VRIJE
UNIVERSITEIT
BRUSSEL

SCALABLE ANALYTICS PROJECT

Project Report

Dana Tabatabaie Irani dana.tabatabaie.irani@vub.be
Faroukh Davouzov faroukh.davouzov@vub.be
Youri Langhendries youri.langhendries@vub.be

Monday 3rd June, 2024

Sciences and Bio-Engineering Sciences

Contents

1 Introduction

The increasing adoption of generative models in education [1, 2] has resulted in a surge in text obfuscation, making the challenge of detecting plagiarism more critical than ever. Plagiarism detection, the process of identifying instances where content is copied without proper attribution, is crucial in academic, professional, and creative fields. Traditional plagiarism detection methods often involve manual reviews or simple string-matching algorithms, which can be time-consuming and inefficient for large datasets. Locality Sensitive Hashing (LSH) offers a promising alternative by enabling efficient similarity searches in high-dimensional data spaces, potentially transforming the domain of plagiarism detection.

Despite LSH's potential advantages, such as lower memory requirements and faster searches [8, 10], the impact of preprocessing and hyperparameter tuning on the detection of near-duplicate documents remains underexplored, especially in the context of large datasets. Additionally, the optimal configurations for LSH, such as shingling approaches and preprocessing techniques, require systematic evaluation to maximise efficiency and accuracy.

1.1 Research Questions and Objectives

Our research aims to answer the following questions (RQ) regarding using Locality Sensitive Hashing for detecting near-duplicates (plagiarism) in a large dataset of text documents. For each research question, we have formulated a hypothesis (H) based on our investigation and expectations:

- **RQ1**: How effective is LSH in detecting plagiarism within a large dataset of text documents?
  H1: LSH is highly effective in detecting plagiarism within large datasets of text documents due to its efficient similarity searches.
- **RQ2**: Which shingling approach is most suitable for detecting plagiarism?
  H2: We hypothesise that character-based shingling approaches are more suitable due to their ability to capture general patterns.
- **RQ3**: Which benefits does preprocessing offer when using LSH for plagiarism detection?
  H3: We hypothesise that preprocessed documents significantly improve the accuracy and efficiency of plagiarism detection using LSH by reducing noise and document size.
- **RQ4**: How does the performance of an LSH tool change when only a subset of the keywords in sentences is considered?
  H4: We anticipate that focusing on a subset of keywords within sentences will lead to a trade-off between performance and efficiency in LSH tools. However, we hypothesise that this trade-off will be justified by the gains in computational efficiency, making it a worthwhile approach.

The first Research Question (RQ1) focuses on the applicability of LSH for plagiarism detection tools. It seeks to determine LSH's accuracy and efficiency when applied to large-scale datasets, exploring its potential as a robust solution for identifying similar text segments.

The second Research Question (RQ2) aims to find whether character-based shingling is more suitable for finding plagiarism than word-based shingling. This question addresses the comparative effectiveness of different shingling approaches in capturing textual similarities that indicate

plagiarism.

Research Question 3 (RQ3) investigates the impact of preprocessing on LSH's performance for plagiarism detection. It aims to determine how preprocessing steps, such as stop-word removal, enhance or hinder LSH's accuracy and efficiency.

The last Research Question 4 (RQ4) attempts to find how an LSH tool's performance is affected when only a subset of keywords within sentences is considered. Understanding this allows us to balance computational efficiency with detection accuracy.

To address these research questions, we developed a custom LSH-based tool tailored for plagiarism detection. Our methodology involved creating various shingling configurations, applying different preprocessing techniques, and conducting extensive experiments on a large dataset of text documents. Preprocessing steps included removing stop-words, paraphrasing specific words within sentences, and selecting a subset of keywords for analysis. This step allows us to create consistency between all documents. Performance metrics such as F1-score and computational efficiency were used to evaluate the effectiveness of the LSH tool under different conditions to balance the recall and precision of our model.

This study aims to advance the field of plagiarism detection by providing empirical insights into the effectiveness of LSH. By identifying optimal shingling approaches and preprocessing techniques, our research seeks to improve the accuracy and efficiency of plagiarism detection tools. This improvement can have significant implications for academic integrity, intellectual property protection, and the broader field of text analysis.

1.2 Report Outline

The structure of this study is delineated into several sections. Section 2 elaborates on acquiring interesting datasets. Section 3 details the steps to prepare the dataset, encompassing dataset extraction, text file processing, and paraphrasing. Section 4 discusses the LSH technique used for plagiarism detection, covering shingling, MinHashing, the LSH algorithm, hashing techniques, and optimisations introduced to speed up the algorithm. After this, we discuss the process of assessing the performance of the LSH tool, including hyperparameter tuning and evaluation metrics in Section 5. Section 6 addresses the tool's scalability, encompassing discussions on multithreading, MapReduce, and Spark. Future work prospects are outlined in the penultimate Section 7, followed by the conclusion in Section 8, summarising the study's findings and offering closing remarks.

2 Data Collection

The initial phase in the development of our tool involves identifying appropriate datasets that meet the following essential criteria:

1. The dataset must comprise two types of documents:
    - A set of original documents (1) may include text files, code files, or any other file format.
    - A corresponding set of suspicious documents (2) that may or may not contain plagiarised content derived from the documents in the first set. These documents are typically sourced from pools of submissions, such as student assignments.
2. A metadata document, also called a ground truth document, is required. This document provides detailed information about which documents from Set (2) contain plagiarised content from Set (1). The presence of this document is crucial for assessing the accuracy of our tool. It enables establishing an objective target function to optimise our tool's performance.

It is also possible to construct a dataset from scratch. However, we preferred to simulate the real-world use-case of actual data usage, so we searched for a dataset based on these criteria. We have found one dataset [6] existing of Java code-files, one dataset [4] containing English Wikipedia articles and finally, a dataset [9] containing regular documents.

2.1 Wikipedia Dataset

The Wikipedia dataset is accessible at: `https://deepblue.lib.umich.edu/data/concern/data_sets/2801pg45f?locale=en`. This dataset comprises multiple folders, one of which includes various machine learning classifier models developed by Foltynek et al. [5]. The primary data of interest are contained within the "corpus" zip folder, which stores two types of documents: "wikipedia_documents_test" and "wikipedia_paragraphs_test". The former includes complete pairs of original and plagiarised documents, whereas the latter consists of individual paragraphs from these documents, each paired as original and plagiarised. Our study focuses only on the complete document zip to ensure it applies in real-world situations. This zip file separates documents into two distinct folders, as illustrated in Figure 1: the "og" folder, which contains the unaltered original files, and the "mg" folder, which comprises documents that have been paraphrased or plagiarised. Each original document, identified by a number, is matched with a corresponding plagiarised version bearing the same number, for example, (666-ORIG, 666-SPUN). The plagiarised versions are created by modifying the original texts using the online tool "Spinbot"[1]. The folder naming scheme, however, was slightly modified by us: the folders have been renamed from "og" to "a_og" and from "mg" to "b_mg" to ensure that all original files are processed first in the tool, followed by the plagiarised ones.

[1] `https://spinbot.com/`

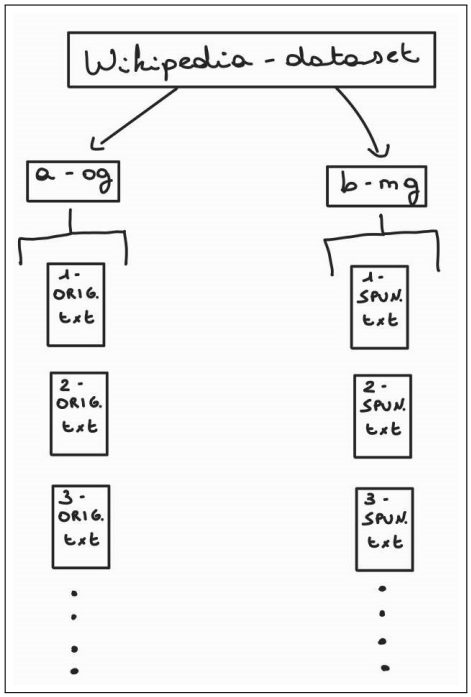

Figure 1: Hierarchy of the Wikipedia Dataset.

2.2 PAN Plagiarism Corpus Dataset

The PAN Plagiarism Corpus 2011 is accessible at: `https://zenodo.org/records/3250095`. This dataset comprises two .rar files, each constituting a part of the complete archive. Both files must be downloaded to enable the complete extraction of the dataset. The combined outcome of unzipping these files results in two primary folders: "external-detection-corpus" and "intrinsic-detection-corpus", as shown in Figure 2. The former includes two subfolders named "source-document" and "suspicious-document". The "source-document" folder contains original documents, whereas the "suspicious-document" folder includes documents that may incorporate (parts of) text from the source documents. Each folder is accompanied by a corresponding XML file that provides metadata about its contents, such as the authors and title. This metadata also includes the source of the potentially plagiarised text for suspicious documents. Unlike the Wikipedia dataset, the numerical identifiers of source documents do not match those of the suspicious documents. Consequently, if only a subset of documents from both folders is utilised for testing, it may not be possible to include any fraud pairs beyond a certain file cutoff point. This absence of usable file cutoff points reduces the utility of this dataset for benchmarking purposes, as it requires processing the entire dataset. Therefore, we plan to utilise this dataset solely for applying LSH with the finalised hyperparameters determined through the evaluation process.

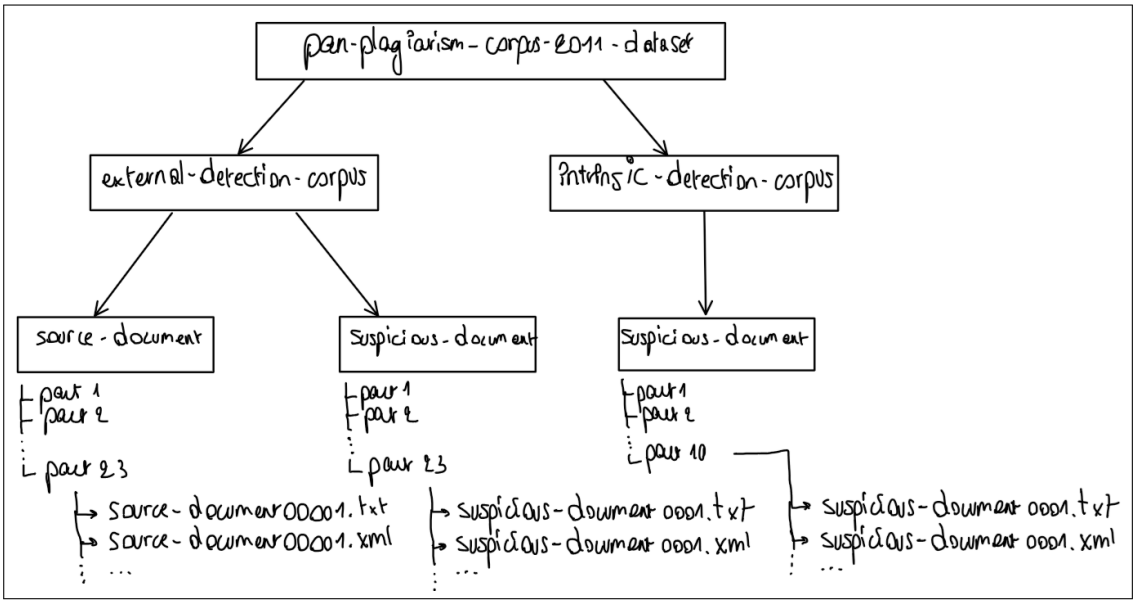

Figure 2: Hierarchy of the Corpus Dataset.

