2. [25 points]

Building a Code Query System with LLMs and LangChain for GitHub Repositories for Apache Spark, and other codebases that you manually examined earlier.

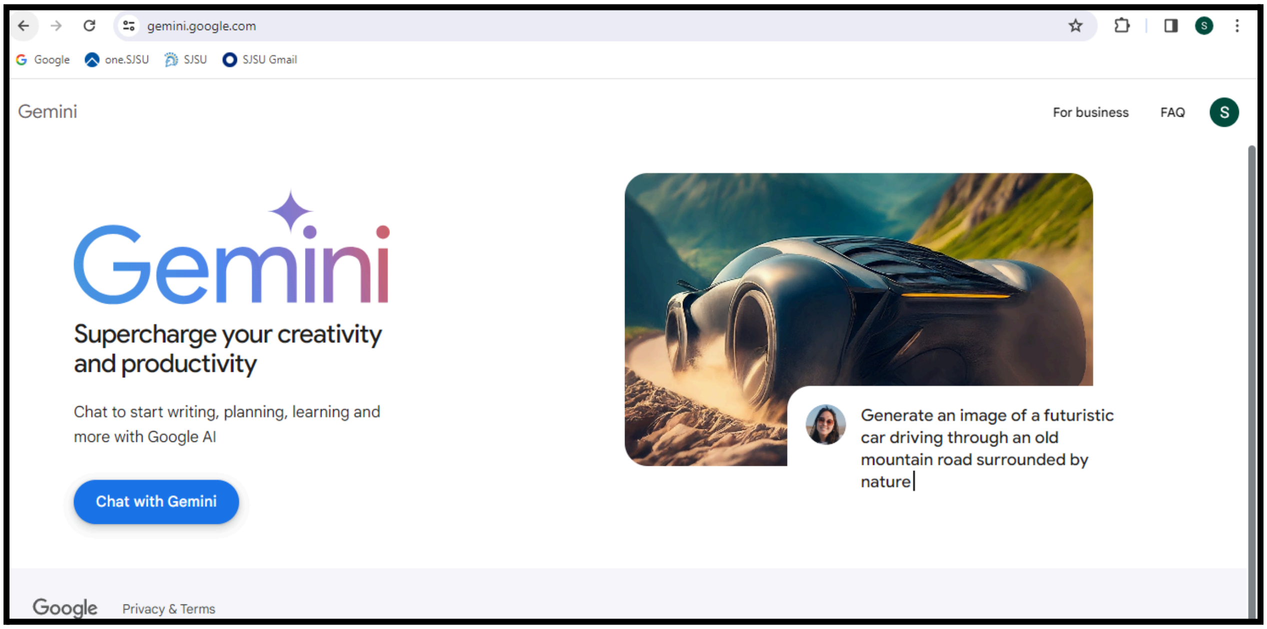
Objectives:

- Gain practical experience with LLMs (Large Language Models) and LangChain for code comprehension.
- Develop a system that can answer user queries about the code within the Apache Spark and other GitHub repositories, in a way automating some of what you did in earlier HW.

2.1 [5 points] Setup

Choose an LLM:

Gemini is Google's newest family of Large Language Models (LLMs). The Gemini suite currently houses 3 different model sizes: Nano, Pro, and Ultra.



Supercharge your creativity and productivity

Chat to start writing, planning, learning and more with Google AI

Generate an image of a futuristic car driving through an old mountain road surrounded by nature

Advantage:

High-Quality Text Generation: Gemini, being a state-of-the-art language model, can generate high-quality text across various domains. It can understand context, generate coherent responses, and provide valuable insights.

Versatility: Gemini can be used for a wide range of **natural language processing (NLP) tasks, including text generation, summarization, translation, question answering, and more**. Its versatility makes it suitable for diverse applications.

Efficiency: Gemini is capable of **generating text quickly**, which can be advantageous in scenarios where real-time or near-real-time responses are required. It can handle large volumes of text efficiently.

Customization: Gemini allows fine-tuning on specific datasets or tasks, enabling users to customize its behavior for particular applications. **Fine-tuning can improve performance on domain-specific tasks.**

Access to Latest Research: OpenAI regularly updates its models based on the latest research advancements. Users of Gemini benefit from these updates, which may include improvements in text quality, efficiency, and capabilities.

Disadvantage:

Compute Resources: Training and fine-tuning large language models like **Gemini require significant computational resources, including powerful hardware and large-scale datasets**. Access to such resources may be limited for some users.

Environmental Impact: **Training and using large language models consume a considerable amount of energy,** contributing to carbon emissions and environmental impact. This aspect raises concerns about the sustainability of AI development.

Data Privacy: Large language models like Gemini are trained on vast amounts of text data, which may include sensitive or private information. There are concerns about data privacy and potential misuse of models for unethical purposes.

Bias and Fairness: Language models trained on diverse datasets may inherit biases present in the training data. Gemini may exhibit biases related to gender, race, or other sensitive attributes, which can lead to biased outputs and reinforce stereotypes.

Ethical Considerations: The widespread adoption of large language models raises ethical concerns related to misinformation, manipulation, and unintended consequences. Users of Gemini must be mindful of ethical considerations and responsible AI practices.

Learning from LLM and Langchain:

Language Understanding: Interacting with LLMs and LangChains can improve language comprehension skills as users formulate prompts and questions, and analyze responses to understand the model's reasoning and capabilities.

Creative Writing: Experimenting with LLMs and LangChains can foster creativity by generating stories, poems, or dialogues based on prompts. Users can explore different writing styles, genres, and narrative structures, enhancing their creative writing abilities.

Problem Solving: Using LLMs and LangChains to tackle complex problems can sharpen problem-solving skills. Users can formulate queries, analyze generated responses, and iterate on their approach to find solutions or insights across various domains.

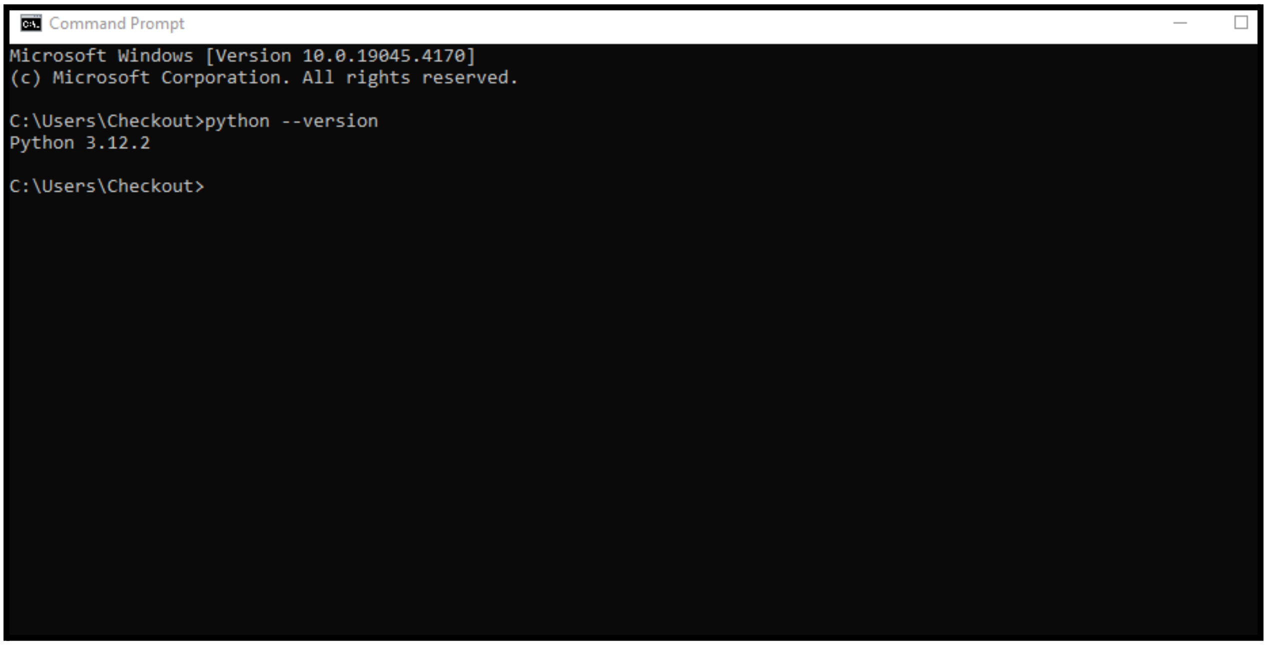
Critical Thinking: Evaluating the quality and relevance of responses generated by LLMs and LangChains encourages critical thinking. Users learn to discern between plausible and implausible answers, identify biases or inaccuracies, and refine their questions to elicit meaningful responses.

Programming and Automation: Leveraging LLMs and LangChains for code generation and automation tasks can enhance programming skills. Users can explore code snippets, generate scripts, and automate repetitive tasks, improving their coding proficiency and efficiency.

Research and Knowledge Acquisition: Engaging with LLMs and LangChains can facilitate research and knowledge acquisition across diverse topics. Users can pose questions, explore explanations, and delve into complex concepts, expanding their understanding and expertise in various domains.

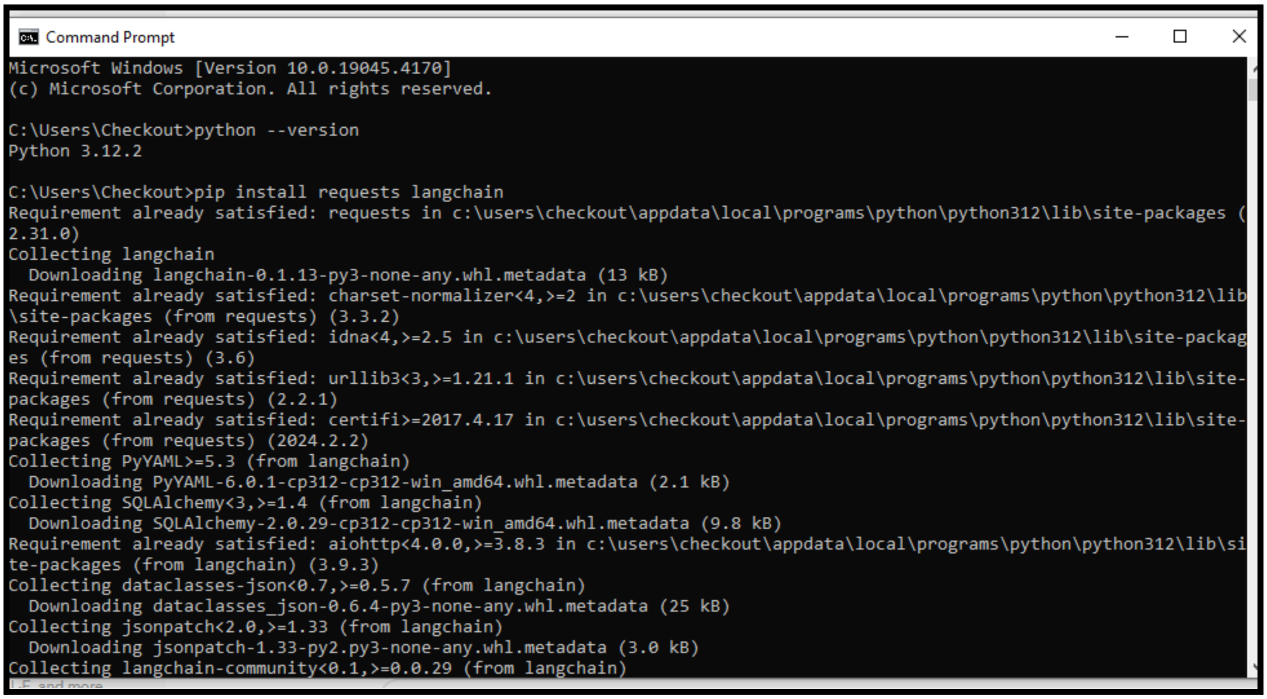
Set Up Your Development Environment:

- Install Python (version 3.6 or later recommended).
- Install required libraries using pip:

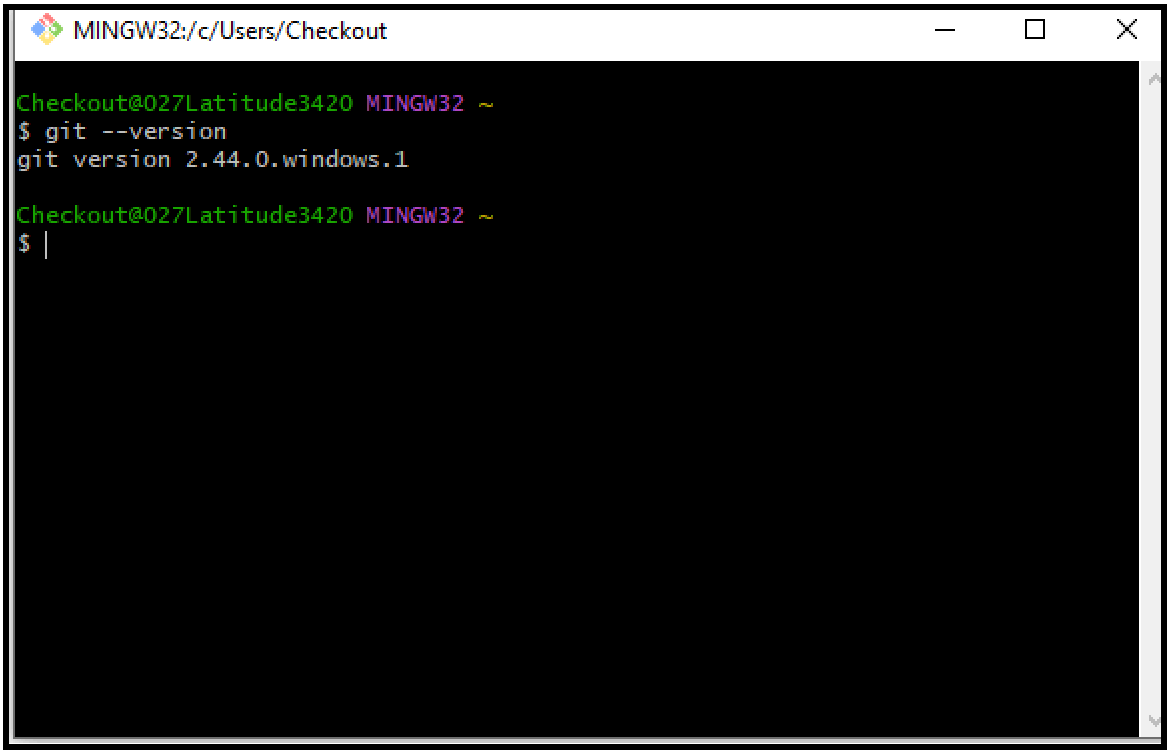


```
Microsoft Windows [Version 10.0.19045.4170]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Checkout>python --version
Python 3.12.2

C:\Users\Checkout>
```

