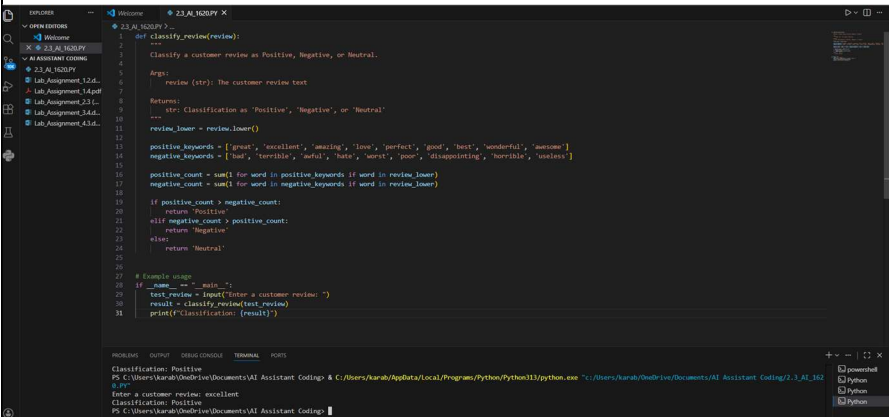


**K.Akshay**  
**2303A51817**  
**B-26**  
**Assignment- 4.4**

| Q.No. | Question  | Expected Time to complete |
|-------|---|---------------------------|
| 1     | <p><b>1. Sentiment Classification for Customer Reviews</b></p> <p><b>Scenario:</b></p> <p>An e-commerce platform wants to analyze customer reviews and classify them into Positive, Negative, or Neutral sentiments using prompt engineering.</p> <p><b>Tasks:</b></p> <ol style="list-style-type: none"> <li>Prepare 6 short customer reviews mapped to sentiment labels.</li> <li>Design a Zero-shot prompt to classify sentiment.</li> <li>Design a One-shot prompt with one labeled example.</li> <li>Design a Few-shot prompt with 3–5 labeled examples.</li> <li>Compare the outputs and discuss accuracy differences.</li> </ol> <p><b>Zero-Shot PROMPT:</b> Classify this customer review as Positive, Negative, or Neutral: {review}</p>  <p><b>One-shot prompt:</b> Great product!" → Positive. "{review}</p> | Week2                     |

```
23_AI_162PY X
def classify_review(review):
    if "great" in review.lower():
        return "Positive"
    else:
        return "Neutral"

# Example usage
review = "Great product!"
classification = classify_review(review)
print(f"The review is categorized as: {classification}")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
"Great product!" -> Positive.
PS C:\Users\karab\OneDrive\Documents\AI Assistant Coding> C:\Users\karab\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/karab/OneDrive/Documents/AI Assistant Coding/2.3_AI_162_0.py"
"Great product!" -> Positive.
PS C:\Users\karab\OneDrive\Documents\AI Assistant Coding>
```

**Few-shot prompt:** Positive: "Love it!"; Negative: "Terrible"; Neutral: "OK".

```
23_AI_162PY X
def categorize_review(review):
    positive_keywords = ["love", "great", "excellent", "amazing", "fantastic"]
    negative_keywords = ["terrible", "bad", "awful", "horrible", "poor"]

    review_lower = review.lower()

    if any(keyword in review_lower for keyword in positive_keywords):
        return "Positive"
    elif any(keyword in review_lower for keyword in negative_keywords):
        return "Negative"
    else:
        return "Neutral"

# Example usage
review = "Love it!"
category = categorize_review(review)
print(f"The review is categorized as: {category}")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
"Great product!" -> Positive.
PS C:\Users\karab\OneDrive\Documents\AI Assistant Coding> C:\Users\karab\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/karab/OneDrive/Documents/AI Assistant Coding/2.3_AI_162_0.py"
"Great product!" -> Positive.
PS C:\Users\karab\OneDrive\Documents\AI Assistant Coding> C:\Users\karab\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/karab/OneDrive/Documents/AI Assistant Coding/2.3_AI_162_0.py"
The review is categorized as: Positive
PS C:\Users\karab\OneDrive\Documents\AI Assistant Coding>
```

**Explanation: Zero-shot:** Works without examples but has lower accuracy for unclear reviews.

**One-shot:** One example improves understanding and gives better accuracy than zero-shot.

**Few-shot:** Multiple examples give the highest accuracy and most consistent results.

## 2. Email Priority Classification

### Scenario:

A company wants to automatically prioritize incoming emails into **High Priority, Medium Priority, or Low Priority**.

