CodeRAG: A Synthetic Dataset Driven Retrieval-Augmented Code Intelligence System

CS787 Project Report

S. Abhinay Teja
Dept. of AE, IITK
sabhinayt23@iitk.ac.in

Vattikuti Susruth
Dept. of EE, IITK
vsusruth23@iitk.ac.in

Vikas Saini
Dept. of EE, IITK
svikas23@iitk.ac.in

JLS Laharii
Dept. of EE, IITK
lakshmi23@iitk.ac.in

Abstract

This project explores the effectiveness of synthetic data for training lightweight, domain-specific code embedding models inspired by the paper "Towards a Generalist Code Embedding Model Based on Massive Data Synthesis." Due to the limited computational resources of personal laptops, we generated a compact synthetic corpus covering four key tasks: text-to-code, code-to-text, code-to-code, and hybrid transformations. From each task category, ten representative samples were curated, forming a 40-example supervised dataset. Using this dataset, we fine-tuned a SentenceTransformer bi-encoder to learn joint embeddings for natural language queries and code snippets. The fine-tuned encoder was then integrated into a complete Retrieval-Augmented Generation (RAG) pipeline, where the synthetic corpus was embedded, indexed in a vector database, and used for similarity-based retrieval. At inference time, user queries were embedded using the same encoder, the top-K relevant code chunks were retrieved, and a downstream LLM generated the final response. The system was evaluated on a downstream text-to-SQL task to assess the quality of retrieval and the practical usability of embeddings trained on small-scale synthetic data. The results demonstrate that even a modest synthetic dataset can meaningfully support retrieval-driven code understanding, highlighting the potential of data synthesis for rapid, resource-constrained model development.

1 Introduction

Recent advances in large language models have significantly improved machine understanding of source code, enabling applications such as code search, summarization, repair, and automated reasoning. However, most existing code intelligence systems rely heavily on supervised datasets that are narrow in scope, limited in size, and biased toward specific programming languages or tasks. As a result, these models struggle to generalize across diverse coding styles, unseen languages, and new downstream requirements.

To address these challenges, our project explores a generalist code embedding framework inspired by the paper "Towards a Generalist Code Embedding Model Based on Massive Data Synthesis." The key idea is to move beyond manually curated datasets by generating large-scale synthetic code–query pairs, enabling the model to learn rich semantic representations without depending on expensive human annotations. By training on diverse synthetic transformations—such as renaming, code-to-query generation, structural edits, and semantic perturbations—the encoder learns to capture language-agnostic properties of code.

Our pipeline consists of three major components:

1. Synthetic Data Generation: We generate a diverse set of synthetic code–query or code–description pairs using simple rule-based transformations and LLM-assisted augmentation. Although the scale is smaller, the goal is to capture a wide variety of structural and semantic patterns in code.

2. Generalist Code Embedding Model: Using the synthesized dataset, we train a unified encoder that maps code snippets into a shared embedding space. The encoder is designed to be task-agnostic, capturing generic semantic information that can be reused for a variety of downstream tasks.

3. Adapter-Based Specialization: Instead of fine-tuning the entire encoder for each task, we attach lightweight adapters on top of the frozen base model. This allows efficient task-specific adaptation while preserving the generality and stability of the core embeddings.

By combining synthetic data with adapter-based learning, this project aims to examine whether general-purpose code representations can still be learned effectively at a student-project scale. The methodology demonstrates a practical and resource-efficient alternative to large industrial code intelligence systems, while retaining the benefits of modularity, generality, and extensibility.

2 Related Work

Research on code intelligence has advanced rapidly in recent years, especially in the areas of code embeddings, synthetic data generation, and retrieval-augmented generation (RAG). Traditional code understanding models relied on large, manually collected datasets such as CodeSearchNet, HackerRank, and open-source repositories. These datasets enabled the development of supervised and contrastive learning models for tasks like search, summarization, and translation. However, their limited diversity and manual effort requirements motivated new approaches.

A major breakthrough came from synthetic data generation, particularly demonstrated by the paper "Towards a Generalist Code Embedding Model Based on Massive Data Synthesis". That work introduced a scalable framework for automatically generating large volumes of paired code–text data across tasks such as text-to-code, code-to-text, and code-to-code transformations. Their results showed that synthetic data can significantly improve embedding quality and model generalization, even in the absence of large human-labeled datasets. Our project adopts a similar idea but at a much smaller scale demonstrating that the methodology remains useful even with limited resources.

Parallel to this, bi-encoder architectures, especially those implemented via the SentenceTransformers framework, have become widely used for encoding text and code independently into a shared embedding space. These models support efficient retrieval using vector similarity search and form the backbone of many practical search systems. Their simplicity, speed, and modularity make them ideal for experimenting with custom code embeddings and synthetic datasets.

