SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 3rd semester

Health Care Generative AI Lab Manual (PGI20G07J)

This lab manual provides a structured guide for practical exercises in the development of healthcare-focused generative AI applications using Google Generative AI Studio. Each lab outlines the aim, procedure, conceptual source code (where applicable), and expected outcomes.

Lab 1: Test models using prompt samples

Title: Testing Generative AI Models with Prompt Samples

Aim: To understand the basic functionality of generative AI models by interacting with pre-defined prompt samples in Google Generative AI Studio.

Procedure:

1. Access Google Generative AI Studio.
2. Navigate to the "Test your model" or "Playground" section.
3. Select a pre-trained model (e.g., Gemini).
4. Browse through the available prompt samples provided by the studio.
5. Select a sample prompt relevant to healthcare (if available, otherwise a general one).
6. Observe the model's generated response.
7. Experiment with a few different prompt samples to see varied outputs.

Source Code:

```
// This lab primarily involves interaction with the Google Generative AI
Studio UI.
// No explicit source code is written by the user for this step,
// but conceptually, the studio makes API calls similar to:

// Example (pseudo-code for API interaction):
// const response = await model.generateContent({
//   prompt: "Sample prompt text goes here",
//   // other parameters like temperature, top_k, etc.
// });
// console.log(response.text);
```

Input:

- **Prompt Sample 1:** "Summarize the key findings of a clinical trial on a new diabetes drug."
- **Prompt Sample 2:** "Explain the process of cellular respiration in simple terms for a high school student."

Expected Output:

- **For Prompt Sample 1:** A concise summary of hypothetical clinical trial findings, including drug efficacy, side effects, and patient outcomes.
- **For Prompt Sample 2:** A clear, easy-to-understand explanation of cellular respiration, avoiding overly technical jargon.

Lab 2: Design and save our own prompts

Title: Designing and Saving Custom Prompts

Aim: To learn how to craft effective custom prompts and save them within Google Generative AI Studio for future use and iteration.

Procedure:

1. Access Google Generative AI Studio.
2. Navigate to the "Create new prompt" or "Prompt design" section.
3. Choose a model to work with.
4. Start designing a prompt from scratch. Consider a healthcare-related scenario (e.g., generating patient discharge instructions, summarizing medical research).
5. Refine the prompt by adding context, examples, or specific instructions to guide the model's output.
6. Test the prompt and make adjustments based on the generated response.
7. Use the "Save" or "Export" functionality to store your custom prompt.

Source Code:

```
// This lab focuses on prompt engineering within the Google Generative AI
Studio UI.
// While no direct code is written, the saved prompt can be thought of as a
structured input:

// Example of a saved prompt structure (conceptual):
// {
//   "name": "PatientDischargeSummaryGenerator",
//   "description": "Generates concise discharge summaries for common
conditions.",
//   "model": "gemini-pro",
//   "prompt_template": "Generate a concise patient discharge summary for a
patient diagnosed with [DIAGNOSIS] who underwent [PROCEDURE]. Include key
instructions for [MEDICATION], [DIET], and [FOLLOW_UP_APPOINTMENT].",
//   "variables": ["DIAGNOSIS", "PROCEDURE", "MEDICATION", "DIET",
"FOLLOW_UP_APPOINTMENT"]
// }
```

Input:

- **Designed Prompt:** "Write a short, empathetic message for a patient recovering from knee surgery, encouraging them to follow their physical therapy exercises. Focus on the benefits of consistency."

Expected Output:

- **Saved Prompt:** The designed prompt saved with a meaningful name and description within the Google Generative AI Studio, ready for reuse.
- **Model Response (during testing):** An encouraging message for a patient, emphasizing the importance of physical therapy for recovery.

Lab 3: Convert text-to-speech and speech-to-text

Title: Implementing Text-to-Speech and Speech-to-Text Conversion

Aim: To explore and implement functionalities for converting text into spoken audio and spoken audio back into text, crucial for accessible healthcare applications.

Procedure:

1. Identify and utilize a text-to-speech (TTS) and speech-to-text (STT) service or API (e.g., Google Cloud Text-to-Speech, Google Cloud Speech-to-Text).
2. For TTS: Provide a text input (e.g., a medical instruction).
3. For TTS: Configure parameters like voice, language, and speaking rate.
4. For TTS: Generate and play the audio output.
5. For STT: Provide an audio input (e.g., a recorded patient query).
6. For STT: Process the audio to obtain the transcribed text.
7. Observe the accuracy of the conversions.

