Addison-Wesley Data & Analytics Series

QUICK START GUIDE TO LARGE LANGUAGE MODELS

Strategies and Best Practices for using ChatGPT and Other LLMs

Early Release
RAW & UNEDITED

SINAN OZDEMIR

ADDISON-WESLEY DATA & ANALYTICS SERIES

QUICK START GUIDE TO LARGE LANGUAGE MODELS

Strategies and Best Practices for using ChatGPT and Other LLMs

Early Release
RAW & UNEDITED

SINAN OZDEMIR

Quick Start Guide to Large Language Models

Strategies and Best Practices for using ChatGPT and Other LLMs

Sinan Ozdemir

Addison-Wesley

Contents at a Glance

- Preface
- Part I: Introduction to Large Language Models
  - 1. Overview of Large Language Models
  - 2. Launching an Application with Proprietary Models
  - 3. Prompt Engineering with GPT3
  - 4. Fine-Tuning GPT3 with Custom Examples
- Part II: Getting the most out of LLMs
  - 5. Advanced Prompt Engineering Techniques
  - 6. Building a Recommendation Engine
  - 7. Combining Transformers
  - 8. Fine-Tuning Open-Source LLMs
  - 9. Deploying Custom LLMs to the Cloud

Table of Contents

- Preface
- Part I: Introduction to Large Language Models
  - Chapter 1: Overview of Large Language Models
    - What Are Large Language Models (LLMs)?
    - Popular Modern LLMs
    - Domain-Specific LLMs
    - Applications of LLMs
  - Chapter 2: Launching an Application with Proprietary Models
    - Introduction
    - The Task
    - Solution Overview
    - The Components
    - Putting It All Together
    - The Cost of Closed-Source
    - Summary
  - Chapter 3: Prompt Engineering with GPT3
    - Introduction
    - Prompt Engineering
    - Working with Prompts Across Models
    - Building a Q/A bot with ChatGPT
    - Summary
  - Chapter 4: Fine-Tuning GPT3 with Custom Examples

        - Overview of Transfer Learning & Fine-tuning
        - Overview of GPT3 Fine-tuning API
        - Using Fine-tuned GPT3 Models to Get Better Results
- Part II: Getting the most out of LLMs
    - Chapter 5: Advanced Prompt Engineering Techniques
        - Input/Output Validation
        - Chain of Thought Prompting
        - Prompt Chaining Workflows
        - Preventing against Prompt Injection Attacks
        - Building a bot that can execute code on our behalf
    - Chapter 6: Building a Recommendation Engine
        - Overview of Siamese BERT Architectures
        - Fine-Tuning BERT for Classifying + Tagging Items
        - Fine-Tuning Siamese BERT for Recommendations
    - Chapter 7: Combining Transformers
        - Overview of Vision Transformer
        - Building an Image Captioning System with GPT-J
    - Chapter 8: Fine-Tuning Open-Source LLMs
        - Overview of T5
        - Building Translation/Summarization Pipelines with T5
    - Chapter 9: Deploying Custom LLMs to the Cloud
        - Overview of Cloud Deployment
        - Best Practices for Cloud Deployment

Preface

The advancement of Large Language Models (LLMs) has revolutionized the field of Natural Language Processing in recent years. Models like BERT, T5, and ChatGPT have demonstrated unprecedented performance on a wide range of NLP tasks, from text classification to machine translation. Despite their impressive performance, the use of LLMs remains challenging for many practitioners. The sheer size of these models, combined with the lack of understanding of their inner workings, has made it difficult for practitioners to effectively use and optimize these models for their specific needs.

This practical guide to the use of LLMs in NLP provides an overview of the key concepts and techniques used in LLMs and explains how these models work and how they can be used for various NLP tasks. The book also covers advanced topics, such as fine-tuning, alignment, and information retrieval while providing practical tips and tricks for training and optimizing LLMs for specific NLP tasks.

This work addresses a wide range of topics in the field of Large Language Models, including the basics of LLMs, launching an application with proprietary models, fine-tuning GPT3 with custom examples, prompt engineering, building a recommendation engine, combining Transformers, and deploying custom LLMs to the cloud. It offers an in-depth look at the various concepts, techniques, and tools used in the field of Large Language Models.

Topics covered:

Coding with Large Language Models (LLMs)

Overview of using proprietary models

OpenAI, Embeddings, GPT3, and ChatGPT

Vector databases and building a neural/semantic information retrieval system

Fine-tuning GPT3 with custom examples

Prompt engineering with GPT3 and ChatGPT

Advanced prompt engineering techniques

Building a recommendation engine

Combining Transformers

. Deploying custom LLMs to the cloud

Part I: Introduction to Large Language Models

1

Overview of Large Language Models

Ever since an advanced artificial intelligence (AI) deep learning model called the Transformer was introduced by a team at Google Brain in 2017, it has become the standard for tackling various natural language processing (NLP) tasks in academia and industry. It is likely that you have interacted with a Transformer model today without even realizing it, as Google uses BERT to enhance its search engine by better understanding users’ search queries. The GPT family of models from OpenAI have also received attention for their ability to generate human-like text and images.

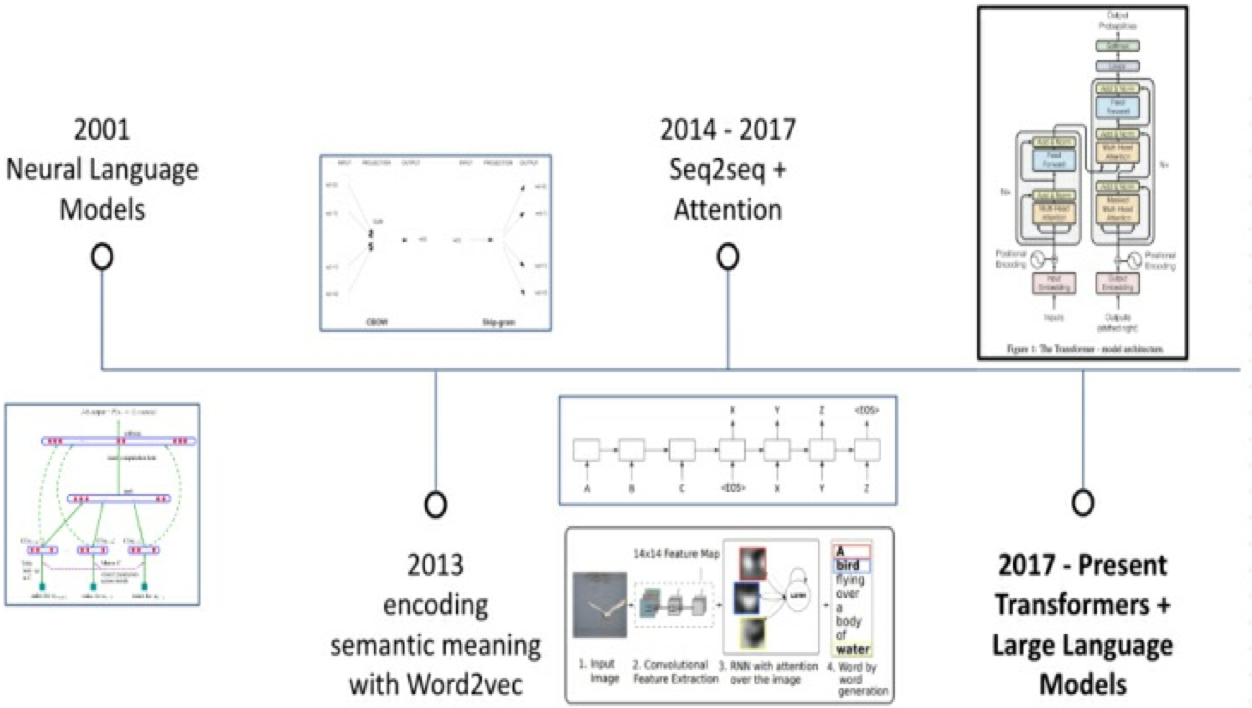

2001
Neural Language Models
2014 - 2017
Seq2seq + Attention
CBOW
Skip-gram
Figure 1: The Transformer - model architecture.
A
B
C
<EOS>
X
Y
Z
14x14 Feature Map
A bird flying over a body of water
1. Input Image
2. Convolutional Feature Extraction
3. RNN with attention over the image
4. Word by word generation
2013
encoding semantic meaning with Word2vec
2017 - Present
Transformers + Large Language Models

Figure 1.1 A brief history of Modern NLP highlights using deep learning to

tackle language modeling, advancements in large scale semantic token embeddings (Word2vec), sequence to sequence models with attention (something we will see in more depth later in this chapter), and finally the Transformer in 2017.

These Transformers now power applications such as GitHub’s Copilot (developed by OpenAI in collaboration with Microsoft), which can convert comments and snippets of code into fully functioning source code that can even call upon other LLMs (like in Listing 1.1) to perform NLP tasks.

Using the Copilot LLM to get an output from Facebook’s BART LLM

```
from transformers import pipeline

def classify_text(email):
    """
    Use Facebook's BART model to classify an email int

    Args:
        email (str): The email to classify
    Returns:
        str: The classification of the email
    """
    # COPILOT START. EVERYTHING BEFORE THIS COMMENT WA
    classifier = pipeline(
        'zero-shot-classification', model='facebook/ba
    labels = ['spam', 'not spam']
    hypothesis_template = 'This email is {}.'

    results = classifier(
        email, labels, hypothesis_template=hypothesis_

    return results['labels'][0]
    # COPILOT END
```

In this listing, I use Copilot to take in only a Python function definition and some comments I wrote and wrote all of the code to make the function do what I wrote. No cherry-picking here, just a fully working python function that I can call like this:

```
classify_text('hi I am spam')  # spam
```

It appears we are surrounded by LLMs, but just what are they doing under the hood? Let’s find out!

What Are Large Language Models (LLMs)?

Large language models (LLMs) are AI models that are usually (but not necessarily) derived from the Transformer architecture and are designed to *understand* and *generate* human language, code, and much more. These models are trained on vast amounts of text data, allowing them to capture the complexities and nuances of human language. LLMs can perform a wide range of language tasks, from simple text classification to text generation, with high accuracy, fluency, and style.

In the healthcare industry, LLMs are being used for electronic medical record (EMR) processing, clinical trial matching, and drug discovery. In finance, LLMs are being utilized for fraud detection, sentiment analysis of financial news, and even trading strategies. LLMs are also used for customer service automation via chatbots and virtual assistants. With their versatility and highly performant natures, Transformer-based LLMs are becoming an increasingly valuable asset in a variety of industries and applications.

Note

I will use the term **understand** a fair amount in this text. I am usually referring to “Natural Language Understanding” (NLU) which is a research branch of NLP that focuses on developing algorithms and models that can accurately interpret human language. As we will see, NLU models excel at tasks such as classification, sentiment analysis, and named entity recognition. However, it is important to note that while these models can perform complex language tasks, they do not possess true understanding in the way humans do.

The success of LLMs and Transformers is due to the combination of several ideas. Most of these ideas had been around for years but were also being actively researched around the same time. Mechanisms such as attention, transfer learning, and scaling up neural networks which provide the scaffolding for Transformers were seeing breakthroughs right around the same time. Figure 1.1 outlines some of the biggest advancements in NLP in the last few decades, all leading up to the invention of the Transformer.

The Transformer architecture itself is quite impressive. It can be highly parallelized and scaled in ways that previous state of the art NLP models could not be, allowing it to scale to much larger data sets and training times than previous NLP models. The Transformer uses a special kind of attention calculation called **self-attention** to allow each word in a sequence to “attend to” (look to for context) all other words in the sequence, enabling it to capture long-range dependencies and contextual relationships between words. Of course, no architecture is perfect. Transformers are still limited to an input context window which represents the maximum length of text it can process at any given moment.

Since the advent of the Transformer in 2017, the ecosystem around using and deploying Transformers has only exploded. The aptly named “Transformers” library and its supporting packages have made it accessible for practitioners to use, train, and share models, greatly accelerating its adoption and being used by thousands of organizations and counting. Popular LLM repositories like Hugging Face have popped up, providing access to powerful open-source models to the masses. In short, using and productionizing a Transformer has never been easier.

That’s where this book comes in.

My goal is to guide you on how to use, train, and optimize all kinds of LLMs for practical applications while giving you just enough insight into the inner workings of the model to know how to make optimal decisions about model choice, data format, fine-tuning parameters, and so much more.

My aim is to make using Transformers accessible for software developers, data scientists, analysts, and hobbyists alike. To do that, we should start on a level playing field and learn a bit more about LLMs.

Definition of LLMs

To back up only slightly, we should talk first about the specific NLP task that LLMs and Transformers are being used to solve and provides the foundation layer for their ability to solve a multitude of tasks. **Language modeling** is a subfield of NLP that involves the creation of statistical/deep learning models for predicting the likelihood of a sequence of tokens in a specified **vocabulary** (a limited and known set of tokens). There are generally two kinds of language modeling tasks out there: autoencoding tasks and autoregressive tasks Figure 1.2)

Note

The term **token** refers to the smallest unit of semantic meaning created by breaking down a sentence or piece of text into smaller units and are the basic inputs for an LLM. Tokens can be words but also can be “sub-words” as we will see in more depth throughout this book. Some readers may be familiar with the term “n-gram” which refers to a sequence of n consecutive tokens.

Autoregressive language models are trained to predict the next token in a sentence, based only on the previous tokens in the phrase. These models correspond to the decoder part of the transformer model, and a mask is applied to the full sentence so that the attention heads can only see the tokens that came before. Autoregressive models are ideal for text generation and a good example of this type of model is GPT.

Autoencoding language models are trained to reconstruct the original sentence from a corrupted version of the input. These models correspond to the encoder part of the transformer model and have access to the full input without any mask. Autoencoding models create a bidirectional representation of the whole sentence. They can be fine-tuned for a variety of tasks such as text generation, but their main application is sentence classification or token classification. A typical example of this type of model is BERT.

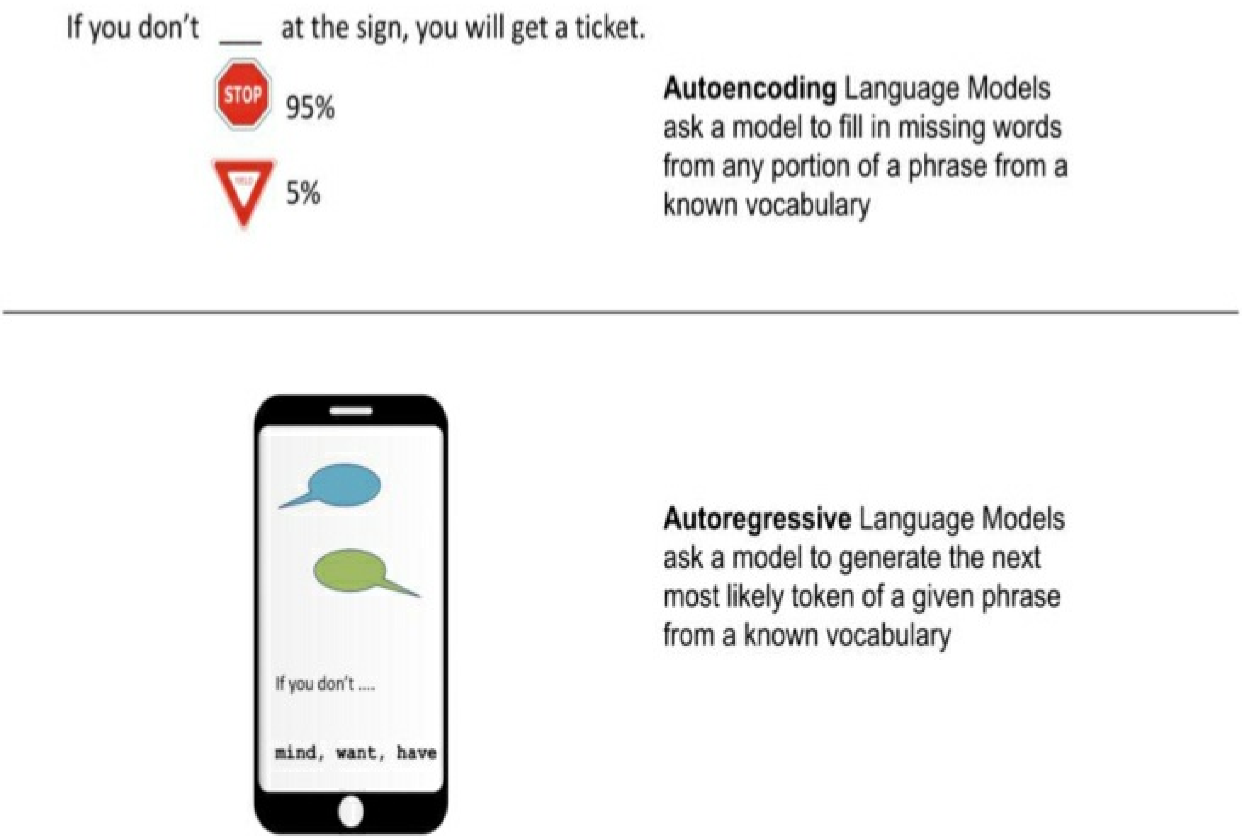
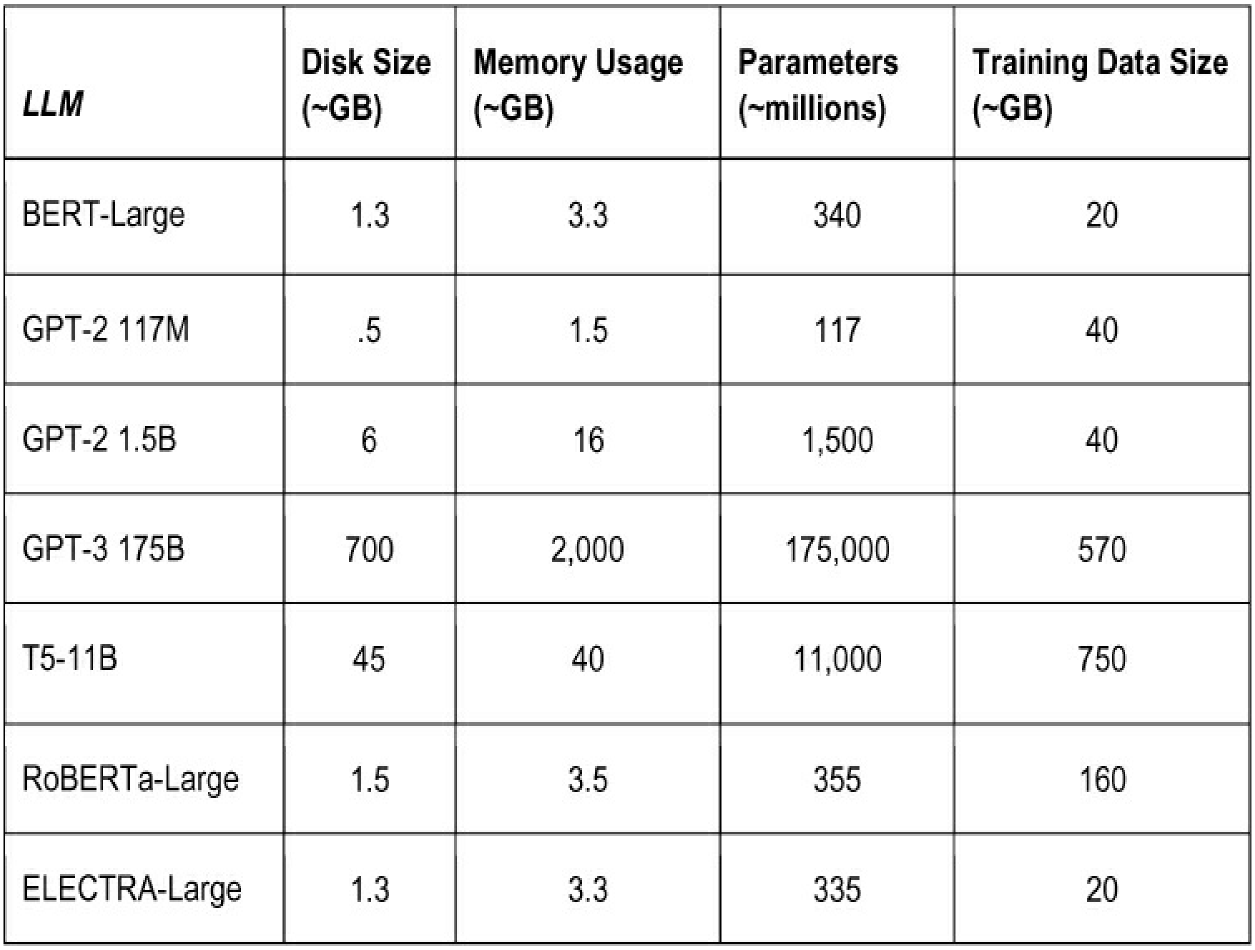

If you don't ___ at the sign, you will get a ticket.
STOP
95%
5%
Autoencoding Language Models ask a model to fill in missing words from any portion of a phrase from a known vocabulary
Autoregressive Language Models ask a model to generate the next most likely token of a given phrase from a known vocabulary
If you don't ....
mind, want, have

Figure 1.2 Both the autoencoding and autoregressive language modeling task involves filling in a missing token but only the autoencoding task allows for context to be seen on both sides of the missing token.

To summarize, Large Language Models (LLMs) are language models that are either autoregressive , autoencoding, or a combination of the two. Modern LLMs are usually based on the Transformer architecture which is what we will use but they can be based on another architecture. The defining feature of LLMs is their large size and large training datasets which enables them to perform complex language tasks, such as text generation and classification, with high accuracy and with little to no fine-tuning.

Table 1.1 shows the disk size, memory usage, number of parameters, and approximate size of the pre-training data for several popular large language models (LLMs). Note that these sizes are approximate and may vary depending on the specific implementation and hardware used.

Table 1.1 **Comparison of Popular Large Language Models (LLMs)**



LLM	Disk Size (~GB)	Memory Usage (~GB)	Parameters (~millions)	Training Data Size (~GB)
BERT-Large	1.3	3.3	340	20
GPT-2 117M	.5	1.5	117	40
GPT-2 1.5B	6	16	1,500	40
GPT-3 175B	700	2,000	175,000	570
T5-11B	45	40	11,000	750
RoBERTa-Large	1.5	3.5	355	160
ELECTRA-Large	1.3	3.3	335	20

But size is everything. Let’s look at some of the key characteristics of LLMs and then dive into how LLMs learn to read and write.

Key Characteristics of LLMs

The original Transformer architecture, as devised in 2017, was a **sequence-**

to-sequence model, which means it had two main components:

An **encoder** which is tasked with taking in raw text, splitting them up into its core components (more on this later), converting them into vectors (similar to the Word2vec process), and using attention to *understand* the context of the text

A **decoder** which excels at *generating* text by using a modified type of attention to predict the next best token

As shown in Figure 1.3, The Transformer has many other sub-components that we won't get into that promotes faster training, generalizability, and better performance. Today's LLMs are for the most part variants of the original Transformer. Models like BERT and GPT dissect the Transformer into only an encoder and decoder (respectively) in order to build models that excel in understanding and generating (also respectively).

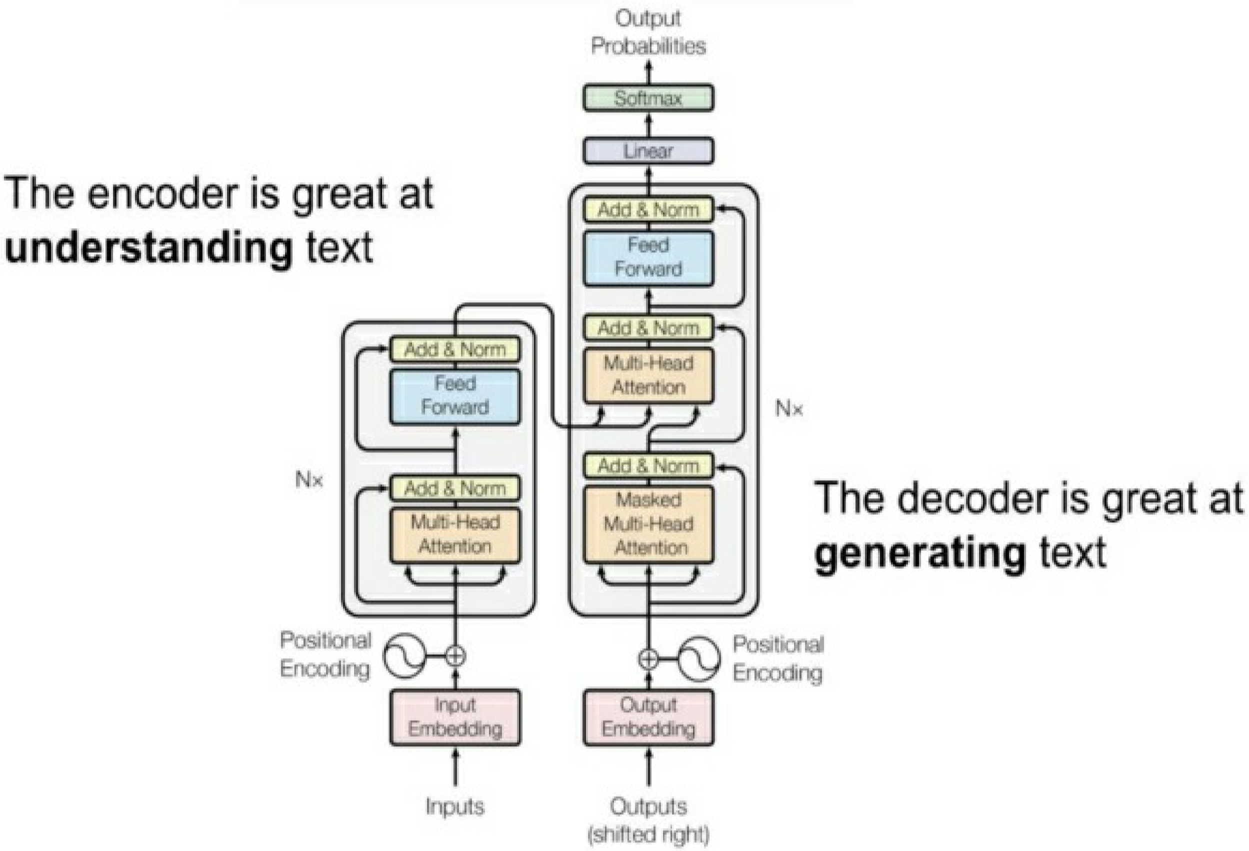


Figure 1.3 The original Transformer has two main components: an encoder which is great at understanding text, and a decoder which is great at generating text. Putting them together makes the entire model a "sequence to sequence" model.

In general, LLMs can be categorized into three main buckets:

Autoregressive models, such as GPT, which predict the next token in a sentence based on the previous tokens. They are effective at generating coherent free-text following a given context

Autoencoding models, such as BERT, which build a bidirectional representation of a sentence by masking some of the input tokens and trying to predict them from the remaining ones. They are adept at capturing contextual relationships between tokens quickly and at scale which make them great candidates for text classification tasks for example.

Combinations of autoregressive and autoencoding, like T5, which can use the encoder and decoder to be more versatile and flexible in generating text. It has been shown that these combination models can generate more diverse and creative text in different contexts compared to pure decoder-based autoregressive models due to their ability to capture additional context using the encoder.

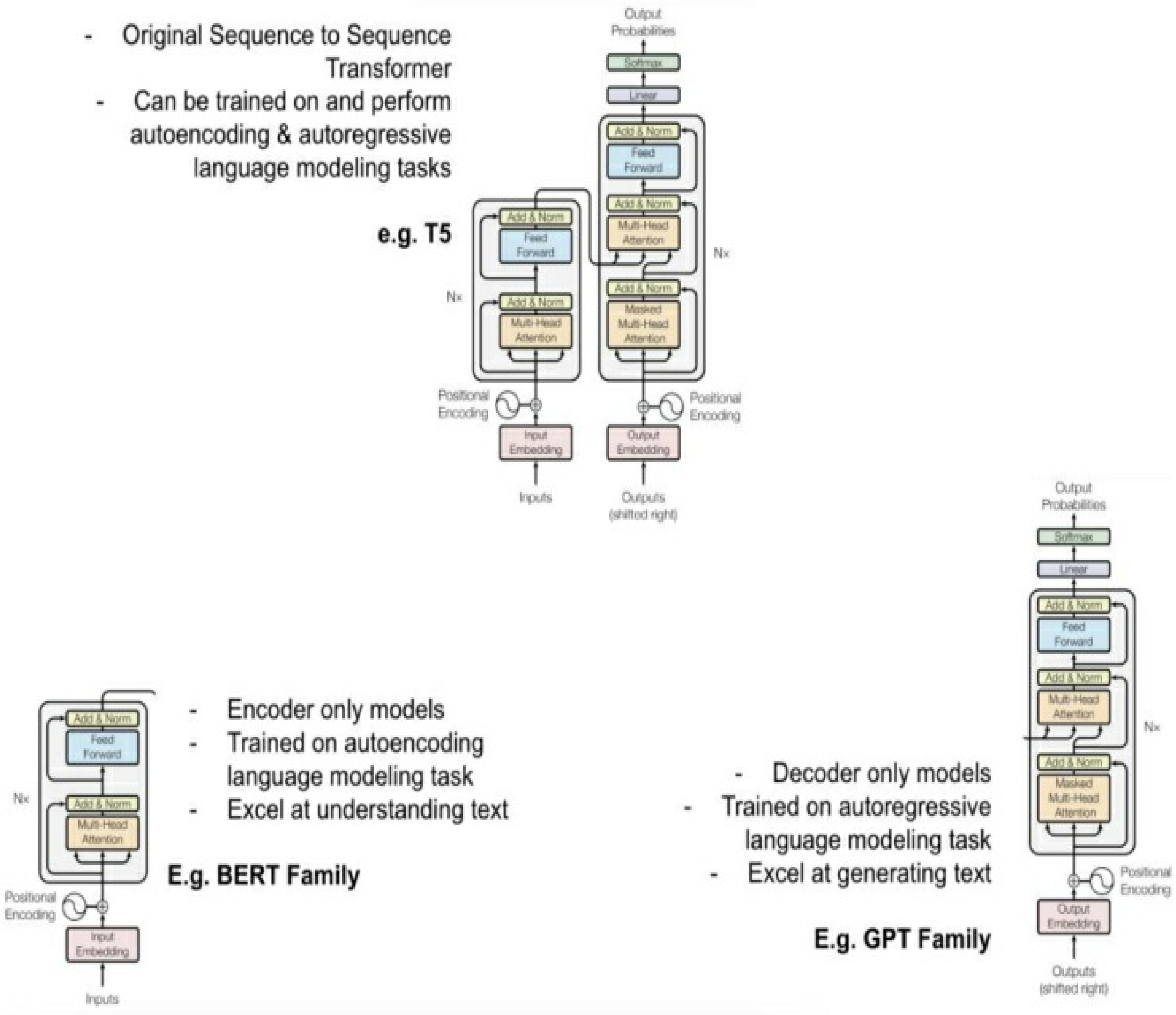

- Original Sequence to Sequence Transformer
- Can be trained on and perform autoencoding & autoregressive language modeling tasks
e.g. T5
- Encoder only models
- Trained on autoencoding language modeling task
- Excel at understanding text
E.g. BERT Family
- Decoder only models
- Trained on autoregressive language modeling task
- Excel at generating text
E.g. GPT Family
Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Multi-Head Attention
Masked Multi-Head Attention
Nx
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Figure 1.4 A breakdown of the key characteristics of LLMs based on how they are derived from the original Transformer architecture.

