P.S.R ENGINEERING COLLEGE

(An Autonomous Institution, Affiliated to Anna University,Chennai)

SIVAKASI-626 140

A MINI PROJECT REPORT

on

TEXT SIMILARITY CHECKERS

Submitted by

M PITCHUMANI (Reg No.95192202073)

In partial fulfillment for the award of the degree Of

BACHELOR OF ENGINEERING

ELECTRONICS AND COMMUNICATION ENGINEERING

P.S.R. ENGINEERING COLLEGE, SIVAKASI

MARCH 2025

BONAFIDE CERTIFICATE

This is to certified that this project report titled **"TEXT SIMILARITY CHECKERS"** in the bonafide work of **PITCHUMANI M (95192202073)** who carried out the project work under my supervision.

SIGNATURE OF THE TRAINER
Mr.V.NAVEEN KUMAR
EVORIEA INFOTECH PRIVATE LIMITED,
BANGALORE.

SIGNATURE OF THE HOD
Dr.K.VALARMATHI, M.Tech.,Ph.D.
PROFESSOR & HOD,
DEPARTMENT OF ELECTRONICS AND
COMMUNICATION
ENGINEERING,
P.S.R. ENGINEERING COLLEGE ,
SIVAKASI – 626 140,
VIRUDHUNAGAR,
TAMILNADU ,INDIA.

Submitted for the project viva-voce held on…………..

ACKNOWLEDGEMENT

I take this opportunity to put record my sincere thanks to all who enlightened my path towards the successful completion of this project. At very outset, I thank the **Almighty** for this abundant blessings showered on me.

It is my greatest pleasure to convey my thanks to **Thiru.R.Solaisamy, Correspondent & Managing Trustee, P.S.R Engineering College,** for having provided me with all required facilities and suitable infrastructure to complete my project without thrones.

It is my greatest privilege to convey my thanks to beloved **Dr.J.S.Senthilkumar, M.E., Ph.D, Principal, P.S.R. Engineering College,** for having provided me with all required facilities to complete my project without hurdles.

I pour our profound gratitude to my beloved Head of the Department, **Dr.K.Valarmathi,M.Tech., Ph.D.,** for providing ample facilities made available to undergo my project successfully.

I thank my project trainer **Mr.V.Naveenkumar, B.E, (EVORIEA)** for his excellent supervision patiently throughout my project work and endless support helped me to complete my project work on time.

I also bound to thank our placement coordinators, Teaching and Non-Teaching faculty members for support to complete my project work.

ABSTRACT

Text similarity measurement is a fundamental problem in natural language processing (NLP) and information retrieval. This project implements a Java-based text similarity checker that employs multiple methodologies to compare and quantify the similarity between two input text strings. The implemented approaches include Jaccard Similarity, Cosine Similarity, Dice Coefficient, Levenshtein Distance, and Longest Common Subsequence (LCS), each analyzing different aspects of textual similarity, from token-based and vector-based matching to edit distance calculations. Jaccard Similarity measures similarity based on the ratio of the intersection to the union of unique words in both texts, making it effective for determining word overlap but insensitive to word frequency. In contrast, Cosine Similarity calculates the cosine of the angle between word frequency vectors, considering word occurrence and importance, making it more sensitive to variations in wording. The Dice Coefficient, another token-based measure, evaluates similarity by computing twice the intersection size divided by the sum of both sets' sizes, providing an effective metric for partial overlaps without accounting for word order. Levenshtein Distance, or edit distance, determines the minimum number of insertions, deletions, or substitutions required to transform one text into another, making it suitable for spell-checking and fuzzy string matching. Additionally, the LCS algorithm identifies the longest sequence of characters appearing in both texts while maintaining their relative order, making it useful for structural similarity evaluation. The Java implementation efficiently processes text inputs, tokenizes words, and computes similarity scores using data structures like HashSets for Jaccard and Dice calculations, HashMaps for Cosine Similarity, and dynamic programming matrices for Levenshtein Distance and LCS computations. By integrating these techniques, the program provides a comprehensive analysis of text similarity, applicable to plagiarism detection, document comparison, sentiment analysis, and text clustering. Future improvements could include semantic similarity approaches using word embeddings or deep learning models to capture contextual relationships between words.

TABLE OF CONTENT

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION TO JAVA

Java is one of the most widely used programming languages in the world, known for its simplicity, platform independence, robustness, and versatility. Developed by **James Gosling** at **Sun Microsystems** in 1995, Java was designed with the philosophy of "**Write Once, Run Anywhere (WORA)**," meaning that Java programs can run on any platform that supports the **Java Virtual Machine (JVM)** without needing modification. This cross-platform capability, enabled by Java's bytecode execution model, has made it a preferred language for developers working on web applications, mobile applications, enterprise software, and even embedded systems.

Java follows an **object-oriented programming (OOP)** paradigm, which means that it organizes code into reusable structures known as **classes and objects**. This enhances modularity, code reusability, and scalability, making Java a powerful language for software development. The four pillars of OOP—**encapsulation, inheritance, polymorphism, and abstraction**—help developers write cleaner, more efficient, and maintainable code. Additionally, Java supports multi-threading, allowing developers to write concurrent applications that can handle multiple tasks simultaneously, improving performance and responsiveness.

One of Java's greatest strengths is its **rich standard library (Java API)**, which provides pre-built classes and methods for various functionalities, including file handling, networking, data structures, multithreading, database connectivity, and graphical user interfaces (GUIs). Java also has a vast ecosystem of frameworks and tools such as **Spring, Hibernate, Apache Struts, and JavaFX**, which simplify and enhance application development. The language's built-in security features, such as **automatic memory management (garbage collection), exception handling, and strong type-checking**, ensure reliable and secure applications, reducing the risk of common programming errors such as memory leaks and pointer-related issues.

Java is widely used in multiple domains, making it one of the most versatile languages. In **web development**, Java powers back-end systems using frameworks like **Spring Boot** and **Jakarta EE**, enabling the development of scalable and secure web applications. In **mobile**

development, Java is the primary language for **Android application development**, providing developers with a rich set of tools, such as **Android Studio** and **the Android SDK**. In **enterprise applications**, Java is heavily used for building large-scale software systems, financial applications, and business solutions due to its robustness, scalability, and security. Additionally, Java is used in **cloud computing, artificial intelligence (AI), game development, and the Internet of Things (IoT)**.

Another key feature that contributes to Java's popularity is its **automatic memory management system**, which eliminates the need for manual memory allocation and deallocation. Java's **Garbage Collector (GC)** automatically removes unused objects, ensuring efficient memory utilization and preventing memory leaks. Moreover, Java's **exception handling mechanism** provides a structured way to handle runtime errors, making programs more reliable and fault-tolerant.

Java's large and active developer community, extensive documentation, and open-source nature contribute to its continued growth and evolution. With the release of new versions, Java continues to introduce powerful features such as **records, sealed classes, enhanced switch expressions, and improved performance optimizations**, keeping it relevant in modern software development. Its long-standing presence in the industry, strong support from tech companies, and adaptability to emerging technologies make Java an essential language for developers worldwide. Whether for beginners or experienced programmers, Java remains a **powerful, flexible, and future-proof programming language** that continues to drive innovation across industries.

1.2 INTRODUCTION TO TEXT SIMILARITY CHECKERS

Text similarity is a crucial concept in natural language processing (NLP), information retrieval, and text mining. The ability to compare and measure the similarity between different text samples plays a vital role in applications such as plagiarism detection, spell-checking, sentiment analysis, document clustering, and recommendation systems. This project presents a **Java-based text similarity checker** that utilizes multiple algorithmic approaches to assess the similarity between two text inputs. The implemented methods include **Jaccard Similarity, Cosine Similarity, Dice Coefficient, Levenshtein Distance, and Longest Common**

Subsequence (LCS). Each of these techniques offers a unique way of comparing textual data, allowing for a comprehensive and detailed similarity analysis.

The **Jaccard Similarity** method is a set-based approach that computes the ratio of the intersection to the union of words in two texts. This technique is particularly useful for determining how much overlap exists between two sets of words, making it effective for document comparison and plagiarism detection. However, it does not consider word frequency, which limits its accuracy in cases where repeated words matter. In contrast, **Cosine Similarity** is a vector-based method that calculates the cosine of the angle between two word frequency vectors. By taking word frequency into account, this approach is particularly useful for scenarios where the relative importance of words is crucial, such as in search engines and keyword-based ranking systems.

