Automatic Grading using LLM

Anand Kamble

Florida State University
Department of Scientific Computing

November 14, 2024

Abstract

This project explores the potential of using Large Language Models (LLMs) to automate grading in educational contexts. The focus is on leveraging retrieval-augmented generation (RAG) techniques, distributed processing, and fine-tuning approaches to improve efficiency, scalability, and consistency. Initial experiments involved setting up the Llama.cpp environment for model deployment and configuring tools like LangChain and Ollama for document retrieval and response generation. Through systematic evaluations with tools such as Phoenix, TruLens, and LangSmith, and synthetic data generation via Ragas TestsetGenerator, the project assesses model accuracy across various educational datasets. Additionally, scalability experiments highlight the use of a distributed Ollama system with load balancing across classroom machines, aiming to handle large query volumes efficiently. The findings demonstrate LLMs' ability to provide reliable, scalable grading solutions, with further recommendations for fine-tuning, user interface development, and LMS integration.

Contents

Glossary

Automated Grading System A system that evaluates student assignments and provides scores or feedback without human intervention, often using machine learning or AI technologies. 2, 8, 9, 11, 13

Chroma A vector database commonly used to store embeddings, which allows for efficient similarity search and retrieval tasks in LLM applications. 17, 18

Embedding A representation of text data as numerical vectors, enabling semantic similarity comparisons for tasks such as document retrieval and classification. 2, 17, 24, 27, 32, 33

Faithfulness A metric assessing how accurately an AI model's response reflects the input source, ensuring factual consistency in applications like automated grading. 27

FastAPI A high-performance web framework for Python, widely used to create API endpoints for serving AI models in production environments. 2, 14

Hallucination In AI, a phenomenon where the model generates inaccurate or fabricated information, which can undermine the reliability of responses in applications such as automated grading. 13

LangChain An open-source framework that simplifies the development of applications with LLMs, providing tools for prompt management, memory, and document retrieval. 2, 3, 7, 13, 14, 16, 18, 32, 33

LangSmith Tracing A tool for detailed tracking of RAG processes, enabling fine-grained analysis of retrieval and response phases, useful for debugging and optimizing LLM systems. 3, 7, 24

Llama.cpp An open-source C++ library enabling efficient inference of large language models, such as Meta's LLaMA series, on consumer-grade hardware. It supports various hardware backends, including CPUs and GPUs, facilitating local execution of advanced AI models without extensive computational resources. 2, 13, 15–18

Load Balancing A technique for distributing workloads evenly across multiple servers or machines, enhancing the system's capacity to handle large query volumes efficiently. 3, 7, 29

Natural Questions Dataset A benchmark dataset from Google containing question-answer pairs, commonly used to evaluate the performance of question-answering models. 31

Ngrok A tunneling tool that provides secure access to local servers over the internet, commonly used to expose services running on a local machine to a public URL. 2, 3, 7, 15, 16, 18, 19, 29

Ollama A tool designed to run and manage LLMs locally, offering a Docker-like interface for deploying open-source language models on personal machines. 2, 7, 13, 14, 20, 21, 23, 24, 26, 27

PerfCounterTimer A utility in Python used to measure the execution time of code, often employed in performance analysis for time-critical applications. 31

Phoenix A framework for evaluating RAG capabilities, allowing detailed assessment of document retrieval accuracy and response quality in LLM applications. 3, 7, 23, 24

RAG A hybrid AI approach combining pre-trained models with retrieval systems that fetch relevant information from external sources, enhancing the model's ability to provide accurate, context-rich answers. 2, 3, 7–9, 12, 13, 20, 23, 24, 32–34

Ragas TestsetGenerator A tool used to create synthetic test data, generating varied question and answer pairs to test and benchmark the performance of automated grading systems. 7

Round-robin Scheduling A scheduling algorithm that assigns tasks cyclically to different processors or servers, ensuring equal distribution of workload across resources. 30

Synthetic Data Generation The process of creating artificial data that mimics real-world scenarios, often used for training and evaluating machine learning models when limited real data is available. 3, 7, 23, 26

TruLens A tracing and evaluation tool that provides feedback on relevance, groundedness, and context within LLM responses, particularly useful for quality assessment in AI applications. 3, 7, 24, 25

VectorStoreIndex A data structure used in retrieval tasks, storing document embeddings to facilitate efficient similarity searches for relevant text retrieval. 23, 24, 32

WAN Access Wide Area Network access that allows a local server to be accessible over the internet, typically configured via tunneling tools like Ngrok for LLM deployment. 2, 16, 18

Chapter 1

Overview of Experiments

This document provides a comprehensive exploration of experiments conducted to evaluate the potential of Large Language Models (LLMs) in automated grading. The initial setup focused on configuring the Llama.cpp environment for reliable access via Docker and Ngrok, enabling WAN capabilities. LangChain experiments were then performed to assess document retrieval and response generation using RetrievalQA, followed by challenges in model setup and evaluation.

Further experiments involved testing available LLM tools, such as LMStudio and AnythingLLM, integrated with Ollama for efficient local model deployment. The evaluation techniques section covers methods such as Retrieval-Augmented Generation (RAG) evaluation using Phoenix, LangSmith Tracing, TruLens, and Synthetic Data Generation with Ragas TestsetGenerator. Comparative analysis of Llama 3 and 3.1 models further examined performance improvements.

Finally, distributed processing and scalability tests using a distributed Ollama system highlighted the system's Load Balancing and parallelization capabilities, while data extraction experiments with LlamaParse and LangChain facilitated Q&A pair generation, enhancing the system's grading accuracy.

Chapter 2

Introduction

2.1 Background on Large Language Models (LLMs) in Education

In recent years, the advancement of Large Language Models (LLMs) has revolutionized various sectors, including education as well. OpenAI's GPT series and Meta's LLaMA are a few of the top models that have the ability to understand and generate human-like text, making them adaptable tools for tasks ranging from natural language comprehension to content creation. This progress has opened up new opportunities to use artificial intelligence (AI) in education, particularly in automating repetitive tasks, improving learning experiences, and providing personalized feedback.

Traditionally, educational systems have relied on human graders to evaluate student assignments, a process that is often time-consuming, inconsistent, and subject to bias. With the increasing prevalence of online education, where assignments must be graded on a large scale and in real-time, the demand for Automated Grading Systems has become more evident. LLMs present a promising solution that offers scalable, consistent, and efficient methods to evaluate student work.

A key strength of LLMs in education is their ability to understand complex, open-ended responses, enabling Automated Grading Systems to go beyond simple multiple-choice or fill-in-the-blank questions. Thanks to their contextual understanding, LLMs can assess essay-based questions, programming code, and other qualitative data in ways that traditional automated systems cannot.

In addition, LLMs can provide immediate feedback to students, fostering real-time learning and improvement. This ability is especially valuable in massive open online courses (MOOCs), where immediate feedback is crucial to keep students motivated and engaged.

In this project, we examine the potential of LLMs not only to automate the grading process but also to improve the overall quality of the assessment. Using fine-tuned models and retrieval-augmented generation (RAG) techniques, our aim is to develop a system that offers accurate, reliable, and scalable classification solutions. As we delve into the technical details in the following sections, it will become clear how LLMs are reshaping the educational landscape by improving the efficiency, scalability, and quality of grading and evaluation systems.

2.2 Motivation for an LLM-Driven Grading System

Educational systems all over the world are shifting toward digital platforms, and demand for scalable and unbiased assessment methods is growing along with it. With the rise of remote learning, online courses, and more people relying on online submission systems, combined with the soaring volumes of assignments, traditional grading methods are under strain. This challenge is particularly acute in fields that require subjective evaluation, e.g., essay writing, open-ended problem solving, and critical analyses.

2.2.1 Scalability

Manual grading becomes unsustainable as the student population grows. Educators may need to grade hundreds of assignments with tight deadlines in large classes. This workflow can lead to delayed feedback on submissions. An Automated Grading System using LLMs can process large volumes of assignments quickly.

2.2.2 Consistency

Human grading is subjective and can differ from one grader to another or even for the same grader over time due to fatigue or bias. This inconsistency can result in unfair assessments. On the other hand, LLMs can have predefined standardized evaluation criteria, which will reduce the variability.

2.2.3 Cost Reduction

Educational institutions often have to spend huge amounts of budget on hiring qualified staff for grading, by automating the grading process, schools and universities can reduce operational costs associated with assessments without compromising quality.

2.2.4 Real-time Feedbacks

The integration of LLM-based agents in the grading process introduces a transformative capability: the provision of instantaneous detailed feedback on submitted assignments. Traditional grading methods inherently involve significant delays between submission and feedback, often spanning days or weeks, which can impede the learning process. An LLM-driven system overcomes this limitation by delivering immediate responses that can substantially improve the educational experience.

2.3 Objectives and Scope of the Project

The objective of this project is to design and develop an Automated Grading System powered by Large Language Models (LLM) by implementing advanced capabilities of LLMs, the system aims to assess student assignments efficiently, accurately, and consistently. The project focuses on integrating fine-tuned models and/or retrieval-augmented generation (RAG) techniques to enhance grading quality and scalability.

2.3.1 Structure of the Document

This document is organized as follows:

- **Overview of Experiments**: This section provides a summary of the experiments conducted in this project. It includes details on the initial setup and testing with `Llama.cpp`, `LangChain`, and other tools to evaluate the feasibility of automated grading using Large Language Models (LLMs).
- **Literature Review**: This section reviews the background information on existing automated grading systems, applications of LLMs, retrieval-augmented generation (RAG) techniques, and essential tools, such as `LangChain` and `Ollama`, used in this study.
- **Initial Experiments with LLMs**: Here, we describe the environment setup for `Llama.cpp`, Docker installation, and initial testing using LangChain's `RetrievalQA` feature to assess document retrieval and response accuracy.
- **Testing Available LLM Tools and Software**: This section explains the implementation and testing of `LMStudio` and `AnythingLLM` using `Ollama`, focusing on model performance in different grading scenarios.
- **Evaluation Techniques and Tools**: We outline the tools and methods, such as `Phoenix`, `LangSmith`, `TruLens`, and `Ragas TestsetGenerator`, used to evaluate the LLM grading system in terms of accuracy, relevance, and scalability.

- **Distributed Processing and Scalability**: This section discusses the setup of a distributed `Ollama` system with load balancing and parallelized query processing to handle large-scale grading tasks effectively.
- **Data Extraction and Processing**: We describe methods for parsing educational documents, generating Q&A pairs using `LangChain`, and formatting the extracted data into JSON for integration with the grading system.
- **Conclusion**: The final section summarizes the achievements, key findings, and recommendations for future work, including user interface development, and integration.

2.4 Codebase

The complete codebase for this project is available on GitHub:
`https://github.com/anand-kamble/automatic-grading-using-llm`.

Chapter 3

Literature Review

3.1 Overview of Automated Grading Systems

Automated Grading Systems have become increasingly prevalent in educational institutions and various platforms, offering efficient and accurate assessment of student work. These tools are designed to evaluate assignments, quizzes, and exams, significantly reducing the time and effort required for manual grading.

3.1.1 Gradescope

One prominent example is Turnitin Gradescope [1], which is particularly suitable for STEM courses and can grade programming assignments. The system allows educators to pre-define the expected output of the code by implementing unit testing. For example, to grade a Python code, an educator can write tests using the `pytest` package.

3.1.2 Canvas

Another comprehensive learning management system Canvas [2] also offers automated grading features. Canvas allows educators to set up quizzes and assignments that are automatically graded based on predefined criteria. The automated grading features in Canvas are flexible enough to support a variety of question formats like multiple-choice, true/false, and short-answer questions.

While Gradescope and Canvas can automatically grade factual questions, they lack the sophistication required to understand the complexity of a student's argument in an essay or to evaluate the coherence of an explanation in social sciences or humanities. Essays demand a deeper understanding of language, the ability to assess multiple valid interpretations, and the need to recognize well-constructed arguments versus superficial responses.

3.2 Large Language Models and Their Applications

Large Language Models (LLMs) are advanced artificial intelligence models designed to understand, generate, and manipulate human-like text by learning from vast amounts of linguistic data. Built upon deep learning architectures like the Transformer model, LLMs such as OpenAI's GPT series and Meta's LLaMA have significantly advanced the field of natural language processing (NLP).

LLMs have a wide range of applications across various domains:

- **Text Generation**: They can produce coherent and contextually relevant text, aiding in content creation for articles, stories, and reports.

- **Language Translation**: LLMs enable high-quality machine translation, bridging communication gaps between different languages.
- **Summarization**: They can condense lengthy documents into concise summaries, facilitating quick information retrieval.
- **Question Answering**: LLMs are capable of providing accurate answers to user queries based on the information they have been trained on.
- **Sentiment Analysis**: They analyze text to determine the sentiment or emotional tone, useful in market research and social media monitoring.
- **Chatbots and Virtual Assistants**: LLMs enhance user interaction by enabling more natural and dynamic conversations in customer service and personal assistant applications.

In the educational sector, LLMs offer transformative potential:

- **Automated Grading**: They can assess open-ended responses, essays, and complex explanations, providing consistent and objective evaluations that traditional systems struggle with.
- **Personalized Learning**: LLMs can tailor educational content to individual student needs, adapting to different learning styles and paces.
- **Content Creation**: Educators can utilize LLMs to generate exercises, quizzes, and instructional materials, saving time and enriching curriculum development.

