Leveraging AI for Advanced Cricket Statistical Analytics -Vishrut Kannan

In today's data-driven world, the ability to analyze vast amounts of information quickly and accurately is crucial. Artificial Intelligence (AI) tools, especially those leveraging natural language processing and embeddings, have revolutionized data analysis. This article explores a practical application of AI in providing advanced analytics for local cricket leagues that may not have access to comprehensive platforms like ESPNcricinfo. By using Python scripts integrated with LangChain, we aim to deliver detailed and insightful statistics and analyses. We will delve deeply into the code's workings to understand how it processes data and generates insightful responses using Retrieval-Augmented Generation (RAG). RAG combines data retrieval with generative models to enhance the quality of AI-generated responses by leveraging relevant information.

Code Overview

create_database.py

Loading environment variables:

```
from dotenv import load_dotenv
import os

load_dotenv()
openai.api_key = os.environ['OPENAI_API_KEY']
import shutil
```

This section of the script is responsible for securely loading environment variables from a .env file. The dotenv package is used to read the .env file, which typically contains sensitive information such as API keys. By calling load_dotenv(), the script populates the environment variables into the Python process.

The os.environ['OPENAI_API_KEY'] line retrieves the OpenAI API key from the environment variables, allowing the script to authenticate with OpenAI's services without hardcoding the key directly into the code. This approach helps in maintaining security by keeping sensitive information out of the source code. The import shutil statement indicates that the script also uses the shutil module, which is commonly used for high-level file operations, although its specific use is not detailed in this snippet.

Generating the Data Store

```
def generate_data_store():
    documents = load_documents()
    chunks = split_text(documents)
    save_to_chroma(chunks)
```

The generate_data_store function is designed to prepare and store data for future queries. Here’s a breakdown of its steps:

1. load_documents(): This function is responsible for loading the documents from a specified source or directory. The documents could be in various formats such as text files, PDFs, or any other type of document. The purpose is to gather the raw data that will be processed.
2. split_text(documents): After loading the documents, this function takes the raw text and splits it into manageable chunks. This is often necessary to handle large texts efficiently and to improve search and retrieval performance. Chunking helps in breaking down the text into smaller, more manageable pieces that can be individually indexed and queried.
3. save_to_chroma(chunks): Finally, this function saves the chunks into a Chroma vector store. The Chroma vector store is a type of database designed to store and manage vector embeddings, which are numerical representations of text. Saving the chunks in this manner allows for efficient and scalable querying in the future, forming the retrieval component of RAG.

Loading Documents

```
def load_documents():
    loader = DirectoryLoader(DATA_PATH, glob="*.txt")
    documents = loader.load()
    return documents
```

The load_documents function is designed to efficiently load text files from a specified directory for further processing. It leverages the DirectoryLoader class to scan the directory path defined by DATA_PATH and reads all files with a .txt extension. This approach allows for a flexible and scalable integration of various text data sources, as it can accommodate any number of text files within the directory. By encapsulating the loading logic within this function, the process of importing and managing text data becomes streamlined and manageable, ensuring that all relevant documents are available for subsequent analysis or processing tasks, forming the initial step in the RAG process.

Splitting Text

```
def split_text(documents: list[Document]):
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=300,
        chunk_overlap=100,
        length_function=len,
        add_start_index=True,
    )
    chunks = text_splitter.split_documents(documents)
    print(f"Split {len(documents)} documents into {len(chunks)} chunks.")

    document = chunks[10]
    print(document.page_content)
    print(document.metadata)

    return chunks
```

The split_text function addresses the need to manage and process large text documents by breaking them into smaller, more manageable chunks. It utilizes the RecursiveCharacterTextSplitter to divide the documents into chunks of 300 characters each, with a 100-character overlap between chunks. This overlapping ensures that important contextual information is preserved across chunks, which is crucial for maintaining the coherence and relevance of the data during retrieval or analysis. The function prints out a summary indicating the number of documents split and the total number of chunks generated, providing an immediate overview of the processing results. Additionally, it showcases an example chunk, displaying its content and metadata to illustrate how the text is divided and indexed. This methodical chunking process enhances the efficiency of data handling and retrieval, making it easier to work with extensive text corpora and forming a key part of the retrieval component in RAG.

Saving to Chroma

```
def save_to_chroma(chunks: list[Document]):
    # Clear out the database first.
    if os.path.exists(CHROMA_PATH):
        shutil.rmtree(CHROMA_PATH)

    # Create a new DB from the documents.
    db = Chroma.from_documents(
        chunks, OpenAIEmbeddings(), persist_directory=CHROMA_PATH
    )
    db.persist()
    print(f"Saved {len(chunks)} chunks to {CHROMA_PATH}.")
```

The save_to_chroma function is crucial for storing text chunks into a Chroma vector store, ensuring efficient and scalable data retrieval, which is central to the RAG approach. The process begins by checking if the Chroma directory, specified by CHROMA_PATH, already exists. If it does, the function uses shutil.rmtree to delete the existing directory and its contents, effectively clearing out any previous data. This step ensures that the database starts fresh, avoiding potential conflicts or outdated information.