Another important metric used in this project is the **Dice Coefficient**, which measures the similarity between two sets of words by comparing their overlap. This technique is widely used in fuzzy matching and bioinformatics, as it effectively quantifies the similarity between two sequences. The **Levenshtein Distance**, also known as edit distance, is a character-based metric that determines the minimum number of operations—insertions, deletions, or substitutions—needed to transform one text into another. This method is particularly useful in applications such as autocorrect systems, DNA sequence matching, and error correction algorithms.

In addition to these methods, the project implements the **Longest Common Subsequence (LCS)** algorithm, which identifies the longest ordered sequence of characters that appear in both texts. The LCS method is highly effective for detecting structural similarities in sentences and documents, making it valuable in text comparison tasks where word order is important. The LCS percentage is also calculated, providing a normalized similarity score based on the length of the longer text. This metric ensures that comparisons are fair even when the input texts have significant length differences.

To implement these similarity measures efficiently, the program utilizes various data structures such as **HashSets** for Jaccard Similarity and Dice Coefficient, **HashMaps** for Cosine Similarity, and **dynamic programming matrices** for Levenshtein Distance and LCS computations. This ensures optimal performance and scalability, making the system suitable for real-time text analysis. The Java-based implementation allows users to enter two text inputs, and the program computes and displays similarity scores using all five methods, providing a multi-perspective analysis.

This project serves as a valuable tool for students, researchers, and professionals who need to analyze text similarity accurately. By integrating multiple similarity measures, it offers a well-rounded approach that accommodates different use cases, from simple keyword matching to complex sentence structure analysis. The potential for future enhancements includes incorporating **semantic similarity techniques** using word embeddings or deep learning models to capture contextual relationships between words. Overall, this Java-based text similarity checker is a **powerful and efficient solution for comparing textual data**, helping users understand the relationships between different texts and facilitating further advancements in NLP applications.

CHAPTER 2

ANALYSIS

2.1 EXISTING SYSTEM

In the field of text similarity analysis, various existing systems and approaches are used to compare and evaluate the similarity between two text documents. These systems play a crucial role in applications such as plagiarism detection, search engines, document classification, natural language processing (NLP), and automated text summarization. Traditional text similarity detection techniques rely on keyword matching, statistical methods, and mathematical models to measure how closely two texts resemble each other. However, these existing systems have their own advantages and limitations based on the methods they employ.

One of the most common approaches used in existing text similarity systems is **string-based similarity measurement**, which includes techniques such as **Jaccard Similarity, Cosine Similarity, and Dice Coefficient**. These methods rely on word-based or character-based tokenization, where the text is broken into individual words or characters, and similarity is computed based on shared terms. For example, **Jaccard Similarity** calculates the ratio of common words to the total number of unique words in both texts, while **Cosine Similarity** treats text as a vector of word frequencies and determines the angle between two vectors. These methods are effective in basic similarity detection but fail to capture semantic meaning and word variations.

Another common approach in existing systems is **edit distance-based techniques**, such as the **Levenshtein Distance** algorithm. This method calculates the minimum number of insertions, deletions, or substitutions needed to convert one text into another. While edit distance is useful in applications like **spell-checking, autocorrection, and fuzzy matching**, it does not consider sentence structure or word importance, making it less effective for complex text comparison tasks.

Some advanced systems implement **Longest Common Subsequence (LCS)** to determine the longest sequence of characters or words that appear in both texts while maintaining their relative order. This technique is useful in scenarios where word order matters, such as **plagiarism detection and text alignment**. However, traditional LCS-based systems do not account for synonyms or variations in sentence phrasing, limiting their effectiveness in semantic similarity analysis.

Despite the effectiveness of these traditional methods, most existing systems struggle with **semantic understanding**—the ability to recognize meaning beyond exact word matches. To address this limitation, modern approaches use **semantic similarity techniques** based on **natural language processing (NLP) and machine learning (ML)**. For example, **word embeddings** such as **Word2Vec, GloVe, and BERT (Bidirectional Encoder Representations from Transformers)** allow systems to understand contextual meaning by mapping words into numerical vectors that capture semantic relationships. These advanced methods provide better accuracy but require more computational power and large datasets for training.

Furthermore, existing text similarity systems often suffer from **performance issues when handling large documents**. Some systems rely on simple word matching, which leads to inefficiencies in processing long texts. Others require extensive pre-processing steps, such as stemming, stop-word removal, and tokenization, which increase the computational complexity. Additionally, many existing systems lack **real-time processing capabilities**, making them unsuitable for time-sensitive applications such as **live search engines, chatbots, and recommendation systems**.

Overall, while existing systems provide various approaches to text similarity detection, they often face challenges in **scalability, efficiency, and semantic understanding**. Modern systems continue to evolve by integrating **deep learning, artificial intelligence (AI), and cloud computing** to improve accuracy, speed, and contextual understanding. However, traditional text similarity methods remain relevant due to their simplicity, ease of implementation, and lower computational requirements.

2.2 PROPOSED SYSTEM

The proposed system aims to enhance text similarity analysis by integrating multiple algorithms that improve accuracy, efficiency, and contextual understanding. Unlike traditional systems that rely on simple word-matching techniques, this system incorporates **Jaccard Similarity, Cosine Similarity, Dice Coefficient, Levenshtein Distance, and Longest Common Subsequence (LCS)** to provide a comprehensive comparison of two text documents.

By combining different mathematical models and computational approaches, the system ensures a more reliable and precise evaluation of textual similarity.

One of the major improvements of the proposed system is the **hybrid approach**, where both word-based and character-based similarity measures are used. **Jaccard Similarity and Dice Coefficient** help in determining common words between the texts, making them effective for **document clustering and keyword-based analysis**. **Cosine Similarity** evaluates the frequency distribution of words, making it suitable for applications such as **search engines and ranking algorithms**. **Levenshtein Distance** measures the number of character-level edits required to transform one text into another, making it ideal for **error correction and fuzzy matching**. Additionally, **Longest Common Subsequence (LCS)** is implemented to detect **structural similarities between sentences**, which is beneficial in **plagiarism detection and paraphrase identification**.

To enhance performance, the proposed system utilizes **efficient data structures** such as **HashSets, HashMaps, and dynamic programming matrices** to store and process text efficiently. The **tokenization process** ensures that text is broken down into meaningful components before similarity computations begin. This reduces processing time and allows for real-time text comparison, making the system suitable for **large-scale applications** like **real-time document comparison, content filtering, and AI-driven recommendation systems**.

Another key feature of the proposed system is its ability to handle **semantic similarity** to some extent. Unlike traditional systems that rely solely on exact word matching, this system includes **longest common subsequence percentage**, which evaluates the structural similarity of sentences even if they are not exact word matches. This feature enhances the system's ability to recognize **paraphrased content and reworded sentences**, which is crucial in plagiarism detection and AI-generated text analysis. In future versions, the system can be further enhanced by integrating **word embeddings** such as **Word2Vec, GloVe, or BERT**, which capture **contextual meaning and synonyms**.

The **usability and flexibility** of the system make it suitable for different domains, including **education, research, business, and cybersecurity**. In education, the system can help detect **plagiarism in student assignments**, while researchers can use it for **document similarity analysis**. Businesses can utilize the system for **sentiment analysis, chatbot responses, and text classification**, while cybersecurity applications can benefit from **fraud detection and**

spam filtering.Compared to existing systems, the proposed approach offers **greater accuracy, scalability, and real-time processing capabilities**. By integrating multiple similarity techniques, it minimizes the weaknesses of individual methods and provides a **more robust and reliable** text similarity analysis tool. Additionally, the modular design of the system ensures that it can be **easily extended** with advanced **machine learning and artificial intelligence (AI) models** in the future.

Overall, the proposed system provides a **powerful and efficient** solution for text similarity analysis by combining **mathematical models, optimized data structures, and real-time processing capabilities**. It overcomes the limitations of existing systems and serves as a **versatile tool for text comparison across multiple industries**.

