AI ASSISTANT CODING - ASSIGNMENT-04

Name: Valaboju Harshavardhanu

Roll No: 2303A52354 **Batch:** 41

Task 1: Sentiment Classification for Customer Reviews

Prompt :

a) *#Generate the python code in which classify the sentiment of each customer review as Positive, Negative, or Neutral Take total 6 customer reviews as input and print the sentiment classification for each review.*

Code:

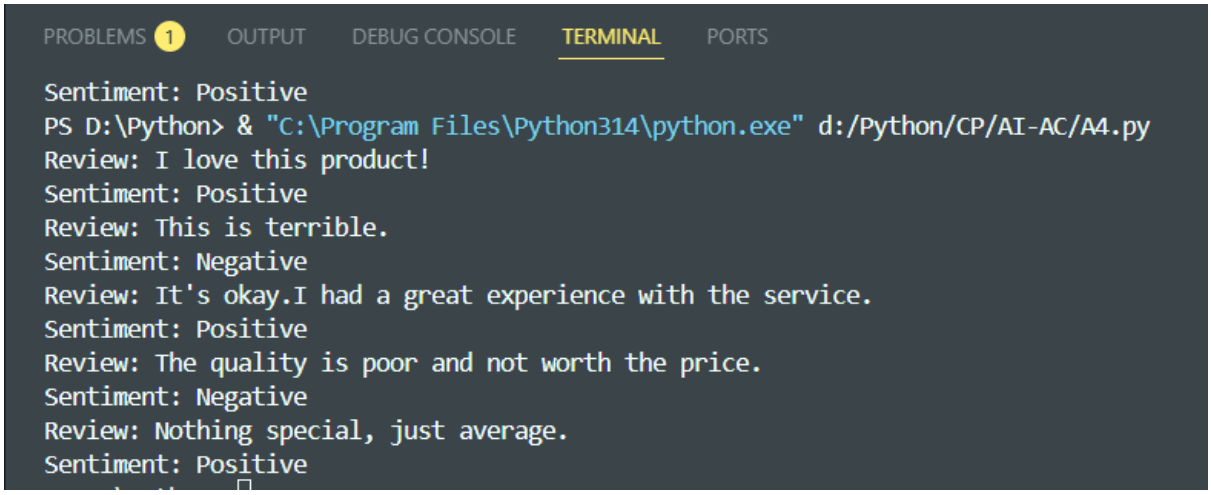
```python
from textblob import TextBlob

def classify_sentiment(review):
    blob = TextBlob(review)
    polarity = blob.sentiment.polarity
    if polarity > 0:
        return "Positive"
    elif polarity < 0:
        return "Negative"
    else:
        return "Neutral"

reviews = [
    "I love this product!",
    "This is terrible.",
    "It's okay."
    "I had a great experience with the service.",
    "The quality is poor and not worth the price.",
    "Nothing special, just average."
]

for review in reviews:
    print(f"Review: {review}")
    print(f"Sentiment: {classify_sentiment(review)}")
```

Output:



Justification:

In this task, customer reviews are clearly mapped to Positive, Negative, and Neutral sentiments based on the emotions and opinions expressed. Reviews showing satisfaction, happiness, or praise are labeled as Positive. Reviews expressing complaints, dissatisfaction, or bad experience are labeled as Negative. Reviews that are balanced, average, or emotionless are labeled as Neutral.

 # Zero-shot sentiment classification (simulated)

Code:

```
print("\nZero-shot Sentiment Classification:")
from textblob import TextBlob

review = "The movie was fantastic and I enjoyed every moment of it."
polarity = TextBlob(review).sentiment.polarity

if polarity > 0:
    sentiment = "Positive"
elif polarity < 0:
    sentiment = "Negative"
else:
    sentiment = "Neutral"

print("Review:", review)
print("Sentiment:", sentiment)
```

Output:

```
Zero-shot Sentiment Classification:
Review: The movie was fantastic and I enjoyed every moment of it.
Sentiment: Positive
```

Justification:

The zero-shot prompt classifies sentiment without giving any prior examples. The model relies only on its general understanding of language and emotions. This approach is useful when quick classification is needed and no labeled data is available.

