SRM Institute of Science and Technology

Department of Computer Applications

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

Circular – 2020-21

MCA GAI 1st semester

Prompt Engineering in Generative AI (Lab: Google Generative AI Studio) (PGI20S01J)

Lab Manual

LAB 1 - Apply 5 Principles of Prompting and Generate an Image Prompt

Title: Applying Prompting Principles for Image Generation

Aim: To understand and apply the five core principles of effective prompting to generate a high-quality image prompt using a generative AI model.

Procedure:

1. **Understand the 5 Principles:** Review the fundamental principles of effective prompting (e.g., be specific, provide context, use examples, specify format, iterate and refine).
2. **Choose a Subject:** Select a subject or scene for the image you wish to generate.
3. **Draft Initial Prompt:** Write a basic prompt describing the desired image.
4. **Apply Principles:** Refine the prompt by incorporating the five principles. For example:
    - **Specificity:** Add details about colors, lighting, style, and composition.
    - **Context:** Describe the setting, time of day, or mood.
    - **Examples (if applicable):** While not direct examples in a single prompt, think of visual references or styles.
    - **Format:** Specify the artistic style (e.g., "oil painting," "photorealistic," "cyberpunk").
    - **Iteration:** Experiment with different wordings and details.
5. **Generate Image (Conceptual):** Imagine or conceptually generate the image using the refined prompt. (In a real lab, you would use an image generation tool).
6. **Evaluate:** Assess how well the generated image prompt adheres to the principles and how effectively it describes the desired visual.

Source Code: *(This lab focuses on prompt construction rather than code. The "source code" here is the prompt itself, which would be input into an AI image generation model.)*

```
# Example Image Prompt (Conceptual)
"A majestic, ancient dragon with iridescent scales, soaring through a twilight
sky filled with swirling nebulae and distant stars. The dragon's eyes glow with
an ethereal light, and its wings are vast, catching the last rays of a setting
sun. Photorealistic, high detail, epic fantasy art."
```

Input: *(The input is the thought process and the iterative refinement of the prompt.)*

- Initial idea: "Dragon in sky."
- Refinement steps applying principles.

Expected Output: A well-structured and detailed image prompt that clearly conveys the desired visual, incorporating specificity, context, and stylistic elements, ready to be used by an image generation AI.

LAB 2 - Working with Chat GPT Prompt-I

Title: Introduction to Prompting with ChatGPT

Aim: To familiarize with the basic interaction and prompt formulation techniques for conversational AI models like ChatGPT, focusing on simple information retrieval and text generation.

Procedure:

1. **Access ChatGPT:** Open the ChatGPT interface.
2. **Simple Query:** Ask a straightforward question (e.g., "What is the capital of France?").
3. **Information Retrieval:** Request information on a specific topic (e.g., "Explain the concept of photosynthesis in simple terms.").
4. **Creative Text Generation:** Ask for a short creative piece (e.g., "Write a haiku about a rainy day.").
5. **Constraint-Based Prompting:** Add a simple constraint (e.g., "Summarize the plot of Romeo and Juliet in 50 words.").
6. **Observe Responses:** Analyze the quality, relevance, and coherence of ChatGPT's responses.

Source Code: *(This lab involves direct interaction with a conversational AI, so "source code" refers to the prompts used.)*

```
# Example Prompts
1. "What is the capital of France?"
2. "Explain the concept of photosynthesis in simple terms."
3. "Write a haiku about a rainy day."
4. "Summarize the plot of Romeo and Juliet in 50 words."
```

Input:

- User prompts as listed in the Source Code section.

Expected Output:

- Accurate answers to factual questions.
- Clear and concise explanations of concepts.
- Creative text adhering to specified formats (e.g., haiku).
- Summaries that meet length constraints.

LAB 3 - Working with Chat GPT Prompt-II

Title: Advanced Prompting Techniques with ChatGPT

Aim: To explore more advanced prompting strategies for ChatGPT, including role-playing, persona definition, and multi-turn conversations to achieve more complex and nuanced outputs.

Procedure:

1. **Role-Playing Prompt:** Instruct ChatGPT to act as a specific persona (e.g., "Act as a history professor. Explain the causes of World War I.").
2. **Persona Definition:** Provide a detailed persona for ChatGPT to adopt (e.g., "You are a witty travel blogger. Write a paragraph about visiting Paris.").
3. **Multi-Turn Conversation:** Engage in a conversation where subsequent prompts build upon previous responses (e.g., start with a topic, then ask follow-up questions for clarification or expansion).
4. **Complex Instruction:** Provide a multi-part instruction or a scenario requiring logical reasoning (e.g., "Plan a one-day itinerary for a tourist in London interested in history and art. Include specific landmarks and estimated times.").
5. **Refinement and Iteration:** Experiment with rephrasing prompts or adding more details to guide ChatGPT towards desired outcomes.

Source Code: *(This lab involves direct interaction with a conversational AI, so "source code" refers to the prompts used.)*

```
# Example Prompts
1. "Act as a history professor. Explain the causes of World War I."
2. "You are a witty travel blogger. Write a paragraph about visiting Paris,
focusing on its charm and hidden gems."
3. "Plan a one-day itinerary for a tourist in London interested in history and
art. Include specific landmarks and estimated times."
```

Input:

- User prompts, potentially in a conversational flow.

Expected Output:

- Responses that align with the specified persona or role.
- Coherent and logical multi-turn conversations.
- Detailed and structured outputs for complex instructions.
- Demonstration of the ability to guide the AI's response through advanced prompting.

