**MULTI-AGENT PROMPT ITERATION SYSTEM**

**ASSESSMENT**

BY:-

SATHISH R

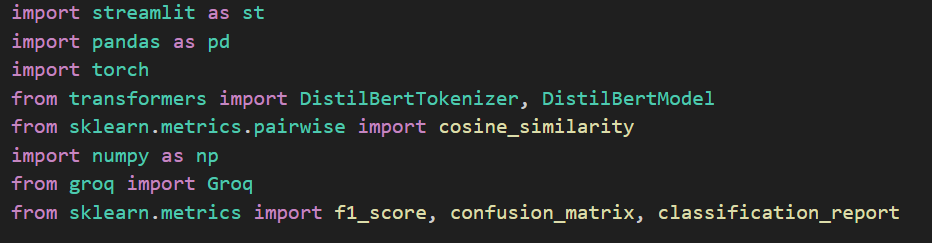
Abstract

In this project, we develop an Agentic AI Tool that iteratively refines prompts to optimize text classification using a Large Language Model (LLM). The system consists of four AI agents—Prompt Creation, Classification, Evaluation, and Feedback Creation—that work in a continuous loop to improve classification accuracy. The objective is to achieve a macro-average F1 score of at least 95% on a dataset of online discussion posts categorized into 12 predefined labels. Each iteration involves generating an initial prompt, classifying posts, evaluating performance, and refining the prompt based on structured feedback. This approach demonstrates the potential of LLM-driven iterative learning for high-accuracy text classification tasks.

INDEX

|  |  |  |
| --- | --- | --- |
| S.no | CONTENT | Pg.no |
| 1 | INSTALL REQUIREMENTS | 4 |
| 2 | AI AGENTS | 5 |
| 3 | DATA PREPROCESSING AND LOADING | 20 |
| 4 | ITERATIVE WORKFLOW | 21 |
| 5 | FRONT END STREAMLIT | 25 |

Install Requirements:



This script imports essential libraries for:  
✅ **Building a UI** (Streamlit)  
✅ **Handling data** (Pandas, NumPy)  
✅ **Using Deep Learning Models** (PyTorch, Hugging Face Transformers)  
✅ **Performing Similarity Calculations** (Cosine Similarity)  
✅ **Evaluating Model Performance** (F1-score, Confusion Matrix, etc.)  
✅ **Interacting with Groq’s AI models**

The Requirements were also added in specific Requirements.txt file

**MULTI-AGENT PROMPT ITERATION SYSTEM**

# Prompt Creation Agent

 The Prompt Creation Agent is responsible for generating and refining the prompt used to guide the LLM in classifying text. Initially, it creates a structured prompt defining the classification task and category descriptions. In subsequent iterations, it incorporates feedback to improve clarity, add examples, and enhance category definitions.

 **Goal**: To generate a high-quality prompt that maximizes classification accuracy.

## Classification Agent

## The Classification Agent applies the latest prompt to classify text using the LLM. It processes the dataset and assigns category labels based on the prompt's instructions.

* **Goal**: To generate accurate predictions for each post in the dataset using the refined prompt.

### Evaluation Agent.

* The Evaluation Agent measures the classification performance by comparing predicted labels with ground-truth labels. It calculates key metrics such as the macro-average F1 score, per-category F1 scores, and a confusion matrix to identify misclassifications.
* **Goal**: To assess classification performance and determine whether the 95% macro-average F1 score threshold has been met.

#### Feedback Creation Agent.

* The Feedback Creation Agent analyzes classification errors and evaluation metrics to generate insights for improving the prompt. It identifies patterns in misclassified posts and provides suggestions to refine category definitions, add examples, or modify language.

##### **Goal**: To provide actionable feedback that helps the Prompt Creation Agent improve future iterations of the prompt.

Code Implementation

### **Using GROQ API to Create an AI Agent**

GROQ provides an API for running AI models efficiently. To create an AI agent using GROQ, you must obtain an API key and set up a client to interact with the service.

Here's a step-by-step guide:

### **1. Get a GROQ API Key**

1. **Visit GROQ’s official website** and sign up or log in.
2. **Go to the API section** and generate an API key.
3. **Copy and store the API key securely** (do not share it publicly).

