Creating an Extractive Text Summarizer in Java

Juan Felix Kusnadi (2802536386)

Jonathan Mulyono (2802537054)

Iglesias Sidharta Handjo (2802530621)

Object Oriented Programming

BINUS University International

Introduction

A summary is a brief description that gives the main facts or ideas of something (Cambridge Dictionary, n.d.). Text summarization plays a significant role in a world with vast amounts of information, enabling humans to understand the importance of a lengthy text within a shorter period. Our interest in text summarization started from the desire to explore natural language processing and to deepen our understanding of object-oriented programming and the characteristics of different data structures. As such, we will develop an extractive text summarizer based on the Term Frequency-Inverse Document Frequency (TF-IDF) algorithm, a widely used statistical method for ranking the importance of sentences. This project aims to produce a robust and modular extractive text summarizer. As large volumes of unstructured information continue to flood, this study offers insight to anyone willing to build an efficient summarization tool.

Problem Description

In the current modern world, vast amounts of textual data are generated every day. This makes it increasingly difficult for humans to process and extract key information efficiently. This overloading of information often creates a need for summarization tools that shorten texts into a meaningful, smaller piece without losing the essential content. Extractive summarization, which selects and compiles sentences based on their importance (Payong, 2024), offers a practical and effective solution. However, implementing this solution efficiently requires choosing the correct type of data structure to yield a balance between accuracy and performance, ensuring that the tool runs on varying text sizes (Ardash, 2024).

Approach

Term Frequency-Inverse Document Frequency (TF-IDF) is a statistical approach used in natural language processing (NLP) to evaluate the importance of a word within a document relative to a collection of documents (corpus) (Jain, 2025). The key workings of TF-IDF state that:

1. Words that appear more often in a document are more important (higher Term Frequency) (Jain, 2025).
2. Words that are common across different documents in the corpus are less important as they are less useful for distinguishing different documents (lower Inverse Document Frequency) (Jain, 2025).

Unlike word frequency, TF-IDF balances common and rare terms to grasp the most important and meaningful words, hence helping to produce a relevant summary. In the context of this research, the algorithm will be modified to account for a single document or text input only, instead of a collection of documents. As such, calculating the Inverse Document Frequency will be based on the number of sentences from the input.

Methodology

Preparing the text input

1. Tokenize the input into distinct sentences, and tokenize each sentence into words.
2. Remove common stop words such as “the”, “is”, “in”, “and”, etc., that are meaningless in identifying important terms.
3. Convert all words into lowercase for case-insensitive comparison.
4. Reduce words to their root form using stemming.

Calculate Term Frequency (TF)

1. Calculate w, the number of times word w appears.

2. Calculate S, the number of words in sentence S.
3. Calculate Term Frequency by $TF = \frac{w}{S}$.

Calculate Inverse Document Frequency (IDF)

1. Calculate N, the total number of sentences.
2. Calculate $SF(w)$, the sentence frequency for the number of sentences containing the word w.
3. Calculate Inverse Document Frequency by $IDF = log(\frac{N}{SF(w)})$.

Creating a summary

1. Calculate TF-IDF by $TF \times IDF$.
2. Store this key (word) and value (TF-IDF score) pair, and repeat this calculation for all words in all sentences.
3. Rank the importance of every sentence by finding the sum of the TF-IDF scores of all words in that sentence.
4. Select the top 20%-40% of sentences from the total number of sentences based on the user input (Demilie, 2022).

Data Structures Used

1. Map

   A Map is a tool that stores pairs of data (key-value), like a word and its number. It helps the program quickly find the value that belongs to a specific word or sentence. In this project, the Map is used to store the score of each word and sentence. This is helpful because it makes it easy to look up and update the score when deciding which sentences are most important for the summary.

2. HashSet

   A HashSet is a list that only keeps unique items and doesn't allow duplicates. It can quickly check if something is already in the list. In this program, HashSet is used to store stopwords (like “the” or “is”), honorifics (like “Mr.” or “Dr.”), and words that only appear once. It’s useful because the program needs to skip unimportant words and avoid checking the same word multiple times.

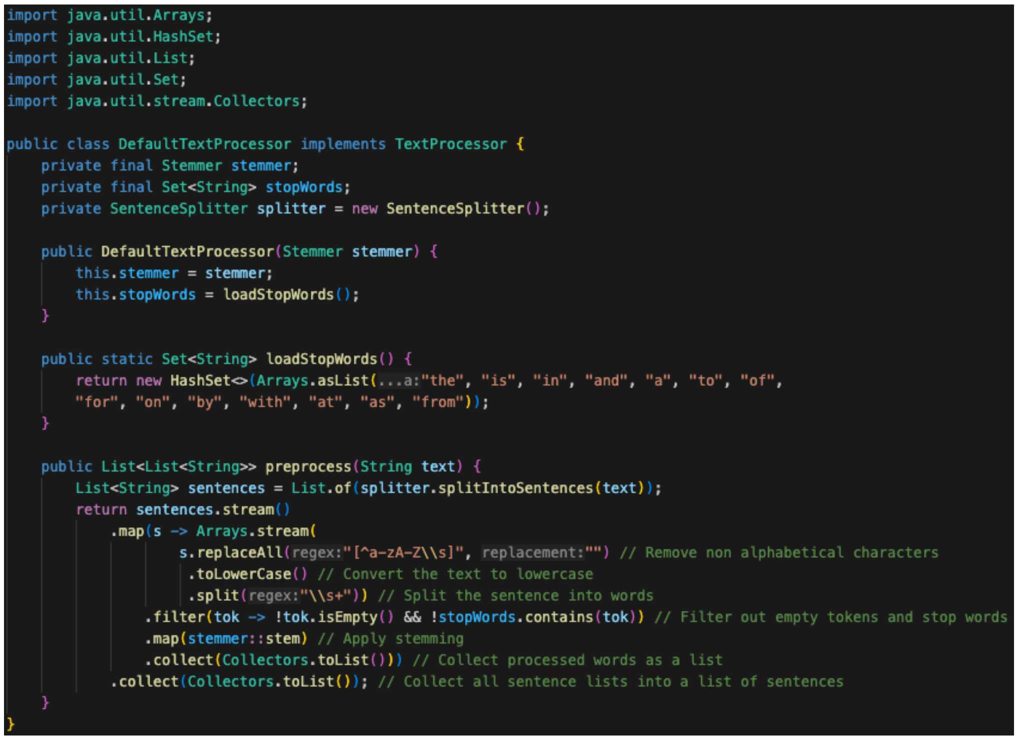
3. ArrayList

   An ArrayList is a flexible list that can grow or shrink depending on how much data it holds. It keeps the items in the order they were added and allows access by number (like position 1, 2, 3, etc.). In this project, the ArrayList is used to store all the sentences from the input text and their scores. This is a good fit because the program needs to keep the order of sentences and go through them one by one to create the summary.

Preliminary Testing

A preliminary test was conducted on the command line version of the program for various types of input text, including informative articles, narrative texts, and news. While there was no significant problem found for informative and narrative texts, a major issue regarding the splitting of sentences was found for news articles. As news articles often contain honorifics, such as Mr, Ms, Dr, Prof, etc., splitting the input only using regular expressions is not accurate. The program recognizes the “.” after the honorifics as the end of a sentence. As such, a customized sentence splitter was built based on three main strategies: skipping closing quotes, checking whether the next part after a full stop looks like a new sentence (either opening quotes or capital letters), and checking whether a word is an honorific (Appendix 1).

