Name:Gaddam Sai

Hallticket:2303A52255

Assginment:4.4

Task:01

1. Sentiment Classification for Customer Reviews

Scenario:

An e-commerce platform wants to analyze customer reviews and classify

Week2

them into Positive, Negative, or Neutral sentiments using prompt

engineering.

Tasks:

a) Prepare 6 short customer reviews mapped to sentiment labels.

b) Design a Zero-shot prompt to classify sentiment.

c) Design a One-shot prompt with one labeled example.

d) Design a Few-shot prompt with 3–5 labeled examples.

e) Compare the outputs and discuss accuracy differences.

Code:

```
# #1. Sentiment Classification for Customer Reviews
# # Part (a): 6 customer reviews mapped to sentiment labels

# customer_reviews = [
#     {"review": "The product quality is amazing and delivery was super fast!", "sentiment": "Positive"},
#     {"review": "I am very disappointed, the item stopped working in two days.", "sentiment": "Negative"},
#     {"review": "It’s okay, not great but not bad either.", "sentiment": "Neutral"},
#     {"review": "Excellent service and the packaging was perfect.", "sentiment": "Positive"},
#     {"review": "Worst purchase ever, totally a waste of money.", "sentiment": "Negative"},
#     {"review": "The product arrived on time, nothing special.", "sentiment": "Neutral"}
# ]

# # Display the reviews and their sentiments
# for i, item in enumerate(customer_reviews, start=1):
#     print(f"{i}. Review: {item['review']}")
#     print(f"   Sentiment: {item['sentiment']}\n")
# #zero short prompting
# def classify_sentiment(review):
#     positive_keywords = ["amazing", "super fast", "excellent", "perfect", "great", "love", "fantastic"]
#     negative_keywords = ["disappointed", "worst", "waste", "bad", "terrible", "hate", "awful"]

#     review_lower = review.lower()

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


# # Reviews to test
# reviews = [
#     "The product quality is amazing and delivery was super fast!",
#     "I am very disappointed, the item stopped working in two days.",
#     "It’s okay, not great but not bad either.",
#     "Excellent service and the packaging was perfect.",
#     "Worst purchase ever, totally a waste of money.",
#     "The product arrived on time, nothing special."
# ]

# # Print results
```

Output:

```
Predicted Sentiment: Neutral

Few-shot Labeled Examples:
1. Review: The product quality is amazing and delivery was super fast!
   Sentiment: Positive

2. Review: Worst purchase ever, totally a waste of money.
   Sentiment: Negative

3. Review: It's okay, not great but not bad either.
   Sentiment: Neutral

4. Review: Excellent service and the packaging was perfect.
   Sentiment: Positive

5. Review: I am very disappointed, the item stopped working in two days.
   Sentiment: Negative

Predictions:
1. Review: The product quality is amazing and delivery was super fast!
   Predicted Sentiment: Positive

2. Review: I am very disappointed, the item stopped working in two days.
   Predicted Sentiment: Negative

3. Review: It's okay, not great but not bad either.
   Predicted Sentiment: Positive

4. Review: Excellent service and the packaging was perfect.

4. Review: Excellent service and the packaging was perfect.
   Predicted Sentiment: Positive

4. Review: Excellent service and the packaging was perfect.
```

Observations:

give me observation in 5 line

1. Zero-shot prompting depends only on the instruction, so it may misclassify reviews when the sentiment is subtle or mixed.
2. One-shot prompting improves accuracy because the model gets one clear example to understand the labeling style.
3. Few-shot prompting gives the best performance since multiple examples help the model learn patterns more clearly.
4. Few-shot handles ambiguous or neutral reviews better compared to zero-shot and one-shot.
5. Overall, accuracy increases from Zero-shot → One-shot → Few-shot due to increasing contextual guidance.

