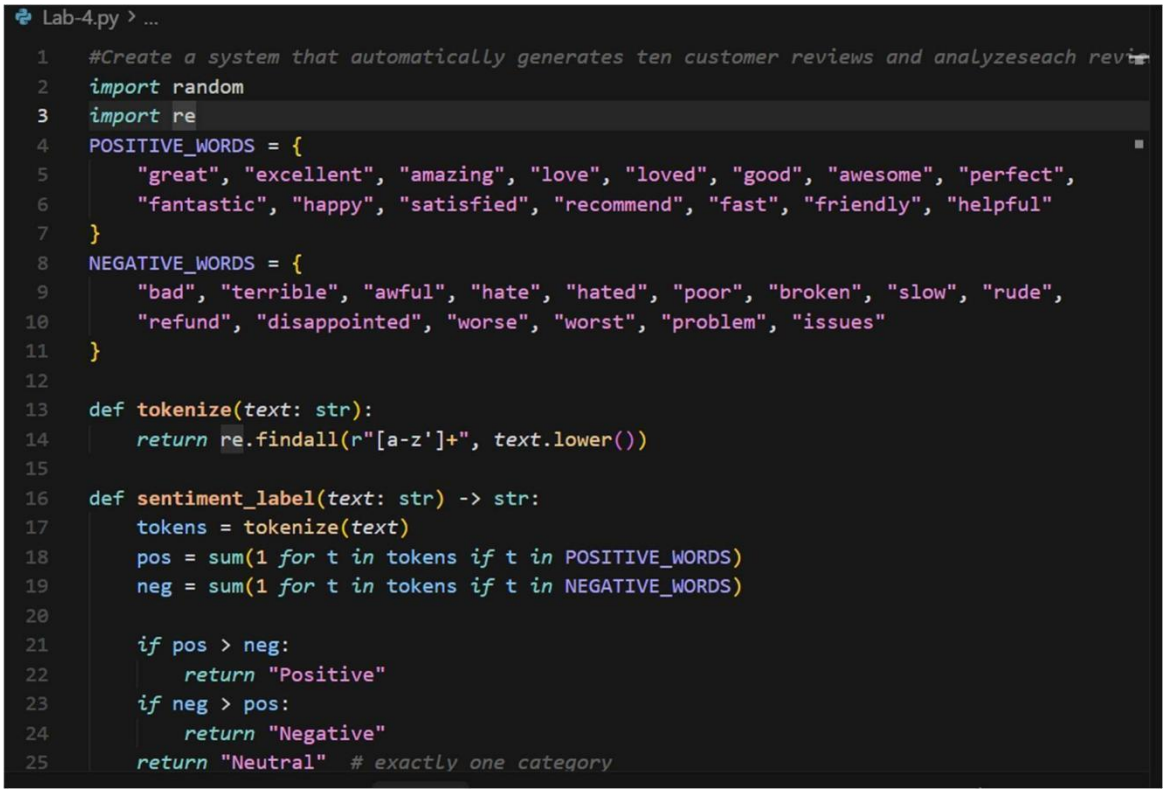
ASSIGNMENT-04

G.SAI TEJA
2303A52135
BATCH-41

Task 1: Sentiment Classification for Customer Reviews

ZERO SHOT PROMPT: Create a system that automa cally generates ten customer reviews and analyzeseach review to determine its sen ment. Every review must be classified intoexactly one category: Posi ve, Nega ve, or Neutral. The solu on should be clear, efficient, and easy to understand.

CODE:



```python
#Create a system that automatically generates ten customer reviews and analyzeseach revi
import random
import re
POSITIVE_WORDS = {
    "great", "excellent", "amazing", "love", "loved", "good", "awesome", "perfect",
    "fantastic", "happy", "satisfied", "recommend", "fast", "friendly", "helpful"
}
NEGATIVE_WORDS = {
    "bad", "terrible", "awful", "hate", "hated", "poor", "broken", "slow", "rude",
    "refund", "disappointed", "worse", "worst", "problem", "issues"
}

def tokenize(text: str):
    return re.findall(r"[a-z']+", text.lower())

def sentiment_label(text: str) -> str:
    tokens = tokenize(text)
    pos = sum(1 for t in tokens if t in POSITIVE_WORDS)
    neg = sum(1 for t in tokens if t in NEGATIVE_WORDS)

    if pos > neg:
        return "Positive"
    if neg > pos:
        return "Negative"
    return "Neutral"  # exactly one category
```

Lab-4.py > ...

```
    return "Neutral"  # exactly one category

def generate_reviews(n=10, seed=7):
    random.seed(seed)

    templates = [
        "The delivery was {delivery} and the staff was {staff}.",
        "Product quality is {quality}; I'm {feeling} with the purchase.",
        "Support was {support}, but the app is {app}.",
        "Packaging was {packaging} and the item arrived {arrival}.",
        "Overall experience was {overall}.",
    ]

    delivery_opts = ["fast", "slow", "on time"]
    staff_opts = ["friendly", "rude", "professional"]
    quality_opts = ["excellent", "good", "poor", "average"]
    feeling_opts = ["happy", "satisfied", "disappointed", "okay"]
    support_opts = ["helpful", "great", "terrible", "okay"]
    app_opts = ["amazing", "broken", "fine", "average"]
    packaging_opts = ["perfect", "bad", "standard", "fine"]
    arrival_opts = ["on time", "late", "early"]
    overall_opts = ["excellent", "great", "neutral", "terrible", "average"]

    reviews = []
```

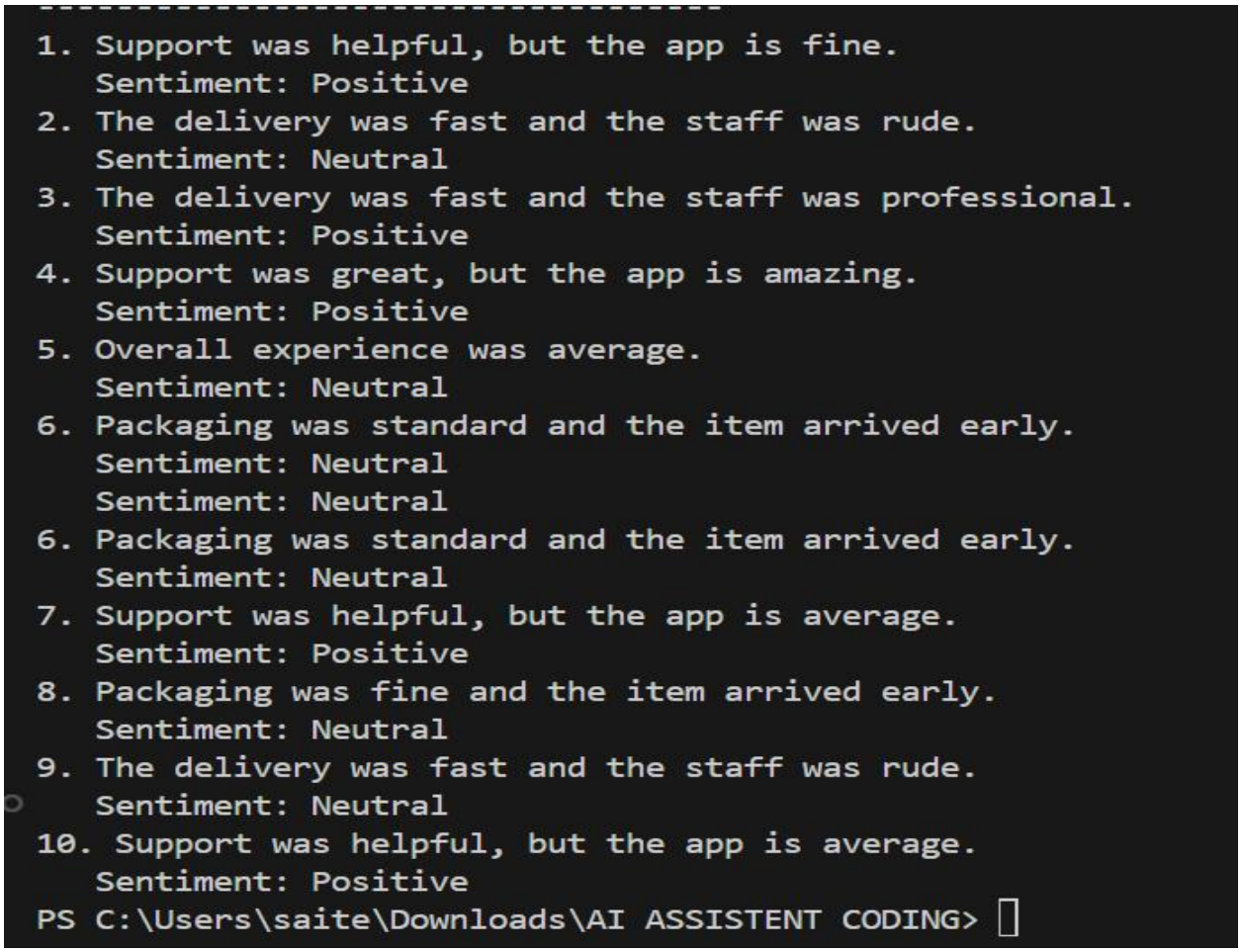
Lab-4.py > ...

```
def generate_reviews(n=10, seed=7):
    reviews = []
    for _ in range(n):
        t = random.choice(templates)
        review = t.format(
            delivery=random.choice(delivery_opts),
            staff=random.choice(staff_opts),
            quality=random.choice(quality_opts),
            feeling=random.choice(feeling_opts),
            support=random.choice(support_opts),
            app=random.choice(app_opts),
            packaging=random.choice(packaging_opts),
            arrival=random.choice(arrival_opts),
            overall=random.choice(overall_opts),
        )
        reviews.append(review)
    return reviews

if __name__ == "__main__":
    reviews = generate_reviews(n=10, seed=7)

    print("Generated Reviews + Sentiment")
    print("-" * 35)
    for i, r in enumerate[Any](reviews, 1):
        print(f"{i}. {r}")
        print(f"   Sentiment: {sentiment_label(r)}")
```