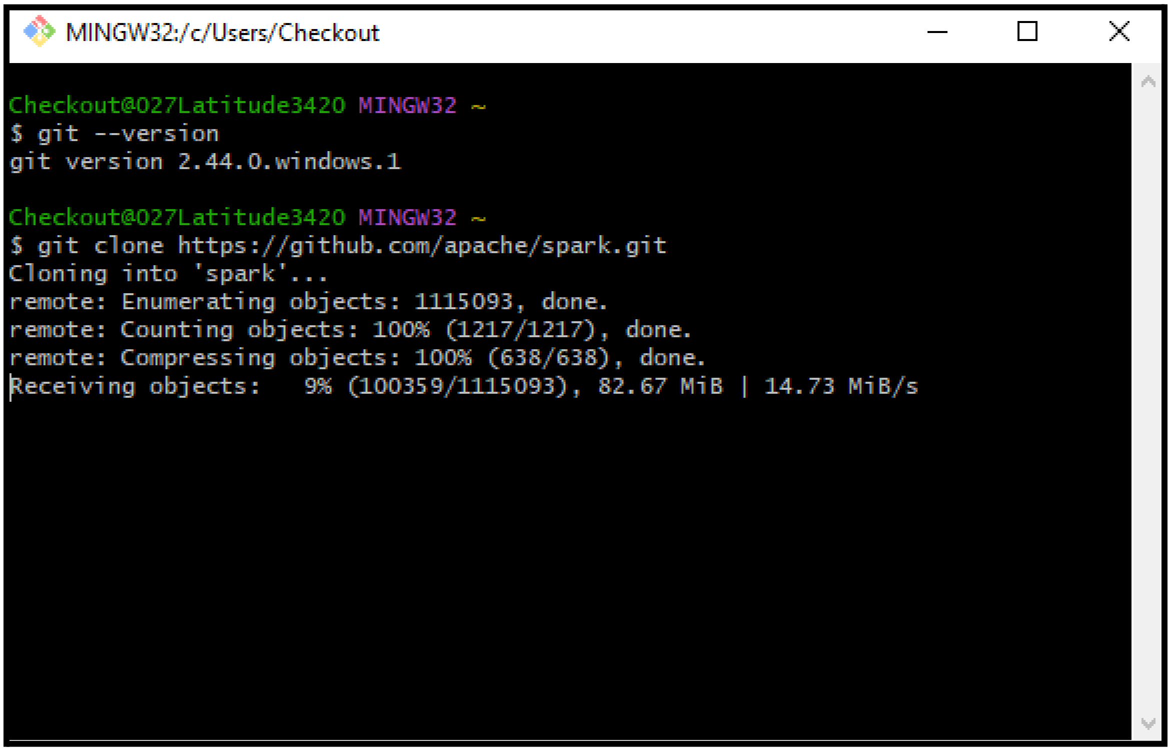
pip install requests langchain



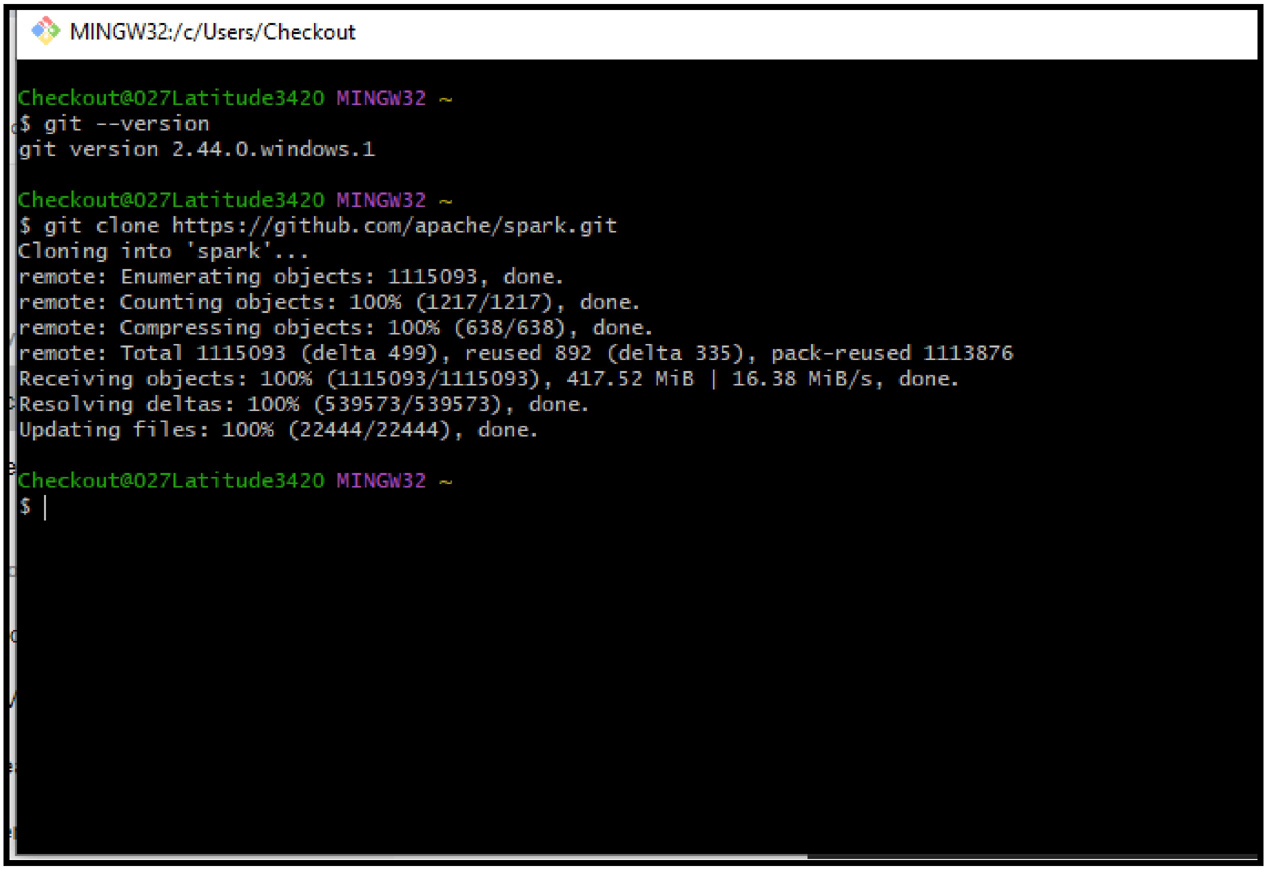
```
Microsoft Windows [Version 10.0.19045.4170]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Checkout>python --version
Python 3.12.2

C:\Users\Checkout>pip install requests langchain
Requirement already satisfied: requests in c:\users\checkout\appdata\local\programs\python\python312\lib\site-packages (
2.31.0)
Collecting langchain
  Downloading langchain-0.1.13-py3-none-any.whl.metadata (13 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\checkout\appdata\local\programs\python\python312\lib
\site-packages (from requests) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in c:\users\checkout\appdata\local\programs\python\python312\lib\site-packag
es (from requests) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\checkout\appdata\local\programs\python\python312\lib\site-
packages (from requests) (2.2.1)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\checkout\appdata\local\programs\python\python312\lib\site-
packages (from requests) (2024.2.2)
Collecting PyYAML>=5.3 (from langchain)
  Downloading PyYAML-6.0.1-cp312-cp312-win_amd64.whl.metadata (2.1 kB)
Collecting SQLAlchemy<3,>=1.4 (from langchain)
  Downloading SQLAlchemy-2.0.29-cp312-cp312-win_amd64.whl.metadata (9.8 kB)
Requirement already satisfied: aiohttp<4.0.0,>=3.8.3 in c:\users\checkout\appdata\local\programs\python\python312\lib\si
te-packages (from langchain) (3.9.3)
Collecting dataclasses-json<0.7,>=0.5.7 (from langchain)
  Downloading dataclasses_json-0.6.4-py3-none-any.whl.metadata (25 kB)
Collecting jsonpatch<2.0,>=1.33 (from langchain)
  Downloading jsonpatch-1.33-py2.py3-none-any.whl.metadata (3.0 kB)
Collecting langchain-community<0.1,>=0.0.29 (from langchain)
```



```
Checkout@027Latitude3420 MINGW32 ~
$ git --version
git version 2.44.0.windows.1

Checkout@027Latitude3420 MINGW32 ~
$ |
```

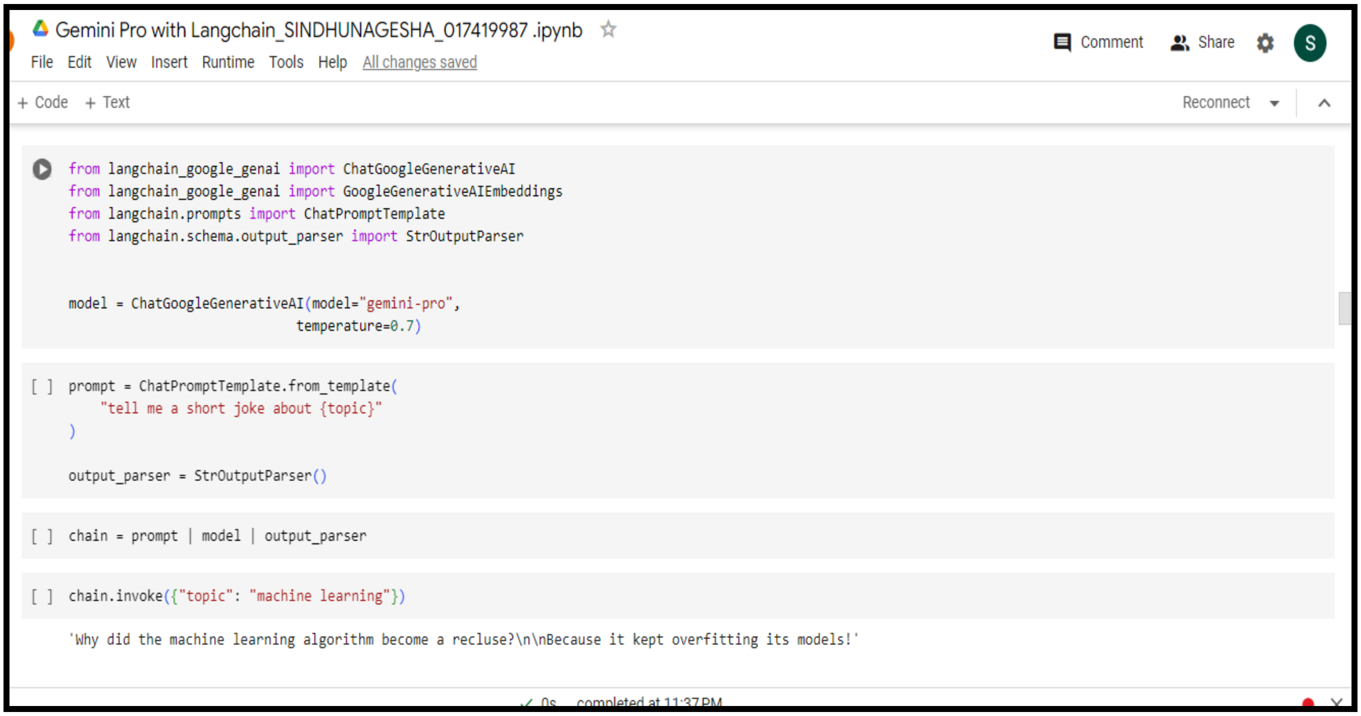


```
Checkout@027Latitude3420 MINGW32 ~
$ git --version
git version 2.44.0.windows.1

Checkout@027Latitude3420 MINGW32 ~
$ git clone https://github.com/apache/spark.git
Cloning into 'spark'...
remote: Enumerating objects: 1115093, done.
remote: Counting objects: 100% (1217/1217), done.
remote: Compressing objects: 100% (638/638), done.
Receiving objects:   9% (100359/1115093), 82.67 MiB | 14.73 MiB/s
```



```
Checkout@027Latitude3420 MINGW32 ~
$ git --version
git version 2.44.0.windows.1

Checkout@027Latitude3420 MINGW32 ~
$ git clone https://github.com/apache/spark.git
Cloning into 'spark'...
remote: Enumerating objects: 1115093, done.
remote: Counting objects: 100% (1217/1217), done.
remote: Compressing objects: 100% (638/638), done.
remote: Total 1115093 (delta 499), reused 892 (delta 335), pack-reused 1113876
Receiving objects: 100% (1115093/1115093), 417.52 MiB | 16.38 MiB/s, done.
Resolving deltas: 100% (539573/539573), done.
Updating files: 100% (22444/22444), done.

Checkout@027Latitude3420 MINGW32 ~
$ |
```

- Create a Python script to interact with the LLM and process queries.


Gemini Pro with Langchain_SINDHUNAGESHA_017419987 .ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
S
+ Code + Text
Reconnect

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_google_genai import GoogleGenerativeAIEmbeddings
from langchain.prompts import ChatPromptTemplate
from langchain.schema.output_parser import StrOutputParser


model = ChatGoogleGenerativeAI(model="gemini-pro",
                             temperature=0.7)
```

```
[ ] prompt = ChatPromptTemplate.from_template(
        "tell me a short joke about {topic}"
    )

    output_parser = StrOutputParser()
```

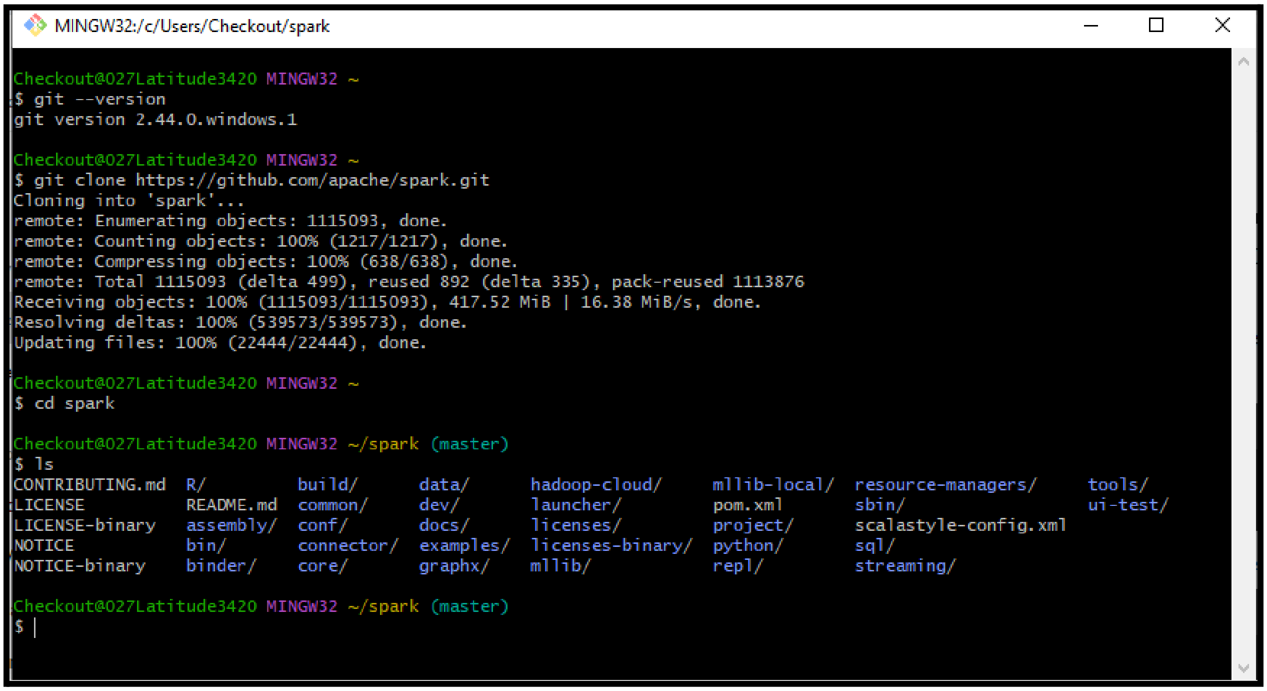
```
[ ] chain = prompt | model | output_parser
```

```
[ ] chain.invoke({"topic": "machine learning"})

    'Why did the machine learning algorithm become a recluse?\n\nBecause it kept overfitting its models!'
```

Prepare the Apache Spark GitHub Repository:

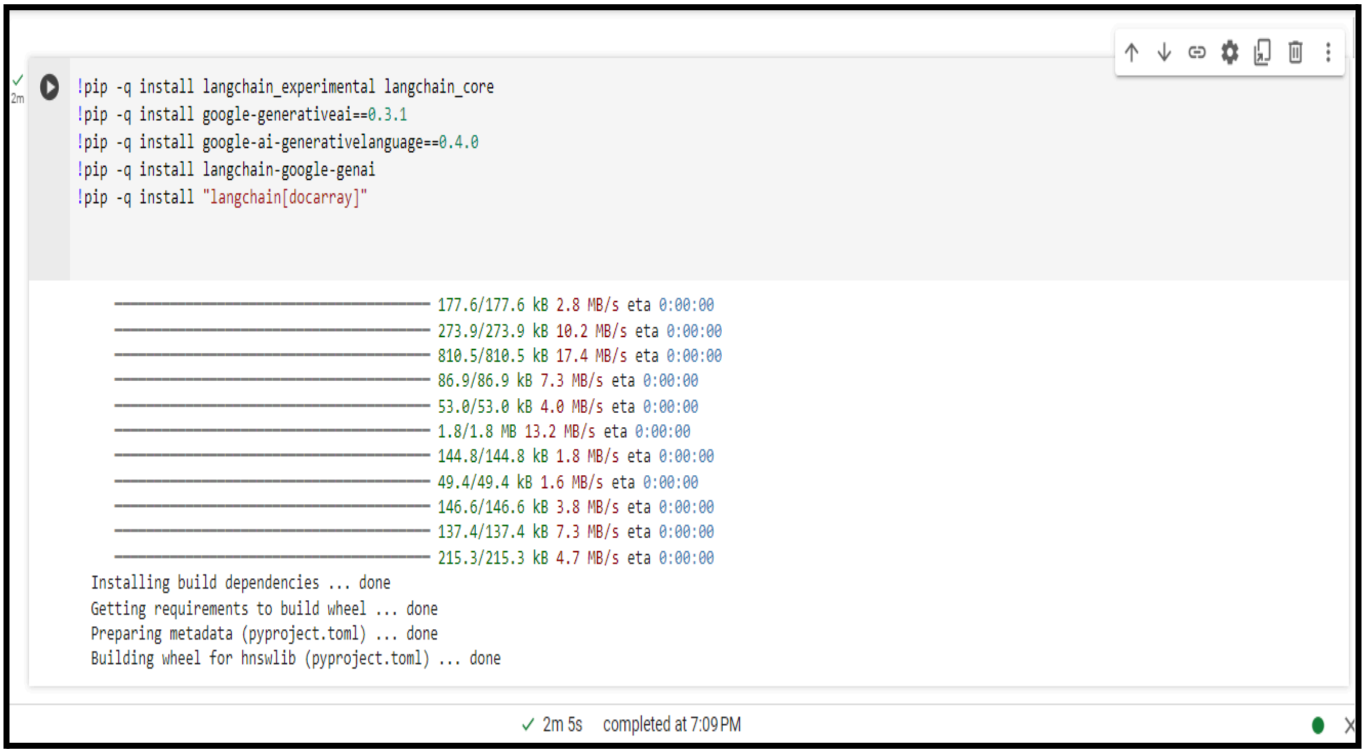
- Clone the Apache Spark and other repositories locally using Git:

  git clone https://github.com/apache/spark.git



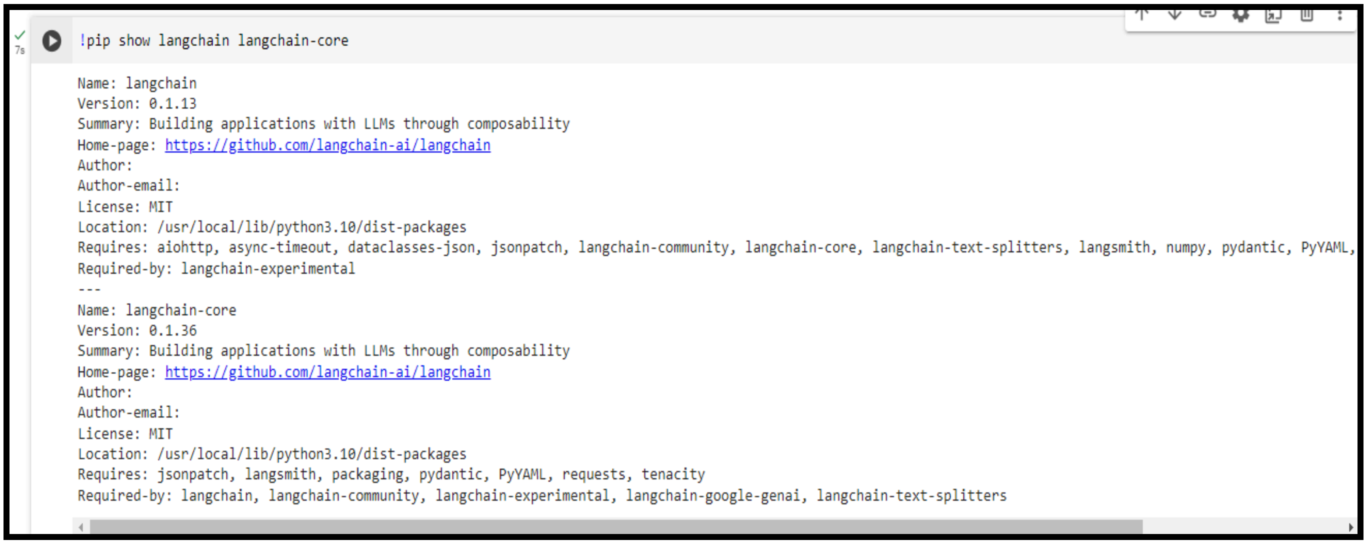
```
Checkout@027Latitude3420 MINGW32 ~
$ git --version
git version 2.44.0.windows.1

Checkout@027Latitude3420 MINGW32 ~
$ git clone https://github.com/apache/spark.git
Cloning into 'spark'...
remote: Enumerating objects: 1115093, done.
remote: Counting objects: 100% (1217/1217), done.
remote: Compressing objects: 100% (638/638), done.
remote: Total 1115093 (delta 499), reused 892 (delta 335), pack-reused 1113876
Receiving objects: 100% (1115093/1115093), 417.52 MiB | 16.38 MiB/s, done.
Resolving deltas: 100% (539573/539573), done.
Updating files: 100% (22444/22444), done.