### Tasks:

1. Create 6 sample email messages with priority labels.
2. Perform intent classification using **Zero-shot prompting**.

3. Perform classification using **One-shot prompting**.
4. Perform classification using **Few-shot prompting**.
5. Evaluate which technique produces the most reliable results and why.

**Zero-shot PROMPT:** Classify email priority as High, Medium, or Low

```

1 def classify_email_priority(email):
2     # Simple keyword-based classification
3     high_priority_keywords = ['urgent', 'important', 'asap', 'immediate']
4     medium_priority_keywords = ['follow up', 'reminder', 'please']
5
6     email_lower = email.lower()
7
8     if any(keyword in email_lower for keyword in high_priority_keywords):
9         return 'High'
10    elif any(keyword in email_lower for keyword in medium_priority_keywords):
11        return 'Medium'
12    else:
13        return 'Low'
14
15    # Example usage
16    email = "Please respond to this urgent request ASAP."
17    priority = classify_email_priority(email)
18    print(f"The email priority is: {priority}")

```

NOBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

he email priority is: High  
 S:\Users\karab\OneDrive\Documents\AI Assistant Codings []

**One-shot prompt:** "URGENT server down" → High.

```

1 import re
2
3 def parse_and_format_string(input_string: str, email_to_use: str = None) -> dict:
4     message_priority_pattern = r"^(.*?)\s+([^\s]+)$"
5     email_placeholder_pattern = r"^\{email\}$"
6
7     # Extract message and priority
8     msg_prio_match = re.match(message_priority_pattern, input_string)
9     message = None
10    priority = None
11    if msg_prio_match:
12        message = msg_prio_match.group(1)
13        priority = msg_prio_match.group(2)
14
15    # Extract and format email part
16    formatted_email_segment = None
17    # Find the part containing the email pattern
18    email_part_start_index = input_string.find("{email}")
19    if email_part_start_index != -1:
20        if email_to_use:
21            formatted_email_segment = f"{email_to_use}"
22        else:
23            # If no email is provided for substitution, keep the original placeholder
24            formatted_email_segment = "{email}"
25
26    return {
27        "message": message,
28        "priority": priority,
29        "formatted_email_segment": formatted_email_segment
30    }
31
32    # Example Usage:
33    input_str_example = "URGENT server down" + High. "{email}" +
34    print(f"Example 1: No email substitution")
35    result_no_sub = parse_and_format_string(input_str_example)
36    print(f"Input: {input_str_example}")
37    print(f"Parsed Output: {result_no_sub}")

```

----- Example 1: No email substitution -----  
 Input: "URGENT server down" + High. "{email}" +  
 Parsed Output: {'message': 'URGENT server down', 'priority': 'High', 'formatted\_email\_segment': None}  
 <19: SyntaxWarning: invalid escape sequence '\{'  
 <19: SyntaxWarning: invalid escape sequence '\{'  
 /tmp/python-input-736938104.py:19: SyntaxWarning: invalid escape sequence '\{'  
 email\_part\_start\_index = input\_string.find("{email}")

