GENERATIVE PRE-TRAINED TEACHER

Evangelos Lamprou
s232462@dtu.dk

Spyridon Pikoulas
s230284@dtu.dk

ABSTRACT

In this project, we present the development of a Large Language Model (LLM)-powered tutoring application. We provide a system which, given a set of teaching material in the form of lecture transcripts can provide a student with help in understanding the material, create tests for assessing the student's understanding and prepare notes based on the lecture content. We briefly go over the theoretical background of LLMs and discuss the different engineering decisions we took during the system's development, showcasing the merits as well as difficulties of creating LLM-based applications both using large commercial language models as well as smaller models that can run on a modern laptop.

Index Terms— Large Language Models, Education, Open Source Models, Software Engineering

1. BACKGROUND

1.1. Large Language Models

Large language models (LLMs) represent a transformative force in artificial intelligence, offering very strong capabilities in understanding and generating human language [1]. Originating from early computational linguistics, these models have rapidly advanced in the past decade, driven by breakthroughs in machine learning and data processing. Notable examples are OpenAI's GPT-4 model [2] as well as Meta's open source LLaMa model [3].

1.2. Model Utilization

Integrating the capabilities of LLMs into applications is done through the use of an application programming interface (API), which is a protocol through a programmer can programmatically send prompts to the model and retrieve it's results. The majority of public APIs provide the following capabilities:

- Text Completion: Text completion is the core capability of language, where a a piece of text is sent to the model, the "prompt", with the model returning a possible continuation of the given text.
- Text Embeddings: Text embeddings are a sophisticated method of representing words, phrases, or entire documents as vectors in a high-dimensional space. This technique provides a much denser and semantically rich representation. For example, two semantically related pieces of text like "Where did John go?" and "John went to the park." will be placed close to each other in the embeddings space.
- Fine-tuning involves updating a Language Learning Model's (LLM's) weights using question-answer pairs to align with those data points. Its effectiveness is debated: some sources claim it imparts new knowledge and reduces "hallucinations", while others suggest it only refines the *form* of responses [4].

1.3. LLM Software Frameworks

To facilitate faster development of LLM-powered applications, there have been released a number of frameworks that abstract some of the common logic that goes into such projects. They offer features such as document loaders (tools that convert data from various formats like .pdf and .srt to text), memory structures (tools to allow for the LLM to recite previous parts of the conversation), retrievers (tools that allow for retrieval augmented generation from a plethora of sources) and agents (a methodology for having an LLM make highger level planning decisions towards achieving a goal). In this project, we used the LangChain[1] framework because of it's popularity. Similar tools exist such as RagChain[2] and LlaMaIndex[3].

2. SYSTEM OVERVIEW

2.1. System

The interface of *GPTeacher* is built using *Streamlit*[4], chosen for its user-friendly design and range of pre-built components, such as the chat interface which mirrors the LLM's conversational system-assistant-user format.

[1] https://www.langchain.com/
[2] https://github.com/NomaDamas/RAGchain
[3] https://www.llamaindex.ai/
[4] https://streamlit.io/

Users can choose from a variety of lectures, each accompanied by a video, a transcript, and lecture notes. Then they select from a range of features, each offering a unique value to the learning experience.

In the background, each feature makes use of LLMs in a distinct manner:

- **Chat-Based:** For features that require continuous interaction, such as question-answering or discussions, we leverage the LLM's conversational capabilities, including its memory function. This allows for a cohesive and context-aware dialogue.
- **One-Time Use:** Some features need a single interaction with the LLM, leveraging its text comprehension abilities, like creating content sections.
- **Combined LLM Usage:** Certain features integrate multiple LLMs. This approach is beneficial for dividing complex tasks into smaller, reducing the complexity of the model's task. ensuring better results. It also utilizes the LLM's ability to generate outputs in specific formats, which can be directly used in the code or serve as input for another LLM instance.

2.2. Features

Our supported features are as follows:

1. **Question & Answering:** This feature enables students to ask any questions related to the lecture and receive answers, enhanced with conversation memory for context-aware dialogues.
2. **Tests:** The application generates tests based on the student's level and lecture content. These tests are designed to evaluate how much the student has learned from the lecture.
3. **Notes Summary:** This feature provides a summary of the most important information from the lecture notes and/or book pages. It condenses the material to its essential points while also structuring the content effectively.
4. **Student Adaptation:** We humanize the LLM's output by matching responses the the student's emotional state using sentiment analysis.