LAB 4 - Build a Simple Chunking Algorithm in Python

Title: Implementing a Basic Text Chunking Algorithm

Aim: To develop a Python script that divides a large text document into smaller, manageable chunks, which is a common preprocessing step for many NLP tasks.

Procedure:

1. **Define Chunk Size:** Determine the maximum number of words or characters for each chunk.
2. **Load Text:** Read a sample text document into a Python string.
3. **Implement Chunking Logic:**
    - Iterate through the text.
    - Split the text into sentences or paragraphs.
    - Group sentences/paragraphs until the chunk size limit is reached.
    - Ensure chunks do not break in the middle of a sentence or word.
4. **Store Chunks:** Save the generated chunks into a list or array.
5. **Test:** Print the chunks to verify the algorithm's correctness.

Source Code:

```
# chunking_algorithm.py
def simple_chunker(text, max_chunk_size=500, overlap=0):
    """
    Splits text into chunks of a specified maximum size,
    attempting to break at sentence boundaries.
    """
    import re

    # Split text into sentences
    sentences = re.split(r'(?<=[.!?])\s+', text)
    chunks = []
    current_chunk = []
    current_length = 0

    for sentence in sentences:
        sentence_length = len(sentence.split()) # Count words
        if current_length + sentence_length <= max_chunk_size:
            current_chunk.append(sentence)
            current_length += sentence_length
        else:
            chunks.append(" ".join(current_chunk))
            # Implement overlap if needed (for simplicity, no overlap in this
basic version)
            current_chunk = [sentence]
            current_length = sentence_length

    if current_chunk:
        chunks.append(" ".join(current_chunk))

    return chunks

if __name__ == "__main__":
    sample_text = """
    Large language models (LLMs) are advanced AI systems capable of
understanding and generating human-like text.
    They are trained on vast amounts of text data, allowing them to perform a
wide range of natural language processing tasks.
```

```
    These tasks include translation, summarization, question answering, and
creative writing.
    The development of LLMs has revolutionized the field of artificial
intelligence.
    However, they also present challenges related to bias, ethical use, and
computational resources.
    Further research is ongoing to address these issues and enhance their
capabilities.
    """

    # Example usage
    chunks = simple_chunker(sample_text, max_chunk_size=30)
    for i, chunk in enumerate(chunks):
        print(f"Chunk {i+1}:\n{chunk}\n---")
```

Input: A long string of text, for example, an article, a book chapter, or a document.

Expected Output: A list of strings, where each string represents a chunk of the original text, and the length of each chunk (in words or characters, depending on implementation) does not exceed the specified maximum chunk size.

LAB 5 - Implement Encoding and Decoding of Text

Title: Text Encoding and Decoding Implementation

Aim: To implement basic text encoding and decoding mechanisms, demonstrating how text data is converted into a machine-readable format and back, often using schemes like ASCII, UTF-8, or custom mappings.

Procedure:

1. **Choose an Encoding Scheme:** Decide on a simple encoding scheme (e.g., a custom mapping of characters to numbers, or using Python's built-in `encode()`/`decode()` with UTF-8).
2. **Implement Encoding Function:**
    - Take a string as input.
    - Convert each character according to the chosen scheme.
    - Return the encoded representation (e.g., a list of integers, a byte string).
3. **Implement Decoding Function:**
    - Take the encoded representation as input.
    - Convert each encoded unit back to its original character.
    - Return the decoded string.
4. **Test:** Encode a sample text and then decode the result to ensure the original text is recovered accurately. Handle potential errors (e.g., characters not in the mapping).

Source Code:

```
# text_encoder_decoder.py
def custom_encode(text):
    """
    Encodes text using a simple custom mapping (e.g., character to ASCII value).
    """
    encoded_values = [ord(char) for char in text]
    return encoded_values

def custom_decode(encoded_values):
    """
    Decodes a list of ASCII values back to text.
    """
    decoded_text = "".join([chr(val) for val in encoded_values])
    return decoded_text

def utf8_encode(text):
    """
    Encodes text to UTF-8 bytes.
    """
    return text.encode('utf-8')

def utf8_decode(byte_string):
    """
    Decodes UTF-8 bytes back to text.
    """
    return byte_string.decode('utf-8')

if __name__ == "__main__":
    sample_text = "Hello, World! 👋"

    print("--- Custom Encoding/Decoding ---")
    encoded_custom = custom_encode(sample_text)
```

```
print(f"Original: '{sample_text}'")
print(f"Encoded (Custom): {encoded_custom}")
decoded_custom = custom_decode(encoded_custom)
print(f"Decoded (Custom): '{decoded_custom}'")
print(f"Match: {sample_text == decoded_custom}\n")

print("--- UTF-8 Encoding/Decoding ---")
encoded_utf8 = utf8_encode(sample_text)
print(f"Original: '{sample_text}'")
print(f"Encoded (UTF-8): {encoded_utf8}")
decoded_utf8 = utf8_decode(encoded_utf8)
print(f"Decoded (UTF-8): '{decoded_utf8}'")
print(f"Match: {sample_text == decoded_utf8}")
```

Input: A string of text to be encoded and decoded.

Expected Output:

- The encoded representation of the input text (e.g., a list of integers or a byte string).
- The decoded text, which should be identical to the original input text.

LAB 6 - Build a Classification Model

Title: Building a Simple Text Classification Model

Aim: To develop a basic machine learning model capable of classifying text into predefined categories, demonstrating the core steps of data preparation, feature extraction, model training, and evaluation.

