Project Report: Anime Recommender System

Overview

This project aims to create an anime recommendation system using NLP techniques, fine-tuning embedding models, and leveraging open-source large language models (LLMs). The system uses user queries to recommend anime based on an anime synopsis and genre dataset, providing personalized suggestions.

Goals and Objectives

The main goal of this project was to develop an efficient anime recommendation system by using:

1. Fine-tuned embeddings to capture relevant context from anime descriptions.
2. A local LLM to generate coherent and relevant responses.
3. A retrieval-based QA approach to retrieve the most appropriate anime recommendations based on user queries.

This report presents the implementation steps and explains the techniques employed.

Environment Setup

Cell 1 - Setup and Installations

In this step, we set up the environment by disabling logging from Weights & Biases (WANDB) and silencing warnings that may clutter the output.

We also installed the necessary Python libraries, such as:

1. **Transformers** and **Sentence Transformers** for pre-trained models.
2. **LangChain** and **LangChain-HuggingFace** for efficient chaining of LLMs.
3. **ChromaDB** for managing vector stores and **datasets** to manage data.

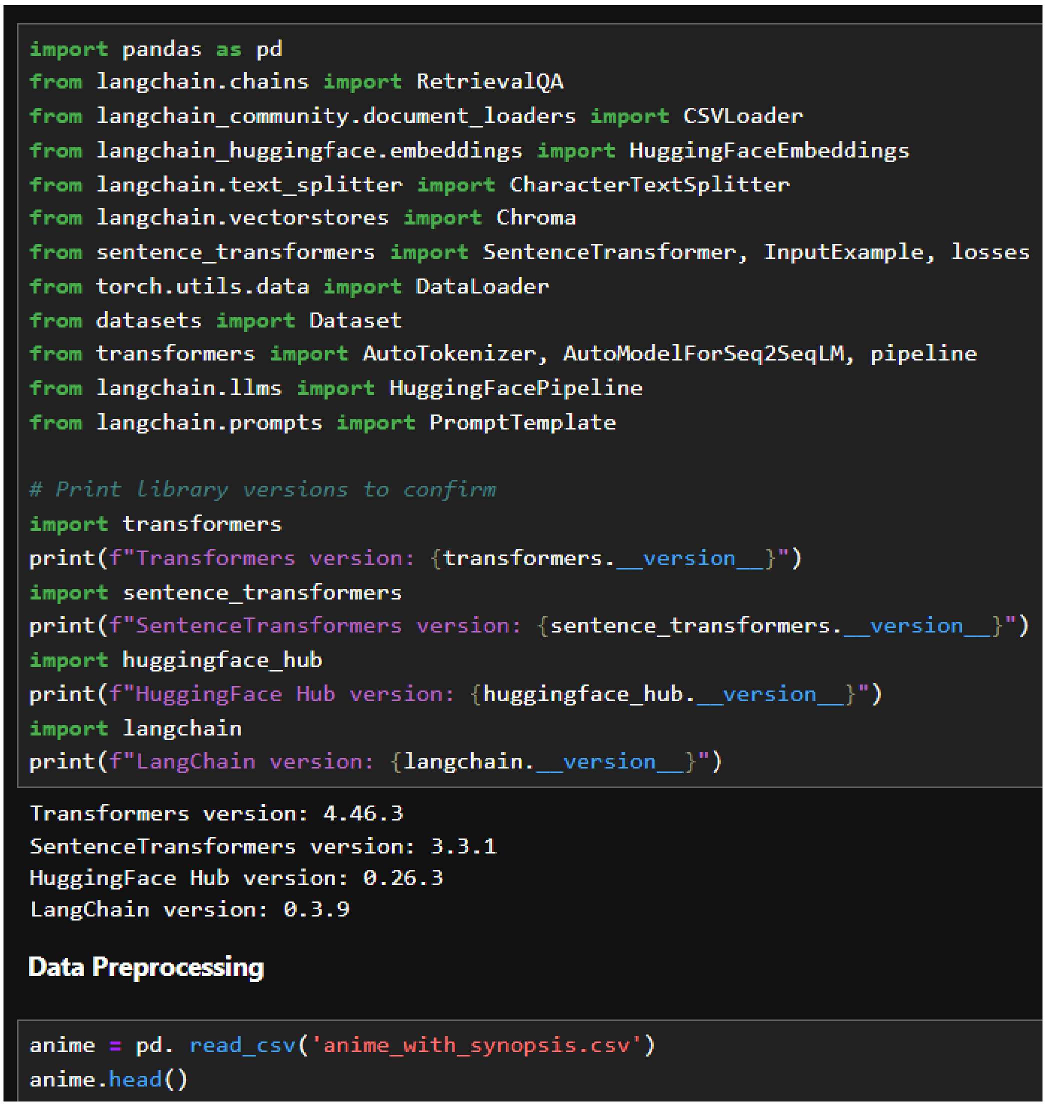
```
import os
os.environ['WANDB_MODE'] = 'disabled'
import warnings
warnings.filterwarnings('ignore')

!pip install --upgrade transformers sentence-transformers huggingface-hub
!pip install langchain langchain-huggingface
!pip install chromadb tiktoken
!pip install langchain-community
!pip install datasets
```

