**COMMUNICATION ANALYSIS TOOL**

A School project submitted by

**GROUP 16**

Department of Computer Science Ladoke Akintola University of Technology, Ogbomosho, Oyo State, Nigeria.

# **Group Details and Course Information.**

* **COURSE TITLE**: **INTRODUCTION TO COMPUTER PROGRAMMING II LABORATORY PRACTICAL (INTERMEDIATE PYTHON PROGRAMMING LABORATORY)**
* **COURSE CODE**: **LAUTECH – CSC 202**
* **GROUP**: **16**
* **PROJECT: Develop a communication analysis tool that processes email archives and message logs, performs sentiment analysis on communication patterns, identifies spam characteristics using Bayesian filtering, analyzes writing style and formality levels, tracks communication frequency and response times, and generates communication effectiveness reports with improvement suggestions for professional correspondence**

***List of Students in Group 16***

1. **Bakare Olamilekan Abdul-Karim – 2023002385**
2. **Suleiman Nasirudeen Adekunle - 2023004354**
3. **Akano Isaac Oluwamayowa - 2023007240**
4. **Bello Ibraheem Olasunkanmi - 2023004247**
5. **Ayoade Faizhah Kehinde - 2023003248**

# 

# **II. Brief Introducti­­­­on.**

This report presents the design and implementation of the Communication Analysis Tool, an object orient programming python only based CLI developed and programmed as a group project by group 16 in the department of Computer Science, Ladoke Akintola University of Technology (LAUTECH). The tool analyzes the communication pattern by detecting spam, evaluating messages tone and examining stylistic features even adding suggestions. This document combines the project overview, requirements and user manual for a comprehensive understanding of the system purpose and design.

Contents

[**I.** **Group Details and Course Information.** 2](#_Toc204686491)

[**II. Brief Introduction.** 3](#_Toc204686492)

[**1.** **Communication Analysis Tool: Project Overview and Architecture** 6](#_Toc204686493)

[**1.1 Core Modules** 6](#_Toc204686494)

[**1.2 Data Flow and Interactions** 7](#_Toc204686495)

[**1.3 Error Handling** 8](#_Toc204686496)

[**2.** **Project Requirements, Object-Oriented Principles and Applying it to Real-World Problem Solving** 9](#_Toc204686497)

[**2.1 Fulfillment of Project Requirements** 9](#_Toc204686498)

[**2.2 Object-Oriented Programming (OOP) Principles** 9](#_Toc204686499)

[**2.3 Standard Library Modules Utilized** 10](#_Toc204686500)

[**2.4 Solving a Real-World Problem** 11](#_Toc204686501)

[**3.** **Technical Documentation of the Communication Analysis Tool.** 12](#_Toc204686502)

[**3.1. Overview** 12](#_Toc204686503)

[**3.2. Features.** 12](#_Toc204686504)

[**3.4 Setup.** 12](#_Toc204686505)

[**3.5. Usage Guidelines.** 14](#_Toc204686506)

[**3.6. Report Output Details.** 16](#_Toc204686507)

[**3.7. Customization.** 17](#_Toc204686508)

[**4.** **Structure of the Source Code of the Communication Analysis Tool** 17](#_Toc204686509)

[**4.1: Data\_parser.py** 19](#_Toc204686510)

[**4.2: Spam\_detector.py** 22](#_Toc204686511)

[**4.3: Sentiment\_analyzer.py** 24](#_Toc204686512)

[**4.4: Style\_analyzer.py** 26](#_Toc204686513)

[**4.5: Metrics\_calculator.py** 28](#_Toc204686514)

[**4.6: Report\_generator.py** 32](#_Toc204686515)

[**4.7: Main.py (CLI Controller)** 49](#_Toc204686516)

[**5.** **Testing and Evaluation** 54](#_Toc204686517)

[**5.1. Unit testing strategy** 54](#_Toc204686518)

[**5.2 CLI Testing** 55](#_Toc204686519)

[**5.4 Sample Results and Evaluation Summary** 62](#_Toc204686520)

[**6.** **Communication Analysis Tool: User Manual for Non-Programmers** 65](#_Toc204686521)

[**6.1. Run full analysis on sample dataset** 65](#_Toc204686522)

[**6.2. Analyze a custom text file** 66](#_Toc204686523)

[**6.3. Type and analyze a custom message** 66](#_Toc204686524)

[**6.4. Exit** 67](#_Toc204686525)

[**7.** **Conclusion** 69](#_Toc204686526)

[**i.** **APPENDIX A – UNIT TEST CODES FOR EACH MODULES** 70](#_Toc204686527)

[**Unit Test Code A.1: test\_sentiment\_analysis.py** 70](#_Toc204686528)

[**Unit Test Code A.2: test\_spam\_detector.py** 72](#_Toc204686529)

[**Unit Test code A.3: test\_style\_analyzer** 74](#_Toc204686530)

[**Unit Test Code A.4: test\_metrics\_calculator.py** 76](#_Toc204686531)

[**Unit Test Code A.5: test\_report\_generator.py** 80](#_Toc204686532)

[**ii.** **APPENDIX B: SAMPLE DATA FOR INTEGRATION AND TESTING** 89](#_Toc204686533)

[**Sample data B.1: Formal email/message sample** 89](#_Toc204686534)

[**Sample data B.2: Informal message sample** 90](#_Toc204686535)

[**Sample data B.3: Spam sample** 92](#_Toc204686536)

[**Sample data B.4: Non-Spam(Legitimate)message sample.** 94](#_Toc204686537)

[**APPENDIX C: Spam Detector trainer** 95](#_Toc204686538)

[**C.1: train\_spam\_detector.py** 95](#_Toc204686539)

[**C.2: Trained data** 96](#_Toc204686540)

[**C.2.1: ham\_word\_count.json** 96](#_Toc204686541)

[**C.2.2: spam\_word\_count.json** 97](#_Toc204686542)

1. **Communication Analysis Tool: Project Overview and Architecture**

This document provides a high-level overview of the Communication Analysis CLI tool's architecture and the interaction between its core Python modules. The tool is designed with a modular approach, where each component handles a specific aspect of the communication analysis pipeline.

**1.1 Core Modules**

The application is structured around several key Python modules, each residing in the modules/ directory:

* **main.py**:
  + **Role:** The entry point of the application. It provides the command-line interface (CLI) and orchestrates user interaction.
  + **Functionality:** Displays the main menu, handles user input for selecting analysis options, and calls the appropriate functions from report\_generator.py based on user choice. It also manages the flow for custom message input, including prompting for message type, sender, and conversation ID.
* **spam\_detector.py**:
  + **Role:** Identifies messages as spam or legitimate (ham).
  + **Functionality:** Likely implements a Bayesian filtering approach. It's initialized with training data to learn patterns associated with spam and ham messages. Its predict() method takes a message string and returns a boolean (True for spam, False for ham).
* **sentiment\_analyzer.py**:
  + **Role:** Determines the emotional tone of a message.
  + **Functionality:** Uses a rule-based approach with predefined lexicons (positive\_words, negative\_words). It counts the occurrences of words from these lexicons in a given text to classify its sentiment as 'positive', 'negative', or 'neutral'.
* **style\_analyzer.py**:
  + **Role:** Assesses the writing style and formality of a message.
  + **Functionality:** Calculates a numerical style score (e.g., based on sentence length, word complexity, or specific formal/informal phrases) and categorizes the message's formality (e.g., 'formal', 'informal').
* **metrics\_calculator.py**:
  + **Role:** Computes aggregate metrics and behavioral insights from a list of analyzed messages.
  + **Functionality:** Takes the structured output from report\_generator (a list of message dictionaries) and calculates:
    - Spam distribution (count of SPAM vs. HAM).
    - Sentiment breakdown (count of positive, neutral, negative).
    - Average style score.
    - Formality breakdown.
    - Top senders (frequency of messages per sender).
    - Average response times within conversations (by grouping messages by Conversation ID and analyzing timestamps).
    - Generates behavioral suggestions based on these metrics.
* **report\_generator.py**:
  + **Role:** The central processing unit that orchestrates the analysis of input data and compiles the final reports.
  + **Functionality:**
    - **Input Handling:** Reads raw text files from data/sample\_emails/ or processes direct user input.
    - **Format Detection:** Employs helper functions (\_is\_whatsapp\_format, \_has\_email\_boundary) to intelligently detect the format of input files (single email, multi-email, WhatsApp chat).
    - **Parsing:** Uses specialized parsing functions (\_parse\_single\_email\_block, \_parse\_multi\_email\_file, \_parse\_whatsapp\_content) to extract message content and metadata (Sender, Conversation ID, Timestamp, Subject) from various formats.
    - **Analysis Orchestration:** For each extracted message, it calls the predict()/analyze() methods of spam\_detector, sentiment\_analyzer, and style\_analyzer.
    - **Result Compilation:** Gathers all analysis results and extracted metadata into a list of structured dictionaries.
    - **Metric Calculation:** Passes the compiled results to metrics\_calculator.py to obtain summary and behavioral data.
    - **Report Generation:** Writes a comprehensive human-readable report to a .txt file in data/reports/.
    - **CLI Feedback:** Prints a condensed summary of metrics and behavioral insights directly to the console using \_print\_summary\_to\_cli.

**1.2 Data Flow and Interactions**

The analysis process follows a clear pipeline:

1. **Input Acquisition:** main.py prompts the user to choose an input method (full dataset, custom file, or typed message).
2. **Raw Data Reading & Parsing:** report\_generator.py reads the raw text content of selected files or directly receives typed input. It then intelligently parses this raw content based on detected format:
   * **Emails:** Headers (Sender, Date, Convo ID, Subject) and message body are extracted.
   * **WhatsApp:** Each line is parsed to extract sender, timestamp, and message. Multi-line messages are reassembled.
   * **Multi-Email Files:** The file is split by ---EMAIL\_BOUNDARY---, and each block is parsed as a separate email.
3. **Individual Message Analysis:** For each distinct message (whether from a file or typed), report\_generator.py sends its content to:
   * spam\_detector.predict()
   * sentiment\_analyzer.analyze()
   * style\_analyzer.analyze() The results (spam status, sentiment, style score, formality) are then added to the message's dictionary along with its extracted metadata.
4. **Aggregate Metric Calculation:** Once all messages from the input source are processed and compiled into a results list, this list is passed to metrics\_calculator.py.
5. **Report Generation & Feedback:** report\_generator.py receives the summary and behavioral insights from metrics\_calculator.py. It then formats and writes these insights, along with the individual message analyses, into a detailed report file. Finally, it calls \_print\_summary\_to\_cli to display a quick overview to the user in the terminal.

**1.3 Error Handling**

The modules incorporate try-except blocks to gracefully handle common issues such as:

* File not found errors.
* ValueError during timestamp parsing (e.g., if a date string is in an unexpected format).
* General exceptions during message processing to ensure the application doesn't crash but logs the error within the report.

This modular design promotes reusability, maintainability, and allows for easier extension of new analysis features or input formats in the future.

1. **Project Requirements, Object-Oriented Principles and Applying it to Real-World Problem Solving**

This Communication Analysis CLI tool has been developed to meet specific project requirements, emphasizing a clear modular design and adherence to Object-Oriented Programming (OOP) principles. It leverages Python's standard library to provide a robust solution to a common real-world communication challenge.

**2.1 Fulfillment of Project Requirements**

The tool successfully addresses the core requirements for analyzing communication logs. It provides functionalities for spam detection, sentiment analysis, writing style assessment, and behavioral pattern identification, all delivered through a user-friendly command-line interface. The generated reports offer comprehensive insights, and the modular structure ensures maintainability and extensibility.

**2.2 Object-Oriented Programming (OOP) Principles**

The application's design heavily utilizes OOP principles to create a scalable, maintainable, and understandable codebase.

* **Encapsulation:**
  + **Demonstrated by:** Each analysis module (SpamDetector, SentimentAnalyzer, StyleAnalyzer) encapsulates its internal data (e.g., word lexicons, trained probabilities) and methods (\_tokenize, \_train, \_calculate\_word\_probability) within its respective class. These internal details are hidden from other parts of the application, and interaction occurs only through well-defined public interfaces (predict(), analyze()).
  + **Benefit:** This prevents unintended modification of internal states, promotes data integrity, and makes each module self-contained and easier to debug or replace.
* **Abstraction:**
  + **Demonstrated by:** Users of the SpamDetector, SentimentAnalyzer, and StyleAnalyzer classes (e.g., report\_generator.py) only interact with high-level methods like predict() or analyze(). They don't need to know the complex internal algorithms (e.g., Bayesian calculations, lexicon lookups, or style scoring formulas).
  + **Benefit:** Simplifies the use of complex functionalities, allowing developers to focus on what a module *does* rather than *how* it does it.
* **Inheritance (Implicit/Composition over Inheritance):**
  + While explicit class inheritance is not heavily used between the main analysis classes (SpamDetector, SentimentAnalyzer, StyleAnalyzer are distinct entities), the design implicitly promotes code reuse through **composition**. For instance, report\_generator.py *composes* its functionality by creating and using instances of SpamDetector, SentimentAnalyzer, and StyleAnalyzer.
  + **Benefit:** This flexible approach allows modules to collaborate without rigid hierarchical relationships, often leading to more adaptable designs.
* **Polymorphism (Implicit):**
  + **Demonstrated by:** Although not through explicit method overriding in a class hierarchy, a form of polymorphism is seen in how report\_generator.py interacts with the analysis modules. Each module offers a consistent analyze() or predict() method signature for processing text, even though their internal implementations differ significantly.
  + **Benefit:** Allows the report\_generator to treat different analysis types uniformly, simplifying the orchestration logic.

**2.3 Standard Library Modules Utilized**

The project strictly adheres to the requirement of using only Python's standard library, ensuring no external dependencies and maximizing portability. Here are at least five different modules used:

1. **os module:**
   * **Usage:** Extensively used for interacting with the operating system, particularly for file and directory operations. Examples include os.path.exists() to check for file presence, os.makedirs() to create report directories, os.listdir() to list files in sample directories, and os.path.join() for constructing platform-independent file paths.
   * **Benefit:** Provides cross-platform compatibility for file system interactions.
2. **datetime module:**
   * **Usage:** Crucial for handling timestamps. It's used to parse date/time strings from communication logs (emails, WhatsApp), format datetime objects into strings for reports, and generate current timestamps for manual input. Functions like datetime.datetime.strptime() for parsing and datetime.datetime.now() for current time are key.
   * **Benefit:** Enables accurate time-based analysis, such as calculating response delays.
3. **re module (Regular Expressions):**
   * **Usage:** Fundamentally important for parsing structured text formats. It's used in report\_generator.py to define and apply complex patterns (re.compile(), re.match()) for extracting sender, timestamp, and message content from WhatsApp chat lines and for detecting email boundaries.
   * **Benefit:** Provides powerful and flexible text pattern matching for robust data extraction from varied log formats.
4. **collections module (defaultdict):**
   * **Usage:** Employed in modules/metrics\_calculator.py and modules/spam\_detector.py. collections.defaultdict is used to simplify counting occurrences (e.g., sender\_freq in metrics\_calculator, ham\_words and spam\_words in spam\_detector) without needing to explicitly check if a key exists before incrementing its value.
   * **Benefit:** Reduces boilerplate code and makes counting operations more concise and efficient.
5. **unittest module:**
   * **Usage:** Although not part of the core application runtime, unittest is a critical standard library module for ensuring the quality and correctness of the code. It provides the framework for writing automated tests for each analysis module and the CLI's functionality.
   * **Benefit:** Facilitates robust development by allowing developers to verify that individual components work as expected and that new changes don't introduce regressions.

**2.4 Solving a Real-World Problem**

This Communication Analysis CLI tool directly addresses the real-world problem of **understanding and managing digital communication overload and effectiveness**. In an era dominated by emails, chat applications, and SMS, individuals and businesses often struggle to:

* **Identify and filter unwanted communication (spam):** Manually sifting through spam is time-consuming and risky. The tool automates this.
* **Gauge emotional tone in text:** Misinterpreting tone can lead to misunderstandings. Sentiment analysis helps users understand the emotional context of messages they receive or send.
* **Assess professional communication quality:** Understanding one's own writing style and formality can help improve professional correspondence.
* **Analyze communication efficiency and responsiveness:** In teams or customer interactions, knowing response times and top communicators can highlight bottlenecks or effective engagement patterns.

By automating these analyses, the tool provides actionable insights that can lead to:

* **Improved productivity:** Less time spent on spam, quicker understanding of message intent.
* **Better relationships:** Enhanced awareness of communication tone and responsiveness.
* **More effective communication strategies:** Data-driven insights to refine writing style and engagement approaches.

It transforms raw, overwhelming communication logs into structured, understandable data, empowering users to make informed decisions about their digital interactions.

# **Technical Documentation of the Communication Analysis Tool.**

## **3.1. Overview**

This is a command-line interface (CLI) tool designed to analyze various forms of text-based communication, providing insights into content characteristics and behavioral patterns. It can process email files, WhatsApp chat exports, and custom user-typed messages.

