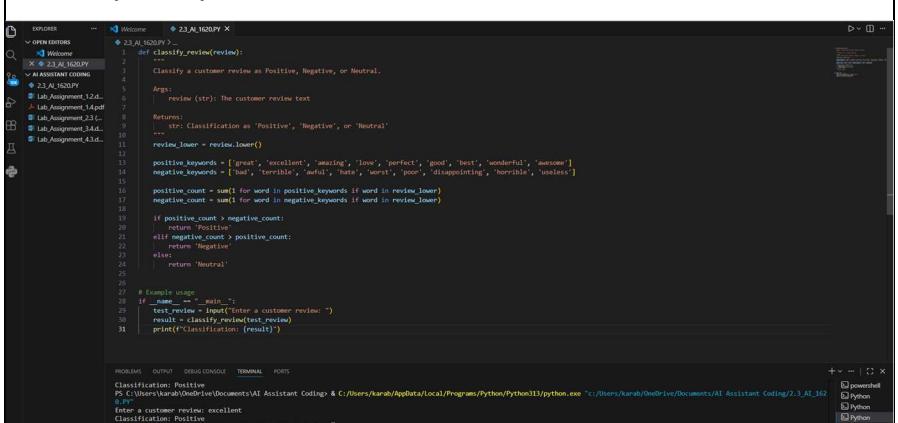


K.Akshay
2303A51817
B-26
Assignment- 4.4

Q.No.	Question	Expected Time to complete
1	<p>1. Sentiment Classification for Customer Reviews</p> <p>Scenario:</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>Tasks:</p> <ol style="list-style-type: none"> Prepare 6 short customer reviews mapped to sentiment labels. Design a Zero-shot prompt to classify sentiment. Design a One-shot prompt with one labeled example. Design a Few-shot prompt with 3–5 labeled examples. Compare the outputs and discuss accuracy differences. <p>Zero-Shot PROMPT: Classify this customer review as Positive, Negative, or Neutral: {review}</p>  <pre> 1 #!/usr/bin/env python 2 # 23_AL_1620PY 3 4 def classify_review(review): 5 """ 6 Classify a customer review as Positive, Negative, or Neutral. 7 """ 8 Args: 9 review (str): The customer review text 10 Returns: 11 str: Classification as "Positive", "Negative", or "Neutral" 12 13 review_lower = review.lower() 14 15 positive_keywords = ["great", "excellent", "amazing", "love", "perfect", "good", "best", "wonderful", "awesome"] 16 negative_keywords = ["bad", "terrible", "awful", "hate", "worst", "poor", "disappointing", "horrible", "useless"] 17 18 positive_count = sum(1 for word in positive_keywords if word in review_lower) 19 negative_count = sum(1 for word in negative_keywords if word in review_lower) 20 21 if positive_count > negative_count: 22 return "Positive" 23 elif negative_count > positive_count: 24 return "Negative" 25 else: 26 return "Neutral" 27 28 # Example usage 29 if __name__ == "__main__": 30 result = classify_review("Great product!") 31 print(f"Classification: {result}") </pre> <p>One-shot prompt: Great product!" → Positive. "{review}"</p>	Week2

```

1 Welcome  23_AI_1620PY X
2 # 23_AI_1620PY >
3     def classify_review(review):
4         if 'great' in review.lower():
5             return "Positive"
6         else:
7             return "Neutral"
8
9 # Example usage
10 review = "Great product!"
11 classification = classify_review(review)
12 print(f'{review} - {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_1620PY.py"
Great product! - Positive.
PS C:\Users\karab\OneDrive\Documents\AI Assistant Coding> 