Source Code:

```
// Example using conceptual API calls for TTS and STT
async function convertTextToSpeech(text) {
    console.log("Converting text to speech...");
    // In a real application, this would be an API call to a TTS service
    // For example:
    /*
    const response = await
fetch('https://texttospeech.googleapis.com/v1/text:synthesize', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json', 'Authorization':
'Bearer YOUR_API_KEY' },
        body: JSON.stringify({
            input: { text: text },
            voice: { languageCode: 'en-US', ssmlGender: 'FEMALE' },
            audioConfig: { audioEncoding: 'MP3' }
        })
    });
    const data = await response.json();
    // Play the audio from data.audioContent
    console.log("Audio generated successfully.");
    */
    console.log(`Simulated TTS output for: "${text}"`);
    // In a browser, you might use SpeechSynthesisUtterance
    const utterance = new SpeechSynthesisUtterance(text);
    speechSynthesis.speak(utterance);
}

async function convertSpeechToText() {
    console.log("Converting speech to text...");
    // In a real application, this would involve using a SpeechRecognition
API
    // or sending audio to an STT service.
    /*
    const recognition = new (window.SpeechRecognition ||
window.webkitSpeechRecognition)();
    recognition.lang = 'en-US';
    recognition.interimResults = false;
    recognition.maxAlternatives = 1;

    recognition.start();
```

```
    recognition.onresult = (event) => {
        const speechResult = event.results[0][0].transcript;
        console.log(`Simulated STT output: "${speechResult}"`);
        return speechResult;
    };

    recognition.onerror = (event) => {
        console.error('Speech recognition error:', event.error);
    };
    */
    console.log("Simulated STT output: 'The patient reported feeling much
better today.'");
    return "The patient reported feeling much better today.";
}

// Example usage (conceptual):
// convertTextToSpeech("Please take your medication twice a day.");
// convertSpeechToText();
```

Input:

- **For TTS:** Text: "Your appointment is scheduled for tomorrow at 10 AM. Please arrive 15 minutes early."
- **For STT:** Audio recording of a user saying: "How do I check my blood sugar levels?"

Expected Output:

- **For TTS:** An audio playback of the provided text.
- **For STT:** Transcribed text: "How do I check my blood sugar levels?"

Lab 4: Google AI Studio quick start

Title: Google AI Studio Quick Start Guide

Aim: To gain hands-on experience with the fundamental features and workflow of Google AI Studio, enabling rapid prototyping of AI applications.

Procedure:

1. Access Google AI Studio (ai.google.dev).
2. Log in with your Google account.
3. Explore the user interface, identifying key sections like "Get started," "Build with prompts," "API key," and "My projects."
4. Create a new project.
5. Generate your first API key (if not already done).
6. Run a simple "freeform" prompt to confirm basic functionality.
7. Understand how to navigate between different prompt types (e.g., freeform, chat, structured).

Source Code:

```
// This lab is primarily about setting up and navigating the Google AI Studio
environment.
// No specific code is written by the user at this stage, but the outcome
enables
// future coding. The API key generated is crucial for programmatic access.

// Example of how an API key might be used in a later lab:
// const { GoogleGenerativeAI } = require('@google/generative-ai');
// const API_KEY = "YOUR_GENERATED_API_KEY"; // This is obtained from AI
Studio
// const genAI = new GoogleGenerativeAI(API_KEY);
// const model = genAI.getGenerativeModel({ model: "gemini-pro" });
```

Input:

- N/A (The input is the user's interaction with the AI Studio interface).

Expected Output:

- A successfully configured Google AI Studio environment.
- A generated API key.
- A basic understanding of the AI Studio interface and its main functionalities.

Lab 5: Writing scripts with Gemini AI

Title: Scripting with Gemini AI for Content Generation

Aim: To learn how to leverage Gemini AI models programmatically to generate various forms of content, such as creative writing, summaries, or dialogues.

Procedure:

1. Set up your development environment with the necessary Gemini AI client library (e.g., Node.js, Python).
2. Obtain your API key from Google AI Studio.
3. Write a script that initializes the Gemini model.
4. Define a prompt within your script (e.g., "Write a short story about a doctor who invents a new diagnostic tool.").
5. Make an API call to the Gemini model to generate content based on your prompt.
6. Process and display the generated output.

Source Code:

```
// This is a conceptual example for a JavaScript environment.
// Ensure you have the Google Generative AI SDK installed: npm install
@google/generative-ai

async function generateScriptWithGemini(promptText) {
    // IMPORTANT: Replace with your actual API key.
    // In a real application, this should be loaded securely (e.g., from
environment variables).
    const API_KEY = ""; // Your API key from Google AI Studio

    if (!API_KEY) {
        console.error("API_KEY is not set. Please get your API key from
Google AI Studio.");
        return "Error: API Key missing.";
    }

    try {
        let chatHistory = [];
        chatHistory.push({ role: "user", parts: [{ text: promptText }] });

        const payload = { contents: chatHistory };
        const apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-
flash:generateContent?key=${API_KEY}`;

        // Display a loading indicator
        console.log("Generating content with Gemini AI...");

        const response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });

        const result = await response.json();

        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            const generatedText = result.candidates[0].content.parts[0].text;
```

```
            console.log("Generated Content:\n", generatedText);
            return generatedText;
        } else {
            console.error("Error: Unexpected response structure or no content
generated.", result);
            return "Error: Could not generate content.";
        }
    } catch (error) {
        console.error("Error calling Gemini API:", error);
        return `Error: ${error.message}`;
    }
}

// Example Usage:
// generateScriptWithGemini("Write a short motivational script for a
healthcare worker facing burnout.");
```

Input:

- **Prompt:** "Write a short script for a public service announcement about the importance of regular health check-ups."

