Domain-Specific AI Code Search with LLMs as a Ranker

Course: COM S 402 – Senior Design

Client: Dr. Phan

Instructor: Dr. Mitra

TAs: Mohammed Rahman

Team Members SD-18:

- Carter Cutsforth
- Jacob Duba
- Keenan Jacobs
- Conor O'Shea
- Diego Perez

Domain-Specific AI Code Search with LLMs as a Ranker 1
Letter from the Client 3
Chapter 1: Requirements 4
Functional Requirements 4
Non-Functional Requirements 5
Tech Stack 5
Software architecture 6
Overview 6
Detailed breakdown of architecture 6
Chapter 2: Design (and API) 8
Design 8
APIs 10
REST API 10
Chapter 3: Work Done by Team 10
Keenan Jacobs 10
Diego Perez 10
Conor O’Shea 10
Carter Cutsforth 11
Jacob Duba 11
Chapter 4: Results Achieved 12
Appendix: Demo Slides 13

Letter from the Client

[To be attached by the TAs]

Chapter 1: Requirements

SD-18's goal with our project was to increase the performance of natural language code search, particularly in domain-specific and more niche areas where traditional models fail to provide adequate results. We implemented retrieval augmented generation to address this challenge, grounding natural language queries in relevant documentation, code snippets, and context specific data. This approach was to help enable models to produce more accurate and context-aware code search results, with emphasis on trying to increase the results in domains such as high performance computing, where specialized terminology and coding patterns are coupled with specific problems.

We developed an app that takes a user input as a natural language query and searches a code base for the most similar code segments. The innovation described by Dr. Phan's research is asking an LLM to rerank the top 10 most similar code segments after performing a vector similarity search. By using UniXcoder we were able to embed the user input and do vector comparison to get a similarity score for every code segment in the code base. Sorting this gives us a top 10 most similar code segments. We then feed the LLM the user natural language query as well as each individual code segment and ask it to give the code segment a new ranking out of 10. This new list is returned to the user, in hopes that the LLM's reranking can provide a better result than just the vector search.

To accomplish this task we were also provided with a collection of high-quality domain-specific datasets which alongside our project we were able to benchmark and determine whether there were improvements in model search accuracy. This project contributes to enhancing LLM capabilities by trying to improve the abilities of models in underrepresented technical areas. Alongside the benefits to AI this project also provided each team member with critical and experience with cutting-edge research on data curation, model development processes, and performance evaluation.

Functional Requirements

1. Natural Language Input
2. Comment-Code Embedding using UniXCoder
3. Vector Similarity Search
4. LLM-Based Ranking via OpenRouter
5. Searchable Dataset: CodeSearchNet, others
6. Code Snippet Output with confidence
7. Basic UI for user interaction

