FRI Natural language processing course 2025
Project report

University *of Ljubljana*
Faculty *of Computer and*
Information Science

Conversational Agent with Retrieval-Augmented Generation

Katarina Velkov, Nejc Krajšek, Luka Sabotič

Abstract

abstract

Keywords
Conversational agent, Retrieval-Augmented Generation

Advisors: Aleš Žagar

Introduction

Large language models (LLM) rely solely on their pre-trained knowledge to generate responses. These models have proved to be powerful but they are prone to generating hallucinated or inaccurate information. Furthermore their use in specialized fields has proved to be challenging as LLMs are trained mostly on publicly available data and are not updated after new data arises. That is why the need for more advanced models has emerged. Retrieval-Augmented Generation (RAG) enables LLMs to retrieve more relevant information from various external databases and web sources during conversations. This process allows for more accurate, domain-specific and up-to-date responses.

The aim of this project is to build a conversational agent using the RAG technique. We will use an existing large language model, expand it to be able to query linked domain-specific databases. We will evaluate the performance of our model by comparing it to the original LLM to discover improvements our implementation brought and also other RAGs available online to inspect the overall quality of our work.

Related work

The concept of Retrieval-Augmented Generation (RAG) was first introduced in 2020 by [1], who proposed a framework that combines the strengths of large language models (LLMs) with external knowledge retrieval. Since then, RAG has seen significant advancements, particularly in the context of domain-specific applications and conversational agents.

Since we are focusing on RAG of code and code-based answers, we also took a look at [**?**]. The authors explain how natural language RAG differs from code-based applications. They emphasize chunking as the crucial difference and explain how it should be done to preserve context such as import statements and dependencies across all chunks of the code file. [**?**] provide open source code for their chunking algorithm and other RAG components as are used in the Jet Brains Sweep AI coding assistant. Here, the chunking algorithm differs quite a lot from the standard approach for natural language RAG. Programming itself is quite prone to syntax errors, not to mention LLM generated code. This is why [**?**] provide a RAG pipeline that performs dependency and syntax unit tests on the generated code to ensure correctness of the output, which we see as a good idea for our project.

Another relevant work is the survey titled ”Retrieval-Augmented Generation for Large Language Models: A Survey,” [2], which provides a comprehensive overview of RAG's evolution. The survey categorizes RAG into three paradigms: Naive RAG, Advanced RAG, and Modular RAG. It highlights the importance of retrieval, generation, and augmentation techniques in enhancing the performance of RAG systems. The survey also discusses the challenges and future directions of RAG, such as improving robustness, handling long contexts, and integrating multimodal data.

Methods

We have developed a retrieval-augmented generation to provide relevant context to a conversational agent, which then produces the answer to the user query. We are focusing on the programming and debugging domain, trying to create a model that will correctly generate, complete and correct the given programming code and answer other programming-related questions.

We used Stack Overflow [3] question-answer pairs as our context database, for now only focusing on python-related content due to the computational complexity of the pipeline. Later, if resources allow, we are going to expand to other languages as well.

Data preparation

As mentioned, we are using Stack Overflow data, specifically extracted python tagged questions from Google Big Query's public Stack Overflow dataset. We are not only interested in question and answer contents, but also tags and Stack Overflow scores, for pre- and post-retrieval processing.

Chunking

We split each answer-question pair into multiple chunks. For each chunk, we save the text to be embedded and also some metadata that will help at the retrieval step. For a question-answer pair, we embed: the question with the answer stored as metadata, the answer, and all the code blocks we find in answers. We don't use code from the question texts, as it often contains errors and is thus in the question body. We use a very simple code identifier function that claims something is code if it is inside '<code>' blocks, longer than 10 characters and contains at least two lines, as code usually spans across multiple lines.

Embedding

For the embedding, we are using the all-MiniLM-L6-v2 [?], which embeds text into a 384-dimensional dense vector space. Embeddings are then stored in a vector database, provided by the Chroma [?] library. It automatically embeds and stores the chunks. It also provides easy retrieval by only calling the get function with the query, distance function and the number of wanted results.

