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

Lab Manual: Building GPT Powered Business Applications (PGI20G06J)

This lab manual provides a structured outline for each program listed, covering the Title, Aim, Procedure, Source Code, Input, and Expected Output sections.

Lab 1: Case Study on NLP Tool

Title: Case Study on NLP Tool

Aim: To understand the fundamental concepts and practical applications of a chosen Natural Language Processing (NLP) tool, and to analyze its capabilities and limitations through a specific use case.

Procedure:

1. **Tool Selection:** Choose a prominent NLP tool (e.g., NLTK, spaCy, Hugging Face Transformers, Google Cloud NLP API).
2. **Installation/Setup:** Install the chosen tool and set up the necessary environment.
3. **Basic Functionality Exploration:** Experiment with core functionalities like tokenization, stemming, lemmatization, part-of-speech tagging, and named entity recognition.
4. **Case Study Implementation:** Select a small text dataset and apply the NLP tool to perform a specific task (e.g., sentiment analysis, text summarization, language translation).
5. **Analysis:** Document the steps, observe the output, and analyze the tool's effectiveness, accuracy, and performance for the given task.
6. **Reporting:** Summarize findings, including strengths, weaknesses, and potential improvements.

Source Code:

```
# Placeholder for Python code demonstrating the use of an NLP tool.
# Example: Using NLTK for tokenization and POS tagging.

import nltk
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag

# Ensure you have the necessary NLTK data downloaded:
# nltk.download('punkt')
# nltk.download('averaged_perceptron_tagger')

text = "Natural Language Processing is a fascinating field."
```

```
tokens = word_tokenize(text)
pos_tags = pos_tag(tokens)

print("Tokens:", tokens)
print("POS Tags:", pos_tags)

# Further code for a specific case study would go here.
```

Input:

```
Sample text for analysis, e.g., "The quick brown fox jumps over the lazy
dog."
```

Expected Output:

```
Depending on the tool and task, typical output could include:
- Tokenized words
- Part-of-speech tags for each word
- Named entities identified
- Sentiment scores (positive, negative, neutral)
- Summarized text
```

Lab 2: Case Study on GPT

Title: Case Study on GPT

Aim: To explore the capabilities of a pre-trained Generative Pre-trained Transformer (GPT) model by interacting with it for various text generation tasks and understanding its underlying principles.

Procedure:

1. **Access GPT Model:** Utilize an accessible GPT model (e.g., via a public API, a local open-source model, or a web interface).
2. **Prompt Engineering Basics:** Formulate different types of prompts to elicit desired responses from the GPT model.
3. **Text Generation Tasks:** Experiment with tasks such as:
    - Creative writing (stories, poems)
    - Question answering
    - Summarization
    - Code generation (simple snippets)
    - Dialogue generation
4. **Output Analysis:** Evaluate the quality, coherence, relevance, and creativity of the generated text. Identify instances of factual inaccuracies or biases.
5. **Parameter Tuning:** If applicable, experiment with generation parameters (e.g., temperature, top-p, max tokens) to observe their effect on output.
6. **Reporting:** Document observations, successful prompts, challenges encountered, and insights into GPT's strengths and limitations.

Source Code:

```
# Placeholder for Python code interacting with a GPT model API.
# This example assumes a hypothetical 'gpt_api_client' library.

# from gpt_api_client import GPTClient

# client = GPTClient(api_key="YOUR_API_KEY")

# prompt = "Write a short story about a robot who discovers art."
# response = client.generate_text(prompt, max_tokens=150, temperature=0.7)

# print(response)

# Note: Actual implementation would depend on the specific GPT API used.
```

Input:

```
Various prompts, e.g.,
- "Explain the concept of quantum entanglement in simple terms."
- "Generate a marketing slogan for a new eco-friendly coffee cup."
- "Continue the following sentence: 'In a galaxy far, far away...'"
```

Expected Output:

```
Generated text based on the prompt, e.g.,
- A simple explanation of quantum entanglement.
- A creative marketing slogan.
- A continuation of the story.
```

Lab 3: Case Study on Prompt Engineering

Title: Case Study on Prompt Engineering

Aim: To understand and apply various prompt engineering techniques to guide large language models (LLMs) to produce more accurate, relevant, and desired outputs for specific tasks.

Procedure:

1. **Task Definition:** Choose a specific task for an LLM (e.g., generating product descriptions, summarizing articles, classifying text).
2. **Baseline Prompting:** Start with simple, direct prompts for the chosen task.
3. **Iterative Prompt Refinement:** Apply different prompt engineering techniques:
    - **Zero-shot prompting:** No examples provided.
    - **Few-shot prompting:** Provide a few input-output examples.
    - **Chain-of-thought prompting:** Instruct the model to think step-by-step.
    - **Role-playing:** Assign a persona to the model.
    - **Constraint-based prompting:** Specify format, length, or content restrictions.
4. **Output Evaluation:** Compare the outputs generated by different prompts against desired criteria (accuracy, completeness, style).
5. **Analysis:** Document which techniques were most effective for the given task and why.
6. **Reporting:** Present the findings, including examples of effective and ineffective prompts, and best practices derived.

