# Finding Similar Book Reviews

Using MinHash and Locality Sensitive Hashing

**Algorithm for Massive Datasets**
Final Project – Spring 2024–2025

Student: Yusuf Kemahlı
Instructor: Prof. Dario Malchiodi
Date: June 19, 2025

# Contents

# Chapter 1

# Dataset and Preprocessing

## 1.1 Introduction

This report presents a project developed for the course *Algorithm for Massive Datasets (AMD) – Spring 2024–2025*. The goal of the project is to detect pairs of similar book reviews by applying approximate similarity detection techniques, with a particular focus on the Jaccard similarity and MinHash-based Locality Sensitive Hashing (LSH). The entire pipeline was implemented from scratch in Python and tested on a real-world dataset.

The motivation behind this project lies in the scalability challenges of processing large text corpora. Given the substantial volume and sparsity of high-dimensional textual data, exact pairwise similarity comparison is computationally infeasible. Therefore, we employ approximate methods to efficiently and scalably identify highly similar review pairs.

## 1.2 Dataset Description

We used the public dataset titled **Amazon Books Reviews**, available on Kaggle[1]. The dataset contains approximately **3 million reviews** covering more than **200,000 unique books**, along with metadata such as user ID, book ID, review text, rating, and timestamp. The latest update to this dataset was made three years ago.

## 1.3 Selected Data and Organization

For this project, a random sample of 100,000 reviews was selected from the full dataset. This sample was large enough to preserve key statistical properties of the original data, while also ensuring that the computational pipeline remained efficient and scalable.

[1] https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews

From each review, only two fields were retained: The first element, `Id`, represents the book identifier, while the second element, `review/text`, contains the textual content of the review.

The data was structured as a Pandas DataFrame, with each row corresponding to a single review. This tabular format facilitated the straightforward application of preprocessing, shingling, hashing, and candidate generation operations. Additional fields such as user ID, rating, or timestamp were excluded, as they were not relevant for the similarity detection task.

## 1.4 Preprocessing

In order to prepare the textual data for similarity comparison, several preprocessing steps were applied to the `review/text` field:

- **Lowercasing:** All characters were converted to lowercase to ensure uniformity and avoid case-sensitive mismatches.
- **Stopword removal:** Common English stopwords (e.g., "the", "is", "and") were removed using the NLTK stopword list, as they carry limited semantic meaning.
- **Shingling:** Each review was tokenized into overlapping 5-grams (i.e., sequences of 5 consecutive words), forming a set of word-level shingles.
- **Shingle set creation:** The resulting tokens were stored as Python sets to facilitate Jaccard similarity computation and MinHashing.

The choice of $k = 5$ for shingling balances two conflicting goals: smaller values of $k$ increase recall by allowing more overlapping shingles between similar reviews, while larger $k$ increases precision by capturing more context. A value of 5 was found to be effective in capturing semantically meaningful phrases while avoiding overly frequent generic patterns.

Reviews with too few informative words (i.e., fewer than $k = 5$ after stopword removal) were unable to produce any shingles, resulting in empty sets. These reviews were automatically discarded after the shingling phase to prevent erroneous candidates or misleading comparisons.

These preprocessing steps ensured that the resulting shingle sets were normalized, compact, and semantically meaningful.

# Chapter 2

# Methodological Approach

## 2.1 Jaccard Similarity and the Need for Approximation

Jaccard similarity is a widely used measure for comparing sets. In our context, it is used to measure the overlap between two reviews' sets of word shingles:

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

While exact Jaccard computation is intuitive and effective for small datasets, it becomes impractical when scaling to millions of documents. Computing Jaccard similarity for all pairs of $n$ reviews requires $\mathcal{O}(n^2)$ set comparisons, which is computationally infeasible even for moderately sized datasets.