2.3. Material

In the realm of academic lectures, there is a vast array of topics and formats, each presenting distinct challenges to an LLM when answering questions. To effectively demonstrate the capabilities of the *GPTeacher* application, we selected three primary categories of lectures based on their prevalence and essential nature in academic environments:

- **Theoretical Lectures:** These are primarily text-based, a domain where LLMs typically excel. However, they often contain overlapping information, making it challenging to answer questions due to the vast and sometimes indistinct information. **Example** - "Plato's Republic, I-II" by Professor Steven Smith at Yale.
- **Technical Lectures:** These have a more defined structure, which aids in comprehension. However, they frequently introduce unique terms, symbols, numerical data, and operations, which can be more challenging for LLMs to interpret accurately. **Example** - "Matrices Multiplication" by Professor Gilbert Strang at MIT.
- **Theoretical Technical Lectures:** This category combines the advantages of both theoretical and technical lectures. **Example** - "Deep Learning State of the Art (2020)" by Lex Fridman at MIT.

3. RETRIEVER & MODEL SELECTION

3.1. Context Window & Challenges

In order for our application to achieve its main functionality, which enables students to ask questions about a lecture and receive precise, accurate answers, it is crucial for the Large Language Model (LLM) to process the relevant parts of the lecture.

A seemingly straightforward solution to this challenge is to input the entire lecture transcript into the LLM. Recent advancements have significantly increased the context window of LLMs to accommodate up to 128k tokens, which is approximately 100,000 words. This capacity is more than sufficient to encompass an entire lecture. For instance, our test lectures indicate that 10 minutes of spoken content roughly equates to 3,000 tokens. Therefore, a lecture lasting 90 minutes, such as our longest lecture titled "Deep Learning State of the Art", comprises about 26,000 tokens according to the OpenAI tokenizer, fitting comfortably within the OpenAI's GPT-4 128K LLM's context window.

LLMs struggle to extract vital information from large inputs, especially when important data is in the middle [5]. This underscores the need for effective retriever strategies to ensure accurate information interpretation by LLMs.

3.2. Retriever vs Context Window

In order to determine the more effective strategy, we have developed two distinct question-answering LLM implementations. The first approach involves the LLM receiving the entire lecture transcript as input. In contrast, the second approach utilizes a more dynamic method. Depending on the nature of the question, an LLM determines a combination of context-aware retrievers and an SQL retriever to use. The context-aware retrievers are used for general inquiries,

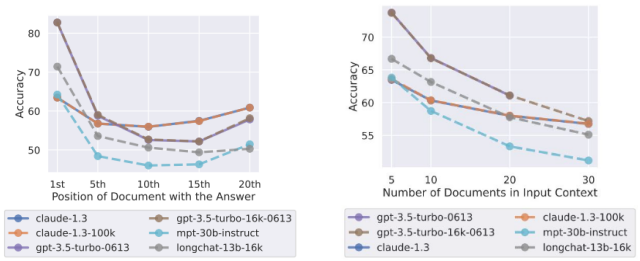


Fig. 1. How position of the right answer and different sizes of input context affect the model's accuracy [5].

while the SQL retriever specifically addresses timestamp or keyword-related questions. Once the relevant information is retrieved, another LLM is tasked with generating the answers. The model used for both cases is GPT-4 128k.

3.2.1. Retriever Selection Mechanism & Final Form

With the *Sections Retriever*, *Topics Retriever*, and *SQL Retriever* at our disposal(See 4.1.3, 4.1.4, 4.1.5), the next challenge is to determine the most effective combination of these for any given query. For this, we employ another LLM, which has been informed with details about each retriever's implementation and capabilities, as well as contextual information about the lecture, such as its title, duration, and overall content structure.

The LLM's role is to analyze a question and decide the most appropriate retriever(s) to use. If it opts for content-based retrievers (*Sections* and *Topics*), it specifies the number (K) of most relevant documents to be fetched from each. In cases where the *SQL Retriever* is also chosen, the LLM generates an SQL query to extract the pertinent information from the database.