OUTPUT:



JUSTIFICATION:

This code generates 10 random customer reviews using templates and fills them with random options like delivery speed, staff behavior, product quality, etc. It then tokenizes each review into words using regular expressions and counts how many words match predefined positive or negative word lists. Based on the count, it classifies the review as Positive, Negative, or Neutral, ensuring only one label is assigned. The program prints each review along with its sentiment label. Overall, it demonstrates basic sentiment analysis using keyword matching and random review generation.

□ZERO SHOT PROMPT: Analyze each customer review and identify its sentiment.

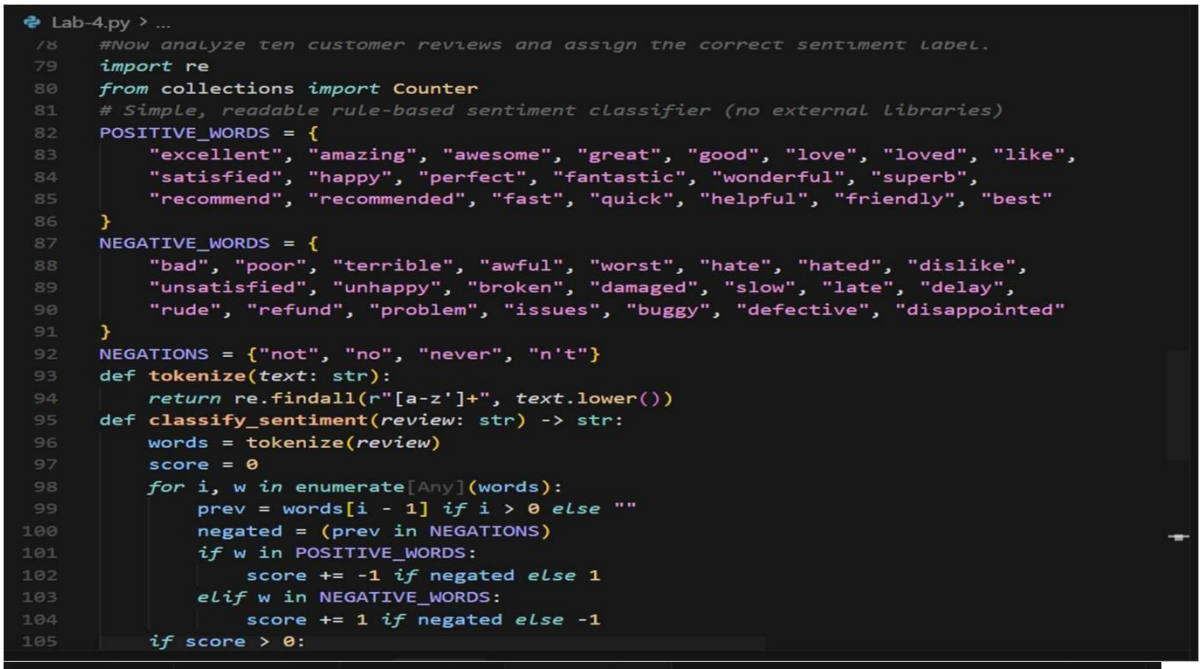
Classify every review into exactly one category: Positive, Negative, or Neutral.

Example:

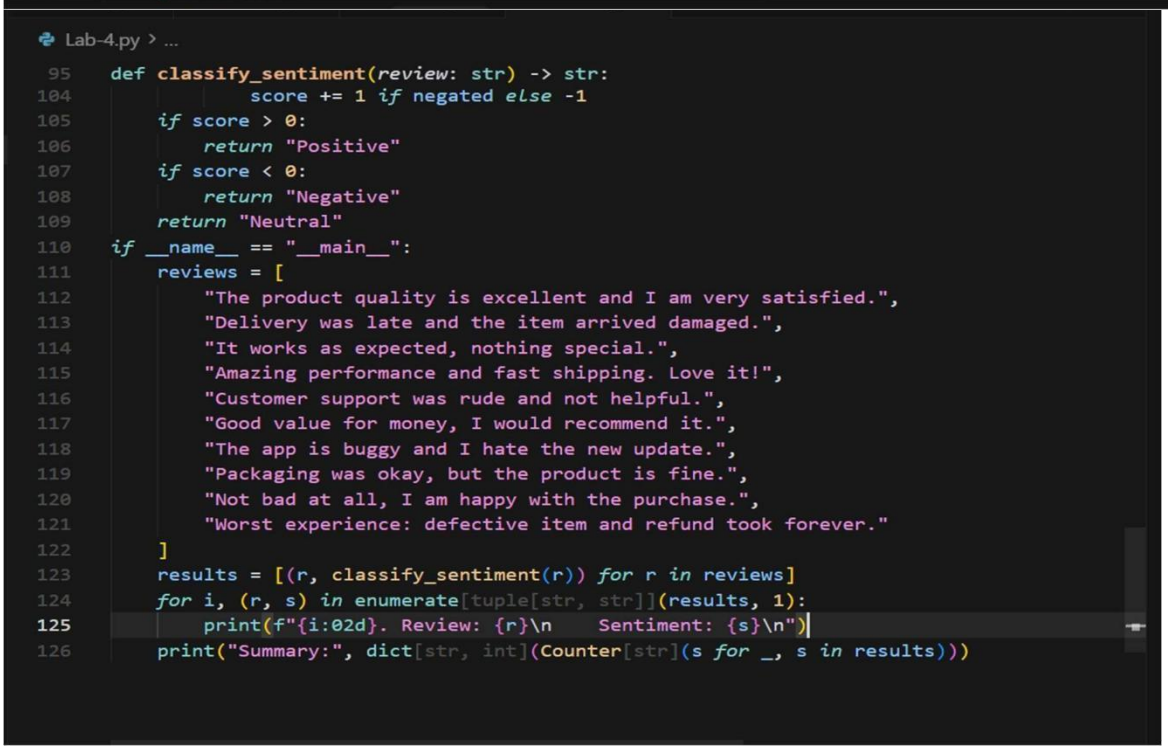
Customer Review: The product quality is excellent and I am very satisfied.

