Internship Report: Week 3

Author: Yahya Alnwsany
Period: Internship Week 3
Company: Cellula AI
Department: NLP Engineer Internship
Supervisor: Jannah Mahmoud
Week 3 Repo: Week 3 Repo

Project Context

In Week 3, I expanded the internship project into the domain of code intelligence by building a retrieval-augmented code generation assistant. This work leverages the HumanEval dataset and state-of-the-art LLMs to create a developer tool that can generate Python code from natural language prompts, grounded in real coding examples. This continues the theme of modular, production-ready NLP systems established in previous weeks.

Executive Summary

This week, I designed and implemented **CodeGenBot**: a Streamlit-based chatbot that generates Python code from user queries using retrieval-augmented generation (RAG). The system combines semantic search over the HumanEval dataset, context retrieval, and LLM-based code synthesis. Key accomplishments:

- Implemented semantic embedding and vector search for relevant code examples
- Integrated DeepSeek LLM via HuggingFace Inference API for high-quality code generation
- Developed a conversational UI with chat history and code formatting using Streamlit
- Modularized the pipeline for extensibility and maintainability

1. Data & Problem Setup

- **Dataset:** `openai_humaneval` (Python coding problems and solutions)
- **Why this dataset?** It provides real-world coding tasks and reference solutions, making it ideal for retrieval-augmented code generation and benchmarking LLMs on practical developer tasks.
- **Preprocessing:** Extracted prompts and solutions, embedded prompts using Sentence Transformers, and stored them for fast similarity search.

2. Approach & Methodology

2.1 Retrieval-Augmented Generation (RAG)

- **Why RAG?** Combining retrieval with generation grounds the LLM's output in real, relevant examples, improving accuracy and reliability for code synthesis tasks.
- **Semantic Embedding:** Used `all-MiniLM-L6-v2` via Sentence Transformers to embed prompts for similarity search.
- **Vector Database:** Used an in-memory vector store (ChromaDB in prototyping, custom class in production) for fast, scalable retrieval of similar coding problems.

2.2 Code Generation

- **Model:** `deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B` via HuggingFace Inference API
- **Why this model?** It is a lightweight, high-quality code LLM with fast inference and strong Python support, suitable for interactive applications.
- **Prompt Engineering:** Concatenated retrieved context (problem and solution) with the user query to guide the LLM toward relevant, accurate code generation.

2.3 User Interface

- **Framework:** Streamlit
- **Why Streamlit?** Enables rapid prototyping, interactive UI, and easy deployment for ML-powered web apps.

- **Features:** Chat history, code formatting, error handling, and session state caching for a smooth user experience.

3. Implementation Details

- **Pipeline:** Modularized in `pipeline.py` with the `CodeGenPipeline` class, orchestrating embedding, retrieval, and code generation.
- **Embedding:** `embedding.py` defines the `Embedder` class for efficient batch encoding of prompts.
- **Retrieval:** `retrieval.py` manages vector search and context fetching using cosine similarity.
- **Code Generation:** `codegen.py` wraps LLM API calls and prompt construction, abstracting away API details.
- **App:** `app.py` (Streamlit UI) handles user interaction, chat logic, and code display.
- **Data:** `data/test-00000-of-00001.parquet` (HumanEval problems)

Key Code Snippet: Pipeline Usage

```
from pipeline import CodeGenPipeline

pipeline =
CodeGenPipeline("hf://datasets/openai/openai_humaneval/openai_humaneval/test-
00000-of-00001.parquet")
result = pipeline.generate_code_from_prompt("Write a function that returns
the factorial of a number")
print(result)
```

UI Example

```
# In app.py (Streamlit)
st.title("💻 CodeGenBot")
user_input = st.chat_input("Ask CodeGenBot to generate Python code...")
if user_input:
```

```
    code_output =
st.session_state.pipeline.generate_code_from_prompt(user_input)
    st.chat_message("assistant").code(code_output, language="python")
```

4. Results & Evaluation

- **Qualitative:** The bot generates correct, well-documented Python code for a variety of prompts, leveraging retrieved context to improve relevance and accuracy.
- **Quantitative:** Tested on a sample of HumanEval prompts, the system produced valid and functional code in most cases, demonstrating the effectiveness of retrieval-augmented generation.
- **User Experience:** Fast response, readable code, and an intuitive chat interface make the tool accessible for both novice and experienced developers.

5. Challenges & Solutions

- **Handling ambiguous user prompts:** Solution: Retrieve multiple similar examples and clarify intent in the UI.
- **API rate limits and latency:** Solution: Implemented error handling, retries, and user feedback for long-running requests.
- **Efficient embedding and retrieval:** Solution: Used batch processing and session caching to minimize redundant computation.

6. Next Steps

- Expand to multi-language code generation
- Support more datasets and problem types
- Improve retrieval ranking and context selection
- Deploy as a web service or integrate with IDEs

Appendix: Folder Structure

```
Week3/
├── app.py                  # Streamlit UI
├── pipeline.py             # Main pipeline logic
├── codegen.py              # LLM code generation wrapper
├── embedding.py            # Embedding utility
├── retrieval.py            # Vector search and retrieval
├── data/
│   └── test-00000-of-00001.parquet  # HumanEval dataset
├── requirements.txt        # Dependencies
├── CodeGen.ipynb           # Experiments and prototyping
```


Prepared by:

Yahya Alnwsany

Cellula AI Intern – Week 3

My Portfolio

Week 3 on GitHub