

5

Optimizing LLMs with Customized Fine-Tuning

Introduction

So far, we've almost exclusively used LLMs, both open- and closed-source, just as they are off the shelf. We were relying on the power of the Transformer's attention mechanisms and their speed of computation to perform some pretty complex problems with relative ease. As you can probably guess, that isn't always enough.

In [Chapter 2](#), I showcased the power of updating LLMs with custom data to increase accuracy in information retrieval. But that's just the tip of the iceberg. In this chapter, we will dive deeper into the world of fine-tuning LLMs to unlock their full potential. Fine-tuning updates off-the-shelf models—specifically, the values of their parameters—and empowers them to achieve higher-quality results on specific tasks. It can lead to cost savings, shorter prompts, and often lower-latency requests. While GPT-like LLMs' pre-training on extensive text data enables impressive few-shot learning capabilities, fine-tuning takes matters a step further by refining the model on a multitude of examples, resulting in superior performance across various tasks.

Running inference with fine-tuned models can be extremely cost-effective in the long run, particularly when working with smaller models. For instance, a fine-tuned Babbage model (a 1.3 billion parameter model from the GPT-3 family) will vastly outperform ChatGPT in terms of cost over a long period of time, as shown in [Figure 5.1](#). The graph in [Figure 5.1](#) shows five options for using LLMs to solve a classification task—that is, inputting a text phrase and outputting a single token representing a class label:

- GPT_3_5_just_ask (50 input tokens, 1 output token): Asks non-fine-tuned GPT-3.5 to solve a classification task and output a single token representing a class
- GPT_3_5_few_shot_prompt (150 input tokens, 1 output token): Includes a few-shot prompt (hence the increase in input tokens) with still only 1 output token, the class label

- GPT_3_5_few_shot_CoT (150 input tokens, 100 output tokens): Includes the same few-shot prompt plus a chain-of-thought output, resulting in more output tokens—which cost more
- GPT_3_5_fine_tuned (50 input tokens, 1 output token): A fine-tuned GPT-3.5 model that won't use few-shot learning or chain-of-thought prompting
- fine_tuned_babbage (50 input tokens, 1 output token): A fine-tuned Babbage model (a smaller autoregressive model in use by OpenAI) that won't use few-shot learning or chain-of-thought prompting

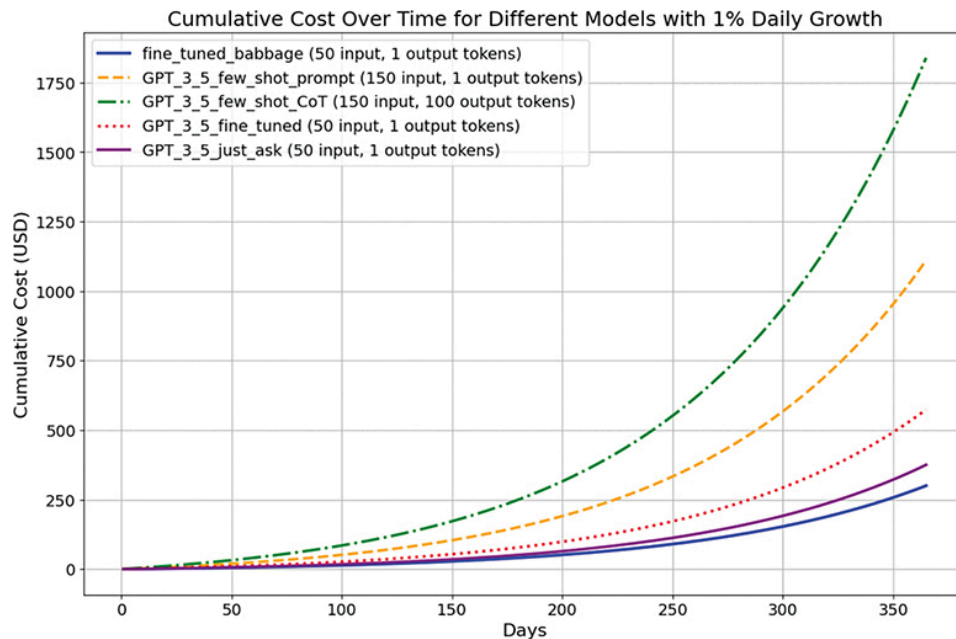


Figure 5.1 Assuming a steady 1% growth daily in number of classifications a day and a relatively liberal prompt ratio (approximately 150 tokens [for few-shot examples, instructions, and other items] for Babbage or ChatGPT), the cost of a fine-tuned Babbage model tends to win the day over all cost-wise. Note that this does not consider the cost of fine-tuning a model, which we will explore later in this chapter.

The data in the figure covers pricing for each model as of May 2024 and represents a daily increase in volume by 1% from the previous day. Notice the two fine-tuned models toward the bottom of the graph: They show that, in the long term, fine-tuning LLMs often offers a cost savings.

My goal in this chapter is to guide you through the fine-tuning process, beginning with the preparation of training data, strategies for training a new or existing fine-tuned model, and a discussion of how to incorporate

your fine-tuned model into real-world applications. This is a big topic, so we will have to assume some big pieces are being handled behind the scenes, such as data labeling. Labeling data can be a huge expense in many cases of complex and specific tasks, but for now we'll assume we can rely on the labels in our data for the most part. For more information on how to handle cases like these, feel free to check out some of my other content on feature engineering and label cleaning.

By understanding the nuances of fine-tuning and mastering its techniques, you will be well equipped to harness the power of LLMs and create tailored solutions for your specific needs.

Transfer Learning and Fine-Tuning: A Primer

Fine-tuning hinges on the idea of transfer learning. **Transfer learning** is a technique that leverages pre-trained models to build upon existing knowledge for new tasks or domains. In the case of LLMs, this involves utilizing the pre-training to transfer general language understanding, including grammar and general knowledge, to particular domain-specific tasks. However, the pre-training may not be sufficient to understand the nuances of certain closed or specialized topics, such as a company's legal structure or guidelines.

Fine-tuning is a specific form of transfer learning that adjusts the parameters of a pre-trained model to better suit a "downstream" target task. Through fine-tuning, LLMs can learn from custom examples and become more effective at generating relevant and accurate responses.

The Fine-Tuning Process Explained

Fine-tuning a deep learning model involves updating the model's parameters to improve its performance on a specific task or dataset.

- **Training set:** A collection of labeled examples used to train the model. The model learns to recognize patterns and relationships in the data by adjusting its parameters based on the training examples.
- **Validation set:** A separate collection of labeled examples used to evaluate the model's performance during training.
- **Test set:** A third collection of labeled examples that is separate from both the training and validation sets. It is used to evaluate the final performance of the model after the training and fine-tuning processes are complete. The test set provides a final, unbiased estimate of the model's ability to generalize to new, unseen data.
- **Loss function:** A function that quantifies the difference between the model's predictions and the actual target values. It serves as a metric of error to evaluate the model's performance and guide the optimization.

tion process. During training, the goal is to minimize the loss function to achieve better predictions.