2.3 Java Dataset

The Java dataset is accessible at: `https://github.com/oscarkarnalim/sourcecodeplagiarismdataset`. The README file within this repository provides an excellent description of the dataset's structure, summarised and partially taken over as follows. This repository comprises 467 Java source code files, categorised across seven introductory programming assessment tasks. Each task is systematically organised into three directories, as depicted in Figure 3:

1. "Original" contains the original code for each task.
2. "Non-plagiarized" includes multiple sub-directories, each containing a code file developed independently of the original task code.
3. "Plagiarized" features six sub-directories, each representing different levels of plagiarism. Further sub-directories contain code files explicitly plagiarised from the original task code.

The diverse forms of plagiarism present in this dataset offer a comprehensive foundation for evaluating the efficacy of our detection tool across various levels of plagiarism severity [6]. Currently, our focus is limited to text file datasets, as their processing is inherently more straightforward and sequential.

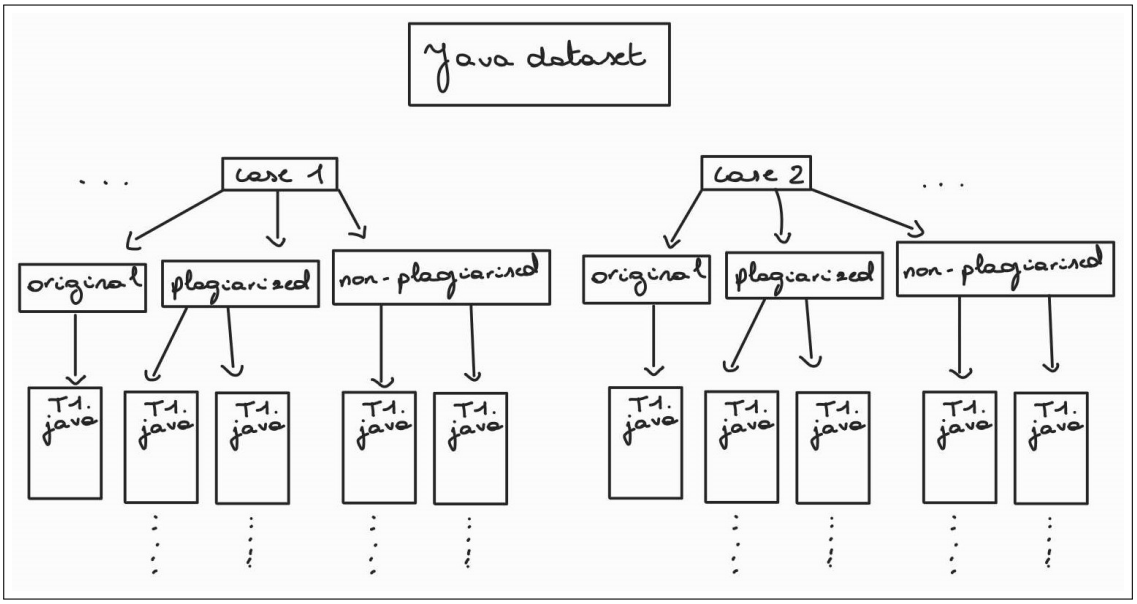

Figure 3: Hierarchy of the Java Dataset.

3 Preprocessing

After acquiring our datasets, the subsequent step involves further processing them in preparation for LSH. This preprocessing consists of several steps, including the removal of stopwords—such as "the," "in," "our," and "these"— and normalising words, which primarily serve to increase the textual volume without contributing meaningful information. Removing these predictable words can increase the relative entropy of the sentences, as per concepts in information theory, by focusing on more meaningful and less predictable content. This section will detail all the steps to prepare the datasets.

3.1 Dataset Extraction

Preprocessing begins with extracting the dataset, which should be located in `assets/`. In our `constants` file, we have defined several constants for this step, namely: "FRESH_RUN", "WIKIPEDIA_DATA", "EXTRACT_ALL_FILES", and "FILE_CUTOFF". The "FRESH_RUN" boolean determines whether to delete all files except those in the `assets/evaluation` folder to start with a clean slate. This automatic removal is beneficial when changing the file cutoff point or switching datasets. If set to false, and existing data is detected in the `assets/` folder, this data will be loaded into memory to avoid re-computation of the preprocessing steps detailed subsequently. This practice of checking for and utilising existing processed data, if available, is applied consistently throughout all project phases. By doing so, we maximise the use of cached data across all steps, ensuring efficiency and minimising unnecessary processing workload. The "WIKIPEDIA_DATA" boolean specifies whether to use the Wikipedia dataset or the Corpus dataset. The "FILE_CUTOFF" constant defines the number of original documents we intend to process from our dataset. If this value is set to *-1*, "EXTRACT_ALL_FILES" is set to true, extracting the entire dataset.

Given this configuration, it is evident that our project utilises both the Corpus and Wikipedia datasets. However, each dataset requires a unique extraction script due to differing structures. Following the extraction step, all original and suspicious documents will be accessible in `/assets/dataset`. Concurrently, a set of (original, suspicious) document (name) pairs is generated, maintained in memory, and saved in `/assets/dataset/processed/fraud_pairs.csv`. This methodology is expected to be similarly applicable to the Java dataset in future work.

3.2 Processing the Files

With the (partial) dataset now prepared in `assets/dataset/` and the fraud pairs readily available in memory, we can begin preprocessing these files. The specific steps involved are detailed in the subsequent subsections.

3.2.1 Preprocessing Text Files

The initiation of the preprocessing step launches a distributed, step-by-step procedure. Our system scans all files within the `assets/dataset` directory and maintains their file paths in memory. Subsequently, these paths are allocated among the available threads (# number of threads divided by 2) to facilitate parallel processing of each file. The specific steps for each file are illustrated in Figure 4. A crucial step not depicted in the figure involves identifying the type of each file. In our system, files may either contain code or text. The preprocessing approach varies accordingly, reflecting a *normalisation behaviour* specific to each file type, similar to the

strategy design pattern in software engineering. This approach ensures that the code remains adaptable to potentially new file types that may be introduced in future developments.

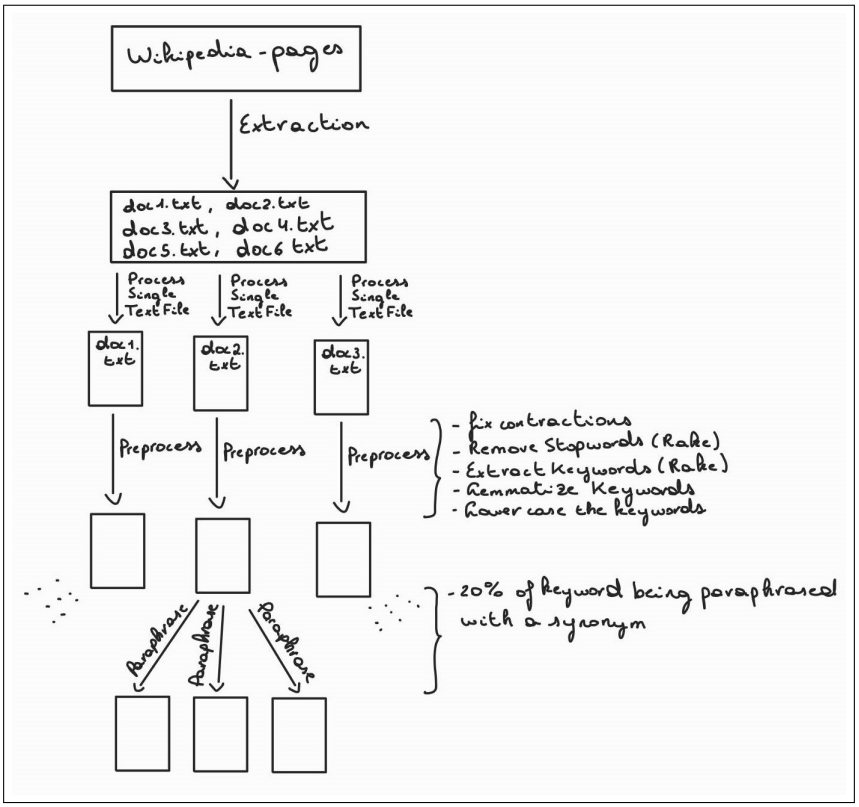

Figure 4: Linear diagram illustrating the step-by-step preprocessing of each text file in the dataset.

Processing a single file, as depicted in the figure above, commences with fixing the contractions of the text. Contractions, such as "don't", "we'll" or even shorthand descriptions of months, such as "Jan", are transformed into their expanded forms, such as "do not", "we will" and "January". This initial step is already the first step to bringing forth consistency in all texts.

Next, we utilise an external library named "Rake", encapsulated within a "CustomRake" wrapper class, to extract key phrases from the text. We developed this custom Rake class because the standard Rake implementation lacked the flexibility required for our specific keyword extraction needs; particularly, we needed the capability to adjust the proportion of text retained from each sentence. This proportion is governed by the constant "KEYWORD_SELECTION_RATIO". Setting this ratio to *1.0* preserves the entire sentence in its original order. Any value less than this initially tokenises the sentence into individual keywords, removes all stopwords and punctuation, assigns a score to each keyword, and sorts the keywords by their score in descending order. Subsequently, only the top percentage of keywords, as specified by the ratio, is retained. By default, we retain only the top 10% of a sentence's keywords. An additional safeguard is implemented to ensure no loss of content: if a sentence's keywords are empty after applying the 10% retention, at least one word is retained. This measure prevents the loss of sentences, which

could disrupt the text's structure and impact the efficacy of LSH.