Checkout@027Latitude3420 MINGW32 ~
$ cd spark

Checkout@027Latitude3420 MINGW32 ~/spark (master)
$ ls
CONTRIBUTING.md  R/          build/      data/      hadoop-cloud/     mllib-local/  resource-managers/     tools/
LICENSE          README.md   common/     dev/       launcher/         pom.xml       sbin/                  ui-test/
LICENSE-binary   assembly/   conf/       docs/      licenses/         project/      scalastyle-config.xml
NOTICE           bin/        connector/  examples/  licenses-binary/  python/       sql/
NOTICE-binary    binder/     core/       graphx/    mllib/            repl/         streaming/

Checkout@027Latitude3420 MINGW32 ~/spark (master)
$ |
```

Step 1: Importing required libraries:



```
!pip -q install langchain_experimental langchain_core
!pip -q install google-generativeai==0.3.1
!pip -q install google-ai-generativelanguage==0.4.0
!pip -q install langchain-google-genai
!pip -q install "langchain[docarray]"
```

```
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 177.6/177.6 kB 2.8 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 273.9/273.9 kB 10.2 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 810.5/810.5 kB 17.4 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 86.9/86.9 kB 7.3 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 53.0/53.0 kB 4.0 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 13.2 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 144.8/144.8 kB 1.8 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 49.4/49.4 kB 1.6 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 146.6/146.6 kB 3.8 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 137.4/137.4 kB 7.3 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 215.3/215.3 kB 4.7 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
  Building wheel for hnswlib (pyproject.toml) ... done
```

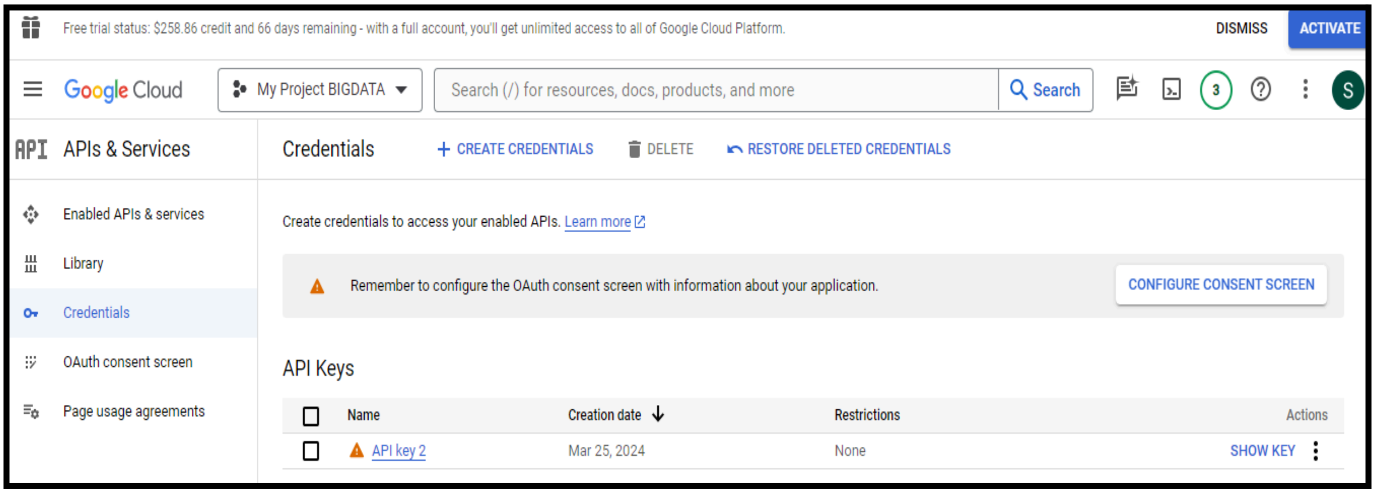


```
!pip show langchain langchain-core
```

```
Name: langchain
Version: 0.1.13
Summary: Building applications with LLMs through composability
Home-page: https://github.com/langchain-ai/langchain
Author:
Author-email:
License: MIT
Location: /usr/local/lib/python3.10/dist-packages
Requires: aiohttp, async-timeout, dataclasses-json, jsonpatch, langchain-community, langchain-core, langchain-text-splitters, langsmith, numpy, pydantic, PyYAML,
Required-by: langchain-experimental
---
Name: langchain-core
Version: 0.1.36
Summary: Building applications with LLMs through composability
Home-page: https://github.com/langchain-ai/langchain
Author:
Author-email:
License: MIT
Location: /usr/local/lib/python3.10/dist-packages
Requires: jsonpatch, langsmith, packaging, pydantic, PyYAML, requests, tenacity
Required-by: langchain, langchain-community, langchain-experimental, langchain-google-genai, langchain-text-splitters
```

and many others according to the requirement.

Step 2: Setting up Authentication



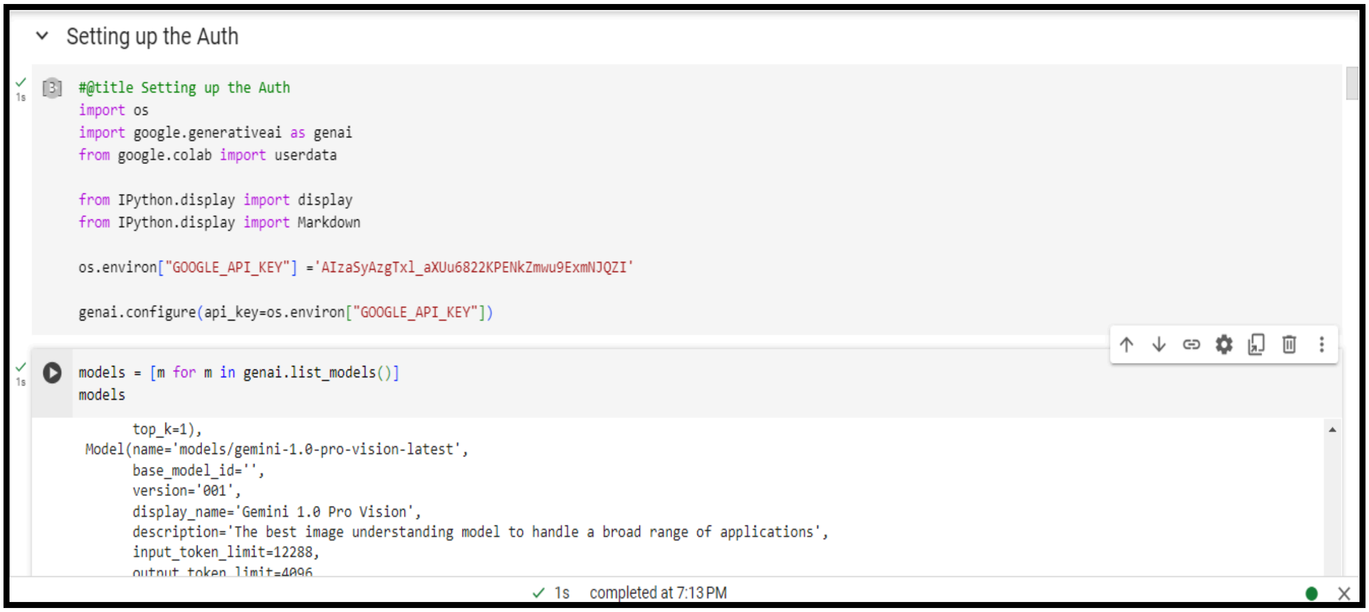
Free trial status: $258.86 credit and 66 days remaining - with a full account, you'll get unlimited access to all of Google Cloud Platform.

Create credentials to access your enabled APIs. Learn more

Remember to configure the OAuth consent screen with information about your application.

API Keys

□	Name	Creation date ↓	Restrictions	Actions
□	API key 2	Mar 25, 2024	None	SHOW KEY ⋮

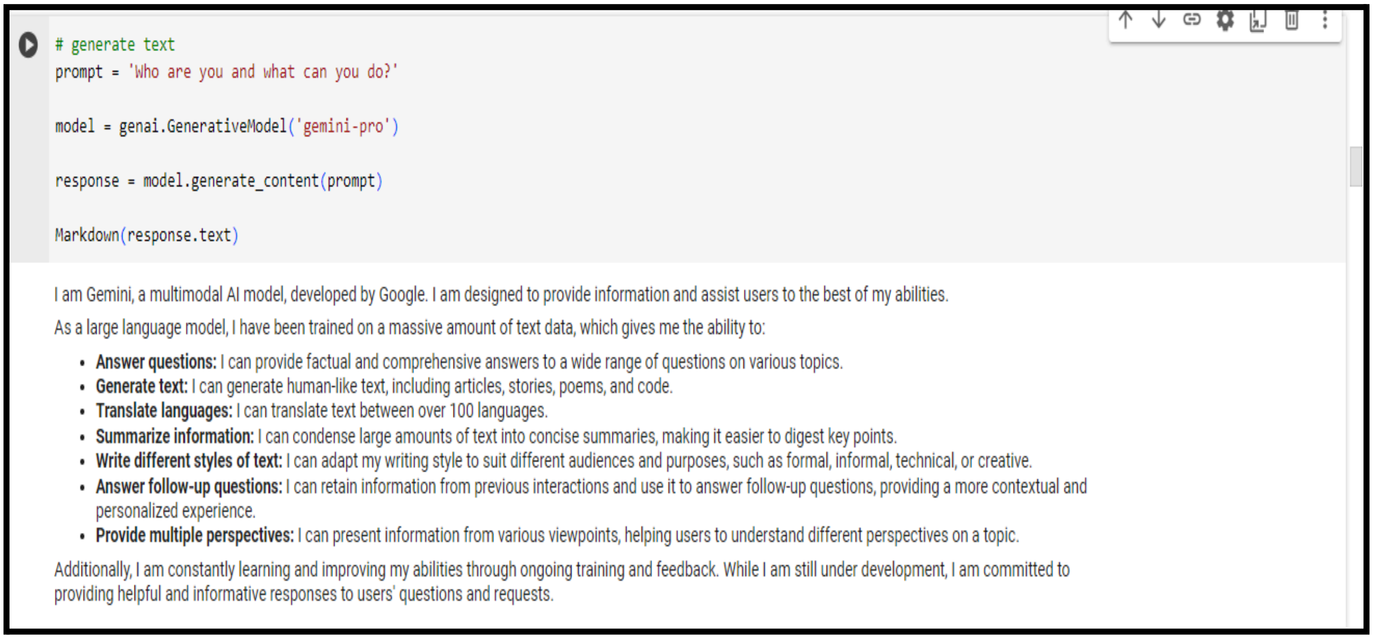


```
#@title Setting up the Auth
import os
import google.generativeai as genai
from google.colab import userdata

from IPython.display import display
from IPython.display import Markdown

os.environ["GOOGLE_API_KEY"] ='AIzaSyAzgTxl_aXUu6822KPENkZmwu9ExmNJQZI'

genai.configure(api_key=os.environ["GOOGLE_API_KEY"])
```

```
models = [m for m in genai.list_models()]
models
```

```
       top_k=1),
 Model(name='models/gemini-1.0-pro-vision-latest',
       base_model_id='',
       version='001',
       display_name='Gemini 1.0 Pro Vision',
       description='The best image understanding model to handle a broad range of applications',
       input_token_limit=12288,
       output_token_limit=4096
```

Step 3: Basic Interaction with LLM



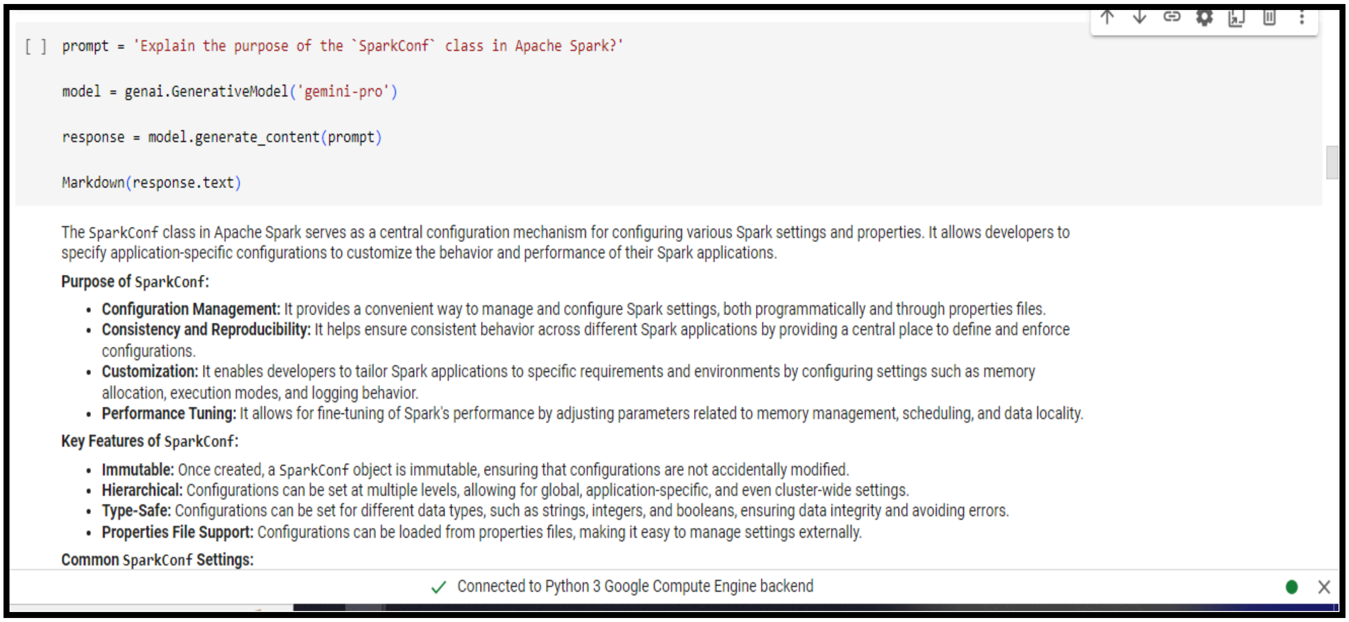
```
# generate text
prompt = 'Who are you and what can you do?'

model = genai.GenerativeModel('gemini-pro')

response = model.generate_content(prompt)

Markdown(response.text)
```

I am Gemini, a multimodal AI model, developed by Google. I am designed to provide information and assist users to the best of my abilities.

As a large language model, I have been trained on a massive amount of text data, which gives me the ability to:

- **Answer questions:** I can provide factual and comprehensive answers to a wide range of questions on various topics.
- **Generate text:** I can generate human-like text, including articles, stories, poems, and code.
- **Translate languages:** I can translate text between over 100 languages.
- **Summarize information:** I can condense large amounts of text into concise summaries, making it easier to digest key points.
- **Write different styles of text:** I can adapt my writing style to suit different audiences and purposes, such as formal, informal, technical, or creative.
- **Answer follow-up questions:** I can retain information from previous interactions and use it to answer follow-up questions, providing a more contextual and personalized experience.
- **Provide multiple perspectives:** I can present information from various viewpoints, helping users to understand different perspectives on a topic.

Additionally, I am constantly learning and improving my abilities through ongoing training and feedback. While I am still under development, I am committed to providing helpful and informative responses to users' questions and requests.



```
[ ] prompt = 'Explain the purpose of the `SparkConf` class in Apache Spark?'

    model = genai.GenerativeModel('gemini-pro')

    response = model.generate_content(prompt)

    Markdown(response.text)
```

The `SparkConf` class in Apache Spark serves as a central configuration mechanism for configuring various Spark settings and properties. It allows developers to specify application-specific configurations to customize the behavior and performance of their Spark applications.

Purpose of `SparkConf`:

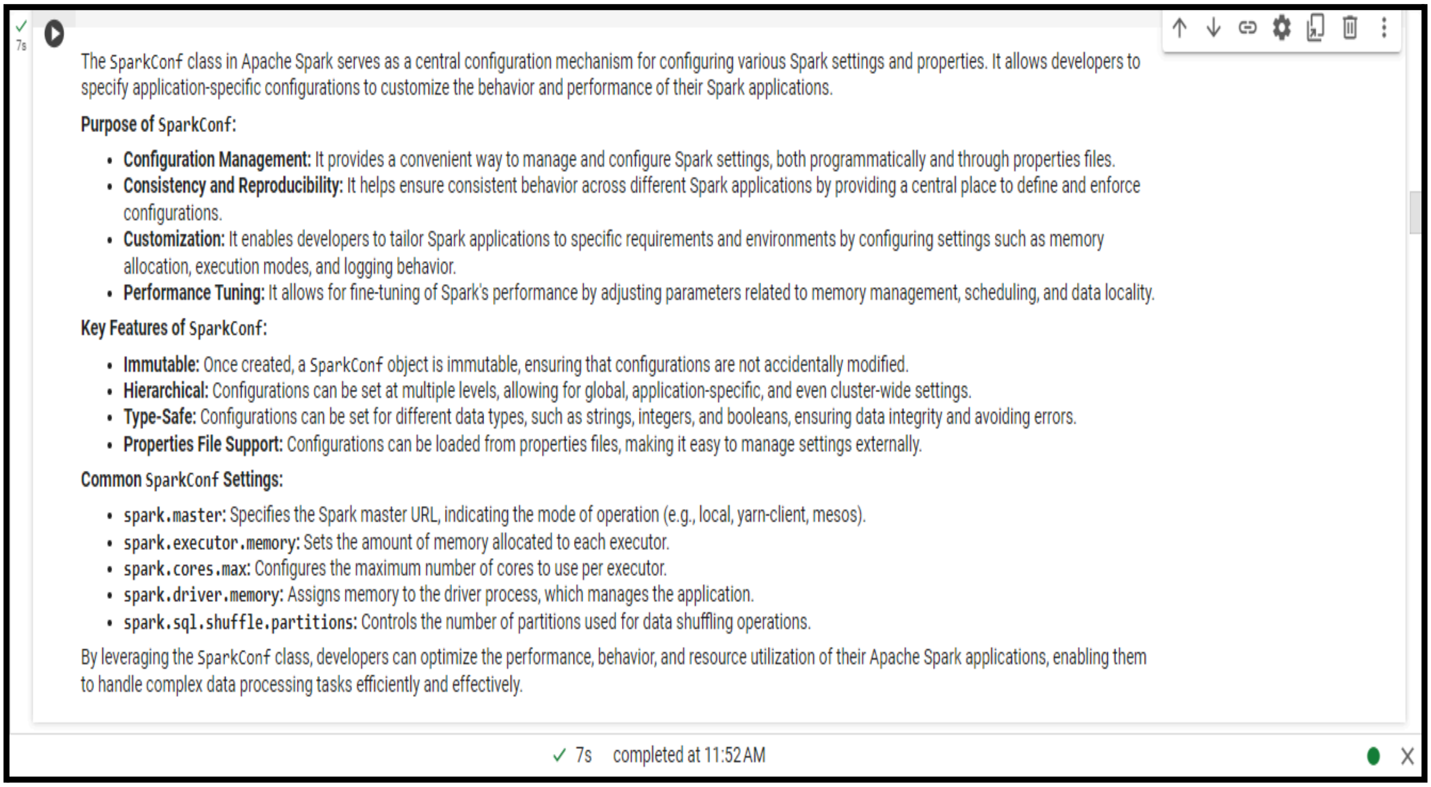
- **Configuration Management:** It provides a convenient way to manage and configure Spark settings, both programmatically and through properties files.
- **Consistency and Reproducibility:** It helps ensure consistent behavior across different Spark applications by providing a central place to define and enforce configurations.
- **Customization:** It enables developers to tailor Spark applications to specific requirements and environments by configuring settings such as memory allocation, execution modes, and logging behavior.
- **Performance Tuning:** It allows for fine-tuning of Spark's performance by adjusting parameters related to memory management, scheduling, and data locality.

Key Features of `SparkConf`:

- **Immutable:** Once created, a `SparkConf` object is immutable, ensuring that configurations are not accidentally modified.
- **Hierarchical:** Configurations can be set at multiple levels, allowing for global, application-specific, and even cluster-wide settings.
- **Type-Safe:** Configurations can be set for different data types, such as strings, integers, and booleans, ensuring data integrity and avoiding errors.
- **Properties File Support:** Configurations can be loaded from properties files, making it easy to manage settings externally.

Common `SparkConf` Settings:

✓ Connected to Python 3 Google Compute Engine backend



The `SparkConf` class in Apache Spark serves as a central configuration mechanism for configuring various Spark settings and properties. It allows developers to specify application-specific configurations to customize the behavior and performance of their Spark applications.

Purpose of `SparkConf`:

- **Configuration Management:** It provides a convenient way to manage and configure Spark settings, both programmatically and through properties files.
- **Consistency and Reproducibility:** It helps ensure consistent behavior across different Spark applications by providing a central place to define and enforce configurations.
- **Customization:** It enables developers to tailor Spark applications to specific requirements and environments by configuring settings such as memory allocation, execution modes, and logging behavior.
- **Performance Tuning:** It allows for fine-tuning of Spark's performance by adjusting parameters related to memory management, scheduling, and data locality.

Key Features of `SparkConf`:

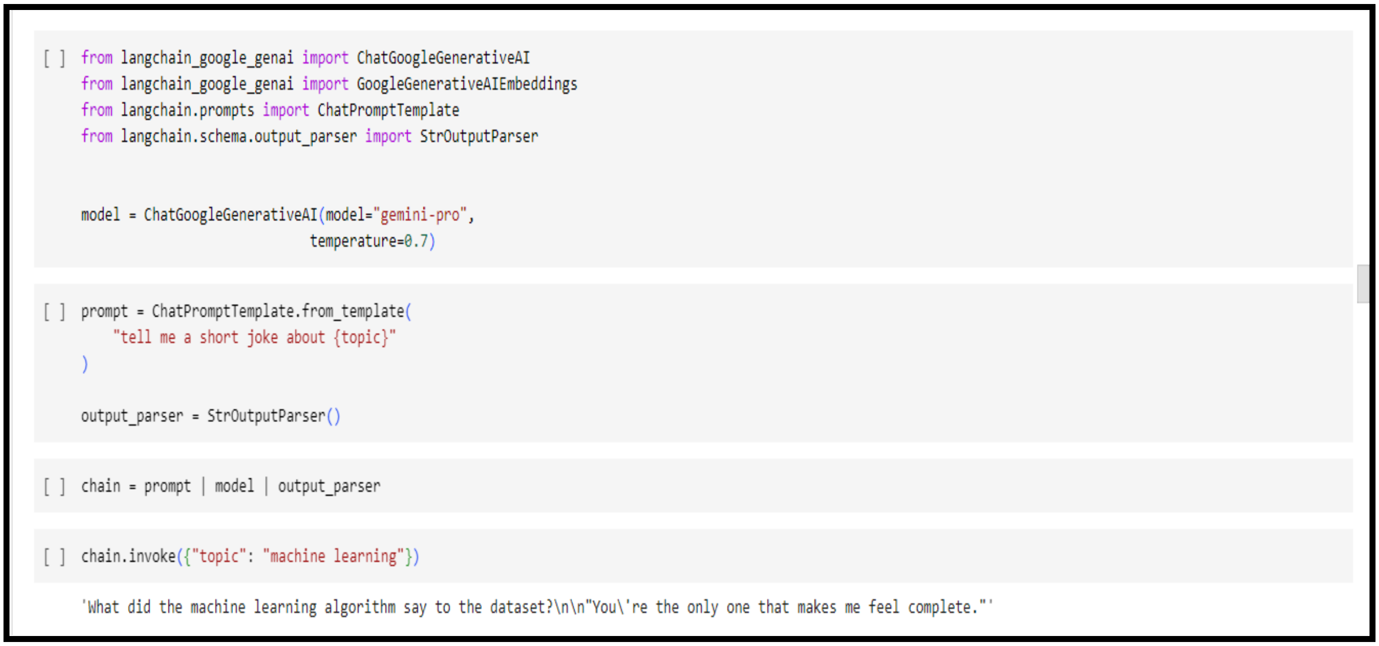
- **Immutable:** Once created, a `SparkConf` object is immutable, ensuring that configurations are not accidentally modified.
- **Hierarchical:** Configurations can be set at multiple levels, allowing for global, application-specific, and even cluster-wide settings.
- **Type-Safe:** Configurations can be set for different data types, such as strings, integers, and booleans, ensuring data integrity and avoiding errors.
- **Properties File Support:** Configurations can be loaded from properties files, making it easy to manage settings externally.

Common `SparkConf` Settings:

- `spark.master`: Specifies the Spark master URL, indicating the mode of operation (e.g., local, yarn-client, mesos).
- `spark.executor.memory`: Sets the amount of memory allocated to each executor.
- `spark.cores.max`: Configures the maximum number of cores to use per executor.
- `spark.driver.memory`: Assigns memory to the driver process, which manages the application.
- `spark.sql.shuffle.partitions`: Controls the number of partitions used for data shuffling operations.

By leveraging the `SparkConf` class, developers can optimize the performance, behavior, and resource utilization of their Apache Spark applications, enabling them to handle complex data processing tasks efficiently and effectively.

✓ 7s completed at 11:52 AM