The ability of LLMs to understand context, generate human-like text, and learn from extensive datasets makes them invaluable tools in various applications. As these models continue to evolve, their integration into educational systems holds the promise of enhancing teaching methodologies, improving student engagement, and addressing challenges in scalability and personalization.

3.3 Retrieval-Augmented Generation (RAG) Techniques

Retrieval-Augmented Generation (RAG)[3] is an innovative approach that combines parametric models (like pre-trained language models) with non-parametric memory through retrieval, enabling the model to access and incorporate external information when generating text. This dual approach addresses the limitations of relying solely on the fixed knowledge stored in a model's parameters, as it enables real-time retrieval of relevant documents and facts from large knowledge bases.

Lewis et al. (2023) demonstrated the effectiveness of RAG models in tasks like open-domain question answering and fact verification, showing that the combination of retrieval and generation leads to more accurate, factual, and contextually relevant outputs [4]. These findings have implications for the development of AI-driven grading systems, where real-time access to external content can lead to more informed and precise evaluations. Finardi et al. extended this work by experimenting with different retrieval strategies, such as dense and sparse retrievers, and highlighted the importance of chunking techniques to optimize the integration of retrieved content into the generative process. [5]

Key insights from the literature on RAG include:

- **Enhanced Precision**: By retrieving and conditioning on specific documents, RAG models outperform purely parametric models, especially in scenarios that require external knowledge
- **Diversity in Generation**: RAG has been shown to generate more diverse and contextually appropriate responses, which is crucial for grading complex assignments like essays and problem-solving tasks
- **Scalability**: The use of non-parametric memory in RAG models allows for scalable and adaptable grading systems that can accommodate a wide range of academic subjects and question types

In conclusion, the integration of RAG techniques into Automated Grading Systems can significantly enhance the model's capacity to provide accurate, scalable, and contextually grounded evaluations.

3.4 Evaluation Metrics for LLM Performance

Evaluating the performance of Large Language Models (LLMs) involves a mix of traditional NLP metrics and task-specific measures. The following metrics are commonly used:

- **BLEU (Bilingual Evaluation Understudy)**: Originally developed for machine translation, BLEU is now used to evaluate text generation by comparing the model's output to reference texts2.
- **ROUGE (Recall-Oriented Understudy for Gisting Evaluation)**: This metric is particularly useful for assessing summarization tasks.
- **Relevancy**: Answer relevancy evaluates how well the LLM's response addresses the given input in an informative and concise manner. This metric helps gauge the model's understanding of the query and its ability to generate appropriate responses.
- **Hallucination**: This metric identifies whether an LLM's output contains fake or made-up information. Minimizing Hallucinations is critical for maintaining the trustworthiness of the model.

To evaluate the LLMs for metric mentioned above we can use frameworks like `deepeval` which also provides integration with llama-index.

3.5 Existing Tools and Frameworks

3.5.1 LangChain

LangChain is an open-source framework designed to simplify the development of applications powered by large language models (LLMs). It provides a standardized interface for combining LLMs with other sources of computation or knowledge, enabling developers to create complex chains of operations. LangChain's architecture includes essential components like prompts management, memory systems for maintaining conversation context, and agents that can make decisions and use tools. The framework supports multiple LLM providers including OpenAI, Huggingface, and open-source models, making it a versatile choice for building applications ranging from chatbots to document analysis systems.

3.5.2 Llama.cpp

Llama.cpp is a groundbreaking C++ implementation for running Meta's Llama models efficiently on consumer-grade hardware. Originally created by Georgi Gerganov, this project has become fundamental in the democratization of AI by enabling local execution of large language models without requiring expensive GPU infrastructure. The framework employs various optimization techniques, including 4-bit quantization and efficient CPU inference, allowing users to run powerful language models on personal computers with reasonable performance. Its success has led to widespread adoption in the open-source community and has inspired numerous derivative projects focused on local AI deployment.

3.5.3 Ollama

Ollama is a modern framework that simplifies the local deployment and management of large language models, particularly focused on making open-source models accessible to developers. It provides a Docker-like experience for running LLMs, with simple commands to pull, run, and manage different models. The framework includes built-in support for popular open-source models like Llama 2, Mistral, and their variants, while offering optimized performance through efficient model loading and execution. Ollama's user-friendly approach and emphasis on local deployment have made it increasingly popular among developers who need

to run AI models without cloud dependencies.

In addition to its ease of setup, Ollama supports API calls, enabling seamless integration with other applications and workflows. This API capability allows developers to interact with models programmatically, making it possible to automate tasks and connect Ollama-powered models to larger systems without requiring cloud dependencies. Ollama's user-friendly approach and versatility have made it increasingly popular among developers seeking a robust, locally deployed AI solution.

3.5.4 FastAPI

FastAPI is a modern, fast (high-performance) web framework for building APIs with Python based on standard Python-type hints. While not specifically an AI framework, it has become a crucial tool in deploying machine learning and AI models in production environments. The framework offers automatic API documentation, asynchronous request handling, and data validation out of the box, making it particularly well-suited for creating robust API endpoints for AI services. Its compatibility with popular machine learning libraries and excellent performance characteristics have made it a preferred choice for serving AI models in production, especially when combined with tools like LangChain for building comprehensive AI applications.

Chapter 4

Initial Experiments with LLMs

This chapter outlines the setup and the experiments conducted with large language models (LLMs). The goal was to set up a reliable environment for hosting `Llama.cpp`, which can be accessed over the internet (WAN).

4.1 Setting Up the Llama.cpp Environment

4.1.1 Installing Llama.cpp via Docker

To set up a reliable and replicable environment using docker is preferred [6]. For building a docker image, we need to define a `Dockerfile` which includes all the instructions necessary for installing the dependencies and the execution of required packages or programs. The `Dockerfile` (see B) used in this project performs the following key tasks:

- **Base Image**: Specifies the base image to build the docker container using the keyword `FROM`. In our case, it is `ubuntu:latest` which means the use of the latest available image of ubuntu.

  ```
  FROM ubuntu:latest
  ```

- **Copying Files**: Copy required files from the local machine into the container. This is done by using the keyword `COPY` followed by path of the file relative to `Dockerfile` in local machine and then the destination path inside the container.

  ```
  COPY <source> <destination>
  ```

- **Update Permission**: The copied file, which is a `Llama.cpp` executable binary of a specific model. This file is made executable by using the `chmod` command. Since this is a Unix operating system command it needs to be defined with keyword `RUN`

  ```
  RUN chmod +x /llava-v1.5-7b-q4.llamafile
  ```

- **Install Ngrok**: Install curl in the system and then add the `Ngrok` repository to the system's trusted sources. Then update the package list again to include `Ngrok`, and finally install the `Ngrok` package.

To start the container which was created using the `Dockerfile` we have used Docker Compose, which is a tool for defining and running multiple containers. In this case, even though we have only a single container, docker-compose is used to make the process of starting the containers easier. Since docker-compose allows us to specify all the variables in a `yaml` file named `docker-compose.yaml` it eliminates the need to use complex commands in the terminal specifying each variable.

In this configuration:

- The service is defined to use the build context and a specified image name for the container.

- The container is set to start with `"/bin/sh"` as the entry point, executed in an interactive mode. The command runs the Llama.cpp binary in server mode and launches Ngrok to expose port 8080.
- GPU resources are reserved, ensuring that the container uses the NVIDIA driver for GPU acceleration, as specified in the deployment section.

4.1.2 Configuring WAN Access with Ngrok

To enable access to the Llama.cpp server from a public endpoint, Ngrok is used which creates a tunnel from the public endpoint to the locally running server (in this case the container). It is a reverse proxy and ensures that the server is accessible from the same public domain independent of the hardware it is running on. This also improves the security of the system as there is no need to expose the local network.

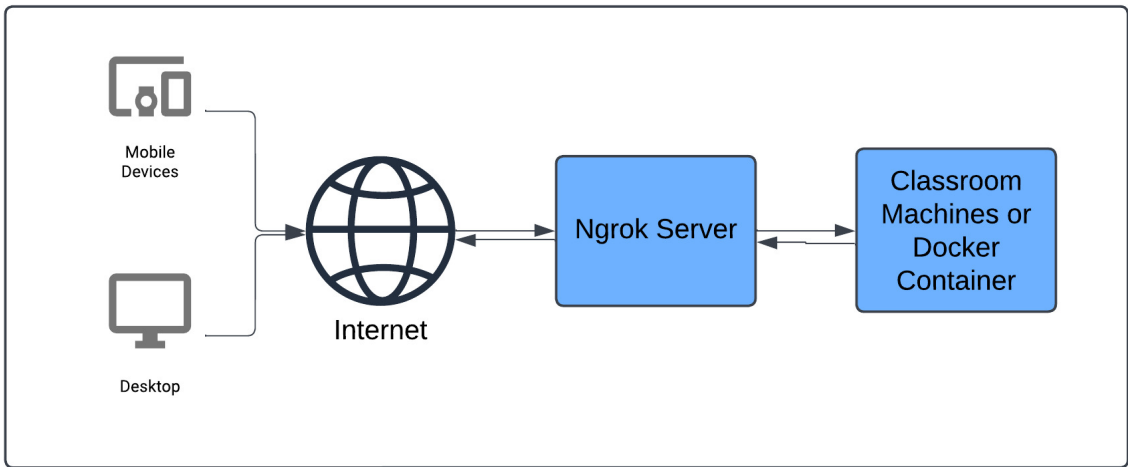

Figure 4.1: NGrok Reverse Proxy

After installation of the `Ngrok` package, it needs to be configured by creating a configuration directory which will store the authentication token. Once the token is configured, the Llama.cpp server is started in server mode, and Ngrok exposes port 8080 using a provided custom domain. This makes the server accessible remotely.

All of these steps are done automatically by the bash script `run_server.sh` (see D)

4.2 LangChain Experiments Using RetrievalQA

In this experiment, we utilized LangChain's `RetrievalQA` chain to assess the performance of the Llama.cpp model in providing responses to domain-specific questions based on embedded documents. This experiment aimed to determine the accuracy of the model's retrieval and response generation when querying technical documents related to data mining. Below, we describe the setup and execution of the experiment. (For code see E)

4.2.1 Setup and Initialization

The experiment begins by importing the necessary libraries from LangChain, setting up the Llama.cpp model, and loading the PDF documents that contain the course material. The Llama.cpp model was initialized with the following configuration:

- **Model Path**: The path used was `Models.MISTRAL.value`, which refers to the pre-trained Mistral model `mistral-7b-instruct-v0.1.Q3_K_L.gguf`.

- **Context Size**: A context window size of 2048 tokens was specified for the model, allowing it to process large chunks of text at once.
- **Precision**: We employed the `f16_kv` setting to optimize memory usage while maintaining model performance.

Once the model was initialized, we utilized the `PyPDFLoader` to load and split the content of the provided PDF documents into manageable chunks.

4.2.2 Embedding Documents

After loading the documents, they were processed by the Llama.cpp embeddings module, using the pre-trained embeddings model:

- **Seed**: A fixed seed of 100 was set for reproducibility.
- **Model Path**: Similar to the LLM configuration, the same pre-trained Mistral model was used for embedding the document chunks.

The resulting embeddings were stored in a Chroma vector store, which acted as the retriever for the next step of the experiment.

4.2.3 Prompt and Chain Setup

To query the model and retrieve relevant chunks, a custom prompt template was designed:

```
You are a professor of graduate level course data mining.
The question asked is: `{question}`
rate the following answer on a scale of 1 to 10, where 1 is the worst and 10 is the
↪  best:
`{answer}`
Give answer in format: Rating = x/10
```

Using this prompt template the `LLMChain` was constructed, followed by the creation of a `RetrievalQA` chain that utilized the Chroma retriever. This setup allowed the model to both retrieve relevant sections of the course material and provide a response based on the input question and answer.

4.2.4 Running the Experiment

The experiment was designed to simulate an automated grading system by evaluating student answers. For example, a sample question and answer pair was tested as follows:

```
Question: Is K-means a clustering method?
Answer: K-means is not a clustering method.
```

The `RetrievalQA` chain retrieved relevant portions of the course material, evaluated the answer using the model, and returned a rating. The result of the model's evaluation was:

```
Rating = 3/10
```

4.3 Challenges in Initial Setup and Experiments

During the initial setup and experiments with Llama.cpp and LangChain, several challenges were encountered that impacted the overall workflow:

- **Hardware Limitations**: The experiments required significant computational resources, particularly for handling the large model sizes and performing inference on longer texts. Systems without sufficient GPU memory faced performance bottlenecks.
- **Docker Configuration**: While Docker offered a reliable environment, setting up GPU support for Llama.cpp inside the Docker container required careful configuration of the NVIDIA drivers. Ensuring compatibility across different hardware setups was another hurdle. This was especially harder on the classroom machines as those required loading the CUDA module.
- **WAN Access**: Configuring WAN Access directly on the classroom machines network presented some security concerns and was not allowed by the university. Hence I had to search for another methods like reverse proxy.
- **Execution Time**: Running RetrievalQA with the Llama.cpp model was computationally expensive, with each query taking up to 112 seconds on classroom machines. This limited the throughput of the experiments and posed challenges for scaling the system.

Despite these challenges, the experiments provided valuable insights into model performance and areas for future optimization.