Following the processing by Custom Rake, we further refine the output by manually removing any remaining punctuation using Python's built-in regex capabilities. This step uses a regular expression designed to identify and eliminate punctuation, addressing the shortcomings of Rake's punctuation removal, which is not comprehensive. Although termed a list of "keywords", each keyword may comprise multiple words. Consequently, we perform word tokenisation on the result and flatten the resulting list to ensure each word is processed separately. Subsequently, each keyword undergoes lemmatisation to reduce the variety of unique words. For example, forms such as "running" and "ran" are normalised to "run", "dogs" to "dog", and "churches" to "church". Moreover, we strip any *'s* from each keyword and convert all keywords to lowercase if not already handled by Rake.

The list of keywords is then transformed into a character sequence without whitespace to optimise it for our LSH algorithm. Subsequent sections will discuss the specifics of this transformation in detail. Finally, the preprocessed data is stored in `assets/preprocessed/` under the same original filename. After implementing this preprocessing step, we observed a reduction in file size ranging between 80% and 85%.

3.2.2 Preprocessing Code Files

Preprocessing code files are similar to text files; each is processed in a separate thread. Initially, we remove comments using a regex pattern, simplifying the subsequent steps. Following this, we eliminate all new lines and tabs. We then employ an external library named "tokenize-all"[2], which processes the code and converts it into tuples consisting of keywords and token types, along with their start and end indices relative to the original file. Maintaining these indices enables us to reconstruct the file for validation purposes. Furthermore, having access to these token types assists us in determining which types are crucial for our analysis.

3.3 Paraphrasing

The preprocessed files are in `assets/preprocessed/`. At this stage, each file should show consistent characteristics relative to the others, enhancing the detectability of similarities and potential plagiarism. Nonetheless, it became clear that conventional plagiarism detection mechanisms could be circumvented using synonyms as a paraphrasing technique. Such modifications often result in different shinglings, regardless of the shingling method employed, and thus may not be identified by LSH. To address this challenge, we generate suspicious documents and create paraphrased versions of them. This approach aims to enhance our system's detection capabilities against such sophisticated forms of plagiarism. The subsequent sections will elaborate on the processes involved in generating these paraphrased versions.

3.3.1 Paraphrasing Text Files

For each suspicious document, up to three paraphrased versions are generated by substituting each keyword with one of its synonyms. There is a 20% probability assigned to each keyword for undergoing paraphrasing. The paraphrased files retain the original filenames but are appended with "paraphrased" and the iteration number to distinguish them from the original files. These versions are subsequently stored in `/assets/paraphrased/`.

[2] `https://pypi.org/project/tokenize-all/`

3.3.2 Paraphrasing Code Files

The paraphrasing of code files involves replacing keywords with their corresponding token type names. This replacement occurs a predetermined number of times based on the specific run configuration. This method aims to simulate various degrees of code manipulation which might occur in real-world scenarios.

3.4 Multithreading

The processing of data files is ideally suited for parallel execution, as a separate thread can independently handle each file. This approach, previously mentioned in Section 3.2.1, is implemented using Python's *ThreadPoolExecutor* class, facilitating task distribution at the thread level. However, during the initial attempts, we encountered challenges related to our preprocessing packages. Specifically, the method for fetching and caching the list of stopwords in a single file created a blockage; when one thread accessed this file to retrieve the stopwords, it imposed a lock on the file, thereby preventing other threads from accessing it simultaneously, which led to exceptions being thrown. To address this issue, we adapted our strategy to preload all necessary data, including stopwords, during the application's startup phase and store them in the commonly accessible `constants` file. This modification ensures that these resources are available as shared, in-memory assets, enabling multiple threads to access them concurrently without facing locking issues.

4 LSH

Following preprocessing and generating paraphrased files, LSH is performed to identify similar text documents. LSH consists of three key steps:

- **Shingling**: This initial process converts text documents into sets of contiguous subsequences, known as "shingles". This technique is crucial for capturing the structural structure of the text, which facilitates similarity comparisons. We employ two distinct approaches to shingling—character-based and word-based—and will explore both methods in greater detail in subsequent sections.
- **MinHashing**: This technique efficiently approximates the similarity between sets of shingles. MinHashing compresses these sets into compact representations. This step significantly reduces the dimensionality of the data, making the comparison process more manageable and less resource-intensive.
- **LSH Algorithm**: The final stage involves applying the LSH algorithm, which hashes the MinHash signatures into buckets. This strategy ensures that similar documents are likely to be hashed into the same bucket, thus optimising the process for quickly retrieving near-duplicate documents.

4.1 Shingling

Shingling is breaking down text into smaller units called *shingles* or *n-grams*. We tested various sizes, such as 2, 5, 7, and 9, and also varied the shingling approach between character-based and word-based methods to compare the effectiveness and impact of each factor. Additionally, we experimented with different window step sizes to determine the optimal configuration for capturing the structure and similarity of the text documents. The window step determines how the text is traversed when creating shingles. For example, with a window step size of 1, we move forward in the text by one character for each shingle. However, if the shingle size is two characters, the same character may appear in multiple shingles.

Character-based shingling breaks down text into smaller units based on individual characters, such as letters or punctuation marks, while word-based shingling breaks down text into units based on whole words. We found that character-based shingling captures more granular text details, including character sequences and punctuation, making it suitable for capturing structural similarities in texts. In contrast, word-based shingling focuses on whole words, preserving semantic meaning and context, which is more effective for capturing contextual relationships between words in the text.

After the shingling process is completed, we use the shingles to create a matrix that encodes the presence of shingles in each document. Given the nature of shingling, this resulting matrix often exhibits sparsity, especially when dealing with extensive document sets, as numerous potential shingle combinations may emerge. In this step, we implemented the sparse matrix in the "Compressed Sparse Row" format to reduce the memory overhead. Figure 5 shows an example of such a matrix. The columns represent the various text documents, while the rows represent all the possible shingles across all documents. Each cell in the matrix denotes the presence or absence of a shingle in a particular document.

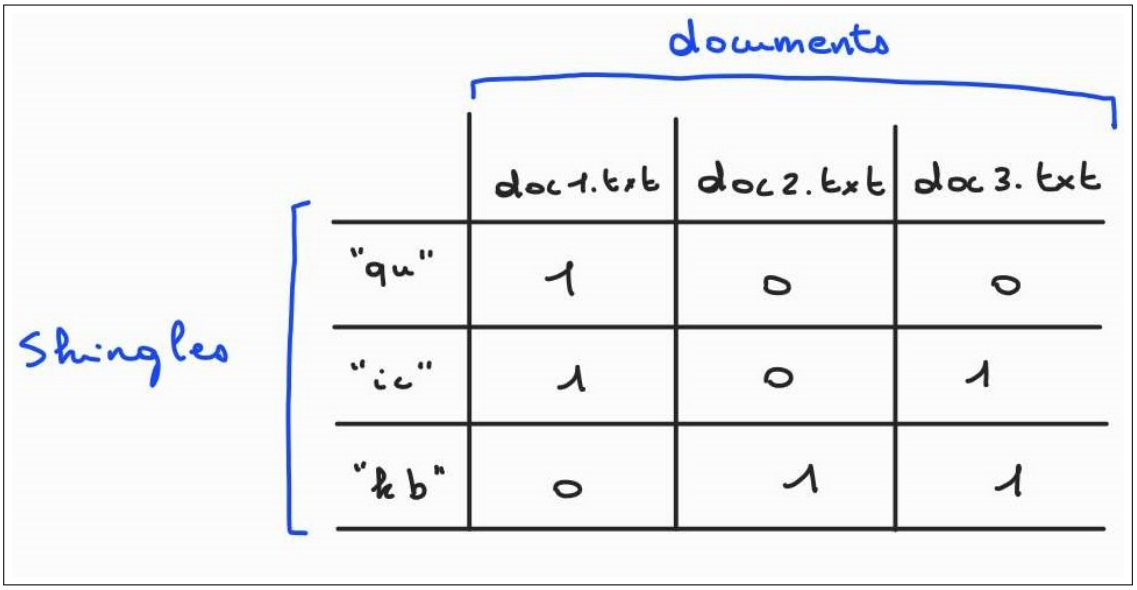

Figure 5: Example of a Sparse Matrix Shingle-Document.

4.2 MinHashing

MinHashing is a technique used to encode the similarity between documents more efficiently. This similarity is achieved by creating a signature matrix where each column represents the signature for a particular document. This signature is a compact document representation and maintains document similarity using minhash functions. By reducing the dimensionality of the data while preserving its essential characteristics, MinHashing facilitates quicker and more efficient similarity searches in large datasets [7].

The basic implementation of MinHashing relies on permuting the rows of the boolean shingle matrix with a minhash function. The resulting matrix has dimensions of the number of hash functions by the number of documents matrix, with each cell containing the smallest permuted index of a shingle present in the document. An illustration of this process can be seen in Figure 6. As the number of hash functions grows, permuting all the rows for each function becomes computationally expensive. Thus, a "one-pass implementation" is recommended [7].

In the one-pass implementation, the signature matrix is initialised with maximum values. For each shingle present in a document, only the index of the current shingle is hashed using each hash function. If the hash value is smaller than the current value in the signature matrix for that document and hash function, it updates the signature matrix. Given the natural sparsity of shingle matrices, we implemented a variant of this method. We optimise computation by using sparse matrix implementations and NumPy broadcasting. Instead of iterating over each document for every shingle, we simply loop through the shingles, select only those documents containing the current shingle, and update their values for all hash functions simultaneously with broadcasting. This results in significantly reduced memory overhead and enhanced computational speed. When performing MinHashing on only fifty documents with a character-based approach on the Corpus Dataset, see Section 2.2, the original one-pass implementation took 120 seconds. After implementing our optimisations, the time required for this step was 14 seconds

in the same environment. Indicating a performance increase of around 88%. The environment for these tests consisted of the following hyperparameters:

```
{'window_step': 3, 'shingle_size': 9, 'n_hash': 300, 'n_bands': 50, 'K': 100000}
```