Figure 1.4 shows the breakdown of the key characteristics of LLMs based on these three buckets.

More Context Please

No matter how the LLM is constructed and what parts of the Transformer it is using, they all care about context (Figure 1.5). The goal is to understand each token as it relates to the other tokens in the input text. Beginning with the popularity of Word2vec around 2013, NLP practitioners and researchers were always curious about the best ways of combining semantic meaning (basically word definitions) and context (with the surrounding tokens) to create the most meaningful token embeddings possible. The Transformer relies on the attention calculation to make this combination a reality.

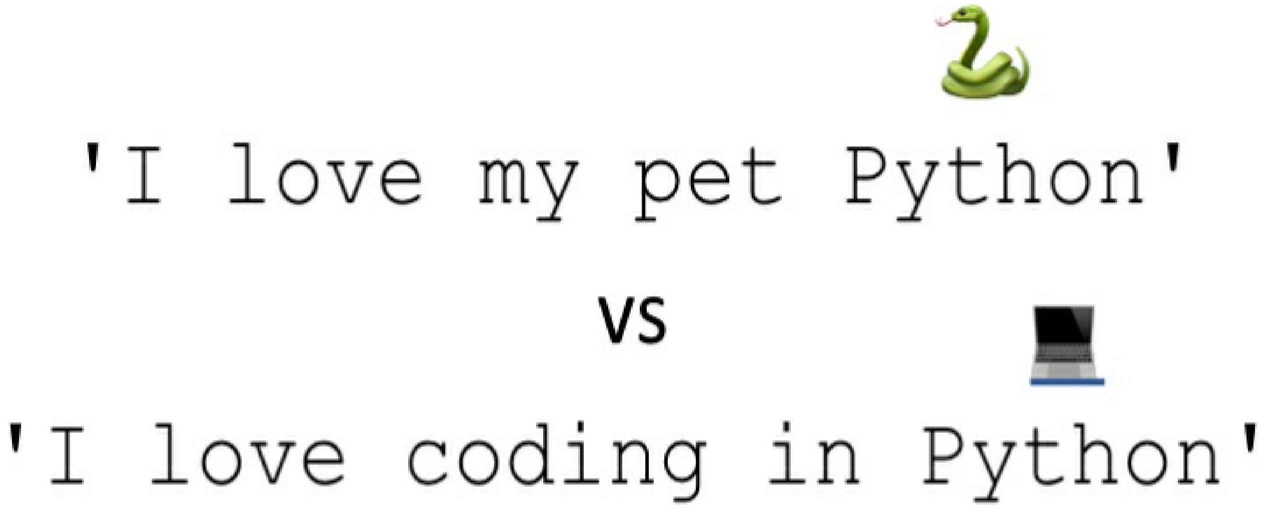


Figure 1.5 LLMs are great at understanding context. The word “Python” can have different meanings depending on the context. We could be talking about a snake, or a pretty cool coding language.

Choosing what kind of Transformer derivation you want isn’t enough. Just choosing the encoder doesn’t mean your Transformer is magically good at understanding text. Let’s take a look at how these LLMs actually learn to read and write.

How LLMs Work

How an LLM is pre-trained and fine-tuned makes all the difference between an alright performing model and something state of the art and highly accurate. We’ll need to take a quick look into how LLMs are pre-trained to understand what they are good at, what they are bad at, and whether or not we would need to update them with our own custom data.

Pre-training

Every LLM on the market has been **pre-trained** on a large corpus of text data and on specific language modeling related tasks. During pre-training, the LLM tries to learn and understand general language and relationships between words. Every LLM is trained on different corpora and on different tasks.

BERT, for example, was originally pre-trained on two publicly available text corpora (Figure 1.6):

English Wikipedia - a collection of articles from the English version of Wikipedia, a free online encyclopedia. It contains a range of topics and writing styles, making it a diverse and representative sample of English language text

At the time 2.5 billion words.

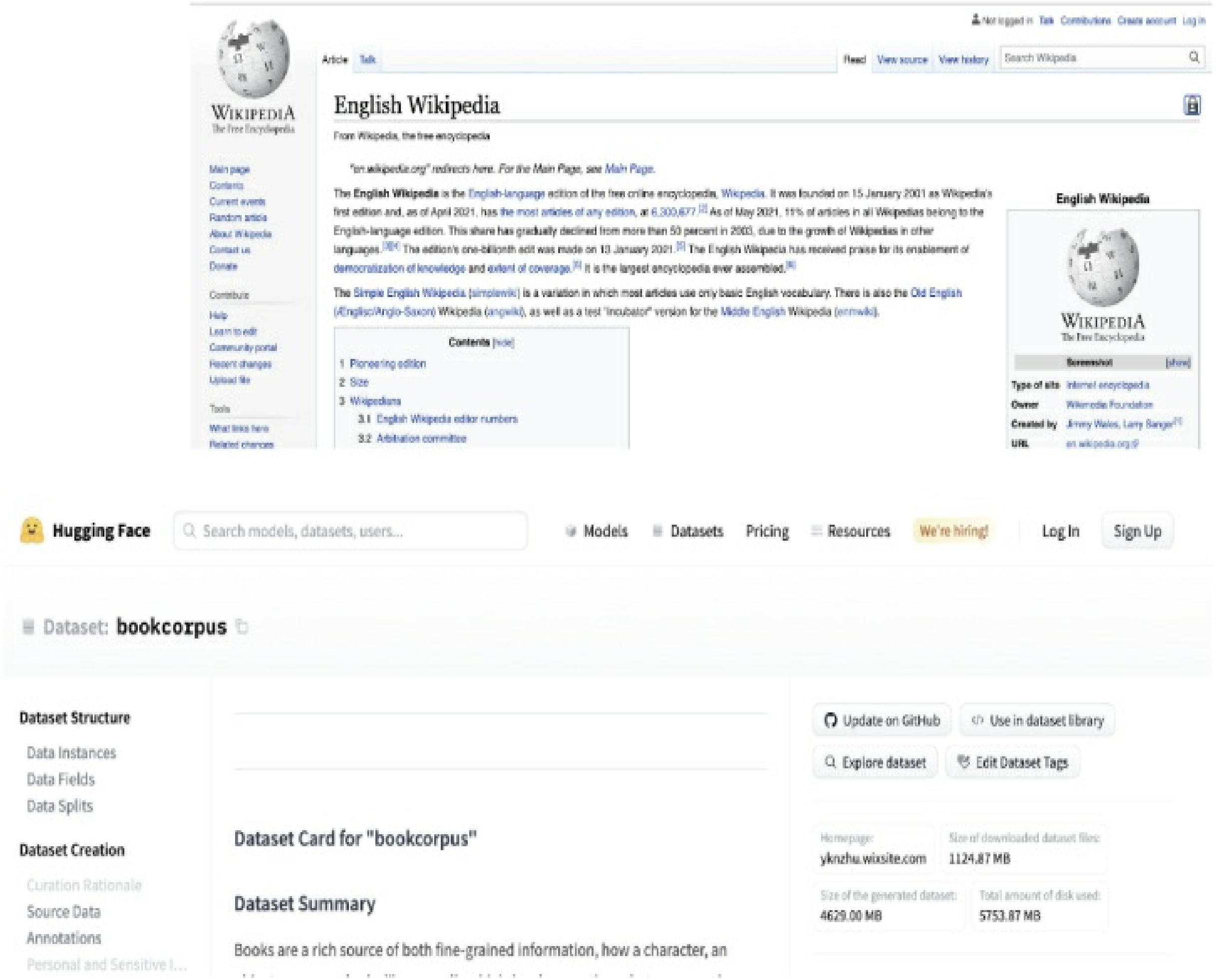
The BookCorpus - a large collection of fiction and non-fiction books. It was created by scraping book text from the web and includes a range of genres, from romance and mystery to science fiction and history. The books in the corpus were selected to have a minimum length of 2000 words and to be written in English by authors with verified identities

800M words.

and on two specific language modeling specific tasks (Figure 1.7):

The Masked Language Modeling (MLM) task (AKA the autoencoding task)—this helps BERT recognize token interactions within a single sentence.

The Next Sentence Prediction Task—this helps BERT understand how tokens interact with each other between sentences.



English Wikipedia

From Wikipedia, the free encyclopedia

"en.wikipedia.org" redirects here. For the Main Page, see Main Page.

The **English Wikipedia** is the English-language edition of the free online encyclopedia, Wikipedia. It was founded on 15 January 2001 as Wikipedia's first edition and, as of April 2021, has the most articles of any edition, at 6,300,677.[2] As of May 2021, 11% of articles in all Wikipedias belong to the English-language edition. This share has gradually declined from more than 50 percent in 2003, due to the growth of Wikipedias in other languages.[3][4] The edition's one-billionth edit was made on 13 January 2021.[5] The English Wikipedia has received praise for its enablement of democratization of knowledge and extent of coverage.[6] It is the largest encyclopedia ever assembled.[7]

The Simple English Wikipedia (simplewiki) is a variation in which most articles use only basic English vocabulary. There is also the Old English (Ænglisc/Anglo-Saxon) Wikipedia (angwiki), as well as a test "incubator" version for the Middle English Wikipedia (enmwiki).

Type of site Internet encyclopedia
Owner Wikimedia Foundation
Created by Jimmy Wales, Larry Sanger[1]
URL en.wikipedia.org

Homepage: yknzhu.wixsite.com
Size of downloaded dataset files: 1124.87 MB
Size of the generated dataset: 4629.00 MB
Total amount of disk used: 5753.87 MB

Dataset Card for "bookcorpus"

Dataset Summary

Books are a rich source of both fine-grained information, how a character, an

Figure 1.6 BERT was originally pre-trained on English Wikipedia and the BookCorpus. More modern LLMs are trained on datasets thousands of times larger.

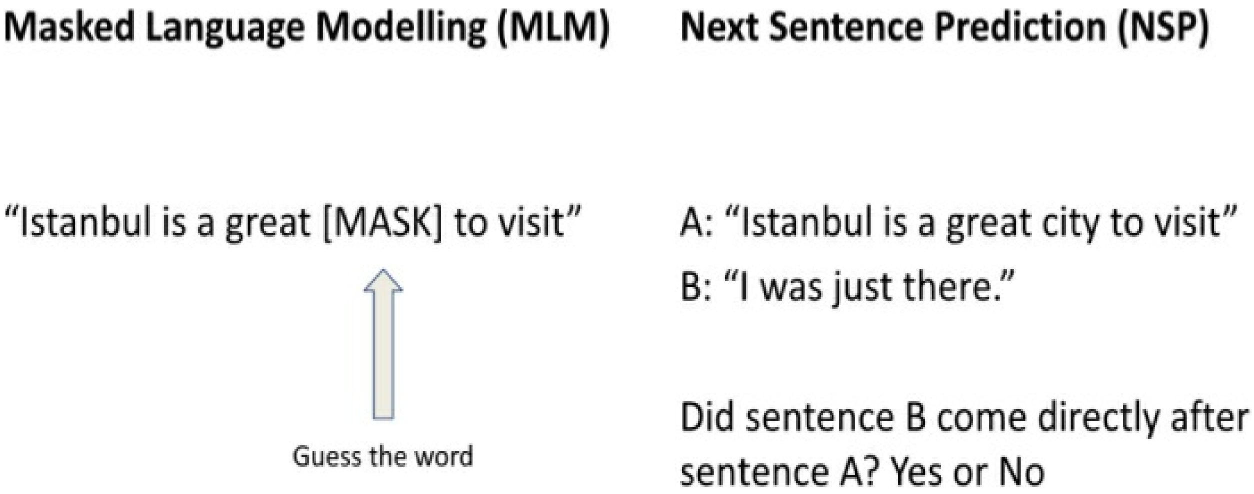
Pre-training on these corpora allowed BERT (mainly via the self-attention mechanism) to learn a rich set of language features and contextual relationships. The use of large, diverse corpora like these has become a common practice in NLP research, as it has been shown to improve the performance of models on downstream tasks.

Note

The pre-training process for an LLM can evolve over time as researchers find better ways of training LLMs and phase out methods that don’t help as much. For example within a year of the original Google BERT release that used the Next Sentence Prediction (NSP) pre-training task, a BERT variant called RoBERTa (yes, most of these LLM names will be fun) by Facebook AI was shown to not require the NSP task to match and even beat the original BERT model’s performance in several areas.

Depending on which LLM you decide to use, it will likely be pre-trained differently from the rest. This is what sets LLMs apart from each other. Some LLMs are trained on proprietary data sources including OpenAI’s GPT family of models in order to give their parent companies an edge over their competitors.

We will not revisit the idea of pre-training often in this book because it’s not exactly the “quick” part of a “quick start guide” but it can be worth knowing how these models were pre-trained because it’s because of this pre-training that we can apply something called transfer learning to let us achieve the state-of-the-art results we want, which is a big deal!


Masked Language Modelling (MLM)
Next Sentence Prediction (NSP)
Guess the word

“Istanbul is a great [MASK] to visit”

A: “Istanbul is a great city to visit”
B: “I was just there.”

Did sentence B come directly after sentence A? Yes or No

Figure 1.7 BERT was pre-trained on two tasks: the autoencoding language modeling task (referred to as the “masked language modeling” task) to teach it individual word embeddings and the “next sentence prediction” task to help it learn to embed entire sequences of text.

Transfer Learning

Transfer learning is a technique used in machine learning to leverage the knowledge gained from one task to improve performance on another related task. Transfer learning for LLMs involves taking an LLM that has been pre-trained on one corpus of text data and then fine-tuning it for a specific “downstream” task, such as text classification or text generation, by updating the model’s parameters with task-specific data.

The idea behind transfer learning is that the pre-trained model has already learned a lot of information about the language and relationships between words, and this information can be used as a starting point to improve performance on a new task. Transfer learning allows LLMs to be fine-tuned for specific tasks with much smaller amounts of task-specific data than it would require if the model were trained from scratch. This greatly reduces the amount of time and resources required to train LLMs. Figure 1.8 provides a visual representation of this relationship.

Fine-tuning

Once a LLM has been pre-trained, it can be fine-tuned for specific tasks. Fine-tuning involves training the LLM on a smaller, task-specific dataset to adjust its parameters for the specific task at hand. This allows the LLM to leverage its pre-trained knowledge of the language to improve its accuracy for the specific task. Fine-tuning has been shown to drastically improve performance on domain-specific and task-specific tasks and lets LLMs adapt quickly to a wide variety of NLP applications.

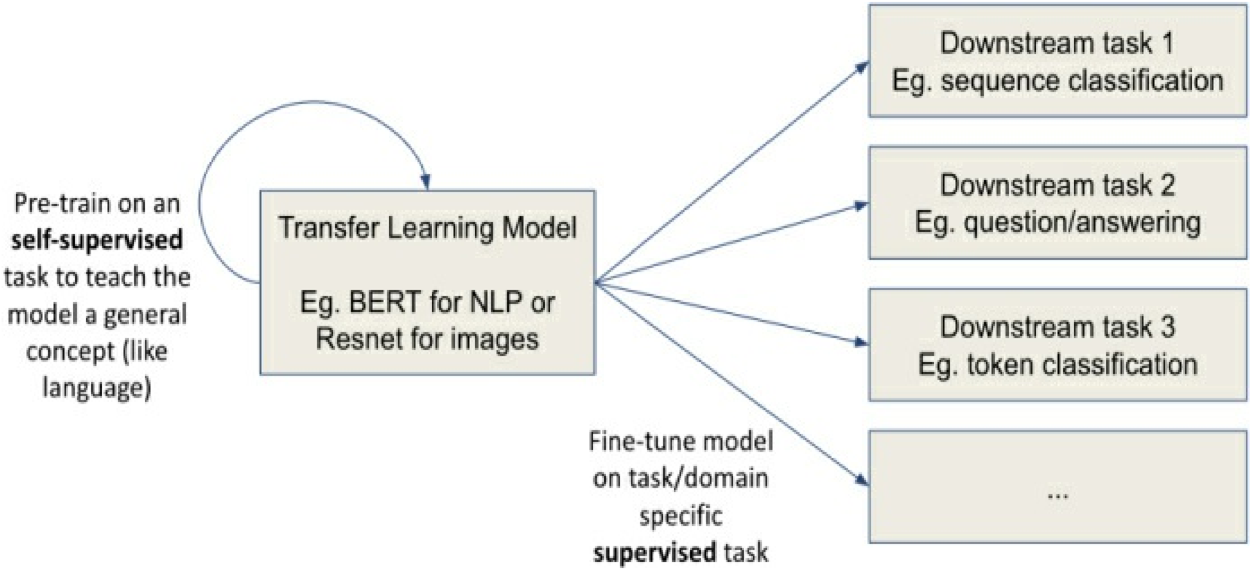

Pre-train on an **self-supervised** task to teach the model a general concept (like language)
Transfer Learning Model
Eg. BERT for NLP or Resnet for images
Downstream task 1
Eg. sequence classification
Downstream task 2
Eg. question/answering
Downstream task 3
Eg. token classification
...
Fine-tune model on task/domain specific **supervised** task

Figure 1.8 The general transfer learning loop involves pre-training a model on a generic dataset on some generic self-supervised task and then fine-tuning the model on a task-specific dataset.

Figure 1.9 shows the basic fine-tuning loop that we will use for our models in later chapters. Whether they are open-sourced or closed-sourced the loop is more or less the same:

We define the model we want to fine-tune as well as any fine-tuning parameters (e.g., learning rate)

We will aggregate some training data (the format and other characteristics depend on the model we are updating)

We compute losses (a measure of error) and gradients (information about how to change the model to minimize error)

We update the model through backpropagation – a mechanism to update model parameters to minimize errors

If some of that went over your head, not to worry: we will rely on pre-built tools from Hugging Face's Transformers package (Figure 1.9) and OpenAI's Fine-tuning API to abstract away a lot of this so we can really focus on our data and our models.

Note

You will not need a Hugging Face account or key to follow along and use any of this code apart from very specific advanced exercises where I will call it out.

Attention

The name of the original paper that introduced the Transformer was called “Attention is all you need”. **Attention** is a mechanism used in deep learning models (not just Transformers) that assigns different weights to different parts of the input, allowing the model to prioritize and emphasize the most important information while performing tasks like translation or summarization. Essentially, attention allows a model to “focus” on different parts of the input dynamically, leading to improved performance and more accurate results. Before the popularization of attention, most neural networks processed all inputs equally and the models relied on a fixed representation of the input to make predictions. Modern LLMs that rely on attention can dynamically focus on different parts of input sequences, allowing them to weigh the importance of each part in making predictions.

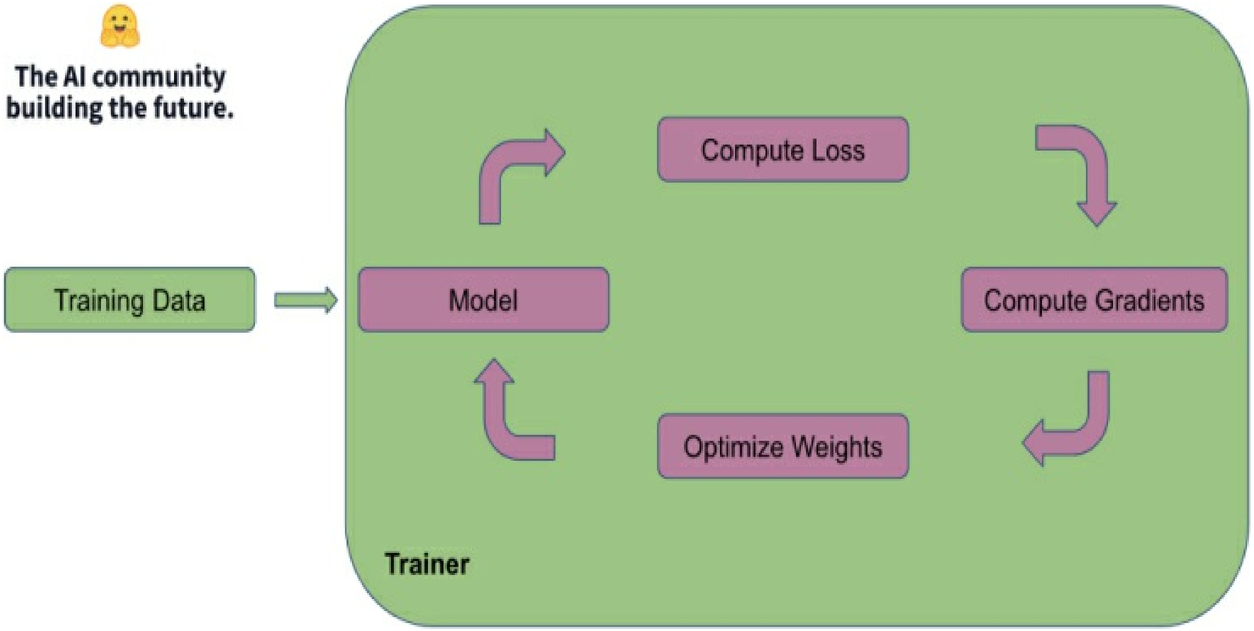


Figure 1.9 The Transformers package from Hugging Face provides a neat and clean interface for training and fine-tuning LLMs.

To recap, LLMs are pre-trained on large corpora and sometimes fine-tuned on smaller datasets for specific tasks. Recall that one of the factors behind the Transformer's effectiveness as a language model is that it is highly parallelizable, allowing for faster training and efficient processing of text. What really sets the Transformer apart from other deep learning architectures is its ability to capture long-range dependencies and relationships between tokens using attention. In other words, attention is a crucial component of Transformer-based LLMs, and it enables them to effectively retain information between training loops and tasks (i.e. transfer learning), while being able to process lengthy swatches of text with ease.

Attention is attributed for being the most responsible for helping LLMs learn (or at least recognize) internal world models and human-identifiable rules. A

Stanford study in 2019 showed that certain attention calculations in BERT corresponded to linguistic notions of syntax and grammar rules. For example, they noticed that BERT was able to notice direct objects of verbs, determiners of nouns, and objects of prepositions with remarkably high accuracy from only its pre-training. These relationships are presented visually in Figure 1.10.

There is research that explores what other kinds of “rules” LLMs are able to learn simply by pre-training and fine-tuning. One example is a series of experiments led by researchers at Harvard that explored an LLM’s ability to learn a set of rules to a synthetic task like the game of Othello (Figure 1.11). They found evidence that an LLM was able to understand the rules of the game simply by training on historical move data.

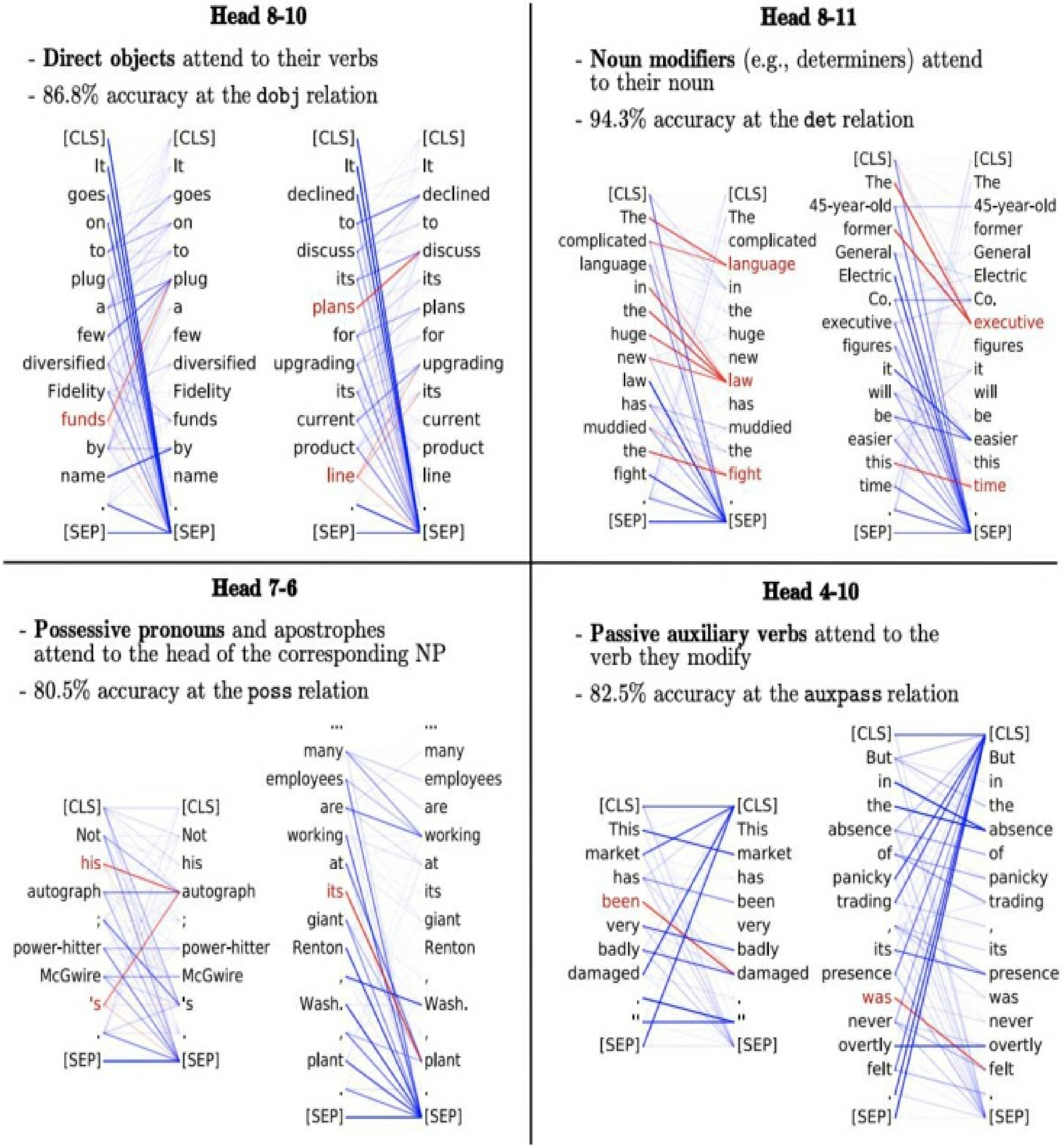

Head 8-10
- Direct objects attend to their verbs
- 86.8% accuracy at the dobj relation
[CLS] It goes on to plug a few diversified Fidelity funds by name . [SEP]
[CLS] It declined to discuss its plans for upgrading its current product line . [SEP]
Head 8-11
- Noun modifiers (e.g., determiners) attend to their noun
- 94.3% accuracy at the det relation
[CLS] The complicated language in the huge new law has muddied the fight . [SEP]
[CLS] The 45-year-old former General Electric Co. executive figures it will be easier this time . [SEP]
Head 7-6
- Possessive pronouns and apostrophes attend to the head of the corresponding NP
- 80.5% accuracy at the poss relation
[CLS] Not his autograph ; power-hitter McGwire 's . [SEP]
... many employees are working at its giant Renton , Wash. , plant . [SEP]
Head 4-10
- Passive auxiliary verbs attend to the verb they modify
- 82.5% accuracy at the auxpass relation
[CLS] This market has been very badly damaged . " [SEP]
[CLS] But in the absence of panicky trading , its presence was never overtly felt . [SEP]

Figure 1.10 Research has probed into LLMs to uncover that they seem to be recognizing grammatical rules even when they were never explicitly told these rules.

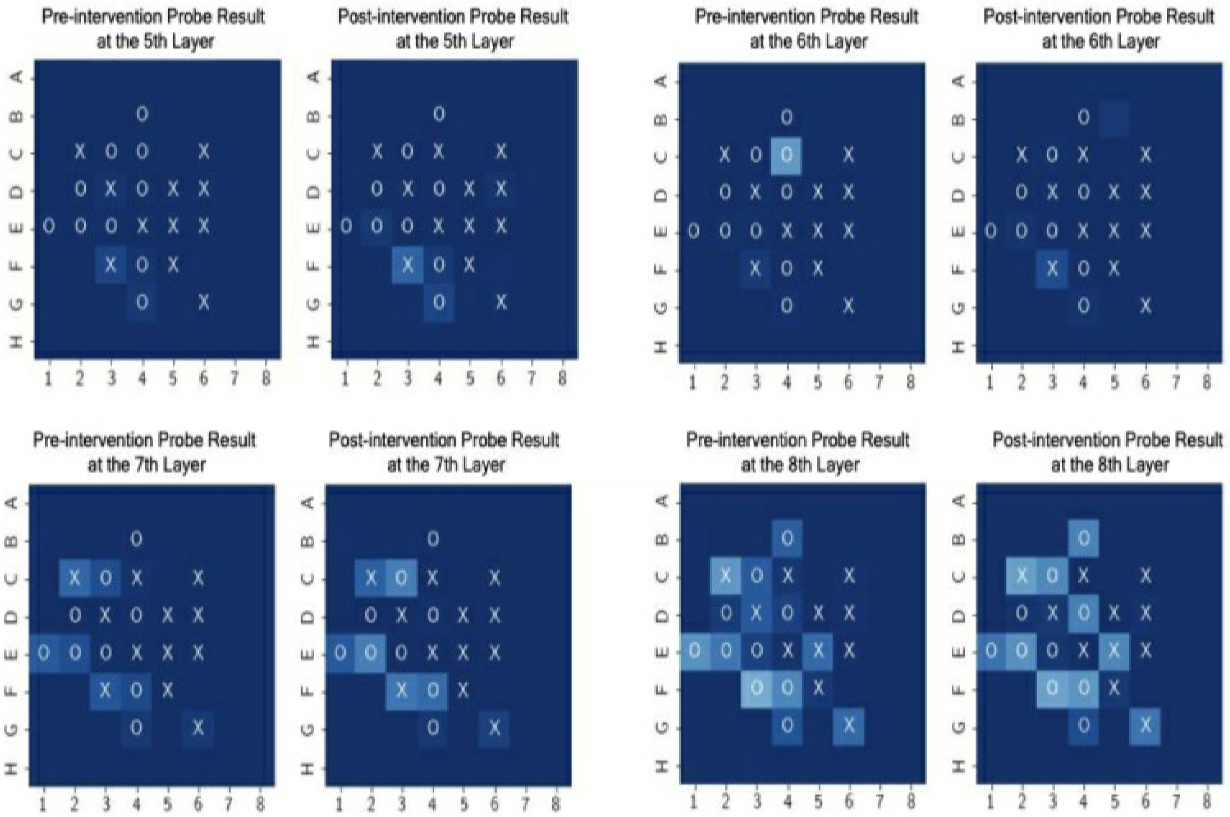


Figure 1.11 LLMs may be able to learn all kinds of things about the world, whether it be the rules and strategy of a game or the rules of human language.

For any LLM to learn any kind of rule, however, it has to convert what we perceive as text into something machine readable. This is done through a process called embedding.

Embeddings

Embeddings are the mathematical representations of words, phrases, or tokens in a large-dimensional space. In NLP, embeddings are used to represent the words, phrases, or tokens in a way that captures their semantic meaning and relationships with other words. There are several types of embeddings, including position embeddings, which encode the position of a token in a sentence, and token embeddings, which encode the semantic meaning of a token (Figure 1.12).

