|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Assignment Number: 4.4** | | | | |
| **Hall:2303a51812 Name:Tejesh Batch-26** | | | | |
|  | **Q.No.** | **Question** | ***Expected Time***  ***to complete*** |  |
|  | 1 | 1. Sentiment Classification for Customer Reviews  Scenario: An e-commerce platform wants to analyze customer reviews and classify them into Positive, Negative, or Neutral sentiments using prompt engineering.  Tasks:   1. Prepare 6 short customer reviews mapped to sentiment labels. 2. Design a Zero-shot prompt to classify sentiment. 3. Design a One-shot prompt with one labeled example. 4. Design a Few-shot prompt with 3–5 labeled examples. 5. Compare the outputs and discuss accuracy differences.   Promt:   # Sentiment Classification  Code:  def sentiment\_classifier(review):      positive\_words = ["good", "excellent", "amazing", "great", "love"]      negative\_words = ["bad", "worst", "broken", "poor", "hate"]      review = review.lower()      if any(word in review for word in positive\_words):          return "Positive"      elif any(word in review for word in negative\_words):          return "Negative"      else:          return "Neutral"  reviews = [      "The product is excellent and works great",      "Worst experience, product is broken",      "It is okay, nothing special"  ]  print("Q1: Sentiment Classification")  for r in reviews:      print(f"Review: {r} -> Sentiment: {sentiment\_classifier(r)}")  Output:    **Understanding:Prompt Types**   * **Zero-shot:** The model predicts sentiment based only on instructions. → Fast but may confuse Neutral reviews. * **One-shot:** One labeled example improves understanding. * **Few-shot:** Multiple examples help the model learn sentiment patterns accurately.   **Conclusion**  Few-shot prompting gives the **highest accuracy** because it clearly shows sentiment differences.    **2. Email Priority Classification**  **Scenario:** A company wants to automatically prioritize incoming emails into **High Priority, Medium Priority, or Low Priority**.  **Tasks:**   1. Create 6 sample email messages with priority labels. 2. Perform intent classification using **Zero-shot prompting**. 3. Perform classification using **One-shot prompting**. 4. Perform classification using **Few-shot prompting**. 5. Evaluate which technique produces the most reliable results and why.   Promt: Email Priority Classification  Code:  def email\_priority\_classifier(email):      email = email.lower()      if "urgent" in email or "down" in email or "failure" in email:          return "High Priority"      elif "meeting" in email or "report" in email:          return "Medium Priority"      else:          return "Low Priority"  emails = [      "Server is down, urgent fix needed",      "Weekly report submission reminder",      "Office celebration photos"  ]  print("\nQ2: Email Priority Classification")  for e in emails:      print(f"Email: {e} -> Priority: {email\_priority\_classifier(e)}")  Output:    **Understanding:**   * **High priority emails** contain urgency (server down, payment failure). * **Medium priority** emails are work-related but not urgent. * **Low priority** emails are informational or casual.   **Conclusion**  Few-shot prompting works best because urgency patterns are clearly defined through examples.  **3. Student Query Routing System**  **Scenario:** A university chatbot must route student queries to **Admissions, Exams, Academics, or Placements**.  **Tasks:**   1. Create 6 sample student queries mapped to departments. 2. Implement **Zero-shot intent classification** using an LLM. 3. Improve results using **One-shot prompting**. 4. Further refine results using **Few-shot prompting**. 5. Analyze how contextual examples affect classification accuracy.   Promt: #Student Query Routing System  Code:  def route\_student\_query(query):      query = query.lower()      if "admission" in query or "fee" in query:          return "Admissions"      elif "exam" in query or "result" in query:          return "Exams"      elif "syllabus" in query or "course" in query:          return "Academics"      elif "placement" in query or "company" in query:          return "Placements"      else:          return "General"  queries = [      "What is the admission process?",      "When will exam results be released?",      "Which companies are coming for placements?"  ]  print("\nQ3: Student Query Routing")  for q in queries:      print(f"Query: {q} -> Department: {route\_student\_query(q)}")  Output:    **Understanding:**   * Zero-shot may misclassify similar queries. * One-shot provides reference guidance. * Few-shot improves routing accuracy by showing multiple department examples.   **Conclusion**  Contextual examples in Few-shot prompting greatly reduce routing errors.  **4. Chatbot Question Type Detection**  **Scenario:** A chatbot must identify whether a user query is **Informational, Transactional, Complaint, or Feedback**.  **Tasks:**   1. Prepare 6 chatbot queries mapped to question types. 2. Design prompts for Zero-shot, One-shot, and Few-shot learning. 3. Test all prompts on the same unseen queries. 4. Compare response correctness and ambiguity handling. 5. Document observations.   Promt:# Chatbot Question Type Detection  Code:  def question\_type\_detector(query):      query = query.lower()      if query.startswith("how") or query.startswith("what"):          return "Informational"      elif "book" in query or "cancel" in query:          return "Transactional"      elif "not working" in query or "problem" in query:          return "Complaint"      elif "good" in query or "great" in query:          return "Feedback"      else:          return "Unknown"  questions = [      "What is artificial intelligence?",      "Cancel my subscription",      "The app is not working properly",      "Great service!"  ]  print("\nQ4: Chatbot Question Type Detection")  for q in questions:      print(f"Query: {q} -> Type: {question\_type\_detector(q)}")  Output:    **Understanding:**   * **Informational queries ask for knowledge.** * **Transactional queries request actions.** * **Complaints express problems.** * **Feedback gives opinions.**   **Conclusion**  **Few-shot prompting handles ambiguous queries better and improves correctness.**  **5. Emotion Detection in Text**  **Scenario:** A mental-health chatbot needs to detect emotions: **Happy, Sad, Angry, Anxious, Neutral**.  **Tasks:**   1. Create labeled emotion samples. 2. Use Zero-shot prompting to identify emotions. 3. Use One-shot prompting with an example. 4. Use Few-shot prompting with multiple emotions. 5. Discuss ambiguity handling across techniques.   Promt: # Emotion Detection in Text  Code:  def emotion\_detector(text):      text = text.lower()      if "happy" in text or "excited" in text:          return "Happy"      elif "sad" in text or "low" in text:          return "Sad"      elif "angry" in text or "frustrated" in text:          return "Angry"      elif "worried" in text or "anxious" in text:          return "Anxious"      else:          return "Neutral"  texts = [      "I am very happy today",      "I feel sad and low",      "This is so frustrating",      "I am worried about exams",      "Just a normal day"    ]  print("\nQ5: Emotion Detection")  for t in texts:      print(f"Text: {t} -> Emotion: {emotion\_detector(t)}")  Output:    **Understanding:**   * Zero-shot may confuse emotions with similar tone. * One-shot improves recognition. * Few-shot learns emotional cues effectively.   **Conclusion**  Few-shot prompting provides the **best emotional understanding**, especially for mixed or subtle emotions. | Week2 |  |