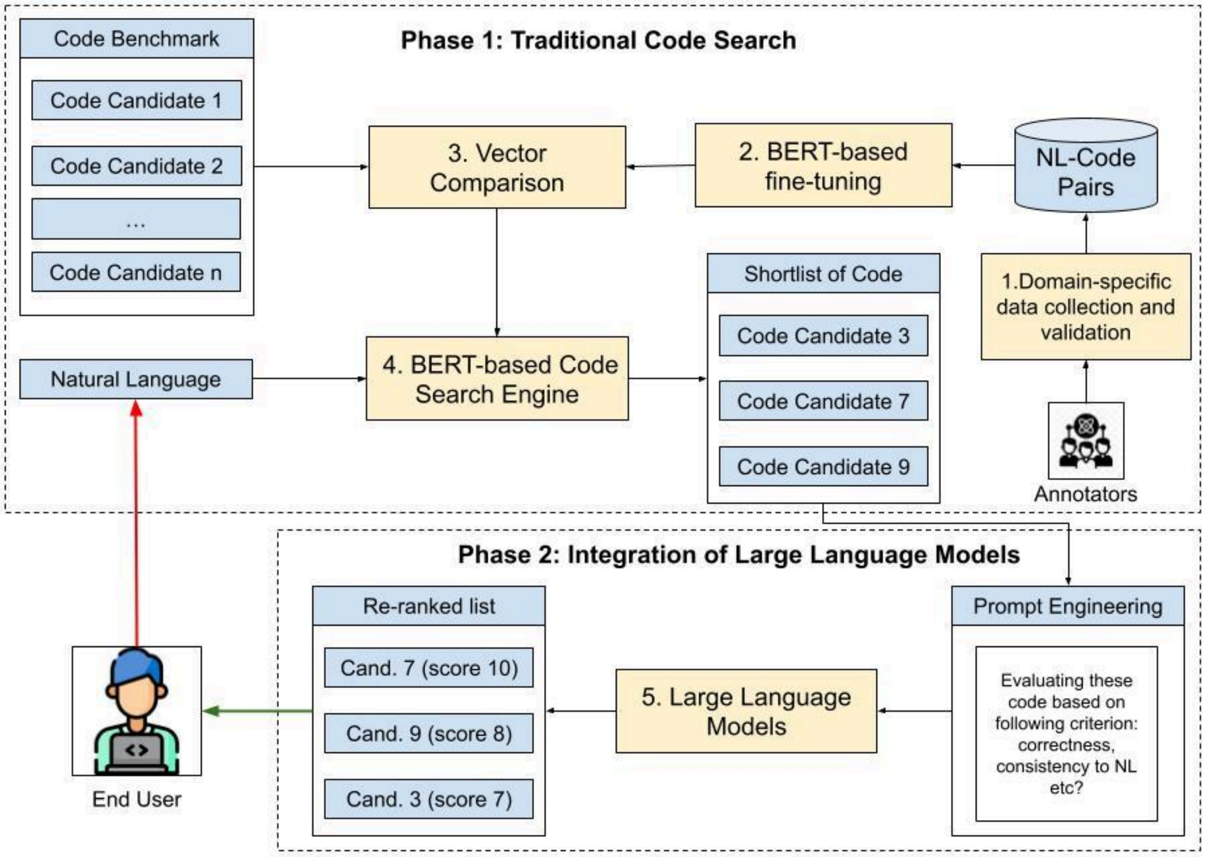
Non-Functional Requirements

1. Query response under 5 seconds.
2. Scalable to large datasets.
3. Robust to API failures (including retries or fallback behavior for LLM responses)
4. Secure internal codebases
5. Modular design
6. Intuitive UI

Tech Stack

1. Python
2. Flask
3. UniXcoder
4. Language Models
    a. DeepSeek via OpenRouter (For Demo 1)
    b. DeepCoderV2 via Ollama (For Demo 2)
5. GitLab
6. SQLite
7. Numpy
8. Pytorch

Software architecture



Overview

The architecture consists of a backend for embedding generation and vector search, a Flask-based web frontend, and an LLM-based re-ranking system using OpenRouter. Embeddings are stored in SQLite for fast access. LLM scoring is invoked via a HTTP call to a LLM. Then we display it with a Flask server.

Detailed breakdown of architecture

The program takes a user input and embeds it with UniXcoder. Then it performs a vector similarity search comparing the user's natural language query to the codebase. This will return a vector consisting of vector comparison scores. Our vector search consists of two parts: pre-processing our data and then doing code product on each vector in vector search. For processing our data, we use the Hugging Face datasets library to iterate through Code Search Net (our demo codebase of choice). Then we use UniXcoder to embed the code snippet and store the comment-code snippet pairs in a SQLite table. The embeddings are stored as SQLite blobs as they are found in numpy's memory. This allows us to reduce the amount of time of loading the embeddings into memory. Then we load the embeddings into a numpy matrix. We then multiply the matrix with the vectorized search query to find dot products of every embedding in the dataset with the embedding of the search query.

After sorting and retrieving the top 10 most similar code segments, the innovation of this research is entering Phase 2, the integration of the LLM. By giving the LLM the natural language query and a code segment, we asked the LLM to give it a new ranking score from 1 to 10. This will be performed for all 10 of the code segments from the top 10 most similar to form a new top 10 list, this time the list is the LLM's rankings. This new list will then be returned to the user. The hope is that the LLM giving its input will allow for improvements on the quality of the code segments returned when compared to just vector comparison.

We then allow user friendly access to the model with a Flask server. We give our Flask server access to the functions that call the code described in previous chapters. The Flask server takes a textbox input, runs it through the functions, and then returns a web page with the results from the functions. It is a simple architecture that has stood the test of time.