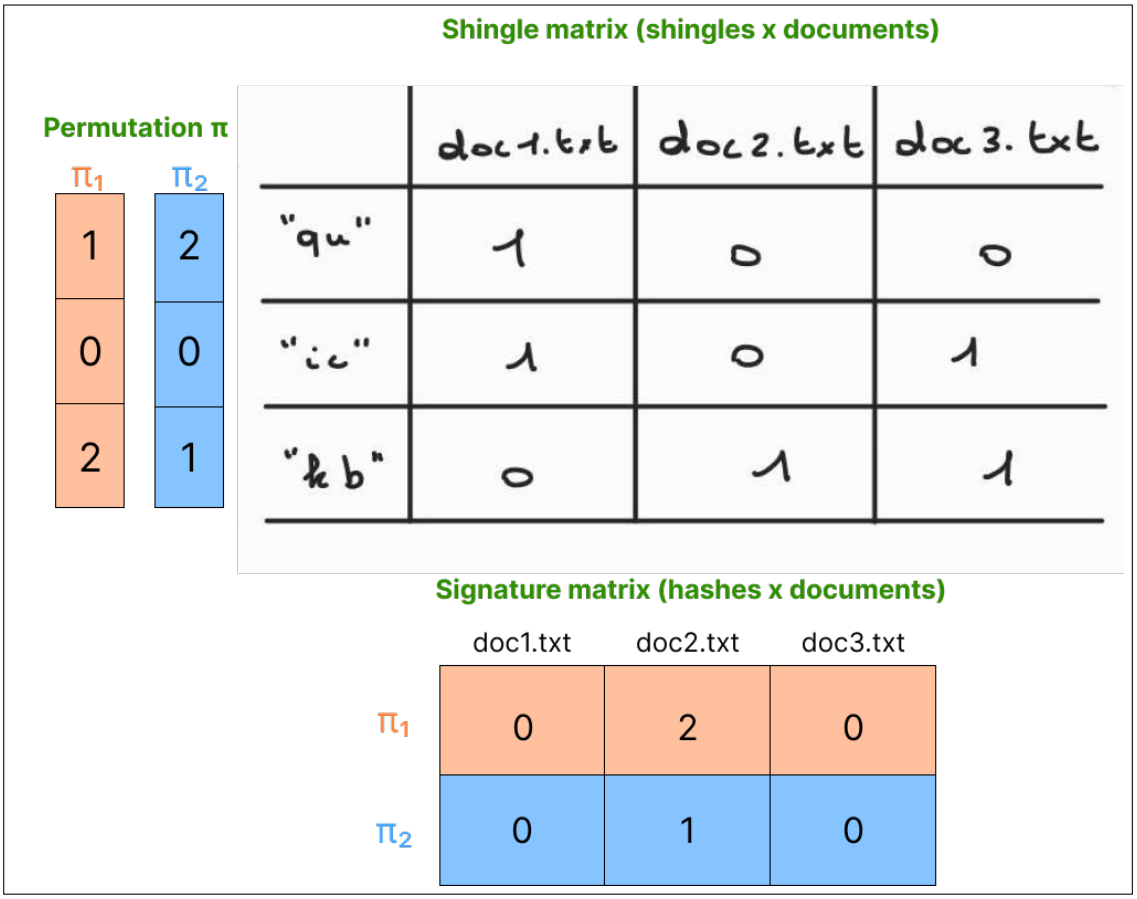

Figure 6: Example creation of a signature matrix with MinHashing

4.3 LSH Algorithm

The LSH algorithm is a crucial step in detecting near-duplicate documents. It functions by hashing fractions of the signatures of documents generated earlier into buckets. This hashing process strategically groups similar items into the same bucket, facilitating the efficient retrieval of near-duplicate documents. By consolidating similar documents into common buckets, LSH significantly diminishes the search space, thereby expediting the identification of potential duplicates within large datasets [7]. Figure 7 illustrates an example of how the LSH step operates. We optimised the memory overhead of the LSH algorithm by storing only a one-dimensional vector of buckets in each band and instantaneously comparing documents within this band. This one-dimensionality allows us to overwrite the buckets when iterating the various bands. Furthermore, we implemented NumPy's "apply_along_axis" method to hash all the signatures in a band

more efficiently. When applied in the same environment as the MinHashing, these enhancements reduced the LSH step's runtime from 90 seconds to 0.05 seconds.

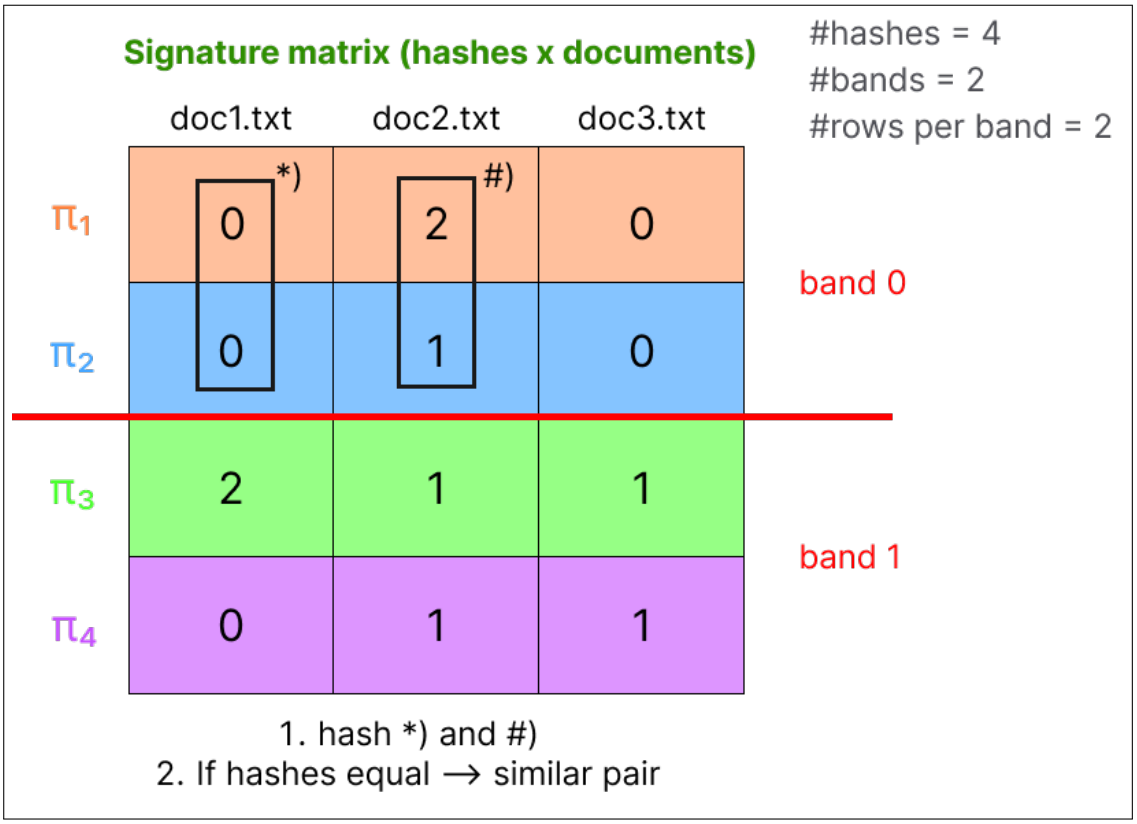

Figure 7: Example of finding similar documents using the LSH algorithm

4.4 Hashing

In LSH applications, selecting the appropriate hashing technique is crucial, and universal hashing emerges as a recommended approach due to its ability to preserve various pre-defined attributes [7]. Universal hashing involves randomly selecting a hash function from a family of hash functions. This family is distinguished by its ability to uniformly distribute keys across the hash table, thereby minimising collisions and ensuring efficient data storage and retrieval. For our implementation, we opted for a linear hash family. This choice was made based on its potential to generate hash functions that distribute keys uniformly and are computationally efficient. It is worth noting that different hash families may offer distinct advantages, but the linear approach proved effective for our purposes. However, one drawback of universal hashing is its instability. Even minor alterations to the coefficients within a function can lead to significantly different outcomes [3]. To address this concern, we implemented a seed mechanism to maintain consistency when hashing across different environments. By anchoring the hashing process with a predetermined seed, we mitigate the risk of unpredictable variations in hashing outcomes, ensuring stability and reliability in our application.

5 Evaluation

The initial stage of the evaluation process involves establishing a baseline by performing Locality Sensitive Hashing on the raw dataset without preprocessing or paraphrasing. This base-case analysis helps us understand the effects of subsequent preprocessing steps and shingling methods we apply. We implement an exhaustive evaluation to measure the effectiveness of our LSH tool in identifying near-duplicate documents and to assess the impact of various preprocessing hyper-parameters. This approach is consistent across all subsequent evaluations. We then normalise the data and evaluate the influence of the number of versions of a file containing paraphrased words on the efficiency and accuracy of our program. After determining an optimal value for this parameter, we explore the effects of selecting varying fractions of keywords from sentences based on their importance. To exhaustively assess these factors and identify the best LSH model configuration under these conditions, we conduct four distinct runs:

- Word-based run, retaining a percentage of the keywords from preprocessing in each sentence as defined by $(KEYWORD\_RATIO)\%$.
- Character-based run, also retaining a percentage of the keywords as per $(KEYWORD\_RATIO)\%$.
- Character-based run, retaining all keywords from preprocessing.
- Word-based run, using all keywords from preprocessing.

5.1 Processing Pipeline

Each major evaluation run follows a complete "pipeline": Firstly, all preprocessed, paraphrased, and extracted data are removed. Data extraction and preprocessing are then performed. Next, we initiate our optimisation process to determine the best hyperparameters for our LSH model. To identify optimal parameters, we exhaustively optimise the LSH process (shingling, min-hashing, LSH) over the entire parameter space. Each run is conducted with the following parameters:

- Shingle size.
- Window step: This parameter adjusts the step size for the sliding window that generates the shingles, influencing the overlap of shingles based on shingle size.
- Number of hash functions (n_{hash}): Used in the MinHashing algorithm.
- Number of bands (n_{bands}): Used in the LSH algorithm.
- K: Defines the bucket size in LSH.

5.2 Performance Metrics

Through tuning these parameters, we aim to discover the optimal combination that maximises the model's effectiveness in detecting similar documents. Each parameter combination yields:

- F1 score: The harmonic mean of precision and recall, used to measure the performance of LSH with specific parameters.
- Parameters: The specific parameters used in each run.
- Precision: The accuracy of positive predictions, calculated as the ratio of true positives to the total predicted positives.

- Recall: The model's capability to identify all true positives is computed as the ratio of true positives to the actual positives.
- False Positives: The number of incorrect positive predictions.
- False Negatives: The number of missed true positives.
- True Positives: The number of correctly predicted positives.
- True Negatives: The number of correctly predicted negatives.

5.3 Location of Results Data

Data from all runs are stored under `/assets/evaluation`. Our main function, which executes the four major runs, is repeated 12 times. These iterations are labelled in our project under `/assets/` as `/evaluation-iter-{mainiteration}`. Each main iteration includes a `complete_evaluation.csv`, which details the four major runs, their execution times, and the best F1 scores. Figure 8 illustrates the folder structure for a typical large run.