Expected Output:

- A short script for a PSA, including dialogue and potential scene descriptions, emphasizing preventive healthcare.

Lab 6: Creating text prompts with Google AI Studio and Gemini AI

Title: Advanced Text Prompt Engineering with Google AI Studio and Gemini AI

Aim: To master the art of crafting precise and effective text prompts using the features of Google AI Studio and the capabilities of Gemini AI for specific generative tasks.

Procedure:

1. Open Google AI Studio and select the Gemini model.
2. Choose the "Freeform" or "Structured" prompt type.
3. Experiment with different prompt engineering techniques:
    - **Zero-shot prompting:** A simple instruction.
    - **Few-shot prompting:** Providing examples within the prompt.
    - **Chain-of-thought prompting:** Guiding the model through reasoning steps.
    - **Role-playing:** Assigning a persona to the AI.
4. Apply these techniques to create prompts for healthcare-related tasks (e.g., generating patient education materials, summarizing medical articles, creating hypothetical patient scenarios).
5. Utilize parameters like temperature, top-k, and top-p to control the creativity and determinism of the output.
6. Iteratively refine prompts based on the model's responses.

Source Code:

```
// This lab focuses on the iterative process of prompt creation and
refinement
// within Google AI Studio. While the core is UI interaction, the underlying
// API calls involve structured prompts and parameters.

// Example of a structured prompt (conceptual):
// {
//   "model": "gemini-pro",
//   "prompt": {
//     "text": "As a medical assistant, explain the importance of vaccination
to a hesitant parent. Keep it concise and address common concerns about
safety and efficacy.",
//     "examples": [
//       {
//         "input": {"text": "Explain why vaccines are important."},
//         "output": {"text": "Vaccines protect your child from serious
diseases by building their immunity."}
//       }
//     ]
//   },
//   "parameters": {
//     "temperature": 0.7,
//     "top_p": 0.9,
//     "top_k": 40
//   }
// }
```

Input:

- **Prompt Idea:** Generate a list of questions a patient should ask their doctor before surgery.
- **Parameters:** Temperature: 0.5, Top-k: 20

Expected Output:

- A well-structured prompt that, when run, generates a comprehensive list of questions for pre-surgery consultation.
- Varied outputs demonstrating the effect of different parameters on the generated text.

Lab 7: Code completion and generation

Title: Utilizing Gemini AI for Code Completion and Generation

Aim: To understand how generative AI, specifically Gemini, can assist developers in writing code more efficiently through intelligent code completion and generation based on natural language descriptions.

Procedure:

1. Set up a development environment (e.g., VS Code with a Gemini AI extension, or a simple script using the Gemini API).
2. Provide a natural language description of a function or code snippet you want to generate (e.g., "Python function to calculate BMI").
3. Observe the AI's ability to complete partial code or generate entire functions.
4. Test the generated code for correctness and make necessary adjustments.
5. Experiment with generating code in different programming languages or for different tasks relevant to healthcare data processing or analysis.

Source Code:

```
// This is a conceptual example of how Gemini AI can be used for code
generation.
// In a real scenario, this would often be integrated into an IDE or a
development tool.

async function generateCodeWithGemini(codePrompt) {
    const API_KEY = ""; // Your API key from Google AI Studio

    if (!API_KEY) {
        console.error("API_KEY is not set. Please get your API key from
Google AI Studio.");
        return "Error: API Key missing.";
    }

    try {
        let chatHistory = [];
        chatHistory.push({ role: "user", parts: [{ text: `Generate code based
on the following description:\n\n${codePrompt}` }] });

        const payload = { contents: chatHistory };
        const apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-
flash:generateContent?key=${API_KEY}`;

        console.log("Requesting code generation from Gemini AI...");

        const response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });

        const result = await response.json();

        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            const generatedCode = result.candidates[0].content.parts[0].text;
            console.log("Generated Code:\n", generatedCode);
```

```
            return generatedCode;
        } else {
            console.error("Error: Unexpected response structure or no code
generated.", result);
            return "Error: Could not generate code.";
        }
    } catch (error) {
        console.error("Error calling Gemini API for code generation:",
error);
        return `Error: ${error.message}`;
    }
}

// Example Usage:
// generateCodeWithGemini("Python function to calculate the average age from
a list of patient dictionaries, where each dictionary has an 'age' key.");
```

Input:

- **Code Description 1:** "JavaScript function to validate an email address using a regular expression."
- **Code Description 2:** "Python function to connect to a PostgreSQL database and fetch all records from a 'patients' table."

Expected Output:

- **For Code Description 1:** A JavaScript function with a regular expression for email validation.
- **For Code Description 2:** A Python function demonstrating database connection and data retrieval (with placeholder credentials).

