Large Language Models & Their Applications
Hackathon – Report

Team Details:

1. C Hemachandra – PES1UG22AM044
2. Chaitra V – PES1UG22AM045
3. Chandana B S – PES1UG22AM046
4. Arpitha Venugopal – PES1UG22AM900

Date: 22/04/2025

Class: 6A CSE (AI & ML)

Lecture Notes Generation Hackathon

1. 📦 Dependency Installation: Set Up Environment for Lecture Content Extraction:

```
[3]  # ✅ Install all necessary dependencies
     !pip install -q openai-whisper
     !pip install -q moviepy
     !pip install -q python-pptx
     !pip install -q PyMuPDF
     !pip install -q pdfminer.six
     !pip install -q ffmpeg-python
```

2. 🎥 📄 Lecture Content Extractor: Transcribe Videos and Extract Text from Slides (PDF/PPT/PPTX) using Whisper and Python:

```
import os
import shutil
import json
from pathlib import Path
from moviepy.editor import VideoFileClip
import whisper
from pptx import Presentation
import fitz  # PyMuPDF
from pdfminer.high_level import extract_text as extract_pdfminer_text
from IPython.display import display
from google.colab import files
import io

# Setup
os.makedirs("Uploads", exist_ok=True)
os.makedirs("Output", exist_ok=True)

# Load Whisper
print("Loading Whisper model...")
model = whisper.load_model("base")  # use 'medium' or 'large' for better accuracy
```

```python
# File upload helpers
def upload_files(label):
    print(f"\nUpload your {label} files:")
    uploaded = files.upload()
    saved_paths = []
    for name, data in uploaded.items():
        path = os.path.join("Uploads", name)
        with open(path, 'wb') as f:
            f.write(data)
        saved_paths.append(path)
    return saved_paths

# Extractors
def transcribe_video(video_path):
    print(f"Transcribing: {video_path}")
    temp_audio = "temp_audio.wav"
    video = VideoFileClip(video_path)
    video.audio.write_audiofile(temp_audio, logger=None)
    result = model.transcribe(temp_audio)
    os.remove(temp_audio)
    return result['text']

def extract_text_from_pptx(pptx_path):
    print(f"Extracting text from PPTX: {pptx_path}")
    prs = Presentation(pptx_path)
    text = []
    for slide in prs.slides:
        for shape in slide.shapes:
            if shape.has_text_frame:
                text.append(shape.text)
    return "\n".join(text)

def extract_text_from_pdf(pdf_path):
    print(f"Extracting text from PDF: {pdf_path}")
    try:
        return extract_pdfminer_text(pdf_path)
    except Exception as e:
        print("Fallback to PyMuPDF due to error:", e)
        doc = fitz.open(pdf_path)
        text = "\n".join(page.get_text() for page in doc)
        return text

# Upload Inputs
video_files = upload_files("Lecture Video (only MP4)")
pdf_files = upload_files("PDF slides")
pptx_files = upload_files("PPTX slides")
ppt_files = upload_files("PPT slides")
```

PTO

```python
# Process all inputs
lecture_text = ""
slides_text = ""

# Process video
if video_files:
    lecture_text = transcribe_video(video_files[0])  # assuming 1 video

# Process slides
for pdf in pdf_files:
    slides_text += extract_text_from_pdf(pdf) + "\n\n"

for pptx in pptx_files:
    slides_text += extract_text_from_pptx(pptx) + "\n\n"

# Output
with open("Output/lecture_transcript.txt", "w", encoding="utf-8") as f:
    f.write(lecture_text.strip())

with open("Output/slides_text.txt", "w", encoding="utf-8") as f:
    f.write(slides_text.strip())

print("\n✅ Files processed and saved in /Output:")
print("- lecture_transcript.txt")
print("- slides_text.txt")
```

OUTPUT:

Here we are doing content extraction pipeline designed to process lecture materials, including video lectures (MP4 format) and slide decks (PDF, PPT, and PPTX formats). It runs in a Google Colab environment and uses several libraries such as whisper for audio transcription, moviepy for video-to-audio conversion, python-pptx and pdfminer for extracting text from slides. When the script is executed, it sets up two folders—Uploads for storing incoming files and Output for saving the processed text. The user is prompted to upload a lecture video, PDF slides, and PPT/PPTX slides. For video processing, the script extracts the audio from the uploaded MP4 file, uses OpenAI's Whisper model to transcribe the audio, and stores the transcript in lecture_transcript.txt.