## **3.2. Features.**

 - Spam Detection: Classifies messages as 'SPAM' or 'HAM'.

 - Sentiment Analysis: Determines the emotional tone (positive, neutral, negative) of messages.

 - Style Analysis: Assesses the formality and calculates a numerical style score for communication.

 - Behavioral Insights: Analyzes communication patterns to identify top senders, average response times (for conversations), and offers suggestions for improving engagement.

 - Comprehensive Reporting: Generates detailed text reports for analysis results, saved to a data/reports/ directory.

 - CLI Summary: Provides a quick, condensed summary of key metrics directly in the terminal after a report is generated.

 - Flexible Input: Supports various file formats and direct user input.

#### **3.3 Supported Communication Formats.**

The tool is designed to intelligently parse the following formats:

 - Standard Email Files: Single .txt files containing email headers (From: , Date:, Subject:, Conversation-ID:) followed by the message body.

 - Multi-Email Files: Single .txt files containing multiple standard emails separated by the ---EMAIL\_BOUNDARY--- string on its own line.

 - WhatsApp Chat Exports: .txt files exported directly from WhatsApp, following the typical MM/DD/YY, HH:MM AM/PM - Sender: Message format.

 - SMS/Other Generic Text: Custom messages typed directly into the CLI.

## **3.4 Setup.**

 - Clone or download: Get the project files.

 - Python Environment: Ensure you have Python 3.8+ installed.

 - Directory Structure:

Make sure your project directory has the following structure:

**Group16/**

**├── main.py                             # Main CLI entry point for the tool**

**├── train\_spam\_detector.py             # Script to train a spam detection model**

**├── test\_main.py                       # General test file to run multiple modules (manual or CLI tests)**

**├── README.md                          # Brief project overview and usage instructions**

**├── technical\_documentation.docx       # Detailed technical insights (architecture, modules, etc.)**

**├── Communication Analysis Tool - Project Overview.docx   # High-level project vision & objectives**

**├── Project Requirements.docx          # Functional and non-functional requirements**

**├── Analysis Tool User Manual.docx     # End-user manual with usage instructions**

**├── data/                              # Stores all data-related subfolders**

**│   ├── reports/                       # Generated reports will be saved here**

**│   ├── sample\_emails/                # Default sample emails for analysis**

**│   └── training\_data/                # Data used for training the spam classifier**

**├── digitallogsample/                 # Raw log samples from digital conversations**

**│   ├── ham\_messages.txt              # Clean/ham messages sample**

**│   └── spam\_messages.txt             # Spam messages sample**

**├── modules/                          # Core analysis modules**

**│   ├── \_\_init\_\_.py                   # Makes 'modules' a Python package**

**│   ├── data\_parser.py                # Parses and structures raw input data**

**│   ├── spam\_detector.py              # Detects spam using trained model**

**│   ├── sentiment\_analyzer.py         # Analyzes sentiment (positive, negative, neutral)**

**│   ├── style\_analyzer.py             # Scores writing style and formality**

**│   ├── report\_generator.py           # Orchestrates overall analysis and report formatting**

**│   └── metrics\_calculator.py         # Calculates summary stats and behavioral metrics**

**├── tests/                            # Unit tests for each module**

**│   ├── \_\_init\_\_.py                   # Makes 'tests' a Python package**

**│   ├── test\_data\_parser.py           # Tests for `data\_parser.py`**

**│   ├── test\_spam\_detector.py         # Tests for `spam\_detector.py`**

**│   ├── test\_sentiment\_analyzer.py    # Tests for `sentiment\_analyzer.py`**

**│   ├── test\_style\_analyzer.py        # Tests for `style\_analyzer.py`**

**│   ├── test\_report\_generator.py      # Tests for `report\_generator.py`**

**│   └── test\_metrics\_calculator.py    # Tests for `metrics\_calculator.py`**

 - data/training\_data/: This directory should contain files necessary for your SpamDetector to train (e.g., ham\_messages.txt, spam\_messages.txt)

 - data/sample\_emails/: This is where you'll place your email and WhatsApp .txt files for analysis

## **3.5. Usage Guidelines.**

Navigate to the your\_project\_root/ directory in your terminal or command prompt. #in this case which is Group 16

 'cd C:\Users\ABDULL\Desktop\Group 16 **#the project's path directory**

 'python main.py'

You will be presented with the main menu:

 'Communication Analysis CLI'

 '1. Run full analysis on sample dataset'

 '2. Analyze a custom text file'

 '3. Type and analyze a custom message'

 '4. Exit'

1. Run full analysis on sample dataset

This option processes all **'.txt'** files found in the **'data/sample\_emails/'** directory. It will automatically detect if a file is a single email, a multi-email file, or a WhatsApp chat export.

 -Input: Enter '1'.

 -Output: A comprehensive report will be saved to **'data/sample\_emails/'** and a summary will be printed to the CLI.

2. Analyze a custom text file

This option allows you to select a specific file from your **'data/sample\_emails/'** directory or provide a full path to any `**.txt'** file on your system.

 -Input: Enter '2'.

  You will see a numbered list of available sample files.

  To select a sample file: Enter its corresponding number.

  To enter a path manually: Enter '0', then type the full path to your '.txt' file.

 -Output: A detailed report for the selected file will be saved to data/reports/ and a summary will be printed to the CLI.

3. Type and analyze a custom message

This option allows you to manually type a message and its associated metadata for immediate analysis.

 -Input: Enter '3'.

 -Message Type Selection:

  You will first choose the type of message you're typing:

   'Select Message Type:'

      '1. WhatsApp Chat Message'

      '2. Email Message (with optional headers)'

      '3. SMS Message'

      '4. Other / Generic Text'

      '5. Back to Main Menu'

Choose 1 for WhatsApp, 2 for Email, 3 for SMS, 4 for Other.

Choose 5 to go back to the main menu.

Metadata Input:

 -You will be prompted for a Sender name and Conversation ID. These are optional.

 -To go back to message type selection: Type 'back' at either the sender or conversation ID prompt.

Message Content Input:

 -You will then be prompted to type your message.

 To finish typing: Press [Enter] on an empty line.

 To cancel and go back to message type selection: Type cancel on a new line and press [Enter].

Output: A report for your typed message will be saved to 'data/reports/' and a summary will be printed to the CLI.

Continuous Analysis: After analysis, you will be asked if you want to "Analyze another custom message? (y/n)". Type 'y' to analyze another message of any type, or 'n' to return to the main menu.

4. Exit

Input: Enter '4'.

Output: The CLI will close.

## **3.6. Report Output Details.**

Reports are saved as '**.txt'** files in the **'data/reports/'** directory. Each report includes:

 - Individual Message Analysis: Detailed breakdown for each parsed message, including source, message ID, subject, sender, conversation ID, timestamp, message body preview, spam status, sentiment, style score, and formality.

 - Summary Metrics: Aggregated statistics like total messages, spam distribution, sentiment breakdown, average style score, and formality distribution.

 - Behavioral Insights: Analysis of communication patterns, including top senders, average response delay (if sufficient data is available within a conversation), and actionable suggestions.

## **3.7. Customization.**

 - Lexicons: You can modify the positive\_words and negative\_words sets in modules/sentiment\_analyzer.py to refine sentiment detection for specific contexts.

 - WhatsApp Regex: If your WhatsApp export format differs, you might need to adjust the whatsapp\_line\_pattern regex in modules/report\_generator.py.

 - Email Boundary: The EMAIL\_BOUNDARY\_SEPARATOR in modules/report\_generator.py can be changed if you prefer a different separator for multi-email files but this was used.

# **Structure of the Source Code of the Communication Analysis Tool**

This is the arrangement of the IDLE that was used in this case Microsoft Visual Studio Code(VS code) and the structural architecture is a follows:

**├── main.py                             # Main CLI entry point for the tool**

**├── train\_spam\_detector.py             # Script to train a spam detection model**

**├── test\_main.py                       # General test file to run multiple modules (manual or CLI tests)**

**├── README.md                          # Brief project overview and usage instructions**

**├── technical\_documentation.docx       # Detailed technical insights (architecture, modules, etc.)**

**├── Communication Analysis Tool - Project Overview.docx   # High-level project vision & objectives**

**├── Project Requirements.docx          # Functional and non-functional requirements**

**├── Analysis Tool User Manual.docx     # End-user manual with usage instructions**

**├── data/                              # Stores all data-related subfolders**

**│   ├── reports/                       # Generated reports will be saved here**

**│   ├── sample\_emails/                # Default sample emails for analysis**

**│   └── training\_data/                # Data used for training the spam classifier**

**├── digitallogsample/                 # Raw log samples from digital conversations**

**│   ├── ham\_messages.txt              # Clean/ham messages sample**

**│   └── spam\_messages.txt             # Spam messages sample**

**├── modules/                          # Core analysis modules**

**│   ├── \_\_init\_\_.py                   # Makes 'modules' a Python package**

**│   ├── data\_parser.py                # Parses and structures raw input data**

**│   ├── spam\_detector.py              # Detects spam using trained model**

**│   ├── sentiment\_analyzer.py         # Analyzes sentiment (positive, negative, neutral)**

**│   ├── style\_analyzer.py             # Scores writing style and formality**

**│   ├── report\_generator.py           # Orchestrates overall analysis and report formatting**

**│   └── metrics\_calculator.py         # Calculates summary stats and behavioral metrics**

**├── tests/                            # Unit tests for each module**

**│   ├── \_\_init\_\_.py                   # Makes 'tests' a Python package**

**│   ├── test\_data\_parser.py           # Tests for `data\_parser.py`**

**│   ├── test\_spam\_detector.py         # Tests for `spam\_detector.py`**

**│   ├── test\_sentiment\_analyzer.py    # Tests for `sentiment\_analyzer.py`**

**│   ├── test\_style\_analyzer.py        # Tests for `style\_analyzer.py`**

**│   ├── test\_report\_generator.py      # Tests for `report\_generator.py`**

**│   └── test\_metrics\_calculator.py    # Tests for `metrics\_calculator.py`**

## **4.1: Data\_parser.py**

***This particular module/class is what is taking control of breaking down the particular log formats into a more python readable set of strings for more analysis and to bring about uniformity****:*

"""

This module contains functions responsible for parsing various communication log formats

(single emails, multi-email files, and WhatsApp chat exports) into a structured

dictionary format for further analysis.

"""

from datetime import datetime

#logic for breaking down data to be readable for python

class Message:

    def \_\_init\_\_(self, sender, recipient, timestamp, subject, body, is\_outgoing):

        self.sender = sender  # who sent the message

        self.recipient = recipient  # who received the message

        self.timestamp = timestamp  # Datetime object for when it was sent

        self.subject = subject  # subject line

        self.body = body  # full message content

        self.is\_outgoing = is\_outgoing  # True if sent by the user

class MessageParser:

    def \_\_init\_\_(self, user\_alias=None):

        self.user\_alias = user\_alias  # Helps flag outgoing messages

    def parse\_file(self, filepath):

        messages = []  # final list of parsed Message objects

        current\_msg = {}  # Temp dictionary to store one message at a time

        try:

            with open(filepath, 'r', encoding='utf-8') as f:

                for line in f:

                    line = line.strip()  # remove leading/trailing whitespace

                    # Parse sender

                    if line.startswith("From:"):

                        current\_msg['sender'] = line[5:].strip()

                    # Parse recipient

                    elif line.startswith("To:"):

                        current\_msg['recipient'] = line[3:].strip()

                    # Parse date and convert to datetime object

                    elif line.startswith("Date:"):

                        date\_str = line[5:].strip()

                        try:

                            current\_msg['timestamp'] = datetime.strptime(date\_str, "%Y-%m-%d %H:%M:%S")

                        except ValueError:

                            current\_msg['timestamp'] = datetime.now()  # fallback if date format is wrong

                    # Parse subject

                    elif line.startswith("Subject:"):

                        current\_msg['subject'] = line[8:].strip()

                    # Start body section

                    elif line.startswith("Body:"):

                        current\_msg['body'] = ""

                    # End of one message, create Message object and add to list

                    elif line == "---":

                        msg = Message(

                            sender=current\_msg.get('sender', ''),

                            recipient=current\_msg.get('recipient', ''),

                            timestamp=current\_msg.get('timestamp', datetime.now()),

                            subject=current\_msg.get('subject', ''),

                            body=current\_msg.get('body', ''),

                            is\_outgoing=(current\_msg.get('sender') == self.user\_alias)

                        )

                        messages.append(msg)

                        current\_msg = {}  # reset for next message

                    # Add lines to body if in body section

                    else:

                        if 'body' in current\_msg:

                            current\_msg['body'] += line + "\n"

        except FileNotFoundError:

            print(f"X File not found: {filepath}")  # error message

        return messages

# This parser assumes a specific format for the archives.

# The format is expected to have "From:", "To:", "Date:", "Subject:", and "Body:" lines,

# with "---" separating each

class WhatsappLogParser:

    def \_\_init\_\_(self, user\_alias=None):

        self.user\_alias = user\_alias #whether the message is incoming or outgoing

    def parse\_file(self, filepath):

        messages = []

        try:

            with open(filepath, 'r', encoding='utf-8') as f:

                for line in f:

                    line = line.strip()

                    #pattern: "20/07/2025, 10:04 - Bakare: Messsage text

                    if " - " in line and ":" in line:  # checking if it looks like a whatsapp text

                     try:

                         #spliting timestamp from message

                         timestamp\_part, content\_part= line.split(" - " , 1)

                         timestamp = datetime.strptime(timestamp\_part, "%d/%m/%y, %H:%M")

                         #seperating sender from body

                         sender\_part, body = content\_part.split(":", 1)

                         sender= sender\_part.strip()

                         recipient = self.user\_alias if sender != self.user\_alias else "Unknown" #determining the recipient

                         msg = Message(

                             sender=sender,

                             recipient=recipient,

                             timestamp=timestamp,

                             subject="",

                             body=body.strip(),

                             is\_outgoing=(sender == self.user\_alias)

                         )

                         messages.append(msg)

                     except ValueError:

                         continue #skip malformed lines

        except FileNotFoundError:

                             print(f"X File not found: {filepath}")

#format detection function

def get\_parser\_for\_file(filepath, user\_alias=None):

    with open(filepath, 'r', encoding='utf-8') as f:

        first\_line = f.readline().strip()

        # Check the first line to determine the format

        if first\_line.startswith("From:") or first\_line.startswith("To:"):

            return MessageParser(user\_alias)

        elif "-" in first\_line and ":" in first\_line and "," in first\_line:

            return WhatsappLogParser(user\_alias)

        else:

            raise ValueError("Unknown file format")

## **4.2: Spam\_detector.py**

***Now once the data has been parsed, it would be passed through this particular module which implements the Probability function called Bayes’ theorem or Bayesian spam detection filtering in python and uses a trainer which spam phrases are being saved and used to train the model, the source code of the trainer can be found in the appendix section*.**

"""

spam\_detector.py

This module implements a Bayesian spam detection system.

It uses word frequency analysis to differentiate spam from ham (non-spam) messages.

Training and prediction are handled using only Python standard libraries.

Classes:

    SpamDetector: Main class that supports training, predicting, and model persistence.

"""

import json

import math

import os

from collections import defaultdict

class SpamDetector:

    def \_\_init\_\_(self, data\_path='data/training\_data'):

        self.spam\_word\_counts = defaultdict(int)

        self.ham\_word\_counts = defaultdict(int)

        self.spam\_total = 0

        self.ham\_total = 0

        self.data\_path = data\_path

        self.\_load\_model()

    def train(self, text, is\_spam):

        words = text.lower().split()

        for word in words:

            if is\_spam:

                self.spam\_word\_counts[word] += 1

                self.spam\_total += 1

            else:

                self.ham\_word\_counts[word] += 1

                self.ham\_total += 1

    def train\_from\_text(self, message, is\_spam=True):

        """

        Trains the spam detector using a single string message.

        Useful for unit testing and quick training without saving state.

