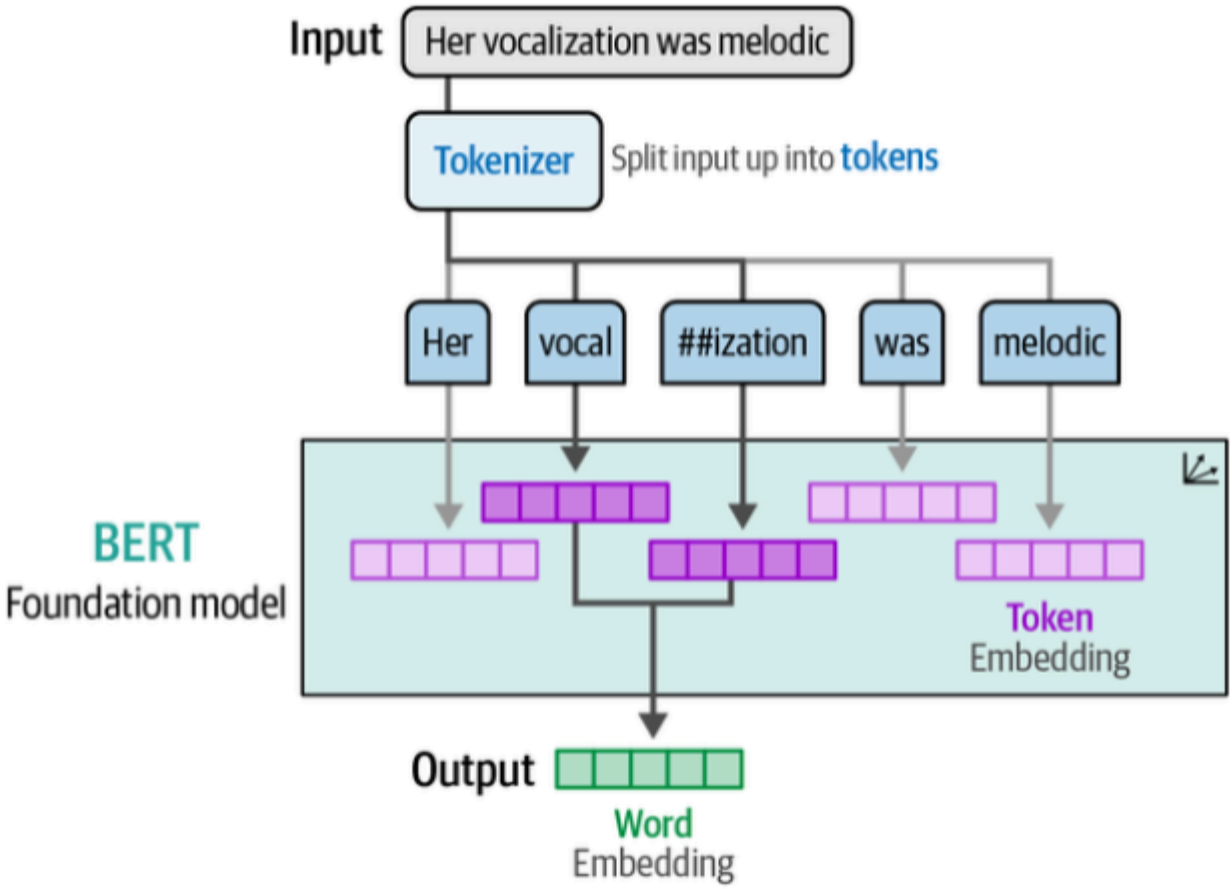

Figure 4-7. By breaking down an unknown word into tokens, word embeddings can still be generated.

Running Inference

```
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)
```

The next step after generating predictions is evaluation:

```python
from sklearn.metrics import classification_report

def evaluate_performance(y_true, y_pred):
    performance = classification_report(
        y_true, y_pred,
        target_names=["Negative Review", "Positive Review"]
    )
    print(performance)

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

Results:

```
                  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
```

F1 score: 0.80

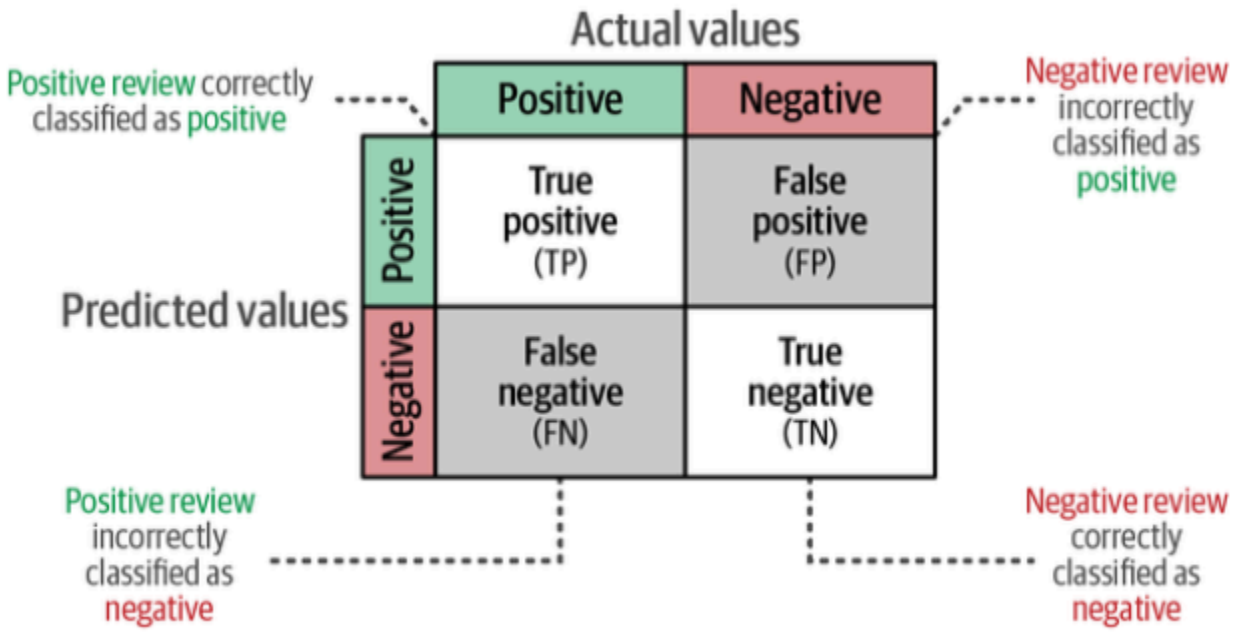

Figure 4-8. The confusion matrix describes four types of predictions we can make.

Key metrics:

- **Precision** - Of items we called positive, how many actually were? (accuracy of positive predictions)
- **Recall** - Of all positive items, how many did we find? (completeness of positive predictions)
- **F1 Score** - Harmonic mean of precision and recall (balanced metric)
- **Accuracy** - Overall correctness (can be misleading with imbalanced classes)
  That's solid for a model trained on tweets (not movie reviews).

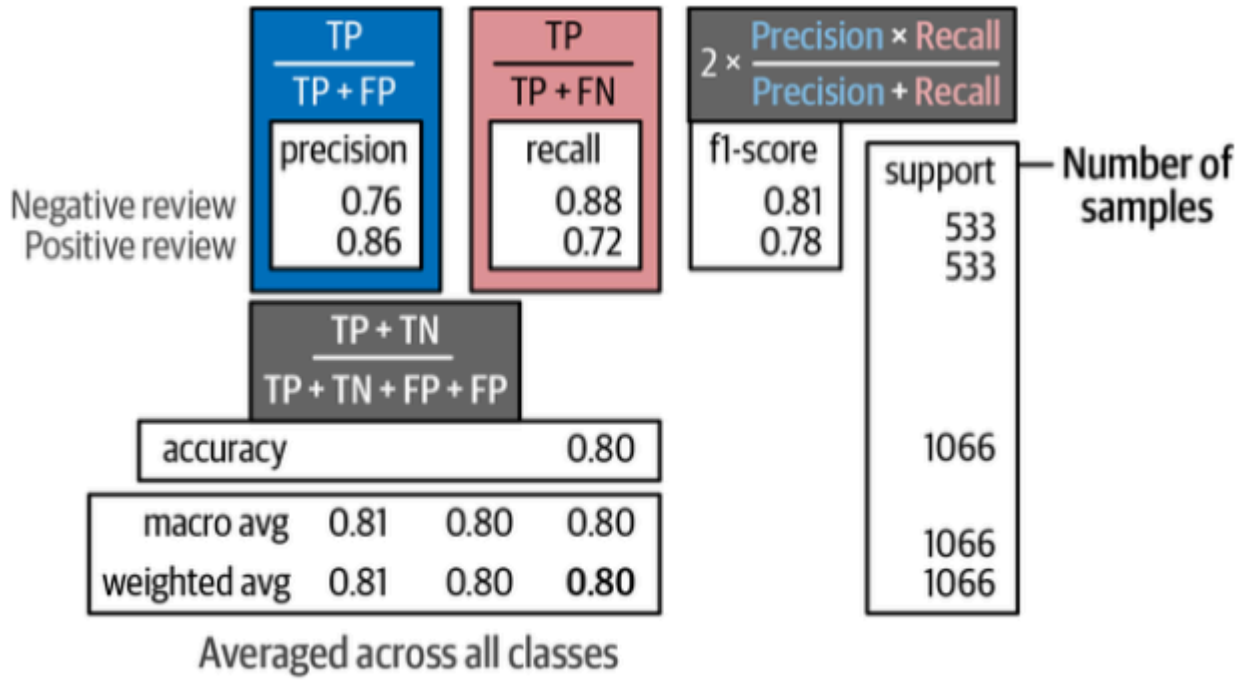

Figure 4-9. The classification report describes several metrics for evaluating a model's performance.

In this case, we will give more emphasis on the weighted average of F1 score throughout the rest of the examples.

Classification Tasks That Leverage Embeddings

While there are available models for different tasks, not every task has a model finetuned for it. You could fine-tune BERT yourself (Chapter 11), but that requires GPUs and time.

This is where general-purpose embedding models come in.

Supervised Classification

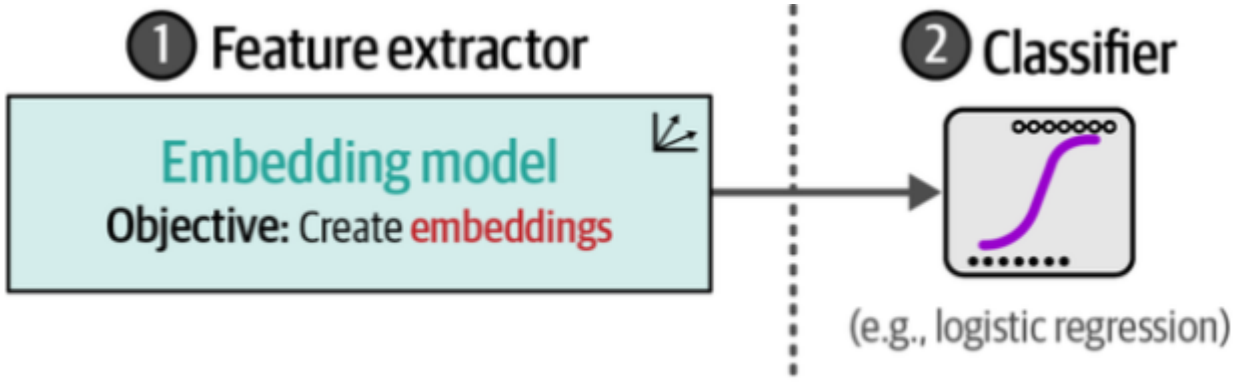

Figure 4-10. The feature extraction step and classification steps are separated.

Stage 1: Embedding model converts text → vectors (frozen, no training needed)

Stage 2: Simple classifier (logistic regression) learns from those vectors (fast, runs on CPU)

> **Representation Model vs Embedding Model in this Context**
>
> **Representation Models** are end-to-end models with a built-in classification head. An example is finetuned BERT like tinybert, where the classification model was used to finetune and update the weights and parameters of the language model.
>
> **Embedding Models** use only the embedding part where the model is *frozen* (parameters and weights are not updated). In this form, there is no classification head attached but is a separate classifier.

Benefits of using a separate classifier:

- **No GPU needed for training** - can be expensive and time-consuming
- **Embeddings work for any task** - because the model is *frozen*, the model is not changed and stays the same as before, therefore, this model can be used for other tasks and is general. Can be attached to other classifiers for other specific tasks.
- **Lightweight classifier trains in seconds** - e.g., *logistic regression*

Generate Embeddings

```
from sentence_transformers import SentenceTransformer

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

train_embeddings = model.encode(data["train"]["text"], show_progress_bar=True)
test_embeddings = model.encode(data["test"]["text"], show_progress_bar=True)

print(train_embeddings.shape)
```

This outputs:

```
(8530, 768)
```

Each document is now a 768-dimensional vector.

In the **second step,** these embeddings are the *input* features to the classifier illustrated below.

The classifier could be logistic regression, random forest, etc.

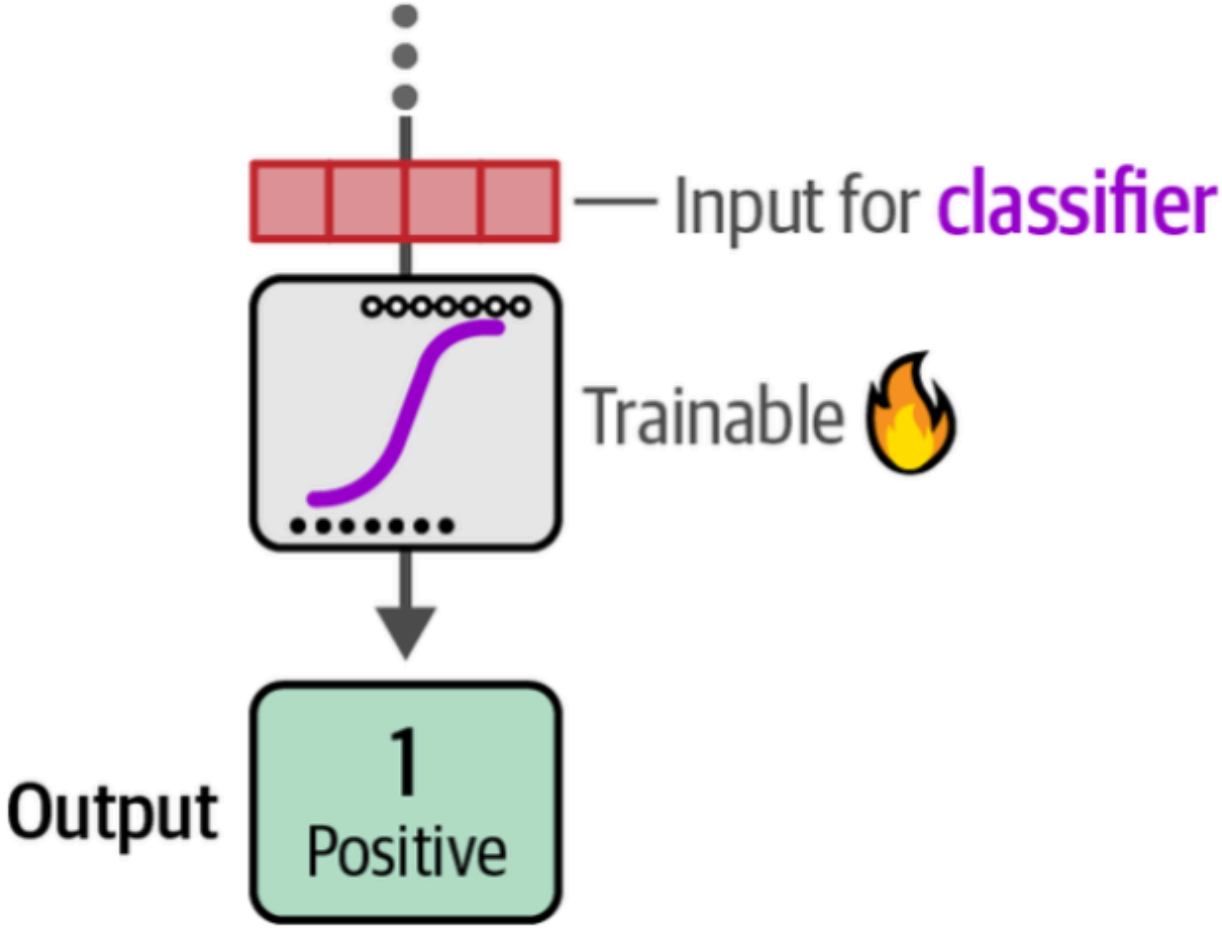

Figure 4-12. Using the embeddings as our features, we train a logistic regression model on our training data.

In this example, we make it simple and use a logistic regression.

Train Classifier

To train the classifier, we need the embeddings from the frozen language model together with the 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"])
```

Let's evaluate the trained model:

```
y_pred = clf.predict(test_embeddings)
evaluate_performance(data["test"]["label"], y_pred)
```

Results:

```
                  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
```

F1 score: 0.85

Better than the task-specific model! And we only trained a simple classifier.

> ⓘ **To run entirely on CPU:**
>
> Use Cohere or OpenAI's embedding API instead of sentence-transformers. Then you don't need a GPU at all.

Zero-Shot Classification

In the previous example, we assume that there are labelled data to use for the classification. However, this is not always the case.

The solution to this is to use **zero-shot classification**: Describe your labels, embed them, find which label is most similar to each document.

