



Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

Text Forge: A Java library for Text Processing.

By
Cian Smith

September 1, 2025

Masters Thesis

Submitted in partial fulfillment for the award of **Master of Science in Computing** to the Department of Computer Science & Applied Physics, Atlantic Technological University (ATU), Galway.

Contents

1	Introduction	1
2	Research Methodology	3
2.1	Research	3
2.2	Development	3
2.2.1	Java Features	4
2.3	Evaluation	5
2.3.1	Testing	5
2.3.2	Comparisons	7
3	Literature Review	8
3.1	Tokenisation	8
3.2	Stemming	9
3.3	Vectorisation	9
3.4	Similarity	9
3.4.1	Alignment-Based Methods	9
3.4.2	Alignment-Free methods	10
3.5	Existing Libraries	11
3.5.1	Smile NLP	11
3.5.2	Stanford CoreNLP	11
3.5.3	Apache OpenNLP	12
4	System Design	13
4.1	Similarity Measures	14
4.1.1	Alignment Based	14
4.1.2	Alignment Free	26
4.2	Stemmers	36
4.2.1	Lancaster Stemmer	37
4.2.2	Lovins Stemmer	39
4.2.3	Porter Stemmer	41
4.3	Tokenisers	43
4.3.1	Byte-Pair Encoding (BPE)	44
4.3.2	N-gram	49
4.3.3	Shingle	50

4.4	Vectorisers	51
4.4.1	Bag-Of-Words	51
4.4.2	TF-IDF	53
4.5	Tool Box	56
4.5.1	Matrix Loader	56
4.5.2	Scrubber	57
4.5.3	String Exporter	57
5	System Evaluation	58
5.1	Testing	58
5.2	Comparisons	59
5.2.1	Available Algorithms	59
5.3	Accuracy	61
5.3.1	Tokenisers	61
5.3.2	Stemmers	61
5.3.3	Similarity measures	62
5.3.4	Vectorisers	64
5.4	Performance	65
5.4.1	Stemmers	66
5.4.2	Byte-Pair Encoding	66
6	Conclusion	67

List of Figures

4.1	TextForge directory structure.	13
4.2	Mathematical definition of the Hamming distance between two equal-length strings. [1]	15
4.3	Recursive definition of the Levenshtein distance $D(i, j)$ between the first i characters of string s and the first j characters of string t . [2]	17
4.4	Recursive definition of the restricted Damerau–Levenshtein distance $D(i, j)$. [2] [3]	18
4.5	Recurrence relation for the Needleman–Wunsch global alignment algorithm, using $D(i, j)$ for the alignment score. [4]	20
4.6	Recurrence relation for the Smith–Waterman local alignment algorithm, using $D(i, j)$ for the alignment score. [5]	21
4.7	Greedy no-gap extension of a seed k -mer: extend left and right as long as characters match exactly.	23
4.8	Definition of the Jaro similarity between strings s_1 and s_2 , where m is the number of matching characters and t is half the number of transpositions. [6]	25
4.9	Definition of the Jaro–Winkler similarity between strings s_1 and s_2 , where d_J is the Jaro similarity, ℓ is the length of the common prefix (max 4), and p is a scaling factor (typically 0.1). [6] [7]	26
4.10	Cosine similarity between vectors \mathbf{x} and \mathbf{y} . [8]	27
4.11	Euclidean distance between two points \mathbf{x} and \mathbf{y} in n -dimensional space. [9]	28
4.12	Manhattan (L_1) distance between two points \mathbf{x} and \mathbf{y} in n -dimensional space. [9]	29
4.13	Canberra distance between two vectors \mathbf{x} and \mathbf{y} in n -dimensional space. [10]	30
4.14	Chebyshev (L_∞) distance between two points \mathbf{x} and \mathbf{y} in n -dimensional space. [11]	30
4.15	Minkowski distance of order p between two points \mathbf{x} and \mathbf{y} in n -dimensional space. [12]	31
4.16	Jaccard similarity between two sets A and B . [13]	32

4.17	Generalized Jaccard similarity between two multisets S_1 and S_2 . [14] [15] [16]	33
4.18	Sørensen–Dice similarity between two sets A and B . [17] [18]	33
4.19	Overlap similarity between two sets A and B . [19]	34
4.20	Tversky similarity between two sets A and B , with parameters α and β controlling the weighting of asymmetrical differences. [20] . .	34
4.21	Estimated Sørensen–Dice similarity between two sets A and B using minhash signatures.	36
4.22	Sample of lancaster stemmer rules (Paice & Husk, 1990) showing termination (\cdot), continuation ($>$), intact check ($*$), and suffix re- placement.	37
4.23	Sample of Lovins Stemmer conditions (Julie Beth Lovins, 1968 [21], Appendix B).	41
4.24	Sample of Lovins Stemmer recoding rules (Julie Beth Lovins, 1968 [21], Appendix C).	41
4.25	Porter’s measure m counts the number of VC sequences after reduc- ing a word to alternating consonant/vowel runs. Formally, a stem has the form $(C)(VC)^m(V)$, where C is a non-empty consonant sequence and V a non-empty vowel sequence.	44
4.26	Excerpt from a BPE hex JSON file.	49
4.27	Excerpt from a BPE ASCII JSON file.	50
4.28	Term frequency of term t in document d , where $f_{t,d}$ is the raw count of t in d and the denominator sums counts of all terms in d . [22] . .	55
4.29	Inverse document frequency of term t , where N is the total number of documents in the corpus and n_t is the number of documents containing term t . [23]	56
4.30	TF-IDF weight of term t in document d , combining term frequency (TF) and inverse document frequency (IDF). [24]	56
5.1	Test Results.	58

List of Tables

3.1	Comparison of Java NLP libraries for preprocessing, ML models, and similarity measures.	12
4.1	Levenshtein distance DP matrix for transforming “cat” into “cart”. .	16
4.2	Restricted Damerau–Levenshtein distance DP matrix for transforming “form” into “from”. The optimal edit is a single adjacent transposition of ‘o’ and ‘r’.	18
4.3	Needleman–Wunsch DP matrix for “cat” (rows) vs. “cart” (columns); match = +1, mismatch = −1, gap = −1.	20
4.4	Smith–Waterman DP matrix for “carts” vs. “carting”; match = +1, mismatch = −1, gap = −1. Bold cells show one optimal local trace-back.	21
4.5	BLOSUM30 substitution matrix [25] with bolded self-match scores on the diagonal.	57
5.1	Availability of NLP features across libraries	61
5.2	Comparison of Lancaster stemming results between TextForge and Smile	62
5.3	Comparison of stemmer times between TextForge and Smile (in milliseconds)	66
5.4	Comparison of BPE training times across different implementations	66

Abstract

Natural Language Processing (NLP) libraries in Java tend to be full-stack implementations, containing machine learning (ML) models as well as preprocessing tools. While useful, these libraries can often have unnecessary overhead for simpler text preprocessing applications. This dissertation addresses this issue by creating a new Java library called TextForge.

TextForge implements four core preprocessing modules: tokenisation, stemming, similarity measures, and vectorisation. Within these modules, several algorithms were introduced that are not available in widely used Java NLP libraries such as Smile NLP, OpenNLP, or CoreNLP. The library is a self-contained package without reliance on external dependencies, making it lightweight and easily integrable into existing Java projects.

The library is not without limitations. Lemmatisation and stopword removal are techniques available in many other libraries that are not present in TextForge. The stemming performance is also not as good as libraries such as Smile NLP. However, the performance of its Byte-Pair Encoding is significantly better than some other Java alternatives.

Overall, this dissertation demonstrates the feasibility and value of a dedicated preprocessing library for Java. TextForge provides a flexible foundation that broadens the range of NLP tools available to developers and offers a base for future enhancements.

Keywords - Natural Language Processing (NLP), Text Preprocessing, Stemming, Vectorisation, Tokenisation, Similarity Measures, Byte-Pair Encoding (BPE), Bag-of-Words (BoW), Term Frequency-Inverse Document Frequency (TF-IDF)

Acknowledgments

I would like to express my deepest appreciation to my supervisor, Michael Duignan, for his guidance, encouragement, and valuable feedback throughout this project. His support has been instrumental in shaping this dissertation.

To my family and friends, I am grateful for their support, patience, and encouragement without which this work would not have been possible.

Chapter 1

Introduction

From text classification to search engines to conversational Artificial Intelligence (AI), Natural Language processing (NLP) has become a central component of many modern applications. A fundamental stage in any NLP pipeline is preprocessing, which transforms raw text into a more suitable representation for analysis. Common preprocessing steps include tokenisation, stemming, lemmatisation, similarity measures, and vectorisation, which aim to enable downstream models (e.g. Machine Learning (ML) models such as Neural Networks (NN)) and algorithms to work more efficiently.

While there are many existing libraries that offer NLP solutions in Java, most of these options are designed as comprehensive frameworks that bundle together preprocessing tools with machine learning models for a smooth NLP stack. Although powerful, these frameworks can be unnecessarily complex, rigid, and heavy for applications that only require efficient preprocessing, and often limit the methods available, as they were developed with specific models and algorithms in mind. As such, developers who wish to use lightweight, modular tools are often left with limited options.

This study (the TextForge project) aims to address that gap, creating a standalone (no external library dependencies) Java library focused exclusively on core NLP preprocessing tasks. Tokenisation, stemming, similarity measures, and vectorisation methods are all offered in TextForge, in a form that is easy to integrate into Java projects. TextForge has a secondary focus on including algorithms that are not currently available with its competitors, allowing developers to choose the right algorithm for their needs. The inclusion of niche algorithms broadens the range of available preprocessing techniques, improving flexibility and reducing the limitations encountered when using existing toolkits. Additionally, TextForge aims to produce generic responses to each developer's requests, allowing the developer to manipulate the results to best fit their task.

This dissertation is structured as follows: Chapter 2 discusses the research and

development methodology used in this project. Chapter 3 provides a review of the literature relevant to the goals of the library. Chapter 4 walks through the overall system design of TextForge, discussing the exact methods and algorithms available. Chapter 5 is an evaluation of the library, with comparisons to other libraries. Finally, chapter 6 concludes this dissertation with a discussion of the objectives achieved and the limitations of the project.

Chapter 2

Research Methodology

2.1 Research

The focus of this project is the development of a new Java library implementing some of the text preprocessing steps used in NLP. As such, it was first necessary to decide what techniques and algorithms should be included. To do this, a Literature Review was conducted (Chap. 3), and common NLP and NLP adjacent terms were used as keyword searches on scholarly databases to find relevant papers and the techniques used to achieve their goals. While most of the techniques found were already being considered, this process particularly helped with time management of this project, allowing features being used in other research to be focused on first. Having said this, it was also found that often the exact techniques used in such papers were not mentioned (e.g. what form of stemming was used). The scholarly databases used for this project were IEEE Explore [26], Google Scholar [27], and ACM Digital Library [28]. Additionally, during this research, papers from between the years 2020 and 2025 were focused on. Terms searched for were Natural Language Processing (NLP), Information Retrieval (IR), Sentiment Analysis, and Text Analysis. Additionally, a key part of the Literature review looks at other available libraries that can be used to fulfil the same tasks as this dissertation.

2.2 Development

During development, the original papers introducing techniques were used as a reference (e.g. Lovins stemmer [21] includes a list of endings and their replacements). It was decided to make the inputs and outputs of each class to be as general as possible, with the intention of making it as intuitive and flexible as possible for developers. It was intended to include lemmatisation with WordNet [29], however, as this is done using an external library, it would go against the goal of creating a

library without external dependencies.

2.2.1 Java Features

Before beginning this project, it was anticipated that many recent Java features would be used during development. While there are newer features used in TextForge, often basic constructs such as standard arrays (`[]`) or characters (`char`) are used instead of potentially more convenient features such as `ArrayLists` or `StringBuilders`.

2.2.1.1 Structured Concurrency

Structured Concurrency is used throughout TextForge for multi-threading. It became a preview feature in Java 21 and is still a preview feature as of Java 24 (forth preview) [30]. The idea of Structured Concurrency is to group tasks together in a scope, instead of submitting tasks to an executor and tracking them individually. When the scope exits, the runtime ensures that all child tasks have finished.

Here is an example of how Structured Concurrency was used in `SeedAndExtend.java`:

```
try(var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    List<StructuredTaskScope.Subtask<Extension>> tasks = new
        ↪ ArrayList<>();

    for(Seed seed: seeds) {
        // Each seed can be checked independently.
        var task = scope.fork(() -> extendSeed(seed, subject,
            ↪ query, kmerLength));
        tasks.add(task);
    }

    scope.join().throwIfFailed();

    // Get each string returned by each task and return them as an
    ↪ array.
    Extension[] extensions =
        ↪ tasks.stream().map(StructuredTaskScope.Subtask::get)
        .toArray(Extension[]::new);

    return filterExtensions(extensions, kmerLength);
}
```

As Structured Concurrency is a preview feature, Java preview features must be enabled when using TextForge. This was used despite its preview status in anticipation of its release within the next Java version or two.

2.2.1.2 Records

Records were a preview features of Java 14 and finalised in Java 16 [31]. A record is a way to model immutable data without having to write getters or constructors. Its main purpose is to hold data. Records are used throughout TextForge, both within other classes (A pair within BPE.java, Sec. 4.3.1) and as dedicated files (ByteSequence Sec. 4.3.1.5). Here is an example of a record from BPE.java (Pair):

```
record Pair(int first, int second) {
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Pair)) return false;
        Pair other = (Pair) o;
        return (this.first == other.first) && (this.second ==
            ↪ other.second);
    }

    @Override
    public int hashCode() {
        return Objects.hash(first, second);
    }
}
```

Records can be created simply by using the “new” keyword and providing values for any parameters necessary:

```
Pair pair = new Pair(corpus[i], corpus[i+1]);
```

2.3 Evaluation

The TextForge library will be evaluated with tests and comparisons to the other libraries mentioned in the literature review.

2.3.1 Testing

Each of the files in the TextForge library will have associated test files (with the exception of ByteSequence.java, Extension.java, and the ToolBox classes) located

in `src/Tests`. These tests will be created with the intention of ensuring the class does what it was intended to do, and that edge cases are being handled correctly. Tests will be created after an initial implementation of a class. After tests have been created and run, refactoring will occur to correct issues in the implementation that lead to failures.

The methodology for test creation will be to prompt an Large Language Model (LLM) service with an explanation of what a class should do and provide it with the available public functions to test. This will result in JUnit tests, that will be then placed in the Tests folder and called `<Classname>Test.java` (where `<Classname>.java` is the corresponding file in `ie/atuforge`). The tests will be divided into subfolders in the same way the classes in `ie/atuforge` are (e.g. `Similarity/Alignment`, `Tokenisers`, `Stemmers`, etc.).

The output from the LLM will be evaluated and refactored where needed, especially in edge cases where the result of a function is not always clear. An example of this is the output of a stemmer (Sec. 4.2) when the input is a null or empty string.

```
// Example function call
String output = Porter.stem("");
```

In such cases, the expected result in the tests generated by the LLM was for an exception to be thrown. This is an understandable conclusion. However, in this implementation, a new empty string is returned. As such, each test case generated needs to be verified and edited when necessary. The following is a test class generated by an LLM for this study (chatGPT in this case):

```
public class HammingTest {

    @Test
    public void testExactMatch() {
        assertEquals(0, Hamming.distance("karolin", "karolin"));
        assertEquals(0, Hamming.distance("", "")); // Both empty
    }

    @Test
    public void testSimpleDifferences() {
        assertEquals(3, Hamming.distance("karolin", "kathrin"));
        // Positions differing: 3 ('r' -> 't', 'o' -> 'h', 'l' -> 'r')

        assertEquals(2, Hamming.distance("1011101", "1001001"));
        // Bits differing: positions 2, 4, 6
    }
}
```

```
}

@Test
public void testAllDifferent() {
    assertEquals(7, Hamming.distance("1010101", "0101010"));
    assertEquals(4, Hamming.distance("AAAA", "TTTT"));
}

@Test
public void testEdgeCases() {
    assertEquals(0, Hamming.distance("A", "A"));
    assertEquals(1, Hamming.distance("A", "B"));
}

@Test
public void testUnequalLength() {
    // Expect an exception or some specific handling
    assertEquals(-1, Hamming.distance("ABC", "AB"));
}

@Test
public void testLongStrings() {
    String s1 = "A".repeat(1000);
    String s2 = "A".repeat(999) + "B"; // Only last char
    ↪ differs
    assertEquals(1, Hamming.distance(s1, s2));
}
}
```

2.3.2 Comparisons

The coverage of niche algorithms for preprocessing tasks will be evaluated by comparing the available methods in each of the libraries. Any overlap with other libraries will be pointed out, as well as any algorithms that are available in TextForge but not the others. Techniques not present in TextForge, but available in other libraries will also be discussed.

