COLORADO SCHOOL OF MINES
EARTH • ENERGY • ENVIRONMENT

CSCI 370 Final Report

HiLabs Group 2

Andrew Huycke
Levi Sprung
Ben Tran
James Watts

Revised December 10, 2023

HiLabs™

CSCI 370 Fall 2023

Dr. Painter-Wakefield

Table 1: Revision history

Revision	Date	Comments
Rough Draft	12/5/2023	Created a rough draft of the document so that we could complete our peer review.
Final Revisions	12/10/2023	Revised our rough draft in accordance with the feedback we received in the peer review.

Table of Contents

I. Introduction.......... 3
II. Functional Requirements.......... 3
III. Non-Functional Requirements.......... 4
IV. Risks.......... 4
V. Definition of Done.......... 4
VI. System Architecture.......... 5
VII. Software Test and Quality.......... 11
VIII. Project Ethical Considerations.......... 13
IX. Project Completion Status.......... 14
X. Future Work.......... 15
XI. Lessons Learned.......... 16
XII. Acknowledgments.......... 17
XIII. Team Profile.......... 17
References.......... 19
Appendix A – Key Terms.......... 20
Appendix B – Program Dependencies.......... 20

I. Introduction

Our client, HiLabs, is committed to leveraging AI, machine learning, and domain knowledge to provide clean data to members in the healthcare industry [1]. While this is their primary mission, they also are looking into delivering innovative solutions in other sectors, one of which is the education sector. More specifically, HiLabs is looking to create a virtual TA that can assist college students in their coursework. With the emergence of LLMs in the past year, the ability for AI to improve and modernize education has only grown, and HiLabs is looking to take steps to facilitate this improvement. During this semester, our group was tasked with researching the feasibility of an AI teaching assistant.

Over the course of the field session term, our group was tasked with implementing a chatbot that can interact with a student and identify areas of strength and weakness within a course. Accomplishing this goal required several steps, the first of which was to gather course materials. After gathering these materials, our group was tasked with parsing them using both LLMs and natural language processing. From here, we could leverage an LLM to assess student knowledge.

II. Functional Requirements

Course Knowledge Graph

- Knowledge graph must contain information taught in the course and encapsulate the relationships between topics
- Start with one course's materials
- Directed graph with labeled nodes and labeled edges
- Knowledge graph creation:
  - Read in pdf, Word document, or PowerPoint presentation files
    - Convert other file types to supported file type
  - Use Large Language Model (LLM) to extract relationships in the data
  - Represent information in knowledge graph
  - Output knowledge graph to .gml file
- Visualize knowledge graph
  - Use Python library to load in .gml file and visualize represented graph

Chatbot for Knowledge Assessment

- Use LangChain to implement chatbot to assess student knowledge using Llama model
  - Ask a series of questions to gauge knowledge on various topics

        - Start broad, and get more specific if student demonstrates some knowledge in the given topic
            - If a student doesn't demonstrate knowledge in a broad topic, no need to ask specific questions about that topic
    - Color triplets in the knowledge graph based on student's responses
- Visualize resulting colored knowledge graph
    - Use aforementioned Python library to load and visualize .gml file
    - Goal is to easily view which topics a student is and isn't proficient in

III. Non-Functional Requirements

- The course knowledge graph and corresponding questions should accurately represent the materials that are uploaded.
- Each graph should be stored in a .gml file.
- Project will be contained in a centralized repository, preferably on GitHub.

IV. Risks

The largest risk that our group identified is that the outputs of the LLM are nondeterministic, meaning that there is some random element at play. Since we are creating something that should assess student performance, the negative ramifications of this could be considerable. It is important to ensure that each student is evaluated as objectively as possible. As such, our group incorporated significant unit testing so that we could get an understanding of how confident we should be in the LLM output.

Another risk that our group considered is related to the scalability of our project. Due to hardware limitations, our group is not able to test our program on super large inputs. As such, it is impossible to know for certain that every part of the project will function correctly when scaling up to larger inputs. With this being said, we determined in client meetings that we should be focused on creating a proof of concept for an AI teaching assistant. As such, testing on smaller inputs suffices for this project.

V. Definition of Done

In order to create a minimum viable product, there is one overarching requirement that our group needed to satisfy. This requirement was to create a functional chatbot that can assess student knowledge on any course, provided that adequate materials are available for that course. To achieve this, our group needed to automate the

process of parsing materials, generating questions, and validating human responses. An additional requirement that our group had to satisfy was uploading all relevant files to a centralized repository on GitHub.