        """

        if not isinstance(message, str) or not message.strip():

            return  # Ignore empty or invalid input

        words = message.lower().split()

        for word in words:

            if is\_spam:

                self.spam\_word\_counts[word] += 1

                self.spam\_total += 1

            else:

                self.ham\_word\_counts[word] += 1

                self.ham\_total += 1

    def predict(self, text):

        words = text.lower().split()

        spam\_score = 0

        ham\_score = 0

        # Avoid division by zero

        spam\_total = self.spam\_total if self.spam\_total > 0 else 1

        ham\_total = self.ham\_total if self.ham\_total > 0 else 1

        vocab = set(self.spam\_word\_counts) | set(self.ham\_word\_counts)

        vocab\_size = len(vocab) if vocab else 1

        for word in words:

            spam\_prob = (self.spam\_word\_counts[word] + 1) / (spam\_total + vocab\_size)

            ham\_prob = (self.ham\_word\_counts[word] + 1) / (ham\_total + vocab\_size)

            spam\_score += math.log(spam\_prob)

            ham\_score += math.log(ham\_prob)

        return spam\_score > ham\_score  # True means spam

    def \_save\_model(self):

        os.makedirs(self.data\_path, exist\_ok=True)

        with open(os.path.join(self.data\_path, "spam\_word\_counts.json"), "w") as f:

            json.dump(dict(self.spam\_word\_counts), f)

        with open(os.path.join(self.data\_path, "ham\_word\_counts.json"), "w") as f:

            json.dump(dict(self.ham\_word\_counts), f)

    def \_load\_model(self):

        try:

            with open(os.path.join(self.data\_path, "spam\_word\_counts.json")) as f:

                self.spam\_word\_counts.update(json.load(f))

            with open(os.path.join(self.data\_path, "ham\_word\_counts.json")) as f:

                self.ham\_word\_counts.update(json.load(f))

            self.spam\_total = sum(self.spam\_word\_counts.values())

            self.ham\_total = sum(self.ham\_word\_counts.values())

        except FileNotFoundError:

            pass  # Start fresh if no saved data exists

## **4.3: Sentiment\_analyzer.py**

***The parsed logs also pass through this module with the aim of bring out the feeling behind the log, whether it is positive, negative or neutral conveying the exact emotions behind the message that has been sent for future uses.***

"""

sentiment\_analyzer.py

This module provides rule-based sentiment analysis using an internal lexicon.

Each message is analyzed for emotional tone: positive, neutral, or negative.

"""

class SentimentAnalyzer:

    def \_\_init\_\_(self):

        # Basic lexicon for demonstration

        # Expanded with words relevant to project updates and alerts

        self.positive\_words = {

            "good", "great", "happy", "love", "excellent", "awesome", "fantastic", "yes", "thank", "thanks",

            "congratulations", "well", "best", "nice", "perfect", "fresh", "gifted", "positive", "meaningful", "bless",

            "successfully", "ahead", "smoothly", "promising", "appreciated", "progress", "achieved", "strong",

            "effective", "efficient", "valuable", "improved", "optimistic", "beneficial", "resolved", "completed"

        }

        self.negative\_words = {

            "bad", "terrible", "poor", "sad", "hate", "awful", "boring", "disappointing", "horrible", "nasty", "ugly",

            "disgusting", "no", "never", "regret", "waste", "failure", "problem", "stupid", "negative", "meaningless",

            "urgent", "breach", "detected", "suspended", "critical", "issue", "error", "risk", "concern",

            "delay", "difficult", "challenge", "unacceptable", "warning", "alert", "security"

        }

        # Neutral words can be added if you want to explicitly balance counts,

        # but for simple pos/neg/neutral based on counts, it's often not strictly necessary

        # unless you want to explicitly 'cancel out' a positive/negative word in certain contexts.

        # self.neutral\_words = {"the", "a", "is", "are", "it", "this", "that", "and", "or", "but"}

    def analyze(self, text):

        """

        Analyzes the sentiment of the given text.

        Returns 'positive', 'negative', or 'neutral'.

        """

        text = text.lower()

        # Split text into words. You might consider more sophisticated tokenization

        # if punctuation attached to words is an issue (e.g., "good." vs "good")

        words = text.split()

        # Corrected logic: Count words from the input text that are present in the lexicons

        pos\_count = sum(1 for word in words if word in self.positive\_words)

        neg\_count = sum(1 for word in words if word in self.negative\_words)

        # You could introduce a threshold here if you want a stronger bias for positive/negative

        # For example, if pos\_count is only 1 greater than neg\_count, maybe it's still neutral.

        # This is a simple example:

        # sentiment\_difference = pos\_count - neg\_count

        # if sentiment\_difference > 2: # Needs at least 3 more positive words than negative

        #     return "positive"

        # elif sentiment\_difference < -2: # Needs at least 3 more negative words than positive

        #     return "negative"

        # else:

        #     return "neutral"

        if pos\_count > neg\_count:

            return "positive"

        elif neg\_count > pos\_count:

            return "negative"

        else:

            return "neutral"

## **4.4: Style\_analyzer.py**

***This calculates the level of formality in a message log, it has a list of informal and formal words to check from and using those lists, it determines whether it is formal or informal and classifies them after they have been passed through this module.***

"""

style\_analyzer.py

Analyzes writing style and formality of messages.

Evaluates lexical density, sentence structure, informal and formal terms,

filler words, and produces a style score with interpretation.

"""

import re

class StyleAnalyzer:

    def \_\_init\_\_(self):

        # Terms flagged for informal tone or filler usage

        self.informal\_words = {

            "gonna", "wanna", "gotta", "kinda", "sorta", "btw", "gimme", "lemme",

            "lol", "whatcha", "y'all", "hey", "yo", "wassup", "lmao", "bruh", "bro",

            "dude", "chill", "c'mon", "omg", "idk"

        }

        self.contractions = {"i'm", "you're", "can't", "won't", "isn't", "don't"}

        self.filler\_words = {"just", "really", "actually", "basically", "kind of"}

        # Phrases indicating formal tone or politeness

        self.formal\_phrases = {"please", "kindly", "regards", "thank you", "best wishes"}

    def split\_sentences(self, text):

        """More robust sentence splitter using basic regex."""

        return re.findall(r"[^.!?]+[.!?]?", text.strip())

    def analyze(self, text):

        """

        Analyze a text message for stylistic quality and formality.

        Returns a dictionary with metrics like lexical density, filler use, style score, etc.

        """

        text\_lower = text.lower()

        # Tokenize words (including contractions)

        words = re.findall(r"\b[\w']+\b", text\_lower)

        word\_count = len(words)

        unique\_count = len(set(words))

        # Sentence analysis

        sentences = self.split\_sentences(text)

        sentence\_lengths = [len(re.findall(r"\b[\w']+\b", s)) for s in sentences if s.strip()]

        avg\_sentence\_length = sum(sentence\_lengths) / len(sentence\_lengths) if sentence\_lengths else 0

        # Style feature counts

        informal\_count = sum(1 for w in words if w in self.informal\_words)

        contraction\_count = sum(1 for w in words if w in self.contractions)

        filler\_count = sum(1 for w in words if w in self.filler\_words and w not in self.informal\_words)

        formal\_count = sum(1 for phrase in self.formal\_phrases if phrase in text\_lower)

        # Lexical density

        lexical\_density = unique\_count / word\_count if word\_count > 0 else 0

        # Style score (scaled to message length)

        score = 100

        penalty = informal\_count \* 3 + contraction\_count \* 2 + filler\_count \* 2

        if word\_count > 0:

            penalty\_scale = min(1, 20 / word\_count)  # Less penalty for long texts

            score -= penalty \* penalty\_scale \* 5

        if avg\_sentence\_length < 7:

            score -= 5

        score += formal\_count \* 2

        score += lexical\_density \* 10

        score = max(0, min(round(score), 100))  # Bound within 0–100

        # Determine formality

        if informal\_count == 0 and formal\_count > 0:

            formality = "formal"

        elif informal\_count >= 2:

            formality = "informal"

        else:

            formality = "neutral"

        return {

            "Word Count": word\_count,

            "Lexical Density": round(lexical\_density, 2),

            "Average Sentence Length": round(avg\_sentence\_length, 1),

            "Informal Terms": informal\_count,

            "Contractions": contraction\_count,

            "Filler Words": filler\_count,

            "Formal Indicators": formal\_count,

            "Style Score": score,

            "Formality": formality

        }

## **4.5: Metrics\_calculator.py**

***This calculate the total amount of spam, legitimate, the feeling breaks down, the formality of the style and joins them to work with the report generator to bring about a compiled summary stat in the command line interface in the output.***

""""

metrics\_calculator.py

Calculates summary metrics from analyzed messages , including:

- Spam distribution

- Sentiment breakdown

- Style score average

- Formality scores

Accepts results lst from repor\_generator and returns a summary dictionary.

"""

from collections import defaultdict

from datetime import datetime

def calculate\_content\_metrics(results):

    #initialize counters and containers

    total = 0 #total number of successfully analyzed messages

    spam\_counts = {"SPAM":0, "HAM":0} #Tracks spam/ham predictions

    sentiment\_counts = {"positive":0, "neutral":0, "negative":0} #sentiment distribution

    style\_scores = [] #collects all nmerc style scores to calculate average

    # FIX: Initialize formality\_counts with lowercase keys to match style\_analyzer output

    formality\_counts = {"formal":0 , "informal":0} #tracks formality tags

    #loop through each anlyzed message in the results list

    for entry in results:

        if "Error" in entry:

            continue #skip entries that had processing errors

        total += 1 #count valid messages

        #count SPAM/HAM predictions

        spam\_value = entry.get("Spam")

        if spam\_value in spam\_counts:

            spam\_counts[spam\_value] += 1

        #Count sentiment labels

        sentiment\_value = entry.get("Sentiment")

        if sentiment\_value in sentiment\_counts:

            sentiment\_counts[sentiment\_value] += 1

        #Collect style scores for averaging

        style\_score = entry.get("Style Score")

        if isinstance(style\_score,(int,float)):

            style\_scores.append(style\_score)

        # Count formality labels

        formality = entry.get("Formality")

        # Ensure formality value is lowercase before checking against counts

        if formality and formality.lower() in formality\_counts:

            formality\_counts[formality.lower()] += 1

    #Calculate average style score (rounded to 3 decimals), or 0.0 if no scores if no scores found

    avg\_style\_score = round(sum(style\_scores)/ len(style\_scores),3) if style\_scores else 0.0

    #return all metrics as a dictionary to be included to be included in the report

    return{

        "Total Messages": total,

        "Spam Counts": spam\_counts,

        "Sentiment Counts": sentiment\_counts,

        "Average Style Score": avg\_style\_score,

        # Return formality counts with original casing or standardized casing if preferred

        "Formality Counts": formality\_counts

    }

#BEHAVIOURAL ANALYSIS

def calculate\_engagement\_metrics(results):

    sender\_freq= defaultdict(int)

    conversations\_times = defaultdict(list)

    response\_gaps = []

    # FIX: Initialize formality\_counts with lowercase keys

    formality\_counts = {"formal":0 , "informal":0}

    style\_scores = []

    for entry in results:

        if "Error" in entry:

            continue

        sender = entry.get("Sender")

        if sender:

            sender\_freq[sender] +=1

        style\_score = entry.get("Style Score")

        if isinstance(style\_score,(int,float)):

            style\_scores.append(style\_score)

        formality = entry.get("Formality")

        # Ensure formality value is lowercase before checking against counts

        if formality and formality.lower() in formality\_counts:

            formality\_counts[formality.lower()] += 1

        convo\_id = entry.get("Conversation ID")

        timestamp = entry.get("Timestamp")

        if convo\_id and timestamp:

            try:

                # Ensure timestamp is a datetime object or convert it

                if isinstance(timestamp, datetime):

                    time\_obj = timestamp

                elif isinstance(timestamp, str):

                    time\_obj = datetime.strptime(timestamp, "%Y-%m-%d %H:%M:%S")

                else:

                    # Handle cases where timestamp is neither datetime nor string

                    print(f"Warning: Unexpected timestamp type for convo '{convo\_id}': {type(timestamp)}")

                    continue # Skip this entry for time calculation

                conversations\_times[convo\_id].append(time\_obj)

            except ValueError:

                # This will catch if the string format doesn't match

                print(f"Warning: Could not parse timestamp '{timestamp}' for convo '{convo\_id}'.")

                pass # Continue processing other entries

    for times in conversations\_times.values():

        if len(times) >= 2:

            times.sort()

            for i in range(1, len(times)):

                gap = (times[i] - times[i - 1]).total\_seconds()

                response\_gaps.append(gap)

    avg\_response\_time = round(sum(response\_gaps)/len(response\_gaps),2) if response\_gaps else None

    avg\_style\_score = round(sum(style\_scores)/len(style\_scores),3) if style\_scores else 0.0

    suggestions=[]

    if avg\_style\_score < 0.4:

        suggestions.append("Improve clarity and structure in messages.")

    # Use lowercase for comparison

    if formality\_counts["informal"] > formality\_counts["formal"]:

        suggestions.append("Use more formal phrasing for formal tone.")

    if avg\_response\_time and avg\_response\_time > 86400: # 86400 seconds = 1 day

        suggestions.append("Respond to the messages more promptly to improve conversation engagement.")

    return{

        "Top Senders": dict(sender\_freq),

        "Average Response time (sec)":avg\_response\_time,

        "Suggestions": suggestions,

        # Include formality counts here for completeness if needed in behavioral report

        "Formality Counts": formality\_counts

    }

## **4.6: Report\_generator.py**

***This is the core back bone of our code, it handles all the modules together, use them one after the other to bring about a detailed report to the interface after getting the summary stat from the metrics\_calculator.py and based on it, a report and suggestion is automatically generated and saved to the data/reports folder***

"""

This module is the core orchestrator for the Communication Analysis Tool.

It handles reading various input message formats (single emails, multi-email files,

WhatsApp chat exports, and direct user input), parsing them, and then

coordinating the analysis by other modules (spam\_detector, sentiment\_analyzer,

style\_analyzer). Finally, it compiles the results, calculates aggregate metrics

using metrics\_calculator, and generates comprehensive reports both to a file

and as a summary printed to the command-line interface.

"""

import os

import datetime

import re # Import regular expression module for parsing headers

from modules.spam\_detector import SpamDetector

from modules.sentiment\_analyzer import SentimentAnalyzer

from modules.style\_analyzer import StyleAnalyzer

from modules.metrics\_calculator import (

    calculate\_content\_metrics,

    calculate\_engagement\_metrics

)

# Define the separator for multiple emails within a single file

EMAIL\_BOUNDARY\_SEPARATOR = "---EMAIL\_BOUNDARY---"

# --- Helper functions for parsing different message formats ---

def \_parse\_single\_email\_block(email\_block\_content):

    """

    Parses a single email's content block (headers + body).

    Returns a dictionary of headers and the email body.

    """

    headers = {

        "Sender": "Unknown",

        "Conversation ID": "N/A",

        "Timestamp": None,

        "Subject": "No Subject"

    }

    body = []

    header\_end\_found = False

    lines = email\_block\_content.splitlines()

    for line in lines:

        if not header\_end\_found and line.strip() == "":

            header\_end\_found = True

            continue

        if not header\_end\_found:

            line\_lower = line.lower()

            if line\_lower.startswith("from:"):

                headers["Sender"] = line[len("from:"):].strip()

            elif line\_lower.startswith("date:"):

                date\_str = line[len("date:"):].strip()

                try:

                    headers["Timestamp"] = datetime.datetime.strptime(date\_str, "%Y-%m-%d %H:%M:%S")

                except ValueError:

                    headers["Timestamp"] = date\_str # Keep as string if parsing fails

            elif line\_lower.startswith("conversation-id:"):

                headers["Conversation ID"] = line[len("conversation-id:"):].strip()

            elif line\_lower.startswith("subject:"):

                headers["Subject"] = line[len("subject:"):].strip()

        else:

            body.append(line)

    return headers, "\n".join(body).strip()

def \_parse\_multi\_email\_file(file\_content, filename):

    """

    Parses a file containing multiple emails separated by EMAIL\_BOUNDARY\_SEPARATOR.

    Returns a list of dictionaries, each representing a single parsed email.

    """

    parsed\_emails = []

    # Split the file content by the defined boundary

    email\_blocks = file\_content.split(EMAIL\_BOUNDARY\_SEPARATOR)

    for i, block in enumerate(email\_blocks):

        block = block.strip()

        if not block:

            continue # Skip empty blocks (e.g., if separator is at start/end or multiple in a row)

        try:

            headers, message\_body = \_parse\_single\_email\_block(block)

            parsed\_emails.append({

                "Source": filename,

                "Message ID": f"{filename}\_email\_{i+1}", # Unique ID for each email in the file

                "Sender": headers["Sender"],

                "Conversation ID": headers["Conversation ID"],

                "Timestamp": headers["Timestamp"], # Keep as datetime for sorting

                "Subject": headers["Subject"],

                "Message": message\_body,

            })

        except Exception as e:

            print(f"[X] Error parsing email block {i+1} in {filename}: {e}")

            parsed\_emails.append({

                "Source": filename,

                "Message ID": f"{filename}\_email\_{i+1}\_error",

                "Error": str(e),

                "Message": block[:200] # Store part of the block for context

            })

    return parsed\_emails

def \_parse\_whatsapp\_content(file\_content, filename):

    """

    Parses WhatsApp chat export content.

    Returns a list of dictionaries, each representing a single chat message.