Sentiment: Positive

Now analyze ten customer reviews and assign the correct sentiment label. CODE:



```python
#Now analyze ten customer reviews and assign the correct sentiment label.
import re
from collections import Counter
# Simple, readable rule-based sentiment classifier (no external libraries)
POSITIVE_WORDS = {
    "excellent", "amazing", "awesome", "great", "good", "love", "loved", "like",
    "satisfied", "happy", "perfect", "fantastic", "wonderful", "superb",
    "recommend", "recommended", "fast", "quick", "helpful", "friendly", "best"
}
NEGATIVE_WORDS = {
    "bad", "poor", "terrible", "awful", "worst", "hate", "hated", "dislike",
    "unsatisfied", "unhappy", "broken", "damaged", "slow", "late", "delay",
    "rude", "refund", "problem", "issues", "buggy", "defective", "disappointed"
}
NEGATIONS = {"not", "no", "never", "n't"}
def tokenize(text: str):
    return re.findall(r"[a-z']+", text.lower())
def classify_sentiment(review: str) -> str:
    words = tokenize(review)
    score = 0
    for i, w in enumerate[Any](words):
        prev = words[i - 1] if i > 0 else ""
        negated = (prev in NEGATIONS)
        if w in POSITIVE_WORDS:
            score += -1 if negated else 1
        elif w in NEGATIVE_WORDS:
            score += 1 if negated else -1
    if score > 0:
```



```python
def classify_sentiment(review: str) -> str:
            score += 1 if negated else -1
    if score > 0:
        return "Positive"
    if score < 0:
        return "Negative"
    return "Neutral"
if __name__ == "__main__":
    reviews = [
        "The product quality is excellent and I am very satisfied.",
        "Delivery was late and the item arrived damaged.",
        "It works as expected, nothing special.",
        "Amazing performance and fast shipping. Love it!",
        "Customer support was rude and not helpful.",
        "Good value for money, I would recommend it.",
        "The app is buggy and I hate the new update.",
        "Packaging was okay, but the product is fine.",
        "Not bad at all, I am happy with the purchase.",
        "Worst experience: defective item and refund took forever."
    ]
    results = [(r, classify_sentiment(r)) for r in reviews]
    for i, (r, s) in enumerate[tuple[str, str]](results, 1):
        print(f"{i:02d}. Review: {r}\n    Sentiment: {s}\n")
    print("Summary:", dict[str, int](Counter[str](s for _, s in results)))
```

OUTPUT:

```
te/Downloads/AI ASSISTENT CODING/As4.py/task2.py
01. Review: The product quality is excellent and I am very satisf
ied.
    Sentiment: Positive

02. Review: Delivery was late and the item arrived damaged.
    Sentiment: Negative

03. Review: It works as expected, nothing special.
    Sentiment: Neutral

04. Review: Amazing performance and fast shipping. Love it!
    Sentiment: Positive

05. Review: Customer support was rude and not helpful.
    Sentiment: Negative

06. Review: Good value for money, I would recommend it.
    Sentiment: Positive

07. Review: The app is buggy and I hate the new update.
    Sentiment: Negative

08. Review: Packaging was okay, but the product is fine.
    Sentiment: Neutral

09. Review: Not bad at all, I am happy with the purchase.
    Sentiment: Positive

10. Review: Worst experience: defective item and refund took fore
ver.
    Sentiment: Negative
```

JUSTIFICATION:

This Python program performs basic rule-based sentiment analysis without using any external libraries. It defines lists of positive and negative words, and also handles negation words like "not" and "never" to reverse sentiment when needed. Each review is tokenized into words, and a sentiment score is calculated by adding +1 for positive words and -1 for negative words, reversing the score if the word is preceded by a negation. Based on the final score, the review is classified as Positive, Negative, or Neutral. Finally, the program prints each review with its sentiment and shows a summary count of each sentiment category.

Task 2: Email Priority Classification

ONE SHOT PROMPT: *You are an AI system that classifies emails into exactly one priority level:High Priority, Medium Priority, or Low Priority.Example:Input Email: "The production server is down and needs immediate attention."Output Priority: High Priority.*

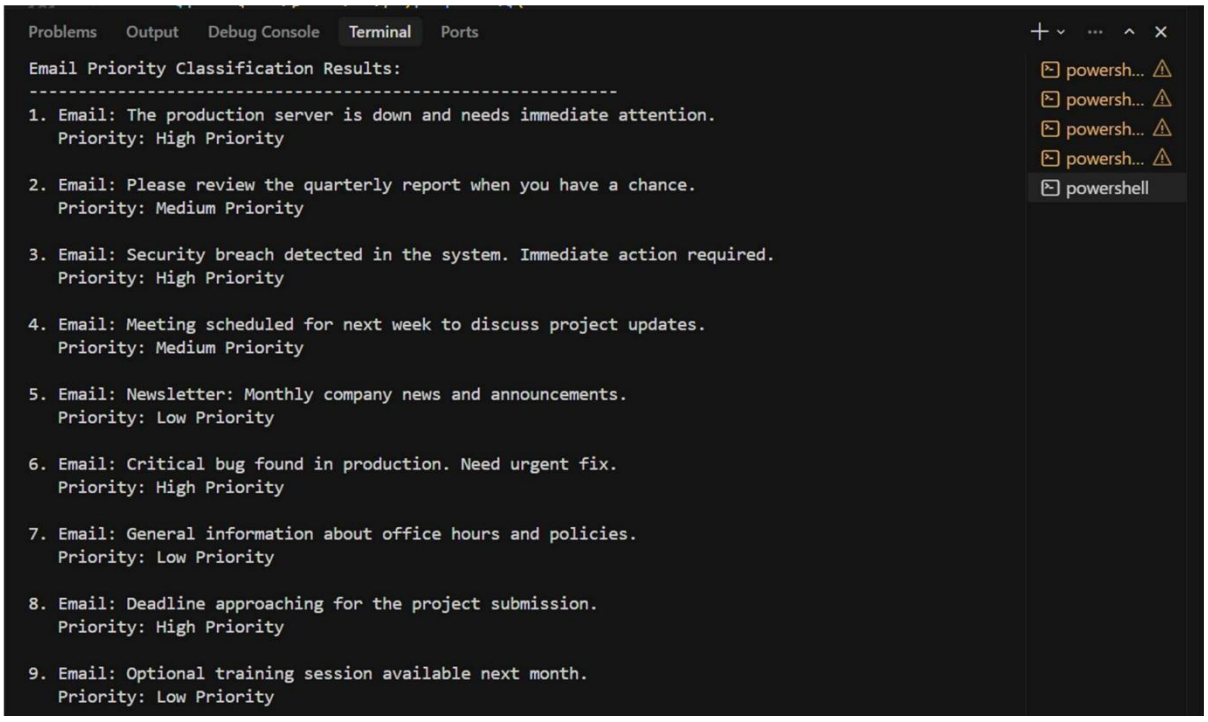
CODE:

```python
import re
# Keyword sets for priority classification
HIGH_PRIORITY_WORDS = {
    "urgent", "immediate", "critical", "emergency", "down", "broken", "failed",
    "crash", "outage", "security", "breach", "hack", "attack", "deadline",
    "asap", "as soon as possible", "important", "escalate", "priority"
}
MEDIUM_PRIORITY_WORDS = {
    "request", "needed", "required", "update", "review", "meeting", "schedule",
    "follow up", "reminder", "pending", "issue", "problem", "concern"
}
LOW_PRIORITY_WORDS = {
    "newsletter", "update", "information", "general", "when convenient",
    "no rush", "optional", "news", "announcement"
}
def tokenize(text: str):
    """Convert text to lowercase tokens."""
    return re.findall(r"[a-z']+", text.lower())
def classify_priority(email: str) -> str:
    """
    Classify email into exactly one priority level: High, Medium, or Low Priority.
    Args:
        email: The email text to classify
    Returns:
        "High Priority", "Medium Priority", or "Low Priority"
    """
    words = tokenize(email)
    text_lower = email.lower()
```

```python
    words = tokenize(email)
    text_lower = email.lower()
    # Count keyword matches
    high_count = sum(1 for word in words if word in HIGH_PRIORITY_WORDS)
    medium_count = sum(1 for word in words if word in MEDIUM_PRIORITY_WORDS)
    low_count = sum(1 for word in words if word in LOW_PRIORITY_WORDS)
    # Check for multi-word phrases
    if any(phrase in text_lower for phrase in ["immediate attention", "as soon as possible", "asap"]):
        high_count += 2
    if any(phrase in text_lower for phrase in ["follow up", "no rush", "when convenient"]):
        if "no rush" in text_lower or "when convenient" in text_lower:
            low_count += 2
        else:
            medium_count += 1
    # Classification logic: High > Medium > Low
    if high_count > 0:
        return "High Priority"
    elif medium_count > 0:
        return "Medium Priority"
    else:
        return "Low Priority"
if __name__ == "__main__":
    # Test with the example
    test_email = "The production server is down and needs immediate attention."
    result = classify_priority(test_email)
    print(f"Input Email: {test_email}")
```

```python
if __name__ == "__main__":
    # Test with the example
    test_email = "The production server is down and needs immediate attention."
    result = classify_priority(test_email)
    print(f"Input Email: {test_email}")
    print(f"Output Priority: {result}\n")
    # Additional test cases
    test_emails = [
        "The production server is down and needs immediate attention.",
        "Please review the quarterly report when you have a chance.",
        "Security breach detected in the system. Immediate action required.",
        "Meeting scheduled for next week to discuss project updates.",
        "Newsletter: Monthly company news and announcements.",
        "Critical bug found in production. Need urgent fix.",
        "General information about office hours and policies.",
        "Deadline approaching for the project submission.",
        "Optional training session available next month.",
        "System outage affecting all users. Escalate immediately."
    ]
    print("Email Priority Classification Results:")
    print("-" * 60)
    for i, email in enumerate[str](test_emails, 1):
        priority = classify_priority(email)
        print(f"{i}. Email: {email}")
        print(f"   Priority: {priority}\n")
```

OUTPUT:



JUSTIFICATION :

This Python program classifies emails into High, Medium, or Low priority using keyword matching. It tokenizes the email text, counts matching keywords from predefined priority sets, and boosts scores for key phrases like “immediate attention.” The classification follows a hierarchy where High overrides Medium and Low. Finally, it tests the classifier on sample emails and prints the results.