**Few-shot Prompt:** CRITICAL outage"; Medium: "Meeting tomorrow"; Low: "Newsletter". "{email}"

```

import re

def parse_and_forward_string(input_string:str, email_to:str = None) -> dict:
    parsed_items = {}

    formatted_email_segment = None
    email_placeholder_pattern = r'(?P=email1)%(?P=)'
    email_match = re.search(email_placeholder_pattern, input_string)
    email_segment = input_string[:email_match.start()] if email_match else None
    if email_match:
        original_email_placeholder_text = email_match.group(1)
        email_segment = input_string[:email_match.start()] + str(email_match.group(1))

    # Prepare the formatted email segment
    if email_to:
        # Reconstruct the email segment with the provided email
        if '% ' in original_email_placeholder_text:
            formatted_email_segment = f'({email_to}) % '
        else:
            formatted_email_segment = f'({email_to}) %'

    else:
        formatted_email_segment = original_email_placeholder_text

    # 2. Parse the email segment for messages and priorities
    if email_segment.endswith(')') or email_segment.endswith(') '):
        email_segment = email_segment[:-1].strip()

    # Parse email segment
    # 3. Parse patterns for different line formats
    pattern_group = re.compile('^(?P=)(?P=)') # e.g., Pattern: "Meeting tomorrow"
    pattern_group = re.compile('^(?P=)') # e.g., Pattern: "Meeting tomorrow"
    for part in parts:
        part = part.strip()
        if not part:
            continue
        match_group = pattern_group.match(part)
        if match_group:
            priority = match_group.group(1).strip()
            message = match_group.group(2).strip()
            parsed_items.append((message, priority, priority))
        else:
            match_group = pattern_group.match(part)
            if match_group:
                message = match_group.group(1).strip()
                parsed_items.append((message, message, priority)) # No explicit priority
            else:
                # 4. fallback for unrecognized parts, though examples should fit patterns
                parsed_items.append((f'raw_unparsed_part: {part}, priority: None'))

    return {
        "messages_with_priorities": parsed_items,
        "formatted_email_segment": formatted_email_segment
    }

# Example usage with one input string
input_str_example = "Meeting tomorrow"
parsed_items_example = parse_and_forward_string(input_str_example)
print(f"Example 1: No email substitution -----")
print(f"Result: {parsed_items_example}")
print(f"Forward string: {input_str_example}")
print(f"Forward output: {input_str_example}")

# Example 2: With email substitution
input_str_example = "Meeting tomorrow"
parsed_items_example = parse_and_forward_string(input_str_example, "user@example.com")
print(f"Example 2: With email substitution -----")
print(f"Result: {parsed_items_example}")
print(f"Forward string: {input_str_example}")
print(f"Forward output: {input_str_example}")

```

**EXPLANATION:**

**Zero-shot:** Classifies priority without examples, accuracy may drop for unclear emails.

**One-shot:** One example helps the model identify urgency better than zero-shot.

**Few-shot:** Multiple examples give the most accurate and consistent priority classification.

### 3. Student Query Routing System

**Scenario:**

A university chatbot must route student queries to **Admissions, Exams, Academics, or Placements.**

## Tasks:

1. Create 6 sample student queries mapped to departments.
2. Implement **Zero-shot intent classification** using an LLM.
3. Improve results using **One-shot prompting**.
4. Further refine results using **Few-shot prompting**.
5. Analyze how contextual examples affect classification accuracy.