Explanation:

1. **Weights & Biases (WANDB)** is a tool often used for tracking experiments. Since it's not required for this implementation, we disabled it.
2. **Transformers** and **Sentence Transformers** are used for model fine-tuning and text embeddings. **LangChain** is a robust framework for connecting LLMs with various downstream tasks.

Data Preparation

Cell 2 & Cell 3 - Loading Libraries and Importing Data



```
import pandas as pd
from langchain.chains import RetrievalQA
from langchain_community.document_loaders import CSVLoader
from langchain_huggingface.embeddings import HuggingFaceEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import Chroma
from sentence_transformers import SentenceTransformer, InputExample, losses
from torch.utils.data import DataLoader
from datasets import Dataset
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, pipeline
from langchain.llms import HuggingFacePipeline
from langchain.prompts import PromptTemplate

# Print library versions to confirm
import transformers
print(f"Transformers version: {transformers.__version__}")
import sentence_transformers
print(f"SentenceTransformers version: {sentence_transformers.__version__}")
import huggingface_hub
print(f"HuggingFace Hub version: {huggingface_hub.__version__}")
import langchain
print(f"LangChain version: {langchain.__version__}")
```

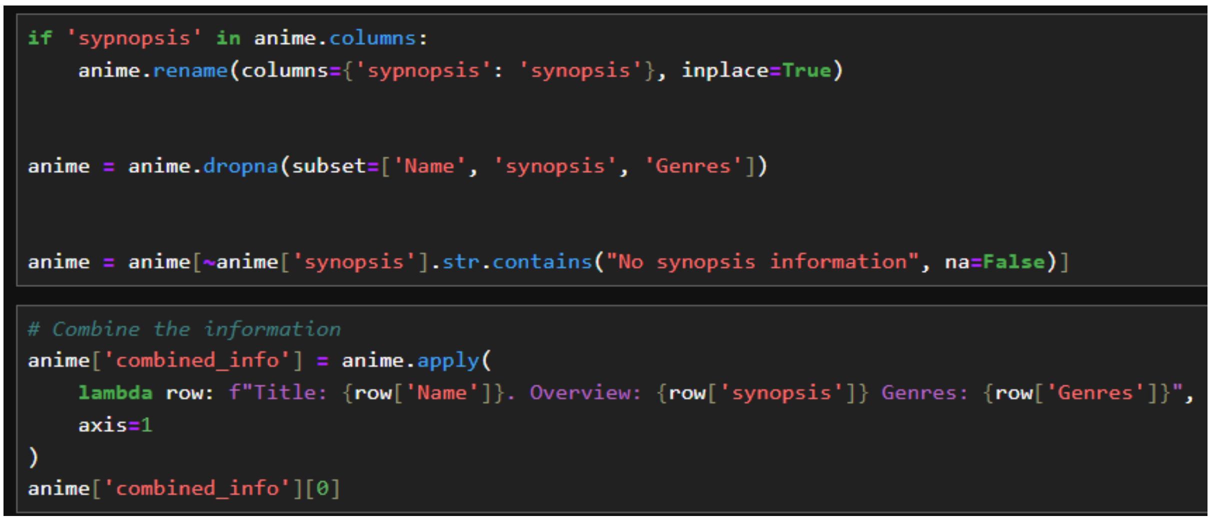
```
Transformers version: 4.46.3
SentenceTransformers version: 3.3.1
HuggingFace Hub version: 0.26.3
LangChain version: 0.3.9
```

Data Preprocessing

```
anime = pd. read_csv('anime_with_synopsis.csv')
anime.head()
```

We loaded the necessary libraries and the dataset containing anime titles, synopses, and genres.