FEW SHOT PROMPT: Develop a program that performs email priority classification by categorizing incoming emails into one of three labels: High Priority, Medium Priority, or Low Priority. First, study the provided examples such as an email about a system outage affecting all customers being labeled as High Priority, a request to review a document and provide feedback by evening being labeled as Medium Priority, and a seasonal greeting from the HR team being labeled as Low Priority. Using these examples as reference, generate ten new incoming email messages and determine the appropriate priority label for each one. The solution should include efficient, well-organized, and easily readable code that accurately assigns exactly one priority level to every email.

CODE:

Lab-4.py > ...

```
206  import re
207  from typing import List, Tuple
208  # Keyword sets for priority classification
209  HIGH_PRIORITY_KEYWORDS = {
210      "urgent", "immediate", "critical", "emergency", "down", "broken", "failed",
211      "crash", "outage", "security", "breach", "hack", "attack", "deadline",
212      "asap", "escalate", "unable", "cannot", "can't", "stopped", "blocked",
213      "unavailable", "disaster", "severe", "critical", "production"
214  }
215  HIGH_PRIORITY_PHRASES = [
216      "as soon as possible", "immediate attention", "right away",
217      "cannot access", "unable to", "system down", "server down",
218      "production down", "out of service", "security breach"
219  ]
220  MEDIUM_PRIORITY_KEYWORDS = {
221      "request", "needed", "required", "update", "review", "meeting", "schedule",
222      "follow up", "reminder", "pending", "issue", "problem", "concern",
223      "discuss", "proposal", "feedback", "approval", "response"
224  }
225  MEDIUM_PRIORITY_PHRASES = [
226      "when you have time", "at your convenience", "follow up",
227      "would like to", "please review", "need your input"
228  ]
229  LOW_PRIORITY_KEYWORDS = {
230      "newsletter", "information", "general", "optional", "news",
231      "announcement", "update", "reminder", "fyi", "for your information",
232      "no rush", "whenever", "leisurely"
```

Lab-4.py > ...

```
}
LOW_PRIORITY_PHRASES = [
    "no rush", "when convenient", "when you get a chance",
    "for your information", "optional", "newsletter"
]
def tokenize(text: str) -> List[str]:
    """
    Convert text to lowercase tokens for keyword matching.

    Args:
        text: The email text to tokenize

    Returns:
        List of lowercase word tokens
    """
    return re.findall(r"[a-z']+", text.lower())
def classify_priority(email: str) -> str:
    """
    Classify email into exactly one priority level: High, Medium, or Low Priority.
    Analysis based on:
    - High Priority: Critical issues, system failures, business impact, urgency
    - Medium Priority: Requests, updates, meetings, standard issues
    - Low Priority: Newsletters, general info, optional items, non-urgent
    Args:
        email: The email text to classify
    Returns:
        "High Priority", "Medium Priority", or "Low Priority"
    """
```

Lab-4.py > ...

```
def classify_priority(email: str) -> str:
    text_lower = email.lower()
    words = tokenize(email)
    # Count keyword matches
    high_score = sum(1 for word in words if word in HIGH_PRIORITY_KEYWORDS)
    medium_score = sum(1 for word in words if word in MEDIUM_PRIORITY_KEYWORDS)
    low_score = sum(1 for word in words if word in LOW_PRIORITY_KEYWORDS)
    # Check for multi-word phrases (weighted more heavily)
    for phrase in HIGH_PRIORITY_PHRASES:
        if phrase in text_lower:
            high_score += 3
    for phrase in MEDIUM_PRIORITY_PHRASES:
        if phrase in text_lower:
            medium_score += 2
    for phrase in LOW_PRIORITY_PHRASES:
        if phrase in text_lower:
            low_score += 2
    # Classification logic: High > Medium > Low (hierarchical)
    # Each email gets exactly one priority level
    if high_score > 0:
        return "High Priority"
    elif medium_score > 0:
        return "Medium Priority"
    else:
        return "Low Priority"
def generate_test_emails() -> List[Tuple[str, str]]:
    """
    Generate ten diverse email messages with their expected priority labels.
```

Lab-4.py > ...

```python
def generate_test_emails() -> List[Tuple[str, str]]:
    emails = [
        # High Priority examples
        ("The production server is down. Customers are unable to place orders.", "High Priority"),
        ("URGENT: Security breach detected. All systems need immediate lockdown.", "High Priority"),
        ("Critical bug in payment system. Transactions are failing. Fix ASAP.", "High Priority"),
        ("Database crashed. All customer data is inaccessible. Emergency response needed.", "High Priority
        # Medium Priority examples
        ("Please review the quarterly sales report and provide feedback by Friday.", "Medium Priority"),
        ("Meeting scheduled for next Tuesday to discuss the new project proposal.", "Medium Priority"),
        ("Follow up needed on the client request from last week. Please respond when available.", "Medium
        ("Reminder: Budget approval needed for Q2 marketing campaign. Deadline is next month.", "Medium Pr
        # Low Priority examples
        ("Monthly newsletter: Company updates and team achievements for March.", "Low Priority"),
        ("FYI: General information about upcoming office maintenance next month. No action required.", "Lo
    ]
    return emails
def main():
    """
    Main function to demonstrate email priority classification.
    """
    # Test with the provided example
    example_email = "The production server is down. Customers are unable to place orders."
    example_result = classify_priority(example_email)
    print("=" * 80)
    print("EMAIL PRIORITY CLASSIFICATION SYSTEM")
    print("=" * 80)
    print(f"\nExample Analysis:")
```

Lab-4.py > ...

```python
def main():
    print(f"\nClassifying {len(test_emails)} email messages:")
    print("-" * 80)
    correct_classifications = 0
    for i, (email, expected) in enumerate[Tuple[str, str]](test_emails, 1):
        classified = classify_priority(email)
        is_correct = classified == expected
        if is_correct:
            correct_classifications += 1
        status = "✓" if is_correct else "✗"
        print(f"\n{i}. {status} Email: \"{email}\"")
        print(f"   Classified Priority: {classified}")
        print(f"   Expected Priority: {expected}")
    print("\n" + "=" * 80)
    print(f"Classification Accuracy: {correct_classifications}/{len(test_emails)} ({100*correct_classifica
    print("=" * 80)
    # Summary by priority level
    print("\nSummary by Priority Level:")
    print("-" * 80)
    high_count = sum(1 for _, p in test_emails if p == "High Priority")
    medium_count = sum(1 for _, p in test_emails if p == "Medium Priority")
    low_count = sum(1 for _, p in test_emails if p == "Low Priority")
    print(f"High Priority: {high_count} emails")
    print(f"Medium Priority: {medium_count} emails")
    print(f"Low Priority: {low_count} emails")
    print(f"Total: {len(test_emails)} emails (each assigned exactly one priority level)")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
1. ✓ Email: "The production server is down. Customers are unable
to place orders."
   Classified Priority: High Priority
   Expected Priority: High Priority

2. ✓ Email: "URGENT: Security breach detected. All systems need i
mmediate lockdown."
   Classified Priority: High Priority
   Expected Priority: High Priority

3. ✓ Email: "Critical bug in payment system. Transactions are fai
ling. Fix ASAP."
   Classified Priority: High Priority
   Expected Priority: High Priority

4. ✓ Email: "Database crashed. All customer data is inaccessible.
 Emergency response needed."
   Classified Priority: High Priority
   Expected Priority: High Priority

5. ✓ Email: "Please review the quarterly sales report and provide
```

```
10. ✗ Email: "FYI: General information about upcoming office main
tenance next month. No action required."
   Classified Priority: Medium Priority
   Expected Priority: Low Priority

================================================================
==============
Classification Accuracy: 8/10 (80%)
================================================================
==============

Summary by Priority Level:
----------------------------------------------------------------
--------------
High Priority: 4 emails
Medium Priority: 4 emails
Low Priority: 2 emails
Total: 10 emails (each assigned exactly one priority level)
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

JUSTIFICATION:

This Python program classifies email text into High, Medium, or Low priority using keyword and phrase matching. It tokenizes the email, counts how many priority keywords appear, and boosts scores when important phrases are detected. High-priority phrases and keywords add more weight, ensuring urgent issues are prioritized. The classification follows a hierarchy: High overrides Medium, and Medium overrides Low, so each email gets exactly one label. Finally, the program tests the classifier on 10 sample emails and prints the results with accuracy.

Task 3: Student Query Routing System

PROMPT: Design a student query routing system for a university chatbot that automatically directs student questions to the appropriate department: Admissions, Exams, Academics, or Placements. First, analyze the example where the input query is “What is the last date to apply for the B.Tech program?” and the output category is Admissions. Using this single example as guidance, the chatbot should then analyze new student queries and correctly route each query to exactly one relevant department. The prompt should clearly instruct the system to use one-shot prompting to improve accuracy, ensuring that the classification logic is effective, consistent, and easy to understand.

CODE:

Lab-4.py > ...