The process of fine-tuning can be broken down into a few steps:

- 1. Collecting labeled data:** The first step in fine-tuning is to gather our training, validation, and testing datasets of labeled examples relevant to the target task or domain. Labeled data serves as a guide for the model to learn the task-specific patterns and relationships. For example, if the goal is to fine-tune a model for sentiment classification (our first example), the dataset should contain text examples along with their respective sentiment labels, such as positive, negative, or neutral.
- 2. Hyperparameter selection:** Fine-tuning involves adjusting hyperparameters that influence the learning process—for example, the learning rate, batch size, and number of epochs. The learning rate determines the step size of the model's weight updates, while the batch size refers to the number of training examples used in a single update. The number of epochs denotes how many times the model will iterate over the entire training dataset. Properly setting these hyperparameters can significantly impact the model's performance and help prevent issues such as overfitting (i.e., when a model learns the noise in the training data more than the signals) and underfitting (i.e., when a model fails to capture the underlying structure of the data).
- 3. Model adaptation:** Once the labeled data and hyperparameters are set, the model may have to be adapted to the target task. This involves modifying the model's architecture, such as adding custom layers or changing the output structure, to better suit the target task. For example, BERT's architecture cannot perform sequence classification as is, but it can be modified very slightly to carry out this task. In our case study, we will not need to deal with that modification because OpenAI will handle it for us. We will, however, have to deal with this issue in a later chapter.
- 4. Evaluation and iteration:** After the fine-tuning process is complete, we have to evaluate the model's performance on a separate holdout validation set to ensure that it generalizes well to unseen data. Performance metrics such as accuracy, F1 score, or mean absolute error (MAE) can be used for this purpose, depending on the task. If the performance is not satisfactory, adjustments to the hyperparameters or dataset may be necessary, followed by retraining the model.
- 5. Model implementation and further training:** Once the model is fine-tuned and we are happy with its performance, we need to integrate it with existing infrastructures in a way that can handle any errors and collect feedback from users. Doing so will enable us to add to our total dataset and rerun the process in the future.

This process is outlined in **Figure 5.2**. Note that the process may require several iterations and careful consideration of hyperparameters, data quality, and model architecture to achieve the desired results.

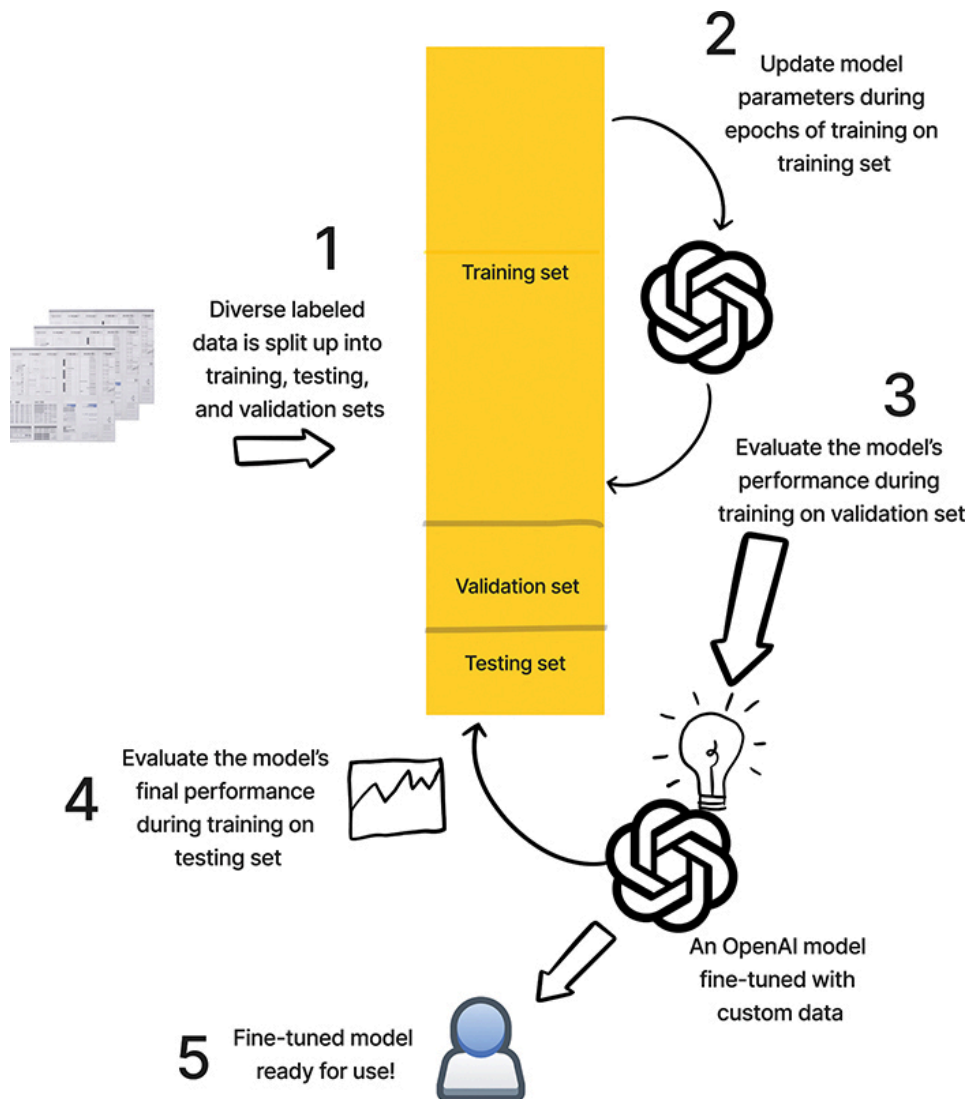


Figure 5.2 The fine-tuning process visualized. A dataset is broken up into training, validation, and testing tests. The training set is used to update the model's weights and evaluate the model, whereas the validation set is used to evaluate the model during training. The final model is then tested against the testing set and evaluated against a set of criteria. If the model passes all of these tests, it is used in production and monitored for further iterations.

Closed-Source Pre-trained Models as a Foundation

Pre-trained LLMs play a vital role in transfer learning and fine-tuning, providing a foundation of general language understanding and knowledge. This foundation allows for efficient adaptation of the models to specific tasks and domains, reducing the need for extensive training resources and data.

This chapter focuses on fine-tuning LLMs using OpenAI's infrastructure, which has been specifically designed to facilitate this process. OpenAI has developed tools and resources to make it easier for researchers and developers to fine-tune smaller models, such as Babbage, for their specific needs. The infrastructure offers a streamlined approach to fine-tuning, allowing users to efficiently adapt pre-trained models to a wide variety of tasks and domains.

Benefits of Using OpenAI's Fine-Tuning Infrastructure

Leveraging OpenAI's infrastructure for fine-tuning offers several advantages:

- Access to powerful pre-trained models, such as GPT-3.5 or GPT-4, which have been trained on extensive and diverse datasets
- A relatively user-friendly interface that simplifies the fine-tuning process for people with varying levels of expertise
- A range of tools and resources that help users optimize their fine-tuning process, such as guidelines for selecting hyperparameters, tips on preparing custom examples, and advice on model evaluation

This streamlined process saves time and resources while ensuring the development of high-quality models capable of generating accurate and relevant responses in a wide array of applications. We will dive deep into open-source fine-tuning and the benefits and drawbacks it offers in [Chapters 6 through 9](#).