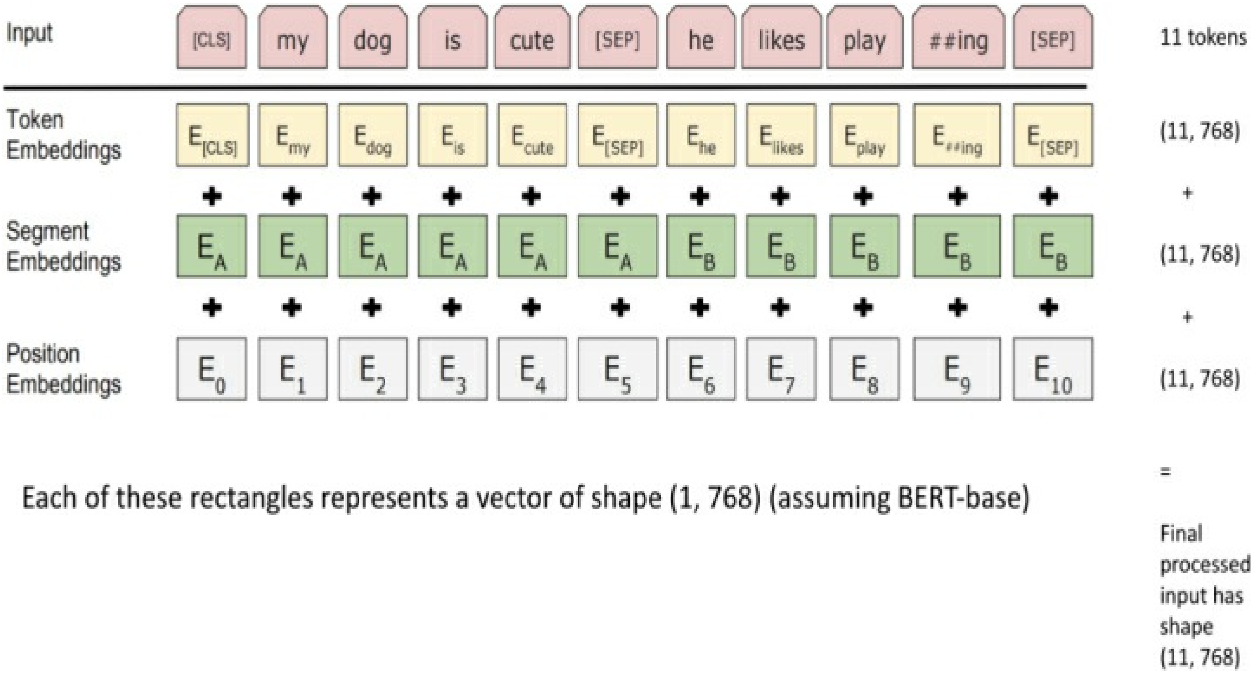


Figure 1.12 An example of how BERT uses three layers of embedding for a given piece of text. Once the text is tokenized, each token is given an embedding and then the values are added up, so each token ends up with an initial embedding before any attention is calculated. We won't focus too much on the individual layers of LLM embeddings in this text unless they serve a more practical purpose but it is good to know about some of these parts and how they look under the hood!

LLMs learn different embeddings for tokens based on their pre-training and can further update these embeddings during fine-tuning.

Tokenization

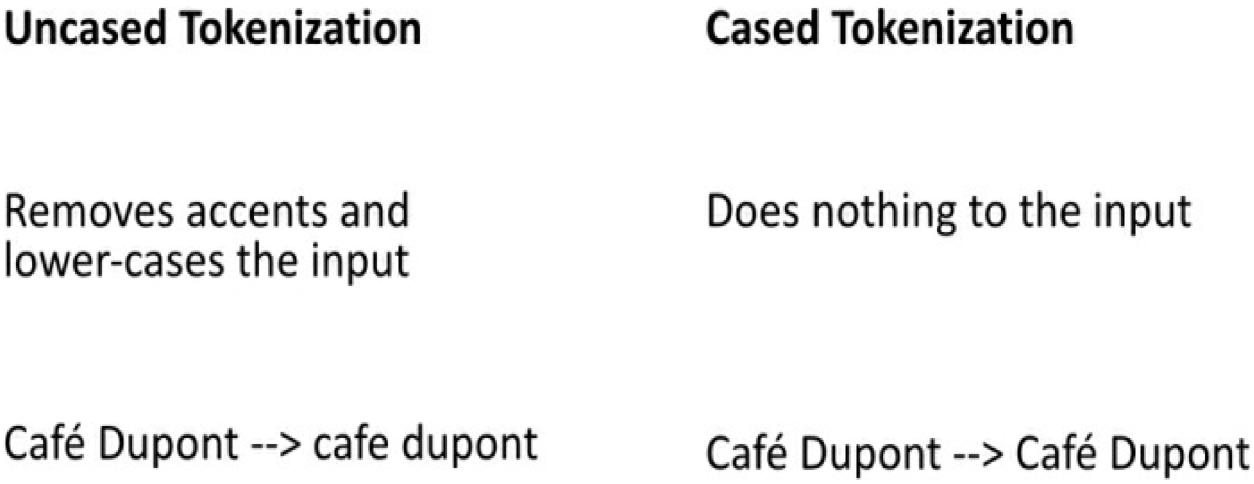
Tokenization, as mentioned previously, involves breaking text down into the smallest unit of understanding - tokens. These tokens are the pieces of information that are embedded into semantic meaning and act as inputs to the attention calculations which leads to ... well, the LLM actually learning and working. Tokens make up an LLMs static vocabulary and don't always represent entire words. Tokens can represent punctuation, individual characters, or even a sub-word if a word is not known to the LLM. Nearly all LLMs also have *special tokens* that have specific meaning to the model. For example, the BERT model has a few special tokens including the **[CLS]** token which BERT automatically injects as the first token of every input and is meant to represent an encoded semantic meaning for the entire input sequence.

Readers may be familiar with techniques like stop words removal, stemming, and truncation which are used in traditional NLP. These techniques are not used nor are they necessary for LLMs. LLMs are designed to handle the inherent complexity and variability of human language, including the usage of stop words like “the” and “an” and variations in word forms like tenses and misspellings. Altering the input text to an LLM using these techniques could potentially harm the performance of the model by reducing the contextual information and altering the original meaning of the text.

Tokenization can also involve several preprocessing steps like **casing**, which refers to the capitalization of the tokens. There are two types of casing: uncased and cased. In uncased tokenization, all the tokens are lowercased and usually accents from letters are stripped, while in cased tokenization, the capitalization of the tokens is preserved. The choice of casing can impact the performance of the model, as capitalization can provide important information about the meaning of a token. An example of this can be found in Figure 1.13.

Note

It is worth mentioning that even the concept of casing has some bias to it depending on the model. To uncase a text - lowercasing and stripping of accents - is a pretty Western style preprocessing step. I myself speak Turkish and know that the umlaut (e.g. the Ö in my last name) matters and can actually help the LLM understand the word being said. Any language model that has not been sufficiently trained on diverse corpora may have trouble parsing and utilizing these bits of context.



Uncased Tokenization	Cased Tokenization
Removes accents and lower-cases the input	Does nothing to the input
Café Dupont --> cafe dupont	Café Dupont --> Café Dupont

Figure 1.13 The choice of uncased versus cased tokenization depends on the

task. Simple tasks like text classification usually prefer uncased tokenization while tasks that derive meaning from case like Named Entity Recognition prefer a cased tokenization.

Figure 1.14 shows an example of tokenization, and in particular, an example of how LLMs tend to handle Out of Vocabulary (OOV) phrases. OOV phrases are simply phrases/words that the LLM doesn't recognize as a token and has to split up into smaller sub-words. For example, my name (Sinan) is not a token in most LLMs (story of my life) so in BERT, the tokenization scheme will split my name up into two tokens (assuming uncased tokenization):

sin - the first part of my name

##an - a special sub-word token that is different from the word "an" and is used only as a means to split up unknown words

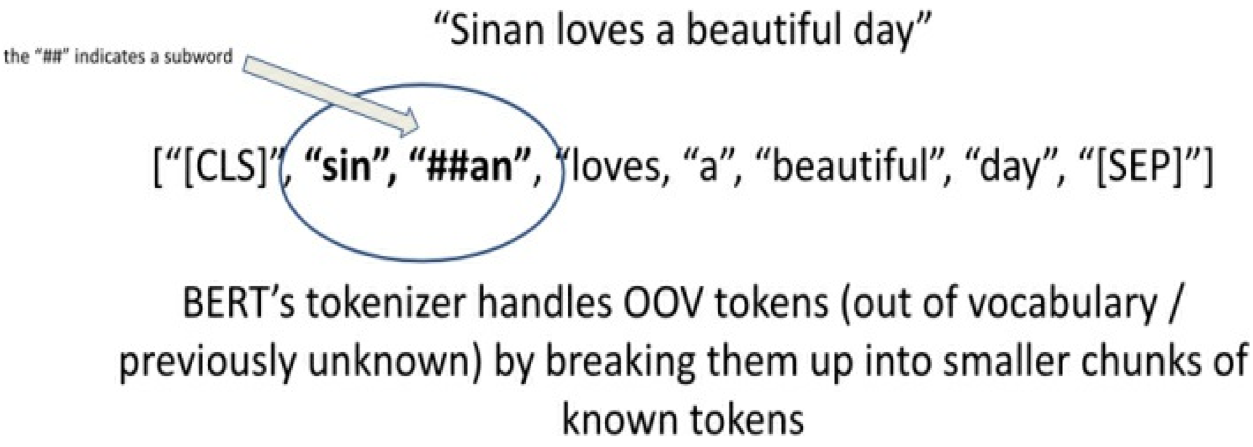

"Sinan loves a beautiful day"
the "##" indicates a subword
["[CLS]", "sin", "##an", "loves, "a", "beautiful", "day", "[SEP]"]
BERT's tokenizer handles OOV tokens (out of vocabulary / previously unknown) by breaking them up into smaller chunks of known tokens

Figure 1.14 Any LLM has to deal with words they've never seen before.

How an LLM tokenizes text can matter if we care about the token limit of an

LLM.

Some LLMs limit the number of tokens we can input at any one time so how an LLM tokenizes text can matter if we are trying to be mindful about this limit.

So far, we have talked a lot about language modeling - predicting missing/next tokens in a phrase, but modern LLMs also can also borrow from other fields of AI to make their models more performant and more importantly more **aligned** - meaning that the AI is performing in accordance with a human's expectation. Put another way, an aligned LLM has an objective that matches a human's objective.

Beyond Language Modeling—Alignment + RLHF

Alignment in language models refers to how well the model can respond to input prompts that match the user's expectations. Standard language models predict the next word based on the preceding context, but this can limit their usefulness for specific instructions or prompts. Researchers are coming up with scalable and performant ways of aligning language models to a user's intent. One such broad method of aligning language models is through the incorporation of reinforcement learning (RL) into the training loop.

RL with Human Feedback (RLHF) is a popular method of aligning pre-trained LLMs that uses human feedback to enhance their performance. It allows the LLM to learn from feedback on its own outputs from a relatively small, high-quality batch of human feedback, thereby overcoming some of the limitations of traditional supervised learning. RLHF has shown significant improvements in modern LLMs like ChatGPT. RLHF is one example of approaching alignment with RL, but there are other emerging approaches like RL with AI feedback (e.g. Constitutional AI).

Let’s take a look at some of the popular LLMs we’ll be using in this book.

Popular Modern LLMs

BERT, T5, and GPT are three popular LLMs developed by Google, Google, and OpenAI respectively. These models differ in their architecture pretty greatly even though they all share the Transformer as a common ancestor. Other widely used variants of LLMs in the Transformer family include RoBERTa, BART (which we saw earlier performing some text classification), and ELECTRA.

BERT

BERT (Figure 1.15) is an autoencoding model that uses attention to build a bidirectional representation of a sentence, making it ideal for sentence classification and token classification tasks.

BERT uses the encoder of the Transformer and ignores the decoder to become exceedingly good at processing/understanding massive amounts of text very quickly relative to other, slower LLMs that focus on generating text one token at a time. BERT-derived architectures, therefore, are best for working with and analyzing large corpora quickly when we don't need to write free text.

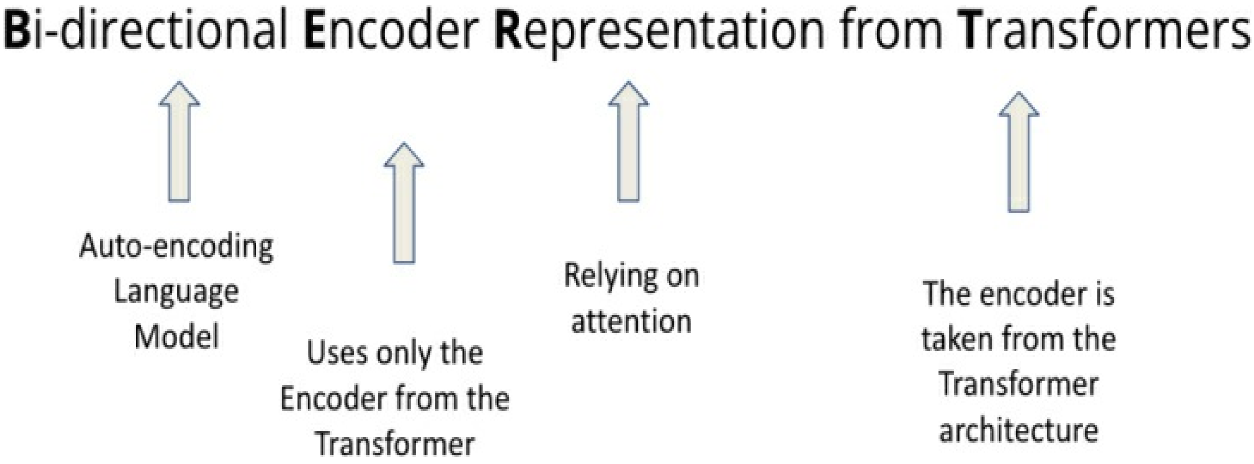

Bi-directional Encoder Representation from Transformers
Auto-encoding Language Model
Uses only the Encoder from the Transformer
Relying on attention
The encoder is taken from the Transformer architecture

Figure 1.15 BERT was one of the first LLMs and continues to be popular for many NLP tasks that involve fast processing of large amounts of text.

BERT itself doesn't classify text or summarize documents but it is often used as a pre-trained model for downstream NLP tasks. BERT has become a widely used and highly regarded LLM in the NLP community, paving the way for the development of even more advanced language models.

GPT-3 and ChatGPT

GPT (Figure 1.16), on the other hand, is an autoregressive model that uses attention to predict the next token in a sequence based on the previous tokens. The GPT family of algorithms (including ChatGPT and GPT-3) is primarily used for text generation and has been known for its ability to generate natural sounding human-like text.

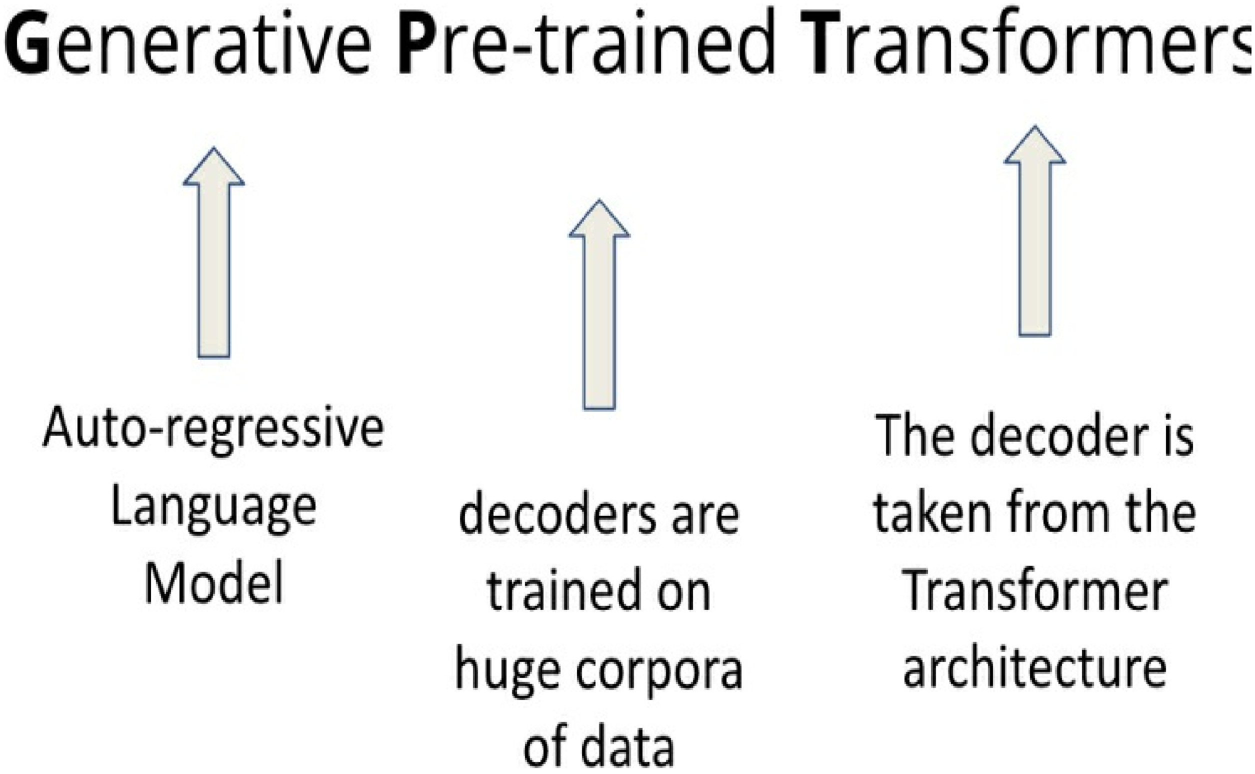

Generative Pre-trained Transformers
Auto-regressive Language Model
decoders are trained on huge corpora of data
The decoder is taken from the Transformer architecture

Figure 1.16 The GPT family of models excels at generating free text aligned with a user’s intent.

GPT relies on the decoder portion of the Transformer and ignores the encoder to become exceptionally good at generating text one token at a time. GPT-based models are best for generating text given a rather large context window. They can also be used to process/understand text as we will see in an upcoming chapter. GPT-derived architectures are ideal for applications that require the ability to freely write text.

T5

T5 is a pure encoder/decoder transformer model that was designed to perform several NLP tasks, from text classification to text summarization and generation, right off the shelf. It is one of the first popular models to be able to boast such a feat, in fact. Before T5, LLMs like BERT and GPT-2 generally had to be fine-tuned using labeled data before they could be relied on to perform such specific tasks.

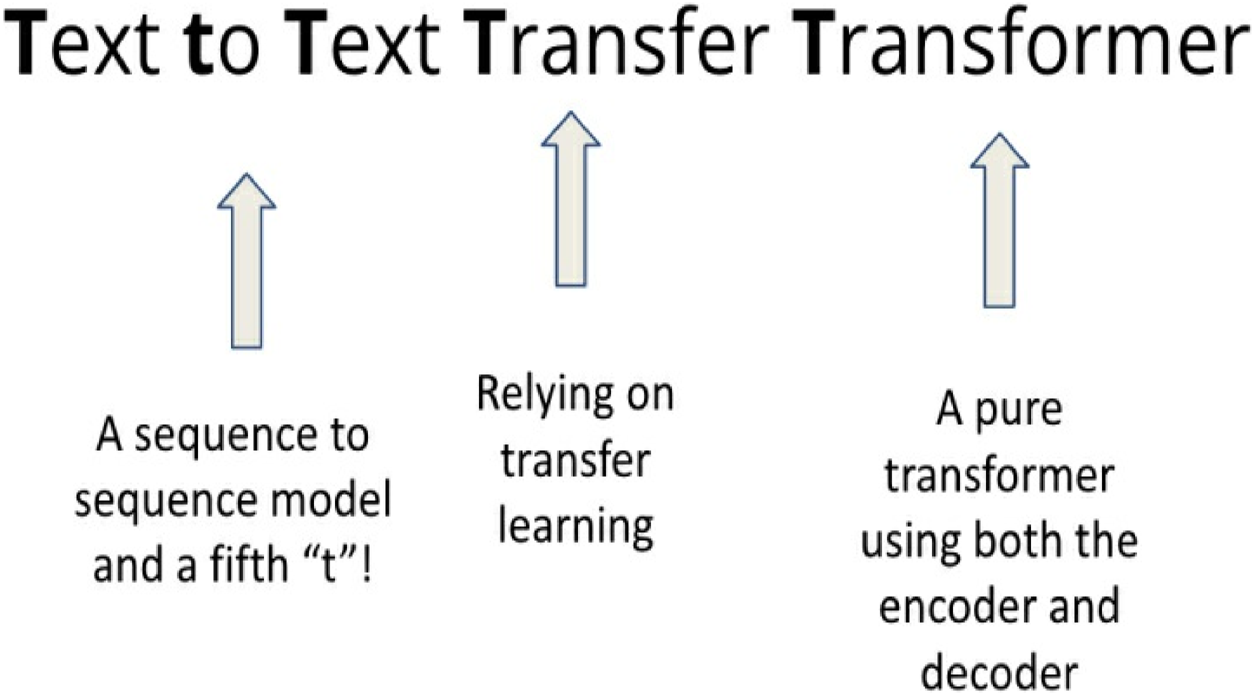

Text to Text Transfer Transformer
A sequence to sequence model and a fifth "t"!
Relying on transfer learning
A pure transformer using both the encoder and decoder

Figure 1.17 T5 was one of the first LLMs to show promise in solving multiple tasks at once without any fine-tuning.

T5 uses both the encoder and decoder of the Transformer to become highly versatile in both processing and generating text. T5-based models can perform a wide range of NLP tasks, from text classification to text generation, due to their ability to build representations of the input text using the encoder and generate text using the decoder (Figure 1.17). T5-derived architectures are ideal for applications that require both the ability to process and understand text and generate text freely.

T5’s ability to perform multiple tasks with no fine-tuning spurred the development of other versatile LLMs that can perform multiple tasks with efficiency and accuracy with little/no fine-tuning. GPT-3, released around the same time at T5, also boasted this ability.

These three LLMs are highly versatile and are used for various NLP tasks, such as text classification, text generation, machine translation, and sentiment analysis, among others. These three LLMs, along with flavors (variants) of them will be the main focus of this book and our applications.

Domain-Specific LLMs

Domain-specific LLMs are LLMs that are trained specifically in a particular subject area, such as biology or finance. Unlike general-purpose LLMs, these models are designed to understand the specific language and concepts used within the domain they were trained on.

One example of a domain-specific LLM is BioGPT (Figure 1.18); a domain-specific LLM that is pre-trained on large-scale biomedical literature. The model was developed by the AI healthcare company, Owkin, in collaboration with Hugging Face. The model is trained on a dataset of over 2 million biomedical research articles, making it highly effective for a wide range of biomedical NLP tasks such as named entity recognition, relationship extraction, and question-answering.

BioGPT, whose pre-training encoded biomedical knowledge and domain-specific jargon into the LLM, can be fine-tuned on smaller datasets, making it adaptable for specific biomedical tasks and reducing the need for large amounts of labeled data.

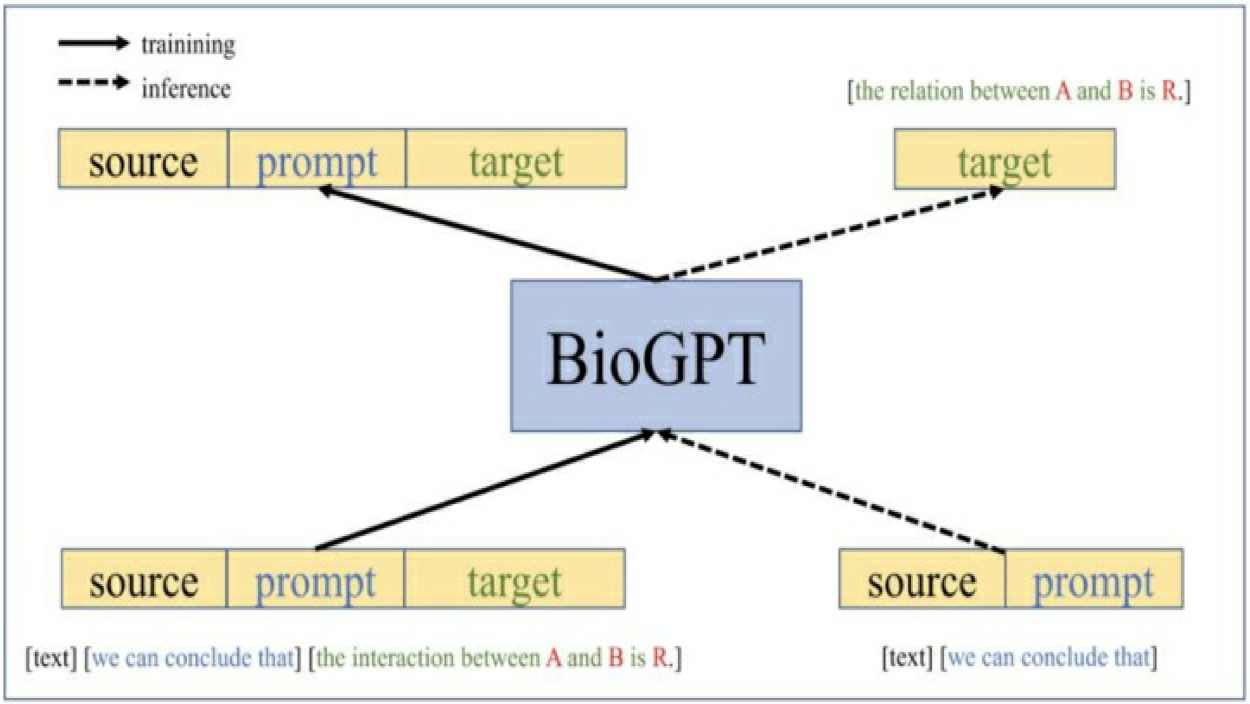

traininig
inference
[the relation between A and B is R.]
source
prompt
target
target
BioGPT
source
prompt
target
source
prompt
[text] [we can conclude that] [the interaction between A and B is R.]
[text] [we can conclude that]

Figure 1.18 BioGPT is a domain-specific Transformer model pre-trained on large-scale biomedical literature. BioGPT's success in the biomedical domain has inspired other domain-specific LLMs such as SciBERT and BlueBERT.

The advantage of using domain-specific LLMs lies in their training on a specific set of texts. This allows them to better understand the language and concepts used within their specific domain, leading to improved accuracy and fluency for NLP tasks that are contained within that domain. By comparison, general-purpose LLMs may struggle to handle the language and concepts used in a specific domain as effectively.

Applications of LLMs

As we’ve already seen, applications of LLMs vary widely and researchers continue to find novel applications of LLMs to this day. We will use LLMs in this book in generally three ways:

Using a pre-trained LLM’s underlying ability to process and generate text with no further fine-tuning as part of a larger architecture.

For example, creating an information retrieval system using a pre-trained BERT/GPT.

Fine-tuning a pre-trained LLM to perform a very specific task using Transfer Learning.

For example, fine-tuning T5 to create summaries of documents in a specific domain/industry.

Asking a pre-trained LLM to solve a task it was pre-trained to solve or could reasonably intuit.

For example, prompting GPT3 to write a blog post.

For example, prompting T5 to perform language translation..

These methods use LLMs in different ways and while all options take advantage of an LLM’s pre-training, only option 2 requires any fine-tuning.

Let’s take a look at some specific applications of LLMs.

Classical NLP Tasks

A vast majority of applications of LLMs are delivering state of the art results in very common NLP tasks like classification and translation. It’s not that we weren’t solving these tasks before Transformers and LLMs, it’s just that now developers and practioners can solve them with comparatively less labeled data (due to the efficient pre-training of the Transformer on huge corpora) and with a higher degree of accuracy.

Text Classification

The text classification task assigns a label to a given piece of text. This task is commonly used in sentiment analysis, where the goal is to classify a piece of text as positive, negative, or neutral, or in topic classification, where the goal is to classify a piece of text into one or more predefined categories. Models like BERT can be fine-tuned to perform classification with relatively little labeled data as seen in Figure 1.19.

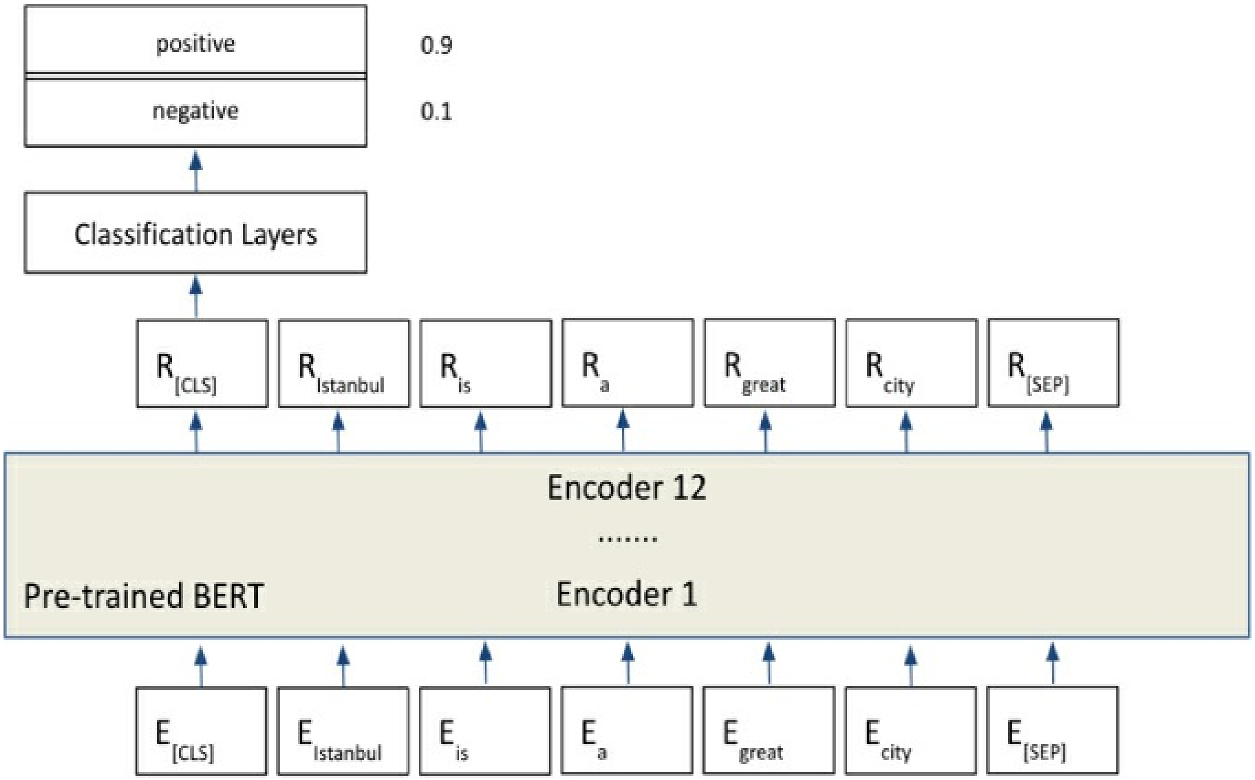


Figure 1.19 A peek at the architecture of using BERT to achieve fast and accurate text classification results. Classification layers usually act on that special [CLS] token that BERT uses to encode the semantic meaning of the entire input sequence.

Text classification remains one of the most globally recognizable and solvable NLP tasks because when it comes down to it, sometimes we just need to know whether this email is “spam” or not and get on with our days!

Translation Tasks

A harder and yet still classic NLP task is machine translation where the goal is to automatically translate text from one language to another while preserving meaning and context. Traditionally, this task is quite difficult because it involves having sufficient examples and domain knowledge of both languages to accurately gauge how well the model is doing but modern LLMs seem to have an easier time with this task again due to their pre-training and efficient attention calculations.

Human Language <> Human Language

One of the first applications of attention even before Transformers was for machine translation tasks where AI models were expected to translate from one human language to another. T5 was one of the first LLMs to tout the ability to perform multiple tasks off the shelf (Figure 1.20). One of these tasks was the ability to translate English into a few languages and back.