Application using Java

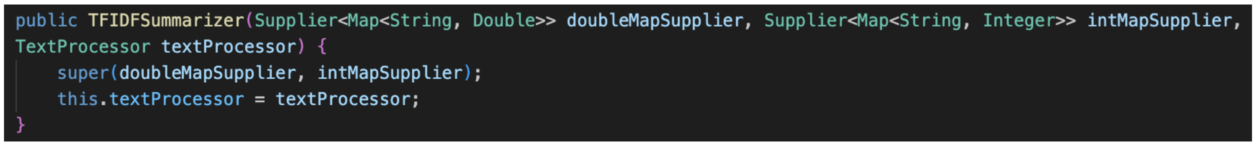


```java
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class DefaultTextProcessor implements TextProcessor {
    private final Stemmer stemmer;
    private final Set<String> stopWords;
    private SentenceSplitter splitter = new SentenceSplitter();

    public DefaultTextProcessor(Stemmer stemmer) {
        this.stemmer = stemmer;
        this.stopWords = loadStopWords();
    }

    public static Set<String> loadStopWords() {
        return new HashSet<>(Arrays.asList(...a:"the", "is", "in", "and", "a", "to", "of",
        "for", "on", "by", "with", "at", "as", "from"));
    }

    public List<List<String>> preprocess(String text) {
        List<String> sentences = List.of(splitter.splitIntoSentences(text));
        return sentences.stream()
            .map(s -> Arrays.stream(
                    s.replaceAll(regex:"[^a-zA-Z\\s]", replacement:"") // Remove non alphabetical characters
                     .toLowerCase() // Convert the text to lowercase
                     .split(regex:"\\s+")) // Split the sentence into words
                .filter(tok -> !tok.isEmpty() && !stopWords.contains(tok)) // Filter out empty tokens and stop words
                .map(stemmer::stem) // Apply stemming
                .collect(Collectors.toList())) // Collect processed words as a list
            .collect(Collectors.toList()); // Collect all sentence lists into a list of sentences
    }
}
```

Figure 1. DefaultTextProcessor class.

Preparing the text input was done by the DefaultTextProcessor class, which implements the TextProcessor interface (Appendix 2). After the input text has been split by an instance of the SentenceSplitter class, and stopwords have been removed by filtering based on the set generated by the loadStopWords() method, stemming is handled by the SnowballStemmer class (Appendix 3), implementing the Stemmer interface (Appendix 4). SnowballStemmer uses an external implementation of the Porter stemming method to reduce words into their root form. This stemming algorithm uses NLP to remove common endings of a word, such as from running to run and ran to run. This ensures that different forms of a word are treated as a single word.

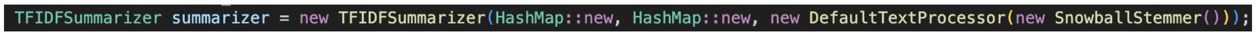
After the input text is processed, sentences will be extracted to generate a summary based on the TF-IDF algorithm. This is done by the TFIDFSummarizer concrete class (Appendix 5), which extends the AbstractSummarizer abstract class (Appendix 6).



```
public TFIDFSummarizer(Supplier<Map<String, Double>> doubleMapSupplier, Supplier<Map<String, Integer>> intMapSupplier,
TextProcessor textProcessor) {
    super(doubleMapSupplier, intMapSupplier);
    this.textProcessor = textProcessor;
}
```

Figure 2. TFIDFSummarizer constructor.

As the aim of this study is to create a flexible extractive text summarizer, the constructor of the TDIDFSummarizer class uses the Supplier<T> functional interface, which returns type T when get() is called. The use of Supplier creates flexibility for this class, allowing it to seamlessly use any implementation of Map without any change of code. As shown by the figure below, the constructor initializes a new instance of TFIDFSummarizer that uses two HashMaps, making it more reusable and testable.



```
TFIDFSummarizer summarizer = new TFIDFSummarizer(HashMap::new, HashMap::new, new DefaultTextProcessor(new SnowballStemmer()));
```

Figure 3. TFIDFSummarizer constructor.

Finally, a Graphical User Interface (GUI) version of the program is created using JavaFX to allow an easier testing process, which involves large input and output sizes. Furthermore, several features are also added to enhance user experience, which include customizing the summary to be in paragraph or bullet points, showing word count for input and summarized text, a copy and paste button, a clear button, customizing the length of the summary, and statistics of the summary. Figure 4 shows the GUI window of this program.

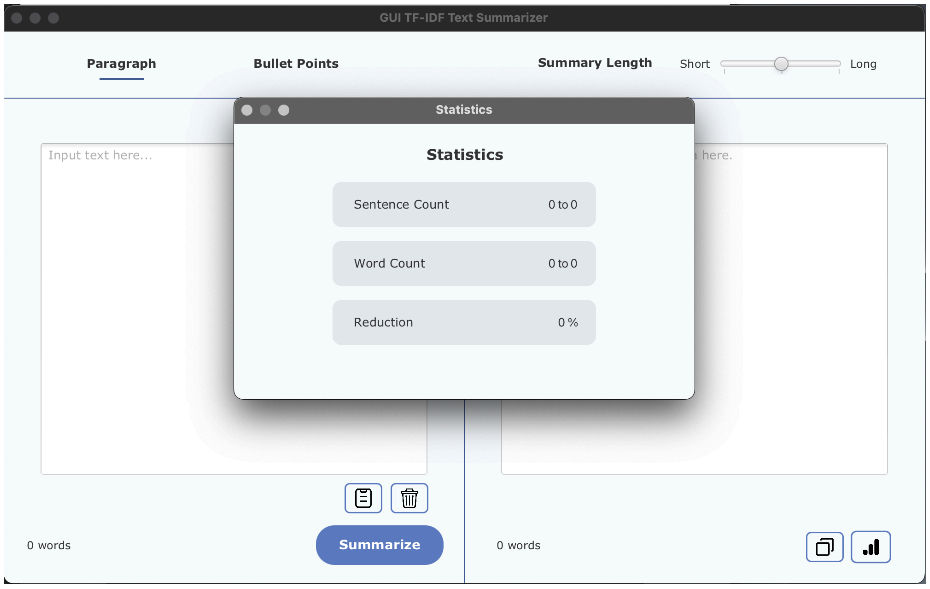
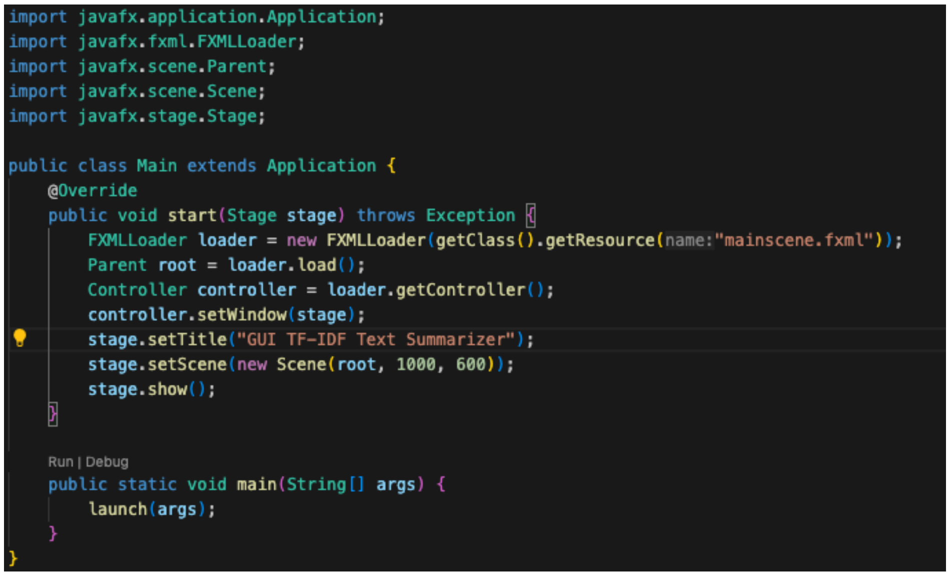


Figure 4. GUI program window.



```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        FXMLLoader loader = new FXMLLoader(getClass().getResource(name:"mainscene.fxml"));
        Parent root = loader.load();
        Controller controller = loader.getController();
        controller.setWindow(stage);
        stage.setTitle("GUI TF-IDF Text Summarizer");
        stage.setScene(new Scene(root, 1000, 600));
        stage.show();
    }

    Run | Debug
    public static void main(String[] args) {
        launch(args);
    }
}
```

Figure 5. Class Main.java.