### **2. Install Required Libraries**

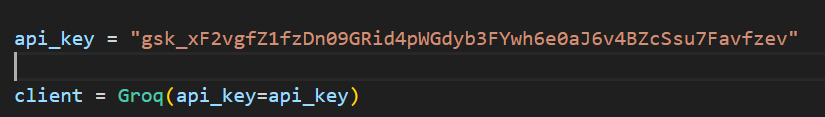
You'll need the groq Python package to interact with the API. Install it using:

Code:

pip install groq

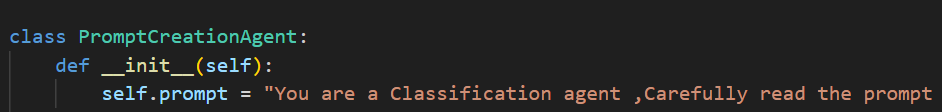
### **3. Initialize the GROQ API Client**

You can use the API key to create a client and make requests.



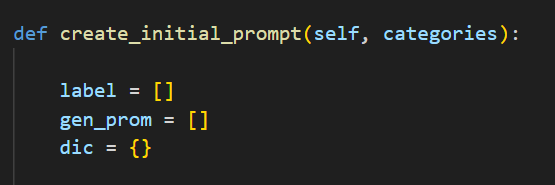
Now we can create AI agents using groq

**1. Prompt Creation Agent**



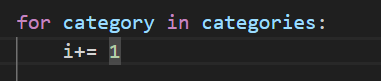
* This defines a **class** named PromptCreationAgent
* This is the **constructor method** (\_\_init\_\_), which initializes the class when an instance is created.
* self.prompt it sets an **initial prompt** that guides the classification agent.
* This prompt acts as a system message for the AI model to understand its role.

**Creating initial prompt for each category:**

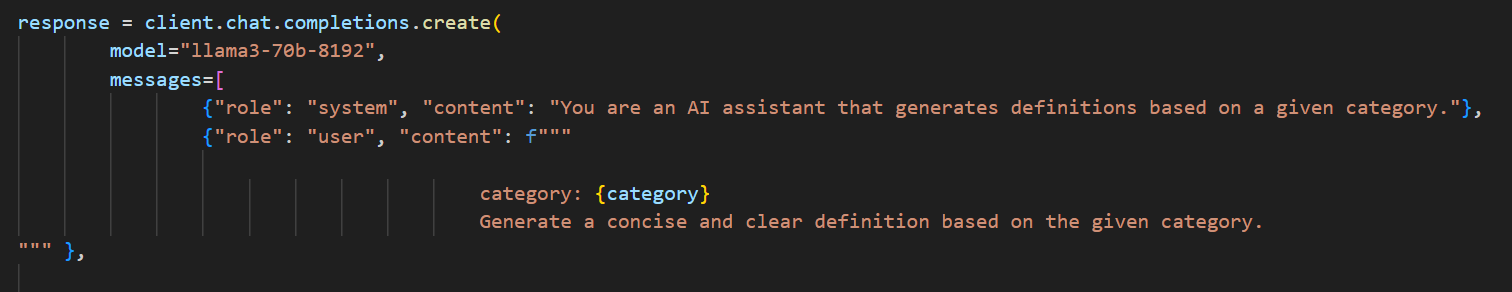


* This function takes a **list of categories** as input.
* It generates a **prompt** that includes definitions for each category using an AI model.
* **label** → Stores category names.
* **gen\_prom** → Stores generated definitions for each category.
* **dic** → A dictionary to map category names to their definitions.
* Generating definitions for each category
* Generating definitions for each category

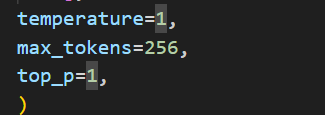
**Generating definitions for each category:**



* Loops through each **category** in the given list

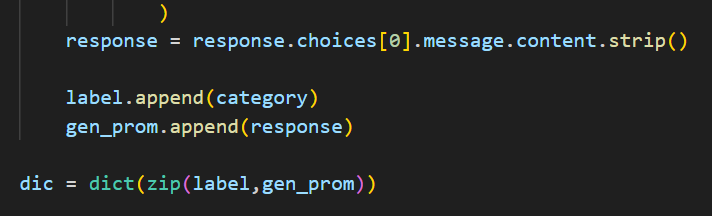


* Calls the LLaMA 3 (70B parameters) model to generate responses.
* client.chat.completions.create(...) sends a prompt to the model.
* The **system message** sets the assistant’s role: "Generate definitions based on a given category".
* The **user message** provides the category name and asks the AI to generate a definition.