To address this, we apply the **MinHash** technique as introduced in the textbook *Mining of Massive Datasets*. MinHash allows us to approximate the Jaccard similarity between sets using compact signature vectors, constructed by applying a family of randomized hash functions. The probability that two sets have the same hash value under MinHash is equal to their Jaccard similarity:

$$\Pr[h(A) = h(B)] = J(A, B)$$

By using $n$ independent hash functions, we obtain $n$-dimensional signatures that approximate the Jaccard similarity with increasing accuracy as $k$ grows. In our project, we used $n = 100$.

This transformation reduces the dimensionality and allows for efficient comparison using integer vectors instead of full sets.

## 2.2 Locality Sensitive Hashing (LSH)

To avoid computing approximate similarities for all pairs of reviews, we adopted **Locality Sensitive Hashing (LSH)**, a technique designed to efficiently identify similar pairs with high probability.

LSH works by splitting each MinHash signature into $b$ bands of $r$ rows each. The banding method divides each group into separate buckets. Only pairs within the same bucket across all bands are considered for similarity checking. This banding strategy is rooted in the "S-curve" probability function:

$$\text{Probability(match)} = 1 - (1 - s^r)^b$$

where $s$ is the true Jaccard similarity between two items.

This probability curve exhibits a threshold-like behavior: below a certain similarity, the probability of being detected is near zero; above it, it rapidly approaches one.

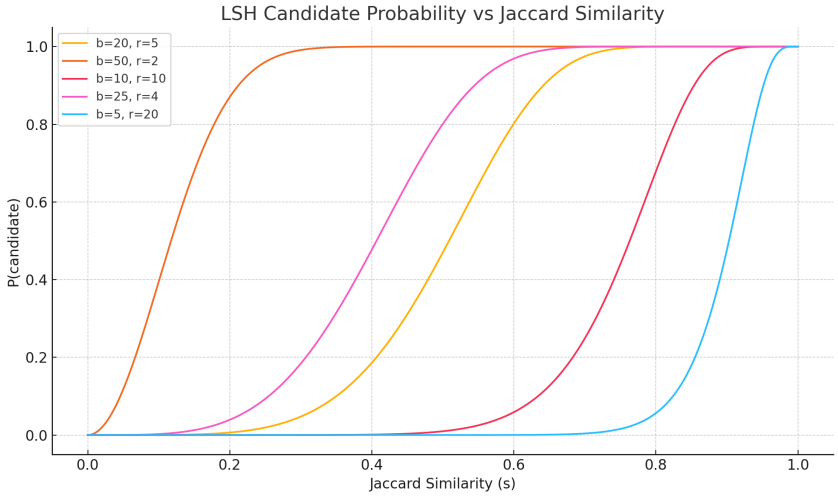


Figure 2.1: LSH Candidate Probability vs. Jaccard Similarity (with $r = 5$, $b = 20$).

Figure 2.1 shows the threshold behavior for $r = 5$ and $b = 20$. The curve demonstrates that pairs with similarity $s < 0.6$ are unlikely to be matched, whereas pairs with $s > 0.8$ are detected with high probability. This justifies our choice of threshold at 0.8 and the corresponding $(r, b)$ parameters.

This setting provides a good balance between reducing false positives and preserving recall.

## 2.3 Prefix Filtering for Early Pruning

To further reduce the number of comparisons after LSH, we applied a prefix-based filtering method, often referred to as **prefix filtering** or $m$**-prefix pruning**.

This technique leverages the observation that if two MinHash signatures do not match in their first $m$ entries, it is improbable that they will have a high Jaccard similarity. Therefore, pairs whose signatures differ in the first $m$ rows are skipped before computing full similarity.

This optimization is discussed in the course textbook [1], and offers a practical speed-up by avoiding expensive Jaccard estimations for unlikely pairs.

In our implementation, we chose $m = 5$, which balances filtering aggressiveness and recall. This approach enabled the early elimination of numerous candidate pairs, particularly those exhibiting spurious LSH collisions.

