Project Report

Text File Analyzer

1. What the Task Was

The goal of this project was to implement a **Text File Analyzer** application, which provides the user an efficient way to analyze text files and obtain structured data.

The project ensures interaction by providing the following features:

- A console-based user interface to accept input from the user.
- The ability to read text files and compute basic statistics such as word count, line count, and character count.
- Identification of the most frequent words in the text.
- Implementation of n-gram analysis to study word sequences.
- A textual word cloud representation.
- Error handling for file access issues, invalid content or invalid input.
- Thorough testing using unit and property-based tests to ensure reliability.

2. Architecture of the Solution

The solution is structured in three main components:

1. **Main.hs**:
    - Serves as the entry point and controls program flow.
    - Interacts with the user via a console interface, collects file paths, and indirectly provides access to the main menu.
2. **Lib.hs**:
    - Includes the core functionality of the analyzer:
        - Checks if the file exists and if it contains data.
        - File reading and content parsing.
        - Main menu
        - Computing text statistics (words, lines, characters counts).
        - Finding most frequent words using frequency analysis.
        - Implementing n-gram analysis to word sequences.
        - Further error handling.
    - This file has a modular design to allow for easy reuse of its functions, which also helps to create a better experience for the user.
3. **Spec.hs**:
    - Contains automated tests to validate functionality.

    - Includes:
        - Unit tests to verify individual functions.
        - Property-based tests for broader validation.

3. Why Certain Architecture Decisions Were Made

Modularity:

- Separating the application into **Main.hs, Lib.hs,** and **Spec.hs** enables clear distinction of responsibilities.
- By separating the logic (**Lib.hs**) of the application from its entry point (**Main.hs**), the solution adheres to modularity principles. This makes the program easier to test, maintain, and extend. By using **Main.hs** as an entry point to our application, this also provides an easier and efficient way to test the whole project.

User Experience:

- The decision to implement a console-based user interface makes the program lightweight and accessible. The simplicity of text-based input/output allows it to be used on a wide range of systems without requiring a graphical user interface. The main menu and interactive prompts are intuitive and guide the user effectively. Error handling is used to ensure the user interacts with the application as designed.

Testability:

- To facilitate easier debugging and testing, more complex features such as n-gram analysis, word cloud generation, and identifying common words are split into smaller, focused functions. By isolating functionality, bugs can be identified in specific parts of the logic without affecting the rest of the program.
- These smaller, modular functions also allow their properties to be individually tested, ensuring correctness across a wide range of inputs.
- The use of **Spec.hs** ensures that every function can be tested in isolation. Property-based testing provides testing for correctness across a broad range of inputs, while unit-based testing is used for more common tests and edge cases.

Error Handling:

- Explicit handling of file access issues (e.g., missing or corrupted files) ensures that the application fails rather than crash. This was implemented to provide meaningful feedback to the user and maintain usability.

- Input validation ensures that invalid commands or data don't lead to unexpected behavio. It is designed so that the user can correct their input instead of closing the program directly.

Reusability:

- Breaking down the logic into smaller, reusable functions not only supports testability but also allows these components to be easily adapted for other projects.

Why use Data.Map for the frequency maps?

- Data.Map is implemented as a balanced binary tree, providing efficient operations for:
    - **Insertion/Update**: O(log n) for adding or updating an entry.
    - **Lookup**: O(log n) for querying the frequency of a word or n-gram.
- It is efficient for frequency analysis as these require constant updates and lookups.
- Provides access to Map.insertWith function, which makes it easy to efficiently update the count for an existing word or initialize it if it is not present.
- Provides uniqueness of the keys, which ensures no duplication of entries in the frequency maps.
- A list based approach was implemented initially, but it required manual searching for existing keys and therefore resulted in a O(n) operations for each word. This approach is also more prone to errors.

Functionality behind the creation of the textual word cloud:

- Transforms the input by calculating the frequency map of the content, and structuring it into tuples of words and importance levels. The list of words is generally split into three parts with different importance levels, which is used for the visualization of the cloud. The importance levels are determined based on the frequencies of the different words. With the initial implementation of the transform function there were occurrences when words with the same frequencies were distributed into different importance level classes. Therefore, a helper function was introduced which checks the bounds of bordering classes and ensures that if a lower level class has a top element with frequency that also occurs in the higher class, then the elements from the higher class have to be “dropped down”. This ensures that the words are properly displayed in the textual word cloud.
- The words are shuffled in order to ensure that the cloud is randomized.
- Groups of random lengths are created in an attempt to display a cloud-like structure. Spacing is also used for this purpose.

- The words are styled differently based on their importance levels in order to ensure distinction when presented.

4. Why Certain Libraries Were Chosen

- Data.List was used because it provides functions for list manipulation, which are fundamental for analyzing the text, generating the frequency maps, generating n-grams and creating the word clouds.
- Data.Char was used because it facilitates operations with characters such as case conversion or character filtering, which were needed to filter the data for proper analysis and for specific transformations.
- Control.Exception was used to handle runtime exceptions and for error reporting.
- System.Directory was used to provide the user with details about their current directory (as they might get an error when using file names out of the current direcory) and to check the existence of their file.
- Text.Read was used for validation of inputs.
- System.Random was used to add a randomization effect (adding spaces, shuffling the tuples, creating word groups of random lengths) when the cloud is created in an effort to replicate an actual cloud.
- Control.Monad was included specifically for the mapM and mapM_ functions as they build on the functionality of monadic I/O actions in the file.
- Data.Map was used for creating frequency maps.
- Test.Tasty and Hedgehog were used to provide extensive testing of specific unit tests and properties of the logic of the application.

4. Investigation of the performance

Overall the console application produces great results in terms of user interactability and error handling, by providing detailed content and feedback information. Specific libraries were used to improve the functionality and performance of the project, notably Data.Map, Data.List and Control.Monad. The deterministic functions which use randomization can naturally be omitted, but they are used to bring a creative aspect to the project and improve the visualization. The modularity of the logic of the project also allows for the extendability and the reusability. This is also useful during testing as it allows to test specific properties of the functionality, despite the I/O nature of the console application. The performance of the application is tested through extensive unit test cases and important property tests, to ensure it is working as intended.

Summary of Time Complexities:

- wordCount: O(n)
- linesCount: O(n)
- charCount: O(n)
- commonWordsHelper: O(n + m log m)
- nGramHelper:O(n + m log m)
- createCloud: O(n + m log m)
- shuffle: O(n)
- groupWords: O(n)
- generateNGram: O(n*m)
- transform: O(n + m log m)