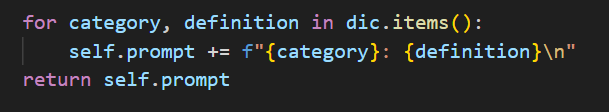
* **temperature=1** → Higher randomness (more creativity in responses).
* **max\_tokens=256** → Limits the response to 256 tokens.
* **top\_p=1** → Consider the full probability distribution for tokenselection

**Processing the AI Response:**



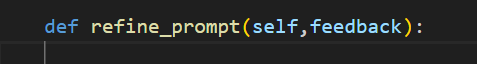
* Extracts and **cleans** the AI-generated definition from the response.
* **Appends** the category and generated definition to label and gen\_prom lists.
* Creates a **dictionary (dic)** mapping category names to their definitions.

**Building final prompt:**

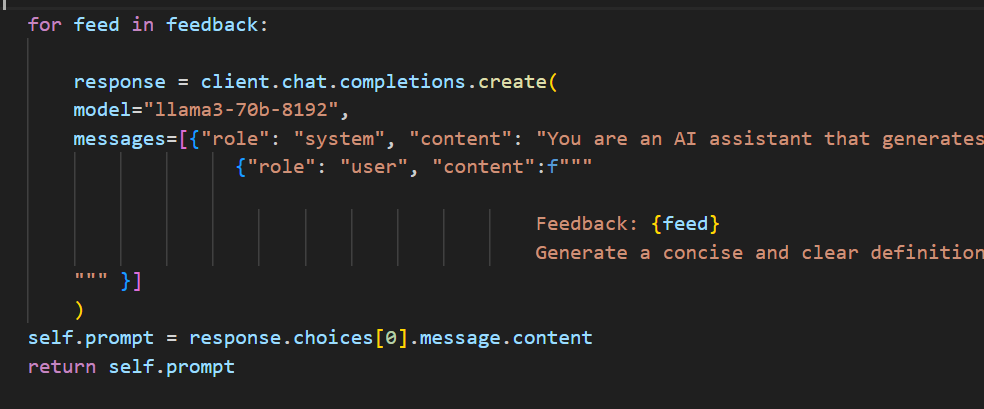
****

* Iterates over the dictionary and appends each **category-definition pair** to the prompt.
* Then returns the **final cons**

**Refine Prompt:**

****

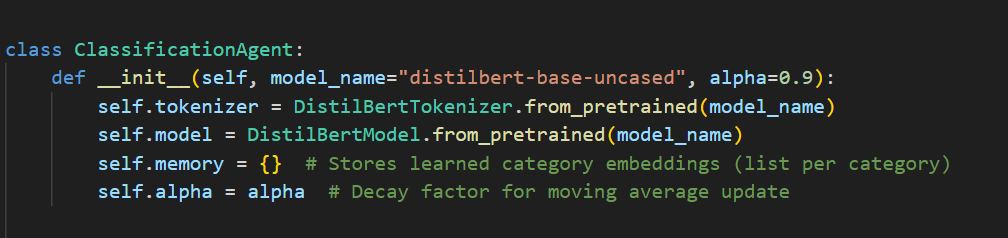
* Takes **feedback** as input and refines the prompt based on it.

****

* Loops through each piece of **feedback**.
* Calls the LLaMA model again to **refine** the definitions based on feedback.
* Updates self.prompt with the **refined definition**.
* Returns the **updated prompt**.

**2. Classification Agent**

Class Definition and Initialization

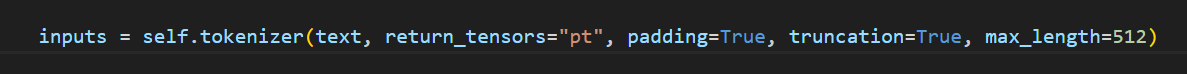


* Defines a **class** named ClassificationAgent.
* It is designed for **text classification** using **DistilBERT** and **cosine similarity**.
* **model\_name="distilbert-base-uncased"** → Uses **DistilBERT** as the embedding model.
* **alpha=0.9** → A **decay factor** for updating category embeddings (used in self-learning).
* Loads **DistilBERT tokenizer** and **model** from Hugging Face's pretrained models.
* **Tokenization** converts text into numerical input for the model.
* **DistilBERT model** generates **word embeddings** (vector representations of text).
* **self.memory** → Stores embeddings for each category.
*  It **learns from misclassifications** and **improves over time**.
* **alpha** controls **how much weight** is given to new embeddings vs. existing ones.