## 2.4 Summary

The proposed methodology integrates several algorithmic components to detect similar book reviews in a large corpus:

- **Shingling**: Reviews are converted into sets of 5-word shingles, preserving local semantic context while reducing dimensionality.
- **MinHashing**: Each shingle set is encoded into a compact signature vector using 100 randomized hash functions, allowing fast estimation of Jaccard similarity.
- **Locality Sensitive Hashing (LSH)**: The signatures are divided into 20 bands of 5 rows each. Items that hash to the same bucket in at least one band are selected as candidates, drastically reducing pairwise comparisons.
- **Prefix Filtering**: An early pruning technique is applied to skip pairs that differ in the first $m$ entries of their signatures (with $m = 5$), improving performance without sacrificing accuracy.

This pipeline is rooted in well-established theory from large-scale data mining and has proven effective in practice for real-world datasets. It achieves a balance between precision and scalability, enabling the efficient discovery of duplicate or near-duplicate reviews without exhaustive comparison.

# Chapter 3

# Implementation

This chapter describes the technical structure of the implementation, the tools and libraries used, and the modular design of the processing pipeline.

## 3.1 Tools Used

All components of this project were implemented and tested entirely within **Google Colab**, leveraging its free and scalable cloud environment. The Colab instance used Python version 3.10, which is currently the default runtime provided.

We relied on minimal external libraries to keep the solution lightweight and portable. The primary libraries used were:

- **Pandas**: Used for data manipulation and tabular operations. Its DataFrame abstraction enabled efficient handling of the dataset and intermediate results. The code was intentionally written in a modular fashion, allowing for future replacement with distributed frameworks such as `dask`, `pyspark`, or `cuDF` for large-scale deployments.
- **Matplotlib**: Employed for visualizations, including similarity thresholds and LSH behavior analysis.
- **NLTK**: Used solely for English stopword removal during preprocessing.

The codebase was written in a modular fashion, with each major processing step encapsulated in standalone functions. This design choice not only improves readability and maintainability but also makes it easy to adapt the implementation to distributed or GPU-accelerated frameworks in future work.

No third-party MinHash or LSH libraries (such as `datasketch`) were used. Instead, all algorithmic components—including hashing, signature computation, LSH banding, and prefix filtering—were implemented **from scratch**.

## 3.2 Pipeline Overview

The overall structure of the implementation was organized into a modular pipeline consisting of the following steps:

1. **Data Loading and Sampling**: A random subsample of 100,000 reviews was extracted from the original dataset using `pandas`. Only the `Id` and `review/text` fields were retained for further analysis. The data loading function was designed with a `full_size` parameter that allows switching between subsampling and full dataset processing, enabling flexibility for different experimental scenarios.

2. **Preprocessing and Shingling**: The process involved converting each review to lowercase, eliminating stopwords, and segmenting it into contiguous five-word segments. This produced a set of normalized features per document. Reviews too short to generate any shingles were automatically excluded.

3. **MinHash Signature Generation**: For each review, a signature vector of size 100 was computed using a family of randomly generated hash functions. Each value in the signature corresponds to the minimum hash value over all shingles.

4. **Locality Sensitive Hashing (LSH)**: The signature matrix was divided into 20 bands of 5 rows each. Reviews sharing identical bands were placed into the same hash bucket, identifying candidate pairs.

5. **Prefix Filtering and Similarity Computation**: Candidate pairs were filtered using a 5-value prefix match to avoid unnecessary computations. The Jaccard similarity between the MinHash signatures was then approximated, and only pairs with similarity above 0.8 were retained.

6. **Output and Result Formatting**: The final similar pairs were stored and optionally displayed alongside review fragments. Furthermore, the results were consolidated into a `DataFrame` for subsequent analysis or export.

This pipeline ensures both scalability and clarity, making it easy to experiment with different components (e.g., shingle size, number of hash functions, band configuration).

