Word Frequency Analyzer

Ryan Collingham
CMIS202

Introduction

Who are the users of the software?

The users of the software could be students, educators, researchers, or anyone who deals with large amounts of text and needs to analyze word frequency for various purposes.

How does the software work?

The software tokenizes input text, counts the frequency of each word using HashMap, sorts the frequencies, and presents them to the user via a JavaFX interface. A timer functionality is used to calculate words per minute of the user.

What is the purpose of the software?

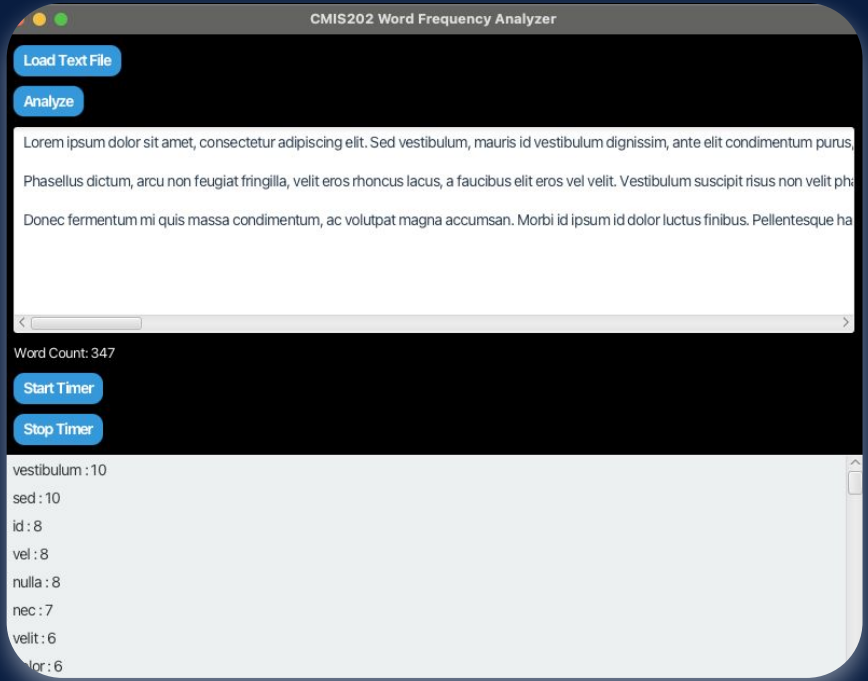
The software provides insights into the distribution of words within a text, helping users understand which words are used most frequently.

Why use the software over existing processes?

The software is open source and free to use, which makes it more customizable and accessable to students. The software uses algorithms to accurately count word frequencies, reducing the risk of human error.