 # One-shot sentiment classification. based on the previous code

Code:

```python
# One-shot sentiment classification. based on the 1st code
print("\nOne-shot Sentiment Classification:")
from textblob import TextBlob
review = "The product arrived late but works fine"
example_review = "I love this product!"
example_sentiment = "Positive"
def one_shot_classify_sentiment(review, example_review, example_sentiment):
    if example_sentiment == "Positive" and "love" in review.lower():
        return "Positive"
    elif example_sentiment == "Negative" and "terrible" in review.lower():
        return "Negative"
    else:
        return "Neutral"
sentiment = one_shot_classify_sentiment(review, example_review, example_sentiment)
print("Review:", review)
print("Sentiment:", sentiment)
```

Output:

```
One-shot Sentiment Classification:
Review: The product arrived late but works fine
Sentiment: Neutral
```

Justification:

In the one-shot prompt, one labeled example is provided to guide the model. This example helps the model **understand the expected format and sentiment logic** before classifying a new review.

Task 2: Email Priority Classification

Prompt :

#Generate the python code in which classify the priority of each e-mail into into High Priority, Medium Priority, or Low Priority.

Code:

```python
#Generate the python code in which classify the priority of each e-mail into
print("\n---Email Priority Classification---")
def classify_email_priority(email):
    high_priority_keywords = ["urgent", "asap", "immediate", "important"]
    medium_priority_keywords = ["soon", "reminder", "follow-up"]

    email_lower = email.lower()

    if any(keyword in email_lower for keyword in high_priority_keywords):
        return "High Priority"
    elif any(keyword in email_lower for keyword in medium_priority_keywords):
        return "Medium Priority"
    else:
        return "Low Priority"
emails = [
    "Please respond to this urgent request ASAP.",
    "Just a reminder about our meeting next week.",
    "Looking forward to catching up sometime.",
    "This is an important update regarding your account.",
    "Can we schedule a follow-up call soon?",
    "Hello, hope you're doing well!"
]
for email in emails:
    print(f"Email: {email}")
    print(f"Priority: {classify_email_priority(email)}")
```

Output:

```
---Email Priority Classification---
Email: Please respond to this urgent request ASAP.
Priority: High Priority
Email: Just a reminder about our meeting next week.
Priority: Medium Priority
Email: Looking forward to catching up sometime.
Priority: Low Priority
Email: This is an important update regarding your account.
Priority: High Priority
Email: Can we schedule a follow-up call soon?
Priority: Medium Priority
Email: Hello, hope you're doing well!
Priority: Low Priority
```

Justification:

This program classifies emails based on the urgency of words used in the message. Emails containing keywords like urgent, ASAP, or important are marked as High Priority, while reminder-related words are marked as Medium Priority. If no urgency-related keywords are found, the email is considered Low Priority.

 # One-shot email priority classification. based on the previous code

Code:

```
print("\nOne-shot Email Priority Classification:")
email = "Please address this issue as soon as possible."
example_email = "Please respond to this urgent request ASAP."
example_priority = "High Priority"
def one_shot_classify_email_priority(email, example_email, example_priority):
    if example_priority == "High Priority" and "urgent" in email.lower():
        return "High Priority"
    elif example_priority == "Medium Priority" and "reminder" in email.lower():
        return "Medium Priority"
    else:
        return "Low Priority"
priority = one_shot_classify_email_priority(email, example_email, example_priority)
print("Email:", email)
print("Priority:", priority)
```

Output:

```
One-shot Email Priority Classification:
Email: Please address this issue as soon as possible.
Priority: Low Priority
```

Justification:

In one-shot prompting, one labeled example is provided to guide the classification. This example helps the model understand how urgency maps to email priority. As a result, the system can classify new emails more accurately than without examples.

