Generative AI for Risk and Reliability

Final Report

Jérémy Mathet - Clovis Piedallu

January 19, 2025

Contents

1 Introduction

This report outlines the plan to analyze the file structure, document each function, explain the prompting strategy, and describe fine-tuning techniques. The technical report includes the implementation details of the Kaggle submission, focusing on the core components and import structure. The implementation relies on key libraries such as the OpenAI API for LLM interactions, pandas for data handling, the RAG system for context retrieval, and custom agents for Python execution.

So we present the different fine-tuning techniques we used to improve the model's performance. We detail the adjustments made to hyperparameters, regularization strategies, and validation methods employed to optimize the results.

Prompt engineering is a crucial step to obtain precise and relevant responses from the model. We carefully designed the prompts to define a clear output format, allow multiple answers with comma separation, limit to a maximum of two distinct answers, include context awareness instructions, and support Python calculations when necessary. However, we will see that our prompt can surely be improved, which could lead to a better score.

Our RAG system uses a custom class to retrieve relevant documentation chunks, combine multiple chunks for broader context, and maintain the original text structure with separators. This integration allows for providing precise contextual responses using similarity search and retrieval-augmented generation techniques. We tried to use a full reliability documentation, which is actually perhaps too big.

The AI agent is responsible for executing Python code when necessary. It uses the OpenAI function calling format to enable numerical calculations, handle execution results in follow-up prompts, and adjust temperature control for more deterministic or exploratory outputs. This agent plays a key role in managing reliability engineering QCM questions.

2 Prompt Engineering

Here is our prompt:

```
You will be given a multiple choice question about reliability engineering.
Choose the correct answer.
You will be given some context followed by [Context].
Use it to reason step-by-step.
If you need any calculation, follow the steps below:

* First think step-by-step and derive an equation for the calculation
while explaining your reasonning.
* Generate a python script for the calculation. Use primarily 'reliability' or 'scipy'
python libraries. When using 'reliability', be careful on the importation of desired
classes and functions. When you need a particular Distributions
import it using from reliability.Distributions import [distrbution you need].
Always print what is relevant to answer the question at the end of the script.
For example, if the result you want is called QoI, add this line at the end of your script:
Be careful when data is provided in the question, you should use it in your script,
* Next, Use the tool "execute_python_script" to execute the python script. This tool runs
the code in a sandbox and catch the std.out. You will receive the std.out from the script
your generated that is why you need to use print in your script.
* Choose the correct answer from the given choices based on the results of your script.
If no exact match exists, choose the choice that is the closest to the calculated result.
```

```
If the calculation fails:
- DO NOT return None or invalid results
- Fall back to theoretical reasoning using the reliability documentation context
- Use approximations if necessary
- Always provide a reasoned answer choice
- Never generate a script without a print statement at the end providing the needed calculat

At the end of your response, start a new line and use the following format
to output your answer: [Answer] [The letters you choose]. For example, if you think
the answer [a] is correct, output [Answer] [a]. If you think there are multiple correct
answer, using a comma to separate them, e.g., [Answer] [a], [b].
Limit your output to 400 words maximum.
```

This prompt leverages several key techniques:

- Chain of Thought (CoT): Enforces step-by-step reasoning through explicit calculation instructions and equation derivation requirements.
- Output Structuring: Mandates specific answer format with "[Answer] [x]" syntax and comma-separated multiple choices.
- Error Recovery: Provides clear fallback instructions to use theoretical reasoning when calculations fail.
- Execution Guidelines: Details requirements for Python script generation using reliability/scipy libraries and proper print statements.
- Context Integration: Specifies how to utilize provided [Context] sections for informed decision making.

Our initial submission using only prompt engineering achieved average performance, leading us to explore additional enhancement methods.

3 RAG (Retrieval-Augmented Generation)

3.1 Cheatsheet

The RAG integration uses a custom RAG class to retrieve relevant documentation chunks, combines multiple chunks for broader context, and maintains the original text structure with separators.

Thus, our RAG class core functions include cheatsheet processing where the document is loaded from markdown, split into semantic chunks, embeddings generated and cached, and similarity search on query. The search implementation uses cosine similarity between embeddings, top-k retrieval (default k=3), context preservation with headers, and token count optimization. This implementation focuses on maintaining document structure while enabling efficient semantic search through the reliability engineering documentation.

Let's delve deeper into our functions. We used an open-source reliability library, which we extracted into a 20,000-line markdown file. We managed to get it reduces to about 12,000 lines but it is still a huge amount to deal with. For the code structure, we were inspired by the simple_rag_from_scratch file seen in class.

Our RAG (Retrieval Augmented Generation) class is a response generation system augmented by information retrieval, designed to answer questions using a markdown cheatsheet. Here is a detailed explanation of its operation, including embeddings and various parts of the code:

3.2 RAG Class

The RAG class is initialized with several parameters:

client: an instance of the OpenAI client to interact with OpenAI APIs. cheatsheet_path: the path to the markdown file containing the cheatsheet. embedding_model: the embedding model to use for generating text embeddings. max_chunk_size: the maximum size of each text chunk in characters. embeddings_path: the path to save or load the embeddings.

- Initialization

During initialization, the class loads and processes the cheatsheet by splitting it into chunks, then computes or loads the embeddings for these chunks.