2.3 OBJECTIVES

The primary objective of this project is to develop an efficient and accurate **text similarity analysis system** that utilizes multiple algorithms to compare two textual documents. Traditional text similarity techniques often rely on simple word-matching methods, which may not capture deeper relationships between words and phrases. This system aims to overcome such limitations by integrating **Jaccard Similarity, Cosine Similarity, Dice Coefficient, Levenshtein Distance, and Longest Common Subsequence (LCS)** to provide a more comprehensive and precise comparison of textual data.

One of the key goals of this system is to **enhance accuracy in text comparison** by using a hybrid approach. By combining multiple similarity measures, the system ensures that results are not solely dependent on a single method, which might be limited in capturing different types of similarities. **Jaccard Similarity and Dice Coefficient** focus on set-based word comparisons, **Cosine Similarity** analyzes the frequency distribution of words, **Levenshtein Distance** measures character-level transformations, and **LCS** evaluates structural similarities in sentences. The integration of these algorithms provides a **multi-dimensional approach to text similarity detection**, making the system highly reliable for various applications.

Another crucial objective is to **optimize performance and scalability**. The system utilizes **efficient data structures** such as **HashSets, HashMaps, and dynamic programming matrices** to store and process text efficiently. This ensures fast computation and minimal memory usage, making the system suitable for **large-scale text analysis tasks**, such as processing thousands of documents in **real-time applications** like search engines, plagiarism detection tools, and AI-driven content recommendations.

Additionally, the project aims to **improve semantic understanding** in text similarity analysis. Traditional methods primarily focus on exact word matching and fail to recognize paraphrased content or variations in sentence structures. By incorporating **Longest Common Subsequence Percentage (LCS %)**, the system can measure structural similarity beyond direct word matches, improving its ability to detect reworded sentences and paraphrased content. This feature is especially useful in **academic integrity checks, content moderation, and automated text evaluation systems**. Future enhancements can integrate **machine learning-based techniques** such as **Word2Vec, GloVe, and BERT** to capture the contextual meaning of words, further improving semantic text analysis.

Another objective of the proposed system is to **provide a user-friendly and flexible solution** that can be adapted across various domains. In **education**, the system can be used for **plagiarism detection in academic papers**. In **business and marketing**, it can help analyze **customer feedback and sentiment analysis**. In **cybersecurity**, it can aid in **detecting spam and fraud by identifying similar malicious texts**. Its flexibility ensures that it can be customized to meet different needs in industries that rely on textual data processing.

Finally, the system is designed to be **scalable and extendable**, allowing future enhancements to integrate **advanced artificial intelligence (AI) and deep learning models** for even greater accuracy and efficiency. By ensuring modularity in its implementation, developers can easily add new features, making the system adaptable to the rapidly evolving field of text analysis.

CHAPTER 3

LITERATURE REVIEW

Dashti et al. (2024) proposed an automatic real-word error correction system for Persian text using neural computing applications. Their method focuses on detecting and correcting real-word errors that traditional spell-checkers often miss. By leveraging machine learning techniques, their approach enhances text quality and readability, particularly in Persian language processing.

Shipurkar et al. (2023) developed an end-to-end system for handwritten text recognition and plagiarism detection using Convolutional Neural Networks (CNNs) and Bidirectional Long Short-Term Memory (BLSTM) networks. Their study demonstrates the effectiveness of deep learning models in processing handwritten documents and detecting textual similarities, which is crucial for academic and research-based plagiarism detection.

Panchal and Shah (2024) implemented Norvig's algorithm for spell-checking in Gujarati text. Their research addresses challenges in low-resource language processing, ensuring efficient text correction and enhanced accuracy. This approach improves text preprocessing for Gujarati-based applications such as document processing and machine translation.

Patil et al. (2010) explored different word suggestion techniques for handling non-word text errors using similarity measures. Their study compares edit distance-based and cosine similarity-based methods, highlighting the advantages of each technique in correcting misspelled words and improving NLP applications.

Dashti and Dashti (2024) focused on improving Persian clinical text quality through an advanced spelling correction system. Their study highlights the importance of error-free medical documentation in electronic health records (EHRs), ensuring better data management and communication in the healthcare sector.

Ramdani et al. (2022) conducted an empirical study on the use of information retrieval techniques in text similarity checker tools. Their work emphasizes the role of semantic similarity measures in plagiarism detection, particularly in academic integrity and research ethics enforcement.

Ravindhar et al. (2023) proposed a plagiarism detection system aimed at preventing copyright infringement and piracy in NCERT textbooks. Their model integrates artificial intelligence (AI) techniques to analyze text similarities, ensuring protection of intellectual property in the educational sector.

Patil et al. (2022) performed a comparative study on minimum edit distance and cosine similarity for correcting Marathi spelling errors. Their research demonstrates that a combination of these methods provides a more effective spell-checking solution for the Marathi language, improving accuracy in text correction applications.

Zarrabi et al. (2023) introduced Hamtajoo, a Persian plagiarism detection system designed specifically for academic manuscripts. This system leverages natural language processing (NLP) techniques to analyze and detect plagiarism in Persian-language academic content, enhancing the authenticity of scholarly work.

Barve and Saini (2023) proposed a Text Resemblance Index (TRI) method for fact-checking applications. Their model evaluates the similarity between textual sources to verify factual accuracy and combat online misinformation, making it an essential tool for journalism and social media monitoring.

Orellana et al. (2024) focused on pre-processing techniques for analyzing ECU 911 emergency call transcripts. Their study demonstrates how text analysis and information retrieval techniques can improve emergency response systems, ensuring better communication and faster decision-making in critical situations.

Sánchez-Vega et al. (2024) explored data augmentation and adversarial attack methods to improve low-resource text classification models. Their research highlights the importance of security and robustness in natural language processing applications, particularly in preventing biases and adversarial attacks.

Jamwal and Gupta (2022) developed a hybrid spell-checking system for the Dogri language, combining rule-based and machine learning approaches. Their work contributes to preserving and improving computational linguistics for regional languages, ensuring accurate text processing in low-resource NLP.

Balaha and Saafan (2021) introduced an Automatic Exam Correction Framework (AECF) for grading multiple-choice questions (MCQs), essays, and equations. Their model automates exam evaluation using AI-driven techniques, enhancing efficiency in education and assessment systems.

Nagaraj et al. (2023) proposed an automatic text correction system using probabilistic error modeling. Their work enhances search engine query correction, chatbot responses, and text-generation models, improving the overall accuracy and usability of text-based AI applications.