Finally, a performance comparison will be done between TextForge and Smile NLP, Open NLP, and Core NLP where possible, where the times to complete tasks will be compared.

Chapter 3

Literature Review

Before text is used in Natural Language Processing (NLP) and Information Retrieval (IR), it often goes through preprocessing [32] . Preprocessing methods include normalisation, tokenisation, stemming, lemmatisation, and vectorisation. Similarity measurements are also often a core part of the NLP pipeline [33] [34]. There are currently many Java libraries that offer preprocessing methods for NLP, however these libraries include full NLP stacks (e.g. including machine learning (ML) algorithms such as Neural Networks (NN) or clustering methods), meaning they tend to be quite broad and bulky for simple text preprocessing needs. The exact preprocessing techniques included in each library varies, with each library having a different focus (e.g. Stemming vs Lemmatisation (retrieving a words root form from a dictionary)).

3.1 Tokenisation

Taking a sequence of character and converting it into tokens is a practice known as tokenisation [35]. Another way of thinking of it is the process of segmenting text into meaningful, discrete units. These units (tokens) can be as small as a single character or as large as multiple words. There are many method that can be used to achieve tokenisation, some common ones include:

1. Byte-Pair Encoding [36]: Originally developed for data compression, but now commonly used for tokenisation in NLP.
2. N-grams [37]: A simple tokeniser which slides a window over a text and creates a token from 'n' continuous characters.
3. Unigram tokenisation[38]: A probabilistic algorithm that iteratively removes tokens until a specified size is reached.

3.2 Stemming

Stemming aims to find the root of a word using heuristic, rule, or statistic based approaches. Lovins stemmer [21] is considered to be the first stemmer, and removes the endings of words based on the longest match with a list of suffixes. Porter stemmer [39] is a heuristic based approach that applies a series of steps to remove the ending of a word. The Lancaster (Paice-Husk) stemmer [40] is a rules-based suffix stripper that iteratively applies a suffix removal rule until rules can no longer be applied.

3.3 Vectorisation

Text Vectorisation is the encoding of text as a multi-dimensional array (vector) of numbers. Text can refer to documents, paragraphs, sentences, words, or even sub-words. Modern developments in vectorisation techniques have lead to the introduction of static and contextualised embeddings (e.g. Word2Vec [41], BERT [42]), however due to the complexity of these techniques, classic vectorisation techniques remain popular.

Bag-Of-Words (BoW) [43] is considered to be the first form of text vectorisation, converting documents to fixed size vectors based on the presence of words. BoW can easily be used with other types of tokens, such as N-grams, creating a Bag-Of-Ngrams (BoN).

Term Frequency-Inverse Document Frequency (TF-IDF) [24] allows the importance of a word (term) to a document to be captured in the vector. The frequency at which the term appears in the document (TF [22]) is combined with the number of documents containing the word (IDF [23]) to find the terms overall importance to the document.

3.4 Similarity

Similarity measures can be used for calculating how similar two documents are, or in the case of bioinformatics, analysing genome sequences.

3.4.1 Alignment-Based Methods

These methods attempt to align two sequences to identify regions of similarity and difference. Character insertions, deletions, substitutions, and transpositions can be considered to achieve this. Several methods have been developed to achieve this, including:

- Hamming Distance [1] finds the minimum number of substitutions between two strings of equal length.
- Levenshtein Distance [2] quantifies the minimum number of single-character edits (insertions, deletions, or substitutions) needed to transform one string into another.
- Damerau-Levenshtein Distance [3] extends Levenshtein Distance to include measuring transpositions.
- Needleman-Wunsch [4] finds the global optimal alignment string between two strings.
- Smith-Waterman [5] finds the local optimal alignment string between two strings.
- Seed and Extend find short highly similar segments in text and then extends them to find longer, possibly gapped alignments. This approach is used by BLAST [44].
- Jaro Similarity [6] is a similarity measure between two strings that allows for transpositions and typos.
- Jaro-Winkler Similarity [7] extends Jaro Similarity to reward shared prefixes between the two strings begin compared.

3.4.2 Alignment-Free methods

These method focus on the overall properties or patterns in a document. Exact positional information is ignored. Some of these methods include:

- Cosine Similarity [8] can be used to find the angle between two n-dimensional vectors.
- Euclidean Distance [45] is the straight line distance between two points in n-dimensional space.
- Manhattan Distance [46] sums the absolute difference between each of the coordinates of two strings to find the distance between them.
- Canberra Distance [10] calculates a distance value from 0 to 1, with 0 being identical strings, and one being completely different. Relative difference between coordinate values is used.

- Chebyshev Distance [47] finds the maximum absolute difference between vector coordinates and uses that as the distance between strings.
- Minkowski Distance [12] was developed by Hermann Minkowski and is a metric that includes both Manhattan and Euclidean distance as special cases.
- Jaccard Similarity [13] finds the similarity between two sets by comparing the number of shared elements to the total number of elements.
- Sørensen–Dice Similarity [17] [18] finds the similarity between two sets by dividing twice the size of their intersection by the sum of their sizes.
- Overlap Distance

3.5 Existing Libraries

There are numerous libraries available for natural language processing (NLP) in Java, each offering different combinations of preprocessing tools, linguistic analysis, and ML capabilities. This section will briefly survey three representative Java NLP libraries—Smile NLP, Stanford CoreNLP, and Apache OpenNLP—highlighting their approaches to tokenisation, stemming/lemmatisation, ML, and similarity measures. A comparison of the libraries can be seen in Tab. 3.1

3.5.1 Smile NLP

Smile (Statistical Machine Intelligence and Learning Engine) NLP [48] [49] is a comprehensive ML framework in Java. ML models, data processing and visualisation, and NLP functionality are all provided within the framework. A range of different text preprocessing techniques are included in the library, such as tokenisation (Penn-Treebank [50] style segmentation and regex based tokenisation), stemming (Lancaster and Porter stemmers), vectorisation (BoW, TF-IDF, word embeddings), and similarity methods (Cosine, Euclidean, Manhattan, Levenshtein and Damerau-Levenshtein distances are provided in the math library).

3.5.2 Stanford CoreNLP

Stanford CoreNLP [51] is a full NLP stack with coverage of eight languages. CoreNLP’s centrepiece is its pipeline, which takes text and applies processing techniques to generate annotation objects. The processing techniques that are applied are customisable, and include tokenisation (Penn-Treebank tokenisation), part-of-speech (POS) tagging, lemmatisation, and sentence splitting. CoreNLP

also provides Name Entity Recognition (NER) models and a Recurrent-Neural Network for Sentiment Analysis.

3.5.3 Apache OpenNLP

Apache OpenNLP [52] is "a machine learning based toolkit for the processing of natural language text". Common NLP tasks to prepare text for ML are provided within the library, such as tokenisation (Maximum-Entropy tokeniser), POS tagging, and lemmatisation. This library also include other useful tools such as a language detector and a document classifier.

Library	Focus	Token.	Stem./Lem.	ML Models	Similarity
Smile NLP	Lightweight preprocessing & ML	Regex, PTB-like	Stemming	Classification, clustering	Cosine, Jaccard, Minkowski on vectors
Stanford CoreNLP	Full NLP pipeline	PTB Tokenizer (rule-based)	Lemmatisation	POS, NER, parsing, sentiment	None built-in; can export vectors
Apache OpenNLP	Modular NLP with ML models	SimpleTokenizer, TokenizerME	Lemmatisation	POS, NER, sentence detection	None built-in; vector-based possible

Table 3.1: Comparison of Java NLP libraries for preprocessing, ML models, and similarity measures.

Chapter 4

System Design

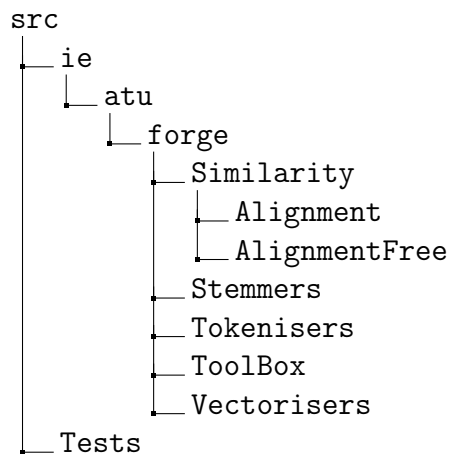


Figure 4.1: TextForge directory structure.

This section will describe the overall design of TextForge, which can be found at the following URL: <https://github.com/cianmacg/TextForge>. The library has been divided into sub-packages for each of the preprocessing tasks (Fig. 4.1). Within each of the sub-packages are classes that implement algorithms to achieve the goal of the sub-package name (e.g. “Porter” (Sec. 4.2.3) within the “Stemmers” sub-package is an algorithm to stem words to their root). Additionally, JUnit tests are located in the “Tests” package (see Sec. 2.3.1). This package is sub-divided in the same way as the main TextForge package. Each class in TextForge has an associated test file.

4.1 Similarity Measures

Text similarity measures have various uses. From plagiarism detection to biological sequencing, the wide range of applications for alignment measures make them a valuable part of the TextForge library.

Similarity algorithms have been divided into 2 categories in the library: alignment based and alignment free.

4.1.1 Alignment Based

Alignment based algorithms calculate the number of actions required to transform one string into another. In these algorithms, the position at which a character appears matters, and they tend to focus on some combination of the following 4 actions:

1. Insertions: Inserting a character at a position.
2. Deletions: Deleting a character at a position.
3. Substitutions: Changing a character at a position for another one.
4. Transposition: Swapping the positions of 2 adjacent character.

Insertions and deletions are often grouped together under the term “Indel”. As previously stated, not all algorithms make use of the 4 actions. Additionally, some algorithms create global alignments (i.e. aligning the entirety of both strings), while others perform local alignments (i.e. find the region(s) within the text that are most similar).

Needleman-Wunsch (Sec. 4.1.1.4) and Smith-Waterman (Sec. 4.1.1.5) use match, mismatch, and gap scores to find optimal alignments. Alternatively, these algorithms can take advantage of pre-calculated matrices. This is especially useful when aligning 2 biological sequences, where matrices such as PAM and BLOSUM are commonly used [25]. Unlike Hamming, Levenshtein, and Damerau-Levenshtein, the implementations of these algorithms do not return a score, but rather return optimal alignment strings.

These classes can be imported using the following code:

```
import ie.atu.forge.Similarity.Alignment.*;
```

With the exception of Seed-And-Extend (Sec. 4.1.1.6), every alignment based algorithm accepts both string inputs and character array inputs. It was decided to accept both to improve the flexibility of the library.

```
public static int distance(char[] s1, char[] s2)
public static int distance(String s1, String s2)
```

If a string input is passed, it will be converted into a character array before being processed.

```
public static int distance(String s1, String s2) {
    return distance(s1.toCharArray(), s2.toCharArray());
}
```

4.1.1.1 Hamming Distance

Hamming distance [1] finds the minimum number of substitutions to transform one string into another (see Fig. 4.2). A restriction of this algorithm is that both of the strings being queried must be of equal length. If this restriction is violated, an exception will be thrown:

```
if(s1.length != s2.length) throw new IllegalArgumentException("Both
↪ strings must be of equal length.");
```

At its core, this implementation compares the characters at each index of both strings and checks if they are equal. If they are different, the overall distance increases by 1.

```
for(int i = 0; i < s1.length; i++) {
    if(s1[i] != s2[i]) dist++;
}
```

After all comparisons have been completed, the distance is returned to the developer.

$$d_H(x, y) = \sum_{i=1}^n \delta(x_i, y_i)$$

$$\delta(a, b) = \begin{cases} 0 & \text{if } a = b, \\ 1 & \text{if } a \neq b \end{cases}$$

Figure 4.2: Mathematical definition of the Hamming distance between two equal-length strings. [1]

4.1.1.2 Levenshtein Distance

Levenshtein distance [2] allows for substitutions, insertions, and deletions (see Fig. 4.3). As insertions and deletions are included, Levenshtein distance does not suffer from the same equal string length restriction as Hamming distance (Sec. 4.1.1.1).

The implementation of Levenshtein distance utilises dynamic programming (DP). Essentially, a 2-dimensional matrix is built, calculating the costs of substitutions, insertions, and deletions at each character position in the strings. The final distance returned is the score in the bottom right-most position in the matrix.

		c	a	r	t
	0	1	2	3	4
c	1	0	1	2	3
a	2	1	0	1	2
t	3	2	1	1	1

Table 4.1: Levenshtein distance DP matrix for transforming “cat” into “cart”.

The columns and rows are representations of the characters in each string. The algorithm begins by populating the first row and column with increasing numbers (0, 1, 2, ..., n). From this point, each individual cell is populated using the formula in Fig. 4.3. For example, for string A and string B, if the character at position i of A is equal to the character at position j of B, the score of cell(i,j) is the same as cell(i-1, j-1). Otherwise, the minimum score of cell(i-1, j) (a deletion of a character from string A), cell (i, j-1) (an insertion of a character into string A), and cell(i-1, j-1) + 1 (the substitution of character i of A to a different character) will be given to cell i,j.

This is the DP code implementation in Levenshtein.java:

```
// Initialize first row and column
for(int i = 0; i <= m; i++) {
    matrix[i][0] = i; // cost of deleting all characters from s1
}

for(int j = 0; j <= n; j++) {
    matrix[0][j] = j; // cost of inserting all characters of s2
}

// Fill in the table
for(int i = 1; i <= m; i++) {
    for(int j = 1; j <= n; j++) {
        if(s1[i-1] == s2[j-1]) {
```



```

        matrix[i][j] = matrix[i-1][j-1]; // no cost
    } else {
        matrix[i][j] = 1 + Math.min(
            matrix[i-1][j],      // deletion
            Math.min(
                matrix[i][j-1], // insertion
                matrix[i-1][j-1] // substitution
            )
        );
    }
}
}

```

$$D(i, j) = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0, \\ \min \begin{cases} D(i-1, j) + 1, \\ D(i, j-1) + 1, \\ D(i-1, j-1) + \mathbf{1}_{s_i \neq t_j} \end{cases} & \text{otherwise} \end{cases}$$

Figure 4.3: Recursive definition of the Levenshtein distance $D(i, j)$ between the first i characters of string s and the first j characters of string t . [2]

4.1.1.3 Damerau-Levenshtein Distance

Damerau-Levenshtein distance expands on Levenshtein distance (Sec. 4.1.1.2) by including transpositions [3]. This implementation of the algorithm is a restricted version: a transposition can only occur between adjacent characters (see Fig. 4.4).