A Look at the OpenAI Fine-Tuning API

The OpenAI API offers developers access to one of the most advanced LLMs available. This API provides a range of fine-tuning capabilities, allowing users to adapt the model to specific tasks, languages, and domains. This section discusses the key features of the OpenAI fine-tuning API, the supported methods, and best practices for successfully fine-tuning models.


The OpenAI Fine-Tuning API

The OpenAI fine-tuning API is a way to make the already powerful GPT family of models even more powerful by letting them learn from our own customized data. It's truly a one-stop shop for tailoring the model to your specific tasks, languages, or domains. This section aims to make the OpenAI fine-tuning API even more accessible through specific examples and a case study, highlighting the tools and techniques that make it such an invaluable resource.

Case Study: App Review Sentiment Classification

Let's introduce our first case study. We will be working with the `app_reviews` dataset (previewed in [Figure 5.3](#)). This dataset is a collection of app reviews of 395 different Android apps spanning multiple review types and app versions. Each review in the dataset is accompanied by a rating on a scale of 1 to 5 stars, with 1 star being the lowest rating (denoted as 0) and 5 stars being the highest (denoted as 4). Our goal in this case study is to fine-tune a pre-trained model from OpenAI to perform sentiment classification on these reviews, enabling it to predict the number of stars given in a review. Taking a page out of my own book (albeit one from just a few pages ago), let's start looking at the data.

	review	star
0	Nice 😊	4
1	Google play service Just one ward its amazing ...	4
2	Mr Perfect	0
3	Does not work with Tmobile S4 If you try to in...	0
4	Ok	2
5	Say App Ka nam to the other than a few months	4
6	Owk	4
7	Coc	4
8	Not working bad	0
9	After downloading this app my phone slowed do...	0


**The Android
App Review**



**Our class to predict
(the response)**

Figure 5.3 A snippet of the `app_reviews` dataset shows our input context (review titles and bodies) and our response (the thing we are trying to predict—the number of stars given out by the reviewer).

We will care about three columns in the dataset for this round of fine-tuning:

- `review_title`: The text title of the review

- `review_body` : The text body of the review
- `stars` : An integer between 1 and 5 indicating the number of stars

Our goal will be to use the context of the title and body of the review and predict the rating that was given.

Guidelines and Best Practices for Data

In general, there are a few items to consider when selecting data for fine-tuning:

- **Data quality:** Ensure that the data used for fine-tuning is of high quality, is free from noise, and accurately represents the target domain or task. This will enable the model to learn effectively from the training examples.
- **Data diversity:** Make sure the dataset is diverse, covering a broad range of scenarios to help the model generalize well across different situations.
- **Data balancing:** Maintaining a balanced distribution of examples across different tasks and domains helps prevent overfitting and biases in the model's performance. This can be achieved with unbalanced datasets by undersampling majority classes, oversampling minority classes, or adding synthetic data. Our sentiment is perfectly balanced due to the fact that this dataset was curated—but check out an even harder example in our code base, where we attempt to classify the very unbalanced category classification task.
- **Data quantity:** Determine the total amount of data needed to fine-tune the model. Generally, larger language models like LLMs require more extensive data to capture and learn various patterns effectively, but smaller datasets if the LLM was pre-trained on similar enough data. The exact quantity of data needed can vary based on the complexity of the task at hand. Any dataset should be not only extensive, but also diverse and representative of the problem space to avoid potential biases and ensure robust performance across a wide range of inputs. While using a large quantity of training data can help to improve model performance, it also increases the computational resources required for model training and fine-tuning. This trade-off needs to be considered in the context of the specific project requirements and resources.

Preparing Custom Examples with the OpenAI CLI

Before diving into fine-tuning, we need to prepare the data by cleaning and formatting it according to the API's requirements. This includes the following steps:

- **Removing duplicates:** To ensure the highest data quality, start by removing any duplicate reviews from the dataset. This will prevent the model from overfitting to certain examples and improve its ability to generalize to new data.
- **Splitting the data:** Divide the dataset into training, validation, and test sets, maintaining a random distribution of examples across each set. If necessary, consider using stratified sampling to ensure that each set contains a representative proportion of the different sentiment labels, thereby preserving the overall distribution of the dataset.
- **Shuffling the training data:** Shuffling training data before fine-tuning helps to avoid biases in the learning process by ensuring that the model encounters examples in a random order, reducing the risk of learning unintended patterns based on the order of the examples. It also improves model generalization by exposing the model to a more diverse range of instances at each stage of training, which also helps to prevent overfitting, as the model is less likely to memorize the training examples and instead will focus on learning the underlying patterns. Ideally, the data will be shuffled before every single epoch to reduce the chance of the model overfitting on the data as much as possible.
- **Creating the OpenAI JSONL format:** OpenAI's API expects the training data to be in JSONL (newline-delimited JSON) format. For each example in the training and validation sets, create a JSON object with two fields: "prompt" (the input) and "completion" (the target class). The "prompt" field should contain the review text, and the "completion" field should store the corresponding sentiment label (stars). Save these JSON objects as newline-delimited records in separate files for the training and validation sets.

For completion tokens in our dataset, we should ensure a leading space appears before the class label, as this enables the model to understand that it should generate a new token. Additionally, when preparing the prompts for the fine-tuning process, there's no need to include few-shot examples, as the model has already been fine-tuned on the task-specific data. Instead, we provide a prompt that includes the review text and any necessary context, followed by a suffix. **Figure 5.4** shows an example of a single line of our JSONL file.