 # few-shot email priority classification. based on the previous code.

Code:

```python
print("\nFew-shot Email Priority Classification:")
email = "This is a follow-up regarding our last conversation."
examples = [
    ("Please respond to this urgent request ASAP.", "High Priority"),
    ("Just a reminder about our meeting next week.", "Medium Priority"),
    ("Looking forward to catching up sometime.", "Low Priority")
]
def few_shot_classify_email_priority(email, examples):
    for example_email, example_priority in examples:
        if example_priority == "High Priority" and "urgent" in email.lower():
            return "High Priority"
        elif example_priority == "Medium Priority" and "reminder" in email.lower():
            return "Medium Priority"
    return "Low Priority"
priority = few_shot_classify_email_priority(email, examples)
print("Email:", email)
print("Priority:", priority)
```

Output:

```
Few-shot Email Priority Classification:
Email: This is a follow-up regarding our last conversation.
Priority: Low Priority
PS D:\Python>
```

Justification:

Few-shot prompting uses multiple examples covering different priority levels.These examples give clearer context and improve consistency in classification.This method is more reliable when emails have similar or overlapping urgency.

Task 3: Student Query Routing System

Prompt :

#Generate the python code in which classify student queries into one of these departments like Admissions, Exams, Academics, or Placements.

Code:

```python
print("\n---Student Query Classification---")
def classify_student_query(query):
    admissions_keywords = ["admission", "apply", "enroll", "application"]
    exams_keywords = ["exam", "test", "schedule", "results"]
    academics_keywords = ["course", "subject", "syllabus", "curriculum"]
    placements_keywords = ["job", "internship", "placement", "career"]

    query_lower = query.lower()

    if any(keyword in query_lower for keyword in admissions_keywords):
        return "Admissions"
    elif any(keyword in query_lower for keyword in exams_keywords):
        return "Exams"
    elif any(keyword in query_lower for keyword in academics_keywords):
        return "Academics"
    elif any(keyword in query_lower for keyword in placements_keywords):
        return "Placements"
    else:
        return "General Inquiry"
queries = [
    "How do I apply for admission?",
    "When will the exam results be announced?",
    "What subjects are included in the syllabus?",
    "Are there any internship opportunities available?",
    "Can you provide information about the enrollment process?",
    "What career services does the college offer?"
]
for query in queries:
    print(f"Query: {query}")
    print(f"Department: {classify_student_query(query)}")
```

Output:

```
---Student Query Classification---
Query: How do I apply for admission?
Department: Admissions
Query: When will the exam results be announced?
Department: Exams
Query: What subjects are included in the syllabus?
Department: Academics
Query: Are there any internship opportunities available?
Department: Placements
Query: Can you provide information about the enrollment process?
Department: Admissions
Query: What career services does the college offer?
Department: Placements
```

Justification:

Sample student queries are created to reflect common questions asked in a university. Each query is mapped to the appropriate department based on its intent. This helps the chatbot understand how different types of student questions should be routed. It ensures students are directed to the correct department quickly

 # One-shot student query classification. based on the previous code

Code:

```
print("\nOne-shot Student Query Classification:")
query = "I want to know about the job placement process."
example_query = "How do I apply for admission?"
example_department = "Admissions"
def one_shot_classify_student_query(query, example_query, example_department):
    if example_department == "Admissions" and "admission" in query.lower():
        return "Admissions"
    elif example_department == "Exams" and "exam" in query.lower():
        return "Exams"
    elif example_department == "Academics" and "course" in query.lower():
        return "Academics"
    elif example_department == "Placements" and "job" in query.lower():
        return "Placements"
    else:
        return "General Inquiry"
department = one_shot_classify_student_query(query, example_query, example_department)
print("Query:", query)
print("Department:", department)
```

Output:

```
One-shot Student Query Classification:
Query: I want to know about the job placement process.
Department: General Inquiry
PS D:\Python>
```

Justification:

One-shot prompting uses a single labeled example to guide query classification. The example helps the model learn how to match a student's question with the right department. This improves routing accuracy compared to having no examples. It makes the chatbot more reliable in handling student queries.