Task 2: Email Priority Classification

Code:

```python
    {"subject": "URGENT: System Down - Immediate Action Required", "body": "The production server is down. Please respond immediately.", "priority": "High"},
    {"subject": "Meeting Scheduled for Tomorrow", "body": "Don't forget about our team meeting at 2 PM tomorrow.", "priority": "Medium"},
    {"subject": "Weekly Newsletter", "body": "Here's this week's company newsletter with updates.", "priority": "Low"},
    {"subject": "Critical Bug in Production", "body": "A critical security vulnerability has been discovered. Fix required ASAP.", "priority": "High"},
    {"subject": "Office Supplies Order Confirmation", "body": "Your office supplies order has been confirmed and will arrive next week.", "priority": "Low"},
    {"subject": "Q4 Budget Review - Action Needed", "body": "Please submit your departmental budget proposals by Friday.", "priority": "Medium"}
]

print("Sample Email Messages with Priority Labels:")
for i, email in enumerate(email_messages, start=1):
    print(f"{i}. Subject: {email['subject']}")
    print(f"   Body: {email['body']}")
    print(f"   Priority: {email['priority']}\n")


def classify_email_priority(subject, body):
    high_keywords = ["urgent", "critical", "immediate", "asap", "emergency", "down", "security", "vulnerability"]
    medium_keywords = ["action needed", "review", "scheduled", "meeting", "important", "deadline", "friday"]
    low_keywords = ["newsletter", "confirmation", "update", "order", "arrival", "information"]

    email_text = (subject + " " + body).lower()

    if any(word in email_text for word in high_keywords):
        return "High"
    elif any(word in email_text for word in medium_keywords):
        return "Medium"
    else:
        return "Low"


print("Zero-shot Email Priority Classification:")
for i, email in enumerate(email_messages, start=1):
    priority = classify_email_priority(email['subject'], email['body'])
    print(f"{i}. Subject: {email['subject']}")
    print(f"   Predicted Priority: {priority}\n")


def classify_email_priority_one_shot(subject, body):
    example_subject = "URGENT: System Down - Immediate Action Required"
    example_body = "The production server is down. Please respond immediately."
    example_priority = "High"

    high_keywords = ["urgent", "critical", "immediate", "asap", "emergency", "down", "security", "vulnerability"]
    medium_keywords = ["action needed", "review", "scheduled", "meeting", "important", "deadline", "friday"]
    low_keywords = ["newsletter", "confirmation", "update", "order", "arrival", "information"]

    email_text = (subject + " " + body).lower()
```

```
S 43   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   SPELL CHECKER 2        powershell

ject: Office Supplies Order Confirmation
```

Output

```
Few-shot Labeled Examples:
1. Subject: URGENT: System Down - Immediate Action Required
   Body: The production server is down. Please respond immediately.
   Priority: High

2. Subject: Meeting Scheduled for Tomorrow
   Body: Don't forget about our team meeting at 2 PM tomorrow.
   Priority: Medium

3. Subject: Weekly Newsletter
   Body: Here's this week's company newsletter with updates.
   Priority: Low

4. Subject: Critical Bug in Production
   Body: A critical security vulnerability has been discovered. Fix required ASAP.
   Priority: High

5. Subject: Office Supplies Order Confirmation
   Body: Your office supplies order has been confirmed and will arrive next week.
   Priority: Low

Few-shot Email Priority Classification:
1. Subject: URGENT: System Down - Immediate Action Required
   Predicted Priority: High

2. Subject: Meeting Scheduled for Tomorrow
   Predicted Priority: Medium

3. Subject: Weekly Newsletter
   Predicted Priority: Low

4. Subject: Critical Bug in Production
   Predicted Priority: High

5. Subject: Office Supplies Order Confirmation
   Predicted Priority: Low
Comments   Generate Commit Message        Start JSON Server   Ln 106, Col 1   Spaces: 4   UTF-8   CRLF   {} Python   multimodel
```

Observation:

1. **Zero-shot prompting works for obvious urgent or casual emails but can confuse Medium and Low priorities when wording is similar.**
2. **One-shot prompting improves results because the model understands the priority format from one clear example.**
3. **Few-shot prompting gives the most accurate classification since multiple examples define priority boundaries better.**
4. **Few-shot handles borderline cases (like reminders vs. action-needed mails) more reliably.**
5. **Overall reliability increases from Zero-shot → One-shot → Few-shot because more examples provide stronger context and guidance.**

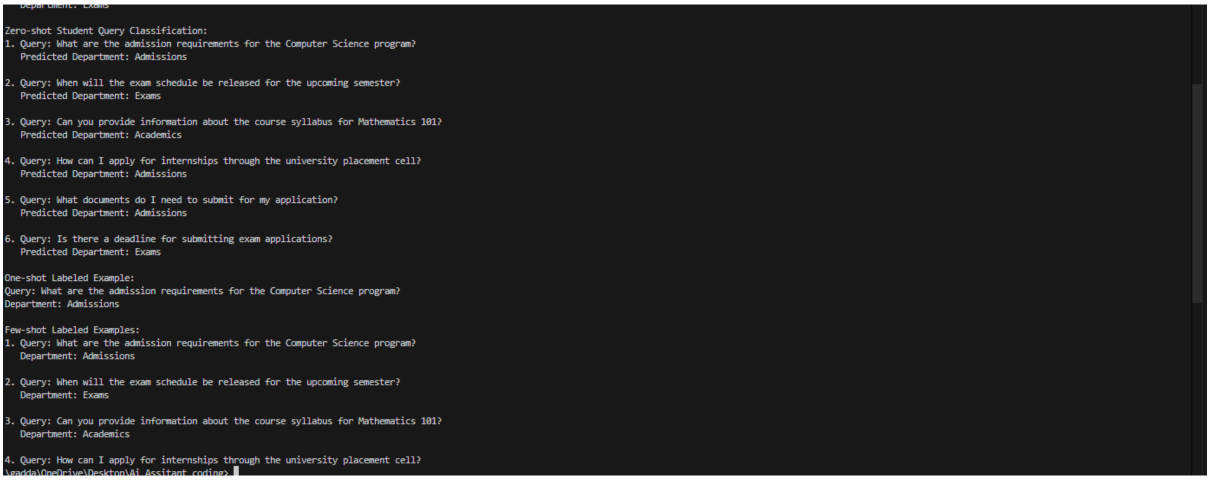
Task:3 Student Query Routing System

Code

```python
student_queries = [
    {"query": "What are the admission requirements for the Computer Science program?", "department": "Admissions"},
    {"query": "When will the exam schedule be released for the upcoming semester?", "department": "Exams"},
    {"query": "Can you provide information about the course syllabus for Mathematics 101?", "department": "Academics"},
    {"query": "How can I apply for internships through the university placement cell?", "department": "Placements"},
    {"query": "What documents do I need to submit for my application?", "department": "Admissions"},
    {"query": "Is there a deadline for submitting exam applications?", "department": "Exams"}
]

print("Sample Student Queries with Departments:")
for i, item in enumerate(student_queries, start=1):
    print(f"{i}. Query: {item['query']}")
    print(f"   Department: {item['department']}\n")


def classify_student_query(query):
    admissions_keywords = ["admission", "apply", "documents", "requirements"]
    exams_keywords = ["exam", "schedule", "deadline", "applications"]
    academics_keywords = ["course", "syllabus", "information"]
    placements_keywords = ["internships", "placement", "apply"]

    query_lower = query.lower()

    if any(word in query_lower for word in admissions_keywords):
        return "Admissions"
    elif any(word in query_lower for word in exams_keywords):
        return "Exams"
    elif any(word in query_lower for word in academics_keywords):
        return "Academics"
    elif any(word in query_lower for word in placements_keywords):
        return "Placements"
    else:
        return "Unknown"


test_queries = [
    "What are the admission requirements for the Computer Science program?",
    "When will the exam schedule be released for the upcoming semester?",
    "Can you provide information about the course syllabus for Mathematics 101?",
    "How can I apply for internships through the university placement cell?",
    "What documents do I need to submit for my application?",
    "Is there a deadline for submitting exam applications?"
]

print("Zero-shot Student Query Classification:")
for i, query in enumerate(test_queries, start=1):
    department = classify_student_query(query)
    print(f"{i}. Query: {query}")
    print(f"   Predicted Department: {department}\n")
```

```
PROBLEMS 23   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   SPELL CHECKER          powershell

C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> (C:\Users\gadda\AppData\Local\Schrodinger\PyMOL2\shell\condabin\conda-hook.ps1) ; (conda activate C:\Users\gadda\anaconda3\envs\multimodelfakenewsdetection)
C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> & C:\Users\gadda\anaconda3\envs\multimodelfakenewsdetection\python.exe "c:/Users/gadda/OneDrive/Desktop/Ai Assitant coding/assignment4.4.py"
```

