Name: Aaditya Raj

Registration number: 21BCI0273

Comprehensive Documentation: Real-Time Financial Assistant Chatbot with Multi-Modal Context Retrieval

1. Project Overview

Our Real-Time Financial Assistant Chatbot is a cloud-native application capable of answering user queries related to financial topics by integrating multi-modal context retrieval. The system leverages advanced language models, retrieval-augmented generation, and real-time financial data APIs to provide accurate and timely responses to user queries.

The chatbot handles multiple data modalities including:

- Text (PDFs, financial reports)
- Images (financial charts via OCR)
- Tabular data (CSV files)
- Real-time API data (stock prices, cryptocurrency rates)

2. System Architecture

2.1 High-Level Architecture

Our system comprises the following core components:

- **Multi-Modal Data Processing Pipeline**: Handles various data formats via specialized loaders
- **Vector Database**: Stores embeddings for efficient context retrieval
- **Retrieval-Augmented Generation (RAG) Module**: Retrieves relevant context and generates responses
- **Financial Data API Integration Layer**: Connects to external financial data sources
- **Streamlit Frontend**: Provides user interface for interaction

2.2 Component Interaction

The system flow begins with a user query submitted through the Streamlit interface. The query is analyzed to determine if it requires real-time financial data (e.g., stock prices) or information retrieval from stored documents. For real-time data, the system calls appropriate financial APIs. For contextual queries, the system performs RAG, retrieving relevant contexts from the vector database and generating responses using the language model.

3. Setup Instructions

3.1 Prerequisites

- Python 3.9+
- Tesseract OCR (for image processing)
- Conda environment manager (recommended)

3.2 Environment Setup

```
# Create and activate conda environment
conda create -n fin39 python=3.9
conda activate fin39

# Install required packages
pip install streamlit langchain transformers sentence-transformers chromadb pandas
pytesseract pillow yfinance PyPDF2
```

3.3 Tesseract OCR Installation

For Windows:

1. Download the installer from the UB Mannheim GitHub repository
2. Run the installer and ensure it's added to PATH

For Linux:

```
sudo apt-get install tesseract-ocr
```

3.4 Project Structure

```
financial_chatbot/
├── app.py                  # Streamlit frontend application
├── utils.py                # Utility functions for data processing
├── data_processing.py      # Text splitting and preprocessing
├── embeddings.py           # Embedding generation and vector DB setup
├── retrieval_qa.py         # RAG implementation
└── data/                   # Directory for storing example data
```

4. API Integration

4.1 Financial Data APIs

We have integrated the following financial data APIs:

4.1.1 Yahoo Finance (via yfinance)

Used for retrieving real-time stock prices and market data:

```
def get_stock_price(ticker_symbol: str) -> str:
    try:
        ticker = yf.Ticker(ticker_symbol)
        price = ticker.fast_info.last_price
        if price:
            return f"{ticker_symbol.upper()} current price: ${price:.2f}"
        else:
            return f"Price data unavailable for {ticker_symbol.upper()}."
    except Exception as e:
        return f"Error fetching data for {ticker_symbol.upper()}: {e}"
```

4.2 Multi-Modal Data Processing

4.2.1 PDF Processing

```
def extract_text_from_pdf(pdf_path):
    reader = PdfReader(pdf_path)
    text = ""
    for page in reader.pages:
        text += page.extract_text()
```

```
    return text
```

4.2.2 Image Processing with OCR

```
def extract_text_from_image(image_path):
    img = Image.open(image_path)
    text = pytesseract.image_to_string(img)
    return text
```

4.2.3 CSV Processing

```
def load_csv_data(csv_path):
    df = pd.read_csv(csv_path)
    return df.to_string(index=False)
```

5. LLM Implementation and Fine-Tuning

5.1 Model Selection

We chose the open-source `EleutherAI/gpt-neo-125M` model for our chatbot, providing a balance between performance and resource requirements.

