Final Report

Jérémy Mathet - Clovis Piedallu

January 18, 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.

When performing calculations:
1. Think step-by-step and derive an equation.
2. Generate a python script using primarily the 'reliability' library.
If needed, import specific distributions using:
from reliability.Distributions import [distribution].
3. Include error handling in your script using try-except blocks.
4. Print results with print(QoI).
5. Use execute_python_script to run the code.

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
```

```
For reliability library usage:
- Reference the provided documentation
- Prefer reliability library over scipy when documentation shows clear advantages
- Double-check distribution parameter requirements

At the end, use format:
[Answer] [letter choice]
For multiple answers: [Answer] [a] , [b]

Maximum response: 400 words.
```

This prompt uses several classic prompt engineering techniques, including:

- Chain of Thought (CoT):

The prompt encourages step-by-step thinking to solve calculations. This helps structure thought and improve response accuracy ("Think step-by-step and derive an equation").

- Few-Shot Learning:

Although the prompt does not provide explicit examples, it guides the user through a series of specific steps to solve the problem. This can be considered an implicit form of few-shot learning where detailed instructions serve as models. Example: "Generate a python script using primarily the 'reliability' library."

- Error Handling:

The prompt includes instructions for handling errors, which is crucial for robust and reliable responses. Example: "Include error handling in your script using try-except blocks."

- Fallback Mechanism:

If the calculation fails, the prompt asks to revert to theoretical reasoning using the reliability library documentation. This ensures that the user always provides a reasoned answer (theorytically). Example: "Fall back to theoretical reasoning using the reliability documentation context."

- Specific Instructions:

The prompt gives precise instructions on the use of libraries and distributions, which helps guide the user towards correct and optimized solutions. Example: "Prefer reliability library over scipy when documentation shows clear advantages."

- Output Formatting:

The prompt specifies the response format, which helps standardize responses and make them easier to evaluate, which was already given.

Our first submission only with the prompt engineering was very poor (quite average, such as the benchmark), so now we tried to use the different methods seen in class.

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

We got some issues with... Dire qu'on a tenté une cheatsheet plus courte ? Problème de pertinence des chunks ?

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.

Here is how we decomposed our agent with the help of the demo_agent notebook.

4.1 llm_runner Function

This function is responsible for interacting with a LLM client to generate responses based on a system prompt and a user question.

- Input Messages:

It creates a list of messages with roles “system” and “user” containing the system prompt and the user question.

- Completion Request:

It sends these messages to the LLM client to generate a response. If tools are provided, they are included in the request.

- Return: The function returns the completion generated by the LLM client.

4.2 execute_python_script Function

This function executes a given Python script in a sandboxed environment and returns the output.

- Timeout and Executable:

It sets a timeout for the script execution and determines the Python executable to use.

- Write Script:

It writes the provided script to a file named sandbox_script.py.

- Run Script:

It runs the script using subprocess.run, capturing the output and handling any exceptions.

- Cleanup:

It deletes the script file after execution and returns the script’s output or an error message if execution fails.

4.3 kaggle_agent Function

This function uses the llm_runner to interact with the LLM and execute Python code if needed.

- System Prompt:

It defines a system prompt explaining the assistant’s capabilities.

- Tools Definition:

It defines a tool for executing Python scripts.

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

- 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.

4.4 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.

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.