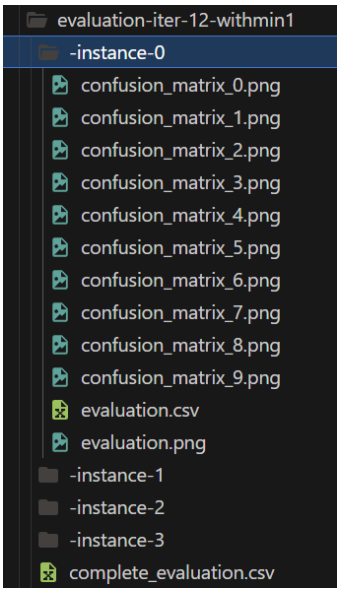

Figure 8: Large Run Folder Structure.

The confusion matrix, as depicted in Figure 9, compares the predicted labels to the actual labels, distinguishing between predicted positives and actual positives, as well as predicted negatives and actual negatives. All LSH runs, and their respective parameters, precision, recall, false positives, F1 scores, and other metrics are documented in `evaluation.csv`. The top 10 F1 scores and their corresponding parameters are visually represented in `evaluation.png`, as illustrated in Figure 10.

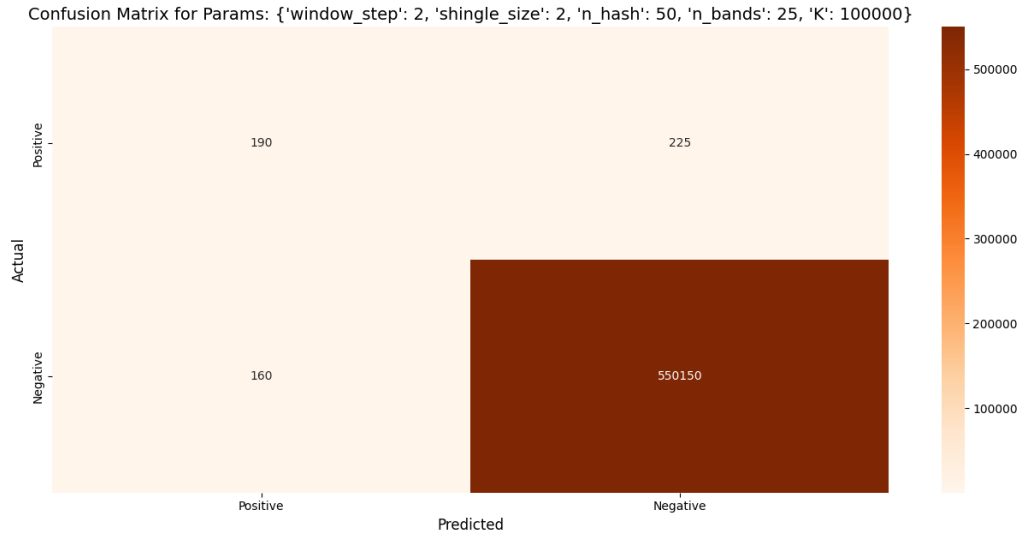

Figure 9: Example of a Confusion Matrix.

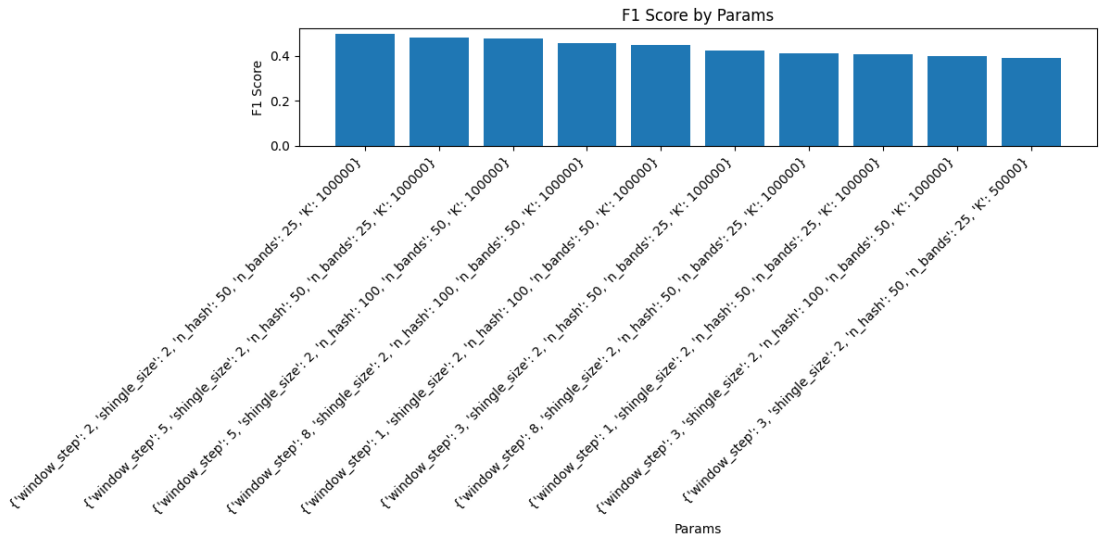

Figure 10: Example of an Evaluation.

5.4 Benchmarking PC

The computer's specifications used for the evaluations are fully detailed below for completeness. This information provides context on the computational resources employed for the runs. It is important to note that while the same evaluations can be conducted on computers with less powerful specifications, the processing times may be longer.

For instance, the benchmarking computer was configured with the following specifications:

- **CPU**: Intel® Core™ i7 processor 14700K (5.6GHz) with 8P- and 12E-cores, totalling 28 threads.
- **RAM**: G.Skill Trident Z5 Neo 64 GB (2 x 32 GB) DDR5-6000 CL30 Memory
- **SSD**: Western Digital Black SN850X 2 TB M.2-2280 PCIe 4.0 X4 NVME Solid State Drive with 7.300MB/s READ, 6.600MB/s WRITE

5.5 Results

This section discusses the changes in model performance across different preprocessing environments. For each environment, every word in a paraphrased version had a 20% probability of being paraphrased. Iterative evaluation is recommended to mitigate noise and anomalies. However, it was impossible to do this due to the long run times needed to optimise the LSH hyperparameters. Initially, we examine the performance of LSH applied directly to the raw dataset. The results from this baseline analysis provide a reference point for understanding the impact of subsequent modifications. Subsequently, we analyse the influence of varying the number of paraphrased versions on the model's efficiency and accuracy. Finally, we explore the effects of different keyword selection percentages from sentences and identify the best-performing model configurations.

5.5.1 LSH on Raw Dataset

Applying LSH to the raw dataset, without preprocessing, yielded F1 scores of 0.56 for Word-based shingling and 0.53 for Char-based shingling. These results indicate that the tool predicts fraudulent pairs within the dataset with an accuracy of approximately 53-56%. Both shingling methods took around 15.5 seconds to complete the entire process. The findings are illustrated in Figure 11. With these baseline F1 scores and durations established, we now process the dataset further to evaluate the impact of these preprocessing steps.

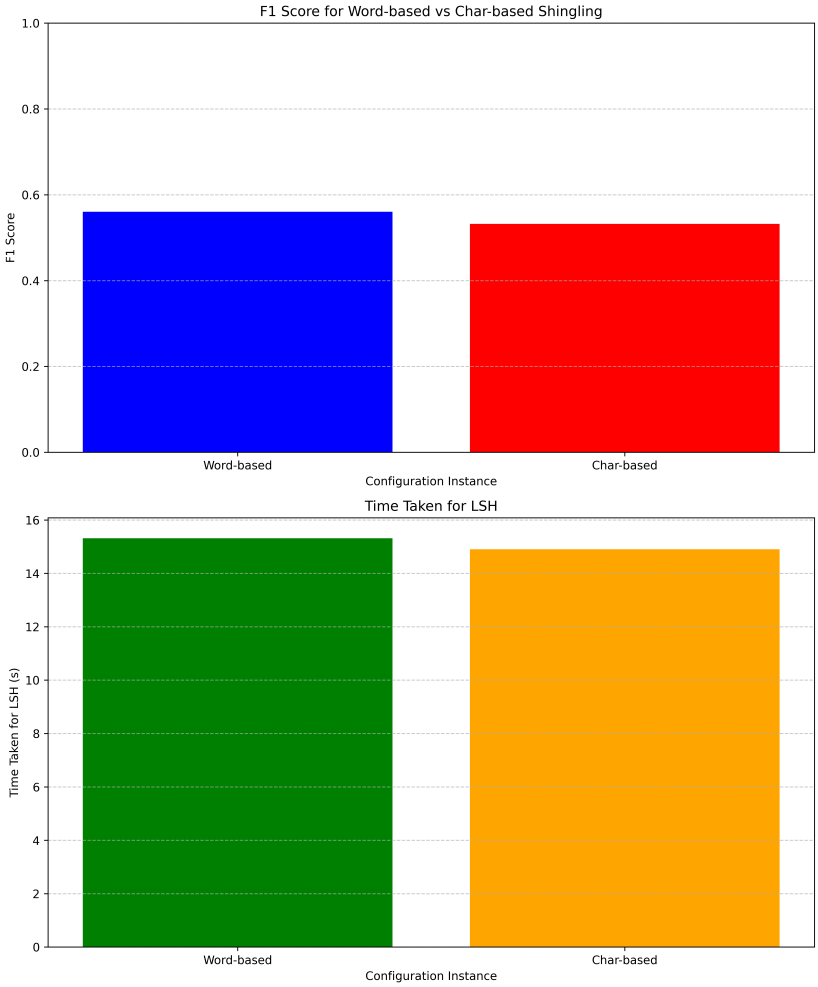

Figure 11: Performance of the LSH model on the Raw Dataset.

5.5.2 Impact of Paraphrased Versions

In the first experimental environment, our goal is to understand the effect of creating copies of a file with certain words paraphrased. We create these copies to simulate plagiarism committed through machine obfuscation. Using a library to paraphrase specific words can help detect such cases and improve model performance. If no obfuscation exists in the documents, these additional versions should not affect the model's accuracy. We kept all other preprocessing hyperparameters constant and only varied the number of paraphrased files to isolate the impact of this parameter on our program.

Our findings indicate that increasing the number of paraphrased versions of a file from 0 to 1 and 3 diminished the accuracy and efficiency of our model. This can be attributed to the 20% probability for a word to be paraphrased, which could create excessive, unwanted pairs between documents. The only exception to this result is the run-time for the character-based shingling approach on full sentences, which runs faster when creating one paraphrased version than when

no additional files are created. The results can be seen in Figure 12 and Figure 13. We observe that the F1 scores are higher when selecting a fraction of keywords from sentences than when working on the full sentences. Additionally, the word-based shingling approach slightly outperforms the character-based approach, which, combined with running nearly twice as fast, can give some initial insights into plausible results later on. The best model, namely the word-based approach when operating on simply the documents and no additional paraphrased versions, attained an F1 score of 0.54, similar to that of our base-line, but also took around 7.5 seconds to perform LSH, which is 50% faster than our base-line duration of 15 seconds when using the following LSH hyperparameters:

```
{'window_step': 3, 'shingle_size': 2, 'n_hash': 100, 'n_bands': 50, 'K': 100000
```

Next, we will investigate the reliability of these results further.