## 3.3 Key Functions

The core logic of our implementation relies on modular and reusable functions, each tailored to a specific part of the pipeline. This modular design ensures the code is readable, testable, and extensible. For instance, it could be adapted to distributed frameworks like Dask or PySpark with minimal changes. Below are some of the most important functions used in the project:

Listing 3.1: Signature Generation Function

```
def compute_signature(shingle_set, hash_functions):
```

```
    signature = []
    for h in hash_functions:
        min_val = float('inf')
        for s in shingle_set:
            val = h(hash(s))
            if val < min_val:
                min_val = val
        signature.append(min_val)
    return signature
```

This function generates the MinHash signature for a review's shingle set. Instead of relying on external libraries or cryptographic hashing, we use Python's built-in `hash()` function combined with custom hash function generators. This allows full transparency and control over the hashing process.

Listing 3.2: LSH Candidate Generation

```
def lsh_create_candidates(df, bands, rows):
    assert bands * rows == len(df['signature'].iloc[0])
    buckets = [defaultdict(set) for _ in range(bands)]
    candidates = set()
    for index, signature in enumerate(df['signature']):
        for b in range(bands):
            start = b * rows
            end = start + rows
            band = tuple(signature[start:end])
            if band in buckets[b]:
                for other in buckets[b][band]:
                    pair = tuple(sorted((index, other)))
                    candidates.add(pair)
            buckets[b][band].add(index)
    return candidates
```

Here, the MinHash signatures are partitioned into `bands`, and review indices are grouped into hash buckets. Only reviews that share a bucket in at least one band are considered candidate pairs, which greatly reduces the number of necessary comparisons.

Listing 3.3: Prefix Filtering and Similarity Computation

```
def find_similar_pairs(df, candidate_pairs, threshold=0.8, m=5):
    results = []
    for i, j in candidate_pairs:
        sig1, sig2 = df.iloc[i]['signature'], df.iloc[j]['signature']
        if sig1[:m] != sig2[:m]:
            continue
        if len(df.iloc[i]['shingles']) < 1 or len(df.iloc[j]['shingles'
           ]) < 1:
            continue
        sim = approximate_jaccard(sig1, sig2)
        if sim >= threshold:
            results.append((i, j, sim))
    return results
```

This function refines candidate pairs by applying an **`m`-prefix filtering** optimization, skipping expensive Jaccard calculations for unlikely matches. This technique is discussed in the course textbook *Mining of Massive Datasets* by Rajaraman and Ullman, and offers

a practical speed-up for MinHash pipelines.

In addition, the function discards any reviews with fewer than $k$ informative words, as these are unable to form even a single valid shingle. Such cases would produce empty shingle sets and result in MinHash signatures containing only infinite values. By excluding these degenerate inputs from the similarity check, we avoid false positives and improve the overall precision of the similarity detection process.

Each of these functions has been carefully designed to operate independently, allowing us to efficiently scale or adapt the pipeline to new tasks or larger datasets.

## 3.4 Performance Considerations

The design incorporates several techniques to ensure that the pipeline performs efficiently, even as the dataset size increases.

- **MinHash Compression**: Instead of comparing full sets of shingles, which would be computationally expensive, each review is represented by a compact signature vector. This reduces the time complexity of pairwise comparisons from set operations to fixed-length vector comparisons.
- **Locality Sensitive Hashing (LSH)**: By grouping only signatures that match in specific bands, LSH allows us to bypass the majority of non-similar pairs, making the process sub-quadratic instead of brute-force $\mathcal{O}(n^2)$.
- **$m$-Prefix Filtering**: This additional layer of filtering verifies whether the first $m$ values of two signatures match before determining their full Jaccard similarity. It offers a significant speed increase by avoiding unnecessary calculations for dissimilar pairs.
- **Shingle Filtering**: Reviews that fail to produce any valid shingles (due to short length or stopword-only content) are excluded early in the pipeline. This prevents noise and false positives from impacting downstream steps.
- **Batch-Oriented Modularity**: The pipeline is designed to be modular, enabling the processing of reviews in logical batches. For instance, all shingling, hashing, and comparison steps are vectorized or looped in sequence, ensuring memory efficiency and code clarity.