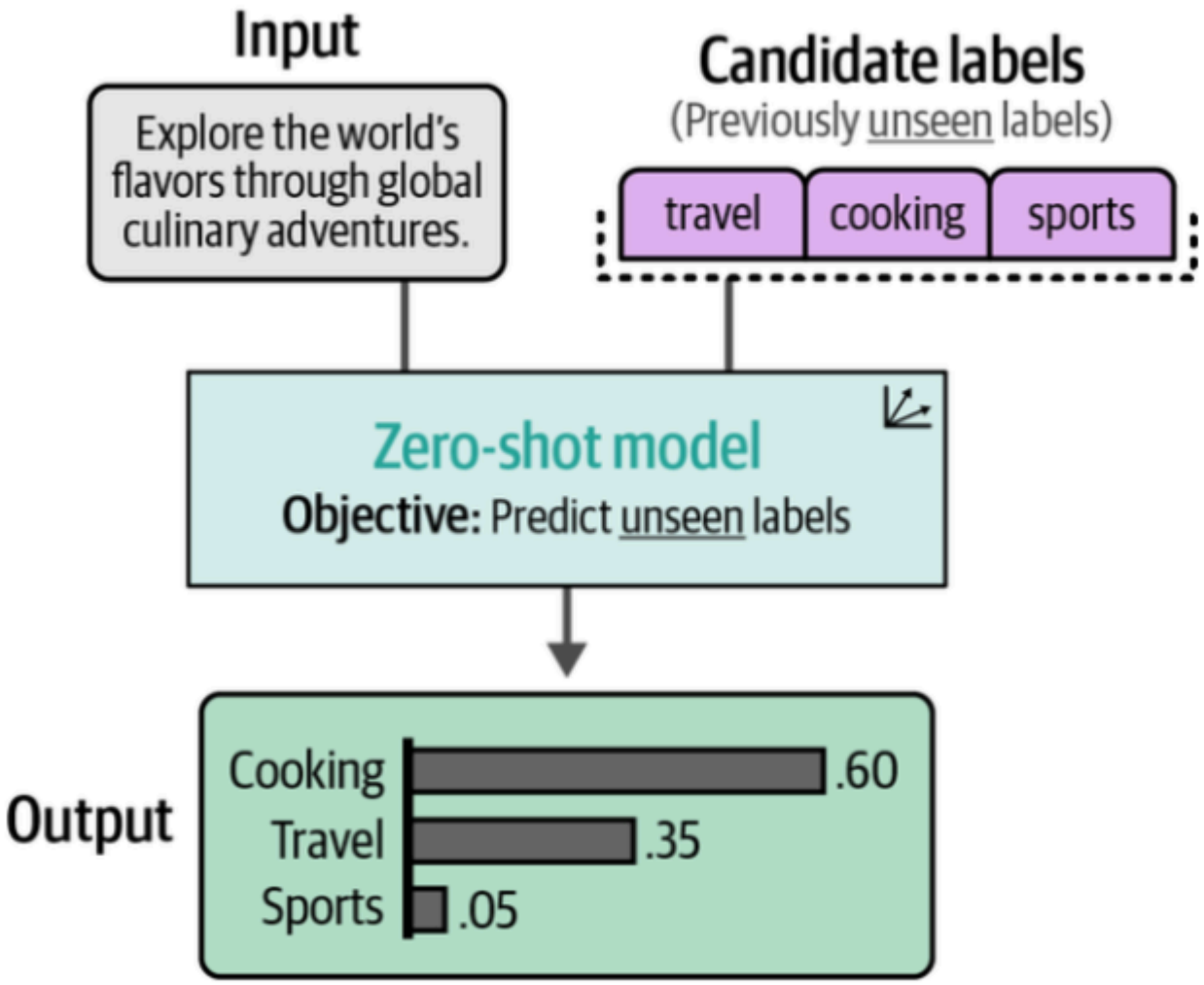

Figure 4-13. In zero-shot classification, we have no labeled data, only the labels themselves. The zero-shot model decides how the input is related to the candidate labels.

How It Works

Instead of:

- Training examples with labels

You just have:

- Label names (0 = negative, 1 = positive)

The trick: Turn label names into descriptions:

- 0 → "A negative review"
- 1 → "A positive review"

Then embed both documents and label descriptions. Use cosine similarity to match.

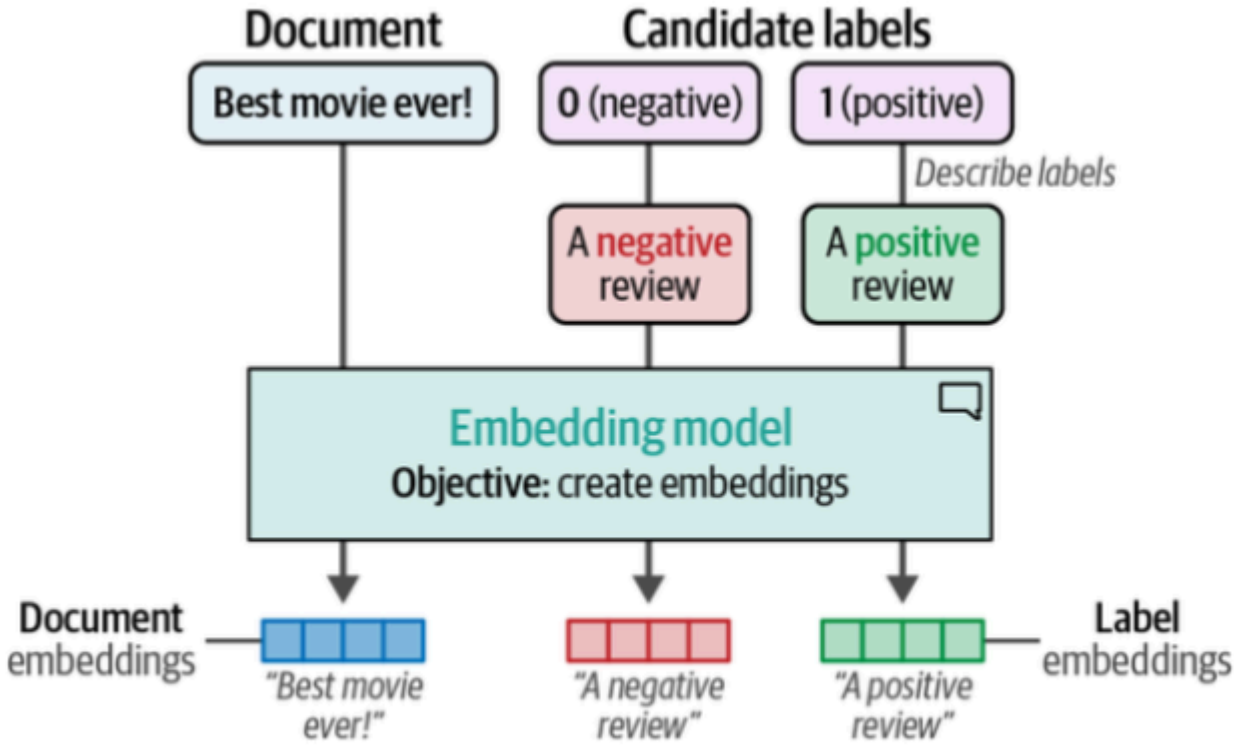

Figure 4-14. To embed the labels, we first need to give them a description, such as "a negative movie review." This can then be embedded through sentence-transformers.

Implementation

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

```
# Embed the labels
label_embeddings = model.encode([
    "A negative review",
    "A positive review"
])
```