Lab 8: Generate and Customize Images

Title: Generating and Customizing Images with Generative AI

Aim: To explore the capabilities of generative AI models (like Imagen) in creating and modifying images based on textual descriptions, with potential applications in medical visualization or educational content.

Procedure:

1. Identify and utilize an image generation API (e.g., Imagen 3.0).
2. Provide a text prompt describing the desired image (e.g., "A microscopic view of a healthy human cell, vibrant colors, scientific illustration style").
3. Generate the initial image.
4. Experiment with customizing the image by modifying the prompt (e.g., adding details, changing styles, specifying colors).
5. Explore parameters to control image generation (e.g., aspect ratio, resolution, number of variations).
6. Observe the quality and relevance of the generated and customized images.

Source Code:

```
// This is a conceptual example for image generation using the Imagen API.
// Ensure you have the necessary setup for API calls.

async function generateAndCustomizeImage(imagePrompt) {
    const API_KEY = ""; // Your API key from Google AI Studio

    if (!API_KEY) {
        console.error("API_KEY is not set. Please get your API key from
Google AI Studio.");
        return "Error: API Key missing.";
    }

    try {
        const payload = { instances: { prompt: imagePrompt }, parameters: {
"sampleCount": 1 } };
        const apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/imagen-3.0-generate-
002:predict?key=${API_KEY}`;

        // Display a loading indicator
        console.log("Generating image with Imagen AI...");

        const response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });

        const result = await response.json();

        if (result.predictions && result.predictions.length > 0 &&
result.predictions[0].bytesBase64Encoded) {
            const imageUrl =
`${result.predictions[0].bytesBase64Encoded}`;
            console.log("Generated Image URL (Base64):",
imageUrl.substring(0, 50) + "..."); // Log first 50 chars
            // In a web app, you would display this image:
            // const imgElement = document.createElement('img');
            // imgElement.src = imageUrl;
```

```
            // document.body.appendChild(imgElement);
            return imageUrl;
        } else {
            console.error("Error: Unexpected response structure or no image
generated.", result);
            return "Error: Could not generate image.";
        }
    } catch (error) {
        console.error("Error calling Imagen API:", error);
        return `Error: ${error.message}`;
    }
}

// Example Usage:
// generateAndCustomizeImage("A stylized illustration of a human heart with
glowing arteries, medical abstract art.");
```

Input:

- **Initial Prompt:** "A serene hospital waiting room, modern design, natural light."
- **Customization Prompt:** "A serene hospital waiting room, modern design, natural light, with a small potted plant on a table."

Expected Output:

- **Initial Output:** An image depicting a modern hospital waiting room.
- **Customized Output:** An image of the same waiting room, but with the addition of a small potted plant.

Lab 9: Universal speech model

Title: Exploring Universal Speech Models for Multilingual Applications

Aim: To understand and experiment with universal speech models that can handle various languages and accents, enabling broader accessibility in healthcare communication.

Procedure:

1. Identify a universal speech model API (e.g., Google's Universal Speech Model).
2. Provide audio input in different languages or with different accents.
3. Observe the model's ability to accurately transcribe or translate the speech regardless of the input language.
4. Test the model with healthcare-specific terminology in various languages.
5. Consider how such a model could be integrated into a multilingual patient information system or a global telehealth platform.

Source Code:

```
// This lab involves using a sophisticated universal speech model API.
// The code will conceptually demonstrate interaction with such a service.

async function processUniversalSpeech(audioData, sourceLanguage = 'auto',
targetLanguage = 'en') {
    console.log(`Processing audio for transcription/translation from
${sourceLanguage} to ${targetLanguage}...`);
    // In a real application, audioData would be a binary representation of
the audio file.
    // The API call would look something like this (conceptual):
    /*
    const response = await
fetch('https://universalspeechmodel.googleapis.com/v1/speech:process', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json', 'Authorization':
'Bearer YOUR_API_KEY' },
        body: JSON.stringify({
            audioContent: audioData.toString('base64'), // Base64 encoded
audio
            config: {
                sourceLanguage: sourceLanguage,
                targetLanguage: targetLanguage, // For translation
                enableAutomaticLanguageDetection: true // If sourceLanguage
is 'auto'
            }
        })
    });
    const data = await response.json();
    console.log("Transcription:", data.transcription);
    if (data.translation) {
        console.log("Translation:", data.translation);
    }
    return { transcription: data.transcription, translation: data.translation
};
    */
    console.log("Simulated universal speech processing.");
    if (sourceLanguage === 'es' && targetLanguage === 'en') {
        return { transcription: "Hola, ¿cómo estás?", translation: "Hello,
how are you?" };
    } else if (sourceLanguage === 'en' && targetLanguage === 'en') {
        return { transcription: "I need to schedule an appointment.",
translation: null };
```

```
    }
    return { transcription: "Simulated transcription", translation:
"Simulated translation" };
}

// Example Usage (conceptual):
// Assume 'audioDataSpanish' is a binary representation of Spanish audio
// processUniversalSpeech(audioDataSpanish, 'es', 'en');
// processUniversalSpeech(audioDataEnglish, 'en', 'en');
```

Input:

- **Audio 1:** A short audio clip of someone speaking "Buenos días, necesito un médico" (Spanish).
- **Audio 2:** A short audio clip of someone speaking "I have a severe headache" (English, with a non-native accent).