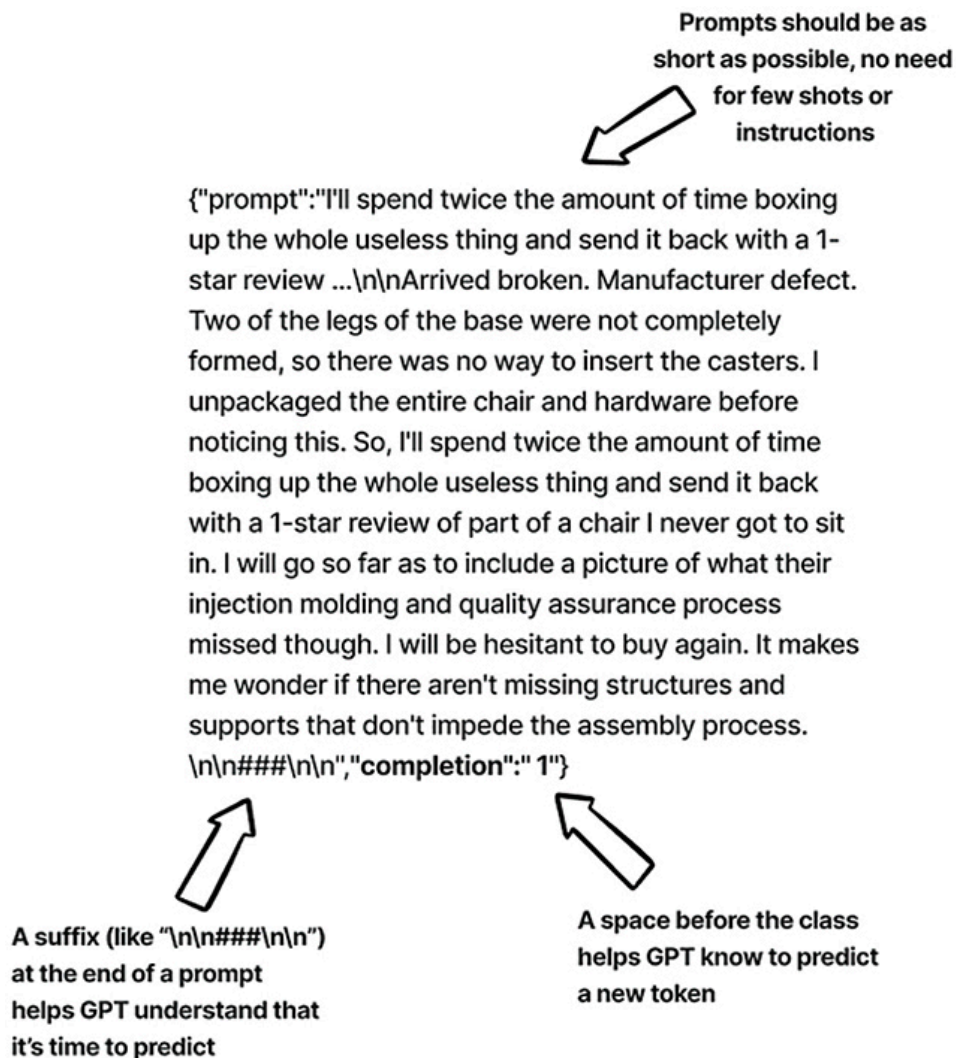


Figure 5.4 A single JSONL example for our training data that we will feed to OpenAI. Every JSON has a prompt key, denoting the input to the model sans any few-shot examples, instructions, or other data, and a completion key, denoting what we want the model to output—a single classification token, in this case. In this example, the user is rating the product with one star.

For our input data, I have concatenated the title and the body of the review as the singular input. This was a personal choice, reflecting my belief that the title can have more direct language to indicate general sentiment while the body likely has more nuanced language to pinpoint the exact number of stars the reviewer will give. Feel free to explore different ways of combining text fields together! We will explore this topic further in later case studies, along with other ways of formatting fields for a single text input.

Listing 5.1 loads the dataset and converts the `train` subset into a pandas DataFrame. Then, it preprocesses the DataFrame using the custom `prepare_df_for_openai` function, which combines the review title and review body into a prompt, creates a new completion column, and filters

the DataFrame to include only English-language reviews. Finally, it removes duplicate rows based on the “prompt” column and returns a DataFrame with only the “prompt” and “completion” columns.

Listing 5.1 Generating a JSONL file for our sentiment training data

[Click here to view code image](#)

```
from datasets import load_dataset
import pandas as pd

# Load the App Review dataset
dataset = load_dataset("amazon_reviews_multi", "all_languages")

...# split into train/test/val

# Creating the 'prompt' column in each dataset (training, validation, and test)
# adding a separator '###\n' to the 'review' column.
# This separator is often used in fine-tuning to signal where the prompt ends and
# the expected output begins.
training_df['prompt'] = training_df['review'] + '\n###\n'
val_df['prompt'] = val_df['review'] + '\n###\n'
test_df['prompt'] = test_df['review'] + '\n###\n'

# Converting the 'star' column in each dataset to a string format and storing it as
# the 'completion' column.
# The 'completion' column will be used as the target variable for sentiment analysis.
training_df['completion'] = training_df['star'].astype(str) # for sentiment
val_df['completion'] = val_df['star'].astype(str) # for sentiment
test_df['completion'] = test_df['star'].astype(str) # for sentiment

# Creating a training dataset in JSONL format after dropping duplicates based on the
# 'prompt' column.
# Random sampling ensures the data is shuffled.
training_df.sample(
    len(training_df)
).drop_duplicates(subset=['prompt'])[['prompt', 'completion']].to_jsonl(
    "app-review-full-train-sentiment-random.jsonl", orient='records', lines=True
)

# Creating a validation dataset in JSONL format after dropping duplicates based on the
# 'prompt' column.
val_df.sample(
    len(val_df)
).drop_duplicates(subset=['prompt'])[['prompt', 'completion']].to_jsonl(
    "app-review-full-val-sentiment-random.jsonl", orient='records', lines=True
)

# Creating a test dataset in JSONL format after dropping duplicates based on the
# 'prompt' column.
test_df.sample(
```

```
len(test_df)
).drop_duplicates(subset=['prompt'])[['prompt', 'completion']].to_json(
    "app-review-full-test-sentiment-random.jsonl", orient='records', lines=True
) orient='records', lines=True)
```

We would follow a similar process with the `validation` subset of the dataset and the holdout `test` subset for a final test of the fine-tuned model. A quick note: We are filtering for English only in this case, but you are free to train your model by mixing in more languages. In this case, I simply wanted to get some quick results at an efficient price.

Setting Up the OpenAI CLI

The OpenAI command line interface (CLI) simplifies the process of fine-tuning and interacting with the API. The CLI allows you to submit fine-tuning requests, monitor training progress, and manage your models, all from your command line. Ensure that you have the OpenAI CLI installed and configured with your API key before proceeding with the fine-tuning process.

To install the OpenAI CLI, you can use `pip`, the Python package manager. First, make sure you have Python 3.6 or later installed on your system. Then, follow these steps:

1. Open a terminal (on macOS or Linux) or a command prompt (on Windows).
2. Run the following command to install the openai package: `pip install openai`
 1. This command installs the OpenAI Python package, which includes the CLI.
3. To verify that the installation was successful, run the following command: `openai --version`
 1. This command should display the version number of the installed OpenAI CLI.

Before you can use the OpenAI CLI, you need to configure it with your API key. To do this, set the `OPENAI_API_KEY` environment variable to your API key value. You can find your API key in your OpenAI account dashboard.

Hyperparameter Selection and Optimization

With our JSONL document created and OpenAI CLI installed, we are ready to select our hyperparameters. Here's a list of key hyperparameters and their definitions:

- **Learning rate:** The learning rate determines the size of the steps the model takes during optimization. A smaller learning rate leads to slower convergence but potentially better accuracy, while a larger learning rate speeds up training but may cause the model to overshoot the optimal solution.
- **Batch size:** Batch size refers to the number of training examples used in a single iteration of model updates. A larger batch size can lead to more stable gradients and faster training, while a smaller batch size may result in a more accurate model but slower convergence.
- **Training epochs:** An epoch is a complete pass through the entire training dataset. The number of training epochs determines how many times the model will iterate over the data, allowing it to learn and refine its parameters.

OpenAI has done a lot of work to find optimal settings for most cases, so we will lean on its recommendations for our first attempt. The only thing we will change is to train for one epoch instead of the default four epochs. We're doing this because we want to see how the performance looks before investing too much time and money. Experimenting with different values and using techniques like grid search will help you find the optimal hyperparameter settings for your task and dataset, but be mindful that this process can be time-consuming and costly.

Our First Fine-Tuned LLM

Let's kick off our first fine-tuning. [Listing 5.2](#) makes a call to OpenAI to train a Babbage model (fastest, cheapest, weakest) for one epoch on our training and validation data.