Cell 4 & Cell 5 - Preprocessing the Data



```python
if 'sypnopsis' in anime.columns:
    anime.rename(columns={'sypnopsis': 'synopsis'}, inplace=True)


anime = anime.dropna(subset=['Name', 'synopsis', 'Genres'])


anime = anime[~anime['synopsis'].str.contains("No synopsis information", na=False)]
```

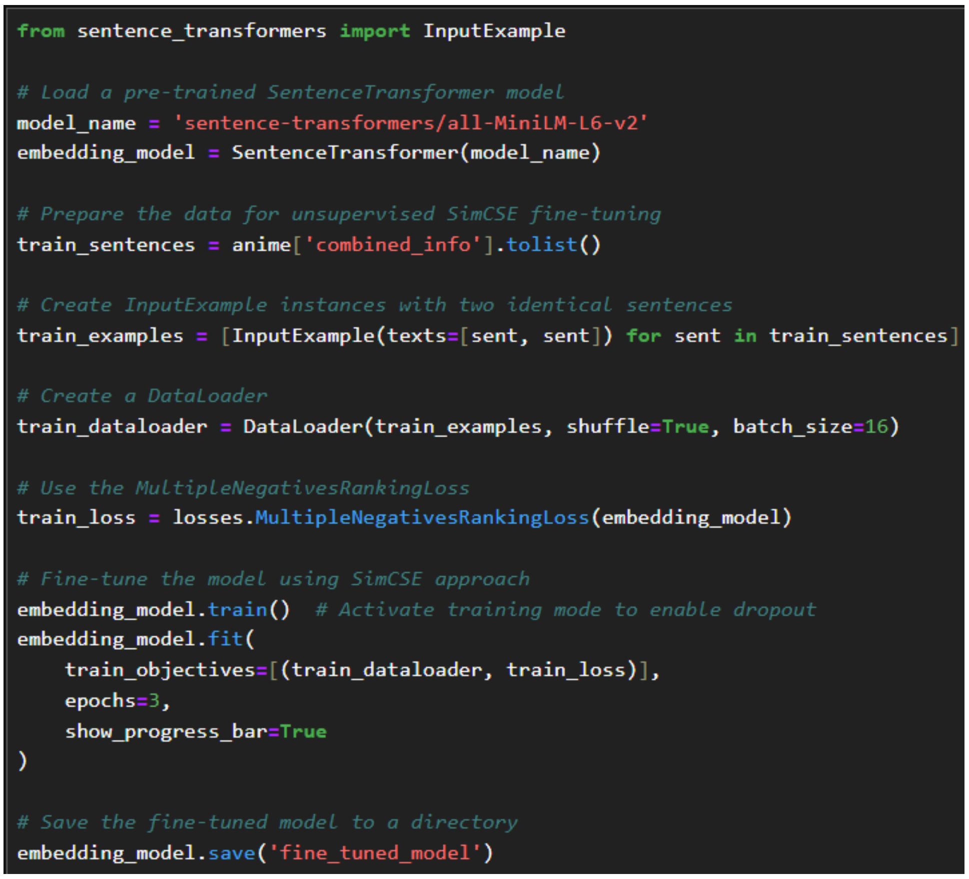
```python
# Combine the information
anime['combined_info'] = anime.apply(
    lambda row: f"Title: {row['Name']}. Overview: {row['synopsis']} Genres: {row['Genres']}",
    axis=1
)
anime['combined_info'][0]
```

Explanation:

1. We corrected a typo in the data and removed entries with incomplete or uninformative descriptions.
2. We combined the title, synopsis, and genres into a unified "combined_info" field, which provides a richer context to the model for embedding generation.