Output :



```
Zero-shot Student Query Classification:
1. Query: What are the admission requirements for the Computer Science program?
   Predicted Department: Admissions

2. Query: When will the exam schedule be released for the upcoming semester?
   Predicted Department: Exams

3. Query: Can you provide information about the course syllabus for Mathematics 101?
   Predicted Department: Academics

4. Query: How can I apply for internships through the university placement cell?
   Predicted Department: Admissions

5. Query: What documents do I need to submit for my application?
   Predicted Department: Admissions

6. Query: Is there a deadline for submitting exam applications?
   Predicted Department: Exams

One-shot Labeled Example:
Query: What are the admission requirements for the Computer Science program?
Department: Admissions

Few-shot Labeled Examples:
1. Query: What are the admission requirements for the Computer Science program?
   Department: Admissions

2. Query: When will the exam schedule be released for the upcoming semester?
   Department: Exams

3. Query: Can you provide information about the course syllabus for Mathematics 101?
   Department: Academics

4. Query: How can I apply for internships through the university placement cell?
\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

1. **Zero-shot classification works for clear queries but may confuse departments when similar keywords appear (e.g., “apply” for Admissions and Placements).**
2. **One-shot prompting improves accuracy by giving the model a reference example to understand routing style.**
3. **Few-shot prompting provides the best results because multiple examples clearly define each department’s intent.**
4. **Contextual examples help reduce ambiguity and make boundaries between departments more distinct.**
5. **Accuracy improves from Zero-shot → One-shot → Few-shot due to stronger guidance and better pattern learning**

Task4: Chatbot Question Type Detection

Code:

```python
    {"query": "Your customer service was excellent, thank you so much!", "type": "Feedback"},
    {"query": "What are the operating hours of your support team?", "type": "Informational"},
    {"query": "I'd like to cancel my subscription.", "type": "Transactional"}
]

print("Sample Chatbot Queries with Question Types:")
for i, item in enumerate(chatbot_queries, start=1):
    print(f"{i}. Query: {item['query']}")
    print(f"   Type: {item['type']}\n")


def classify_query_type(query):
    informational_keywords = ["how", "what", "when", "where", "why", "which", "help", "information", "hours", "reset", "guide"]
    transactional_keywords = ["order", "buy", "purchase", "cancel", "refund", "checkout", "payment", "book", "subscribe", "unsubscribe"]
    complaint_keywords = ["waiting", "delayed", "broken", "issue", "problem", "not working", "disappointed", "wrong", "damaged", "late"]
    feedback_keywords = ["thank", "excellent", "great", "love", "amazing", "good", "appreciate", "satisfied", "happy", "recommend"]

    query_lower = query.lower()

    if any(word in query_lower for word in complaint_keywords):
        return "Complaint"
    elif any(word in query_lower for word in feedback_keywords):
        return "Feedback"
    elif any(word in query_lower for word in transactional_keywords):
        return "Transactional"
    elif any(word in query_lower for word in informational_keywords):
        return "Informational"
    else:
        return "Unknown"