CHAPTER 4
MODULES

4.1 Processing module

The **Preprocessing Module** is a crucial component in any text similarity checker, as it transforms raw text into a clean, standardized format suitable for further analysis. This module plays a fundamental role in ensuring that the input text is prepared for effective comparison, feature extraction, and similarity measurement. Raw text is often noisy, containing irrelevant information, inconsistencies, and extraneous elements that can negatively impact the accuracy of similarity calculations. The preprocessing module addresses these challenges by cleaning and structuring the text in a way that allows the similarity checker to focus on meaningful content. One of the first steps in preprocessing is **tokenization**, which involves breaking down the text into smaller units, typically words or sentences. This step is necessary because the text must be divided into manageable pieces to be analyzed independently. For instance, a sentence like "The quick brown fox" would be tokenized into the words "The", "quick", "brown", and "fox", allowing each word to be compared individually or collectively. After tokenization, **lowercasing** the entire text is essential. Since words like "Apple" and "apple" are technically different due to case sensitivity, converting all text to lowercase ensures that such variations do not lead to incorrect similarity results. This transformation makes sure that words are treated uniformly, regardless of how they appear in the text. Another important step is **stop-word removal**. Stop-words are common words such as "the", "and", "is", or "in", which appear frequently in text but add little value when comparing the semantic meaning of sentences or phrases. Removing these words helps to eliminate noise from the text, allowing the similarity checker to focus on the more meaningful words that convey the core ideas of the text. For example, in the sentence "The dog jumped over the fence," removing "the" and "over" would leave us with the key content words "dog", "jumped", and "fence", which are more relevant for similarity analysis. **Punctuation and special character removal** is another crucial preprocessing task. Punctuation marks, such as commas, periods, or exclamation points, and special characters like hashtags or symbols, do not contribute to the meaning of the text in most similarity comparisons. Thus, they are typically removed or ignored during preprocessing. For example, the sentence "Hello! How are you?" would be reduced to "Hello How are you" to focus solely on the words that carry meaning. **Lemmatization and stemming** are techniques used to reduce words to their root forms. Lemmatization involves transforming words into their base or dictionary form by considering their meaning and context (e.g., "better" becomes "good"). Stemming, on the other hand, simply removes prefixes or suffixes, often producing non-existent words (e.g., "running" becomes "run"). Lemmatization is generally preferred because it preserves the semantic integrity of words, helping to ensure that variations of a word do not lead to inaccurate similarity scores. Additionally, the **handling of negations** is an important preprocessing consideration. Negations, such as "not" or "never", can drastically change the meaning of a sentence. Therefore, the preprocessing module must recognize and account for these negations to maintain the accuracy of the similarity comparison. Finally, **whitespace and formatting removal** is necessary to eliminate unnecessary spaces, tabs, or newline characters that might appear in poorly formatted text. By ensuring that the text is consistently formatted and free of excess whitespace, the preprocessing module helps maintain the integrity of the similarity analysis. In some cases, preprocessing also involves **language detection and normalization**, especially when the input text may contain mixed languages or dialects. This step ensures that the text is processed in the correct linguistic context, improving the accuracy of the analysis. In summary, the Preprocessing Module is indispensable for preparing raw text for meaningful comparison. By cleaning, standardizing, and structuring the text, it helps ensure that the subsequent steps in the similarity detection process focus on relevant and meaningful content, ultimately leading to more accurate and reliable results.

4.2 Feature Extraction Module

The **Feature Extraction Module** is a pivotal component of a text similarity checker, responsible for transforming preprocessed text into numerical representations that can be used for comparing and measuring similarity. After the preprocessing phase, where noise and irrelevant data are removed, the text needs to be converted into a format that a computational model can understand and process. The Feature Extraction Module focuses on capturing the essential features or characteristics of the text that reflect its semantic meaning. These features are then used for comparison with other texts, allowing the system to determine how similar or different two pieces of text are. The most common approach for feature extraction involves representing the text as vectors in a high-dimensional space, where each dimension corresponds to a specific feature or characteristic of the text. One of the simplest and most traditional methods of feature extraction is **Bag-of-Words (BoW)**. In this method, the text is represented as a collection of words, and each word’s frequency of occurrence in the document is counted. This results in a vector where each entry corresponds to the frequency of a specific word in the text. While effective in some cases, BoW has limitations. It does not take into account the order of words or the relationships between them, and it can also result in high-dimensional vectors that are sparse, meaning they contain many zeros. A more sophisticated approach is **TF-IDF (Term Frequency-Inverse Document Frequency)**, which builds on the BoW model by weighing each word's importance based on its frequency in the document (Term Frequency) and its rarity across all documents in the dataset (Inverse Document Frequency). This helps to highlight words that are more unique to a specific document, making the feature vectors more informative. However, both BoW and TF-IDF are limited in their ability to capture the deeper semantic meaning of words and sentences. **Word Embeddings**, such as **Word2Vec** and **GloVe**, provide a more advanced method for feature extraction. These models represent words as dense vectors in a continuous vector space, where semantically similar words are placed closer together. For example, "king" and "queen" would be represented as vectors that are closer to each other than "king" and "dog". Word embeddings capture the relationships between words based on the context in which they appear, making them much more powerful than simple frequency-based models like BoW or TF-IDF. However, word embeddings still focus on individual words, which can be limiting when trying to compare entire sentences or paragraphs. To address this limitation, **sentence embeddings** have been developed. Models like **BERT (Bidirectional Encoder Representations from Transformers)** and **SBERT (Sentence-BERT)** generate dense vector representations not just for individual words, but for entire sentences or passages of text. These embeddings capture the contextual meaning of a sentence, taking into account the relationships between all the words in the sentence. For example, the sentences "The cat chased the mouse" and "The mouse was chased by the cat" would have very similar embeddings, even though the word order is different. This approach enables a more nuanced and accurate comparison of text, particularly when comparing longer pieces of content. Additionally, **n-grams** are another technique used in feature extraction. An n-gram is a sequence of 'n' words taken from the text. For instance, in the sentence "The quick brown fox," 2-grams (bigrams) would include "The quick," "quick brown," and "brown fox." N-grams are useful for capturing context and word combinations, which can be important for similarity detection, as meaning is often derived not just from individual words but from the way they are combined. The Feature Extraction Module may also incorporate techniques to handle specific text attributes, such as part-of-speech (POS) tagging, named entity recognition (NER), or syntactic parsing. These techniques can help the model focus on certain parts of speech (e.g., nouns and verbs) or identify named entities (e.g., people, places, organizations), which can be valuable for measuring similarity in certain contexts, such as comparing news articles or technical documents. Once the text has been converted into numerical features, the system can use these features to measure the similarity between different pieces of text. Common similarity measures include cosine similarity, Euclidean distance, and Jaccard similarity, which all rely on the vector representations generated by the feature extraction process. The more accurately and effectively the text has been transformed into meaningful features, the more reliable the similarity scores will be. In summary, the Feature Extraction Module plays a critical role in the text similarity checking process by converting raw, preprocessed text into structured numerical data that can be compared. The module uses various methods, ranging from simple word counts to advanced deep learning-based embeddings, to capture the essential meaning of the text. This process ensures that the system can evaluate text similarity in a meaningful and accurate way, providing valuable insights for tasks such as plagiarism detection, content comparison, and recommend Sim ation systems.

4.3 Similarity Measurement Module

The **Similarity Measurement Module** is a crucial component in a text similarity checker, as it quantifies how similar or different two pieces of text are based on the features extracted in the previous step. Once the text has been preprocessed and converted into numerical representations or feature vectors, the next logical step is to assess the degree of similarity between these representations. The Similarity Measurement Module calculates this similarity using mathematical models and algorithms, allowing the system to generate meaningful scores that indicate how closely related two texts are. Several methods are employed in the Similarity Measurement Module, each with its advantages and specific use cases. One of the most common approaches is **Cosine Similarity**, which measures the cosine of the angle between two vectors in a multi-dimensional space. Cosine similarity ranges from 0 to 1, where a cosine similarity of 1 indicates that the vectors are identical, while a cosine similarity of 0 means the vectors are completely dissimilar. This method is especially useful when working with high-dimensional text data, as it focuses on the orientation of the vectors rather than their magnitude, making it resilient to variations in document length. For example, in a text similarity task, even if two documents are of different lengths, cosine similarity can still effectively assess their content similarity. **Euclidean Distance** is another popular method used in the Similarity Measurement Module. This technique measures the straight-line distance between two vectors in a vector space. The closer the two vectors are, the smaller the Euclidean distance, indicating higher similarity. Euclidean distance is intuitive and easy to compute, but it can be sensitive to the magnitude of the vectors, which can cause issues in cases where text lengths vary significantly. **Jaccard Similarity**, on the other hand, compares two sets by measuring the ratio of their intersection to their union. It is often used in scenarios where the text is represented as a set of words or tokens, as it evaluates the proportion of shared elements between the two sets. For instance, if two texts share three words out of a total of ten unique words, their Jaccard similarity score would be 0.3. Jaccard similarity is simple and effective, but it may not capture the subtle nuances of meaning in more complex texts. **Manhattan Distance**, also known as the **L1 distance**, is another method used in text similarity analysis. Unlike Euclidean distance, which measures the shortest straight-line distance, Manhattan distance calculates the sum of the absolute differences between corresponding vector components. This method can be more robust in certain contexts, particularly when comparing vectors with large differences in values across multiple dimensions. **Pearson Correlation** is another statistical measure that is sometimes employed in text similarity tasks, particularly when the text representations are vectors of continuous values. Pearson correlation measures the linear relationship between two vectors, ranging from -1 (perfect negative correlation) to +1 (perfect positive correlation), with 0 indicating no linear relationship. Pearson correlation is often used in cases where the relationship between vector components needs to be taken into account, especially when dealing with word embeddings or sentence embeddings where the dimensions may represent continuous values rather than discrete counts. The Similarity Measurement Module also leverages more advanced techniques, particularly when working with deep learning-based models, such as **BERT** or **SBERT**. These models generate dense vector embeddings for entire sentences or documents, which capture not only the individual words' meanings but also their contextual relationships within the text. In these cases, the Similarity Measurement Module may use specialized metrics like **Semantic Textual Similarity (STS)**, which combines traditional similarity measures with the semantic understanding derived from the embeddings. This allows for more nuanced similarity comparisons that take into account the overall meaning of the text, even when the exact words or phrasing differ. Additionally, in some cases, **Clustering** techniques may be applied after similarity measurements to group similar texts together, especially when dealing with large datasets. Clustering methods like **K-means** or **Hierarchical Clustering** can segment texts into categories based on their similarity scores, allowing for more efficient analysis or categorization. These clustering results can be particularly useful for tasks such as document categorization, topic modeling, or duplicate content detection. The output of the Similarity Measurement Module typically includes a similarity score that quantifies the degree of similarity between two texts. This score can then be interpreted to determine if the texts are sufficiently similar for the intended task, whether it's plagiarism detection, content recommendation, or information retrieval. The higher the score, the more similar the texts are. In some systems, a threshold may be applied, and texts with similarity scores above a certain level are considered duplicates or highly related. In conclusion, the Similarity Measurement Module is an integral part of any text similarity checker, as it provides the tools necessary to assess and quantify how closely related two pieces of text are. Through various methods like Cosine Similarity, Euclidean Distance, Jaccard Similarity, and more advanced techniques like BERT-based models, this module enables the system to produce accurate and meaningful similarity scores. These scores are the foundation for a wide range of applications, including plagiarism detection, document clustering, and semantic search, helping to facilitate effective text comparison and analysis.