Fine-Tuning Embedding Model

Cell 8 - Fine-Tuning the Embedding Model



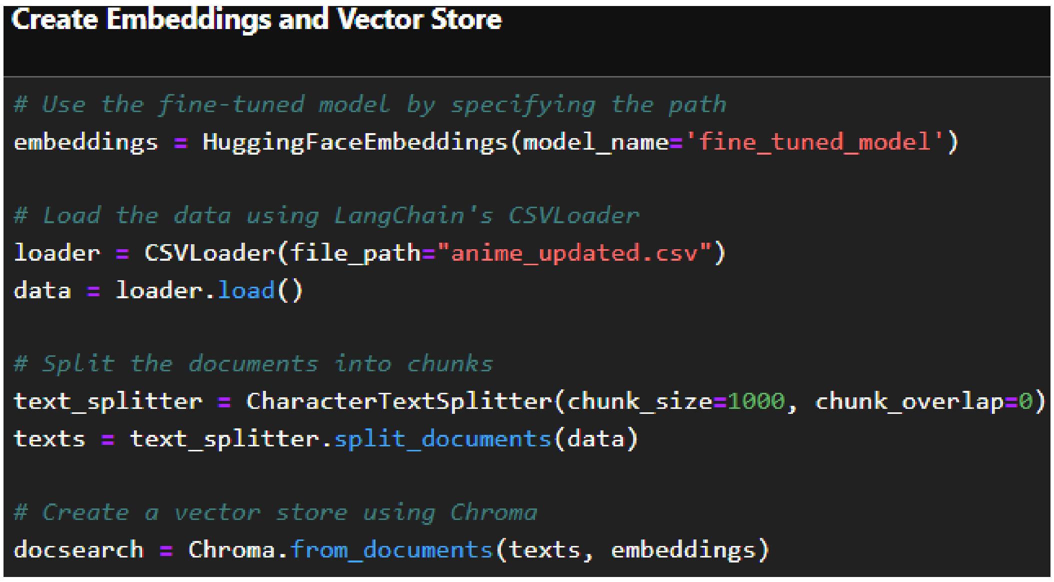
```python
from sentence_transformers import InputExample

# Load a pre-trained SentenceTransformer model
model_name = 'sentence-transformers/all-MiniLM-L6-v2'
embedding_model = SentenceTransformer(model_name)

# Prepare the data for unsupervised SimCSE fine-tuning
train_sentences = anime['combined_info'].tolist()

# Create InputExample instances with two identical sentences
train_examples = [InputExample(texts=[sent, sent]) for sent in train_sentences]

# Create a DataLoader
train_dataloader = DataLoader(train_examples, shuffle=True, batch_size=16)

# Use the MultipleNegativesRankingLoss
train_loss = losses.MultipleNegativesRankingLoss(embedding_model)

# Fine-tune the model using SimCSE approach
embedding_model.train()  # Activate training mode to enable dropout
embedding_model.fit(
    train_objectives=[(train_dataloader, train_loss)],
    epochs=3,
    show_progress_bar=True
)

# Save the fine-tuned model to a directory
embedding_model.save('fine_tuned_model')
```

Explanation:

1. **SentenceTransformer** is used to load a pre-trained embedding model, which we fine-tuned with the SimCSE (Simple Contrastive Learning of Sentence Embeddings) approach.
2. **Fine-tuning** ensures that the embeddings generated are more suited to the anime context, improving recommendation accuracy.
3. **MultipleNegativesRankingLoss** was used to train the model to generate better contextual embeddings, minimizing the loss for relevant recommendations.

Creating Embeddings and Vector Store

Cell 9 - Creating a Vector Store for Retrieval



Create Embeddings and Vector Store

```python
# Use the fine-tuned model by specifying the path
embeddings = HuggingFaceEmbeddings(model_name='fine_tuned_model')

# Load the data using LangChain's CSVLoader
loader = CSVLoader(file_path="anime_updated.csv")
data = loader.load()

# Split the documents into chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(data)

# Create a vector store using Chroma
docsearch = Chroma.from_documents(texts, embeddings)
```

Explanation:

1. We used **HuggingFaceEmbeddings** with the fine-tuned model to generate embeddings for each dataset chunk.
2. **CharacterTextSplitter** splits documents into manageable chunks to facilitate efficient vector searches.
3. **Chroma** is a vector store that enables fast retrieval based on cosine similarity between the query and anime embeddings.