Expected Output:

- **For Audio 1:** Transcription: "Buenos días, necesito un médico". Translation: "Good morning, I need a doctor."
- **For Audio 2:** Transcription: "I have a severe headache."

Lab 10: Build a product copy generator

Title: Building an AI-Powered Product Copy Generator for Healthcare Products

Aim: To develop a generative AI application that can automatically create compelling marketing copy for healthcare-related products or services, leveraging the power of large language models.

Procedure:

1. Define the requirements for the product copy generator (e.g., inputs like product name, features, target audience; desired output length, tone).
2. Choose a generative AI model (e.g., Gemini) and integrate its API into your application.
3. Design a robust prompt that guides the AI to produce high-quality, persuasive product copy. This prompt should include placeholders for product details.
4. Implement a user interface (even a simple command-line one) to accept product information as input.
5. Make an API call to the AI model with the filled-in prompt.
6. Display the generated product copy to the user.
7. Add features for customization or regeneration.

Source Code:

```
// This is a conceptual example for a product copy generator.
// It assumes a basic HTML structure for input and output.

async function generateProductCopy() {
    const productName = document.getElementById('productName').value;
    const productFeatures = document.getElementById('productFeatures').value;
    const targetAudience = document.getElementById('targetAudience').value;
    const outputDiv = document.getElementById('generatedCopy');
    const API_KEY = ""; // Your API key from Google AI Studio

    if (!API_KEY) {
        outputDiv.innerText = "Error: API Key missing. Please get your API
key from Google AI Studio.";
        return;
    }

    if (!productName || !productFeatures || !targetAudience) {
        outputDiv.innerText = "Please fill in all product details.";
        return;
    }

    const prompt = `Generate compelling marketing copy for a healthcare
product.
    Product Name: ${productName}
    Key Features: ${productFeatures}
    Target Audience: ${targetAudience}
    Tone: Professional, empathetic, and persuasive.
    Length: Approximately 100-150 words.
    Include a clear call to action.`;

    try {
        let chatHistory = [];
        chatHistory.push({ role: "user", parts: [{ text: prompt }] });

        const payload = { contents: chatHistory };
        const apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-
flash:generateContent?key=${API_KEY}`;
```

```
        outputDiv.innerText = "Generating product copy...";

        const response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });

        const result = await response.json();

        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            outputDiv.innerText = result.candidates[0].content.parts[0].text;
        } else {
            outputDiv.innerText = "Error: Could not generate product copy.";
            console.error("Unexpected response structure or no content
generated.", result);
        }
    } catch (error) {
        outputDiv.innerText = `Error generating copy: ${error.message}`;
        console.error("Error calling Gemini API:", error);
    }
}

// Example HTML structure (conceptual, not part of the JS function itself):
/*
<div style="font-family: 'Inter', sans-serif; max-width: 600px; margin: 20px
auto; padding: 20px; border: 1px solid #ccc; border-radius: 8px; box-shadow:
0 2px 4px rgba(0,0,0,0.1);">
    <h2 style="text-align: center; color: #333;">Product Copy Generator</h2>
    <div style="margin-bottom: 15px;">
        <label for="productName" style="display: block; margin-bottom: 5px;
font-weight: bold;">Product Name:</label>
        <input type="text" id="productName" placeholder="e.g., 'NutriBoost
Vitamin D Supplement'" style="width: 100%; padding: 8px; border: 1px solid
#ddd; border-radius: 4px;">
    </div>
    <div style="margin-bottom: 15px;">
        <label for="productFeatures" style="display: block; margin-bottom:
5px; font-weight: bold;">Key Features:</label>
        <textarea id="productFeatures" rows="3" placeholder="e.g., 'High
potency, easy absorption, supports bone health, boosts immunity'"
style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius:
4px;"></textarea>
    </div>
    <div style="margin-bottom: 20px;">
        <label for="targetAudience" style="display: block; margin-bottom:
5px; font-weight: bold;">Target Audience:</label>
        <input type="text" id="targetAudience" placeholder="e.g., 'Adults
seeking bone health support'" style="width: 100%; padding: 8px; border: 1px
solid #ddd; border-radius: 4px;">
    </div>
    <button onclick="generateProductCopy()" style="width: 100%; padding: 10px
15px; background-color: #4CAF50; color: white; border: none; border-radius:
5px; cursor: pointer; font-size: 16px; transition: background-color 0.3s
ease;">Generate Copy</button>
    <div id="generatedCopy" style="margin-top: 20px; padding: 15px; border:
1px dashed #ccc; border-radius: 4px; background-color: #f9f9f9; min-height:
100px; white-space: pre-wrap;">
        Your generated product copy will appear here.
    </div>
</div>
*/
```

Input:

- **Product Name:** "MediCare Telehealth App"
- **Key Features:** "Secure video consultations, prescription refills, appointment scheduling, symptom checker."
- **Target Audience:** "Patients seeking convenient and accessible healthcare from home."