    """

    messages = []

    whatsapp\_line\_pattern = re.compile(

        r"^(\d{1,2}/\d{1,2}/\d{2,4}),\s\*" # Date (MM/DD/YY or MM/DD/YYYY), comma, optional whitespace

        r"(\d{1,2}:\d{2}(?:\s\*(?i:am|pm))?)\s\*-\s\*" # Time (HH:MM optional AM/PM case-insensitive, with flexible whitespace)

        r"([^:]+?):\s\*(.\*)$" # Non-greedy sender name up to first colon, colon, optional whitespace, then message

    )

    conversation\_id = f"whatsapp\_chat\_{os.path.basename(filename).replace('.', '\_')}"

    for line\_num, line in enumerate(file\_content.splitlines(), 1):

        line = line.strip()

        if not line:

            continue

        match = whatsapp\_line\_pattern.match(line)

        if match:

            date\_part, time\_part, sender, message\_text = match.groups()

            timestamp\_str = f"{date\_part} {time\_part.strip()}"

            parsed\_timestamp = None

            try:

                parsed\_timestamp = datetime.datetime.strptime(timestamp\_str, "%m/%d/%y %I:%M %p")

            except ValueError:

                try:

                    parsed\_timestamp = datetime.datetime.strptime(timestamp\_str, "%m/%d/%Y %I:%M %p")

                except ValueError:

                    try:

                        parsed\_timestamp = datetime.datetime.strptime(timestamp\_str, "%m/%d/%y %H:%M")

                    except ValueError:

                        try:

                            parsed\_timestamp = datetime.datetime.strptime(timestamp\_str, "%m/%d/%Y %H:%M")

                        except ValueError:

                            pass # Keep as None if parsing fails

            messages.append({

                "Source": filename,

                "Message ID": f"{filename}\_line\_{line\_num}",

                "Sender": sender.strip(),

                "Conversation ID": conversation\_id,

                "Timestamp": parsed\_timestamp, # Store as datetime object

                "Subject": "WhatsApp Chat Message", # Default subject for chat messages

                "Message": message\_text.strip()

            })

        else:

            if messages:

                messages[-1]["Message"] += "\n" + line

            elif line\_num == 1: # This handles the initial system message if it doesn't match

                continue # Skip the first line if it's not a chat message

            else:

                messages.append({

                    "Source": filename,

                    "Message ID": f"{filename}\_line\_{line\_num}\_unformatted",

                    "Sender": "Unknown",

                    "Conversation ID": conversation\_id,

                    "Timestamp": None,

                    "Subject": "Unformatted Chat Line",

                    "Message": line

                })

    return messages

def \_is\_whatsapp\_format(first\_lines):

    """

    Heuristically checks if the file content looks like a WhatsApp export.

    Checks the first few lines for the typical WhatsApp timestamp/sender pattern.

    """

    whatsapp\_line\_pattern\_check = re.compile(

        r"^\d{1,2}/\d{1,2}/\d{2,4},\s\*" # Date

        r"\d{1,2}:\d{2}(?:\s\*(?i:am|pm))?\s\*-\s\*" # Time

        r"[^:]+?:\s\*.\*$" # Sender and start of message

    )

    for line in first\_lines:

        if whatsapp\_line\_pattern\_check.match(line.strip()):

            return True

    return False

def \_has\_email\_boundary(file\_content):

    """Checks if the file content contains the multi-email boundary separator."""

    return EMAIL\_BOUNDARY\_SEPARATOR in file\_content

# --- Common function to print summary to CLI ---

def \_print\_summary\_to\_cli(summary, behavior):

    """Prints a condensed summary of metrics and insights to the console."""

    print("\n" + "="\*30)

    print("  Analysis Summary (CLI)  ")

    print("="\*30)

    print("\n--- Summary Metrics ---")

    print(f"Total Messages        : {summary['Total Messages']}")

    print(f"Spam Breakdown        : SPAM = {summary['Spam Counts']['SPAM']}, HAM = {summary['Spam Counts']['HAM']}")

    print(f"Sentiment Breakdown   : {summary['Sentiment Counts']}")

    print(f"Average Style Score   : {summary['Average Style Score']}")

    print(f"Formality Breakdown   : {summary['Formality Counts']}")

    print("\n--- Behavioral Insights ---")

    print(f"Top Senders           : {behavior['Top Senders']}")

    if behavior.get("Average Response time (sec)") is not None:

        print(f"Avg Response Delay    : {behavior['Average Response time (sec)']} seconds")

    else:

        print("Avg Response Delay    : [Not enough data for response time calculation]")

    print("Suggestions           :")

    if behavior["Suggestions"]:

        for tip in behavior["Suggestions"]:

            print(f" - {tip}")

    else:

        print(" - No behavioral recommendations found.")

    print("="\*30 + "\n")

# --- Main report generation functions ---

def generate\_report(source="sample\_emails"):

    # Set path

    input\_items = []

    if source == "digitallogssample":

        input\_items = [

            ("digitallogssample/ham\_messages.txt", "HAM"),

            ("digitallogssample/spam\_messages.txt", "SPAM")

        ]

    elif source == "sample\_emails":

        input\_dir = "data/sample\_emails"

        if not os.path.exists(input\_dir):

            print(f"[X] Sample emails directory not found: {input\_dir}")

            return

        for filename in os.listdir(input\_dir):

            if filename.endswith(".txt"):

                input\_items.append(os.path.join(input\_dir, filename))

    else:

        print(f"[x] Unknown source: {source}")

        return

    timestamp = datetime.datetime.now().strftime("%Y-%m-%d\_%H-%M-%S")

    output\_path = f"data/reports/report\_{timestamp}.txt"

    os.makedirs(os.path.dirname(output\_path), exist\_ok=True)

    # Initializing modules

    spam\_detector = SpamDetector("data/training\_data/")

    sentiment\_analyzer = SentimentAnalyzer()

    style\_analyzer = StyleAnalyzer()

    results = [] # This will store dictionaries for each \*message\*

    for file\_path in input\_items:

        if isinstance(file\_path, tuple): # For digitallogssample format

            path\_to\_open = file\_path[0]

            try:

                with open(path\_to\_open, 'r', encoding='utf-8') as f:

                    for line\_num, line in enumerate(f, 1):

                        message\_body = line.strip()

                        if not message\_body: continue

                        spam\_status = "SPAM" if spam\_detector.predict(message\_body) else "HAM"

                        sentiment = sentiment\_analyzer.analyze(message\_body)

                        style = style\_analyzer.analyze(message\_body)

                        results.append({

                            "Source": os.path.basename(path\_to\_open),

                            "Message ID": f"{os.path.basename(path\_to\_open)}\_line\_{line\_num}",

                            "Message": message\_body,

                            "Spam": spam\_status,

                            "Sentiment": sentiment,

                            "Style Score": style["Style Score"],

                            "Formality": style["Formality"],

                            "Sender": "Unknown", # Default for these files

                            "Conversation ID": "N/A", # Default

                            "Timestamp": datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S") # Default to current time

                        })

            except Exception as e:

                print(f"[X] Error processing file {os.path.basename(path\_to\_open)}: {e}")

                results.append({

                    "Source": os.path.basename(path\_to\_open),

                    "Error": str(e),

                    "Message": "Could not read file."

                })

        else: # For sample\_emails directory (which might contain emails or WhatsApp or multi-emails)

            try:

                with open(file\_path, 'r', encoding='utf-8') as f:

                    full\_content = f.read()

                first\_few\_lines = full\_content.splitlines()[:5] # Check first few lines for format

                parsed\_messages = []

                if \_is\_whatsapp\_format(first\_few\_lines):

                    print(f"[!] Detected WhatsApp format for {os.path.basename(file\_path)}.")

                    parsed\_messages = \_parse\_whatsapp\_content(full\_content, os.path.basename(file\_path))

                elif \_has\_email\_boundary(full\_content): # Check for multi-email boundary

                    print(f"[!] Detected multi-email format for {os.path.basename(file\_path)}.")

                    parsed\_messages = \_parse\_multi\_email\_file(full\_content, os.path.basename(file\_path))

                else:

                    # Assume single standard email format if not WhatsApp or multi-email

                    headers, message\_body = \_parse\_single\_email\_block(full\_content)

                    parsed\_messages.append({

                        "Source": os.path.basename(file\_path),

                        "Message ID": f"{os.path.basename(file\_path)}\_msg",

                        "Sender": headers["Sender"],

                        "Conversation ID": headers["Conversation ID"],

                        "Timestamp": headers["Timestamp"], # Keep as datetime for sorting

                        "Subject": headers["Subject"],

                        "Message": message\_body,

                    })

                for msg\_data in parsed\_messages:

                    message\_to\_analyze = msg\_data.get("Message", "")

                    if not message\_to\_analyze.strip(): # Skip empty messages

                        continue

                    spam\_status = "SPAM" if spam\_detector.predict(message\_to\_analyze) else "HAM"

                    sentiment = sentiment\_analyzer.analyze(message\_to\_analyze)

                    style = style\_analyzer.analyze(message\_to\_analyze)

                    # Update the message data with analysis results

                    msg\_data.update({

                        "Spam": spam\_status,

                        "Sentiment": sentiment,

                        "Style Score": style["Style Score"],

                        "Formality": style["Formality"]

                    })

                    results.append(msg\_data)

            except Exception as e:

                print(f"[X] Error processing file {os.path.basename(file\_path)}: {e}")

                results.append({

                    "Source": os.path.basename(file\_path),

                    "Error": str(e),

                    "Message": full\_content # Store full content for error context

                })

    # Convert datetime objects to string format for metrics\_calculator if needed

    for entry in results:

        if isinstance(entry.get("Timestamp"), datetime.datetime):

            entry["Timestamp"] = entry["Timestamp"].strftime("%Y-%m-%d %H:%M:%S")

    # Run metrics after message parsing

    summary = calculate\_content\_metrics(results)

    behavior = calculate\_engagement\_metrics(results)

    # Write full analysis report

    with open(output\_path, 'w', encoding='utf-8') as out:

        out.write("Communication Analysis Report\n\n")

        out.write("--- Individual Message Analysis ---\n\n")

        for entry in results:

            out.write(f"File: {entry.get('Source', '[unknown]')}\n")

            if "Error" in entry:

                out.write(f" Error: {entry['Error']}\n")

                out.write(f" Original Content: {entry.get('Message', '')[:200]}...\n")

            else:

                out.write(f" Subject: {entry.get('Subject', 'N/A')}\n")

                out.write(f" Sender: {entry.get('Sender', 'N/A')}\n")

                out.write(f" Conversation ID: {entry.get('Conversation ID', 'N/A')}\n")

                out.write(f" Timestamp: {entry.get('Timestamp', 'N/A')}\n")

                out.write(f" Message Body Preview: {entry['Message'][:100]}...\n")

                out.write(f"   Spam      : {entry['Spam']}\n")

                out.write(f"   Sentiment : {entry['Sentiment']}\n")

                out.write(f"   Style     : {entry['Style Score']} ({entry['Formality']})\n")

            out.write("-" \* 50 + "\n")

        # Append Summary Metrics

        out.write("\n--- Summary Metrics ---\n\n")

        out.write(f"Total Messages        : {summary['Total Messages']}\n")

        out.write(f"Spam Breakdown        : SPAM = {summary['Spam Counts']['SPAM']}, HAM = {summary['Spam Counts']['HAM']}\n")

        out.write(f"Sentiment Breakdown   : {summary['Sentiment Counts']}\n")

        out.write(f"Average Style Score   : {summary['Average Style Score']}\n")

        out.write(f"Formality Breakdown   : {summary['Formality Counts']}\n")

        # Append Behavioral Insights

        out.write("\n--- Behavioral Insights ---\n\n")

        out.write(f"Top Senders           : {behavior['Top Senders']}\n")

        if behavior.get("Average Response time (sec)") is not None:

            out.write(f"Avg Response Delay    : {behavior['Average Response time (sec)']} seconds\n")

        else:

            out.write("Avg Response Delay    : [Not enough data for response time calculation]\n")

        out.write("Suggestions           :\n")

        if behavior["Suggestions"]:

            for tip in behavior["Suggestions"]:

                out.write(f" - {tip}\n")

        else:

            out.write(" - No behavioral recommendations found.\n")

    print(f"[] Report saved to: {output\_path}")

    \_print\_summary\_to\_cli(summary, behavior) # Call the new function to print to CLI

# Added message\_type parameter with a default

def generate\_custom\_test\_report(

    file\_path,

    output\_path="data/reports/test\_analysis\_report.txt",

    spam\_detector=None,

    sentiment\_analyzer=None,

    style\_analyzer=None

):

    # Initialize analyzers

    spam\_detector = spam\_detector or SpamDetector("data/training\_data/")

    sentiment\_analyzer = sentiment\_analyzer or SentimentAnalyzer()

    style\_analyzer = style\_analyzer or StyleAnalyzer()

    results = []

    if not os.path.exists(file\_path):

        print(f"[X] Input file not found: {file\_path}")

        return

    try:

        with open(file\_path, 'r', encoding='utf-8') as f:

            full\_content = f.read()

        first\_few\_lines = full\_content.splitlines()[:5]

        parsed\_messages = []

        if \_is\_whatsapp\_format(first\_few\_lines):

            print(f"[!] Detected WhatsApp format for {os.path.basename(file\_path)}.")

            parsed\_messages = \_parse\_whatsapp\_content(full\_content, os.path.basename(file\_path))

        elif \_has\_email\_boundary(full\_content): # Check for multi-email boundary

            print(f"[!] Detected multi-email format for {os.path.basename(file\_path)}.")

            parsed\_messages = \_parse\_multi\_email\_file(full\_content, os.path.basename(file\_path))

        else:

            # Assume single standard email format if not WhatsApp or multi-email

            headers, message\_body = \_parse\_single\_email\_block(full\_content)

            parsed\_messages.append({

                "Source": os.path.basename(file\_path),

                "Message ID": f"{os.path.basename(file\_path)}\_msg",

                "Sender": headers["Sender"],

                "Conversation ID": headers["Conversation ID"],

                "Timestamp": headers["Timestamp"],

                "Subject": headers["Subject"],

                "Message": message\_body,

            })

        for msg\_data in parsed\_messages:

            message\_to\_analyze = msg\_data.get("Message", "")

            if not message\_to\_analyze.strip():

                continue

            spam\_status = "SPAM" if spam\_detector.predict(message\_to\_analyze) else "HAM"

            sentiment = sentiment\_analyzer.analyze(message\_to\_analyze)

            style = style\_analyzer.analyze(message\_to\_analyze)

            msg\_data.update({

                "Spam": spam\_status,

                "Sentiment": sentiment,

                "Style Score": style["Style Score"],

                "Formality": style["Formality"]

            })

            results.append(msg\_data)

    except Exception as e:

        print(f"[X] Error analyzing file {os.path.basename(file\_path)}: {e}")

        results.append({

            "Source": os.path.basename(file\_path),

            "Error": str(e),

            "Message": full\_content

        })

    # Convert datetime objects to string format for metrics\_calculator if needed

    for entry in results:

        if isinstance(entry.get("Timestamp"), datetime.datetime):

            entry["Timestamp"] = entry["Timestamp"].strftime("%Y-%m-%d %H:%M:%S")

    # Run Metrics

    summary = calculate\_content\_metrics(results)

    behavior = calculate\_engagement\_metrics(results)

    # Write Full Report

    os.makedirs(os.path.dirname(output\_path), exist\_ok=True)

    with open(output\_path, 'w', encoding='utf-8') as out:

        out.write("Custom Test Report\n\n")

        out.write("--- Individual Message Analysis ---\n\n")

        for entry in results:

            out.write(f"File: {entry.get('Source', '[unknown]')}\n")

            if "Error" in entry:

                out.write(f" Error: {entry['Error']}\n")

                out.write(f" Original Content: {entry.get('Message', '')[:200]}...\n")

            else:

                out.write(f" Subject: {entry.get('Subject', 'N/A')}\n")

                out.write(f" Sender: {entry.get('Sender', 'N/A')}\n")

                out.write(f" Conversation ID: {entry.get('Conversation ID', 'N/A')}\n")

                out.write(f" Timestamp: {entry.get('Timestamp', 'N/A')}\n")

                out.write(f" Message Body Preview: {entry['Message'][:100]}...\n")

                out.write(f" Spam: {entry['Spam']}\n")

                out.write(f" Sentiment: {entry['Sentiment']}\n")

                out.write(f" Style: {entry['Style Score']} ({entry['Formality']})\n")

            out.write("-" \* 50 + "\n")

        out.write("\n--- Summary Metrics ---\n\n")

        out.write(f"Total Messages        : {summary['Total Messages']}\n")

        out.write(f"Spam Breakdown        : SPAM = {summary['Spam Counts']['SPAM']}, HAM = {summary['Spam Counts']['HAM']}\n")

        out.write(f"Sentiment Breakdown   : {summary['Sentiment Counts']}\n")

        out.write(f"Average Style Score   : {summary['Average Style Score']}\n")

        out.write(f"Formality Breakdown   : {summary['Formality Counts']}\n")

        out.write("\n--- Behavioral Insights ---\n\n")

        out.write(f"Top Senders           : {behavior['Top Senders']}\n")

        if behavior.get("Average Response time (sec)") is not None:

            out.write(f"Avg Response Delay    : {behavior['Average Response time (sec)']} seconds\n")

        else:

            out.write("Avg Response Delay    : [Not enough data for response time calculation]\n")

        out.write("Suggestions           :\n")

        if behavior["Suggestions"]:

            for tip in behavior["Suggestions"]:

                out.write(f" - {tip}\n")

        else:

            out.write(" - No behavioral recommendations found.\n")

    print(f" Custom test report saved to: {output\_path}")