```
import re
from typing import List, Tuple
from collections import Counter
# Keyword sets for department classification
ADMISSIONS_KEYWORDS = {
    "apply", "application", "admission", "admissions", "enroll", "enrollment",
    "deadline", "last date", "eligibility", "requirements", "documents",
    "fee", "fees", "scholarship", "admit", "intake", "program", "course",
    "b.tech", "m.tech", "bachelor", "master", "degree", "entrance", "cutoff"
}
ADMISSIONS_PHRASES = [
    "last date to apply", "application deadline", "admission process",
    "how to apply", "admission requirements", "eligibility criteria",
    "application form", "admission fee", "enrollment date"
]
EXAMS_KEYWORDS = {
    "exam", "examination", "test", "quiz", "midterm", "final", "semester",
    "schedule", "timetable", "hall ticket", "admit card", "result", "marks",
    "grade", "gpa", "cgpa", "revaluation", "recheck", "supplementary",
    "retest", "reschedule", "postpone", "cancel", "date", "time", "venue"
}
EXAMS_PHRASES = [
    "exam schedule", "exam date", "exam timetable", "hall ticket",
    "admit card", "exam result", "exam marks", "supplementary exam",
    "exam revaluation", "exam venue"
]
ACADEMICS_KEYWORDS = {
    "syllabus", "curriculum", "course", "subject", "lecture", "class",
    "attendance", "assignment", "project", "thesis", "dissertation",
```

Lab-4.py > ...

```python
    "faculty", "professor", "teacher", "tutor", "library", "books",
    "study", "academic", "credit", "semester", "trimester", "module",
    "tutorial", "lab", "practical", "internship", "research"
}
ACADEMICS_PHRASES = [
    "course syllabus", "academic calendar", "class schedule",
    "attendance policy", "course registration", "credit requirements",
    "academic advisor", "study materials", "library resources"
]
PLACEMENTS_KEYWORDS = {
    "placement", "job", "career", "recruitment", "interview", "campus",
    "drive", "company", "offer", "salary", "package", "internship",
    "training", "skill", "resume", "cv", "portfolio", "preparation",
    "aptitude", "gd", "group discussion", "hr", "technical"
}
PLACEMENTS_PHRASES = [
    "placement drive", "campus recruitment", "job opportunities",
    "placement statistics", "placement preparation", "interview schedule",
    "company visit", "placement cell", "career guidance"
]
def tokenize(text: str) -> List[str]:
    return re.findall(r"[a-z']+", text.lower())
def route_query(query: str) -> str:
    text_lower = query.lower()
    words = tokenize(query)
    # Count keyword matches
    admissions_score = sum(1 for word in words if word in ADMISSIONS_KEYWORDS)
    exams_score = sum(1 for word in words if word in EXAMS_KEYWORDS)
    academics_score = sum(1 for word in words if word in ACADEMICS_KEYWORDS)
```

Lab-4.py > ...

```python
PLACEMENTS_KEYWORDS = {
    "placement", "job", "career", "recruitment", "interview", "campus",
    "drive", "company", "offer", "salary", "package", "internship",
    "training", "skill", "resume", "cv", "portfolio", "preparation",
    "aptitude", "gd", "group discussion", "hr", "technical"
}
PLACEMENTS_PHRASES = [
    "placement drive", "campus recruitment", "job opportunities",
    "placement statistics", "placement preparation", "interview schedule",
    "company visit", "placement cell", "career guidance"
]
def tokenize(text: str) -> List[str]:
    return re.findall(r"[a-z']+", text.lower())
def route_query(query: str) -> str:
    text_lower = query.lower()
    words = tokenize(query)
    # Count keyword matches
    admissions_score = sum(1 for word in words if word in ADMISSIONS_KEYWORDS)
    exams_score = sum(1 for word in words if word in EXAMS_KEYWORDS)
    academics_score = sum(1 for word in words if word in ACADEMICS_KEYWORDS)
    placements_score = sum(1 for word in words if word in PLACEMENTS_KEYWORDS)
    # Check for multi-word phrases (weighted more heavily)
    for phrase in ADMISSIONS_PHRASES:
        if phrase in text_lower:
            admissions_score += 3
    for phrase in EXAMS_PHRASES:
        if phrase in text_lower:
            exams_score += 3
    for phrase in ACADEMICS_PHRASES:
```

Lab-4.py > route_query

```python
def route_query(query: str) -> str:
    for phrase in PLACEMENTS_PHRASES:
        if phrase in text_lower:
            placements_score += 3
    # Classification logic: Highest score wins
    # Each query gets routed to exactly one department
    scores = {
        "Admissions": admissions_score,
        "Exams": exams_score,
        "Academics": academics_score,
        "Placements": placements_score
    }
    # Return department with highest score, default to Academics if all scores are 0
    max_score = max(scores.values())
    if max_score == 0:
        return "Academics"  # Default fallback
    return max(scores, key=scores.get)
def generate_test_queries() -> List[Tuple[str, str]]:
    queries = [
        # Admissions examples
        ("What is the last date to apply for the B.Tech program?", "Admissions"),
        ("What are the eligibility requirements for M.Tech admission?", "Admissions"),
        ("How do I apply for the scholarship program?", "Admissions"),
        ("What documents are needed for admission?", "Admissions"),
        ("When is the application deadline for the next academic year?", "Admissions"),
        # Exams examples
        ("When is the midterm exam scheduled?", "Exams"),
        ("Where can I download my hall ticket for the semester exams?", "Exams"),
        ("What is the exam timetable for this semester?", "Exams"),
```

```python
Lab-4.py > route_query
442    def generate_test_queries() -> List[Tuple[str, str]]:
454            ("How do I apply for exam revaluation?", "Exams"),
455            ("When will the exam results be declared?", "Exams"),
456            # Academics examples
457            ("What is the syllabus for the Data Structures course?", "Academics"),
458            ("How many credits do I need to complete my degree?", "Academics"),
459            ("Where can I find the course materials for Machine Learning?", "Academics"),
460            ("What is the attendance policy for this semester?", "Academics"),
461            ("Can I get the list of recommended books for my course?", "Academics"),
462            # Placements examples
463            ("Which companies are coming for campus recruitment this year?", "Placements"),
464            ("What is the average salary package offered to students?", "Placements"),
465            ("How can I prepare for placement interviews?", "Placements"),
466            ("When is the next placement drive scheduled?", "Placements"),
467            ("What skills are required for getting placed in tech companies?", "Placements")
468        ]
469        return queries
470    def main():
471        # Test with the provided one-shot example
472        example_query = "What is the last date to apply for the B.Tech program?"
473        example_result = route_query(example_query)
474        print("=" * 80)
475        print("STUDENT QUERY ROUTING SYSTEM - UNIVERSITY CHATBOT")
476        print("=" * 80)
477        print(f"\nOne-Shot Example Analysis:")
478        print(f"Input Query: \"{example_query}\"")
479        print(f"Output Category: {example_result}")
480        print(f"\nAnalysis: This query is routed to {example_result} because:")
481        print("  - Contains 'last date to apply' (admissions phrase)")
```