For slide processing, the script extracts textual content from all uploaded PDF, PPT, and PPTX files. PDFs are primarily processed using pdfminer, with PyMuPDF as a fallback if any error occurs. PPTX files are parsed to collect text from all slide shapes that contain text frames. The extracted text from all slide files is combined and saved into slides_text.txt. Overall, this script helps consolidate lecture content—both spoken and visual—into readable text files for further analysis, summarization, or archiving.

3. 📝 Combine Lecture Transcript and Slide Content into a Single File:

```python
# Path to combined input file
combined_path = "Output/input.txt"

# Combine both texts with clear headers
with open(combined_path, "w", encoding="utf-8") as f:
    f.write("### Lecture Transcript\n\n")
    f.write(lecture_text.strip() + "\n\n")

    f.write("### Slide Content\n\n")
    f.write(slides_text.strip())

print(f"\n📄 Combined input saved as: {combined_path}")
```

The code snippet in the image demonstrates how to combine two separate text inputs — a lecture transcript and slide content — into a single file (Output/input.txt) with clear markdown headers for structure. It uses UTF-8 encoding for compatibility and includes headers ### Lecture Transcript and ### Slide Content to distinguish between the two sections. The strip() method ensures no leading/trailing spaces, and the result is confirmed with a print statement. This setup is ideal for preparing structured input for further processing, such as AI-based note generation.

4. 🎨 Generate and Beautify Lecture Notes in Markdown using Gemini AI:

```python
import google.generativeai as genai

# 🔐 Authenticate using your Google AI Studio API key
genai.configure(api_key="AIzaSyCN0m7WYw9yjLxhTDfFGT-vDqrpTD0x6ko")  # Replace this with your Gemini key

def generate_notes_with_gemini(input_text):
    print("🧠 Generating lecture notes using Gemini...")
    model = genai.GenerativeModel("models/gemini-1.5-pro")
    response = model.generate_content(
        f"""You are an AI assistant helping generate structured lecture notes.
Format content using:
- Headings, subheadings
- Bullet points
- Definitions and examples

Content to summarize:
{input_text}
"""
    )
    return response.text

# Read combined input
with open("Output/input.txt", "r", encoding="utf-8") as f:
    input_text = f.read()

# Generate and save notes
lecture_notes = generate_notes_with_gemini(input_text)

with open("Output/lecture_notes.txt", "w", encoding="utf-8") as f:
    f.write(lecture_notes)

print("✅ Gemini-generated notes saved to Output/lecture_notes.txt")
```

This script leverages Google's Gemini 1.5 Pro model via the google.generativeai package to generate structured lecture notes from a given text input. It begins by configuring the Gemini API with a provided API key (which should ideally be kept private and not hardcoded). The function generate_notes_with_gemini is designed to send a prompt to the Gemini model asking it to format the input content into well-organized lecture notes. The prompt explicitly instructs the model to structure the output using headings, subheadings, bullet points, and to include definitions and examples where relevant. The model processes the input_text, which is read from the Output/input.txt file, and generates a summarized version in a structured format.

Once the response from Gemini is received, the resulting formatted lecture notes are saved into a new file named lecture_notes.txt in the Output directory. The overall purpose of this script is to take raw lecture or slide text (previously extracted and combined into input.txt), and produce a more concise, human-friendly set of notes that are easy to study or reference. This automation is particularly useful for students or educators who want to quickly convert unstructured educational content into clear, well-formatted material.