Chapter 2: Design (and API)

Design

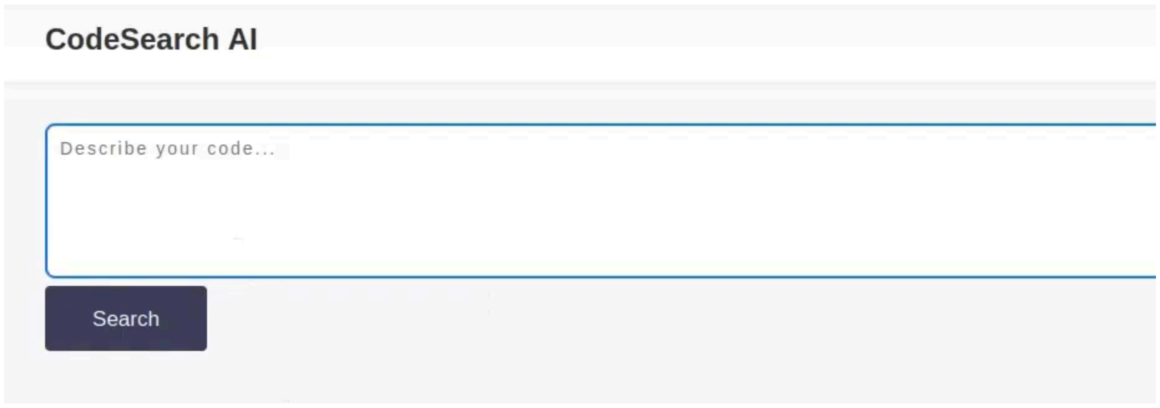
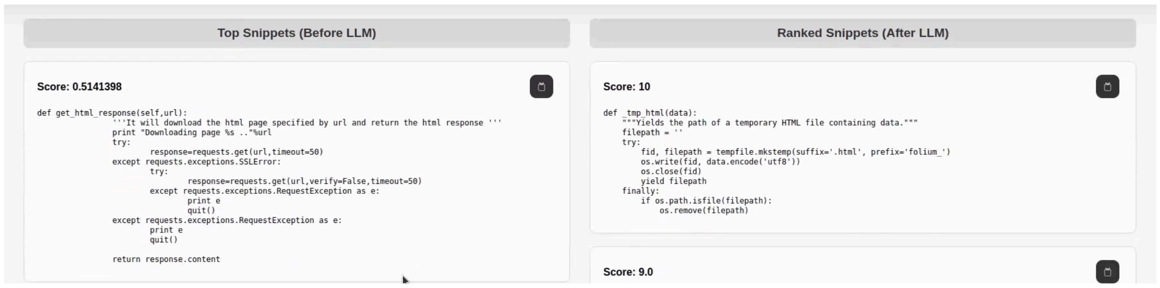


Figure 1. Asking the user for their desired code search



Score: 0.5141398

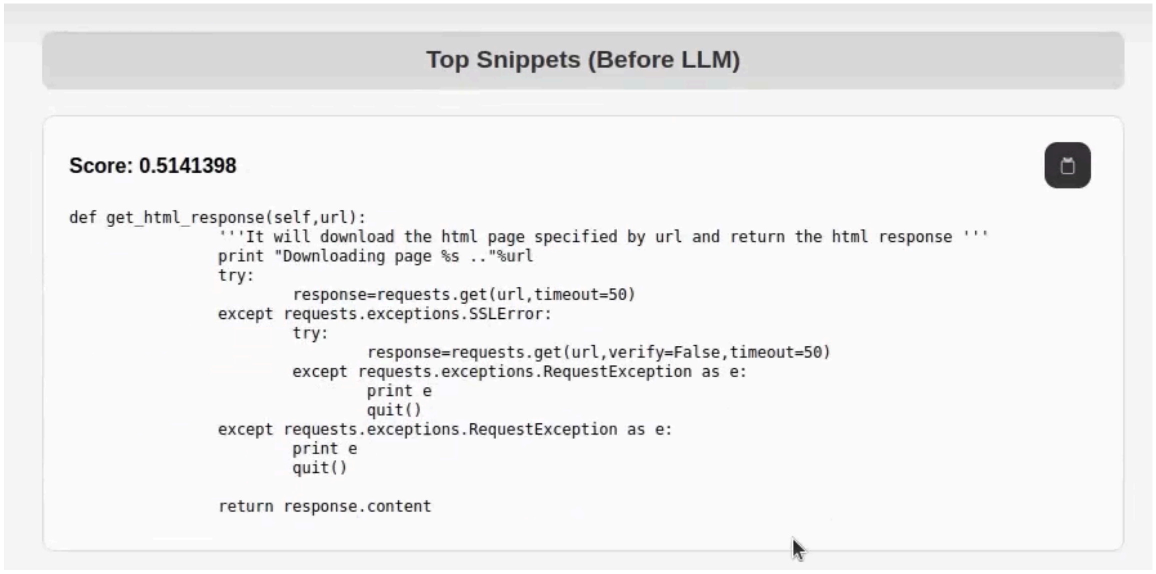
```
def get_html_response(self,url):
		'''It will download the html page specified by url and return the html response '''
		print "Downloading page %s .."%url
		try:
			response=requests.get(url,timeout=50)
		except requests.exceptions.SSLError:
			try:
				response=requests.get(url,verify=False,timeout=50)
			except requests.exceptions.RequestException as e:
				print e
				quit()
		except requests.exceptions.RequestException as e:
			print e
			quit()

		return response.content
```