Programmatically, Levenshtein distance (Sec. 4.1.1.2) and Damerau-Levenshtein are almost identical, with the latter including the following when calculating the value of a cell.:

```

// Transpositions are checked after performing the normal
↪ deletion/insertion/substitution phase
if((i > 1) && (j > 1) && (s1[i - 1] == s2[j - 2]) && (s1[i - 2] ==
↪ s2[j - 1])) {
    matrix[i][j] = Math.min(
        matrix[i][j],
        matrix[i - 2][j - 2] + 1
    );
}

```

```
    );
}
```

Once again, after every cell has been populated, the final distance returned to the developer is the score in the bottom right-most cell.

		f	r	o	m
	0	1	2	3	4
f	1	0	1	2	3
o	2	1	1	1	2
r	3	2	1	1	2
m	4	3	2	2	1

Table 4.2: Restricted Damerau–Levenshtein distance DP matrix for transforming “form” into “from”. The optimal edit is a single adjacent transposition of ‘o’ and ‘r’.

$$D(i, j) = \min \left\{ \begin{array}{l} D(i-1, j) + 1 \quad (\text{deletion}) \\ D(i, j-1) + 1 \quad (\text{insertion}) \\ D(i-1, j-1) + \begin{cases} 0, & s_i = t_j \\ 1, & s_i \neq t_j \end{cases} \quad (\text{substitution}) \\ D(i-2, j-2) + 1 \quad \text{if } i, j > 1, s_i = t_{j-1}, s_{i-1} = t_j \quad (\text{transposition}) \end{array} \right\}$$

Figure 4.4: Recursive definition of the restricted Damerau–Levenshtein distance $D(i, j)$. [2] [3]

4.1.1.4 Needleman-Wunsch

Needleman-Wunsch [4] is a global alignment algorithm that uses match, mismatch, and gap scores, or a precomputed scoring matrix to populate its DP matrix. Once this has been completed, a trace-back phase occurs, where the optimal path is traversed from the bottom right-most position in the DP matrix to position (1, 1). Along the way, optimal alignment strings are constructed and finally returned to the developer.

```
public String[] align(String s1, String s2)
public String[] align(String s1, String s2, boolean useScoringMatrix)
```

The algorithm is called using the “align” method, with an optional “useScoringMatrix” boolean controlling whether to use a scoring matrix or match, mismatch, and gap scores. If a scoring matrix has not been set, but “useScoringMatrix” has been set to true, an exception is thrown:

```
throw new IllegalStateException("useScoringMatrix was set to true,  
↪ but no scoring matrix has been set.");
```

Match, mismatch, and gap scores are 1, -1, and -1 respectively by default. Each have their own getters and setters:

```
public int getMATCH()  
public int getMISMATCH()  
public int getGAP()  
  
public void setMATCH(int newValue)  
public void setMISMATCH(int newValue)  
public void setGAP(int newValue)
```

If a scoring matrix is to be used, it must be set before calling “align”. This can be done using one of 2 methods.

```
public void setScoringMatrix(Map<String, Integer> matrix)
```

To use this method, a scoring matrix (a map of character combination pairs to their score) must be created by the developer. Alternatively, the developer can pass a path to a text file that contains the scoring matrix to the following function:

```
public void loadScoringMatrix(String path)
```

This function will parse the text file and create the expected map. The functionality of this function is described in more detail in Sec. 4.5.1. An example of a scoring matrix can be seen in Tab. 4.5.

Once the scoring system has been established, the DP matrix is created using the formula in Fig. 4.5. If the scoring system is using a scoring matrix, instead of using a match or mismatch score, the character pairs are looked up in the map, and the resulting score is used. An example of a DP matrix for “cat” and “cart” can be seen in Table 4.3.

The trace-back is then performed. From the bottom right-most position of the DP matrix, we follow the highest scores back to position (1, 1) in the matrix. Only the cell to the directly to the left, above, or diagonally above and left can be moved to. As the trace-back is performed, 2 new strings are created. The first

	-	c	a	r	t
-	0	-1	-2	-3	-4
c	-1	1	0	-1	-2
a	-2	0	2	1	0
t	-3	-1	1	1	2

Table 4.3: Needleman–Wunsch DP matrix for “cat” (rows) vs. “cart” (columns); match = +1, mismatch = -1, gap = -1.

is an optimal alignment for the first string passed (S - columns) and the second for the second string passed (T - rows). A diagonal move in trace-back will add the character at that position in S and T to their optimal alignments. A leftwards move will add the character at the position to the optimal alignment of S, but add a gap character (‘-’) to the alignment of T. An upwards move will add a gap character to the optimal alignment of S, but add the character at the position to the alignment of T. The resulting alignments of the DP matrix from Table 4.3 are “cart” and “ca-t”.

$$D(i, j) = \max \begin{cases} D(i-1, j-1) + \text{score}(s[i], t[j]) & \text{(match/mismatch)} \\ D(i-1, j) + d, & \text{(gap in s)} \\ D(i, j-1) + d, & \text{(gap in t)} \end{cases}$$

Figure 4.5: Recurrence relation for the Needleman–Wunsch global alignment algorithm, using $D(i, j)$ for the alignment score. [4]

4.1.1.5 Smith-Waterman

Smith-Waterman [5] is a local alignment algorithm that uses the same method to populate its DP matrix as the Needleman-Wunsch algorithm (Sec. 4.1.1.4) with one alteration: Smith-waterman will not allow any cell’s value to be less than 0.

The trace-back step is also different to Needleman-Wunsch. Instead of beginning at the bottom right of the DP matrix, the cell with the highest score during the matrix creation is kept track of, and the trace-back begins from there. Additionally, the trace-back stops the first time a cell with a score of 0 is entered. If all cells are equal, a diagonal cell is favoured.

The result of this is a local alignment, i.e. an alignment that is less than or equal to the maximum length of the input strings. It must also be noted that the

current implementation only keeps track of a single maximum score, leading to a single position of optimal local alignment. Ideally, if the highest DP score appears in multiple places, multiple optimal alignments would be calculated.

```
public String[] align(String s1, String s2)
```

An example of the DP matrix created by the Smith-Waterman algorithm can be seen in Tab. 4.4, with the optimal alignment highlighted in bold.

	-	c	a	r	t	i	n	g
-	0	0	0	0	0	0	0	0
c	0	1	0	0	0	0	0	0
a	0	0	2	1	0	0	0	0
r	0	0	1	3	2	1	0	0
t	0	0	0	2	4	3	2	1
s	0	0	0	1	3	3	2	1

Table 4.4: Smith–Waterman DP matrix for “carts” vs. “carting”; match = +1, mismatch = −1, gap = −1. Bold cells show one optimal local trace-back.

$$D(i, j) = \max \begin{cases} 0, \\ D(i-1, j-1) + s(x_i, y_j), & \text{(match/mismatch)} \\ D(i-1, j) - d, & \text{(deletion)} \\ D(i, j-1) - d, & \text{(insertion)} \end{cases}$$

Figure 4.6: Recurrence relation for the Smith–Waterman local alignment algorithm, using $D(i, j)$ for the alignment score. [5]

4.1.1.6 Seed-And-Extend

Seed-And-Extend is a local alignment algorithm often associated with BLAST [44]. It begins with a seeding phase, where matching substrings of a given length are searched for (called “seeds”). In this implementation, the length of a substring is controlled by the integer parameter “kmerLength”. A seed is simply the starting position in both the “subject” string and the “query” string where the substring match occurs. Since each seed is of a known length (kmerLength), there is no need to keep track of anything else.

```
record Seed (int queryIndex, int subjectIndex) {}
```

Once all of these seeds have been found, an extension phase begins. There are 2 methods of extension available in the library. The default extension approach is a greedy, no gap method (see Fig. 4.7). Essentially, this method will attempt to extend each seed left and right, as long as both the character at the position in the subject and query string match. Once there are no more left or right extensions possible, the resulting string is returned to the developer as an “Extension” object. An extension object contains the alignment text, the starting position of the text in the subject, and the starting position of the text in the query.

```
public record Extension (int subjectPos, int queryPos, String text)
```

Alternatively, Smith-Waterman (Sec. 4.1.1.5) object can be passed to the align function to be used for extension. An advantage of using a Smith-Waterman object is the ability to use a pre-computed scoring matrix for alignment of biological sequences.

```
public static Extension[] align(String subject, String query, int  
    ↪ kmerLength, SmithWaterman smithWaterman, int windowSize)
```

When passing a Smith-Waterman object, an additional “windowSize” parameter is required. This parameter controls the length of text around each seed that will be passed to the Smith-Waterman extension phase. The seed will be in the centre of this text if possible. If the seed does not have enough characters to the left or right of it for it to be in the centre while maintaining the will “window” width (i.e. the window would be cut off on either end, as it would otherwise cause an array out-of-bounds error), the window will slide right or left in order to maintain the window size. This often occurs if the seed appears at the beginning or end of text. A window of text for both the subject and query string is then passed to the Smith-Waterman object, and a normal Smith-Waterman alignment is computed. The returned alignment is the text for the extension object.

A problem occurred when trying to find the starting points for the extension object when using a Smith-Waterman object. As the Smith-Waterman extension happens in another object, a different solution is required to get the starting position of the extension in the subject and query. The attempt to solve this problem is found in the “findConsensusStartingPoints”:

```
private static int[] findConsensusStartingPoints(Seed seed, char[]  
    ↪ subjectAlignment, char[] queryAlignment, int kmerLength, String  
    ↪ subject, String query)
```

The number of non-gap characters (in the returned optimal alignment) that appear before the initial seed are counted. This value is then subtracted from the

starting positions of the seed, resulting in the starting positions of the extension. However, there are a couple of issues. Firstly, it is possible that during the extension, another identical copy of the original seed appears before the original seed. This approach assumes that the first appearance it finds of the seed in the returned alignment is the original seed, allowing for the possibility of inaccurate starting positions. For example, an alignment of strings "AAAAA" (S1) and "AAA" (S2) with a window of 2 may find a seed "AA" at the end of S1 and extend to find an optimal alignment of "AAA". When the new starting positions are then searched for, this implementation will immediately find what it believes to be the original seed at index 0 of S1, and the starting positions won't be updated. Additionally, the original seed may not appear in the extension at all. When testing with the strings "VVVVVME" and "MEVVVVV", a `kmerLength` of 2, and a `windowsSize` of 10, the seed "ME" returns an optimal alignment of "VVVVV". This occurs as the original seed appears far away from itself in the subject and query strings. As a result, when using "findConsensusStartingPoints", the original seed is not found. In this case, it was decided to simply return the starting points of the original seed. Due to these issues in the implemented approach, care is advised when using a Smith-Waterman object during Seed-And-Extend.

An advantage of using Seed-and-Extend is multiple local alignments are returned, in contrast to Smith-Waterman (Sec. 4.1.1.5), which only returns a single instance of optimal alignment.

$$i_{\text{start}} = \max \{i \leq i_0 \mid s[i' : i_0 + k - 1] = q[j' : j_0 + k - 1] \ \forall i' \in [i, i_0]\},$$

$$i_{\text{end}} = \min \{i \geq i_0 + k - 1 \mid s[i_0 : i'] = q[j_0 : i'] \ \forall i' \in [i_0 + k - 1, i]\},$$

Extension: $s[i_{\text{start}} \dots i_{\text{end}}] = q[j_{\text{start}} \dots j_{\text{end}}]$

Figure 4.7: Greedy no-gap extension of a seed k -mer: extend left and right as long as characters match exactly.

4.1.1.7 Jaro Distance

Jaro Distance [6] is a measure that returns a value between 0 and 1. It allows for typos and transpositions. In this implementation, distance and similarity functions are available. Distance is simply 1 minus the similarity (I.e. a distance of 1 is equal to a similarity of 0).

```
public static double distance(String s1, String s2)
public static double similarity(String s1, String s2)
```

The algorithm begins by calculating a range:

```
int range = Math.max((int) Math.floor((Math.max(s1.length,
↪ s2.length) / 2.0d) - 1), 0);
```

In Jaro distance, a “match” does not have to exist at the same index in both strings. As long as the character appears in both strings within the range calculated above (E.g. if the range is 5, the character at index ‘n’ of string S1 appears within 5 characters before or 5 characters after index ‘n’ of string S2), it will be considered a match when calculating Jaro distance. To help with calculating transpositions, during the match checking phase the position of a match in both strings is tracked:

```
if(s2[j] == c1) { // If there is a match, where c1 is the character
↪ at position i in string s1.
    matches++;

    matched_s1[i] = 1; // Keeping track of the match position for
↪ string s1.
    matched_s2[j] = 1; // Keeping track of the match position for
↪ string s2.

    break;
}
```

For transpositions, the position where a match occurred in string S1 is looked at, and then the next index (starting at index 0) where a match occurred in string S2 is found. The character at the match position of both strings is then compared. If the characters are identical, it is not considered a transposition. Otherwise, the transposition count is increased.

```
int transpositions = 0;
int j = 0;
// If the next appearing match is not of the same character as the
↪ index in s1, this means the characters are not in order and
↪ therefore a transposition.
for(int i = 0; i < s1.length; i++) {
    if(matched_s1[i] == 1) {
        while(matched_s2[j] == 0) j++;

        if(s1[i] != s2[j]) transpositions++;

        j++;
    }
}
```



```
    }
}
```

After using the formula (Fig. 4.8, where m is the number of matches, and t is half the number of transpositions), the Jaro similarity is found. The Jaro distance is 1 minus the similarity.

$$d_J(s_1, s_2) = \begin{cases} 0, & \text{if } m = 0, \\ \frac{1}{3} \left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \right), & \text{otherwise} \end{cases}$$

Figure 4.8: Definition of the Jaro similarity between strings s_1 and s_2 , where m is the number of matching characters and t is half the number of transpositions. [6]

4.1.1.8 Jaro-Winkler Distance

Jaro-Winkler distance [7] was proposed as an improvement over Jaro distance (Sec. 4.1.1.7). The main difference introduced by Jaro-Winkler is in rewarding shared prefixes. Like the Jaro implementation, both distance and similarity functions are available in JaroWinkler.java, with the distance being 1 minus the similarity.

```
public static double distance(String s1, String s2, double p)
public static double similarity(String s1, String s2, double p)
```

This implementation begins by calculating the Jaro similarity between the 2 strings (simply making a call to the Jaro implementation):

```
double jaro = Jaro.similarity(s1, s2);
```

The next step is to find the common prefix length between both strings (called ‘ l ’ in this implementation). The maximum prefix length is limited to 4, as suggested in the original paper [7].

```
int l = 0;
int end = Math.min(Math.min(s1.length, s2.length), 4);

for(int i = 0; i < end; i++) {
    if(s1[i] == s2[i]) l = i + 1;
    else break;
}
```

A scaling factor ‘p’ is then required. This value controls the importance of the prefix. It is provided a function parameter by the developer. If no ‘p’ is provided, the function defaults to a value of 0.1, as this was suggested in the original paper (while the paper also recommends to keep the value less than or equal to 0.25, it is not limited in this implementation).

The next step is to use the Jaro-Winkler formula (Fig. 4.9) to calculate the similarity. Finally, the distance is calculated as 1 minus the similarity and returned to the developer.

$$d_{JW}(s_1, s_2) = d_J(s_1, s_2) + \ell \cdot p \cdot (1 - d_J(s_1, s_2))$$

Figure 4.9: Definition of the Jaro-Winkler similarity between strings s_1 and s_2 , where d_J is the Jaro similarity, ℓ is the length of the common prefix (max 4), and p is a scaling factor (typically 0.1). [6] [7]

4.1.2 Alignment Free

An alignment-free measure is one which ignores positional information. If both of the strings being compared have exactly the same token composition (word or sub-word tokens), but their tokens appear in different orders, they will still be considered identical. The primary benefit of using an alignment-free measure is its performance when dealing with large volumes of text. Common use cases for such measures include plagiarism detection and text classification.

A requirement for Cosine distance (Sec. 4.1.2.1), Euclidean distance (Sec. 4.1.2.2), Manhattan distance (Sec. 4.1.2.3), Canberra distance (Sec. 4.1.2.4), Chebyshev distance (Sec. 4.1.2.5), and Minkowski distance (Sec. 4.1.2.6) is that the strings being compared must first be converted into vectors (integers or doubles). This is commonly achieved using hashing. Alternatively, the implementation of Bag-of-Words (BoW, Sec. 4.4.1) can help create count vectors (if maps from BoWs are passed as parameters to the functions, equal length count vectors will be created from them), or the implementation of Term Frequency - Inverse Document Frequency (TF-IDF, Sec. 4.4.2) can be used to create equal sized vectors (by adding both strings to the corpus, and then calling the “vectoriseDocument” function on both strings). If the provided vectors are not of equal length, an exception is thrown:

```
if(v1.length != v2.length) throw new
    ↳ IllegalArgumentException("Vectors must be the same length:
    ↳ v1.length = " + v1.length + "; v2.length = " + v2.length);
```

Jaccard, Sørensen–Dice, Overlap, and Tversky similarities compare sets of tokens (word or sub-word tokens) to calculate similarity. As such, the sets compared do not need to be of equal size. The implementations of these measures can calculate either distances or similarities (where a distance is $1 - \text{the similarity}$). Each returns a value between 0 and 1. Accepted inputs to their functions are Sets (generic sets are accepted, allowing strings, integers, etc., to be the set type) or maps of strings to integers (e.g. a map returned from a BoW), where the key set will be used.