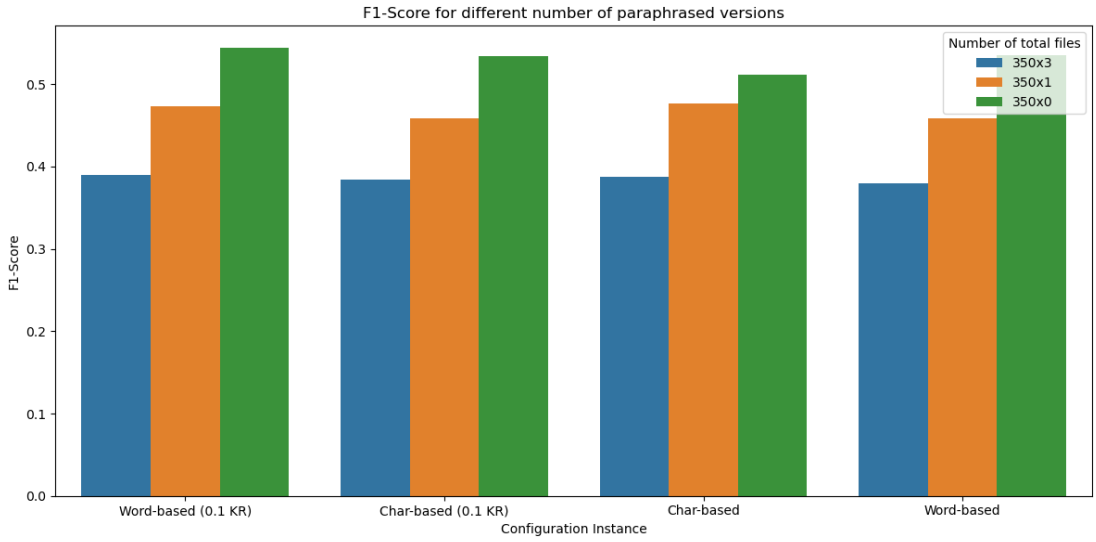

Figure 12: F1 scores for the LSH model for 0, 1 and 3 paraphrased versions.

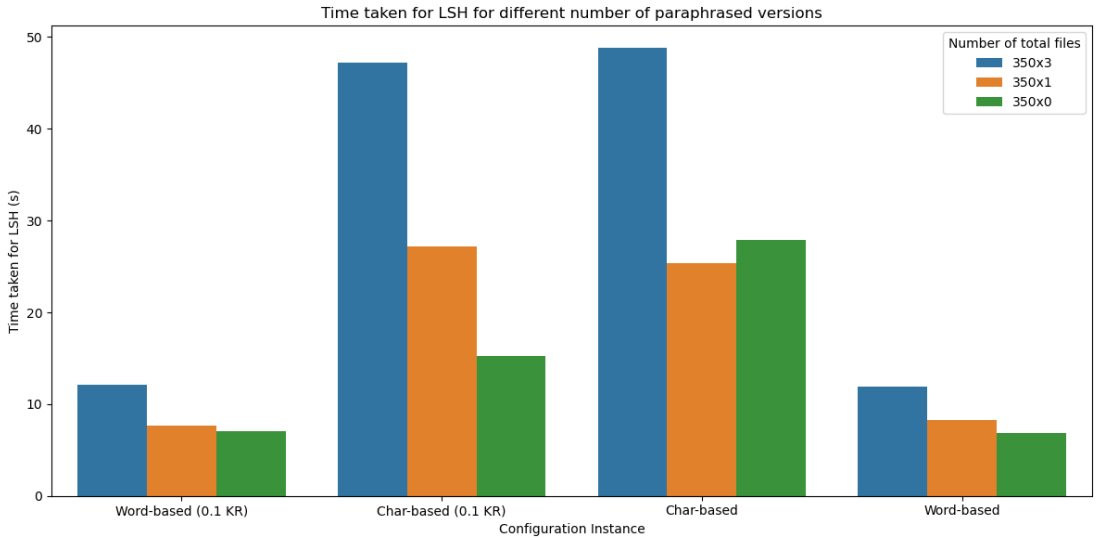

Figure 13: Duration for the LSH model for different numbers of paraphrased versions

5.5.3 Impact of Summarising Sentences

In the second experimental environment, we aimed to understand the impact of summarising sentences to a fraction of their keywords. In (H4), we hypothesised that summarising sentences creates a trade-off between performance and the tool's accuracy, which would be justified. Notably, taking too few keywords could mean we lose out on common patterns between documents. Therefore, it is crucial to find a balance for this hyperparameter. We did not create any additional paraphrased versions of documents in this environment, as we found earlier that these harm the model's accuracy. We performed four different iterations in this environment, where we altered the ratio of the keywords we selected between 10%, 30%, 50%, and 100%.

We found that summarising sentences minimises the run time when opting for a word-based approach. We even see diminishing returns when summarising sentences too much; using only 10% of the keywords in a sentence takes longer than taking complete sentences. This occurrence can be explained since we will, by default, take at least one word when sentences are short. A sentence with only five words will return the same amount of keywords for ratios varying between 0% and 20%. When utilising character-based shingling, summarising the sentences in the documents has a significant positive effect on the run time of our model. Figure 14 and Figure 15 summarise the results obtained from the evaluation in this environment. We found that the F1 scores for the word-based and character-based approaches are highly similar. This similarity indicates that taking fewer keywords does not significantly affect either approach. We note that we optimised the hyperparameters in each run; this explains the similarity of values across different runs. When reducing the keyword ratio, it is possible that the best model reduces its window size and, therefore, has the same run time. Our best model, namely the character-based approach, which operates on the original, complete sentences, attained an F1 score of 0.55, similar to that of our base-line, with the following LSH hyperparameters:

```
{'window_step': 8, 'shingle_size': 2, 'n_hash': 100, 'n_bands': 50, 'K': 100000}
```

We note that this model only slightly outperforms the word-based approach, which operates

on sentences summarised by 10% of their keywords and attains a score of 0.54 while being around five times slower and twice slower than its baseline counterpart.

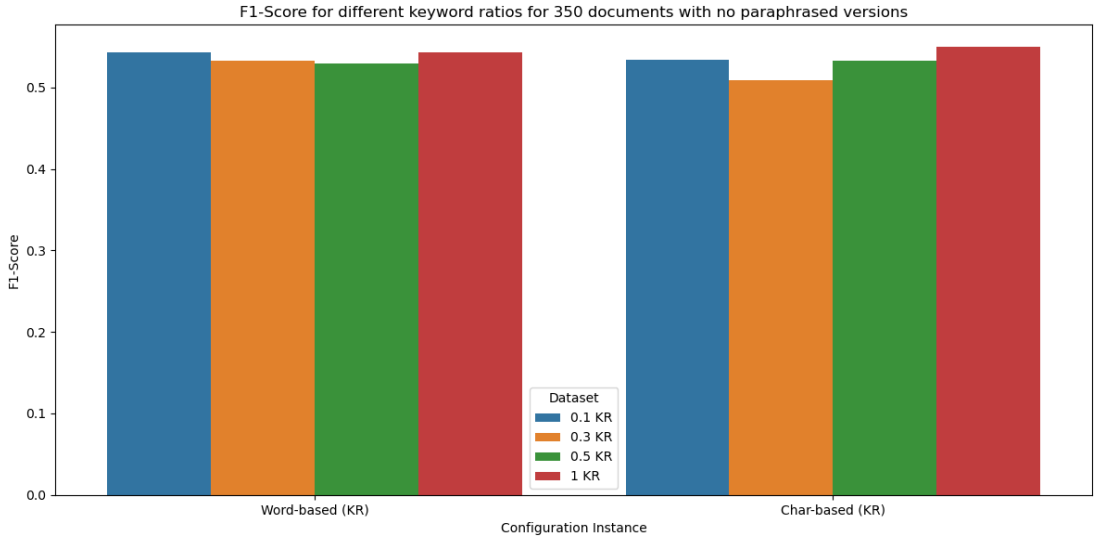

Figure 14: F1 scores for the LSH model when selecting different fractions of keywords from sentences.

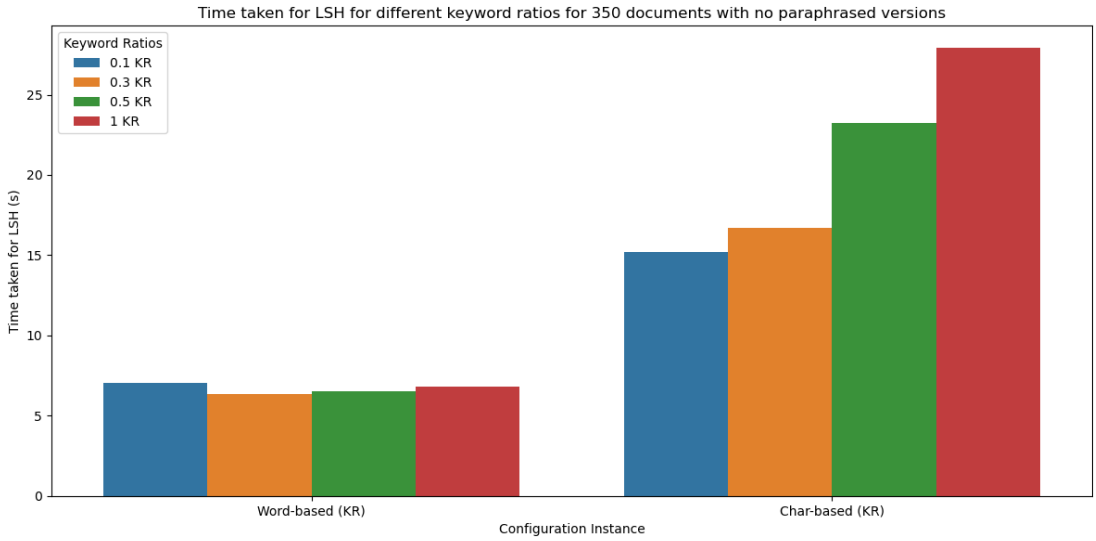

Figure 15: Duration for the LSH model when selecting different fractions of keywords from sentences.