**Get embedding:**



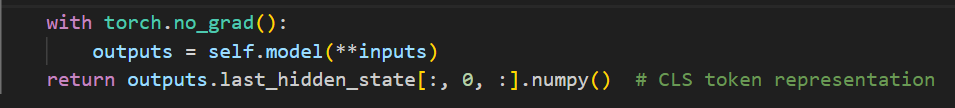
* Converts text into a **numerical vector representation** using DistilBERT.



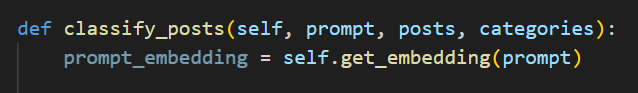
 **Tokenizes** the input text.

 Returns a **PyTorch tensor** (return\_tensors="pt").

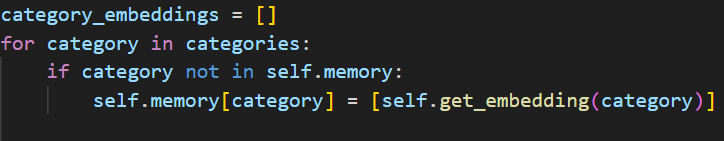
 **Truncates** long texts to **512 tokens** (DistilBERT's limit).



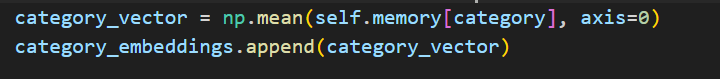
* **Passes** the tokenized text through DistilBERT.
* Uses torch.no\_grad() to **disable gradients** (no training, just inference).
* Extracts **CLS token representation** (first token of output) as the **sentence embedding**.
* Converts the embedding to a **NumPy array** for easier processing.



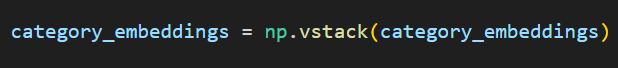
* **Takes** a prompt, a list of posts, and categories as input.
* **Returns** predicted categories for each post
* Converts the **prompt** into an embedding



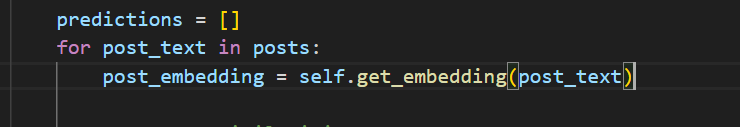
* Creates an **empty list** to store embeddings for each category.
* If a category **does not exist** in memory,
* It **generates an embedding** for that category and **stores it** in memory.



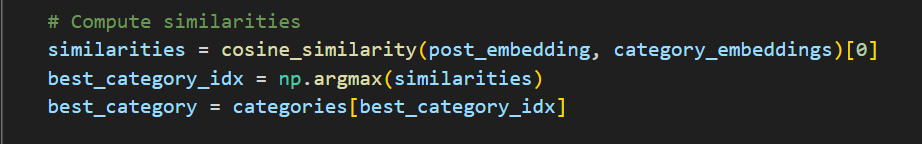
* Computes the **mean embedding** of all stored category vectors.
* This helps **smooth out variations** in the category representation.
* Adds the computed **category embedding** to the list.



* Converts the list into a **NumPy array** for easier matrix operations.

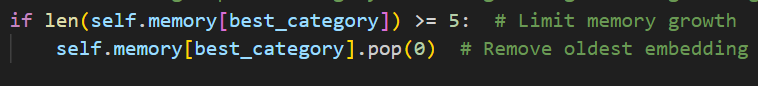


* Initializes an **empty list** to store **predicted categories**.
* Loops through each **post** that needs classification.
* Converts the **post text** into an embedding using DistilBERT.