4.4 Pros and Cons

While the initial experiments provided promising insights, they also highlighted areas that could benefit from further refinement. Below, the pros and cons of this approach are outlined.

4.4.1 Pros

- **Controlled Environment**: Utilizing Docker for containerization ensured a reliable, consistent environment across various machines, which made experimentation more manageable.
- **WAN Accessibility**: The integration of Ngrok enabled access to the grading system remotely, which is essential for testing performance under varied network conditions, simulating different setups and remote access scenarios.
- **Effective Retrieval System**: Combining LangChain's RetrievalQA with Chroma vector store provided quick and relevant access to document sections, which improved grading accuracy and the system's responsiveness in evaluating answers against course materials.
- **Automated Evaluation Potential**: Initial trials showed the model's potential to assess student responses, laying the groundwork for developing more advanced automated grading that can handle open-ended responses effectively.

4.4.2 Cons

- **High Computational Demand**: The model required substantial computational resources, particularly for tasks involving long text passages, which limited its accessibility on lower-end hardware.
- **Setup Complexity**: The environment setup, including GPU support in Docker which depends on OS drivers along with WAN Accessibility, required complex technical setup and time, which could be challenging for wider adoption in educational settings.
- **Response Time**: The inference time per query was relatively long, making it challenging to scale the experiments for real-time applications or high-throughput requirements.

- **Reliance on Network Stability**: Using Ngrok for WAN access introduced a dependency on network stability, and in some cases, university network restrictions posed additional limitations on accessible configurations.

Chapter 5

Testing Available LLM tools and software

5.1 Implementing LMStudio with Ollama

LMStudio [7] is a versatile tool designed to seamlessly integrate and manage language model (LM) workflows. This tool allows us to quickly and easily experiment with different LLM models and RAG and it uses Ollama as a backend to run all the LLM models, so it supports all the models that are supported by Ollama. This tool also has an option to customize the prompt template, making it easier to experiment and evaluate a model's performance based on different documents, contexts, and prompt templates. This avoids building a new pipeline with `langchain` or `llama-index` for each new model we want to test. We can have all the settings saved per model, so we can easily switch between different models during testing.

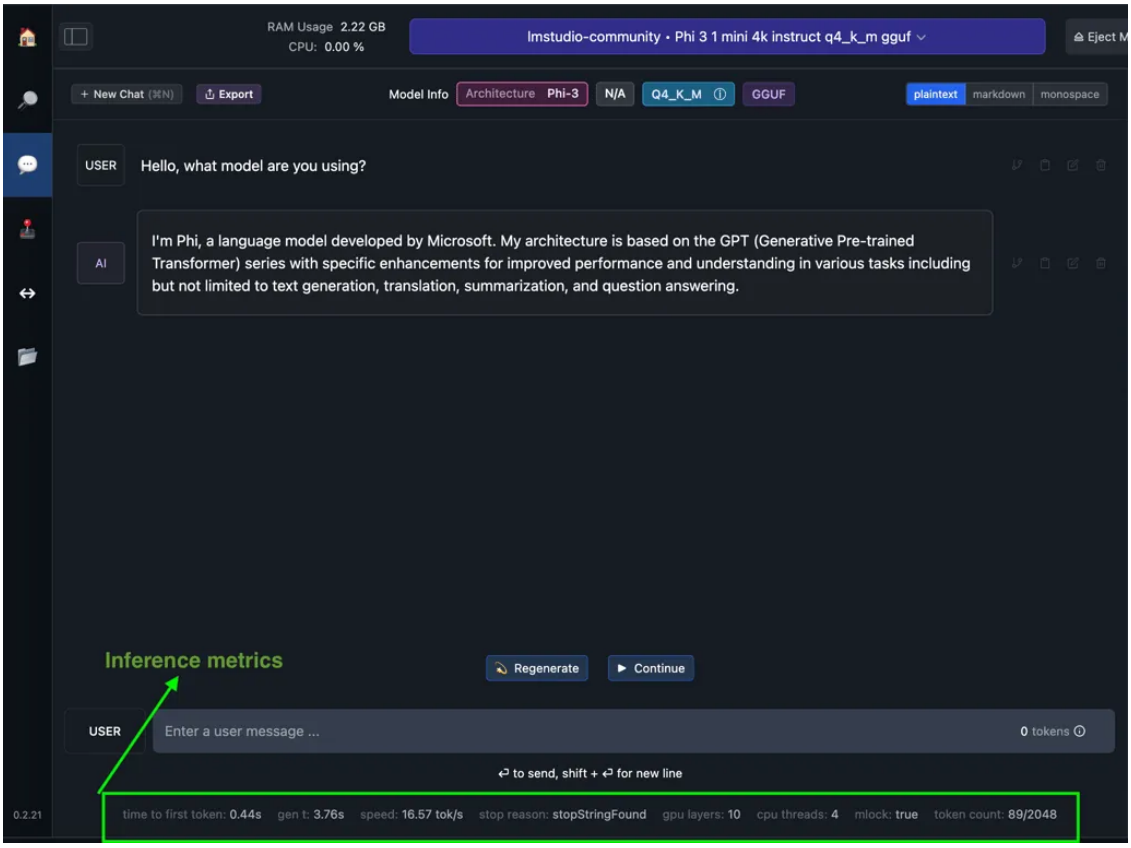

Figure 5.1: LM Studio User Interface

5.2 Implementing AnythingLLM with Ollama

AnythingLLM [8] is an AI application designed for local execution and interaction with large language models (LLMs)1. When combined with Ollama, it provides a powerful and flexible environment for working with various LLMs locally. This is similar to LMStudio 5.1 but offers more features that can be useful for organizations such as chat logs, multiple users and API access.

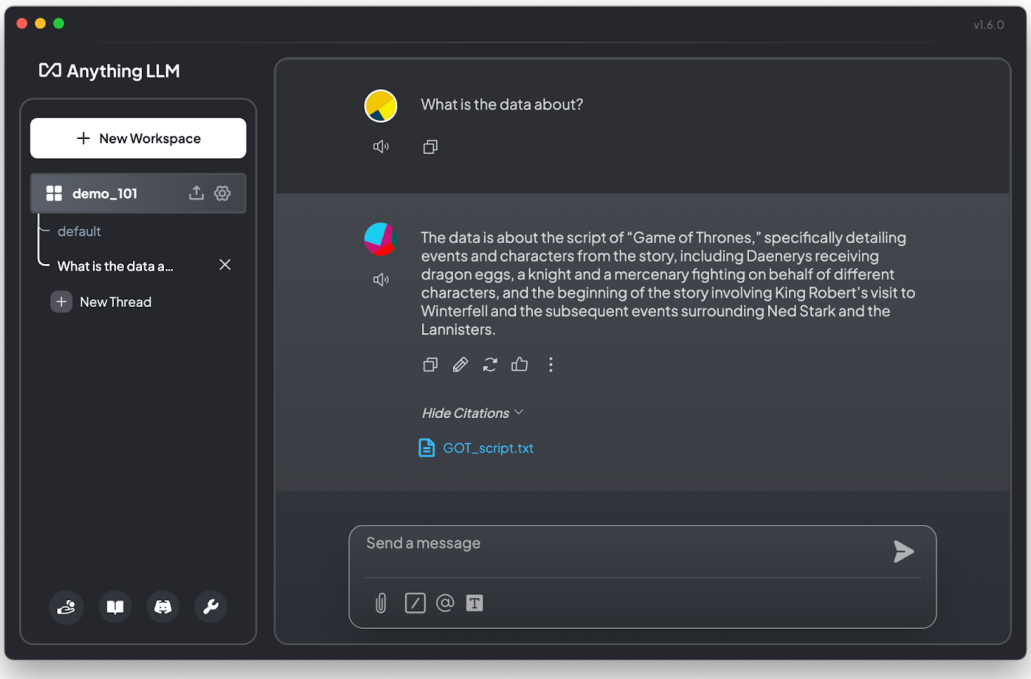

Figure 5.2: Anything LLM User Interface

Chapter 6

Evaluation Techniques and Tools

Effective evaluation is essential in the development of our LLM-driven grading system, ensuring accuracy, reliability, and scalability. This chapter presents the evaluation techniques and tools utilized to assess system performance, focusing on Retrieval-Augmented Generation (RAG) evaluation, tracing, Synthetic Data Generation, and comparative analysis across different LLM models.

6.1 RAG Evaluation Using Phoenix

To evaluate the Retrieval-Augmented Generation (RAG) capabilities of our grading system, we leveraged the Phoenix framework. Phoenix provides a comprehensive suite of tools to assess document retrieval accuracy and response quality, offering detailed insights into model performance across various queries.

In our setup, we employed Ollama's phi3 model for both document embedding and response generation, integrated within the Phoenix environment. Below is a code snippet showcasing the embedding model setup:

```
from llama_index.embeddings.ollama import OllamaEmbedding
from llama_index.llms.ollama import Ollama

# Embedding model setup
embedding_model = OllamaEmbedding(
    model_name="phi3:latest", base_url="http://localhost:11434"
)

# LLM setup
llm = Ollama(model="phi3:latest", temperature=0.7, base_url="http://localhost:11434")

```

Now we can create the `VectorStoreIndex` and query the LLM, but to trace this query using Phoenix we will have to launch the Phoenix app before querying. This has to be done inside an `async` function as launching the Phoenix app is an asynchronous process.

```
import phoenix as px

async def evaluate():
    px.launch_app()
    index = VectorStoreIndex.from_documents(documents)

    qe = index.as_query_engine(llm=llm)
    response1 = qe.query("what is k-means")
```

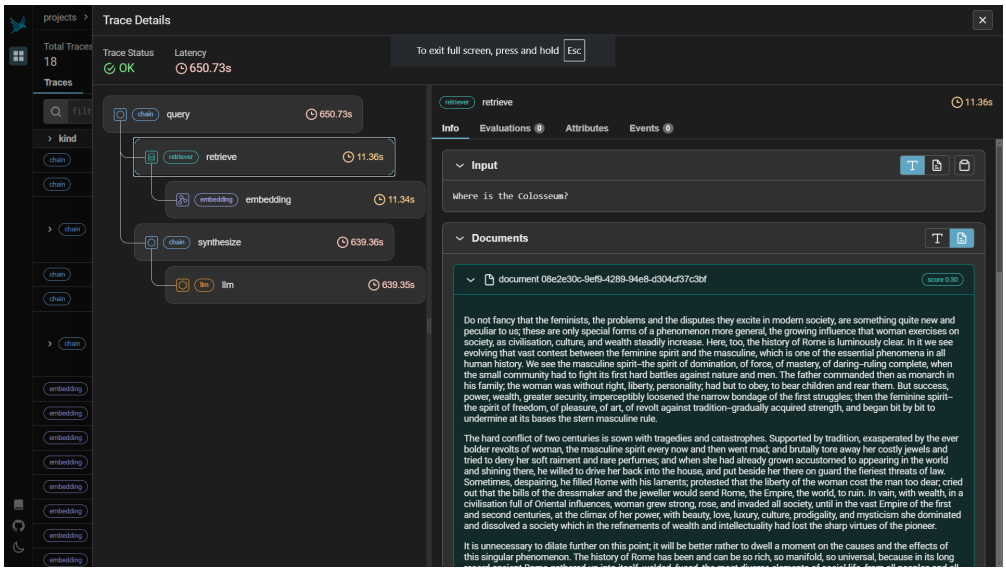

Figure 6.1: Phoenix Dashboard

After we have completed the query processing we will get detailed information about the query such as retrieved documents on the Phoenix dashboard as shown in Figure 6.1.

Detailed code for this setup is available in Appendix G.

6.2 LangSmith Tracing for RAG Evaluation

LangSmith Tracing is another tool that can track each RAG evaluation phase, allowing in-depth inspection of retrieval and response steps. This tracing aids in monitoring the accuracy of specific retrievals, response coherence, and system performance, particularly for debugging and improving model responses.

You can check one of the runs on langsmith online dashboard by following the link:
`https://smith.langchain.com/public/8e4d387e-770f-4a27-8f08-2884b58b66b7/r?runtab=0`

The implementation code for LangSmith Tracing can be found in Appendix H.

6.3 Tracing with TruLens

TruLens allows us to measure the quality and effectiveness using feedback functions on our LLM provided, in this experiment it is the `LiteLLM` object.

