CSAI 422: Laboratory Assignment 3

Due Date: March 10, 2025 at 23:59
Time Estimate: 3 hours
Points: 100

Programming Lab: Taming LLMs with Groq API

Overview

In this lab assignment, you will apply concepts from "Taming the Model" to build a content classification and analysis tool. You'll work with the Groq API (which has a similar interface to OpenAI) to implement techniques for controlling completions, analyzing model confidence, and comparing different prompt strategies.

Time Estimate: 3 hours

Learning Objectives

After completing this lab, you will be able to: - Create well-structured prompts that control model behavior - Implement techniques to manage model completions and extract precise answers - Use streaming responses to optimize performance - Apply logprob analysis to evaluate completion quality - Compare different model prompt strategies for classification tasks

Prerequisites

- Python programming experience
- Basic understanding of HTTP APIs
- Familiarity with prompt engineering concepts
- Groq API key (instructions will be provided by your instructor)

Assignment Description

You will build a content analysis tool that:

1. Takes user input text (such as product reviews, news articles, or emails)
2. Classifies the content using structured prompts
3. Extracts key insights using recognizable start/end patterns
4. Analyzes the model's confidence in its responses
5. Compares different prompt strategies

Setup Instructions

1. Create a new Python project directory
2. Install required libraries:

```
pip install groq python-dotenv
```

3. Create a `.env` file in your project directory and add your Groq API key:

```
GROQ_API_KEY=your_api_key_here
```

4. Create a Python file named `taming_llm.py` for your assignment code

Tasks and Grading Breakdown (100 points total)

Part 1: Configuration and Basic Completion (15 points)

- Set up the Groq API client with proper error handling (5 points)
- Implement a basic completion function that:
  - Takes a prompt and returns a completion (5 points)
  - Properly handles API errors and rate limits (5 points)

Part 2: Structured Completions (25 points)

- Create a prompt template for structured document completion (7 points)
- Implement a function that extracts specific sections from completions using recognizable start/end patterns (8 points)
- Implement streaming response handling that stops generation once a specified pattern is detected (10 points)

Part 3: Classification with Confidence Analysis (30 points)

- Implement a text classification feature that:
  - Takes a text input and classifies it into one of several categories (8 points)
  - Uses a structured prompt to ensure clear classification responses (7 points)
  - Requests and analyzes logprobs to determine the model's confidence (10 points)
  - Implements a threshold-based filtering system that only accepts classifications above a certain confidence level (5 points)

Part 4: Prompt Strategy Comparison (30 points)

- Implement at least three different prompt strategies for the same classification task:
  - Basic prompt with direct question (5 points)
  - Structured prompt with format instructions (5 points)
  - Few-shot prompt with examples (5 points)
- Create a test harness that:
  - Runs the same set of inputs through each strategy (5 points)
  - Compares accuracy, confidence, and response length (5 points)
  - Visualizes the comparison results (5 points)

Detailed Implementation Guidelines

Part 1: Configuration and Basic Completion

Create a client class to handle Groq API interactions:

```
import os
import time
from dotenv import load_dotenv
import groq

class LLMClient:
    def __init__(self):
        load_dotenv()
        self.api_key = os.getenv("GROQ_API_KEY")
        self.client = groq.Client(api_key=self.api_key)
        self.model = "llama3-70b-8192"  # or another Groq model

    def complete(self, prompt, max_tokens=1000, temperature=0.7):
        try:
            response = self.client.chat.completions.create(
                model=self.model,
                messages=[{"role": "user", "content": prompt}],
                max_tokens=max_tokens,
                temperature=temperature
            )
            return response.choices[0].message.content
        except Exception as e:
            # Implement proper error handling with retries
            print(f"Error: {e}")
            return None
```

Part 2: Structured Completions

Implement functions to create structured prompts and extract specific sections:

```
def create_structured_prompt(text, question):
    """
    Creates a structured prompt that will produce a completion with
    easily recognizable sections.
    """
    prompt = f"""
# Analysis Report

## Input Text
{text}

## Question
```

```
{question}

## Analysis
"""
    return prompt

def extract_section(completion, section_start, section_end=None):
    """
    Extracts content between section_start and section_end.
    If section_end is None, extracts until the end of the completion.
    """
    start_idx = completion.find(section_start)
    if start_idx == -1:
        return None

    start_idx += len(section_start)

    if section_end is None:
        return completion[start_idx:].strip()

    end_idx = completion.find(section_end, start_idx)
    if end_idx == -1:
        return completion[start_idx:].strip()

    return completion[start_idx:end_idx].strip()

def stream_until_marker(prompt, stop_marker, max_tokens=1000):
    """
    Streams the completion and stops once a marker is detected.
    Returns the accumulated text up to the marker.
    """
    # Implement this function with streaming API
    pass
```