5.5.4 Conclusion

These experiments yield exciting results. Firstly, it is clear that preprocessing the data yields faster LSH computation time, up to 50%. Secondly, the trade-off between both approaches favours the word-based approach due to its significantly faster run time (5x faster) and similar accuracy. Creating additional paraphrased versions of a file can lead to unwanted pairs being found, which lowers the model's accuracy and causes unwanted overhead w.r.t. run times. Summarising sentences into a fraction of their keywords remains beneficial as it reduces run times for both approaches while maintaining similar accuracy.

6 Scalability

Throughout this project, considerable efforts have been made to optimise performance at every process stage. We have implemented multithreading to accelerate file processing, caching mechanisms for finding synonyms for keywords, utilised sparse matrices and NumPy broadcasting to reduce memory usage and prioritised in-memory operations to avoid significantly slower disk accesses. However, there are still opportunities to enhance these processes' efficiency further. These potential improvements are detailed in this section.

6.1 Multithreading

To enhance the efficiency of our LSH tool, we maximised the use of available resources on our local machines by employing multithreading and using as much RAM as possible. This approach was applied strategically in various process stages where parallel processing could significantly reduce execution time. For example, during the preprocessing phase, each file is handled by a separate thread. Additionally, LSH hyperparameters are optimised concurrently, with each run assigned to a different thread. Consequently, the performance gains from multithreading are directly proportional to the number of threads the system supports.

6.2 Garbage Collection

A significant challenge associated with Python in our tool is its automatic garbage collection. As the LSH optimisation step evaluates approximately 1200 hyperparameter combinations, generating shingling matrices and other data structures, this data quickly becomes redundant, consuming valuable main memory. This memory could be more effectively allocated to other processes within the tool or reserved for essential system operations. To address this issue, data structures such as shingles, the signature matrix, and predicted pairs generated from LSH are promptly removed from memory as soon as they are no longer required.

6.3 MapReduce

When working with large datasets (terabytes or petabytes), fitting all the data into main memory is often impossible. Even with relatively smaller datasets, such as one containing 5,000 files that can fit into main memory, our LSH tool may still require hours to execute due to the extensive hyperparameter tuning involved. To enhance scalability and manage larger datasets more effectively, we consider employing Hadoop's MapReduce programming model. Below, we describe a theoretical approach to integrating MapReduce into our workflow.

The initial step involves adapting our dataset extraction process for MapReduce. Specifically, each MapReduce job could handle pairs of original and plagiarised documents for our Wikipedia dataset. In the Reduce phase, each job would write these pairs into a `fraud_pairs.csv` file and store both documents in the `/assets/dataset/` directory. This process is illustrated in Figure 16.

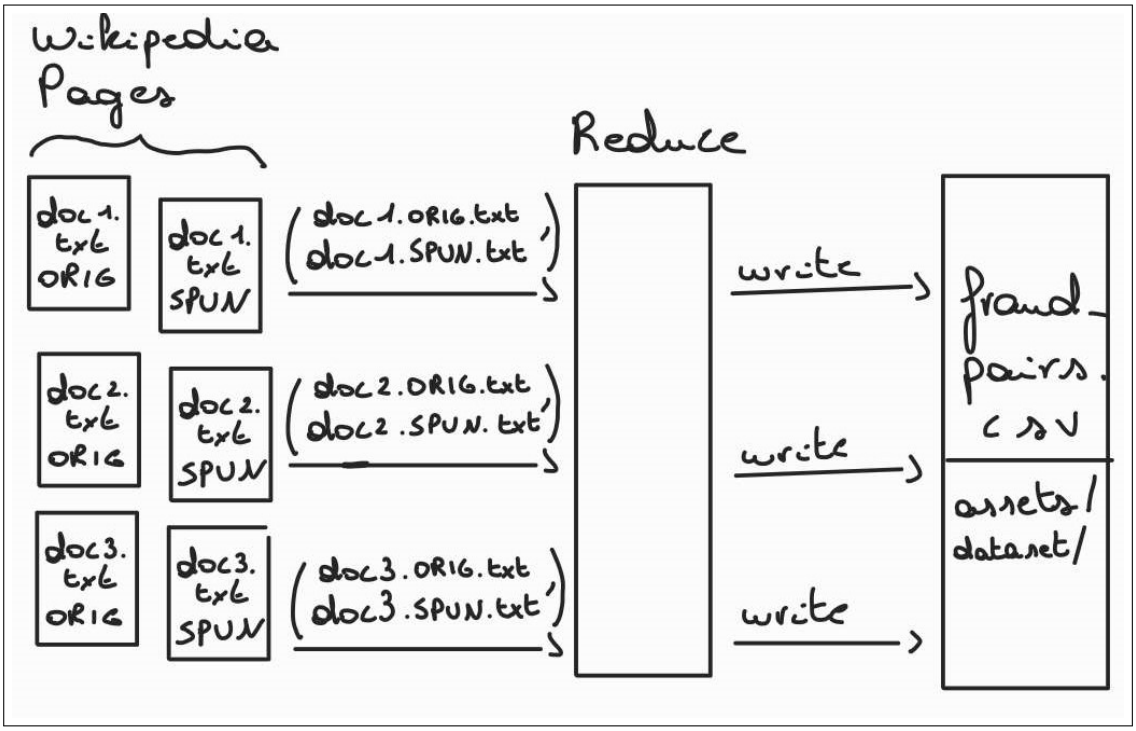

Figure 16: MapReduce (1): Data Extraction Step.

Once all tasks from the extraction step are completed, the entirety of the data will be available in `/assets/dataset/`. Subsequently, the next reduction step could involve distributing the documents into distinct sets, where each worker node processes multiple documents. The output from these nodes would then be directed to `/assets/preprocessed/`, `/assets/paraphrased`, or both directories, depending on the processing requirements. This workflow is depicted in Figure 17.

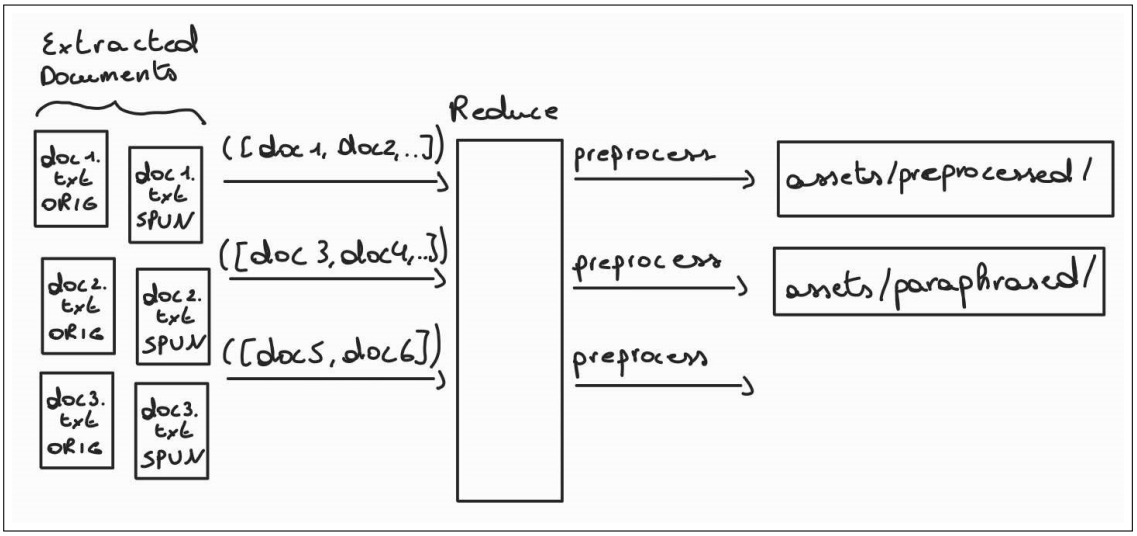

Figure 17: MapReduce (2): Preprocessing Step.

Once preprocessing is complete, the shingling step can be processed using the MapReduce model. Each mapper is tasked with a subset of a document on which word-based or character-based shingling is executed. Mappers generate key-value pairs, where the key is the document ID or name, and the value is the set of shingles. Subsequently, a shuffling step reduces all shingles associated with each document based on the document identifier. Each reducer then aggregates these shingles into a comprehensive list for each document. These steps can be seen in Figure 18.

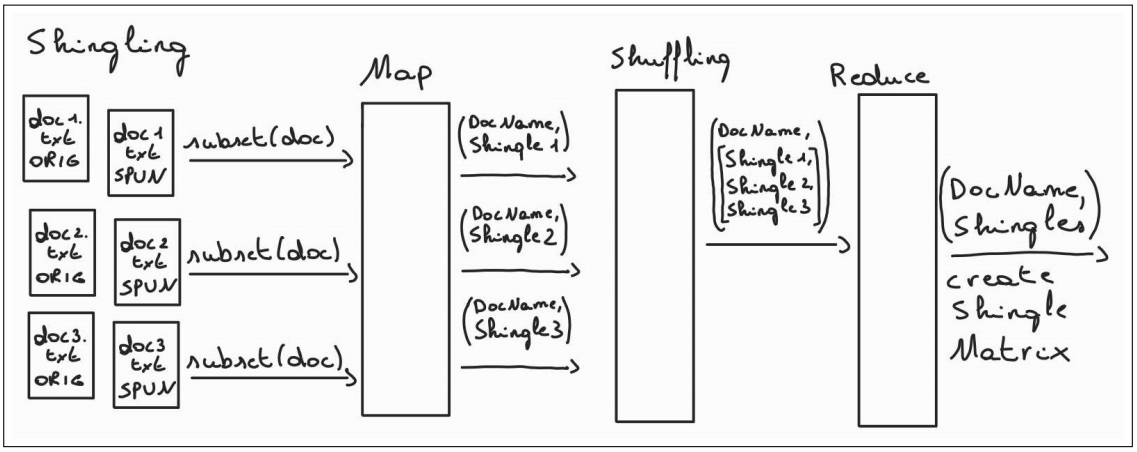

Figure 18: MapReduce (3): Shingling Step.

With the shingle matrix in place, where columns correspond to documents and rows to shingles, we advance to the MinHashing phase to develop the signature matrix. This step is efficiently executed using reducers, as illustrated in Figure 19. Each permutation is applied to the shingle matrix in this phase, generating signatures for their corresponding rows. These signatures collectively form the signature matrix, pivotal for subsequent similarity detection processes.