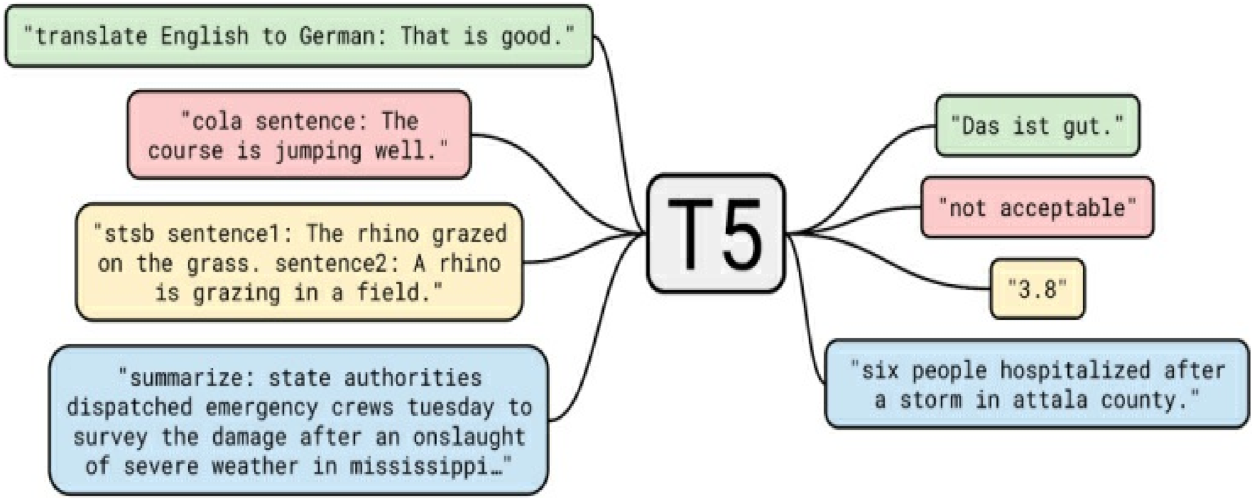
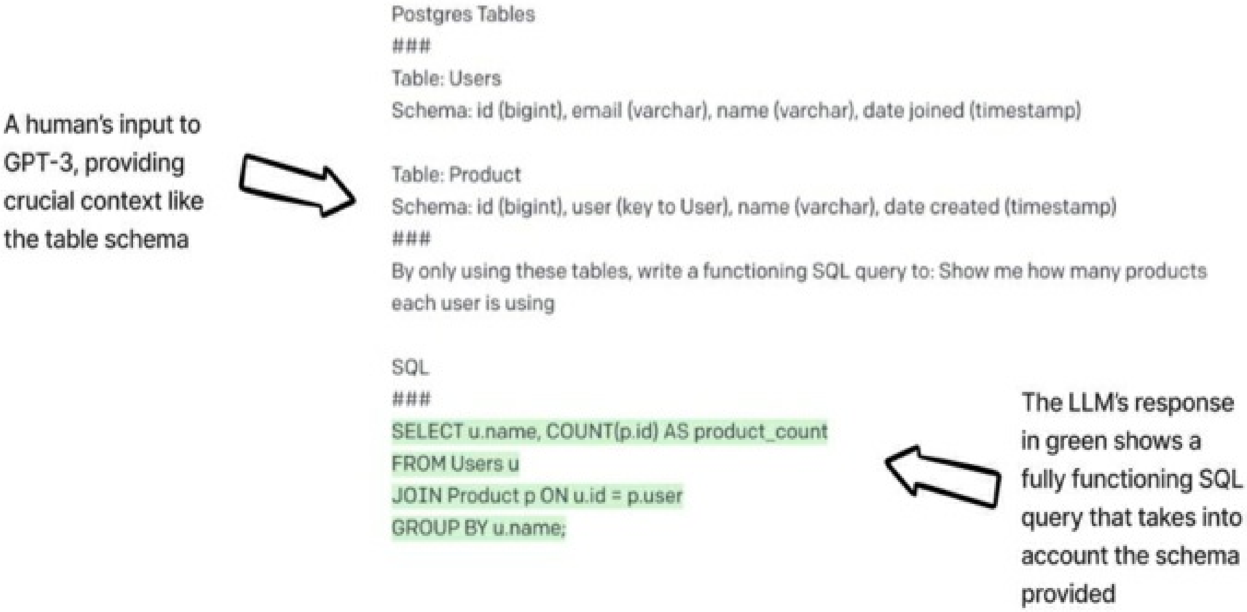

"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

Figure 1.20 T5 could perform many NLP tasks off the shelf, including grammar correction, summarization, and translation.

Since T5, language translation in LLMs has only gotten better and more diverse. Models like GPT-3 and the latest T5 models can translate between dozens of languages with relative ease. Of course this bumps up against one major known limitation of LLMs that they are mostly trained from an English-speaking/usually American point of view so most LLMs can handle English well and non-English languages, well, not as well.

SQL Generation

If we consider SQL as a language, then converting English to SQL is really not that different from converting English to French (Figure 1.21). Modern LLMs can already do this at a basic level off the shelf, but more advanced SQL queries often require some fine-tuning.



A human's input to GPT-3, providing crucial context like the table schema

```
Postgres Tables
###
Table: Users
Schema: id (bigint), email (varchar), name (varchar), date joined (timestamp)

Table: Product
Schema: id (bigint), user (key to User), name (varchar), date created (timestamp)
###
By only using these tables, write a functioning SQL query to: Show me how many products
each user is using

SQL
###
SELECT u.name, COUNT(p.id) AS product_count
FROM Users u
JOIN Product p ON u.id = p.user
GROUP BY u.name;
```

The LLM's response in green shows a fully functioning SQL query that takes into account the schema provided

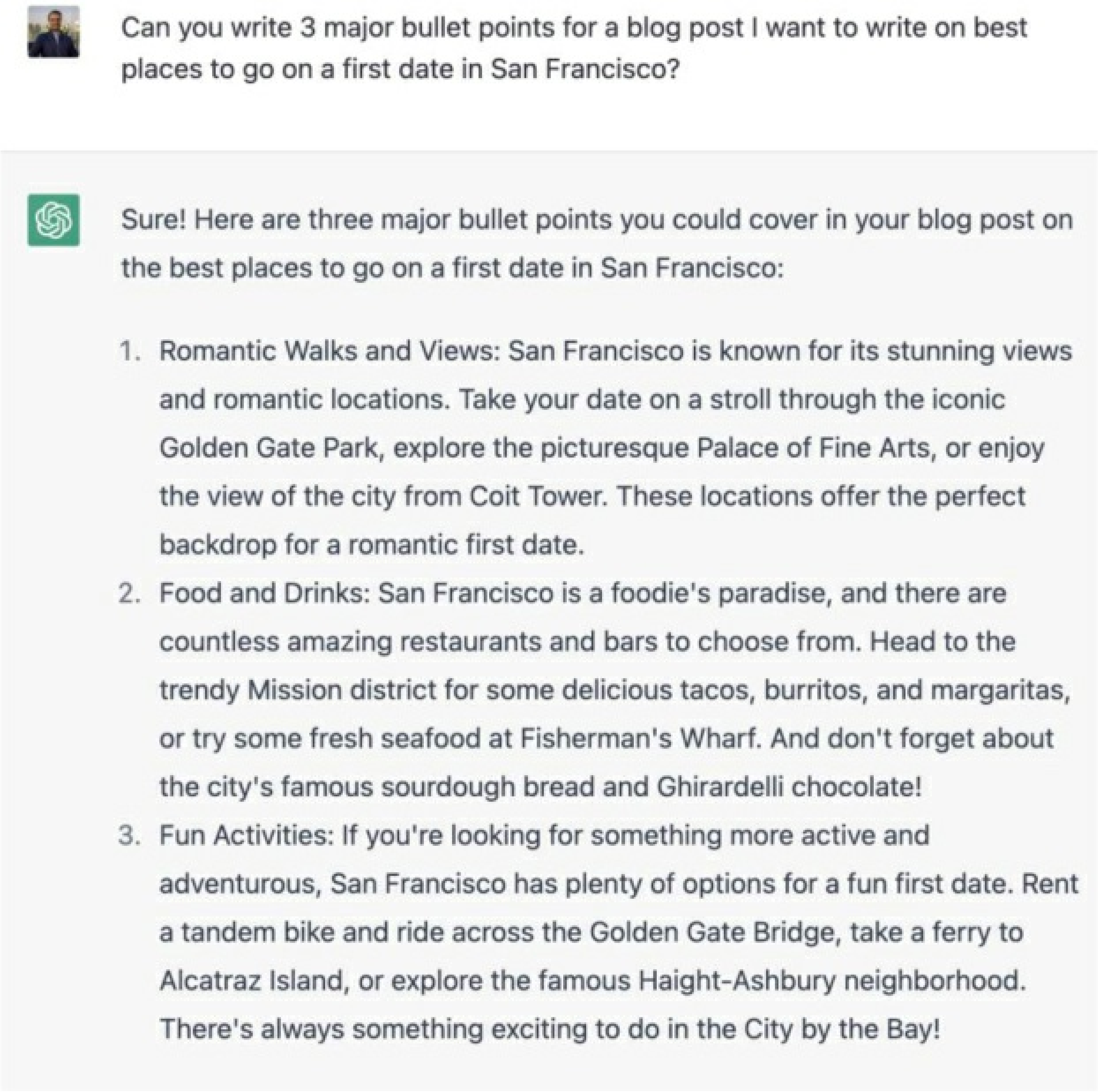
Figure 1.21 Using GPT-3 to generate functioning SQL code from an (albeit simple) Postgres schema

If we expand our thinking of what can be considered a “translation” then a lot of new opportunities lie ahead of us. For example, what if we wanted to “translate” between English and a series of wavelengths that a brain might interpret and execute as motor functions. I’m not a neuro-scientist or anything, but that seems like a fascinating area of research!

Free Text Generation

What first caught the world’s eye in terms of modern LLMs like ChatGPT was their ability to freely write blogs, emails, and even academic papers. This notion of text generation is why many LLMs are affectionately referred to as “Generative AI”, although that term is a bit reductive and imprecise. I will not often use the term “Generative AI” as the specific word “generative” has its own meaning in machine learning as the analogous way of learning to a “discriminative” model. For more on that, check out my first book: The Principles of Data Science)

We could for example prompt (ask) ChatGPT to help plan out a blog post like in Figure 1.22. Even if you don’t agree with the results, this can help humans with the “tabula rasa” problem and give us something to at least edit and start from rather than staring at a blank page for too long.



Can you write 3 major bullet points for a blog post I want to write on best places to go on a first date in San Francisco?

Sure! Here are three major bullet points you could cover in your blog post on the best places to go on a first date in San Francisco:

1. Romantic Walks and Views: San Francisco is known for its stunning views and romantic locations. Take your date on a stroll through the iconic Golden Gate Park, explore the picturesque Palace of Fine Arts, or enjoy the view of the city from Coit Tower. These locations offer the perfect backdrop for a romantic first date.
2. Food and Drinks: San Francisco is a foodie's paradise, and there are countless amazing restaurants and bars to choose from. Head to the trendy Mission district for some delicious tacos, burritos, and margaritas, or try some fresh seafood at Fisherman's Wharf. And don't forget about the city's famous sourdough bread and Ghirardelli chocolate!
3. Fun Activities: If you're looking for something more active and adventurous, San Francisco has plenty of options for a fun first date. Rent a tandem bike and ride across the Golden Gate Bridge, take a ferry to Alcatraz Island, or explore the famous Haight-Ashbury neighborhood. There's always something exciting to do in the City by the Bay!

Figure 1.22 ChatGPT can help ideate, scaffold, and even write entire blog posts

---

Note

I would be remiss if I didn't mention the controversy that LLMs like this can cause at the academic level. Just because

> an LLM can write entire blogs or even essays doesn't mean we should let them. Just like how the internet caused some to believe that we'd never need books again, some argue that ChatGPT means that we'll never need to write anything again. As long as institutions are aware of how to use this technology and proper regulations/rules are put in place, students and teachers alike can use ChatGPT and other text-generation-focused AIs safely and ethically.

We will be using ChatGPT to solve a few tasks in this book. We will rely on ChatGPT's ability to contextualize information in its context window and freely write back (usually) accurate responses. We will mostly be interacting with ChatGPT through the Playground and the API provided by OpenAI as this model is not open source.

Information Retrieval / Neural Semantic Search

LLMs encode information directly into their parameters via pre-training and fine-tuning but keeping them up to date with new information is tricky. We either have to further fine-tune the model on new data or run the pre-training steps again from scratch. To dynamically keep information fresh, we will architect our own information retrieval system with a vector database (don't worry we will go into more details on all of this in the next chapter). Figure 1.23 shows an outline of the architecture we will build.

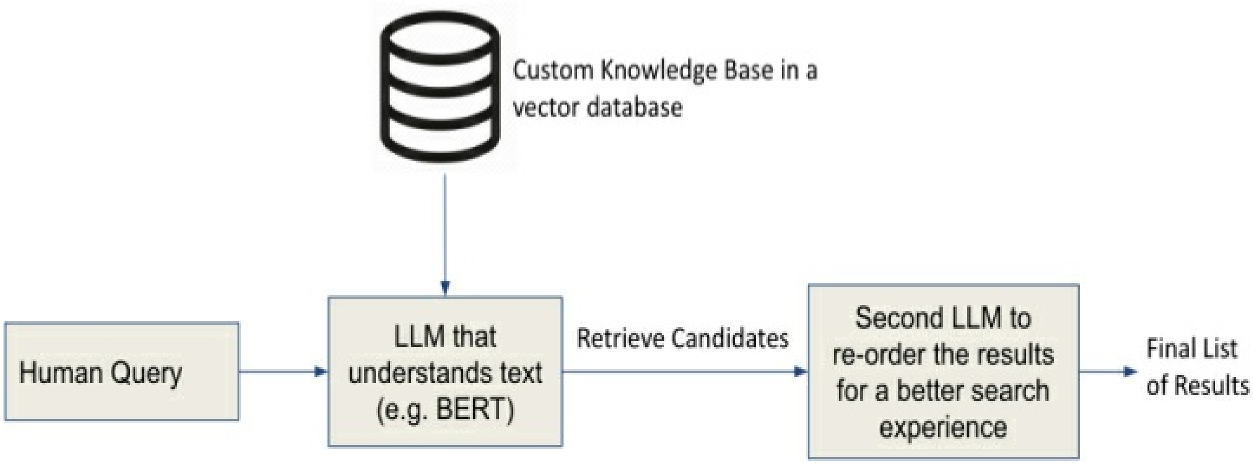

Custom Knowledge Base in a vector database
Human Query
LLM that understands text (e.g. BERT)
Retrieve Candidates
Second LLM to re-order the results for a better search experience
Final List of Results

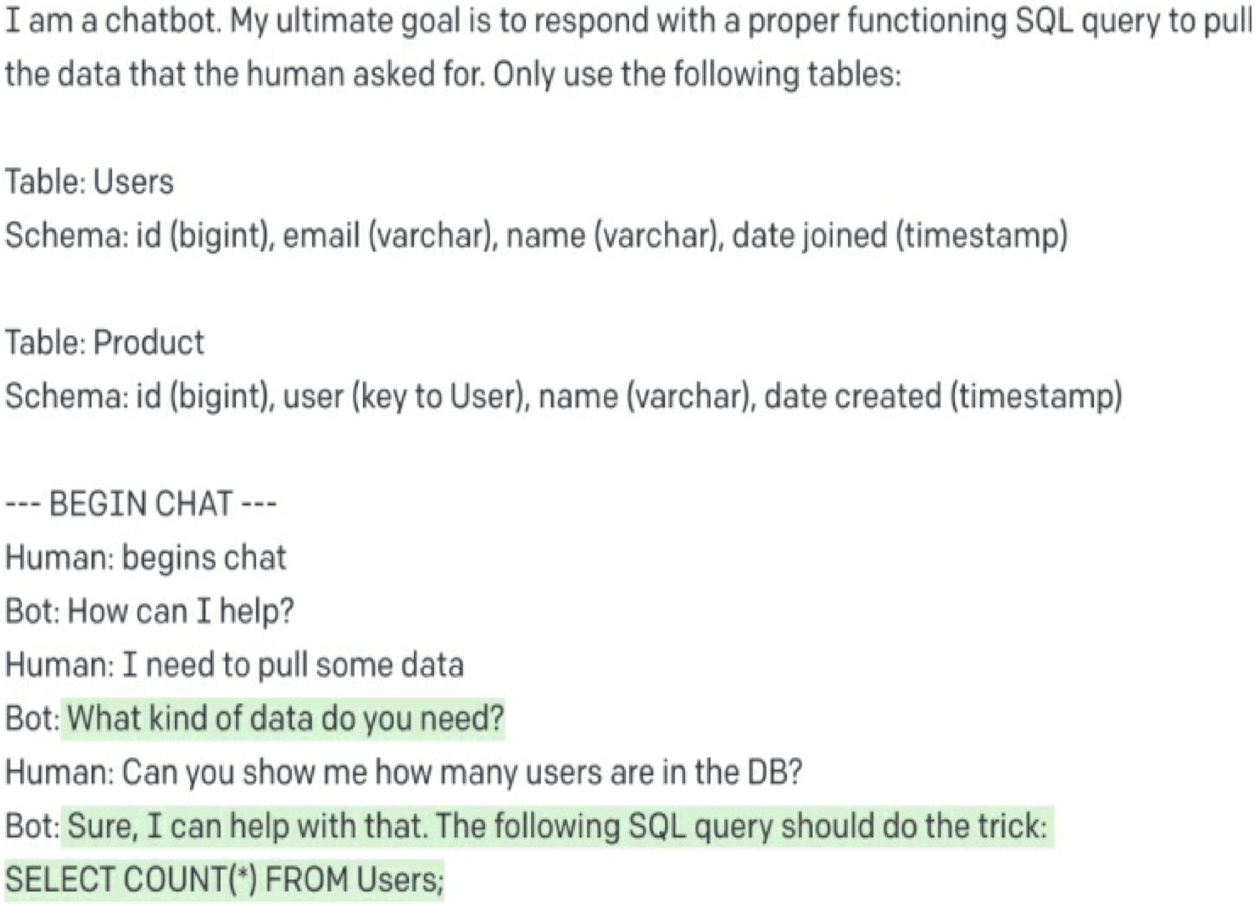
Figure 1.23 Our neural semantic search system will be able to take in new information dynamically and be able to retrieve relevant documents quickly and accurately given a user's query using LLMs.

We will then add onto this system by building a ChatGPT-based chatbot to conversationally answer questions from our users.

Chatbots

Everyone loves a good chatbot, right? Well, whether you love them or hate them, LLMs' capacity for holding a conversation is evident through systems like ChatGPT and even GPT-3 (as seen in Figure 1.24). The way we architect chatbots using LLMs will be quite different from the traditional way of designing chatbots through intents, entities, and tree-based conversation flows. These concepts will be replaced by system prompts, context, and personas – all of which we will dive into in the coming chapters.

We have our work cut out for us. I'm excited to be on this journey with you and I'm excited to get started!



I am a chatbot. My ultimate goal is to respond with a proper functioning SQL query to pull the data that the human asked for. Only use the following tables:

Table: Users
Schema: id (bigint), email (varchar), name (varchar), date joined (timestamp)

Table: Product
Schema: id (bigint), user (key to User), name (varchar), date created (timestamp)

--- BEGIN CHAT ---
Human: begins chat
Bot: How can I help?
Human: I need to pull some data
Bot: What kind of data do you need?
Human: Can you show me how many users are in the DB?
Bot: Sure, I can help with that. The following SQL query should do the trick:
SELECT COUNT(*) FROM Users;

Figure 1.24 ChatGPT isn’t the only LLM that can hold a conversation. We can use GPT-3 to construct a simple conversational chatbot. The text highlighted in green represents GPT-3’s output. Note that before the chat even begins, I inject context to GPT-3 that would not be shown to the end-user but GPT-3 needs to provide accurate responses.

Summary

LLMs are advanced AI models that have revolutionized the field of NLP. LLMs are highly versatile and are used for a variety of NLP tasks, including text classification, text generation, and machine translation. They are pre-trained on large corpora of text data and can then be fine-tuned for specific tasks.

Using LLMs in this fashion has become a standard step in the development of NLP models. In our first case study, we will explore the process of launching an application with proprietary models like GPT-3 and ChatGPT. We will get a hands-on look at the practical aspects of using LLMs for real-world NLP tasks, from model selection and fine-tuning to deployment and maintenance.

2

Semantic Search with LLMs

Introduction

In the last chapter, we explored the inner workings of language models and the impact that modern LLMs have had on NLP tasks like text classification, generation, and machine translation. There is another powerful application of LLMs that has been gaining traction in recent years: semantic search.

Now you might be thinking that it’s time to finally learn the best ways to talk to ChatGPT and GPT-4 to get the optimal results, and we will do that as early as the next chapter, I promise. In the meantime, I want to show you what else we can build on top of this novel transformer architecture. While text-to-text generative models like GPT are extremely impressive in their own right, one of the most versatile solutions that AI companies offer is the ability to generate text embeddings based on powerful LLMs.

Text embeddings are a way to represent words or phrases as vectors in a high-dimensional space based on their contextual meaning within a corpus of text data. The idea is that if two phrases are similar (we will explore that word in more detail later on in this chapter) then the vectors that represent those phrases should be close together and vice versa. Figure 2.1 shows an example of a simple search algorithm. When a user searches for an item to buy – say a magic the gathering trading card they might simply search for “a vintage magic card”. The system should then embed the query such that if two text embeddings that are near each other should indicate that the phrases that were used to generate them are similar.

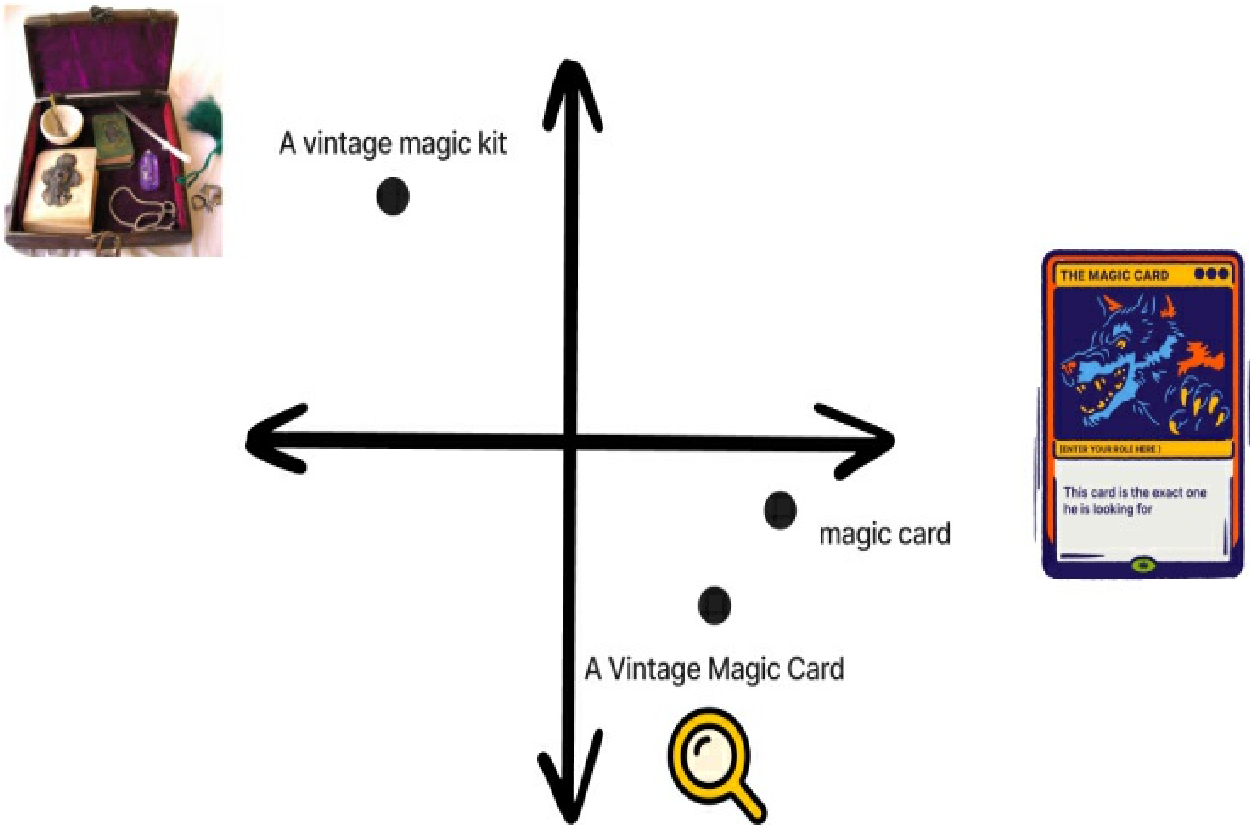


Figure 2.1 *Vectors that represent similar phrases should be close together and those that represent dissimilar phrases should be far apart. In this case, if a user wants a trading card they might ask for “a vintage magic card”. A proper semantic search system should embed the query in such a way that it ends up near relevant results (like “magic card”) and far apart from non relevant items (like “a vintage magic kit”) even if they share certain keywords.*

This map from text to vectors can be thought of as a kind of hash with meaning. We can’t really reverse vectors back to text but rather they are a representation of the text that has the added benefit of carrying the ability to compare points while in their encoded state.

LLM-enabled text embeddings allow us to capture the semantic value of words and phrases beyond just their surface-level syntax or spelling. We can rely on the pre-training and fine-tuning of LLMs to build virtually unlimited applications on top of them by leveraging this rich source of information about language use.

This chapter introduces us to the world of semantic search using LLMs to explore how they can be used to create powerful tools for information retrieval and analysis. In the next chapter, we will build a chatbot on top of GPT-4 that leverages a fully realized semantic search system that we will build in this chapter.

Without further ado, let’s get right into it, shall we?

The Task

A traditional search engine would generally take what you type in and then give you a bunch of links to websites or items that contain those words or permutations of the characters that you typed in. So if you typed in “Vintage Magic the Gathering Cards” on a marketplace, you would get items with a title/description that contains combinations of those words. That’s a pretty standard way to search, but it’s not always the best way. For example I might get vintage magic sets to help me learn how to pull a rabbit out of a hat. Fun but not what I asked for.

The terms you input into a search engine may not always align with the *exact* words used in the items you want to see. It could be that the words in the query are too general, resulting in a slew of unrelated findings. This issue often extends beyond just differing words in the results; the same words might carry different meanings than what was searched for. This is where semantic search comes into play, as exemplified by the earlier-mentioned Magic: The Gathering cards scenario.

Asymmetric Semantic Search

A **semantic search** system can understand the meaning and context of your search query and match it against the meaning and context of the documents that are available to retrieve. This kind of system can find relevant results in a database without having to rely on exact keyword or n-gram matching but rather rely on a pre-trained LLM to understand the nuance of the query and the documents (Figure 2.2).

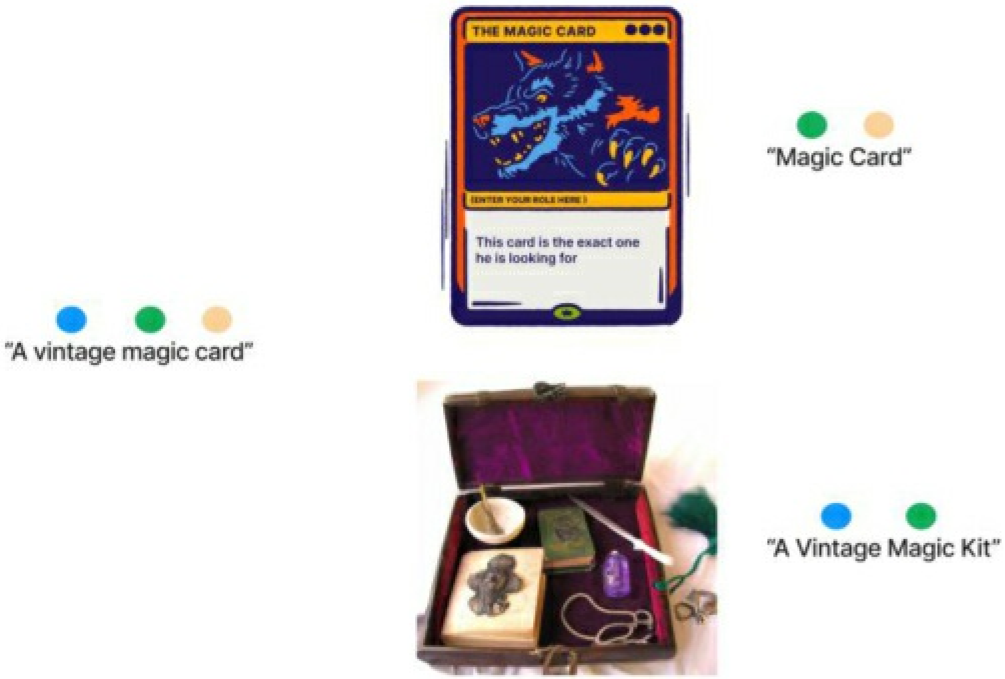


Figure 2.2 *A traditional keyword-based search might rank a vintage magic kit with the same weight as the item we actually want whereas a semantic search system can understand the actual concept we are searching for*

The **asymmetric** part of asymmetric semantic search refers to the fact that there is generally an imbalance between the semantic information (basically the size) of the input query and the documents/information that the search system has to retrieve. For example, the search system is trying to match “magic the gathering card” to paragraphs of item descriptions on a marketplace. The four-word search query has much less information than the paragraphs but nonetheless it is what we are comparing.

Asymmetric semantic search systems can get very accurate and relevant search results, even if you don’t use the exact right words in your search. They rely on the learnings of LLMs rather than the user being able to know exactly what needle to search for in the haystack.

I am of course, vastly oversimplifying the traditional method. There are many ways to make them more performant without switching to a more complex LLM approach and pure semantic search systems are not always the answer. They are not simply “the better way to do search”. Semantic algorithms have their own deficiencies like:

They can be overly sensitive to small variations in text, such as differences in capitalization or punctuation.

They struggle with nuanced concepts, such as sarcasm or irony that rely on localized cultural knowledge.

They can be more computationally expensive to implement and maintain than the traditional method, especially when launching a home-grown system with many open-source components.

Semantic search systems can be a valuable tool in certain contexts, so let’s jump right into how we will architect our solution.