Source Code:

```
# Placeholder for Python code demonstrating prompt engineering.
# This would involve making multiple API calls with different prompts.

# Example structure:

# def get_llm_response(prompt):
#     # Call your LLM API here and return the response
#     # For demonstration:
#     return f"LLM response to: '{prompt}'"

# # Zero-shot
# prompt_zero_shot = "Classify the sentiment of 'This movie was terrible.'"
# print("Zero-shot:", get_llm_response(prompt_zero_shot))

# # Few-shot
# prompt_few_shot = """
# Classify the sentiment:
# Text: 'I loved the book.' Sentiment: Positive
# Text: 'It was okay.' Sentiment: Neutral
# Text: 'This movie was terrible.' Sentiment:
# """
# print("Few-shot:", get_llm_response(prompt_few_shot))

# # Chain-of-thought
# prompt_cot = """
# Explain step-by-step how to make a cup of tea, then summarize it.
# """
# print("Chain-of-thought:", get_llm_response(prompt_cot))
```

Input:

```
A series of prompts designed to test different engineering techniques, e.g.,
- "Summarize the following text: [long text]"
- "Translate 'Hello, how are you?' to French."
- "Write a Python function to calculate the factorial of a number."
```

Expected Output:

```
Varied outputs based on prompt engineering, e.g.,
- A concise summary.
- "Bonjour, comment allez-vous ?"
- A correct Python function.
```

Lab 4: Word2Vec Exploration

Title: Word2Vec Exploration

Aim: To understand the concept of word embeddings and explore the functionality of the Word2Vec model for generating vector representations of words, and to visualize word relationships.

Procedure:

1. **Theoretical Understanding:** Review the Skip-gram and CBOW architectures of Word2Vec.
2. **Data Preparation:** Obtain a text corpus (e.g., a collection of news articles, books). Preprocess the text (tokenization, lowercasing, removing stop words).
3. **Model Training:** Train a Word2Vec model on the prepared corpus using a library like Gensim.
4. **Vector Exploration:**
    - Retrieve vector representations for specific words.
    - Calculate cosine similarity between word vectors to find similar words.
    - Perform vector arithmetic (e.g., "king" - "man" + "woman" = "queen").
5. **Visualization:** Use dimensionality reduction techniques (e.g., t-SNE, PCA) to project high-dimensional word vectors into 2D or 3D space for visualization, observing clusters of related words.
6. **Analysis:** Interpret the semantic relationships captured by the word embeddings.
7. **Reporting:** Document the training process, examples of word similarities, vector arithmetic results, and visualizations.

Source Code:

```
# Placeholder for Python code demonstrating Word2Vec using Gensim.

# from gensim.models import Word2Vec
# from nltk.tokenize import word_tokenize
# import nltk
# import matplotlib.pyplot as plt
# from sklearn.manifold import TSNE
# import pandas as pd

# # Ensure NLTK data for tokenization
# # nltk.download('punkt')

# # Sample corpus
# corpus = [
#     "The quick brown fox jumps over the lazy dog.",
#     "I love to read books and articles.",
#     "Dogs are loyal animals.",
#     "Cats are independent pets."
# ]

# # Tokenize and preprocess
# tokenized_corpus = [word_tokenize(sentence.lower()) for sentence in corpus]

# # Train Word2Vec model
# model = Word2Vec(tokenized_corpus, vector_size=100, window=5, min_count=1,
workers=4)

# # Find similar words
# print("Words similar to 'dog':", model.wv.most_similar('dog'))
```

```
# # Example of vector arithmetic (if applicable to your corpus)
# # result = model.wv['king'] - model.wv['man'] + model.wv['woman']
# # print("King - Man + Woman:", model.wv.most_similar([result]))

# # Visualization (simplified)
# # words = list(model.wv.index_to_key)
# # vectors = [model.wv[word] for word in words]

# # tsne_model = TSNE(perplexity=2, n_components=2, init='pca', n_iter=2500,
random_state=23)
# # new_values = tsne_model.fit_transform(vectors)

# # x = []
# # y = []
# # for value in new_values:
# #     x.append(value[0])
# #     y.append(value[1])

# # plt.figure(figsize=(16, 16))
# # for i in range(len(x)):
# #     plt.scatter(x[i], y[i])
# #     plt.annotate(words[i],
# #                  xy=(x[i], y[i]),
# #                  xytext=(5, 2),
# #                  textcoords='offset points',
# #                  ha='right',
# #                  va='bottom')
# # plt.show()
```

Input:

A text corpus (e.g., a few paragraphs of text, or a larger text file).

Expected Output:

- List of words similar to a query word (e.g., words similar to "cat").
- Results of vector arithmetic (e.g., the word closest to "king" - "man" + "woman").
- A 2D or 3D plot showing word clusters.

Lab 5: Applying Tokenization Techniques on text samples

Title: Applying Tokenization Techniques on Text Samples

Aim: To understand and implement various tokenization techniques (e.g., word tokenization, sentence tokenization, subword tokenization) and analyze their impact on text processing for different NLP tasks.

Procedure:

1. **Theoretical Review:** Study different tokenization methods and their use cases (e.g., white-space tokenization, rule-based tokenization, BPE, WordPiece, SentencePiece).
2. **Tool Setup:** Use NLP libraries like NLTK, spaCy, or Hugging Face `transformers` for tokenization.
3. **Text Samples:** Prepare diverse text samples (e.g., formal English, informal chat, code snippets, text with punctuation and numbers).
4. **Implementation:**
    - Apply basic tokenization (e.g., `word_tokenize`, `sent_tokenize` from NLTK).
    - Apply subword tokenization (e.g., using a pre-trained tokenizer from `transformers` library).
    - Experiment with custom tokenization rules if applicable.
5. **Analysis:**
    - Compare the output of different tokenizers on the same text sample.
    - Discuss the advantages and disadvantages of each technique for different types of text and NLP tasks.
    - Observe how punctuation, numbers, and special characters are handled.
6. **Reporting:** Document the code, tokenized outputs, and a comparative analysis of the techniques.