VI. System Architecture

Our system architecture consists of several components. Starting out, our program requires a pdf input, which contains all of the course materials. At this stage, the program is storing a large amount of text, so it is paramount to store this information in a compact data structure. We achieved this by creating a knowledge graph. A knowledge graph, sometimes referred to as “kg”, is a directed labeled graph in which there are associated domain specific meanings with nodes and edges [2]. To be more exact, edges represent relationships between nodes. In our problem, nodes correspond to certain topics, and edges relate those topics together. From the knowledge graph, we generate questions to ask a human, and after recording all of the human responses, we determine whether those responses are correct or incorrect. After all of this is complete, we display the results to the user, showing which topics they did and did not understand. All of this functionality sits on top of a graphical user interface to allow a seamless user experience. A diagram that encapsulates our system architecture can be seen below.

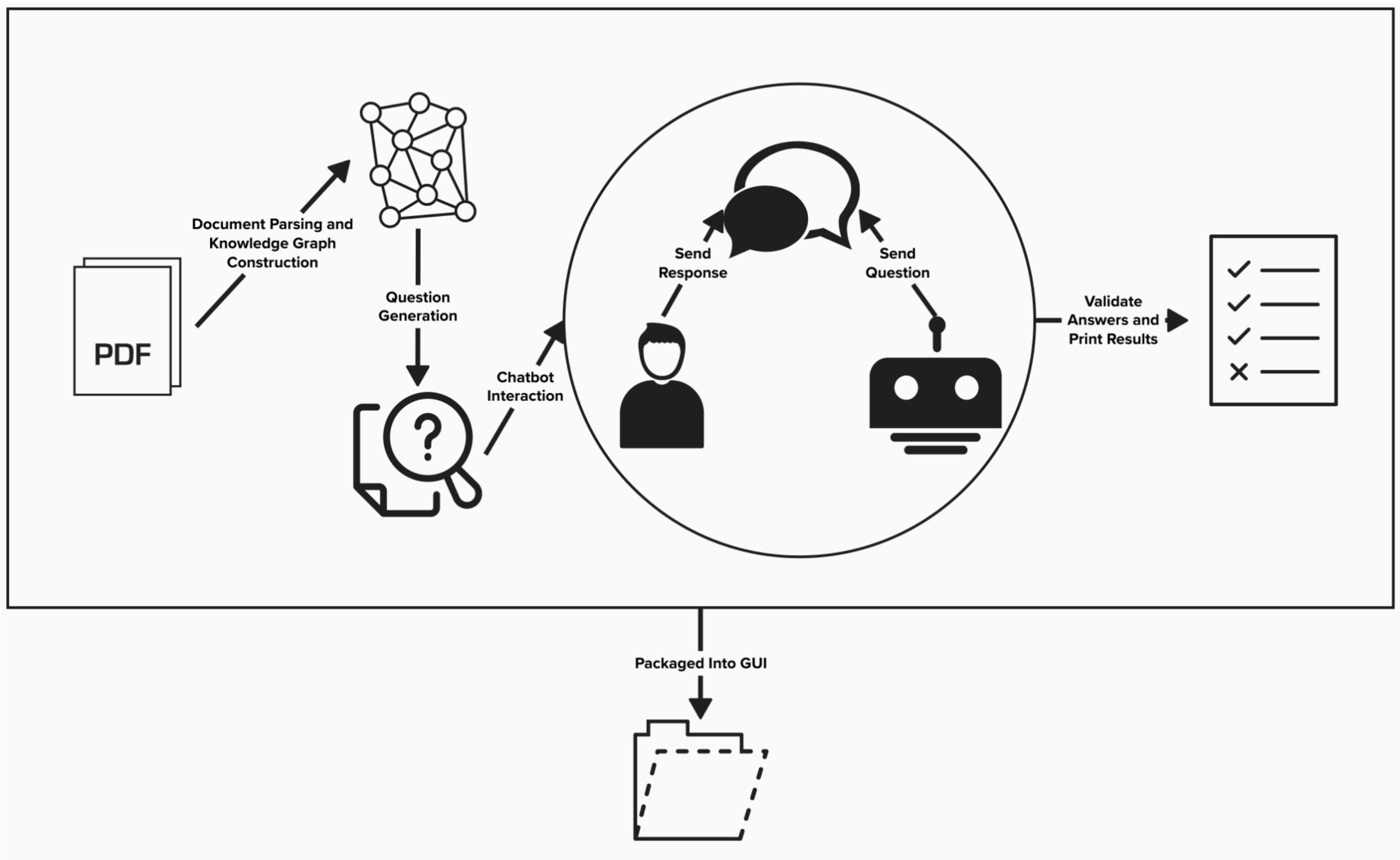

Figure 1: A high level representation of our system architecture

While this diagram gives a solid idea of how the different components in our system interface with each other, it is still a very high level overview of our project. As such, the different components are explained in more detail below.