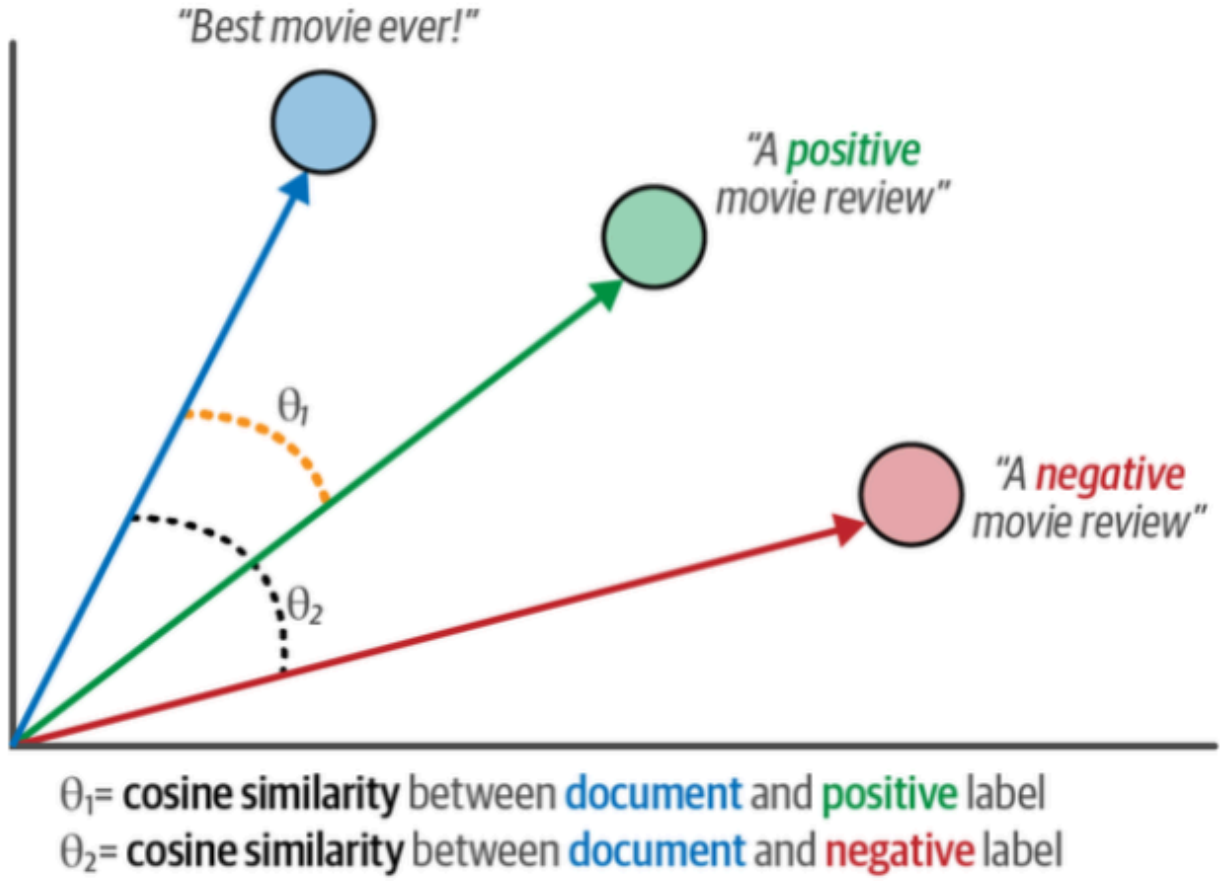

Figure 4-15. The cosine similarity is the angle between two vectors or embeddings. In this example, we calculate the similarity between a document and the two possible labels, positive and negative.

```
# Find most similar label for each document
from sklearn.metrics.pairwise import cosine_similarity

sim_matrix = cosine_similarity(test_embeddings, label_embeddings)
y_pred = np.argmax(sim_matrix, axis=1)
```

As you can see above, we 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.

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

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

Results:

```
                  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
```

F1 score: 0.78

Tip: Make your label descriptions more specific:

- Bad: "A negative review"
- Better: "A very negative movie review"

More specific descriptions return better results.

Text Classification with Generative Models

Generative models output text, not classes. You need to prompt them.

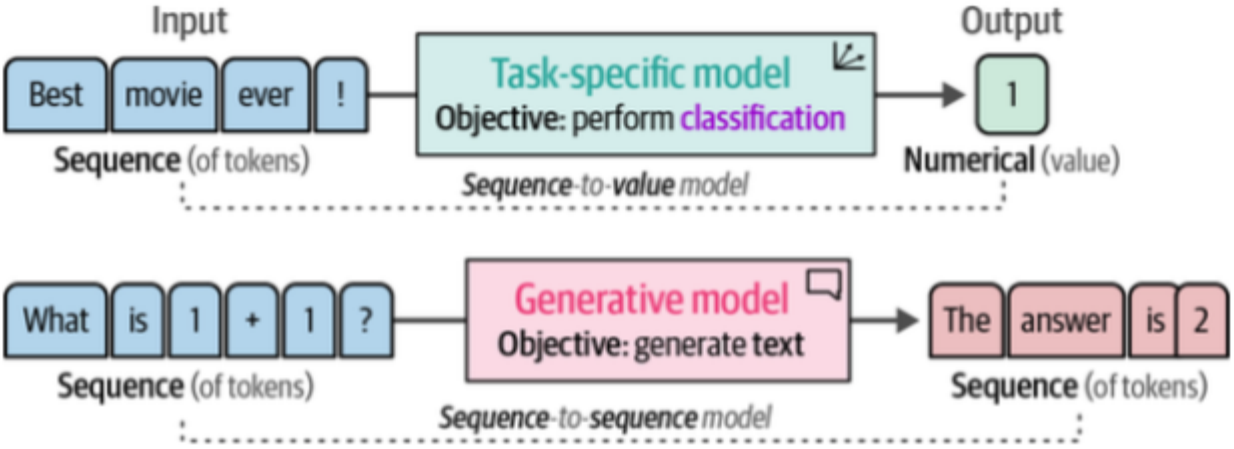

Figure 4-17. A task-specific model generates numerical values from sequences of tokens while a generative model generates sequences of tokens from sequences of tokens.

Key differences:

- **Task-specific:** Input → Class number
- **Generative:** Input + Prompt → Generated text → Parse into class

A generative model wouldn't know what to do with a review, for example. It needs an instruction or *prompt*