Source Code:

```
# Placeholder for Python code demonstrating tokenization.

# import nltk
# from nltk.tokenize import word_tokenize, sent_tokenize
# from transformers import AutoTokenizer # Requires transformers library

# # Ensure NLTK data for tokenization
# # nltk.download('punkt')

# text_sample_1 = "Hello, world! How are you doing today? I'm fine."
# text_sample_2 = "This is a sentence. And another one! With some numbers
like 123 and symbols like @#$."

# print("--- Word Tokenization (NLTK) ---")
# print("Sample 1:", word_tokenize(text_sample_1))
# print("Sample 2:", word_tokenize(text_sample_2))

# print("\n--- Sentence Tokenization (NLTK) ---")
# print("Sample 1:", sent_tokenize(text_sample_1))
# print("Sample 2:", sent_tokenize(text_sample_2))

# print("\n--- Subword Tokenization (Hugging Face - BERT) ---")
# # Load a pre-trained tokenizer (e.g., BERT base uncased)
# # tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

# # print("Sample 1 (BERT):", tokenizer.tokenize(text_sample_1))
# # print("Sample 2 (BERT):", tokenizer.tokenize(text_sample_2))
```

```
# # For full pipeline, you'd use tokenizer(text_sample_1) to get input IDs,
etc.
```

Input:

```
Various text samples, e.g.,
- "Dr. Smith went to N.Y. on Tuesday."
- "I have a new iPhone 15 Pro Max."
- "This is a long sentence, with multiple clauses, and commas."
```

Expected Output:

```
Tokenized lists of words or subwords, e.g.,
- `['Dr.', 'Smith', 'went', 'to', 'N.Y.', 'on', 'Tuesday', '.']` (word
tokenization)
- `['Hello', ',', 'world', '!', 'How', 'are', 'you', 'doing', 'today', '?',
'I', "'m", 'fine', '.']`
- `['This', 'is', 'a', 'sentence', '.', 'And', 'another', 'one', '!', 'With',
'some', 'numbers', 'like', '123', 'and', 'symbols', 'like', '@', '#', '$',
'.']`
- Subword tokens like `['this', 'is', 'a', 'long', 'sentence', ',', 'with',
'multiple', 'clauses', ',', 'and', 'commas', '.']`
```

Lab 6: Case Study on ChatGPT API

Title: Case Study on ChatGPT API

Aim: To learn how to programmatically interact with the ChatGPT API to build conversational applications, understand its response structure, and manage conversation flow.

Procedure:

1. **API Key Setup:** Obtain an API key from OpenAI and set up the environment for API calls.
2. **Basic API Call:** Make a simple API call to the ChatGPT model with a single user message.
3. **Conversation Management:** Implement logic to maintain conversation history (messages array) to enable multi-turn dialogues.
4. **Role-Playing/System Messages:** Experiment with system messages to guide the model's behavior or persona.
5. **Parameter Tuning:** Adjust parameters like `temperature`, `max_tokens`, and `top_p` to control the creativity and length of responses.
6. **Error Handling:** Implement basic error handling for API calls.
7. **Simple Application:** Build a small interactive console-based chatbot or a web-based interface that uses the ChatGPT API.
8. **Analysis:** Evaluate the conversational quality, coherence, and ability of the model to follow instructions and maintain context.

Source Code:

```
# Placeholder for Python code interacting with the ChatGPT API.
# This example uses the 'openai' library.

# from openai import OpenAI

# # client = OpenAI(api_key="YOUR_OPENAI_API_KEY")

# def chat_with_gpt(messages):
#     try:
#         # response = client.chat.completions.create(
#         #     model="gpt-3.5-turbo", # or "gpt-4"
#         #     messages=messages
#         # )
#         # return response.choices[0].message.content
#         return "Simulated response from ChatGPT API." # For demonstration
#     except Exception as e:
#         return f"Error: {e}"

# # Example of a simple conversation
# # conversation_history = [
# #     {"role": "system", "content": "You are a helpful assistant."},
# #     {"role": "user", "content": "What is the capital of France?"}
# # ]

# # response = chat_with_gpt(conversation_history)
# # print("ChatGPT:", response)

# # # Add assistant's response to history for multi-turn
# # conversation_history.append({"role": "assistant", "content": response})
# # conversation_history.append({"role": "user", "content": "And what is its
main river?"})

# # response_2 = chat_with_gpt(conversation_history)
```

```
# # print("ChatGPT:", response_2)
```

Input:

```
User messages in a conversational format, e.g.,
- "Hi there! How are you?"
- "Can you tell me a joke?"
- "What's the weather like in London today?" (Note: LLMs might not have real-
time data)
```

Expected Output:

```
Conversational responses from ChatGPT, e.g.,
- "Hello! I'm an AI, so I don't have feelings, but I'm ready to assist you!"
- "Why don't scientists trust atoms? Because they make up everything!"
- "I don't have real-time weather data. You might want to check a weather
app."
```

Lab 7: Experimenting with Prompts

Title: Experimenting with Prompts

Aim: To systematically test different prompt structures, phrasing, and content to observe their influence on the output of a large language model, focusing on achieving specific desired outcomes.

Procedure:

1. **Define a Target Task:** Choose a clear task (e.g., generating a product review, writing a short news headline, extracting information from text).
2. **Design Prompt Variations:** Create multiple prompts for the same task, varying:
    - **Clarity and Specificity:** Vague vs. highly detailed instructions.
    - **Tone and Style:** Formal, informal, humorous, professional.
    - **Format Instructions:** Requesting JSON, bullet points, paragraphs.
    - **Contextual Information:** Providing background or examples.
    - **Negative Constraints:** Specifying what *not* to include.
3. **Execute Prompts:** Send each prompt variation to an LLM (e.g., via API or web interface).
4. **Evaluate Outputs:** Compare the generated outputs for each prompt against the desired outcome, noting differences in quality, adherence to instructions, and style.
5. **Identify Best Practices:** Determine which prompt elements consistently lead to better results for the given task.
6. **Reporting:** Present the different prompts and their corresponding outputs, along with an analysis of why certain prompts performed better.