**Zero-shot prompt:** Route to Admissions, Exams, Academics, or Placements

```

Untitled44.ipynb
File Edit View Insert Runtime Tools Help
Commands Code Text Run all

Start coding or @Generate with AI.

import google.generativeai as genai
from google.colab import userdata

# Assuming genai_model is already configured and CATEGORIES and user_query are defined

# Initialize the Generative Model if not already done
# Fetch the API key from Colab's secrets manager
# Ensure MODEL_API_KEY is set in Colab secrets
if 'genai_model' not in locals() and 'genai_model' not in globals():
    try:
        MODEL_API_KEY=userdata.get('GOOGLE_API_KEY')
        genai.configure(api_key=MODEL_API_KEY)
        genai_model = genai.GenerativeModel('gemini-pro')
        print('Genai model configured successfully.')
    except Exception as e:
        print(f"Error configuring Genai API: {e}. Please ensure GOOGLE_API_KEY is set in Colab secrets.")
        # Exit or handle the error appropriately if model cannot be initialized
        # For now, we'll let the NameError propagate if initialization fails

def route_query_with_genai(query: str, categories: list) -> str:
    """Routes a user query to one of the given categories using the Genai model.
    Expects genai_model to be initialized globally.

    Args:
        query: The user's query string.
        categories: A list of possible categories.

    Returns:
        The predicted category as a string.

    """
    category_list_str = ', '.join(categories)
    prompt = f"""Classify the following user query into one of these categories: {category_list_str}.
    Return only the category name, without any additional text.

    Query: '{query}'
    Category: """
    response = genai_model.generate_content(prompt)
    return response.text.strip()

# Define CATEGORIES and user_query if they are not globally defined yet
# For demonstration purposes I'll use some hard-coded values, but in a real scenario, they might come from previous cells.
if 'CATEGORIES' not in locals() and 'CATEGORIES' not in globals():
    CATEGORIES = ['Admissions', 'Exam Dates', 'Placements']
    print('Initialized CATEGORIES: ', CATEGORIES)

if 'user_query' not in locals() and 'user_query' not in globals():
    user_query = "What is the procedure for applying to master's programs?" # Example query
    print('Initialized user_query: ', user_query)

# Route the query using the existing variables
if 'genai_model' in locals() or 'genai_model' in globals():
    routed_category = route_query_with_genai(user_query, CATEGORIES)
    print(f"Original Query: '{user_query}'")
    print(f"Routed to Category: '{routed_category}'")
else:
    print("Genai model not initialized, cannot route query.")

# Error configuring Genai API: Secret GOOGLE_API_KEY does not exist.. Please ensure GOOGLE_API_KEY is set in Colab secrets.
Genai model not initialized, cannot route query.

```

One-shot prompt: MBA application" → Admissions. "{query}

```

Untitled44.ipynb
File Edit View Insert Runtime Tools Help
Commands Code Text Run all

import re

def parse_and_route_string(input_string: str, user_query_text: str = None) -> dict:
    """Parses a string in the format "Message" -> Category, "{query}"
    and optionally substitutes the "{query}" placeholder.

    Args:
        input_string: The string to parse.
        user_query_text: An optional string to substitute for "{query}".

    Returns:
        A dictionary containing the parsed message, category, and
        formatted query string.

    """
    # Define patterns for "Message" -> Category
    message_category_patterns = r'^(.*?) -> (.*?)$'
    query_placeholder_patterns = r'"{query}"$'

    # Extract message and category
    msg_cat_match = re.match(message_category_patterns, input_string)
    message = None
    category = None
    if msg_cat_match:
        message = msg_cat_match.group(1)
        category = msg_cat_match.group(2)

    # Extract and format the query part
    formatted_query_segment = None
    query_match = re.search(query_placeholder_patterns, input_string)

    if query_match:
        if user_query_text:
            formatted_query_segment = f'"{user_query_text}"'
        else:
            formatted_query_segment = query_match.group(1) # Keep original if no substitution

    return {
        "message": message,
        "category": category,
        "formatted_query_segment": formatted_query_segment
    }

# Example usage
input_str_example = "MBA application" -> Admissions, "{query}"

print("----- Example 1: No query substitution -----")
result_no_sub = parse_and_route_string(input_str_example)
print(f"Input: {input_str_example}")
print(f"Parsed Output: {result_no_sub}")

print("----- Example 2: With query substitution -----")
result_with_sub = parse_and_route_string(input_str_example, user_query_text="What are the exam dates?")
print(f"Input: {input_str_example}")
print(f"Parsed Output: {result_with_sub}")

# ----- Example 1: No query substitution -----
# Input: "MBA application" -> Admissions, "{query}"
# Parsed Output: {'message': 'MBA application', 'category': 'Admissions', 'formatted_query_segment': '"{query}"'}

# ----- Example 2: With query substitution -----
# Input: "MBA application" -> Admissions, "{query}"
# Parsed Output: {'message': 'MBA application', 'category': 'Admissions', 'formatted_query_segment': '"What are the exam dates?"'}

