

LAB ASSIGNMENT 4.1

NAME : Harshita Anupoju

2303A52211

SUBJECT : AI ASSISTED CODING

The screenshot shows a Jupyter Notebook interface in Google Colab. The code cell contains the following Python script:

```
# Classify customer emails into categories
simple_email = """
Hello, I am charged value for my subscription this month. "Billing"
My application creates whenever I open it. "Technical Support"
I really like the new feature. "Feedback"
What are your customer support working hours? "Others"
I am unable to login to my account. "Technical Support"

# Prompt classification
def classify_email():
    print("Simple Email")
    print("Billing")
    print("Feedback")
    print("Others")
    print("Technical Support")

# Main loop
while True:
    user_input = input("Enter email content: ")
    if user_input == "exit":
        break
    else:
        classify_email()
        print("\n")
```

The screenshot shows a Jupyter Notebook interface in Google Colab. The code cell contains the following Python script:

```
# Define a function to classify email
def classify_email():
    print("Classify the following customer email into one of three categories: Billing, Technical Support, Feedback, Others.")
    user_input = input("Hello, I am charged value for my subscription this month. ")
    if user_input == "Billing":
        print("Billing")
    elif user_input == "Feedback":
        print("Feedback")
    else:
        print("Others")

# Main loop
while True:
    user_input = input("Enter email content: ")
    if user_input == "exit":
        break
    else:
        classify_email()
        print("\n")
```

```
Job Assignment 4 | google-mail - Search | Untitled0.pyw - Code | Advanced Prompt Engineering | Python3 (100%) - Coding
https://docs.google.com/drive/u/0/p/p/1BQZ2aC_CMrHmzg4KvU7kmTt-187WfUFV9B
File Edit View Insert Home Tools Help
O, Compose → Code → Mail → Mail...
I More
documents
mails["My application status frequently."]
Category: Technical Support

Recalibrate the following mail into:
Billing, Technical Support, Feedback, or Others.

mail1: "I usually like the new update. Great job!"

REBIL: BOTTTFY:
Feedback

mails_new_group = []
mails[2]
mail2["I was charged today for my subscription."]
Category: Billing

Example 1:
mail3["My tax credits after the update."]
Category: Technical Support

Example 3:
mail4["Great service and fast response."]
Category: Feedback

Recalibrate the following mail1:
mail1["How are your customer support working hours?"]
Category: Others

run_group(mails_new_group, "Others")

REPORT:
Example 1:
mail1["I was charged twice for my subscription."]
Category: Billing

Example 3:
```

```
# Lab Assignment 0.1                                     Search                                     Untitled Slides - Code                                     Advanced Python (beginning)                                     Python with GUI Using

File Edit View Insert Runtime Tools Help
File Commands + Code + Run + Reset
formula = []
mail = "We are excited after the update."
category = "Technical Support"

formula = []
mail = "Great service and fast response."
category = "Feedback

You classify the following email.
mail = "What are your customer support setting here?"  
  
MAIL SUBJECT: ...  
SUBJECT:  
  
import re  
def classification(mail):  
    if re.search("Great service and fast response.", mail):  
        return "Feedback"  
    elif re.search("What are your customer support setting here?", mail):  
        return "Technical Support"  
    else:  
        return "Informational"  
    else:  
        return "Informational"  
  
print(classification(mail))  
for mail in observation:  
    print(classification(mail))  
  
Explanation of Prompting Techniques:  
The code starts by defining a function named 'classification' which takes a single argument 'mail'. Inside the function, it uses a regular expression to search for specific patterns in the email content. If it finds the pattern for 'fast response', it returns 'Feedback'. If it finds the pattern for 'customer support setting', it returns 'Technical Support'. If neither pattern is found, it returns 'Informational'. Finally, the code prints the classification for each email in the 'observation' list.  
  
A simple interview question with answer:  
simply_answer = [  
    "Can you tell me about your previous projects?",  
    "What is your strength?", "What weakness?",  
    "What are your current support levels?", "What's your goal?",  
    "Why did you join us?", "What do you expect from us?",  
    "What will you offer to our team?", "What's your target?"  
]
```