In this experiment, we are using the `phi3:latest` model along with Ollama Embedding created using the same base model. We are encoding the "chap7_basic_cluster_analysis_98p%20.pdf" in the `VectorStoreIndex`. After this, we create a provider object which is the instance of `LiteLLM` with the `phi3` model and endpoint directed to the local Ollama server. This provider object has the feedback function added and it checks for `f_groundedness`, `f_answer_relevance`, and `f_context_relevance`. (See Appendix I)

After querying, we can see the results along with timings as shown in the Figures 6.2 and 6.3

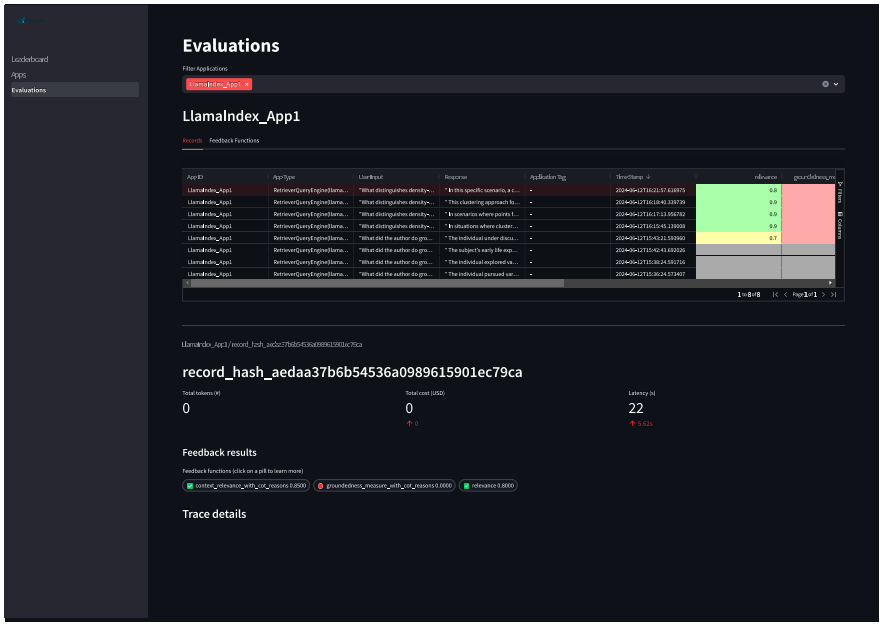

Figure 6.2: TruLens dashboard - relevance & groundedness score

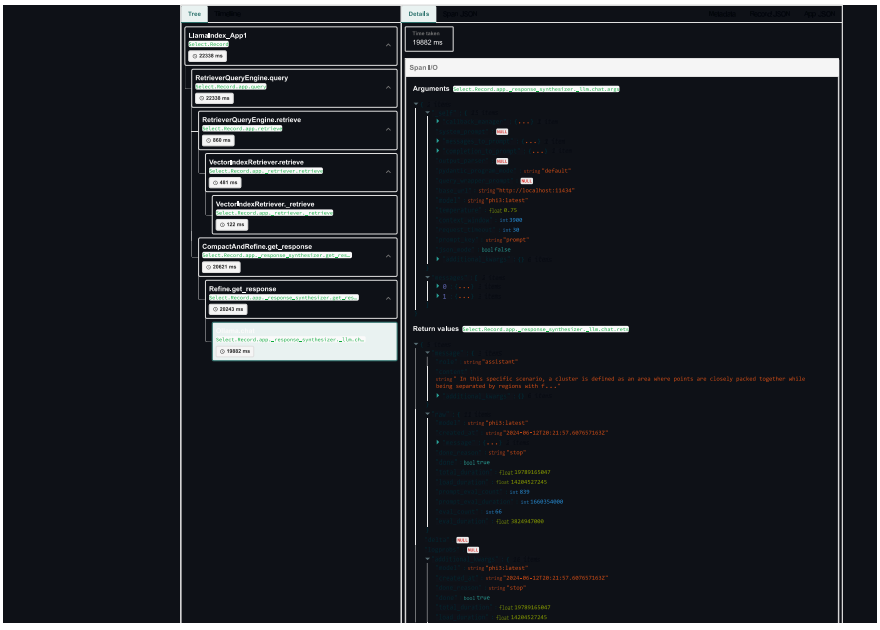

Figure 6.3: TruLens dashboard - Timings

6.4 Synthetic Data Generation for Comprehensive Testing

Given the limitations of real-world datasets, Synthetic Data Generation was implemented to simulate varied student submissions. This step enabled the evaluation of the grading model across a spectrum of complexities and response styles.

6.4.1 Using Ragas TestsetGenerator for Synthetic Data

To effectively evaluate the grading system across diverse scenarios, we implemented Synthetic Data Generation using the Ragas TestsetGenerator. This tool allows for the creation of realistic, diverse test data that reflects common student response patterns and edge cases. The TestsetGenerator relies on a set of documents loaded from a specified directory, with each document processed to create synthetic responses that simulate real-world grading scenarios.

The TestsetGenerator is designed to vary the distribution of response types within the dataset. For example, responses are generated with distributions tailored to include a mix of simple, reasoning-based, and multi-context questions. This blend enables the grading model to encounter typical responses, as well as more complex cases that might involve partial correctness or ambiguity. By tuning these parameters, the test data reflects a broad range of response scenarios, making the grading system's evaluation comprehensive.

Once generated, the synthetic dataset is saved in CSV format, ensuring ease of use for subsequent analysis and evaluation. This file format facilitates loading and querying the data during the evaluation process, allowing for efficient interaction with the grading system. The synthetic test set acts as a benchmark for understanding the grading model's adaptability and robustness under varied response patterns. For full implementation details, see Appendix J, which includes the configuration and generation process.

By leveraging Ragas TestsetGenerator, we ensure that the grading model undergoes extensive testing, simulating both typical student responses and complex edge cases. This approach enhances the model's resilience and prepares it for practical deployment in real-world educational settings.

6.4.2 Generating Diverse Test Datasets

The generated datasets simulate grading scenarios, preparing the model for a range of response qualities and complexities. These datasets are essential for developing a model that is resilient to varying answer accuracies and unique response styles.

6.5 Comparative Evaluation of Llama 3 and Llama 3.1

The comparative evaluation of Llama 3 and Llama 3.1 aimed to determine the optimal model for our grading system. By leveraging the Ragas evaluation framework in conjunction with Llama Index and Ollama embeddings, we systematically compared these models on key performance metrics to ensure robust, accurate grading capabilities. This section describes the setup, configuration, and results of this comprehensive evaluation.

6.5.1 Evaluation Setup

To conduct the experiment setup was prepared with the following prerequisites:

- **Environment Setup**: Python 3.10 environment with essential packages such as ‘ragas‘, ‘llama-index‘, ‘ollama‘, ‘datasets‘, and ‘dotenv‘.
- **Installation and Configuration**: The project repository was cloned and dependencies were installed using the:

```
git clone https://github.com/anand-kamble/eval-comparison/
pip install -r requirements.txt
```

Each model was evaluated using the different datasets from *llama-datasets*, configured to employ Ollama's embedding with both of the `llama3` and `llama3.1` models.

6.5.2 Evaluation Methodology and Metrics

The evaluation followed a structured approach:

1. **Embedding Configuration**: Ollama embeddings were initialized to embed the query model using 'OllamaEmbedding' as shown below:

```
embeddings = OllamaEmbedding(model_name=QUERY_MODEL, base_url="http://class02:11434")
Settings.embed_model = embeddings
```

2. **Document Loading and Vector Indexing**: Documents from the specified dataset directory (supporting '.pdf' and '.txt' files) were loaded, followed by vector index building for efficient retrieval during evaluation.

3. **Query Engine Setup**: A query engine was initialized for each model configuration, facilitating the querying of generated test questions.

4. **Metric-Based Evaluation**: The models were compared on a range of metrics provided by Ragas, including:

   - **Faithfulness**: Measures how accurately answers reflect the source content.
   - **Answer Relevancy**: Evaluates the relevance of answers to the specific queries.
   - **Context Precision**: Assesses how precisely context is captured in responses.
   - **Context Recall**: Measures the comprehensiveness of context in answers.
   - **Harmfulness**: Evaluates the content's potential harmfulness or appropriateness.

6.5.3 Results and Analysis

The evaluation generated results for each model on different datasets (e.g., *History of Alexnet Dataset*, *Paul Graham Essay Dataset*, *Llm Survey Paper Dataset*, and *Mini Truthful QA Dataset*). Figure 6.4 illustrates the performance of Llama 3 and Llama 3.1 across the various metrics.

- **Llama 3 (self-evaluation)**: Used as the baseline for evaluating the system's initial responses.
- **Llama 3.1 (self-evaluation)**: Employed to observe the improvements in understanding and relevance within the updated model.
- **Cross-Evaluation (Llama 3 evaluated by Llama 3.1 and vice versa)**: This comparison helped in identifying potential strengths in response quality, relevancy, and model robustness across versions.

6.5.4 Discussion of Findings

Given these results, Llama 3.1 emerged as the more reliable model for the grading system, displaying higher accuracy, improved contextual understanding, and better consistency across varied test cases. For details on the specific implementation and configuration, refer to Appendix K.

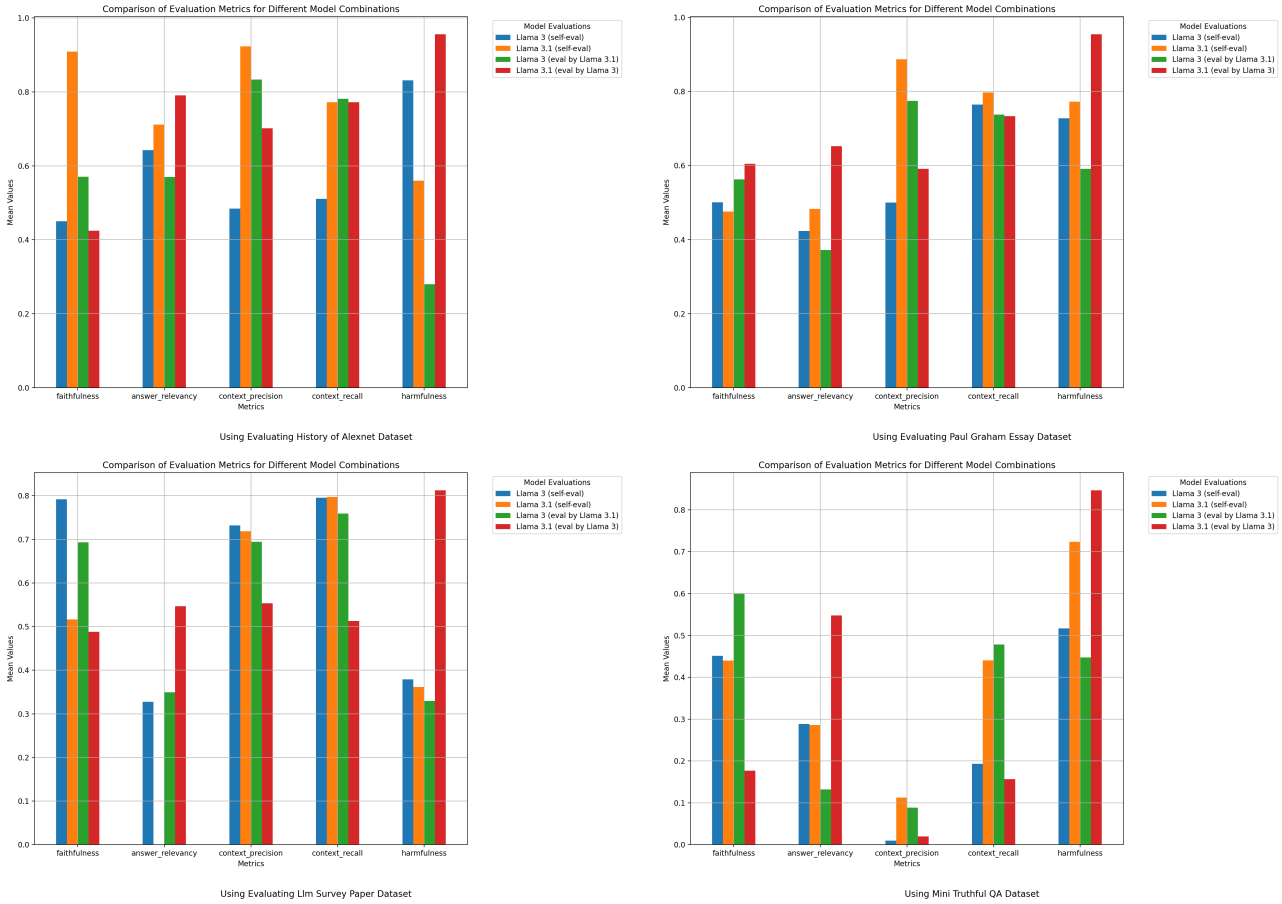

Figure 6.4: A 2x2 grid of images

6.5.5 Result Storage and Access

The evaluation results, including CSV files with metric comparisons and timing details, are saved in the ‘results‘ directory with the naming pattern:

`results/<DATASET>_query_<QUERY_MODEL>_eval_<EVALUATION_MODEL>.csv`

Timing results are saved as text files, facilitating future comparisons and performance tracking.

This thorough evaluation confirms Llama 3.1 as the preferred model for our system, laying a strong foundation for reliable, accurate, and fair automated grading.

Chapter 7

Distributed Processing and Scalability

As an educational institute adopts the grading system, we have to make sure that it can be scaled to potentially to hundreds or thousands of users and assignments that need to be graded. The ideal scaling for this system would be constant time i.e. $\mathcal{O}(1)$ but this is not possible as the number of questions and their complexity in an assignment might differ vastly.

7.1 Developing a Distributed Ollama System

7.1.1 Load Balancing

Load Balancing is a technique used to distribute network traffic or the computational load across multiple machines or servers. [9] This method stands out as it matches perfectly with our use case of the grader system where we are implementing HTTP API calls.

7.1.2 Testings with Ngrok

A quick search for load-balancing tools leads us to `Nginx`, which is a web server that can be used for Load Balancing. Nginx is widely used in industries and has good documentation.