To run a GUI program using JavaFX in Java, the class Main must extend the Application class. In overriding the default start() method, the UI definition of the program must be loaded from an FXML file via FXMLLoader. Next, retrieve the Controller instance that the loader created, so that the primary Stage can be passed into setWindow(). Set up the window title, size, and call show() to display the window. Finally, add launch(args) inside main().

JavaFX mainly works by separating the UI from the algorithm that supports the overall GUI by defining a separate FXML file containing UI definitions only, such as Label, TextArea, Button, etc. In this program, an FXML file named mainscene is created (Appendix 7). It points to the controller class Controller.java and the stylesheet mainscene.css to connect logic with UI elements. Lastly, within the initialize() method in Controller.java, it creates listeners to the slider and text area properties to update word counts and summary length, and handle additional buttons, including copy, paste, clear, and statistics buttons.

Class Diagram

The figure below shows the simplified version of the UML class diagram of this program without including class attributes and methods. The complete UML class diagram is available in Appendix 8.

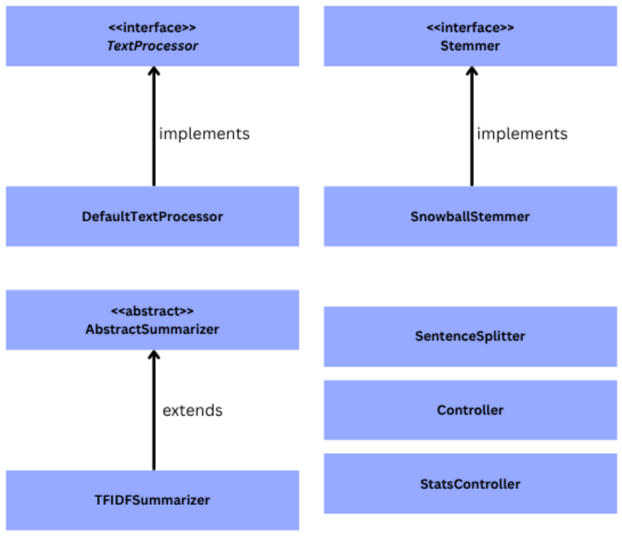


Figure 6. Simplified UML class diagram.

The 4 Principles of Object Oriented Programming

1. Encapsulation

   Encapsulation refers to the principle of bundling data and methods that operate on the data into a single, self-contained unit, often known as a class. This principle prevents unwanted external access and modifications. In Controller.java, FXML UI components (inputArea, summarizeBtn, lengthSlider, etc.) and attributes such as summaryRatio and paragraphFormat are all declared as private. External code outside of Controller.java is unable to reach and manipulate them directly without interacting with the public methods. In DefaultTextProcessor.java, stopWords and stemmer are also private attributes. When preprocess(text) is called, outside codes do not see how the words are filtered or stemmed.

2. Abstraction

   Abstraction refers to the concept of hiding complex implementation details and only showing the necessary features of an object. It simplifies complex details by only showing a simplified view to the users. In AbstractSummarizer.java, it defines summarize(text, ratio) without explicitly defining the details of the summarizing algorithm. Furthermore, TextProcessor and Stemmer interfaces only tell the operations available, like preprocess and stem, without describing how they work. In this context, abstraction is particularly beneficial when a summarizer is built based on a different summarizing or stemming algorithm.

3. Inheritance

   Inheritance refers to a concept that allows a class (subclass) to inherit the properties and methods from another class, known as the superclass. This creates an “is-a” relationship, showing that the subclass “is a” superclass. This concept is shown by the inheritance of TFIDFSummarizer from AbstractSummarizer. It inherits the doubleMapSupplier and intMapSupplier attributes, and it needs to implement the summarize() method. As AbstractSummarizer does not provide a concrete implementation of summarize(), every subclass that inherits it is able to focus on its own summarizing algorithm.

4. Polymorphism

Polymorphism refers to a principle that allows objects from different classes to be treated in a uniform way, even if they have unique implementations. In simpler words, polymorphism means “many forms”, allowing an entity to have multiple behaviours or representations. In this project, polymorphism is demonstrated through interchangeable interface and abstract references. In Controller.initialize(), the following line of code is written.

$$TFIDFSummarizer\ summarizer\ =\ new\ TFIDFSummarizer(...);$$

However, it is also possible to write:

$$AbstractSummarizer\ summarizer\ =\ new\ TFIDFSummarizer(...);$$

This is due to the fact that TFIDFSummarizer inherits AbstractSummarizer. Likewise, DefaultTextProcessor takes a stemmer interface when instantiating an object, and it currently takes SnowballStemmer. However, it is possible to pass any other Stemmer implementations. In addition, polymorphism is also shown through the overriding of methods. As TFIDFSummarizer extends AbstractSummarizer, it overrides summarize() to have its own implementation.

Program GUI Walkthrough

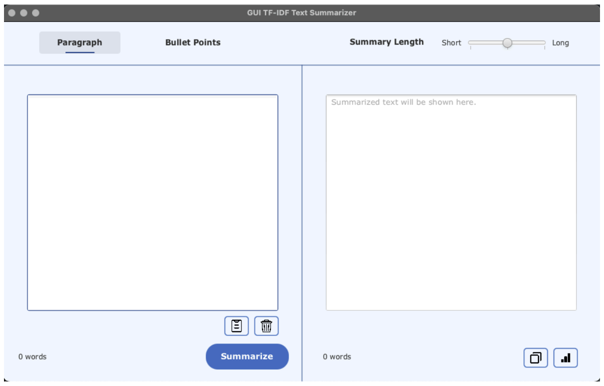


Figure 7. Paragraph as chosen format.

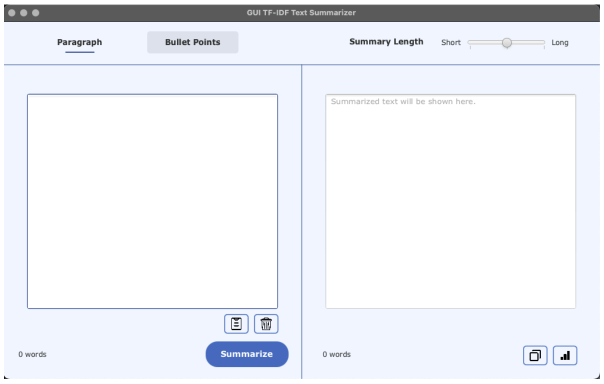


Figure 8. Bullet points as chosen format.

The GUI version of this program allows users to choose the format of the output, either in paragraph or in bullet points, to improve readability. To choose the desired format, the user needs to click on either “Paragraph” or “Bullet Points”, and the chosen format is marked by the blue line underneath. The default format is set to “Paragraph”.

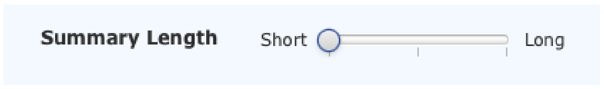


Figure 9. Short summary length.

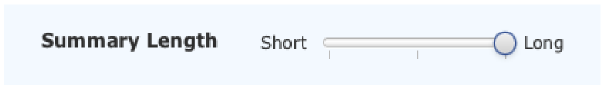


Figure 10. Long summary length.

The GUI version also includes a slider that allows users to choose how long they want their summary to be. It could range from short to long. As shown in Figure 9, by moving the slider to the left it will give a short summary. When the slider is moved towards the “Short” end, the program will generate a more concise summary, which will be useful for quick overviews. On the other hand, if users move the slider towards the “Long” end (Figure 10), the program will generate a more detailed summary that retains more of the original content. This function helps users pick a summary length that works best for them.