4.4 TEXT FREQUENCY ANALYSIS MODULE

The **Text Frequency Analysis Module** is an essential component of text analysis, especially in the context of natural language processing (NLP) and text similarity computations. This module focuses on counting the frequency of individual words or tokens in a given text. Understanding word frequency plays a crucial role in a wide range of applications, including information retrieval, search engines, content analysis, and machine learning tasks like document classification and clustering. In essence, the Text Frequency Analysis Module transforms raw textual data into a structured form that can be further processed, analyzed, and compared. It forms the foundation for algorithms like **Cosine Similarity** and **TF-IDF (Term Frequency-Inverse Document Frequency)**, which rely on the frequency of words to assess similarity between texts.

In its most basic form, text frequency analysis involves counting how many times each word appears in the text. This process typically starts with text preprocessing, which may include steps like tokenization, converting the text to lowercase, removing stopwords, and handling punctuation. Tokenization is the first crucial step, where the text is split into individual words or phrases. These tokens are then processed to normalize the text for further analysis. For instance, common words (known as stopwords), such as "and," "the," or "is," are often removed because they do not provide significant meaning in similarity calculations. Additionally, punctuation marks, numbers, and special characters may also be discarded to focus purely on the words themselves.

Once the text is tokenized and cleaned, the next step in the Text Frequency Analysis Module is the actual counting of word occurrences. This can be achieved using data structures like hash maps or dictionaries, where the key is the word, and the value is its frequency of occurrence in the text. For example, consider the sentence: "Text analysis is crucial for text mining." After tokenization, the words would be "text," "analysis," "is," "crucial," "for," "text," and "mining." The word "text" would appear twice, while the other words would appear once. The resulting frequency map would look something like this: {"text": 2, "analysis": 1, "is": 1, "crucial": 1, "for": 1, "mining": 1}.

This word frequency map is important because it helps quantify the relative importance of words within the text. Higher frequency words are usually more significant in identifying key themes or topics, and this is why algorithms like **Cosine Similarity** use word frequencies to measure the similarity between two texts. In Cosine Similarity, word frequency vectors are created for two texts, and the cosine of the angle between these vectors is computed. The closer the vectors, the higher the similarity. Similarly, other techniques such as **TF-IDF** use word frequency to weigh words based on their occurrence across multiple documents, where words that occur frequently in a document but not across many documents are considered more important.

One important aspect of the Text Frequency Analysis Module is its scalability. As the volume of text grows, the efficiency of counting word frequencies becomes critical. In large datasets, advanced data structures like **trie** or **suffix trees** may be used to handle large vocabularies efficiently. Additionally, techniques like **stream processing** or distributed computing frameworks such as **Apache Spark** can be employed for real-time analysis of text data that may not fit into memory.

Furthermore, this module also allows for more sophisticated analysis, such as detecting **word patterns**, finding **collocations**, and examining **contextual usage** of words. Word frequency data can also serve as a basis for other text analytics tasks, such as **sentiment analysis**, **topic modeling**, and **document clustering**.

CHAPTER 5
DESIGN METHODOLOGY

The design methodology for a text similarity checker involves multiple stages, including data preprocessing, feature extraction, similarity measurement, and result evaluation. The process begins with data collection, where input text is gathered from various sources such as academic papers, online articles, or handwritten documents. The collected text undergoes preprocessing, which includes tokenization, stop word removal, stemming, and lemmatization. These steps standardize the text format, eliminate irrelevant words, and reduce words to their root forms, ensuring more accurate similarity detection.

Next, feature extraction plays a critical role in transforming the text into a numerical representation that can be analysed computationally. Techniques such as Term Frequency-Inverse Document Frequency (TF-IDF), word embeddings (Word2Vec, GloVe), and deep learning-based vectorization methods like BERT are commonly used. These representations help in capturing semantic and syntactic relationships between words, improving the precision of similarity detection.

The similarity measurement phase employs various algorithms to compute text similarity scores. Traditional methods include cosine similarity, Jaccard similarity, and Euclidean distance, which analyze word overlap and vector relationships. Advanced approaches incorporate deep learning models, such as Siamese networks and transformer-based architectures, which learn contextual representations of text for better similarity detection. Additionally, machine learning classifiers like Support Vector Machines (SVM) and Random Forest can be used to categorize text pairs as similar or dissimilar.

Once similarity scores are computed, the final stage involves result evaluation and performance analysis. Metrics such as precision, recall, F1-score, and accuracy help assess the effectiveness of the model. Benchmark datasets, including plagiarism detection corpora and academic text similarity datasets, are used for validation. The system can also integrate visualization tools to display similarity heatmaps, highlighting overlapping sections in text comparisons.

Overall, the design methodology ensures a structured approach to developing a robust text similarity checker. By incorporating efficient preprocessing, advanced feature extraction, and powerful similarity measurement techniques, the system can accurately detect duplicate or plagiarized content while maintaining high performance and scalability.