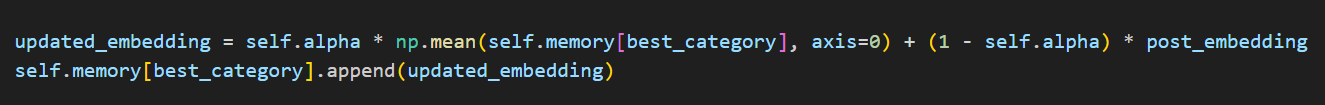


* **Computes cosine similarity** between the **post embedding** and **category embeddings**.
* **Higher similarity** means the post is more likely to belong to that category.
* Finds the **category with the highest similarity score**.
* **Assigns** the corresponding category to the post

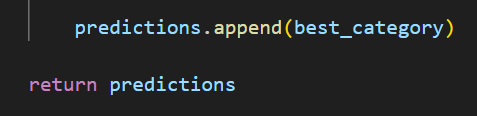
**Self Learning**

****

* If a category has **more than 5 stored embeddings**,
* It **removes the oldest embedding** to **prevent memory overflow**.

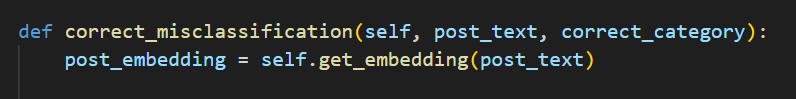


* Updates the **category embedding** using a **moving average**:
* **alpha** controls how much weight is given to previous embeddings vs. new ones.
* **Stores** the updated category embedding for future classifications.

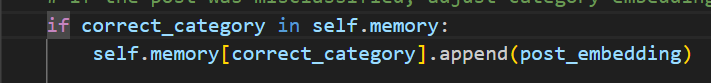


* Adds the **predicted category** to the list

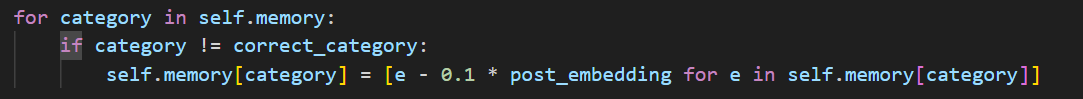
Correct misclassifications



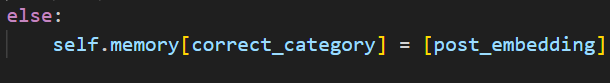
* Updates **category embeddings** when a **misclassification occurs**.
* Takes the **incorrectly classified post** and its **correct category** as input.
* Converts the **misclassified post** into an embedding.

****

* **Adds the post embedding** to the correct category's memory.

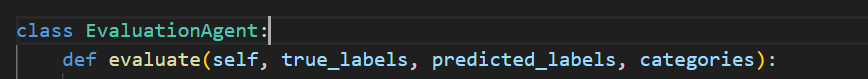
****

* **Pushes the incorrect category's embeddings slightly away** from the misclassified post.
* This helps **reduce future misclassification**.

****

* If the **correct category** does not exist in memory,
* It **initializes** the memory with the new embedding

3.Evaluation Agent



* Defines a method **evaluate** that **calculates evaluation metrics** for the classification model.
* It takes the following **inputs**:
* **true\_labels** → The actual (ground truth) labels of the posts.
* **predicted\_labels** → The labels predicted by the classification agent.
* **categories** → The list of all possible category labels.

1. Compute Macro F1 Score



* Uses **f1\_score** from sklearn.metrics to calculate the **macro-averaged F1 score**.

**What is the F1-score?**

* It measures the **balance between precision and recall**.
* Formula: F1=2×Precision×RecallPrecision+RecallF1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}F1=2×Precision+RecallPrecision×Recall​
* **Why average="macro"?**
* It **computes F1-score separately for each category** and then takes the **unweighted mean**.
* This treats all classes **equally**, regardless of how many samples belong to each.

2. Compute Confusion Matrix



* Uses **confusion\_matrix** from sklearn.metrics to create a **confusion matrix**.
* The confusion matrix helps analyze **where the model makes mistakes**:
* Rows represent **actual categories**.
* Columns represent **predicted categories**.
* Each cell (i, j) shows how many times a **true label i** was classified as **category j**.

3. Classification Report

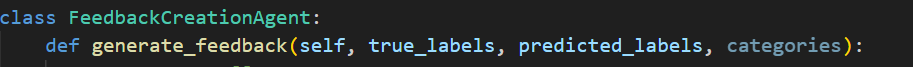


* This line is **commented out**, but if enabled, it would generate a **detailed classification report**.
* It provides **Precision, Recall, F1-score, and Support** for each category.