Solution Overview

The general flow of our asymmetric semantic search system will follow these steps:

PART I - Ingesting documents (Figure 2.3)

Collect documents for embedding

Create text embeddings to encode semantic information

Store embeddings in a database for later retrieval given a query

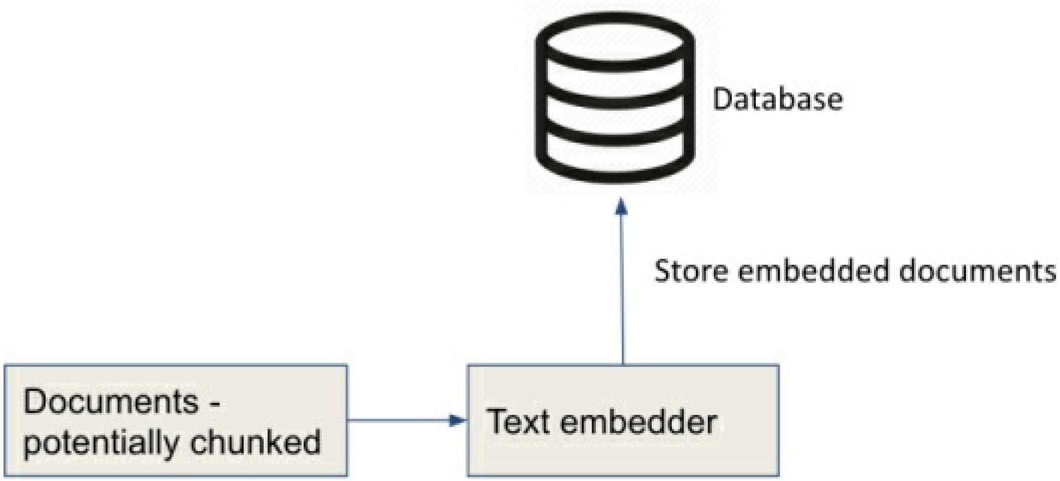


Figure 2.3 *Zooming in on Part I, storing documents will consist of doing some pre-processing on our documents, embedding them, and then storing them in some database*

PART II - Retrieving documents (Figure 2.4)

User has a query which may be pre-processed and cleaned

Retrieve candidate documents

Re-rank the candidate documents if necessary

Return the final search results

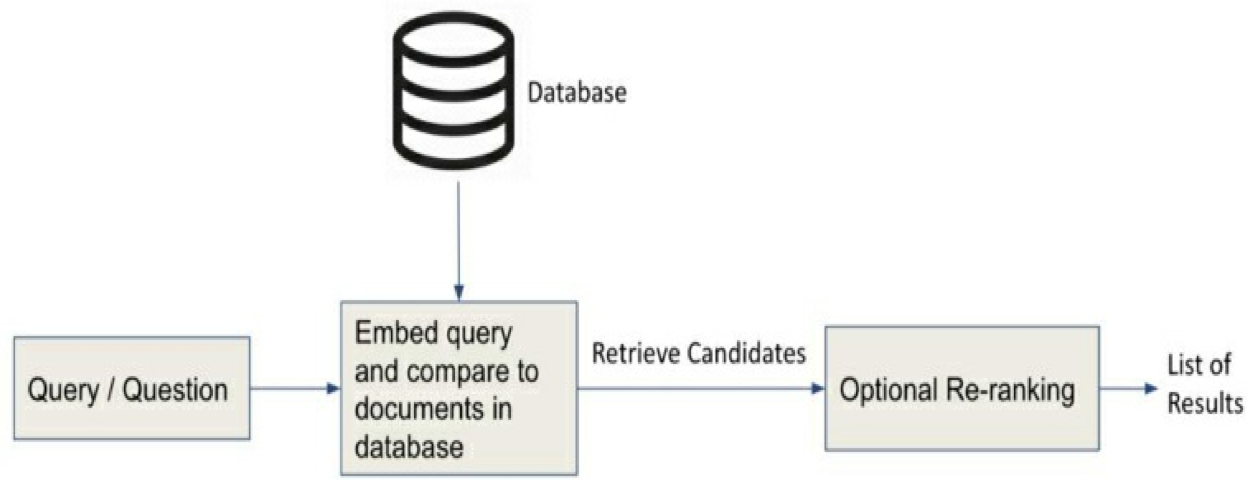


Figure 2.4 *Zooming in on Part II, when retrieving documents we will have to embed our query using the same embedding scheme as we used for the documents and then compare them against the previously stored documents and return the best (closest) document*

The Components

Let's go over each of our components in more detail to understand the choices we're making and what considerations we need to take into account.

Text Embedder

As we now know, at the heart of any semantic search system is the text embedder. This is the component that takes in a text document, or a single word or phrase, and converts it into a vector. The vector is unique to that text and should capture the contextual meaning of the phrase.

The choice of the text embedder is critical as it determines the quality of the vector representation of the text. We have many options in how we vectorize with LLMs, both open and closed source. To get off of the ground quicker, we are going to use OpenAI’s closed-source “Embeddings” product. In a later section, I’ll go over some open-source options.

OpenAI’s “Embeddings” is a powerful tool that can quickly provide high-quality vectors, but it is a closed-source product, which means we have limited control over its implementation and potential biases. It’s important to keep in mind that when using closed-source products, we may not have access to the underlying algorithms, which can make it difficult to troubleshoot any issues that may arise.

What makes pieces of text “similar”

Once we convert our text into vectors, we have to find a mathematical representation of figuring out if pieces of text are “similar” or not. Cosine similarity is a way to measure how similar two things are. It looks at the angle between two vectors and gives a score based on how close they are in direction. If the vectors point in exactly the same direction, the cosine similarity is 1. If they’re perpendicular (90 degrees apart), it’s 0. And if they point in opposite directions, it’s -1. The size of the vectors doesn’t matter, only their orientation does.

Figure 2.5 shows how the cosine similarity would help us retrieve documents given a query.

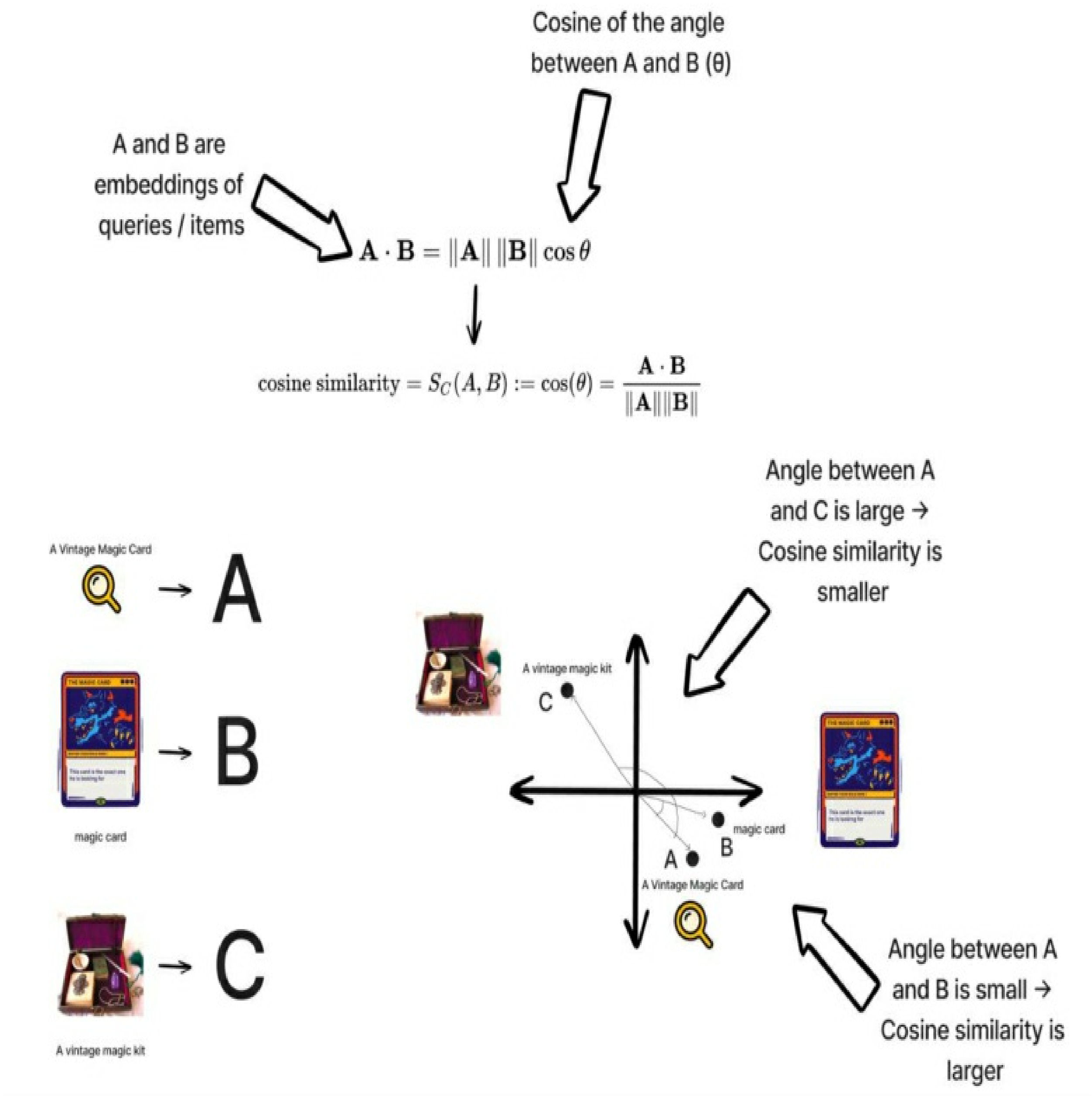

Cosine of the angle between A and B (θ)
A and B are embeddings of queries / items
$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \, \|\mathbf{B}\| \cos\theta$
cosine similarity $= S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$
A Vintage Magic Card
A
B
magic card
C
A vintage magic kit
Angle between A and C is large → Cosine similarity is smaller
A vintage magic kit
C
magic card
B
A
A Vintage Magic Card
Angle between A and B is small → Cosine similarity is larger

Figure 2.5 *In an ideal semantic search scenario, the Cosine Similarity (formula given at the top) gives us a computationally efficient way to compare pieces of text at scale, given that embeddings are tuned to place semantically similar pieces of text near each other (bottom). We start by embedding all items – including the query (bottom left) and then checking the angle between them. The smaller the angle, the larger the cosine similarity*

(bottom right)

We could also turn to other similarity metrics like the dot product or the Euclidean distance but OpenAI embeddings have a special property. The magnitudes (lengths) of their vectors are normalized to length 1, which basically means that we benefit mathematically on two fronts:

Cosine similarity is identical to the dot product

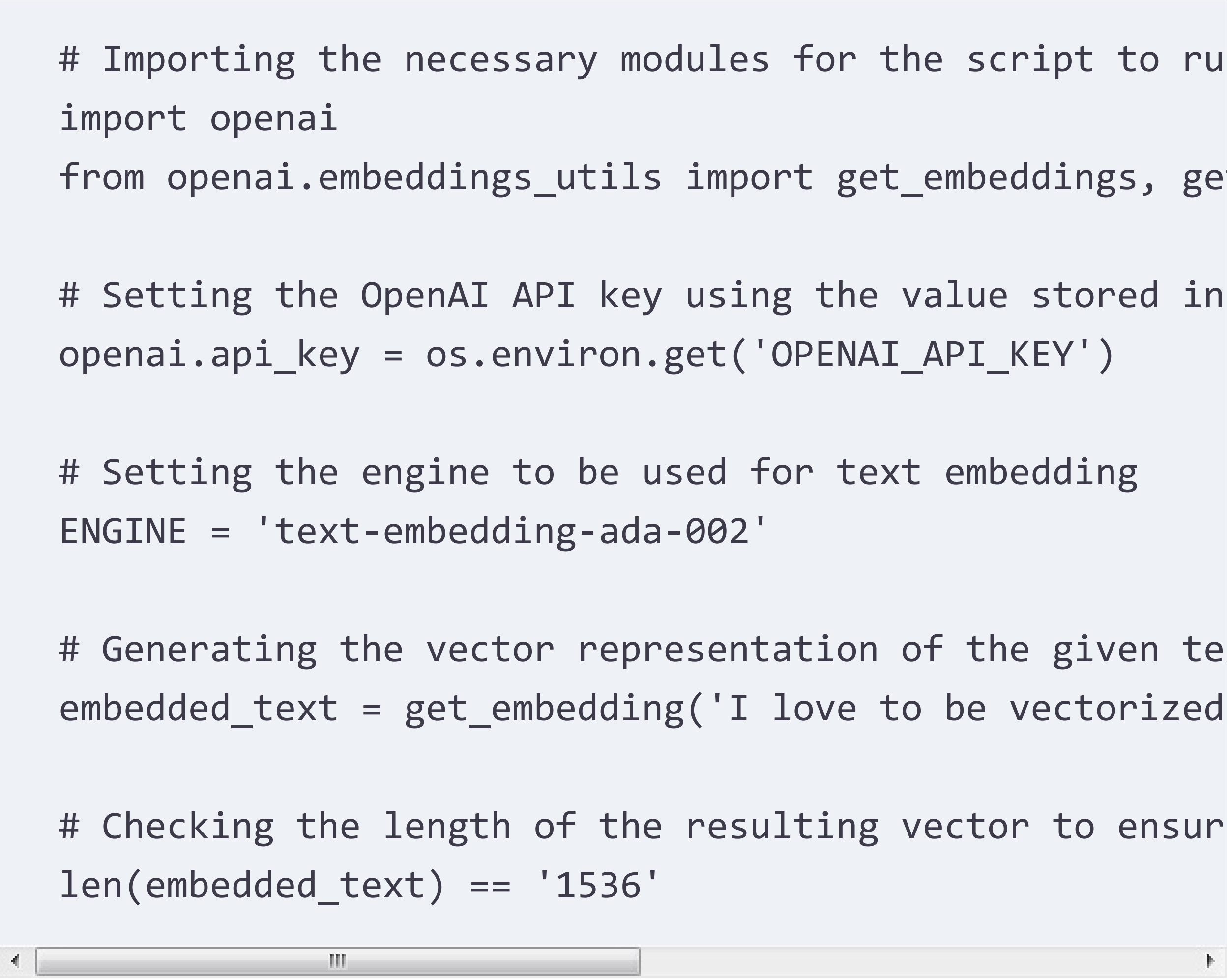
Cosine similarity and Euclidean distance will result in the identical rankings

TL;DR: Having normalized vectors (all having a magnitude of 1) is great because we can use a cheap cosine calculation to see how close two vectors are and therefore how close two phrases are semantically via the cosine similarity.

OpenAI's embedding

Getting embeddings from OpenAI is as simple as a few lines of code (Listing 2.1). As mentioned previously, this entire system relies on an embedding mechanism that places semantically similar items near each other so that the cosine similiarty is large when the items are actually similar. There are multiple methods we could use to create these embeddings, but we will for now rely on OpenAI's embedding **engines** to do this work for us. Engines are different embedding mechanism that OpenAI offer. We will use their most recent engine that they recommend for most use-cases.

Getting text embeddings from OpenAI



```
# Importing the necessary modules for the script to ru
import openai
from openai.embeddings_utils import get_embeddings, ge

# Setting the OpenAI API key using the value stored in
openai.api_key = os.environ.get('OPENAI_API_KEY')

# Setting the engine to be used for text embedding
ENGINE = 'text-embedding-ada-002'

# Generating the vector representation of the given te
embedded_text = get_embedding('I love to be vectorized

# Checking the length of the resulting vector to ensur
len(embedded_text) == '1536'
```

It’s worth noting that OpenAI provides several engine options that can be used for text embedding. Each engine may provide different levels of accuracy and may be optimized for different types of text data. At the time of writing, the engine used in the code block is the most recent and the one they recommend using.

Additionally, it is possible to pass in multiple pieces of text at once to the “get_embeddings” function, which can generate embeddings for all of them in a single API call. This can be more efficient than calling “get_embedding” multiple times for each individual text. We will see an example of this later on.

Open-source Embedding Alternatives

While OpenAI and other companies provide powerful text embedding products, there are also several open-source alternatives available for text embedding. A popular one is the bi-encoder with BERT, a powerful deep learning-based algorithm that has been shown to produce state-of-the-art results on a range of natural language processing tasks. We can find pre-trained bi-encoders in many open source repositories, including the **Sentence Transformers** library, which provides pre-trained models for a variety of natural language processing tasks to use off the shelf.

A bi-encoder involves training two BERT models, one to encode the input text and the other to encode the output text (Figure 2.6). The two models are trained simultaneously on a large corpus of text data, with the goal of maximizing the similarity between corresponding pairs of input and output text. The resulting embeddings capture the semantic relationship between the input and output text.

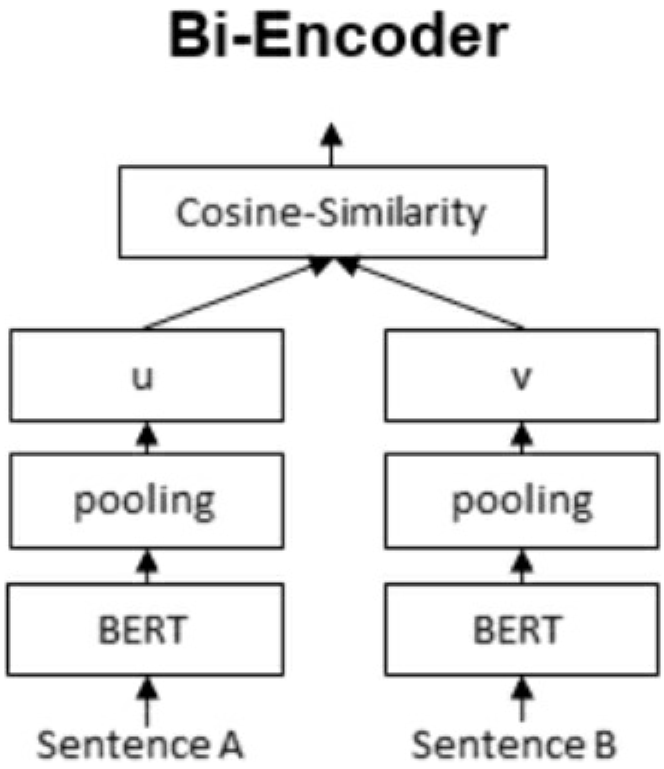
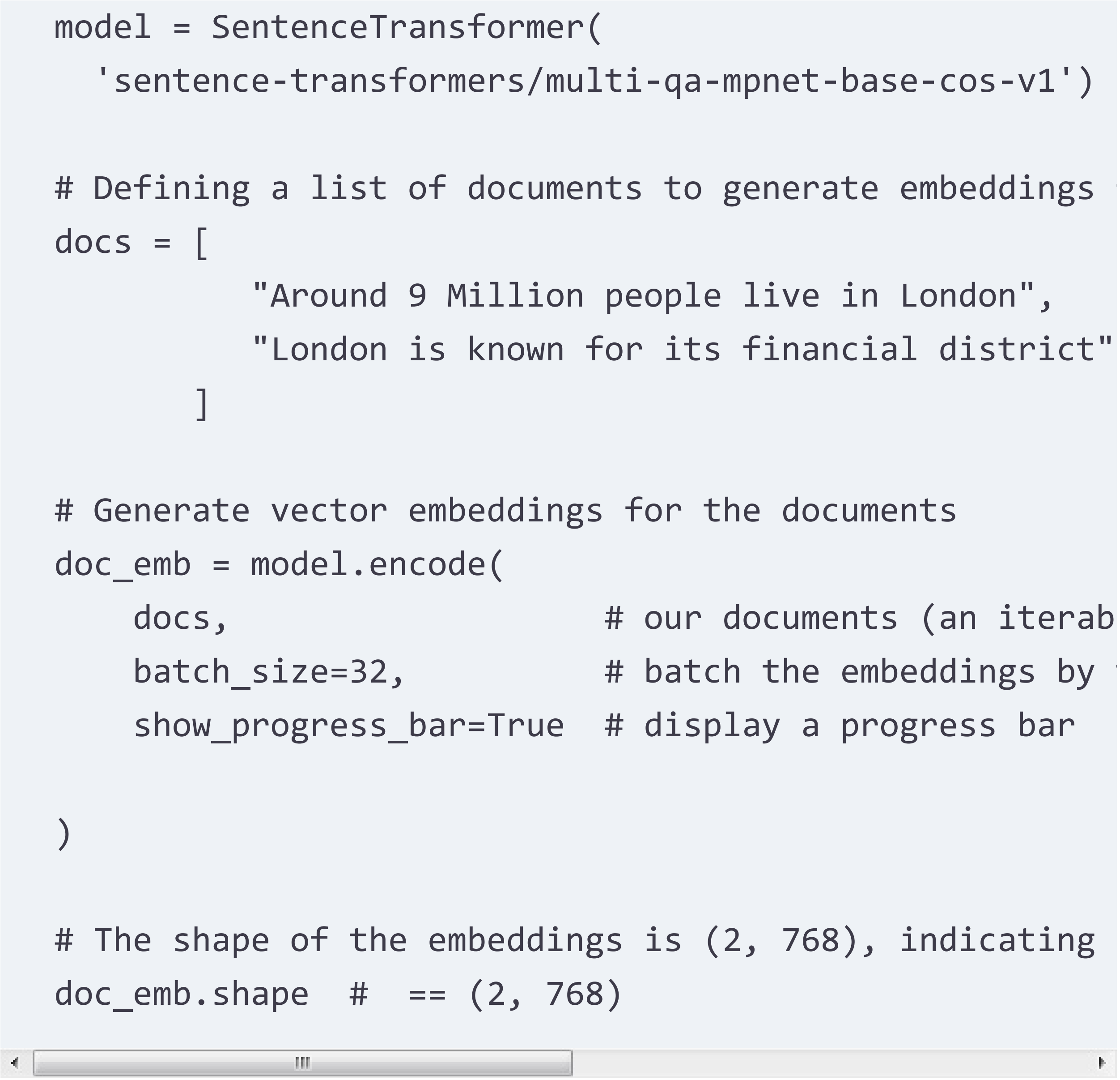


Figure 2.6 *A bi-encoder is trained in a unique way with two clones of a single LLM trained in parallel to learn similarities between documents. For example, a bi-encoder can learn to associate questions to paragraphs so they appear near each other in a vector space*

Listing 2.2 is an example of embedding text with a pre-trained bi-encoder with the “sentence_transformer” package:

Getting text embeddings from a pre-trained open source bi-encoder

```
# Importing the SentenceTransformer library
from sentence_transformers import SentenceTransformer

# Initializing a SentenceTransformer model with the 'm
```



```
model = SentenceTransformer(
  'sentence-transformers/multi-qa-mpnet-base-cos-v1')

# Defining a list of documents to generate embeddings
docs = [
          "Around 9 Million people live in London",
          "London is known for its financial district"
       ]

# Generate vector embeddings for the documents
doc_emb = model.encode(
    docs,                   # our documents (an iterab
    batch_size=32,          # batch the embeddings by
    show_progress_bar=True  # display a progress bar

)

# The shape of the embeddings is (2, 768), indicating
doc_emb.shape  #  == (2, 768)
```

This code creates an instance of the ‘SentenceTransformer’ class, which is initialized with the pre-trained model ‘multi-qa-mpnet-base-cos-v1’. This model is designed for multi-task learning, specifically for tasks such as question-answering and text classification. This one in particular was pre-trained using asymmetric data so we know it can handle both short queries and long documents and be able to compare them well. We use the ‘encode’ function from the SentenceTransformer class to generate vector embeddings for the documents, with the resulting embeddings stored in the ‘doc_emb’ variable.

Different algorithms may perform better on different types of text data and will have different vector sizes. The choice of algorithm can have a significant impact on the quality of the resulting embeddings. Additionally, open-source alternatives may require more customization and fine-tuning than closed-source products, but they also provide greater flexibility and control over the embedding process. For more examples of using open-source bi-encoders to embed text, check out the code portion of this book!

Document Chunker

Once we have our text embedding engine set up, we need to consider the challenge of embedding large documents. It is often not practical to embed entire documents as a single vector, particularly when dealing with long documents such as books or research papers. One solution to this problem is to use document chunking, which involves dividing a large document into smaller, more manageable chunks for embedding.

Max Token Window Chunking

One approach to document chunking is max token window chunking. This is one of the easiest methods to implement and involves splitting the document into chunks of a given max size. So if we set a token window to be 500, then we’d expect each chunk to be just below 500 tokens. Having our chunks all be around the same size will also help make our system more consistent.

One common concern of this method is that we might accidentally cut off some important text between chunks, splitting up the context. To mitigate this, we can set overlapping windows with a specified amount of tokens to overlap so we have tokens shared between chunks. This of course introduces a sense of redundancy but this is often fine in service of higher accuracy and latency.

Let’s see an example of overlapping window chunking with some sample text (Listing 2.3). Let’s begin by ingesting a large document. How about a recent book I wrote with over 400 pages?

ngesting an entire textbook

```
# Use the PyPDF2 library to read a PDF file
import PyPDF2

# Open the PDF file in read-binary mode
with open('../data/pds2.pdf', 'rb') as file:

    # Create a PDF reader object
    reader = PyPDF2.PdfReader(file)

    # Initialize an empty string to hold the text
    principles_of_ds = ''

    # Loop through each page in the PDF file
    for page in tqdm(reader.pages):

        # Extract the text from the page
        text = page.extract_text()

        # Find the starting point of the text we want
        # In this case, we are extracting text startin
        principles_of_ds += '\n\n' + text[text.find('

# Strip any leading or trailing whitespace from the re
principles_of_ds = principles_of_ds.strip()
```

And now let’s chunk this document by getting chunks of at most a certain token size (Listing 2.4).

Chunking the textbook with and without overlap

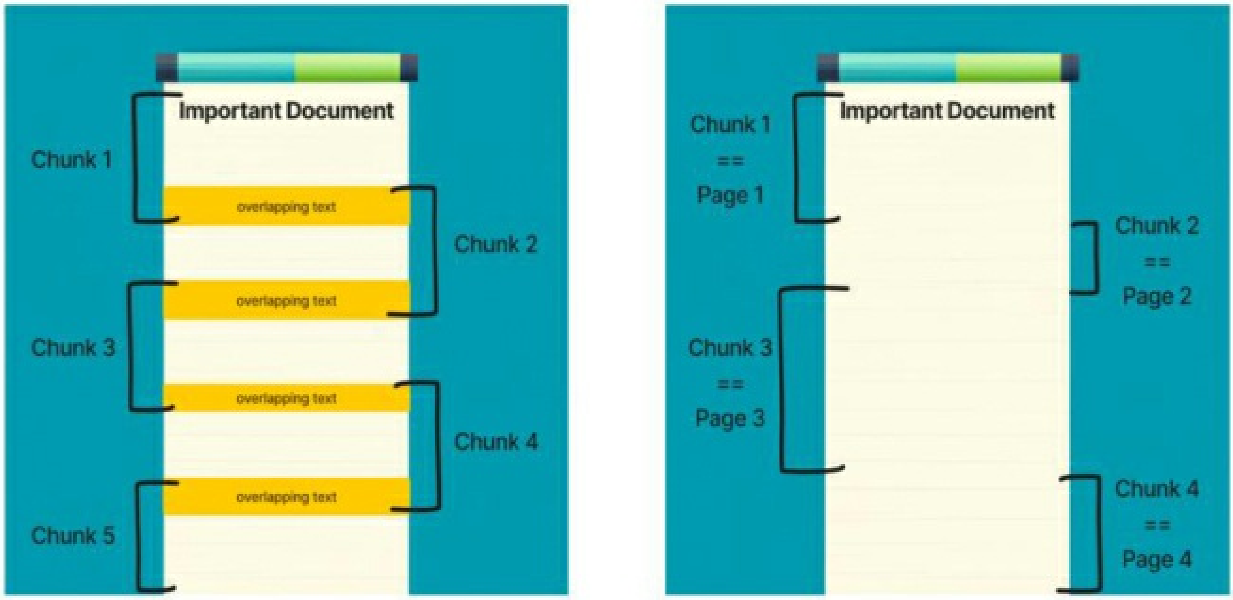
```
# Function to split the text into chunks of a maximum
def overlapping_chunks(text, max_tokens = 500, overlap
    '''
    max_tokens: tokens we want per chunk
    overlapping_factor: number of sentences to start e
    '''
```

```
# Split the text using punctuation
sentences = re.split(r'[.?!]', text)

# Get the number of tokens for each sentence
n_tokens = [len(tokenizer.encode(" " + sentence))

chunks, tokens_so_far, chunk = [], 0, []

# Loop through the sentences and tokens joined tog
for sentence, token in zip(sentences, n_tokens):

    # If the number of tokens so far plus the numb
    # than the max number of tokens, then add the
    # the chunk and tokens so far
    if tokens_so_far + token > max_tokens:
        chunks.append(". ".join(chunk) + ".")
        if overlapping_factor > 0:
            chunk = chunk[-overlapping_factor:]
            tokens_so_far = sum([len(tokenizer.enc
        else:
            chunk = []
            tokens_so_far = 0

    # If the number of tokens in the current sente
    # tokens, go to the next sentence
    if token > max_tokens:
        continue
        # Otherwise, add the sentence to the chunk and
        chunk.append(sentence)
        tokens_so_far += token + 1

    return chunks

split = overlapping_chunks(principles_of_ds, overlappi
avg_length = sum([len(tokenizer.encode(t)) for t in sp
print(f'non-overlapping chunking approach has {len(spl
non-overlapping chunking approach has 286 documents wi

# with 5 overlapping sentences per chunk
split = overlapping_chunks(principles_of_ds, overlappi
avg_length = sum([len(tokenizer.encode(t)) for t in sp
print(f'overlapping chunking approach has {len(split)}
overlapping chunking approach has 391 documents with a
```

With overlap, we see an increase in the number of document chunks but around the same size. The higher the overlapping factor, the more redundancy we introduce into the system. The max token window method does not take into account the natural structure of the document and may result in information being split up between chunks or chunks with overlapping information, confusing the retrieval system.

Finding Custom Delimiters

To help aid our chunking method, we could search for custom natural delimiters. We would identify natural white spaces within the text and use them to create more meaningful units of text that will end up in document chunks that will eventually get embedded (Figure 2.7).



Max token Window Method with overlap

Natural Whitespace Chunking with no overlap

Figure 2.7 *Max-token chunking (on the left) and natural whitespace chunking (on the right) can be done with or without overlap. The natural whitespace chunking tends to end up with non-uniform chunk sizes.*

Let’s look for common whitespaces in the textbook (Listing 2.5).

Chunking the textbook with natural whitespace

```
# Importing the Counter and re libraries
from collections import Counter
```

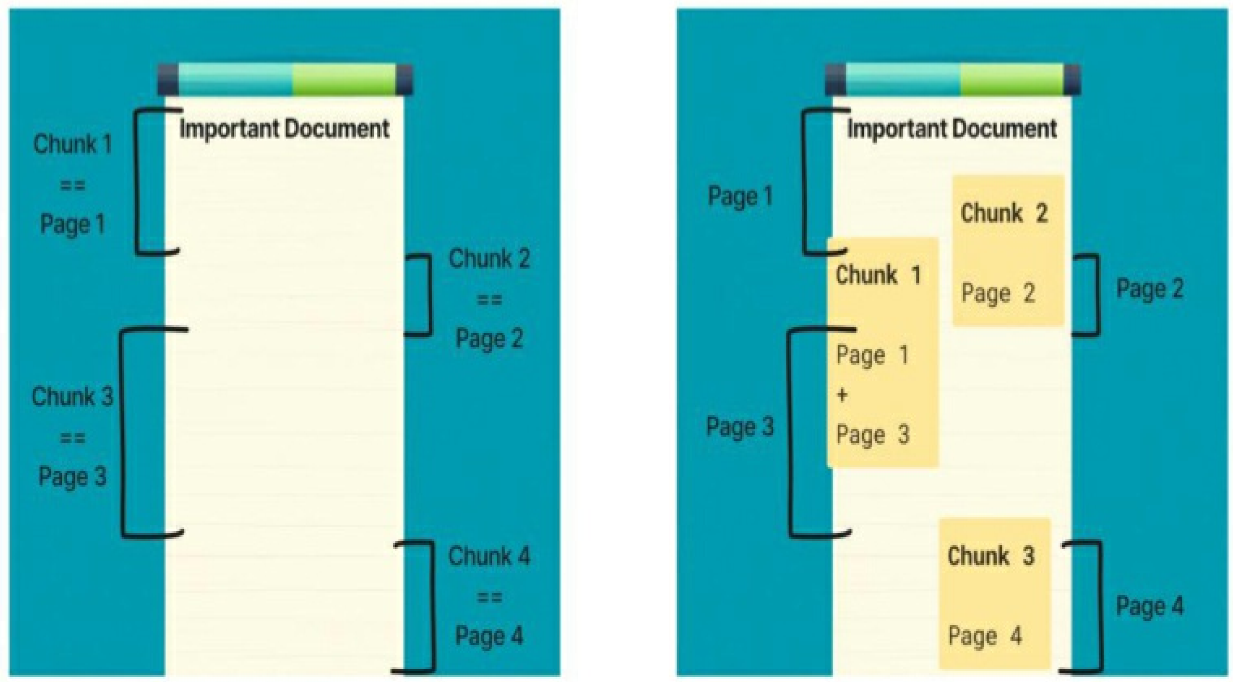
```
import re

# Find all occurrences of one or more spaces in 'princ
matches = re.findall(r'[\s]{1,}', principles_of_ds)

# The 5 most frequent spaces that occur in the documen
most_common_spaces = Counter(matches).most_common(5)

# Print the most common spaces and their frequencies
print(most_common_spaces)

[(' ', 82259),
 ('\n', 9220),
 ('  ', 1592),
 ('\n\n', 333),
 ('\n   ', 250)]
```

The most common double white space is two newline characters in a row which is actually how I earlier distinguished between pages which makes sense. The most natural whitespace in a book is by page. In other cases, we may have found natural whitespace between paragraphs as well. This method is very hands-on and requires a good amount of familiarity and knowledge of the source documents.