Source Code:

```
# Placeholder for Python code for prompt experimentation.
# This would involve repetitive calls to an LLM API with different prompts.

# def generate_text_with_prompt(prompt_text):
#     # Simulate LLM API call
#     return f"Output for prompt: '{prompt_text}'"

# prompts = [
#     "Write a short, positive review for a new coffee shop.",
#     "Generate a 3-sentence positive review for 'The Daily Grind' coffee
shop, focusing on the atmosphere and coffee quality.",
#     "Act as a satisfied customer. Write a review for 'The Daily Grind'
coffee shop. Mention the cozy atmosphere and excellent espresso. Keep it
under 50 words."
# ]

# for i, prompt in enumerate(prompts):
#     print(f"\n--- Prompt {i+1} ---")
#     print("Prompt:", prompt)
#     print("Output:", generate_text_with_prompt(prompt))
```

Input:

```
A set of carefully crafted prompts for a specific task, e.g.,
- "Write a poem about autumn."
- "Write a haiku about autumn."
- "Write a rhyming poem about autumn, focusing on falling leaves and crisp
air."
```

Expected Output:

```
Varied text generations, each reflecting the nuances of the specific prompt,
e.g.,
- A general poem about autumn.
- A three-line haiku about autumn.
- A rhyming poem with specific thematic elements.
```

Lab 8: Working Functionality of GPT-3

Title: Working Functionality of GPT-3

Aim: To gain hands-on experience with the core functionalities and capabilities of the GPT-3 model, understanding its strengths in various natural language generation and understanding tasks.

Procedure:

1. **Access GPT-3:** Utilize an environment that provides access to GPT-3 (e.g., OpenAI Playground, API access).
2. **Text Completion:** Experiment with basic text completion by providing a starting prompt and observing the model's continuation.
3. **Instruction Following:** Test GPT-3's ability to follow explicit instructions for tasks like summarization, translation, Q&A, and classification.
4. **Code Generation (if applicable):** Provide natural language descriptions and observe if GPT-3 can generate corresponding code snippets.
5. **Creative Generation:** Explore its capacity for creative writing, such as generating stories, poems, or marketing copy.
6. **Parameter Exploration:** Adjust parameters (e.g., `temperature`, `max_tokens`, `frequency_penalty`, `presence_penalty`) to see how they influence the output style and content.
7. **Limitations Observation:** Identify scenarios where GPT-3 might produce irrelevant, nonsensical, or factually incorrect information.
8. **Reporting:** Document various interactions, successful use cases, and observed limitations, providing concrete examples.

Source Code:

```
# Placeholder for Python code demonstrating GPT-3 interaction.
# This would be similar to Lab 6, but focusing on GPT-3's specific
capabilities.

# from openai import OpenAI

# # client = OpenAI(api_key="YOUR_OPENAI_API_KEY")

# def interact_with_gpt3(prompt, model="text-davinci-003", max_tokens=100,
temperature=0.7):
#     try:
#         # response = client.completions.create(
#         #     model=model,
#         #     prompt=prompt,
#         #     max_tokens=max_tokens,
#         #     temperature=temperature
#         # )
#         # return response.choices[0].text.strip()
#         return f"Simulated GPT-3 output for: '{prompt}'" # For
demonstration
#     except Exception as e:
#         return f"Error: {e}"

# # Example prompts
# # print("Completion:", interact_with_gpt3("Once upon a time in a land far
away,"))
# # print("Summarization:", interact_with_gpt3("Summarize the following text:
[long text about climate change]"))
```

```
# # print("Translation:", interact_with_gpt3("Translate 'Bonjour le monde' to
English."))
```

Input:

```
Various prompts designed to test different aspects of GPT-3, e.g.,
- "Generate a list of 5 healthy breakfast ideas."
- "Write a short email to a colleague about meeting rescheduling."
- "Explain the difference between supervised and unsupervised learning."
```

Expected Output:

```
Generated text demonstrating GPT-3's capabilities, e.g.,
- A list of breakfast ideas.
- A well-structured email.
- A clear explanation of machine learning concepts.
```

Lab 9: Experimenting Naïve Bayes

Title: Experimenting Naïve Bayes

Aim: To understand the principles of the Naïve Bayes classification algorithm and implement it for a text classification task, such as spam detection or sentiment analysis.

Procedure:

1. **Theoretical Understanding:** Review Bayes' theorem and the "naïve" independence assumption of the Naïve Bayes classifier.
2. **Dataset Preparation:** Obtain a labeled text dataset suitable for classification (e.g., SMS spam collection, movie review sentiment dataset).
3. **Text Preprocessing:** Clean and preprocess the text data (tokenization, lowercasing, removing stop words, stemming/lemmatization).
4. **Feature Extraction:** Convert text into numerical features using techniques like Bag-of-Words (CountVectorizer) or TF-IDF (TfidfVectorizer).
5. **Model Implementation:**
    - Split the dataset into training and testing sets.
    - Implement a Naïve Bayes classifier (e.g., Multinomial Naïve Bayes from scikit-learn).
    - Train the model on the training data.
6. **Model Evaluation:** Evaluate the model's performance on the test set using metrics such as accuracy, precision, recall, and F1-score.
7. **Analysis:** Discuss the strengths and weaknesses of Naïve Bayes for text classification, considering the independence assumption.
8. **Reporting:** Document the dataset, preprocessing steps, code, evaluation metrics, and conclusions.