Setting up a Nginx service is simple as it is available officially as a Docker image. The docker file for the Nginx server is listed below:

```
# Use the official NGINX image from the Docker Hub
FROM nginx:latest
# Remove the default NGINX configuration file
RUN rm /etc/nginx/nginx.conf
# Copy your custom NGINX configuration file to the appropriate location
COPY nginx.conf /etc/nginx/nginx.conf
# Expose port 80 to allow external traffic
EXPOSE 80
# Start the NGINX server
CMD ["nginx", "-g", "daemon off;"]
```

We need to provide a configuration file named `nginx.conf` that is in the `ini` format. This allows us to specify how many servers (classroom machines) are there and the port used to establish the connection. The configuration file in our case to distribute the traffic over the classroom machines would look like this:

```
events {
    worker_connections 1024;  # Max connections per worker
}
```

```
http {
    upstream backend_servers {
        least_conn;  # Least connections load balancing
        # Ollama servers
        server class01:11434;
        server class02:11434;
        server class03:11434;
        # Add or remove servers as needed
    }
    server {
        listen 11434;
        location / {
            proxy_pass http://backend_servers;
        }
    }
}
```

After implementing this `Nginx` server, it did demonstrate its effectiveness as a load balancer, however, this tool provides an extensive set of features that exceed our needs for our application and introduce added complexity and potential points of failure to the system.

7.1.3 Architecture Across Classroom Machines

The resources available for testing are from the Department of Scientific Computing, which includes the classroom machines. These machines are connected to each other over the same network that has a sufficient speed of 1000 MBit/s [1] and share the same naming schema i.e. `class01` to `class19` which will come in handy later. Also, these machines have the same shared storage which minimizes the need to download the LLM model and source code on each machine separately.

7.1.4 Parallelizing Query Processing

As our grading system relies on querying a LLM model, we will be parallelizing the query processing. For this we are using **Round-robin Scheduling** [10], which is a scheduling algorithm where each process (in our case a query) is assigned a machine in a cyclic order. It rotates through all the machines, restarting at the first machine once it reaches the end. This is visualized in figure 7.1 below:

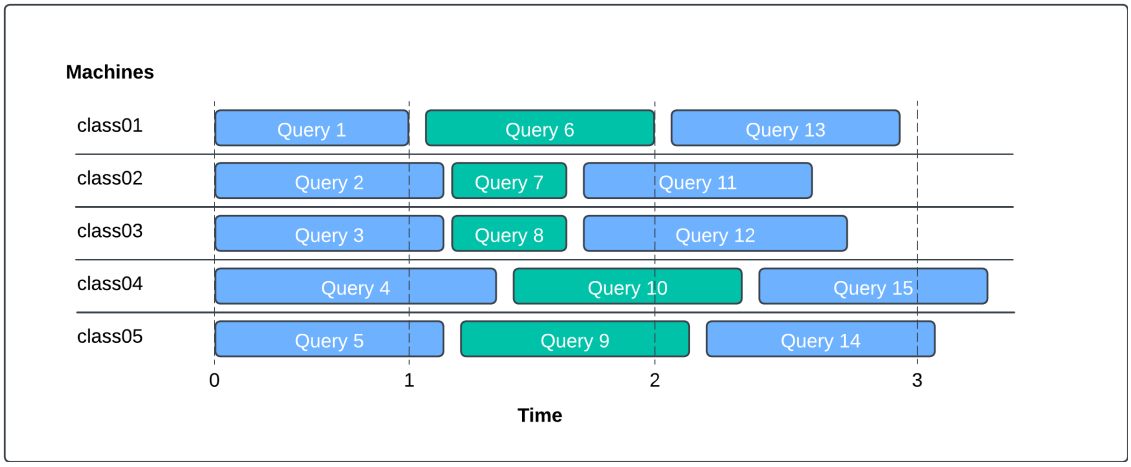

Figure 7.1: Visual representation of query scheduling

[1] Verified with command `$ cat /sys/class/net/eno1/speed`

To simplify the implementation of this algorithm, we created a Python class named `TaskScheduler` (see Appendix L), which handles the creation, scheduling, and execution of all tasks, that is, the processing of queries. This `TaskScheduler` class also captures any failed processes and also records the time taken by each task into a Pandas data frame using the `PerfCounterTimer` class (see Appendix M). However, the distribution of queries across different machines is done in a separate file where we are creating multiple LLM objects with different `base_url` parameters. This `base_url` parameter defines the machine on which the API call to execute the query will be executed.

This scheduling and load distribution is done by a separate machine, let's call it the 'master node', which will start Ollama servers on each of the classroom machines by running the `ollama serve` through SSH. This master node will then load all the queries, schedule them, and then distribute the queries by synchronizing the API calls to the machine running the Ollama server. Please take a look at Appendix N for the full script.

7.2 Scalability Results and Analysis

To test how well this method scales, we used the Natural Questions Dataset [11] from Google. This dataset includes questions along with their answers that can be used to test an LLM.

We are using the first 10000 questions from the training set to test how large amounts of queries can be processed over different numbers of machines. The plot 7.2 below shows how much time is required to process the 10000 queries for different numbers of machines.

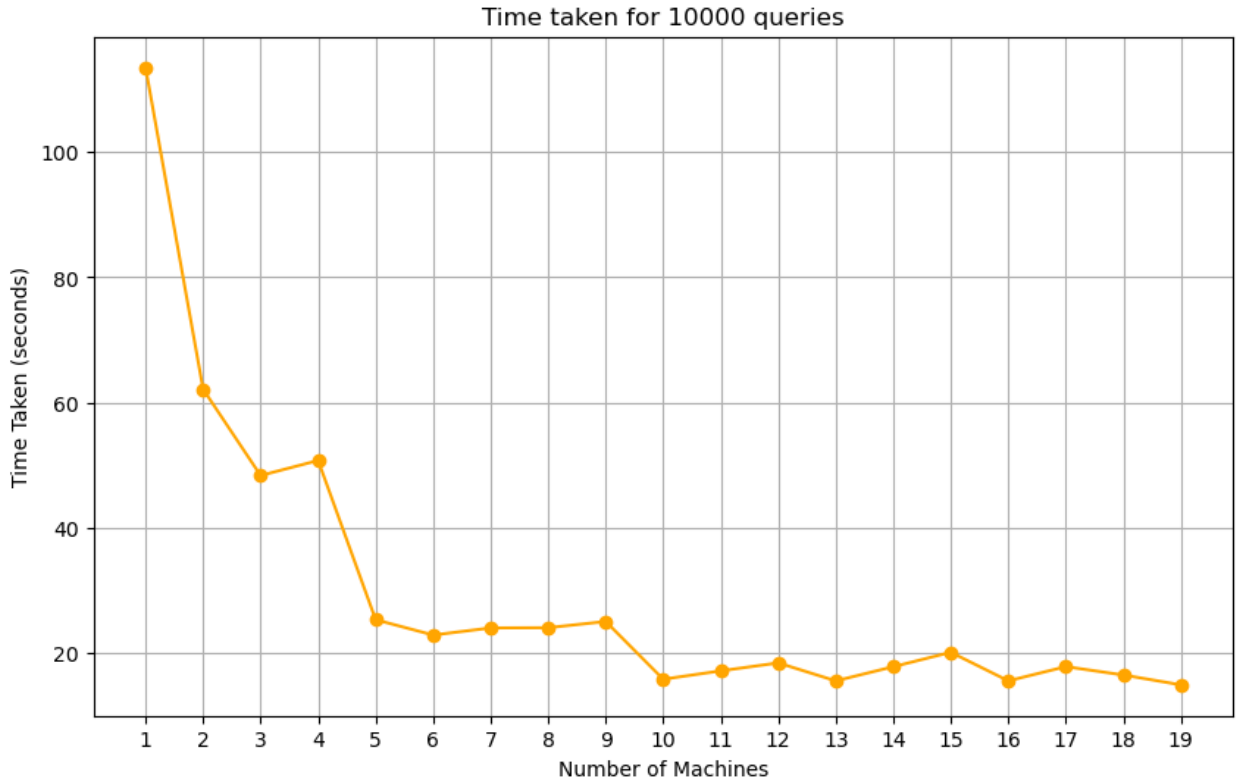

Figure 7.2: Time vs No. of Machines

This experiment was performed on classroom machines with Nvidia A5000 GPUs and a machine in the intelligence lab with Ryzen 9 5950X was used as a master node.

Chapter 8

Data Extraction and Processing

The grading system heavily depends on the data that is being provided to it beforehand, which includes the context and the embeddings. To create this embedding we are using various documents which include the presentation slides, reference material, and textbooks of a specific course. These documents can contain data in different formats ranging from simple text to complex flow charts. These documents present a challenge of extracting meaningful data that can be used by the grading system to improve its evaluations.

8.1 Parsing Documents with Llama Parse

LlamaParse is a document parsing tool developed by LlamaIndex that is designed to enhance retrieval-augmented generation (RAG) applications by efficiently parsing complex documents, especially those with embedded tables and charts. This is suitable for the type of documents that we are dealing with.

Also, LlamaParse can be integrated with LlamaIndex and it provides us with a `query_engine` which can be used to query our documents directly. This `query_engine` is based on the `VectorStoreIndex` created using the parsed documents and it can be used the LLM of choice.

8.2 Extracting Q&A Pairs Using LangChain

To improve the grading system's effectiveness, we conducted an experiment to generate question-and-answer (Q&A) pairs using the LangChain framework with Ollama. This approach allows us to structure the data extracted from educational documents into a format the can be used for both grading and also in future applications.

8.2.1 Conversion to JSON Format

The goal of this experiment is to convert text data into a structured JSON format with entries containing questions with their corresponding answers. Using LangChain's `LLMChain`, we set up a specific prompt template to guide the LLM in generating accurate Q&A pairs directly from the text. The output is a JSON object that represents the structured information in a format compatible with the grading system. This process involved several key steps:

- **Document Loading**: We used the 'PyPDFLoader' module to load and split the PDF document, titled "Question Bank 1 (Tan et al 2nd Edition)," into individual pages for easier processing.
- **Embeddings Generation**: The 'OllamaEmbeddings' library generated embeddings for each page's content, allowing us to capture the semantic meaning of the text accurately.
- **Prompt Template Creation**: A custom prompt template was designed to instruct the LLM to extract Q&A pairs in JSON format.

8.2.2 Preparing Data for Future Applications

After the Q&A pairs were extracted, the JSON format was stored and organized for potential future uses, such as retrieval-augmented generation (RAG) applications. The JSON structure allows seamless integration with various LLM-based tools, which can utilize these Q&A pairs to improve automated grading accuracy and provide meaningful feedback to students.

8.2.3 Project Setup and Execution

Setting up the LangChain experiment involved the following steps:

Requirements The project was developed using Python 3.7+ with the following dependencies:

- LangChain and 'langchain-community' for the main framework
- A suitable LLM model ('llama3.1') compatible with LangChain
- 'PyPDFLoader' for loading PDF documents

Installation To install the necessary libraries, the following command was used:

```
pip install langchain langchain-community
```

File Structure The project consists of the following files:

- `main.py`: Contains the code for loading the PDF, processing the text, and generating the JSON output.
- `Question Bank 1 (Tan et al 2nd Edition).pdf`: The source PDF document for QA extraction.

Execution Process

1. Load the PDF and split it into individual pages.
2. Generate embeddings for each page using 'OllamaEmbeddings'.
3. Use the 'PromptTemplate' and 'LLMChain' with 'llama3.1' model to extract questions and answers in JSON format.

The following prompt template was used:

```
template: str = """You are a teacher who is creating a list of questions and answers.
You are generating a usable json from the given text. The json has an array of objects
with two keys: question, answer. Given text is: {text} json: """
```

8.2.4 Sample Output

Below is a sample JSON output generated by LangChain for a section of text:

```
[
    {
        "question": "For each data set given below, give specific examples of
        ↪  classification, clustering,
        association rule mining, and anomaly detection tasks that can be performed on
        ↪  the data.",
        "answer": ""
    },
    {
```

```
        "question": "(a) Ambulatory Medical Care data1, which contains demographic and
        ↪  medical visit
        information for each patient (e.g., gender, age, duration of visit, physician's
        ↪  diagnosis,
        symptoms, medication, etc)",
        "answer": [
            {
                "task": "Classification",
                "description": "Diagnose whether a patient has a disease.",
                "rows": "Patient",
                "columns": "Patient's demographic and hospital visit information"
            },
            {
                "task": "Clustering",
                "description": "Find groups of patients with similar medical
                ↪  conditions",
                "rows": "A patient visit",
                "columns": "List of medical conditions of each patient"
            },
            {
                "task": "Association rule mining",
                "description": "Identify symptoms and medical conditions that co-occur
                ↪  together",
                "rows": "A patient visit",
                "columns": "List of symptoms and diagnosed conditions of the patient"
            },
            {
                "task": "Anomaly detection",
                "description": "Identify patients with rare medical disorders",
                "rows": "A patient visit",
                "columns": "Demographic attributes, symptoms, test results"
            }
        ]
    }
]
```

This JSON output format makes it easy to retrieve specific questions and their answers for both automated grading and supplementary educational tools. The structured JSON also supports further processing, enabling the data to be used across various retrieval-augmented generation (RAG) applications.