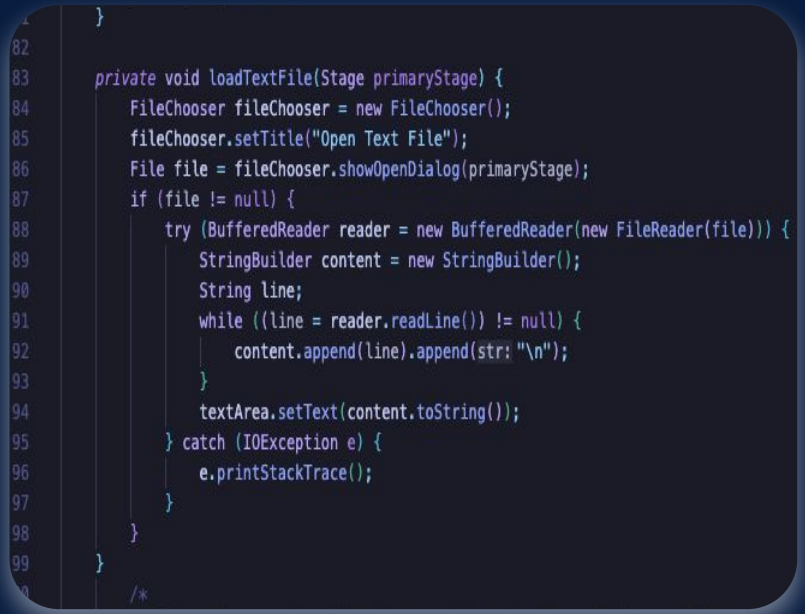
UI Components

- **TextArea:** Allows users to input or view text content, serving as the main area for text interaction within the application.
- **ListView:** Displays sorted word frequencies, presenting users with a clear overview of the most common words in the text.
- **Buttons:** Includes functionalities like "Load Text File" for importing text, "Analyze" for initiating text analysis, "Start Timer" and "Stop Timer" for tracking processing time.
- **Labels:** Features a "Word Count" label to inform users of the total number of words analyzed, facilitating efficient text exploration and analysis.



Loading Text Files

- **User Interaction:** When the user clicks the "Load Text File" button in the interface, it triggers an event handler associated with the button.
- **File Chooser Dialog:** This event handler prompts the operating system's file chooser dialog to open, allowing the user to navigate their file system and select a text file.
- **File Selection:** Once the user selects a text file and confirms their choice, the file chooser dialog closes, and the selected file is passed back to the application.
- **File Reading:** The application attempts to open and read the contents of the selected text file using a BufferedReader, which reads text from a character-input stream.

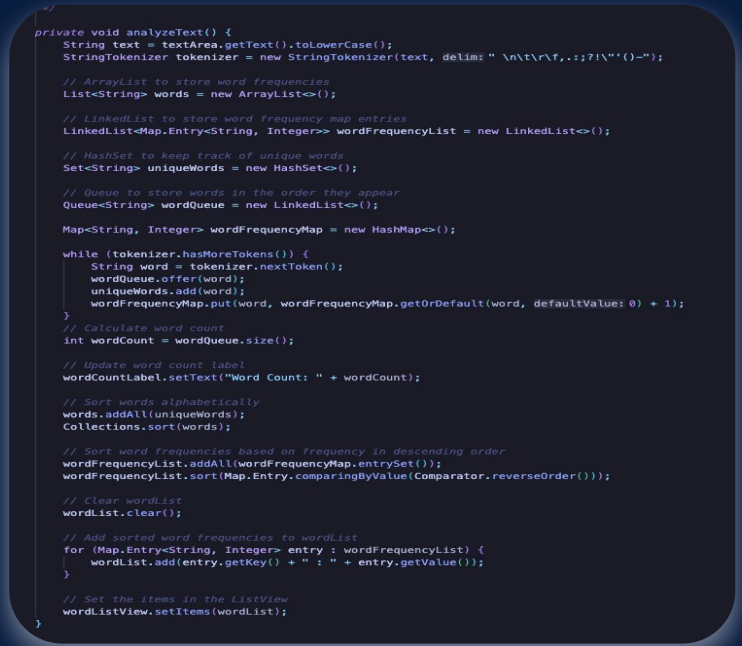


```
}

private void loadTextFile(Stage primaryStage) {
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Open Text File");
    File file = fileChooser.showOpenDialog(primaryStage);
    if (file != null) {
        try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
            StringBuilder content = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                content.append(line).append(str: "\n");
            }
            textArea.setText(content.toString());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
/*
```