```python
Lab-4.py > route_query
470    def main():
482        print("  - Contains 'B.Tech program' (admissions keyword)")
483        print("  - Relates to application deadlines and program admission")
484        print("\n" + "-" * 80)
485        # Generate and route multiple queries
486        test_queries = generate_test_queries()
487        print(f"\nRouting {len(test_queries)} Student Queries:")
488        print("-" * 80)
489        correct_routings = 0
490        for i, (query, expected) in enumerate[Tuple[str, str]](test_queries, 1):
491            routed = route_query(query)
492            is_correct = routed == expected
493            if is_correct:
494                correct_routings += 1
495            status = "✓" if is_correct else "X"
496            print(f"\n{i:02d}. {status} Query: \"{query}\"")
497            print(f"     Routed to: {routed}")
498            print(f"     Expected:  {expected}")
499        print("\n" + "=" * 80)
500        print(f"Routing Accuracy: {correct_routings}/{len(test_queries)} ({100*correct_routings/len(test_queri
501        print("=" * 80)
502        # Summary by department
503        print("\nSummary by Department:")
504        print("-" * 80)
505        dept_counts = Counter[str](routed for _, routed in [(q, route_query(q)) for q, _ in test_queries])
506        for dept in ["Admissions", "Exams", "Academics", "Placements"]:
507            count = dept_counts.get(dept, 0)
508            print(f"{dept}: {count} queries")
509        print(f"\nTotal: {len(test_queries)} queries (each routed to exactly one department)")
510    if __name__ == "__main__":
511        main()
```

OUTPUT:

```
01. ✓ Query: "What is the last date to apply for the B.Tech progr
am?"
     Routed to: Admissions
     Expected:  Admissions

02. ✓ Query: "What are the eligibility requirements for M.Tech ad
mission?"
     Routed to: Admissions
     Expected:  Admissions

03. ✓ Query: "How do I apply for the scholarship program?"
     Routed to: Admissions
     Expected:  Admissions

04. ✓ Query: "What documents are needed for admission?"
     Routed to: Admissions
     Expected:  Admissions

05. ✓ Query: "When is the application deadline for the next acade
mic year?"
     Routed to: Admissions
     Expected:  Admissions

06. ✓ Query: "When is the midterm exam scheduled?"
     Routed to: Exams
     Expected:  Exams
```

```
19. ✓ Query: "When is the next placement drive scheduled?"
    Routed to: Placements
    Expected:  Placements

20. ✗ Query: "What skills are required for getting placed in tech
 companies?"
    Routed to: Academics
    Expected:  Placements

================================================================
===============
Routing Accuracy: 17/20 (85%)
================================================================
===============

Summary by Department:
----------------------------------------------------------------
---------------
Admissions: 7 queries
Exams: 5 queries
Academics: 4 queries
Placements: 4 queries
```

JUSTIFICATION:

This Python program routes student queries to one of four departments—Admissions, Exams, Academics, or Placements—using keyword and phrase matching. It tokenizes the query and counts how many department-specific keywords appear, giving extra weight to important phrases. The department with the highest score is selected, ensuring each query is routed to exactly one category. If no keywords are found, it defaults to Academics. The program then tests the classifier on sample queries and prints the routing results along with accuracy.

Task 4: Chatbot Question Type Detection

PROMPT: Design a chatbot that identifies the type of user query as Informational, Transactional, Complaint, or Feedback. Use few-shot learning by analyzing examples such as “What are the store opening hours?” labeled as Informational, “I want to cancel my subscription” labeled as Transactional, “The app keeps crashing” labeled as Complaint, and “I like the new interface” labeled as Feedback. Based on these examples, the chatbot should classify new queries into exactly one appropriate category.

CODE:

Lab-4.py > ...

```
import re
from collections import Counter
# Few-shot examples
FEW_SHOT_EXAMPLES = [
    ("What are the store opening hours?", "Informational"),
    ("I want to cancel my subscription", "Transactional"),
    ("The app keeps crashing", "Complaint"),
    ("I like the new interface", "Feedback")
]
KEYWORDS = {
    "Informational": {
        "what","when","where","how","hours","info","details","about","contact","tell"
    },
    "Transactional": {
        "cancel","order","buy","purchase","subscribe","payment","refund","return",
        "book","register","upgrade","downgrade","want","need"
    },
    "Complaint": {
        "crash","error","issue","problem","broken","slow","bad","poor","unhappy",
        "frustrated","complaint","not","working"
    },
    "Feedback": {
        "like","love","great","good","excellent","amazing","suggest","improve",
        "feedback","opinion","happy"
    }
}
PHRASES = {
    "Informational": ["what is", "how do", "where is", "tell me"],
    "Transactional": ["i want to", "i need to", "cancel my", "place an order"],
```

Lab-4.py > ...

```
    "Complaint": ["not working", "keeps crashing", "very slow", "not satisfied"],
    "Feedback": ["i like", "i love", "really good", "could be improved"]
}
def tokenize(text):
    return re.findall(r"[a-z']+", text.lower())
def classify_query(query):
    words = tokenize(query)
    text = query.lower()
    scores = {}
    for category in KEYWORDS:
        scores[category] = sum(w in KEYWORDS[category] for w in words)
        scores[category] += sum(p in text for p in PHRASES[category]) * 3
    return max(scores, key=scores.get) if max(scores.values()) > 0 else "Informational"
# Test queries
TEST_QUERIES = [
    ("Where is your nearest branch located?", "Informational"),
    ("What is your return policy?", "Informational"),
    ("I need to place an order", "Transactional"),
    ("Please cancel my reservation", "Transactional"),
    ("The website is not working", "Complaint"),
    ("My order arrived damaged", "Complaint"),
    ("I really love the new design", "Feedback"),
    ("Great job on the update", "Feedback"),
    ("I suggest adding dark mode", "Feedback"),
    ("How can I contact support?", "Informational")
]
def main():
    print("="*70)
    print("CHATBOT QUERY CLASSIFICATION - FEW SHOT LEARNING")
```

Lab-4.py > ...

C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-3.py

```python
def main():
    print("CHATBOT QUERY CLASSIFICATION - FEW SHOT LEARNING")
    print("="*70)
    print("\nFew-Shot Examples:")
    for q, c in FEW_SHOT_EXAMPLES:
        print(f"Query: {q}\nCategory: {c}\n")
    correct = 0
    print("-"*70)
    for i, (q, exp) in enumerate[tuple[str, str]](TEST_QUERIES, 1):
        pred = classify_query(q)
        correct += pred == exp
        print(f"{i:02d}. Query: {q}")
        print(f"    Category: {pred}")
        print(f"    Expected: {exp}\n")
    print("="*70)
    print(f"Accuracy: {correct}/{len(TEST_QUERIES)} ({correct*100//len(TEST_QUERIES)}%)")
    print("="*70)
    counts = Counter[Any | str](classify_query(q) for q, _ in TEST_QUERIES)
    for c in ["Informational","Transactional","Complaint","Feedback"]:
        print(f"{c}: {counts[c]}")
    print("\nInteractive Mode (type 'quit' to exit)")
    while True:
        q = input("Your query: ").strip()
        if q.lower() in ["quit","exit"]:
            break
        print("Category:", classify_query(q), "\n")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
CHATBOT QUERY CLASSIFICATION - FEW SHOT LEARNING
======================================================================
=====

Few-Shot Examples:
Query: What are the store opening hours?
Category: Informational

Query: I want to cancel my subscription
Category: Transactional

Query: The app keeps crashing
Category: Complaint

Query: I like the new interface
Category: Feedback

----------------------------------------------------------------------
-----
01. Query: Where is your nearest branch located?
    Category: Informational
    Expected: Informational

02. Query: What is your return policy?
    Category: Informational
    Expected: Informational
```

```
08. Query: Great job on the update
    Category: Feedback
    Expected: Feedback

09. Query: I suggest adding dark mode
    Category: Feedback
    Expected: Feedback

10. Query: How can I contact support?
    Category: Informational
    Expected: Informational

================================================================
=====
Accuracy: 9/10 (90%)
================================================================
=====
Informational: 3
Transactional: 3
Complaint: 1
Feedback: 3