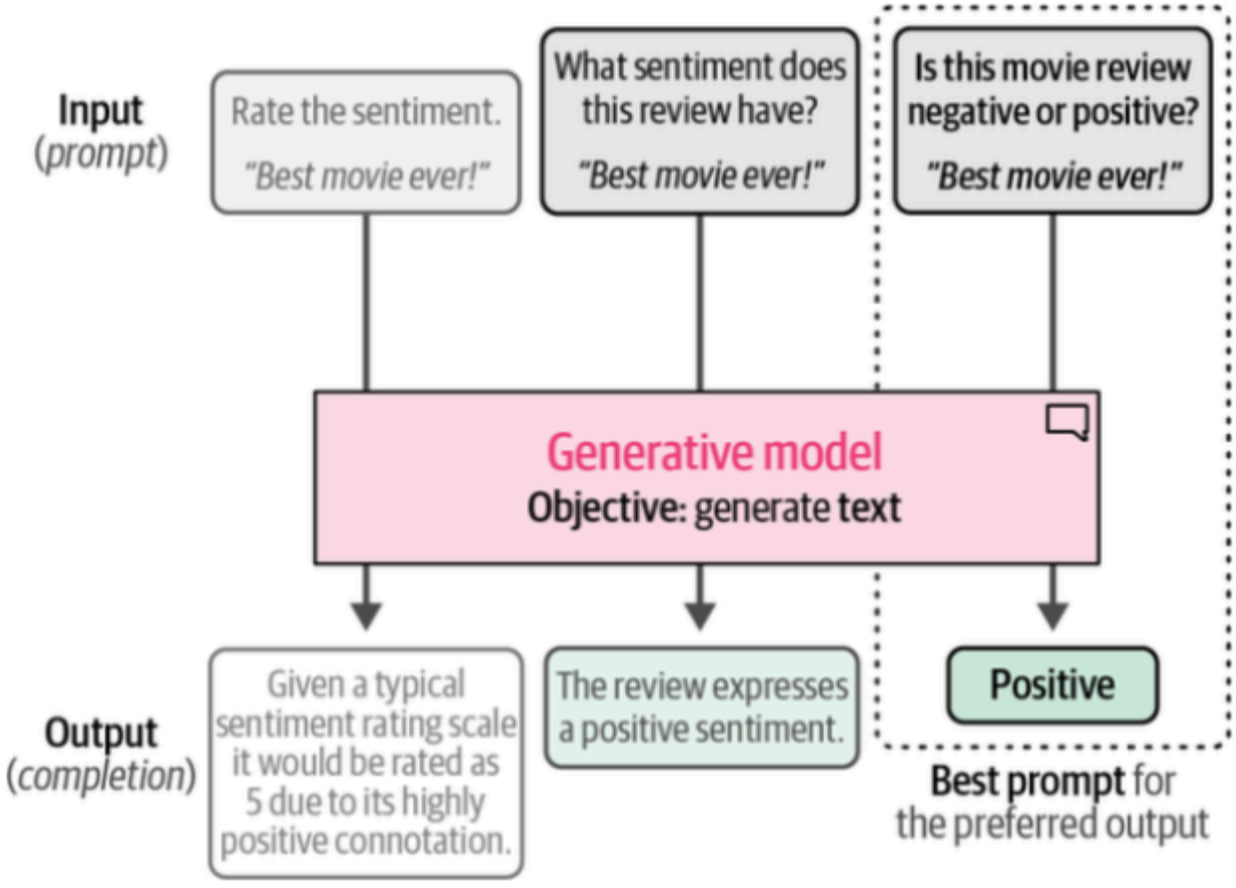

Figure 4-18. Prompt engineering allows prompts to be updated to improve the output generated by the model.

The quality of your prompt matters a lot.

Using T5 (Text-to-Text Transfer Transformer)

Throughout the chapters we explored and the coming chapters, we mostly used encoder-only (representation) like BERT and decoder-only (generative) models like ChatGPT.

T5 uses the original Transformer architecture: encoder + decoder. This falls in the category of generative models.

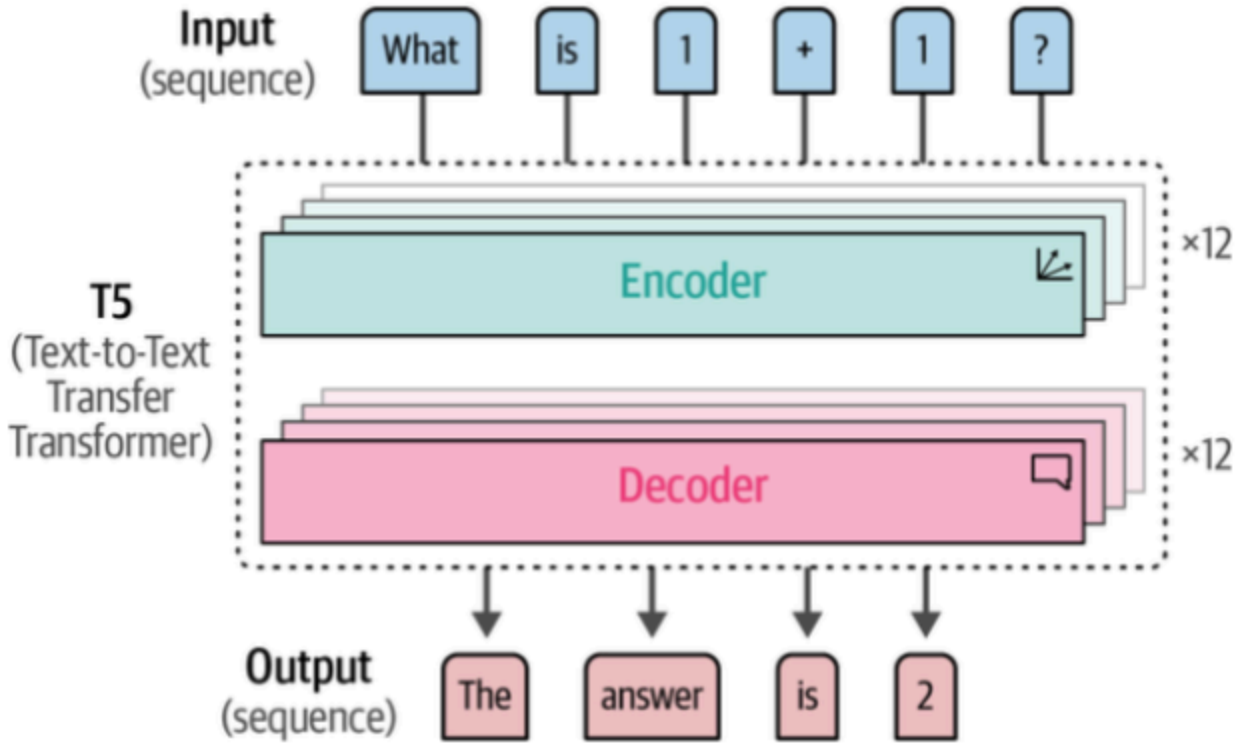

Figure 4-19. The T5 architecture is similar to the original Transformer model, a decoder-encoder architecture.

How T5 Was Trained

Step 1: Pretraining

Masked language modelling with token spans (not just single tokens).

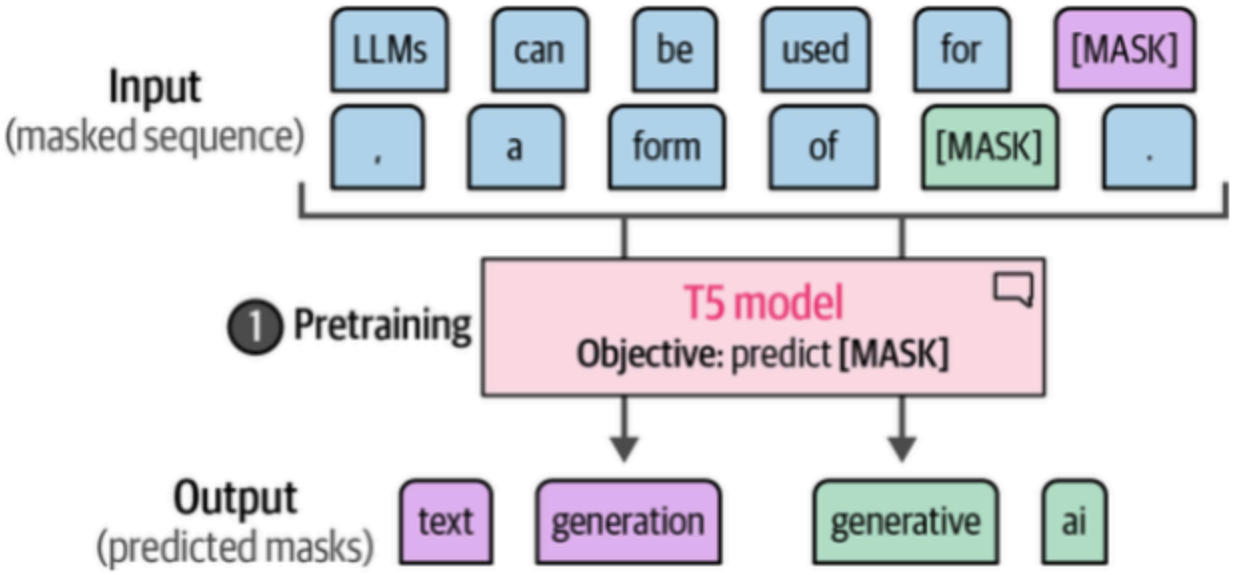

Figure 4-20. In the first step of training, namely pretraining, the T5 model needs to predict masks that could contain multiple tokens.