    \_print\_summary\_to\_cli(summary, behavior) # Call the new function to print to CLI

# Modified to accept message\_type parameter

def generate\_report\_from\_custom\_input(message, sender=None, conversation\_id=None, message\_type="Other"):

    timestamp = datetime.datetime.now().strftime("%Y-%m-%d\_%H-%M-%S")

    output\_path = f"data/reports/report\_{timestamp}.txt"

    os.makedirs(os.path.dirname(output\_path), exist\_ok=True)

    spam\_detector = SpamDetector("data/training\_data/")

    sentiment\_analyzer = SentimentAnalyzer()

    style\_analyzer = StyleAnalyzer()

    results = [] # Prepare a list to hold the single result for consistency

    # Determine metadata based on message\_type and user input

    final\_sender = sender

    final\_convo\_id = conversation\_id

    final\_subject = "Custom Message"

    current\_timestamp = datetime.datetime.now()

    parsed\_message\_content = message # Default to raw message

    if message\_type == "WhatsApp":

        # Attempt to parse as a single WhatsApp line

        # We'll create a dummy file content for \_parse\_whatsapp\_content

        dummy\_filename = "typed\_whatsapp\_msg.txt"

        # The parser expects a file-like structure, so we simulate it.

        # It will return a list, we take the first item if successful.

        temp\_parsed = \_parse\_whatsapp\_content(message, dummy\_filename)

        if temp\_parsed:

            parsed\_data = temp\_parsed[0] # Take the first parsed message

            final\_sender = parsed\_data.get("Sender", sender if sender else "WhatsApp User")

            final\_convo\_id = parsed\_data.get("Conversation ID", conversation\_id if conversation\_id else f"typed\_whatsapp\_convo\_{timestamp}")

            current\_timestamp = parsed\_data.get("Timestamp", current\_timestamp)

            final\_subject = parsed\_data.get("Subject", "WhatsApp Chat Message")

            parsed\_message\_content = parsed\_data.get("Message", message) # Use parsed body

            print(f"[!] Parsed as WhatsApp: Sender='{final\_sender}', Convo ID='{final\_convo\_id}'")

        else:

            print("[X] Could not parse as WhatsApp format. Using generic defaults.")

            final\_sender = sender if sender else "WhatsApp User"

            final\_convo\_id = conversation\_id if conversation\_id else f"typed\_whatsapp\_convo\_{timestamp}"

            final\_subject = "WhatsApp Message (Unparsed)"

    elif message\_type == "Email":

        # Attempt to parse as a single email block

        temp\_headers, temp\_body = \_parse\_single\_email\_block(message)

        if temp\_headers.get("Sender") != "Unknown" or temp\_headers.get("Subject") != "No Subject":

            final\_sender = temp\_headers.get("Sender", sender if sender else "Email Sender")

            final\_convo\_id = temp\_headers.get("Conversation ID", conversation\_id if conversation\_id else f"typed\_email\_convo\_{timestamp}")

            current\_timestamp = temp\_headers.get("Timestamp", current\_timestamp)

            final\_subject = temp\_headers.get("Subject", "Email Message")

            parsed\_message\_content = temp\_body # Use parsed body

            print(f"[!] Parsed as Email: Sender='{final\_sender}', Subject='{final\_subject}'")

        else:

            print("[X] Could not parse as Email format. Using generic defaults.")

            final\_sender = sender if sender else "Email Sender"

            final\_convo\_id = conversation\_id if conversation\_id else f"typed\_email\_convo\_{timestamp}"

            final\_subject = "Email Message (Unparsed)"

    elif message\_type == "SMS":

        final\_sender = sender if sender else "SMS Sender"

        final\_convo\_id = conversation\_id if conversation\_id else f"typed\_sms\_convo\_{timestamp}"

        final\_subject = "SMS Message"

    else: # "Other" or fallback

        final\_sender = sender if sender else "User Input"

        final\_convo\_id = conversation\_id if conversation\_id else f"manual\_convo\_{timestamp}"

        final\_subject = "Custom Message"

    try:

        spam\_status = "SPAM" if spam\_detector.predict(parsed\_message\_content) else "HAM"

        sentiment = sentiment\_analyzer.analyze(parsed\_message\_content)

        style = style\_analyzer.analyze(parsed\_message\_content)

        result = {

            "Source": f"manual\_input\_{message\_type.lower()}",

            "Message ID": "manual\_msg\_1", # Still 1 message per analysis session

            "Sender": final\_sender,

            "Conversation ID": final\_convo\_id,

            "Timestamp": current\_timestamp.strftime("%Y-%m-%d %H:%M:%S") if isinstance(current\_timestamp, datetime.datetime) else current\_timestamp,

            "Subject": final\_subject,

            "Message": parsed\_message\_content,

            "Spam": spam\_status,

            "Sentiment": sentiment,

            "Style Score": style["Style Score"],

            "Formality": style["Formality"]

        }

        results.append(result)

        # Run metrics even for single message, though behavioral will be limited

        summary = calculate\_content\_metrics(results)

        behavior = calculate\_engagement\_metrics(results)

        with open(output\_path, 'w', encoding='utf-8') as out:

            out.write(f"Single {message\_type} Message Analysis Report\n\n")

            out.write("--- Individual Message Analysis ---\n\n")

            out.write(f" Source: {result['Source']}\n")

            out.write(f" Subject: {result['Subject']}\n")

            out.write(f" Sender: {result['Sender']}\n")

            out.write(f" Conversation ID: {result['Conversation ID']}\n")

            out.write(f" Timestamp: {result['Timestamp']}\n")

            out.write(f" Message Body: {result['Message']}\n")

            out.write(f" Spam: {result['Spam']}\n")

            out.write(f" Sentiment: {result['Sentiment']}\n")

            out.write(f" Style: {result['Style Score']} ({result['Formality']})\n")

            out.write("-" \* 50 + "\n")

            out.write("\n--- Summary Metrics ---\n\n")

            out.write(f"Total Messages        : {summary['Total Messages']}\n")

            out.write(f"Spam Breakdown        : SPAM = {summary['Spam Counts']['SPAM']}, HAM = {summary['Spam Counts']['HAM']}\n")

            out.write(f"Sentiment Breakdown   : {summary['Sentiment Counts']}\n")

            out.write(f"Average Style Score   : {summary['Average Style Score']}\n")

            out.write(f"Formality Breakdown   : {summary['Formality Counts']}\n")

            out.write("\n--- Behavioral Insights (Limited for single message) ---\n\n")

            out.write(f"Top Senders           : {behavior['Top Senders']}\n")

            if behavior.get("Average Response time (sec)") is not None:

                out.write(f"Avg Response Delay    : {behavior['Average Response time (sec)']} seconds\n")

            else:

                out.write("Avg Response Delay    : [Not enough data for response time calculation]\n")

            out.write("Suggestions           :\n")

            if behavior["Suggestions"]:

                for tip in behavior["Suggestions"]:

                    out.write(f" - {tip}\n")

            else:

                out.write(" - No behavioral recommendations found.\n")

        print(f" Manual input report saved to: {output\_path}")

        \_print\_summary\_to\_cli(summary, behavior) # Call the new function to print to CLI

    except Exception as e:

        print(f"[X] Error analyzing message: {e}")

## **4.7: Main.py (CLI Controller)**

***The main command line interface source code, to combine the modules into one and it is the “Power switch” of all the modules for them to run together and specifically switches on the report generator then causing a butterfly effect for all the analysis in the modules to be triggered and be used at that particular point in time***

"""

This is the main command-line interface (CLI) for the Communication Analysis Tool.

It provides a menu-driven system for users to:

1. Run a full analysis on a predefined sample dataset.

2. Analyze a custom text file (either from samples or a user-provided path).

3. Type and analyze a custom message, with options for message type (WhatsApp, Email, SMS, Other),   and flexible input for sender/conversation ID, including "back" and "cancel" options for user control.

4. Exit the application.

The CLI orchestrates calls to various modules in the 'modules/' directory,

primarily 'report\_generator.py', to perform the analysis and generate reports.

"""

from modules.spam\_detector import SpamDetector

from modules.style\_analyzer import StyleAnalyzer

from modules.sentiment\_analyzer import SentimentAnalyzer

from modules.report\_generator import (

    generate\_custom\_test\_report,

    generate\_report,

    generate\_report\_from\_custom\_input

)

import os

import time

def list\_sample\_files(directory="data/sample\_emails/"):

    """

    Lists sample files in the specified directory, returning a list of

    (display\_name, full\_path) tuples.

    """

    try:

        files = os.listdir(directory)

        if not files:

            print(f"[X] No sample files found in '{directory}'.")

            return []

        # Store files with their full relative paths for selection

        sample\_file\_paths = []

        print("\nAvailable sample files:")

        for i, f in enumerate(files):

            full\_path = os.path.join(directory, f)

            sample\_file\_paths.append((f, full\_path)) # Store display name and full path

            print(f" {i + 1}. {f}") # Display with number and just the filename

        return sample\_file\_paths

    except FileNotFoundError:

        print(f"[X] Sample directory '{directory}' not found.")

        return []

def print\_menu():

    print("\nCommunication Analysis CLI")

    print("1. Run full analysis on sample dataset")

    print("2. Analyze a custom text file")

    print("3. Type and analyze a custom message")

    print("4. Exit")

def print\_message\_type\_menu():

    print("\nSelect Message Type:")

    print("  1. WhatsApp Chat Message")

    print("  2. Email Message (with optional headers)")

    print("  3. SMS Message")

    print("  4. Other / Generic Text")

    print("  5. Back to Main Menu")

def main():

    while True:

        print\_menu()

        choice = input("Enter your choice (1-4): ").strip()

        if choice == '1':

            print("\n[!] Running full analysis on sample dataset...")

            generate\_report("sample\_emails")

        elif choice == '2':

            print("\nListing available sample files...")

            available\_files = list\_sample\_files()

            if not available\_files:

                print("[X] No files to select from. Please add files to the sample directory or provide a full path.")

                continue

            print(" 0. Enter a custom file path manually")

            file\_selection\_input = input("\nEnter the number of the file to analyze, or '0' for a custom path: ").strip()

            selected\_file\_path = None

            try:

                selection\_number = int(file\_selection\_input)

                if selection\_number == 0:

                    selected\_file\_path = input("Please enter the full path to your custom text file: ").strip()

                elif 1 <= selection\_number <= len(available\_files):

                    selected\_file\_path = available\_files[selection\_number - 1][1]

                else:

                    print("[X] Invalid number. Please enter a number from the list or '0'.")

                    continue

            except ValueError:

                selected\_file\_path = file\_selection\_input

            if not selected\_file\_path:

                print("[X] No file path provided.")

                continue

            if not os.path.exists(selected\_file\_path):

                print(f"[X] File not found at '{selected\_file\_path}'. Please check the path and try again.")

                continue

            timestamp = time.strftime("%Y-%m-%d\_%H-%M-%S")

            output\_name = f"data/reports/custom\_report\_{timestamp}.txt"

            generate\_custom\_test\_report(selected\_file\_path, output\_path=output\_name)

        elif choice == '3':

            while True: # Loop for selecting message type

                print\_message\_type\_menu()

                type\_choice = input("Enter message type choice (1-5): ").strip()

                message\_type = "Other" # Default type

                if type\_choice == '1':

                    message\_type = "WhatsApp"

                elif type\_choice == '2':

                    message\_type = "Email"

                elif type\_choice == '3':

                    message\_type = "SMS"

                elif type\_choice == '4':

                    message\_type = "Other"

                elif type\_choice == '5':

                    break # Go back to main menu

                else:

                    print("[X] Invalid message type option. Try again.")

                    continue

                print(f"\n--- Analyzing Custom {message\_type} Message ---")

                # Prompt for sender name with a back option

                sender\_input = input("Enter sender name (optional, leave blank for default, type 'back' to change type): ").strip()

                if sender\_input.lower() == 'back':

                    print("Going back to message type selection.")

                    continue # Restart the message type loop

                # Prompt for conversation ID with a back option

                convo\_id\_input = input("Enter conversation ID (optional, leave blank for unique ID, type 'back' to change type): ").strip()

                if convo\_id\_input.lower() == 'back':

                    print("Going back to message type selection.")

                    continue # Restart the message type loop

                print("\nType or paste your message below.")

                print("Press [Enter] on an empty line to finish input, or type 'cancel' on a new line to go back:")

                lines = []

                while True:

                    line = input()

                    if line.strip().lower() == "cancel": # Check for cancel keyword

                        print("Message input cancelled. Going back to message type selection.")

                        lines = [] # Clear any partial input

                        break # Exit inner input loop

                    if line.strip() == "":

                        break # Finish input

                    lines.append(line)

                user\_input = "\n".join(lines).strip()

                if not user\_input: # This covers empty input or if 'cancel' was typed

                    if lines: # If 'cancel' was typed, lines list might not be empty, but user\_input is empty.

                              # This check ensures we truly cancel if 'cancel' was the last thing.

                        print("[X] No input received or message cancelled.")

                    continue # Go back to message type selection

                # Pass the optional sender, convo ID, and the new message\_type

                generate\_report\_from\_custom\_input(

                    user\_input,

                    sender=sender\_input if sender\_input else None,

                    conversation\_id=convo\_id\_input if convo\_id\_input else None,

                    message\_type=message\_type

                )

                # After analysis, give option to analyze another custom message or go back

                another\_message = input("\nAnalyze another custom message? (y/n): ").strip().lower()

                if another\_message != 'y':

                    break # Go back to main menu

        elif choice == '4':

            print("[!] Exiting CLI. Goodbye!")

            break

        else:

            print("[X] Invalid option. Try again.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

# **Testing and Evaluation**

Testing was an essential part of the development process to ensure all modules performed reliably and accurately. Both unit and integrating testing approaches were used to validate functionality.

This section documents the strategy used for testing, the environment in which the tests where conducted, and the actual samples obtained from the test run, it includes the sample output of the tests, the test scripts and reports generated. The goal of this testing phase was to catch logical errors, validate code robustness and to verify the correctness of all core functions before running the program.

Each functional component of the Communication Analysis Tool was treated as an independent module and tested separately using unit tests, while the entire workflow was verified using CLI-based integration testing.

The primary objectives were:

* To verify that each module performs its function accurately
* To ensure seamless integration between modules
* To validate outputs using sample data for consistency and correctness

All test scripts and data samples are located in the tests/group16(directory) and sample\_emails/data/group16 respectively and in the appendix section of this documentation, and testing covered both functionality and performance under real-world message inputs.

## **5.1. Unit testing strategy**

Unit tests were written for each module to ensure that internal logic works correctly. These tests are found in the tests/ folder and are named according to the module being tested.

The sample: test\_data\_parser.py

from modules.data\_parser import parse\_spam\_file, parse\_ham\_file

def test\_parse\_spam\_file():

result = parse\_spam\_file("data/digitallogsample/spam\_messages.txt")

assert isinstance(result, list)

assert all(isinstance(msg, str) for msg in result)

def test\_parse\_ham\_file():

result = parse\_ham\_file("data/digitallogsample/ham\_messages.txt")

assert isinstance(result, list)

assert all(isinstance(msg, str) for msg in result)

This test confirms that the data parsing functions:

* Correctly return lists
* Contain only string-type entries
* Handle file loading without failure

Additional tests were written similarly for:

* spam\_detector.py
* sentiment\_analyzer.py
* style\_analyzer.py
* report\_generator.py

Each test was designed to be lightweight, fast, and cover common edge cases.

## **5.2 CLI Testing**

A complete pipeline test was performed using the test\_main.py script located at the root of the project. This test simulates user input and verifies whether the integrated modules work together as expected.

"""

Integration/Unit tests for the main.py CLI.

This module tests the overall flow of the command-line interface,

ensuring that user input correctly triggers the intended functions

from other modules. It uses mocking to simulate user input and

to prevent actual file operations or external module calls during tests.