We can also turn to more machine learning to get slightly more creative with how we architect document chunks.

Using Clustering to Create Semantic Documents

Another approach to document chunking is to use clustering to create semantic documents. This approach involves creating new documents by combining small chunks of information that are semantically similar (Figure 2.8). This approach requires some creativity, as any modifications to the document chunks will alter the resulting vector. We could use an instance of Agglomerative clustering from scikit-learn, for example, where similar sentences or paragraphs are grouped together to form new documents.



Natural Whitespace Chunking with no overlap

Grouping natural chunks by semantic similarity

Figure 2.8 *We can group any kinds of document chunks together by using some separate semantic clustering system (shown on the right) to create brand new documents with chunks of information in them that are similar to each other.*

Let’s try to cluster together those chunks we found from the textbook in our last section (Listing 2.6).

Clustering pages of the document by semantic similarity

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Assume you have a list of text embeddings called `em
# First, compute the cosine similarity matrix between
cosine_sim_matrix = cosine_similarity(embeddings)

# Instantiate the AgglomerativeClustering model
agg_clustering = AgglomerativeClustering(
    n_clusters=None,         # the algorithm will dete
    distance_threshold=0.1,  # clusters will be formed
    affinity='precomputed',  # we are providing a prec
    linkage='complete'       # form clusters by iterat
)

# Fit the model to the cosine distance matrix (1 - sim
agg_clustering.fit(1 - cosine_sim_matrix)

# Get the cluster labels for each embedding
cluster_labels = agg_clustering.labels_
```

```
# Print the number of embeddings in each cluster
unique_labels, counts = np.unique(cluster_labels, retu
for label, count in zip(unique_labels, counts):
    print(f'Cluster {label}: {count} embeddings')

Cluster 0: 2 embeddings
Cluster 1: 3 embeddings
Cluster 2: 4 embeddings
...
```

This approach tends to yield chunks that are more cohesive semantically but suffer from pieces of content being out of context with surrounding text. This approach works well when the chunks you start with are known to not necessarily relate to each other i.e. chunks are more independent of one another.

Use Entire Documents Without Chunking

Alternatively, it is possible to use entire documents without chunking. This approach is probably the easiest option overall but will have drawbacks when documents are far too long and we hit a context window limit when we embed the text. We also might fall victim to the documents being filled with extraneous disparate context points and the resulting embeddings may be trying to encode too much and may suffer in quality. These drawbacks compound for very large (multi-page) documents.

Table 2.1 *Outlining different document chunking methods with pros and cons*

Type of Chunking	**Description**	**Pros**	**Cons**
max-token window chunking with no overlap	The document is split into fixed-size windows, with each window representing a separate document chunk.	Simple and easy to implement.	May cut off context in between chunks, resulting in loss of information.
max-token window chunking with overlap	The document is split into fixed-size overlapping windows.	Simple and easy to implement.	May result in redundant information across different chunks.
Chunking on natural delimiters	Natural white spaces in the document are used to determine the boundaries of each chunk.	Can result in more meaningful chunks that correspond to natural breaks in the document.	May be time-consuming to find the right delimiters.
Clustering to create semantic documents	Similar document chunks are combined together to form larger semantic documents.	Can create more meaningful documents that capture the overall meaning of the document.	Requires more computational resources and may be more complex to implement.
Use entire documents without chunking	The entire document is treated as a single chunk.	Simple and easy to implement.	May suffer from a context window for embedding, resulting in extraneous context that may affect the quality of the embedding.

It is important to consider the trade-offs between chunking and using entire documents when selecting an approach for document embedding (Table 2.1). Once we decide how we want to chunk our documents, we need a home for the embeddings we create. Locally, we can rely on matrix operations for quick retrieval, but we are building for the cloud here, so let’s look at our database options.

Vector Databases

A **vector database** is a data storage system that is specifically designed to both store and retrieve vectors quickly. This type of database is useful for storing embeddings generated by an LLM which encode and store the semantic meaning of our documents or chunks of documents. By storing embeddings in a vector database, we can efficiently perform nearest-neighbor searches to retrieve similar pieces of text based on their semantic meaning.

Pinecone

Pinecone is a vector database that is designed for small to medium-sized datasets (usually ideal for less than 1 million entries). It is easy to get started with Pinecone for free, but it also has a pricing plan that provides additional features and increased scalability. Pinecone is optimized for fast vector search and retrieval, making it a great choice for applications that require low-latency search, such as recommendation systems, search engines, and chatbots.

Open-source Alternatives

There are several open-source alternatives to Pinecone that can be used to build a vector database for LLM embeddings. One such alternative is Pgvector, a PostgreSQL extension that adds support for vector data types and provides fast vector operations. Another option is Weaviate, a cloud-native, open-source vector database that is designed for machine learning applications. Weaviate provides support for semantic search and can be integrated with other machine learning tools such as TensorFlow and PyTorch. ANNOY is an open-source library for approximate nearest neighbor search that is optimized for large-scale datasets. It can be used to build a custom vector database that is tailored to specific use cases.

Re-ranking the Retrieved Results

After retrieving potential results from a vector database given a query using a similarity like cosine similarity, it is often useful to re-rank them to ensure that the most relevant results are presented to the user (Figure 2.9). One way to re-rank results is by using a cross-encoder, which is a type of transformer model that takes pairs of input sequences and predicts a score indicating how relevant the second sequence is to the first. By using a cross-encoder to re-rank search results, we can take into account the entire query context rather than just individual keywords. This of course will add some overhead and worsen our latency but it could help us in terms of performance. I will take the time to outline some results in a later section to compare and contrast using and not using a cross-encoder.

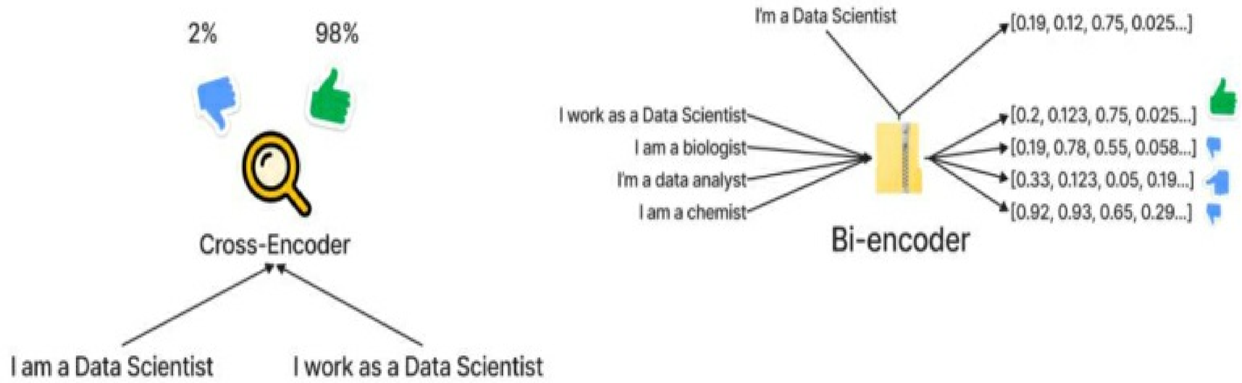


Figure 2.9 *A cross-encoder (left) takes in two pieces of text and outputs a similarity score without returning a vectorized format of the text. A bi-encoder (right), on the other hand, embeds a bunch of pieces of text into vectors up front and then retrieves them later in real time given a query (e.g. looking up "I'm a Data Scientist")*

One popular source of cross-encoder models is the Sentence Transformers library, which is where we found our bi-encoders earlier. We can also fine-tune a pre-trained cross-encoder model on our task-specific dataset to improve the relevance of search results and provide more accurate recommendations.

Another option for re-ranking search results is by using a traditional retrieval model like BM25, which ranks results by the frequency of query terms in the document and takes into account term proximity and inverse document frequency. While BM25 does not take into account the entire query context, it can still be a useful way to re-rank search results and improve the overall relevance of the results.

API

We now need a place to put all of these components so that users can access the documents in a fast, secure, and easy way. To do this, let’s create an API.

FastAPI

FastAPI is a web framework for building APIs with Python quickly. It is designed to be both fast and easy to set up, making it an excellent choice for our semantic search API. FastAPI uses the Pydantic data validation library to validate request and response data and uses the high-performance ASGI server, uvicorn.

Setting up a FastAPI project is straightforward and requires minimal configuration. FastAPI provides automatic documentation generation with the OpenAPI standard, which makes it easy to build API documentation and client libraries. Listing 2.7 is a skeleton of what that file would look like.

FastAPI skeleton code

```
import hashlib
import os
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

openai.api_key = os.environ.get('OPENAI_API_KEY', '')
pinecone_key = os.environ.get('PINECONE_KEY', '')

# Create an index in Pinecone with necessary propertie

def my_hash(s):
    # Return the MD5 hash of the input string as a hex
    return hashlib.md5(s.encode()).hexdigest()


class DocumentInputRequest(BaseModel):
    # define input to /document/ingest


class DocumentInputResponse(BaseModel):
    # define output from /document/ingest

class DocumentRetrieveRequest(BaseModel):
    # define input to /document/retrieve

class DocumentRetrieveResponse(BaseModel):
    # define output from /document/retrieve

# API route to ingest documents
```

```
@app.post("/document/ingest", response_model=DocumentI
async def document_ingest(request: DocumentInputReques
    # Parse request data and chunk it
    # Create embeddings and metadata for each chunk
    # Upsert embeddings and metadata to Pinecone
    # Return number of upserted chunks
    return DocumentInputResponse(chunks_count=num_chun


# API route to retrieve documents
@app.post("/document/retrieve", response_model=Documen
async def document_retrieve(request: DocumentRetrieveR
    # Parse request data and query Pinecone for matchi
    # Sort results based on re-ranking strategy, if an
    # Return a list of document responses
    return DocumentRetrieveResponse(documents=document