Step 2: Multi-task Fine-tuning

Trained on many tasks simultaneously by converting each to text-to-text format.

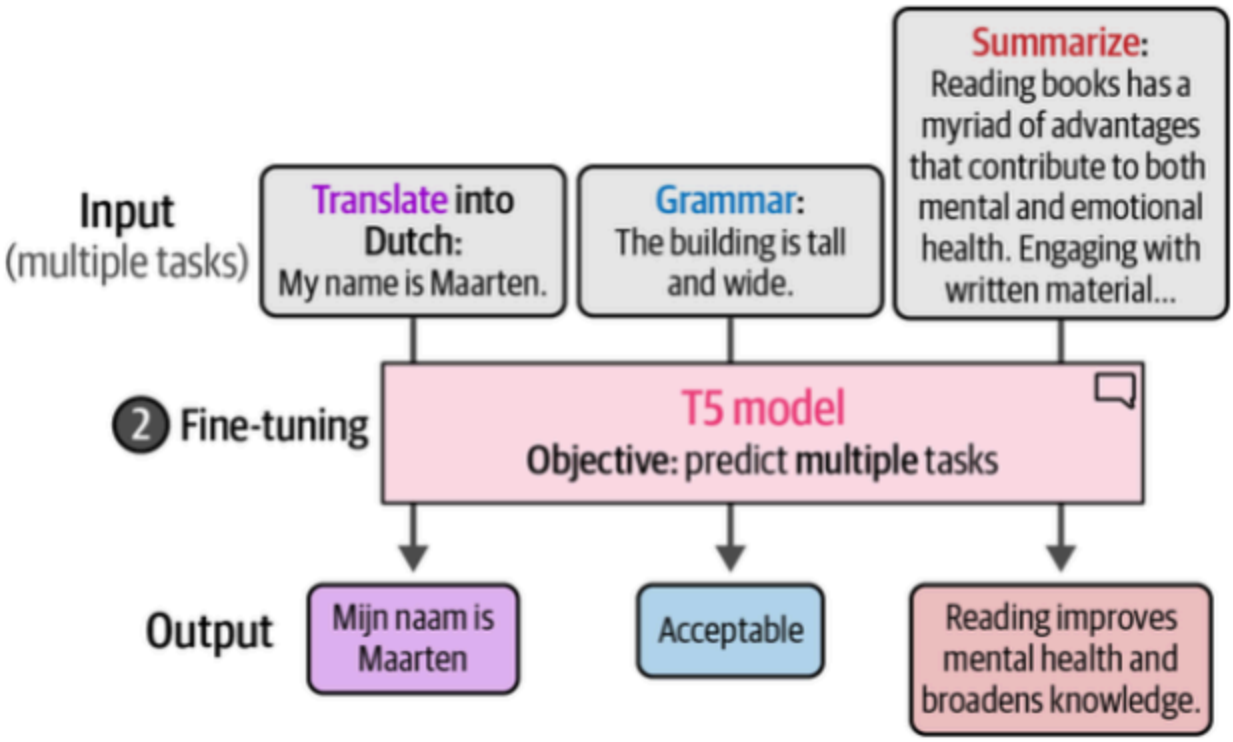

Figure 4-21. By converting specific tasks to textual instructions, the T5 model can be trained on a variety of tasks during fine-tuning.

Flan-T5 variant: Fine-tuned on 1000+ tasks with better instruction following.

Using Flan-T5

```
pipe = pipeline(
    "text2text-generation",
    model="google/flan-t5-small",  # 80M params
    device="cuda:0"
)

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

Results:

```
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
    })
})
```

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)

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

Results:

```
                  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
```

F1 score: 0.84

Solid performance for a small (80M param) model.

Model sizes:

- flan-t5-small (80M)
- flan-t5-base (250M)
- flan-t5-large (780M)
- flan-t5-xl (3B)
- flan-t5-xxl (11B)

ChatGPT for Classification

Architecture and Training

ChatGPT uses a decoder-only architecture based on the GPT family of models. While OpenAI hasn't publicly shared the complete architecture of GPT-3.5, we can assume from its name that it is a decoder-only model.

The training process for ChatGPT involves two distinct phases (instruction tuning & preference tuning) beyond the initial pretraining on large text corpora. These phases are what differentiate ChatGPT from base GPT models and enable it to follow instructions effectively.

Instruction Tuning

The first phase is instruction tuning, also called supervised fine-tuning. Human annotators manually create datasets consisting of prompts paired with ideal responses. The base language model is then fine-tuned on these prompt-completion pairs.

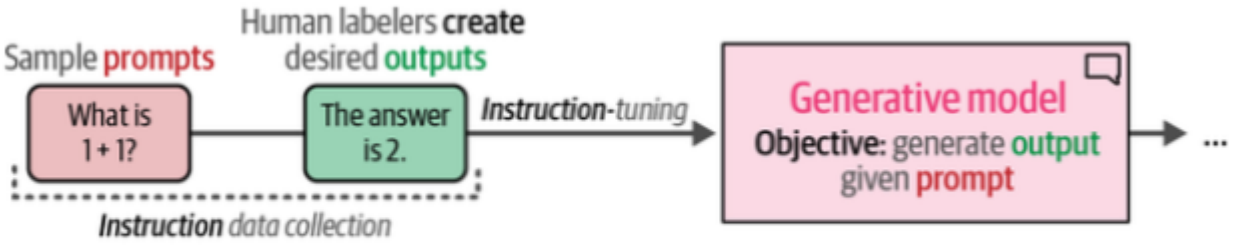

Figure 4-22. Manually labeled data consisting of an instruction (prompt) and output was used to perform fine-tuning (instruction-tuning).

This process teaches the model to understand and follow instructions, but it only captures what annotators think is a good response at the time of labeling and doesn't capture the relative quality between multiple plausible responses.

Preference Tuning (RLHF)

The second phase addresses this limitation through preference tuning, implemented via ***Reinforcement Learning from Human Feedback (RLHF)***. The instruction-tuned model generates multiple candidate responses to the same prompt. Human annotators then rank these outputs from best to worst rather than writing ideal responses from scratch.

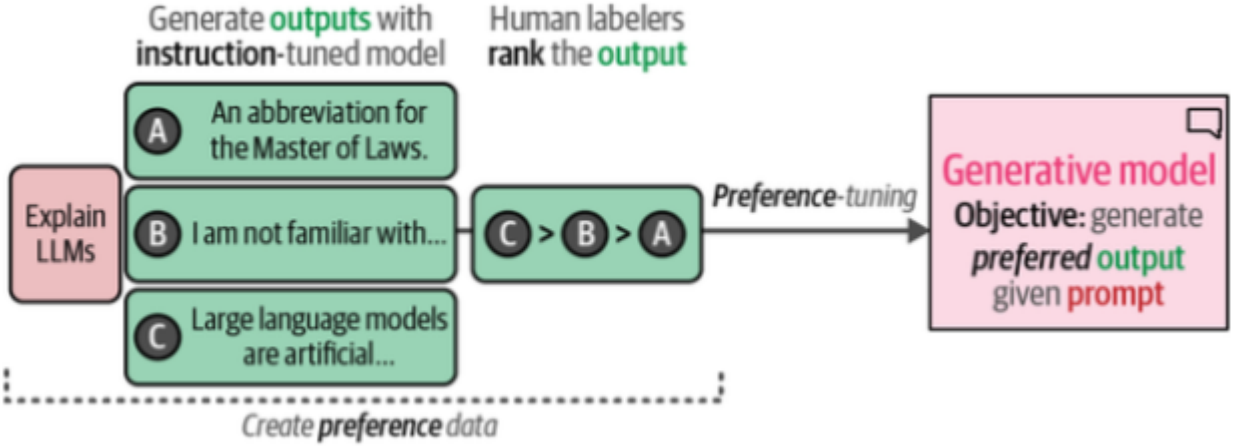

Figure 4-23. Manually ranked preference data was used to generate the final model, ChatGPT.

For instance, given the prompt "Explain LLMs", the model might generate:

- Output A: "An abbreviation for the Master of Laws"
- Output B: "I am not familiar with..."
- Output C: "Large language models are artificial intelligence systems..."

Annotators would rank these as C > B > A. This ranking information trains a reward model that learns human preferences, which then guides further training of the language model through reinforcement learning.

The key advantage of preference tuning over instruction tuning alone is nuance. By showing the model the difference between acceptable and excellent responses, it learns to generate outputs

that better align with human expectations and preferences. This is why ChatGPT tends to produce more helpful, harmless, and honest responses compared to base GPT models.

Implementation with OpenAI API

Unlike open source models where you load weights locally, accessing ChatGPT requires using OpenAI's REST API. This means your data is sent to OpenAI's servers for processing.

Setup

First, create an account at platform.openai.com and generate an API key from the API keys section. The free tier includes initial credits that should be sufficient for experimentation.