LLM and Retrieval Chain Setup

Cell 10 - Defining the Language Model

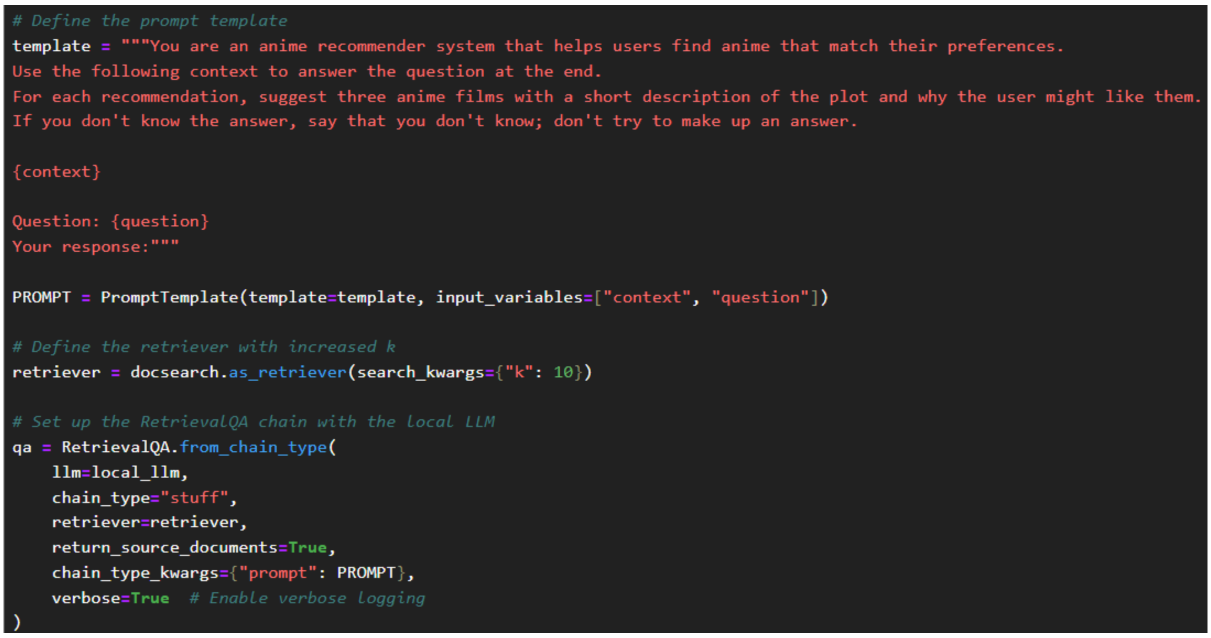
```python
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, pipeline
from langchain.llms import HuggingFacePipeline

# Load the Flan-T5 model and tokenizer
model_name = 'google/flan-t5-large'

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

# Create a pipeline for text generation
hf_pipeline = pipeline(
    'text2text-generation',
    model=model,
    tokenizer=tokenizer,
    max_length=512,
    temperature=0.9,  # Increased temperature for more creativity
    top_p=0.95,       # Increased top_p to allow more diverse tokens
    repetition_penalty=1.1,
    do_sample=True
)

# Wrap the pipeline in a LangChain LLM
local_llm = HuggingFacePipeline(pipeline=hf_pipeline)
```

Explanation:

1. We used **Flan-T5**, an open-source LLM, which we loaded using HuggingFace's AutoTokenizer and AutoModelForSeq2SeqLM.
2. The **pipeline** configuration controls the generation characteristics, such as **temperature** (controls randomness), **top_p** (sampling diversity), and **repetition_penalty** (to avoid repetitive responses).
3. The model was wrapped in a LangChain LLM wrapper for our retrieval-based QA chain.