5. 🎨 Generate and Beautify Lecture Notes in Markdown using Gemini AI:

```python
import google.generativeai as genai
import re

# Configure your API key
genai.configure(api_key="AIzaSyCN0m7WYw9yjLxhTDfFGT-vDqrpTD0x6ko")  # Replace with your key

model = genai.GenerativeModel(model_name="models/gemini-1.5-pro")

def format_notes_md(text):
    # Beautify: Convert "###" to bold headers
    text = re.sub(r'(?m)^### (.*?)$', r'### \1\n', text)
    text = re.sub(r'(?m)^## (.*?)$', r'## \1\n---', text)

    # Bold important words like 'Key Takeaway', 'Definition', etc.
    keywords = ['Key Takeaway', 'Definition', 'Example', 'Benefits', 'Challenges', 'Significance', 'Applications', 'Importance']
    for word in keywords:
        text = re.sub(fr'(?i)\b({word}):', r'**\1:**', text)

    # Add line breaks between bullets and paragraphs
    text = text.replace("* ", "\n* ")

    # Optional: Emojis for engagement
    emoji_map = {
        "Key Takeaway": "📌",
        "Benefits": "✅",
        "Challenges": "⚠️",
        "Example": "💡",
        "Definition": "📖",
        "Applications": "🛠️",
        "Importance": "🌟",
        "Conclusion": "🧠"
    }
    for word, emoji in emoji_map.items():
        text = text.replace(f"**{word}:**", f"{emoji} **{word}:**")

    return text
```

```python
def generate_notes_with_gemini(input_text):
    print("🧠 Generating lecture notes using Gemini with beautified formatting...")

    prompt = f"""
You are an AI assistant creating beautifully formatted lecture notes from transcripts and slides.

✅ Use clear formatting with:
- Markdown headings (##, ###)
- Bullet points and line breaks
- Bold important concepts like 'Definition', 'Example', etc.
- Use emojis sparingly to make sections engaging
- Insert horizontal dividers (---) between major sections

Here is the lecture content to be transformed:
{input_text}
"""

    response = model.generate_content(prompt)
    return format_notes_md(response.text)

# Read combined input
with open("Output/input.txt", "r", encoding="utf-8") as f:
    input_text = f.read()

# Generate and save markdown notes
lecture_notes = generate_notes_with_gemini(input_text)

with open("Output/lecture_notes.md", "w", encoding="utf-8") as f:
    f.write(lecture_notes)

print("✅ Beautiful lecture notes saved as Output/lecture_notes.md")
```

The later part of this code, enhances the previous Gemini-based lecture note generator by not only summarizing the content but also formatting the output in **Markdown** with added styling and emojis for better readability and engagement. After configuring the Gemini API with a provided key, it defines a custom function format_notes_md() to beautify the generated text. This function transforms standard headings into Markdown-style (##, ###), emphasizes key educational terms like “Definition,” “Example,” or “Challenges” with bold formatting, and adds emojis to visually distinguish sections. Additionally, it ensures better spacing by inserting line breaks and horizontal rules (---) to separate major content blocks. A predefined Gemini prompt guides the AI to produce engaging, well-structured, and clearly sectioned lecture notes from raw transcript and slide text.

The script then reads the combined lecture content from input.txt, passes it through Gemini using the generate_notes_with_gemini() function, and applies the beautification logic to format the AI-generated response. Finally, it saves the formatted Markdown notes into a file named lecture_notes.md inside the Output directory. The final result is a polished and visually pleasant Markdown document that can be easily rendered on various platforms (like GitHub, Notion, or Markdown viewers), making it ideal for study guides, documentation, or content revision.

6. 📊 Auto-Generate Quiz Questions & Answers and Concept Diagrams from Lecture Notes using Gemini AI:

```python
# -------- Q&A Generation -------- #
def generate_questions_with_gemini(note_text):
    print("❓ Generating quiz questions...")
    prompt = f"""
Based on the following lecture notes, generate a set of 5-10 quiz questions to test understanding.

Use the format:
**Q1:** Question text
**A1:** Correct answer

Lecture Notes:
{note_text}
"""
    response = model.generate_content(prompt)
    return response.text


# -------- Diagram/Concept Mapping -------- #
def generate_diagrams_with_gemini(note_text):
    print("📊 Generating text-based diagrams/concept maps...")
    prompt = f"""
From the lecture notes below, extract key concepts and represent them using ASCII-style diagrams or concept maps.

Use markdown format, and try to visually link concepts.

Lecture Notes:
{note_text}
"""
    response = model.generate_content(prompt)
    return response.text


# Read markdown notes
with open("Output/lecture_notes.md", "r", encoding="utf-8") as f:
    notes_text = f.read()

# Generate Q&A
questions_md = generate_questions_with_gemini(notes_text)
with open("Output/lecture_questions.md", "w", encoding="utf-8") as f:
    f.write(questions_md)

# Generate Diagrams
diagrams_md = generate_diagrams_with_gemini(notes_text)
with open("Output/diagrams.md", "w", encoding="utf-8") as f:
    f.write(diagrams_md)

print("✅ Q&A and Diagrams saved to Output folder.")
```