Chapter 9

Conclusion

9.1 Summary of Achievements

This project has explored the potential of using Large Language Models (LLMs) to improve the efficiency, scalability, and consistency of automated grading in educational settings. By investigating the capabilities of Retrieval-Augmented Generation (RAG), distributed processing, and fine-tuning approaches, this research demonstrates the feasibility of leveraging LLMs for tasks traditionally performed by human graders. Key tools such as LangChain, Llama.cpp, and Ollama provided insights into how a structured LLM framework could support efficient model management and query processing, indicating that LLMs could be a promising solution for addressing the growing demands in academic assessment.

9.2 Possibilities and Limitations

Current technology allows for grading factual and structured responses effectively, but struggles with nuanced, creative, or argumentative content that requires subjective evaluation. The potential to scale automated grading to handle large volumes is feasible, as demonstrated through distributed processing experiments, but achieving consistency in assessing subjective responses remains challenging. Thus, a hybrid approach, blending automated and human evaluation, may be the most effective and ethical path forward.

9.3 Future Work and Recommendations

While the project achieved significant milestones, there are areas that warrant further exploration and development:

Enhanced Fine-Tuning Techniques : Investigate more advanced fine-tuning methods, such as Low-Rank Adaptation (LoRA), to improve the model's ability to adapt to specific grading criteria and educational contexts while maintaining computational efficiency.

User Interface Development : Build a user-friendly interface for educators and students to interact with the grading system, incorporating features for feedback, customization of grading rubrics, and real-time analytics.

Expanded Evaluation Metrics : Incorporate additional evaluation metrics focused on educational impact, such as student learning outcomes and engagement levels, to assess the effectiveness of the automated grading system beyond technical performance.

Ethical Considerations and Bias Mitigation : Conduct thorough assessments of the system's fairness and potential biases, implementing strategies to ensure equitable treatment of all student responses regardless of language proficiency or writing style.

Integration with Learning Management Systems (LMS) : Explore seamless integration with existing LMS platforms like Canvas or Moodle to facilitate broader adoption and streamline the workflow for educators.

9.4 Future Vision for Automated Grading

I envision a future where automated grading systems are seamlessly integrated into Learning Management Systems (LMS) and support educators by providing reliable, consistent feedback. Ideally, this system would allow for real-time, nuanced grading, while also enabling educators to review and adjust criteria as needed. While automation can substantially reduce grading burdens, human oversight will remain vital, especially for complex or open-ended responses that require deeper contextual understanding.

Chapter 10

References

[1] Turnitin, Inc. *Gradescope*. https://www.turnitin.com/products/gradescope/. Accessed: 2024-10-18.

[2] Instructure, Inc. *Canvas*. https://www.instructure.com/canvas. Accessed: 2024-10-18.

[3] Wikipedia contributors. *Retrieval-augmented generation — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Retrieval-augmented_generation. Accessed: 2024-10-18. 2024.

[4] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401.

[5] Paulo Finardi et al. *The Chronicles of RAG: The Retriever, the Chunk and the Generator*. 2024. arXiv: 2401.07883 [cs.LG]. URL: https://arxiv.org/abs/2401.07883.

[6] Pankaj Saha et al. "Evaluation of Docker Containers for Scientific Workloads in the Cloud". In: *Proceedings of the Practice and Experience on Advanced Research Computing: Seamless Creativity*. PEARC '18. Pittsburgh, PA, USA: Association for Computing Machinery, 2018. ISBN: 9781450364461. DOI: 10.1145/3219104.3229280. URL: https://doi.org/10.1145/3219104.3229280.

[7] *LM Studio - Experiment with local LLMs*. en. URL: https://lmstudio.ai (visited on 10/28/2024).

[8] *AnythingLLM — The all-in-one AI application for everyone*. en. URL: https://anythingllm.com/ (visited on 10/28/2024).

[9] Einollah Jafarnejad Ghomi, Amir Masoud Rahmani, and Nooruldeen Nasih Qader. "Load-balancing algorithms in cloud computing: A survey". en. In: *Journal of Network and Computer Applications* 88 (June 2017), pp. 50–71. ISSN: 10848045. DOI: 10.1016/j.jnca.2017.04.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S1084804517301480 (visited on 10/30/2024).

[10] *Round-robin scheduling*. en. Page Version ID: 1237396566. July 2024. URL: https://en.wikipedia.org/w/index.php?title=Round-robin_scheduling&oldid=1237396566 (visited on 10/30/2024).

[11] Tom Kwiatkowski et al. "Natural Questions: A Benchmark for Question Answering Research". en. In: *Transactions of the Association for Computational Linguistics* 7 (Nov. 2019), pp. 453–466. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00276. URL: https://direct.mit.edu/tacl/article/43518 (visited on 10/30/2024).

Appendices

Appendix A

Related work

A.1 Integration of `PaperQA` with `Open-WebUI`

A.1.1 Open-WebUI

Open WebUI is a self-hosted AI interface that operates entirely offline, supporting various LLM runners like Ollama and OpenAI-compatible APIs. It offers seamless integration with multiple AI models and features like full Markdown and LaTeX support, image generation, and role-based access control. Installation is straightforward via Docker or Kubernetes, and the platform ensures strict data privacy by storing all information locally.

A.1.2 PaperQA

PaperQA is a Retrieval-Augmented Generation (RAG) system designed to assist researchers in navigating and extracting information from scientific literature. By leveraging LLMs and RAG techniques, PaperQA autonomously retrieves, processes, and synthesizes information from full-text scientific articles to provide accurate, context-rich responses with reliable citations. It outperforms existing LLMs on science question-answering benchmarks and matches human researchers in tasks requiring retrieval and synthesis of information from multiple papers.

A.1.3 Motivation for Integration

PaperQA although being a powerful tool for researchers lacks ease of use. It does not offer a user-friendly interface to interact with it, and the user has to go through the command line interface to interact with it. Integrating PaperQA with Open WebUI addresses this limitation by providing a user-friendly, web-based platform. By combining PaperQA's robust retrieval capabilities with Open WebUI's intuitive interface, researchers can focus more on analysis and interpretation, thereby improving productivity and the overall research experience.

A.1.4 Implementation

Backend

The Open WebUI applications backend is made using FastAPI and offer multiple API endpoints for the frontend. All the API traffic is handled by different FastAPI apps which include Ollama, OpenAI, RAG where each app handles its respective API calls and their execution.

We had to develop a new app named PaperQA which handles the inference through `paper-qa` python package. This app also handles other API calls related to the PaperQA such as retrieving a chat or getting inference statistics.

Additionally, we modified the existing RAG application to accommodate PaperQA's dependency on user-uploaded documents. Since the RAG application already manages document uploads, we introduced a new state variable, `RAG_PAPERQA_ACTIVE`, a boolean that indicates whether the uploaded documents are intended for the PaperQA application. This approach leverages the existing upload mechanisms, allowing documents to be efficiently forwarded to PaperQA without the need to develop new functions, thereby streamlining the integration process.

Frontend

The front end of the Open-WebUI which is built using Svelte and handles majority of the logic for user interactions and initialization of API calls to the backend. To integrate PaperQA, we had to add new functions that will handle chats with PaperQA, this function is named `generatePaperQAChatCompletion` and is alongside Ollama chat handlers as PaperQA uses Ollama for its inference.

We also added a new toggle switch in the model selection menu, where user can select to use PaperQA, shown in A.1. This toggle switch changes the URLs where the API calls are made and redirects those from `/ollama` to `/paperqa`. We also have to ensure all the configurations selected by the user in the settings menu are applied to PaperQA as well, for this we are using the same config object that is created for Ollama in the frontend and forwarding that to the PaperQA backend.

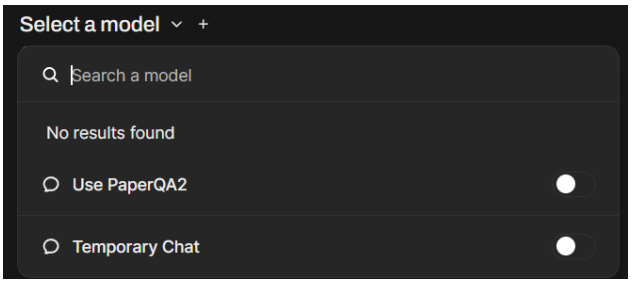

Figure A.1: Model selection menu with the new toggle for PaperQA integration

A.2 Execution

For using PaperQA through Open WebUI, follow the steps below:

- Clone the repository from `https://github.com/anand-kamble/open-webui/`
- Follow the instructions from official documentation: `https://docs.openwebui.com/getting-started/`
- Once you have everything working and an account setup in the local instance of Open WebUI, start by creating a new chat by clicking the option in top left corner.
- Start by typing in your query and uploading the documents as per usual.
- Select the desired model from the dropdown menu.
- Turn on the toggle to use PaperQA in the dropdown menu.

Now the chat you started will be utilizing PaperQA for inference. Since PaperQA takes more time than usual RAG applications it might look like the WebUI has frozen, but please be patient as it can take a couple of minutes depending on the documents.

A.3 Usage Example

To demonstrate the PaperQA, I uploaded one of my papers to the chat and asked the query *"Can you explain the details from this paper in short?"* and the response from the PaperQA answered in detail using the `llama3.1` as the base model. The image A.2 below shows how the PaperQA responded and also provided citations.

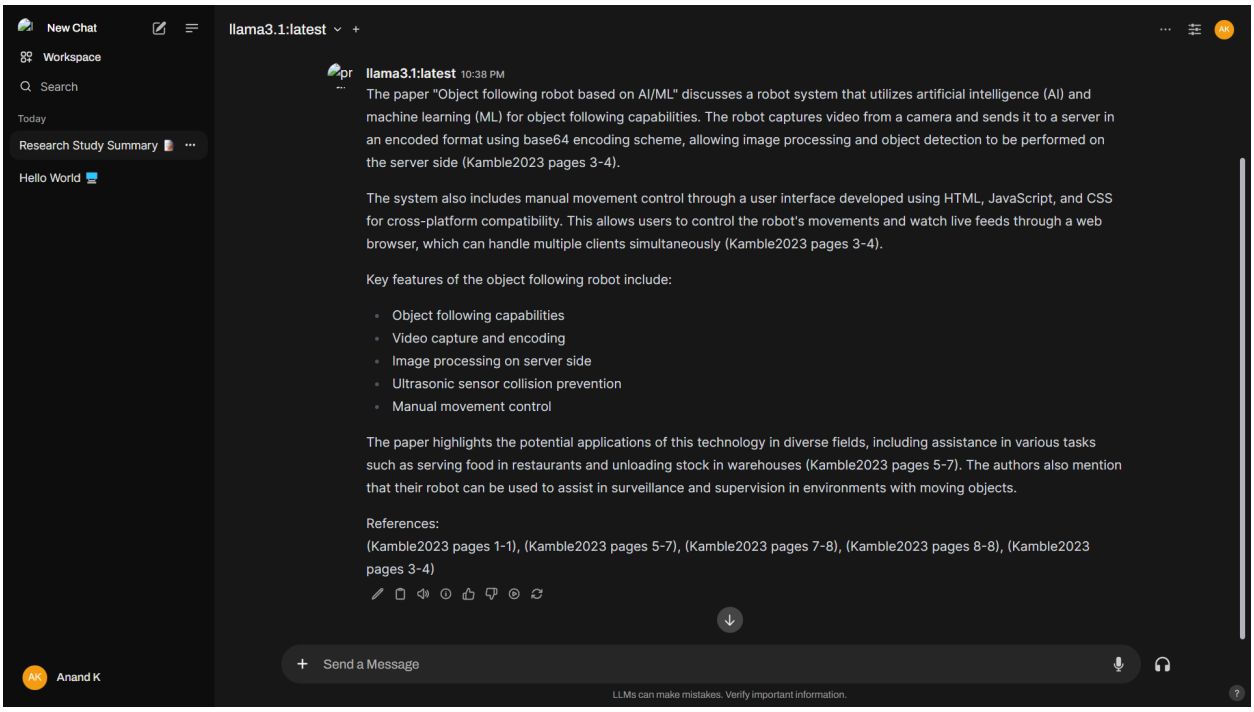

Figure A.2: PaperQA providing a summary based on the uploaded paper

Appendix B

Dockerfile - Llama.cpp

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_3/Dockerfile

```
# Use a base image, for example, Ubuntu
FROM ubuntu:latest

# Copy the file from the host machine to the container
COPY llava-v1.5-7b-q4.llamafile /llava-v1.5-7b-q4.llamafile


# Make the file executable
RUN chmod +x /llava-v1.5-7b-q4.llamafile

RUN  apt-get update && apt install -y curl &&\
curl -s https://ngrok-agent.s3.amazonaws.com/ngrok.asc |\
tee /etc/apt/trusted.gpg.d/ngrok.asc >/dev/null &&\
echo "deb https://ngrok-agent.s3.amazonaws.com buster main" |\
tee /etc/apt/sources.list.d/ngrok.list &&\
apt update &&\
apt install ngrok

COPY run_server.sh /run_server.sh
# Define the command to run the file
CMD ["/bin/bash", "/run_server.sh"]
```

Appendix C

Docker Compose - Llama.cpp

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_3/docker-compose.yaml

```yaml
services:
  web:
    build: .
    image: test:llamafile
    entrypoint: "/bin/sh"
    stdin_open: true
    tty: true
    command: bash -c "./llava-v1.5-7b-q4.llamafile --server --nobrowser & ngrok http 8080"
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [ gpu ]