Listing 5.2 Making our first fine-tuning job creation call

[Click here to view code image](#)

```
# Creating a file object for the training dataset with OpenAI's API.
# The 'file' parameter specifies the path to the training data in JSONL format
# The 'purpose' is set to 'fine-tune,' indicating the file's intended use.
no_system_training_file = client.files.create(
    file=open("app-review-full-train-sentiment-random.jsonl", "rb"),
    purpose='fine-tune'
)
# Creating a file object for the validation dataset with OpenAI's API.
no_system_val_file = client.files.create(
    file=open("app-review-full-val-sentiment-random.jsonl", "rb"),
    purpose='fine-tune'
)
# Initiating the fine-tuning process with OpenAI's API.
# The 'client.fine_tuning.jobs.create' method is used to start the training.
```

```
# Parameters include:
# - 'training_file': The ID of the previously uploaded training dataset file.
# - 'validation_file': The ID of the previously uploaded validation dataset file.
# - 'model': The base model to be fine-tuned. In this case, "babbage-002" is chosen.
# - 'hyperparameters': Dictionary containing training hyperparameters. Here, we specify the number of epochs as 1.

babbage_job = client.fine_tuning.jobs.create(
    training_file=no_system_training_file.id,
    validation_file=no_system_val_file.id,
    model="babbage-002",
    hyperparameters={'n_epochs': 1}
)
```

Evaluating Fine-Tuned Models with Quantitative Metrics

Measuring the performance of fine-tuned models is essential for understanding their effectiveness and identifying areas for improvement. Utilizing metrics and benchmarks, such as accuracy, F1 score, or perplexity, will provide quantitative measures of the model's performance. In addition to quantitative metrics, qualitative evaluation techniques, such as human evaluation and analyzing example outputs, can offer valuable insights into the model's strengths and weaknesses, helping identify areas ripe for further fine-tuning.

After one epoch of training a Babbage model (the training metrics are shown in [Figure 5.5](#)), our classifier has roughly 70% accuracy on the training dataset and on the validation dataset.

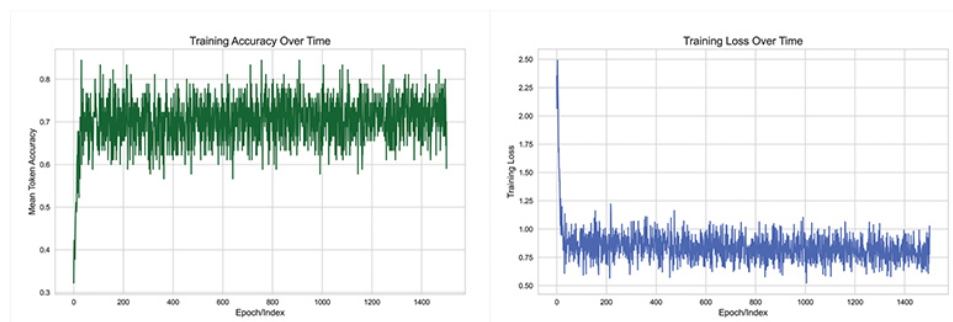


Figure 5.5 Our model is performing well after only one epoch on de-duplicated shuffled training data. These accuracy/loss metrics are calculated from the training set and not the testing set, as OpenAI was never given the final testing set. We should not use these numbers to report but they are an indication that training was successful.

A 70% training accuracy rate might sound low, but predicting the *exact* number of stars is tricky because people aren't always consistent in what they write and how they finally review the product. So, I'll offer two more metrics:

- Relaxing our accuracy calculation to be binary (did the model predict three or fewer stars and was the review three or fewer stars). This will tell us if the model can distinguish between “good” and “bad.”
- Relaxing the calculation to be “one-off” so that, for example, the model predicting two stars would count as correct if the actual rating was one, two, or three stars.

We will highlight all metrics for all models in just a few pages. For our next experiment, let's see if our model gets any better if we train for a further three epochs. This process of taking smaller steps in training and updating already fine-tuned models for more training steps/epochs with new labeled datapoints is called incremental learning, also known as continuous learning or online learning. Incremental learning often results in more controlled learning, which can be ideal when working with smaller datasets or when you want to preserve some of the model's general knowledge. Let's try some incremental learning! We'll take our already fine-tuned Babbage model and let it run for three more epochs on the same data. The results are shown in [Figure 5.6](#).

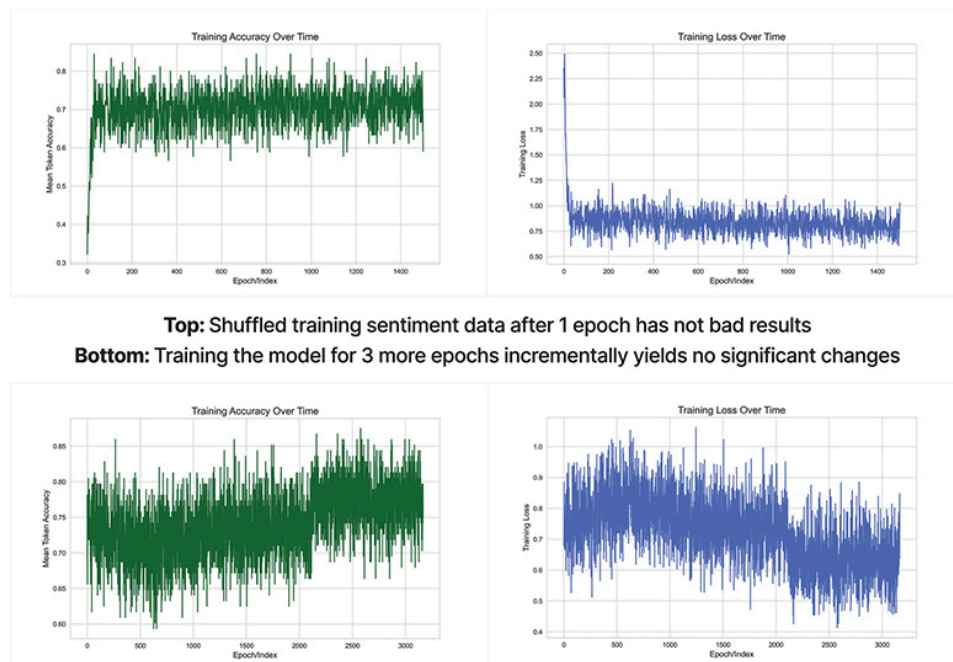


Figure 5.6 Babbage’s training performance seems to barely move during a further three epochs of incremental learning after a successful single epoch. Of course, none of this really matters compared to our models’ final results on the out-of-sample testing set.

Uh oh, more epochs didn’t seem to really do anything. But nothing is set in stone until we test on our holdout `test` data subset and compare it to our first model as well as two fine-tuned gpt-3.5 models. **Figure 5.7** shows cases two prompt variants we will test while fine-tuning GPT-3.5—one with and one without a system prompt.

