

STARS Tutoring Chatbot Documentation

Overview

The STARS Tutoring Chatbot is a Streamlit application designed to assist college students in learning programming concepts and improving their coding skills. The application features two interactive chatbots:

- **TutorBot:** An advanced computer science tutor that helps students understand programming concepts and problem-solving techniques without providing direct answers.
- **CodeBot:** Engages users in an educational game to explore how to build programs using Language Learning Models (LLMs), focusing on prompt engineering and code generation.

Both bots leverage OpenAI's GPT models via the LangChain framework to generate contextual and educational responses.

Table of Contents

1. [Application Structure](#)
2. [Components](#)
3. [Packages and Dependencies](#)
4. [Class and Method Explanations](#)
 - [LoggingSemanticSimilarityExampleSelector](#)
 - [ChatChainSingleton](#)
 - [TutorBot and CodeBot Classes](#)
5. [Usage Instructions](#)
6. [Configuration and Setup](#)
7. [Security Considerations](#)
8. [Conclusion](#)

Application Structure

The application is organized into the following components:

components/

└─ chatbot.py # Base Chatbot class

└─ tutorbot.py # TutorBot implementation

```
|— codebot.py    # CodeBot implementation
main.py         # Main Streamlit application
```

Components

TutorBot (components/tutorbot.py)

- Implements an advanced tutoring assistant using LangChain.
- Features:
 - **Custom Example Selector:** Selects relevant examples for few-shot learning.
 - **Singleton Pattern:** Ensures only one instance of the chat chain is created.
 - **Chat Chain Initialization:** Prepares prompt templates and initializes the language model.

CodeBot (components/codebot.py)

- Implements an educational game bot.
 - Features:
 - Custom examples and prompt templates for interactive code learning.
 - Similar structure to TutorBot.
-

Main Application (main.py)

The Streamlit application provides:

- **User Authentication:** Sign-up and sign-in functionality using MongoDB to store user credentials.
 - **Chat Interface:** Allows users to interact with the selected chatbot.
 - **Bot Selection Sidebar:** Users can choose between TutorBot and CodeBot.
 - **Session Management:** Maintains user session state, including login status and message history.
-

Packages and Dependencies

The application relies on several external packages:

Package	Purpose
streamlit	For building the web application interface.
pymongo	For MongoDB integration to store and retrieve user data.
langchain	Provides tools for chaining together prompts and GPT model interactions.
openai	Interfaces with OpenAI GPT models for response generation.
logging	Logs information and errors for debugging and monitoring.
os	Accesses environment variables.
typing	Adds type hints for better code readability and maintainability.

Class and Method Explanations

LoggingSemanticSimilarityExampleSelector

- **Inheritance:** Inherits from SemanticSimilarityExampleSelector.
 - **Purpose:** Selects the most relevant few-shot examples based on semantic similarity to the user's input and logs the selected examples.
 - **Methods:**
 - `select_examples(self, input_variables)`: Overrides the parent method to include logging of the selected examples.
-

ChatChainSingleton

- **Purpose:** Implements the Singleton design pattern to ensure only one instance of the chat chain is created.
- **Attributes:**
 - `_instance`: Stores the singleton instance.
 - `chain`: The language model chain used for generating responses.
 - `prompt`: The final prompt template used in the chain.
 - `model`: The name of the OpenAI GPT model to use.
- **Methods:**

- `__new__(cls, *args, **kwargs)`: Creates a new instance if one doesn't exist.
 - `initialize_chain(cls, model: str)`: Sets up the chain with examples, vector stores, prompt templates, and the chat model.
 - `change_model(new_model: str)`: Allows changing the language model used.
-

TutorBot and CodeBot Classes

- **Inheritance:** Both inherit from the base Chatbot class.
 - **Purpose:** Provide specialized implementations for the two types of chatbots, utilizing their respective ChatChainSingleton instances.
 - **Key Methods:**
 - `__init__(self, api_key, mongo_uri)`: Initializes the chatbot by setting up the chain and prompt.
 - `generate_response(self, user_id, messages)`: Generates a response based on user input and conversation history.
-

Usage Instructions

Clone the Repository

```
git clone https://github.com/your-repository/stars-tutoring-chatbot.git
```

```
cd stars-tutoring-chatbot
```

Install Dependencies

```
pip install streamlit pymongo langchain openai
```

Configure Environment Variables

Set up your OpenAI API key:

```
export OPENAI_API_KEY='your-openai-api-key'
```

Configure MongoDB

Ensure MongoDB is running locally or update the connection string in `main.py` to match your MongoDB instance.

Run the Application

```
streamlit run main.py
```

Configuration and Setup

Environment Variables

- OPENAI_API_KEY: Your OpenAI API key for GPT model access.

MongoDB Configuration

- Default connection: mongodb://localhost:27017/
 - Database: user_data
 - Collection: users (stores user credentials and chat history).
-

Security Considerations

1. API Key Security:

- Store the OpenAI API key in environment variables.
- Avoid exposing sensitive keys in logs or code repositories.