For instance, consider the question: "What is the lottery ticket hypothesis and when was it discussed?" The LLM might output a decision like this:

```
{
  'retrievers': ['sections_retriever',
  'topics_retriever',
  'SQL_retriever'],
  'SQL_retriever': "SELECT text FROM table
  WHERE text LIKE
  '%lottery ticket hypothesis%'",
  'sections_retriever': 2,
  'topics_retriever': 3
}
```

This output indicates the combination of retrievers to be employed and the specific SQL query to be executed.

With the relevant parts of the lecture retrieved through the selected retrievers, we prune duplicate content, and then we provide them along with the student's question to a second LLM to get the final answer.

3.2.2. Questions & Evaluation

The above strategies will be evaluated using a set of questions, each with different requirements and predefined answers. The testing will be qualitative in nature. The questions are divided into categories based on their unique requirements and difficulty levels, with two questions allocated for each category. The categories are as follows:

- **Simple Questions** that can be easily searched and answered, testing the basic capability of the implementations.
- **Semantic Inference Questions**, that require to understand and infer information not explicitly stated in the lecture.
- **Synthesizing content Questions** that require synthesizing content from different smaller parts of the lecture.
- **Holistic Questions** that demand understanding of content from larger sections of the lecture.
- **Time-related Questions** which necessitate identifying and extracting information based on specific timestamps or chronological order within the lecture.

The results of the comparison of the two model on the above questions are highlighted in the table below

Question Category	Retrievers	Context Window
Simple	2/2	2/2
Semantic	2/2	2/2
Synthesizing	1/2	1/2
Holistic	1/2	2/2
Time	1/2	2/2
Total	7/10	9/10

Table 1. Results of how many questions for each category have been successfully answered from the two different LLM Q&A implementations

The Context Window implementation exhibited a higher overall success rate, correctly answering 9 out of 10 questions. In contrast, the Retrievers model achieved a 7 out of 10. It is important to note that the incorrect answers from the Retrievers model were not entirely wrong; rather, they sometimes lacked completeness or specific details. The Retrievers model successfully answered a question under 'Synthesizing content' that the Context Window model failed. The Context Window model lacked many necessary details, while the Retrievers model effectively integrated information from various parts of the lecture.

4. FEATURES

4.1. Question Answering

This is the main feature of our application, where the student can ask questions about the lecture and get answers.

4.1.1. Details

For this, we first employed what is referred to by most LLM frameworks as an *Agent*. This is the idea of having a language model prompt itself and choose between a set of options. In particular we employed a zero-shot *ReAct* (Reason + Act) [6] agent. This agent is presented with a question, a set of tools and their descriptions. Then, through a series of thinking, acting and observation steps, they are able to arrive at an answer. These frameworks can offer powerful capabilities where the LLM can try alternative solutions if one it's choices did not retrieve the correct answer. In section 3.2 we present a more stripped-down version of this approach. Essentially we have a ReAct agent that goes through a single loop of the Think-Act-Observe cycle.

For retrieval, we used two different databases, a vector store (FAISS) and a relational database (SQLite). The LLM, when asked questions that corresponds to concepts taught inside the lecture such as "How does Matrix multiplication work?" is supposed to retrieve relevant documents from the vector store based on their L2 Distance with the question's embedded vector. On the other hand, for questions that refer to specific time-points of the lecture such as "What did the professor talk about during the last 5 minutes?", the model is supposed to retrieve the corresponding lecture parts from the SQL database since the time-point information has been lost during the embedding process.

4.1.2. Embeddings

For calculating the embedding vector in order to create the vector database, we used two embedding models: *text-embedding-ada-002* (from OpenAI) and *all-MiniLM-L6-v2* (based on BERT). These are embeddings trained on natural language and worked very well for answer retrieval. After testing our system with both embedding models, we found results to be motivating but often not all relevant documents would be returned, where document is a section of the lecture contents after they have been split into smaller parts. For this reason, we developed an LLM assisted text-splitting algorithm presented in sections 4.1.3 and 4.1.4

4.1.3. Sections Retriever

The primary challenge in designing an effective retriever lies in balancing detail and comprehensiveness. A straightforward approach might involve dividing the lecture text into predefined chunks. However, large chunks could overlook finer details, while small chunks might miss the overarching context. Additionally, there's a risk that segments of the lecture discussing the same topic could be fragmented across multiple chunks.