Gathering Course Materials

The first step of the process is gathering course materials. Currently, the project only has support for pdf files, so other file formats, such as Word documents or PowerPoints, would need to be converted to pdf. Initially, the project converted the pdfs to plain text files, but it made more sense to use the pdf loader that LangChain provided instead.

Parsing Documents and Creating a Knowledge Graph

Once the pdfs are gathered, the next steps are all handled within LangChain. First, LangChain's PyPDFLoader loads in the text from the pdf files. This text can then be used as input for an LLM prompt.

The next step is to create the knowledge graph. We used LangChain's ConversationKGMemory class to help with this. This class has a kg (knowledge graph) attribute which, in addition to the class methods, are helpful for constructing a knowledge graph. Our group created a knowledge extraction prompt, fed that into the memory class, and ran the prompt for each slide. The LLM output is then parsed and written to the memory's kg attribute, which is an instance of a wrapper class for the NetworkX DiGraph class (models a directed graph). This is all to say that the LLM output is written to a place that we can access later in our system architecture pipeline. After the knowledge graph is created, it is written to a .gml file.

Once the knowledge graph is made, we deemed some additional preprocessing necessary. This is because some nodes exist that are synonymous with each other, but are not exactly the same, and they should instead just be one node. In order to do this, we implemented a function which would read in all the node names for a .gml file, capitalize all proper nouns and ensure all other words are lowercase using the spaCy library (Python library for natural language processing), and then find each pair of adequately similar words and combine them into one node. The result is written to another .gml file, or the same one that was used as the input.

Question Generation and Answer Validation

The input to this step of the project is a knowledge graph in .gml format. From here, we had to find a way to generate questions to ask. In order to do this, our group first created a knowledge graph class to make dealing with the input easier. This class inherited the NetworkX.DiGraph() class, and it is able to read in a .gml

file and create a knowledge graph from it. Additionally the class is able to find all weakly connected components in our graph, which is vital.

As alluded to above, the approach to generating questions that our group used is to inject information from weakly connected components into a prompt. To provide some background, a weakly connected component is a subgraph for which there is a path connecting every node, ignoring edge direction. The intuition behind this is that in our knowledge graph, the edges relate topics to each other. By finding weakly connected components, we are hoping to extract nodes that are inherently related. Thus, instead of having to ask questions about every single topic in our graph, we can ask questions about groups of topics to quickly gauge student understanding.

After a question has been displayed to the user of our program, they will eventually provide an answer that must be evaluated. To do this, our group once again makes use of LLM prompting. In this step, the information we inject into the prompt are the question that was asked and the human response. In the prompt, we instruct the LLM to output an answer between 0 and 1, where 0 indicates no understanding of the topics and 1 indicates complete understanding. This is done because the evaluation of answers is inherently subjective, and our group wanted to have a way to assess the confidence level of the LLM. A picture of a prompt that our group engineered for the Llama 70B model can be seen below. In the prompt, we feed the LLM information about the question that was asked and the respective response so that it can make a decision.

```
1  answer_validation_template = """
2  [INST] <<SYS>>
3  You are a helpful, respectful and honest assistant. Always answer as helpfully as possible, while being safe.
4  Your answers should not include any harmful, unethical, racist, sexist, toxic, dangerous, or illegal content.
5  Please ensure that your responses are socially unbiased and positive in nature.
6  If a question does not make any sense, or is not factually coherent, explain why instead of answering something not correct.
7  If you don't know the answer to a question, please don't share false information.
8  <</SYS>>
9  You are a helpful assistand that will validate whether or not a student understands a topic based on their response to a question.
10 Respond with a float between 0 and 1, where '0' indicates no understanding and '1' signifies complete understanding.
11 Keep your response concise and do not use any words.
12
13 The question that the student was asked is: {question}
14
15 The student's response to the question is: {student_response}
16 [/INST]
17 """
18
19 prompt = PromptTemplate(template=answer_validation_template, input_variables=["question","student_response"])
```

Figure 2: Picture of an answer validation prompt that our group used

Admittedly, there are some potential concerns that our team encountered when generating questions via weakly connected components. On certain knowledge graph inputs, our team noticed that there would be one or two weakly connected components that contained a lot of nodes while the other components were sparse. This

can cause issues when scaling our program up, as it is incredibly difficult to evaluate knowledge of these components with one question.

Chatbot Feedback

The chatbot feedback for our program occurs in two phases: after each individual question, and at the end once all questions have been answered. After a response is provided to each individual question, the program tells the participant whether or not they answered correctly. This response is based on the result of the answer validation prompt; if the result is above 0.5, then the question is marked as correct, and it is marked incorrect otherwise. Then, at the end of the chatbot interaction, our program displays a list of topics that the student is proficient and struggling with. These topics are simply the nodes in each weakly connected component.