```

Few-shot prompt: Exam dates" → Exams; "Job fair" → Placements;  
 "Course list" → Academics.

```
import re

def parse_and_route_string(input_string: str, user_query_text: str = None) -> dict:
    """
    Parse and route a string based on the user query text.
    """
    # Extract the query segment from the rest of the string
    query_placeholder_pattern = r'{{query}}' # Placeholder for the query at the end of the string
    query_match = re.search(query_placeholder_pattern, input_string)

    main_segment = input_string.strip()
    if query_match:
        # Extract the part of the string before the query placeholder
        main_segment = input_string[:query_match.start()].strip()

    # Prepare the formatted query segment
    if user_query_text:
        formatted_query_segment = f'{{user_query_text}}'
    else:
        formatted_query_segment = query_match.group(1) # Keep original if no substitution
        # Remove any trailing ' or ' from the main segment if present
        if main_segment.endswith(' ') or main_segment.endswith(' '):
            main_segment = main_segment[:-1].strip()

    # Format the main segment for message and category
    # Split by semicolon to get individual message/category pairs
    parts = main_segment.split(';')

    message_category_pair_pattern = re.compile(r'({})'.format('{}'))

    for part in parts:
        part = part.strip()
        if not part:
            continue
        msg_cat_match = message_category_pair_pattern.match(part)
        if msg_cat_match:
            message = msg_cat_match.group(1).strip()
            category = msg_cat_match.group(2).strip()
            parsed_items.append({'message': message, 'category': category})
        else:
            # Fallback for unrecognized parts
            parsed_items.append({'raw_unparsed_part': part, 'message': None, 'category': None})

    return {
        "message_category_pairs": parsed_items,
        "formatted_query_segment": formatted_query_segment
    }

# Example usage with one input string
input_str_example = "Exam dates" + Exam; "Job fair" + Placement; "Course list" + Academic; "Query"

print("----- Example 1: No query substitution -----")
result_no_sub = parse_and_route_string(input_str_example)
print("Input: input_str_example")
print("Parsed Output: (result_no_sub)")

print("----- Example 2: With query substitution -----")
result_with_sub = parse_and_route_string(input_str_example, user_query_text="What are the exam dates?")
print("Input: input_str_example")
print("Parsed Output: (result_with_sub)")

# ----- Example 1: No query substitution -----
# Input: "Exam dates" + Exam; "Job fair" + Placement; "Course list" + Academic; "Query"
# Parsed Output: [{"message": "Exam dates", "category": "Exam"}, {"message": "Job fair", "category": "Placement"}, {"message": "Course list", "category": "Academic"}, {"formatted_query_segment": "Exam"}]

# ----- Example 2: With query substitution -----
# Input: "Exam dates" + Exam; "Job fair" + Placement; "Course list" + Academic; "Query"
# Parsed Output: [{"message": "Exam dates", "category": "Exam"}, {"message": "Job fair", "category": "Placement"}, {"message": "Course list", "category": "Academic"}, {"formatted_query_segment": "What"}]
```

## EXPLANATION:

**Zero-shot:** Routes the query using instructions only, accuracy may be lower for vague queries.

**One-shot:** One example improves routing accuracy compared to zero-shot.

**Few-shot:** Multiple examples give the highest accuracy and most reliable routing.

## 4. Chatbot Question Type Detection

### Scenario:

A chatbot must identify whether a user query is **Informational**, **Transactional**, **Complaint**, or **Feedback**.

### Tasks:

1. Prepare 6 chatbot queries mapped to question types.
2. Design prompts for Zero-shot, One-shot, and Few-shot learning.
3. Test all prompts on the same unseen queries.
4. Compare response correctness and ambiguity handling.
5. Document observations.