2. Password Storage:

- Currently, passwords are stored as plain text.
- Use a hashing algorithm (e.g., bcrypt) for secure password storage.

3. Input Validation:

- Sanitize and validate user inputs to prevent injection attacks.

4. HTTPS:

- Use HTTPS to encrypt data in transit if deploying the application online.
-

Conclusion

The **STARS Tutoring Chatbot** combines advanced natural language processing with an interactive user interface to provide an engaging and educational experience. By leveraging LangChain and OpenAI GPT models, the application delivers a personalized tutoring experience while promoting independent learning.

TutorBot and CodeBot Classes

Inheritance:

Both inherit from the base Chatbot class.

Purpose:

Provide specialized implementations for the two types of chatbots, utilizing their respective ChatChainSingleton instances.

Methods:

- `__init__(self, api_key, mongo_uri)`: Initializes the chatbot by setting up the chain and prompt.
- `generate_response(self, user_id, messages)`: Generates a response based on user input and conversation history.

Detailed Explanation of LangChain Usage

LangChain plays a pivotal role in the STARS Tutoring Chatbot by providing the infrastructure to build complex language model applications. Below is a comprehensive explanation of how LangChain is utilized for few-shot learning, embeddings, vector stores, prompt templates, and chain construction.

Few-Shot Learning with LangChain

Few-shot learning enables the language model to perform a task with only a few examples provided. By including examples similar to the user's input, the model can generate more accurate and contextually relevant responses.

Implementation in the Chatbot:

1. Examples Definition:

- A set of example interactions (input and output pairs) are defined within the code.
- **Purpose:** These examples serve as guidance for the model, illustrating how to handle specific types of user inputs.

2. Dynamic Example Selection:

- Instead of using a fixed set of examples, the chatbot selects the most relevant examples based on the user's current input using semantic similarity.

Embeddings and Semantic Similarity

Embeddings are numerical representations of text that capture semantic meaning. By converting text into embeddings, we can compute the similarity between different pieces of text.

Implementation in the Chatbot:

- **OpenAI Embeddings:** The application uses OpenAIEmbeddings to generate embeddings for the examples.

```
embeddings = OpenAIEmbeddings(api_key=os.environ["OPENAI_API_KEY"])
```

- **Vectorization of Examples:**

- The text from each example is combined (" ".join(example.values())) and converted into embeddings.
 - **Semantic Similarity Computation:**
 - By comparing the embeddings of the user's input with the embeddings of the examples, the chatbot can select the most relevant examples.
-

Vector Stores with Chroma

A vector store is a database optimized for storing and querying vector embeddings. Chroma is used as the vector store in this application.

Implementation in the Chatbot:

- **Creating the Vector Store:**

```
vectorstore = Chroma.from_texts(to_vectorize, embeddings, metadatas=examples)
```

- `to_vectorize`: List of combined text from examples.
 - `embeddings`: Embedding function from OpenAI.
 - `metadatas`: Original examples to be retrieved later.
 - **Purpose:** Stores the embeddings of examples along with their metadata, enabling efficient retrieval based on similarity.
-

Prompt Templates and Example Selection

Prompt templates define the structure of the messages sent to the language model. LangChain allows for complex prompt constructions, including dynamic insertion of selected examples.

Implementation in the Chatbot:

1. **Example Selector:**

```
class LoggingSemanticSimilarityExampleSelector(SemanticSimilarityExampleSelector):
```

```
    def select_examples(self, input_variables):
        selected_examples = super().select_examples(input_variables)
        logging.info("Selected few-shot examples:")
        return selected_examples
```

2. **Few-Shot Prompt Template:**

```
few_shot_prompt = FewShotChatMessagePromptTemplate(
```

```

input_variables=["input"],
example_selector=example_selector,
example_prompt=ChatPromptTemplate.from_messages(
    [("user", "{input}"), ("assistant", "{output}")]
),
)

```

Chain Construction and Execution

A chain in LangChain connects prompts and models to form a complete application logic.

Implementation in the Chatbot:

- **Chat Model Initialization:**

```

chat_model = ChatOpenAI(
    model=model,
    temperature=0.0,
    api_key=os.environ["OPENAI_API_KEY"]
)

```

- **Chain Creation:**

```
chain = LLMChain(llm=chat_model, prompt=final_prompt)
```

- **Invocation:**

```
response = self.chain.invoke(prompt_values)
```

End-to-End Workflow with LangChain

1. **User Input:** The user sends a message to the chatbot.
2. **Example Selection:**
 - LoggingSemanticSimilarityExampleSelector computes embeddings for the user's input.
 - Retrieves the most similar examples from the vector store.
3. **Prompt Formatting:**

- The FewShotChatMessagePromptTemplate includes the selected examples in the prompt.
- The final_prompt assembles the system message, examples, conversation history, and user input.

4. **Model Invocation:**

- The ChatOpenAI model receives the formatted prompt.
- Generates a response based on the provided context.

5. **Response Handling:**

- The assistant's response is extracted and appended to the conversation history.
- The response is displayed to the user.