"""

import unittest

from unittest.mock import patch, MagicMock

import os

import sys

from io import StringIO

import main

class TestMainCLI(unittest.TestCase):

def setUp(self):

self.mock\_generate\_report = patch('main.generate\_report').start()

self.mock\_generate\_custom\_test\_report = patch('main.generate\_custom\_test\_report').start()

self.mock\_generate\_report\_from\_custom\_input = patch('main.generate\_report\_from\_custom\_input').start()

self.mock\_list\_sample\_files = patch('main.list\_sample\_files').start()

self.mock\_list\_sample\_files.return\_value = [

("test\_file\_1.txt", "/path/to/test\_file\_1.txt"),

("test\_file\_2.txt", "/path/to/test\_file\_2.txt")

]

self.held\_stdout = sys.stdout

sys.stdout = StringIO()

self.addCleanup(patch.stopall)

self.addCleanup(self.\_restore\_stdout)

def \_restore\_stdout(self):

sys.stdout = self.held\_stdout

def \_get\_stdout(self):

return sys.stdout.getvalue()

@patch('builtins.input', side\_effect=['1', '4'])

def test\_option\_1\_full\_analysis(self, mock\_input):

main.main()

self.mock\_generate\_report.assert\_called\_once\_with("sample\_emails")

self.assertIn("Running full analysis on sample dataset", self.\_get\_stdout())

@patch('builtins.input', side\_effect=['2', '1', '4'])

@patch('main.os.path.exists', return\_value=True)

def test\_option\_2\_analyze\_sample\_file(self, mock\_exists, mock\_input):

main.main()

self.mock\_generate\_custom\_test\_report.assert\_called\_once()

self.assertEqual(self.mock\_generate\_custom\_test\_report.call\_args[0][0], "/path/to/test\_file\_1.txt")

self.assertIn("Listing available sample files", self.\_get\_stdout())

@patch('builtins.input', side\_effect=['2', '0', '/custom/path/my\_file.txt', '4'])

@patch('main.os.path.exists', return\_value=True)

def test\_option\_2\_analyze\_custom\_path(self, mock\_exists, mock\_input):

main.main()

self.mock\_generate\_custom\_test\_report.assert\_called\_once()

self.assertEqual(self.mock\_generate\_custom\_test\_report.call\_args[0][0], "/custom/path/my\_file.txt")

@patch('builtins.input', side\_effect=['3', '1', 'Alice', 'convo1', 'Hello!', '', 'n', '4'])

def test\_option\_3\_whatsapp\_message(self, mock\_input):

main.main()

self.mock\_generate\_report\_from\_custom\_input.assert\_called\_once\_with(

'Hello!', sender='Alice', conversation\_id='convo1', message\_type='WhatsApp'

)

self.assertIn("Analyzing Custom WhatsApp Message", self.\_get\_stdout())

@patch('builtins.input', side\_effect=['3', '2', '', '', 'Subject: Test\n\nBody', '', 'n', '4'])

def test\_option\_3\_email\_message(self, mock\_input):

main.main()

self.mock\_generate\_report\_from\_custom\_input.assert\_called\_once\_with(

'Subject: Test\n\nBody', sender=None, conversation\_id=None, message\_type='Email'

)

self.assertIn("Analyzing Custom Email Message", self.\_get\_stdout())

@patch('builtins.input', side\_effect=['3', '1', 'back', '5', '4']) # Added '4'

def test\_option\_3\_back\_from\_sender(self, mock\_input):

main.main()

self.mock\_generate\_report\_from\_custom\_input.assert\_not\_called()

self.assertIn("Going back to message type selection", self.\_get\_stdout())

self.assertIn("Select Message Type", self.\_get\_stdout())

@patch('builtins.input', side\_effect=['3', '1', 'Alice', 'back', '5', '4']) # Added '4'

def test\_option\_3\_back\_from\_convo\_id(self, mock\_input):

main.main()

self.mock\_generate\_report\_from\_custom\_input.assert\_not\_called()

self.assertIn("Going back to message type selection", self.\_get\_stdout())

self.assertIn("Select Message Type", self.\_get\_stdout())

@patch('builtins.input', side\_effect=['3', '1', 'Alice', 'convo1', 'message line 1', 'cancel', '5', '4']) # Added '4'

def test\_option\_3\_cancel\_message\_input(self, mock\_input):

main.main()

self.mock\_generate\_report\_from\_custom\_input.assert\_not\_called()

self.assertIn("Message input cancelled", self.\_get\_stdout())

self.assertIn("Select Message Type", self.\_get\_stdout())

@patch('builtins.input', side\_effect=['4'])

def test\_option\_4\_exit(self, mock\_input):

main.main()

self.assertIn("Exiting CLI. Goodbye!", self.\_get\_stdout())

@patch('builtins.input', side\_effect=['99', '4']) # Added '4'

def test\_invalid\_main\_menu\_choice(self, mock\_input):

main.main()

self.assertIn("Invalid option. Try again.", self.\_get\_stdout())

self.assertGreaterEqual(self.\_get\_stdout().count("Communication Analysis CLI"), 2)

@patch('builtins.input', side\_effect=['3', '99', '5', '4']) # Added '4'

def test\_invalid\_message\_type\_choice(self, mock\_input):

main.main()

self.assertIn("Invalid message type option. Try again.", self.\_get\_stdout())

self.assertGreaterEqual(self.\_get\_stdout().count("Select Message Type"), 2)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

This allowed us to manually observe the classification result, style breakdown, and sentiment tone in real time, confirming functional correctness in an integrated environment.

## **5.4 Sample Results and Evaluation Summary**

We ran the tool using real messages from the data/sample\_emails and Here are some actual results observed during testing of the unit testing, test CLI, main CLI and the results summary came out as:

* 1. Unit testing:

......................

----------------------------------------------------------------------

Ran 35 tests in 0.854s

OK

* 1. Test run CLI:

Custom Test Report

--- Individual Message Analysis ---

File: single\_email.txt

Subject: Test Email

Sender: test@example.com

Conversation ID: email\_test\_001

Timestamp: 2025-01-01 12:00:00

Message Body Preview: This is a test email body....

Spam: HAM

Sentiment: neutral

Style: 50 (informal)

--------------------------------------------------

--- Summary Metrics ---

Total Messages : 1

Spam Breakdown : SPAM = 0, HAM = 1

Sentiment Breakdown : {'positive': 0, 'neutral': 1, 'negative': 0}

Average Style Score : 50.0

Formality Breakdown : {'formal': 0, 'informal': 1}

--- Behavioral Insights ---

Top Senders : {'TestSender': 1}

Avg Response Delay : [Not enough data for response time calculation]

Suggestions :

- No behavioral recommendations found.

3. Main CLI:

==============================

Analysis Summary (CLI)

==============================

--- Summary Metrics ---

Total Messages : 2

Spam Breakdown : SPAM = 0, HAM = 2

Sentiment Breakdown : {'positive': 0, 'neutral': 2, 'negative': 0}

Average Style Score : 74.0

Formality Breakdown : {'formal': 0, 'informal': 1}

--- Behavioral Insights ---

Top Senders : {'sarah.davis@example.com': 1, 'mark.wilson@example.com': 1}

Avg Response Delay : 5400.0 seconds

Suggestions :

- Use more formal phrasing for formal tone.

==============================

These results show that:

* The spam detector is able to differentiate promotional content from normal communication.
* Style Analyzer identifies characteristics like tone, punctuation, and structure.
* Sentiment Analyzer reflects emotional cues accurately.

Overall, the tool passed all functional and integration tests with strong, reliable performance across diverse message samples.

The list of the codes that underwent unit testing, the email log samples used and the words for training the spam detector can be found at the end of the document in the Appendix Section.

1. **Communication Analysis Tool: User Manual for Non-Programmers**

This simple program helps you understand your messages better. It can tell you if a message looks like spam, what its general feeling is (happy, sad, or neutral), how formal or informal it sounds, and even give you insights into how people communicate in a conversation.

This manual will guide you through how to use the tool, step by step. You don't need to know anything about computer programming to use this!

Getting Started

First, you need to open your computer's "Command Prompt" (on Windows) or "Terminal" (on Mac/Linux).

1. **Find your project folder:** Navigate to the folder where you saved all the program files. For example, if your folder is on your Desktop named "Group 16", you might type:
2. cd C:\Users\YourName\Desktop\Group 16 #On Windows
3. # or
4. cd ~/Desktop/Group\ 16/ #On Mac/Linux
5. **Start the program:** Once you are in the correct folder, type this command and press Enter:
6. python main.py

You will then see the main menu:

Communication Analysis CLI

1. Run full analysis on sample dataset

2. Analyze a custom text file

3. Type and analyze a custom message

4. Exit

2. How to Use Each Option

**6.1. Run full analysis on sample dataset**

This option is great if you have a collection of message files (like emails or WhatsApp chats) that you want to analyze all at once. The program will look at every .txt file in the data/sample\_emails/ folder.

* **What to do:**
  1. Type 1 and press Enter.
* **What happens:**
  1. The program will go through all your sample files.
  2. It will create a detailed report file in the data/reports/ folder. This file will have a name like report\_YYYY-MM-DD\_HH-MM-SS.txt.
  3. After the report is saved, a quick summary of the main findings will appear right in your command prompt/terminal window.

**6.2. Analyze a custom text file**

Use this option if you want to analyze just one specific message file.

* **What to do:**
  1. Type 2 and press Enter.
  2. The program will show you a list of files already in your data/sample\_emails/ folder, each with a number next to it.
  3. **To pick a file from the list:** Type the number next to the file you want to analyze and press Enter.
  4. **To analyze a file not on the list (from anywhere on your computer):** Type 0 and press Enter. Then, the program will ask you to type the full path to your file (e.g., C:\MyDocuments\my\_special\_email.txt). Type the path and press Enter.
* **What happens:**
  1. The program will analyze that single file.
  2. It will create a detailed report file in the data/reports/ folder.
  3. A quick summary will also be printed in your command prompt/terminal.

**6.3. Type and analyze a custom message**

This option is perfect for quickly analyzing a message you type directly into the program, without needing to save it as a file first.

* **What to do:**
  1. Type 3 and press Enter.
  2. **Select Message Type:** You'll see a small menu asking what kind of message you're typing:
  3. Select Message Type:
  4. 1. WhatsApp Chat Message
  5. 2. Email Message (with optional headers)
  6. 3. SMS Message
  7. 4. Other / Generic Text
  8. 5. Back to Main Menu
     + Choose the number that best describes your message (e.g., 1 for a WhatsApp-style message, 2 for an email).
     + If you're unsure, 4. Other / Generic Text is always a safe choice.
     + If you want to go back to the *main menu*, type 5.
  9. **Enter Sender Name (Optional):** The program will ask for a sender name (e.g., John Doe). You can type a name and press Enter, or just press Enter to leave it blank.
     + **To go back to the Message Type Selection:** Type back and press Enter.
  10. **Enter Conversation ID (Optional):** It will then ask for a Conversation ID (e.g., Project\_Meeting\_Chat). This helps group messages that belong to the same discussion. You can type one or leave it blank.
      + **To go back to the Message Type Selection:** Type back and press Enter.
  11. **Type Your Message:**
      + Start typing or paste your message.
      + **To finish:** Press [Enter] on an empty line (just press Enter twice in a row).
      + **To cancel this message and go back to Message Type Selection:** Type cancel on a new line and press [Enter].
  12. **Analyze Another?** After your message is analyzed, the program will ask if you want to "Analyze another custom message? (y/n)".
      + Type y and press Enter to analyze another message of any type.
      + Type n and press Enter to go back to the main menu.
* **What happens:**
  1. The program analyzes your typed message.
  2. A report for this message will be saved to data/reports/.
  3. A summary will be printed in your command prompt/terminal.

**6.4. Exit**

This option simply closes the program.

* **What to do:**
  1. Type 4 and press Enter.
* **What happens:**
  1. The program will close.

3. Understanding the Reports

After analysis, reports are saved as text files in the data/reports/ folder. You can open these with any text editor (like Notepad, VS Code, etc.) or a word processor.

Each report provides:

* **Individual Message Analysis:** For each message, you'll see:
  + **Source:** Where the message came from (e.g., my\_email.txt, personal\_whatsapp.txt, manual\_input\_email).
  + **Subject:** The email subject, or a default like "WhatsApp Chat Message" or "Custom Message".
  + **Sender:** Who sent the message.
  + **Conversation ID:** A label to group related messages.
  + **Timestamp:** When the message was sent.
  + **Message Body Preview:** A short peek at the message content.
  + **Spam:** Whether it's SPAM or HAM (not spam).
  + **Sentiment:** The overall feeling: positive, neutral, or negative.
  + **Style:** A score (higher is often more formal/structured) and its Formality (e.g., formal, informal).
* **Summary Metrics:** Overall numbers for all messages analyzed in that report:
  + Total number of messages.
  + How many were spam vs. ham.
  + The breakdown of positive, neutral, and negative sentiments.
  + The average style score.
  + How many messages were formal vs. informal.
* **Behavioral Insights:** This section looks at communication patterns:
  + **Top Senders:** Who sent the most messages.
  + **Average Response Delay:** If there are multiple messages in the same conversation, this shows the average time it took for someone to reply.
  + **Suggestions:** Tips for improving communication based on the analysis (e.g., "Respond to messages more promptly").

This tool is designed to give you valuable insights into your communication data!

# **Conclusion**

The Communication Analysis Tool developed by Group 16 of the Department of Computer Science at Ladoke Akintola University of Technology provides a practical solution for detecting spam, analyzing writing style, and evaluating sentiment in email texts. Through the integration of various Python modules, including spam detection models, linguistic style analysis, and sentiment classifiers, the system demonstrates how automated text analysis can enhance communication filtering and monitoring.

The modular design of the tool encourages scalability and future integration of machine learning techniques or more advanced natural language processing models. Testing was approached both manually and programmatically, with test cases verifying each component's functionality.

Although certain limitations were observed, such as false positives in spam classification and occasional ambiguity in sentiment detection, the overall performance of the tool remains commendable. With continued refinement, the Communication Analysis Tool holds strong potential for wider application in email clients, digital communication analysis, and academic research.

This project reflects not only our understanding of Python programming and modular application but also our ability to apply computational thinking to solve real-world problems. We believe this work provides a strong foundation for further exploration in artificial intelligence and natural language processing domains.

THANK YOU!

**REFERENCES FOR EXTERNAL RESEARCH:**

* <https://docs.pythton.org/3/library/index.html>
* <https://docs.python.org/py-modinex.html> Python module index – python 3.13.5 documentation
* <https://github.com/topics/python-project-beginner>
* Intermediate Python Programming Laboratory Manual, Ladoke Akintola University of Technology, Ogbomosho, Oyo State, Nigeria.

# **APPENDIX A – UNIT TEST CODES FOR EACH MODULES**

## **Unit Test Code A.1: test\_sentiment\_analysis.py**

"""

Unit tests for the SentimentAnalyzer module.

This module verifies the sentiment classification logic, ensuring messages

are correctly categorized as positive, negative, or neutral based on lexicons.

"""

import unittest

from modules.sentiment\_analyzer import SentimentAnalyzer

class TestSentimentAnalyzer(unittest.TestCase):

    def setUp(self):

        self.analyzer = SentimentAnalyzer()

    def test\_positive\_sentiment(self):

        self.assertEqual(self.analyzer.analyze("This is a truly wonderful and amazing experience!"), "positive")

        self.assertEqual(self.analyzer.analyze("I love this great idea."), "positive")

    def test\_negative\_sentiment(self):

        self.assertEqual(self.analyzer.analyze("This is a terrible and awful problem."), "negative")

        self.assertEqual(self.analyzer.analyze("I hate this bad issue."), "negative")

    def test\_neutral\_sentiment\_no\_keywords(self):

        self.assertEqual(self.analyzer.analyze("The quick brown fox jumps over the lazy dog."), "neutral")

        self.assertEqual(self.analyzer.analyze("A cat sat on the mat."), "neutral")

    def test\_neutral\_sentiment\_balanced\_keywords(self):

        self.assertEqual(self.analyzer.analyze("It was a good idea but had a terrible implementation."), "neutral")

        self.assertEqual(self.analyzer.analyze("I like the concept, but the execution was bad."), "neutral")

    def test\_empty\_message(self):

        self.assertEqual(self.analyzer.analyze(""), "neutral")

    def test\_message\_with\_punctuation\_and\_numbers(self):

        self.assertEqual(self.analyzer.analyze("This is great! (123)"), "positive")

        self.assertEqual(self.analyzer.analyze("What a terrible-problem!"), "negative")

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

## **Unit Test Code A.2: test\_spam\_detector.py**

"""

Unit tests for the SpamDetector module.

This module contains test cases to verify the correct functionality of:

- Message tokenization.

- Training process for ham and spam messages.

- Probability calculation for words.

- Prediction accuracy for classifying messages as spam or ham.

"""

import unittest

import os

from collections import defaultdict

from modules.spam\_detector import SpamDetector

class TestSpamDetector(unittest.TestCase):

    def setUp(self):

        self.test\_dir = "test\_training\_data"

        os.makedirs(self.test\_dir, exist\_ok=True)

        with open(os.path.join(self.test\_dir, "ham\_messages.txt"), "w") as f:

            f.write("hello world\n")

            f.write("how are you\n")

        with open(os.path.join(self.test\_dir, "spam\_messages.txt"), "w") as f:

            f.write("buy now free money\n")

            f.write("win lottery prize\n")

        self.detector = SpamDetector(self.test\_dir)

    def tearDown(self):

        import shutil

        shutil.rmtree(self.test\_dir)

    def test\_tokenize(self):

        text = "Hello, World! 123 Buy now. Free money!!!"

        expected = ["hello", "world", "buy", "now", "free", "money"]

        self.assertEqual(self.detector.\_tokenize(text), expected)

    def test\_train\_ham\_message(self):

        initial\_ham\_count = self.detector.ham\_messages\_count

        initial\_ham\_words = dict(self.detector.ham\_words)

        self.detector.\_train("test ham message", False)

        self.assertEqual(self.detector.ham\_messages\_count, initial\_ham\_count + 1)

        self.assertEqual(self.detector.ham\_words["test"], 1)

        self.assertEqual(self.detector.ham\_words["ham"], 1)

        self.assertEqual(self.detector.ham\_words["message"], 1)

    def test\_predict\_ham(self):

        self.assertFalse(self.detector.predict("this is a normal email"))

    def test\_predict\_spam(self):

        self.assertTrue(self.detector.predict("urgent claim your prize now"))

    def test\_predict\_unknown\_word(self):

        self.assertFalse(self.detector.predict("this is a completely new sentence"))

    def test\_empty\_message\_prediction(self):

        self.assertFalse(self.detector.predict(""))

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

## **Unit Test code A.3: test\_style\_analyzer**

"""

Unit tests for the StyleAnalyzer module.