To address these issues, we introduce the *Sections Retriever*, a context-aware splitting mechanism. The lecture text is initially segmented into chunks of approximately 4000 tokens. Each chunk is then processed by a specialized LLM, which further divides it into coherent sections. This approach ensures that each section is a self-contained unit of meaning, reducing the likelihood of thematic fragmentation.

To ensure continuity and context retention, the final section of a chunk is not treated as a standalone segment. Instead, its content is concatenated to the beginning of the subsequent chunk. This strategy effectively mitigates the risk of losing crucial contextual information at the boundaries of each chunk.

4.1.4. Topics Retriever

While the *Sections Retriever* effectively segments the lecture into meaningful sections, these sections can still encompass various subtopics. To further refine the retrieval process, we introduce the *Topics Retriever*. This component aims to break down each section into more specific topics.

An LLM is employed again for this task, analyzing each section to identify and isolate distinct topics within it. This process results in dividing the lecture text into smaller, topic-specific segments. By achieving this, the *Topics Retriever* enhances the specificity of our search capability. This means that when a question is asked, the retriever can now pinpoint more precisely relevant information within the lecture content, when that retriever is used.

4.1.5. SQL Retriever

Alongside the content-based retrievers, the *SQL Retriever* offers a structured approach to querying the lecture transcript based on timestamps and words. The database consists of three main columns. The first column contains segments of the lecture text. The second, labeled 'From', records the timestamp marking the start of each text segment. The third column, 'To', notes the timestamp when the segment ends. This structured format allows for efficient SQL queries, enabling the retrieval of information based on specific timeframes or keyword searches.

By integrating the *SQL Retriever*, we enhance the system's ability to handle queries that are specifically time or word oriented, complementing the content-based retrievers.

4.1.6. Evaluation

We wanted to see how plausible it would be to use a locally-runable model as the "backend" of our application. We tested our application's question answering pipeline using three

Topic	LLaMa	Mistral	ChatGPT 3.5
Plato's Republic	7.0	7.0	7.0
Deep Learning State of the Art	8.0	7.0	5.0
Matrix Multiplication	6.0	8.0	3.5
Total	21.0	22.0	15.5

Table 2. Results comparing the base question-answering feature of our app using two locally-runnable models and one larger model.

models: LLaMA (13B parameters), Mistral (7.3B parameters) and ChatGPT 3.5 (model `turbo-1106`, undisclosed number of parameters, close to 175B) For the local models, we set up the inference pipeline using *Ollama*[5], a wrapper around *llama.cpp*[6].

The experiment setup is as follows: we give the three LLMs the same exact question. Inside the prompt, we provide them with $k = 5$ relevant documents that are retrieved from the vector store. We asked each model 10 questions for each lecture. Then, we employed LLM-Guided Evaluation [7]. We take the answers and give them as input together with the question to *ChatGPT 4*, asking it to rate each of the answers as "Good" $(+1)$, "Medium" $(+0.5)$ or "Bad" $(+0)$. The results are in favor of the local models. We found this to be surprising, especially considering that the ChatGPT 3.5 is a lot larger that the other two candidates. One theory is that the company behind ChatGPT "weakens" temporarily some of it's services. There have been reports of model performance degregading [8]. In any case, after manually inspecting the LLMs outputs, we saw that the local models could provide high quality answers to a variety of questions for each lecture.

4.2. Student Adaptation

In this project we wanted to experiment with ways that using an LLM with could adapt the learning experience to specific student needs.

One way to do this, is by detecting the student's current emotional state and modifying the form of the LLM's answers to that. For example, if by the wording of a student's question we realize that they are starting to feel frustrated, we can adapt our answers so that they will positively reinforce the student.

LLMs, though not fully explored for Sentiment Analysis (SA), perform well in various SA tasks [9]. They exhibit a "theory of mind" in larger models, understanding the conversational partner's mental state. However, for smaller models, explicit instructions improve response humanization, aiding in future development of a comprehensive emotional map of the student's learning journey.

In our approach, the model first detects the student's sentiment from their question, outputting a single word like "Happy" or "Frustrated". It then adjusts its response based on this detected emotion. Initially, given prompts asking the model to differentiate only between "Neutral" and "Frustrated" proved ineffective, as the model mostly identified questions as "Neutral". However, allowing the model to freely identify any emotion significantly improved results.