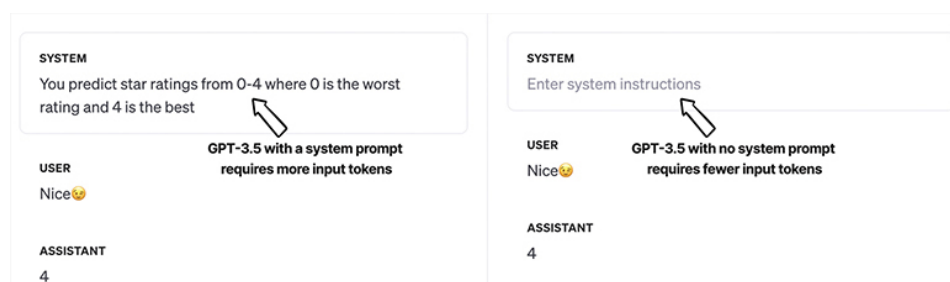


Figure 5.7 We fine-tuned two versions of GPT-3.5: one with a system prompt (left) and one without a system prompt (on the right). Each model will, like Babbage, only take in the review as the user message.

Table 5.1 shows the results. Recall that the testing subset was not given to OpenAI; instead, we held it out for final model comparisons.

Table 5.1 **OpenAI Fine-Tuning Results**

Metric (on hold-out test set)	Babbage - 1 epoch	Babbage - 4 epochs	GPT-3.5 - 1 epoch - No sys- tem prompt	GPT-3.5 - 1 epoch - With sys- tem prompt
Accuracy	64.68%	63.21%	63.45%	64.42%
“Good” versus “bad”	72.36%	71.09%	71.46%	72.13%

Metric (on hold-out test set)	Babbage - 1 epoch	Babbage - 4 epochs	GPT-3.5 - 1 epoch - No system prompt	GPT-3.5 - 1 epoch - With system prompt
<i>One-off accuracy</i>	79.72%	78.48%	78.48%	79.51%
<i>Cost to fine-tune (overall in USD)</i>	\$1.13	\$4.53	\$39.88	\$70.30

[Figures 5.8](#) and [5.9](#) show these results as well.

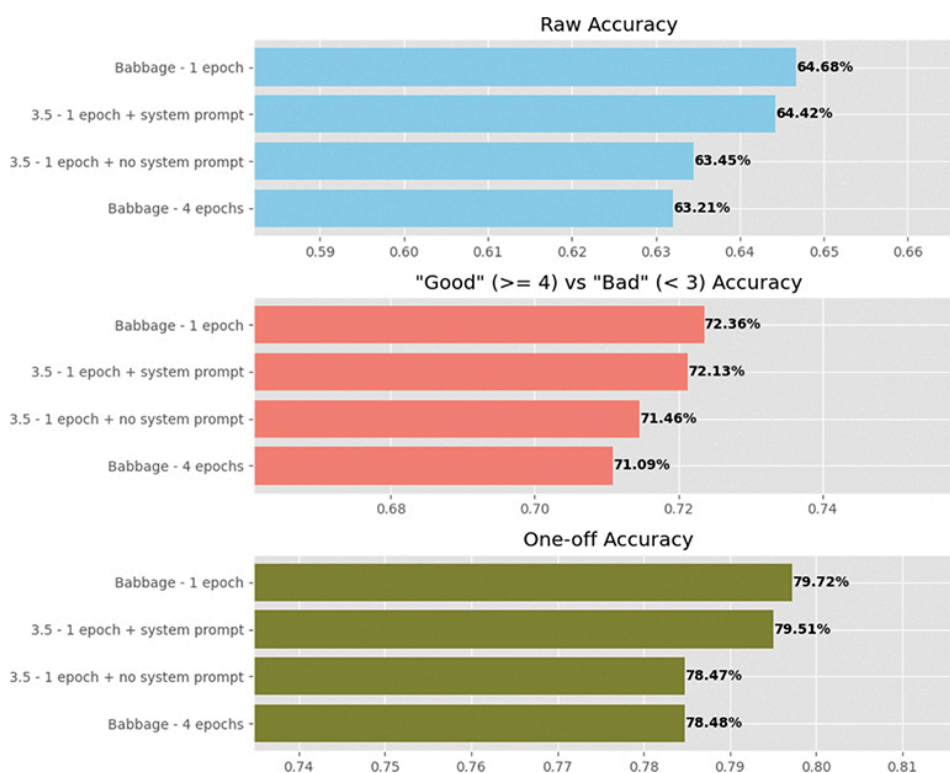


Figure 5.8 Our four fine-tuned OpenAI models tested across the same holdout testing set.

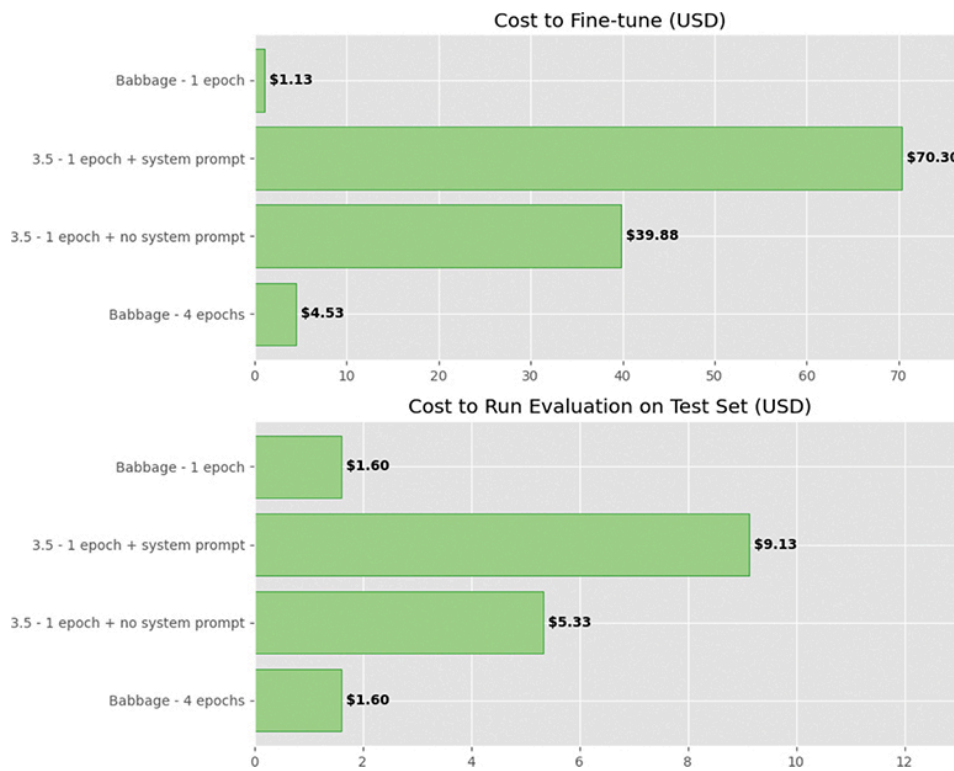


Figure 5.9 Cost projections of our four fine-tuned OpenAI models.

This is stunning! Our 1.3 billion parameter is outperforming (albeit not by that much) the 175 billion parameter GPT-3.5 models. This outcome is not so surprising given that, regardless of parameter size, there is often a ceiling to the number of patterns a machine learning model can encode from a static training set. Put another way, the dataset itself is likely filled with inconsistencies such as conflicting data points where the reviews are similar, but the star ratings are different. For this reason, a model, no matter how large, can learn only so much. And when the task is as simple as a single next-token prediction, larger model sizes—which are better for tasks involving larger and more varied vocabularies—aren’t needed as much.