Score: 10

```
def _tmp_html(data):
    """Yields the path of a temporary HTML file containing data."""
    filepath = ''
    try:
        fid, filepath = tempfile.mkstemp(suffix='.html', prefix='folium_')
        os.write(fid, data.encode('utf8'))
        os.close(fid)
        yield filepath
    finally:
        if os.path.isfile(filepath):
            os.remove(filepath)
```

Score: 9.0

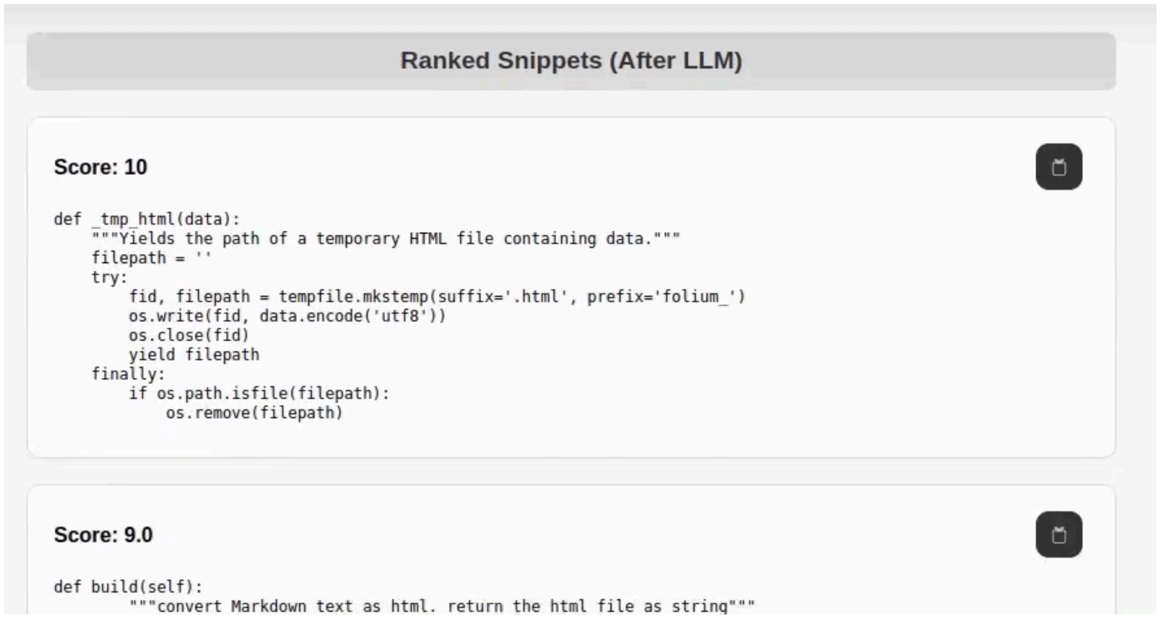
Figure 2. Example output showing a side-by-side list of the vector search results and the reranked LLM segments


Top Snippets (Before LLM)

Score: 0.5141398

```
def get_html_response(self,url):
                '''It will download the html page specified by url and return the html response '''
                print "Downloading page %s .."%url
                try:
                        response=requests.get(url,timeout=50)
                except requests.exceptions.SSLError:
                        try:
                                response=requests.get(url,verify=False,timeout=50)
                        except requests.exceptions.RequestException as e:
                                print e
                                quit()
                except requests.exceptions.RequestException as e:
                        print e
                        quit()

                return response.content
```