- Splitting the Cheatsheet

The chunk_markdown method splits the markdown content into chunks of the specified maximum size. It detects section headers to logically structure the chunks.

- Computing Embeddings

Embeddings are vector representations of texts. The _compute_embeddings method computes embeddings for a list of texts using the specified model.

```
def _compute_embeddings(self, texts: list, show_progress: bool = True) -> list:
    if show_progress:
        return [self._get_embedding(chunk) for chunk in tqdm(texts,
        desc="Computing embeddings")]
    return [self._get_embedding(chunk) for chunk in texts]
```

The _get_embedding method uses the OpenAI client to obtain the embedding of a text.

```
def _get_embedding(self, text: str) -> list:
  response = self.client.embeddings.create(model=self.embedding_model, input=text)
  return response.data[0].embedding
```

- Loading or computing embeddings

The _load_or_compute_embeddings method loads embeddings from a file if they exist, otherwise it computes and saves them.

```
def _load_or_compute_embeddings(self) -> list:
  if os.path.exists(self.embeddings_path):
    with open(self.embeddings_path, "r") as file:
      return json.load(file)
  else:
    embeddings = self._compute_embeddings(self._chunks)
    self._save_embeddings(embeddings)
    return embeddings
```

- Saving embeddings

The _save_embeddings method saves the embeddings in a JSON file.

```
def _save_embeddings(self, embeddings: list):
  with open(self.embeddings_path, "w") as file:
    json.dump(embeddings, file)
```

- Computing similarity

The _compute_similarity method calculates the cosine similarity between two embeddings.

```
def _compute_similarity(self, query_embedding: list, chunk_embedding: list) -> float:
  return cosine_similarity(query_embedding, chunk_embedding)
```

The cosine_similarity function calculates the cosine similarity between two vectors.

```
def cosine_similarity(v1, v2):
  return np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))
```

- Retrieving relevant chunks

The get_relevant_chunks method retrieves the most relevant chunks for a given query by calculating the similarity between the query embedding and the chunk embeddings.

```
def get_relevant_chunks(self, query: str, top_k: int = 1) -> list:
  query_embedding = self._get_embedding(query)
  similarities = [self._compute_similarity(query_embedding, emb) for emb in self._chunk_embe
  top_indices = np.argsort(similarities)[-top_k:][::-1]
  return [self._chunks[i] for i in top_indices]
```

3.3 Limits of our RAG

Because of the massive size of our RAG, we got some issues with the relevance of the chunks selected regarding cosine similarity. Indeed, a lot of the reliability library documentation was dedicated to specific examples with sometimes large and detailed answers. When selecting only the most relevant chunk, it was often not useful and even sometimes inapropriate because it was encouraging at using a particular function within `reliability` library even if it wasn't suit for the particular question.

4 AI Agent

The RAG slightly improved our accuracy, but our model performs very poorly when calculations are required. As we explained in our previous report, using a Python script could prevent these hallucination issues.

4.1 Agent definition

Our agent architecture follows the same workflow as presented in the lecture, implementing function calling through the OpenAI API to execute Python code when calculations are needed. The RAG system helped improve our model's understanding of theoretical concepts, but we still faced challenges with numerical calculations where Python execution was required.

Our implementation uses a standardized tool definition:

```
tools = [{
    "type": "function",
    "function": {
        "name": "execute_python_script",
        "description": """Executes a python script in a sandboxed environment.
        Return the stdout printed by the script.""",
        "strict": True,
        "parameters": {
            "type": "object",
            "required": ["script"],
            "properties": {
                "script": {
```

```
                    "type": "string",
                    "description": "The python script to execute."
                }
            },
            "additionalProperties": False,
        },
    },
}]
```

- LLM Interaction:

It calls llm_runner with the prompt and tools to get a response from the LLM.

- Tool Execution:

If the LLM response indicates a tool call, it extracts the script and executes it using execute_python_script.

- Final Response:

It sends the execution result back to the LLM client to generate a final response.

4.2 Issues encountered

For some questions that require a script calling, our LLM started answering in a wrong format, like "1,3" or even several times the same letter, or no letter at all. That lowers our accuracy because some predictions were irrelevant. The functions described above are from the `agent_jerem.py` file, but we achieved the best scores with the `agents.py` file. However, we encountered difficulties with the agent as it sometimes generated incorrect code.

When script execution fails or returns no output, we provide the LLM with clear fallback instructions:

```
result = """Please ignore the result of the script and generate your response independently
 with a theoretical reasoning. Don't forget : at the end of your response, start a new line
and use the following format to output your answer: [Answer] [The letters you choose]"""
```

This ensures that even when numerical calculations fail, the model can fall back to theoretical reasoning using the RAG context.

However, some scripts would still fails or returns no output, we did our best to understand why but even with a parameter `temperature=0`, it could generates different scripts for a same question.

5 Conclusion

Our private score reflects that our model is correct as it does not overfit the train_test, unlike others, which may explain why we were only at 0.816 in the public score and not above. Despite the challenges, our approach demonstrates a balanced performance, indicating potential for further improvements with refined prompt engineering and better handling of edge cases in code generation. Additionally, writing a more concise cheatsheet could also lead to more relevant chunks and thus improve our accuracy.