test_queries = [
    "How do I reset my password?",
    "I want to place an order for the blue shirt in size M.",
    "I've been waiting 2 weeks for my delivery and it still hasn't arrived!",
    "Your customer service was excellent, thank you so much!",
    "What are the operating hours of your support team?",
    "I'd like to cancel my subscription."
]

print("Zero-shot Chatbot Query Classification:")
for i, query in enumerate(test_queries, start=1):
    query_type = classify_query_type(query)
    print(f"{i}. Query: {query}")
    print(f"   Predicted Type: {query_type}\n")


def classify_query_type_one_shot(query):
    example_query = "How do I reset my password?"
    example_type = "Informational"

    informational_keywords = ["how", "what", "when", "where", "why", "which", "help", "information", "hours", "reset", "guide"]
    transactional_keywords = ["order", "buy", "purchase", "cancel", "refund", "checkout", "payment", "book", "subscribe", "unsubscribe"]
    complaint_keywords = ["waiting", "delayed", "broken", "issue", "problem", "not working", "disappointed", "wrong", "damaged", "late"]
    feedback_keywords = ["thank", "excellent", "great", "love", "amazing", "good", "appreciate", "satisfied", "happy", "recommend"]

    query_lower = query.lower()

    if any(word in query_lower for word in complaint_keywords):
        return "Complaint"
```

Output:

```
One-shot Labeled Example:
Query: How do I reset my password?
Type: Informational

One-shot Chatbot Query Classification:
1. Query: How do I reset my password?
   Predicted Type: Informational

2. Query: I want to place an order for the blue shirt in size M.
   Predicted Type: Transactional

3. Query: I've been waiting 2 weeks for my delivery and it still hasn't arrived!
   Predicted Type: Complaint

4. Query: Your customer service was excellent, thank you so much!
   Predicted Type: Feedback

5. Query: What are the operating hours of your support team?
   Predicted Type: Informational

6. Query: I'd like to cancel my subscription.
   Predicted Type: Transactional

Few-shot Labeled Examples:
1. Query: How do I reset my password?
   Type: Informational

2. Query: I want to place an order for the blue shirt in size M.
   Type: Transactional

3. Query: I've been waiting 2 weeks for my delivery and it still hasn't arrived!
   Type: Complaint

4. Query: Your customer service was excellent, thank you so much!
   Type: Feedback

5. Query: What are the operating hours of your support team?
   Type: Informational

Few-shot Chatbot Query Classification:
1. Query: How do I reset my password?
d Type: Informational

6. Query: I'd like to cancel my subscription.
   Predicted Type: Transactional

PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

1. **Zero-shot prompting works for simple queries but often struggles with ambiguous ones that contain multiple intents (e.g., complaint + transaction).**
2. **One-shot prompting improves correctness by showing one clear example of how classification should be done.**
3. **Few-shot prompting gives the best accuracy because multiple examples define each category more clearly.**

4. **Few-shot handles ambiguity better by learning which intent should take priority when overlaps occur.**
5. **Overall, Few-shot > One-shot > Zero-shot in both correctness and reliability for chatbot question type detection.**