```
[ ] from langchain_google_genai import ChatGoogleGenerativeAI
    from langchain_google_genai import GoogleGenerativeAIEmbeddings
    from langchain.prompts import ChatPromptTemplate
    from langchain.schema.output_parser import StrOutputParser


    model = ChatGoogleGenerativeAI(model="gemini-pro",
                                   temperature=0.7)

[ ] prompt = ChatPromptTemplate.from_template(
        "tell me a short joke about {topic}"
    )

    output_parser = StrOutputParser()

[ ] chain = prompt | model | output_parser

[ ] chain.invoke({"topic": "machine learning"})

    'What did the machine learning algorithm say to the dataset?\n\n"You\'re the only one that makes me feel complete."'
```

2.2 [10 points] Interact with the LLM:

- Attempt to generate the takeaways that you manually extracted from the codebase(s) that you analyzed in the previous homework. Analyze the code for Apache Spark as well. Compare the insights generated with the ones that you manually generated earlier.

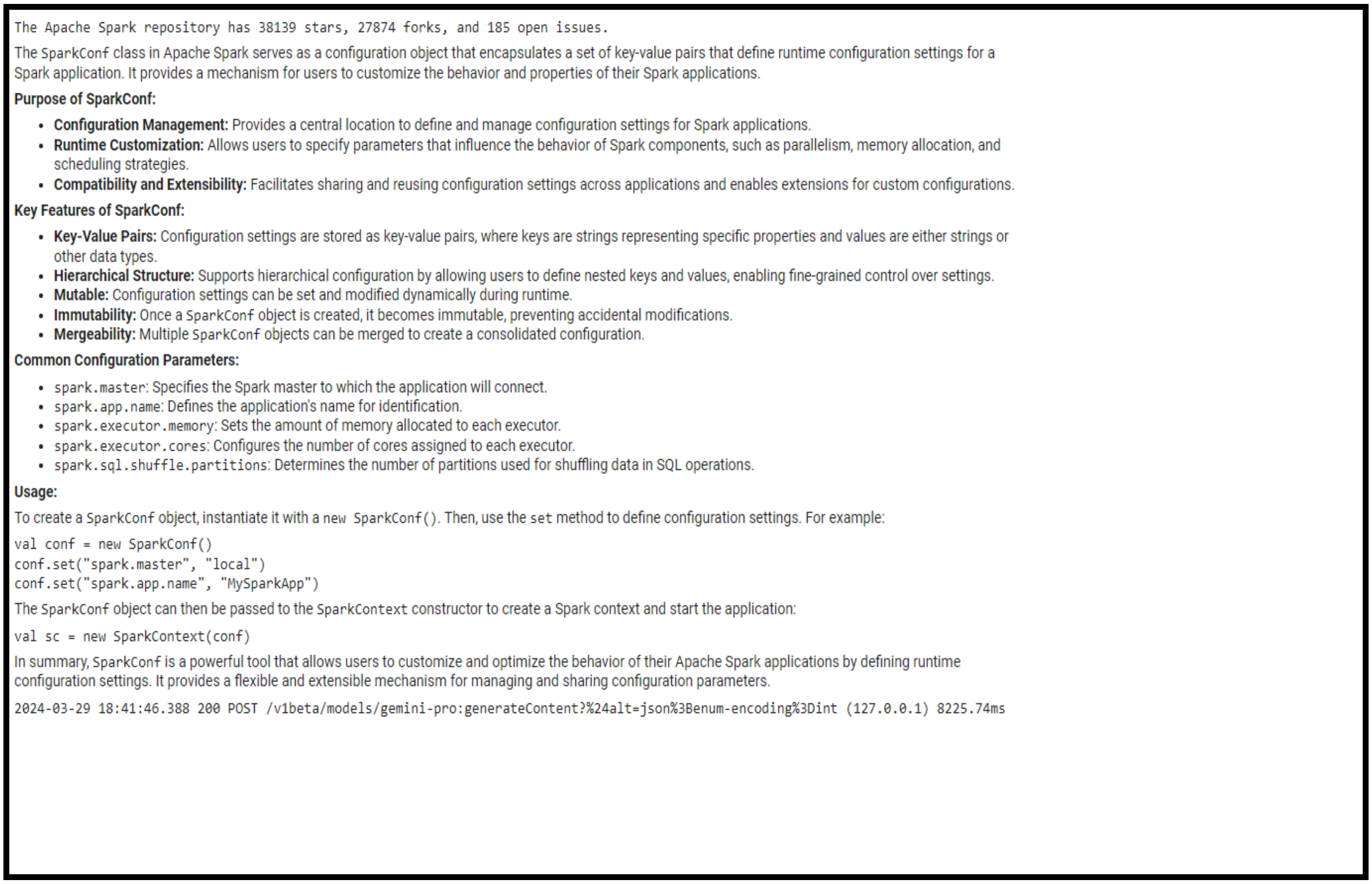
```
import requests

import google.generativeai as genai


def get_repo_stats(repo_url):

    """

    Function to get statistics for a given GitHub repository.

    """

    # Extracting owner and repo name from the URL

    parts = repo_url.split('/')

    owner = parts[-2]

    repo_name = parts[-1]
    # Constructing the GitHub API URL
    api_url = f"https://api.github.com/repos/{owner}/{repo_name}"

    # Sending a GET request to the GitHub API
    response = requests.get(api_url)

    if response.status_code == 200:
        repo_data = response.json()
        stars = repo_data['stargazers_count']
        forks = repo_data['forks_count']
        issues = repo_data['open_issues_count']
        return stars, forks, issues
    else:
        return None, None, None

def interact_with_codebase():
    """
    Function to interact with a codebase and provide insights.
    """
    repo_url = "https://github.com/apache/spark"

    stars, forks, issues = get_repo_stats(repo_url)
    if stars is not None:
        print(f"The Apache Spark repository has {stars} stars, {forks} forks, and {issues} open issues.")
    else:
        print("Failed to fetch repository statistics. Please check the URL and try again.")


if __name__ == "__main__":
    interact_with_codebase()
prompt = 'Explain the purpose of the `SparkConf` class in Apache Spark?'
model = genai.GenerativeModel('gemini-pro')
response = model.generate_content(prompt)
Markdown(response.text)
```

Manual takeaways:

Spark provides nice user APIs for computations on small datasets. Spark can scale these same code examples to large datasets on distributed clusters. It’s fantastic how Spark can handle both large and small datasets.

Spark also has an expansive API compared with other query engines. Spark allows you to perform DataFrame operations with programmatic APIs, write SQL, perform streaming analyses, and do machine learning. Spark saves you from learning multiple frameworks and patching together various libraries to perform an analysis.

Gemini takeaways:



```
The Apache Spark repository has 38139 stars, 27874 forks, and 185 open issues.
```

The `SparkConf` class in Apache Spark serves as a configuration object that encapsulates a set of key-value pairs that define runtime configuration settings for a Spark application. It provides a mechanism for users to customize the behavior and properties of their Spark applications.

Purpose of SparkConf:

- **Configuration Management:** Provides a central location to define and manage configuration settings for Spark applications.
- **Runtime Customization:** Allows users to specify parameters that influence the behavior of Spark components, such as parallelism, memory allocation, and scheduling strategies.
- **Compatibility and Extensibility:** Facilitates sharing and reusing configuration settings across applications and enables extensions for custom configurations.

Key Features of SparkConf:

- **Key-Value Pairs:** Configuration settings are stored as key-value pairs, where keys are strings representing specific properties and values are either strings or other data types.
- **Hierarchical Structure:** Supports hierarchical configuration by allowing users to define nested keys and values, enabling fine-grained control over settings.
- **Mutable:** Configuration settings can be set and modified dynamically during runtime.
- **Immutability:** Once a `SparkConf` object is created, it becomes immutable, preventing accidental modifications.
- **Mergeability:** Multiple `SparkConf` objects can be merged to create a consolidated configuration.

Common Configuration Parameters:

- `spark.master`: Specifies the Spark master to which the application will connect.
- `spark.app.name`: Defines the application's name for identification.
- `spark.executor.memory`: Sets the amount of memory allocated to each executor.
- `spark.executor.cores`: Configures the number of cores assigned to each executor.
- `spark.sql.shuffle.partitions`: Determines the number of partitions used for shuffling data in SQL operations.

Usage:

To create a `SparkConf` object, instantiate it with a `new SparkConf()`. Then, use the `set` method to define configuration settings. For example:

```
val conf = new SparkConf()
conf.set("spark.master", "local")
conf.set("spark.app.name", "MySparkApp")
```

The `SparkConf` object can then be passed to the `SparkContext` constructor to create a Spark context and start the application:

```
val sc = new SparkContext(conf)
```

In summary, `SparkConf` is a powerful tool that allows users to customize and optimize the behavior of their Apache Spark applications by defining runtime configuration settings. It provides a flexible and extensible mechanism for managing and sharing configuration parameters.

```
2024-03-29 18:41:46.388 200 POST /v1beta/models/gemini-pro:generateContent?%24alt=json%3Benum-encoding%3Dint (127.0.0.1) 8225.74ms
```

Manual Takeaways focuses on :

Configuration Management: Provides a central location to define and manage configuration settings for Spark applications.
Runtime Customization: Allows users to specify parameters influencing Spark components' behavior, such as parallelism, memory allocation, and scheduling strategies.
Compatibility and Extensibility: Facilitates sharing and reusing configuration settings across applications and enables extensions for custom configurations.
Key Features:

- Key-Value Pairs
- Hierarchical Structure
- Mutable
- Immutability
- Mergeability

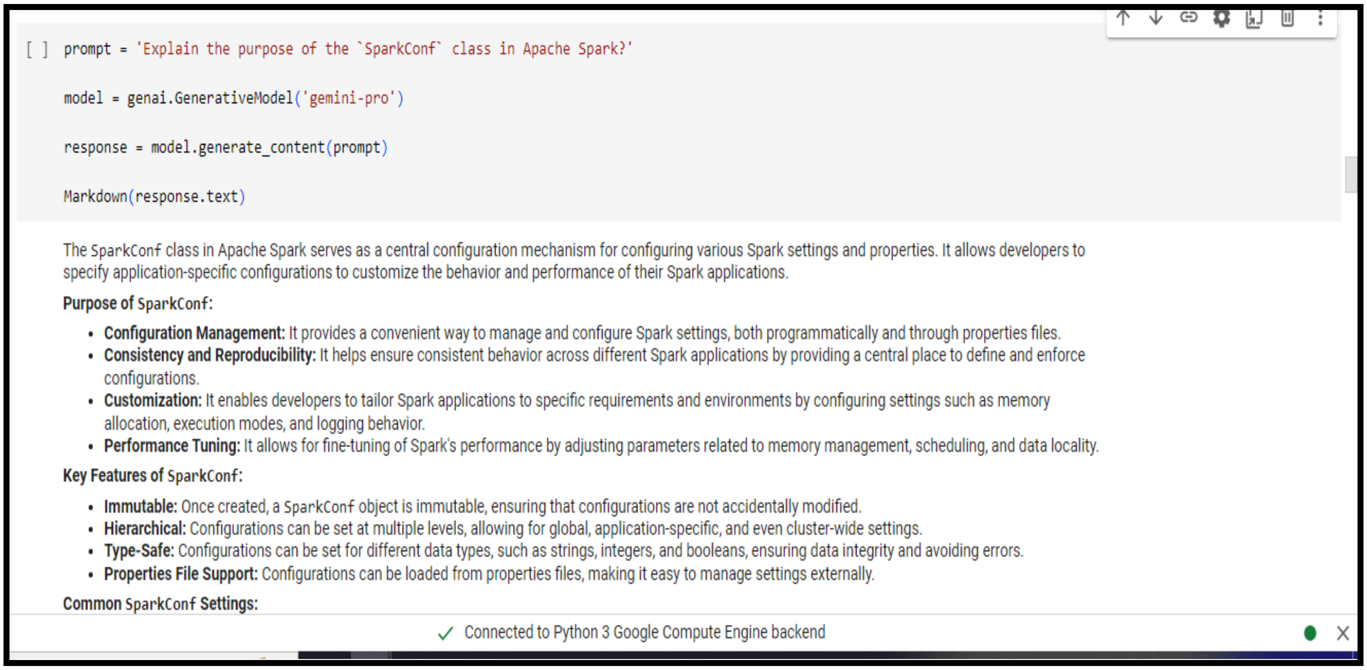
Common Configuration Parameters: Examples such as spark.master, spark.app.name, spark.executor.memory, etc.
Usage: Demonstrated instantiation and usage of SparkConf objects.

AI-generated Takeaways focuses on :

Ease of Scaling: Spark's ability to scale from small to large datasets on distributed clusters.
Expansive API: Highlights Spark's comprehensive API, covering DataFrame operations, SQL queries, streaming analysis, and machine learning.
Unified Framework: Emphasizes Spark's advantage of providing a single framework for various types of analyses, saving users from having to learn multiple frameworks and libraries.

Comparison:

- Content Coverage: Both the manual and AI-generated takeaways cover aspects of SparkConf's purpose, key features, common parameters, and usage.
- Depth of Explanation: The manual takeaways delve deeper into the technical aspects of SparkConf, such as its hierarchical structure, immutability, and manageability. In contrast, the AI-generated takeaway focuses more on the broader advantages and capabilities of Spark, highlighting its scalability, expansive API, and unified framework.
- Technical vs. Practical Focus: The manual takeaways provide more technical insights into SparkConf's workings and usage. In contrast, the AI-generated takeaway emphasizes practical benefits and advantages of using Spark in data analysis.