Procedure:

1. **Prepare Dataset:** Obtain a small dataset of labeled text (e.g., movie reviews labeled as positive/negative, or news articles labeled by topic).
2. **Preprocess Text:** Clean the text data (e.g., lowercase, remove punctuation, stop words).
3. **Feature Extraction:** Convert text into numerical features using techniques like Bag-of-Words (CountVectorizer) or TF-IDF.
4. **Split Data:** Divide the dataset into training and testing sets.
5. **Choose Model:** Select a simple classification algorithm (e.g., Naive Bayes, Logistic Regression, or a basic scikit-learn classifier).
6. **Train Model:** Train the chosen model on the training data.
7. **Evaluate Model:** Test the model's performance on the unseen test data using metrics like accuracy, precision, recall, and F1-score.

Source Code:

```
# text_classification_model.py
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

def train_and_evaluate_classifier(texts, labels):
    """
    Trains a simple Naive Bayes text classifier and evaluates its performance.
    """
    # 1. Feature Extraction
    vectorizer = TfidfVectorizer(stop_words='english', max_features=1000)
    X = vectorizer.fit_transform(texts)
    y = labels

    # 2. Split Data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    # 3. Choose and Train Model
    model = MultinomialNB()
    model.fit(X_train, y_train)

    # 4. Evaluate Model
    y_pred = model.predict(X_test)

    print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    return model, vectorizer

if __name__ == "__main__":
    # Sample Dataset (simplified for demonstration)
    sample_texts = [
```

```
        "This movie is fantastic and truly enjoyable.",
        "A terrible film, I hated every minute of it.",
        "Great acting and a compelling storyline.",
        "Absolutely boring, a waste of time.",
        "Highly recommended, a masterpiece.",
        "The worst movie I've seen in years.",
        "A delightful experience, very well made.",
        "So bad, I walked out halfway through.",
        "Intriguing plot and strong performances.",
        "Could not finish, it was so dull."
    ]
    sample_labels = [
        "positive", "negative", "positive", "negative", "positive",
        "negative", "positive", "negative", "positive", "negative"
    ]

    print("Training and evaluating text classifier...")
    trained_model, trained_vectorizer =
train_and_evaluate_classifier(sample_texts, sample_labels)

    # Example prediction
    new_texts = [
        "This was an amazing film!",
        "I regret watching this."
    ]
    new_X = trained_vectorizer.transform(new_texts)
    predictions = trained_model.predict(new_X)
    print("\n--- Predictions for new texts ---")
    for text, pred in zip(new_texts, predictions):
        print(f"Text: '{text}' -> Predicted: '{pred}'")
```

Input: A dataset of text documents with corresponding labels.

Expected Output:

- Model performance metrics (e.g., accuracy, precision, recall, F1-score) indicating how well the model classifies new, unseen text.
- A trained classification model capable of predicting labels for new input texts.

LAB 7 - Creating a Generator in LangChain

Title: Building a Text Generator with LangChain

Aim: To create a simple text generation pipeline using LangChain, demonstrating how to integrate a Large Language Model (LLM) for generating coherent and contextually relevant text based on a given prompt.

Procedure:

1. **Install LangChain:** Ensure LangChain and necessary LLM libraries (e.g., `openai` or `google-generativeai`) are installed.
2. **Set up LLM:** Initialize an LLM instance (e.g., `ChatOpenAI` or `ChatGoogleGenerativeAI`) with appropriate API keys.
3. **Define Prompt Template:** Create a `PromptTemplate` to structure the input for the LLM.
4. **Create LLMChain:** Combine the LLM and the prompt template into an `LLMChain`.
5. **Generate Text:** Invoke the chain with an input to generate text.
6. **Experiment:** Try different prompts and observe the generated outputs.

Source Code:

```
# langchain_generator.py
# Note: This code requires LangChain and an LLM provider (e.g., Google
Generative AI or OpenAI).
# Please ensure you have the necessary API key configured (e.g., as an
environment variable).

# For Google Generative AI:
# pip install langchain-google-genai
# For OpenAI:
# pip install langchain-openai

from langchain_core.prompts import PromptTemplate
from langchain_google_genai import ChatGoogleGenerativeAI # Or from
langchain_openai import ChatOpenAI
from langchain.chains import LLMChain
import os

def create_and_run_generator(api_key):
    """
    Creates a simple text generator using LangChain and a Google Generative AI
model.
    """
    # Set up the LLM (using Google Generative AI as an example)
    # Replace with your actual model if different (e.g.,
ChatOpenAI(api_key=api_key))
    llm = ChatGoogleGenerativeAI(model="gemini-pro", google_api_key=api_key)

    # Define a prompt template
    prompt_template = PromptTemplate.from_template(
        "Write a short story about a {character} who discovers a {item} in a
{setting}. The story should be {mood}."
    )

    # Create an LLMChain
    story_chain = LLMChain(llm=llm, prompt=prompt_template)

    # Generate text
    print("Generating a story...")
```

```
    response = story_chain.invoke({
        "character": "young wizard",
        "item": "glowing ancient scroll",
        "setting": "hidden magical library",
        "mood": "mysterious and adventurous"
    })

    print("\n--- Generated Story ---")
    print(response['text'])

if __name__ == "__main__":
    # IMPORTANT: Replace "YOUR_GOOGLE_API_KEY" with your actual Google API key.
    # It's recommended to set this as an environment variable (e.g.,
GOOGLE_API_KEY)
    # and retrieve it using os.getenv("GOOGLE_API_KEY").
    # For demonstration, a placeholder is used.
    google_api_key = os.getenv("GOOGLE_API_KEY", "YOUR_GOOGLE_API_KEY")

    if google_api_key == "YOUR_GOOGLE_API_KEY":
        print("WARNING: Please replace 'YOUR_GOOGLE_API_KEY' with your actual
Google API key or set it as an environment variable.")
        print("Skipping LLM generation for now.")
    else:
        create_and_run_generator(google_api_key)
```

Input:

- A dictionary of variables that fill the `PromptTemplate` (e.g., `character`, `item`, `setting`, `mood`).
- An API key for the chosen LLM.