```

Appendix D

Bash Script for running the server - Llama.cpp

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_3/run_server.sh

```
mkdir ~/.config &&\
mkdir ~/.config/ngrok &&\
touch ~/.config/ngrok/ngrok.yml &&\
echo "version: 2" > ~/.config/ngrok/ngrok.yml &&\
echo "authtoken: <your-authtoken-here>" >> ~/.config/ngrok/ngrok.yml &&\
./llava-v1.5-7b-q4.llamafile --server --nobrowser &\
ngrok http --domain=vastly-pleasing-sunbeam.ngrok-free.app 8080
```

Appendix E

Langchain RetrievalQA

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_3/langchain_experiment.py

```python
from langchain.llms import llamacpp
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores.chroma import Chroma
from langchain_community.document_loaders import PyPDFLoader
from langchain_community.embeddings import LlamaCppEmbeddings
from utils import Models, list_files

llm = llamacpp.LlamaCpp(model_path=Models.MISTRAL.value, n_ctx=2048, f16_kv=True)

DATA_ROOT = "../data/"
pdf_filenames = list_files(DATA_ROOT)

splitData = []
for pdf in pdf_filenames:
    pdfLoader = PyPDFLoader(DATA_ROOT + pdf)
    data = pdfLoader.load_and_split()
    for d in data:
        splitData.append(d)

llamaEmed = LlamaCppEmbeddings(seed=100, model_path=Models.MISTRAL.value)

template = """
You are a professor of graduate level course data mining.
The question asked is: `{question}`
rate the following answer on a scale of 1 to 10, where 1 is the worst and 10 is the best:
`{answer}`
Give answer in format: Rating = x/10
"""

prompt = PromptTemplate(
    template=template,
    input_variables=["answer", "question"],
)
```

```
36
37 text_splitter = RecursiveCharacterTextSplitter(chunk_size=256, chunk_overlap=20)
38 docs = text_splitter.split_documents(splitData)
39 vec_store = Chroma.from_documents(splitData, llamaEmed)
40 base_retriever = vec_store.as_retriever()
41
42
43 from langchain.chains import LLMChain, RetrievalQA
44
45 chain = LLMChain(llm=llm, prompt=prompt)
46
47 qa = RetrievalQA.from_chain_type(
48     llm=llm, chain_type="map_reduce", retriever=base_retriever
49 )
50
51
52 question = "Is K-means a clustering method?"
53 answer = "K-means is not a clustering method"
54 qa.run(answer=answer, question=question)
```

Appendix F

`create-llama` Directory structure

Directory structure and files generated by the `create-llama@0.3.8` package when a `NextJS` back-end is selected.

```
.
|-- Dockerfile
|-- README.md
|-- app
|   |-- api
|   |-- components
|   |-- favicon.ico
|   |-- globals.css
|   |-- layout.tsx
|   |-- markdown.css
|   |-- observability
|   |-- page.tsx
|-- config
|   |-- tools.json
|-- data
|   |-- 101.pdf
|-- next-env.d.ts
|-- next.config.json
|-- next.config.mjs
|-- output
|   |-- llamacloud
|   |-- tools
|   |-- uploaded
|-- package.json
|-- postcss.config.js
|-- prettier.config.js
|-- public
|   |-- llama.png
|-- tailwind.config.ts
|-- tsconfig.json
|-- webpack.config.mjs
```

Appendix G

Phoenix Evaluation

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_5/Phoenix-Evaluation.py

```python
# %%
from phoenix.session.evaluation import get_qa_with_reference, get_retrieved_documents
from llama_index.core import SimpleDirectoryReader, VectorStoreIndex, set_global_handler
import phoenix as px
from llama_index.embeddings.ollama import OllamaEmbedding
from llama_index.core import Settings
from llama_index.llms.ollama import Ollama
from llama_index.core import set_global_handler
import asyncio

# %%
# Without this command, OpenAI model is used
# embedding_model = HuggingFaceEmbedding(
#     model_name="BAAI/bge-small-en-v1.5"  # small model
# )

embedding_model = OllamaEmbedding(
    model_name="phi3:latest", base_url="http://localhost:11434")

# %%
llm = Ollama(model="phi3:latest", temperature=0.7,
             base_url="http://localhost:11434", request_timeout=3600.0)
set_global_handler("arize_phoenix")
Settings.embed_model = embedding_model
Settings.llm = llm
# %%

corpus_schema = px.Schema(
    id_column_name="id",
    document_column_names=px.EmbeddingColumnNames(
        vector_column_name="embedding",
        raw_data_column_name="text",
    ),
)

```

```python
async def evaluate():
    """
    This asynchronous function is responsible for launching a Phoenix application.
    The Phoenix application is launched in the current event loop.
    """
    # %%
    px.launch_app()

    # %%
    documents = SimpleDirectoryReader("files").load_data()
    #%%
    index = VectorStoreIndex.from_documents(documents)
    #%%
    qe = index.as_query_engine(llm=llm)
    #%%
    response1 = qe.query("what is k-means")
    # %%
    response2 = qe.query("what is fuzzy clustering?")
    # %%
    print(str(response1) + "\n" + str(response2))

    # %%
    model = Ollama(model="phi3:latest") #OpenAIModel(model="gpt-3.5-turbo-instruct")
    # %%
    print(f"{px.Client()=}")
    Client = px.Client(endpoint="http://localhost:6006")

    # %%
    Client.get_evaluations()
    # %%
    Client.get_spans_dataframe()
    # %%
    retrieved_documents_df = get_retrieved_documents(Client)

    # %%
    Client.query_spans
    # %%
    get_qa_with_reference(Client)

    # %%
    queries_df = get_qa_with_reference(Client)
    # %%
    # print("queries_df= ", queries_df)
    print(queries_df.head())
    print("Press Ctrl+C to stop the server.")


if __name__ == "__main__":
    """
    This is the main entry point of the program. It sets up an asyncio event loop,
    ensures that the evaluate() coroutine is scheduled to run, and starts the event loop.
    If a KeyboardInterrupt is raised (which usually happens when the user hits Ctrl+C),
    it simply passes and proceeds to the finally block.
    In the finally block, it prints a message, closes the event loop, and quits the program.
```

```
    """
    loop = asyncio.get_event_loop()
    try:
        asyncio.ensure_future(evaluate())
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        print("Closing Loop")
        loop.close()
        quit()
```

Appendix H

Langsmith Tracing

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_5/Langsmith-Tracing.py

```python
# %%
import os
from typing import Any, List

from dotenv import load_dotenv
from langchain_huggingface import HuggingFacePipeline
from langsmith import traceable
from utils import constants, pickel_loader

load_dotenv()


def load_correct_answers():
    return pickel_loader(constants.CORRECT_ANSWER_PATH)


def load_student_answers(path: str) -> Any:
    return pickel_loader(path)


def format_prompt(
    correct_answer: str, student_answer: str, context: str = "No context provided"
) -> List[dict[str, str]]:
    return [
        {
            "role": "system",
            "content": f"You are a assignment grader, you grade explainations from 0 to 10.\nContext:{context} \nThe expected explaination is:{correct_answer}",
        },
        {
            "role": "user",
            "content": f"Please grade the following answer: {student_answer}.",
        },
    ]

```

```python

# The @traceable decorator is used to track the execution of the function.
# It can be used for logging, debugging, or performance monitoring.
@traceable
def invoke_llm(messages: List[dict[str, str]]) -> Any:
    llm = HuggingFacePipeline.from_model_id(
        model_id="microsoft/Phi-3-mini-128k-instruct",  # The model to be used
        task="text-generation",  # The task to be performed
        pipeline_kwargs={
            "max_length": 1024,  # The maximum length of the generated text
        },
        device_map="auto",  # Automatically select the device to run the model on
    )

    return llm.invoke(messages)


def main():
    print("Running grader ...")

    # Load the correct answers
    correct_answers = load_correct_answers()

    # Load the student's answers
    # Path is relative to the ./run.sh script.
    student_answers = load_student_answers("./student_code/answers.pkl")

    # Define the context for the grading prompt
    context = "K-Means and Agglomerative Clustering are two popular unsupervised machine
    ↪  learning algorithms used for data clustering. While both methods have their
    ↪  strengths and weaknesses, K-Means is generally more efficient for large datasets
    ↪  due to its linear time complexity compared to Agglomerative Clustering's
    ↪  quadratic time complexity. This efficiency allows K-Means to handle large
    ↪  datasets more quickly and effectively, making it a preferred choice for
    ↪  production-scale systems. Additionally, K-Means is more scalable and can handle
    ↪  datasets with over 10,000 data points, whereas Agglomerative Clustering can
    ↪  become computationally expensive and impractical for such large datasets."

    # Format the grading prompt
    messages = format_prompt(
        correct_answers["question1"]["(c) explain"],
        student_answers["question1"]["(c) explain"],
        context=context,
    )

    # Invoke the language model to grade the student's answer
    response = invoke_llm(messages)
    print("Response is: ", response)


if __name__ == "__main__":
    main()
```

Appendix I

TruLens Evaluation

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_5/Trulens-Evaluation.py

```python
import time

import numpy as np
import pandas as pd

from langchain_community.embeddings import OllamaEmbeddings
from llama_index.core import SimpleDirectoryReader, VectorStoreIndex
from llama_index.core.settings import Settings
from llama_index.llms.ollama import Ollama
from trulens_eval import Feedback, Tru, TruLlama
from trulens_eval.app import App
from trulens_eval.feedback.provider import LiteLLM

num_ctx = 2048 * 4
embeddings = OllamaEmbeddings(model="phi3:latest", num_ctx=num_ctx, show_progress=True)

Settings.embed_model = embeddings

# Load documents and create a vector store index
documents = SimpleDirectoryReader("../data").load_data()
start = time.time()
index = VectorStoreIndex.from_documents(documents)
end = time.time()
print(f"Indexing took {end - start} seconds with num_ctx={num_ctx}")

# %% Initialize the LLM and create a query engine
generator_llm = Ollama(model="phi3:latest")
query_engine = index.as_query_engine(llm=generator_llm)

# %%
# Configure the LiteLLM provider for feedback functions
provider = LiteLLM(
    model_engine="ollama/phi3:latest",
    endpoint="http://localhost:11434",
    kwargs={"set_verbose": True},  # Verbose mode for easier debugging
```

```
)

# Select the context for the application
context = App.select_context(query_engine)

# %%
# Define feedback functions

# Groundedness feedback function
f_groundedness = (
    Feedback(provider.groundedness_measure_with_cot_reasons)
    .on(context.collect())  # Collect context chunks into a list
    .on_output()
)

# Relevance feedback functions
f_answer_relevance = Feedback(provider.relevance).on_input_output()
f_context_relevance = (
    Feedback(provider.context_relevance_with_cot_reasons)
    .on_input()
    .on(context)
    .aggregate(np.mean)  # Aggregate relevance scores using mean
)

# %%
# Initialize the TruLlama query engine recorder with feedback functions
tru_query_engine_recorder = TruLlama(
    query_engine,
    app_id="LlamaIndex_App1",
    feedbacks=[f_groundedness, f_answer_relevance, f_context_relevance],
)

# %%
# Load the test dataset
testset = pd.read_csv("../testset.csv")

# Create a dictionary to hold test questions and ground truths
testset_dict = {
    "question": list(testset["question"]),
    "ground_truth": list(testset["ground_truth"]),
}

# %%
# Query the engine with the first question in the testset and record the process
with tru_query_engine_recorder as recording:
    for question in testset_dict["question"]:
        print(f"Querying the engine with question: {question}")
        query_engine.query(question)
    # print(f"Querying the engine with question: {testset['question'][0]}")
    # query_engine.query(testset["question"][0])

    # %%
    # Retrieve the record of the app invocation
    rec = recording.get()  # Use .get if only one record
```

```
    # recs = recording.records  # Use .records if multiple

    # %%
    # Initialize Tru for accessing records and feedback
    tru = Tru()
    # Uncomment the following line to run the Tru dashboard
    # tru.run_dashboard()

    # %%
    # Retrieve the feedback results and print them
    for feedback, feedback_result in rec.wait_for_feedback_results().items():
        print(feedback.name, feedback_result.result)
    # Retrieve records and feedback for the specified app_id
    records, feedback = tru.get_records_and_feedback(app_ids=["LlamaIndex_App1"])
    # Display the records
    print(records.head())

    # %%
    tru.run_dashboard()

# %%
```