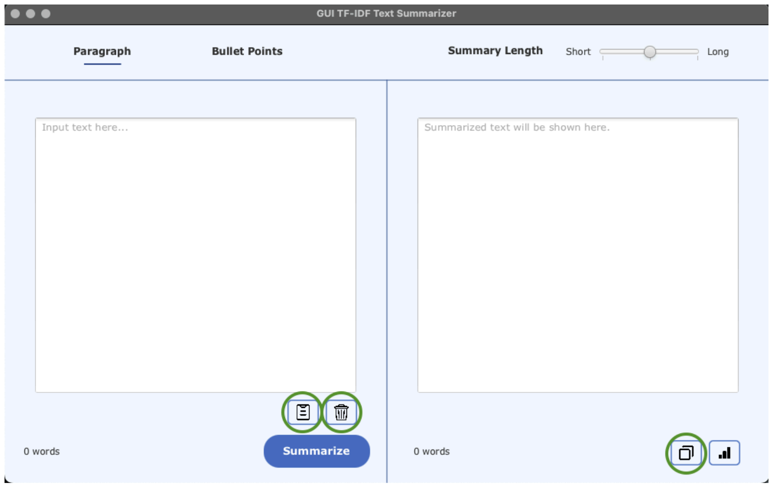


Figure 11. Copy, paste, and delete buttons.

The program also includes helpful buttons for easier use. As shown in Figure 11, users can copy, paste, or delete text using the buttons under the text boxes. The copy and paste button help to move the text quickly, and also there is also a delete button to clear the text box. These functions help to make the program faster and also more convenient to use.

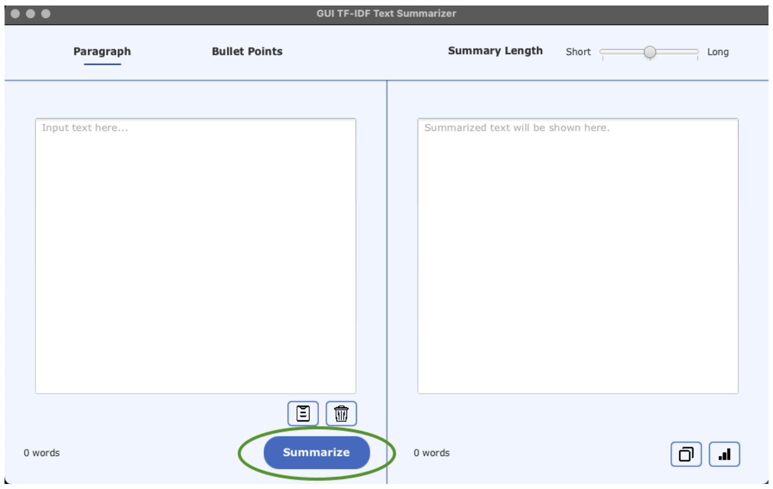


Figure 12. Summarize button.

There is also a summarize button, which is the core function of the program. As shown in Figure 12, once the user enters or pastes text into the input area and then clicks the summarize button, the program will generate a summary. The summarized content is then displayed in the output box on the right. This feature allows users to quickly and easily turn longer texts into shorter texts and also more readable with just one step.

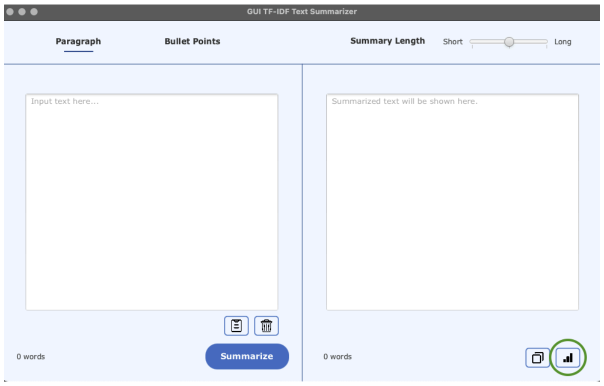
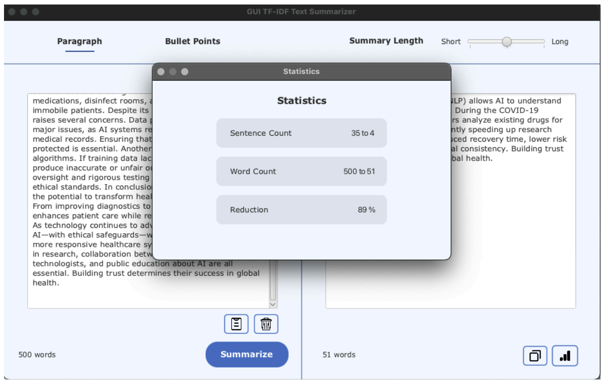


Figure 13. Statistics button.



Statistics

Sentence Count	35 to 4
Word Count	500 to 51
Reduction	89 %

medications, disinfect rooms,
immobile patients. Despite its
raises several concerns. Data
major issues, as AI systems re
medical records. Ensuring that
protected is essential. Another
algorithms. If training data lac
produce inaccurate or unfair o
oversight and rigorous testing
ethical standards. In conclusio
the potential to transform heal
From improving diagnostics to
enhances patient care while re
As technology continues to adv
AI—with ethical safeguards—w
more responsive healthcare sy
in research, collaboration betw
technologists, and public education about AI are all
essential. Building trust determines their success in global
health.

LP) allows AI to understand
During the COVID-19
rs analyze existing drugs for
ntly speeding up research
ced recovery time, lower risk
al consistency. Building trust
bal health.

Figure 14. Statistics window.

The program also has a statistics button, as shown in Figure 13, located at the bottom-right corner of the window, marked with a bar chart icon. When the button is clicked, this button opens a new window, as demonstrated in Figure 14. This pop-up window in Figure 14 provides some useful metrics, including the original and summarized sentence count, the word count before and after summarization, and the overall reduction percentage. These statistics help users evaluate the effectiveness and conciseness of the generated summary using this program.

References

Ardash. (2024, October 5). DSA for developers: Why data structures matter in real projects. *Medium*. https://medium.com/@adarshgs.909/dsa-for-developers-why-data-structures-matter-in-real-projects-9b373f706a17

Cambridge Dictionary. (n.d.). Meaning of summary in English. *Cambridge Dictionary*. https://dictionary.cambridge.org/dictionary/english/summary