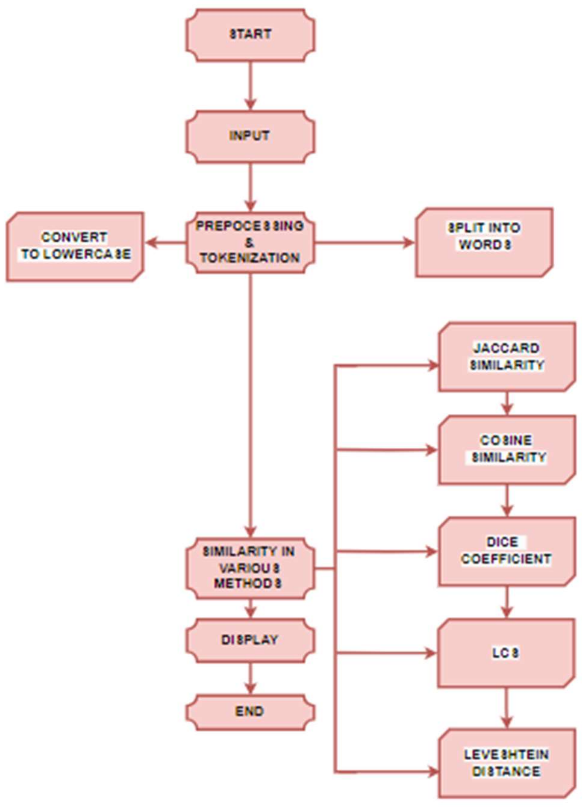


Fig 5.1 Flow chart

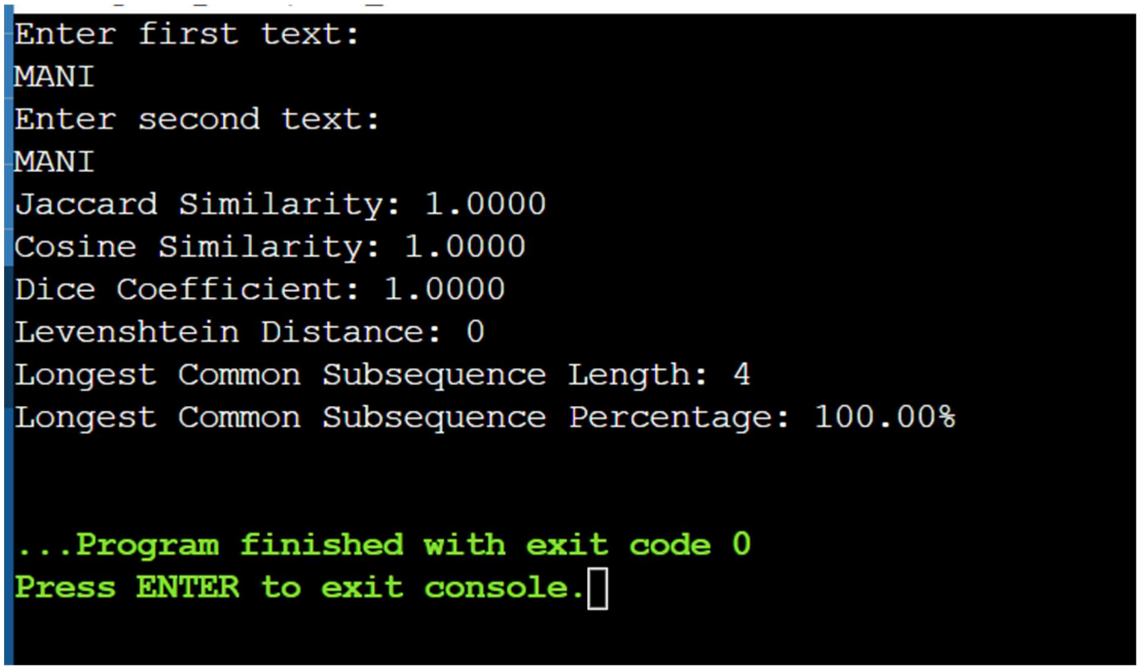
In fig 5.1 is represents the process of a Java-based text similarity checker, beginning with the **Start** node and moving to the **Input** stage, where users enter two text strings for comparison. The next phase involves **Preprocessing & Tokenization**, which consists of two key steps: **Convert to Lowercase** to ensure uniformity and **Split into Words** to break the text into individual tokens. This preprocessing stage prepares the text for similarity computations.

Once tokenized, the system evaluates text similarity using multiple methodologies, categorized under **Similarity in Various Methods**. The first method, **Jaccard Similarity**, calculates the ratio of the intersection to the union of unique words in both texts, providing insight into word overlap. **Cosine Similarity** uses vector-based calculations to measure the cosine angle between word frequency vectors, making it effective for analyzing text variations. **Dice Coefficient**, another token-based method, measures similarity based on twice the intersection size divided by the total number of words, making it effective for partial overlaps.

Additionally, the system incorporates sequence-based approaches such as **Longest Common Subsequence (LCS)**, which determines the longest ordered sequence of matching characters between two texts. The **Levenshtein Distance** calculates the minimum number of insertions, deletions, or substitutions required to convert one text into another, making it useful for spelling correction and fuzzy matching.

Once all similarity scores are computed, the results are displayed in the **Display** step, providing users with a comprehensive similarity analysis from multiple perspectives. The process then reaches the **End** stage. This structured approach ensures a robust comparison of textual data, making it suitable for applications such as plagiarism detection, document comparison, and text clustering. The integration of multiple similarity measures allows for a more accurate and flexible text comparison system.

CHAPTER 6
RESULT ANALYSIS



```
Enter first text:
MANI
Enter second text:
MANI
Jaccard Similarity: 1.0000
Cosine Similarity: 1.0000
Dice Coefficient: 1.0000
Levenshtein Distance: 0
Longest Common Subsequence Length: 4
Longest Common Subsequence Percentage: 100.00%


...Program finished with exit code 0
Press ENTER to exit console.
```

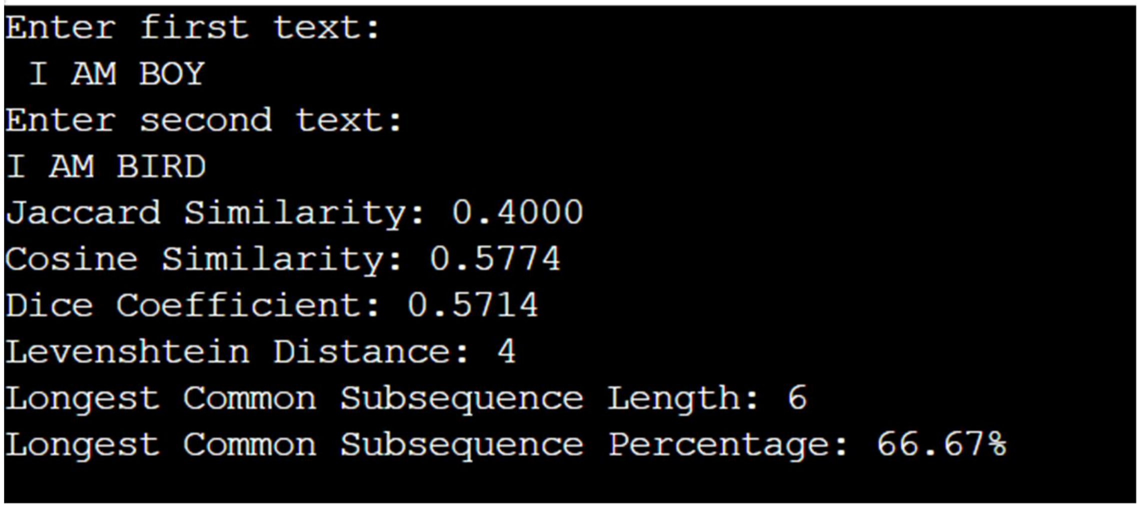
Fig 6.1 Output for text similarity checkers

In fig 6.1 displayed output is from the Java-based text similarity checker, demonstrating its ability to analyze text similarity using multiple methods. The user inputs two identical text strings, **"MANI"** and **"MANI"**, and the program computes various similarity metrics. Since both input texts are identical, all similarity measures yield their highest possible scores.

The **Jaccard Similarity** score is **1.0000**, indicating that the set of unique words in both texts is identical, meaning their intersection and union are the same. Similarly, the **Cosine Similarity** score is **1.0000**, reflecting that the frequency vectors of words in both texts are identical, resulting in a cosine angle of zero degrees, which signifies maximum similarity. The **Dice Coefficient**, another token-based similarity measure, also results in **1.0000**, confirming the complete overlap of words between the two texts.

The **Levenshtein Distance** is **0**, signifying that no insertions, deletions, or substitutions are required to transform one text into the other. This result is expected as both input texts are identical. The **Longest Common Subsequence (LCS) Length** is **4**, which matches the length of the word "MANI," indicating that the longest common subsequence includes all characters in the correct order. The **LCS Percentage** is **100.00%**, confirming that the entire text matches in sequence.