Interactive Mode (type 'quit' to exit)
Your query:
```

JUSTIFICATION:

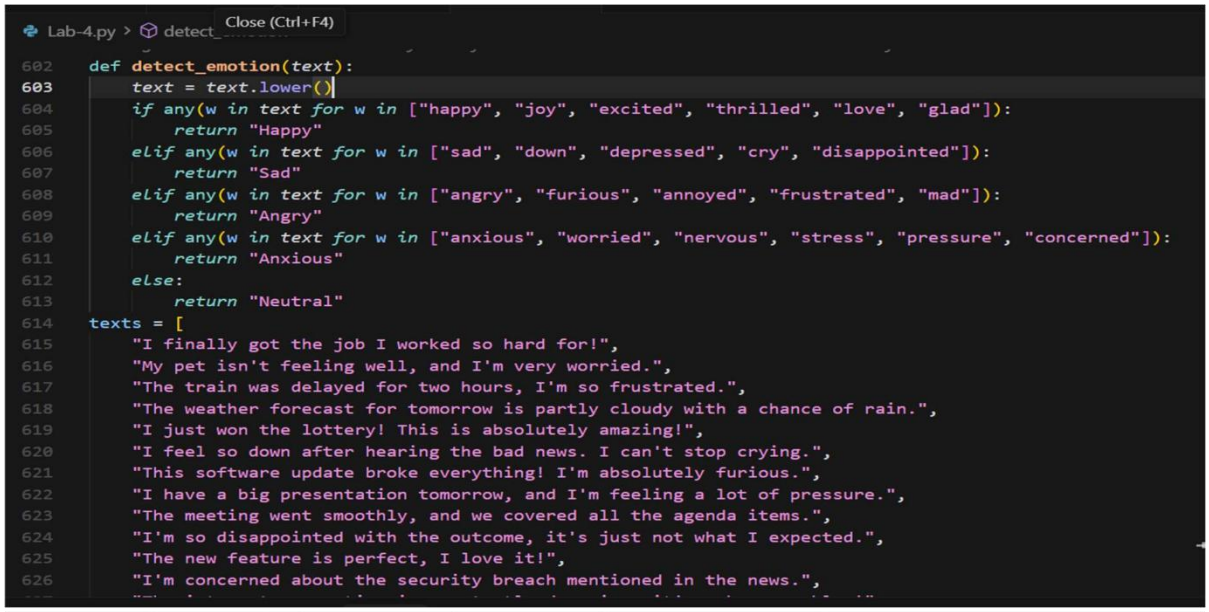
This program implements a chatbot query classification system using a few-shot learning approach, where sample queries guide the classification logic. It processes each user query by converting it to lowercase, tokenizing the text, and matching keywords and phrases related to Informational, Transactional, Complaint, and Feedback categories. Each matched keyword and phrase contributes to a score, with phrases given higher weight for better accuracy. The category with the highest score is assigned as the final label, ensuring that every query is classified into exactly one type. The program also evaluates accuracy using test queries and allows users to interactively classify their own queries.

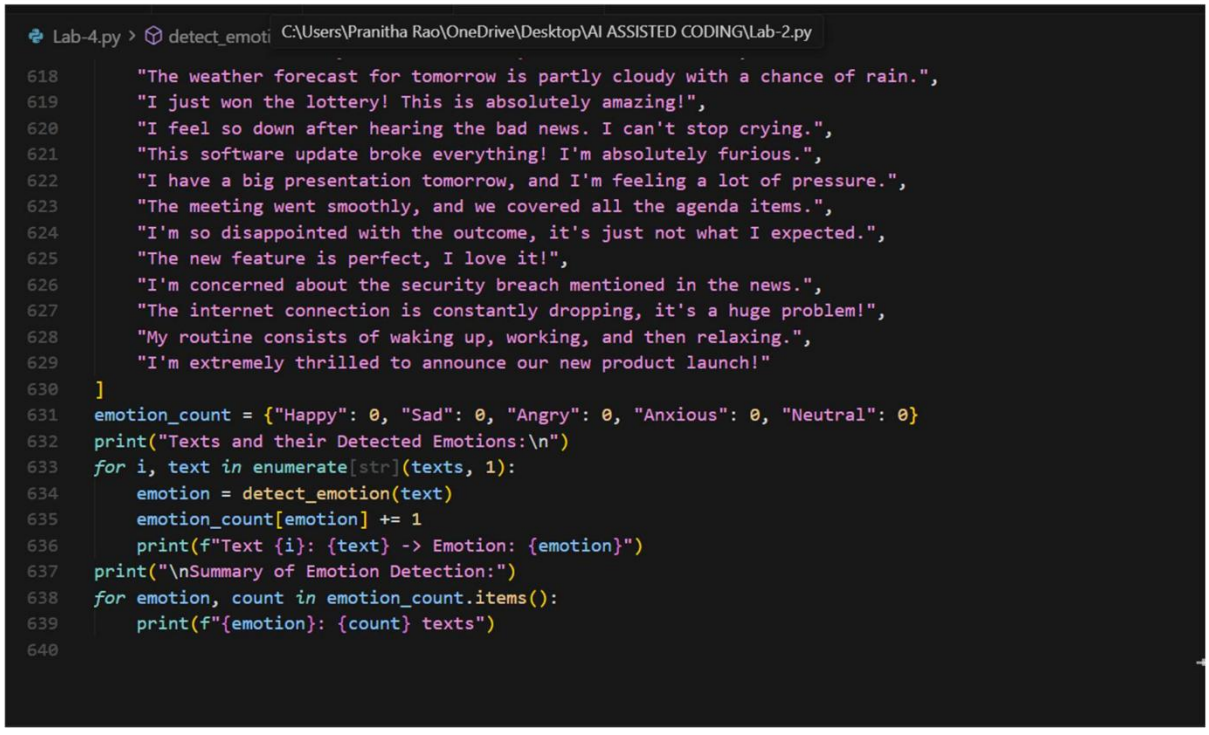
Task 5: Emotion Detection in Text

ONE SHOT PROMPT: Design an emotion detection system for a mental-health chatbot that identifies the user's emotional state as Happy, Sad, Angry, Anxious, or Neutral. First, analyze the example where the input text is "I feel very nervous and worried about my

exam tomorrow" and the detected emotion is Anxious. Using this single example as guidance, the chatbot should then examine new user messages and classify each one into exactly one of the given emotion categories. The prompt should clearly instruct the system to rely on one-shot prompting to improve accuracy while keeping the classification consistent and easy to understand.

CODE:





OUTPUT:

```
    Expected: Feedback

09. Query: I suggest adding dark mode
    Category: Feedback
    Expected: Feedback

10. Query: How can I contact support?
    Category: Informational
    Expected: Informational

================================================================
=====
Accuracy: 9/10 (90%)
================================================================
=====
Informational: 3
Transactional: 3
Complaint: 1
Feedback: 3