The script expands the functionality of the lecture note pipeline by introducing automated quiz question generation and text-based concept mapping from previously generated lecture notes. It defines two core functions using Google's Gemini model: generate_questions_with_gemini() and generate_diagrams_with_gemini(). The first function takes Markdown-formatted notes and prompts the model to output 5 to 10 well-structured questions and answers in a clear Q1: / A1: format, ideal for self-assessment or quiz preparation. The second function prompts the model to extract core concepts from the notes and represent them visually using ASCII-style diagrams in Markdown, enabling conceptual clarity through structured visual aids.

The script reads the lecture notes from lecture_notes.md, sends them to the Gemini model via both functions, and saves the outputs to lecture_questions.md and diagrams.md

respectively in the Output directory. The generated Q&A can be used to reinforce learning or build assessments, while the concept maps offer a compact, visual summary of the lecture content. This setup is especially useful for creating study kits, revision guides, or learning resources—all automatically, with minimal manual input.

7. 🧠 Build an Interactive Gradio App for Lecture Notes, Q&A, and Diagrams:

```
import gradio as gr
import os
import google.generativeai as genai

# Configure Gemini API
GENAI_API_KEY = "AIzaSyCN0m7WYw9yjLxhTDfFGT-vDqrpTD0x6ko"  # Replace this with your actual key
genai.configure(api_key=GENAI_API_KEY)
model = genai.GenerativeModel(model_name="models/gemini-1.5-pro")

def generate_notes(input_text):
    prompt = f"""
You are an AI assistant creating beautifully formatted lecture notes from transcripts and slides.

✅ Use clear formatting with:
- Markdown headings (##, ###)
- Bullet points and line breaks
- Bold important concepts like 'Definition', 'Example', etc.
- Use emojis sparingly to make sections engaging
- Insert horizontal dividers (---) between major sections

Here is the lecture content to be transformed:
{input_text}
"""
    response = model.generate_content(prompt)
    return response.text

def generate_questions(notes_text):
    prompt = f"""
Based on the following lecture notes, generate a set of 5-10 quiz questions to test understanding.

Use the format:
**Q1:** Question text
**A1:** Correct answer

Lecture Notes:
{notes_text}
"""
    response = model.generate_content(prompt)
    return response.text

def generate_diagrams(notes_text):
    prompt = f"""
From the lecture notes below, extract key concepts and represent them using ASCII-style diagrams or concept maps.

Use markdown format, and try to visually link concepts.

Lecture Notes:
{notes_text}
"""
    response = model.generate_content(prompt)
    return response.text

def process(input_text):
    notes = generate_notes(input_text)
    questions = generate_questions(notes)
    diagrams = generate_diagrams(notes)
    return notes, questions, diagrams

with gr.Blocks(title="📘 AI Lecture Note Generator") as demo:
    gr.Markdown("""
        # 📘 AI Lecture Note Generator
        Upload raw lecture content (transcripts + slides) and get:
        - ✅ Structured lecture notes
        - ❓ Auto-generated Q&A
        - 📊 Conceptual diagrams
```

```
    with gr.Row():
        input_textbox = gr.Textbox(label="📥 Paste Combined Lecture + Slides Text Here", lines=20, placeholder="Paste your raw content from lecture transcript + slid

    with gr.Row():
        submit_btn = gr.Button("🧠 Generate Notes, Q&A & Diagrams")

    with gr.Row():
        notes_output = gr.Markdown(label="📘 Lecture Notes")
    with gr.Row():
        qna_output = gr.Markdown(label="❓ Q&A")
    with gr.Row():
        diagram_output = gr.Markdown(label="📊 Diagrams")

    submit_btn.click(fn=process, inputs=input_textbox, outputs=[notes_output, qna_output, diagram_output])

# Launch the interface
demo.launch(debug=True, share=True)
```

This final script wraps the entire lecture content transformation pipeline into a Gradio web application, offering a user-friendly interface to generate lecture notes, quiz questions, and concept diagrams using Google's Gemini model. It consists of three backend functions: generate_notes(), generate_questions(), and generate_diagrams(), each designed to format content, extract questions, and create ASCII-style diagrams from input lecture text. These functions are orchestrated in the process() function.