Graphical User Interface

Our Graphical User Interface (henceforth referred to as GUI) is our program's capability to display the chatbot interactions in real time as they occur, similar to ChatGPT. The message generated by the LLM is displayed in the chatlog, and the student types in the answer to be evaluated. It is a fairly simple user interface, however it abstracts away a lot of the work that our code is doing in the background, allowing information such as the knowledge graph generation and response evaluation to be hidden. This was done through both the tkinter library in Python and through the command line in the example shown in our presentation. We were not able to completely synthesize the project to have a working user interface, as our client was not planning on using it regardless, but our prototype is shown below.

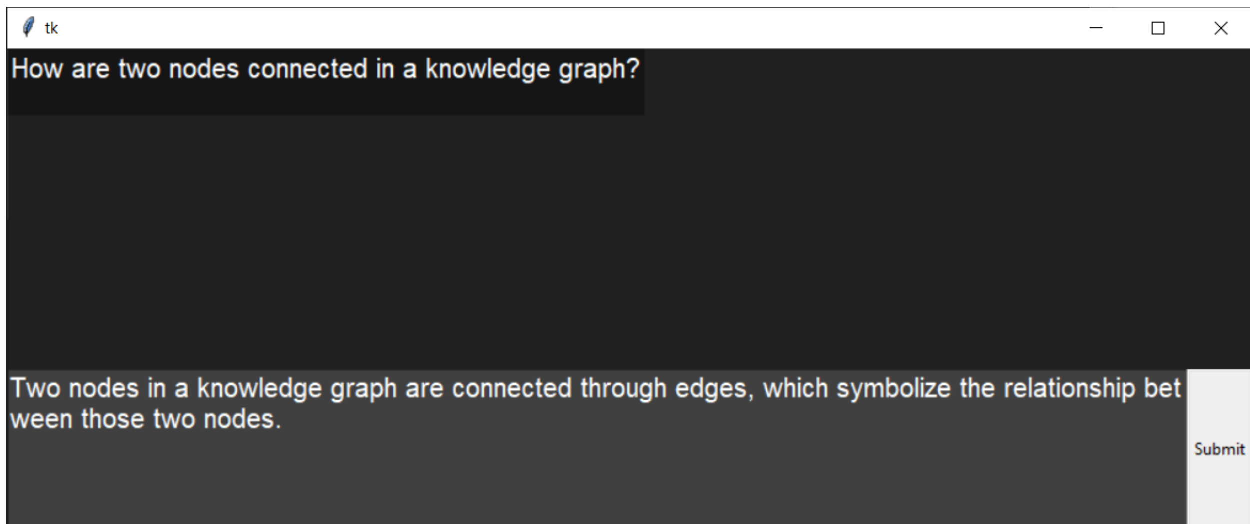

Figure 3: Example of our GUI for the chatbot

Remote Operation

The GUI also programmatically implements an SSH connection to the compute cluster at mines to offload the heavy lifting of the model to the higher performance computing devices. To implement the connection, we made use of the paramiko library. On top of establishing an ssh connection, it is able to do standard I/O operations as if it was a user.

```
<paramiko.ChannelFile from <paramiko.Channel 0 (open) window=2097152 -> <paramiko.Transport at 0xf28ae410 (cipher aes128-ctr, 128 bits) (active; 1 open channel(s))>>
bins
CSCI370
ondemand
scratch
```

Figure 4: A sample output from calling "ls" using a paramiko SSH connection.

Technical Design Aspects

One interesting part of the project was integrating the knowledge graph generation and question generation processes. In order to keep our codebase as organized as possible, we decided to create a knowledge graph class in order to read in .gml files, populate a graph, extract weakly connected components, and visualize the knowledge graph. A UML diagram of this class can be seen below.

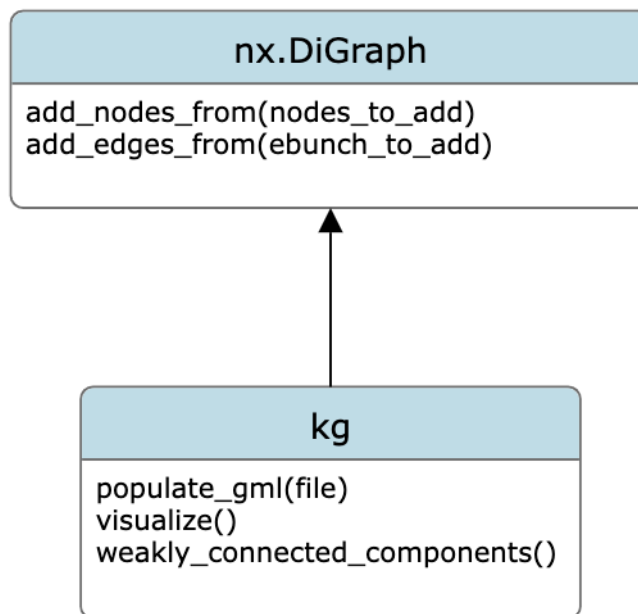

Figure 5: A UML diagram knowledge graph class, named "kg", which inherits the nx.DiGraph class.