**Zero-shot prompt:** "Classify as Informational, Transactional, Complaint, or Feedback

```
def zero_shot_prompt(query):  
    prompt = f"""  
    Classify the user query into one category:  
    Informational, Transactional, Complaint, or Feedback.  
    Query: "{query}"  
    Category:  
    """  
    return prompt  
  
# Test  
print(zero_shot_prompt("Where is my order?"))  
  
"""  
Classify the user query into one category:  
Informational, Transactional, Complaint, or Feedback.  
Query: "Where is my order?"  
Category:  
"""
```

**One-shot prompt:** "'Store hours?' → Informational.

```
def one_shot_prompt(query):  
    prompt = f"""  
    "Store hours?" = Informational  
    "{query}" =  
    """  
    return prompt  
  
# Test  
print(one_shot_prompt("Cancel my order"))  
  
"""  
"Store hours?" = Informational  
"Cancel my order" =  
"""
```

**Few-shot prompt:** "'Cancel order' → Transactional; 'Bad service' → Complaint; 'Love update' → Feedback.

```
def few_shot_prompt(query):  
    prompt = f"""  
    "Cancel order" = Transactional  
    "Bad service" = Complaint  
    "Love the update" = Feedback  
    "{query}" =  
    """  
    return prompt  
  
# Test  
print(few_shot_prompt("The app is very slow"))  
  
"""  
"Cancel order" = Transactional  
"Bad service" = Complaint  
"Love the update" = Feedback  
"The app is very slow" =  
"""
```

## EXPLANATION:

**Zero-shot:** Uses only instructions, so accuracy can be lower for ambiguous queries.

**One-shot:** One example improves understanding and gives better results than zero-shot.

**Few-shot:** Multiple examples provide the highest accuracy and most consistent classification

## 5. Emotion Detection in Text

### Scenario:

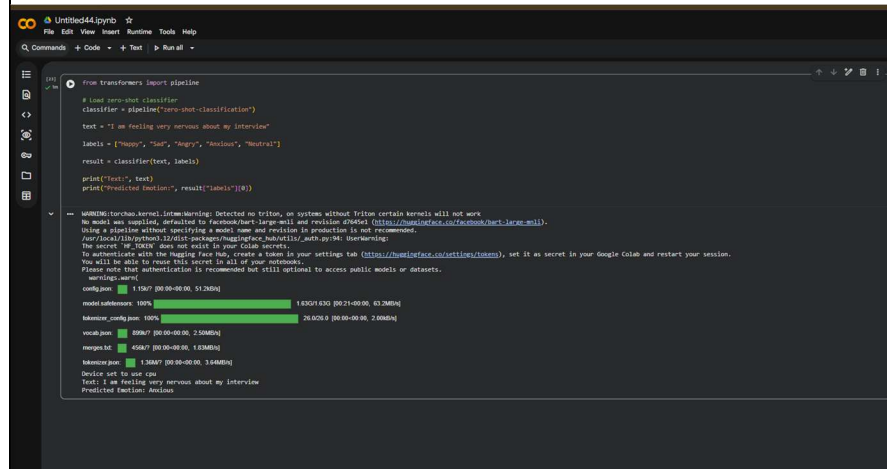
A mental-health chatbot needs to detect emotions: **Happy, Sad, Angry, Anxious, Neutral.**

### Tasks:

1. Create labeled emotion samples.
2. Use Zero-shot prompting to identify emotions.

3. Use One-shot prompting with an example.
4. Use Few-shot prompting with multiple emotions.
5. Discuss ambiguity handling across techniques.