RetrievalQA Chain Setup

Cell 11 - Setting Up RetrievalQA Chain



```
# Define the prompt template
template = """You are an anime recommender system that helps users find anime that match their preferences.
Use the following context to answer the question at the end.
For each recommendation, suggest three anime films with a short description of the plot and why the user might like them.
If you don't know the answer, say that you don't know; don't try to make up an answer.

{context}

Question: {question}
Your response:"""

PROMPT = PromptTemplate(template=template, input_variables=["context", "question"])

# Define the retriever with increased k
retriever = docsearch.as_retriever(search_kwargs={"k": 10})

# Set up the RetrievalQA chain with the local LLM
qa = RetrievalQA.from_chain_type(
    llm=local_llm,
    chain_type="stuff",
    retriever=retriever,
    return_source_documents=True,
    chain_type_kwargs={"prompt": PROMPT},
    verbose=True  # Enable verbose logging
)
```

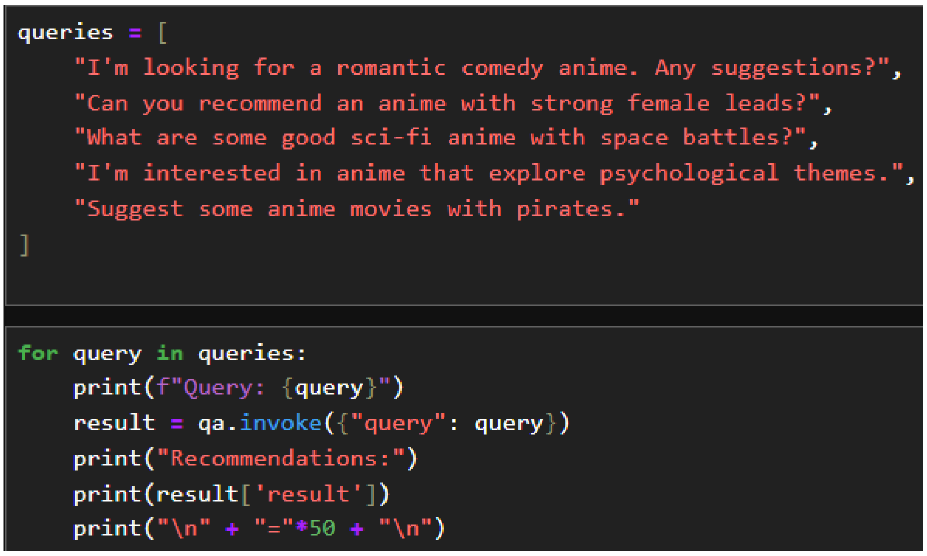
Explanation:

1. **PromptTemplate** defines how queries are formatted before being sent to the LLM.
2. The **retriever** retrieves the most relevant documents from the vector store (k=10 implies we retrieve 10 documents for each query).
3. **RetrievalQA** uses these retrieved documents as context for generating coherent anime recommendations.