- The frontend is built using gr.Blocks to define a clean layout.
- A large Textbox allows users to input raw transcript + slide content.
- A single button 🧠 Generate Notes, Q&A & Diagrams triggers the content transformation pipeline.
- Three separate Markdown display areas show the generated lecture notes, questions, and diagrams respectively.

Once launched (demo.launch()), users can interact with this tool in a browser, making it ideal for educators, students, or content creators who want to instantly convert unstructured content into study-ready material. The share=True option even lets you generate a public link for broader access.

8. 📊 Automated Evaluation of AI-Generated Lecture Notes using Gemini:

```
import re
import google.generativeai as genai

# 🔐 Configure Gemini
genai.configure(api_key="AIzaSyCN0m7WYw9yjLxhTDfFGT-vDqrpTD0x6ko")  # Replace with your valid Gemini key
model = genai.GenerativeModel("models/gemini-1.5-pro")

# 🧠 Gemini-based evaluation with full prompt and lecture note injection
def evaluate_notes(input_text, notes_text):
    prompt = f"""
You are an expert education evaluator.

Evaluate the quality of the generated lecture notes using the following 5 criteria.
For each, assign a score from 0 to 5 and explain your reasoning:

1. *Accuracy*
2. *Completeness*
3. *Organization*
4. *Readability*
5. *Value-Added*

---

### Raw Input (Transcript + Slides):
{input_text}

---
```

```
### Generated Lecture Notes:
{notes_text}

---

Return your evaluation in markdown format with the format:
*Accuracy (4/5):* explanation...
*Completeness (3/5):* explanation...
...
Then at the bottom, include a short summary of your overall impression.
"""
    response = model.generate_content(prompt)
    return response.text

# 📊 Extract score values and compute percentage breakdown
def extract_scores_and_convert_to_percentages(report_text):
    score_data = {}
    total_score = 0
    max_score = 0

    # Updated regex handles bold or plain markdown (e.g., Accuracy (4/5))
    matches = re.findall(r"(Accuracy|Completeness|Organization|Readability|Value-Added)\s*\((\d)/5\)", report_text, re.IGNORECASE)

    for metric, score_str in matches:
        score = int(score_str)
        percentage = (score / 5) * 100
        score_data[metric.strip().title()] = {
            "score": score,
            "percentage": percentage
        }
        total_score += score
        max_score += 5

    overall_percentage = (total_score / max_score) * 100 if max_score > 0 else 0
    return score_data, total_score, overall_percentage

# 📂 Load generated notes and input
with open("Output/input.txt", "r", encoding="utf-8") as f:
    raw_input = f.read()
with open("Output/lecture_notes.md", "r", encoding="utf-8") as f:
    generated_notes = f.read()

# 📝 Evaluate the notes
eval_markdown = evaluate_notes(raw_input, generated_notes)

# 📈 Extract and format scores
score_data, total, overall_pct = extract_scores_and_convert_to_percentages(eval_markdown)
score_table = "\n".join([f"- *{k}*: {v['score']}/5 ({v['percentage']}%)" for k, v in score_data.items()])
summary = f"### 📑 Evaluation Summary\n\n{score_table}\n\n*Total Score: {total}/25\nOverall Percentage*: {overall_pct:.2f}%"

# 📄 Combine into final report
final_report = f"# 📊 Evaluation Report\n\n{eval_markdown}\n\n{summary}"

# 💾 Save to file
with open("Output/evaluation_report.md", "w", encoding="utf-8") as f:
    f.write(final_report)

print("✅ Evaluation complete. Results saved to Output/evaluation_report.md")
```

This script automates the evaluation of AI-generated lecture notes using Gemini with a structured scoring system. It begins by configuring the Gemini API and loading the raw input content (input.txt) and the corresponding generated notes (lecture_notes.md). The core of the process is the evaluate_notes function, which sends both the raw and generated content to Gemini with a detailed prompt asking it to evaluate across five key educational criteria: Accuracy, Completeness, Organization, Readability, and Value-Added. For each criterion, Gemini provides a score out of 5 along with an explanation, all returned in a clean markdown format.