As well, the design of the knowledge graph generation is worth detailing. In this step, we did not employ object-oriented programming, but rather just wrote a lot of functions for each part of the process. The first function is import_pdf, which just loads a pdf and returns the text contents, followed by create_llm, which creates the LLM object used for the knowledge graph generation prompt, and then update_kg_and_results,

which is essentially an additional function for the ConversationKGMemory class which updates the knowledge graph given the input text and returns the updated nodes and edges. The get_kg_triplets function runs this and returns it as a list of tuples, which was included in case we needed to do something with the LLM output. slides_to_kg is the function that does the most dirty work, so to speak, as it runs get_kg_triplets for each slide in the inputted pdf and returns the resulting NetworkXEntityGraph object. kg_to_gml then writes this to a .gml file. pdf_to_kg combines all the aforementioned steps to convert a pdf to a knowledge graph, given paths for the pdf, resulting knowledge graph, and model, and, optionally, the start and end indices for the pdf slides. These functions are all located in pdf_to_kg.py.

Node label regularization is handled by four functions in node_processing.py. The first is process_text, which uses spaCy to ensure correct capitalization of proper nouns and other words, and also replaces underscores with spaces. A helper function entitled similar was used to find the similarity ratio between two words using the SequenceMatcher class from the difflib library, which is in the Python standard library. This helper function is then used in the combine_similar function, which looks at each pair of node names and combines them into one node if the similarity ratio is above a certain threshold, which is 0.7 by default. It then returns a list of the new node labels at the indices of the old labels. The process_gml_nodes function runs combine_similar on the inputted .gml file, and then builds a new graph by adding edges based on the new node names, which will only create new nodes if a node name does not exist. This resulting graph is written to a .gml file as well.

Lastly, the usage of .gml files will be detailed. GML stands for graph modeling language, and is aptly a file format used for graphs. It uses square brackets to organize graph, node, and edge information. An example .gml file is shown below, with intermediate info replaced with an ellipsis. Our group chose to use .gml files because they seamlessly integrate knowledge graph construction, visualization, and use, since they are supported by the graph libraries we used, in addition to most other libraries that could potentially be used in the future.

```
graph [
  directed 1
  node [
    id 0
    label "data type"
  ]
  node [
    id 1
    label "basic type"
  ]

...

  edge [
    source 17
    target 0
    relation "is a"
  ]
  edge [
    source 18
    target 0
    relation "is a"
  ]
]
```

Figure 6: Example of a knowledge graph stored as a .gml file

VII. Software Test and Quality

Our initial quality assurance plan consisted of a few steps in order to ensure that we managed to deliver a quality product to our client. For our project, we decided that we must use unit testing, interface/output testing, and adequate code metrics to deliver the best product possible.

Unit Testing

Our unit testing consisted of a couple of phrases to use as input to the LLM. These phrases consisted of simple phrases whose interpretations could be easily falsifiable if our model was doing a poor job. Examples of these phrases are "Alice is friends with Bob, who is an electrician." and "Alan Turing was born on June 23, 1912 in London, England.". These simple statements provide variety as well as simplicity in order for us to see if our code is incorrect in a fundamental way.

In our code, we used these statements at two key steps in order to validate our output. The first was the output of the knowledge graph generation. We passed these statements verbatim to the LLM, and would test to see if their output was consistent with our expectations. Looking at the outputs in node-edge-node format quickly determined if what we were doing was correct. The other was question generation. As we put in the expected knowledge graph outputs into the LLM, we were able to generate questions that would test the users'

knowledge of these steps. These were fairly falsifiable as well, though they required a knowledge of all the topics involved, hence why the statements consisted of simple, easily memorizable facts.

Interface/Output Testing

Since our output is minimal, there was not a whole lot of need for testing the UI. However, getting actual results that were able to be visually displayed to the client were extremely important as we kept up our weekly meetings. Typically, these would be in the form of the actual LLM output given to us, but other times we had to succinctly present a large amount of data. For this, we used visualization libraries such as matplotlib to communicate our ideas to Dr. Stolman, our client. An example of how we visualized the accuracy of our data based on natural language processing comparisons is shown below.

