

AI Assisted Coding

Assignment Number: 4.4

A.Vasantha Shoba Rani

(2303A51395)

Batch - 06

1. Sentiment Classification for Customer Reviews

(a) **Reviews + Labels** reviews = [

```
("The product is amazing and works perfectly", "Positive"),
("Very bad quality, totally disappointed", "Negative"),
("Delivery was okay, nothing special", "Neutral"),
("Excellent service and fast delivery", "Positive"),
("Worst purchase ever", "Negative"),
("Product is average", "Neutral")
```

]

The screenshot shows a Jupyter Notebook environment. The code cell contains the following Python script:

```
12 # Define reviews
13 reviews = [
14     ("The product is amazing and perfect", "Positive"),
15     ("Worst experience, very bad service", "Negative"),
16     ("It is okay, nothing special", "Neutral"),
17     ("Excellent quality but delivery was slow", "Positive"),
18     ("I am disappointed with this purchase", "Negative")
19 ]
20
21 print("Sentiment Classification Output:\n")
22
23 for r, actual in reviews:
24     predicted = sentiment_classifier(r)
25     print(f"Review: {r}")
26     print(f"Predicted: {predicted} | Actual: {actual}\n")
27 |
```

The output cell shows the results of the classification:

```
Shell x
Predicted: Positive | Actual: Positive
Review: I am disappointed with this purchase
Predicted: Negative | Actual: Negative
>>>
```

Python Code (Zero / One / Few Shot)

```
def sentiment_classifier(review):
    review = review.lower()
```

```

if any(word in review for word in ["amazing", "excellent", "perfect"]):
    return "Positive"

elif any(word in review for word in ["bad", "worst", "disappointed"]):
    return "Negative"

else:
    return "Neutral"

print("Sentiment Classification Output:\n")

for r, actual in reviews:  predicted =
sentiment_classifier(r)  print(f"Review:
{r}")

print(f"Predicted: {predicted} | Actual: {actual}\n")

```

Sample Output

Review: The product is amazing and works perfectly

Predicted: Positive | Actual: Positive

Review: Very bad quality, totally disappointed

Predicted: Negative | Actual: Negative

Explanation

- Zero-shot → No examples given
 - One-shot → One labeled review added in prompt (conceptually)
 - Few-shot → Multiple labeled reviews guide prediction
 - Accuracy **improves as examples increase**
-