4.1.2.1 Cosine Distance (Similarity)

Cosine distance is a method of determining how different 2 vectors are based on the angle between them. The actual magnitude of the vectors is not relevant when calculating the distance, and only their direction matters. The Cosine distance is 1 minus the Cosine similarity (see Fig. 4.10).

Both similarity and distance functions are offered in Cosine.java, and will always be a value between 0 and 1. Identical vectors will return a value of 0 distance, and 1 for similarity. The similarity calculation is mostly done in a single loop:

```
double dotProd = 0.0d, mag1 = 0.0d, mag2 = 0.0d;

for(int i = 0; i < v1.length; i++) {
    int d1 = v1[i], d2 = v2[i];

    dotProd += d1 * d2;
    mag1 += d1 * d1;
    mag2 += d2 * d2;
}

return dotProd / Math.sqrt(mag1 * mag2);
```

$$s_{\cos}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}}$$

Figure 4.10: Cosine similarity between vectors \mathbf{x} and \mathbf{y} . [8]

4.1.2.2 Euclidean Distance

Euclidean distance (also known as L_2 norm) is the distance in a straight line between 2 points in n -dimensional space, where ‘ n ’ is the size of the vectors. The implementation of the Euclidean formula (Fig. 4.11) is mostly performed within a loop, with the square root performed when returning the value:

```
double result = 0.0d;

for (int i = 0; i < v1.length; i++) {
    int d1 = v1[i];
    int d2 = v2[i];

    result += (d1 - d2) * (d1 - d2);
}

return Math.sqrt(result);
```

Identical vectors will return a value of 0. Unlike Cosine (Sec. 4.1.2.1), Euclidean distance is not constrained to a range, and can be any positive number.

$$d_E(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Figure 4.11: Euclidean distance between two points \mathbf{x} and \mathbf{y} in n -dimensional space. [9]

4.1.2.3 Manhattan Distance

Manhattan distance (also known as L_1 norm) is the sum of the absolute differences between each coordinate of the provided vectors (Fig. 4.12) (e.g. for (5, 2) and (6, -1), the coordinate differences are (1, 3). These numbers are summed to find the Manhattan distance). If an integer vector is passed, an integer is returned, if a double vector is passed, a double is returned:

```
int result = 0;

for(int i = 0; i < v1.length; i++) {
    result += Math.abs(v1[i] - v2[i]);
}
```

```
}
```

```
return result;
```

Similar to Euclidean distance (Sec. 4.1.2.2), Manhattan distance can return any positive number.

$$d_m(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^n |x_i - y_i|$$

Figure 4.12: Manhattan (L_1) distance between two points \mathbf{x} and \mathbf{y} in n -dimensional space. [9]

4.1.2.4 Canberra Distance

Canberra distance [10] is similar to Manhattan distance (Sec. 4.1.2.3), but with a sensitivity to relative differences, rather than absolute. Each coordinate difference is constrained to a range from 0 to 1. The implementation of the Canberra formula (Fig. 4.13) can be seen here:

```
double result = 0.0d;

for (int i = 0; i < v1.length; i++) {
    int d1 = v1[i];
    int d2 = v2[i];

    result += (double) Math.abs(d1 - d2) / (Math.abs(d1) +
        ↪ Math.abs(d2));
}

return result;
```

Canberra distance can return any positive number.

4.1.2.5 Chebyshev Distance

Chebyshev distance (also known as L_∞ norm) finds the maximum absolute difference between vector coordinates (Fig. 4.14). Simply put, each index of the vectors is compared, the absolute difference between them is found, and then the maximum of these values is returned to the developer as the distance:

$$d_{\text{Canberra}}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^n \frac{|x_i - y_i|}{|x_i| + |y_i|}$$

Figure 4.13: Canberra distance between two vectors \mathbf{x} and \mathbf{y} in n -dimensional space. [10]

```
double result = 0.0d;

for (int i = 0; i < v1.length; i++) {
    double difference = Math.abs(v1[i] - v2[i]);

    // Only if the difference is greater than the current maximum
    //   do we care about it.
    if(difference > result) result = difference;
}

return result;
```

$$d_{\infty}(\mathbf{x}, \mathbf{y}) = \max_{i=1, \dots, n} |x_i - y_i|$$

Figure 4.14: Chebyshev (L_{∞}) distance between two points \mathbf{x} and \mathbf{y} in n -dimensional space. [11]

4.1.2.6 Minkowski Distance

Minkowski distance [12] is a generalised distance measurement (Fig. 4.15). The type of distance calculated is controlled by a parameter ‘p’. As p increases, the importance of the maximum difference between coordinates also increases. As such, if p is infinity, the distance calculated is Chebyshev distance (Sec. 4.1.2.5). Other special values of p include 1, which will calculate Manhattan distance (Sec. 4.1.2.3), and 2, which will calculate Euclidean distance (Sec. 4.1.2.2).

```
double result = 0.0d;

for(int i = 0; i < v1.length; i++) {
```

```

        result += Math.pow(Math.abs(v1[i] - v2[i]), p);
    }

    result = Math.pow(result, 1/p);

    return result;

```

Since passing a p value of infinity is impossible, a special case is included where if a p value of -1 is passed into the distance function a Chebyshev distance will be calculated and returned to the developer:

```

// If p is infinity, the similarity becomes Chebyshev. Since it is
↪ not possible to pass infinity, -1 will represent it.
    if(p == -1.0d) {
        return Chebyshev.distance(v1, v2);
    }

```

Additionally, since a division of p is happening in the distance calculation, division by 0 errors are protected against by throwing an exception when p is 0:

```

// A p value of 0 will cause a division by 0 error.
    if(p == 0.0d) {
        throw new IllegalArgumentException("P cannot be 0. Leads to
        ↪ division by 0 error.");
    }

```

$$d_p(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^n |x_i - y_i|^p \right)^{1/p}$$

Figure 4.15: Minkowski distance of order p between two points \mathbf{x} and \mathbf{y} in n -dimensional space. [12]

4.1.2.7 Jaccard Similarity

Jaccard similarity [13] is simply the size of the intersection of 2 sets divided by the union of the 2 sets (Fig. 4.16). The result is a value ranging from 0 to 1.

```

Set<T> inter = new HashSet<T>(s1);
inter.retainAll(s2);

```

```
return (double) inter.size() / (s1.size() + s2.size() -
    ↪ inter.size());
```

Jaccard is unique in the implementations of similarity measures, as it has an second, different algorithm when maps (of strings to counts) are given to the similarity/distance functions. This algorithm is commonly referred to as "Generalised Jaccard" or "Multiset Jaccard" [14] [15] [16]. The count of words are factored into the similarity. The sum of the minimum counts for each word between the two sets is divided by the sum of the maximum of the counts (Fig. 4.17). The implementation can be seen here:

```
Set<String> union = new HashSet<>(s1.keySet());
union.addAll(s2.keySet());

for(String key: union) {
    int v1 = s1.getOrDefault(key, 0), v2 = s2.getOrDefault(key, 0);
    min_count += Math.min(v1, v2);
    max_count += Math.max(v1, v2);
}

if(max_count == 0) return 1.0d; // If every key in both sets has a
    ↪ value of 0, they are identical.

return (double) min_count / max_count;
```

To use the multiset version, "generalisedSimilarity" and "generalisedDistance" must be called.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Figure 4.16: Jaccard similarity between two sets A and B . [13]

4.1.2.8 Sørensen–Dice Similarity

Sørensen–Dice similarity [17] [18] is twice the size of the intersections of 2 sets, divided by the sum of the sizes of the sets (Fig. 4.18). The result is a value ranging from 0 to 1.

$$J_{\text{gen}}(S_1, S_2) = \frac{\sum_{k \in S_1 \cup S_2} \min(\text{count}_{S_1}(k), \text{count}_{S_2}(k))}{\sum_{k \in S_1 \cup S_2} \max(\text{count}_{S_1}(k), \text{count}_{S_2}(k))}$$

Figure 4.17: Generalized Jaccard similarity between two multisets S_1 and S_2 . [14] [15] [16]

```
Set<T> inter = new HashSet<T>(s1);
inter.retainAll(s2);

return (double) (2 * inter.size()) / (s1.size() + s2.size());
```

$$\text{Dice}(A, B) = \frac{2|A \cap B|}{|A| + |B|}$$

Figure 4.18: Sørensen–Dice similarity between two sets A and B . [17] [18]

4.1.2.9 Overlap Similarity

Overlap distance [19] divides the intersection size of 2 sets by the size of the smaller set (Fig. 4.19). This method of similarity/distance is useful when comparing 2 sets of very different sizes. The result is a value ranging from 0 to 1. The implementation can be seen here:

```
if(s1.isEmpty() && s2.isEmpty()) return 0.0d;
if(s1.isEmpty() || s2.isEmpty()) return 1.0d;

Set<T> inter = new HashSet<T>(s1);
inter.retainAll(s2);

double divider = Math.min(s1.size(), s2.size());

return 1 - (double) inter.size() / divider;
```

$$\text{Overlap}(A, B) = \frac{|A \cap B|}{\min(|A|, |B|)}$$

Figure 4.19: Overlap similarity between two sets A and B . [19]

4.1.2.10 Tversky Similarity

Tversky similarity [20] or index divides the size of the intersection of 2 sets by the size of the intersection plus the size of the items in set A but not set B plus the size of the items in set B but not set A (Fig. 4.20). The size of items in A but not B , and the size of items in B but not A are multiplied by α and β respectively. This controls their importance to the overall similarity. The result is a value ranging from 0 to 1. The implementation can be seen here:

```
Set<T> setA = new HashSet<T>(s1);
Set<T> setB = new HashSet<T>(s2);
Set<T> aNotB = new HashSet<T>(setA);
Set<T> bNotA = new HashSet<T>(setB);
aNotB.removeAll(setB);
bNotA.removeAll(setA);

Set<T> inter = new HashSet<T>(setA);
inter.retainAll(setB);

return (double) inter.size() / (inter.size() + a*(aNotB.size()) +
    ↪ b*(bNotA.size()));
```

Where α is represented by ‘a’ and β is represented by b.

There are special cases for α and β values. When $\alpha = \beta = 1$, the result is a standard Jaccard similarity (Sec. 4.1.2.7). When $\alpha = \beta = 0.5$, the result is a Sørensen–Dice similarity.

$$T(A, B) = \frac{|A \cap B|}{|A \cap B| + \alpha|A \setminus B| + \beta|B \setminus A|}$$

Figure 4.20: Tversky similarity between two sets A and B , with parameters α and β controlling the weighting of asymmetrical differences. [20]

4.1.2.11 MinHash

MinHash [53] is a method used to generalise some other similarity measures. It is particularly useful when dealing with large sets (potentially millions of entries) [54]. The idea is to generate a number of hashing functions (e.g. 128, 256, etc.) and then apply them to each of the values in the set (if the values are strings, an initial hashing function must be applied to convert them to a number). Here is the code to create hashing functions (where *k* is the number of hashing functions to be generated):

```
funcs = new int[k][2]; // Each function will have an a and b values
    ↪ (hence the 2).
Random rand = new Random();

for(int i = 0; i < k; i++) {
    funcs[i][0] = rand.nextInt(prime - 1) + 1; // Ensures non-0
    ↪ value.
    funcs[i][1] = rand.nextInt(prime);
}
```

A hashing function is essentially 2 integer numbers that will be used to manipulate the set entries later. The variable “prime” is the largest prime number for a 32-bit integer. A large prime number helps with even distributions or random numbers. Its use here guarantees a random number less than the prime value.

Once a hashing function has been applied to each value in the set, the lowest value observed is stored and next hashing function is applied to each value. A hashing function is applied in the following way (where the values of *a* and *b* were generated previously in the previous code block):

```
private int applyFunc(int a, int b, int val) {
    long hash = ((long) a * val + b) % prime;

    return (int) hash;
}
```

The use of a large prime number in a hashing function helps with evenly distributing the hashed values.

After the hashing functions have been applied to both sets, and the minimum values for each are obtained, they are then compared. The count of matching values is kept track of, and then divided by the total number of hashing functions. The result is an approximation of the Jaccard similarity. Here is the code for the approximation:

```

int[] v1 = minHashSet(s1), v2 = minHashSet(s2);
int count = 0;

for(int i = 0; i < k; i++) {
    if(v1[i] == v2[i]) count++;
}

return (double) count / k;

```

Additionally, Sørensen–Dice similarity (Sec. 4.1.2.8) can be derived from a Jaccard approximation (Fig. 4.21).

$$\widehat{\text{Dice}}(A, B) = \frac{2 \hat{J}(A, B)}{1 + \hat{J}(A, B)}$$

Figure 4.21: Estimated Sørensen–Dice similarity between two sets A and B using minhash signatures.

The current implementation can be improved in a notable way. The conversion of a set entry to a valid integer is done by using the `hashCode` function. This is a rather poor way of obtaining an integer representation of, for example, a string, as it returns a 32-bit integer, so collisions are likely for large sets. Additionally, for shorter strings the distribution can be poor. A solution is to use something like `MurmurHash3`, which is designed for uniformity. It is available from external libraries such as `Apache Commons Code` [55].

4.2 Stemmers

Stemming is an algorithmic approach to reducing words to their root (or stem). This is done through ending removals, additions, or both. In their paper introducing the Lancaster stemmer [40], Paice describes stemming as "the removal (or sometimes replacement) of inflectional and derivational suffixes". Stemmers are commonly used in Natural Language Processing (NLP) and Information Retrieval (IR) in an attempt to group words with common stems (and hopefully meanings) together (e.g. "run", "runs", "running", and "runner" may all be reduced to "run"). This has the benefit of improving performance [39] by reducing the number of unique words being worked on (reducing the vocabulary size), however this comes at the expense of accuracy (unrelated words can be reduced to the same stem e.g. "organisation" and "organ" may both be reduced to "organ").

For each stemmer implementation, a string, or an array of strings, can be stemmed using the “stem” function:

```
public static String stem(String input)
public static String[] stem(String[] inputs)
```

It is assumed that the developer will not pass in a full sentence as a single string, and the each word will be its own string.

Additionally, each implementation has the rules coded in, stored in a map. A potential improvement would be to store the rules in external files (e.g. text, Json, etc.). This would allow rules to be easily changed and experimented with, without the need to change any code.

4.2.1 Lancaster Stemmer

The Lancaster stemmer (also known as the Paice-Husk stemmer) [40] is an iterative rules based stemmer. Rules are applied based on the final letters of a word. An example of some rules can be seen here:

```
ai*2. { -ia > - if intact }
al*2. { -la > - if intact }
bb1. { -bb > -b }
nois4j> { -sion > -j }
ing3> { -ing > - continue }
```

Figure 4.22: Sample of lancaster stemmer rules (Paice & Husk, 1990) showing termination (.), continuation (>), intact check (*), and suffix replacement.

In each rule, the letters that appear before the number indicate the ending the rule can be applied to. If a ‘*’ appears in a rule, the word must be “intact”. An intact word is one that is yet to have had any stemming rules applied to it. The number in the rule indicates how many characters should be removed from the end of the word (starting from the last character). Any letters that appear after the number should then be appended to the end of the word. Finally, a ‘.’ indicates that stemming should be concluded, while ‘>’ indicates that stemming should continue, and the rules should be checked on the output from the rule that was just applied. It should be noted that in the original paper, the ending on the left of the number in the rules are reversed (e.g. “ai*2.” applies to a word ending in “ia”, not “ai”).