After the initial evaluation is generated, the script uses regex to extract each score from the markdown response and calculate both individual percentages and the overall performance

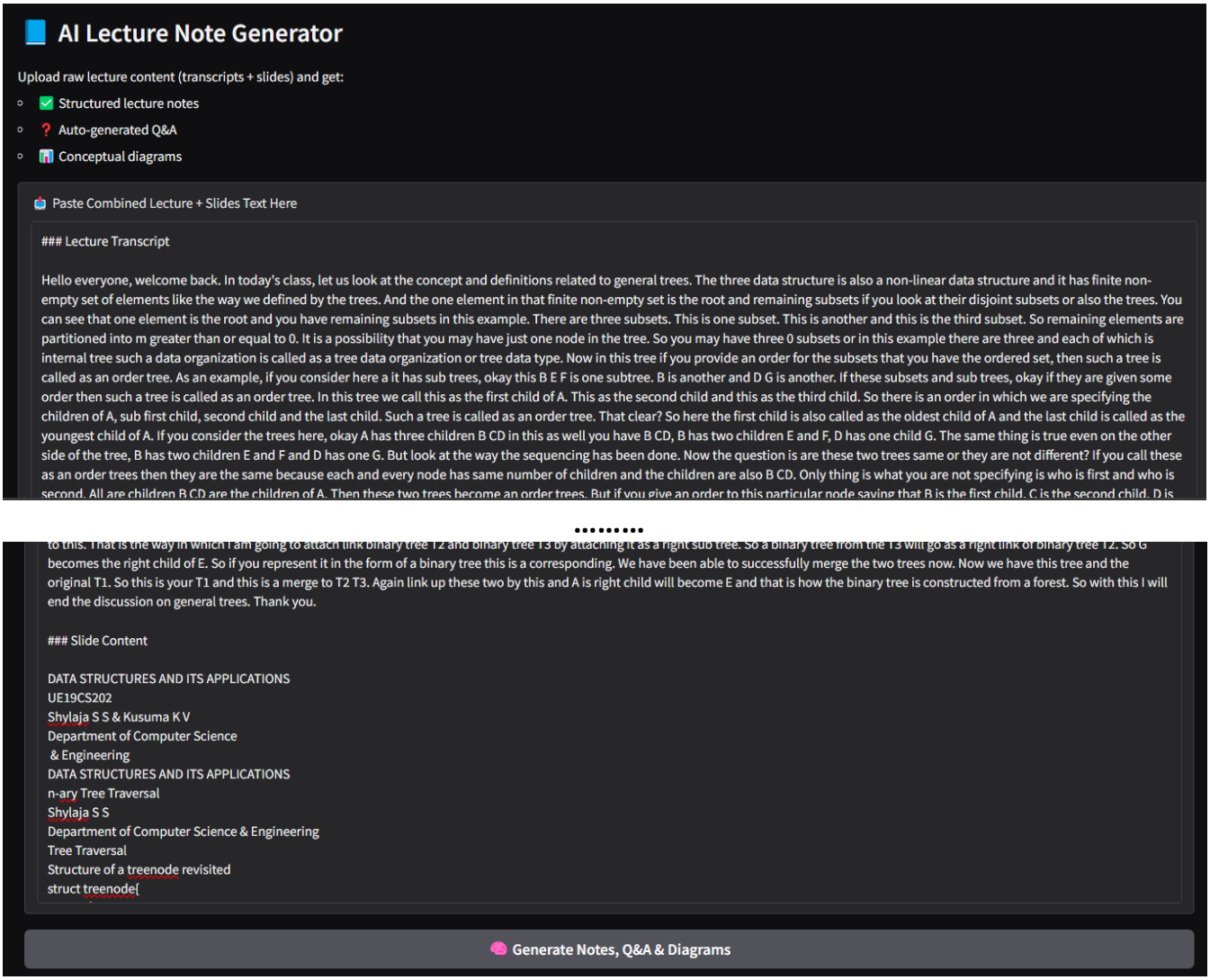
percentage. It compiles this information into a summary section that includes the total score out of 25 and the calculated overall percentage.