After clearing the old data, the function proceeds to create a new Chroma database. This is accomplished using the Chroma.from_documents method, which takes the list of text chunks and embeds them using OpenAIEmbeddings. The embedding process converts the text into vector representations, allowing for efficient similarity searches and queries. The newly created Chroma database is then persisted to the specified directory using the db.persist() method, which saves the data to disk for future access.

The function concludes by printing a summary that indicates the number of chunks successfully saved to the Chroma vector store and confirms the storage location. This provides a clear and immediate feedback on the operation, helping to ensure that the data has been correctly stored and is ready for use in subsequent queries.

query_data.py

Setting up CLI and Query Processing

```
def main():
    # Create CLI.
    parser = argparse.ArgumentParser()
    parser.add_argument("query_text", type=str, help="The query text.")
    args = parser.parse_args()
    query_text = args.query_text
```

The main function is designed to facilitate user interaction with the script through a command-line interface (CLI). At its core, this section leverages the argparse library to manage command-line arguments. By creating an ArgumentParser instance, the script defines a command-line argument query_text, which expects a string input from the user. This argument represents the query text that the user wishes to process. The parser.parse_args() method captures the command-line inputs and assigns them to the args variable. From this, the query_text is extracted, making it available for further processing within the script. This setup allows users to provide their queries directly when running the script, ensuring a flexible and user-friendly interface for interacting with the query processing functionality.

Preparing the Database

```
    embedding_function = OpenAIEmbeddings()
    db = Chroma(persist_directory=CHROMA_PATH, embedding_function=embedding_function)
```

This section of the script is focused on initializing and preparing the Chroma vector store for use in similarity searches, an integral part of the RAG process. First, an OpenAIEmbeddings instance is created, which is responsible for converting text into vector embeddings using OpenAI's pre-trained models. These embeddings represent the text in a numerical format that facilitates efficient similarity comparisons. Next, the Chroma object is instantiated with the persist_directory set to CHROMA_PATH and the embedding_function set to the previously created OpenAIEmbeddings. This initialization process loads the Chroma database from the specified directory, ensuring that it is ready for performing similarity searches based on the vector embeddings. By preparing the database in this manner, the script sets up a robust framework for querying and retrieving relevant information efficiently, aligning with the retrieval component of RAG.

Formatting and Generating the Best Response

```
    context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in results])
    prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
    prompt = prompt_template.format(context=context_text, question=query_text)
    print(prompt)

    model = ChatOpenAI()
    response_text = model.predict(prompt)
```

In this section, the script is focused on crafting a prompt for the AI model and generating a response based on the processed query, representing the generative component of RAG. First, it formats the context by concatenating the content of each relevant document into a single string. This is achieved by joining the page_content of each document with a separator \n\n---\n\n, which helps delineate

individual documents within the context. The resulting context_text provides a comprehensive background for the AI model to consider when formulating a response.

Next, a ChatPromptTemplate is created using a predefined template stored in PROMPT_TEMPLATE. The prompt_template.format() method is then used to insert the formatted context_text and the user's query_text into the template, generating a complete prompt that is ready for the AI model. This prompt serves as the input for the AI model, guiding it in producing a relevant and accurate response.

The ChatOpenAI model is instantiated, and the model.predict(prompt) method is used to generate a response based on the crafted prompt. This step involves the AI model analyzing the prompt and producing a textual response that addresses the user's query in the context provided.

Outputting Results

```
    sources = [doc.metadata.get("source", None) for doc, _score in results]
    formatted_response = f"Response: {response_text}\nSources: {sources}"
    print(formatted_response)
```

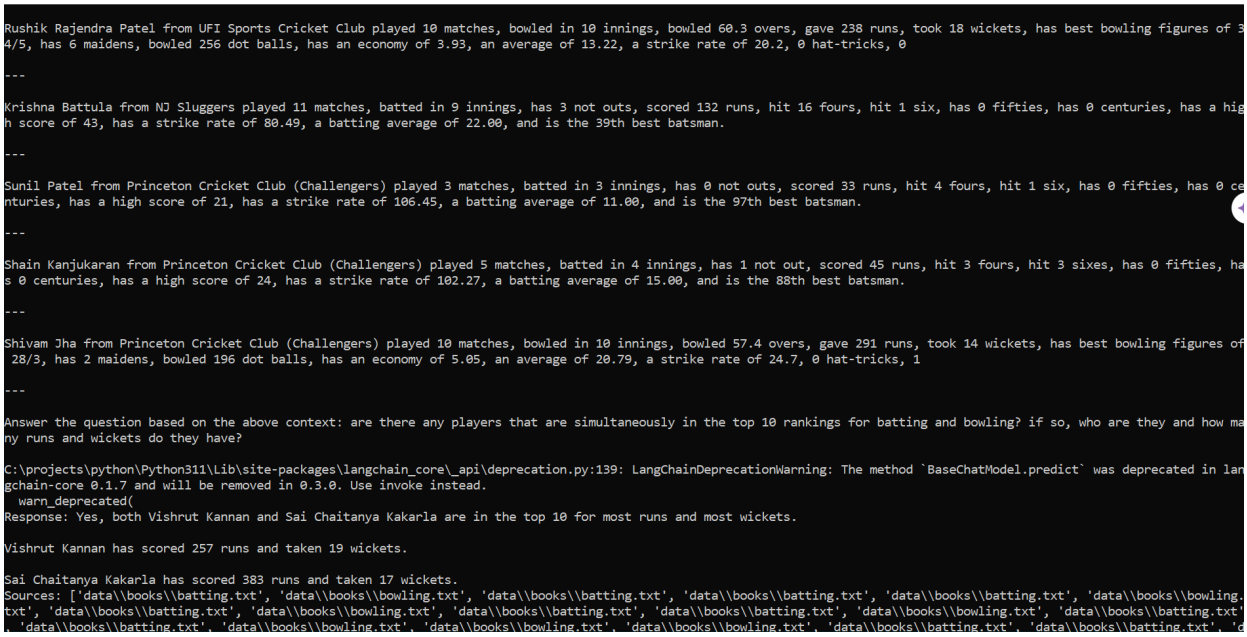
Once the AI model generates a response, this section handles the output of results, ensuring that both the response and its sources are clearly presented. The script first extracts the sources associated with each document from the results, accessing the metadata for each document to retrieve the source information. These sources are collected into a list.