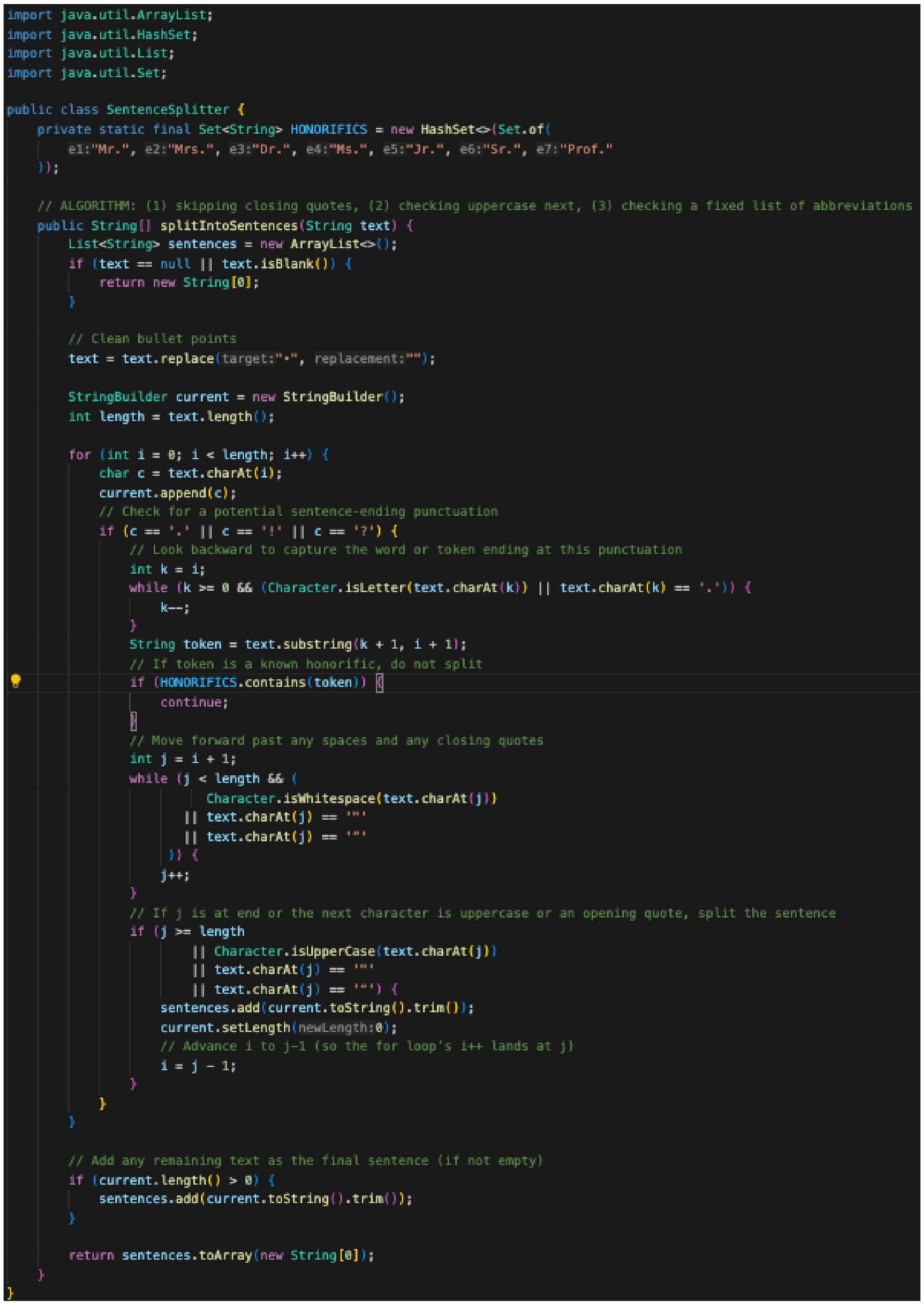
Demilie, W. (2022, September 13). Comparative analysis of automated text summarization techniques: The case of Ethiopian languages. *Wireless Communication and Mobile Computing*, *2022*, 13-15. https://doi.org/10.1155/2022/3282127

Jain, S. (2025, February 7). Understanding TF-IDF (Term Frequency-Inverse Document Frequency). *GeeksforGeeks*. https://www.geeksforgeeks.org/understanding-tf-idf-term-frequency-inverse-document-frequency/

Payong, A. (2024, October 25). Introduction to extractive and abstractive summarization techniques. *DigitalOcean*. https://www.digitalocean.com/community/tutorials/extractive-and-abstractive-summarization-techniques

Appendices

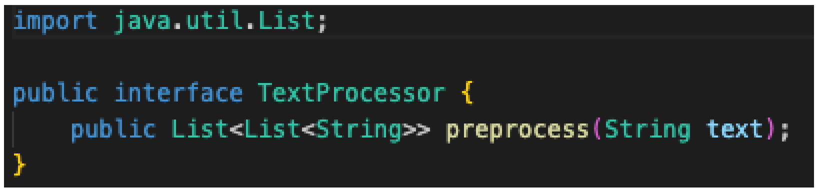
Appendix 1



```java
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class SentenceSplitter {
    private static final Set<String> HONORIFICS = new HashSet<>(Set.of(
        e1:"Mr.", e2:"Mrs.", e3:"Dr.", e4:"Ms.", e5:"Jr.", e6:"Sr.", e7:"Prof."
    ));

    // ALGORITHM: (1) skipping closing quotes, (2) checking uppercase next, (3) checking a fixed list of abbreviations
    public String[] splitIntoSentences(String text) {
        List<String> sentences = new ArrayList<>();
        if (text == null || text.isBlank()) {
            return new String[0];
        }

        // Clean bullet points
        text = text.replace(target:"•", replacement:"");

        StringBuilder current = new StringBuilder();
        int length = text.length();

        for (int i = 0; i < length; i++) {
            char c = text.charAt(i);
            current.append(c);
            // Check for a potential sentence-ending punctuation
            if (c == '.' || c == '!' || c == '?') {
                // Look backward to capture the word or token ending at this punctuation
                int k = i;
                while (k >= 0 && (Character.isLetter(text.charAt(k)) || text.charAt(k) == '.')) {
                    k--;
                }
                String token = text.substring(k + 1, i + 1);
                // If token is a known honorific, do not split
                if (HONORIFICS.contains(token)) {
                    continue;
                }
                // Move forward past any spaces and any closing quotes
                int j = i + 1;
                while (j < length && (
                        Character.isWhitespace(text.charAt(j))
                        || text.charAt(j) == '”'
                        || text.charAt(j) == '“'
                      )) {
                    j++;
                }
                // If j is at end or the next character is uppercase or an opening quote, split the sentence
                if (j >= length
                         || Character.isUpperCase(text.charAt(j))
                         || text.charAt(j) == '”'
                         || text.charAt(j) == '“') {
                    sentences.add(current.toString().trim());
                    current.setLength(newLength:0);
                    // Advance i to j-1 (so the for loop's i++ lands at j)
                    i = j - 1;
                }
            }
        }

        // Add any remaining text as the final sentence (if not empty)
        if (current.length() > 0) {
            sentences.add(current.toString().trim());
        }

        return sentences.toArray(new String[0]);
    }
}
```

Appendix 1. SentenceSplitter class.

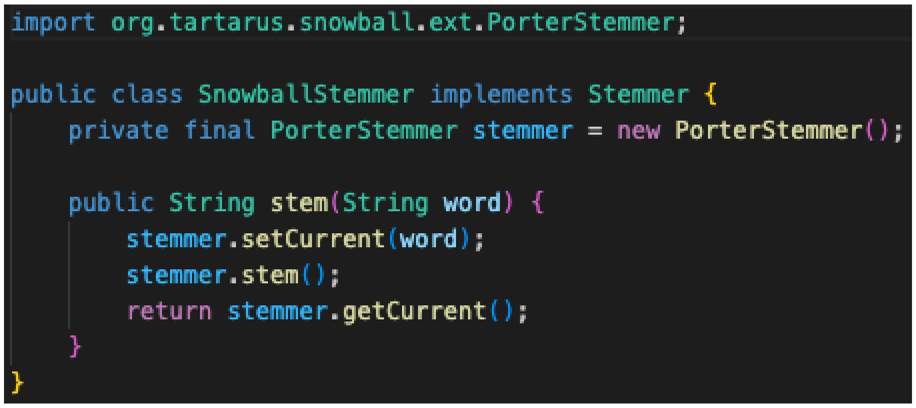
Appendix 2



```java
import java.util.List;

public interface TextProcessor {
    public List<List<String>> preprocess(String text);
}
```

Appendix 2. TextProcessor interface.

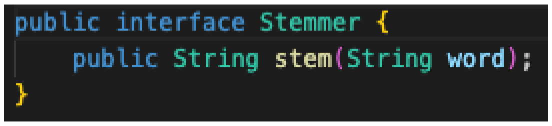
Appendix 3



```java
import org.tartarus.snowball.ext.PorterStemmer;

public class SnowballStemmer implements Stemmer {
    private final PorterStemmer stemmer = new PorterStemmer();

    public String stem(String word) {
        stemmer.setCurrent(word);
        stemmer.stem();
        return stemmer.getCurrent();
    }
}
```

Appendix 3. SnowballStemmer class.