* **macro\_f1** → The overall **macro F1-score**.
* **confusion** → The **confusion matrix**.
* Classification report with Precision, Recall, and F1-score for each category.

**4.Feedback Agent**



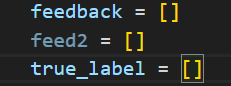
* Defines a class named **FeedbackCreationAgent**.
* This method is responsible for **analyzing misclassified instances** and generating **feedback** to improve classification.

**Inputs:**

**true\_labels** → The actual (correct) labels.

**predicted\_labels** → The labels predicted by the classification agent.

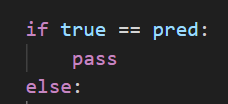
**categories** → The list of possible classification categories.



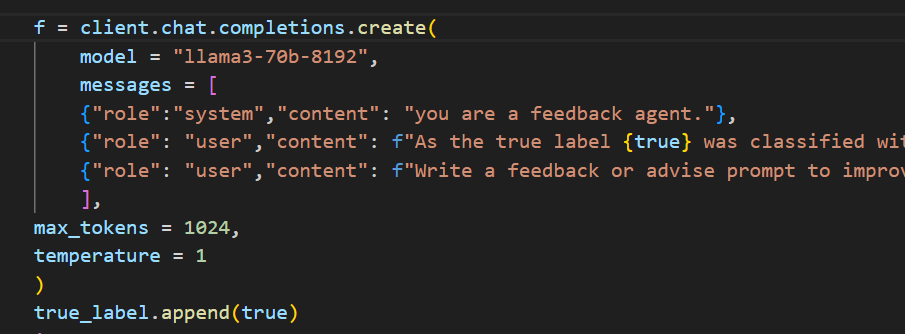
* **feedback** → Stores feedback messages generated by the AI model.
* **feed2** → **(Unused in this code)**—this variable seems unnecessary.
* **true\_label** → Stores the true labels of misclassified examples.



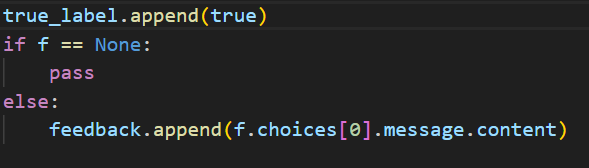
* Iterates through both **true labels** and **predicted labels** at the same time.
* **enumerate(zip(...))** ensures that true and pred are picked **at the same index** from true\_labels and predicted\_labels



* If the prediction (pred) is **correct** (true == pred), it does **nothing** (pass).
* If the prediction is **incorrect**, it proceeds to **generate feedback**.



* Uses the **LLaMA-3 model (llama3-70b-8192)** to generate **feedback**.
* **messages**:
* **System Message:** "you are a feedback agent." → This tells the model its role.
* **User Message 1:** "As the true label {true} was classified with incorrect predicted label {pred} provide more definitions for true label to improve"
  + Asks the model to **generate better definitions** for the correct category.
* **User Message 2:** "Write a feedback or advise prompt to improve the definitions for true label : {true} with examples"
  + Requests the model to **suggest better prompts** for defining the true category.
* **Hyperparameters**:
  + **max\_tokens = 1024** → The response can be up to **1024 tokens**.
  + **temperature = 1** → Higher randomness to encourage **diverse responses**.



* If the **LLaMA-3 response (f) is None**, it does nothing.
* Otherwise, it **extracts the response** and appends it to the **feedback list**.



* Returns a list of feedback messages for **misclassified labels**

**Data Preprocessing & loading Data**

1. Read the Excel file into a DataFrame



* **pd.read\_excel(uploaded\_file)**: Reads an Excel file (uploaded\_file) into a Pandas DataFrame.
* **dataset**: Stores the loaded data as a DataFrame.

2. Clean the category column by removing numeric prefixes



* **dataset['category']**: Accesses the category column
* Uses a regular expression (regex=True) to remove numeric prefixes
* ^\d+\.: Matches numbers at the beginning (^) followed by a dot (.).
* \s\*: Matches any whitespace after the dot.
* Replaces the match with an empty string (''), effectively removing it.
* **dataset['cleaned\_category']**: Stores the cleaned values in a new column.