5.2 Model Initialization

```
def initialize_llm():
    llm_pipeline = pipeline(
        "text-generation",
        model="EleutherAI/gpt-neo-125M",
        max_new_tokens=100,
        temperature=0.2,
        do_sample=True,
    )
    llm = HuggingFacePipeline(pipeline=llm_pipeline)
    return llm
```

5.3 Prompt Engineering

We designed a specialized prompt template to guide the LLM in providing accurate financial information:

```
prompt_template = """
You are a financial assistant chatbot. Answer briefly and accurately based on the
context below.

Context: {context}

Question: {question}

Answer:"""

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

6. Context Retrieval Mechanism

6.1 Text Splitting

```
def split_text(text):
    splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
    return splitter.split_text(text)
```

6.2 Embedding Generation

```
def create_vector_db(chunks):
    embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-
v2")
    vectordb = Chroma.from_texts(chunks, embeddings, persist_directory="./chroma_db")
    return vectordb
```

6.3 Retrieval-Augmented Generation (RAG)

```
def setup_retrieval_qa(vectordb):
    prompt_template = """
    You are a financial assistant chatbot. Answer briefly and accurately based on the
```

```
context below.

    Context: {context}

    Question: {question}

    Answer:"""

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

    qa_chain = RetrievalQA.from_chain_type(
        llm=llm,
        chain_type="stuff",
        retriever=vectordb.as_retriever(),
        chain_type_kwargs={"prompt": PROMPT},
        return_source_documents=True,
    )
    return qa_chain
```

7. Fallback Mechanisms

7.1 Real-Time Data Fallback

When live financial data cannot be retrieved (e.g., API limits or network issues), the system provides an appropriate error message:

```
try:
    # Attempt to get stock price
    ticker = yf.Ticker(ticker_symbol)
    price = ticker.fast_info.last_price
    return f"{ticker_symbol.upper()} current price: ${price:.2f}"
except Exception as e:
    # Fallback response with error information
    return f"Error fetching data for {ticker_symbol.upper()}: {e}"
```

7.2 No Context Fallback

When no relevant context is found in the vector database, the system provides a generic response:

```
if not source_documents:
    return "I couldn't find specific information about that in my knowledge base."
```

7.3 Error Handling in RAG Chain

```
try:
    response_dict = qa_chain({"query": user_query})
    response = response_dict.get("result", "No result found.")
except Exception as e:
    response = f"Error generating response: {str(e)}"
```

8. Testing and Validation

8.1 Unit Tests

We developed unit tests for core functionalities:

```
def test_extract_text_from_pdf():
    text = extract_text_from_pdf("sample.pdf")
    assert len(text) > 0

def test_extract_text_from_image():
    text = extract_text_from_image("sample.jpg")
    assert len(text) > 0

def test_get_stock_price():
    price = get_stock_price("AAPL")
    assert "AAPL current price" in price
```

8.2 Integration Testing

```
def test_rag_pipeline():
    vectordb = create_vector_db(["Financial data for testing"])
    qa_chain = setup_retrieval_qa(vectordb)
    response = qa_chain({"query": "What is financial data?"})
```

```
    assert response is not None
```

9. Frontend Implementation

We implemented a Streamlit-based user interface for interacting with the chatbot:

```
def app():
    st.title("📈 Real-Time Financial Assistant Chatbot")

    st.sidebar.header("Upload Multi-Modal Data")
    uploaded_pdf = st.sidebar.file_uploader("Upload PDF", type="pdf")
    uploaded_image = st.sidebar.file_uploader("Upload Image", type=["png", "jpg",
"jpeg"])
    uploaded_csv = st.sidebar.file_uploader("Upload CSV", type="csv")

    # Process uploads and setup RAG
    # ...

    user_query = st.text_input("Ask your financial query:")

    if user_query:
        # Query processing logic
        # ...

        st.session_state.history.append(("Bot", response))

    # Display conversation history
    for speaker, msg in reversed(st.session_state.history):
        st.markdown(f"**{speaker}:** {msg}")
```

10. Running the Application

To run the Streamlit application:

```
streamlit run app.py
```

Access the chatbot interface at `http://localhost:8501`.

11. Future Enhancements

1. **Advanced Model Integration**: Replace GPT-Neo with larger models for improved performance
2. **Expanded API Integrations**: Add cryptocurrency and forex APIs
3. **User Authentication**: Implement secure authentication for personalized financial advice
4. **Deployment Optimization**: Containerize the application with Docker and deploy on Kubernetes
5. **Continuous Learning**: Implement feedback-based model retraining

This documentation provides a comprehensive overview of our Real-Time Financial Assistant Chatbot with Multi-Modal Context Retrieval, covering its architecture, setup process, core functionalities, and future enhancement plans.