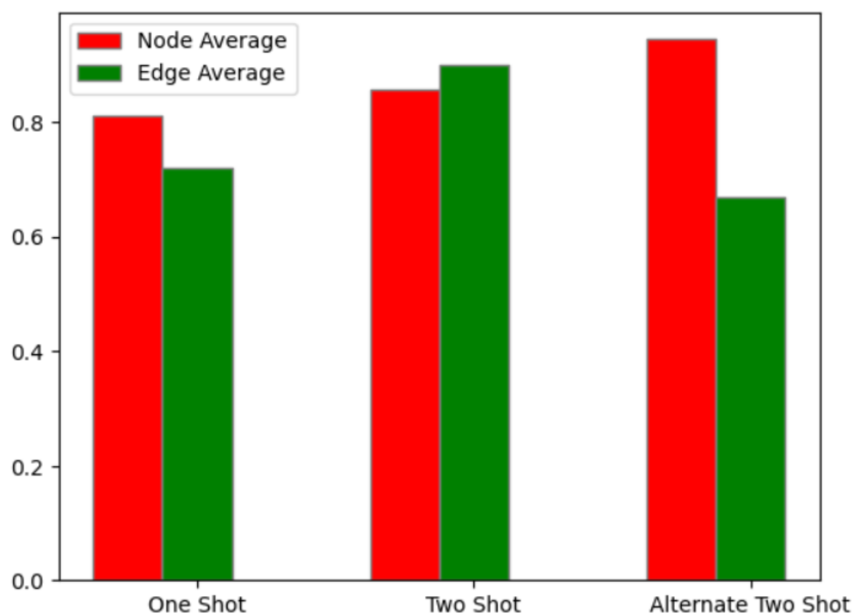

Figure 7: A graph generated to visualize accuracy among prompts

Code Metrics

Another point in our project where we incorporated unit testing was during the answer validation phase. It was necessary to use unit testing to ensure that our LLM prompts were working as expected. To test our answer validation prompts, we created sample questions and sample responses that were labeled as either correct or incorrect. In total, we created a total of 66 test cases. When our group tested these cases with our best performing Llama 70B model prompt, it obtained an accuracy of 0.79. The confusion matrix for these results can be seen below.

	precision	recall	f1-score	support
correct	0.77	0.82	0.79	33
incorrect	0.81	0.76	0.78	33
accuracy			0.79	66
macro avg	0.79	0.79	0.79	66
weighted avg	0.79	0.79	0.79	66

Figure 8: Confusion matrix for our best-performing answer validation prompt

This table also includes metrics like the precision, recall, and f1-score for each class. With this being said, our testing data was balanced, so the primary metric that our group tried to maximize is the overall accuracy.

One of the most important aspects of maintaining quality output with our code this semester was the response time of our LLM. A lot of this came down to one specific metric: input length vs. response time. Seeing as the end goal for our product is to eventually take in entire course loads of material, we needed to ensure that response times from our LLM were not unreasonable. This was mainly attended to by investing time into researching the various models available to us via huggingface's website, and seeing what we were able to work with in our various high-performance computing environments. This allowed us to achieve much faster speeds than if we had stuck to a fixed model throughout the entire project.

VIII. Project Ethical Considerations

One of the most complex issues in the development of our project was the consideration of ethics. Since AI and LLMs are at the forefront of new and developing technologies, we had to be careful in the ways in which we implemented our project. Alongside these considerations, we had to follow established ethical guidelines for engineers. With both these main focuses in mind, we came up with five pertinent ACM principles that were pertinent and unique to our overall project, and decided to handle them in ways we felt was in the best interest of all parties involved.

Principle 1.05. Cooperate in efforts to address matters of grave public concern caused by software, its installation, maintenance, support or documentation.

It is no secret that there is a large sector of the public concerned with the integration of AI into modern software. In terms of the type of AI that we used for the project, many have issues with biased datasets, producing inaccurate information, and the security of its user base. Our focus on alleviating these concerns was through good prompt engineering and being up front about how our software is used. The types of prompts we used asked the LLM to perform a specific task, and was rigorously tested to ensure that we were able to output the best response possible, without the need to have an LLM deem information as factually correct. In disclosing the process in which we created our model, as well as receiving consent from all of the parties involved in which we provided our data, we provided the security we needed in order to create our project.

Principle 2.01. Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.

We displayed a competency and level of expertise that helped HiLabs move forwards in their integration of AI into their new learning products. With our weekly meetings between us and our point of contact, we were able to discuss the issues we were having with an area of Computer Science that was mostly new to us all. With us being forthright about this information, we were able to learn and grow as engineers during this process. It also allowed our client to be aware of our limitations and assist us whenever we needed.