These components work together to ensure the pipeline remains performant and scalable, even when processing tens or hundreds of thousands of reviews.

# Chapter 4

# Scalability Analysis

To empirically validate the scalability of our implementation, we conducted runtime experiments across increasing dataset sizes. The dataset was preloaded into memory, and sampling was applied in each run to eliminate I/O overhead from the measurement.

## Experiment: Sample Size vs Runtime

The pipeline was executed for four sample sizes: 1,000; 5,000; 25,000; and 125,000 reviews. Each run included the stages of shingling, MinHash signature generation, LSH candidate detection, and final similarity filtering. The initial data loading time was excluded to isolate pure processing performance. Additionally, reviews with insufficient content to generate valid shingle sets (i.e., those with fewer than $k$ informative words) were filtered prior to processing, ensuring clean inputs for signature generation.

Table 4.1: Runtime and Pair Statistics for Varying Sample Sizes

| **Sample Size** | **Runtime (s)** | **Candidates** | **Similar Pairs** |
|---|---|---|---|
| 1,000 | 3.21 | 1 | 1 |
| 5,000 | 19.88 | 10 | 9 |
| 25,000 | 96.29 | 256 | 247 |
| 125,000 | 516.06 | 6,002 | 5,847 |

## Graphical Analysis

The following figure plots the processing time as a function of sample size:

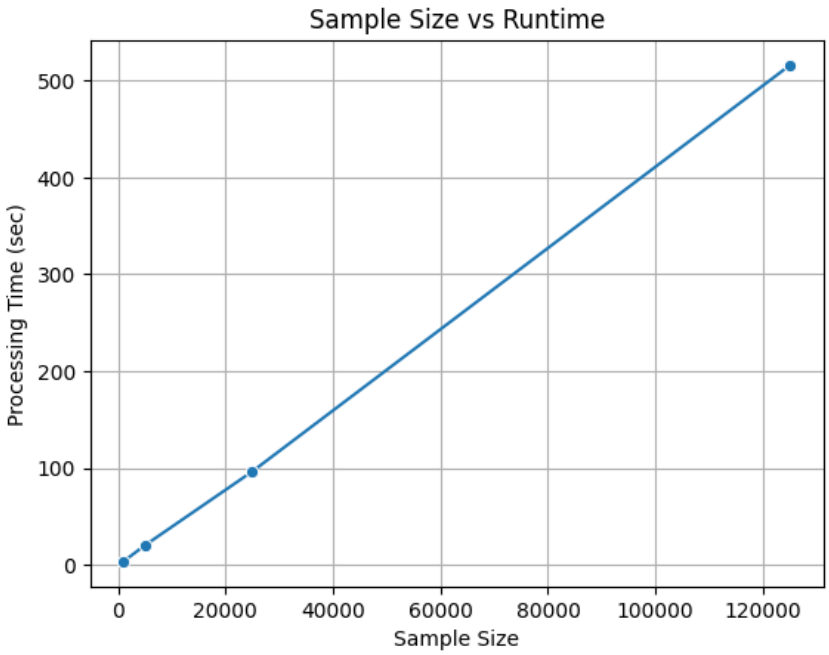


Figure 4.1: Processing Time vs Sample Size (Filtered Shingles)

# Discussion

The graph in Figure 4.1 demonstrates that the total processing time increases approximately linearly with the number of reviews. This aligns with theoretical expectations: shingling and signature generation scale linearly with the dataset size ($\mathcal{O}(n \cdot k)$), while LSH reduces the need for exhaustive pairwise comparisons.

The number of detected candidates and final similar pairs also rises with the sample size, reflecting the natural increase in potential review redundancy as the corpus grows. It is important to note that the pipeline is designed to avoid false candidate inflation by filtering out invalid or empty shingle sets during the early stages of the process. This ensures that the match rate remains accurate. These results confirm that the pipeline remains scalable and efficient, even when processing large samples of over 100,000 reviews.