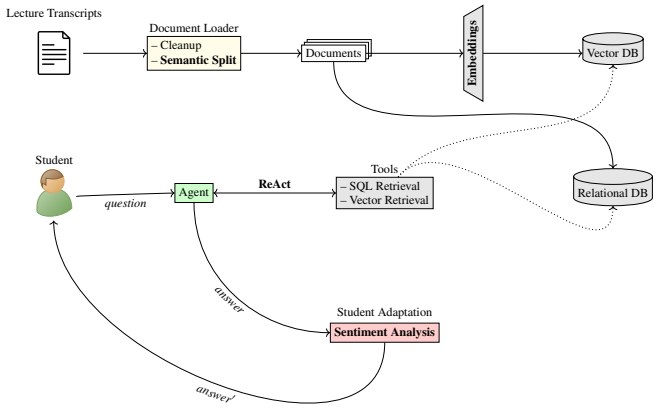


Fig. 2. Overview of the question-answering system together with the student adaptation pipeline. We **note** the parts of our system where we employed LLMs.

```
User: I really don't get why Plato's work is
    important.

Agent: I understand that Plato's significance
    might seem unclear at first. Let's look at it
    this way: Plato was...
```

Listing 1. Example of a modified answer. The sentiment detected is: Confused

4.3. Test Creation

For tests generation, the teacher gives a set of learning objectives for the course as a list of strings. Then, our retriever returns a set of relevant documents based on the learning objectives. Finally, these documents are given to the LLM with a prompt instructing it to create multiple choice questions. Since we need to create actual tests in our application's frontend we need the output to follow a particular structure. For that, we used *Langchain*'s `ResponseSchema` feature. This is a developer-friendly way of specifying to the LLM the format the output needs to be in which can also take advantage of *Python*'s type annotations. Through this, we can automatically generate the necessary natural-language instructions for the model.

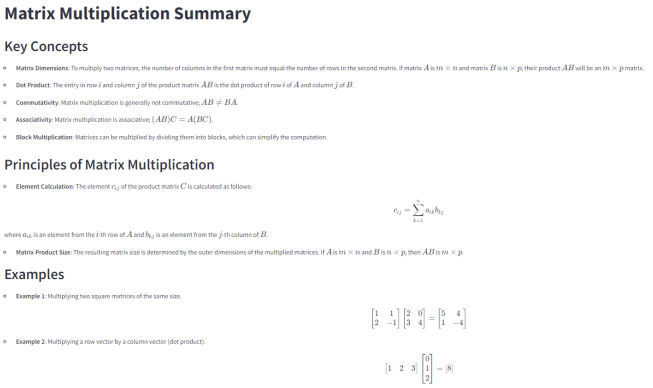
Some alternative approaches for controlling the LLM's

[5] `https://ollama.ai/`
[6] `https://github.com/ggerganov/llama.cpp`

output exist such as OpenAI functions [10] and Guidance [11]. These technologies work by getting the LLM's token probability output (usually the log probabilities of the next possible token) and tapping into the tokens sampling routine.

```
[
  {
    "question": "When was Plato born?",
    "options": [
      "427 BC",
      "407 BC",
      "387 BC",
      "367 BC"
    ],
    "answer": 0
  },
  ...
```

Listing 2. An example of the JSON output returned by the model.



Matrix Multiplication Summary

Key Concepts

- **Matrix Dimensions**: To multiply two matrices, the number of columns in the first matrix must equal the number of rows in the second matrix. If matrix A is $m \times n$ and matrix B is $n \times p$, their product AB will be an $m \times p$ matrix.
- **Dot Product**: The entry in row i and column j of the product matrix AB is the dot product of row i of A and column j of B.
- **Commutativity**: Matrix multiplication is generally not commutative, $AB \neq BA$.
- **Associativity**: Matrix multiplication is associative; $(AB)C = A(BC)$.
- **Block Multiplication**: Matrices can be multiplied by dividing them into blocks, which can simplify the computation.

Principles of Matrix Multiplication

- **Element Calculation**: The element c_{ij} of the product matrix C is calculated as follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

where a_{ik} is an element from the i-th row of A and b_{kj} is an element from the j-th column of B.