Expected Output: A generated text (e.g., a short story, a poem, an answer) that is coherent, contextually relevant to the prompt, and demonstrates the LLM's ability to create new content.

LAB 8 - Working with Prompt Template and Vector Database

Title: Integrating Prompt Templates with Vector Databases for RAG

Aim: To demonstrate how to use LangChain's `PromptTemplate` in conjunction with a vector database to implement a basic Retrieval-Augmented Generation (RAG) system, improving the LLM's ability to generate informed responses.

Procedure:

1. **Install Libraries:** Install LangChain, an embedding model library (e.g., `sentence-transformers`), and a vector database library (e.g., `chromadb` or `faiss-cpu`).
2. **Create Embeddings:** Generate numerical embeddings for a set of text documents.
3. **Populate Vector Database:** Store the document embeddings in a vector database.
4. **Define Prompt Template:** Create a `PromptTemplate` that includes a placeholder for retrieved context.
5. **Implement Retrieval:** When a query comes, use the vector database to retrieve relevant document chunks based on semantic similarity.
6. **Combine and Generate:** Inject the retrieved context into the `PromptTemplate` and pass it to the LLM for generation.
7. **Test:** Query the system and observe how the retrieved context influences the LLM's response.

Source Code:

```
# prompt_template_vector_db.py
# Note: This code requires LangChain, an embedding model, and a vector database.
# For simplicity, we'll use a local in-memory ChromaDB and a basic embedding.
# pip install langchain-google-genai langchain-community chromadb sentence-
transformers

from langchain_core.prompts import PromptTemplate
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_community.vectorstores import Chroma
from langchain_community.embeddings import SentenceTransformerEmbeddings
from langchain.chains import RetrievalQA
import os

def run_rag_example(api_key):
    """
    Demonstrates a simple RAG system using LangChain, ChromaDB, and an embedding
model.
    """
    # 1. Sample Documents
    documents = [
        "The quick brown fox jumps over the lazy dog.",
        "Artificial intelligence is a rapidly evolving field.",
        "Machine learning is a subset of AI that focuses on algorithms.",
        "Deep learning is a specialized area within machine learning.",
        "Natural Language Processing (NLP) deals with human language and
computers."
    ]

    # 2. Create Embeddings and Vector Database
    # Using a local embedding model for simplicity
    embedding_function = SentenceTransformerEmbeddings(model_name="all-MiniLM-
L6-v2")
```

```
    # Create a ChromaDB instance from documents
    print("Creating vector database from documents...")
    vectorstore = Chroma.from_texts(documents, embedding_function)
    print("Vector database created.")

    # 3. Set up the LLM
    llm = ChatGoogleGenerativeAI(model="gemini-pro", google_api_key=api_key)

    # 4. Create a RetrievalQA chain
    # This chain handles retrieval from the vectorstore and then passes to the
LLM.
    qa_chain = RetrievalQA.from_chain_type(
        llm=llm,
        chain_type="stuff", # "stuff" means it stuffs all retrieved docs into
the prompt
        retriever=vectorstore.as_retriever()
    )

    # 5. Query the system
    query = "What is machine learning and how is it related to AI?"
    print(f"\nQuery: {query}")
    print("Generating response with RAG...")
    response = qa_chain.invoke({"query": query})

    print("\n--- RAG Response ---")
    print(response['result'])

if __name__ == "__main__":
    google_api_key = os.getenv("GOOGLE_API_KEY", "YOUR_GOOGLE_API_KEY")

    if google_api_key == "YOUR_GOOGLE_API_KEY":
        print("WARNING: Please replace 'YOUR_GOOGLE_API_KEY' with your actual
Google API key or set it as an environment variable.")
        print("Skipping RAG example for now.")
    else:
        run_rag_example(google_api_key)
```

Input:

- A set of text documents to populate the vector database.
- A user query for which an informed response is desired.
- An API key for the chosen LLM.

Expected Output: A generated response from the LLM that leverages the context retrieved from the vector database, providing more accurate and specific information than the LLM might generate on its own.

LAB 9 - Working with FAISS

Title: Implementing Vector Search with FAISS

Aim: To understand and implement FAISS (Facebook AI Similarity Search), a library for efficient similarity search and clustering of dense vectors, which is crucial for large-scale retrieval in RAG systems.

Procedure:

1. **Install FAISS:** Install the `faiss-cpu` or `faiss-gpu` library.
2. **Generate Vectors:** Create a set of sample numerical vectors (e.g., random vectors or embeddings from text).
3. **Build FAISS Index:** Initialize a FAISS index (e.g., `IndexFlatL2` for L2 distance) and add the generated vectors to it.
4. **Perform Search:** Define a query vector and use the FAISS index to find the `k` most similar vectors.
5. **Retrieve Original Data:** Map the indices returned by FAISS back to the original data points (if applicable).
6. **Test:** Verify that the retrieved vectors are indeed the most similar to the query vector.