```

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

```

1 Welcome  23_AI_1620PY X
2 # 23_AI_1620PY >
3     def categorize_review(review):
4         positive_keywords = ["love", "great", "excellent", "amazing", "fantastic"]
5         negative_keywords = ["terrible", "bad", "awful", "horrible", "poor"]
6
7         review_lower = review.lower()
8
9         if any(keyword in review_lower for keyword in positive_keywords):
10             return "Positive"
11         elif any(keyword in review_lower for keyword in negative_keywords):
12             return "Negative"
13         else:
14             return "Neutral"
15
16 # Example usage
17 review = "love it."
18 category = categorize_review(review)
19 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_1620PY.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_1620PY.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}")

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
The email priority is: High
\$ C:\Users\karab\OneDrive\Documents\AI Assistant Coding> []

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

```

import re

def parse_and_format_string(input_string: str, email_to_use: str = None) -> dict:
    message_priority_pattern = r"^(+*)> (+*)\.\s*"
    email_placeholder_pattern = r"\{(\email)\}\s*"

    # Extract message and priority
    msg_prio_match = re.match(message_priority_pattern, input_string)
    message = None
    priority = None
    if msg_prio_match:
        message = msg_prio_match.group(1)
        priority = msg_prio_match.group(2)

    # Extract and format email part
    formatted_email_segment = None
    # If no email part is provided, keep the original placeholder
    if email_to_use:
        formatted_email_segment = f"({email_to_use})>"
    else:
        formatted_email_segment = f"\{email\}\s*"

    return {
        "message": message,
        "priority": priority,
        "formatted_email_segment": formatted_email_segment
    }

# Example Usage:
input_str_example = "URGENT server down" + High, "\{email\}" + " "
print("----- Example 1: No email substitution -----")
print(f"Input: {input_str_example}")
result_no_sub = parse_and_format_string(input_str_example)
print(f"Input: {input_str_example}")
print(f"Output: {result_no_sub}")
print(f"----- Example 1: No email substitution -----")

```

... ---- Example 1: No email substitution -----
Input: "URGENT server down" + High, "\{email\}" + " "
Parsed Output: {"message": "URGENT server down", "priority": "High", "formatted_email_segment": None}
----- Example 2: Email substitution -----
>>> result_with_sub = parse_and_format_string(input_str_example)
>>> print(f"Input: {input_str_example}")
>>> print(f"Output: {result_with_sub}")
/tmp/ipython-input-736938104.py:19: SyntaxWarning: invalid escape sequence '\\'
email_part_start_index = input_string.find("\{email\}\s*")

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

```

● import re
def parse_email(input_string: str, email_to_use: str = None) -> dict:
    parsed_items = []
    formatted_email_segment = None
    original_email_placeholder_text = r'{{email}}'
    mail_match = re.search(original_email_placeholder_text, input_string)
    if mail_match:
        mail_segment = input_string[mail_match.start():mail_match.end()]
        if mail_to_use:
            original_email_placeholder_text = email_to_use
        else:
            original_email_placeholder_text = mail_match.group(1).strip()
        mail_segment = mail_segment.replace(original_email_placeholder_text, '')
        parsed_items.append({'segment': mail_segment, 'type': 'email'})
        formatted_email_segment = f'{{{email_to_use}}}'
    else:
        formatted_email_segment = f'{{{email_to_use}}}'  

    # Parse the main_segment for messages and priorities
    if '-' in original_email_placeholder_text:
        main_segment = original_email_placeholder_text.replace(original_email_placeholder_text, '-').strip()
        parts = main_segment.split('-')
        parts[-1] = parts[-1].strip()
        pattern_pr1_msg = re.compile(r'([^\(\)]*)\((\d+)\)([^\(\)]*)') # e.g., Medium: "Meeting tomorrow"
        pattern_pr2_msg = re.compile(r'([^\(\)]*)\((\d+)\)([^\(\)]*)') # e.g., "CRITICAL outage"
        part_in_parts = parts[-1]
        for part in part_in_parts:
            part = part.strip()
            continue
            match_pr1_msg = pattern_pr1_msg.match(part)
            if match_pr1_msg:
                priority = match_pr1_msg.group(2).strip()
                message = match_pr1_msg.group(1).strip()
                parsed_items.append({'message': message, 'priority': priority})
            else:
                match_pr2_msg = pattern_pr2_msg.match(part)
                if match_pr2_msg:
                    message = match_pr2_msg.group(1).strip()
                    parsed_items.append({'message': message, 'priority': None}) # No explicit priority
                else:
                    # Fall back for unenclosed parts, though examples should fit patterns
                    parsed_items.append({'raw_unparsed_part': part, 'priority': None})
    return {
        'segment': formatted_email_segment,
        'parsed_items': parsed_items
    }