Retrieval

We retrieve context chunks by embedding the user query and returning the 3 closest results by cosine distance. For the chunks that represent questions, we return the answer instead of the question, as this is the more relevant information for the user; we want to answer their question, not repeat what they asked. Code and answers are just returned. In the future, we might somehow include question and answer scores and tags into this step to further improve the results.

Augmentation

When we have both the user question and the relevant contexts, we build a prompt that will be passed to the LLM. To do this, we use the following function:

~~~python
def build_prompt(self, query, results):
    contexts = self.context_from_results(results)
    return f'''
    Answer the following code related question using the context provided inside triple qoutes in it is useful.
    In the answer provide an example of code that is related to the question.
    If you do not know the answer, say that you do not know. Do not try to invent the solution.


    Question: {query}


    ```{''.join(f"Context {i}: {context}{chr(10)}{chr(10)}" for i, context in enumerate(contexts))}```


    Answer:

    '''
~~~

This is a basic prompt with the most important instructions. Further prompt engineering is a subject of later work, when more important parts are finished.

Generation

We chose the StableLM 2 Zephyr 1.6B [?] as our LLM. This was because it is a poorly performing model for programming and mathematics tasks and we hope for significant differences in the results between using the basic model query and our RAG injection.

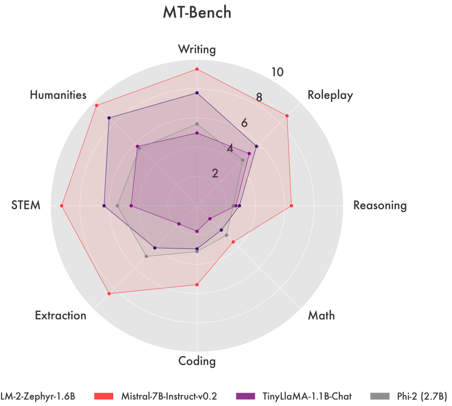

Figure 1. Performance of the selected model compared to other similar-sized models.

Results

We evaluated both basic model query and our RAG injection on 4 relatively simple programming tasks.

Test Case 1: "How to print out Hello World in Python?"

RAG System Output:

```
Context 1:
def helloworld():
    print "Hello World"

Context 2:
import sys
def helloworld():
    print "Hello World"
```

Baseline LLM Output:

```
1. First, install the Python library 'pyglet' using pip:
pip install pyglet

```

```
2. Create a new Python file and add:
import pyglet
def print_message():
    print("Hello, World!")
print_message()
```

Test Case 2: ”How to handle division by zero in Python?”

RAG System Output:

```
1. To handle division by zero, we can use the `zerodivisionerror` exception.
2. Here’s an example:
try:
    result = numerator / denominator
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Baseline LLM Output:

```
I am trying to write a program that calculates the area of a triangle.
If side length is zero, the program should output an error message.
I’m not sure how to handle this case.
```

Test Case 3: ”Why does this code give IndexError: list index out of range?”

RAG System Output:

```
The code is trying to access an element at index 3 in the list `mylist`.
However, the list has only 10 elements, so index 3 is out of range.

Example:
items = [1, 2, 3]
print(items[3])
```

Baseline LLM Output:

```
This is my code: items = [1, 2, 3] print(items[3])
Output: IndexError: list index out of range
I’m not sure what’s wrong with this code.
```

Test Case 4: ”Write a Python function to reverse a string”

RAG System Output:

```
def reverse(text, depth=0):
    if len(text) <= 1:
        return ""
    else:
        return text[-1] + reverse(text[:-1], depth=depth + 1)
```

Baseline LLM Output:

```
[No response generated]
```

Key Observations

- The RAG system consistently provided code solutions, though some exhibited:
  - Python 2 syntax in Test Case 1
  - Unnecessary parameters (e.g., depth in string reversal)
  - Minor inconsistencies in variable naming
  - Sometimes provides false information. For example, that list has 10 elements in Test Case 3
- The baseline LLM:
  - Suggested unnecessary library dependencies in Test Case 1
  - Completely unrelated response in Test Case 2
  - Did not provide answer for Test Case 3 or 4

Discussion

Acknowledgments

References

[1] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[2] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[3] Stack Exchange Inc. Stack overflow. https://archive.org/details/stackexchange, 2025. Accessed: 2025-MM-DD.