Before the application of a rule on a word is accepted, the resulting stem must satisfy a condition. If the word begins with a vowel, the stem must contain at least 2 letters after stemming. If the word begins with a consonant, the stem must

contain at least 3 letters, one of which must be a vowel or the letter ‘y’. These “Acceptability Conditions” help prevent against over-stemming (reducing a word to something that offers no meaning). Without these conditions, Paice states in the paper [40], “the words “rent”, “rant”, “rice”, “rage”, “rise”, “rate”, “ration” and “river” would all be reduced to “r””.

In this implementation, it was decided to store each rule in a map, where the key is the ending the rule applies to. Intact rules are put into a separate map:

```
rules.put("mui", "3.");
intactRules.put("mu", "2.");
```

The values of each entry contain the action to be applied. The first character in the value is always the number of letters to remove. The final character in the value will indicate whether to keep looking for endings to remove (‘>’) or to end the stemming process (‘.’). Any character in between will be added to the end of the stem.

The approach to applying rules begins by looking at the largest possible endings, and decreasing the ending size whenever no ending is found. Intact rules will only be searched for on the first pass. If a rule is applied, and doesn’t contain the termination character, the resulting stem is once again checked for applicable rules. This continues until a termination character is found or no applicable rules can be found. Before a stem is accepted, it is checked using a “verifyStem” function to make sure the acceptability conditions are satisfied:

```
// Before we go any further, make sure the stem satisfies the
→ minimum stem rules. If it doesn't, return the original input.
if(!verifyStem(stem)){
    return input;
}
```

If the stem was not accept, the ending length is decreased and a smaller ending is searched for:

```
if(rule!=null) {
    char[] stem = applyRule(input, rule.toCharArray());

    if(!Arrays.equals(input, stem)) { // If the rule was applied,
→ and the stem was accepted, the 'stem' will be different
→ from the input.
        return stem;
    }
}
```

```
// No rule was found, reduce the ending size and try again.
endLen--;
```

Having re-read the original paper [40], the current approach to selecting which rule to apply is not correct. Paice states the following when describing the algorithm: "Inspect the final letter of the form; if there is no section corresponding to that letter, then terminate; otherwise, consider the first rule in the relevant section". This indicates that rules should be checked based on their order in the rules list, not by ending length, as the current approach does. Due to this, the implementation will produce different results to a more faithful implementation of the Lancaster stemmer.

4.2.2 Lovins Stemmer

The Lovins stemmer is a “two-phase stemming system” [21]. It begins by removing the longest possible ending (or suffix) from the word to retrieve its stem. These endings come from the original paper, and all have an associated condition the resulting stem must satisfy for the stem to be considered valid (see Fig. 4.23). If the stem is not valid, the process is repeated with a different, shorter ending (if a matching one exists). The second phase handles “spelling exceptions” by recoding the end of a stemmed word. This handles cases such as “absorbing” and “absorption”, where the stem of both words should be the same. Without the recoding step, these words would be stemmed to “absorb” and “absorpt” respectively. In this case, the recoding step changes the final ‘rpt’ in “absorpt” to “rb”, resulting in both “absorbing” and “absorption” being stemmed to “absorb” (this is talked about in more detail in Section V of the original paper [21]). Some of the recoding rules can be seen in Fig. 4.24.

The implementation stores each ending and their corresponding condition in an array of maps. The index in the array of the map for each ending is the length of the ending - 1. Each map is populated in the following way:

```
// 11
endings[10] = new HashMap<>();
endings[10].put("alistically", "b");
endings[10].put("arizability", "a");
endings[10].put("arisability", "a");
endings[10].put("izationally", "b");
endings[10].put("isationally", "b");
...
// 1
endings[0] = new HashMap<>();
```

```

endings[0].put("a", "a");
endings[0].put("e", "a");
endings[0].put("i", "a");
endings[0].put("o", "a");
endings[0].put("s", "w");
endings[0].put("y", "b");

```

When searching for an ending, the longest possible ending is looked for. As the minimum stem length is suggested to be 2 (from Section III of Lovins' original paper [21]), the search begins with endings that are at most the length of the input minus 2. A substring of length 'n' is taken from the input, and searched for in the appropriate map (endings[n]). If the ending exists, a condition (string) is returned, otherwise null will be returned and the process repeats with a shorter ending.

If there is a condition string (i.e. the ending exists in the map), a "conditions" map is searched using the string. The conditions map tells us what function to call to validate the stem if the ending is removed:

```

private static final Map<String, Condition> conditions =
    ↪ Map.ofEntries(
        Map.entry("a", Lovins::conditionA),
        Map.entry("b", Lovins::conditionB),
        ...
        Map.entry("bb", Lovins::conditionBb),
        Map.entry("cc", Lovins::conditionCc)
    );

```

A "Condition" is a function that follows the following interface:

```

@FunctionalInterface
interface Condition {
    boolean test(char[] input);
}

// Example Condition functions
// No restrictions on stem
private static boolean conditionA(char[] input) {
    return true;
}

// Minimum stem length == 3
private static boolean conditionB(char[] input) {
    return input.length >= 3;
}

```


If the condition function returns true, the ending is removed. Regardless of if an ending was removed or not, the second phase is moved to. In this phase, the ending of the stem is checked to see if it needs to be changed, based on the recoding rules. Here is an example for recoding rule 1:

```
// Remove one of double b, d, g, l, m, n, p, r, s, t
if(input[stemLength - 1] == input[stemLength - 2]) {
    char[] values = {'b', 'd', 'g', 'l', 'm', 'n', 'p', 'r', 's',
        ↪ 't'};
    for(char value : values) {
        if(input[stemLength - 1] == value) {
            char[] result = new char[stemLength - 1];
            System.arraycopy(input, 0, result, 0, result.length);
            return result;
        }
    }
}
```

- A ... No restrictions on stem
- B ... Minimum stem length = 3
- C ... Minimum stem length = 4
- D ... Minimum stem length = 5
- E ... Do not remove ending after e

Figure 4.23: Sample of Lovins Stemmer conditions (Julie Beth Lovins, 1968 [21], Appendix B).

- 1 ... Remove one of double b, d, g, l, m, n, p, r, s, t
- 2 ... iev -> ief
- 3 ... uct -> uc
- 4 ... umpt -> um
- 5 ... rpt -> rb

Figure 4.24: Sample of Lovins Stemmer recoding rules (Julie Beth Lovins, 1968 [21], Appendix C).

4.2.3 Porter Stemmer

The Porter stemmer [39], like the Lancaster (Sec. 4.2.1) and Lovins (Sec. 4.2.2) stemmers, is a rules based stemmer. It is divided in to series of steps, all of which

are applied to each word to reduce them to a stem. The main difference between the Porter stemmer and others lies in the calculation of a “measure” (Fig. 4.25). Each character of a word is labelled ‘C’ (for a consonant) or ‘V’ (for a vowel). Adjacent Cs are merged into a single C, and the same for Vs. The measure is then the number of times a VC combination is observed. This measure is often used to decide whether a rule should be applied to a word or not.

```
// From step1B

// If the measure (m) is > 0 and the end of the word is 'eed', we
→ remove the last 'd', making the new ending 'ee'.
if(input.length > 3 && input[input.length - 3] == 'e') {
    result = new char[input.length - 1];
    System.arraycopy(input, 0, result, 0, result.length);

    if(measure(result) > 0) {
        return result;
    }

    // If the measure is 0, no stemming is needed and the original
    → input is returned.
    return input;
}
```

Occasionally a stem must satisfy a condition to be valid. An example of such a condition is “*S” (from Porter’s paper [39]), where the resulting stem must end with ‘s’ (or any specified letter).

```
// Check if the stem ends with a specific letter.
private static boolean conditionS(char[] input, char letter) {
    return input[input.length - 1] == letter;
}
```

In summary, steps are simply replacing an ending with another ending, or removing an ending, should a condition be satisfied. The implementation of the Porter stemmer simply applies each step one after the other.

```
char[] chars = input.replaceAll("[^a-zA-Z ]",
    → "").toLowerCase().toCharArray();

result.append(
    step5B(
```

```

        step5A(
            step4(
                step3(
                    step2(
                        step1C(
                            step1B(
                                step1A(chars)))))))))
    );

    return result.toString();

```

As the number of endings in steps 1 and 5 are quite low, it was decided that it was unnecessary to store them in a data structure. Steps 2, 3, and 4, however, contain quite a number of endings, so they are each stored in their own maps (or set in step 4's case, where endings are not replaced, only removed).

```

step2Endings.put("ational", "ate");
step2Endings.put("tional", "tion");
...
step3Endings.put("icate", "ic");
step3Endings.put("ative", "");
...
step4Endings.add("al");
step4Endings.add("ance");

```

4.3 Tokenisers

Tokenisation has been defined as “the practice of converting strings of characters from an alphabet into sequences of tokens over a vocabulary” [35]. A simpler definition of tokenisation is the process of representing text (sentences, paragraphs, documents, etc.) with smaller meaningful units (tokens). This is generally achieved in one of 2 ways:

1. Beginning with a large text, split it up into smaller tokens (N-gram Sec. 4.3.2, Shingle Sec. 4.3.3).
2. Beginning with characters, build them up into larger tokens of meaning, based on a larger text (training data) (Byte-Pair Encoding (BPE) Sec. 4.3.1).

The result of a tokenisation process is a vocabulary. A vocabulary is simply the set of unique tokens created from tokenisation. A challenge of tokenisation is

Example: “trouble”

```

t r o u b l e  C C V V C C V   (classify each letter as consonant/vowel)
                C V C V       (merge adjacent C's and V's)
                (C)(VC)(VC)(V)
                m = 2

```

Example: “trees”

```

t r e e s      C C V V C
                C V C
                (C)(VC)
                m = 1

```

Figure 4.25: Porter’s measure m counts the number of VC sequences after reducing a word to alternating consonant/vowel runs. Formally, a stem has the form $(C)(VC)^m(V)$, where C is a non-empty consonant sequence and V a non-empty vowel sequence.

out-of-vocabulary (OOV) words. These are words or sequences of text that are not able to be expressed with the tokens available in a vocabulary. The idea of tokenisation is to be able to express as much text as possible with as few tokens as possible, while minimising OOV words.

4.3.1 Byte-Pair Encoding (BPE)

Byte-Pair Encoding was originally developed by Philip Gage [36] as a form of data compression, but it now commonly used as a tokenisation method for NLP [56] [57] [58] [59]. The idea is to build a vocabulary based on how frequently tokens appear beside each other. It begins with the a vocabulary of all possible single byte values, and associates them with an integer.

```

for(int i = 0; i < 256; i++) {
    ByteSequence token = new ByteSequence(new byte[] {(byte) i});
    inverseVocab.put(token, i);
    vocab.put(i, token);
}

```

2 maps are kept for this, one where the key is the integer, and the value is the ByteSequence (Sec. 4.3.1.5), and another with the opposite. This helps with encoding and decoding later. Whenever a map is looked up for bytes, the bytes are first added to a ByteSequence object.

4.3.1.1 Training

The next step requires training data to be input from the developer. This comes in the form of a string, “corpus”, passed in when using the “train” function. Additionally, the developer also passes the desired vocabulary size (not including the 256 vocabulary entries created in the first step) with the “vocabSize” parameter.

```
public void train(String corpus, int vocabSize) {  
    ...  
    int[] tokenCorpus = encode(corpus);  
    ...  
}
```

Training begins by encoding (tokenising) the training corpus with the initial vocabulary. The text is converted into bytes, and each byte (as a `ByteSequence`) is searched for in the “inverseVocab” map for its integer representation. All of these integer representations (tokens) are stored in a integer array (in order) and returned to the training function (the encoding function is talked about in greater detail in Sec. 4.3.1.2).

Token pairs are then counted. A pair is simply 2 adjacent tokens. Pairs and their counts are stored in a map called “pairFreq”, where the key is the token pair (a record object) and the value is its frequency (integer). In a pair object, the “first” integer is the left token, and the “second” integer is the right token.

```
private final Map<Pair, Integer> pairFreq = new HashMap<>();  
...  
record Pair(int first, int second) {  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof Pair)) return false;  
        Pair other = (Pair) o;  
        return (this.first == other.first) && (this.second ==  
            ↪ other.second);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(first, second);  
    }  
}
```

Once all pair frequencies have been counted, the pair with the highest frequency is merged. Merging takes the bytes associated with both tokens in the pair, and adds them to a new ByteSequence. The lowest integer not currently being used by another token (i.e. the first merged token will be 256, the next 257, etc.) will be assigned to this new ByteSequence, and added to the vocabulary maps. The tokenised corpus is then updated to replace the old token pair with the new merged token. The old token pair is then removed from the “pairFreq” map, and pair frequencies for the new merged token are calculated and added to the map.

```
Pair best_pair = findBestPair(tokenCorpus); // Finds the most
    ↪ frequent pair

if(best_pair!=null) {
    int leftToken = best_pair.first(), rightToken =
        ↪ best_pair.second();
    int tokenId = addToken(leftToken, rightToken); // Adds the new
        ↪ merged pair token to the vocabulary

    // Update the token corpus with the new merged token.
    tokenCorpus = mergeTokens(tokenId, leftToken, rightToken,
        ↪ tokenCorpus);
    countPair(tokenId, leftToken, rightToken, tokenCorpus); //
        ↪ Counts token pairs for the new merged token
    count++; // Updates the current vocabulary size
}
```

This process is repeated until the desired vocabulary size has been reached, or there are no tokens pairs left to merge. A BPE object may only be trained once. Once training has been completed, or a token vocabulary has been loaded (Sec. 4.3.1.4), a “trained” boolean is set to true. When the “train” function is called again, this boolean is checked, and if it is true, retraining will not occur.

4.3.1.2 Encoding

Once BPE has been trained, it is possible to encode any string using the tokens created. The result of an encoding will be an integer array of tokens.

```
public int[] encode(String text)
```

This implementation attempts to shrink the text to as few tokens as possible by looking for the most possible bytes to include in a single token. Encoding begins by converting the text into bytes. The first byte (as a ByteSequence) is then searched

for in the vocabulary (“inverseVocab”). If the ByteSequence is in the vocabulary, it is saved as the “goodSequence”, and the next byte from the text is added to the ByteSequence and searched for in the map again. This continues until the ByteSequence is not in the vocabulary, and the token of the “goodSequence” is added to the token array to be returned.

```
while (end <= byteText.length) {
    byte[] slice = Arrays.copyOfRange(byteText, start, end);
    ByteSequence candidate = new ByteSequence(slice);

    if (inverseVocab.containsKey(candidate)) {
        goodSequence = candidate; // Update the best valid match
        end++;
    } else {
        break;
    }
}

// Store the best match found
encoding[tokenCounter++] = inverseVocab.get(goodSequence);

// Move start forward by the *length of the match*
start += goodSequence.bytes().length;
```

The next token is then searched for beginning with the byte after the last byte of the previously added token. This process continues until the full text has been converted into tokens, and returned the tokens to the developer.

4.3.1.3 Decoding

Decoding is the opposite process of encoding (Sec. 4.3.1.2). A string is extracted from an integer array of tokens.

```
public String decode(int[] tokens)
```

Beginning at index 0, each integer in the “tokens” array is search for in the vocabulary (“vocab”), and the bytes returned are added to a byte list. This list is then converted into a byte array that is used to create a string.

```
List<Byte> tokenBytes = new ArrayList<>();

for(int token: tokens) {
```

```
byte[] bytes = vocab.get(token).bytes();

for(byte b: bytes) {
    tokenBytes.add(b);
}

byte[] stringBytes = new byte[tokenBytes.size()];

for(int i = 0; i < stringBytes.length; i++) {
    stringBytes[i] = tokenBytes.get(i);
}

return new String(stringBytes,
    ↪ java.nio.charset.StandardCharsets.UTF_8);
```

The returned value is the decoded UTF-8 string.

4.3.1.4 Saving/Loading A Vocabulary

A vocabulary can be saved as a JSON file. The output is a JSON of a mapping of integer keys to either hex or character representations of the ByteSequences (Sec. 4.3.1.5). JSON requires keys to be strings, so the integers are warped in quotation marks. The format of the values depends on the function called:

```
public void saveVocabToJsonHex(String path)
public void saveVocabToJsonASCII(String path)
```

The location of the saved file will be the string that is passed. If no path is passed, the file is saved in the current working directory. An example of the output of each function can be viewed in Fig. 4.26 (hex) and Fig. 4.27 (ASCII).