This module verifies the style analysis logic, ensuring text is correctly

assigned a style score and classified as formal or informal.

"""

import unittest

from modules.style\_analyzer import StyleAnalyzer

class TestStyleAnalyzer(unittest.TestCase):

    def setUp(self):

        self.analyzer = StyleAnalyzer()

    def test\_formal\_style(self):

        formal\_text = "Furthermore, we endeavor to facilitate the optimal utilization of resources."

        result = self.analyzer.analyze(formal\_text)

        self.assertEqual(result["Formality"], "formal")

        self.assertGreaterEqual(result["Style Score"], 50)

        formal\_text\_2 = "Pursuant to our previous correspondence, please find the attached document."

        result\_2 = self.analyzer.analyze(formal\_text\_2)

        self.assertEqual(result\_2["Formality"], "formal")

    def test\_informal\_style(self):

        informal\_text = "Hey dude, what's up? Wanna grab a coffee asap?"

        result = self.analyzer.analyze(informal\_text)

        self.assertEqual(result["Formality"], "informal")

        self.assertLessEqual(result["Style Score"], 50)

        informal\_text\_2 = "Lol, I'm gonna chill this weekend, btw."

        result\_2 = self.analyzer.analyze(informal\_text\_2)

        self.assertEqual(result\_2["Formality"], "informal")

    def test\_empty\_message(self):

        result = self.analyzer.analyze("")

        self.assertEqual(result["Formality"], "informal")

        self.assertEqual(result["Style Score"], 0.0)

    def test\_mixed\_style(self):

        mixed\_text = "Hi team, I'm writing to confirm our meeting. Furthermore, let's prioritize the next steps."

        result = self.analyzer.analyze(mixed\_text)

        self.assertEqual(result["Formality"], "formal")

        self.assertGreater(result["Style Score"], 0)

    def test\_contractions\_impact(self):

        text\_with\_contractions = "I'm sure you'll agree it's a good idea."

        result = self.analyzer.analyze(text\_with\_contractions)

        self.assertEqual(result["Formality"], "informal")

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

## **Unit Test Code A.4: test\_metrics\_calculator.py**

"""

Unit tests for the metrics\_calculator module.

This module verifies the correct calculation of:

- Summary content metrics (total messages, spam/ham counts, sentiment breakdown, avg style, formality).

- Behavioral insights (top senders, average response delay, suggestions).

"""

import unittest

from datetime import datetime, timedelta

from modules.metrics\_calculator import (

    calculate\_content\_metrics,

    calculate\_engagement\_metrics

)

class TestMetricsCalculator(unittest.TestCase):

    def test\_calculate\_content\_metrics\_basic(self):

        results = [

            {"Spam": "HAM", "Sentiment": "positive", "Style Score": 70, "Formality": "formal"},

            {"Spam": "SPAM", "Sentiment": "negative", "Style Score": 30, "Formality": "informal"},

            {"Spam": "HAM", "Sentiment": "neutral", "Style Score": 60, "Formality": "formal"},

        ]

        metrics = calculate\_content\_metrics(results)

        self.assertEqual(metrics["Total Messages"], 3)

        self.assertEqual(metrics["Spam Counts"], {"SPAM": 1, "HAM": 2})

        self.assertEqual(metrics["Sentiment Counts"], {"positive": 1, "neutral": 1, "negative": 1})

        self.assertAlmostEqual(metrics["Average Style Score"], (70+30+60)/3)

        self.assertEqual(metrics["Formality Counts"], {"formal": 2, "informal": 1})

    def test\_calculate\_content\_metrics\_empty(self):

        metrics = calculate\_content\_metrics([])

        self.assertEqual(metrics["Total Messages"], 0)

        self.assertEqual(metrics["Spam Counts"], {"SPAM": 0, "HAM": 0})

        self.assertEqual(metrics["Sentiment Counts"], {"positive": 0, "neutral": 0, "negative": 0})

        self.assertEqual(metrics["Average Style Score"], 0.0)

        self.assertEqual(metrics["Formality Counts"], {"formal": 0, "informal": 0})

    def test\_calculate\_engagement\_metrics\_response\_time(self):

        t1 = datetime(2025, 7, 26, 10, 0, 0)

        t2 = datetime(2025, 7, 26, 10, 5, 0)

        t3 = datetime(2025, 7, 26, 10, 10, 0)

        results = [

            {"Sender": "Alice", "Conversation ID": "convo1", "Timestamp": t1, "Style Score": 50, "Formality": "informal"},

            {"Sender": "Bob", "Conversation ID": "convo1", "Timestamp": t2, "Style Score": 50, "Formality": "informal"},

            {"Sender": "Alice", "Conversation ID": "convo1", "Timestamp": t3, "Style Score": 50, "Formality": "informal"},

        ]

        engagement = calculate\_engagement\_metrics(results)

        self.assertAlmostEqual(engagement["Average Response time (sec)"], 300.0)

        self.assertEqual(engagement["Top Senders"], {"Alice": 2, "Bob": 1})

    def test\_calculate\_engagement\_metrics\_multiple\_conversations(self):

        t1\_a = datetime(2025, 7, 26, 9, 0, 0)

        t1\_b = datetime(2025, 7, 26, 9, 10, 0)

        t2\_a = datetime(2025, 7, 26, 11, 0, 0)

        t2\_b = datetime(2025, 7, 26, 11, 5, 0)

        results = [

            {"Sender": "UserA", "Conversation ID": "chatX", "Timestamp": t1\_a, "Style Score": 50, "Formality": "informal"},

            {"Sender": "UserB", "Conversation ID": "chatX", "Timestamp": t1\_b, "Style Score": 50, "Formality": "informal"},

            {"Sender": "UserC", "Conversation ID": "chatY", "Timestamp": t2\_a, "Style Score": 50, "Formality": "informal"},

            {"Sender": "UserD", "Conversation ID": "chatY", "Timestamp": t2\_b, "Style Score": 50, "Formality": "informal"},

        ]

        engagement = calculate\_engagement\_metrics(results)

        self.assertAlmostEqual(engagement["Average Response time (sec)"], (600 + 300) / 2)

        self.assertEqual(engagement["Top Senders"], {"UserA": 1, "UserB": 1, "UserC": 1, "UserD": 1})

    def test\_calculate\_engagement\_metrics\_no\_response\_data(self):

        results = [

            {"Sender": "Alice", "Conversation ID": "convo1", "Timestamp": datetime(2025, 7, 26, 10, 0, 0), "Style Score": 50, "Formality": "informal"},

            {"Sender": "Bob", "Conversation ID": "convo2", "Timestamp": datetime(2025, 7, 26, 10, 10, 0), "Style Score": 50, "Formality": "informal"},

        ]

        engagement = calculate\_engagement\_metrics(results)

        self.assertIsNone(engagement["Average Response time (sec)"])

        self.assertEqual(engagement["Top Senders"], {"Alice": 1, "Bob": 1})

    def test\_suggestions\_generation(self):

        results\_high\_delay = [

            {"Sender": "A", "Conversation ID": "c1", "Timestamp": datetime(2025, 1, 1, 10, 0), "Style Score": 20, "Formality": "informal"},

            {"Sender": "B", "Conversation ID": "c1", "Timestamp": datetime(2025, 1, 2, 10, 0), "Style Score": 20, "Formality": "informal"},

        ]

        engagement\_high\_delay = calculate\_engagement\_metrics(results\_high\_delay)

        self.assertIn("Respond to the messages more promptly to improve conversation engagement.", engagement\_high\_delay["Suggestions"])

        self.assertIn("Use more formal phrasing for formal tone.", engagement\_high\_delay["Suggestions"])

        results\_good = [

            {"Sender": "A", "Conversation ID": "c1", "Timestamp": datetime(2025, 1, 1, 10, 0), "Style Score": 80, "Formality": "formal"},

            {"Sender": "B", "Conversation ID": "c1", "Timestamp": datetime(2025, 1, 1, 10, 10), "Style Score": 80, "Formality": "formal"},

        ]

        engagement\_good = calculate\_engagement\_metrics(results\_good)

        self.assertListEqual(engagement\_good["Suggestions"], [])

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

## **Unit Test Code A.5: test\_report\_generator.py**

"""

Unit tests for the report\_generator module.

This module verifies the parsing capabilities for different message formats

(single email, multi-email, WhatsApp), and the overall report generation process.

It uses mocking to isolate the report\_generator from external dependencies

like SpamDetector, SentimentAnalyzer, and MetricsCalculator.

"""

import unittest

import os

import shutil

from unittest.mock import Mock, patch

from datetime import datetime

from modules.report\_generator import (

    generate\_report,

    generate\_custom\_test\_report,

    generate\_report\_from\_custom\_input,

    \_parse\_single\_email\_block,

    \_parse\_multi\_email\_file,

    \_parse\_whatsapp\_content,

    \_is\_whatsapp\_format,

    \_has\_email\_boundary,

    EMAIL\_BOUNDARY\_SEPARATOR

)

TEST\_EMAIL\_SINGLE = """Subject: Test Email

From: test@example.com

Date: 2025-01-01 12:00:00

Conversation-ID: email\_test\_001

This is a test email body.

"""

TEST\_EMAIL\_MULTI = f"""Subject: First Email

From: multi1@example.com

Date: 2025-01-01 10:00:00

Conversation-ID: multi\_test\_001

Body of first email.

{EMAIL\_BOUNDARY\_SEPARATOR}

Subject: Second Email

From: multi2@example.com

Date: 2025-01-01 10:05:00

Conversation-ID: multi\_test\_001

Body of second email.

"""

TEST\_WHATSAPP = """1/1/25, 10:00 AM - Alice: Hey Bob!

1/1/25, 10:05 AM - Bob: Hi Alice! How are you?

1/1/25, 10:10 AM - Alice: I'm great, thanks!

"""

class TestReportGenerator(unittest.TestCase):

    def setUp(self):

        self.test\_data\_dir = "test\_data\_reports"

        self.test\_sample\_emails\_dir = os.path.join(self.test\_data\_dir, "sample\_emails")

        self.test\_reports\_dir = os.path.join(self.test\_data\_dir, "reports")

        os.makedirs(self.test\_sample\_emails\_dir, exist\_ok=True)

        os.makedirs(self.test\_reports\_dir, exist\_ok=True)

        with open(os.path.join(self.test\_sample\_emails\_dir, "single\_email.txt"), "w") as f:

            f.write(TEST\_EMAIL\_SINGLE)

        with open(os.path.join(self.test\_sample\_emails\_dir, "multi\_email.txt"), "w") as f:

            f.write(TEST\_EMAIL\_MULTI)

        with open(os.path.join(self.test\_sample\_emails\_dir, "whatsapp\_chat.txt"), "w") as f:

            f.write(TEST\_WHATSAPP)

        self.mock\_spam\_detector = Mock()

        self.mock\_spam\_detector.predict.return\_value = False

        self.mock\_sentiment\_analyzer = Mock()

        self.mock\_sentiment\_analyzer.analyze.return\_value = "neutral"

        self.mock\_style\_analyzer = Mock()

        self.mock\_style\_analyzer.analyze.return\_value = {"Style Score": 50, "Formality": "informal"}

        self.mock\_metrics\_calculator\_content = Mock(return\_value={

            "Total Messages": 1, "Spam Counts": {"SPAM": 0, "HAM": 1},

            "Sentiment Counts": {"positive": 0, "neutral": 1, "negative": 0},

            "Average Style Score": 50.0, "Formality Counts": {"formal": 0, "informal": 1}

        })

        self.mock\_metrics\_calculator\_engagement = Mock(return\_value={

            "Top Senders": {"TestSender": 1}, "Average Response time (sec)": None,

            "Suggestions": ["No behavioral recommendations found."]

        })

        patcher\_sd = patch('modules.report\_generator.SpamDetector', return\_value=self.mock\_spam\_detector)

        patcher\_sa = patch('modules.report\_generator.SentimentAnalyzer', return\_value=self.mock\_sentiment\_analyzer)

        patcher\_st = patch('modules.report\_generator.StyleAnalyzer', return\_value=self.mock\_style\_analyzer)

        patcher\_cm = patch('modules.report\_generator.calculate\_content\_metrics', new=self.mock\_metrics\_calculator\_content)

        patcher\_em = patch('modules.report\_generator.calculate\_engagement\_metrics', new=self.mock\_metrics\_calculator\_engagement)

        self.mock\_spam\_detector\_start = patcher\_sd.start()

        self.mock\_sentiment\_analyzer\_start = patcher\_sa.start()

        self.mock\_style\_analyzer\_start = patcher\_st.start()

        self.mock\_metrics\_calculator\_content\_start = patcher\_cm.start()

        self.mock\_metrics\_calculator\_engagement\_start = patcher\_em.start()

        self.addCleanup(patcher\_sd.stop)

        self.addCleanup(patcher\_sa.stop)

        self.addCleanup(patcher\_st.stop)

        self.addCleanup(patcher\_cm.stop)

        self.addCleanup(patcher\_em.stop)

        self.addCleanup(lambda: shutil.rmtree(self.test\_data\_dir))

    def test\_parse\_single\_email\_block(self):

        headers, body = \_parse\_single\_email\_block(TEST\_EMAIL\_SINGLE)

        self.assertEqual(headers["Subject"], "Test Email")

        self.assertEqual(headers["Sender"], "test@example.com")

        self.assertIn("test email body", body)

    def test\_parse\_multi\_email\_file(self):

        parsed\_messages = \_parse\_multi\_email\_file(TEST\_EMAIL\_MULTI, "multi\_email.txt")

        self.assertEqual(len(parsed\_messages), 2)

        self.assertEqual(parsed\_messages[0]["Subject"], "First Email")

        self.assertEqual(parsed\_messages[1]["Sender"], "multi2@example.com")

        self.assertEqual(parsed\_messages[0]["Conversation ID"], "multi\_test\_001")

        self.assertEqual(parsed\_messages[1]["Conversation ID"], "multi\_test\_001")

    def test\_parse\_whatsapp\_content(self):

        parsed\_messages = \_parse\_whatsapp\_content(TEST\_WHATSAPP, "whatsapp\_chat.txt")

        self.assertEqual(len(parsed\_messages), 3)

        self.assertEqual(parsed\_messages[0]["Sender"], "Alice")

        self.assertEqual(parsed\_messages[1]["Message"], "Hi Alice! How are you?")

        self.assertIsNotNone(parsed\_messages[0]["Timestamp"])

        self.assertEqual(parsed\_messages[0]["Conversation ID"], "whatsapp\_chat\_whatsapp\_chat\_txt")

    @patch('builtins.open', new\_callable=unittest.mock.mock\_open, read\_data=TEST\_EMAIL\_SINGLE)

    @patch('modules.report\_generator.os.path.exists', return\_value=True)

    def test\_generate\_custom\_test\_report\_single\_email(self, mock\_exists, mock\_open):

        test\_file\_path = os.path.join(self.test\_sample\_emails\_dir, "single\_email.txt")

        generate\_custom\_test\_report(test\_file\_path, output\_path=os.path.join(self.test\_reports\_dir, "custom\_report.txt"))

        mock\_open.assert\_called\_with(unittest.mock.ANY, 'w', encoding='utf-8')

        self.mock\_sentiment\_analyzer.analyze.assert\_called\_once()

        self.mock\_metrics\_calculator\_content.assert\_called\_once()

    @patch('builtins.open', new\_callable=unittest.mock.mock\_open, read\_data=TEST\_WHATSAPP)

    @patch('modules.report\_generator.os.path.exists', return\_value=True)

    def test\_generate\_custom\_test\_report\_whatsapp(self, mock\_exists, mock\_open):

        test\_file\_path = os.path.join(self.test\_sample\_emails\_dir, "whatsapp\_chat.txt")

        generate\_custom\_test\_report(test\_file\_path, output\_path=os.path.join(self.test\_reports\_dir, "whatsapp\_report.txt"))

        expected\_calls = len(TEST\_WHATSAPP.strip().split('\n'))

        self.assertGreaterEqual(self.mock\_sentiment\_analyzer.analyze.call\_count, 2)

        self.mock\_metrics\_calculator\_content.assert\_called\_once()

        self.mock\_metrics\_calculator\_engagement.assert\_called\_once()

    @patch('modules.report\_generator.\_print\_summary\_to\_cli')

    def test\_cli\_summary\_called(self, mock\_print\_summary):

        test\_file\_path = os.path.join(self.test\_sample\_emails\_dir, "single\_email.txt")

        generate\_custom\_test\_report(test\_file\_path)

        mock\_print\_summary.assert\_called\_once()

    def test\_generate\_report\_from\_custom\_input\_whatsapp\_format(self):

        message = "2/14/25, 4:26 PM - Bobola 🤖✨: Ohhh okay"

        generate\_report\_from\_custom\_input(message, message\_type="WhatsApp")

        args, kwargs = self.mock\_sentiment\_analyzer.analyze.call\_args

        self.assertEqual(args[0], "Ohhh okay")

        content\_metrics\_call\_args = self.mock\_metrics\_calculator\_content.call\_args[0][0]

        self.assertEqual(content\_metrics\_call\_args[0]["Sender"], "Bobola 🤖✨")

        self.assertIn("whatsapp\_chat\_typed\_whatsapp\_msg\_txt", content\_metrics\_call\_args[0]["Conversation ID"])

        self.assertIn("2025-02-14 16:26:00", content\_metrics\_call\_args[0]["Timestamp"])

    def test\_generate\_report\_from\_custom\_input\_email\_format(self):

        message = """Subject: Test Email Input

From: typed@example.com

Date: 2025-07-26 15:00:00

Conversation-ID: typed\_email\_001

This is the body of the typed email."""

        generate\_report\_from\_custom\_input(message, message\_type="Email")

        content\_metrics\_call\_args = self.mock\_metrics\_calculator\_content.call\_args[0][0]

        self.assertEqual(content\_metrics\_call\_args[0]["Sender"], "typed@example.com")

        self.assertEqual(content\_metrics\_call\_args[0]["Subject"], "Test Email Input")

        self.assertEqual(content\_metrics\_call\_args[0]["Conversation ID"], "typed\_email\_001")

        self.assertIn("2025-07-26 15:00:00", content\_metrics\_call\_args[0]["Timestamp"])

        self.assertEqual(content\_metrics\_call\_args[0]["Message"], "This is the body of the typed email.")