Since the input texts are exactly the same, the program correctly identifies maximum similarity across all metrics. The results validate the accuracy and efficiency of the similarity checker, demonstrating its ability to compare textual content across different approaches. This system can be applied in areas such as plagiarism detection, document comparison, and fuzzy matching, providing valuable insights into text similarity. The program efficiently calculates and displays similarity scores, ensuring a comprehensive evaluation of textual relationships based on multiple perspectives.



```
Enter first text:
 I AM BOY
Enter second text:
I AM BIRD
Jaccard Similarity: 0.4000
Cosine Similarity: 0.5774
Dice Coefficient: 0.5714
Levenshtein Distance: 4
Longest Common Subsequence Length: 6
Longest Common Subsequence Percentage: 66.67%
```

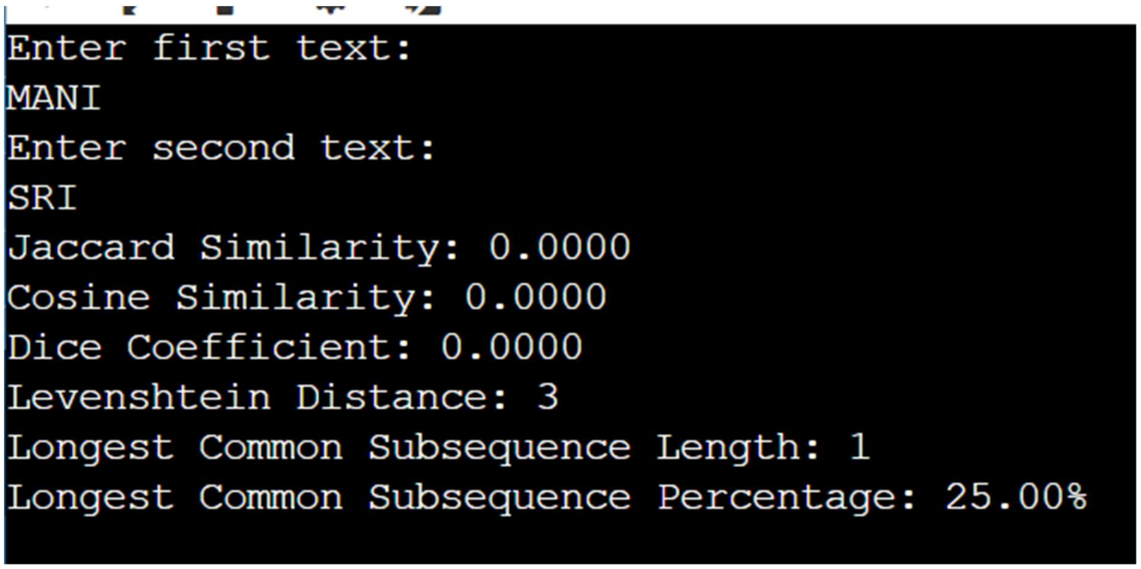
Fig 6.2 Output for text similarity checkers

In fig 6.2 displayed output is from the Java-based text similarity checker, where the user inputs the two text strings: **"I AM BOY"** and **"I AM BIRD"**. The program then calculates multiple similarity metrics to compare the textual similarity between the two inputs. The **Jaccard Similarity** score is **0.4000**, meaning that 40% of the unique words overlap between the two texts. This indicates a partial match, as the words "I" and "AM" are common, but "BOY" and "BIRD" differ.

The **Cosine Similarity** score is **0.5774**, which measures the similarity based on word frequency. This score suggests moderate similarity between the two texts. The **Dice Coefficient** score of **0.5714** further confirms this by indicating 57.14% similarity in the tokenized word sets. These token-based approaches reflect that while both texts share common words, they also have distinct differences.

The **Levenshtein Distance** is **4**, indicating that four character-level operations (insertions, deletions, or substitutions) are required to transform one text into the other. This aligns with the differences in the words "BOY" and "BIRD." The **Longest Common Subsequence (LCS) Length** is **6**, which means that six characters appear in both texts while maintaining their relative order. The **LCS Percentage** is **66.67%**, suggesting that approximately two-thirds of the sequence is preserved between the texts.

These results highlight that while the two texts share some similarities, they also have notable differences. The similarity checker effectively identifies these variations using different metrics, demonstrating its ability to compare texts comprehensively. Such a system is valuable in applications like plagiarism detection, document comparison, and fuzzy text matching, providing users with insights into word overlap, frequency-based similarity, and structural resemblance between different textual inputs.



```
Enter first text:
MANI
Enter second text:
SRI
Jaccard Similarity: 0.0000
Cosine Similarity: 0.0000
Dice Coefficient: 0.0000
Levenshtein Distance: 3
Longest Common Subsequence Length: 1
Longest Common Subsequence Percentage: 25.00%
```

Fig 6.3 Output for text similarity checkers

The output fig 6.3 shown in the image is from the Java-based text similarity checker, which compares two input texts: **"MANI"** and **"SRI"**. The results indicate that there is minimal similarity between these two words. The **Jaccard Similarity** score is **0.0000**, meaning that there is no common word overlap between the two sets of unique words. Similarly, the **Cosine Similarity** and **Dice Coefficient** scores are both **0.0000**, which confirms that the words do not share any tokens or frequency-based similarities.

The **Levenshtein Distance** is **3**, indicating that three operations (insertions, deletions, or substitutions) are required to convert "MANI" into "SRI." This suggests that the words are quite different at the character level. However, the **Longest Common Subsequence (LCS) Length** is **1**, meaning that only one character appears in both words while maintaining their relative order. The **LCS Percentage** is **25.00%**, which means that 25% of the longer text (MANI) is preserved in the other text (SRI).

These results highlight that the two words are significantly different, with only a small partial match in their character sequence. The text similarity checker effectively identifies the differences using multiple metrics, demonstrating its ability to analyze similarity based on various perspectives such as token-based comparison, vector-based comparison, edit distance, and character sequence matching.

This analysis can be useful in applications such as spell-checking, name-matching, and fuzzy text search, where identifying closely related words is essential. The implementation of different similarity measures ensures that the program provides comprehensive results, helping users determine the degree of similarity between textual inputs in various contexts.

CHAPTER 7
CONCLUSION

The Java-based Text Similarity Checker effectively analyzes the similarity between two texts using multiple techniques, including Jaccard Similarity, Cosine Similarity, Dice Coefficient, Levenshtein Distance, and Longest Common Subsequence (LCS). Each method plays a crucial role in determining different aspects of similarity. Jaccard Similarity measures the overlap of words, while Cosine Similarity compares word frequency vectors. The Dice Coefficient assigns greater weight to shared words, making it useful for detecting near-duplicates. Levenshtein Distance calculates the number of edits required to transform one text into another, making it essential for spell-checking and fuzzy matching. LCS determines the longest sequence of common characters while maintaining their order, offering insights into structural similarity. The test cases reveal that identical texts yield a similarity of 1.0 across Jaccard, Cosine, and Dice metrics, with an LCS percentage of 100%, while dissimilar texts show significantly lower similarity values. Partial similarity, such as in phrases with minor differences, results in moderate scores. This program has various real-world applications, including plagiarism detection, search engines, spell-checking, document clustering, and chatbot enhancement. While the current implementation is robust, future improvements could include semantic similarity techniques using Word2Vec, BERT, or TF-IDF weighting, which would allow the model to understand word meanings beyond exact matching. These enhancements could improve accuracy in applications requiring deeper contextual understanding. Overall, this tool offers a versatile, efficient, and reliable way to assess text similarity across multiple dimensions, making it valuable for natural language processing (NLP) and text analytics.