```
[ ]  prompt = 'Explain the purpose of the `SparkConf` class in Apache Spark?'

     model = genai.GenerativeModel('gemini-pro')

     response = model.generate_content(prompt)

     Markdown(response.text)
```

The `SparkConf` class in Apache Spark serves as a central configuration mechanism for configuring various Spark settings and properties. It allows developers to specify application-specific configurations to customize the behavior and performance of their Spark applications.

Purpose of `SparkConf`:

- **Configuration Management:** It provides a convenient way to manage and configure Spark settings, both programmatically and through properties files.
- **Consistency and Reproducibility:** It helps ensure consistent behavior across different Spark applications by providing a central place to define and enforce configurations.
- **Customization:** It enables developers to tailor Spark applications to specific requirements and environments by configuring settings such as memory allocation, execution modes, and logging behavior.
- **Performance Tuning:** It allows for fine-tuning of Spark's performance by adjusting parameters related to memory management, scheduling, and data locality.

Key Features of `SparkConf`:

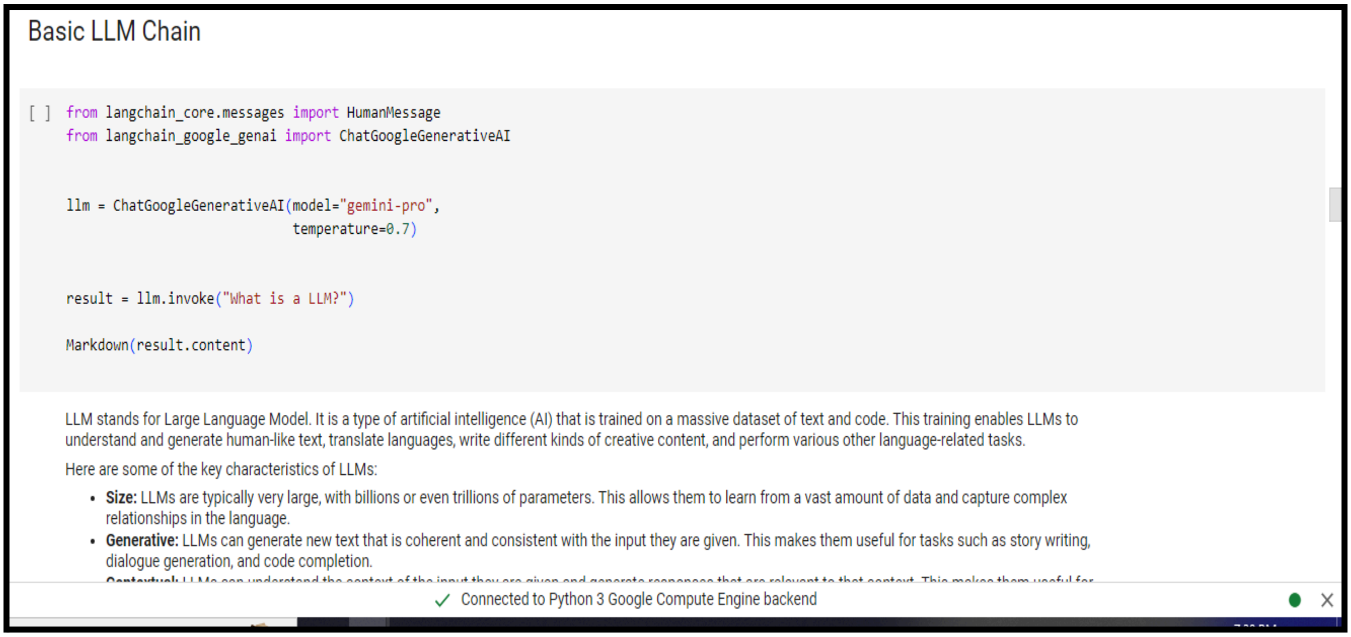
- **Immutable:** Once created, a `SparkConf` object is immutable, ensuring that configurations are not accidentally modified.
- **Hierarchical:** Configurations can be set at multiple levels, allowing for global, application-specific, and even cluster-wide settings.
- **Type-Safe:** Configurations can be set for different data types, such as strings, integers, and booleans, ensuring data integrity and avoiding errors.
- **Properties File Support:** Configurations can be loaded from properties files, making it easy to manage settings externally.

Common `SparkConf` Settings:

✓ Connected to Python 3 Google Compute Engine backend

Process User Queries:

- Design a function to handle user queries:



Basic LLM Chain

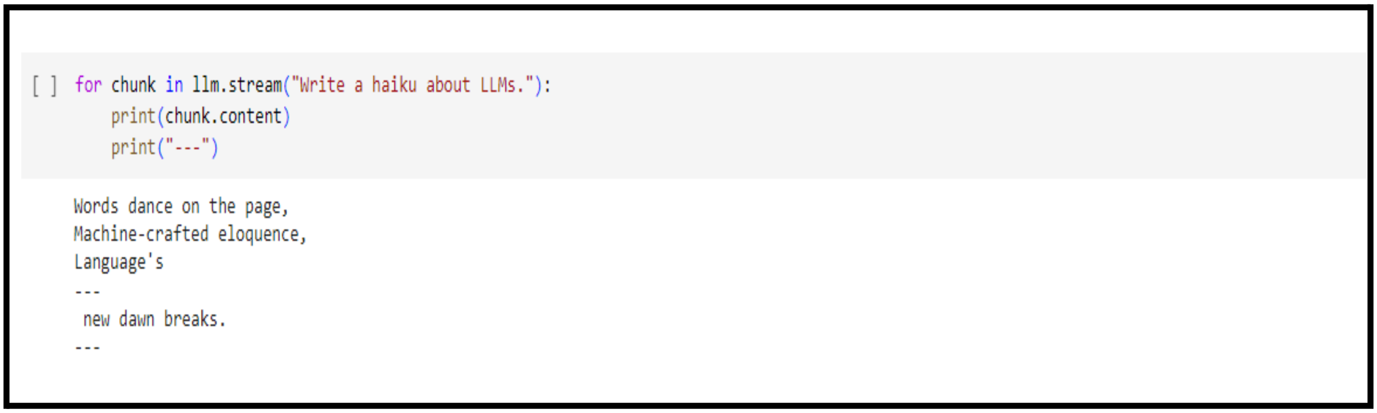
```
[ ]  from langchain_core.messages import HumanMessage
     from langchain_google_genai import ChatGoogleGenerativeAI


     llm = ChatGoogleGenerativeAI(model="gemini-pro",
                                  temperature=0.7)


     result = llm.invoke("What is a LLM?")

     Markdown(result.content)
```

LLM stands for Large Language Model. It is a type of artificial intelligence (AI) that is trained on a massive dataset of text and code. This training enables LLMs to understand and generate human-like text, translate languages, write different kinds of creative content, and perform various other language-related tasks.

Here are some of the key characteristics of LLMs:

- **Size:** LLMs are typically very large, with billions or even trillions of parameters. This allows them to learn from a vast amount of data and capture complex relationships in the language.
- **Generative:** LLMs can generate new text that is coherent and consistent with the input they are given. This makes them useful for tasks such as story writing, dialogue generation, and code completion.
- **Contextual:** LLMs can understand the context of the input they are given and generate responses that are relevant to that context. This makes them useful for

✓ Connected to Python 3 Google Compute Engine backend

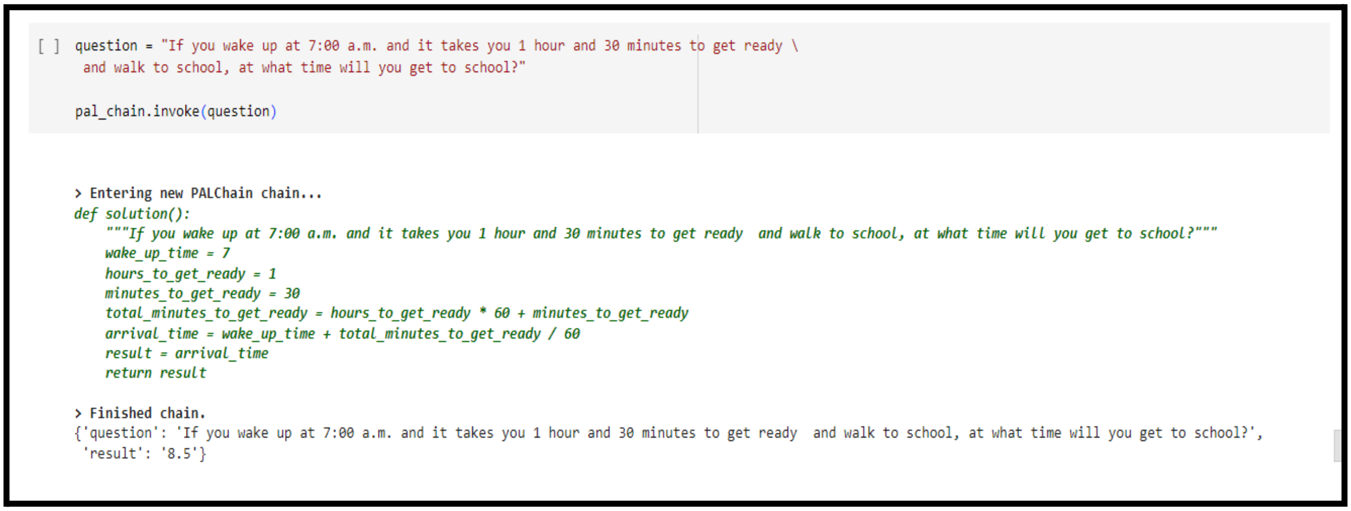


```
[ ]  for chunk in llm.stream("Write a haiku about LLMs."):
         print(chunk.content)
         print("---")

     Words dance on the page,
     Machine-crafted eloquence,
     Language's
     ---
      new dawn breaks.
     ---
```

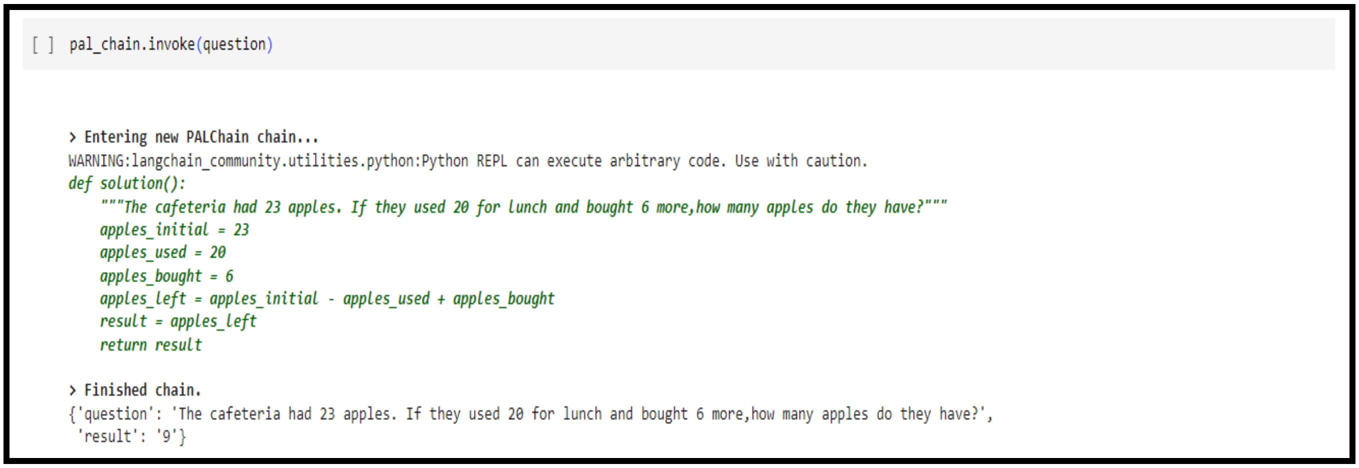
2.3 [5 points] LangChain:

- Assess the value addition of using LangChain

Math solutions:

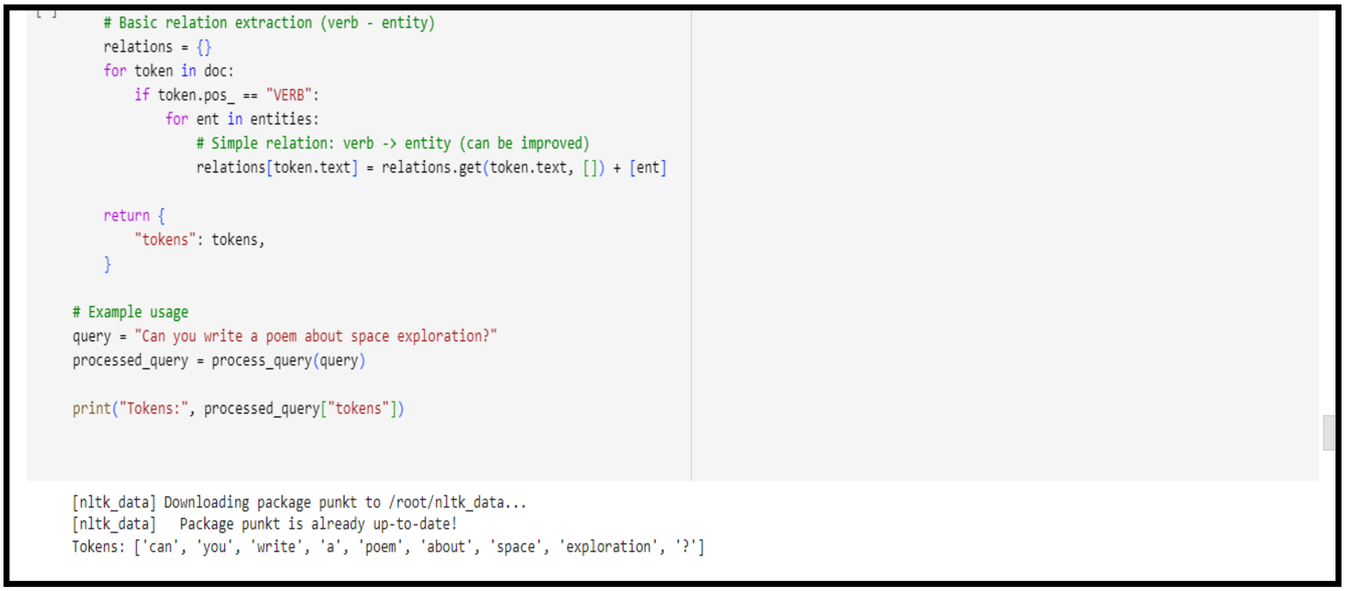


```
[ ]  question = "If you wake up at 7:00 a.m. and it takes you 1 hour and 30 minutes to get ready \
      and walk to school, at what time will you get to school?"

     pal_chain.invoke(question)


     > Entering new PALChain chain...
     def solution():
         """If you wake up at 7:00 a.m. and it takes you 1 hour and 30 minutes to get ready  and walk to school, at what time will you get to school?"""
         wake_up_time = 7
         hours_to_get_ready = 1
         minutes_to_get_ready = 30
         total_minutes_to_get_ready = hours_to_get_ready * 60 + minutes_to_get_ready
         arrival_time = wake_up_time + total_minutes_to_get_ready / 60
         result = arrival_time
         return result

     > Finished chain.
     {'question': 'If you wake up at 7:00 a.m. and it takes you 1 hour and 30 minutes to get ready  and walk to school, at what time will you get to school?',
      'result': '8.5'}
```



```
[ ] pal_chain.invoke(question)


> Entering new PALChain chain...
WARNING:langchain_community.utilities.python:Python REPL can execute arbitrary code. Use with caution.
def solution():
    """The cafeteria had 23 apples. If they used 20 for lunch and bought 6 more,how many apples do they have?"""
    apples_initial = 23
    apples_used = 20
    apples_bought = 6
    apples_left = apples_initial - apples_used + apples_bought
    result = apples_left
    return result

> Finished chain.
{'question': 'The cafeteria had 23 apples. If they used 20 for lunch and bought 6 more,how many apples do they have?',
 'result': '9'}
```

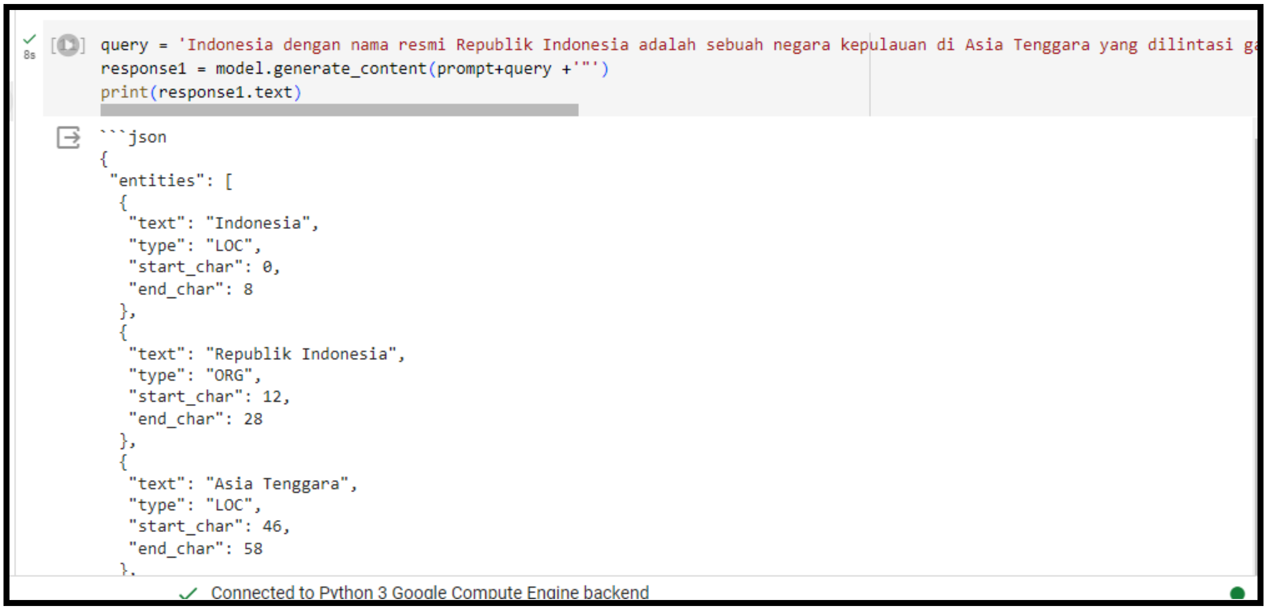
- create a pipeline to process queries in a structured manner, potentially involving:
    - Tokenization (breaking down the query into words)



```
    # Basic relation extraction (verb - entity)
    relations = {}
    for token in doc:
        if token.pos_ == "VERB":
            for ent in entities:
                # Simple relation: verb -> entity (can be improved)
                relations[token.text] = relations.get(token.text, []) + [ent]

    return {
        "tokens": tokens,
    }

# Example usage
query = "Can you write a poem about space exploration?"
processed_query = process_query(query)

print("Tokens:", processed_query["tokens"])
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
Tokens: ['can', 'you', 'write', 'a', 'poem', 'about', 'space', 'exploration', '?']
```

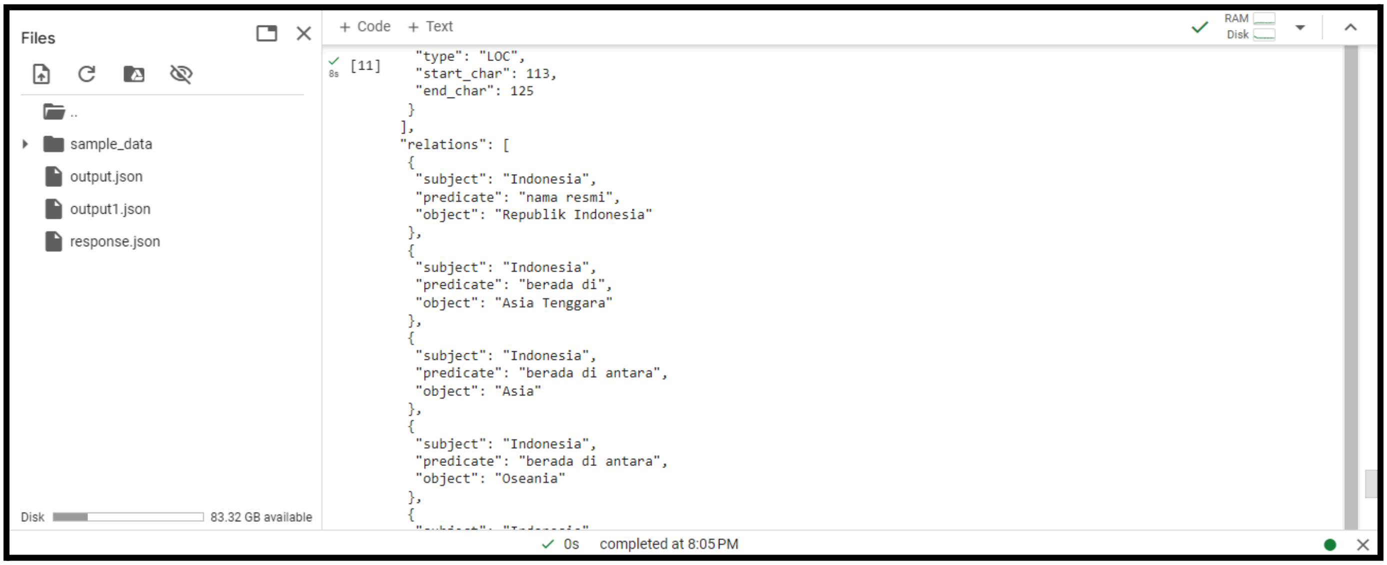
    - Named entity recognition (identifying code elements)