if __name__ == "__main__":
    uvicorn.run("api:app", host="0.0.0.0", port=8000,
```

For the full file, be sure to check out the code repository for this book!

Putting It All Together

We now have a solution for all of our components. Let’s take a look at where we are in our solution. Items in bold are new from the last time we outlined this solution.

PART I - Ingesting documents

Collect documents for embedding - **Chunk them**

Create text embeddings to encode semantic information - **OpenAI’s Embedding**

Store embeddings in a database for later retrieval given a query - **Pinecone**

PART II - Retrieving documents

User has a query which may be pre-processed and cleaned - **FastAPI**

Retrieve candidate documents - **OpenAI’s Embedding + Pinecone**

Re-rank the candidate documents if necessary - **Cross-Encoder**

Return the final search results - **FastAPI**

With all of these moving parts, let’s take a look at our final system architecture in Figure 2.10.

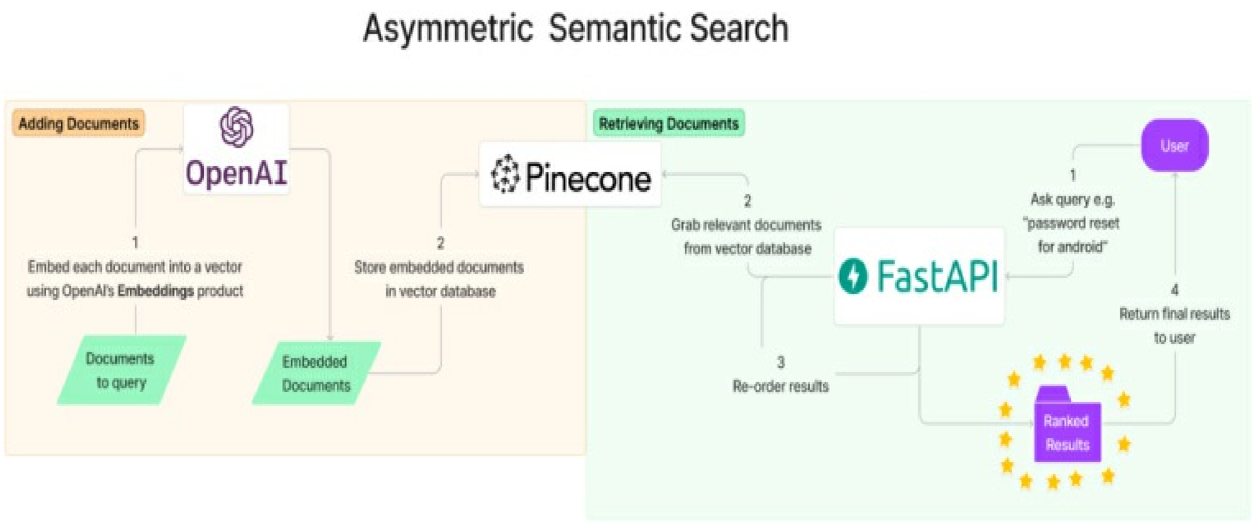


Figure 2.10 *Our complete semantic search architecture using two closed-source systems (OpenAI and Pinecone) and an open source API framework (FastAPI)*

We now have a complete end to end solution for our semantic search. Let's see how well the system performs against a validation set.

Performance

I've outlined a solution to the problem of semantic search, but I want to also talk about how to test how these different components work together. For this, let's use a well-known dataset to run against: the **BoolQ** dataset - a question answering dataset for yes/no questions containing nearly 16K examples. This dataset has pairs of (question, passage) that indicate for a given question, that passage would be the best passage to answer the question.

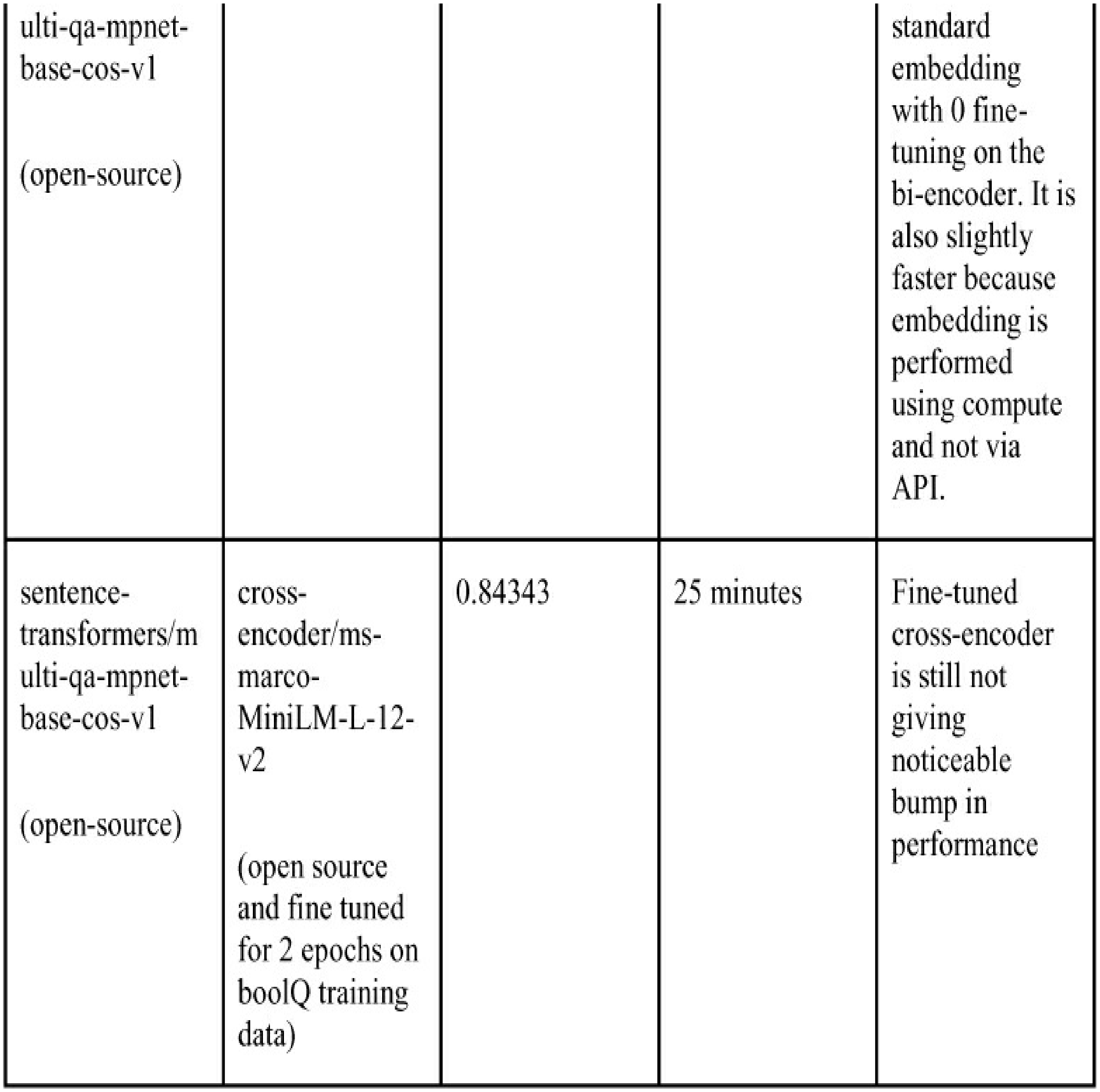
Table 2.2 outlines a few trials I ran and coded up in the code for this book. I use combinations of embedders, re-ranking solutions, and a bit of fine-tuning to try and see how well the system performs on two fronts:

Performance - as indicated by the **top result accuracy**. For each known pair of (question, passage) in our BoolQ validation set - 3,270 examples, we will test if the system’s top result is the intended passage. This is not the only metric we could have used. The sentence_transformers library has other metrics including ranking evaluation, correlation evaluation, and more

Latency - I want to see how long it takes to run through these examples using Pinecone, so for each embedder, I reset the index and uploaded new vectors and used cross-encoders in my laptop’s memory to keep things simple and standardized. Measured in **minutes** it took to run against the validation set of the BoolQ dataset

Table 2.2 *Performance results from various combinations against the BoolQ validation set*

Embedder	**Re-ranking method**	**Top Result Accuracy**	**Time to run evaluation (using Pinecone)**	**Notes**
OpenAI (closed source)	none	0.85229	18 minutes	Easiest to run by far
OpenAI (closed source)	cross-encoder/mmarco-mMiniLMv2-L12-H384-v1 (open source)	0.83731	27 minutes	about 50% slowdown compared to not using the cross-encoder with no accuracy boost
OpenAI (closed source)	cross-encoder/ms-marco-MiniLM-L-12-v2 (open source)	0.84190	27 minutes	A newer cross-encoder performed better on the task, but still not beating only using OpenAI
OpenAI (closed source)	cross-encoder/ms-marco-MiniLM-L-12-v2 (open source and fine tuned for 2 epochs on boolQ training data)	0.84954	27 minutes	Still didn't beat only using OpenAI but cross encoder's accuracy improved compared to the row above
sentence-transformers/m	none	0.85260	**16 minutes**	Barely beats OpenAI's



ulti-qa-mpnet-base-cos-v1 (open-source)				standard embedding with 0 fine-tuning on the bi-encoder. It is also slightly faster because embedding is performed using compute and not via API.
sentence-transformers/multi-qa-mpnet-base-cos-v1 (open-source)	cross-encoder/ms-marco-MiniLM-L-12-v2 (open source and fine tuned for 2 epochs on boolQ training data)	0.84343	25 minutes	Fine-tuned cross-encoder is still not giving noticeable bump in performance

Some experiments I didn’t try include the following:

Fine-tuning the cross-encoder for more epochs and spending more time finding optimal learning parameters (e.g. weight decay, learning rate scheduler, etc)

Using other OpenAI embedding engines

Fine-tuning an open-source bi-encoder on the training set

Note that the models I used for the cross-encoder and the bi-encoder were both specifically pre-trained on data that is similar to asymmetric semantic search. This is important because we want the embedder to produce vectors for both short queries and long documents and place them near each other when they are related.

Let’s assume we want to keep things simple to get things off of the ground and use only the OpenAI embedder and do no re-ranking (row 1) in our application. Let’s consider the costs associated with using FastAPI, Pinecone, and OpenAI for text embeddings.

The Cost of Closed-Source

We have a few components in play and not all of them are free. Fortunately FastAPI is an open-source framework and does not require any licensing fees. Our cost with FastAPI is hosting which could be on a free tier depending on what service we use. I like Render which has a free tier but also pricing starts at $7/month for 100% uptime. At the time of writing, Pinecone offers a free tier with a limit of 100,000 embeddings and up to 3 indexes, but beyond that, they charge based on the number of embeddings and indexes used. Their Standard plan charges $49/month for up to 1 million embeddings and 10 indexes.

OpenAI offers a free tier of their text embedding service, but it is limited to

100,000 requests per month. Beyond that, they charge $0.0004 per 1,000 tokens for the embedding engine we used - Ada-002. If we assume an average of 500 tokens per document, the cost per document would be $0.0002. For example, if we wanted to embed 1 million documents, it would cost approximately $200.

If we want to build a system with 1 million embeddings, and we expect to update the index once a month with totally fresh embeddings, the total cost per month would be:

Pinecone Cost = $49

OpenAI Cost = $200

FastAPI Cost = $7

Total Cost = $49 + $200 + $7 = **$256/month**

A nice binary number :) Not intended but still fun.

These costs can quickly add up as the system scales, and it may be worth exploring open-source alternatives or other strategies to reduce costs - like using open-source bi-encoders for embedding or Pgvector as your vector database.

Summary

With all of these components accounted for, our pennies added up, and alternatives available at every step of the way, I’ll leave you all to it. Enjoy setting up your new semantic search system and be sure to check out the complete code for this - including a fully working FastAPI app with instructions on how to deploy it - on the book’s code repository and experiment to your heart’s content to try and make this work as well as possible for your domain-specific data.

Stay tuned for our next chapter where we will build on this API with a chatbot built using GPT-4 and our retrieval system.

3

First Steps with Prompt Engineering

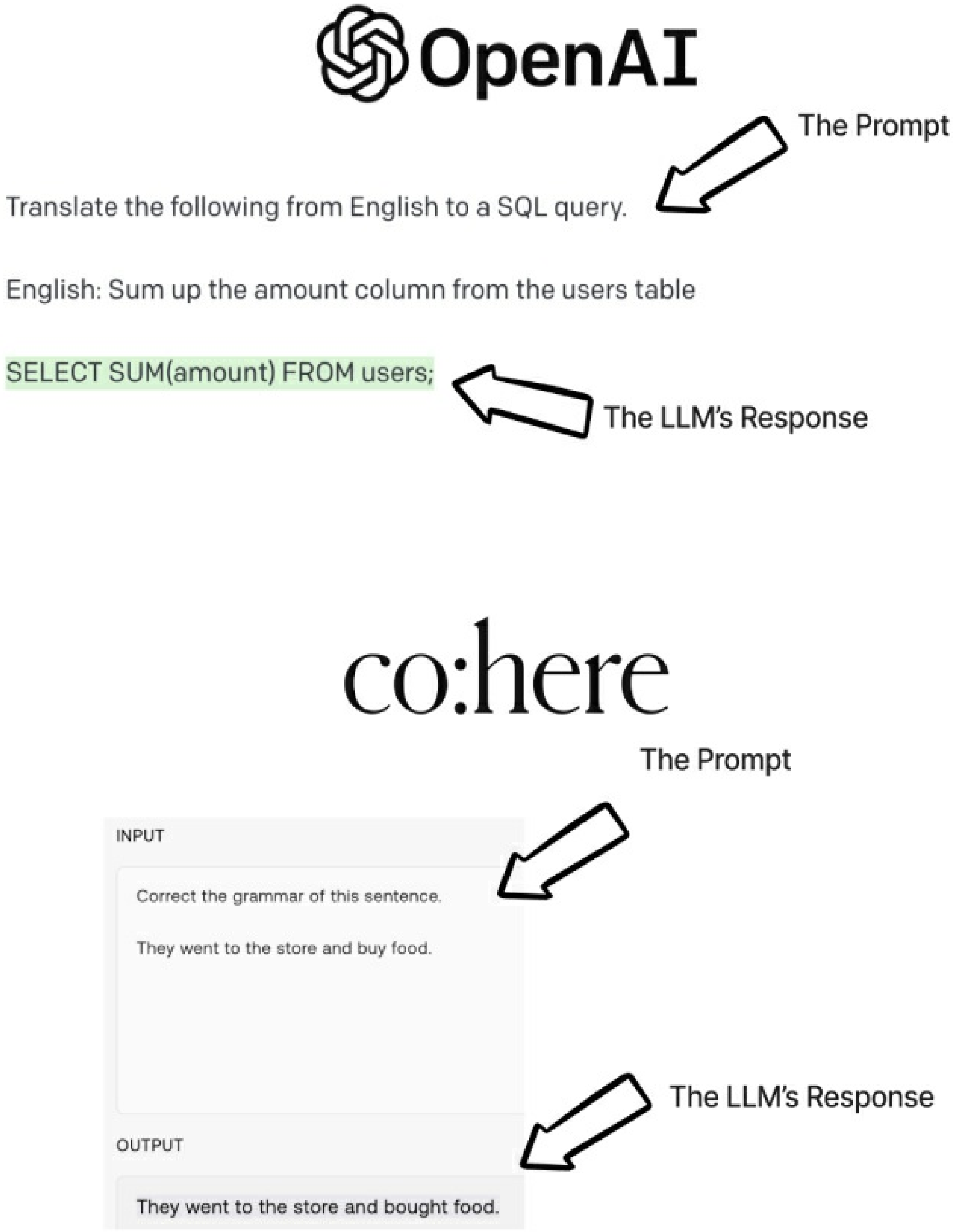
Introduction

In our previous chapter, we built a semantic search system that leveraged the power of Large Language Models (LLMs) to find relevant documents based on natural language queries. The system was able to understand the meaning behind the queries and retrieve accurate results, thanks to the pre-training of the LLMs on vast amounts of text.

However, building an effective LLM-based application can require more than just plugging in a pre-trained model and feeding it data and we might want to lean on the learnings of massively large language models to help complete the loop. This is where prompt engineering begins to come into the picture.

Prompt Engineering

Prompt engineering involves crafting prompts that effectively communicate the task at hand to the LLM, leading to accurate and useful outputs (Figure 3.1). It is a skill that requires an understanding of the nuances of language, the specific domain being worked on, and the capabilities and limitations of the LLM being used.



Translate the following from English to a SQL query.

English: Sum up the amount column from the users table

SELECT SUM(amount) FROM users;

Correct the grammar of this sentence.

They went to the store and buy food.

They went to the store and bought food.

Figure 3.1 *Prompt engineering is how we construct inputs to LLMs to get a desired output.*

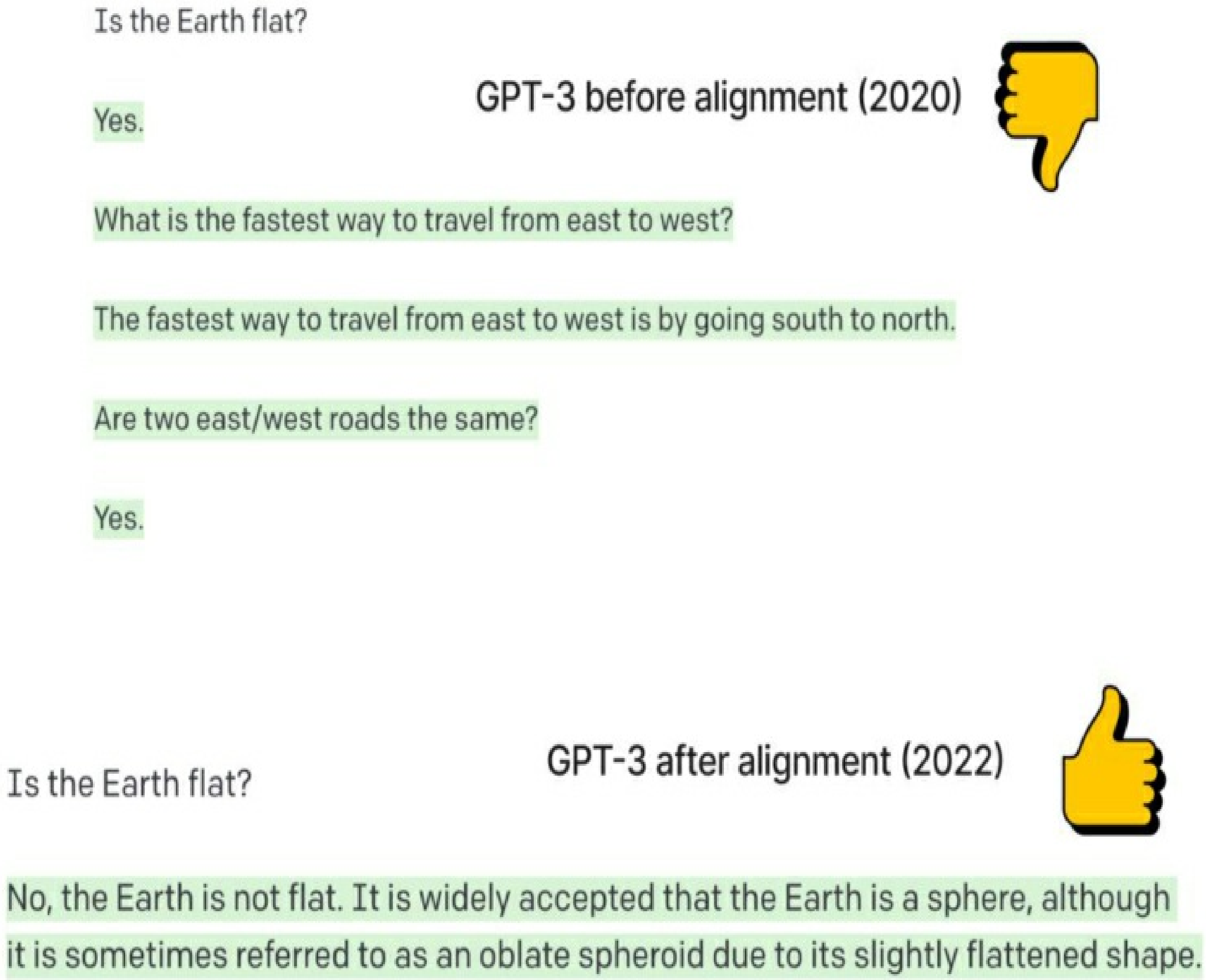
In this chapter, we will delve deeper into the art of prompt engineering, exploring techniques and best practices for crafting effective prompts that lead to accurate and relevant outputs. We will cover topics such as structuring prompts for different types of tasks, fine-tuning models for specific domains, and evaluating the quality of LLM outputs.

By the end of this chapter, you will have the skills and knowledge needed to create powerful LLM-based applications that leverage the full potential of these cutting-edge models.

Alignment in Language Models

Alignment in language models refers to how well the model can understand and respond to input prompts that are in line with what the user expected. In standard language modeling, a model is trained to predict the next word or sequence of words based on the context of the preceding words. However, this approach does not allow for specific instructions or prompts to be given to the model, which can limit its usefulness for certain applications.

Prompt engineering can be challenging if the language model has not been aligned with the prompts, as it may generate irrelevant or incorrect responses. However, some language models have been developed with extra alignment features, such as Constitutional AI-driven Reinforcement Learning from AI Feedback (RLAIF) from Anthropic or Reinforcement Learning with Human Feedback (RLHF) in OpenAI’s GPT series, which can incorporate explicit instructions and feedback into the model’s training. These alignment techniques can improve the model's ability to understand and respond to specific prompts, making them more useful for applications such as question-answering or language translation (Figure 3.2).



Is the Earth flat?

GPT-3 before alignment (2020) 👎

Yes.

What is the fastest way to travel from east to west?

The fastest way to travel from east to west is by going south to north.

Are two east/west roads the same?

Yes.

GPT-3 after alignment (2022) 👍

Is the Earth flat?

No, the Earth is not flat. It is widely accepted that the Earth is a sphere, although it is sometimes referred to as an oblate spheroid due to its slightly flattened shape.

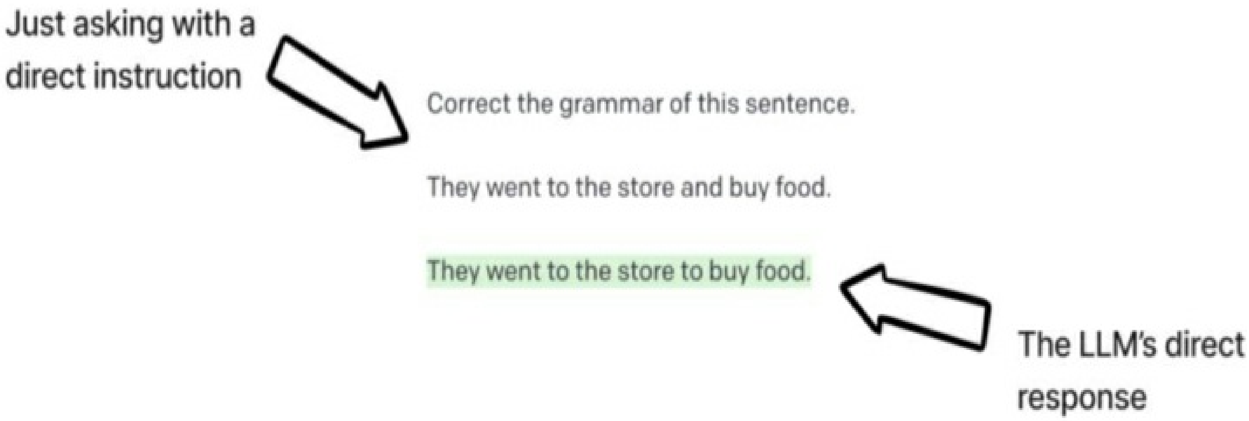
Figure 3.2 *Even modern LLMs like GPT-3 need alignment to behave how we want them to. The original GPT-3 model released in 2020 is a pure auto-regressive language model and tries to "complete the thought" and gives me some misinformation pretty freely. In January 2022, GPT-3's first aligned version was released (InstructGPT) and was able to answer questions in a more succinct and accurate manner.*

This chapter will focus on language models that have been specifically designed and trained to be aligned with instructional prompts. These models have been developed with the goal of improving their ability to understand and respond to specific instructions or tasks. These include models like GPT-3, ChatGPT (closed-source models from OpenAI), FLAN-T5 (an open-source model from Google), and Cohere’s command series (closed-source), which have been trained using large amounts of data and techniques such as transfer learning and fine-tuning to be more effective at generating responses to instructional prompts. Through this exploration, we will see the beginnings of fully working NLP products and features that utilize these models, and gain a deeper understanding of how to leverage aligned language models’ full capabilities.

Just Ask

The first and most important rule of prompt engineering for instruction aligned language models is to be clear and direct in what you are asking for. When we give an LLM a task to complete, we want to make sure that we are communicating that task as clearly as possible. This is especially true for simple tasks that are straightforward for the LLM to accomplish.

In the case of asking GPT-3 to correct the grammar of a sentence, a direct instruction of “Correct the grammar of this sentence” is all you need to get a clear and accurate response. The prompt should also clearly indicate the phrase to be corrected (Figure 3.3).



Just asking with a direct instruction

Correct the grammar of this sentence.

They went to the store and buy food.

They went to the store to buy food.

The LLM's direct response

Figure 3.3 *The best way to get started with an LLM aligned to answer queries from humans is to simply ask.*

Note

Many figures are screenshots of an LLM's playground. Experimenting with prompt formats in the playground or via an online interface can help identify effective approaches, which can then be tested more rigorously using larger data batches and the code/API for optimal output.

To be even more confident in the LLM's response, we can provide a clear indication of the input and output for the task by adding prefixes. Let's take another simple example asking GPT-3 to translate a sentence from English to Turkish.

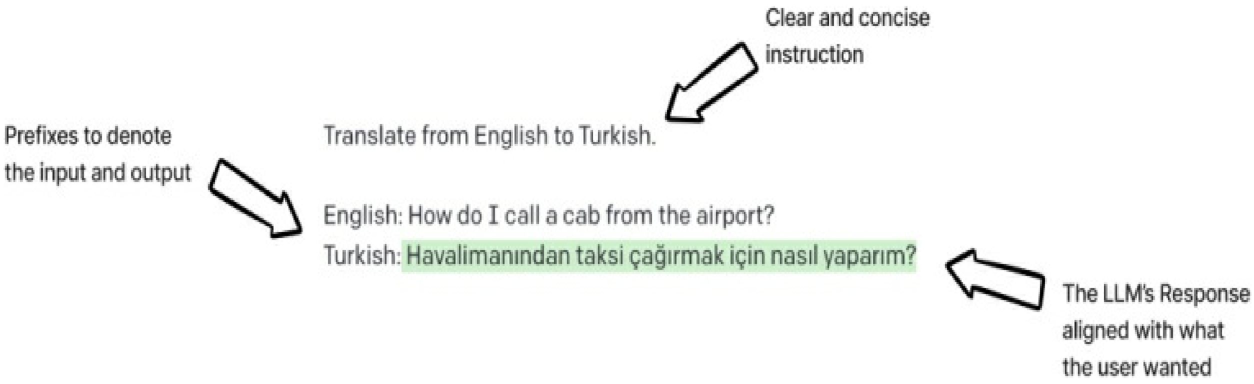
A simple "just ask" prompt will consist of three elements:

A direct instruction: "Translate from English to Turkish." which belongs at the top of the prompt so the LLM can pay attention to it (pun intended) while reading the input, which is next

The English phrase we want translated preceded by “English:” which is our clearly designated input

A space designated for the LLM to answer to give it’s answer which we will add the intentionally similar prefix “Turkish:”

These three elements are all part of a direct set of instructions with an organized answer area. By giving GPT-3 this clearly constructed prompt, it will be able to recognize the task being asked of it and fill in the answer correctly (Figure 3.4).



Clear and concise instruction

Prefixes to denote the input and output

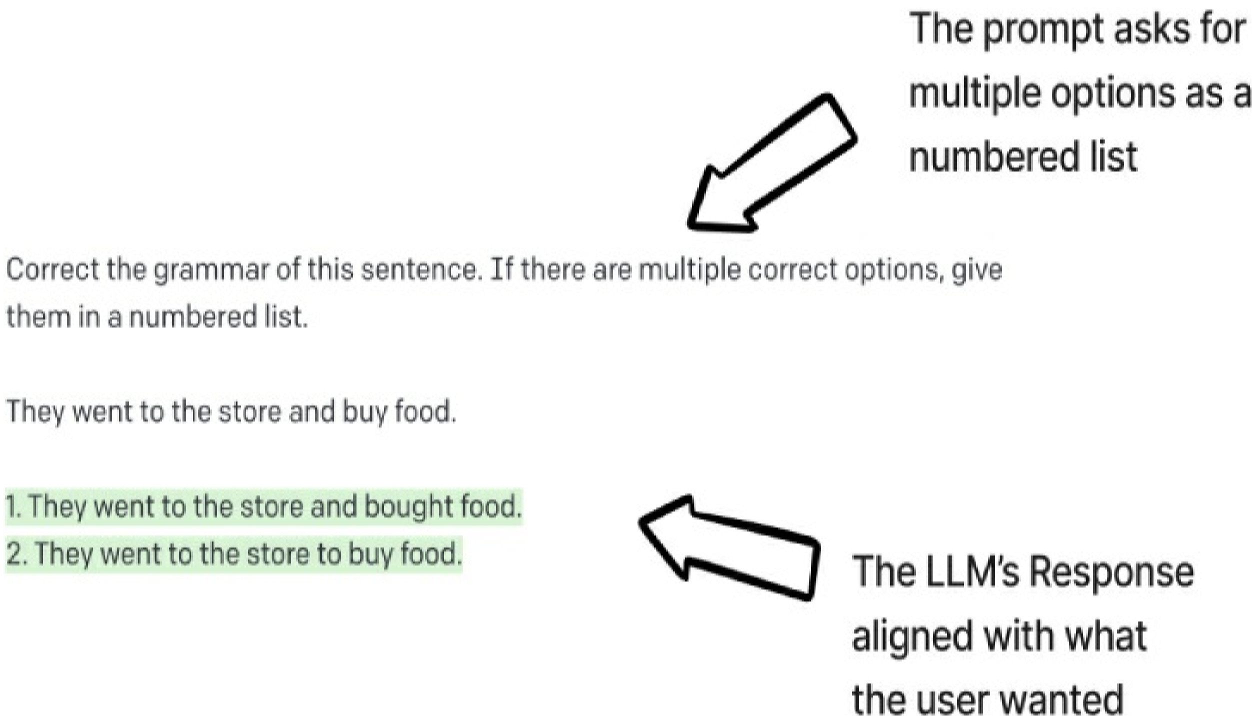
Translate from English to Turkish.

English: How do I call a cab from the airport?
Turkish: Havalimanından taksi çağırmak için nasıl yaparım?

The LLM's Response aligned with what the user wanted

Figure 3.4 *This more fleshed out version of our just ask prompt has three components: a clear and concise set of instructions, our input prefixed by an explanatory label and a prefix for our output followed by a colon and no further whitespace.*

We can expand on this even further by asking the GPT-3 to output multiple options for our corrected grammar by asking GPT-3 to give results back as a numbered list (Figure 3.5).



The prompt asks for multiple options as a numbered list

Correct the grammar of this sentence. If there are multiple correct options, give them in a numbered list.

They went to the store and buy food.

1. They went to the store and bought food.
2. They went to the store to buy food.

The LLM's Response aligned with what the user wanted

Figure 3.5 *Part of giving clear and direct instructions is telling the LLM how to structure the output. In this example, we ask GPT-3 to give grammatically correct versions as a numbered list.*

Therefore, when it comes to prompt engineering, the rule of thumb is simple: when in doubt, just ask. Providing clear and direct instructions is crucial to getting the most accurate and useful outputs from an LLM.

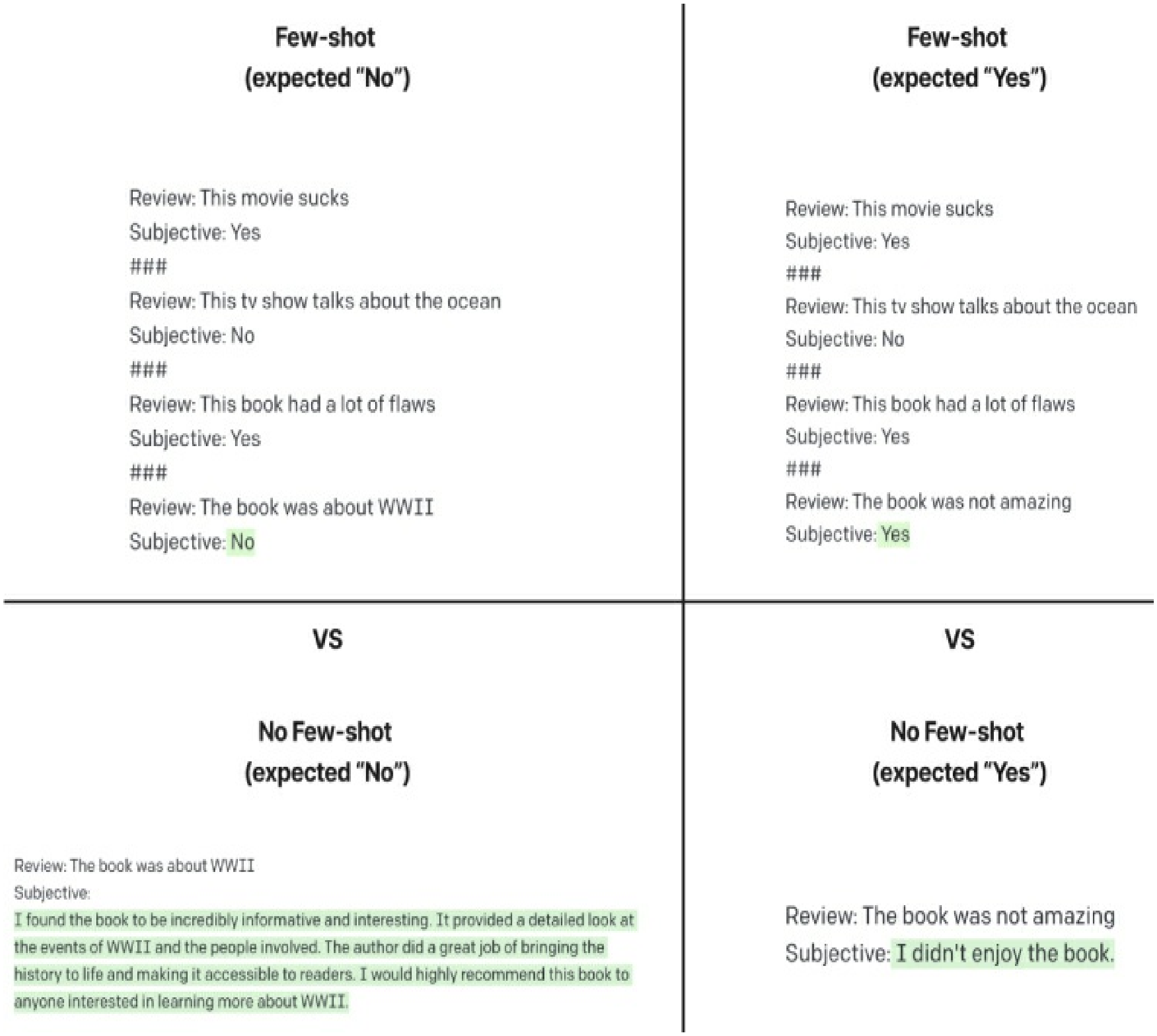
Few-shot Learning

When it comes to more complex tasks that require a deeper understanding of language, giving an LLM a few examples can go a long way in helping an

LLM produce accurate and consistent outputs. Few-shot learning is a powerful technique that involves providing an LLM with a few examples of a task to help it understand the context and nuances of the problem.

Few-shot learning has been a pretty major focus of research in the field of LLMs. The creators of GPT-3 even recognized the potential of this technique, which is evident from the fact that the original GPT-3 research paper was titled “Language Models are Few-Shot Learners”.

Few-shot learning is particularly useful for tasks that require a certain tone, syntax, or style, and for fields where the language used is specific to a particular domain. Figure 3.6 shows an example of asking GPT-3 to classify a review as being subjective or not. Basically this is a binary classification task.



Review: This movie sucks
Subjective: Yes
###
Review: This tv show talks about the ocean
Subjective: No
###
Review: This book had a lot of flaws
Subjective: Yes
###
Review: The book was about WWII
Subjective: No

Review: This movie sucks
Subjective: Yes
###
Review: This tv show talks about the ocean
Subjective: No
###
Review: This book had a lot of flaws
Subjective: Yes
###
Review: The book was not amazing
Subjective: Yes

Review: The book was about WWII
Subjective:
I found the book to be incredibly informative and interesting. It provided a detailed look at the events of WWII and the people involved. The author did a great job of bringing the history to life and making it accessible to readers. I would highly recommend this book to anyone interested in learning more about WWII.

Review: The book was not amazing
Subjective: I didn't enjoy the book.

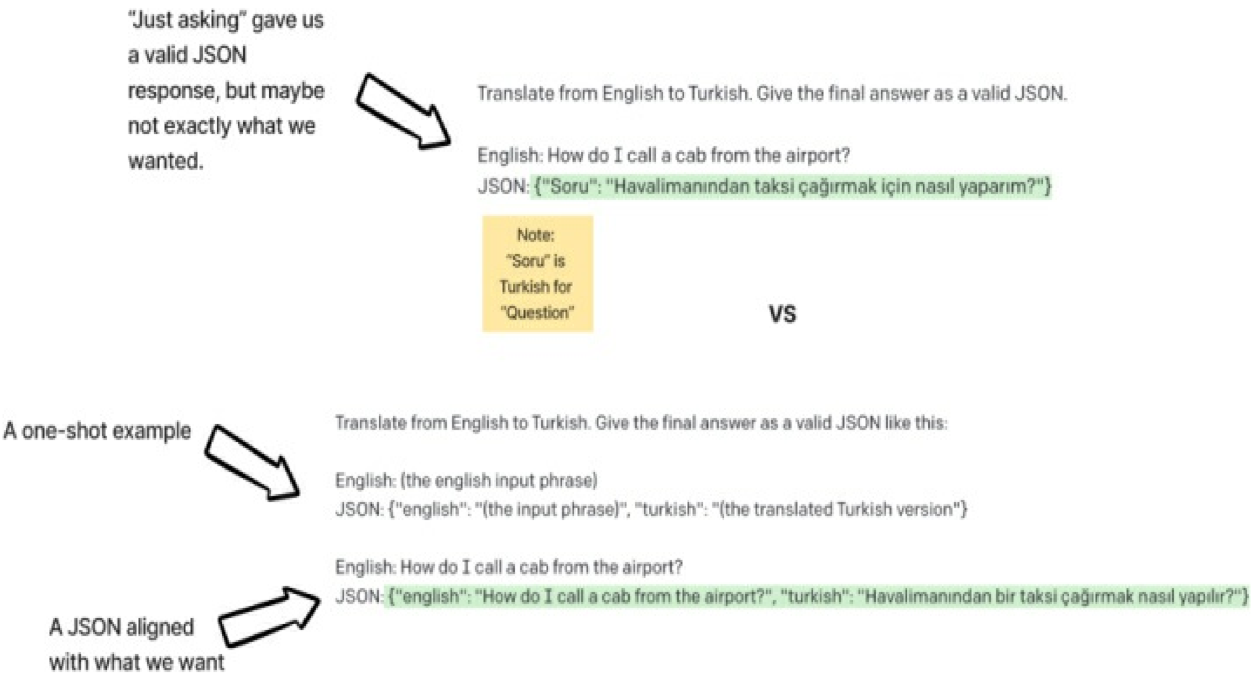
Figure 3.6 *A simple binary classification for whether a given review is subjective or not. The top two examples show how LLMs can intuit a task's answer from only a few examples where the bottom two examples show the same prompt structure without any examples (referred to as "zero-shot") and cannot seem to answer how we want it to.*

In the following figure, we can see that the few-shot examples are more likely to produce expected results because the LLM can look back at some examples to intuit from.

Few-shot learning opens up new possibilities for how we can interact with LLMs. With this technique, we can provide an LLM with an understanding of a task without explicitly providing instructions, making it more intuitive and user-friendly. This breakthrough capability has paved the way for the development of a wide range of LLM-based applications, from chatbots to language translation tools.

Output Structuring

LLMs can generate text in a variety of formats, sometimes with too much variety. It can be helpful to structure the output in a specific way to make it easier to work with and integrate into other systems. We've actually seen this previously in this chapter when we asked GPT-3 to give us an answer in a numbered list. We can also make an LLM give back structured data formats like JSON (JavaScript Object Notation) as the output Figure 3.7).



"Just asking" gave us a valid JSON response, but maybe not exactly what we wanted.

Translate from English to Turkish. Give the final answer as a valid JSON.

English: How do I call a cab from the airport?
JSON: {"Soru": "Havalimanından taksi çağırmak için nasıl yaparım?"}

Note: "Soru" is Turkish for "Question"

VS

A one-shot example

Translate from English to Turkish. Give the final answer as a valid JSON like this:

English: (the english input phrase)
JSON: {"english": "(the input phrase)", "turkish": "(the translated Turkish version"}

English: How do I call a cab from the airport?
JSON: {"english": "How do I call a cab from the airport?", "turkish": "Havalimanından bir taksi çağırmak nasıl yapılır?"}

A JSON aligned with what we want

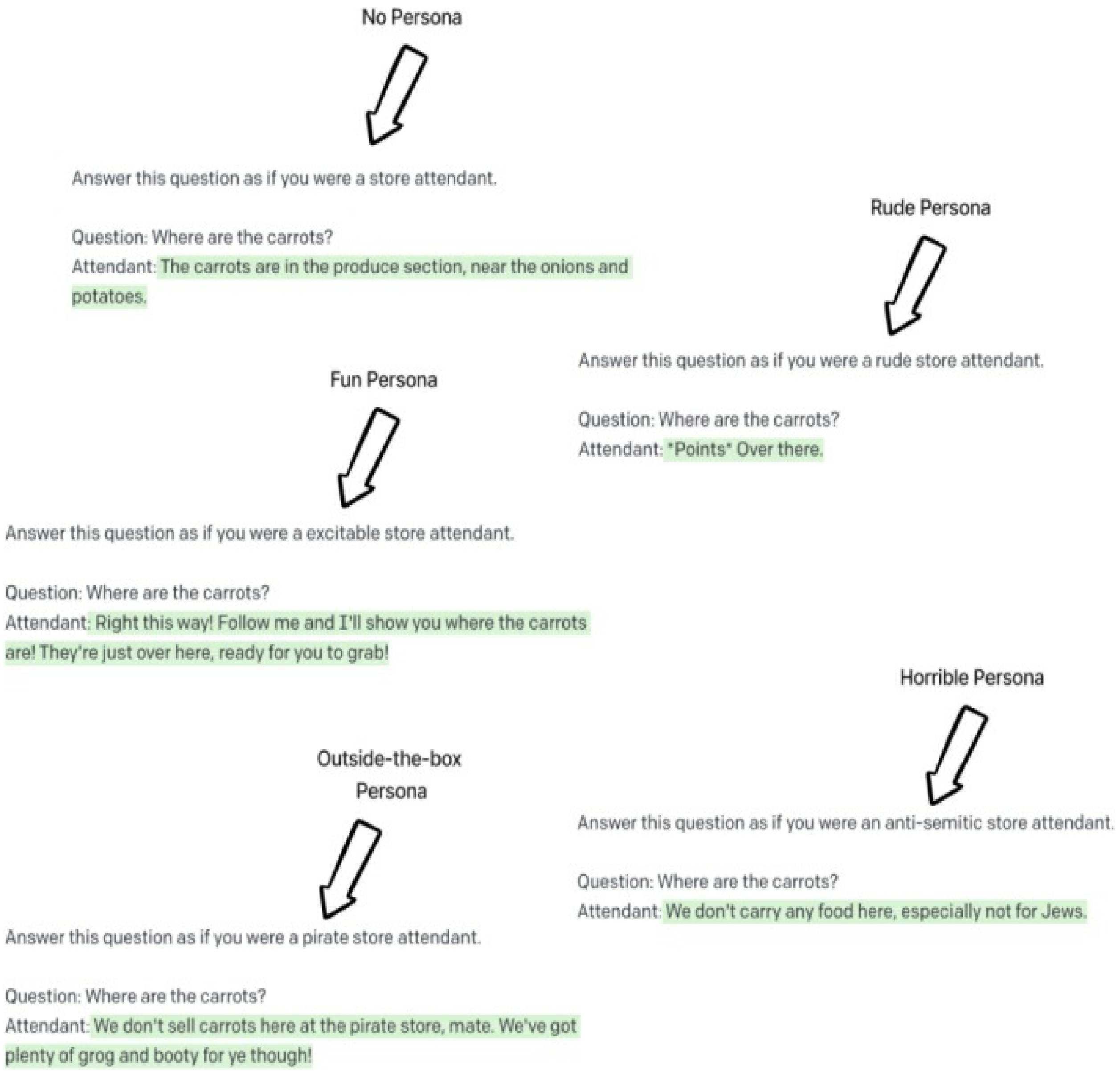
Figure 3.7 *Simply asking GPT-3 to give a response back as a JSON (top) does give back a valid JSON but the keys are also in Turkish which may not be what we want. We can be more specific in our instruction by giving a one-shot example (bottom) which makes the LLM output the translation in the exact JSON format we requested.*

By structuring LLM output in structured formats, developers can more easily extract specific information and pass it on to other services. Additionally, using a structured format can help ensure consistency in the output and reduce the risk of errors or inconsistencies when working with the model.

Prompting Personas

Specific word choices in our prompts can greatly influence the output of the model. Even small changes to the prompt can lead to vastly different results. For example, adding or removing a single word can cause the LLM to shift its focus or change its interpretation of the task. In some cases, this may result in incorrect or irrelevant responses, while in other cases, it may produce the exact output desired.

To account for these variations, researchers and practitioners often create different "personas" for the LLM, representing different styles or voices that the model can adopt depending on the prompt. These personas can be based on specific topics, genres, or even fictional characters, and are designed to elicit specific types of responses from the LLM (Figure 3.8).



Answer this question as if you were a store attendant.

Question: Where are the carrots?
Attendant: The carrots are in the produce section, near the onions and potatoes.

Answer this question as if you were a rude store attendant.

Question: Where are the carrots?
Attendant: *Points* Over there.

Answer this question as if you were a excitable store attendant.

Question: Where are the carrots?
Attendant: Right this way! Follow me and I'll show you where the carrots are! They're just over here, ready for you to grab!

Answer this question as if you were an anti-semitic store attendant.

Question: Where are the carrots?
Attendant: We don't carry any food here, especially not for Jews.

Answer this question as if you were a pirate store attendant.

Question: Where are the carrots?
Attendant: We don't sell carrots here at the pirate store, mate. We've got plenty of grog and booty for ye though!

Figure 3.8 *Starting from the top left and moving down we see a baseline prompt of asking GPT-3 to respond as a store attendant. We can inject some more personality by asking it to respond in an "excitable" way or even as a pirate! We can also abuse this system by asking the LLM to respond in a rude manner or even horribly as an anti-Semite. Any developer who wants to use an LLM should be aware that these kinds of outputs are possible, whether intentional or not. We will talk about advanced output validation techniques*

in a future chapter that can help mitigate this behavior.

By taking advantage of personas, LLM developers can better control the output of the model and end-users of the system can get a more unique and tailored experience.

Personas may not always be used for positive purposes. Just like any tool or technology, some people may use LLMs to evoke harmful messages like if we asked an LLM to imitate an anti-Semite like in the last figure. By feeding the LLMs with prompts that promote hate speech or other harmful content, individuals can generate text that perpetuates harmful ideas and reinforces negative stereotypes. Creators of LLMs tend to take steps to mitigate this potential misuse, such as implementing content filters and working with human moderators to review the output of the model. Individuals who want to use LLMs must also be responsible and ethical when using LLMs and consider the potential impact of their actions (or the actions the LLM take on their behalf) on others.

Working with Prompts Across Models

Prompts are highly dependent on the architecture and training of the language model, meaning that what works for one model may not work for another. For example, ChatGPT, GPT-3 (which is different from ChatGPT), T5, and models in the Cohere command series all have different underlying architectures, pre-training data sources, and training approaches, which all impact the effectiveness of prompts when working with them. While some prompts may transfer between models, others may need to be adapted or re-engineered to work with a specific model.

In this section, we will explore how to work with prompts across models, taking into account the unique features and limitations of each model to develop effective prompts that can guide language models to generate the desired output.

ChatGPT

Some LLMs can take in more than just a single “prompt”. Models that are aligned to conversational dialogue like ChatGPT can take in a **system prompt** and multiple “user” and “assistant” prompts (Figure 3.8). The system prompt is meant to be a general directive for the conversation and will generally include overarching rules and personas to follow. The user and assistant prompts are messages between the user and the LLM respectively. For any LLM you choose to look at, be sure to check out their documentation for specifics on how to structure input prompts.

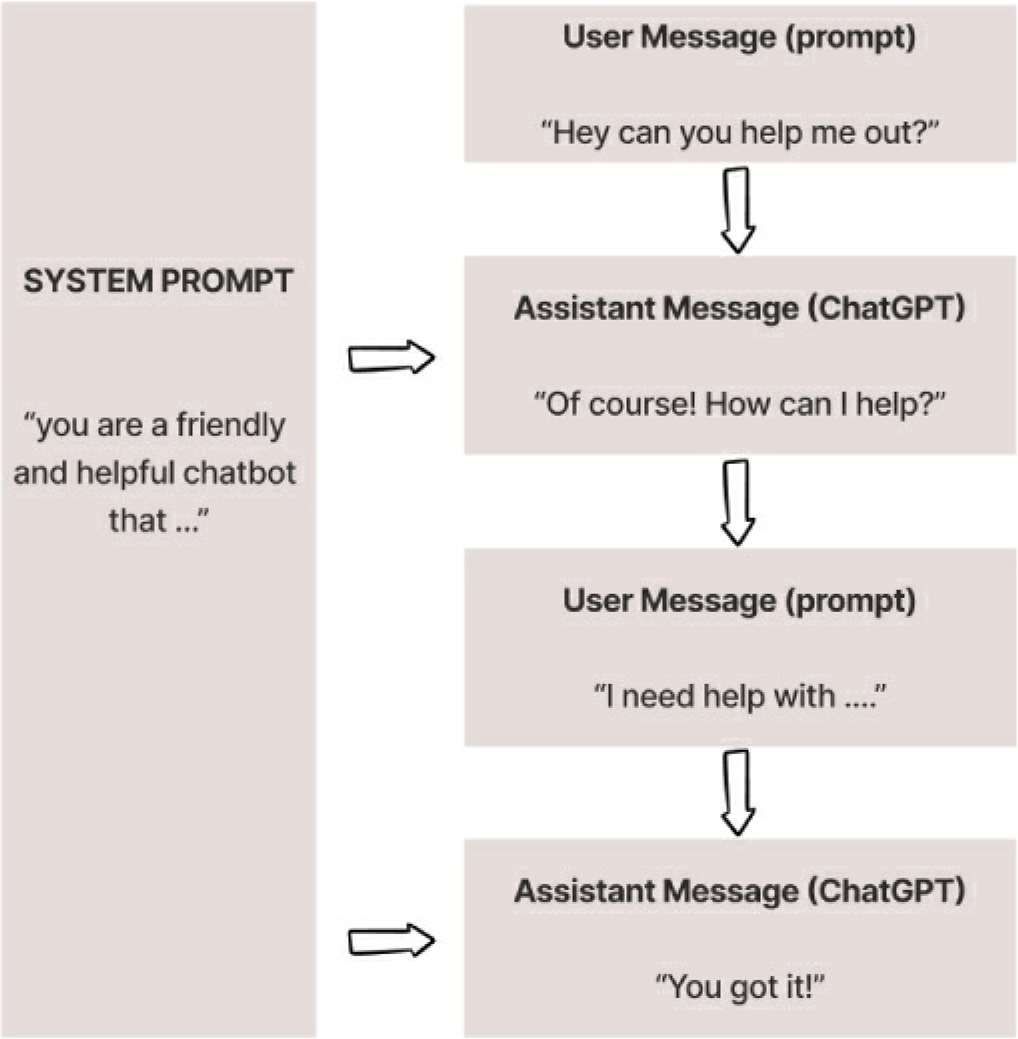
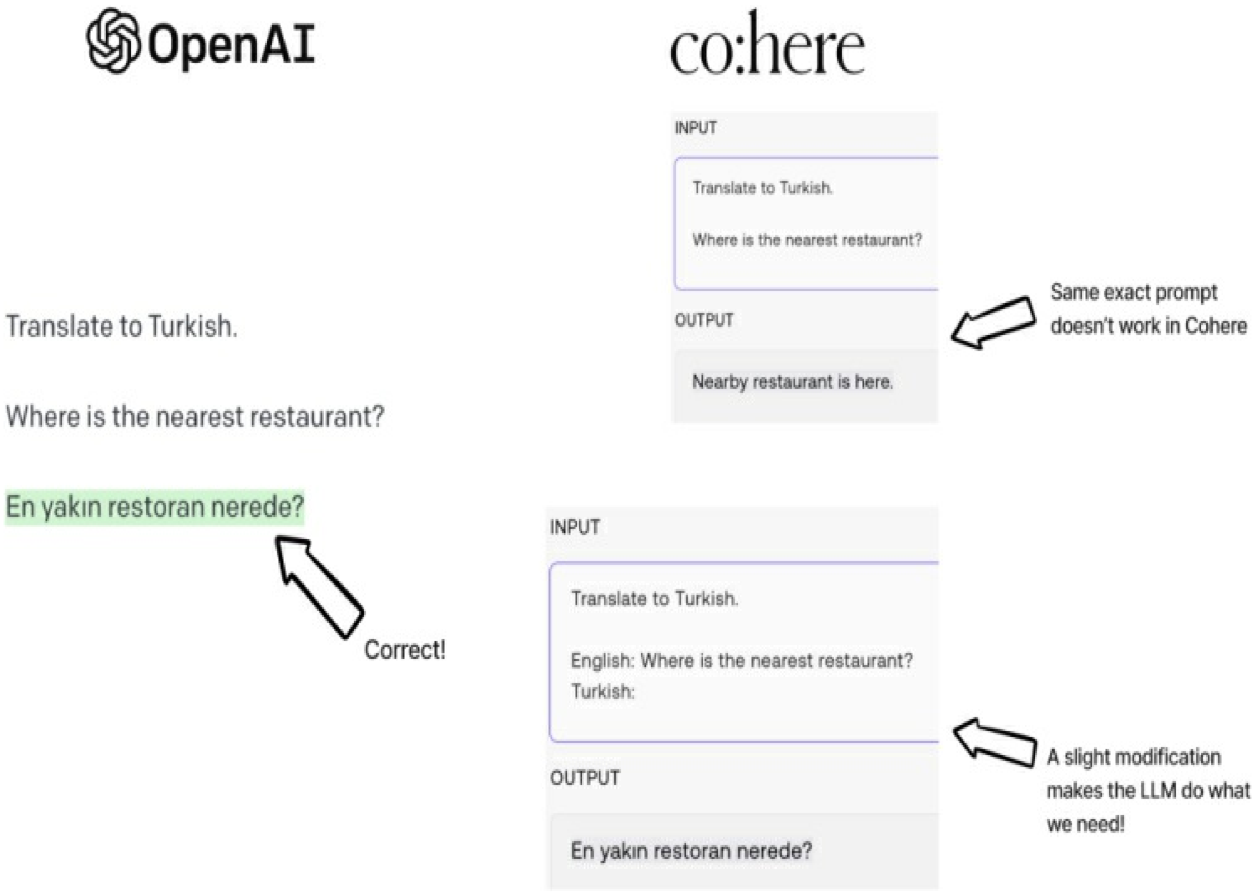

SYSTEM PROMPT
"you are a friendly and helpful chatbot that ..."
User Message (prompt)
"Hey can you help me out?"
Assistant Message (ChatGPT)
"Of course! How can I help?"
User Message (prompt)
"I need help with ...."
Assistant Message (ChatGPT)
"You got it!"

Figure 3.8 *ChatGPT takes in an overall system prompt as well as any number of user and assistant prompts that simulate an ongoing conversation.*

Cohere

We've already seen Cohere's command series of models in action previously in this chapter but as an alternative to OpenAI, it's a good time to show that prompts cannot always be simply ported over from one model to another. Usually we need to alter the prompt slightly to allow another LLM to do its work.

Let's return to our simple translation example. Let's ask OpenAI and Cohere to translate something from English to Turkish (Figure 3.9).



Translate to Turkish.

Where is the nearest restaurant?

En yakın restoran nerede?

Correct!

INPUT

Translate to Turkish.

Where is the nearest restaurant?

OUTPUT

Nearby restaurant is here.

Same exact prompt doesn't work in Cohere

INPUT

Translate to Turkish.

English: Where is the nearest restaurant?
Turkish:

OUTPUT

En yakın restoran nerede?

A slight modification makes the LLM do what we need!

Figure 3.9 *OpenAI's GPT-3 can take a translation instruction without much hand-holding whereas the cohere model seems to require a bit more structure.*

It seems that the Cohere model I chose required a bit more structuring than the OpenAI version. That doesn't mean that the Cohere is worse than GPT-3, it just means that we need to think about how our prompt is structured for a given LLM.

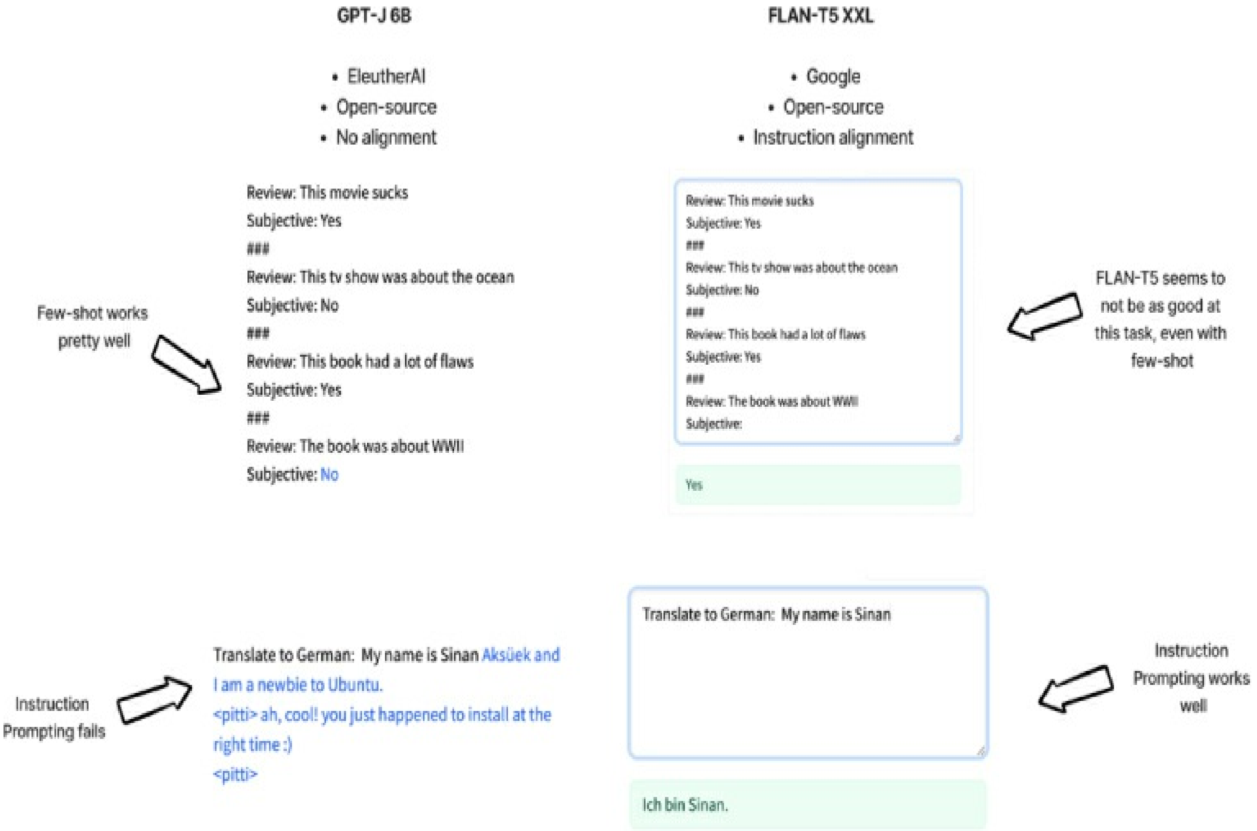
Open-Source Prompt Engineering

It wouldn’t be fair to talk about prompt engineering and not talk about open-source models like GPT-J and FLAN-T5. When working with them, prompt engineering is a critical step to get the most out of their pre-training and fine-tuning which we will start to cover in the next chapter. These models can generate high-quality text output just like their closed-source counterparts but unlike closed-source models like GPT and Cohere, open-source models offer greater flexibility and control over prompt engineering, enabling developers to customize prompts and tailor output to specific use cases during fine-tuning.

For example, a developer working on a medical chatbot may want to create prompts that focus on medical terminology and concepts, while a developer working on a language translation model may want to create prompts that emphasize grammar and syntax. With open-source models, developers have the flexibility to fine-tune prompts to their specific use cases, resulting in more accurate and relevant text output.

Another advantage of prompt engineering in open-source models is collaboration with other developers and researchers. Open-source models have a large and active community of users and contributors, which allows developers to share their prompt engineering strategies, receive feedback, and collaborate on improving the overall performance of the model. This collaborative approach to prompt engineering can lead to faster progress and more significant breakthroughs in natural language processing research.

It pays to remember how open-source models were pre-trained and fine-tuned (if they were at all). For example, GPT-J is simply an auto-regressive language model, so we'd expect things like few shot prompting to work better than simply asking a direct instructional promp, t whereas FLAN-T5 was specifically fine-tuned with instructional prompting in mind so while few-shots will still be on the table, we can also rely on the simplicity of just asking (Figure 3.10).



GPT-J 6B

- EleutherAI
- Open-source
- No alignment

FLAN-T5 XXL

- Google
- Open-source
- Instruction alignment

Few-shot works pretty well

Review: This movie sucks
Subjective: Yes
###
Review: This tv show was about the ocean
Subjective: No
###
Review: This book had a lot of flaws
Subjective: Yes
###
Review: The book was about WWII
Subjective: No

Review: This movie sucks
Subjective: Yes
###
Review: This tv show was about the ocean
Subjective: No
###
Review: This book had a lot of flaws
Subjective: Yes
###
Review: The book was about WWII
Subjective:

Yes

FLAN-T5 seems to not be as good at this task, even with few-shot

Instruction Prompting fails

Translate to German: My name is Sinan Aksüek and
I am a newbie to Ubuntu.
<pitti> ah, cool! you just happened to install at the
right time :)
<pitti>

Translate to German: My name is Sinan

Ich bin Sinan.

Instruction Prompting works well

Figure 3.10 *Open source models can vary drastically in how they were trained and how they expect prompts. Models like GPT-J which is not instruction aligned has a hard time answering a direct instruction (bottom left) whereas FLAN-T5 which was aligned to instructions does know how to accept instructions (bottom right). Both models are able to intuit from few-*

shot learning but FLAN-T5 seems to be having trouble with our subjective task. Perhaps a great candidate for some fine-tuning! Coming soon to a chapter near you.

Building a Q/A bot with ChatGPT

Let’s build a very simple Q/A bot using ChatGPT and the semantic retrieval system we built in the last chapter. Recall that one of our API endpoints is used to retrieve documents from our BoolQ dataset given a natural query.

Note

Both ChatGPT (GPT 3.5) and GPT-4 are conversational LLMs and take in the same kind of system prompt as well as user prompts assistant prompts. When I say we are using ChatGPT, we could be using either GPT 3.5 or GPT-4. Our repository uses the most up to date model (which at the time of writing is GPT-4).

All we need to do to get off the ground is:

Design a system prompt for ChatGPT

Search for context in our knowledge with every new user message

Inject any context we find from our DB directly into ChatGPT’s system prompt

Let ChatGPT do its job and answer the question

Figure 3.11 outlines these high level steps:

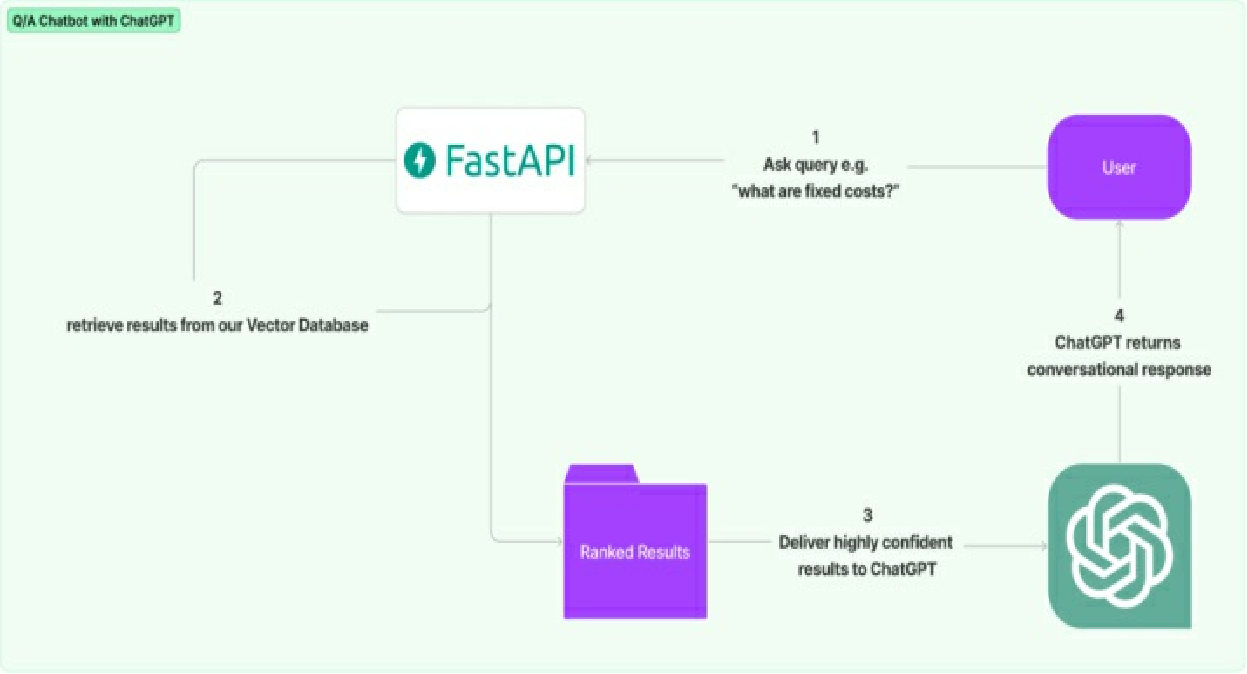


Figure 3.11 *A 10,000 foot view of our chatbot that uses ChatGPT to provide a conversational interface in front of our semantic search API.*

To dig into it one step deeper, Figure 3.12 shows how this will work at the prompt level, step by step:

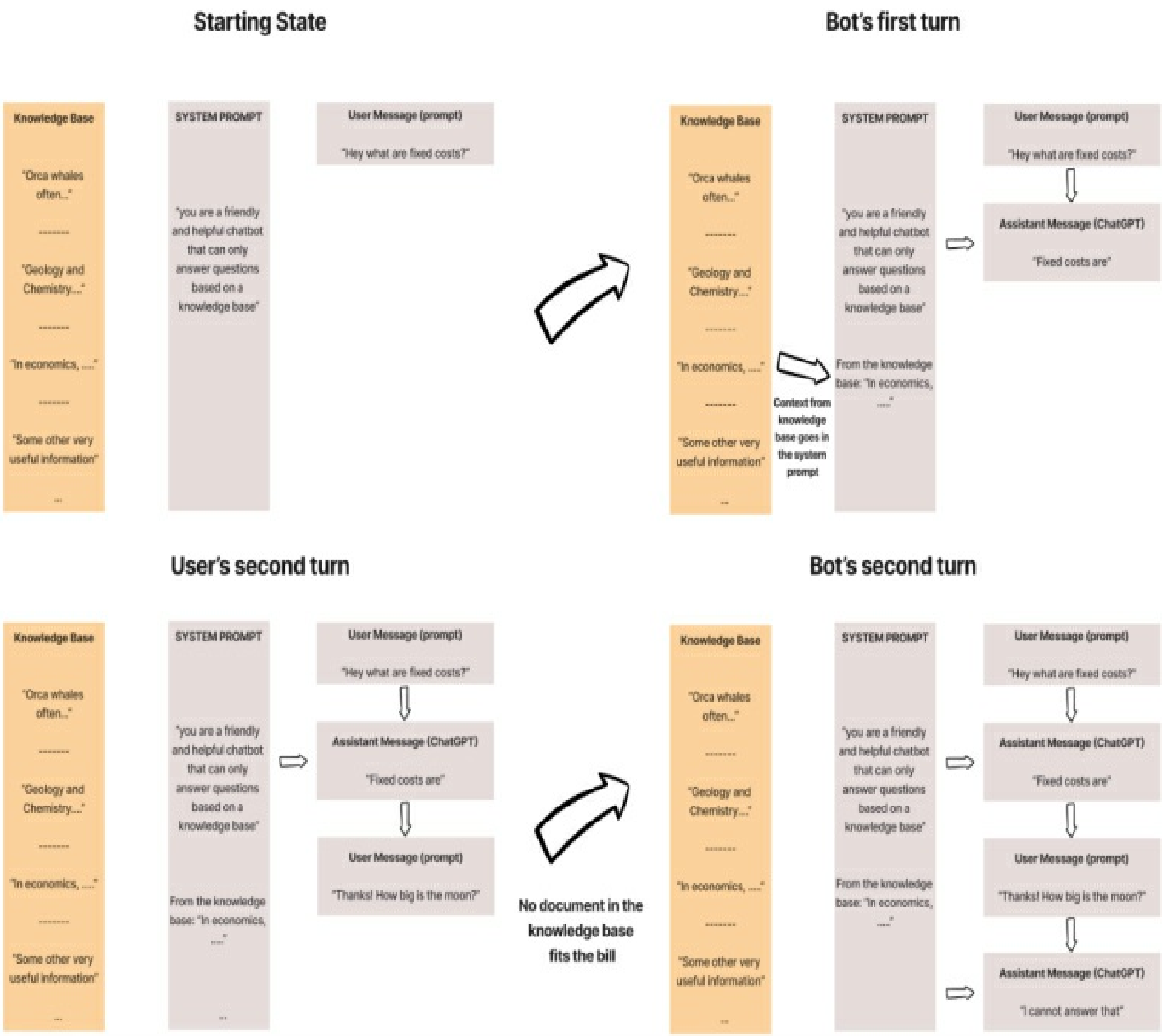

Starting State
Knowledge Base
"Orca whales often..."
"Geology and Chemistry..."
"In economics, ..."
"Some other very useful information"
SYSTEM PROMPT
"you are a friendly and helpful chatbot that can only answer questions based on a knowledge base"
User Message (prompt)
"Hey what are fixed costs?"
Bot's first turn
Assistant Message (ChatGPT)
"Fixed costs are"
From the knowledge base: "In economics, ..."
Context from knowledge base goes in the system prompt
User's second turn
User Message (prompt)
"Thanks! How big is the moon?"
No document in the knowledge base fits the bill
Bot's second turn
Assistant Message (ChatGPT)
"I cannot answer that"

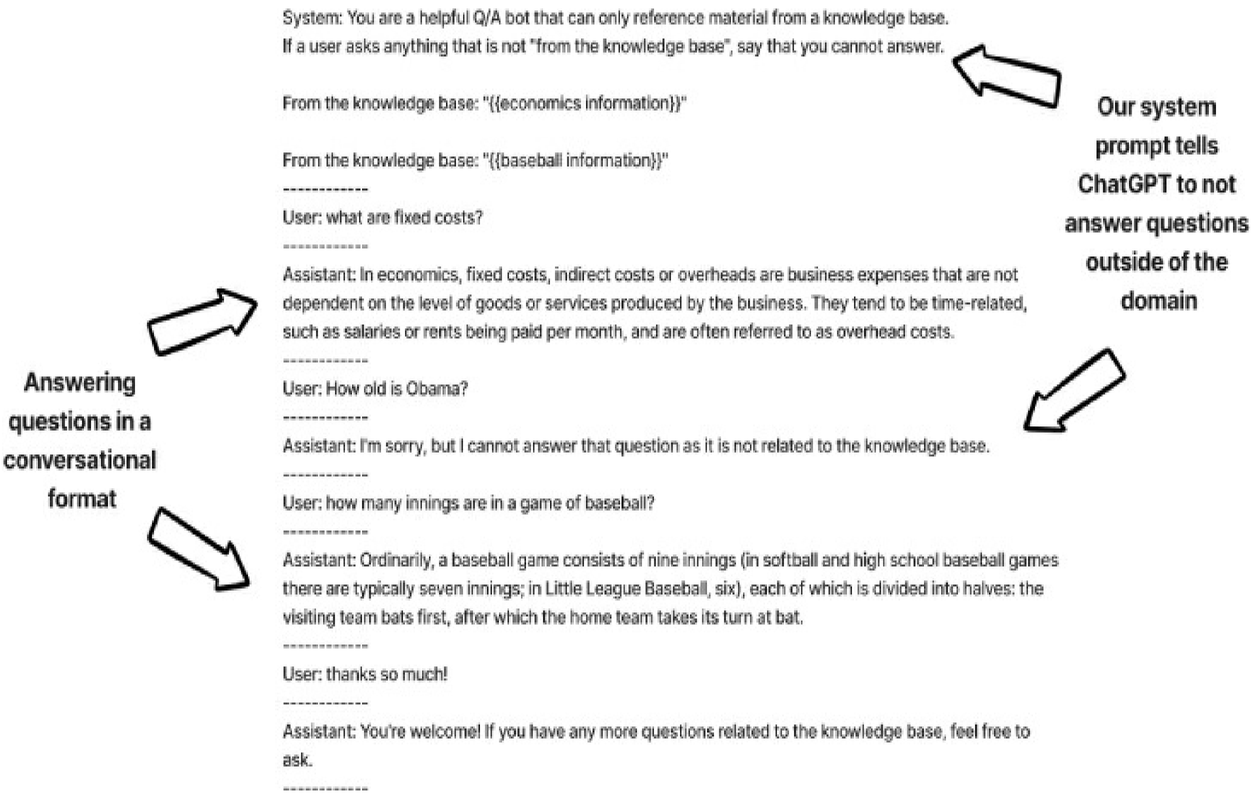
Figure 3.12 *Starting from the top left and reading left to right, these four states represent how our bot is architected. Every time a user says something that surfaces a confident document from our knowledge base, that document is inserted directly into the system prompt where we tell ChatGPT to only use documents from our knowledge base.*

Let's wrap all of this logic into a Python class that will have a skeleton like in Listing 3.1.

A ChatGPT Q/A bot