Part 3: Classification with Confidence Analysis

Implement a classification function with confidence analysis:

```
def classify_with_confidence(text, categories, confidence_threshold=0.8):
    """
    Classifies text into one of the provided categories.
    Returns the classification only if confidence is above threshold.
    """
    # Create a prompt that encourages clear, unambiguous classification
    prompt = f"""
Classify the following text into exactly one of these categories: {', '.join(categories)}.
```

```
Response format:
1. CATEGORY: [one of: {', '.join(categories)}]
2. CONFIDENCE: [high|medium|low]
3. REASONING: [explanation]

Text to classify:
{text}
"""

    # Request completion with logprobs
    response = client.chat.completions.create(
        model=model,
        messages=[{"role": "user", "content": prompt}],
        max_tokens=500,
        temperature=0,
        logprobs=True,
        top_logprobs=5
    )

    # Extract classification
    completion = response.choices[0].message.content
    category = extract_section(completion, "1. CATEGORY: ", "\n")

    # Analyze logprobs to determine confidence
    # Implement this section to calculate confidence score

    # Return classification if confidence exceeds threshold
    if confidence_score > confidence_threshold:
        return {
            "category": category,
            "confidence": confidence_score,
            "reasoning": extract_section(completion, "3. REASONING: ")
        }
    else:
        return {
            "category": "uncertain",
            "confidence": confidence_score,
            "reasoning": "Confidence below threshold"
        }
```

Part 4: Prompt Strategy Comparison

Implement a comparison framework to test different prompt strategies:

```
def compare_prompt_strategies(texts, categories):
    """
```

```
    Compares different prompt strategies on the same classification tasks.
    """
    strategies = {
        "basic": lambda text: f"Classify this text into one of these categories: {', '.join(

        "structured": lambda text: f"""
Classification Task
Categories: {', '.join(categories)}
Text: {text}
Classification: """,

        "few_shot": lambda text: f"""
Here are some examples of text classification:

Example 1:
Text: "The product arrived damaged and customer service was unhelpful."
Classification: Negative

Example 2:
Text: "While delivery was slow, the quality exceeded my expectations."
Classification: Mixed

Example 3:
Text: "Absolutely love this! Best purchase I've made all year."
Classification: Positive

Now classify this text:
Text: "{text}"
Classification: """
    }

    results = {}

    for strategy_name, prompt_func in strategies.items():
        strategy_results = []

        for text in texts:
            prompt = prompt_func(text)
            # Implement timing, classification, and confidence measurement
            # Add results to strategy_results

        results[strategy_name] = strategy_results

    # Compare and visualize results
    return results
```

Deliverables

Submit the following:

1. Your complete Python code implementing all required functionality
2. A brief report (PDF) that includes:
   - A description of your implementation approach
   - Screenshots of your tool in action
   - Analysis of how different prompt strategies performed
   - Discussion of challenges encountered and how you addressed them
   - Reflections on what you learned about controlling model behavior

Bonus Challenges (Extra Credit)

1. Implement a calibration function that tunes classification confidence thresholds based on test data (5 points)
2. Add support for comparing results across different Groq models (5 points)
3. Create a simple web interface using Streamlit or Flask (5 points)

Resources

- Groq API Documentation: https://console.groq.com/docs/quickstart
- Python Groq Client: https://github.com/groq/groq-python
- Prompt Engineering Guide: https://www.promptingguide.ai/

Submission Guidelines

- Submit your code as a GitHub repository
  - The repository should be private and shared with the instructor
  - Include a clear README with setup and usage instructions
  - Ensure your code is well-documented with comments explaining key functionality
  - Do not include your actual API keys in your submission (use .env with a sample template)
- Submit your report as a PDF document through the course learning management system
  - The report should be professionally formatted and include all components listed in the deliverables section
- Include a link to your GitHub repository in your report
- All submissions must be completed by the deadline (March 10, 2025 at 23:59)

Good luck!