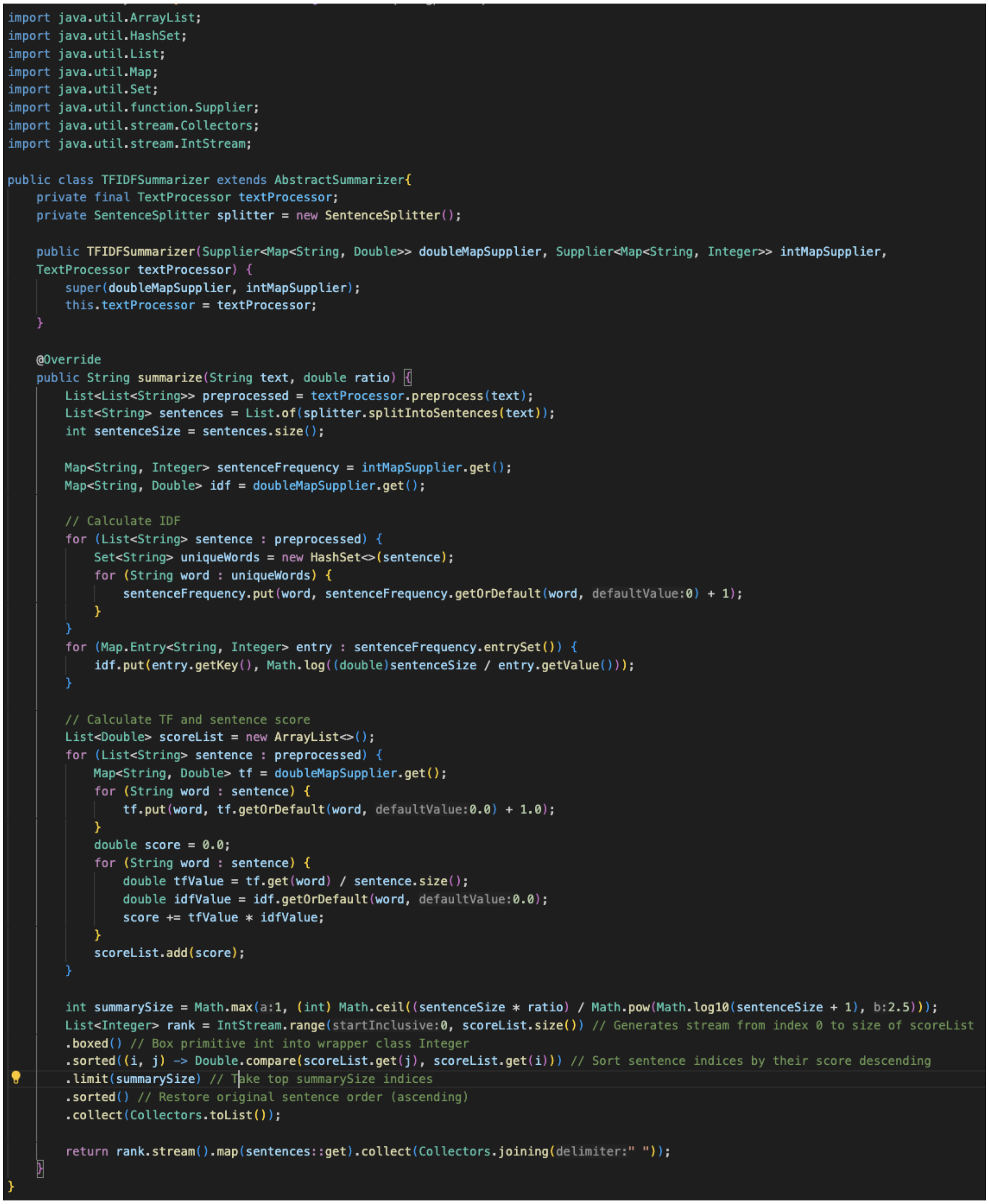
Appendix 4



```java
public interface Stemmer {
    public String stem(String word);
}
```

Appendix 4. Stemmer interface.

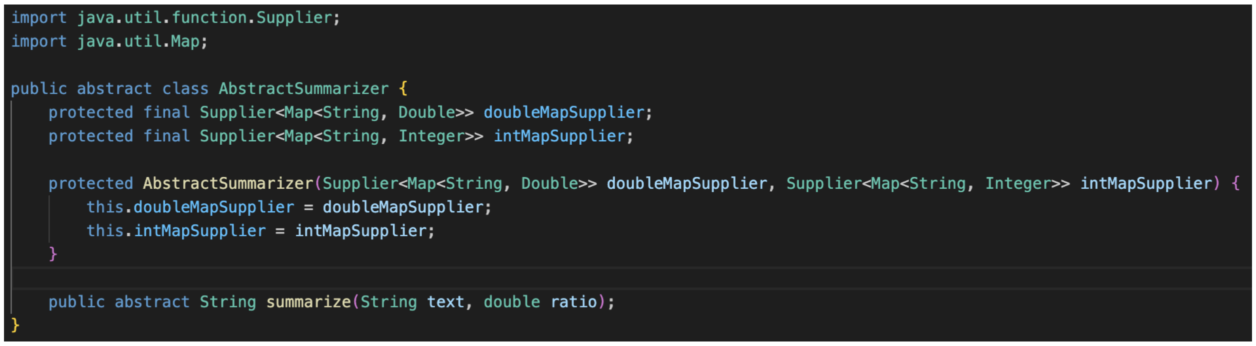
Appendix 5



```java
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.function.Supplier;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class TFIDFSummarizer extends AbstractSummarizer{
    private final TextProcessor textProcessor;
    private SentenceSplitter splitter = new SentenceSplitter();

    public TFIDFSummarizer(Supplier<Map<String, Double>> doubleMapSupplier, Supplier<Map<String, Integer>> intMapSupplier,
    TextProcessor textProcessor) {
        super(doubleMapSupplier, intMapSupplier);
        this.textProcessor = textProcessor;
    }

    @Override
    public String summarize(String text, double ratio) {
        List<List<String>> preprocessed = textProcessor.preprocess(text);
        List<String> sentences = List.of(splitter.splitIntoSentences(text));
        int sentenceSize = sentences.size();

        Map<String, Integer> sentenceFrequency = intMapSupplier.get();
        Map<String, Double> idf = doubleMapSupplier.get();

        // Calculate IDF
        for (List<String> sentence : preprocessed) {
            Set<String> uniqueWords = new HashSet<>(sentence);
            for (String word : uniqueWords) {
                sentenceFrequency.put(word, sentenceFrequency.getOrDefault(word, defaultValue:0) + 1);
            }
        }
        for (Map.Entry<String, Integer> entry : sentenceFrequency.entrySet()) {
            idf.put(entry.getKey(), Math.log((double)sentenceSize / entry.getValue()));
        }

        // Calculate TF and sentence score
        List<Double> scoreList = new ArrayList<>();
        for (List<String> sentence : preprocessed) {
            Map<String, Double> tf = doubleMapSupplier.get();
            for (String word : sentence) {
                tf.put(word, tf.getOrDefault(word, defaultValue:0.0) + 1.0);
            }
            double score = 0.0;
            for (String word : sentence) {
                double tfValue = tf.get(word) / sentence.size();
                double idfValue = idf.getOrDefault(word, defaultValue:0.0);
                score += tfValue * idfValue;
            }
            scoreList.add(score);
        }

        int summarySize = Math.max(a:1, (int) Math.ceil((sentenceSize * ratio) / Math.pow(Math.log10(sentenceSize + 1), b:2.5)));
        List<Integer> rank = IntStream.range(startInclusive:0, scoreList.size()) // Generates stream from index 0 to size of scoreList
        .boxed() // Box primitive int into wrapper class Integer
        .sorted((i, j) -> Double.compare(scoreList.get(j), scoreList.get(i))) // Sort sentence indices by their score descending
        .limit(summarySize) // Take top summarySize indices
        .sorted() // Restore original sentence order (ascending)
        .collect(Collectors.toList());

        return rank.stream().map(sentences::get).collect(Collectors.joining(delimiter:" "));
    }
}
```

Appendix 5. TFIDFSummarizer class.