# Example usage with new input string
input_str = "Meeting tomorrow, priority: Medium: {{email}} Low: {{email}} Newsletter, {{email}}"
print(f'... Example 1: No email substitution ...')
result1 = parse_email(input_str)
print(result1)
print(f'Input: {input_str}, Output: {result1["segment"]}')
print(f'Parsed output: {result1["parsed_items"]}')

--- Example 1: No email substitution ---
Input: 'Meeting tomorrow, priority: Medium: {{email}} Low: {{email}} Newsletter, {{email}}'
Parsed output: {'segment': 'Meeting tomorrow, priority: Medium: (Medium) (Meeting tomorrow)', 'parsed_items': []}
--- Example 2: With email substitution ---
Input: 'Meeting tomorrow, priority: Medium: {{email}} Low: {{email}} Newsletter, {{email}} support@cole.com'
Parsed output: {'segment': 'Meeting tomorrow, priority: Medium: (Medium) (Meeting tomorrow)', 'parsed_items': [{"message": "support@cole.com"}]}


```

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

```

  File Edit View Insert Runtime Tools Help
  Commands + Code + Text | Run all

  Start coding or generate with AI.

  [M] Untitled44.pylib
  import google.generativeai as gemai
  from google.colab import secrets
  # Assuming gemai_model is already configured and CATEGORIES and user_query are defined
  # Initialize the Generative Model if not already done
  # Ensure GOOGLE_API_KEY is set in Colab secrets
  if 'gemai_model' not in locals() and 'gemai_model' not in globals():
    try:
      GOOGLE_API_KEY=secrets.get('GOOGLE_API_KEY')
      gemai.configure(api_key=GOOGLE_API_KEY)
      print("Gemai API configured successfully!")
    except Exception as e:
      print(f"Error configuring Gemai 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 Notebook propagate if initialization fails

  def route_query_with_gemai(query: str, categories: list) -> str:
    """
    Routes a user query to one of the given categories using the Gemai model.
    Assumes gemai_model is 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: (query)
  response = gemai_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 add them here. In a real scenario, they might come from previous cells.
  if 'CATEGORIES' not in locals() and 'CATEGORIES' not in globals():
    CATEGORIES = ['Admissions', 'Academics', 'Placements', 'Exams', 'Job Fair']
    print(f"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(f"Initialized user_query: {user_query}")

  # Route the query using the existing variables
  if user_query in CATEGORIES:
    user_query = route_query_with_gemai(user_query, CATEGORIES)
    print(f"Original Query: '{user_query}'")
    print(f"Routed to Category: {route_category}")
  else:
    print("Gemai model not initialized, cannot route query.")

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

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

  File Edit View Insert Runtime Tools Help
  Commands + Code + Text | Run all

  [M] Untitled44.pylib
  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 the
      formatted query string.
    ...
    # Regex to capture "Message -> Category"
    message_category_pattern = r"^(.*?)->(.*)$"
    query_placeholder_pattern = r"\{(.*)\}\n"
    query_placeholder_replacement = r"\1"

    # Extract message and category
    msg_cat_match = re.search(message_category_pattern, 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_pattern, input_string)

    if query_match:
      if user_query_text:
        formatted_query_segment = f"{query_placeholder_replacement}{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(result_no_sub['message'])
  print(result_no_sub['category'])
  print(result_no_sub['formatted_query_segment'])
  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(result_with_sub['message'])
  print(result_with_sub['category'])
  print(result_with_sub['formatted_query_segment'])
  print("Parsed Output: (result_with_sub)")