The screenshot shows a browser window with several tabs open, all related to AI development and prompt engineering. The main content area is a Jupyter Notebook cell containing Python code and JSON data. The code includes functions for generating responses based on user input and for classifying text. The JSON data defines different response types: 'Answer', 'Text', 'Image', 'Video', 'Audio', and 'Question'. The terminal window below shows examples of how these prompts are used to generate responses like 'Hello, I can't read my account password.' and 'Sorry, there's no video right now!'. The bottom of the screen shows the Windows taskbar with various icons.

```
print("Simple question: {question[0]}")
for query, intent in model._queries:
    print(f"Query: {query[0]}")
    print(f"Intent: {intent[0]}")

# Example function signature
def get_answer(query):
    # Query: I can't read my account password.
    # Intent: Account issue

    # Query: There's no video right now!
    # Intent: Other issue

    # Query: Does this phone support 5G?
    # Intent: Product Inquiry

    # Query: What are your customer support hours?
    # Intent: Service Inquiry

    # Query: My account is locked.
    # Intent: Account issue

    # Query: When will my order be delivered?
    # Intent: Order Status
```

```
def gen_prompt(prompt, intent):
    print("Simple prompt:")
    print("{'text':")
    print(f"\"{prompt}\",")
    print(f"\"{intent}\",")
    print("}")
    print("Simple prompt + intent")
    print("Classify one following sentence into one of these intents:")
    print("Answer, Text, Image, Video, Audio, or Question:")
    print("Query: Does this phone support 5G?")
    print("Intent: Product Inquiry")
```

```
 1  // Lab Assignment 4.1
 2  // | https://datascience.googleusercontent.com/1UPropTIBG3kK_C0HfNeweLdXkzvmtIz-17W7sfUW8
 3  // Advanced Prompt Engineering
 4  // Python (DETA) Coding
 5
 6  #!/usr/bin/python3
 7
 8  import requests
 9  import json
10
11  # Set up the API endpoint URL
12  url = "https://api-inference.huggingface.co/models/llama-2-7b-hf"
13
14  # Set up the headers for the API request
15  headers = {"Authorization": "Bearer "}
16
17  # Set up the prompt for the API request
18  prompt = "What is the capital of France?"
19
20  # Make the API request
21  response = requests.post(url, headers=headers, json={"prompt": prompt})
22
23  # Print the response from the API
24  print(response.json())
25
26
27  # Set up the API endpoint URL
28  url = "https://api-inference.huggingface.co/models/llama-2-7b-hf"
29
30  # Set up the headers for the API request
31  headers = {"Authorization": "Bearer "}
32
33  # Set up the prompt for the API request
34  prompt = "What is the capital of France?"
```

The screenshot shows a Jupyter Notebook interface with several tabs at the top: 'Untitled Document - Colab', 'Untitled Document - Search', 'Untitled Document - Colab', 'Advanced Python (beginning)', and 'Pandas with Python Using'. The main area contains Python code for generating feedback based on student responses.

```
# & Untitled0.ipynb * colab
File Edit View Insert Variables Tools Help
Cell Commands + Code + Text + Markdown + More
See classify the following story:
Story: "There is no water right now."
ANSWER: None
Enter Solution
evaluator = [
    "Incorrect", "Incorrect for class purposes, but may struggle with ambiguity",
    "Incomplete", "Depresses student's clarity by showing one example",
    "Incorrect", "Most accurate and convincing due to multiple correct examples"
]

print("Evaluation of Prompting Techniques:")
for method in evaluator:
    print(f"\n{method}: {evaluator[method]}")

Evaluation of Prompting Techniques:
Incomplete: Incorrect for class purposes, but may struggle with ambiguity
Incomplete: Depresses student's clarity by showing one example
Incomplete: Most accurate and convincing due to multiple correct examples

# sample student feedback with continue logic
sample_feedback = [
    ("The instructions explained concepts very clearly.", "Positive"),
    ("The math sentence was written accurately.", "Positive"),
    ("The solution was accurate.", "Positive"),
    ("Distinguish between helpful and unhelpful.", "Positive"),
    ("Comments were having just near to hollow.", "Negative")
]

print("Sample Student Feedback:")
for feedback in sample_feedback:
    print(f"\n{feedback[0]}, sentiment: {feedback[1]}")
```