```
import openai

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

Creating a Generation Function

We'll create a wrapper function that handles the API interaction:

```
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
```

The messages list follows OpenAI's chat format:

- `system`: Sets the assistant's behavior and context
- `user`: Contains the actual prompt with our classification request

Setting `temperature=0` makes the model deterministic, always selecting the highest probability token. This is important for classification where we want consistent outputs.

Prompt Design

The prompt is critical for generative models. Unlike task-specific models where the task is baked into the weights, generative models need explicit instructions:

```
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.
"""

# Test on a single example
document = "unpretentious, charming, quirky, original"
result = chatgpt_generation(prompt, document)
print(result)  # Should output: 1
```

The prompt structure matters significantly:

- Clear task description ("Predict whether...")
- Explicit output format ("return 1 or 0")
- Constraint on output space ("Do not give any other answers")

Poorly designed prompts will lead to inconsistent outputs that are harder to parse programmatically.

Batch Inference

To evaluate on the full test set:

```
from tqdm import tqdm

# Run inference on all test examples
predictions = []
for doc in tqdm(data["test"]["text"]):
```

```
    result = chatgpt_generation(prompt, doc)
    predictions.append(result)
```

Since the model outputs text rather than class indices, we need to convert the string outputs to integers:

```
# Convert string predictions to integers
y_pred = [int(pred) for pred in predictions]

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

Results:

```
                  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
```

An F1 score of 0.91 is the highest we've achieved across all approaches in this chapter.

Practical Considerations

Cost Tracking

External APIs charge per token processed. For our test set of 1,066 reviews using gpt-3.5-turbo-0125, the cost is approximately $0.03 at current pricing. While this seems negligible, costs scale linearly with dataset size. A production system processing millions of documents would incur substantial expenses.

Always test on small samples first and monitor your usage through the OpenAI dashboard. Set billing limits to prevent unexpected charges.

Rate Limiting

APIs have rate limits to prevent abuse and ensure fair resource allocation. OpenAI's limits vary by tier but typically restrict requests per minute and tokens per day.

When you hit a rate limit, the API returns a 429 status code. The proper handling approach is exponential backoff:

```
import time
from openai import RateLimitError

def chatgpt_generation_with_retry(prompt, document, max_retries=5):
    """Generate with exponential backoff on rate limits."""
    for attempt in range(max_retries):
        try:
            return chatgpt_generation(prompt, document)
        except RateLimitError:
            if attempt == max_retries - 1:
                raise
            wait_time = 2 ** attempt  # 1, 2, 4, 8, 16 seconds
            time.sleep(wait_time)
```

This waits increasingly longer between retries, giving the rate limit time to reset.

Data Privacy

Using external APIs means your data leaves your infrastructure. This raises important considerations:

- Training data usage: OpenAI's API terms specify whether your data can be used for model training. As of the latest terms, data sent through the API is not used for training by default, but verify current policies.
- Sensitive information: Never send personally identifiable information, proprietary data, or confidential information through external APIs without proper data agreements.
- Geographic restrictions: Some jurisdictions have data residency requirements that prohibit sending data to foreign servers.

For sensitive applications, consider using local models (T5, Flan-T5, Llama) even if they perform slightly worse.

When to Use ChatGPT vs Alternatives

ChatGPT achieves the highest performance but comes with tradeoffs:

Use ChatGPT when:

- You need the absolute best performance
- You have budget for API costs
- Data privacy is not a concern

- You want to prototype quickly without infrastructure setup

Use alternatives when:

- Working with sensitive data
- Need to process large volumes cost-effectively
- Require full control over the model and infrastructure
- Want reproducible results without dependency on external services

Note that we cannot verify what data ChatGPT was trained on. The high F1 score of 0.91 could partially reflect data contamination if Rotten Tomatoes reviews were in the training set. For rigorous evaluation, use held-out test sets that definitely postdate the model's training cutoff, or use standardized benchmarks designed to avoid contamination.

Comparison with Other Approaches

Across the classification methods explored in this chapter:

Method	F1 Score	Key Advantage	Main Limitation
Task-specific BERT	0.80	Fast inference	Single task only
Embeddings + Classifier	0.85	Flexible, CPU-trainable	Requires labeled data
Embeddings Zero-shot	0.78	No labeled data needed	Lower accuracy
Flan-T5	0.84	Open source, flexible	GPU needed for inference
ChatGPT	0.91	Best performance	Costs money, privacy concerns

For most production applications, the embeddings + classifier approach offers the best balance of performance, cost, and flexibility. ChatGPT is good for prototyping and cases where the performance difference justifies the cost and privacy trade-offs.