Additionally, a hex JSON can be loaded into BPE to populate the vocabulary. This can only be done if BPE has not yet been trained. ASCII JSONs are only intended to be used for demonstration.

```
public void loadVocabFromJsonHex(String path)
```

4.3.1.5 ByteSequence

A ByteSequence is simply a record that stores an array of bytes. This was necessary for BPE (Sec. 4.3.1), as maps cannot work with primitive types in Java. Additionally, it contains an “equals” function for equality checks.


```
public record ByteSequence(byte[] bytes) {  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof ByteSequence)) return false;  
        ByteSequence other = (ByteSequence) o;  
        return Arrays.equals(this.bytes, other.bytes);  
    }  
  
    @Override  
    public int hashCode() {  
        return Arrays.hashCode(bytes);  
    }  
}
```

```
{  
    "224": "E0",  
    "225": "E1",  
    "223": "DF",  
    "226": "E2",  
    "...",  
    "279": "65 20 73",  
    "280": "65 20 74",  
    "258": "74 20",  
    "260": "64 20"  
}
```

Figure 4.26: Excerpt from a BPE hex JSON file.

4.3.2 N-gram

An n-gram is a small sub-section of a larger text. ‘N’ refers to the length of the sub-section. This can be n characters, or n words. Ngram.java handles the character level tokenisation, while Shingle.java (Sec. 4.3.3) is used for word level tokenisation.

This implementation of N-gram is simply a sliding window over a body of text provided to the developer. A substring of length n is taken as a token, the window slides over by 1 character and the process is repeated until the end of the text

```
{
  "224": "â",
  "225": "á",
  "317": "^&*",
  "223": "ß",
  "...",
  "269": "an",
  "276": "nes",
  "321": "lines\ta",
  "344": "new"
}
```

Figure 4.27: Excerpt from a BPE ASCII JSON file.

is reached. Each token is stored in order in a string array and returned to the developer.

```
public static String[] tokenise(String input, int window) {
    if(input == null || input.length() < window || window <= 0)
        ↪ return new String[0];

    String[] tokens = new String[input.length() - window + 1];

    for (int i = 0; i <= input.length() - window; i++) {
        tokens[i] = input.substring(i, i + window);
    }

    return tokens;
}
```

The window size (‘n’) is controlled by the developer with the “window” parameter.

4.3.3 Shingle

Shingle is simply another term used for n-gram. This term has been used here to differentiate between character n-grams and word n-grams. Ngram.java (Sec. 4.3.2) is used for character n-grams, while Shingle.java is used for word n-grams. A requirement of Shingle is that the text can be split by spaces, meaning languages such as Chinese and Japanese cannot be tokenised.

This implementation of Shingle simply slides a window over a body of text provided by the developer. The text is split by white space, producing a string array. The array is then looped over, and beginning at index 0, the next ‘N’ elements are joined together to create a new string (a token). The windows then slides over by 1 position, and the process is repeated until the end of the array is reached. Each token is stored in order and returned to the developer in a string array.

```
public static String[] tokenise(String input, int window) {
    if(input == null || input.length() <= window || window <= 0)
        ↪ return new String[0];

    String[] words = input.strip().split("\\s+"); // Remove
        ↪ leading/trailing whitespace before splitting

    if(words.length < window) return new String[0];

    String[] tokens = new String[words.length - window + 1];

    for (int i = 0; i <= words.length - window; i++) {
        tokens[i] = String.join(" ", Arrays.copyOfRange(words, i, i
            ↪ + window));
    }

    return tokens;
}
```

The windows size (‘n’) is controlled by the parameter “window”.

4.4 Vectorisers

In NLP, vectors are numerical representations of text (sentences, paragraphs, documents, etc.). In the TextForge library, a vectoriser is a method of converting strings to vectors.

4.4.1 Bag-Of-Words

A Bag-Of-Words (BoW) [43] is a very simple text representation model where a text is represented as an unordered collection (“bag”) of its words. BoW ignores grammar, word order, and syntax but takes note of a words count. While called

Bag-Of-Words, it very easily other forms of text tokens, such as n-grams (Sec. 4.3.2).

At its core, BoW is simply a map of words and counts. Before a word gets added to the bag, it is “cleaned” (at the moment, this simply converts all characters to lower case).

```
private final Map<String, Integer> bag = new TreeMap<String,
    ↪ Integer>();
...
public void add(String word) {
    if(word == null || word.isEmpty()) return;
    String cleanedWord = cleanWord(word);
    bag.put(cleanedWord, bag.computeIfAbsent(cleanedWord, k -> 0) +
    ↪ 1);
}
```

A developer using BoW can also add words in bulk, either with an array of strings or by using the “addSentence” function that will split the provided text on any white space.

A BoW can be queried for its vocabulary (the set of unique words in the bag), the count of a specific word. the number of unique words in the bag, the total number of words in the bag, or simply provide the developer with the underlying map.

```
public Set<String> vocabulary()
public int getCount(String word)
public int size()
public long totalWordCount()
public Map<String, Integer> getBag()
```

For the vectorisation method provided in BoW, the bag acts as a corpus. All words that have been added to the bag, will receive their own dimension in the created vector when calling the “vectorise” function. The developer passes in text in the form of an array of strings (as BoW can also act as a bag-of-ngrams (BoN), etc., it is required that the developer splits the text appropriately before calling the “vectorise” function).

```
public int[] vectorise(String[] text) {
    ...
    // First step is to count how often each word appears in the
    ↪ text array.
    for(String word: text) {
```

```

        String cleanedWord = cleanWord(word);
        counts.put(cleanedWord, counts.computeIfAbsent(cleanedWord,
            ↪ k -> 0) + 1);
    }
    ...
    // For every word in the bag (corpus), find out how often it
    ↪ appears in the text array, and build the vector.
    for(int i = 0; i < vocab.size(); i++) {
        vector[i] = counts.getDefault(vocab.get(i), 0);
    }

    return vector;
}

```

Vectorisation begins by counting each unique word in the text array, and storing it in a map. Then, each word in the corpus is searched for the text map. If the word appears in the text map, the count returned is entered in that words index in vector, otherwise a 0 will be put in that position.

Alternatively, BoW can be used to create count vectors. An example of this can be seen in Cosine distance (Sec. 4.1.2.1) where map representation of text (words and their counts) can be passed into the function and 2 equally sized count vectors are created. Count vector creation in TextForge is only available within certain Alignment-Free classes (Sec. 4.1.2). Count vector creation outside of these classes is not provided.

4.4.2 TF-IDF

Term Frequency - Inverse Document Frequency (TF-IDF) [24] is a vectorisation method what attempts to preserve the importance of a word to a document. It combines 2 separate concepts (Term Frequency (TF) [22] and Inverse Document Frequency (IDF) [23]) to create a vector that reduces the importance (i.e. give a smaller number to) terms or words that appear in many documents, while increasing the importance of a term that appears many times in a single document. TF is simply the frequency of a given term divided by the total number of terms (Fig. 4.28). IDF is essentially the log of the total number of documents divided by the number of documents a term appears in (Fig. 4.29). TF and IDF are then multiplied (Fig. 4.30) to give a final TF-IDF score to a term.

This implementation allows a developer to add a single document or a group of documents to a TFIDF object. A document is expected to be a string of text that is contains white space between words (terms). Each term is added to a corpus map that tracks its overall frequency (i.e. the number of documents in which

the term has appeared), and the number of documents that have been added is recorded.

```
private final Map<String, Integer> corpusTermFreq = new
    ↪ HashMap<>(); // Map of words to frequency count across all
    ↪ documents.
private int documentCount = 0;
...
public void addDocument(String doc)
public void addDocuments(String[] docs)
```

Once all corpus documents have been added, it is possible to vectorise a document. This process creates a vector of a size equal to the total number of unique terms in the corpus. The document is split into its terms, and then the TF-IDF formula (Fig. 4.30) is applied to each term in the corpus. It is important that the document being vectorised is already in the corpus, as such a boolean parameter (“newDocument”) can be set when calling “vectorise” that when set to true will add the document to the corpus before it is vectorised.

```
public double[] vectoriseDocument(String doc, boolean newDocument)
    ↪ {
    ...
    if(newDocument) {
        documentCount++;
        addTermsToDocumentFrequency(addedTerms); // Add terms to
        ↪ overall document frequency (if it hasn't previously
        ↪ been added). Terms should only be incremented once per
        ↪ document (i.e. even if a word appears many times in the
        ↪ document, it should only increase the corpus count by
        ↪ 1).
    }

    // To ensure the same order each time a vector is created
    ↪ (assuming no new terms added)
    List<String> vocab = new ArrayList<>(corpusTermFreq.keySet());
    Collections.sort(vocab);

    double[] vector = new double[corpusTermFreq.size()];

    // Begin calculations
    for(int i = 0; i < vocab.size(); i++) {
```

```

String term = vocab.get(i);
double tf = (double) termCounts.getOrDefault(term, 0) /
    ↪ len;
double idf = Math.log((1.0d + documentCount) / (1.0d +
    ↪ corpusTermFreq.get(term))); // This idf calculation
    ↪ includes smoothing (+ 1.0d) to ensure no division by 0
    ↪ errors.

vector[i] = (tf * idf);
}

return vector;
}

```

The returned result is a double vector representation of the document. Each dimension of the vector is the a term in the corpus, in alphabetical order.

Another similar function called “scoreDocument” is provided. This function returns a map of terms in the document and their TF-IDF score. This differs from “vectoriseDocument” in that only terms in the specified document are included in the map. Finally, functions to return the corpus term frequency map and the overall document count are available.

```

public Map<String, Double> scoreDocument(String doc, boolean
    ↪ newDocument)
public Map<String, Integer> getCorpusTermFreq()
public int getDocumentCount()

```

Potential improvements to TF-IDF include a document being an array of strings (terms) instead of a single string. This would allow other forms of tokens such as n-grams to be considered as terms.

$$\text{TF}_{t,d} = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

Figure 4.28: Term frequency of term t in document d , where $f_{t,d}$ is the raw count of t in d and the denominator sums counts of all terms in d . [22]

$$\text{IDF}_t = \log \frac{N}{1 + n_t}$$

Figure 4.29: Inverse document frequency of term t , where N is the total number of documents in the corpus and n_t is the number of documents containing term t . [23]

$$\text{TF-IDF}_{t,d} = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \times \log \frac{N}{1 + n_t}$$

Figure 4.30: TF-IDF weight of term t in document d , combining term frequency (TF) and inverse document frequency (IDF). [24]

4.5 Tool Box

The Tool Box is simply a collection of classes (Tools) that were created for use in other classes, but don't really fall within any of the other categories of the TextForge library.

4.5.1 Matrix Loader

How does Matrix Loader work? What is it used for? Where to get matrices?

Matrix loader is used by NeedlemanWunsch.java (Sec. 4.1.1.4) and SmithWaterman.java (Sec. 4.1.1.5) to load scoring matrices (e.g. PAM, BLOSUM, etc.) into memory for use in alignment. It is expected that the file will be similar to Tab. 4.5. For reference, the files this class was developed using can be found here: <https://ftp.ncbi.nih.gov/blast/matrices/>.

```
public static Map<String, Integer> load(String matrix_path) throws
    IOException
```

The load function is used to read a matrix from a text file and convert it to a map. It reads the text file line by line, ignore comment lines. A map key will be the column character and the row character (e.g. "AN", "RE", "YP", etc) and the value will be the cell score. Column characters appear on the first line after the comments, while the row character is the first character of every subsequent line.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	4	-1	0	0	-3	1	0	0	-2	0	-1	0	1	-2	-1	1	1	-5	-4	1	0	0	0	-7
R	-1	8	-2	-1	-2	3	-1	-2	-1	-3	-2	1	0	-1	-1	-1	-3	0	0	-1	-2	0	-1	-7
N	0	-2	8	1	-1	-1	-1	0	-1	0	-2	0	0	-1	-3	0	1	-7	-4	-2	4	-1	0	-7
D	0	-1	1	9	-3	-1	1	-1	-2	-4	-1	0	-3	-5	-1	0	-1	-4	-1	-2	5	0	-1	-7
C	-3	-2	-1	-3	17	-2	1	-4	-5	-2	0	-3	-2	-3	-3	-2	-2	-2	-6	-2	-2	0	-2	-7
Q	1	3	-1	-1	-2	8	2	-2	0	-2	-2	0	-1	-3	0	-1	0	-1	-1	-3	-1	4	0	-7
E	0	-1	-1	1	1	2	6	-2	0	-3	-1	2	-1	-4	1	0	-2	-1	-2	-3	0	5	-1	-7
G	0	-2	0	-1	-4	-2	-2	8	-3	-1	-2	-1	-2	-3	-1	0	-2	1	-3	-3	0	-2	-1	-7
H	-2	-1	-1	-2	-5	0	0	-3	14	-2	-1	-2	2	-3	1	-1	-2	-5	0	-3	-2	0	-1	-7
I	0	-3	0	-4	-2	-2	-3	-1	-2	6	2	-2	1	0	-3	-1	0	-3	-1	4	-2	-3	0	-7
L	-1	-2	-2	-1	0	-2	-1	-2	-1	2	4	-2	2	2	-3	-2	0	-2	3	1	-1	-1	0	-7
K	0	1	0	0	-3	0	2	-1	-2	-2	-2	4	2	-1	1	0	-1	-2	-1	-2	0	1	0	-7
M	1	0	0	-3	-2	-1	-1	-2	2	1	2	2	6	-2	-4	-2	0	-3	-1	0	-2	-1	0	-7
F	-2	-1	-1	-5	-3	-3	-4	-3	-3	0	2	-1	-2	10	-4	-1	-2	1	3	1	-3	-4	-1	-7
P	-1	-1	-3	-1	-3	0	1	-1	1	-3	-3	1	-4	-4	11	-1	0	-3	-2	-4	-2	0	-1	-7
S	1	-1	0	0	-2	-1	0	0	-1	-1	-2	0	-2	-1	-1	4	2	-3	-2	-1	0	-1	0	-7
T	1	-3	1	-1	-2	0	-2	-2	-2	0	0	-1	0	-2	0	2	5	-5	-1	1	0	-1	0	-7
W	-5	0	-7	-4	-2	-1	-1	1	-5	-3	-2	-2	-3	1	-3	-3	-5	20	5	-3	-5	-1	-2	-7
Y	-4	0	-4	-1	-6	-1	-2	-3	0	-1	3	-1	-1	3	-2	-2	-1	5	9	1	-3	-2	-1	-7
V	1	-1	-2	-2	-2	-3	-3	-3	-3	4	1	-2	0	1	-4	-1	1	-3	1	5	-2	-3	0	-7
B	0	-2	4	5	-2	-1	0	0	-2	-2	-1	0	-2	-3	-2	0	0	-5	-3	-2	5	0	-1	-7
Z	0	0	-1	0	0	4	5	-2	0	-3	-1	1	-1	-4	0	-1	-1	-1	-2	-3	0	4	0	-7
X	0	-1	0	-1	-2	0	-1	-1	-1	0	0	0	0	-1	-1	0	0	-2	-1	0	-1	0	-1	-7
*	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	1

Table 4.5: BLOSUM30 substitution matrix [25] with bolded self-match scores on the diagonal.