The screenshot shows a browser window with several tabs open. The active tab is titled "Untitled202307 - Code" and contains code for analyzing student feedback. The code uses the `ref` function to process feedback from different sources (student, teacher, peer) and classifies it as Positive, Negative, or Neutral. It also includes a function to analyze a list of student feedback.

```
def ref_prompts(prompt, intent):
    print("intent:", intent)
    print("prompt:", prompt)
    print("refined:", refine(prompt))
    print("-----\n")

def analyze_feedback(feedback):
    for source in feedback:
        for intent in source:
            for item in source[intent]:
                ref_prompts(item, intent)

analyze_feedback([
    {
        "student": [
            {
                "text": "Feedback: The instructor explained concepts very clearly.", "intent": "Positive"
            },
            {
                "text": "Feedback: Excellent.", "intent": "Positive"
            },
            {
                "text": "Feedback: Very good overall and makes sense.", "intent": "Positive"
            },
            {
                "text": "Feedback: Negative.", "intent": "Negative"
            },
            {
                "text": "Feedback: Good explanations and concepts.", "intent": "Positive"
            },
            {
                "text": "Feedback: Assignments were helpful, but will remember.", "intent": "Positive"
            },
            {
                "text": "Feedback: Lectures were boring, and hard to follow.", "intent": "Negative"
            }
        ],
        "teacher": [
            {
                "text": "Refined prompt: Explain concepts clearly.", "intent": "Positive"
            },
            {
                "text": "Refined prompt: Excellent.", "intent": "Positive"
            },
            {
                "text": "Refined prompt: Very good overall and makes sense.", "intent": "Positive"
            },
            {
                "text": "Refined prompt: Negative.", "intent": "Negative"
            },
            {
                "text": "Refined prompt: Good explanations and concepts.", "intent": "Positive"
            },
            {
                "text": "Refined prompt: Assignments were helpful, but will remember.", "intent": "Positive"
            },
            {
                "text": "Refined prompt: Lectures were boring, and hard to follow.", "intent": "Negative"
            }
        ]
    }
])
```

The screenshot shows a Jupyter Notebook interface with three examples of feedback classification. Each example includes a code snippet, an example sentence, feedback text, and a sentiment label.

Example 1:

```
new_chat_prompt = ""
```

Feedback: "Participants were helpful and well designed."

Sentiment: Positive

Example 2:

```
new_chat_prompt = ""
```

Feedback: "Participants were boring and had no follow-up."

Sentiment: Negative

Example 3:

```
new_chat_prompt = ""
```

Feedback: "The instructor was average."

Sentiment: Neutral

Example 4:

```
new_chat_prompt = ""
```

Feedback: "Participants were helpful and well designed."

Sentiment: Positive

Example 5:

```
new_chat_prompt = ""
```

Feedback: "The instructor explained concepts very clearly."

Sentiment: Positive

Example 6:

```
new_chat_prompt = ""
```

Feedback: "The instructor was average."

Sentiment: Neutral

```
# Lab Assignment 0.1           Search           Untitled Slides - Code           Advanced Python (beginning)           Python for Data Using

```

```
class DataClass:
    def one_shot_group(self, query):
        policy = self.get_policy()
        print(query)
        print("one shot group")
        print(policy["one_shot_group"])
        print("-----\n")

    def one_shot_project(self, query):
        classify = self.get_classifier()
        print(query)
        print("one shot project")
        print(classify["one_shot_project"])
        print("-----\n")

    def get_policy(self):
        return {
            "one_shot_group": {
                "basic": "I want basic data and want to impress.",
                "intermediate": "I want to learn deep learning algorithms."
            },
            "one_shot_project": {
                "basic": "I want basic data and want to impress.",
                "intermediate": "I want to learn deep learning algorithms."
            }
        }

    def get_classifier(self):
        return {
            "one_shot_group": {
                "basic": "I want basic data and want to impress.",
                "intermediate": "I want to learn deep learning algorithms."
            },
            "one_shot_project": {
                "basic": "I want basic data and want to impress.",
                "intermediate": "I want to learn deep learning algorithms."
            }
        }
```

```
class DataClass:
    def one_shot_group(self, query):
        policy = self.get_policy()
        print(query)
        print("one shot group")
        print(policy["one_shot_group"])
        print("-----\n")

    def one_shot_project(self, query):
        classify = self.get_classifier()
        print(query)
        print("one shot project")
        print(classify["one_shot_project"])
        print("-----\n")

    def get_policy(self):
        return {
            "one_shot_group": {
                "basic": "I want basic data and want to impress.",
                "intermediate": "I want to learn deep learning algorithms."
            },
            "one_shot_project": {
                "basic": "I want basic data and want to impress.",
                "intermediate": "I want to learn deep learning algorithms."
            }
        }

    def get_classifier(self):
        return {
            "one_shot_group": {
                "basic": "I want basic data and want to impress.",
                "intermediate": "I want to learn deep learning algorithms."
            },
            "one_shot_project": {
                "basic": "I want basic data and want to impress.",
                "intermediate": "I want to learn deep learning algorithms."
            }
        }
```

```
Lab Assignment 4.1 - google.colab - Search Untitled0.pyrb - Code Advanced Prompt Engineering Prantha1024-Coding
```

```
File Edit View Insert Runtime Tools Help
```

```
Untitled0.pyrb
```

```
File Edit View Insert Runtime Tools Help
```

```
Q Commands → Code → Run ⌘ R
```

```
How classify the following user prompt:  
Query: "I want to improve my coding logic."
```

```
run_prompt(query, prompt_type="beginner")
```

```
How???
```

```
Example 1:  
Query: "I am new to programming."  
Level: Beginner
```

```
Example 2:  
Query: "I have experience with data structures."  
Level: Intermediate
```

```
Example 3:  
Query: "I want to optimize memory usage."  
Level: Advanced
```

```
How classify the following user prompt:  
Query: "I want to improve my coding logic."
```

```
print("Beginner")
```

```
Intermediate
```

```
Discussions ↗
```

```
Improve prompting improves overall communication quality by:
```