Appendix J

Testset Generation

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_5/Testset-Generation.py

```python
from llama_index.core import SimpleDirectoryReader, VectorStoreIndex
from llama_index.core.settings import Settings
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.embeddings.ollama import OllamaEmbedding
from llama_index.llms.ollama import Ollama
from ragas.integrations.llama_index import evaluate
from ragas.metrics import (
    answer_relevancy,
    context_precision,
    context_recall,
    faithfulness,
)
from ragas.metrics.critique import harmfulness
from ragas.testset.generator import TestsetGenerator


def main():
    print("Loading documents...")
    # Load documents from the specified directory
    documents = SimpleDirectoryReader("./data").load_data()
    print("Initializing generator and critic models...")

    # Initialize the generator and critic models using the Ollama LLM
    generator_llm = Ollama(model="phi3:latest")
    critic_llm = Ollama(model="phi3:latest")  # Alternatively, OpenAI(model="gpt-4")
    print("Initializing embeddings...")

    # Initialize embeddings using the HuggingFace embedding model
    embeddings = HuggingFaceEmbedding(model_name="BAAI/bge-small-en-v1.5")

    print("Initializing testset generator...")
    # Create a testset generator using the initialized models and embeddings
    generator = TestsetGenerator.from_llama_index(
        generator_llm=generator_llm,
        critic_llm=critic_llm,
```

```
        embeddings=embeddings,
    )
    print("Generating testset...")
    # Generate the test set from the loaded documents
    # Note: This process might take some time; be patient if it seems to be stuck at
    ↪  certain percentages
    testset = generator.generate_with_llamaindex_docs(
        documents,
        test_size=10,
        distributions={simple: 0.5, reasoning: 0.25, multi_context: 0.25},
    )

    print("Writing testset to CSV...")

    testset_df = testset.to_pandas()

    # Save the generated test set as a CSV file
    testset_df.to_csv("testset.csv", index=False)
    print("Testset generation completed successfully.")

    ### ============= BUILDING THE QUERY ENGINE ================== ###

    # Updating the embed model, this is required for the VectorStoreIndex
    # https://docs.llamaindex.ai/en/stable/module_guides/models/embeddings/#modules
    Settings.embed_model = embeddings

    print("Building the vector index...")
    vector_index = VectorStoreIndex.from_documents(documents)

    print("Building the query engine...")
    query_engine = vector_index.as_query_engine(llm=generator_llm)

    print("Querying the engine...")
    response_vector = query_engine.query(testset_df["question"][0])
    print(f"{response_vector=}")

    ### ============= EVALUATING THE QUERY ENGINE ============= ###

    metrics = [
        faithfulness,
        answer_relevancy,
        context_precision,
        context_recall,
        harmfulness,
    ]

    # using GPT 3.5, use GPT 4 / 4-turbo for better accuracy
    evaluator_llm = critic_llm  # OpenAI(model="gpt-3.5-turbo")
    # USING CRITIC LLM TO KEEP EVERYTHING LOCAL FOR NOW.
    testset_dict = (testset.to_dataset()).to_dict()
    print("Evaluating the query engine...")
    result = evaluate(
        query_engine=query_engine,
        metrics=metrics,
```

```
        dataset=testset_dict,
        llm=evaluator_llm,
        embeddings=OllamaEmbedding(model_name="phi3:latest"),
        raise_exceptions=False
    )

    print("========= RESULTS =========")
    print(result)

    result.to_pandas().to_csv("evaluation_results.csv", index=False)


if __name__ == "__main__":
    main()
```

Appendix K

Evaluation Implementation

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_5/Evaluation-Implementati
py

```python
# %%
import time
from ragas.integrations.llama_index import evaluate
from ragas.metrics.critique import harmfulness
from ragas.metrics import (
    faithfulness,
    answer_relevancy,
    context_precision,
    context_recall,

)
from dotenv import load_dotenv
import json
import os
from llama_index.core.settings import Settings
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader
from llama_index.llms.ollama import Ollama
from llama_index.llms.openai import OpenAI
# from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.embeddings.ollama import OllamaEmbedding
from ragas.testset.evolutions import simple, reasoning, multi_context, conditional
from llama_index.core import SimpleDirectoryReader
from datasets import Dataset
# %%

QUERY_MODEL = "llama3.1"
EVALUATION_MODEL = "llama3.1"
DATASET = "PatronusAIFinanceBenchDataset"

# %%
time_dict = {}
start_time: float = time.time()
# %%
embeddings = OllamaEmbedding(model_name=QUERY_MODEL,base_url="http://class02:11434")
Settings.embed_model = embeddings
```

```
end_time = time.time()
time_dict['embedding_setup'] = end_time - start_time
print("Time taken for embedding setup: ", time_dict['embedding_setup'])
# %%
start_time = time.time()
documents = SimpleDirectoryReader(
    f"./data/{DATASET}", required_exts=[".pdf",".txt"], recursive=True).load_data()
end_time = time.time()
time_dict['document_loading'] = end_time - start_time
print("Time taken for document loading: ", time_dict['document_loading'])
# %%
print("Building the vector index...")
start_time = time.time()
vector_index = VectorStoreIndex.from_documents(documents[:2])
end_time = time.time()
time_dict['vector_index_building'] = end_time - start_time
print("Time taken for vector index building: ",
      time_dict['vector_index_building'])
# %%
print("Building the query engine...")
start_time = time.time()
generator_llm = Ollama(model=QUERY_MODEL, request_timeout=600.0,
                       base_url="http://class02:11434",
                       additional_kwargs={"max_length": 512})
query_engine = vector_index.as_query_engine(llm=generator_llm)
end_time = time.time()
time_dict['query_engine_building'] = end_time - start_time
print("Time taken for query engine building: ",
      time_dict['query_engine_building'])
# %%
metrics = [
    faithfulness,
    answer_relevancy,
    context_precision,
    context_recall,
    harmfulness,
    # max_workers = 24 # Found this argument from the RunConfig dataclass.
]
# %%
#   # OpenAI(model="gpt-4")
critic_llm =
↪  Ollama(model=EVALUATION_MODEL,base_url="http://class01:11434",request_timeout=600.0)
# using GPT 3.5, use GPT 4 / 4-turbo for better accuracy
evaluator_llm = critic_llm  # OpenAI(model="gpt-3.5-turbo")
# USING CRITIC LLM TO KEEP EVERYTHING LOCAL FOR NOW.

start_time = time.time()

llama_rag_dataset = None
with open(f"data/{DATASET}/rag_dataset.json", "r") as f:
    llama_rag_dataset = json.load(f)

testset = {
    "question": [],
```

```
    "ground_truth": [],
}


# Here we are using the reference answer as the ground truth.
for item in llama_rag_dataset["examples"]:
    testset["question"].append(item["query"])
    testset["ground_truth"].append(item["reference_answer"])

    """
    Here I realized that for evluation we need to provide a Dataset object to the ragas evaluate function.
    Not just the dictionary. So I will convert the dictionary to a Dataset object.

    I found this by look at the source code, and the type given
    """

dataset: Dataset = Dataset.from_dict(testset)

end_time: float = time.time()
time_dict['testset_loading'] = end_time - start_time
print("Time taken for testset loading: ", time_dict['testset_loading'])
# %%

print("Evaluating the query engine...")
start_time = time.time()
result = evaluate(
    query_engine=query_engine,
    metrics=metrics,
    dataset=testset,
    llm=evaluator_llm,

    embeddings=OllamaEmbedding(model_name=EVALUATION_MODEL,base_url="http://class03:11434"),
    raise_exceptions=False
)

end_time = time.time()
# %%
time_dict['evaluation'] = end_time - start_time
print("Time taken  for evaluation: ", time_dict['evaluation'])
# %%
result.to_pandas().to_csv(f"results/{DATASET}_query_{QUERY_MODEL}_eval_{EVALUATION_MODEL}.csv")
# %%

# Save timing results to a text file
with open(f"results/{DATASET}_query_{QUERY_MODEL}_eval_{EVALUATION_MODEL}.txt", "w") as f:
    for key, value in time_dict.items():
        f.write(f"{key}: {value} seconds\n")

# %%
```

Appendix L

TaskScheduler class

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_6/TaskScheduler.py

```python
from typing import List, Callable, Any
import concurrent.futures
from timer import PerfCounterTimer
import multiprocessing


class TaskScheduler:
    def __init__(self, max_workers: int = multiprocessing.cpu_count(), timeout: int = 60):
        self.executor = concurrent.futures.ThreadPoolExecutor(
            max_workers=max_workers
        )
        self.tasks = []
        self.failed_tasks = []
        self.futures = []
        self.timer = PerfCounterTimer()
        self.timeout = timeout

    def add_task(self, id: str, task: Callable[..., Any], *args):
        with PerfCounterTimer(id).timeit():
            future = self.executor.submit(task, *args)
            self.futures.append(future)
            self.tasks.append((task, args))
            future.add_done_callback(self._task_done)

    def _task_done(self, future):
        try:
            result = future.result()
            print(f"Task completed with result: {result}")
        except Exception as e:
            print(f"An error occurred: {e}")
            self.failed_tasks.append(self.tasks[self.futures.index(future)])

    def execute_tasks(self):
        concurrent.futures.wait(self.futures, timeout=self.timeout)
```

```
35
36      def get_results(self) -> tuple[List[Any], List[tuple[Callable[..., Any], tuple]]]:
37          results = []
38          for future in self.futures:
39              try:
40                  result = future.result()
41                  results.append(result)
42              except Exception as e:
43                  print(f"An error occurred: {e}")
44                  self.failed_tasks.append(
45                      self.tasks[self.futures.index(future)]
46                  )
47          return results, self.failed_tasks
48
```

Appendix M

PerfCounterTimer class

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_6/PerfCounterTimer.py

```python
import time
import numpy as np
from contextlib import contextmanager
from collections import defaultdict
import pandas as pd


class PerfCounterTimer:
    timings = defaultdict(list)
    columns = ["name", "min", "mean", "std", "count"]
    df = pd.DataFrame(columns=columns)

    def __init__(self, name=""):
        self.name = name

    @contextmanager
    def timeit(self, count: int = 1):
        start_time = time.perf_counter()
        yield
        end_time = time.perf_counter()
        elapsed_time = (end_time - start_time) / count
        PerfCounterTimer.timings[self.name].append(elapsed_time)

    @classmethod
    def reset(cls):
        cls.timings = defaultdict(list)

    def __call__(self, count: int = 1):
        return self.timeit(count)

    @classmethod
    def report(cls, msg="") -> defaultdict:
        out_dict = defaultdict(dict)
        if msg:
            print(f"\n{msg}")
```

```
        for name, times in cls.timings.items():
            out_dict[name] = {}
            mean_total_time = np.mean(times)
            std_total_time = np.std(times)
            min_total_time = np.min(times)
            counter = len(times)
            out_dict[name]["mean"] = mean_total_time
            out_dict[name]["std"] = std_total_time
            out_dict[name]["min"] = min_total_time
            out_dict[name]["count"] = counter
            print(
                f"Name: {name}, Count: {counter}, Total Time: {min_total_time:7.4f}
                ↪  seconds, "
                f"Timings: {mean_total_time / counter:7.4f} each"
            )
        print()

        # Update the DataFrame with the new data
        data_dict = defaultdict(dict)
        for name, times in cls.timings.items():
            mean_total_time = np.mean(times)
            std_total_time = np.std(times)
            min_total_time = np.min(times)
            counter = len(times)
            data_dict[name]["name"] = name
            data_dict[name]["mean"] = mean_total_time
            data_dict[name]["std"] = std_total_time
            data_dict[name]["min"] = min_total_time
            data_dict[name]["count"] = counter

        # Convert data_dict to DataFrame and concatenate with existing df
        new_rows = pd.DataFrame.from_dict(data_dict, orient="index")
        cls.df = pd.concat([cls.df, new_rows], ignore_index=True)

    @classmethod
    def get_dataframe(cls) -> pd.DataFrame:
        return cls.df
```

Appendix N

Master Node Script

You can check the file on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_6/Master_node_Script.sh

```bash
#!/bin/bash

# List of machines
machines=("class01" "class02" "class03" "class04" "class05" "class06" "class07" "class08" "class09" "class10" "class11" "class12" "class13" "class14" "class15" "class16" "class17" "class18" "class19")

# List of colors
# This makes it easy to identify the outputs from different machines
colors=(
    "\033[31m" # Red
    "\033[32m" # Green
    "\033[33m" # Yellow
    "\033[34m" # Blue
    "\033[35m" # Magenta
    "\033[36m" # Cyan
    "\033[37m" # White
    "\033[91m" # Bright Red
    "\033[92m" # Bright Green
    "\033[93m" # Bright Yellow
    "\033[94m" # Bright Blue
    "\033[95m" # Bright Magenta
    "\033[96m" # Bright Cyan
    "\033[97m" # Bright White
)

# Get the current machine's hostname
current_machine=$(hostname)

# Function to run the command on each machine and color the output
run_command() {
    local machine=$1
    local color=$2
    ssh -o "StrictHostKeyChecking no" "$machine" 'bash -s' < ./ollama_script.sh | while IFS= read -r line; do
```

```
        echo -e "${color}${machine}: ${line}\033[0m"
    done
}

# Loop over each machine and run the command in the background
for i in "${!machines[@]}"; do
    machine=${machines[$i]}
    color=${colors[$((i % ${#colors[@]}))]}
    if [ "$machine" != "$current_machine" ]; then
        run_command "$machine" "$color" &
    fi
done

# Wait for all background processes to finish
wait
```

The above script depends on the `ollama_script.sh` file which includes the following:

```
#!/bin/bash

# Get the IP address of the machine
IP_ADDRESS=$(hostname -I | awk '{print $1}')

# Set the OLLAMA_HOST environment variable
export OLLAMA_HOST="$IP_ADDRESS:11434"

# Run the ollama serve command with a timeout of 3600 seconds (1 hour)
timeout 21600 ~/binaries/ollama serve
```

You can check the file above on Github:
https://github.com/anand-kamble/automatic-grading-using-llm/blob/main/chapter_6/ollama_script.sh