Source Code:

```
# Placeholder for Python code demonstrating Naïve Bayes.

# from sklearn.feature_extraction.text import CountVectorizer
# from sklearn.naive_bayes import MultinomialNB
# from sklearn.model_selection import train_test_split
# from sklearn.metrics import classification_report, accuracy_score

# # Sample Data (simplified for demonstration)
# texts = [
#     "I love this movie, it's great!",
#     "This is a terrible film, so boring.",
#     "What a wonderful day, I'm happy.",
#     "The worst experience ever, very bad."
# ]
# labels = ["positive", "negative", "positive", "negative"]

# # 1. Feature Extraction (Bag-of-Words)
# vectorizer = CountVectorizer()
# 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=0.25,
random_state=42)

# # 3. Train Naïve Bayes Classifier
# classifier = MultinomialNB()
# classifier.fit(X_train, y_train)
```

```
# # 4. Make Predictions
# y_pred = classifier.predict(X_test)

# # 5. Evaluate
# print("Accuracy:", accuracy_score(y_test, y_pred))
# print("Classification Report:\n", classification_report(y_test, y_pred))

# # Example prediction on new text
# # new_text = ["This is an amazing product!"]
# # new_text_vectorized = vectorizer.transform(new_text)
# # print("Prediction for new text:",
classifier.predict(new_text_vectorized))
```

Input:

```
A labeled dataset of text and categories, e.g.,
- For spam detection: SMS messages labeled as 'spam' or 'ham'.
- For sentiment analysis: Movie reviews labeled as 'positive' or 'negative'.
```

Expected Output:

```
- Accuracy score of the classifier.
- Precision, Recall, F1-score for each class.
- Classification report.
- Predictions for new, unseen text samples.
```

Lab 10: Simple Chatbot

Title: Simple Chatbot

Aim: To build a basic chatbot that can respond to predefined keywords or patterns using rule-based logic, demonstrating fundamental conversational AI concepts.

Procedure:

1. **Design Conversation Flow:** Map out simple user intents and corresponding chatbot responses.
2. **Keyword/Pattern Recognition:** Implement logic to detect keywords or simple patterns in user input.
3. **Response Generation:** Define a set of static responses for each recognized intent.
4. **Fallback Mechanism:** Include a default response for unrecognized inputs.
5. **Implementation:** Write Python code to create an interactive console-based chatbot.
6. **Testing:** Interact with the chatbot using various inputs to test its responses and robustness.
7. **Analysis:** Evaluate the chatbot's ability to handle simple conversations and identify its limitations (e.g., lack of context, inability to handle complex queries).
8. **Reporting:** Document the chatbot's rules, code, and a summary of its performance.

Source Code:

```
# Placeholder for Python code for a simple rule-based chatbot.

# def simple_chatbot():
#     print("Chatbot: Hello! How can I help you today?")
#     while True:
#         user_input = input("You: ").lower()

#         if "hello" in user_input or "hi" in user_input:
#             print("Chatbot: Hi there! How are you?")
#         elif "how are you" in user_input:
#             print("Chatbot: I'm just a program, but I'm doing great! Thanks
for asking.")
#         elif "weather" in user_input:
#             print("Chatbot: I don't have real-time weather information. You
can check a weather app!")
#         elif "bye" in user_input or "exit" in user_input:
#             print("Chatbot: Goodbye! Have a great day.")
#             break
#         else:
#             print("Chatbot: I'm sorry, I don't understand that. Can you
rephrase?")

# # To run the chatbot:
# # simple_chatbot()
```

Input:

```
User inputs in a conversational style, e.g.,
- "Hello"
- "How are you?"
- "Tell me about the weather."
- "Goodbye"
- "What is your name?"
```

Expected Output:

```
Predefined responses based on keywords, e.g.,
- "Hi there! How are you?"
- "I'm just a program, but I'm doing great! Thanks for asking."
- "I don't have real-time weather information. You can check a weather app!"
- "Goodbye! Have a great day."
- "I'm sorry, I don't understand that. Can you rephrase?"
```

Lab 11: Simple rule-based Chatbot

Title: Simple Rule-Based Chatbot

Aim: To design and implement a more sophisticated rule-based chatbot that utilizes regular expressions or more complex pattern matching to handle a wider range of user inputs and provide more specific responses.

Procedure:

1. **Expand Intent Mapping:** Identify more specific user intents and create detailed rules for each.
2. **Pattern Definition:** Use regular expressions or string matching to define patterns that capture variations of user queries for each intent.
3. **Response Logic:** Develop conditional logic to select appropriate responses based on matched patterns.
4. **Context Handling (Basic):** Implement rudimentary context awareness (e.g., remembering the last topic discussed for a short period).
5. **Implementation:** Write Python code for the rule-based chatbot, incorporating the pattern matching and response logic.
6. **Testing with Edge Cases:** Test the chatbot with various inputs, including synonyms, misspellings (if handled), and slightly ambiguous phrases.
7. **Analysis:** Evaluate the chatbot's ability to interpret user intent and provide relevant responses. Discuss the scalability and maintainability of rule-based systems.
8. **Reporting:** Document the rules, patterns, code, and a critical assessment of the chatbot's performance.

Source Code:

```
# Placeholder for Python code for a simple rule-based chatbot with patterns.

# import re

# def rule_based_chatbot():
#     print("Chatbot: Hello! I'm a rule-based assistant. How can I help?")
#     while True:
#         user_input = input("You: ").lower()

#         if re.search(r"hi|hello|hey", user_input):
#             print("Chatbot: Greetings! What's on your mind?")
#         elif re.search(r"how are you|how's it going", user_input):
#             print("Chatbot: I'm functioning perfectly, thank you!")
#         elif re.search(r"your name|who are you", user_input):
#             print("Chatbot: I am a simple rule-based chatbot.")
#         elif re.search(r"tell me a joke", user_input):
#             print("Chatbot: Why don't scientists trust atoms? Because they
make up everything!")
#         elif re.search(r"bye|goodbye|exit", user_input):
#             print("Chatbot: Farewell! Come back anytime.")
#             break
#         else:
#             print("Chatbot: My apologies, I only understand specific
phrases. Could you try something else?")

# # To run the chatbot:
# # rule_based_chatbot()
```

Input:

Varied user inputs, including synonyms and slightly different phrasing, e.g.,
- "Hey there"
- "How are you doing today?"
- "What is your name?"
- "Can you crack a joke?"
- "I want to leave."