Expected Output:

- Marketing copy for "MediCare Telehealth App," highlighting its features, benefits for the target audience, and a call to action (e.g., "Download today!").

Lab 11: Build a custom chat application

Title: Building a Custom Generative AI Chat Application

Aim: To develop a simple chat application that interacts with a generative AI model (like Gemini) to provide conversational responses, simulating a healthcare chatbot.

Procedure:

1. Design the architecture for your chat application (frontend for UI, backend for API calls, or a purely client-side approach with API key).
2. Set up the user interface with an input field for messages and a display area for chat history.
3. Integrate the Gemini AI API to send user messages and receive AI responses.
4. Implement a conversational flow, maintaining chat history to enable context-aware responses.
5. Consider adding features like loading indicators, error handling, and basic input validation.
6. Test the chat application with various healthcare-related queries.

Source Code:

```
// This is a conceptual example for a simple chat application.
// It assumes a basic HTML structure for the chat interface.

let chatHistory = []; // Stores the conversation history

async function sendMessage() {
    const userInput = document.getElementById('userInput').value;
    const chatDisplay = document.getElementById('chatDisplay');
    const API_KEY = ""; // Your API key from Google AI Studio

    if (!API_KEY) {
        chatDisplay.innerHTML += `<div class="message bot-message">Error: API
Key missing.</div>`;
        return;
    }

    if (!userInput.trim()) {
        return; // Don't send empty messages
    }

    // Add user message to display and history
    chatDisplay.innerHTML += `<div class="message user-
message">${userInput}</div>`;
    chatHistory.push({ role: "user", parts: [{ text: userInput }] });
    document.getElementById('userInput').value = ''; // Clear input

    // Add loading indicator
    const loadingMessage = document.createElement('div');
    loadingMessage.className = 'message bot-message loading';
    loadingMessage.innerText = 'AI is typing...';
    chatDisplay.appendChild(loadingMessage);
    chatDisplay.scrollTop = chatDisplay.scrollHeight; // Scroll to bottom

    try {
        const payload = { contents: chatHistory };
        const apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-
flash:generateContent?key=${API_KEY}`;
```

```
        const response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });

        const result = await response.json();

        // Remove loading indicator
        chatDisplay.removeChild(loadingMessage);

        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            const botResponse = result.candidates[0].content.parts[0].text;
            chatDisplay.innerHTML += `<div class="message bot-
message">${botResponse}</div>`;
            chatHistory.push({ role: "model", parts: [{ text: botResponse }]
}); // Add bot response to history
        } else {
            chatDisplay.innerHTML += `<div class="message bot-message error-
message">Error: Could not get a response.</div>`;
            console.error("Unexpected response structure or no content
generated.", result);
        }
    } catch (error) {
        // Remove loading indicator
        chatDisplay.removeChild(loadingMessage);
        chatDisplay.innerHTML += `<div class="message bot-message error-
message">Error: ${error.message}</div>`;
        console.error("Error calling Gemini API:", error);
    }
    chatDisplay.scrollTop = chatDisplay.scrollHeight; // Scroll to bottom
}

// Example HTML structure (conceptual, not part of the JS function itself):
/*
<style>
    body { font-family: 'Inter', sans-serif; margin: 0; padding: 0; display:
flex; justify-content: center; align-items: center; min-height: 100vh;
background-color: #f0f2f5; }
    .chat-container { width: 100%; max-width: 500px; height: 80vh; display:
flex; flex-direction: column; border: 1px solid #ddd; border-radius: 10px;
overflow: hidden; box-shadow: 0 4px 8px rgba(0,0,0,0.1); background-color:
white; }
    .chat-header { background-color: #007bff; color: white; padding: 15px;
text-align: center; font-size: 1.2em; border-top-left-radius: 10px; border-
top-right-radius: 10px; }
    .chat-display { flex-grow: 1; padding: 15px; overflow-y: auto;
background-color: #e9ecef; }
    .message { margin-bottom: 10px; padding: 8px 12px; border-radius: 15px;
max-width: 75%; word-wrap: break-word; }
    .user-message { background-color: #dcf8c6; align-self: flex-end; margin-
left: auto; }
    .bot-message { background-color: #ffffff; align-self: flex-start; margin-
right: auto; border: 1px solid #e0e0e0; }
    .error-message { background-color: #ffcccc; color: #cc0000; }
    .loading { font-style: italic; color: #666; }
    .chat-input-area { display: flex; padding: 15px; border-top: 1px solid
#ddd; background-color: #f8f9fa; }
    .chat-input-area input { flex-grow: 1; padding: 10px; border: 1px solid
#ccc; border-radius: 20px; outline: none; }
    .chat-input-area button { background-color: #28a745; color: white;
border: none; border-radius: 20px; padding: 10px 15px; margin-left: 10px;
cursor: pointer; transition: background-color 0.3s ease; }
```

```
    .chat-input-area button:hover { background-color: #218838; }
</style>

<div class="chat-container">
    <div class="chat-header">Health AI Chatbot</div>
    <div class="chat-display" id="chatDisplay">
        <div class="message bot-message">Hello! How can I assist you with
your health queries today?</div>
    </div>
    <div class="chat-input-area">
        <input type="text" id="userInput" placeholder="Type your message..."
onkeypress="if(event.key === 'Enter') sendMessage()">
        <button onclick="sendMessage()">Send</button>
    </div>
</div>
*/
```