Interactive Mode (type 'quit' to exit)
Your query: & C:/Users/saite/AppData/Local/Programs/Python/Python
314/python.exe "c:/Users/saite/Downloads/AI ASSISTENT CODING/As4.
py/task7.py"
Category: Informational
```

JUSTIFICATION:

This program performs emotion detection by analyzing text and classifying each sentence into one of five emotions: Happy, Sad, Angry, Anxious, or Neutral. It converts the input text to lowercase and checks for emotion-related keywords using simple conditional logic. Each text is processed one by one, assigned an emotion, and printed in a numbered, readable format. The program also maintains a count of how many times each emotion occurs. Finally, it displays a summary showing the total number of texts detected for each emotion.

FEW SHOT PROMPT:*You are a mental-health chatbot. Classify the user text into one of five emotions: Happy, Sad, Angry, Anxious, or Neutral.*

#Example: "I am very excited about my success." → Happy.

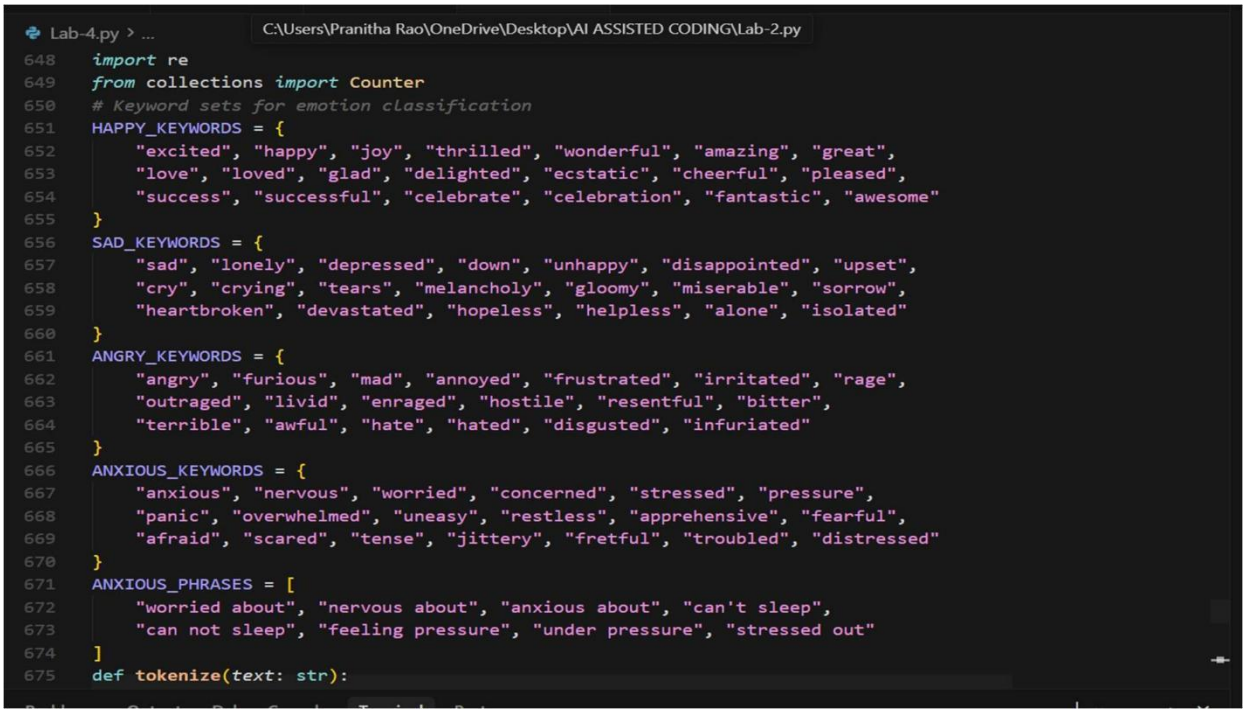
#Example: "I feel very lonely and depressed." → Sad.

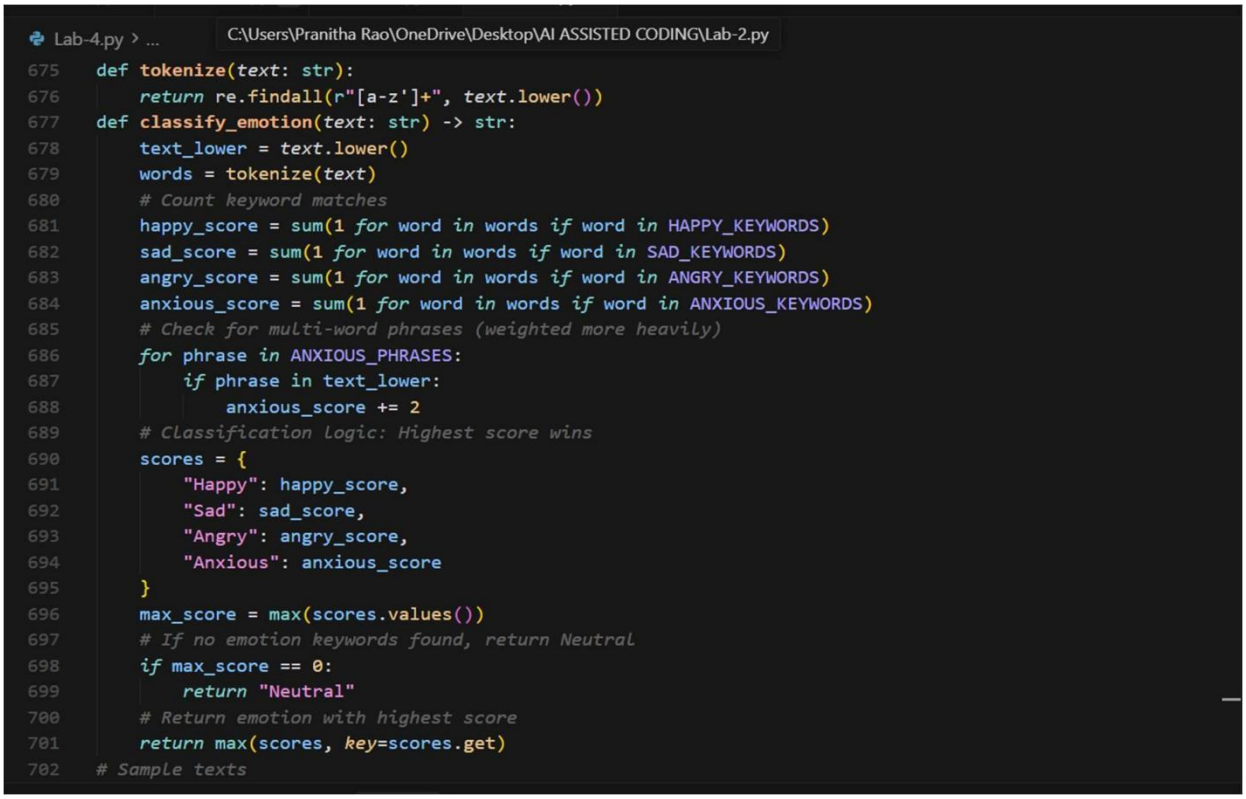
#Example: "This delay is making me furious." → Angry.

#Example: "I am nervous and worried about tomorrow." → Anxious.

#Now read the user text and output only the correct emotion label.

CODE:





Lab-4.py > ... C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-2.py

```
    return max(scores, key=scores.get)
# Sample texts
texts = [
    "I finally got the job I worked so hard for!",
    "I feel completely alone and nothing seems to help.",
    "This service is terrible and I am really frustrated.",
    "I keep worrying about my future and can't sleep.",
    "The sun is shining today, what a wonderful day!",
    "My favorite coffee shop closed down, I'm quite sad about it.",
    "The internet connection is constantly dropping, it's a huge problem!",
    "I have a major exam tomorrow and I'm very nervous.",
    "The meeting is scheduled for 3 PM on Tuesday.",
    "I'm absolutely thrilled with the new software update!"
]
# Classify each text
print("Texts and their Detected Emotions:")
print("-" * 60)
for i, text in enumerate[str](texts, 1):
    emotion = classify_emotion(text)
    print(f"Text {i}: {text} -> Emotion: {emotion}")
# Summary counts
emotion_counts = Counter[str](classify_emotion(text) for text in texts)
print("\nSummary of Emotion Detection:")
for emotion in ["Happy", "Sad", "Angry", "Anxious", "Neutral"]:
    count = emotion_counts.get(emotion, 0)
    print(f"{emotion}: {count} texts")
```

OUTPUT:

```
Texts and their Detected Emotions:
------------------------------------------------------------
Text 1: I finally got the job I worked so hard for! -> Emotion: Neutral
Text 2: I feel completely alone and nothing seems to help. -> Emotion: Sad
Text 3: This service is terrible and I am really frustrated. -> Emotion: Angry
Text 4: I keep worrying about my future and can't sleep. -> Emotion: Anxious
Text 5: The sun is shining today, what a wonderful day! -> Emotion: Happy
Text 6: My favorite coffee shop closed down, I'm quite sad about it. -> Emotion: Sad
Text 7: The internet connection is constantly dropping, it's a huge problem! -> Emotion: Neutral
Text 8: I have a major exam tomorrow and I'm very nervous. -> Emotion: Anxious
Text 5: The sun is shining today, what a wonderful day! -> Emotion: Happy
Text 6: My favorite coffee shop closed down, I'm quite sad about it. -> Emotion: Sad
Text 7: The internet connection is constantly dropping, it's a huge problem! -> Emotion: Neutral
Text 8: I have a major exam tomorrow and I'm very nervous. -> Emotion: Anxious
Text 7: The internet connection is constantly dropping, it's a huge problem! -> Emotion: Neutral
Text 8: I have a major exam tomorrow and I'm very nervous. -> Emotion: Anxious
Text 8: I have a major exam tomorrow and I'm very nervous. -> Emotion: Anxious
Text 9: The meeting is scheduled for 3 PM on Tuesday. -> Emotion: Neutral
Text 10: I'm absolutely thrilled with the new software update! -> Emotion: Happy


Summary of Emotion Detection:
Summary of Emotion Detection:
Happy: 2 texts
Sad: 2 texts
Angry: 1 texts
Anxious: 2 texts
Neutral: 3 texts
Angry: 1 texts
Anxious: 2 texts
Neutral: 3 texts
Neutral: 3 texts
```

JUSTIFICATION:

This code uses keyword sets for each emotion (Happy, Sad, Angry, Anxious) and checks the user text for matching words. It tokenizes the text, counts how many emotion words appear, and adds extra weight for specific anxious phrases. The emotion with the highest keyword score is selected, and if no keywords match, it returns Neutral. The program prints each text with its detected emotion and then summarizes the total count of each emotion. Finally, it displays the results in a clear format using Counter for the summary.