Implementation Report: Module 4 - Text Processing (Summarization & Simplification)

Project: Inclusive Reading Aid for Dyslexic & Visually Impaired Users **Module:** Point 4 - Text Processing
Status: Completed **Date:** 2025-11-08

1.0 Introduction

This report details the technical methodology for constructing the **Text Processing** module. The primary objective of this module is to reduce the cognitive load of complex text, as specified in the project proposal.

This module provides two distinct backend functions:

1. **Summarization:** Condenses lengthy passages into a short summary.
2. **Simplification:** Replaces difficult words with simpler, more common alternatives.

Both operations are computationally intensive. A key part of this implementation was to run them in non-blocking worker threads to ensure the UI remains responsive, adopting the multi-threading pattern established in Module 3.

2.0 Core Technologies & Dependencies

This module introduces a suite of Natural Language Processing (NLP) libraries:

1. **sumy:** The primary library for extractive text summarization. The `LsaSummarizer` (Latent Semantic Analysis) was chosen for its effectiveness.
2. **nltk (Natural Language Toolkit):**
   - Used by `sumy` for tokenization and stop-word removal.
   - Used directly for simplification to `word_tokenize` text and `TreebankWordDetokenizer` to re-join it.
   - `nltk.corpus.wordnet` : Used to find synonyms (`synsets`) for difficult words.
3. **wordfreq:** Used to determine word "difficulty." The `zipf_frequency` function provides a logarithmic score of a word's commonness, which is ideal for setting a "difficulty threshold."
4. **PyQt6:**
   - `QGroupBox` , `QPushButton` : Used to create the "Text Tools" UI.
   - `QProgressDialog` : Used to provide visual feedback and block user input *only* while a task is running.
   - `pyqtSignal` : A new signal (`text_processed_signal`) was created to safely pass the processed text from the worker thread back to the main GUI thread.
5. **threading:** Used to run the summarization and simplification tasks in the background.

3.0 Implementation Methodology

3.1 GUI Construction (Text Tools)

1. **New UI Group:** A new `QGroupBox` titled "Text Tools" was added to the main vertical layout, placing it between the "Mode Selection" and "Controls" groups.
2. **Action Buttons:** Two `QPushButton` widgets, "Summarize Text" and "Simplify Text," were added to this group.
3. **Signal Connection:** The `clicked` signal of each button was connected to a new "starter" function (`start_summarize_task` and `start_simplify_task` , respectively).

3.2 Non-Blocking Task Management

To prevent UI freezes during NLP processing, a generic, reusable task-management system was created.

1. `start_processing_task()` **:** A new function that:
   - Takes the target worker function (e.g., `_summarize_thread`) as an argument.
   - Creates and displays a modal `QProgressDialog` to inform the user and prevent concurrent operations.
   - Starts a new `threading.Thread` , passing it the target function and the text from `self.text_area` .
2. `text_processed_signal` **:** A new `pyqtSignal(str, str)` was defined. It is emitted by worker threads upon task completion, carrying the `processed_text` and the `operation_type` .
3. `on_text_processed()` **Slot:** This main-thread slot is connected to the `text_processed_signal` . It is responsible for:
   - Closing the `QProgressDialog` .
   - Updating the `self.text_area` with the processed text.
   - Resetting the audio player (`stop_tts()`) and clearing highlights.

3.3 Summarization Logic

1. `start_summarize_task()` **:** The "Summarize" button's slot. It retrieves the current text and calls `start_processing_task` , passing it the `_summarize_thread` function.
2. `_summarize_thread()` **(Worker Thread):** This function performs the summarization:
   - It initializes a `PlaintextParser` , `Tokenizer` , `Stemmer` , and `LsaSummarizer` from the `sumy` library.
   - It requests a summary of a fixed length (e.g., 3 sentences) from the summarizer.
   - It joins the resulting sentences into a single string.
   - It emits the `text_processed_signal` with the summary.

3.4 Simplification Logic

1. `start_simplify_task()` **:** The "Simplify" button's slot. It calls `start_processing_task` , passing it the `_simplify_thread` function.
2. `_simplify_thread()` **(Worker Thread):** This function performs the word-replacement logic:
   - The text is tokenized using `nltk.word_tokenize` .
   - It iterates through each `word` .

- **Word Analysis:** It checks if the word is "difficult" by comparing its `wordfreq.zipf_frequency` to a predefined `DIFFICULTY_THRESHOLD` (e.g., `3.0`).
- **Synonym Search:** If a word is difficult, it queries `wordnet.synsets` for synonyms.
- **Synonym Selection:** It loops through the synonyms, checks their `zipf_frequency`, and selects the first synonym that is "simpler" (has a higher frequency) than the original word.
- **Reconstruction:** The list of words (with difficult ones replaced) is re-assembled into a string using `TreebankWordDetokenizer` to handle spacing correctly.
- It emits the `text_processed_signal` with the simplified text.

4.0 Summary of Features Implemented

- [x] **"Text Tools" UI:** A new UI section with "Summarize" and "Simplify" buttons is in place.
- [x] **Non-Blocking Processing:** Both tasks run in background threads, showing a "Processing..." dialog and keeping the UI responsive.
- [x] **Summarization:** The app can successfully generate a 3-sentence summary of the loaded text using `sumy`.
- [x] **Simplification:** The app can analyze text and replace low-frequency (difficult) words with higher-frequency (simpler) synonyms using `nltk`, `wordnet`, and `wordfreq`.

5.0 Conclusion

Module 4 is fully implemented, providing powerful text-processing capabilities as requested. The integration of a non-blocking task manager is a key success, ensuring a smooth user experience even during complex NLP operations.

The application is now feature-complete, with the exception of the final module, **Module 5 (Translation)**.