Input:

- User Message 1: "What are the common symptoms of the flu?"
- User Message 2: "How can I prevent it?"

Expected Output:

- **Response 1:** A list of common flu symptoms (e.g., fever, cough, body aches).
- **Response 2:** Advice on flu prevention (e.g., vaccination, hand washing, avoiding sick individuals).

Lab 12: Experiment with model parameters

Title: Experimenting with Generative AI Model Parameters

Aim: To understand how different model parameters (e.g., temperature, top-k, top-p, max output tokens) influence the behavior and output characteristics of generative AI models.

Procedure:

1. Access Google AI Studio or set up a script to interact with the Gemini API.
2. Choose a simple, open-ended prompt (e.g., "Describe a healthy lifestyle.").
3. Run the prompt multiple times, each time varying one or more parameters:
    - **Temperature:** Start low (0.1) for deterministic output, then increase (0.9) for more creative/diverse output.
    - **Top-k:** Observe how limiting the number of token choices affects coherence.
    - **Top-p:** See how probability mass sampling changes output diversity.
    - **Max Output Tokens:** Control the length of the generated response.
4. Analyze the differences in the generated text for each parameter combination.
5. Document your observations on how each parameter affects creativity, coherence, and length.

Source Code:

```
// This is a conceptual example demonstrating how to vary parameters
// when making an API call to Gemini.

async function experimentWithParameters(promptText, temperature, topK, topP,
maxOutputTokens) {
    const API_KEY = ""; // Your API key from Google AI Studio

    if (!API_KEY) {
        console.error("API_KEY is not set. Please get your API key from
Google AI Studio.");
        return "Error: API Key missing.";
    }

    try {
        let chatHistory = [];
        chatHistory.push({ role: "user", parts: [{ text: promptText }] });

        const payload = {
            contents: chatHistory,
            generationConfig: {
                temperature: temperature,
                topK: topK,
                topP: topP,
                maxOutputTokens: maxOutputTokens,
            }
        };
        const apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-
flash:generateContent?key=${API_KEY}`;

        console.log(`Generating with params: Temp=${temperature},
TopK=${topK}, TopP=${topP}, MaxTokens=${maxOutputTokens}`);

        const response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
```

```
        const result = await response.json();

        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            const generatedText = result.candidates[0].content.parts[0].text;
            console.log("Generated Content:\n", generatedText);
            return generatedText;
        } else {
            console.error("Error: Unexpected response structure or no content
generated.", result);
            return "Error: Could not generate content.";
        }
    } catch (error) {
        console.error("Error calling Gemini API:", error);
        return `Error: ${error.message}`;
    }
}

// Example Usage (conceptual):
// experimentWithParameters("Write a short poem about health.", 0.2, 1, 0.9,
50); // More deterministic
// experimentWithParameters("Write a short poem about health.", 0.9, 40,
0.95, 50); // More creative
```

Input:

- **Prompt:** "Describe the benefits of mindfulness for mental health."
- **Parameter Set 1:** Temperature = 0.2, Top-k = 1, Top-p = 0.9, Max Output Tokens = 100
- **Parameter Set 2:** Temperature = 0.8, Top-k = 40, Top-p = 0.95, Max Output Tokens = 100

Expected Output:

- **For Parameter Set 1:** A concise, factual, and less varied description of mindfulness benefits.
- **For Parameter Set 2:** A more creative, potentially diverse, and slightly less predictable description of mindfulness benefits.

Lab 13: Case Study I

Title: Case Study I: Applying Generative AI in Patient Education

Aim: To apply the learned generative AI concepts to a specific healthcare case study, focusing on creating personalized and accessible patient education materials.

Procedure:

1. **Understand the Case:** Analyze a provided healthcare scenario where patient education is critical (e.g., explaining a chronic condition like diabetes to a newly diagnosed patient).
2. **Define the Task:** Determine the specific generative AI task (e.g., generate a simple explanation, create FAQs, draft a personalized care plan summary).
3. **Prompt Engineering:** Design and refine prompts using Google AI Studio and Gemini AI, incorporating best practices for clarity, empathy, and medical accuracy.
4. **Content Generation:** Generate the required patient education content using the AI model.
5. **Review and Refine:** Critically evaluate the generated content for accuracy, readability, and appropriateness for the target patient demographic. Make manual edits or further prompt refinements.
6. **Discussion:** Discuss the challenges and benefits of using generative AI for patient education in this specific case.