So, for 40 to 70 times the price, GPT-3.5 ended up barely underperforming against the smaller Babbage model. Frankly, that’s not an uncommon story in the world of LLM fine-tuning.

Qualitative Evaluation Techniques

When carried out alongside quantitative metrics, qualitative evaluation techniques offer valuable insights into the strengths and weaknesses of

our fine-tuned model. Examining generated outputs and employing human evaluators can help identify areas where the model excels or falls short, guiding our future fine-tuning efforts.

For example, we can get the probability for our classification by looking at the probabilities of predicting the first token either in the playground (as seen in [Figure 5.10](#)) or via the API's `logprobs` value (as seen in [Listing 5.3](#)).

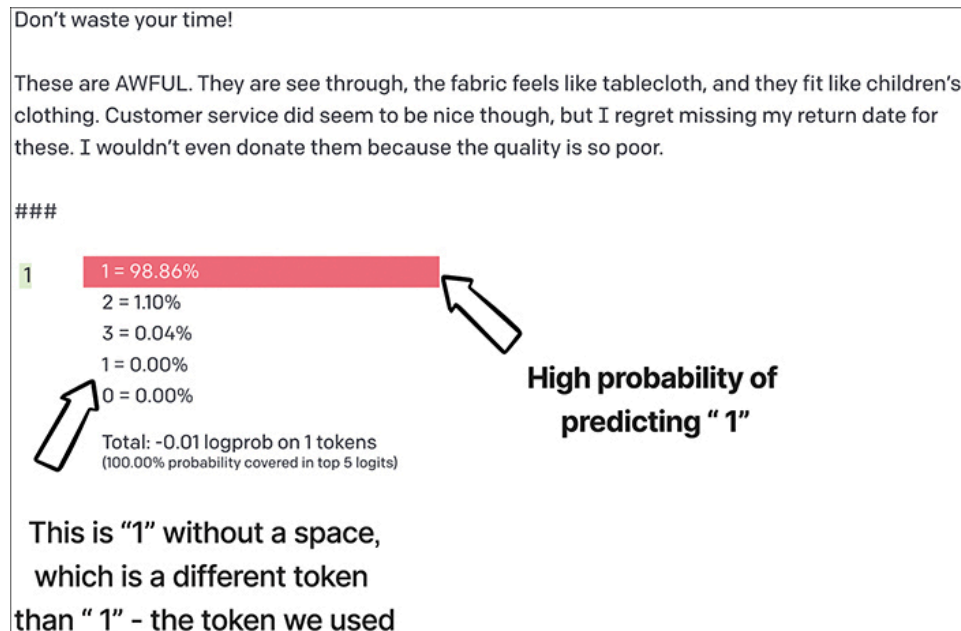


Figure 5.10 The playground and the API for Babbage-like models (including our fine-tuned Babbage model, as seen in this figure) offer token probabilities that we can use to check the model's confidence on a particular classification. Note that the main option is "1" with a leading space, just as in our training data, but one of the tokens on the top of the list is "1" with no leading space. These are two separate tokens according to many LLMs—which is why I am calling this distinction out so often. It can be easy to forget and mix them up.

Listing 5.3 Getting token probabilities from the OpenAI API

[Click here to view code image](#)

```
# Importing the numpy library to perform mathematical operations
import numpy as np

# Define a function to run the fine-tuned model and get the model's response
def run_ft_model(review, ft_id, system='', chat=False):
    """
    Given a review and a fine-tuned model ID, this function uses OpenAI's Comp[
```

API to

generate a completion. It also calculates the exponential of the top log probabilities for the completion.

Parameters:

- review (str): The text of the review.
- ft_id (str): The ID of the fine-tuned model.

Returns:

- str: The completion generated by the model.
- dict: A dictionary of tokens and their corresponding exponential of top log probabilities.

"""

```
# Use OpenAI's API to create a completion using the fine-tuned model
if chat:
    completion = client.chat.completions.create(
        model=ft_id,
        messages=[
            {"role": "system", "content": system},
            {"role": "user", "content": review}
        ],
        max_tokens=1,
        temperature=0.1,
        logprobs=True, # Request the top 5 log probabilities for the completion
        top_logprobs=5

    )
    text = completion.choices[0].message.content.strip()
    probs = {t.token: np.exp(t.logprob) for t in completion.choices[0].logprobs}

    return text, probs
else:
    completion = client.completions.create(
        model=ft_id, # Specify the fine-tuned model ID
        prompt=f'{review}\n###\n', # Format the review with the prompt string
        max_tokens=1, # Limit the response to 1 token (useful for
classification tasks)
        temperature=0.1, # Set a low temperature for deterministic output
        logprobs=5 # Request the top 5 log probabilities for the completion
    )

    # Extract the model's completion text and strip any extra whitespace
    text = completion.choices[0].text.strip()

    # Convert log probabilities to probabilities using exponential function
    # Provide clearer understanding of the model's confidence in its response
    probs = {k: np.exp(v) for k, v in completion.choices[0].logprobs.items()}

    return text, probs
```

[Click here to view code image](#)

```
run_ft_model('i hate it', gpt_3_5_with_system_job.fine_tuned_model,
chat=True, system=system_prompt)

('0',
 {'0': 0.8119405465788642,
  '4': 0.09705203509841609,
  '1': 0.051789419879963904,
  '2': 0.026422253190714406,
  '3': 0.012679542640306907})

run_ft_model(
    'I hated this thing it was the worst',
    client.fine_tuning.jobs.retrieve(babbage_job.id).fine_tuned_model
) # babbage for one epoch

('0',
 {'0': 0.9148366996154271,
  '1': 0.03817410777964789,
  '4': 0.03247224352290873,
  '2': 0.009867273547689607,
  '3': 0.004406479093077916})
```

We will explore more on this idea of probability “calibration” in [Chapter 12](#) on evaluations. For now, between quantitative and qualitative measures, let’s assume we believe our model is ready to go into production—or at least a development or staging environment for further testing. Let’s take a minute to consider how we can incorporate our new model into our applications.

Integrating Fine-Tuned OpenAI Models into Applications

Integrating a fine-tuned GPT-3 model into your application is identical to using a base model provided by OpenAI. The primary difference is that you’ll need to reference your fine-tuned model’s unique identifier when making API calls. Here are the key steps to follow:

- 1. Identify your fine-tuned model:** After completing the fine-tuning process, you will receive a unique identifier for your fine-tuned model—something like `ft:babbage-002:personal::9PWE7zS2`. Make sure to note this identifier, as it will be required for API calls.
- 2. Use the OpenAI API normally:** Use your OpenAI API to make requests to your fine-tuned model. When making requests, replace the base

model's name with your fine-tuned model's unique identifier. [Listing 5.3](#) offers an example of doing this.

3. **Adapt any application logic:** Since fine-tuned models may require different prompt structures or generate different output formats, you may need to update your application's logic to handle these variations. For example, in our prompts, we concatenated the review title with the body and added a custom suffix “\n\n###\n\n”.
4. **Monitor and evaluate performance:** Continuously monitor your fine-tuned model's performance and collect user feedback. You may need to iteratively fine-tune your model with even more data to improve its accuracy and effectiveness.