3. Keep only the first line in cleaned\_category



* **.str.split('\n')**: Splits the string at newlines (\n), creating a list of lines.
* **.str[0]**: Keeps only the first line (removes multi-line text).

4. Drop the original category column



* **.drop(columns=["category"])**: Removes the original category column.
* **dataset =**: Updates the dataset without this column.

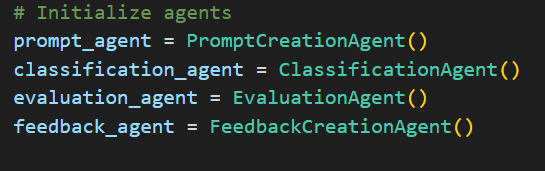
5. Rename cleaned\_category back to category



* Copies cleaned\_category into a new category column.

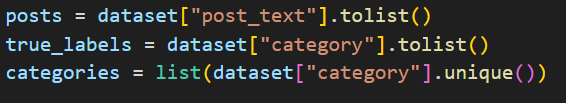
**Iterative Workflow**

1. Initialize Agents



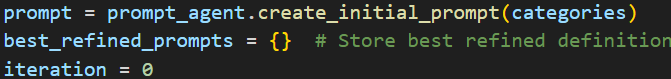
* This initializes the Agents of different classes

2. Extract Posts and Categories from the Dataset



* dataset["post\_text"].tolist(): Extracts the post\_text column (raw text data) as a list.
* dataset["category"].tolist(): Extracts the category column as a list of true labels.
* list(dataset["category"].unique()): Gets unique category labels for classification.

3. Initialize Prompt and Storage for Refined Prompts



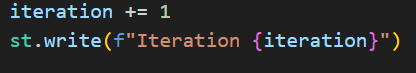
* prompt\_agent.create\_initial\_prompt(categories): Creates an initial prompt using the unique categories.
* best\_refined\_prompts = {}: Initializes a dictionary to store the best refined prompts per category.
* iteration = 0: Sets an iteration counter for tracking the process.

4. Start Iterative Classification and Refinement



* An infinite loop ensures the classification and refinement process continues until a stopping condition (F1 score threshold) is met.

5. Increment Iteration Counter



* Increments iteration and displays it using st.write(), which is part of **Streamlit** (suggesting this code runs in a Streamlit app).

6. Classify Posts Using the Current Prompt.



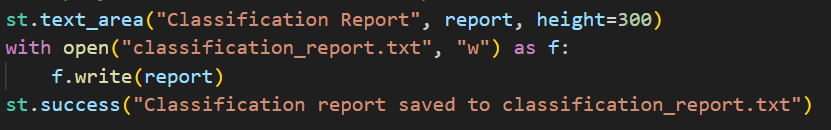
* Uses the classification\_agent to classify posts based on the current prompt.
* Returns predicted\_labels, a list of predicted categories.

7. Evaluate Model Performance



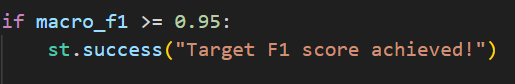
* evaluation\_agent.evaluate(): Evaluates classification performance and returns:
* macro\_f1: The **Macro F1 Score**, a balanced metric for multi-class classification.
* confusion: A confusion matrix (not used explicitly in the code).
* report: A detailed classification report.

8. Display and Save the Classification Report

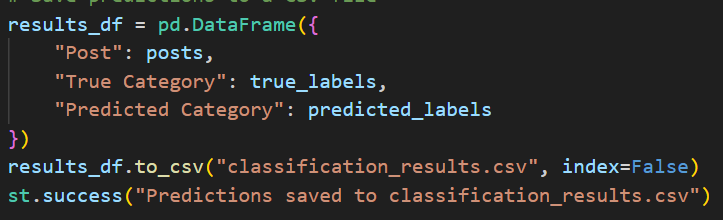


* st.text\_area(): Displays the classification report in a Streamlit text area.
* Saves the report to a text file (classification\_report.txt).
* st.success(): Confirms successful saving.

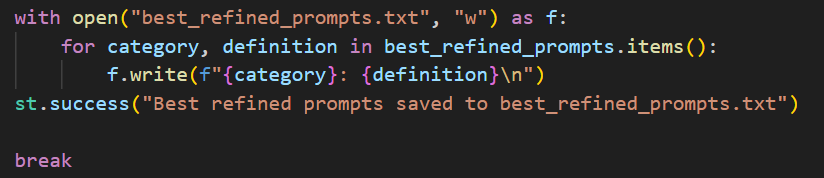
9. Check for Success and Save Results if Target F1 Score is Achieved



* If the Macro F1 Score reaches **0.95 or higher**, the model stops iterating.