Benefits of Using LangChain

- **Modularity:** LangChain allows for a clean separation of components like prompts, models, and chains.
- **Scalability:** Easily extendable to include more complex logic or additional models.
- **Ease of Use:** Simplifies the process of building applications with LLMs by abstracting common patterns.
- **Flexibility:** Supports dynamic prompt construction and example selection, enhancing the adaptability of the chatbot.

The application is organized into the following components:

components/

```
├— chatbot.py    # Base Chatbot class
├— tutorbot.py   # TutorBot implementation
├— codebot.py    # CodeBot implementation
```

```
main.py         # Main Streamlit application
```

Components

TutorBot (components/tutorbot.py)

- Implements an advanced tutoring assistant using LangChain.
- Features:

- **Custom Example Selector:** Selects relevant examples for few-shot learning.
- **Singleton Pattern:** Ensures only one instance of the chat chain is created.
- **Chat Chain Initialization:** Prepares prompt templates and initializes the language model.

CodeBot (components/codebot.py)

- Implements an educational game bot.
 - Features:
 - Custom examples and prompt templates for interactive code learning.
 - Similar structure to TutorBot.
-

Main Application (main.py)

The Streamlit application provides:

- **User Authentication:** Sign-up and sign-in functionality using MongoDB to store user credentials.
 - **Chat Interface:** Allows users to interact with the selected chatbot.
 - **Bot Selection Sidebar:** Users can choose between TutorBot and CodeBot.
 - **Session Management:** Maintains user session state, including login status and message history.
-

Packages and Dependencies

The application relies on several external packages:

Package	Purpose
streamlit	For building the web application interface.
pymongo	For MongoDB integration to store and retrieve user data.
langchain	Provides tools for chaining together prompts and GPT model interactions.
openai	Interfaces with OpenAI GPT models for response generation.
logging	Logs information and errors for debugging and monitoring.

Package	Purpose
os	Accesses environment variables.
typing	Adds type hints for better code readability and maintainability.

Class and Method Explanations

LoggingSemanticSimilarityExampleSelector

- **Inheritance:** Inherits from SemanticSimilarityExampleSelector.
 - **Purpose:** Selects the most relevant few-shot examples based on semantic similarity to the user's input and logs the selected examples.
 - **Methods:**
 - `select_examples(self, input_variables)`: Overrides the parent method to include logging of the selected examples.
-

ChatChainSingleton

- **Purpose:** Implements the Singleton design pattern to ensure only one instance of the chat chain is created.
 - **Attributes:**
 - `_instance`: Stores the singleton instance.
 - `chain`: The language model chain used for generating responses.
 - `prompt`: The final prompt template used in the chain.
 - `model`: The name of the OpenAI GPT model to use.
 - **Methods:**
 - `__new__(cls, *args, **kwargs)`: Creates a new instance if one doesn't exist.
 - `initialize_chain(cls, model: str)`: Sets up the chain with examples, vector stores, prompt templates, and the chat model.
 - `change_model(new_model: str)`: Allows changing the language model used.
-

TutorBot and CodeBot Classes

- **Inheritance:** Both inherit from the base Chatbot class.

- **Purpose:** Provide specialized implementations for the two types of chatbots, utilizing their respective ChatChainSingleton instances.
 - **Key Methods:**
 - `__init__(self, api_key, mongo_uri)`: Initializes the chatbot by setting up the chain and prompt.
 - `generate_response(self, user_id, messages)`: Generates a response based on user input and conversation history.
-

Usage Instructions

Clone the Repository

```
git clone https://github.com/your-repository/stars-tutoring-chatbot.git
```

```
cd stars-tutoring-chatbot
```

Install Dependencies

```
pip install streamlit pymongo langchain openai
```

Configure Environment Variables

Set up your OpenAI API key:

```
export OPENAI_API_KEY='your-openai-api-key'
```

Configure MongoDB

Ensure MongoDB is running locally or update the connection string in `main.py` to match your MongoDB instance.

Run the Application

```
streamlit run main.py
```

Configuration and Setup

Environment Variables

- `OPENAI_API_KEY`: Your OpenAI API key for GPT model access.

MongoDB Configuration

- Default connection: `mongodb://localhost:27017/`
- Database: `user_data`
- Collection: `users` (stores user credentials and chat history).

Security Considerations

1. API Key Security:

- Store the OpenAI API key in environment variables.
- Avoid exposing sensitive keys in logs or code repositories.

2. Password Storage:

- Currently, passwords are stored as plain text.
- Use a hashing algorithm (e.g., bcrypt) for secure password storage.

3. Input Validation:

- Sanitize and validate user inputs to prevent injection attacks.

4. HTTPS:

- Use HTTPS to encrypt data in transit if deploying the application online.

Conclusion

The **STARS Tutoring Chatbot** combines advanced natural language processing with an interactive user interface to provide an engaging and educational experience. By leveraging LangChain and OpenAI GPT models, the application delivers a personalized tutoring experience while promoting independent learning.