````
query = 'Indonesia dengan nama resmi Republik Indonesia adalah sebuah negara kepulauan di Asia Tenggara yang dilintasi ga
response1 = model.generate_content(prompt+query +'"')
print(response1.text)
````

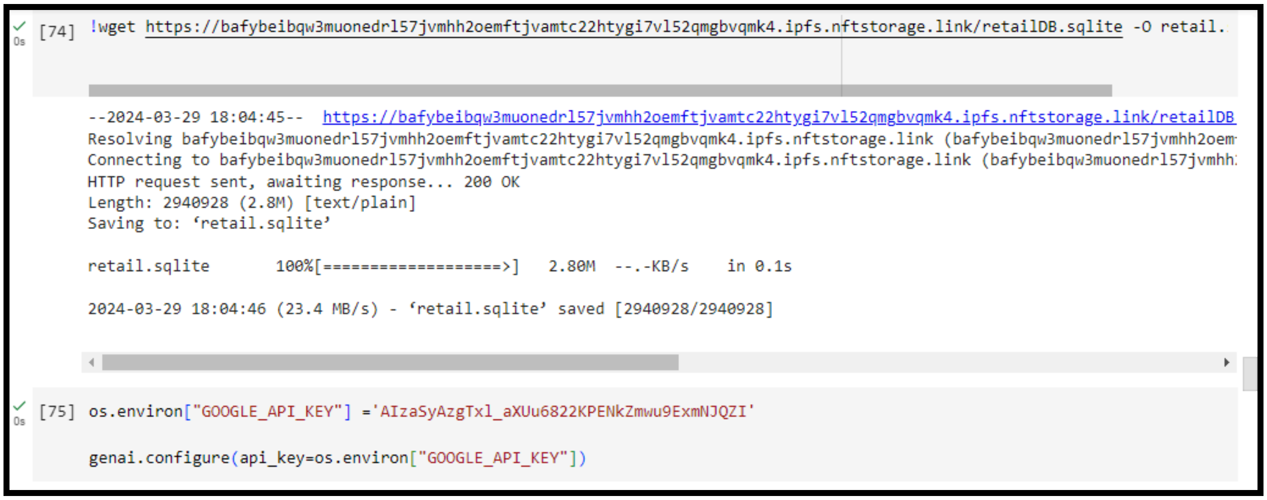
~~~
```json
{
 "entities": [
  {
   "text": "Indonesia",
   "type": "LOC",
   "start_char": 0,
   "end_char": 8
  },
  {
   "text": "Republik Indonesia",
   "type": "ORG",
   "start_char": 12,
   "end_char": 28
  },
  {
   "text": "Asia Tenggara",
   "type": "LOC",
   "start_char": 46,
   "end_char": 58
  },
~~~

- Relation extraction (understanding relationships between entities)



````
   "type": "LOC",
   "start_char": 113,
   "end_char": 125
  }
 ],
 "relations": [
  {
   "subject": "Indonesia",
   "predicate": "nama resmi",
   "object": "Republik Indonesia"
  },
  {
   "subject": "Indonesia",
   "predicate": "berada di",
   "object": "Asia Tenggara"
  },
  {
   "subject": "Indonesia",
   "predicate": "berada di antara",
   "object": "Asia"
  },
  {
   "subject": "Indonesia",
   "predicate": "berada di antara",
   "object": "Oseania"
  },
  {
   "subject": "Indonesia",
````

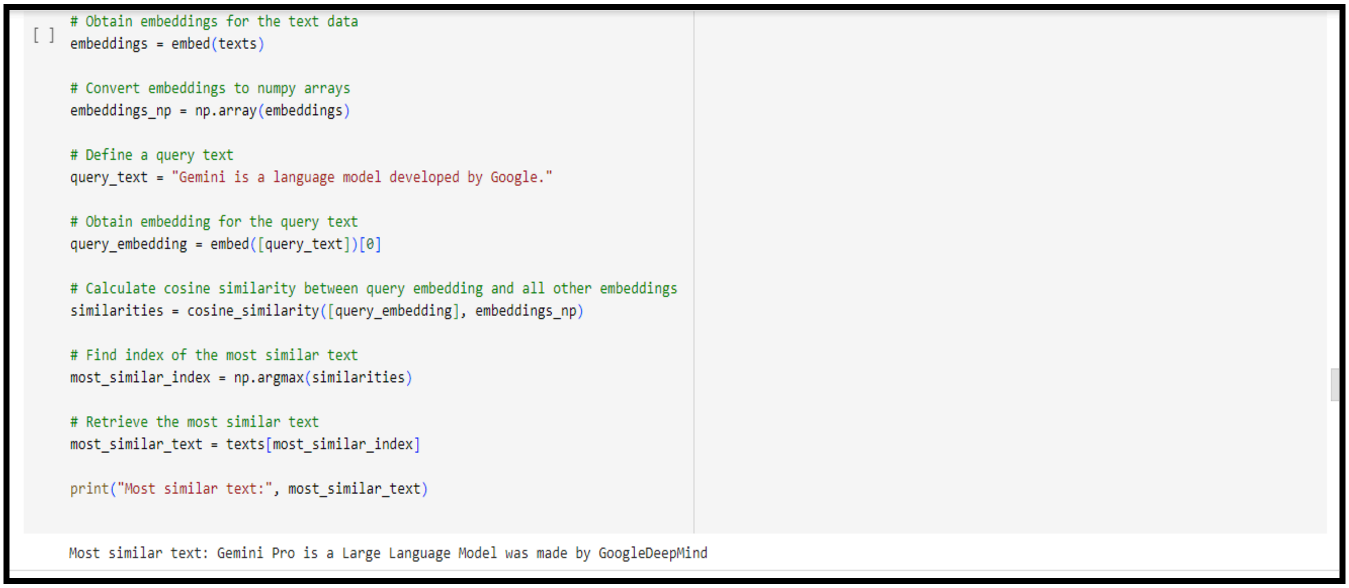
Interacting with one of my old databases and providing solutions.



```
[74] !wget https://bafybeibqw3muonedrl57jvmhh2oemftjvamtc22htygi7vl52qmgbvqmk4.ipfs.nftstorage.link/retailDB.sqlite -O retail.
```

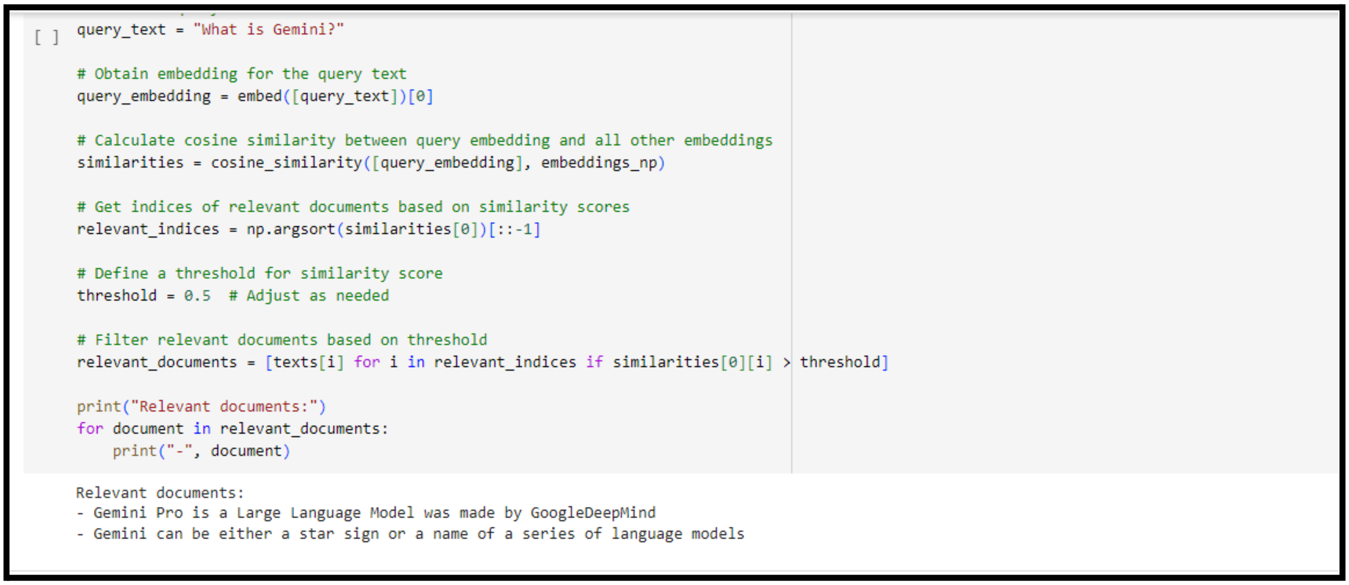
```
--2024-03-29 18:04:45--  https://bafybeibqw3muonedrl57jvmhh2oemftjvamtc22htygi7vl52qmgbvqmk4.ipfs.nftstorage.link/retailDB
Resolving bafybeibqw3muonedrl57jvmhh2oemftjvamtc22htygi7vl52qmgbvqmk4.ipfs.nftstorage.link (bafybeibqw3muonedrl57jvmhh2oem
Connecting to bafybeibqw3muonedrl57jvmhh2oemftjvamtc22htygi7vl52qmgbvqmk4.ipfs.nftstorage.link (bafybeibqw3muonedrl57jvmhh
HTTP request sent, awaiting response... 200 OK
Length: 2940928 (2.8M) [text/plain]
Saving to: ‘retail.sqlite’

retail.sqlite       100%[===================>]   2.80M  --.-KB/s    in 0.1s

2024-03-29 18:04:46 (23.4 MB/s) - ‘retail.sqlite’ saved [2940928/2940928]
```

```
[75] os.environ["GOOGLE_API_KEY"] ='AIzaSyAzgTxl_aXUu6822KPENkZmwu9ExmNJQZI'

     genai.configure(api_key=os.environ["GOOGLE_API_KEY"])
```



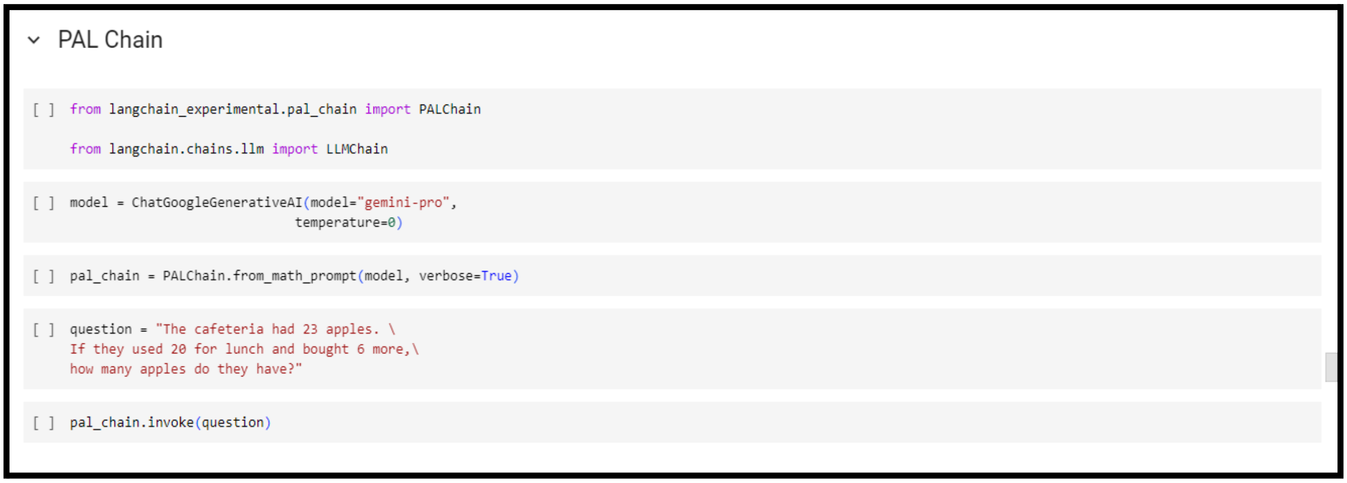
```
# Obtain embeddings for the text data
embeddings = embed(texts)

# Convert embeddings to numpy arrays
embeddings_np = np.array(embeddings)

# Define a query text
query_text = "Gemini is a language model developed by Google."

# Obtain embedding for the query text
query_embedding = embed([query_text])[0]

# Calculate cosine similarity between query embedding and all other embeddings
similarities = cosine_similarity([query_embedding], embeddings_np)

# Find index of the most similar text
most_similar_index = np.argmax(similarities)

# Retrieve the most similar text
most_similar_text = texts[most_similar_index]

print("Most similar text:", most_similar_text)
```

```
Most similar text: Gemini Pro is a Large Language Model was made by GoogleDeepMind
```



```
query_text = "What is Gemini?"

# Obtain embedding for the query text
query_embedding = embed([query_text])[0]

# Calculate cosine similarity between query embedding and all other embeddings
similarities = cosine_similarity([query_embedding], embeddings_np)

# Get indices of relevant documents based on similarity scores
relevant_indices = np.argsort(similarities[0])[::-1]

# Define a threshold for similarity score
threshold = 0.5  # Adjust as needed

# Filter relevant documents based on threshold
relevant_documents = [texts[i] for i in relevant_indices if similarities[0][i] > threshold]

print("Relevant documents:")
for document in relevant_documents:
    print("-", document)
```

```
Relevant documents:
- Gemini Pro is a Large Language Model was made by GoogleDeepMind
- Gemini can be either a star sign or a name of a series of language models
```



PAL Chain

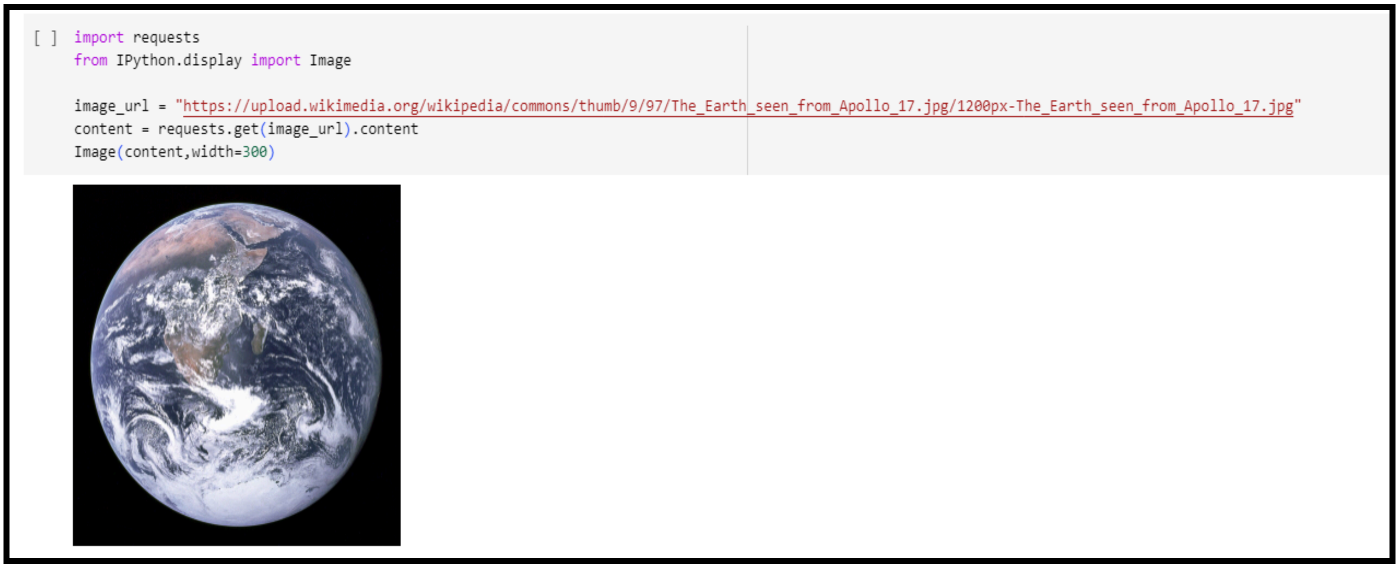
```
from langchain_experimental.pal_chain import PALChain

from langchain.chains.llm import LLMChain
```

```
model = ChatGoogleGenerativeAI(model="gemini-pro",
                             temperature=0)
```

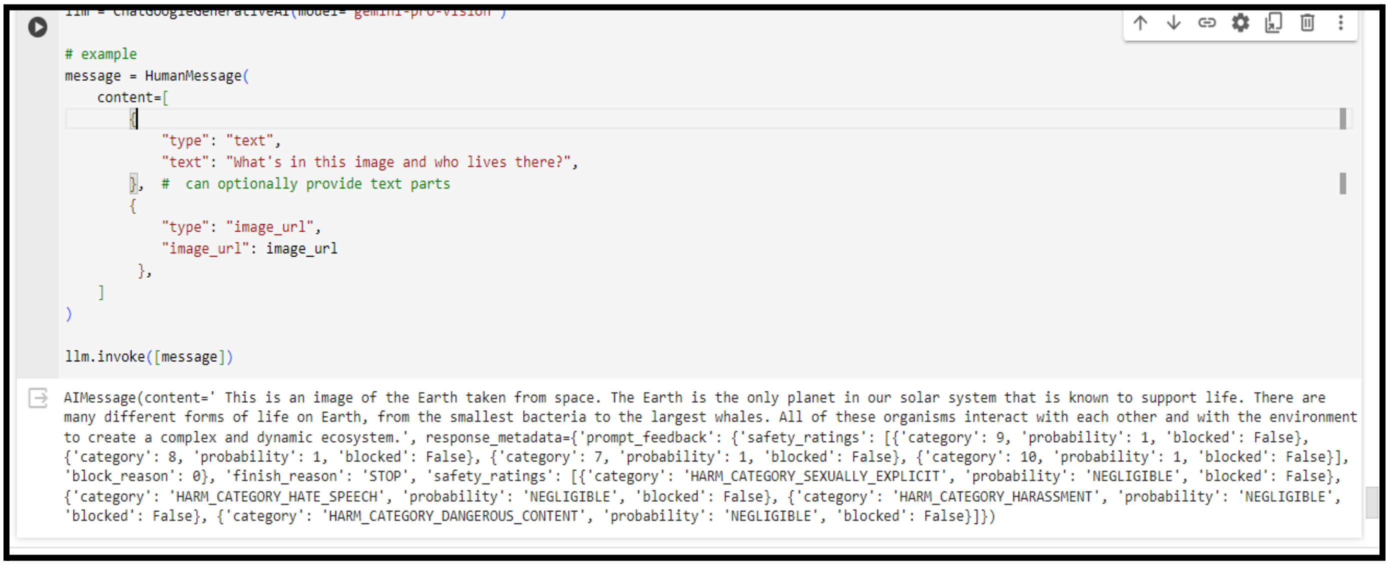
```
pal_chain = PALChain.from_math_prompt(model, verbose=True)
```

```
question = "The cafeteria had 23 apples. \
If they used 20 for lunch and bought 6 more,\
how many apples do they have?"
```

```
pal_chain.invoke(question)
```