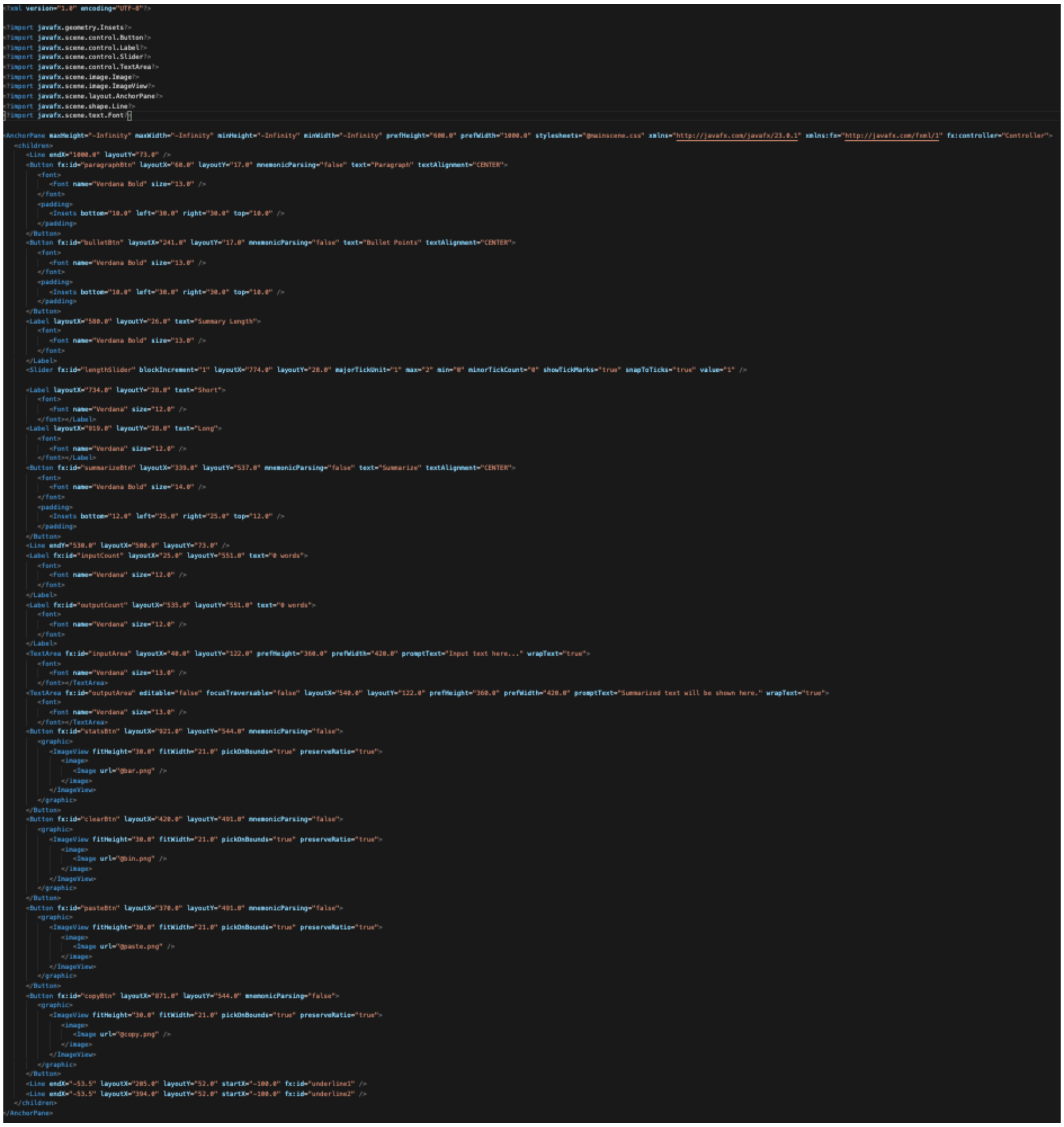
Appendix 6



```java
import java.util.function.Supplier;
import java.util.Map;

public abstract class AbstractSummarizer {
    protected final Supplier<Map<String, Double>> doubleMapSupplier;
    protected final Supplier<Map<String, Integer>> intMapSupplier;

    protected AbstractSummarizer(Supplier<Map<String, Double>> doubleMapSupplier, Supplier<Map<String, Integer>> intMapSupplier) {
        this.doubleMapSupplier = doubleMapSupplier;
        this.intMapSupplier = intMapSupplier;
    }

    public abstract String summarize(String text, double ratio);
}
```

Appendix 6. AbstractSummarizer abstract class.