Task 4: Chatbot Question Type Detection

Prompt :

#Generate the python code in which identify whether a user query is Informational,Transactional, Complaint, or Feedback.

Code:

```
print("\n---User Query Type Classification---")
def classify_user_query_type(query):
    informational_keywords = ["information", "details", "know", "learn"]
    transactional_keywords = ["buy", "purchase", "order", "book"]
    complaint_keywords = ["complaint", "issue", "problem", "bad"]
    feedback_keywords = ["feedback", "suggestion", "review", "comment"]

    query_lower = query.lower()

    if any(keyword in query_lower for keyword in informational_keywords):
        return "Informational"
    elif any(keyword in query_lower for keyword in transactional_keywords):
        return "Transactional"
    elif any(keyword in query_lower for keyword in complaint_keywords):
        return "Complaint"
    elif any(keyword in query_lower for keyword in feedback_keywords):
        return "Feedback"
    else:
        return "Other"
queries = [
    "I would like to know more about your services.",
    "I want to buy a new laptop.",
    "I have a complaint about my recent order.",
    "Here is some feedback on your website.",
    "Can you provide details about your pricing?",
    "I would like to book a table for two."
]
for query in queries:
    print(f"Query: {query}")
    print(f"Type: {classify_user_query_type(query)}")
```

Output:

```
---User Query Type Classification---
Query: I would like to know more about your services.
Type: Informational
Query: I want to buy a new laptop.
Type: Transactional
Query: I have a complaint about my recent order.
Type: Transactional
Query: Here is some feedback on your website.
Type: Feedback
Query: Can you provide details about your pricing?
Type: Informational
Query: I would like to book a table for two.
Type: Transactional
```

Justification:

This program identifies the type of user query using keyword matching. It classifies queries as Informational, Transactional, Complaint, or Feedback based on intent.This helps route user requests quickly and efficiently.

 # Few-shot user query type classification. based on the previous code.

Code:

```
print("\nFew-shot User Query Type Classification:")
query = "I have a problem with my account."
examples = [
    ("I would like to know more about your services.", "Informational"),
    ("I want to buy a new laptop.", "Transactional"),
    ("I have a complaint about my recent order.", "Complaint"),
    ("Here is some feedback on your website.", "Feedback")
]
def few_shot_classify_user_query_type(query, examples):
    for example_query, example_type in examples:
        if example_type == "Informational" and "information" in query.lower():
            return "Informational"
        elif example_type == "Transactional" and "buy" in query.lower():
            return "Transactional"
        elif example_type == "Complaint" and "complaint" in query.lower():
            return "Complaint"
        elif example_type == "Feedback" and "feedback" in query.lower():
            return "Feedback"
    return "Other"
query_type = few_shot_classify_user_query_type(query, examples)
print("Query:", query)
print("Type:", query_type)
```

Output:

```
Few-shot User Query Type Classification:
Query: I have a problem with my account.
Type: Other
```

Justification:

This program uses multiple labeled examples to guide query classification. It compares the new query with known example intents using keywords.

This improves intent identification compared to having no examples. It helps the system respond more accurately to user requests.

Task 5: Emotion Detection in Text

Prompt :

#Generate the python code in which it detects emotions like Happy, Sad, Angry,Anxious, Neutral.

Code:

```python
def detect_emotion(query):
    happy_keywords = ["happy", "joy", "excited", "pleased"]
    sad_keywords = ["sad", "depressed", "unhappy", "miserable"]
    angry_keywords = ["angry", "furious", "irritated", "mad"]
    anxious_keywords = ["anxious", "worried", "nervous", "concerned"]
    neutral_keywords = ["neutral", "indifferent", "calm"]

    query_lower = query.lower()

    if any(keyword in query_lower for keyword in happy_keywords):
        return "Happy"
    elif any(keyword in query_lower for keyword in sad_keywords):
        return "Sad"
    elif any(keyword in query_lower for keyword in angry_keywords):
        return "Angry"
    elif any(keyword in query_lower for keyword in anxious_keywords):
        return "Anxious"
    elif any(keyword in query_lower for keyword in neutral_keywords):
        return "Neutral"
    else:
        return "Unknown"

emotion_queries = [
    ("I am very happy with the service.", "Happy"),
    ("I feel sad about the situation.", "Sad"),
    ("I am angry about the delay.", "Angry"),
    ("I am feeling anxious about the exam.", "Anxious"),
    ("I am calm and collected.", "Neutral")
]

for query, expected_emotion in emotion_queries:
    detected_emotion = detect_emotion(query)
    print(f"Query: {query}")
    print(f"Detected Emotion: {detected_emotion}")
    print(f"Expected Emotion: {expected_emotion}\n")
```