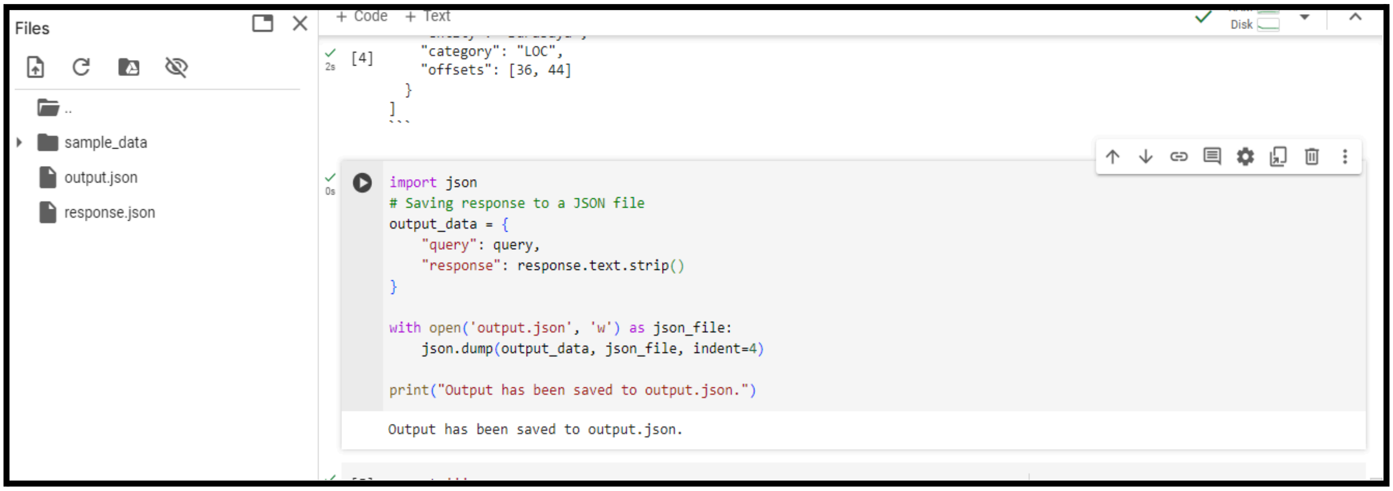
```
[ ] import requests
    from IPython.display import Image

    image_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/9/97/The_Earth_seen_from_Apollo_17.jpg/1200px-The_Earth_seen_from_Apollo_17.jpg"
    content = requests.get(image_url).content
    Image(content,width=300)
```



```
llm = ChatGoogleGenerativeAI(model="gemini-pro-vision")

# example
message = HumanMessage(
    content=[
        {
            "type": "text",
            "text": "What's in this image and who lives there?",
        },  #  can optionally provide text parts
        {
            "type": "image_url",
            "image_url": image_url
         },
    ]
)

llm.invoke([message])
```

```
AIMessage(content=' This is an image of the Earth taken from space. The Earth is the only planet in our solar system that is known to support life. There are
many different forms of life on Earth, from the smallest bacteria to the largest whales. All of these organisms interact with each other and with the environment
to create a complex and dynamic ecosystem.', response_metadata={'prompt_feedback': {'safety_ratings': [{'category': 9, 'probability': 1, 'blocked': False},
{'category': 8, 'probability': 1, 'blocked': False}, {'category': 7, 'probability': 1, 'blocked': False}, {'category': 10, 'probability': 1, 'blocked': False}],
'block_reason': 0}, 'finish_reason': 'STOP', 'safety_ratings': [{'category': 'HARM_CATEGORY_SEXUALLY_EXPLICIT', 'probability': 'NEGLIGIBLE', 'blocked': False},
{'category': 'HARM_CATEGORY_HATE_SPEECH', 'probability': 'NEGLIGIBLE', 'blocked': False}, {'category': 'HARM_CATEGORY_HARASSMENT', 'probability': 'NEGLIGIBLE',
'blocked': False}, {'category': 'HARM_CATEGORY_DANGEROUS_CONTENT', 'probability': 'NEGLIGIBLE', 'blocked': False}]})
```

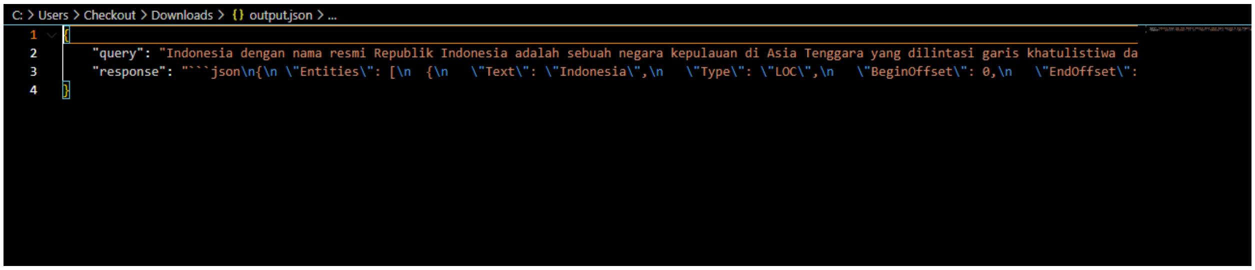
Storing result in JSON:



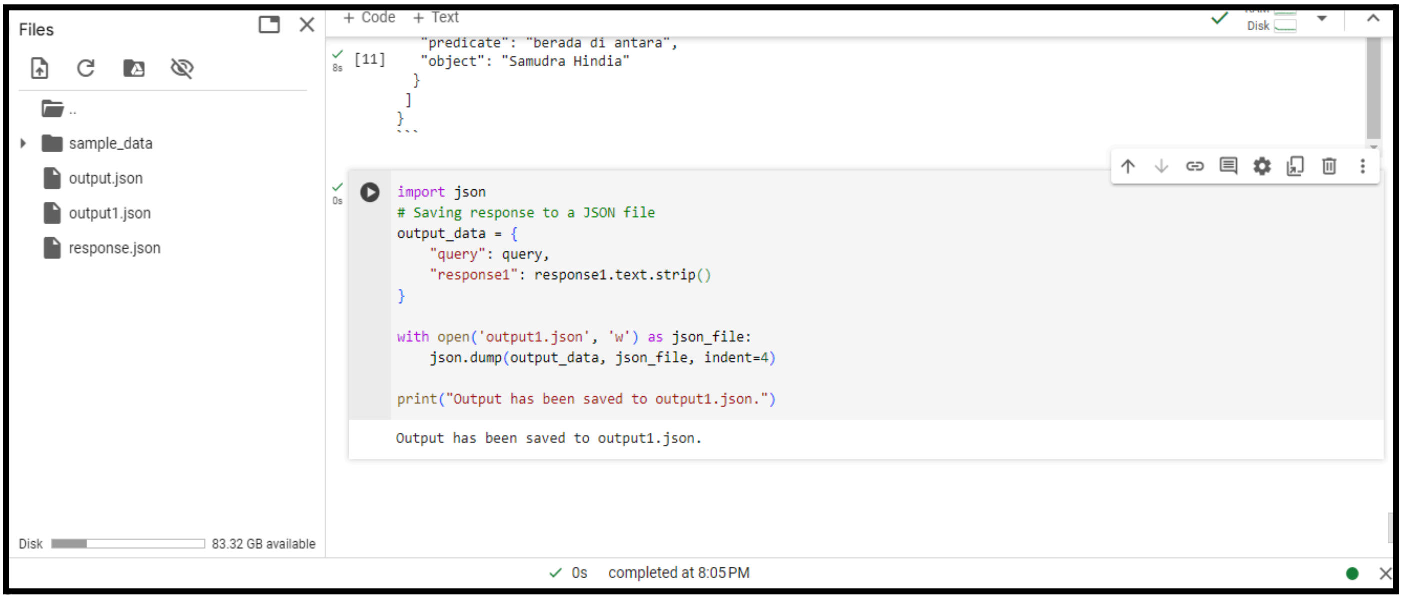
```
[4]      "category": "LOC",
         "offsets": [36, 44]
       }
     ]
     ```
```

```
import json
# Saving response to a JSON file
output_data = {
    "query": query,
    "response": response.text.strip()
}

with open('output.json', 'w') as json_file:
    json.dump(output_data, json_file, indent=4)

print("Output has been saved to output.json.")
```

```
Output has been saved to output.json.
```
```

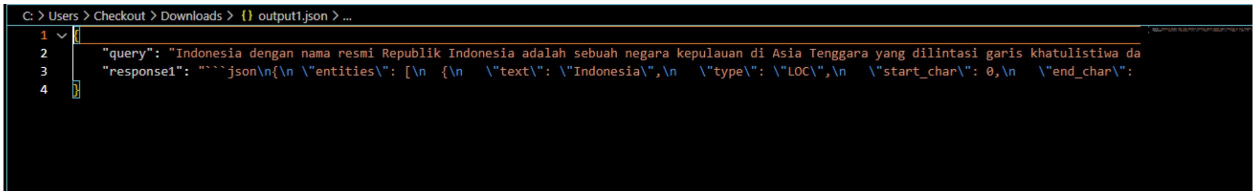


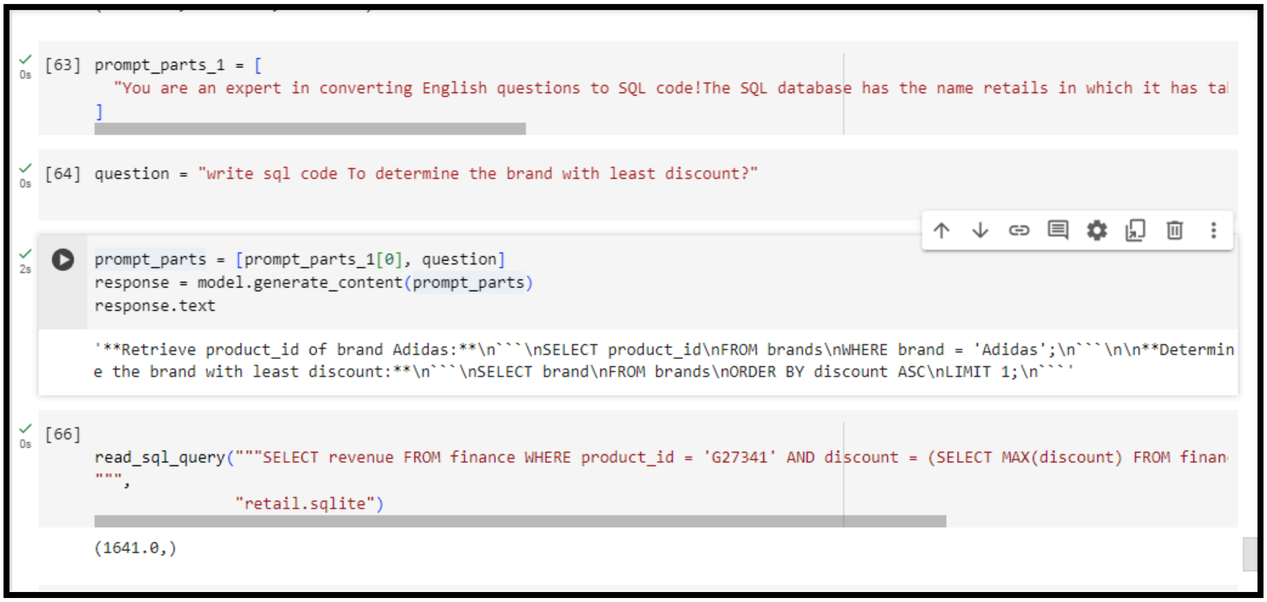
```
{
    "query": "Indonesia dengan nama resmi Republik Indonesia adalah sebuah negara kepulauan di Asia Tenggara yang dilintasi garis khatulistiwa da
    "response": "```json\n{\n \"Entities\": [\n  {\n   \"Text\": \"Indonesia\",\n   \"Type\": \"LOC\",\n   \"BeginOffset\": 0,\n   \"EndOffset\":
}
```



```
[11]   "predicate": "berada di antara",
       "object": "Samudra Hindia"
      }
     ]
    }
    ```
```

```
import json
# Saving response to a JSON file
output_data = {
    "query": query,
    "response1": response1.text.strip()
}

with open('output1.json', 'w') as json_file:
    json.dump(output_data, json_file, indent=4)

print("Output has been saved to output1.json.")
```

```
Output has been saved to output1.json.
```

✓ 0s   completed at 8:05PM


C: > Users > Checkout > Downloads > {} output1.json > ...

```
{
    "query": "Indonesia dengan nama resmi Republik Indonesia adalah sebuah negara kepulauan di Asia Tenggara yang dilintasi garis khatulistiwa da
    "response1": "```json\n{\n \"entities\": [\n  {\n   \"text\": \"Indonesia\",\n   \"type\": \"LOC\",\n   \"start_char\": 0,\n   \"end_char\":
}
```

Storing the preprocessed code in a database (SQLite)

Asking the LLM to go through the retail.sqlite from a previous work and answering the questions.
```



```
[63] prompt_parts_1 = [
       "You are an expert in converting English questions to SQL code!The SQL database has the name retails in which it has tal
     ]
```

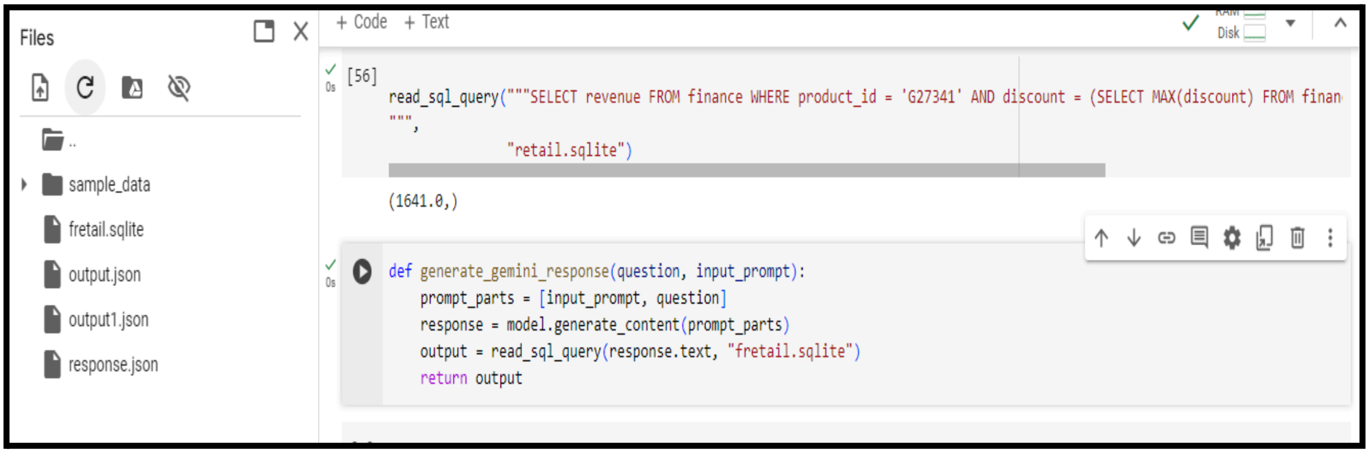
```
[64] question = "write sql code To determine the brand with least discount?"
```

```
prompt_parts = [prompt_parts_1[0], question]
response = model.generate_content(prompt_parts)
response.text
```

```
'**Retrieve product_id of brand Adidas:**\n```\nSELECT product_id\nFROM brands\nWHERE brand = 'Adidas';\n```\n\n**Determin
e the brand with least discount:**\n```\nSELECT brand\nFROM brands\nORDER BY discount ASC\nLIMIT 1;\n```'
```

```
[66]
read_sql_query("""SELECT revenue FROM finance WHERE product_id = 'G27341' AND discount = (SELECT MAX(discount) FROM finan
""",
               "retail.sqlite")
```

```
(1641.0,)
```