We will fine-tune autoregressive models with more complex datasets in later chapters. For now, let's give an open-source model a chance to play in the space.

OpenAI Versus Open-Source Autoencoding BERT

In [Chapter 1](#), we looked at the two sides of the LLM family tree. Autoregressive models (which OpenAI specializes in) learn by predicting the next token in a sequence but are blinded to future context during training. In contrast, autoencoding models (like BERT) have full access to context before and after the blanks during pre-training (the “B” in BERT stands for bi-directional), which make them much more efficient at capturing multiple meanings of words/tokens using fewer parameters and pre-training data.

To continue our fine-tuning experiment, I picked one of the tiniest BERT models out there to compare against OpenAI's models—DistilBERT. DistilBERT is a distilled version of BERT. We will explore distillation in much more detail in [Chapter 11](#). The full code to fine-tune DistilBERT can be found on our GitHub, as usual. [Figures 5.11](#) and [5.12](#) show the drastic performance difference using our open-source autoencoding model.

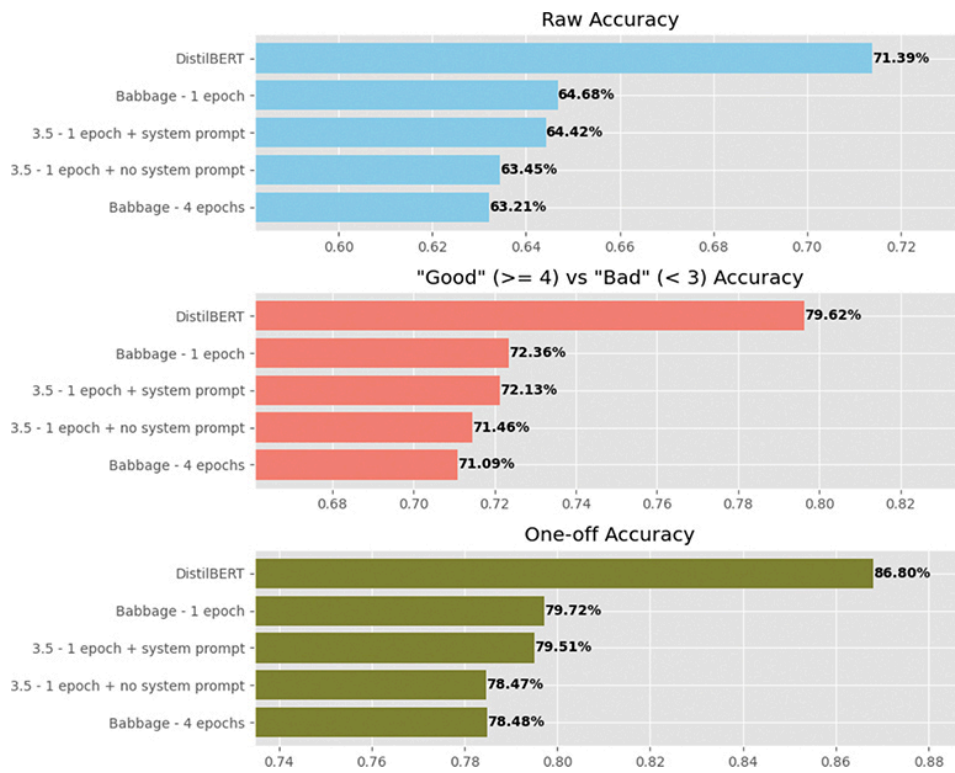


Figure 5.11 BERT is beating all of our OpenAI models on the holdout testing set.

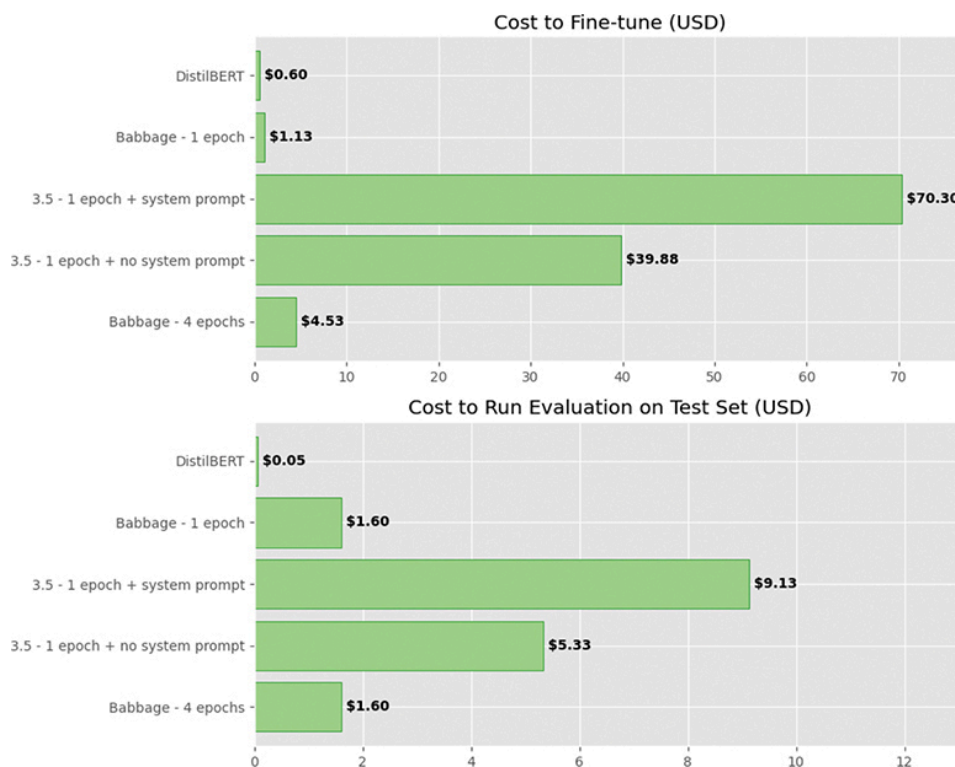


Figure 5.12 Our BERT model is even cheaper than fine-tuning Babbage (estimated based on my colab notebook using the T4 GPU).

This is incredible! Our BERT model handily outperformed all fine-tuned OpenAI models, even though the BERT model I used has only 70 **million** parameters (i.e., it is approximately 2500 times smaller than GPT-3.5 and approximately 18 times smaller than Babbage).

To be clear, open-source autoencoding models won't always be better than closed-source autoregressive models like the ones from OpenAI. I was happy to see such drastic differences in performance, but I was able to produce them only by following my own rules of testing these models fairly against the out-of-sample testing set.

Summary

Fine-tuning LLMs like GPT-4 and BERT is an effective way to enhance their performance on specific tasks or domains. By integrating a fine-tuned model into your application and following best practices for deployment, you can create a more efficient, accurate, and cost-effective language processing solution. Continuously monitor and evaluate your model's performance, and iterate on its fine-tuning to ensure it meets the evolving needs of your application and users.

We will revisit the idea of fine-tuning in later chapters with some more complicated examples while also exploring the fine-tuning strategies for open-source models to achieve even further cost reductions.