The final step combines the detailed markdown evaluation from Gemini with the computed summary section and saves the result to Output/evaluation_report.md. This allows educators or users to not only view qualitative feedback but also get a quantitative breakdown of the note quality — making it a powerful tool for validating and improving AI-generated educational materials.

Individual Contributions:

1. C Hemachandra: Worked on how to extract contents from videos, single file upload & gui.
2. Chaitra V: Worked on how to extract contents from pdf, multiple files upload & gui.
3. Chandana B S: Worked on how to extract contents from PPT, model selections for various tasks & gui.
4. Arpitha Venugopal: Worked on how to extract contents from PPTX, evaluation metrics & gui.

Gradio Input screenshots:



Gradio Output screenshots:

General Trees 🌲

Welcome back! Today, we'll explore the concept and definitions related to general trees.

Introduction

A tree data structure is a non-linear data structure consisting of a finite, non-empty set of elements. One element is designated as the **root**, and the remaining elements form disjoint subsets, which are also trees.

Example:

Consider a tree with root 'A' and three subsets/subtrees.

- Subset 1
- Subset 2
- Subset 3

These remaining elements are partitioned into *m* (>= 0) disjoint subsets, each an internal tree. This organization is called a **tree data organization** or **tree data type**.

Ordered Trees ➡️

If we impose an order on the subsets of a tree, it becomes an **ordered tree**.

Example:

A tree with root 'A' and subtrees:

- B E F
- C
- D G

If these subtrees have a defined order (e.g., first, second, third), the tree is ordered. The first child is also known as the oldest child, and the last child is the youngest.

Definition: In an ordered tree, the children of a node have a specific sequence (first child, second child, etc.).

Unordered: A / |
C B D (Same as ordered if order doesn't matter)

```
**N-ary Tree (5-ary Example):**
```

```
   A
 / | \
B  C  D
```