Source Code:

```
# faiss_example.py
# pip install faiss-cpu numpy

import faiss
import numpy as np

def run_faiss_example():
    """
    Demonstrates basic usage of FAISS for similarity search.
    """
    dimension = 128  # Dimension of vectors
    num_vectors = 1000 # Number of vectors in the database
    num_queries = 5    # Number of query vectors

    # 1. Generate Sample Vectors (Database)
    # These could be text embeddings, image features, etc.
    np.random.seed(42)
    database_vectors = np.random.rand(num_vectors, dimension).astype('float32')

    # Normalize vectors (important for many FAISS indices, especially for dot
product)
    faiss.normalize_L2(database_vectors)

    # 2. Build FAISS Index
    # IndexFlatL2 is a simple index that performs exhaustive search using L2
(Euclidean) distance.
    index = faiss.IndexFlatL2(dimension)

    print(f"Is index trained? {index.is_trained}") # IndexFlatL2 does not need
training
    index.add(database_vectors)
    print(f"Number of vectors in the index: {index.ntotal}")

    # 3. Generate Query Vectors
    query_vectors = np.random.rand(num_queries, dimension).astype('float32')
    faiss.normalize_L2(query_vectors)
```

```
    # 4. Perform Search
    k = 5 # Number of nearest neighbors to retrieve
    print(f"\nSearching for {k} nearest neighbors for {num_queries} queries...")
    distances, indices = index.search(query_vectors, k)

    # 5. Display Results
    for i in range(num_queries):
        print(f"\nQuery {i+1}:")
        print(f"  Query Vector (first 5 dims): {query_vectors[i, :5]}")
        print(f"  Nearest Neighbor Indices: {indices[i]}")
        print(f"  Distances to Neighbors: {distances[i]}")

        # Verify one of the retrieved vectors (e.g., the closest one)
        closest_index = indices[i, 0]
        closest_vector = database_vectors[closest_index]
        print(f"  Closest Vector (first 5 dims): {closest_vector[:5]}")

        # Calculate distance manually to confirm
        manual_distance = np.linalg.norm(query_vectors[i] - closest_vector)
        print(f"  Manual L2 Distance to Closest: {manual_distance:.4f}")
        print(f"  FAISS Reported Distance (L2): {distances[i, 0]:.4f}")

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

Input:

- A set of numerical vectors to be indexed.
- One or more query vectors.
- The desired number of nearest neighbors (`k`).

Expected Output:

- A list of indices corresponding to the `k` nearest neighbors for each query vector.
- A list of distances to these nearest neighbors.
- Demonstration of FAISS's ability to efficiently find similar vectors.

LAB 10 - Implement Simple React

Title: Developing a Basic React Application

Aim: To create a fundamental React application, demonstrating core concepts such as components, JSX, state management, and props, by building a simple interactive UI.

Procedure:

1. **Set up React Environment:** Use `create-react-app` or a similar tool to initialize a new React project.
2. **Create a Functional Component:** Define a simple functional component (e.g., a `Counter` or `Greeting` component).
3. **Use JSX:** Write UI elements using JSX syntax within the component.
4. **Manage State:** Implement `useState` hook to manage component-specific data that changes over time (e.g., a counter value, input text).
5. **Handle Events:** Add event listeners (e.g., `onClick`, `onChange`) to interactive elements.
6. **Pass Props:** If creating multiple components, demonstrate passing data from a parent component to a child component using props.
7. **Render Component:** Render the main component into the DOM.
8. **Style (Optional but Recommended):** Apply basic styling using Tailwind CSS.

Source Code:

```
// App.js
import React, { useState } from 'react';

// A simple functional component that displays a greeting
function Greeting(props) {
  return (
    <div className="p-4 bg-blue-100 rounded-lg shadow-md">
      <h2 className="text-2xl font-semibold text-blue-800">Hello,
{props.name}!</h2>
      <p className="text-blue-700">Welcome to your first React app.</p>
    </div>
  );
}

// A functional component that implements a simple counter
function Counter() {
  const [count, setCount] = useState(0); // Initialize state for count

  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div className="mt-6 p-4 bg-green-100 rounded-lg shadow-md flex flex-col
items-center">
      <h2 className="text-2xl font-semibold text-green-800 mb-2">Counter:
{count}</h2>
      <div className="flex space-x-4">
        <button
          onClick={decrement}
```

```
          className="px-4 py-2 bg-red-500 text-white rounded-lg hover:bg-red-600
transition duration-200"
        >
          Decrement
        </button>
        <button
          onClick={increment}
          className="px-4 py-2 bg-blue-500 text-white rounded-lg hover:bg-blue-
600 transition duration-200"
        >
          Increment
        </button>
      </div>
    </div>
  );
}

// The main App component
function App() {
  return (
    <div className="min-h-screen bg-gray-50 flex flex-col items-center justify-
center p-4 font-sans">
      <h1 className="text-4xl font-bold text-gray-800 mb-8">Simple React
Application</h1>