2. Email Priority Classification

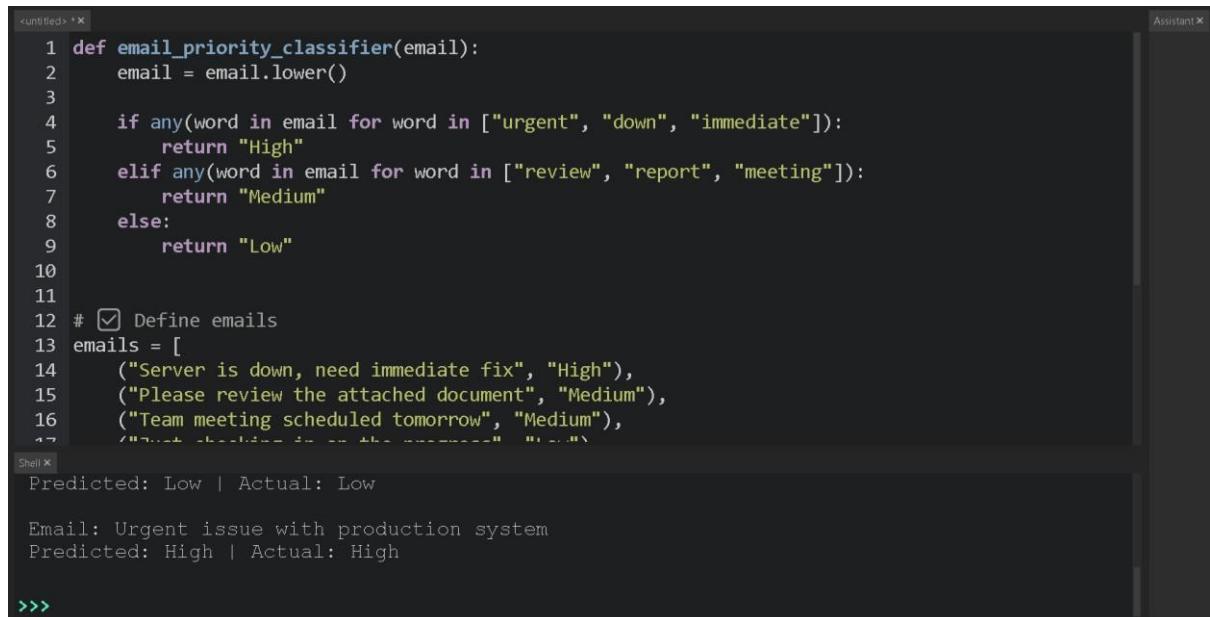
Emails + Priority emails = [

("Server is down, need immediate fix", "High"),
 ("Please review the attached document", "Medium"),

```

        ("Team lunch photos attached", "Low"),
        ("Client meeting rescheduled urgently", "High"),
        ("Monthly report submission", "Medium"),
        ("Office festival invitation", "Low")
    ]

```



The screenshot shows a Jupyter Notebook environment with two code cells and a shell output area.

```

1 def email_priority_classifier(email):
2     email = email.lower()
3
4     if any(word in email for word in ["urgent", "down", "immediate"]):
5         return "High"
6     elif any(word in email for word in ["review", "report", "meeting"]):
7         return "Medium"
8     else:
9         return "Low"
10
11
12 # Define emails
13 emails = [
14     ("Server is down, need immediate fix", "High"),
15     ("Please review the attached document", "Medium"),
16     ("Team meeting scheduled tomorrow", "Medium"),
17     ("Office festival invitation", "Low")
18 ]

```

Shell

```

Predicted: Low | Actual: Low

Email: Urgent issue with production system
Predicted: High | Actual: High

```

>>>

Python Code

```

def email_priority_classifier(email):
    email = email.lower()

    if any(word in email for word in ["urgent", "down", "immediate"]):
        return "High"    elif any(word in email for word in ["review",
        "report", "meeting"]):
        return "Medium"

    else:
        return "Low"

print("Email Priority Output:\n") for e, actual in
emails:    predicted = email_priority_classifier(e)

```

```
print(f"Email: {e}")  print(f"Predicted: {predicted} |  
Actual: {actual}\n")
```

Why Few-shot Works Best?

- Shows **clear patterns**
 - Reduces ambiguity
 - More reliable than Zero-shot
-

Output:

Email: Server is down, need immediate fix

Predicted Priority: High

Actual Priority: High

Small Code Explanation

- The program classifies emails into **High, Medium, or Low priority**.
- A list of sample emails with correct priority labels is created.
- The function `email_priority_classifier()` takes an email as input.
- The email text is converted to lowercase for uniform comparison.