Principle 4.01. Temper all technical judgments by the need to support and maintain human values.

An important aspect of following ethical principles was ensuring that we kept the matters of public concern discussed in Principle 1.05 in mind as we continued to make important design decisions. In order to keep these in mind, we made sure that we consulted our references often throughout the project, ensuring that we were on the right track. Simply defining these goals out gave us a proper mindset to keep us on track to create a project as ethically mindful as possible.

IX. Project Completion Status

Our codebase has all functionality that is required for a minimal viable product, from parsing documents to identifying knowledge graph triplets to evaluating student knowledge. A picture of the chatbot-human interaction that our program produces can be seen below. This screenshot is from the terminal because our group has not fully integrated all functionality with our GUI.

```
(test) [ahuycke@g005 auto-kg]$ python3 src/knowledge-graph/13b_main.py
What is the difference between a local variable and an object member variable?
A local variable can be accessed in the scope that it was declared in while a member vari
able can only be accessed within a certain object
correct
```

Figure 9: Example of chatbot-human interaction for a single question

As can be seen in the figure, the question that the LLM generates is printed to the console. After this the user is prompted for input, and once this input is given, the LLM will validate that answer and respond with either “correct” or “incorrect”. After all of the questions are answered, a comprehensive list of topics that the student showed adequate understanding in and struggled in are printed to the console as well.

Outside of producing a minimum viable product, our group incorporated unit testing at each phase to ensure that prompts operate as intended. All of the code exists in a repository on GitHub, as our client requested, and relevant files are in the main branch. With this being said, some pieces of our code are not integrated as well as we had hoped for, such as the GUI and the rest of the program. Additionally, there are theoretically always improvements that can be made to our LLM prompts.

One concern that our group had towards the end of the project is the ability for our project to scale with regards to the size of input. In client meetings, we have been told not to worry about this too much, so we settled for optimizing our prompts on smaller inputs.

We were not able to dynamically generate questions for students. Instead, all questions are generated prior to being asked. As well, the user interface is barebones. Some aspects of the knowledge graph generation process also could be improved, as outputs are currently inconsistent and flawed.

X. Future Work

While our group achieved a minimum viable product, there is still a considerable amount of work that can be done to improve the product. To start out, our group ran out of time before we were able to fully integrate our program with the GUI. Completing this would greatly improve the user experience when using our product. Additionally, there are some computing optimizations that can be made to decrease the runtime of our program. When generating questions with the Llama 70B model, it would take minutes to generate questions from a medium-sized knowledge graph where it only took a few seconds for the Llama 13B model. This performance was seen on a Wendian compute node with 144 GB of memory and 18 cores per task. Our group isn’t entirely sure if this is an issue with computing resources or a lack of parallel processing between nodes, but either way, research should be done on this to ensure faster runtimes in the future.

XI. Lessons Learned

Throughout the semester, our group learned an incredible amount about large language models. Specifically, we learned how to use LangChain to leverage LLMs, we learned about the computational demands of running LLMs, and we learned that the randomness associated with LLMs carry a significant amount of risk. Outside of this, our group learned a great deal about tackling a large software project in general. A summary of our group's findings can be found below.

- We all gained general experience with Large Language Models (LLMs). Since we didn't have access to OpenAI's models, we used Meta's Llama model. Using this model, we had to deal with inconsistent outputs and difficulties presented by its limitations.
- Prompt engineering is a useful tool for generating LLM outputs for a specific purpose. Using multiple examples which may be similar to the types of inputs the LLM may be receiving is helpful. As well, specifying an output format is helpful for using the LLM output in code, but due to Llama's limitations string processing in addition to standard output parsing was required to account for inconsistencies.
- Our group learned the importance of prompt formatting, especially when using Llama. While models like ChatGPT may be smart enough to give consistent results without a rigid format, best results with Llama were obtained by using the exact prompting format that a specific model was trained on.
- We learned the concept of knowledge graphs and how it directly applies to our project. To expand, we discovered how to capture relationships with a knowledge graph, and how traversing a knowledge graph can reveal information that is related.
- We learned of Python's capabilities with its libraries. There is probably a library of most if not all applications to implement common tasks within that library. For example, we used networks for the manipulation of graphs. LangChain also assisted us in using LLMs which make up a significant portion of our application.
- We learned the importance of consistent communication with our client and within our team. Additionally, we discovered how important it is to demo our work in order to get constructive feedback.
- Some networking knowledge needed to be picked up when we started looking at running models in a distributed manner. In our case we were interested in running a model in a computing cluster while in a local GUI environment.
- We quickly discovered that some of the bigger LLM downloads require a ton of resources to run at a decent speed. Running these models locally was not an option.
- After communicating with HPC at Mines to get provisioned on our school's supercomputers, we learned how to allocate and monitor resources on a gpu node to ensure timely execution of different tasks.