Testing the Model and Personalized Prompts

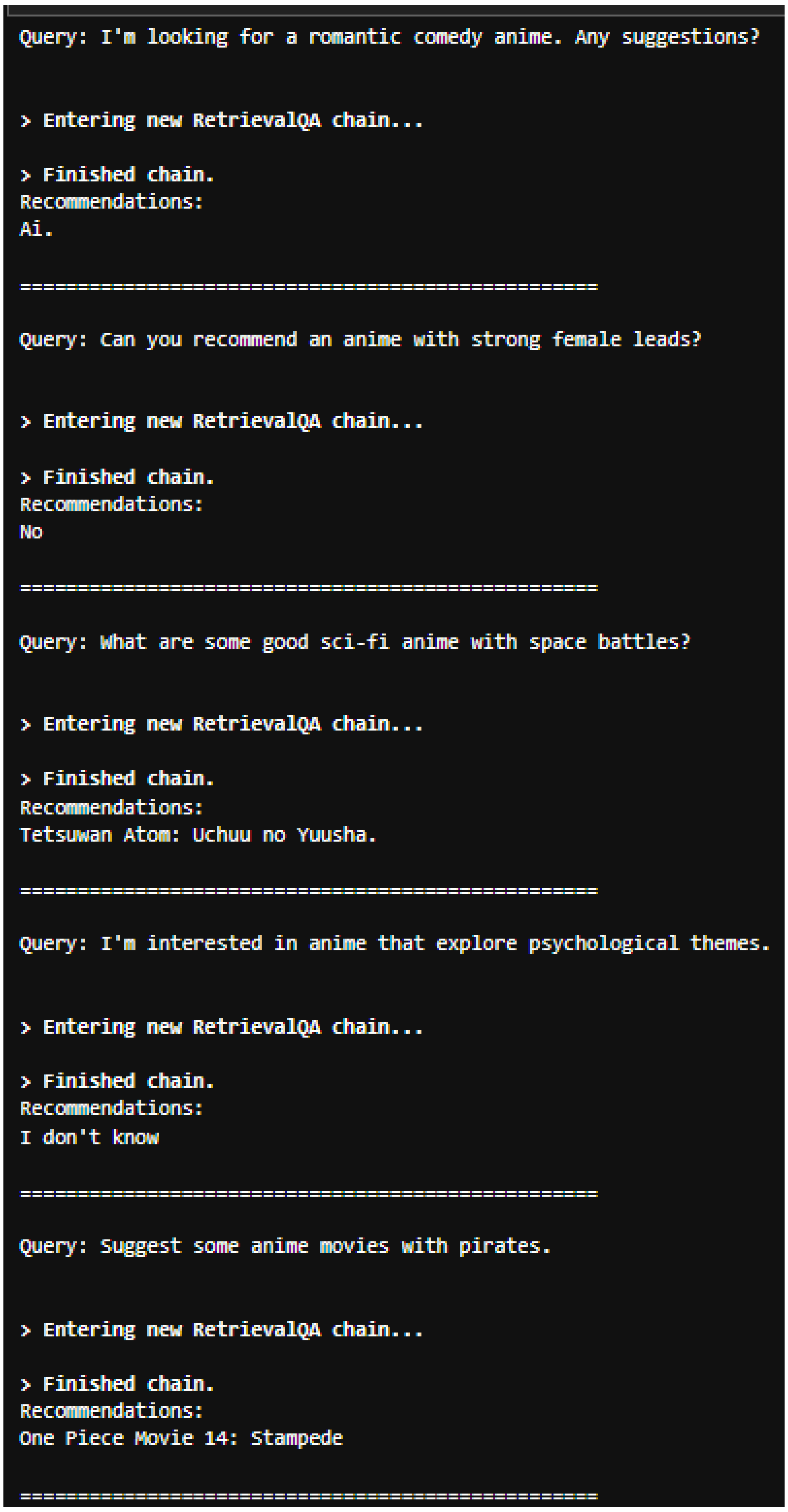
Cells 12-19 - Querying the System

We used different user queries to evaluate the recommender system's output. For personalization, additional user information (e.g., age and gender) was included in the prompts to generate more customized recommendations.



```
queries = [
    "I'm looking for a romantic comedy anime. Any suggestions?",
    "Can you recommend an anime with strong female leads?",
    "What are some good sci-fi anime with space battles?",
    "I'm interested in anime that explore psychological themes.",
    "Suggest some anime movies with pirates."
]
```

```
for query in queries:
    print(f"Query: {query}")
    result = qa.invoke({"query": query})
    print("Recommendations:")
    print(result['result'])
    print("\n" + "="*50 + "\n")
```



```
Query: I'm looking for a romantic comedy anime. Any suggestions?


> Entering new RetrievalQA chain...

> Finished chain.
Recommendations:
Ai.

==================================================

Query: Can you recommend an anime with strong female leads?


> Entering new RetrievalQA chain...

> Finished chain.
Recommendations:
No

==================================================

Query: What are some good sci-fi anime with space battles?


> Entering new RetrievalQA chain...

> Finished chain.
Recommendations:
Tetsuwan Atom: Uchuu no Yuusha.

==================================================

Query: I'm interested in anime that explore psychological themes.


> Entering new RetrievalQA chain...

> Finished chain.
Recommendations:
I don't know

==================================================

Query: Suggest some anime movies with pirates.


> Entering new RetrievalQA chain...

> Finished chain.
Recommendations:
One Piece Movie 14: Stampede

==================================================
```

Explanation:

1. We iterated through multiple user queries and obtained relevant anime recommendations, verifying that the model can have diverse and meaningful suggestions.

Summary: The project integrates several techniques, from embedding-based document retrieval to leveraging an LLM for coherent responses. It uses open-source models and state-of-the-art techniques to generate contextually relevant anime recommendations.