- **Matrix Product Size**: The resulting matrix size is determined by the outer dimensions of the multiplied matrices. If A is $m \times n$ and B is $n \times p$, then AB is $m \times p$.

Examples

- **Example 1**: Multiplying two square matrices of the same size.

$$\begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 1 & -4 \end{bmatrix}$$

- **Example 2**: Multiplying a row vector by a column vector (dot product).

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} = [8]$$

Fig. 3. Notes Summary Feature in the app.

4.4. Notes Summary

Recognizing the value for students to have concise summaries of lectures, we aimed to create this feature using the lecture notes or associated book sections/pages. This is particularly beneficial for complex subjects with visual elements, such as the "Technical lecture on Matrices Multiplication," which includes numerous mathematical equations and matrix operations. We manually created sections to gain more control over the summarizing process. Each section of the lecture was then associated with relevant pages from the textbook, provided in PNG format. To process these images and generate summaries, we utilized the GPT-4-1106-Vision model from OpenAI, capable of understanding visual information. The model was tasked with creating a summary for each section based on the given images. We specifically instructed it to output the summaries in both LaTeX and Markdown formats. The result was a successful generation of concise, informative summaries for each section.

5. CONCLUSIONS

LLMs allow for creating many small specialized APIs, speeding up significantly the software development process. During development, it's important to pinpoint the parts of the application where the LLM works better given explicit instructions rather than allowing it to figure out what to do. This balance between explicit instructions on response format and allowing the model to be less restricted is something that we found to be very important to developing our LLM application.

Specific to education, text generation models can be very powerful because of the inherent text-based nature of learning material. Future research directions should include in developing systems that further adapt to how the student feels and learns.

6. REFERENCES

[1] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian, "A comprehensive overview of large language models," 2023.

[2] OpenAI, "Gpt-4 technical report," 2023.

[3] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample, "Llama: Open and efficient foundation language models," 2023.

[4] Waleed Kadous and Kourosh Hakhamaneshi, "Fine tuning is for form, not facts," *Anyscale Blog*, July 2023.

[5] John Hewitt Ashwin Paranjape Michele Bevilacqua Fabio Petroni Percy Liang Nelson F. Liu, Kevin Lin, "Lost in the middle: How language models use long contexts," 2023.

[6] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao, "React: Synergizing reasoning and acting in language models," 2023.

[7] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica, "Judging llm-as-a-judge with mt-bench and chatbot arena," 2023.

[8] Benj Edwards, "Is chatgpt getting worse over time? study claims yes, but others aren't sure," July 2023.

[9] Wenxuan Zhang, Yue Deng, Bing Liu, Sinno Jialin Pan, and Lidong Bing, "Sentiment analysis in the era of large language models: A reality check," 2023.

[10] Atty Eleti, Jeff Harris, and Logan Kilpatrick, "Function calling and other api updates," *OpenAI Blog*, June 2023.

[11] Guidance-AI contributors, "Guidance: A language for controlling large language models," `https://github.com/guidance-ai/guidance`, 2023, Accessed: 2023-12-19.

A. SOURCE CODE

Source code for the developed application as well as a Jupyter Notebook with results from our project can be found **here**.

B. LLM-ASSISTED TEXT SPLITTING

Here, we present the pseudo-code for the section and topic splitting algorithm we developed. This algorithm utilizes an LLM in order to split a text document into semantically sounds pieces, rather than using hard-coded metrics like word/character/token count. This approach can greatly improve retrieval augmented generation, as more to-the-point pieces of text can be retrieved for question answering.

procedure EXTRACTSECTIONS($lecture\_str, tokens$)
    Determine segment size based on input tokens
    Initialize variables for segment tracking
    Prepare a list for storing identified sections
    while more segments to process in the transcript **do**
        Extract and process current segment
        Analyze segment to find logical sections by passing it to the LLM
        Update list with identified sections
        Adjust segment tracking for next iteration
    end while
    return All identified sections from the transcript
end procedure

procedure EXTRACTTOPICS($lecture\_str, section\_list$)
    Prepare a structure to store topics by section
    for each defined section **do**
        Analyze section to identify sub-topics by passing it to the LLM
        Record topics with their context within the section
    end for
    return Detailed mapping of topics within each section
end procedure

Fig. 4. The section and topic extracting algorithms