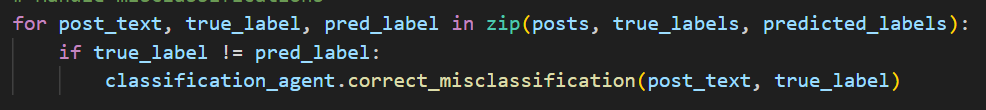


* Creates a Pandas DataFrame with posts, true labels, and predicted labels.
* Saves the results as a CSV file (classification\_results.csv).
* Displays success message.



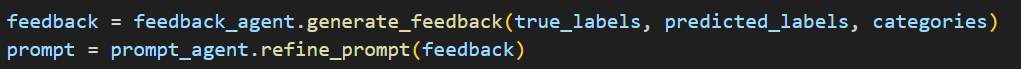
* Saves the best refined prompts per category to a text file (best\_refined\_prompts.txt).
* If the F1 score target is met, the loop terminates.

10. Handle Misclassifications



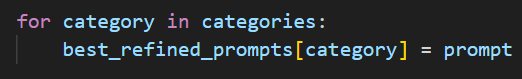
* Iterates through misclassified posts.
* Calls classification\_agent.correct\_misclassification() to adjust classification logic.

11. Generate Feedback for Refining the Prompt



* feedback\_agent.generate\_feedback(): Creates feedback based on misclassifications.
* prompt\_agent.refine\_prompt(feedback): Uses the feedback to improve the prompt.

12. Store Best Refined Prompts



* Updates best\_refined\_prompts with the most recent refined prompt.

### **Summary of Workflow**

1. **Initialize agents** for prompt creation, classification, evaluation, and feedback.
2. **Extract posts & categories** from the dataset.
3. **Generate an initial prompt** based on categories.
4. **Start an iterative process**:
   1. Classify posts.
   2. Evaluate performance (macro f1 score).
   3. Save results.
   4. If the target f1 score is met, stop.
   5. Otherwise, correct misclassifications and refine the prompt.
5. **Store best refined prompts** for future use.

**Frontend using streamlit**

1. Set the Title of the Streamlit App



* **st.title()**: Sets the main title of the Streamlit web app.
* The text "Multi Agent Prompt Iteration System" appears in a large font at the top of the page.

2. File Uploader for Dataset



* **st.file\_uploader()**: Creates a file upload button for users to upload a file.
* "Upload your dataset (Excel file)": Displayed as the upload prompt.
* type=["xlsx"]: Restricts file uploads to **Excel files (.xlsx)** only.
* The uploaded file is stored in uploaded\_file.

3



* **st.text\_input()**: Creates a text input box.
* "Enter your Groq API key": Displayed as the prompt inside the input field.
* type="password": Hides the input text for security, making it a password-style field.
* The entered API key is stored in api\_key.

4. Display Iteration Number



* **st.write()**: Displays text or variables in the Streamlit app.
* Prints the current iteration number dynamically.

5. Display the Macro F1 Score



* Displays the computed **Macro F1 Score** after each iteration.

6. Display the Classification Report



* **st.text\_area()**: Displays a **scrollable text box** with the classification report.
* **height=300**: Sets the height of the text area for better readability.

7. Save and Confirm Classification Report



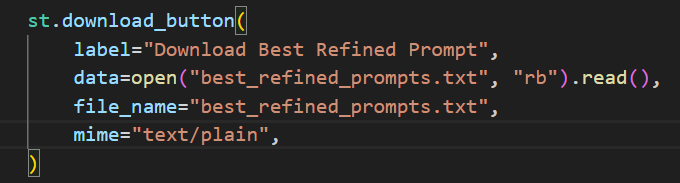
* **st.success()**: Displays a green success message confirming the classification report has been saved.

8. Display Success Message When Target F1 Score is Achieved



Informs the user that the model has met or exceeded the F1 score goal (0.95).

9.Display Download button.



* This **Streamlit** code creates a **download button** that allows users to download the file "best\_refined\_prompts.txt , Classification\_report, Classification\_results.