4.5.2 Scrubber

The Scrubber is a very simple class for normalising text. At the moment, it simply applies a regex pattern to remove all non-alphabetic character from a string and then converts it to lower case.

```
public static String scrub(String dirty) {
    return dirty.replaceAll("[^a-zA-Z ]", "").toLowerCase();
}
```

4.5.3 String Exporter

String Exporter simply writes a string to a text file. If a string array is to be written, each element of the array will be joined with a space as a separator. A path to a directory can be given to specify the location to save the file, otherwise the current working directory will be used. The file is always saved as “Output.txt”, and will overwrite this file if it already exists.

```
public static void toFile(String output, String path) throws
    ↳ IOException
public static void toFile(String[] output, String path) throws
    ↳ IOException
```

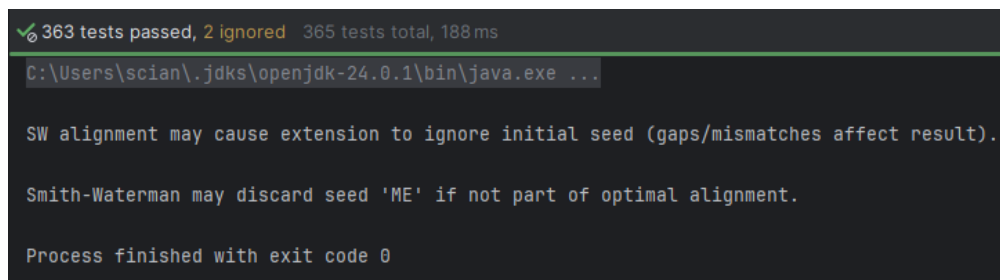
Chapter 5

System Evaluation

In this section, TextForge will be compared to the libraries researched in the Literature Review (Chap. 3). As the secondary goal of TextForge is to provide algorithms that may not be available within other libraries, many of the methods in TextForge cannot be directly compared. An explanation of how classes were tested will be discussed first, followed by a description of the algorithms available in TextForge and their presence in other libraries, and finally a short performance comparison will be conducted. Due to time constraints, this section is not as comprehensive as initially planned.

5.1 Testing

The creation of JUnit tests has been discussed in Sec. 2.3.1. At the moment, all tests are passing except for 2 (Fig. 5.1). The failing tests are currently disabled and are located in SeedAndExtendTest.java. Both are related to the issues discussed in SeedAndExtend (Sec. 4.1.1.6) regarding the use of Smith-Waterman to extend a seed.



```
✓ 363 tests passed, 2 ignored 365 tests total, 188 ms
C:\Users\scian\.jdk\openjdk-24.0.1\bin\java.exe ...

SW alignment may cause extension to ignore initial seed (gaps/mismatches affect result).

Smith-Waterman may discard seed 'ME' if not part of optimal alignment.

Process finished with exit code 0
```

Figure 5.1: Test Results.

5.2 Comparisons

In this section the TextForge library will be compared and contrasted to the libraries discussed in the literature review (Chap. 3.5).

5.2.1 Available Algorithms

A detailed list of the available algorithms available in the TextForge library, and their availability in other NLP libraries will be reviewed here.

5.2.1.1 Tokenisers

The TextForge library provides classes for several tokenisation methods. BPE (Sec. 4.3.1) is included in TextForge, with the ability to create tokens from a training corpus, load and save tokens from and to a JSON file, and to tokenise strings based on a token vocabulary. BPE is not available in any of Smile [49], OpenNLP [52], nor CoreNLP [51] libraries.

N-gram (Sec. 4.3.2) tokenisation splits text into overlapping segments of characters, with each segment being of an equal length specified by the developer. TextForge provides this functionality directly to the developer, with the ability to pass a string and a token length to `Ngram.tokenise()`. Character-level n-gram tokenisation is not provided by any of SmileNLP, OpenNlp, nor CoreNLP.

Word level n-gram tokenisation, known as Shingle tokenisation (Sec. 4.3.3 in the TextForge library, is used to split documents (or paragraphs, sentences, etc.) into overlapping substrings of words. These substrings each contain an equal number of words that is controlled by the developer. Word level n-gram tokenisation is not directly provided by any of SmileNLP, OpenNLP, nor CoreNLP.

All of SmileNLP, OpenNLP, and CoreNLP opt for different tokenisation methods, with OpenNLP and CoreNLP having implementations of Penn-Treebank tokenisation [50].

5.2.1.2 Stemmers

TextForge offers variety in its stemming algorithms. The Porter stemmer (Sec. 4.2.3) applies a set of steps (rules) to text to remove suffixes. This stemmer is available in TextForge, and also SmileNLP. OpenNLP and CoreNLP do not provide any stemming algorithms to developers.

The Lancaster stemmer (Sec. 4.2.1) iteratively applies rules to the end of words to reduce them to their root. This stemmer is available in both TextForge and SmileNLP.

Finally, the Lovins stemmer (Sec. 4.2.2) looks to match the ending of a word to a list of endings and their replacements. This action occurs only once. The Lovins stemmer is unique to TextForge amongst the NLP discussed in this document.

5.2.1.3 Similarity Measures

Similarity measures, both alignment based (Sec. 4.1.1) and alignment free (Sec. 4.1.2), are not part of either OpenNLP nor CoreNLP. TextForge and SmileNLP (within the “math” package) share a variety distance measures that they implement. The majority of distance measures provided by SmileNLP are alignment free measures, with the shared one with TextForge being: Chebyshev, Euclidean, Jaccard, Manhattan, and Minkowski. TextForge also provides Cosine, Canberra, Sørensen–Dice, Overlap, and Tversky similarity measures.

For alignment-based measures, SmileNLP and TextForge both have implementations of Levenshtein, Damerau-Levenshtein (Edit Distance), and Hamming distances, while TextForge also provides Jaro and Jaro-Winkler for similarity measurement, and Smith-Waterman, Needleman-Wunsch, and Seed-And-Extend algorithms for aligning strings.

The mentioned similarity (or distance) measures for the SmileNLP library are not the extend of its offerings. Other measures it provides include: Correlation distance [60], Jensen-Shannon distance [61], and Lee distance [62]. None of these distance measures are currently available in TextForge.

5.2.1.4 Vectorisers

Vectorisation methods (Sec. 4.4) convert strings to numerical representations. There are no available methods to do this in OpenNLP nor CoreNLP. Both TextForge and SmileNLP provide Bag-Of-Words (Sec. 4.4.1) and TF-IDF (Sec. 4.4.2) implementation to developers, while SmileNLP also allows pre-trained embeddings (e.g. Word2Vec [41], GloVe [63]) to be loaded and used.

5.2.1.5 Unavailable

There are many features of other NLP libraries that are not available in TextForge. OpenNLP and CoreNLP forgo stemming entirely in favour of lemmatisation (replacing a word with its stem by looking up the word in a dictionary). Part-of-Speech (PoS) Tagging [50] gives a “tag” to a word based on what type of word it is (e.g. noun, verb, adverb, etc.), and is implemented in each of SmileNLP, OpenNLP, and CoreNLP. As stated in the introduction, each of these libraries include some form of a machine learning model, from Name Entity Recognisers (NER) [64] in OpenNLP and CoreNLP, to Neural Networks (NN) and Random Forests (RF) in

Smile. This is a non-exhaustive list of some of the features available in the other libraries.

Library	Token	Stem	Sim	Vector	Other
TextForge	✓	✓	✓	✓	None
SmileNLP	✓	✓	✓	✓	ML models (NN, RF); Pretrained embeddings
OpenNLP	✓	✗	✗	✗	PoS tagging, NER
CoreNLP	✓	✗	✗	✗	PoS tagging, NER

Table 5.1: Availability of NLP features across libraries

5.3 Accuracy

In this section, the results of the methods in TextForge are compared to their counter-parts in the other NLP libraries, if they exist.

5.3.1 Tokenisers

As none of the other NLP libraries have the same tokenisation methods as TextForge, a comparison cannot be done.

5.3.2 Stemmers

Smile NLP shares two stemming algorithms with TextForge: Lancaster and Porter. The implementations from both libraries are given the same 230 words to stem, and the results are then compared. Any mismatches will be discussed. A sample of the words stemmed can be seen here:

```
String[] words = {
    "running", "runner", "runs", "ran", "easily",
    ↪ "easier", "easiest", "studies", "studying",
    ↪ "studied",
    ...
    "hopeful", "hopefully", "hopeless", "hopelessly",
    ↪ "unhopeful", "useful", "usefully", "useless",
    ↪ "uselessly", "unuseful"
};
```

5.3.2.1 Lancaster

When comparing Lancaster stemmers (Sec. 4.2.1), there were 4 mismatches:

Original Word	TextForge	Smile
ate	at	ate
careless	careles	careless
hopeless	hopeles	hopeless
useless	useles	useless

Table 5.2: Comparison of Lancaster stemming results between TextForge and Smile

In the case of “ate”, it would be expected to remove the ‘e’, due to the rule: “e1>” (if the word ends with ‘e’, remove one letter from the end, and continue the algorithm). Then, as stated in the original paper [40], if the word begins with a vowel, there must be at least two characters in the stem, thus stemming stops with “at”. It is unclear why Smile does not stem the final ‘e’.

In the cases of “careless”, “hopeless”, and “useless”, it seems the issue lies in the mistake discussed in Sec. 4.2.1, where rules are not applied in the expected order. As the TextForge implementation looks to apply intact rules first, the rule “s1*>” gets applied, removing the final ‘s’ of the words. However, the rule that should be applied is “ss0.”, which prevents the final ‘s’ from being removed and concludes the algorithm.

5.3.2.2 Porter

When comparing the results of stemming using the TextForge and Smile implementations, the results for the 230 words were identical.

5.3.2.3 Lovins

As none of the other libraries contain a Lovins stemmer implementation, a comparison cannot be done.

5.3.3 Similarity measures

TextForge and Smile share both alignment based and alignment free similarity measures. However, Smile’s similarity measures are in its “math” package, so that will be used for this comparison.

5.3.3.1 Alignment Based

Levenshtein and Damerau-Levenshtein distances were first compared using two strings of differing lengths. The same strings were passed into each implementation from each library.

```
String s1 = "cats and dogs and also more";
String s2 = "dogs and cats and nothing else";
```

The results of both libraries were the same.

The implementation of Hamming distance in Smile was slightly different. It does not accept characters nor strings, so integer arrays needed to be created before HammingDistance.d could be called.

```
String s1 = "cats";
String s2 = "dogs";

int[] i1 = s1.chars().toArray();
int[] i2 = s2.chars().toArray();
```

In this case, s1 and s2 were compared, and s1 was compared to itself. The results of both TextForge's and Smile's implementations were identical.

5.3.3.2 Alignment Free

First, Chebyshev, Euclidean, Manhattan, and Minkowski distances were compared. Vectors were needed, so they were created using TextForge's TF-IDF (Sec. 4.4.2):

```
String d1 = "This is a document to be added to the tfidf vector. It
→ should have a couple of repeating words.";
String d2 = "Another document to be added. The corpus should be
→ significant enough to produced decent results.";
String d3 = "After this there will be one final document.";
String d4 = "Finally, the last and final document has been reached.
→ It is time to add them all to tfidf.";
```

```
TFIDF tf = new TFIDF();
tf.addDocument(d1);
tf.addDocument(d2);
tf.addDocument(d3);
tf.addDocument(d4);

double[] v1 = tf.vectoriseDocument(d1, false);
double[] v2 = tf.vectoriseDocument(d2, false);
```

The results of both TextForge’s and Smile’s implementations of these algorithms were identical.

Jaccard similarity compares two sets, and thus had to be tested differently. As Smile provides a distance measure, the TextForge distance measure was also used. The sets used were created from strings (d1 and d2 from the previous code block) to emulate use in a NLP task.

```
Set<String> s1 = new HashSet<>(new  
    ↳ ArrayList<>(List.of(Scrubber.scrub(d1).split(" ")))));  
Set<String> s2 = new HashSet<>(new  
    ↳ ArrayList<>(List.of(Scrubber.scrub(d2).split(" ")))));
```

The results of both TextForge’s and Smile’s implementations of these algorithms were identical.

5.3.4 Vectorisers

5.3.4.1 Bag-Of-Words

Both TextForge and Smile have implementations of BoW (Smile’s is not located in the NLP package, but rather in the “core” package). Smile’s implementation requires a tokeniser and a string array during BoW creation, as such the tokeniser used (SimpleTokenizer) for this was also used for on strings before adding them to TextForge’s BoW to maintain consistency. The same word array from the stemming tests was used to populate the BoW implementations. Each BoW was then used to vectorise some test. As the positions of words in the created vectors may not be equal, equality was done by: ensuring the vectors were of equal sizes; ensuring the expected counts were both equal.

```
String s1 = "Here careful is careful a careful string kindly unkind  
    ↳ unkind";
```

The tested string only contains three unique words that were in the initial corpus. As such, all but three of the vector values should be zero. Additionally, as there are three “careful”s, two “unkind”s and a single “kindly” (the words in the corpus), we should see a vector value of one, another of two, and finally a value of three.

Both BoW vectors were of equal length, and both contained a single value of each of the expected values.

5.3.4.2 Term Frequency - Inverse Document Frequency

TF-IDF in TextForge and Smile are fundamentally different. TextForge’s implementation allows the developer to vectorise complete documents, while the goal

of Smile's implementation is to simply score individual terms. Having said this, a comparison can still be made, as TextForge also allows the scoring of each term in a document.

```
String[] docs = {  
    "Here is the first document.",  
    "And another document.",  
    "This is the final document."  
};
```

When scoring the term “document”, both implementations return a value of 0. This is expected, as the term appears in each document. When scoring the term “first”, the scores from each implementation differs. An explanation for this is how each handles smoothing. TextForge applies smoothing to prevent division by 0 errors within the IDF calculation.

```
// TextForge TF-IDF  
double tf = (double) termCounts.get(term) / len;  
double idf = Math.log((1.0d + documentCount) / (1.0d +  
    ↪ corpusTermFreq.get(term))) ); // Adding 1.0d to the denominator  
    ↪ here to prevent zeroing out of scores (log 1)  
termScores.put(term, (tf * idf));  
  
// Smile TF-IDF  
return (a + (1-a) * tf / maxtf) * Math.log((double) N / n); // 'a'  
    ↪ is a smoothing factor specified by the developer.
```

Smile applies smoothing to the TF calculation.

5.4 Performance

With regards to size, TextForge is significantly smaller than other libraries. The JAR for TextForge is 62KB (with the JDoc JAR included, this becomes 4.13MB), while Smile is 203MB (no JDoc, 4.1 MB with just core, base, and nlp JARs), OpenNLP is 5.62MB (no JDoc), and CoreNLP is 972MB (all JARs, including JDoc).

For specific algorithms, times taken to complete the task by each implementation will be used to compare performance.

5.4.1 Stemmers

Lancaster (Sec. 4.2.1) and Porter (Sec. 4.2.3) stemmers were chosen for performance comparison as both are available in a library (Smile) discussed in the Literature Review (Chap. 3). Each stemmer will stem the same words used in the accuracy comparison (Sec. 5.3.2) 10 times (2300 words total).

The times taken (in milliseconds) by each Lancaster and Porter stemmer to stem all words can be seen in Tab. 5.4.1. The results of this test indicate the performance of TextForge’s stemmers is relatively poor.

Stemmer	TextForge	Smile
Lancaster	22	9
Porter	25	2

Table 5.3: Comparison of stemmer times between TextForge and Smile (in milliseconds)

5.4.2 Byte-Pair Encoding

Although BPE is not available from any library discussed in the Literature Review, it was decided to test its performance anyway, as it is a large part of the TextForge library. A pure Java BPE implementation (from skyfalconai on github [65]) that allows for token training was found online to compare to TextForge’s implementation. As training is the most time consuming aspect of BPE, it is the focus of this test. Both implementations were provided a copy of George Orwell’s 1984 (10287 lines, 587,086 characters) to train on, and asked to create 2000 new tokens (merges). As the other implementation includes reading a file, TextForge’s time will also include the time taken to read the file (copying the code from the other implementation. For reference, the time taken for HuggingFace’s [66] BPE implementation (written in Rust [67], called from Python) is also included in the results table (Tab. 5.4.2).

	TextForge	SkyFalconAI	HuggingFace
Training Time (ms)	961	17094	364

Table 5.4: Comparison of BPE training times across different implementations

The results show that, while TextForge’s BPE implementation does not train as quickly as HuggingFace’s, it is more performative than other available Java implementations.

Chapter 6

Conclusion

This dissertation set out to investigate the creation of a Java library for text preprocessing in Natural Language Processing, explicitly avoiding the inclusion of machine learning models or advanced analytical tools such as named entity recognition and language detection, with the secondary goal of including niche or under-represented algorithms. By focusing exclusively on preprocessing, the library fills a gap for developers who require flexible, lightweight tools without the overhead of integrated ML frameworks.