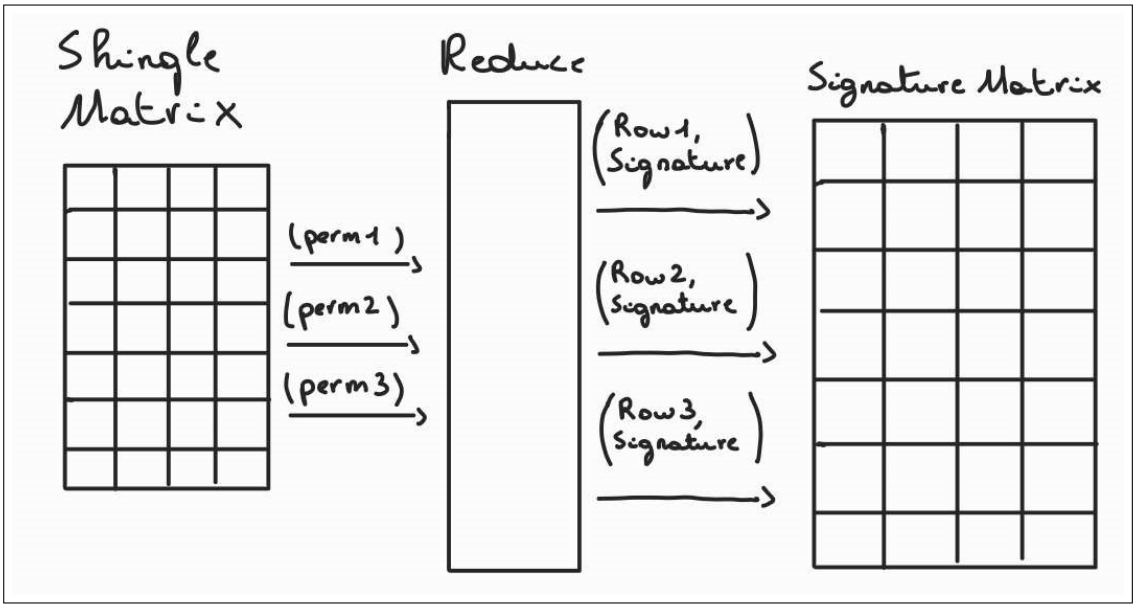

Figure 19: MapReduce (4): Signature Matrix Step.

Once we obtain the signature matrix, we can partition it into multiple bands containing several rows. Mappers can handle individual bands, emitting bucket-document identifier pairs. During the shuffle phase, documents sharing the same bucket are grouped, facilitating the reducers' extraction of candidate pairs. This last step can be seen in Figure 20.

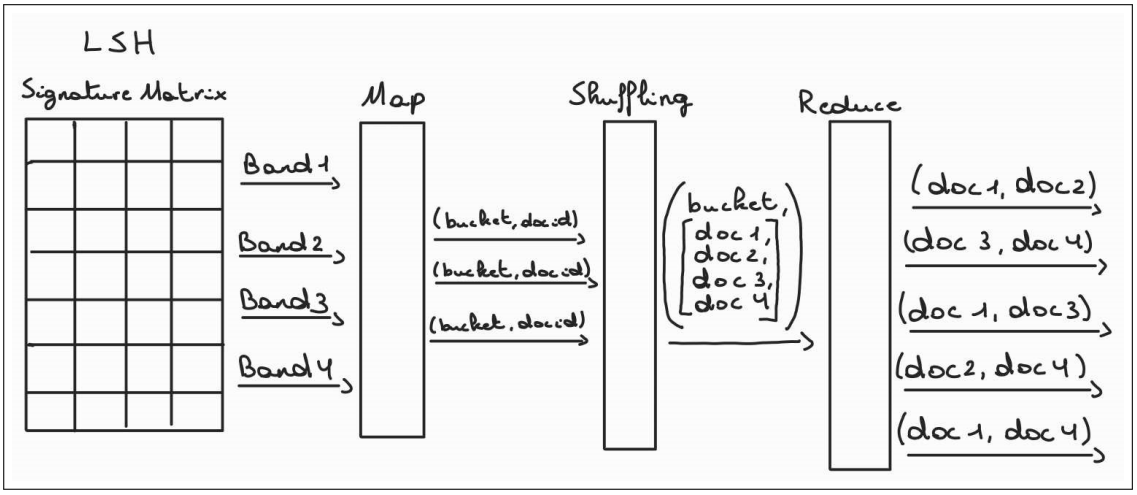

Figure 20: MapReduce (5): LSH Step.

6.4 Spark

An alternative method involves using Spark[3], more actively used in data processing and machine learning than Hadoop's MapReduce. Dataset extraction and preprocessing are effectively executed in Spark through queries and DataFrames, a distributed data collection organised into named columns. Spark also features a built-in query optimiser known as the *Catalyst Optimizer*, which enhances the operations on DataFrames. It achieves this by optimising the query plan, thereby improving performance. By capturing as many expressions over columns as possible and delaying final actions like result collection to the end of the process, the Catalyst Optimizer refines the execution plan. It also minimises the volume of data that needs to be shuffled, reducing (slower) network traffic.

Spark efficiently manages main memory and gracefully handles scenarios where data surpasses memory capacity using disk storage. Additionally, Spark distributes text data across multiple partitions using a specified partitioning strategy.

In the context of shingling, each partition may hold a subset of documents, which are then distributed among worker nodes. Each worker independently carries out shingling for its assigned partitions. After shingling, Spark may require aggregating or combining shingles from various partitions, a task it coordinates effectively.

MinHashing is executed in parallel, with each worker node responsible for processing a permutation. Likewise, for LSH, a MapReduce-like method can be adopted, assigning each worker node a band to process. Aggregation is then conducted to identify the documents within each bucket.

[3] `https://spark.apache.org/`

7 Future Work

Initially, our project intended to process both text and code files. However, we encountered significant challenges handling code files, prompting us to concentrate solely on text files. These complexities included dilemmas on how best to normalise code or whether normalisation was beneficial. Employing syntax trees might offer a more practical approach in this case. While the preprocessing code for handling programming files (e.g., Python or Java), which includes tokenisation and paraphrasing, remains within our code base, we opted not to proceed with code files as this began to extend beyond our project's original scope.

If future work were to adapt LSH to code files, the preprocessing would likely involve:

- Breaking the code into tokens (keywords, identifiers, operators).
- Extracting the Abstract Syntax Tree (AST).

To make this work with LSH, a method would be needed to encode the AST so that similar encoded ASTs are mapped to the same LSH bucket. One potential approach could involve converting these features into vector representations.

Our current approach to paraphrasing simply transforms keywords into their respective token types. This method essentially introduces plagiarism in our processed files. Programming languages like Java also inherently contain many overlapping keywords such as "public", "static", "void", "print", and "function". Therefore, simple text replacement is not a viable option and necessitates a different algorithm. Comparing syntax trees would be a more effective strategy, but implementing this is beyond our current capabilities.

8 Conclusion

In conclusion, our scalable analytics project has successfully explored the effectiveness of Locality Sensitive Hashing (LSH) in identifying near-duplicate documents across large datasets and provided crucial insights into the impact of various preprocessing hyperparameters on the tool's performance. Through an exhaustive evaluation, we have shed light on the influence of the number of paraphrased versions of a file on computation efficiency and accuracy and the impact of selecting different fractions of keywords from sentences.

Our results are promising, showing that preprocessing significantly enhances LSH computation times—up to a 50% reduction. Notably, the trade-off between word-based and character-based approaches favoured the word-based method, which was up to five times faster while maintaining similar accuracy levels. However, creating additional paraphrased versions of files introduced unwanted pairs, thus reducing model accuracy and unnecessarily increasing runtime overhead.

Summarising sentences into a fraction of their keywords proved beneficial, as it consistently reduced runtime for both approaches without compromising the accuracy of the outcomes. Although both word-based and character-based shingling approaches provided comparable F1 scores, the character-based approach operating on original complete sentences slightly outperformed the word-based approach when summarising sentences to just 10% of their keywords.

These findings highlight the importance of balancing performance with runtime efficiency in scalable analytics applications. Optimising LSH hyperparameters is crucial for achieving consistent and reliable results across various iterations. Our project highlights how thoughtful preprocessing strategies are essential in enhancing the efficiency of analytical tools for near-duplicate document detection. This study confirms the practical viability of LSH and makes way for further research to refine these techniques for broader applications.

References

[1] L. Benichou. The Role of Using ChatGPT AI in Writing Medical Scientific Articles. *Journal of Stomatology, Oral and Maxillofacial Surgery*, 124(5):101456, October 2023. doi: 10.1016/j.jormas.2023.101456.

[2] Anna Carobene, Andrea Padoan, Federico Cabitza, Giuseppe Banfi, and Mario Plebani. Rising Adoption of Artificial Intelligence in Scientific Publishing: Evaluating the Role, Risks, and Ethical Implications in Paper Drafting and Review Process. *Clinical Chemistry and Laboratory Medicine*, 62(5):835–843, November 2023. doi: 10.1515/cclm-2023-1136.

[3] Lianhua Chi and Xingquan Zhu. Hashing Techniques: A Survey and Taxonomy. *ACM Computing Surveys*, 50(1):1–36, April 2017. doi: 10.1145/3047307.

[4] Tomáš Foltýnek, Norman Meuschke, and Bela Gipp. Academic Plagiarism Detection: A Systematic Literature Review. *ACM Computing Surveys*, 52(6):1–42, October 2019. doi: 10.1145/3345317.

[5] Tomáš Foltynek, Terry Ruas, Peter Scharpf, Norman Meuschke, Moritz Schubotz, William Grosky, and Bela Gipp. Detecting Machine-obfuscated Plagiarism, 2019.

[6] Oscar Karnalim, Setia Budi, Hapnes Toba, and Mike Joy. Source Code Plagiarism Detection in Academia With Information Retrieval: Dataset and the Observation. *Informatics in Education*, 18(2):321–344, October 2019. doi: 10.15388/infedu.2019.15.

[7] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2nd edition, 2014. ISBN 1107077230.

[8] Jorge Meira, Carlos Eiras-Franco, Verónica Bolón-Canedo, Goreti Marreiros, and Amparo Alonso-Betanzos. Fast Anomaly Detection With Locality-Sensitive Hashing and Hyperparameter Autotuning. *Information Sciences*, 607:1245–1264, 2022. doi: 10.1016/j.ins.2022.06.035.

[9] Martin Potthast, Benno Stein, Andreas Eiselt, Alberto Barrón-Cedeño, and Paolo Rosso. PAN Plagiarism Corpus 2011 (PAN-PC-11), June 2019.

[10] Sadhan Sood and Dmitri Loguinov. Probabilistic Near-Duplicate Detection Using Simhash. In *Proceedings of CIKM 2011, the 20th ACM International Conference on Information and Knowledge Management*, pages 1117—-1126, Glasgow, Scotland, UK, October 2011. doi: 10.1145/2063576.2063737.

List of Figures