Expected Output:

Responses based on pattern matching, e.g.,
- "Greetings! What's on your mind?"
- "I'm functioning perfectly, thank you!"
- "I am a simple rule-based chatbot."
- "Why don't scientists trust atoms? Because they make up everything!"
- "Farewell! Come back anytime."

Lab 12: Working of GPT-4

Title: Working of GPT-4

Aim: To explore the advanced capabilities of the GPT-4 model, focusing on its improved reasoning, creativity, and instruction-following abilities compared to earlier models, and to identify its potential for complex business applications.

Procedure:

1. **Access GPT-4:** Gain access to GPT-4 (e.g., via OpenAI API or a platform that integrates it).
2. **Complex Instruction Following:** Provide multi-step instructions, constraints, and nuanced requests to test GPT-4's ability to understand and execute complex tasks.
3. **Creative Content Generation:** Experiment with generating long-form content, creative writing, and diverse styles (e.g., scripts, legal documents, marketing campaigns).
4. **Reasoning and Problem Solving:** Present reasoning puzzles, coding challenges, or logical problems to assess its problem-solving skills.
5. **Multimodal Capabilities (if available):** If the API supports it, experiment with image input for visual understanding tasks.
6. **Comparison with Earlier Models:** Qualitatively compare GPT-4's outputs with those from GPT-3.5 or other models for the same prompts.
7. **Ethical Considerations:** Observe and document any biases, hallucinations, or inappropriate content generated, and discuss mitigation strategies.
8. **Reporting:** Provide detailed examples of GPT-4's performance on challenging tasks, highlighting its advancements and potential applications.

Source Code:

```
# Placeholder for Python code interacting with the GPT-4 API.
# This is similar to Lab 6, but explicitly targeting "gpt-4" model.

# from openai import OpenAI

# # client = OpenAI(api_key="YOUR_OPENAI_API_KEY")

# def interact_with_gpt4(messages):
#     try:
#         # response = client.chat.completions.create(
#         #     model="gpt-4", # Specify GPT-4 model
#         #     messages=messages
#         # )
#         # return response.choices[0].message.content
#         return "Simulated response from GPT-4 API." # For demonstration
#     except Exception as e:
#         return f"Error: {e}"

# # Example of a complex prompt for GPT-4
# # complex_prompt = [
# #     {"role": "system", "content": "You are a senior technical writer."},
# #     {"role": "user", "content": "Write a detailed explanation of the
Transformer architecture for a technical audience, including attention
mechanisms and positional encoding. Then, provide a simplified analogy for a
non-technical audience. Structure it with clear headings and bullet points."}
# # ]

# # response_gpt4 = interact_with_gpt4(complex_prompt)
# # print("GPT-4 Output:\n", response_gpt4)
```

Input:

Complex and nuanced prompts, e.g.,
- "Draft a business proposal for a new AI-powered customer support system, including a market analysis, proposed features, and a financial projection."
- "Explain the concept of blockchain to a 10-year-old, then explain it to a university computer science student."
- "Write a Python function that sorts a list of dictionaries by a specific key, and handle cases where the key might be missing."

Expected Output:

High-quality, well-structured, and accurate responses demonstrating GPT-4's advanced capabilities, e.g.,
- A comprehensive business proposal.
- Two distinct explanations of blockchain tailored to different audiences.
- A robust Python function with error handling.

Lab 13: Build a website using Gen AI Tools

Title: Build a Website Using Gen AI Tools

Aim: To leverage generative AI tools to assist in the rapid development of web components, content, and potentially even basic website structures, demonstrating the efficiency gains from AI assistance in web development.

Procedure:

1. **Identify AI Tools:** Research and select generative AI tools relevant to web development (e.g., AI code generators, content generators, image generators for assets).
2. **Define Website Concept:** Outline a simple website concept (e.g., a personal portfolio, a small business landing page, a blog).
3. **Component Generation:**
    - Use AI to generate HTML/CSS snippets for specific components (e.g., navigation bar, hero section, contact form).
    - Use AI to generate placeholder text, headlines, or descriptive paragraphs for the website content.
    - (Optional) Use AI image generation for basic visual assets (e.g., icons, background patterns).
4. **Integration:** Assemble the AI-generated components and content into a cohesive HTML/CSS structure.
5. **Refinement:** Manually adjust and refine the generated code and content to ensure functionality, responsiveness, and aesthetic appeal.
6. **Testing:** Test the website across different devices and browsers for responsiveness and usability.
7. **Analysis:** Evaluate the effectiveness and time-saving potential of using generative AI in the web development workflow.
8. **Reporting:** Document the tools used, the AI-generated elements, the integration process, and a reflection on the experience.