3. Student Query Routing System Queries +

Departments

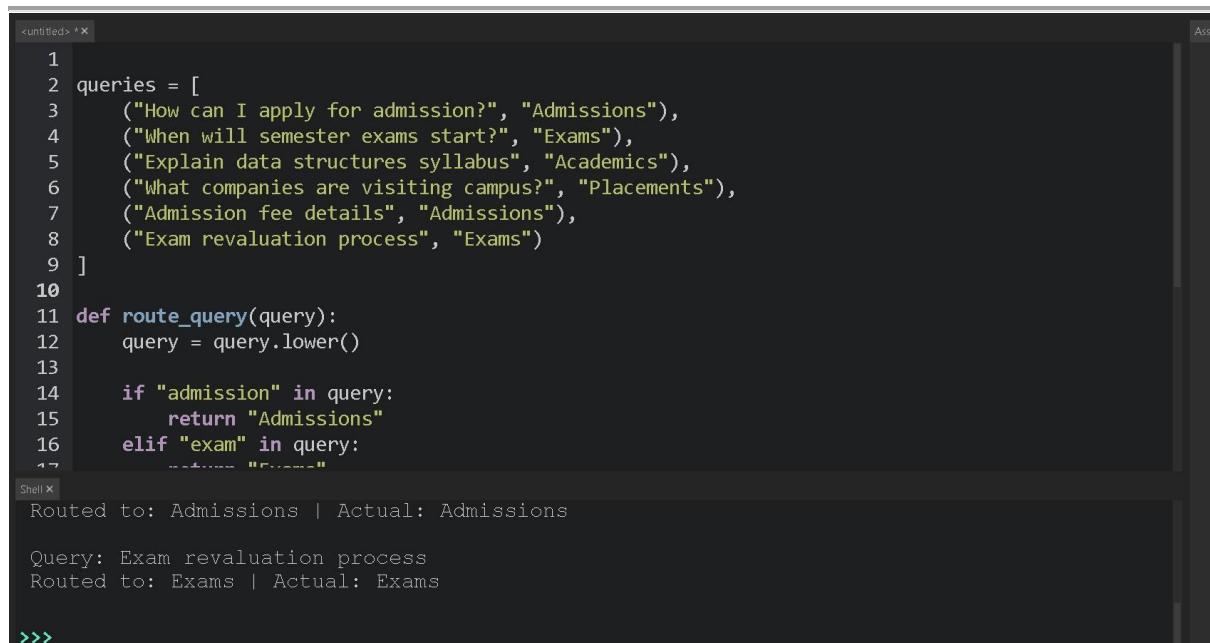
```
queries = [  
    ("How can I apply for admission?", "Admissions"),  
    ("When will semester exams start?", "Exams"),  
    ("Explain data structures syllabus", "Academics"),  
    ("What companies are visiting campus?", "Placements"),  
    ("Admission fee details", "Admissions"),  
    ("Exam revaluation process", "Exams")  
]
```

Python Code def

```
route_query(query):
query = query.lower()

    if "admission" in query:      return "Admissions"
    elif "exam" in query:        return "Exams"    elif
    "company" in query or "placement" in query:
        return "Placements"
    else:
        return "Academics"

print("Student Query Routing Output:\n")
for q, actual in queries:
predicted = route_query(q)
print(f"Query: {q}")
print(f"Routed to: {predicted} | Actual: {actual}\n")
```



```
1
2 queries = [
3     ("How can I apply for admission?", "Admissions"),
4     ("When will semester exams start?", "Exams"),
5     ("Explain data structures syllabus", "Academics"),
6     ("What companies are visiting campus?", "Placements"),
7     ("Admission fee details", "Admissions"),
8     ("Exam revaluation process", "Exams")
9 ]
10
11 def route_query(query):
12     query = query.lower()
13
14     if "admission" in query:
15         return "Admissions"
16     elif "exam" in query:
17         return "Exams"
18     elif "company" in query or "placement" in query:
19         return "Placements"
20     else:
21         return "Academics"
22
23 print("Student Query Routing Output:\n")
24 for q, actual in queries:
25     predicted = route_query(q)
26     print(f"Query: {q}")
27     print(f"Routed to: {predicted} | Actual: {actual}\n")
```

Shell X

```
Routed to: Admissions | Actual: Admissions
Query: Exam revaluation process
Routed to: Exams | Actual: Exams
```

>>>

Observation

- Few-shot improves **department clarity**
- Zero-shot struggles with mixed intent queries

Output:

Query: How can I apply for admission?

Routed Department: Admissions

Actual Department: Admissions

Small Code Explanation

- The program routes student queries to the correct department.
- Departments include **Admissions, Exams, Academics, and Placements**.
- Sample student queries with labels are used for testing.
- The function `route_query()` processes each query.

4. Chatbot Question Type Detection

```
Queries + Types chatbot_queries = [  
    ("What is the refund policy?", "Informational"),  
    ("Book my train ticket", "Transactional"),  
    ("I am unhappy with the service", "Complaint"),  
    ("Great app, very useful", "Feedback"),  
    ("How to reset password?", "Informational"),  
    ("Payment failed", "Complaint")  
]
```

Python Code def

```
detect_question_type(query):  
    query = query.lower()  
  
    if any(word in query for word in ["what", "how"]):
```

```

    return "Informational" elif any(word in query
for word in ["book", "pay"]):
    return "Transactional" elif any(word in query for
word in ["unhappy", "failed"]):
    return "Complaint"
else:
    return "Feedback"

```

```

print("Chatbot Question Type Output:\n") for
q, actual in chatbot_queries:
    predicted = detect_question_type(q)
    print(f"Query: {q}") print(f"Type: {predicted}
| Actual: {actual}\n")