The final response is then formatted into a clear and user-friendly output string, which includes the AI-generated response and the corresponding sources. This string is stored in the variable formatted_response, which is printed to provide the user with both the response and context regarding where the information originated from. By including source information, the script enhances transparency and helps users understand the basis of the provided answers.

Practical Example

Query: are there any players that are simultaneously in the top 10 rankings for batting and bowling? If so, who are they and how many runs and wickets do they have?
Input Data:

- *Batting.txt*
    - *Has records of all players' batting statistics*
- *Bowling.txt*
    - *Has records of all players' bowling statistics*



When given “are there any players that are simultaneously in the top 10 rankings for batting and bowling? If so, who are they and how many runs and wickets do they have?" as a query, the script effectively demonstrates the combination of AI and embeddings in generating insightful responses. By retrieving relevant information and crafting a well-informed reply, the system showcases its ability to handle complex queries using RAG. It retrieves relevant chunks from the Chroma vector store, formats them into a context, and uses this context to generate a comprehensive response, exemplifying the effective integration of retrieval and generation.

Output:

Yes, both Vishrut Kannan and Sai Chaitanya Kakarla are in the top 10 for most runs and most wickets.

Vishrut Kannan has scored 257 runs and taken 19 wickets.

Sai Chaitanya Kakarla has scored 383 runs and taken 17 wickets.
Sources: ['data\\books\\batting.txt', 'data\\books\\bowling.txt', 'data\\books\\batting.txt', 'data\\books\\batting.txt', 'data\\books\\batting.txt', 'data\\books\\bowling.txt', 'data\\books\\batting.txt', 'data\\books\\bowling.txt', 'data\\books\\batting.txt', 'data\\books\\batting.txt', 'data\\books\\bowling.txt', 'data\\books\\batting.txt', 'data\\books\\batting.txt', 'data\\books\\bowling.txt', 'data\\books\\bowling.txt', 'data\\books\\bowling.txt', 'data\\books\\batting.txt', 'data\\books\\batting.txt', 'data\\books\\batting.txt', 'data\\books\\bowling.txt']

Why RAG is the Optimal Choice for This Project

Retrieval-Augmented Generation (RAG) combines a retrieval component with a generative model, making it particularly well-suited for this cricket league analytics project. Unlike fine-tuning, which can

be both costly and time-consuming, RAG offers a more flexible and efficient approach. It dynamically integrates the latest information into responses, ensuring that the answers are not only accurate but also contextually rich and up-to-date.

One of the key advantages of RAG is its flexibility and scalability. It adapts to new data without the need for frequent retraining, a common necessity for fine-tuned models which can quickly become outdated. This adaptability is crucial for cricket analytics, where player performance and match data are constantly changing.

Moreover, RAG enhances contextual understanding by retrieving relevant documents and incorporating them into the response generation process. This results in more informed and precise answers, which is essential for addressing complex queries in cricket analytics. The ability to pull from a broad base of current information helps maintain the accuracy and relevance of the insights provided.

RAG's capability to incorporate real-time information and deliver accurate, context-rich answers makes it superior to fine-tuning for dynamic and information-rich applications like cricket league analytics. Its combination of flexibility, enhanced contextual understanding, and cost efficiency provides significant advantages in the fast-paced environment of sports analytics.

Conclusion

The integration of advanced AI techniques in cricket league analytics provides a powerful tool for generating insightful and accurate responses. By leveraging embeddings and efficient querying mechanisms, we can deliver comprehensive answers to complex questions. This article has provided a detailed look at the code and methodology behind this process, offering a practical example of its application in the context of local cricket league analytics. The use of Retrieval-Augmented Generation (RAG) significantly enhances the quality of AI-generated responses by combining efficient data retrieval with advanced generative techniques, highlighting the substantial benefits of this approach in data-driven decision-making for sports analytics.