      {/* Using the Greeting component with a prop */}
      <Greeting name="User" />

      {/* Using the Counter component */}
      <Counter />

      <p className="mt-8 text-gray-600 text-sm">
        This is a basic demonstration of React components, state, and props.
      </p>
    </div>
  );
}

export default App;
```

Input: User interactions (e.g., clicking buttons).

Expected Output: A functional web page displaying:

- A greeting message.
- A counter with increment and decrement buttons.
- The counter value updating dynamically upon button clicks.
- The application should be styled using Tailwind CSS and be responsive.

LAB 11 - Custom and Build the Agent

Title: Developing a Custom LangChain Agent

Aim: To design and implement a custom agent using LangChain, enabling it to reason, use tools, and interact with the environment to achieve specific goals, demonstrating advanced AI capabilities.

Procedure:

1. **Install Libraries:** Ensure LangChain and necessary LLM/tool libraries are installed.
2. **Define Tools:** Create custom tools that the agent can use (e.g., a calculator tool, a search tool, a custom data lookup tool).
3. **Set up LLM:** Initialize an LLM instance to serve as the agent's "brain."
4. **Create Agent:** Use LangChain's agent functionality (e.g., `initialize_agent` with a specific agent type like `zero-shot-react-description`).
5. **Provide Instructions:** Give the agent a clear objective or question.
6. **Run Agent:** Execute the agent and observe its reasoning process (thought, action, observation) and final answer.
7. **Test with Various Queries:** Test the agent's ability to use its tools and reason across different scenarios.

Source Code:

```
# langchain_agent.py
# Note: This code requires LangChain and an LLM provider (e.g., Google
Generative AI or OpenAI).
# It also requires the `numexpr` library for the calculator tool.
# pip install langchain-google-genai langchain-community numexpr

from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import AgentExecutor, create_react_agent
from langchain_core.tools import Tool
from langchain_core.prompts import PromptTemplate
from langchain_community.utilities import PythonREPL
import os

def run_agent_example(api_key):
    """
    Creates and runs a simple LangChain agent with a calculator tool.
    """
    # 1. Set up the LLM
    llm = ChatGoogleGenerativeAI(model="gemini-pro", google_api_key=api_key,
temperature=0)

    # 2. Define Tools
    # A simple calculator tool using PythonREPL
    python_repl = PythonREPL()
    calculator_tool = Tool(
        name="calculator",
        func=python_repl.run,
        description="Useful for when you need to answer questions about math.
Input should be a math expression."
    )
    tools = [calculator_tool]

    # 3. Define the Agent Prompt
    # This prompt guides the agent's reasoning process (ReAct pattern)
    agent_prompt = PromptTemplate.from_template("""
```

```
    You are a helpful AI assistant. You have access to the following tools:

    {tools}

    Use the following format:

    Question: the input question you must answer
    Thought: you should always think about what to do
    Action: the action to take, should be one of [{tool_names}]
    Action Input: the input to the action
    Observation: the result of the action
    ... (this Thought/Action/Action Input/Observation can repeat N times)
    Thought: I now know the final answer
    Final Answer: the final answer to the original input question

    Begin!

    Question: {input}
    Thought:{agent_scratchpad}
    """)

    # 4. Create the Agent
    agent = create_react_agent(llm, tools, agent_prompt)

    # 5. Create Agent Executor
    agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True,
handle_parsing_errors=True)

    # 6. Run the Agent
    print("Running agent with a math question...")
    try:
        result = agent_executor.invoke({"input": "What is 123 * 456 - 789?"})
        print("\n--- Agent Final Answer ---")
        print(result["output"])
    except Exception as e:
        print(f"\nAn error occurred while running the agent: {e}")
        print("Please ensure your API key is correct and the LLM is
accessible.")

if __name__ == "__main__":
    google_api_key = os.getenv("GOOGLE_API_KEY", "YOUR_GOOGLE_API_KEY")

    if google_api_key == "YOUR_GOOGLE_API_KEY":
        print("WARNING: Please replace 'YOUR_GOOGLE_API_KEY' with your actual
Google API key or set it as an environment variable.")
        print("Skipping agent example for now.")
    else:
        run_agent_example(google_api_key)
```

Input:

- A natural language question or instruction for the agent.
- An API key for the chosen LLM.

Expected Output:

- The agent's "thought process" (if `verbose=True`), showing its reasoning steps, tool calls, and observations.
- A final answer to the input question, derived by the agent using its available tools.

LAB 12 - Implement Callbacks with Constructors

Title: Implementing Callbacks in Python with Constructors

Aim: To understand and implement the concept of callbacks in Python, specifically demonstrating how to pass functions as arguments (callbacks) and how to manage them within class constructors.

Procedure:

1. **Define a Class:** Create a class that will use a callback.
2. **Constructor with Callback:** In the class's `__init__` method, accept a function as an argument and store it as an instance variable.
3. **Trigger Callback:** Implement a method within the class that, at a certain point, calls the stored callback function.
4. **Define Callback Functions:** Create several independent functions that can serve as callbacks.
5. **Instantiate Class and Pass Callbacks:** Create instances of the class, passing different callback functions to their constructors.
6. **Test:** Call the method that triggers the callback and observe the execution of the passed function.

Source Code:

```
# callbacks_with_constructors.py

class EventProcessor:
    """
    A class that processes an event and can trigger a callback function.
    The callback is passed during the object's construction.
    """
    def __init__(self, callback_function=None):
        """
        Constructor that accepts an optional callback function.
        """
        self.callback = callback_function
        print(f"EventProcessor initialized. Callback set to:
{callback_function.__name__ if callback_function else 'None'}")

    def process_data(self, data):
        """
        Simulates processing data and then optionally calls the callback.
        """
        print(f"Processing data: '{data}'")
        processed_result = data.upper() # Simulate some processing

        if self.callback:
            print("Triggering callback...")
            self.callback(processed_result) # Call the stored callback with the
result
        else:
            print("No callback function provided.")
        return processed_result