Source Code:

```
// Source code will be specific to the chosen task within the case study.
// It will likely involve API calls to Gemini, similar to Lab 5 or Lab 10,
// but with prompts tailored to the specific patient education scenario.

// Example conceptual prompt for this case study:
// "As a compassionate healthcare educator, explain Type 2 Diabetes to a 60-
year-old patient
// who is newly diagnosed. Use simple language, focus on diet and exercise,
and
// provide actionable steps. Avoid medical jargon where possible."
```

Input:

- **Case Study Scenario:** A patient, Mr. Sharma (60 years old), has just been diagnosed with Type 2 Diabetes. He is anxious and has limited medical literacy.
- **Prompt:** "Generate a simplified explanation of Type 2 Diabetes for a 60-year-old patient, focusing on diet, exercise, and basic management steps. Use a reassuring and encouraging tone."

Expected Output:

- A clear, concise, and empathetic explanation of Type 2 Diabetes tailored for Mr. Sharma, including practical advice on diet, exercise, and medication adherence.

Lab 14: Case Study II

Title: Case Study II: Generative AI for Clinical Decision Support

Aim: To explore the application of generative AI in assisting healthcare professionals with clinical decision-making by summarizing complex medical information or suggesting differential diagnoses.

Procedure:

1. **Understand the Case:** Analyze a provided clinical scenario (e.g., a patient presenting with ambiguous symptoms, requiring a differential diagnosis or a summary of recent research on a rare condition).
2. **Define the Task:** Determine the specific generative AI task (e.g., summarize recent research papers, list potential diagnoses based on symptoms, suggest treatment options).
3. **Prompt Engineering:** Design prompts that effectively extract or synthesize relevant medical information from the AI model. Emphasize accuracy and source citation (if the model supports it).
4. **Information Generation:** Generate the required clinical information using the AI model.
5. **Review and Validate:** Critically review the AI-generated information against established medical knowledge and guidelines. **Emphasize that AI output is for support only and not a substitute for professional medical judgment.**
6. **Discussion:** Discuss the ethical considerations and limitations of using generative AI in clinical decision support.

Source Code:

```
// Source code will involve API calls to Gemini, with prompts designed
// to process and summarize medical text or generate structured information.

// Example conceptual prompt for this case study:
// "Summarize the latest clinical guidelines for managing hypertension in
elderly patients,
// focusing on medication classes and lifestyle interventions. Provide key
recommendations."
```

Input:

- **Case Study Scenario:** A doctor needs a quick summary of the latest research on novel treatments for autoimmune hepatitis, specifically regarding new immunosuppressants.
- **Prompt:** "Summarize the key findings and recommendations from the five most recent peer-reviewed articles on novel immunosuppressant treatments for autoimmune hepatitis. Include drug names and efficacy rates."

Expected Output:

- A concise summary of recent research on immunosuppressants for autoimmune hepatitis, including drug names, their reported efficacy, and relevant recommendations.

Lab 15: Case Study III

Title: Case Study III: Generative AI in Healthcare Operations and Administration

Aim: To apply generative AI to optimize administrative tasks and operational workflows within a healthcare setting, such as drafting administrative documents, generating reports, or automating communication.

Procedure:

1. **Understand the Case:** Analyze a provided healthcare administrative scenario (e.g., drafting a hospital policy, generating a patient satisfaction report summary, creating internal communication for staff).
2. **Define the Task:** Determine the specific generative AI task (e.g., draft a policy document outline, summarize feedback from patient surveys, compose an email announcement).
3. **Prompt Engineering:** Design prompts that guide the AI to produce professional, accurate, and contextually appropriate administrative content.
4. **Content Generation:** Generate the required administrative content using the AI model.
5. **Review and Edit:** Review the AI-generated content for clarity, tone, adherence to organizational guidelines, and grammatical correctness. Make necessary revisions.
6. **Discussion:** Discuss the potential for efficiency gains and the challenges of integrating generative AI into existing healthcare administrative workflows.

Source Code:

```
// Source code will involve API calls to Gemini, with prompts structured
// to generate formal documents, reports, or communications.

// Example conceptual prompt for this case study:
// "Draft an internal memo to hospital staff announcing a new protocol for
// electronic health record (EHR) updates. Include the effective date,
// a brief explanation of changes, and where to find training resources."
```

Input:

- **Case Study Scenario:** The hospital administration needs to draft a formal announcement for staff regarding a new mandatory cybersecurity training program.
- **Prompt:** "Draft a formal internal announcement for all hospital staff about a new mandatory cybersecurity training program. Include the purpose of the training, key topics covered, the deadline for completion, and instructions on how to access the training module. Maintain a professional and serious tone."

Expected Output:

- A formal internal announcement suitable for hospital staff, detailing the new cybersecurity training program, its importance, content, deadline, and access instructions.