/|\
E F G H /|
I J K

```
**Left-Child Right-Sibling Representation:**
```

Forest: A E A / \ / \ | B C F G B --- C | E --- F --- G

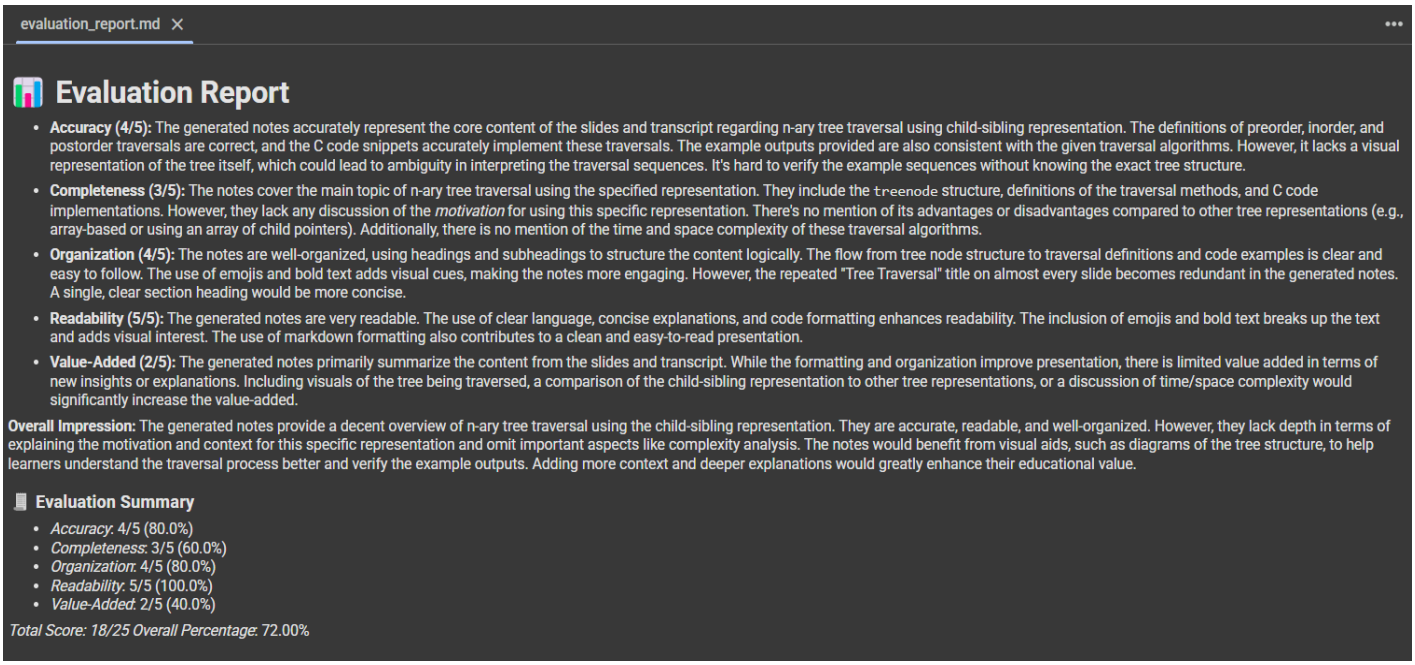
```
**Tree Node Structure (C):**

```c
struct treenode{
        int info;  // Data
        struct treenode *child; // Pointer to oldest child
        struct treenode *sibling; // Pointer to next sibling
};
```

Traversal (Conceptual):

```
Preorder:  Root -> Left Subtree -> Right Sibling Subtree
Inorder:   Left Subtree -> Root -> Right Sibling Subtree
Postorder: Left Subtree -> Right Sibling Subtree -> Root
```
```

Evaluation Report Screenshot:



evaluation_report.md

Evaluation Report

- **Accuracy (4/5):** The generated notes accurately represent the core content of the slides and transcript regarding n-ary tree traversal using child-sibling representation. The definitions of preorder, inorder, and postorder traversals are correct, and the C code snippets accurately implement these traversals. The example outputs provided are also consistent with the given traversal algorithms. However, it lacks a visual representation of the tree itself, which could lead to ambiguity in interpreting the traversal sequences. It's hard to verify the example sequences without knowing the exact tree structure.
- **Completeness (3/5):** The notes cover the main topic of n-ary tree traversal using the specified representation. They include the `treenode` structure, definitions of the traversal methods, and C code implementations. However, they lack any discussion of the *motivation* for using this specific representation. There's no mention of its advantages or disadvantages compared to other tree representations (e.g., array-based or using an array of child pointers). Additionally, there is no mention of the time and space complexity of these traversal algorithms.
- **Organization (4/5):** The notes are well-organized, using headings and subheadings to structure the content logically. The flow from tree node structure to traversal definitions and code examples is clear and easy to follow. The use of emojis and bold text adds visual cues, making the notes more engaging. However, the repeated "Tree Traversal" title on almost every slide becomes redundant in the generated notes. A single, clear section heading would be more concise.
- **Readability (5/5):** The generated notes are very readable. The use of clear language, concise explanations, and code formatting enhances readability. The inclusion of emojis and bold text breaks up the text and adds visual interest. The use of markdown formatting also contributes to a clean and easy-to-read presentation.
- **Value-Added (2/5):** The generated notes primarily summarize the content from the slides and transcript. While the formatting and organization improve presentation, there is limited value added in terms of new insights or explanations. Including visuals of the tree being traversed, a comparison of the child-sibling representation to other tree representations, or a discussion of time/space complexity would significantly increase the value-added.

Overall Impression: The generated notes provide a decent overview of n-ary tree traversal using the child-sibling representation. They are accurate, readable, and well-organized. However, they lack depth in terms of explaining the motivation and context for this specific representation and omit important aspects like complexity analysis. The notes would benefit from visual aids, such as diagrams of the tree structure, to help learners understand the traversal process better and verify the example outputs. Adding more context and deeper explanations would greatly enhance their educational value.

Evaluation Summary

- *Accuracy*: 4/5 (80.0%)
- *Completeness*: 3/5 (60.0%)
- *Organization*: 4/5 (80.0%)
- *Readability*: 5/5 (100.0%)
- *Value-Added*: 2/5 (40.0%)

Total Score: 18/25 Overall Percentage: 72.00%

FINAL ACCURACY(In Percentage) : 72%

-------THE END-------