Finally, Retrieval-Augmented Generation (RAG) has become a standard strategy for enhancing LLM performance by grounding the generation process in retrieved context. Originally developed for knowledge-intensive NLP tasks, RAG has recently been applied to code generation and code explanation systems. By retrieving semantically relevant code snippets or documentation, RAG reduces hallucinations and improves accuracy in tasks such as text-to-SQL, code completion, and debugging.

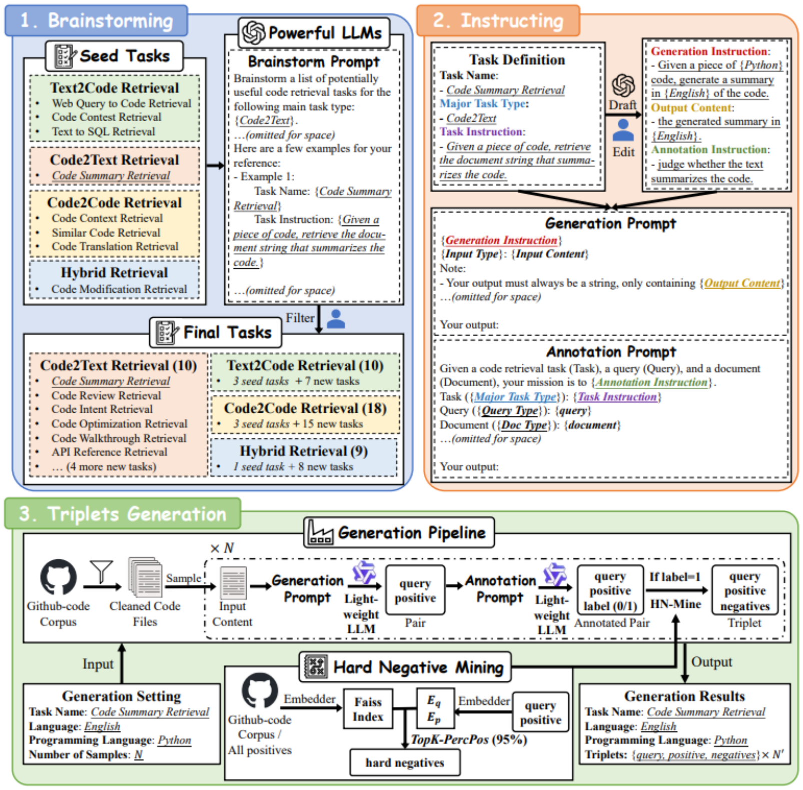
Our work is inspired by these three research directions — synthetic data generation, bi-encoder embeddings, and retrieval-augmented generation — but adapts them to a lightweight student-level project. We demonstrate that even limited synthetic data can be used to train a functioning code embedding model and integrate it into a simple RAG pipeline.

3 Data Synthesis

The creation of a high-quality dataset is a foundational step in building a Retrieval-Augmented Generation (RAG) system for code-related tasks. Due to the hardware limitations of personal computing environments, generating a large-scale dataset was not feasible. Instead, we created a small but representative synthetic dataset, referred to as the **CodeRAG Dataset**, which captures essential semantic and structural characteristics of code and its associated textual descriptions.

The dataset consists of 40 supervised query–code pairs, manually sampled from four distinct transformation types relevant to code understanding:

- **Text-to-Code:** Converts natural language descriptions into code.
- **Code-to-Text:** Converts code into summaries or explanations.
- **Code-to-Code:** Produces semantically equivalent or refactored versions of code.
- **Hybrid Transformations:** Mixes code modification with textual annotations.



1. Brainstorming
Seed Tasks
Text2Code Retrieval
Web Query to Code Retrieval
Code Contest Retrieval
Text to SQL Retrieval
Code2Text Retrieval
Code Summary Retrieval
Code2Code Retrieval
Code Context Retrieval
Similar Code Retrieval
Code Translation Retrieval
Hybrid Retrieval
Code Modification Retrieval
Powerful LLMs
Brainstorm Prompt
Brainstorm a list of potentially useful code retrieval tasks for the following main task type: {Code2Text}.
…(omitted for space)
Here are a few examples for your reference:
- Example 1:
Task Name: {Code Summary Retrieval}
Task Instruction: {Given a piece of code, retrieve the document string that summarizes the code.}
…(omitted for space)
Filter
Final Tasks
Code2Text Retrieval (10)
Code Summary Retrieval
Code Review Retrieval
Code Intent Retrieval
Code Optimization Retrieval
Code Walkthrough Retrieval
API Reference Retrieval
… (4 more new tasks)
Text2Code Retrieval (10)
3 seed tasks + 7 new tasks
Code2Code Retrieval (18)
3 seed tasks + 15 new tasks
Hybrid Retrieval (9)
1 seed task + 8 new tasks
2. Instructing
Task Definition
Task Name:
- Code Summary Retrieval
Major Task Type:
- Code2Text
Task Instruction:
- Given a piece of code, retrieve the document string that summarizes the code.
Draft
Edit
Generation Instruction:
- Given a piece of {Python} code, generate a summary in {English} of the code.
Output Content:
- the generated summary in {English}.
Annotation Instruction:
- judge whether the text summarizes the code.
Generation Prompt
{Generation Instruction}
{Input Type}: {Input Content}
Note:
- Your output must always be a string, only containing {Output Content}
…(omitted for space)
Your output:
Annotation Prompt
Given a code retrieval task (Task), a query (Query), and a document (Document), your mission is to {Annotation Instruction}.
Task ({Major Task Type}): {Task Instruction}
Query ({Query Type}): {query}
Document ({Doc Type}): {document}
…(omitted for space)
Your output:
3. Triplets Generation
Generation Pipeline
× N
Github-code Corpus
Cleaned Code Files
Sample
Input Content
Generation Prompt
Light-weight LLM
query positive
Pair
Annotation Prompt
Light-weight LLM
query positive label (0/1)
Annotated Pair
If label=1
HN-Mine
query positive negatives
Triplet
Input
Output
Generation Setting
Task Name: Code Summary Retrieval
Language: English
Programming Language: Python
Number of Samples: N
Hard Negative Mining
Github-code Corpus / All positives
Embedder
Faiss Index
E_q
E_p
Embedder
query positive
TopK-PercPos (95%)
hard negatives
Generation Results
Task Name: Code Summary Retrieval
Language: English
Programming Language: Python
Triplets: {query, positive, negatives} × N'