XII. Acknowledgments

Firstly, we would like to thank our client, HiLabs, namely our client contact, Andrew Stolman. Andrew was invested in the success of our project and served as a helpful resource for our team, answering our questions and providing suggestions and feedback as needed to help guide our team forward. We also appreciate the flexibility HiLabs gave us by allowing us to focus primarily on the internal functionality of our project.

We would also like to thank our technical advisor, Christopher Painter-Wakefield. He helped our team stay on track, gave us valuable feedback on our presentations, and courteously allowed us to use his past course materials on our project.

Lastly, we would like to thank everybody at High Performance Computing with Mines. Particularly, we would like to express our gratitude towards Nicholas Danes and Richard Gilmore for provisioning us on HPC environments and helping us navigate them. Without their help, it would have been impossible for us to have a consistent environment to run a LLM in.

XIII. Team Profile

Name: Andrew Huycke

Bio: Hello, my name is Andrew Huycke and I am currently a senior in computer science with a specialization in data science. A few of the things I am involved in on campus are club volleyball, Sigma Phi Epsilon (Greek Life), as well as being a teaching assistant for CSCI 128. Outside of these activities, I enjoy skiing, playing basketball, and working through interesting math problems.

Team Role: Throughout the semester, I was the client liaison, so I was primarily responsible for communicating with our client outreach, Andrew Stolman. Some of my technical contributions to the project include building

the framework for LLM question generation and answer validation, as well as creating a class for knowledge graphs to facilitate integration with my teammates' work.

Name: James Watts

Bio: My name is James Watts and I am currently a junior in computer science with a focus area in computer engineering. I am involved with a couple of the arts around campus. I am a member of the school's jazz band as well as on the leadership team for our school's movie club. I enjoy listening to and analyzing music, playing guitar and piano, and going on hikes.

Team Role: I served as the primary contact with our advisor, Christopher Painter-Wakefield, and would help organize meetings and keep things organized for our team. In terms of code contributions, I served as a tester for a lot of my teammates' code as well as contributed to the GUI and integration steps in our overall project flow.

Name: Benjamin Tran

Bio: My name is Benjamin Tran and I am currently a senior in computer science on the general track. I am a member of the game development club on campus. I enjoy working on personal software projects along with playing the piano and building circuits.

Team Role: I primarily performed unit testing during the earlier stages of the project. However, I moved to develop a user interface for proof of concept purposes. More specifically, I did research into different Python libraries for implementing simple UIs. Additionally, I established a connection between the user interface program to a remote compute cluster in order to off load the running of LLMs to said compute cluster while keeping the front end running locally.

Name: Levi Sprung

Bio: My name is Levi and I am a junior in computer science working towards a math minor. On campus, I am primarily involved in the Competitive Programming Club and am going to be a teaching assistant for Algorithms next semester. Some other interests of mine include skiing, singing, rock climbing, chess, and card games such as poker and Magic: The Gathering.

Team Role: I spent the majority of my time with this project working on knowledge graph generation. I spent a lot of time testing knowledge graph node and edge creation prompts, as well as figuring out how to utilize LangChain to best support our project. As well, I generally served as the team's Scrum Master, and helped allocate tasks to myself and my team members.

References

[1] "Hilabs refines dirty data to unlock its hidden potential," *www.hilabs.com*. https://www.hilabs.com/ (accessed Dec. 10, 2023).

[2] "An Introduction to Knowledge Graphs," *SAIL Blog*, May 10, 2021. http://ai.stanford.edu/blog/introduction-to-knowledge-graphs/ (accessed Dec. 10, 2023).

Appendix A – Key Terms

Include descriptions of technical terms, abbreviations and acronyms

Term	Definition
AI (Artificial Intelligence)	*The use of computer engineering to perform tasks that have been typically performed using human intelligence*
Large Language Model (LLM)	*A type of AI trained on large data sets to understand and produce written content.*
Knowledge Graph (KG)	*A knowledge graph is a collection of nodes representing entities, with edges representing relationships between entities.*
Natural Language Processing (NLP)	*A computational method that uses deep learning to analyze and process human language.*

Appendix B – Program Dependencies

Our program requires several dependencies to run properly, so below is a list of the packages that we used in order to bring this project together.

- Python
- LangChain
- llama-cpp-python
- NetworkX
- spaCy
- tkinter