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-Assisted Website</title>
    <script src="https://cdn.tailwindcss.com"></script>
    <link
href="https://fonts.googleapis.com/css2?family=Inter:wght@400;600;700&display
=swap" rel="stylesheet">
    <style>
        body {
            font-family: 'Inter', sans-serif;
        }
    </style>
</head>
<body class="bg-gray-100 text-gray-900">
    <header class="bg-blue-600 text-white p-4 shadow-md rounded-b-lg">
        <nav class="container mx-auto flex justify-between items-center">
            <h1 class="text-2xl font-bold">AI-Powered Solutions</h1>
            <ul class="flex space-x-4">
                <li><a href="#" class="hover:text-blue-200 transition
duration-300">Home</a></li>
```

```
                <li><a href="#" class="hover:text-blue-200 transition
duration-300">Services</a></li>
                <li><a href="#" class="hover:text-blue-200 transition
duration-300">About</a></li>
                <li><a href="#" class="hover:text-blue-200 transition
duration-300">Contact</a></li>
            </ul>
        </nav>
    </header>

    <main class="container mx-auto my-8 p-6 bg-white rounded-lg shadow-lg">
        <section class="text-center mb-12">
            <h2 class="text-4xl font-extrabold text-blue-700 mb-4">Welcome to
Our AI-Driven Platform</h2>
            <p class="text-lg text-gray-700 max-w-2xl mx-auto">
                Leveraging the power of artificial intelligence to transform
your business operations and enhance efficiency.
                Our solutions are designed to be intuitive, scalable, and
highly effective.
            </p>
            <button class="mt-6 px-8 py-3 bg-blue-600 text-white font-
semibold rounded-full shadow-lg hover:bg-blue-700 transition duration-300">
                Learn More
            </button>
        </section>

        <section class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-
8">
            <div class="bg-gray-50 p-6 rounded-lg shadow-md hover:shadow-xl
transition duration-300">
                <h3 class="text-2xl font-semibold text-blue-600 mb-
3">Intelligent Automation</h3>
                <p class="text-gray-600">Automate repetitive tasks and
streamline workflows with our smart AI agents, freeing up your team for
strategic initiatives.</p>
            </div>
            <div class="bg-gray-50 p-6 rounded-lg shadow-md hover:shadow-xl
transition duration-300">
                <h3 class="text-2xl font-semibold text-blue-600 mb-
3">Personalized Experiences</h3>
                <p class="text-gray-600">Deliver tailored content and
recommendations to your users, enhancing engagement and satisfaction through
AI-driven insights.</p>
            </div>
            <div class="bg-gray-50 p-6 rounded-lg shadow-md hover:shadow-xl
transition duration-300">
                <h3 class="text-2xl font-semibold text-blue-600 mb-3">Data-
Driven Decisions</h3>
                <p class="text-gray-600">Gain actionable insights from your
data with advanced AI analytics, enabling smarter and faster business
decisions.</p>
            </div>
        </section>
    </main>

    <footer class="bg-gray-800 text-white p-6 mt-8 rounded-t-lg">
        <div class="container mx-auto text-center">
            <p>&copy; 2025 AI-Powered Solutions. All rights reserved.</p>
        </div>
    </footer>
</body>
</html>
```

Input:

Prompts to the AI tools for specific web components or content, e.g.,
- "Generate HTML for a responsive navigation bar with 4 links."
- "Write a short paragraph about the benefits of AI in business."
- "Create CSS for a stylish button with a hover effect."

Expected Output:

- HTML, CSS, or JavaScript code snippets.
- Text content for website sections.
- (Potentially) Image assets.
- A functional, albeit simple, website assembled from these components.

Lab 14: Gen AI with Custom Dataset

Title: Gen AI with Custom Dataset

Aim: To train or fine-tune a generative AI model on a custom dataset to enable it to generate text or other content that is specific to a particular domain, style, or knowledge base.

Procedure:

1. **Dataset Acquisition:** Obtain or create a custom dataset relevant to the desired generation task (e.g., medical texts, legal documents, specific company FAQs, creative writing in a unique style).
2. **Data Preprocessing:** Clean, format, and prepare the custom dataset for model training (e.g., tokenization, formatting into input-output pairs).
3. **Model Selection:** Choose a suitable generative AI model architecture (e.g., a smaller open-source LLM, or a model designed for fine-tuning).
4. **Training/Fine-tuning:**
    - Set up a training environment (e.g., using Hugging Face Transformers, TensorFlow, PyTorch).
    - Train a new model from scratch or fine-tune a pre-trained model on the custom dataset.
    - Monitor training progress and loss.
5. **Generation and Evaluation:**
    - Generate new content using the trained/fine-tuned model.
    - Evaluate the quality, relevance, and adherence to the custom dataset's characteristics.
    - Compare with a generic model's output if possible.
6. **Analysis:** Discuss the impact of the custom dataset on the model's generation capabilities and identify challenges in data preparation and training.
7. **Reporting:** Document the dataset details, training configuration, code for training/fine-tuning, examples of generated content, and an analysis of the model's performance on the custom domain.

Source Code:

```
# Placeholder for Python code for fine-tuning a Gen AI model.
# This would involve using libraries like Hugging Face Transformers.

# from transformers import AutoTokenizer, AutoModelForCausalLM, Trainer,
TrainingArguments
# from datasets import Dataset

# # 1. Prepare a dummy custom dataset (replace with your actual data)
# # data = {"text": ["This is a custom sentence about apples.", "Another
sentence about oranges.", "A third one about bananas."]}
# # custom_dataset = Dataset.from_dict(data)

# # 2. Load pre-trained tokenizer and model (e.g., GPT-2)
# # model_name = "gpt2"
# # tokenizer = AutoTokenizer.from_pretrained(model_name)
# # model = AutoModelForCausalLM.from_pretrained(model_name)

# # # Add padding token if tokenizer doesn't have one (common for GPT-2)
# # if tokenizer.pad_token is None:
# #     tokenizer.pad_token = tokenizer.eos_token

# # 3. Tokenize the dataset
# # def tokenize_function(examples):
```

```
# #     return tokenizer(examples["text"], truncation=True,
padding="max_length", max_length=128)