# Define some callback functions
def log_result(result):
    """A callback function that logs the result."""
    print(f"Callback (Logger): Data processed successfully: {result}")

def send_notification(result):
```

```
    """A callback function that simulates sending a notification."""
    print(f"Callback (Notifier): Sending notification for result: {result}")

def store_in_db(result):
    """A callback function that simulates storing data in a database."""
    print(f"Callback (DB Store): Storing '{result}' in database.")

if __name__ == "__main__":
    print("--- Example 1: With log_result callback ---")
    processor1 = EventProcessor(callback_function=log_result)
    processor1.process_data("hello world")
    print("-" * 30)

    print("\n--- Example 2: With send_notification callback ---")
    processor2 = EventProcessor(callback_function=send_notification)
    processor2.process_data("important update")
    print("-" * 30)

    print("\n--- Example 3: With store_in_db callback ---")
    processor3 = EventProcessor(callback_function=store_in_db)
    processor3.process_data("user_data_123")
    print("-" * 30)

    print("\n--- Example 4: Without a callback ---")
    processor4 = EventProcessor()
    processor4.process_data("no callback here")
    print("-" * 30)
```

Input:

- A string of data to be processed by the `EventProcessor`.
- Different callback functions passed during the instantiation of `EventProcessor`.

Expected Output:

- Console output showing the data processing step.
- Subsequent console output from the specific callback function that was passed to the `EventProcessor`'s constructor, demonstrating that the callback was successfully triggered and executed with the processed data.

LAB 13 - Generate AI with Various Format Modifiers

Title: AI Text Generation with Format Modifiers

Aim: To explore and apply various format modifiers in prompts to guide generative AI models to produce text in specific structures, styles, or lengths.

Procedure:

1. **Access Generative AI:** Use a generative AI model (e.g., Google Generative AI Studio, or an LLM via API).
2. **Experiment with Length Modifiers:**
    - Request a short response (e.g., "Summarize in 50 words.").
    - Request a longer response (e.g., "Write a detailed explanation...").
3. **Experiment with Style Modifiers:**
    - Request a formal tone (e.g., "Write a formal email...").
    - Request a casual tone (e.g., "Explain this concept like you're talking to a friend.").
    - Request a specific genre (e.g., "Write a sci-fi paragraph...").
4. **Experiment with Structure Modifiers:**
    - Request a list (e.g., "List 5 benefits of...").
    - Request a table (e.g., "Create a table comparing X and Y.").
    - Request a specific format (e.g., "Write a JSON object with...").
5. **Combine Modifiers:** Try combining multiple modifiers in a single prompt.
6. **Evaluate Output:** Assess how well the AI adheres to the specified format modifiers.

Source Code: *(This lab involves direct interaction with a conversational AI, so "source code" refers to the prompts used.)*

```
# Example Prompts with Format Modifiers
1. "Summarize the history of the internet in exactly 100 words." (Length)
2. "Write a formal letter of complaint about a delayed delivery." (Style)
3. "Explain quantum computing in a simple, conversational tone, suitable for a
high school student." (Style, Audience)
4. "List 3 pros and 3 cons of remote work in bullet points." (Structure: List)
5. "Create a JSON object representing a book with 'title', 'author', and 'year'
fields." (Structure: JSON)
6. "Write a short, suspenseful paragraph about a detective finding a clue, in
the style of a film noir narration." (Length, Style, Genre)
```

Input:

- User prompts incorporating various format modifiers.

Expected Output:

- Text generated by the AI that strictly adheres to the specified length constraints.
- Text that matches the requested tone, style, or genre.
- Text presented in the exact structural format requested (e.g., bullet points, numbered list, table, JSON).

LAB 14 - Generate AI with Various Prompts

Title: Exploring AI Generation with Diverse Prompt Types

Aim: To understand the impact of different prompt types (e.g., open-ended, instructional, conversational, creative) on the output of generative AI models and to practice crafting effective prompts for varied tasks.

Procedure:

1. **Access Generative AI:** Use a generative AI model.
2. **Open-Ended Prompt:** Provide a broad topic and allow the AI to generate freely (e.g., "Write about the future of space exploration.").
3. **Instructional Prompt:** Give clear, step-by-step instructions (e.g., "Explain how a combustion engine works, then list its main components.").
4. **Conversational Prompt:** Engage in a dialogue with the AI, asking follow-up questions (e.g., "Tell me about climate change. What are its primary causes?").
5. **Creative Prompt:** Ask for imaginative or fictional content (e.g., "Invent a new mythical creature and describe its habitat and powers.").
6. **Constraint-Based Prompt:** Add specific rules or limitations (e.g., "Write a short story that ends with the phrase 'and then the lights went out', keeping it under 200 words.").
7. **Problem-Solving Prompt:** Present a scenario and ask the AI for solutions or advice (e.g., "I'm planning a road trip across Europe. What are some essential things to consider?").
8. **Compare Outputs:** Analyze how the AI's response changes based on the prompt type.

Source Code: *(This lab involves direct interaction with a conversational AI, so "source code" refers to the prompts used.)*