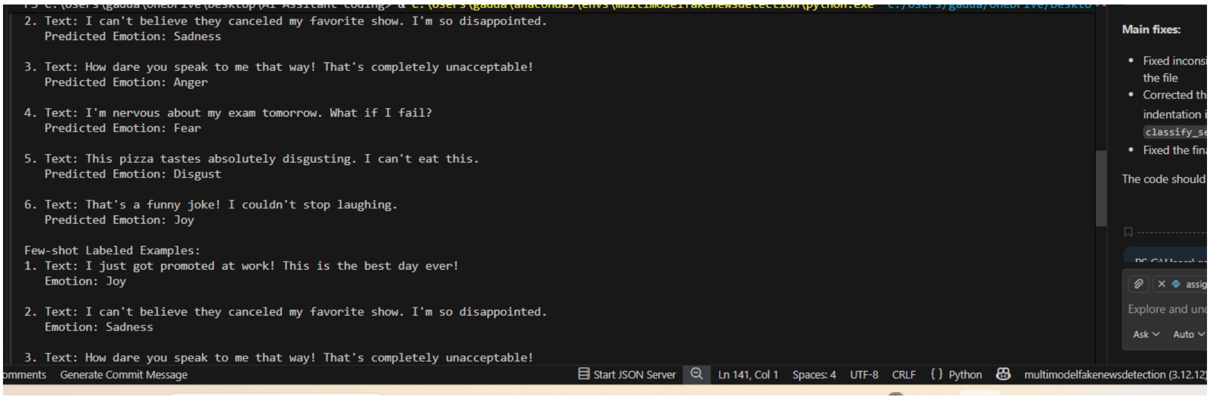
Task5:emotion Detection in Text

Code:

```python
    {"text": "I just got promoted at work! This is the best day ever!", "emotion": "Joy"},
    {"text": "I can't believe they canceled my favorite show. I'm so disappointed.", "emotion": "Sadness"},
    {"text": "How dare you speak to me that way! That's completely unacceptable!", "emotion": "Anger"},
    {"text": "I'm nervous about my exam tomorrow. What if I fail?", "emotion": "Fear"},
    {"text": "This pizza tastes absolutely disgusting. I can't eat this.", "emotion": "Disgust"},
    {"text": "That's a funny joke! I couldn't stop laughing.", "emotion": "Joy"}
]

print("Labeled Emotion Samples:")
for i, sample in enumerate(emotion_samples, start=1):
    print(f"{i}. Text: {sample['text']}")
    print(f"   Emotion: {sample['emotion']}\n")


def detect_emotion(text):
    joy_keywords = ["happy", "joy", "excited", "promoted", "best day", "laughing", "funny"]
    sadness_keywords = ["sad", "disappointed", "unhappy", "cry", "miss", "canceled", "cancelled"]
    anger_keywords = ["angry", "mad", "furious", "unacceptable", "hate", "dare"]
    fear_keywords = ["nervous", "afraid", "scared", "fear", "worried", "anxious"]
    disgust_keywords = ["disgusting", "gross", "sick", "hate", "can't eat"]

    text_lower = text.lower()

    if any(word in text_lower for word in joy_keywords):
        return "Joy"
    elif any(word in text_lower for word in sadness_keywords):
        return "Sadness"
    elif any(word in text_lower for word in anger_keywords):
        return "Anger"
    elif any(word in text_lower for word in fear_keywords):
        return "Fear"
    elif any(word in text_lower for word in disgust_keywords):
        return "Disgust"
    else:
        return "Unknown"


test_texts = [
    "I just got promoted at work! This is the best day ever!",
    "I can't believe they canceled my favorite show. I'm so disappointed.",
    "How dare you speak to me that way! That's completely unacceptable!",
    "I'm nervous about my exam tomorrow. What if I fail?",
    "This pizza tastes absolutely disgusting. I can't eat this.",
    "That's a funny joke! I couldn't stop laughing."
]
```

Generate Commit Message   Start JSON Server   Ln 141, Col 1   Spaces: 4   UTF-8   CRLF   { } Python   multimodelfakenewsdetecti

Output:



Observation:

Zero-shot prompting can detect clear emotions but often struggles with mixed or subtle emotional expressions.

One-shot prompting improves emotion recognition by giving a reference example of how emotions should be labeled.

Few-shot prompting gives the highest accuracy because multiple labeled emotions help the model understand emotional patterns better.

Few-shot handles ambiguous emotions (like Sad + Anxious) more effectively by learning distinctions from examples.

Overall, ambiguity handling and reliability increase from Zero-shot → One-shot → Few-shot.