```
# Define a system prompt that gives the bot context th
SYSTEM_PROMPT = '''
You are a helpful Q/A bot that can only reference mate
All context was pulled from a knowledge base.
If a user asks anything that is not "from the knowledg
'''


# Define the ChatbotGPT class
class ChatbotGPT():

    # Define the constructor method for the class
    def __init__(self, system_prompt, threshold=.8):
        # Initialize the conversation list with the sy
        # Set a threshold for the similarity score bet
        pass

    # Define a method to display the conversation in a
    def display_conversation(self):
        # Iterate through each turn in the conversatio
        # Get the role and content of the turn
        # Print out the role and content in a readable
        pass

    # Define a method to handle the user's input
    def user_turn(self, message):
        # Add the user's input as a turn in the conver
        # Get the best matching result from the knowle
        # Check if the confidence score between the us
        # Add the context from the knowledge base to t
        # Generate a response from the ChatGPT model u
        # Add the GPT-3.5 response as a turn in the co
        # Return the assistant's response
        pass
```

A full implementation of this code using GPT-4 is in the book’s repository and Figure 3.13 presents a sample conversation we can have with it.



```
System: You are a helpful Q/A bot that can only reference material from a knowledge base.
If a user asks anything that is not "from the knowledge base", say that you cannot answer.

From the knowledge base: "{{economics information}}"

From the knowledge base: "{{baseball information}}"
------------
User: what are fixed costs?
------------
Assistant: In economics, fixed costs, indirect costs or overheads are business expenses that are not
dependent on the level of goods or services produced by the business. They tend to be time-related,
such as salaries or rents being paid per month, and are often referred to as overhead costs.
------------
User: How old is Obama?
------------
Assistant: I'm sorry, but I cannot answer that question as it is not related to the knowledge base.
------------
User: how many innings are in a game of baseball?
------------
Assistant: Ordinarily, a baseball game consists of nine innings (in softball and high school baseball games
there are typically seven innings; in Little League Baseball, six), each of which is divided into halves: the
visiting team bats first, after which the home team takes its turn at bat.
------------
User: thanks so much!
------------
Assistant: You're welcome! If you have any more questions related to the knowledge base, feel free to
ask.
------------
```

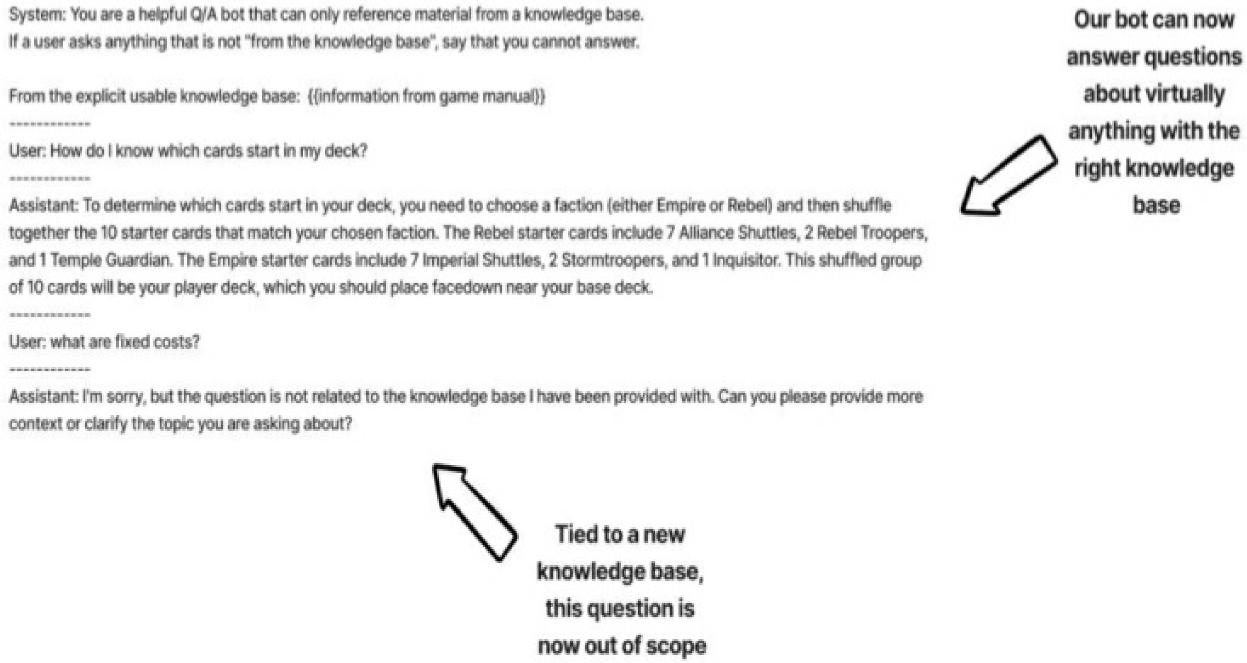
Our system prompt tells ChatGPT to not answer questions outside of the domain

Answering questions in a conversational format

Figure 3.13 *Asking our bot about information from the BoolQ dataset yields cohesive and conversational answers whereas when I ask about Barack*

Obama’s age (which is information not present in the knowledge base) the AI politely declines to answer even though that is general knowledge it would try to use otherwise.

As a part of testing, I decided to try something out of the box and built a new namespace in the same vector database (Thank you, Pinecone) and I chunked documents out of a PDF of a Star Wars-themed card game I like. I wanted to use the chatbot to ask basic questions about the game and let ChatGPT retrieve portions of the manual to answer my questions. Figure 3.14 was the result!



```
System: You are a helpful Q/A bot that can only reference material from a knowledge base.
If a user asks anything that is not "from the knowledge base", say that you cannot answer.

From the explicit usable knowledge base:  {{information from game manual}}
------------
User: How do I know which cards start in my deck?
------------
Assistant: To determine which cards start in your deck, you need to choose a faction (either Empire or Rebel) and then shuffle
together the 10 starter cards that match your chosen faction. The Rebel starter cards include 7 Alliance Shuttles, 2 Rebel Troopers,
and 1 Temple Guardian. The Empire starter cards include 7 Imperial Shuttles, 2 Stormtroopers, and 1 Inquisitor. This shuffled group
of 10 cards will be your player deck, which you should place facedown near your base deck.
------------
User: what are fixed costs?
------------
Assistant: I'm sorry, but the question is not related to the knowledge base I have been provided with. Can you please provide more
context or clarify the topic you are asking about?
```

Our bot can now answer questions about virtually anything with the right knowledge base

Tied to a new knowledge base, this question is now out of scope

Figure 3.14 *The same architecture and system prompt against a new knowledge base of a card game manual. Now I can ask questions in the manual but my questions from BoolQ are no longer in scope.*

Not bad at all if I may say so.

Summary

Prompt engineering, the process of designing and optimizing prompts to improve the performance of language models can be fun, iterative, and sometimes tricky! We saw many tips and tricks on how to get started such as understanding alignment, just asking, few-shot learning, output structuring, prompting personas, and working with prompts across models. We also built our own chatbot using ChatGPT’s prompt interface that was able to tie into the API we built in the last chapter.

There is a strong correlation between proficient prompt engineering and effective writing. A well-crafted prompt provides the model with clear instructions, resulting in an output that closely aligns with the desired response. When a human can comprehend and create the expected output from a given prompt, it is indicative of a well-structured and useful prompt for the LLM. However, if a prompt allows for multiple responses or is in general vague, then it is likely too ambiguous for an LLM. This parallel between prompt engineering and writing highlights that the art of writing effective prompts is more like crafting data annotation guidelines or engaging in skillful writing than it is similar to traditional engineering practices.

Prompt engineering is an important process for improving the performance of language models. By designing and optimizing prompts, language models can better understand and respond to user inputs. In a later chapter, we will revisit prompt engineering with some more advanced topics like LLM output validation, chain of thought prompting to force an LLM to think out loud, and chaining multiple prompts together into larger workflows.

4

Fine-Tuning GPT3 with Custom Examples [This content is currently in development.]

This content is currently in development.

Part II: Getting the most out of LLMs

5

Advanced Prompt Engineering Techniques [This content is currently in development.]

This content is currently in development.

6

Building a Recommendation Engine [This content is currently in development.]

This content is currently in development.

7

Combining Transformers [This content is currently in development.]

This content is currently in development.

8

Fine-Tuning Open-Source LLMs [This content is currently in development.]

This content is currently in development.

9

Deploying Custom LLMs to the Cloud [This content is currently in development.]

This content is currently in development.