1. Clearly defining skill boundaries between levels.
2. Providing reference patterns for lower levels.
3. Reducing ambiguity in queries with unclear requirements.
4. Producing more consistent and accurate classifications.

```
print("Beginner")
```

```
Improve prompting improves overall communication quality by:
```

```
Q Variables Q Terminal
```

```
21:59PM 14:22 10-01-2024
```

```
Lab Assignment 4.1 - google.colab - Search Untitled0.pyrb - Code Advanced Prompt Engineering Prantha1024-Coding
```

```
File Edit View Insert Runtime Tools Help
```

```
Untitled0.pyrb
```

```
File Edit View Insert Runtime Tools Help
```

```
Q Commands → Code → Run ⌘ R
```

```
How classify the following user prompt with categories:  
Query: "I am a beginner developer. I want to learn Python. Can you suggest any resources? I am new to programming."
```

```
simple_query("I am a beginner developer. I want to learn Python. Can you suggest any resources? I am new to programming.", "beginner")
```

```
How???
```

```
Example 1:  
I am a beginner developer. I want to learn Python. Can you suggest any resources? I am new to programming.  
Category: Beginner
```

```
Example 2:  
I am a beginner developer. I want to learn Python. Can you suggest any resources? I am new to programming.  
Category: Intermediate
```

```
Example 3:  
I am a beginner developer. I want to learn Python. Can you suggest any resources? I am new to programming.  
Category: Advanced
```

```
How classify the following user prompt:  
Query: "I have built an app; it works perfectly. Categories: available."
```

```
print("Available")
```

```
How???
```

```
Example 1:  
I have built an app; it works perfectly. Categories: available.
```

```
Example 2:  
I have built an app; it works perfectly. Categories: available.
```

```
Example 3:  
I have built an app; it works perfectly. Categories: available.
```

```
How classify the following user prompt:  
Query: "I want to learn Python. Can you suggest any resources? I am new to programming."
```

```
print("Beginner")
```

```
Intermediate
```

```
Advanced
```

```
Discussions ↗
```

```
Improve prompting improves overall communication quality by:
```

1. Clearly defining skill boundaries between levels.
2. Providing reference patterns for lower levels.
3. Reducing ambiguity in queries with unclear requirements.
4. Producing more consistent and accurate classifications.

```
print("Beginner")
```

```
Intermediate
```

```
Advanced
```

```
Q Variables Q Terminal
```

```
21:59PM 14:22 10-01-2024
```

A screenshot of a web browser window titled "Untitled202307 - Code". The URL is https://colab.research.google.com/drive/1PtpjD8Z2eK_C9THewzg4KQnU79mzTc-U7WfUFV9B. The browser interface includes tabs for "Job Assignment 41", "Advanced Prompt Engineering", and "Python3 (TODAY)-Coding". Below the tabs are standard browser controls like back, forward, and search.