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

# **APPENDIX B: SAMPLE DATA FOR INTEGRATION AND TESTING**

## **Sample data B.1: Formal email/message sample**

Subject: Confirmation: Project Synergy Kick-off Meeting

From: alicia.smith@example.com

To: bob.jones@example.com

Date: 2025-07-26 10:00:00

Conversation-ID: formal\_synergy\_001

Dear Mr. Jones,

This email confirms our Project Synergy kick-off meeting scheduled for Monday, July 29th, at 9:00 AM in Conference Room B.

The agenda and pre-reading materials have been attached for your review. Please come prepared to discuss the initial project scope and resource allocation.

We look forward to a productive discussion.

Sincerely,

Alicia Smith

Project Lead

---EMAIL\_BOUNDARY---

Subject: RE: Confirmation: Project Synergy Kick-off Meeting

From: bob.jones@example.com

To: alicia.smith@example.com

Date: 2025-07-26 10:15:00

Conversation-ID: formal\_synergy\_001

Dear Ms. Smith,

Thank you for the confirmation regarding the Project Synergy kick-off meeting. I have received the agenda and materials and will review them thoroughly.

I anticipate a productive discussion on Monday.

Best regards,

Bob Jones

Senior Consultant

## **Sample data B.2: Informal message sample**

Subject: Hey! What's up?

From: sarah.davis@example.com

To: mark.wilson@example.com

Date: 2025-07-26 14:30:00

Conversation-ID: informal\_weekend\_002

Hey Mark,

What's up for the weekend? Thinking of hitting up that new coffee spot downtown. Wanna join? It's supposed to be super chill.

Let me know!

Cheers,

Sarah

---EMAIL\_BOUNDARY---

Subject: Re: Hey! What's up?

From: mark.wilson@example.com

To: sarah.davis@example.com

Date: 2025-07-26 16:00:00

Conversation-ID: informal\_weekend\_002

Hey Sarah,

Oh, totally! That sounds awesome. I'm free Saturday morning. Is that cool? I heard their pastries are amazing.

Can't wait!

Mark

## **Sample data B.3: Spam sample**

Subject: URGENT: Action Required - Security Breach Detected

From: security.alert@example.com

To: user@example.com

Date: 2025-07-25 10:00:00

Conversation-ID: security\_alert\_convo\_1

Dear User,

Our systems have detected unusual activity on your account. For your protection, we have temporarily suspended your access.

To restore your account access immediately, please click on the following secure link and verify your credentials:

[MALICIOUS\_LINK\_PLACEHOLDER]

Failure to do so within 24 hours will result in permanent account deactivation. This is a critical security measure.

Sincerely,

Security DepartmentSubject: URGENT: Your Account Has Been Compromised - Action Required!

From: security.update@secure-login.com

To: victim@example.com

Date: 2025-07-26 08:00:00

Conversation-ID: spam\_alert\_003

Dear Valued Customer,

We have detected suspicious activity on your account originating from an unrecognized device. For your security, your account has been temporarily locked.

To avoid permanent deactivation, you MUST verify your account details immediately by clicking the link below:

CLICK HERE TO VERIFY: [https://www.google.com/search?q=http://suspicious-link.biz/verify-now]

Failure to comply within 12 hours will result in the permanent closure of your account and loss of all data. This is a critical security measure.

Thank You,

Security Team

---EMAIL\_BOUNDARY---

Subject: FINAL WARNING: Your Account Deactivation Pending!

From: security.update@secure-login.com

To: victim@example.com

Date: 2025-07-26 10:00:00

Conversation-ID: spam\_alert\_003

ATTENTION:

This is your final notification regarding the security breach detected on your account. Our records indicate you have NOT completed the mandatory verification process.

Your account will be permanently terminated in 1 hour. Do not miss this LAST CHANCE to secure your funds and personal information.

ACT NOW: [https://www.google.com/search?q=http://suspicious-link.biz/final-chance]

We are not responsible for any loss of data or funds due to non-compliance.

Sincerely,

Account Management

## **Sample data B.4: Non-Spam(Legitimate)message sample.**

Subject: Project Alpha - Weekly Update

From: project.manager@example.com

To: team@example.com

Date: 2025-07-25 09:30:00

Conversation-ID: project\_alpha\_convo\_1

Hi Team,

Just a quick update on Project Alpha. We've successfully completed Phase 1 ahead of schedule, which is fantastic news! The new module integration went smoothly, and initial tests show promising results.

Next steps for this week include:

1. Reviewing the Q3 financial projections.

2.Preparing for the client demo on Friday.

3.Addressing the minor bug reported in the user interface.

Please ensure all documentation is up-to-date by end of day Wednesday. Your prompt attention to these matters is greatly appreciated.

Thanks,

Project Manager

## **APPENDIX C: Spam Detector trainer**

This was the code and samples used to train our spam detector model in differentiating between spam and legitimate messages, it can be modified by the user.

## **C.1: train\_spam\_detector.py**

***Trains the SpamDetector Class using sample messages from digital logs. It stores word and their frequencies seperately in a folder called training\_data.***

"""

train\_spam\_detector.py

Trains the SpamDetector using sample messages from digitallogs.

Stores word frequencies in ham\_word\_counts.json and spam\_word\_counts.json.

"""

from modules.spam\_detector import SpamDetector

import os

# File paths to training examples

spam\_file = "digitallogssample/spam\_messages.txt"

ham\_file = "digitallogssample/ham\_messages.txt"

# Create detector with correct data path

detector = SpamDetector(data\_path="data/training\_data")

def train\_from\_file(filepath, is\_spam):

    # Check file exists

    if not os.path.exists(filepath):

        print(f" File not found: {filepath}")

        return

    # Feed each line into training model

    with open(filepath, "r", encoding="utf-8") as file:

        for line in file:

            message = line.strip()

            if message:

                detector.train(message, is\_spam)

# Train using your labeled files

train\_from\_file(spam\_file, is\_spam=True)

train\_from\_file(ham\_file, is\_spam=False)

# Save learned word frequencies

detector.\_save\_model()

print(" Training complete. Word counts saved to training data folder.")

## **C.2: Trained data**

***Data that already been separated by the spam detector and was gotten from real time logs and was used as a foundational bases for the spam detector class to function properly without occurring errors and it needs to be trained constantly and be given more samples to detect spam or ham in a message effectively. It is modifiable by adding more data either to the spam or ham directory in the digital logs sample folder in the root folder and it would be saved in their respective json files assigned to them with their frequency, the higher the effectiveness of the spam detector.***

### **C.2.1: ham\_word\_count.json**

The detected legitimate words to check whether your message is legitimate and their frequencies i.e. the amount of times used in all the real time data samples.

***{"please": 7, "review": 4, "the": 30, "attached": 5, "budget": 4, "file.": 4, "reminder:": 5, "our": 7, "team": 4, "sync": 4, "is": 13, "at": 9, "10am": 4, "tomorrow.": 6, "i\u2019ve": 6, "shared": 4, "meeting": 11, "notes": 5, "with": 6, "you.": 4, "let\u2019s": 8, "catch": 6, "up": 7, "about": 4, "project": 5, "update.": 4, "your": 18, "assignment": 4, "due": 4, "by": 4, "friday": 4, "evening.": 4, "can": 6, "we": 5, "reschedule": 2, "to": 11, "tomorrow?": 2, "i\u2019m": 5, "on": 5, "my": 2, "way,": 2, "be": 2, "there": 2, "in": 3, "10": 2, "minutes.": 2, "grab": 2, "lunch": 4, "usual": 2, "place.": 2, "thanks": 3, "for": 9, "help": 3, "project.": 2, "don't": 3, "forget": 4, "submit": 2, "assignment.": 2, "good": 2, "morning!": 2, "hope": 3, "you\u2019re": 3, "doing": 2, "well.": 2, "that\u2019s": 2, "a": 4, "great": 3, "idea,": 2, "in.": 3, "this": 3, "weekend.": 2, "sent": 2, "documents": 2, "email.": 2, "you": 10, "call": 4, "me": 4, "when": 4, "you're": 2, "free?": 2, "i": 3, "enjoyed": 2, "presentation": 2, "today.": 2, "set": 2, "3": 3, "pm,": 2, "conference": 2, "room": 2, "b.": 2, "hey,": 2, "are": 1, "still": 1, "pm?": 1, "pick": 1, "groceries": 1, "way": 1, "back.": 1, "send": 1, "from": 1, "yesterday\u2019s": 1, "class?": 1, "happy": 1, "birthday!": 1, "have": 2, "day!": 1, "i'll": 1, "later,": 1, "right": 1, "now.": 1, "package": 1, "has": 2, "been": 2, "shipped": 1, "and": 1, "will": 1, "arrive": 1, "thank": 1, "payment.": 1, "order": 1, "confirmed.": 1, "let": 1, "know": 1, "free": 1, "week.": 1, "dentist": 1, "appointment": 1, "tomorrow": 1, "11": 1, "am.": 1, "just": 1, "checking": 1, "how": 1, "been?": 1, "rescheduled": 1, "2": 1, "pm.": 1, "confirm": 1, "attendance.": 1, "see": 1, "party": 1, "tonight!": 1, "monthly": 1, "bank": 1, "statement": 1, "now": 1, "available": 1, "online.": 1, "move": 1, "4": 1, "pm": 1, "instead": 1, "of": 1, "3?": 1, "congrats": 1, "new": 1, "job!": 1, "do": 2, "start?": 1, "here's": 1, "document": 1, "asked": 1, "for.": 1, "was": 1, "great!": 1, "should": 1, "it": 1, "again": 2, "soon.": 1, "how\u2019s": 1, "coming": 1, "along?": 1, "don\u2019t": 1, "bring": 1, "id": 1, "test": 1, "agenda": 1, "tomorrow's": 1, "meeting.": 1, "left": 1, "files": 1, "desk.": 1, "today!": 1, "flight": 1, "confirmation:": 1, "lax": 1, "jfk,": 1, "tuesday": 1, "8": 1}***

### **C.2.2: spam\_word\_count.json**

The detected spam words to check whether your message is non legitimate and their frequencies i.e. the amount of times used in all the real time data samples for effectiveness and to detect them in the logs.

***{"click": 13, "here": 11, "to": 29, "claim": 12, "your": 30, "free": 22, "prize!": 5, "you've": 10, "been": 17, "selected": 9, "for": 14, "a": 25, "cash": 4, "reward!": 4, "win": 5, "phone": 4, "now": 8, "\u2014": 4, "limited": 8, "time": 7, "offer.": 4, "act": 5, "fast": 6, "receive": 5, "exclusive": 5, "gift.": 4, "earn": 9, "money": 5, "from": 8, "home": 5, "with": 7, "no": 10, "experience!": 4, "congratulations!": 3, "you\u2019ve": 6, "won": 6, "iphone.": 3, "it.": 2, "$5000": 3, "week": 2, "working": 2, "home.": 3, "sign": 4, "up": 4, "now.": 7, "get": 8, "cheap": 2, "meds": 2, "online.": 2, "prescription": 2, "needed.": 2, "you": 5, "have": 3, "vacation.": 2, "call": 2, "now!": 11, "urgent:": 2, "account": 5, "has": 5, "compromised.": 2, "reset": 2, "here!": 2, "buy": 3, "followers": 2, "and": 4, "likes": 2, "instantly.": 3, "100%": 2, "real": 2, "users!": 2, "lottery": 2, "winnings": 2, "is": 9, "limited.": 2, "this": 7, "not": 3, "scam.": 2, "truly": 2, "big!": 2, "rich": 3, "quick": 2, "crypto": 2, "opportunity.": 2, "offer:": 3, "70%": 2, "discount": 3, "on": 6, "all": 3, "products.": 2, "shop": 2, "hot": 3, "singles": 3, "in": 4, "area": 3, "want": 2, "chat.": 2, "join": 5, "today!": 4, "chosen!": 2, "gift": 3, "card": 3, "waiting": 3, "you.": 3, "$1000": 2, "walmart": 1, "card.": 1, "urgent!": 1, "suspended.": 1, "verify": 2, "avoid": 2, "permanent": 1, "lock.": 1, "offer!": 1, "1": 1, "3": 1, "weight": 2, "loss": 1, "pills.": 1, "claim.": 1, "$500/day": 1, "experience": 1, "required.": 1, "final": 2, "notice:": 1, "car": 1, "warranty": 1, "about": 1, "expire.": 1, "renew": 1, "unlock": 1, "50%": 1, "electronics": 1, "today": 1, "only!": 1, "we": 1, "tried": 1, "deliver": 1, "package.": 1, "please": 1, "reschedule": 1, "at": 1, "scamlink.com.": 1, "bitcoin": 1, "giveaway!": 1, "register": 1, "2": 1, "btc": 1, "are": 2, "meet": 1, "chosen": 1, "$500": 1, "amazon": 1, "reward.": 1, "before": 1, "it": 1, "expires.": 1, "paypal": 1, "restricted.": 1, "login": 1, "restore": 1, "access.": 2, "pre-approved": 2, "$10,000": 1, "loan.": 1, "credit": 3, "check!": 1, "lose": 1, "without": 1, "dieting.": 1, "miracle": 1, "pill": 1, "approved": 1, "by": 2, "doctors!": 1, "deal:": 1, "work": 1, "make": 2, "$2000/week.": 1, "trial": 1, "of": 4, "our": 3, "skincare": 1, "product!": 1, "dear": 1, "user,": 1, "the": 1, "lucky": 1, "winner": 1, "$5": 1, "million.": 1, "contact": 1, "us": 1, "mobile": 1, "number": 1, "$100,000": 1, "monthly": 1, "draw.": 1, "special": 1, "alert:": 1, "unusual": 1, "activity": 1, "detected": 1, "account.": 1, "identity": 1, "trip": 1, "dubai.": 1, "spots": 1, "available!": 1, "$10": 1, "every": 1, "minute": 1, "just": 2, "watching": 1, "videos!": 1, "quick!": 1, "invest": 1, "only": 1, "$100": 1, "guaranteed!": 1, "reply": 1, "\"yes\"": 1, "confirm": 1, "scam!": 1, "really": 1, "won!": 1, "lowest": 1, "mortgage": 1, "rates": 1, "available": 2, "refinance": 1, "netflix": 1, "expiring.": 1, "update": 1, "payment": 1, "info": 1, "here.": 1, "activate": 1, "antivirus": 1, "software.": 1, "you\u2019re": 2, "missing": 1, "out": 1, "big": 1, "earnings.": 1, "program": 2, "last": 1, "chance": 1, "smartwatch!": 1, "score": 1, "check": 1, "-": 1, "needed!": 1, "online": 1, "while": 1, "sleep!": 1, "offer": 1, "\u2013": 2, "fees!": 1, "samsung": 1, "galaxy": 1, "answering": 1, "few": 1, "questions!": 1, "rid": 1, "debt": 1, "government": 2, "hacked!": 1, "fix": 1, "computer!": 1, "warning:": 1, "legal": 1, "action.": 1, "inheritance": 1, "$4.5": 1, "million": 1, "relief": 1, "grant.": 1, "apply": 1, "immediately!": 1}***