**Zero-shot prompt:** Emotion from Happy, Sad, Angry, Anxious, Neutral



```

from transformers import pipeline

# Load zero-shot classifier
classifier = pipeline("zero-shot-classification")

text = "I am feeling very nervous about my interview"

labels = ["Happy", "Sad", "Angry", "Anxious", "Neutral"]

result = classifier(text, labels)

print("text:", text)
print("Predicted emotion:", result["labels"][0])

```

WARNING:torchvision.transforms.functional: Detected no triton, on systems without triton certain kernels will not work. We would like to continue to recommend to torchvision: large-eccv and revision 202404 (https://github.com/pytorch/vision/pull/5444). Using a pipeline without specifying a model name and revision in production is not recommended. /usr/local/lib/python3.10/site-packages/torchvision/models/\_utils.py:94: UserWarning: The secret key 'SECRET\_KEY' does not exist in your Cloud secrets. In authentication with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Cloud and restart your session. You will be able to reuse this secret in all of your notebooks. Please note that authentication is recommended but still optional to access public models or datasets. warnings.warn()

config.json 1.15K [00:00-00:00, 51.2KB/s]

model.safetensors 1.62G [00:21-00:00, 61.2MB/s]

tokenizer\_config.json 10K [00:00-00:00, 2.05MB/s]

vocab.json 459K [00:00-00:00, 2.05MB/s]

merges.txt 459K [00:00-00:00, 1.63MB/s]

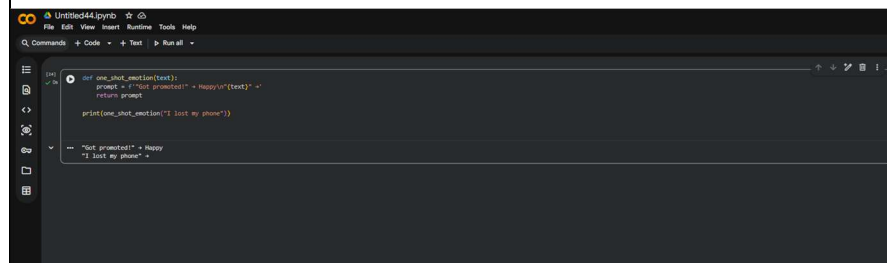
tokenizer.json 1.58K [00:00-00:00, 3.68MB/s]

Device set to use cpu

text: I am feeling very nervous about my interview

Predicted emotion: Anxious

**One-shot prompt:** Got promoted!" → Happy.



```

def one_shot_emotion(text):
    prompt = f"Got promoted!" + text
    return prompt

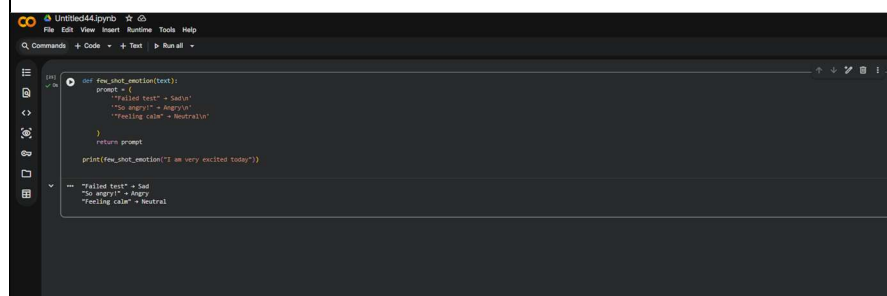
print(one_shot_emotion("I lost my phone"))

```

text: Got promoted! + Happy

text: I lost my phone +

**Few-shot prompt:** Failed test" → Sad; "So angry!" → Angry; "Feeling calm" → Neutral.



```

def few_shot_emotion(text):
    prompt = [
        "Failed test" + "Sad",
        "So angry!" + "Angry",
        "Feeling calm" + "Neutral"
    ]
    return prompt

print(few_shot_emotion("I am very excited today"))

```

text: Failed test + Sad

text: So angry! + Angry

text: Feeling calm + Neutral

**EXPLANATION:**

- **Zero-shot:** Detects emotion using only instructions, accuracy may drop for mixed emotions.
- **One-shot:** One example helps the model detect emotion more accurately than zero-shot.

|  |   |  |
|--|---|--|
|  | <ul style="list-style-type: none"><li>• <b>Few-shot:</b> Multiple examples give the highest accuracy and most reliable emotion detection.</li></ul> |  |
|--|---|--|