The TextForge library successfully implemented four core preprocessing modules: tokenisation (Sec. 4.3), stemming (Sec. 4.2), similarity measures (Sec. 4.1), and vectorisation (Sec. 4.4). Within tokenisation, stemming, and similarity measures, at least one algorithm was implemented that is not available in existing Java NLP libraries such as Smile NLP [49], OpenNLP [52], or CoreNLP [51], thereby expanding the range of preprocessing options available to developers. Furthermore, TextForge was developed as a self-contained library without reliance on external Java dependencies, ensuring lightweight integration and broad usability across projects.

Despite its contributions, the project is not without limitations. The TextForge library currently supports a limited set of preprocessing techniques, omitting important methods such as lemmatisation [29] and stopword removal that are commonly used in NLP pipelines. In addition, while the library provides original implementations of stemming algorithms, their performance (Sec. 5.4) and accuracy (Sec. 5.3) do not consistently match the quality of established alternatives such as Smile NLP. Finally, the development of TextForge relied on Java preview features (Sec. 2.2.1.1), which may impact long-term stability and compatibility as the language evolves.

This dissertation has demonstrated that it is both feasible and valuable to develop a lightweight, self-contained Java library dedicated to text preprocessing for Natural Language Processing. By providing a range of core techniques, in-

cluding several not available in other widely used Java NLP libraries, TextForge offers developers a flexible and accessible tool that can be easily integrated into existing workflows. While there is scope for further refinement and expansion, the project establishes a solid foundation upon which future enhancements can be built. Ultimately, TextForge represents a meaningful step towards broadening the resources available for Java-based NLP, and it is hoped that this work will encourage continued exploration and innovation in this area.

Bibliography

- [1] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, Apr. 1950. [Online]. Available: <https://ieeexplore.ieee.org/document/6772729/>
- [2] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, Feb. 1966, aDS Bibcode: 1966SPhD...10..707L. [Online]. Available: <https://ui.adsabs.harvard.edu/abs/1966SPhD...10..707L>
- [3] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, Mar. 1964. [Online]. Available: <https://dl.acm.org/doi/10.1145/363958.363994>
- [4] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0022283670900574>
- [5] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0022283681900875>
- [6] M. A. Jaro, "Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida," *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989, publisher: [American Statistical Association, Taylor & Francis, Ltd.]. [Online]. Available: <https://www.jstor.org/stable/2289924>
- [7] W. E. Winkler, "String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage," Jan. 1990. [Online]. Available: https://www.researchgate.net/publication/243772975_String_Comparator_Metrics_and_Enhanced_Decision_Rules_in_the_Fellegi-Sunter_Model_of_Record_Linkage

- [8] J. Doshi, “Chatbot User Interface for Customer Relationship Management using NLP models,” in *2021 International Conference on Artificial Intelligence and Machine Vision (AIMV)*, Sep. 2021, pp. 1–4. [Online]. Available: <https://ieeexplore.ieee.org/document/9670914/>
- [9] M. D. Malkauthekar, “Analysis of euclidean distance and Manhattan Distance measure in face recognition,” in *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013)*, Oct. 2013, pp. 503–507. [Online]. Available: <https://ieeexplore.ieee.org/document/6950920/>
- [10] G. N. Lance and W. T. Williams, “Lance G N & Williams W T. A general theory of classificatory sorting strategies. 1. Hierarchical systems. Computer J. 9:373-80, 1967.” 1967. [Online]. Available: <https://academic.oup.com/comjnl/article-abstract/9/4/373/390278?redirectedFrom=fulltext>
- [11] E. O. Rodrigues, “Combining Minkowski and Chebyshev: New distance proposal and survey of distance metrics using k-nearest neighbours classifier,” *Pattern Recognition Letters*, vol. 110, pp. 66–71, Jul. 2018, arXiv:2112.12549 [cs]. [Online]. Available: <http://arxiv.org/abs/2112.12549>
- [12] H. H. Minkowski, *Geometrie der Zahlen*. Leipzig : Teubner, 1910. [Online]. Available: <http://archive.org/details/geometriederzahl00minkrich>
- [13] P. JACCARD, “Etude comparative de la distribution florale dans une portion des Alpes et des Jura,” *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901. [Online]. Available: <https://cir.nii.ac.jp/crid/1573387450552842240>
- [14] T. T. Tanimoto, *An Elementary Mathematical Theory of Classification and Prediction*. International Business Machines Corporation, 1958, google-Books-ID: yp34HAAACAAJ.
- [15] “GENERALIZED JACCARD COEFFICIENT, GENERALIZED JACCARD DISTANCE.” [Online]. Available: <https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/jaccard.htm>
- [16] L. d. F. Costa, “Further Generalizations of the Jaccard Index,” Nov. 2021, arXiv:2110.09619 [cs]. [Online]. Available: <http://arxiv.org/abs/2110.09619>
- [17] T. Sørensen, T. Sørensen, T. Biering-Sørensen, T. Sørensen, and J. T. Sorensen, “A method of establishing group of equal amplitude in plant sociobiology based on similarity of species content and its application to analyses of the vegetation on Danish commons,” 1948. [Online]. Available: <https://www.semanticscholar.org/paper/>

A-method-of-establishing-group-of-equal-amplitude-S%C3%B8rensen-S%C3%B8rensen/d8d3e6d95b60ec6ac8f91f42a6914a87b13a6bc1

- [18] L. R. Dice, “Measures of the Amount of Ecologic Association Between Species,” *Ecology*, vol. 26, no. 3, pp. 297–302, 1945, publisher: [Wiley, Ecological Society of America]. [Online]. Available: <https://www.jstor.org/stable/1932409>
- [19] D. Szymkiewicz, “Une contribution statistique à la géographie floristique,” *Acta Societatis Botanicorum Poloniae*, vol. 11, no. 3, pp. 249–265, 1934. [Online]. Available: <https://pbsociety.org.pl/journals/index.php/asbp/article/view/asbp.1934.012>
- [20] A. Tversky, “Features of similarity,” *Psychological Review*, vol. 84, no. 4, pp. 327–352, 1977, place: US Publisher: American Psychological Association.
- [21] J. B. Lovins, “Development of a stemming algorithm,” *Mechanical Translation and Computational Linguistics*, vol. 11, no. 1-2, pp. 22–31, 1968. [Online]. Available: <https://aclanthology.org/www.mt-archive.info/MT-1968-Lovins.pdf>
- [22] G. Salton, A. Wong, and C. S. Yang, “A vector space model for automatic indexing,” *Commun. ACM*, vol. 18, no. 11, pp. 613–620, Nov. 1975. [Online]. Available: <https://dl.acm.org/doi/10.1145/361219.361220>
- [23] K. SPARCK JONES, “A STATISTICAL INTERPRETATION OF TERM SPECIFICITY AND ITS APPLICATION IN RETRIEVAL,” *Journal of Documentation*, vol. 28, no. 1, pp. 11–21, Jan. 1972, publisher: MCB UP Ltd. [Online]. Available: <https://doi.org/10.1108/eb026526>
- [24] G. Salton and C. Buckley, “Term-weighting approaches in automatic text retrieval,” *Information Processing & Management*, vol. 24, no. 5, pp. 513–523, Jan. 1988. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0306457388900210>
- [25] “Blast Matrices.” [Online]. Available: <https://ftp.ncbi.nih.gov/blast/matrices/>
- [26] “IEEE Xplore.” [Online]. Available: <https://ieeexplore.ieee.org/Xplore/home.jsp>
- [27] “Google Scholar.” [Online]. Available: <https://scholar.google.com/>
- [28] “ACM Digital Library.” [Online]. Available: <https://dl.acm.org/>

- [29] G. A. Miller, “WordNet: a lexical database for English,” *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995. [Online]. Available: <https://dl.acm.org/doi/10.1145/219717.219748>
- [30] “Structured Concurrency,” publisher: April2025. [Online]. Available: <https://docs.oracle.com/en/java/javase/24/core/structured-concurrency.html>
- [31] “Record Classes,” publisher: January2025. [Online]. Available: <https://docs.oracle.com/en/java/javase/17/language/records.html>
- [32] R. Lourdusamy and S. Abraham, “A Survey on Text Pre-processing Techniques and Tools,” *International Journal of Computer Sciences and Engineering*, vol. 06, no. 03, pp. 148–157, Apr. 2018. [Online]. Available: http://www.ijcseonline.org/full_spl_paper_view.php?paper_id=337
- [33] L. Athota, V. K. Shukla, N. Pandey, and A. Rana, “Chatbot for Healthcare System Using Artificial Intelligence,” in *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, Jun. 2020, pp. 619–622. [Online]. Available: <https://ieeexplore.ieee.org/document/9197833/>
- [34] S. Warusawithana, N. Perera, R. Weerasinghe, T. Hindakaraldeniya, and G. U. Ganegoda, “Layout Aware Resume Parsing Using NLP and Rule-based Techniques,” in *2023 8th International Conference on Information Technology Research (ICITR)*, Dec. 2023, pp. 1–5, iSSN: 2831-3399. [Online]. Available: <https://ieeexplore.ieee.org/document/10382773/>
- [35] J. L. Gastaldi, J. Terilla, L. Malagutti, B. DuSell, T. Vieira, and R. Cotterell, “The Foundations of Tokenization: Statistical and Computational Concerns,” Apr. 2025, arXiv:2407.11606 [cs]. [Online]. Available: <http://arxiv.org/abs/2407.11606>
- [36] P. Gage, “A new algorithm for data compression,” *C Users J.*, vol. 12, no. 2, pp. 23–38, Feb. 1994. [Online]. Available: <https://arxiv.org/pdf/1209.1045>
- [37] W. B. Cavnar and J. M. Trenkle, “N-Gram-Based Text Categorization.” [Online]. Available: <https://dsac13-2019.github.io/materials/CavnarTrenkle.pdf>
- [38] T. Kudo, “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates,” Apr. 2018, arXiv:1804.10959 [cs]. [Online]. Available: <http://arxiv.org/abs/1804.10959>

- [39] M. Porter, “An algorithm for suffix stripping,” *Program*, vol. 14, no. 3, pp. 130–137, Jan. 1980, publisher: MCB UP Ltd. [Online]. Available: <https://doi.org/10.1108/eb046814>
- [40] C. D. Paice, “Another stemmer,” *SIGIR Forum*, vol. 24, no. 3, pp. 56–61, Nov. 1990. [Online]. Available: <https://dl.acm.org/doi/10.1145/101306.101310>
- [41] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” Sep. 2013, arXiv:1301.3781 [cs]. [Online]. Available: <http://arxiv.org/abs/1301.3781>
- [42] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” May 2019, arXiv:1810.04805 [cs] version: 2. [Online]. Available: <http://arxiv.org/abs/1810.04805>
- [43] Z. S. Harris, “Distributional Structure,” *WORD*, vol. 10, no. 2-3, pp. 146–162, Aug. 1954, publisher: Routledge _eprint: <https://doi.org/10.1080/00437956.1954.11659520>. [Online]. Available: <https://doi.org/10.1080/00437956.1954.11659520>
- [44] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool,” *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, Oct. 1990. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0022283605803602>
- [45] A. N. Sak, “Using Vector Proximity for NLP Analysis of Specialized Texts,” in *2022 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO)*, Jun. 2022, pp. 1–6, iSSN: 2832-0514. [Online]. Available: <https://ieeexplore.ieee.org/document/9840981/>
- [46] J. Mueller and A. Thyagarajan, “Siamese recurrent architectures for learning sentence similarity,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI’16. Phoenix, Arizona: AAAI Press, Feb. 2016, pp. 2786–2792.
- [47] S. Bereg, Z. Miller, and I. H. Sudborough, “Upper Bounds for Chebyshev Permutation Arrays,” *Entropy (Basel, Switzerland)*, vol. 27, no. 6, p. 558, May 2025.
- [48] “Smile - NLP.” [Online]. Available: <https://haifengl.github.io/nlp.html>
- [49] H. Li, “haifengl/smile,” Aug. 2025, original-date: 2014-11-20T16:28:12Z. [Online]. Available: <https://github.com/haifengl/smile>

- [50] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz, “Building a Large Annotated Corpus of English: The Penn Treebank,” *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993, place: Cambridge, MA Publisher: MIT Press. [Online]. Available: <https://aclanthology.org/J93-2004/>
- [51] “Stanford CoreNLP.” [Online]. Available: <https://stanfordnlp.github.io/CoreNLP/>
- [52] “Apache OpenNLP.” [Online]. Available: <https://opennlp.apache.org/>
- [53] A. Z. Broder, “Min-wise Independent Permutations: Theory and Practice,” in *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, ser. ICALP ’00. Berlin, Heidelberg: Springer-Verlag, Jul. 2000, p. 808.
- [54] W. Wu, B. Li, L. Chen, J. Gao, and C. Zhang, “A Review for Weighted MinHash Algorithms,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 6, pp. 2553–2573, Jun. 2022. [Online]. Available: <https://ieeexplore.ieee.org/document/9184977>
- [55] “MurmurHash3 (Apache Commons Codec 1.19.0 API).” [Online]. Available: https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/MurmurHash3.html?utm_source=chatgpt.com
- [56] M. Berglund and B. v. d. Merwe, “Formalizing BPE Tokenization,” *Electronic Proceedings in Theoretical Computer Science*, vol. 388, pp. 16–27, Sep. 2023, arXiv:2309.08715 [cs]. [Online]. Available: <http://arxiv.org/abs/2309.08715>
- [57] Z. Huang, “An Ensemble LLM Framework of Text Recognition Based on BERT and BPE Tokenization,” in *2024 5th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, Mar. 2024, pp. 1750–1754. [Online]. Available: <https://ieeexplore.ieee.org/document/10581466>
- [58] V. Zouhar, C. Meister, J. L. Gastaldi, L. Du, T. Vieira, M. Sachan, and R. Cotterell, “A Formal Perspective on Byte-Pair Encoding,” Sep. 2024, arXiv:2306.16837 [cs]. [Online]. Available: <http://arxiv.org/abs/2306.16837>
- [59] H. Lian, Y. Xiong, J. Niu, S. Mo, Z. Su, Z. Lin, H. Chen, P. Liu, J. Han, and G. Ding, “Scaffold-BPE: Enhancing Byte Pair Encoding for Large Language Models with Simple and Effective Scaffold Token Removal,” Nov. 2024, arXiv:2404.17808 [cs]. [Online]. Available: <http://arxiv.org/abs/2404.17808>

- [60] L. Hubert and P. Arabie, “Comparing partitions,” *Journal of Classification*, vol. 2, no. 1, pp. 193–218, Dec. 1985. [Online]. Available: <https://doi.org/10.1007/BF01908075>
- [61] J. Lin, “Divergence measures based on the Shannon entropy,” *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 145–151, Jan. 1991. [Online]. Available: <https://ieeexplore.ieee.org/document/61115/>
- [62] C. Lee, “Some properties of nonbinary error-correcting codes,” *IRE Transactions on Information Theory*, vol. 4, no. 2, pp. 77–82, Jun. 1958. [Online]. Available: <https://ieeexplore.ieee.org/document/1057446/>
- [63] J. Pennington, R. Socher, and C. Manning, “GloVe: Global Vectors for Word Representation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: <https://aclanthology.org/D14-1162/>
- [64] A. McCallum and W. Li, “Early results for Named Entity Recognition with Conditional Random Fields, Feature Induction and Web-Enhanced Lexicons,” in *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, 2003, pp. 188–191. [Online]. Available: <https://aclanthology.org/W03-0430/>
- [65] S. Falcon, “skyfalconai/BPE-Byte-Pair-Encoding-Java-,” Jan. 2025, original-date: 2025-01-13T18:16:57Z. [Online]. Available: <https://github.com/skyfalconai/BPE-Byte-Pair-Encoding-Java->
- [66] “Hugging Face – The AI community building the future.” [Online]. Available: <https://huggingface.co/>
- [67] A. Moi and N. Patry, “HuggingFace’s Tokenizers,” Apr. 2023, original-date: 2019-11-01T17:52:20Z. [Online]. Available: <https://github.com/huggingface/tokenizers>