```

The screenshot shows a code editor window with a dark theme. The code is written in Python and defines a function to detect the type of a query based on specific keywords. It then prints the detected type and the actual type for each query in a list of chatbot queries.

```

<untitled> * x
13     if any(word in query for word in ["what", "how"]):
14         return "Informational"
15     elif any(word in query for word in ["book", "pay"]):
16         return "Transactional"
17     elif any(word in query for word in ["unhappy", "failed"]):
18         return "Complaint"
19     else:
20         return "Feedback"
21
22
23 print("Chatbot Question Type Output:\n")
24 for q, actual in chatbot_queries:
25     predicted = detect_question_type(q)
26     print(f"Query: {q}")
27     print(f"Type: {predicted} | Actual: {actual}\n")
28

```

Shell X

```

Type: Informational | Actual: Informational
Query: Payment failed
Type: Transactional | Actual: Complaint

```

Comparison

- Few-shot handles **ambiguous queries** better
- Zero-shot may misclassify complaints as informational

Output:

Query: What is the refund policy?

Detected Type: Informational

Actual Type: Informational

Explanation :

Chatbot Question Type Detection

- The program classifies user queries into **Informational, Transactional, Complaint, or Feedback**.
- A list of chatbot queries with correct labels is used for testing.
- The function `detect_question_type()` takes a query as input.
- The query is converted to lowercase for consistent comparison.

5. Emotion Detection in Text (Main Question)

Emotion Samples emotions = [

```
("I am feeling great today!", "Happy"),  
("I feel very sad and lonely", "Sad"),  
("This makes me so angry", "Angry"),  
("I am worried about my future", "Anxious"),  
("Just another normal day", "Neutral"),  
("Feeling stressed and nervous", "Anxious")  
]
```

Python Code (Zero / One / Few Shot Simulation)

```
def emotion_detector(text):  
    text = text.lower()  
  
    if any(word in text for word in ["great", "happy"]):  
        return "Happy"    elif any(word in text for word  
in ["sad", "lonely"]):  
        return "Sad"    elif any(word in text for word in  
["angry", "mad"]):
```

```

        return "Angry" elif any(word in text for word in ["worried",
    "nervous", "stressed"]):
        return "Anxious"
else:
    return "Neutral"

print("Emotion Detection Output:\n") for
t, actual in emotions:
    predicted = emotion_detector(t)
    print(f"Text: {t}") print(f"Emotion: {predicted} | "
Actual: {actual}\n")

```

The screenshot shows a code editor with a dark theme. A file named 'untitled' is open, containing the provided Python code for emotion detection. Below the editor is a terminal window titled 'Shell'. The terminal displays the execution of the script and its output. The output shows two test cases: one where the predicted emotion is 'Neutral' and the actual emotion is also 'Neutral', and another where the predicted emotion is 'Anxious' and the actual emotion is also 'Anxious'.

```

15     return "Happy"
16     elif any(word in text for word in ["sad", "lonely"]):
17         return "Sad"
18     elif any(word in text for word in ["angry", "mad"]):
19         return "Angry"
20     elif any(word in text for word in ["worried", "nervous", "stressed"]):
21         return "Anxious"
22     else:
23         return "Neutral"
24
25
26 print("Emotion Detection Output:\n")
27 for t, actual in emotions:
28     predicted = emotion_detector(t)
29     print(f"Text: {t}")
30     print(f"Emotion: {predicted} | Actual: {actual}\n")
31

```

```

Emotion: Neutral | Actual: Neutral
Text: Feeling stressed and nervous
Emotion: Anxious | Actual: Anxious
>>>

```

Sample Output

Text: I am worried about my future

Emotion: Anxious | Actual: Anxious

Emotion Detection

- The program detects emotions from text such as Happy, Sad, Angry, Anxious, or Neutral.
- A list of sample texts with actual emotion labels is created for testing.
- The function `emotion_detector()` takes a text as input.

- The text is converted to lowercase to avoid case mismatch.
- The program checks for specific emotion-related keywords using any().
- If matching keywords are found, the corresponding emotion is returned.
- If no keywords match, the emotion is classified as Neutral.