The main content area displays a Python script:

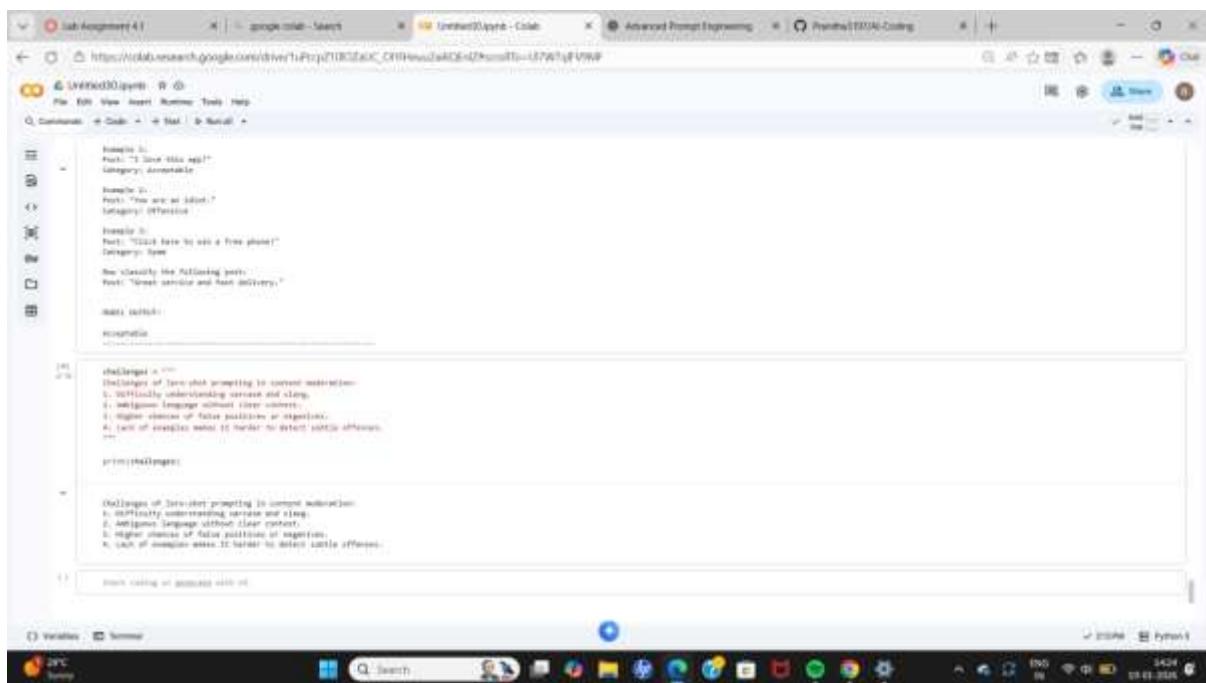
```
def run_prompt(prompt, method):
    print("Method: " + method)
    print("Prompt: " + prompt)
    print("Output: " + method(prompt))
    print("\n", "-" * 40)

# ---#
# user_chat_group = """
# Classify the following social media post as:
# Inappropriate, Offensive, or Spam.
#
# Note: You can get 100 API calls here!
# """
#
# run_prompt(user_chat_group, "spam")

# spam:
#
# classify the following social media post as:
# Inappropriate, offensive, or spam.
#
# Note: You can get 100 API calls here!
# """
#
# run_prompt(user_chat_group, "spam")

# INAPPROPRIATE OUTPUT:
# spam

# ---#
# user_chat_group = """
# Greetings!
# Note: You are stupid and useless.
# Greetings: offensive
#
# Note: Classify the following post:
# Note: Note a first phone by clicking this link!
# """
#
# run_prompt(user_chat_group, "spam")
```



Final Observation for Problem Statement 1

Zero-shot prompting works well for straightforward emails.

One-shot prompting improves understanding by providing context.

Few-shot prompting gives the best performance by clearly defining category boundaries and reducing ambiguity.

Final Observation for Problem Statement 2

Zero-shot prompting works for simple and explicit queries.

One-shot prompting improves understanding with minimal context.

Few-shot prompting provides the best performance by clearly defining intent boundaries and reducing ambiguity.

Final Observation for Problem Statement 3

Zero-shot prompting works for clearly emotional feedback.

One-shot prompting improves understanding with minimal guidance.

Few-shot prompting gives the best accuracy by clearly defining positive, negative, and neutral sentiment patterns

Final Observation for Problem Statement 4

Zero-shot prompting works for very clear beginner or advanced queries.

One-shot prompting improves classification with minimal guidance.

Few-shot prompting provides the best results by clearly distinguishing between beginner, intermediate, and advanced learning needs.

Final Observation for Problem Statement 5

Zero-shot prompting works for clearly spam or offensive posts.

However, it struggles with ambiguity and sarcasm.

One-shot improves clarity, while Few-shot prompting gives the most accurate and reliable moderation results.