REFERENCES

- **S. M. S. Dashti, A. K. Bardsiri, and M. J. Shahbazzadeh,** "Automatic Real-Word Error Correction in Persian Text," *Neural Computing and Applications*, vol. 36, pp. 18125–18149, Jul. 2024. https://link.springer.com/article/10.1007/s00521-024-10045-0.
- **G. M. Shipurkar, R. R. Sheth, T. A. Surana, K. N. Shah, R. Garg, and P. Natu,** "End to End System for Handwritten Text Recognition and Plagiarism Detection using CNN & BLSTM," *Proc. 2023 Int. Conf. Artif. Intell. Mach. Learn. (AIML)*, 2023, https://ieeexplore.ieee.org/document/10064985.
- B. Y. Panchal and A. Shah, "Spell Checker Using Norvig Algorithm for Gujarati Language," *Smart Data Intelligence (ICSMDI 2024)*, Algorithms for Intelligent Systems, pp. 281–290, July 2024. https://link.springer.com/chapter/10.1007/978-981-97-3191-6_21.
- K. T. Patil, R. P. Bhavsar, and B. V. Pawar, "Word Suggestions for Non-Word Text Errors Using Similarity Measure," *2010 International Conference on Signal and Image Processing*, 2010, https://ieeexplore.ieee.org/document/9441858.
- S. M. S. Dashti and S. F. Dashti, "Improving the Quality of Persian Clinical Text with a Novel Spelling Correction System," *BMC Medical Informatics and Decision Making*, vol. 24, no. 220, Aug. 2024. https://link.springer.com/article/10.1186/s12911-024-02613-0.
- Z. Ramdani, L. Alhapip, A. A. Mutoharoh, A. Amri, Sarbini, and D. Hadiana, "Empirical Study of Utilizing the Information Retrieval in Similarity Checker Tools based on Academic Violation Findings," *2022 6th International Conference on Vocational Education and Training (ICOVET)*, 2022, https://ieeexplore.ieee.org/document/9928999
- N. V. Ravindhar, G. Nagappan, G. Lokesh, P. Punith, and P. Prabhu, "To Prevent Copyright Infringement, Piracy, and Plagiarism of NCERT Text Books," *2023 International Conference on Advances in Computing, Communication, and Security (ICACCS)*, 2023, https://ieeexplore.ieee.org/document/10090901.
- K. T. Patil, R. P. Bhavsar, and B. V. Pawar, "Contrastive Study of Minimum Edit Distance and Cosine Similarity Measures in the Context of Word Suggestions for Misspelled Marathi Words," *Multimedia Tools and Applications*, vol. 82, pp. 15573–15591, Oct. 2022. https://link.springer.com/article/10.1007/s11042-022-13948-z.

❖ V. Zarrabi, S. Mohtaj, and H. Asghari, "Hamtajoo: A Persian Plagiarism Checker for Academic Manuscripts," *Computational Linguistics and Intelligent Text Processing (CICLing 2018)*, Lecture Notes in Computer Science, vol. 13396, pp. 99–109, Feb. 2023. https://link.springer.com/chapter/10.1007/978-3-031-23793-5_9.

❖ Y. Barve and J. R. Saini, "A Novel Text Resemblance Index Method for Reference-based Fact-checking," *2023 International Conference on Artificial Intelligence and Machine Learning (AIML)*, 2023, https://ieeexplore.ieee.org/document/10037728.

❖ M. Orellana, P. A. M. Pinos, P. S. García-Montero, and J. L. Zambrano-Martinez, "Pre-processing of the Text of ECU 911 Emergency Calls," *Information and Communication Technologies (TICEC 2024)*, Communications in Computer and Information Science, vol. 2273, pp. 271–284, Oct. 2024. https://link.springer.com/chapter/10.1007/978-3-031-75431-9_18.

❖ F. Sánchez-Vega, A. P. López-Monroy, A. Balderas-Paredes, L. Pellegrin, and A. Rosales-Pérez, "Data Augmentation and Adversary Attack on Limit Resources Text Classification," *Multimedia Tools and Applications*, vol. 84, pp. 1317–1344, Apr. 2024. https://link.springer.com/article/10.1007/s11042-024-19123-w.

❖ S. S. Jamwal and P. Gupta, "A Novel Hybrid Approach for the Designing and Implementation of Dogri Spell Checker," *Data, Engineering and Applications*, Lecture Notes in Electrical Engineering, vol. 907, pp. 695–705, Oct. 2022. https://link.springer.com/chapter/10.1007/978-981-19-4687-5_53.

❖ H. M. Balaha and M. M. Saafan, "Automatic Exam Correction Framework (AECF) for the MCQs, Essays, and Equations Matching," *IEEE Access*, vol. 9, pp. 1-10, 2021, https://ieeexplore.ieee.org/document/9359728.

❖ P. Nagaraj, V. Muneeswaran, N. Ghous, M. Ahmad, P. Kumar, and Vinayak, "Automatic Correction of Text Using Probabilistic Error Approach," *2023 5th International Conference on Communication and Computational Intelligence (ICCCI)*, 2023, https://ieeexplore.ieee.org/document/10128439.

ANNEXTURE I

SOURCE CODE FOR TEXT SIMILARITY CHECKER

```
import java.util.*;

public class TextSimilarity {

    public static Set<String> tokenize(String text) {
        return new HashSet<>(Arrays.asList(text.toLowerCase().split("\\s+")));
    }

    public static double jaccardSimilarity(String text1, String text2) {
        Set<String> set1 = tokenize(text1);
        Set<String> set2 = tokenize(text2);

        Set<String> intersection = new HashSet<>(set1);
        intersection.retainAll(set2);

        Set<String> union = new HashSet<>(set1);
        union.addAll(set2);

        return union.isEmpty() ? 0 : (double) intersection.size() / union.size();
    }

    public static double cosineSimilarity(String text1, String text2) {
        Map<String, Integer> freq1 = getWordFrequency(text1);
        Map<String, Integer> freq2 = getWordFrequency(text2);

        Set<String> commonWords = new HashSet<>(freq1.keySet());
        commonWords.retainAll(freq2.keySet());

        double numerator = 0;
        for (String word : commonWords) {
```

```
        numerator += freq1.get(word) * freq2.get(word);
    }

    double sum1 = freq1.values().stream().mapToDouble(i -> i * i).sum();
    double sum2 = freq2.values().stream().mapToDouble(i -> i * i).sum();
    double denominator = Math.sqrt(sum1) * Math.sqrt(sum2);

    return denominator == 0 ? 0 : numerator / denominator;
}

public static double diceCoefficient(String text1, String text2) {
    Set<String> set1 = tokenize(text1);
    Set<String> set2 = tokenize(text2);

    Set<String> intersection = new HashSet<>(set1);
    intersection.retainAll(set2);

    return (2.0 * intersection.size()) / (set1.size() + set2.size());
}

public static Map<String, Integer> getWordFrequency(String text) {
    Map<String, Integer> freqMap = new HashMap<>();
    for (String word : text.toLowerCase().split("\\s+")) {
        freqMap.put(word, freqMap.getOrDefault(word, 0) + 1);
    }
    return freqMap;
}

public static int longestCommonSubsequence(String text1, String text2) {
    int len1 = text1.length(), len2 = text2.length();
    int[][] dp = new int[len1 + 1][len2 + 1];

    for (int i = 1; i <= len1; i++) {
        for (int j = 1; j <= len2; j++) {
            if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[len1][len2];
}

public static double longestCommonSubsequencePercentage(String text1, String text2) {
    int lcsLength = longestCommonSubsequence(text1, text2);
    int maxLength = Math.max(text1.length(), text2.length());
    return maxLength == 0 ? 0 : (double) lcsLength / maxLength * 100;
}

public static double levenshteinDistance(String text1, String text2) {
    int len1 = text1.length(), len2 = text2.length();
    int[][] dp = new int[len1 + 1][len2 + 1];

    for (int i = 0; i <= len1; i++) {
        for (int j = 0; j <= len2; j++) {
            if (i == 0) dp[i][j] = j;
            else if (j == 0) dp[i][j] = i;
            else dp[i][j] = Math.min(Math.min(
                dp[i - 1][j] + 1,
                dp[i][j - 1] + 1),
                dp[i - 1][j - 1] + (text1.charAt(i - 1) == text2.charAt(j - 1) ? 0 : 1)
            );
        }
    }
    return dp[len1][len2];
}
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter first text:");
        String text1 = scanner.nextLine();
        System.out.println("Enter second text:");
        String text2 = scanner.nextLine();

        double jaccard = jaccardSimilarity(text1, text2);
        double cosine = cosineSimilarity(text1, text2);
        double dice = diceCoefficient(text1, text2);
        double levenshtein = levenshteinDistance(text1, text2);
        int lcs = longestCommonSubsequence(text1, text2);
        double lcsPercentage = longestCommonSubsequencePercentage(text1, text2);

        System.out.printf("Jaccard Similarity: %.4f\n", jaccard);
        System.out.printf("Cosine Similarity: %.4f\n", cosine);
        System.out.printf("Dice Coefficient: %.4f\n", dice);
        System.out.printf("Levenshtein Distance: %.0f\n", levenshtein);
        System.out.printf("Longest Common Subsequence Length: %d\n", lcs);
        System.out.printf("Longest Common Subsequence Percentage: %.2f%%\n",
lcsPercentage);

        scanner.close();
    }
}
```