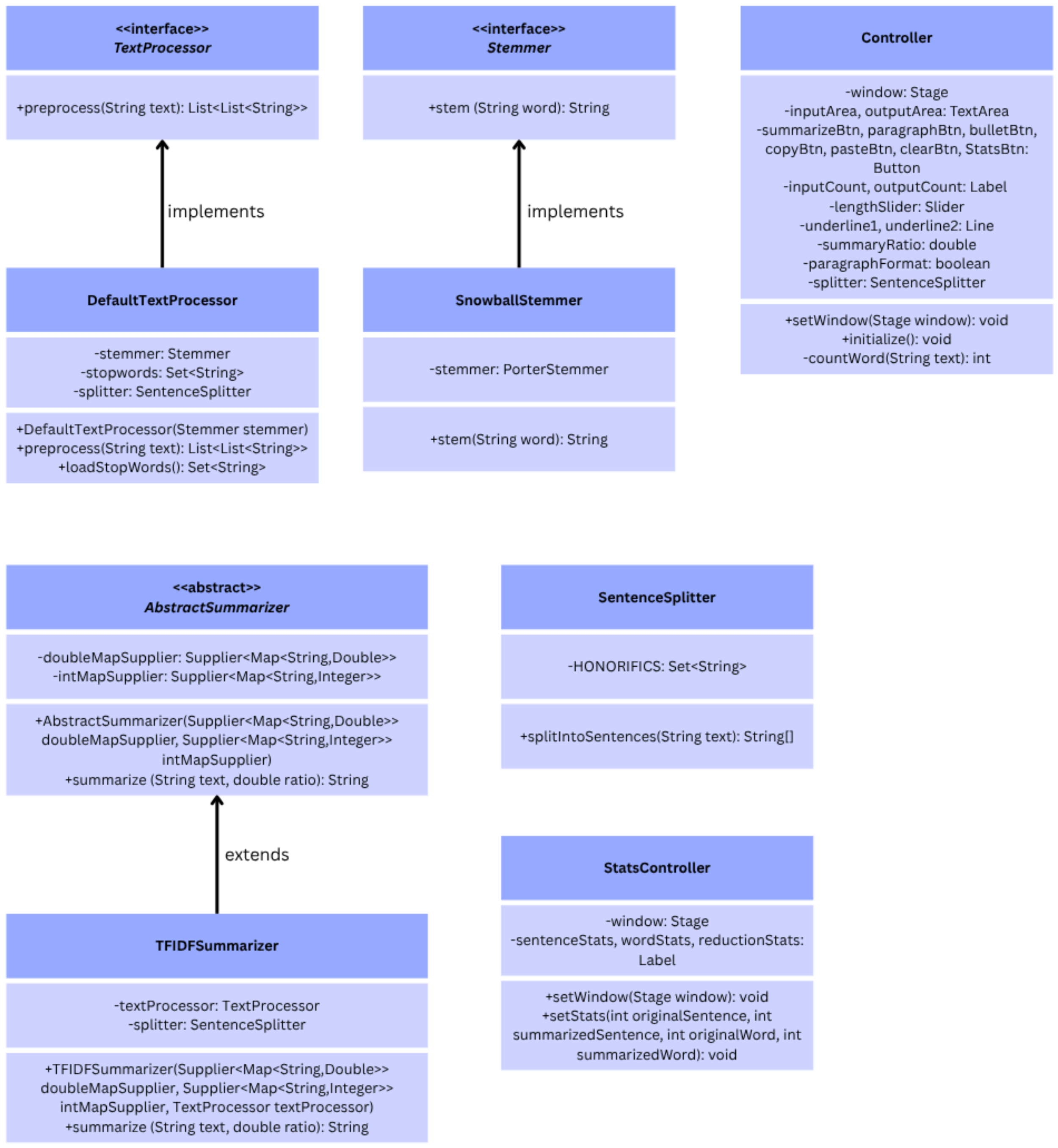
Appendix 7



```xml
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Slider?>
<?import javafx.scene.control.TextArea?>
<?import javafx.scene.image.Image?>
<?import javafx.scene.image.ImageView?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.shape.Line?>
<?import javafx.scene.text.Font?>

<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" prefHeight="600.0" prefWidth="1000.0" stylesheets="@mainscene.css" xmlns="http://javafx.com/javafx/23.0.1" xmlns:fx="http://javafx.com/fxml/1" fx:controller="Controller">
   <children>
      <Line endX="1000.0" layoutY="73.0" />
      <Button fx:id="paragraphBtn" layoutX="60.0" layoutY="17.0" mnemonicParsing="false" text="Paragraph" textAlignment="CENTER">
         <font>
            <Font name="Verdana Bold" size="13.0" />
         </font>
         <padding>
            <Insets bottom="10.0" left="30.0" right="30.0" top="10.0" />
         </padding>
      </Button>
      <Button fx:id="bulletBtn" layoutX="241.0" layoutY="17.0" mnemonicParsing="false" text="Bullet Points" textAlignment="CENTER">
         <font>
            <Font name="Verdana Bold" size="13.0" />
         </font>
         <padding>
            <Insets bottom="10.0" left="30.0" right="30.0" top="10.0" />
         </padding>
      </Button>
      <Label layoutX="580.0" layoutY="26.0" text="Summary Length">
         <font>
            <Font name="Verdana Bold" size="13.0" />
         </font>
      </Label>
      <Slider fx:id="lengthSlider" blockIncrement="1" layoutX="774.0" layoutY="28.0" majorTickUnit="1" max="2" min="0" minorTickCount="0" showTickMarks="true" snapToTicks="true" value="1" />

      <Label layoutX="734.0" layoutY="28.0" text="Short">
         <font>
            <Font name="Verdana" size="12.0" />
         </font></Label>
      <Label layoutX="919.0" layoutY="28.0" text="Long">
         <font>
            <Font name="Verdana" size="12.0" />
         </font></Label>
      <Button fx:id="summarizeBtn" layoutX="339.0" layoutY="537.0" mnemonicParsing="false" text="Summarize" textAlignment="CENTER">
         <font>
            <Font name="Verdana Bold" size="14.0" />
         </font>
         <padding>
            <Insets bottom="12.0" left="25.0" right="25.0" top="12.0" />
         </padding>
      </Button>
      <Line endY="530.0" layoutX="500.0" layoutY="73.0" />
      <Label fx:id="inputCount" layoutX="25.0" layoutY="551.0" text="0 words">
         <font>
            <Font name="Verdana" size="12.0" />
         </font>
      </Label>
      <Label fx:id="outputCount" layoutX="535.0" layoutY="551.0" text="0 words">
         <font>
            <Font name="Verdana" size="12.0" />
         </font>
      </Label>
      <TextArea fx:id="inputArea" layoutX="40.0" layoutY="122.0" prefHeight="360.0" prefWidth="420.0" promptText="Input text here..." wrapText="true">
         <font>
            <Font name="Verdana" size="13.0" />
         </font></TextArea>
      <TextArea fx:id="outputArea" editable="false" focusTraversable="false" layoutX="540.0" layoutY="122.0" prefHeight="360.0" prefWidth="420.0" promptText="Summarized text will be shown here." wrapText="true">
         <font>
            <Font name="Verdana" size="13.0" />
         </font></TextArea>
      <Button fx:id="statsBtn" layoutX="921.0" layoutY="544.0" mnemonicParsing="false">
         <graphic>
            <ImageView fitHeight="30.0" fitWidth="21.0" pickOnBounds="true" preserveRatio="true">
               <image>
                  <Image url="@bar.png" />
               </image>
            </ImageView>
         </graphic>
      </Button>
      <Button fx:id="clearBtn" layoutX="420.0" layoutY="491.0" mnemonicParsing="false">
         <graphic>
            <ImageView fitHeight="30.0" fitWidth="21.0" pickOnBounds="true" preserveRatio="true">
               <image>
                  <Image url="@bin.png" />
               </image>
            </ImageView>
         </graphic>
      </Button>
      <Button fx:id="pasteBtn" layoutX="370.0" layoutY="491.0" mnemonicParsing="false">
         <graphic>
            <ImageView fitHeight="30.0" fitWidth="21.0" pickOnBounds="true" preserveRatio="true">
               <image>
                  <Image url="@paste.png" />
               </image>
            </ImageView>
         </graphic>
      </Button>
      <Button fx:id="copyBtn" layoutX="871.0" layoutY="544.0" mnemonicParsing="false">
         <graphic>
            <ImageView fitHeight="30.0" fitWidth="21.0" pickOnBounds="true" preserveRatio="true">
               <image>
                  <Image url="@copy.png" />
               </image>
            </ImageView>
         </graphic>
      </Button>
      <Line endX="-53.5" layoutX="205.0" layoutY="52.0" startX="-100.0" fx:id="underline1" />
      <Line endX="-53.5" layoutX="394.0" layoutY="52.0" startX="-100.0" fx:id="underline2" />
   </children>
</AnchorPane>
```

Appendix 7. mainscene.fxml.

Appendix 8



<<interface>>
TextProcessor
+preprocess(String text): List<List<String>>
implements
DefaultTextProcessor
-stemmer: Stemmer
-stopwords: Set<String>
-splitter: SentenceSplitter
+DefaultTextProcessor(Stemmer stemmer)
+preprocess(String text): List<List<String>>
+loadStopWords(): Set<String>
<<interface>>
Stemmer
+stem (String word): String
implements
SnowballStemmer
-stemmer: PorterStemmer
+stem(String word): String
Controller
-window: Stage
-inputArea, outputArea: TextArea
-summarizeBtn, paragraphBtn, bulletBtn, copyBtn, pasteBtn, clearBtn, StatsBtn: Button
-inputCount, outputCount: Label
-lengthSlider: Slider
-underline1, underline2: Line
-summaryRatio: double
-paragraphFormat: boolean
-splitter: SentenceSplitter
+setWindow(Stage window): void
+initialize(): void
-countWord(String text): int
<<abstract>>
AbstractSummarizer
-doubleMapSupplier: Supplier<Map<String,Double>>
-intMapSupplier: Supplier<Map<String,Integer>>
+AbstractSummarizer(Supplier<Map<String,Double>> doubleMapSupplier, Supplier<Map<String,Integer>> intMapSupplier)
+summarize (String text, double ratio): String
extends
TFIDFSummarizer
-textProcessor: TextProcessor
-splitter: SentenceSplitter
+TFIDFSummarizer(Supplier<Map<String,Double>> doubleMapSupplier, Supplier<Map<String,Integer>> intMapSupplier, TextProcessor textProcessor)
+summarize (String text, double ratio): String
SentenceSplitter
-HONORIFICS: Set<String>
+splitIntoSentences(String text): String[]
StatsController
-window: Stage
-sentenceStats, wordStats, reductionStats: Label
+setWindow(Stage window): void
+setStats(int originalSentence, int summarizedSentence, int originalWord, int summarizedWord): void

Appendix 8. UML class diagram.

Appendix 9 - Program Manual

1. Download Java SDK 24 (if not yet downloaded).
2. Change the path of Java SDK in launch.json and settings.json.
3. Open Main.java and run main().

Appendix 10

The GitHub repository for this program can be found here.

Appendix 11

The presentation for this project can be found here.