Figure 2: The data synthesis pipeline of **CodeR-Pile**.

To ensure quality and diversity, ten high-quality examples were sampled from each task type, producing a final dataset of 40 examples. While modest in size, this curated dataset is sufficient for fine-tuning a bi-encoder for retrieval tasks in a resource-constrained setting.

The dataset was generated using the lightweight LLM `Qwen2.5-coder-32B-instruct-q5_k_m.gguf`. Following prompting strategies inspired by prior research, the model was guided to produce meaningful query–code pairs. Each entry in the dataset is stored in CSV format with columns `query`, `pos`, and `neg`, suitable for contrastive learning.

Despite its small scale, the dataset captures diverse semantic relationships between code and text, enabling the encoder to learn generalizable embeddings for downstream tasks such as Text-to-SQL.

4 Methodology

The methodology describes the step-by-step construction of an end-to-end Retrieval-Augmented Generation (RAG) system that uses the synthetic dataset for encoder training, builds a similarity index, and integrates with a large language model to generate grounded, context-aware outputs.

4.1 Bi-Encoder Fine-Tuning

A SentenceTransformer bi-encoder is used to embed natural language queries and code snippets into a shared semantic vector space.

- **Training Dataset:** The 40-sample CodeRAG Dataset.
- **Loss Function:** Multiple Negatives Ranking Loss to pull matching pairs together and push negatives apart.
- **Base Model:** `BAAI/bge-code-large-en`,

  with fallback to `sentence-transformers/all-MiniLM-L6-v2` in low-resource environments.
- **Training Script:** train_full_encoder.py
- **Hyperparameters:** 5 epochs, batch size 16, learning rate 2×10^{-5}, max length 512.
- **Output:** A fine-tuned encoder saved in `CodeRAG_encoder/`.

Fine-tuning on a curated synthetic dataset enables the encoder to learn retrieval semantics even with limited examples.

4.2 Retrieval Corpus Indexing

A vector database is built to support fast retrieval:

- **Corpus Source:** The HuggingFace dataset `gretelai/synthetic_text_to_sql`.
- **Document Structure:**

$$\text{Document} = \text{DATABASE CONTEXT} + \text{SQL QUERY}$$

- **Embedding Generation:** The fine-tuned bi-encoder embeds every document.
- **Indexing:** A FAISS `IndexFlatIP` index is built using cosine-similarity-ready embeddings.
- **Index Files:**
  - sql_rag_faiss_index.bin
  - sql_rag_corpus_map.npy
- **Indexing Script:** vectorbase.py

These resources collectively form the searchable retrieval corpus for downstream tasks.

4.3 Retrieval-Augmented Generation (RAG) Pipeline

The complete RAG pipeline integrates the retriever with a generative model:

- **LLM:** `mistralai/Mistral-7B-Instruct-v0.2`
- **Retrieval:** A query is embedded and run through FAISS to fetch Top-K relevant documents.
- **Augmentation:** Retrieved documents are merged with the user query via a structured prompt template.
- **Generation:** The LLM generates the final SQL query using grounded context.
- **Pipeline Script:** rag_query.py

4.4 End-to-End Workflow Summary

Synthetic Dataset $\rightarrow$ Encoder Fine-Tuning $\rightarrow$ Corpus Indexing $\rightarrow$ Query Retrieval $\rightarrow$ RAG Prompt $\rightarrow$ LLM

This pipeline demonstrates that even a small synthetic dataset can power an effective RAG system, enabling context-aware generation in resource-constrained environments.

5 Conclusion

(Your text...)