Analyzing Text

- **Text Tokenization:** The software tokenizes the loaded text into individual words, ignoring common delimiters like spaces, punctuation marks, and line breaks.
- **Word Frequency Calculation:** Using HashMap and other data structures, the software counts the frequency of each word, ensuring accurate tracking of word occurrences.
- **Sorting Algorithm:** After obtaining word frequencies, the software utilizes Collections.sort() to arrange the frequencies in descending order, providing users with a clear view of the most common words.
- **User Feedback:** The user interface is updated dynamically to display the total word count and the sorted list of word frequencies in the ListView component, enabling users to explore and analyze text patterns effectively.



```
*/
private void analyzeText() {
    String text = textArea.getText().toLowerCase();
    StringTokenizer tokenizer = new StringTokenizer(text, delim: " \n\t\r\f,.:;?!\"'()-");

    // ArrayList to store word frequencies
    List<String> words = new ArrayList<>();

    // LinkedList to store word frequency map entries
    LinkedList<Map.Entry<String, Integer>> wordFrequencyList = new LinkedList<>();

    // HashSet to keep track of unique words
    Set<String> uniqueWords = new HashSet<>();

    // Queue to store words in the order they appear
    Queue<String> wordQueue = new LinkedList<>();

    Map<String, Integer> wordFrequencyMap = new HashMap<>();

    while (tokenizer.hasMoreTokens()) {
        String word = tokenizer.nextToken();
        wordQueue.offer(word);
        uniqueWords.add(word);
        wordFrequencyMap.put(word, wordFrequencyMap.getOrDefault(word, defaultValue: 0) + 1);
    }
    // Calculate word count
    int wordCount = wordQueue.size();

    // Update word count label
    wordCountLabel.setText("Word Count: " + wordCount);

    // Sort words alphabetically
    words.addAll(uniqueWords);
    Collections.sort(words);

    // Sort word frequencies based on frequency in descending order
    wordFrequencyList.addAll(wordFrequencyMap.entrySet());
    wordFrequencyList.sort(Map.Entry.comparingByValue(Comparator.reverseOrder()));

    // Clear wordList
    wordList.clear();

    // Add sorted word frequencies to wordList
    for (Map.Entry<String, Integer> entry : wordFrequencyList) {
        wordList.add(entry.getKey() + " : " + entry.getValue());
    }

    // Set the items in the ListView
    wordListView.setItems(wordList);
}
```

Timing Operations

- **Timer Functionality:** The Word Frequency Analyzer includes timer functionality to measure the time taken for text analysis.
- **Start Timer:** Users can initiate the timer by clicking the "Start Timer" button, which records the current system time as the starting point for analysis.
- **Stop Timer:** Upon completion of text analysis, users can stop the timer by clicking the "Stop Timer" button, capturing the end time and calculating the elapsed time.
- **Words Per Minute (WPM):** The elapsed time is used to calculate the Words Per Minute (WPM) metric, providing users with insights into the efficiency of text processing.

```
private void startTimer() {
    if (!isTimerRunning) {
        startTime = System.currentTimeMillis();
        isTimerRunning = true;
        wordCountLabel.setText("Word Count: 0 (Timer started)");
    } else {
        showAlert(Alert.AlertType.ERROR, content: "Timer is already running.");
    }
}

// Calculates WPM by dividing word count by time (in seconds)
private void stopTimer() {
    if (isTimerRunning) {
        isTimerRunning = false;
        long endTime = System.currentTimeMillis();
        long elapsedTimeInMillis = endTime - startTime;
        double elapsedTimeInSeconds = elapsedTimeInMillis / 1000.0;
        int wordCount = textArea.getText().split(regex: "\\s+").length;
        int wordsPerMinute = (int) (wordCount / (elapsedTimeInSeconds / 60));
        showAlert(Alert.AlertType.INFORMATION, "Words Per Minute (WPM): " + wordsPerMinute);
    } else {
        showAlert(Alert.AlertType.ERROR, content: "Timer is not running.");
    }
}
```

Error Handling

- **Event Registration:** Event handlers are associated with specific UI components, such as buttons. For example, I use lambda expressions to register event handlers directly inline with the button's setOnAction method.
- **Event Handling:** Upon triggering, the corresponding event handler method is called to handle the event. For instance, when the "Load Text File" button is clicked, my loadTextFile(Stage primaryStage) method is invoked to handle the event. It opens a file chooser dialog and loads the selected text file.
- **Event Processing:** Inside the event handler method, I implement the necessary logic to respond to the event. For example, in my analyzeText() method, I execute text analysis logic to tokenize the text, count word frequencies, and update the UI with the results.

```
private void showAlert(Alert.AlertType alertType, String content) {
    Alert alert = new Alert(alertType);
    alert.setTitle("Word Frequency Analyzer");
    alert.setHeaderText(null);
    alert.setContentText(content);
    alert.showAndWait();
}
```