  ----- Example 1: No query substitution -----
  Input: "MBA application -> Admissions. '{query}'"
  Parsed Output: {'message': 'MBA application -> Admissions.', 'category': 'Admissions', 'formatted_query_segment': '{query}'}
  ----- Example 2: With query substitution -----
  Input: "MBA application -> Admissions. '{query}'"
  Parsed Output: {'message': 'MBA application -> Admissions.', 'category': 'Admissions', 'formatted_query_segment': 'What are the exam dates?'}

```

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

```

def parse_and_route_string(input_string: str, user_query_text: str = None) -> dict:
    parsed_items: [] = []
    query_plaintext: str = None
    # 1. Separate the query segment from the rest of the string
    query_placeholder: str = re.search(query_placeholder_pattern, input_string + " $match(query)" at the end of the string
    main_segment: str = input_string[query_match.start():].strip()
    # Preprocess 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
    # Process the main segment
    if main_segment.startswith('`') or main_segment.endswith('`'):
        main_segment = main_segment[1:-1].strip()
    # 2. Extract message-category pairs
    message_category_pairs: [] = []
    # Split by semicolon to get individual message-category pairs
    parts = main_segment.split(';')
    message_category_pair_pattern = re.compile(r"(.+?)" + ":" + "(.+)")

    for part in parts:
        part: str
        if part == '':
            continue
        msg_cat_match: re.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:
            # Failover 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 new input string:
input_str_example = "Course list" + Examples["Job fair"] + Placements["Course list"] + Academics["(query)"]
print("----- Example 1: No query substitution -----")
result_no_sub = parse_and_route_string(input=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=input_str_example, user_query_text="What are the exam dates?")
print("Input: ({input_str_example})")
print("Parsed Output: ({result_with_sub})")

```

EXPLAINATION:

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 = """"
    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 = """"
    "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 = """"
    "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(result["text"], result["label"])

```

-- WARMING: torchab_kernel.InternWarning: Detected no triton, on systems without Triton certain kernels will not work
No model was supplied, defaulted to Facebook/bart-large-mnli and revision d7d6e1 (https://huggingface.co/facebook/bart-large-mnli).
This warning is present because the user has not specified a model or revision.
UserWarning: /usr/local/lib/python3.12/dist-packages/huggingface_hub/v11/_auth.py:94: UserWarning:
To authenticate with the Hugging Face Hub, create a token in your settings tab (<https://huggingface.co/settings/token>). You will be able to reuse this secret in all of your notebooks.
From: /usr/local/lib/python3.12/dist-packages/huggingface_hub/v11/_auth.py:94
warnings.warn(config.json 11597 [00:00:00, 51.26MB]
model_sdaccel 100% 1430/1430 [00:21:00, 62.3MB/s]
tokenizer_config.json 100% 24.00/24.00 [00:00:00, 2.00MB/s]
vocab.json 8994/7 [00:00:00, 2.50MB/s]
merges.txt 456/47 [00:00:00, 0.83MB/s]
tokenizer.json 1364/7 [00:00:00, 3.64MB/s]
deepspeed config
Text: I am feeling very nervous about my interview
Predicted Emotion: Anxious

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

```

def one_shot_emotion(text):
    prompt = "got promoted" + Happy_in["text"] + "\n"
    return prompt
print(one_shot_emotion("I lost my phone"))

```

-- "Got promoted!" > Happy
"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_in["text"] + "\n"
    "So angry" + Angry_in["text"] + "\n"
    "Feeling calm" + Neutral_in["text"]
    )
    return prompt
print(few_shot_emotion("I am very excited today"))

```

-- "Failed test" > Sad
"So angry" > Angry
"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">• Few-shot: Multiple examples give the highest accuracy and most reliable emotion detection.	
--	---	--