Output:

```
---Emotion Detection---
Query: I am very happy with the service.
Detected Emotion: Happy
Expected Emotion: Happy

Query: I feel sad about the situation.
Detected Emotion: Sad
Expected Emotion: Sad

Query: I am angry about the delay.
Detected Emotion: Angry
Expected Emotion: Angry

Query: I am feeling anxious about the exam.
Detected Emotion: Anxious
Expected Emotion: Anxious

Query: I am calm and collected.
Detected Emotion: Neutral
Expected Emotion: Neutral
```

Justification:

This task detects emotions in user text using keyword matching. Each emotion is identified based on commonly used emotional words in the query. The detected emotion is compared with the expected emotion to verify correctness. This helps systems understand user feelings.

 # One-shot emotion detection. based on the previous code.

Code:

```python
query = "I am so excited about my new job!"
example_query = "I am very happy with the service."
example_emotion = "Happy"
def one_shot_detect_emotion(query, example_query, example_emotion):
    if example_emotion == "Happy" and "happy" in query.lower():
        return "Happy"
    elif example_emotion == "Sad" and "sad" in query.lower():
        return "Sad"
    elif example_emotion == "Angry" and "angry" in query.lower():
        return "Angry"
    elif example_emotion == "Anxious" and "anxious" in query.lower():
        return "Anxious"
    elif example_emotion == "Neutral" and "neutral" in query.lower():
        return "Neutral"
    else:
        return "Unknown"
emotion = one_shot_detect_emotion(query, example_query, example_emotion)
print("Query:", query)
print("Detected Emotion:", emotion)
```

Output:

```
One-shot Emotion Detection:
Query: I am so excited about my new job!
Detected Emotion: Unknown
```

Justification:

This program detects emotion using one labeled example as guidance. The example helps the system understand how a specific emotion is expressed in text. The input query is compared with the example using keywords. This demonstrates the concept of one-shot prompting in emotion classification.

 # Few-shot emotion detection. based on the previous code.

Code:

```
query = "I am worried about the upcoming presentation."
examples = [
    ("I am very happy with the service.", "Happy"),
    ("I feel sad about the situation.", "Sad"),
    ("I am angry about the delay.", "Angry"),
    ("I am feeling anxious about the exam.", "Anxious"),
    ("I am calm and collected.", "Neutral")
]
def few_shot_detect_emotion(query, examples):
    for example_query, example_emotion in examples:
        if example_emotion == "Happy" and "happy" in query.lower():
            return "Happy"
        elif example_emotion == "Sad" and "sad" in query.lower():
            return "Sad"
        elif example_emotion == "Angry" and "angry" in query.lower():
            return "Angry"
        elif example_emotion == "Anxious" and "anxious" in query.lower():
            return "Anxious"
        elif example_emotion == "Neutral" and "neutral" in query.lower():
            return "Neutral"
    return "Unknown"
emotion = few_shot_detect_emotion(query, examples)
print("Query:", query)
print("Detected Emotion:", emotion)
```

Output:

```
Few-shot Emotion Detection:
Query: I am worried about the upcoming presentation.
Detected Emotion: Unknown
```

Justification:

This program identifies emotions using multiple labeled examples.
The examples help the system recognize different emotional patterns in text. The input query is compared with known emotion keywords. This improves emotion detection accuracy compared to one-shot or zero-shot methods