```
# Example Prompts
1. "Write a short essay on the importance of renewable energy." (Open-ended)
2. "Provide a step-by-step guide on how to bake a chocolate cake."
(Instructional)
3. "What are the benefits of meditation? Can you give me some simple
techniques?" (Conversational)
4. "Describe a futuristic city powered entirely by sustainable energy."
(Creative)
5. "Write a poem about autumn, using only rhyming couplets." (Constraint-based)
6. "Suggest three innovative solutions for reducing plastic waste in urban
areas." (Problem-solving)
```

Input:

- A variety of prompt types, each designed to elicit a different kind of response from the AI.

Expected Output:

- Diverse outputs from the AI, demonstrating its flexibility in handling different prompt structures and intentions.
- Responses that are appropriate for the given prompt type (e.g., a creative story for a creative prompt, a step-by-step list for an instructional prompt).

LAB 15 - Build AI Powered Applications

Title: Developing AI-Powered Applications

Aim: To integrate generative AI capabilities into a simple application, demonstrating how LLMs can be used to add intelligent features like content generation, summarization, or conversational interfaces to software.

Procedure:

1. **Choose an Application Idea:** Select a simple application idea (e.g., a text summarizer, a simple chatbot, a content idea generator).
2. **Set up Development Environment:** Use a programming language (e.g., Python, JavaScript with Node.js) and a framework (e.g., Flask/Django for Python, React/Vue for frontend).
3. **Integrate LLM API:** Connect to a generative AI API (e.g., Google Generative AI API, OpenAI API) using an SDK or direct HTTP requests.
4. **Design User Interface (if applicable):** Create a basic UI for user input and displaying AI output.
5. **Implement AI Logic:**
    - Capture user input.
    - Formulate a prompt based on the input.
    - Send the prompt to the LLM API.
    - Process the LLM's response.
    - Display the AI-generated content to the user.
6. **Error Handling:** Implement basic error handling for API calls.
7. **Test:** Thoroughly test the application with various inputs to ensure proper functionality and AI response quality.

Source Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>AI-Powered Summarizer</title>
    <script src="https://cdn.tailwindcss.com"></script>
    <link
href="https://fonts.googleapis.com/css2?family=Inter:wght@400;600;700&display=sw
ap" rel="stylesheet">
    <style>
        body {
            font-family: 'Inter', sans-serif;
        }
    </style>
</head>
<body class="bg-gray-100 min-h-screen flex items-center justify-center p-4">
    <div class="bg-white p-8 rounded-xl shadow-lg w-full max-w-2xl">
        <h1 class="text-3xl font-bold text-gray-800 mb-6 text-center">AI-Powered
Text Summarizer</h1>

        <div class="mb-6">
            <label for="inputText" class="block text-gray-700 text-sm font-
medium mb-2">Enter Text to Summarize:</label>
            <textarea
                id="inputText"
```

```
                class="w-full p-3 border border-gray-300 rounded-lg
focus:outline-none focus:ring-2 focus:ring-blue-500 transition duration-200
resize-y"
                rows="8"
                placeholder="Paste your text here..."
            ></textarea>
        </div>

        <button
            id="summarizeBtn"
            class="w-full bg-blue-600 text-white py-3 rounded-lg font-semibold
hover:bg-blue-700 transition duration-200 shadow-md"
        >
            Summarize Text
        </button>

        <div id="loadingIndicator" class="hidden text-center mt-4 text-blue-600
font-medium">
            Summarizing... Please wait.
        </div>

        <div class="mt-8">
            <h2 class="text-xl font-semibold text-gray-800 mb-3">Summary:</h2>
            <div
                id="summaryOutput"
                class="bg-gray-50 p-4 border border-gray-200 rounded-lg min-h-
[100px] text-gray-700 leading-relaxed whitespace-pre-wrap"
            >
                Your summary will appear here.
            </div>
        </div>
    </div>

    <script>
        document.addEventListener('DOMContentLoaded', () => {
            const inputText = document.getElementById('inputText');
            const summarizeBtn = document.getElementById('summarizeBtn');
            const summaryOutput = document.getElementById('summaryOutput');
            const loadingIndicator =
document.getElementById('loadingIndicator');

            summarizeBtn.addEventListener('click', async () => {
                const textToSummarize = inputText.value.trim();

                if (!textToSummarize) {
                    summaryOutput.textContent = "Please enter some text to
summarize.";
                    return;
                }

                loadingIndicator.classList.remove('hidden');
                summaryOutput.textContent = ""; // Clear previous summary

                try {
                    // This is where you would call the Gemini API
                    // Replace with your actual API call logic
                    let chatHistory = [];
                    chatHistory.push({ role: "user", parts: [{ text: `Summarize
the following text concisely: ${textToSummarize}` }] });

                    const payload = { contents: chatHistory };
                    const apiKey = ""; // Canvas will provide this at runtime if
empty
                    const apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-
flash:generateContent?key=${apiKey}`;
```

```
                    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 summary =
result.candidates[0].content.parts[0].text;
                        summaryOutput.textContent = summary;
                    } else {
                        summaryOutput.textContent = "Failed to get a summary.
Please try again.";
                        console.error("Unexpected API response structure:",
result);
                    }

                } catch (error) {
                    summaryOutput.textContent = "An error occurred while
summarizing. Please check your network connection or API key.";
                    console.error("Error during API call:", error);
                } finally {
                    loadingIndicator.classList.add('hidden');
                }
            });
        });
    </script>
</body>
</html>
```

Input:

- Text entered by the user in the input area.

Expected Output:

- A web application with an input text area and a "Summarize" button.
- Upon clicking the button, a concise summary of the input text generated by the integrated AI model will be displayed in the output area.
- A loading indicator will be shown while the AI processes the request.