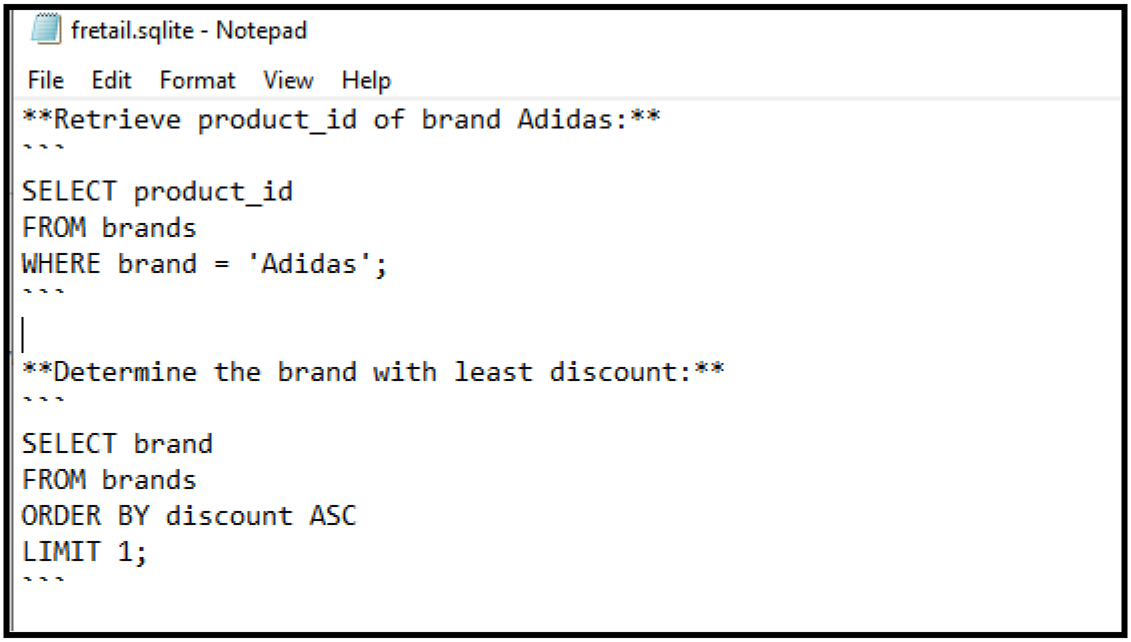


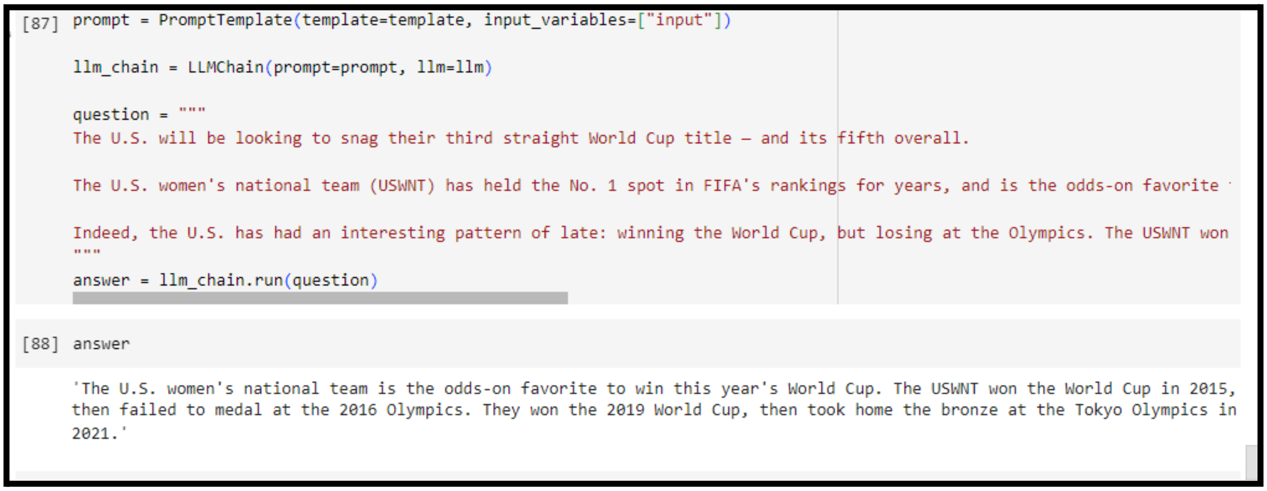
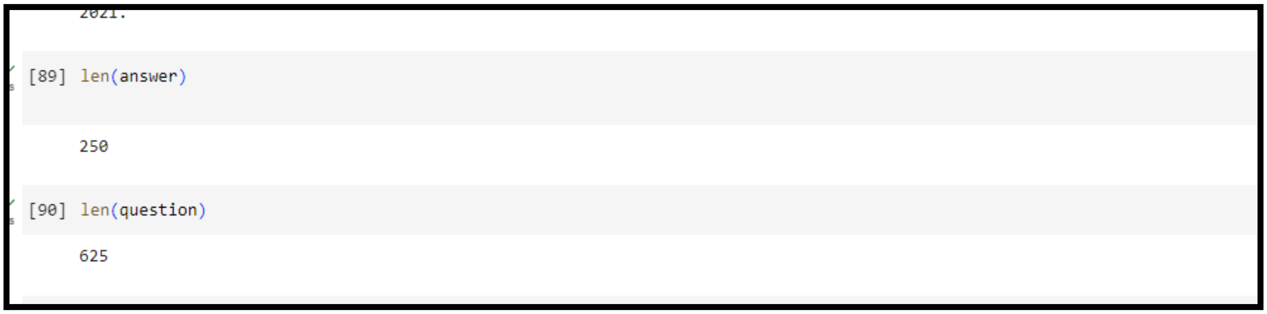
```
[56]
read_sql_query("""SELECT revenue FROM finance WHERE product_id = 'G27341' AND discount = (SELECT MAX(discount) FROM finan
""",
               "retail.sqlite")
```

```
(1641.0,)
```

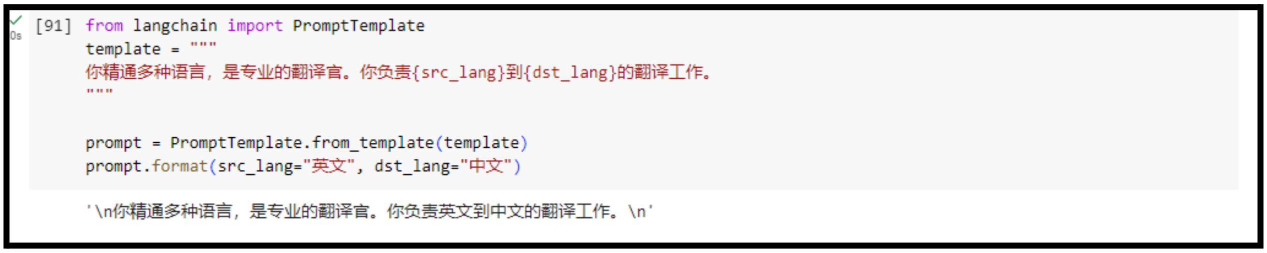
```
def generate_gemini_response(question, input_prompt):
    prompt_parts = [input_prompt, question]
    response = model.generate_content(prompt_parts)
    output = read_sql_query(response.text, "fretail.sqlite")
    return output
```

Storing the solution into sqlite file.

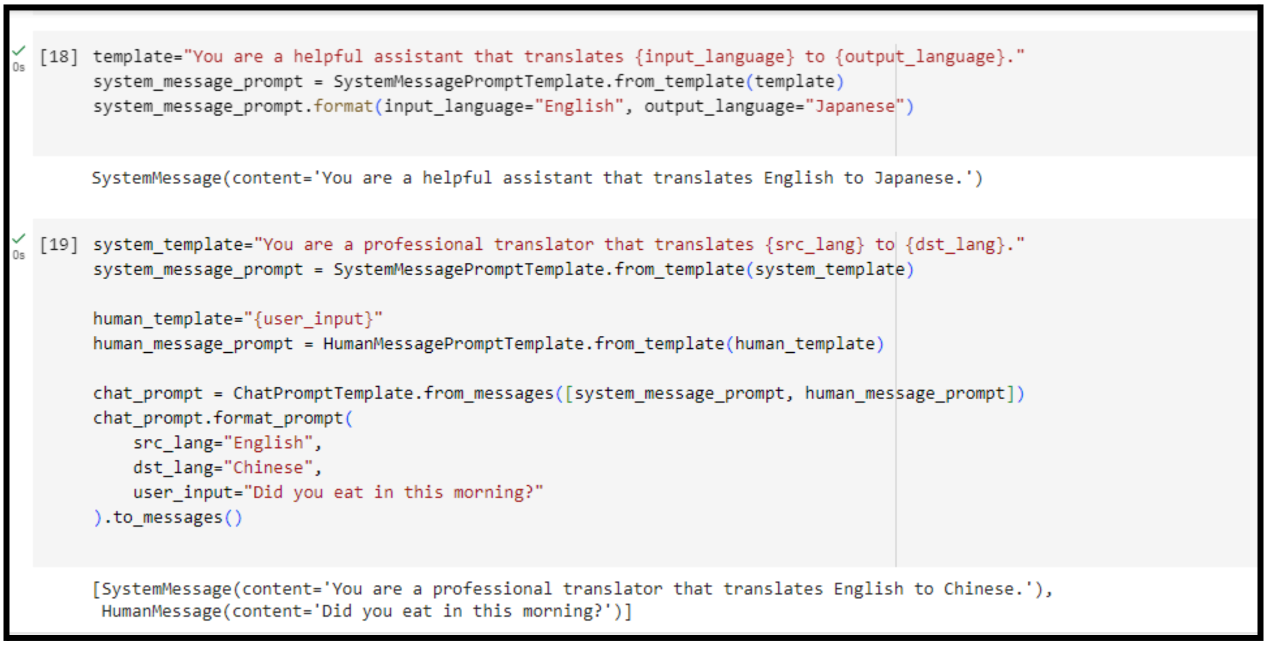


```
**Retrieve product_id of brand Adidas:**
````
SELECT product_id
FROM brands
WHERE brand = 'Adidas';
````

Determine the brand with least discount:
````
SELECT brand
FROM brands
ORDER BY discount ASC
LIMIT 1;
~~~
```

## LLM in summarization:

Here we can see the length of the question and the summarizes answer.



```
[87] prompt = PromptTemplate(template=template, input_variables=["input"])

     llm_chain = LLMChain(prompt=prompt, llm=llm)

     question = """
     The U.S. will be looking to snag their third straight World Cup title — and its fifth overall.

     The U.S. women's national team (USWNT) has held the No. 1 spot in FIFA's rankings for years, and is the odds-on favorite

     Indeed, the U.S. has had an interesting pattern of late: winning the World Cup, but losing at the Olympics. The USWNT won
     """
     answer = llm_chain.run(question)

[88] answer

     'The U.S. women's national team is the odds-on favorite to win this year's World Cup. The USWNT won the World Cup in 2015,
     then failed to medal at the 2016 Olympics. They won the 2019 World Cup, then took home the bronze at the Tokyo Olympics in
     2021.'
```



```
     2021.

[89] len(answer)

     250

[90] len(question)

     625
```
~~~

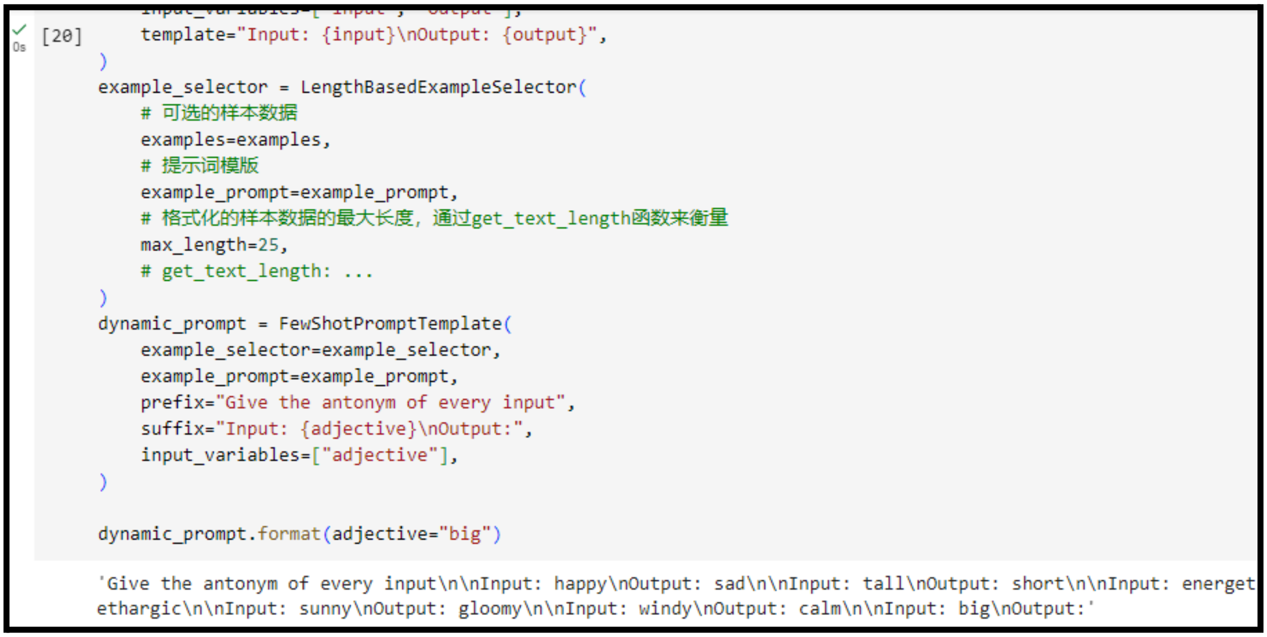
### To see if our llm can offer the same performance when considered to interact with new languages.



```
[91] from langchain import PromptTemplate
     template = """
     你精通多种语言，是专业的翻译官。你负责{src_lang}到{dst_lang}的翻译工作。
     """

     prompt = PromptTemplate.from_template(template)
     prompt.format(src_lang="英文", dst_lang="中文")

     '\n你精通多种语言，是专业的翻译官。你负责英文到中文的翻译工作。\n'
```



```
[18] template="You are a helpful assistant that translates {input_language} to {output_language}."
     system_message_prompt = SystemMessagePromptTemplate.from_template(template)
     system_message_prompt.format(input_language="English", output_language="Japanese")

     SystemMessage(content='You are a helpful assistant that translates English to Japanese.')

[19] system_template="You are a professional translator that translates {src_lang} to {dst_lang}."
     system_message_prompt = SystemMessagePromptTemplate.from_template(system_template)

     human_template="{user_input}"
     human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)

     chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt, human_message_prompt])
     chat_prompt.format_prompt(
         src_lang="English",
         dst_lang="Chinese",
         user_input="Did you eat in this morning?"
     ).to_messages()

     [SystemMessage(content='You are a professional translator that translates English to Chinese.'),
      HumanMessage(content='Did you eat in this morning?')]
```



```
         input_variables=["input", "output"],
[20]     template="Input: {input}\nOutput: {output}",
     )
     example_selector = LengthBasedExampleSelector(
         # 可选的样本数据
         examples=examples,
         # 提示词模版
         example_prompt=example_prompt,
         # 格式化的样本数据的最大长度，通过get_text_length函数来衡量
         max_length=25,
         # get_text_length: ...
     )
     dynamic_prompt = FewShotPromptTemplate(
         example_selector=example_selector,
         example_prompt=example_prompt,
         prefix="Give the antonym of every input",
         suffix="Input: {adjective}\nOutput:",
         input_variables=["adjective"],
     )

     dynamic_prompt.format(adjective="big")

     'Give the antonym of every input\n\nInput: happy\nOutput: sad\n\nInput: tall\nOutput: short\n\nInput: energet
     ethargic\n\nInput: sunny\nOutput: gloomy\n\nInput: windy\nOutput: calm\n\nInput: big\nOutput:'
```

### 2.4 [5 points] Reflect and Share

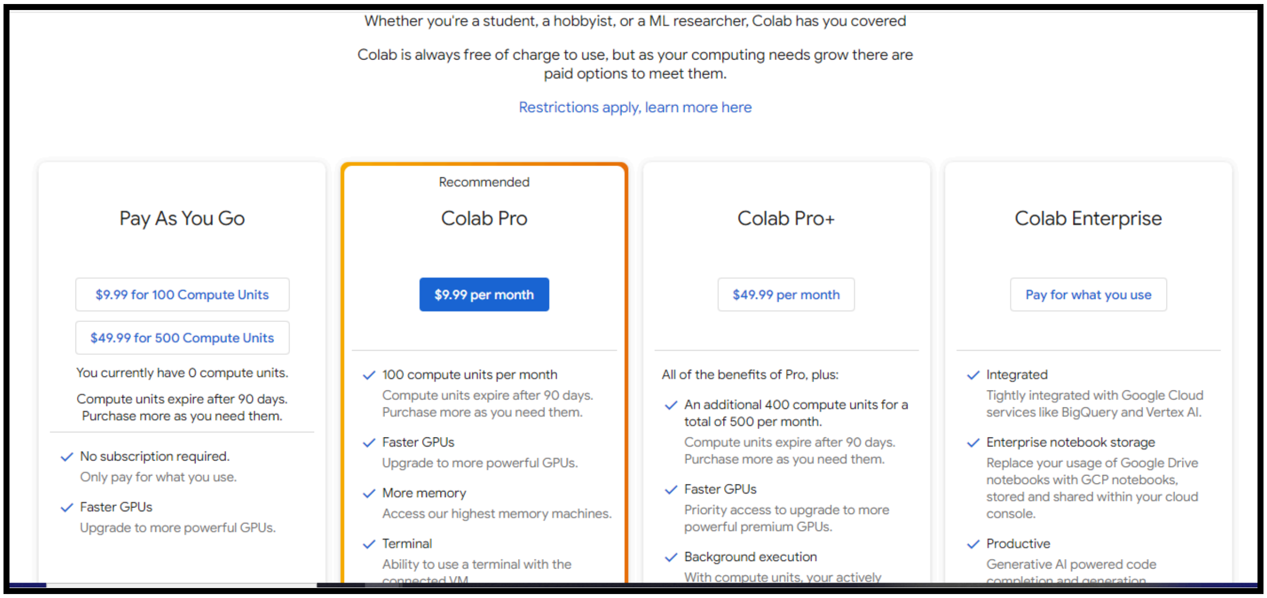
Summarize your learning experience, challenges faced, and insights gained. Document technical difficulties and possible best practices you discovered in the process.

- Challenges:

Gemini is not supported for university accounts, which was a main hindrance.

The free version had very few features to explore.

Working on all the subtasks took a lot of storage and time to install the dependencies.



Whether you're a student, a hobbyist, or a ML researcher, Colab has you covered

Colab is always free of charge to use, but as your computing needs grow there are paid options to meet them.

Restrictions apply, learn more here

Pay As You Go

$9.99 for 100 Compute Units

$49.99 for 500 Compute Units

You currently have 0 compute units.

Compute units expire after 90 days. Purchase more as you need them.

✓ No subscription required.
Only pay for what you use.

✓ Faster GPUs
Upgrade to more powerful GPUs.

Recommended

Colab Pro

$9.99 per month

✓ 100 compute units per month
Compute units expire after 90 days. Purchase more as you need them.

✓ Faster GPUs
Upgrade to more powerful GPUs.

✓ More memory
Access our highest memory machines.

✓ Terminal
Ability to use a terminal with the connected VM

Colab Pro+

$49.99 per month

All of the benefits of Pro, plus:

✓ An additional 400 compute units for a total of 500 per month.
Compute units expire after 90 days. Purchase more as you need them.

✓ Faster GPUs
Priority access to upgrade to more powerful premium GPUs.

✓ Background execution
With compute units, your actively

Colab Enterprise

Pay for what you use

✓ Integrated
Tightly integrated with Google Cloud services like BigQuery and Vertex AI.

✓ Enterprise notebook storage
Replace your usage of Google Drive notebooks with GCP notebooks, stored and shared within your cloud console.

✓ Productive
Generative AI powered code completion and generation

- Learning Experiences:

API Integration: Integrating APIs such as GitHub API for fetching repository statistics and Google's AI Gemini for natural language generation provides practical experience in leveraging external services to enhance application functionality.

Data Processing: Understanding how to parse JSON data retrieved from web APIs (in this case, GitHub API) and extract relevant information demonstrates basic data processing skills essential for working with external data sources.

Configuration Management: Exploring the role of the SparkConf class in Apache Spark highlights the importance of configuration management in distributed computing environments. Learning to configure Spark applications dynamically can optimize performance and resource utilization.

- Insights

Great at understanding other languages.

Natural Language Understanding: Gemini Pro can help users gain insights into natural language understanding by analyzing various text inputs and generating relevant responses or summaries. This can be particularly useful for tasks such as text summarization, sentiment analysis, or question answering.

Content Generation: Users can gain insights into content generation by exploring the capabilities of Gemini Pro in generating human-like text based on prompts or input data. This can be valuable for tasks such as content creation, creative writing, or generating personalized responses.

Language Modeling: Gemini Pro can provide insights into language modeling techniques and approaches by showcasing how language models can generate coherent and contextually relevant text based on input prompts. Users can learn about language model architectures, training data, and fine-tuning techniques.

Creative Exploration: Gemini Pro can inspire creative exploration by generating diverse and imaginative text outputs based on user prompts. Users can experiment with different input scenarios, explore various writing styles, and generate novel ideas or narratives.

Link:

https://medium.com/@sindhukotegar/exploration-with-gemini-pro-14b94261892f

**References:**

Hugging Face Transformers (https://huggingface.co/docs/transformers/en/index

Links to an external site.

LangChain: https://github.com/langchain-ai/langchain

Links to an external site.

OpenAI API (https://openai.com/

Links to an external site.

)

Google AI Platform (https://cloud.google.com/vertex-ai

Links to an external site.

)