# # tokenized_dataset = custom_dataset.map(tokenize_function, batched=True)

# # # Set format for PyTorch
# # tokenized_dataset.set_format(type="torch", columns=["input_ids",
"attention_mask"])

# # 4. Define training arguments
# # training_args = TrainingArguments(
# #     output_dir="./results",
# #     num_train_epochs=3,
# #     per_device_train_batch_size=2,
# #     save_steps=500,
# #     save_total_limit=2,
# #     logging_dir="./logs",
# #     logging_steps=100,
# # )

# # 5. Create Trainer and fine-tune
# # trainer = Trainer(
# #     model=model,
# #     args=training_args,
# #     train_dataset=tokenized_dataset,
# # )

# # trainer.train()

# # 6. Generate text with the fine-tuned model
# # prompt = "Apples are"
# # input_ids = tokenizer.encode(prompt, return_tensors="pt")
# # output = model.generate(input_ids, max_length=50, num_return_sequences=1)
# # generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
# # print("Generated Text:", generated_text)
```

Input:

```
A custom dataset in text format (e.g., a .txt file, a CSV with text columns).
Example: A collection of medical research papers, or a list of product
descriptions from an e-commerce site.
```

Expected Output:

```
- A trained/fine-tuned generative model.
- New text generated by the model that reflects the style, vocabulary, and
content of the custom dataset.
- Evaluation metrics if applicable (e.g., perplexity, loss).
```

Lab 15: Working of Conversational AI

Title: Working of Conversational AI

Aim: To understand the end-to-end architecture and workflow of a conversational AI system, encompassing natural language understanding (NLU), dialogue management, and natural language generation (NLG), and to build a simple conversational agent.

Procedure:

1. **Architectural Overview:** Study the components of a conversational AI system (e.g., intent recognition, entity extraction, dialogue state tracking, response generation).
2. **NLU Implementation:**
    - Define intents (e.g., "order_pizza", "check_status", "greet").
    - Define entities (e.g., "pizza_type", "order_id").
    - Train an NLU model (e.g., using Rasa NLU, spaCy, or a simple rule-based approach) to recognize intents and extract entities from user input.
3. **Dialogue Management:**
    - Design a dialogue flow that dictates how the chatbot responds based on recognized intents and the current conversation state.
    - Implement state tracking to remember context across turns.
4. **NLG Implementation:** Define how the chatbot generates responses, either through predefined templates or by leveraging a generative model.
5. **Integration:** Combine NLU, dialogue management, and NLG components into a working conversational agent.
6. **Testing and Iteration:** Conduct extensive testing with various user inputs, including out-of-scope queries and complex multi-turn dialogues. Iterate on the NLU model and dialogue logic based on testing feedback.
7. **Analysis:** Evaluate the conversational AI's ability to understand user intent, manage dialogue, and provide coherent responses. Discuss the challenges of building robust conversational systems.
8. **Reporting:** Document the system architecture, NLU training data, dialogue flow, code, and a critical analysis of the conversational AI's performance and limitations.

Source Code:

```
# Placeholder for Python code for a simple conversational AI (conceptual).
# This would typically involve a framework like Rasa, or custom NLU/DM/NLG.

# import re

# class ConversationalAI:
#     def __init__(self):
#         self.context = {} # To store conversation context
#         self.intents = {
#             "greet": r"hi|hello|hey",
#             "order_pizza": r"i want to order pizza|order a pizza|pizza
please",
#             "check_status": r"what is my order status|check my order",
#             "goodbye": r"bye|goodbye|exit"
#         }
#         self.entities = {
#             "pizza_type": r"(pepperoni|margherita|vegetarian)"
#         }

#     def recognize_intent(self, user_input):
#         for intent, pattern in self.intents.items():
#             if re.search(pattern, user_input):
```

```
#                 return intent
#         return "unknown"

#     def extract_entities(self, user_input, intent):
#         extracted = {}
#         if intent == "order_pizza":
#             match = re.search(self.entities["pizza_type"], user_input)
#             if match:
#                 extracted["pizza_type"] = match.group(1)
#         return extracted

#     def generate_response(self, intent, entities):
#         if intent == "greet":
#             return "Hello! How can I help you with your order?"
#         elif intent == "order_pizza":
#             pizza_type = entities.get("pizza_type", "a pizza")
#             self.context["last_order_type"] = pizza_type
#             return f"Sure, I can help you order {pizza_type}. What size
would you like?"
#         elif intent == "check_status":
#             return "I need your order ID to check the status."
#         elif intent == "goodbye":
#             return "Goodbye! Enjoy your meal."
#         else:
#             return "I'm sorry, I didn't understand that. Can you please
clarify?"

#     def chat(self, user_input):
#         user_input_lower = user_input.lower()
#         intent = self.recognize_intent(user_input_lower)
#         entities = self.extract_entities(user_input_lower, intent)
#         response = self.generate_response(intent, entities)
#         return response

# # To run the conversational AI:
# # ai = ConversationalAI()
# # print("Chatbot:", ai.chat("Hi"))
# # print("Chatbot:", ai.chat("I want to order a pepperoni pizza"))
# # print("Chatbot:", ai.chat("What is my order status?"))
# # print("Chatbot:", ai.chat("Bye"))
```

Input:

```
Conversational user inputs, e.g.,
- "Hi"
- "I want to book a flight."
- "From New York to London."
- "On what date?"
- "Tomorrow."
- "Thank you."
```

Expected Output:

```
Multi-turn conversational responses, demonstrating:
- Intent recognition (e.g., "greet", "book_flight", "provide_details").
- Entity extraction (e.g., "origin: New York", "destination: London", "date:
tomorrow").
- Dialogue state management (remembering origin/destination).
- Coherent and relevant responses based on the conversation flow.
```