Figure 3. Close-up of the vector search top result with score listed


Ranked Snippets (After LLM)

Score: 10

```
def _tmp_html(data):
    """Yields the path of a temporary HTML file containing data."""
    filepath = ''
    try:
        fid, filepath = tempfile.mkstemp(suffix='.html', prefix='folium_')
        os.write(fid, data.encode('utf8'))
        os.close(fid)
        yield filepath
    finally:
        if os.path.isfile(filepath):
            os.remove(filepath)
```

Score: 9.0

```
def build(self):
        """convert Markdown text as html. return the html file as string"""
```

Figure 4. Close-up of the reranked snippets after the LLM gives them a score

APIs

REST API

GET /

- Returns web page with textbox to enter HTTP information

POST /

- Accepts a natural language query.
- Returns: Top 10 code snippets before and after LLM ranking

Chapter 3: Work Done by Team

Keenan Jacobs

Led integration of LLM-based ranking into the backend and implemented the scoring prompt. Built the entire frontend including JavaScript, CSS, and HTML. Reorganized Flask routes to support dual-column display of results. Added loader animations, copy-to-clipboard buttons, and UX polish. Also contributed to the final website, report, and presentation.

Diego Perez

Built data embedding pipeline and storage architecture. Explored storage and API options. Optimized performance by rewriting the pipeline to store and retrieve embeddings using SQLite. Implemented GPU acceleration for batch embedding generation and vector search. Worked closely with Jacob on performance tuning and system architecture for Demo 2. Prepared the system to handle new datasets as provided by the client.

Conor O’Shea

Built the core vector similarity search algorithm that searches over 450,000 Python code snippets in less than a second. In Demo 2, developed benchmarking logic critical for Dr. Phan’s research. Implement benchmarking based on the same methodology used for CosQA and AdvTest to rigorously calculate performance improvements over regular vector search with UniXcoder.

Carter Cutsforth

Attending weekly meetings with teammates and clients to discuss projects and determine work schedules. Set up the CodeSearchNet dataset and successfully deployed our team’s application and language models onto the university’s computing infrastructure, ensuring smooth integration with existing systems. Carter’s infrastructure improvements significantly enhanced performance and scalability, allowing the team to cost-effectively run a wide range of benchmarks across multiple configurations. Explored the feasibility of running the Flask server on ISU’s internal network, addressing potential deployment issues, and implemented the necessary changes to support Ollama integration. Additionally, conducted extensive testing on an assortment of large language models to evaluate performance, accuracy, and resource requirements, ultimately identifying the most suitable LLM for our specific use case.

Jacob Duba

Architectured the core technical foundation and drove technical decisions throughout the project. Established the initial codebase structure, strategically decomposed the project into trivial steps in tickets, and provided technical guidance to members when challenges arose, especially with embedding processing, storage, and search.

Chapter 4: Results Achieved

- Working demo application.
- Vector similarity search implemented using UniXcoder embeddings and dot product scoring that can search over 450,000 code snippets in a second.
    - App built to take a user input and use it as a natural language query over a codebase.
    - Embedding this user input with UniXcoder allowed us to perform vector similarity search.
- SQLite database integration for efficient, persistent storage and retrieval of code embeddings.
- Flask-based web application with a user-friendly interface for natural language code search.
- Dual-phase retrieval: initial vector search followed by LLM re-ranking implemented locally using DeepSeek models through Ollama
- Fully functional UI supporting side-by-side comparison of search results before and after LLM re-ranking.
- Copy-to-clipboard and loading animation features enhance usability and user experience.
- Benchmarking framework created to evaluate Mean Reciprocal Rank (MRR) using re-ranked results.
    - Benchmarking used to analyze CosQA and AdvTest datasets.
    - We determined through benchmarking that the LLM can perform significantly worse than just vector similar comparison.
- Local LLM inference supported using Ollama for improved speed and reduced dependency on external APIs.
- JSON-based demos deployed for validation, and performance tested across different datasets and embedding configurations, later updated to SQLite

Appendix: Demo Slides

- Demo 1: COMS_402_SD18_DEMO1
- Demo 2: COMS_402_SD18_DEMO2
- Demo 3: COMS_402_SD18_FINAL
- Also for your convenience: Demo of application

- Dr. Phan's paper
    - https://dl.acm.org/doi/10.1145/3661167.3661233