# Chapter 5

# Experiments and Observations

## 5.1 Experimental Objectives and Setup

This chapter evaluates the performance and behavior of the proposed similarity detection pipeline under different parameter configurations. Although the methodology is designed for scalability and approximate matching, its effectiveness depends on the design parameters. These parameters influence the trade-off between precision, efficiency, and runtime.

To evaluate this, we ran two sets of experiments.

1. **LSH Band and Row Configuration:** Different combinations of bands ($b$) and rows ($r$) were tested, keeping $b \cdot r = 100$ constant to maintain the signature length. This shows how varying LSH sensitivity affects the number of candidates retrieved and the quality of matches.
2. **Shingle Length Variation:** We explored the effect of changing the number of words per shingle ($k$). Since shingles are the fundamental features of each review, choosing a different value of $k$ directly affects both recall and precision. Smaller values of $k$ typically produce more overlapping content across reviews (more matches), while larger values yield more discriminative features.

All experiments were carried out on a fixed random sample of **100,000 reviews**, using:

- 100 MinHash functions,
- Similarity threshold: 0.8,
- Prefix filtering with $m = 5$.

The following metrics were tracked:

- **Candidate pairs:** Number of pairs retrieved via LSH bucketing.

- **Similar pairs:** Number of pairs exceeding the similarity threshold.
- **Execution time:** Total time for shingling, signature computation, candidate generation, and filtering.

All tests were executed in Google Colab with Python 3.10. Matplotlib was used to generate visualizations and interpret the effects of parameter tuning.

## 5.2 Effect of LSH Band and Row Configuration

This experiment examined the impact of varying the number of bands ($b$) and rows ($r$) in the LSH scheme on three performance metrics: the number of candidate pairs, the number of similar pairs (Jaccard similarity $\geq 0.8$), and execution time. The product $b \cdot r = 100$ was kept constant to match the signature length.

### Configurations Tested

- $b = 10$, $r = 10$
- $b = 20$, $r = 5$
- $b = 25$, $r = 4$
- $b = 50$, $r = 2$

All experiments were performed on a fixed sample of 100,000 reviews with 5-word shingles and 100 hash functions. The same review sample and hash function family were reused across configurations to ensure fairness.

# Results

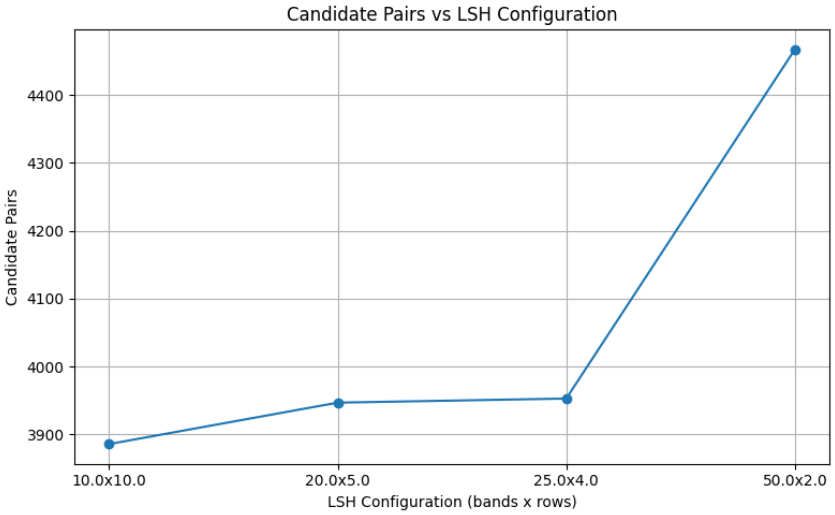


Figure 5.1: Candidate Pairs vs LSH Configuration

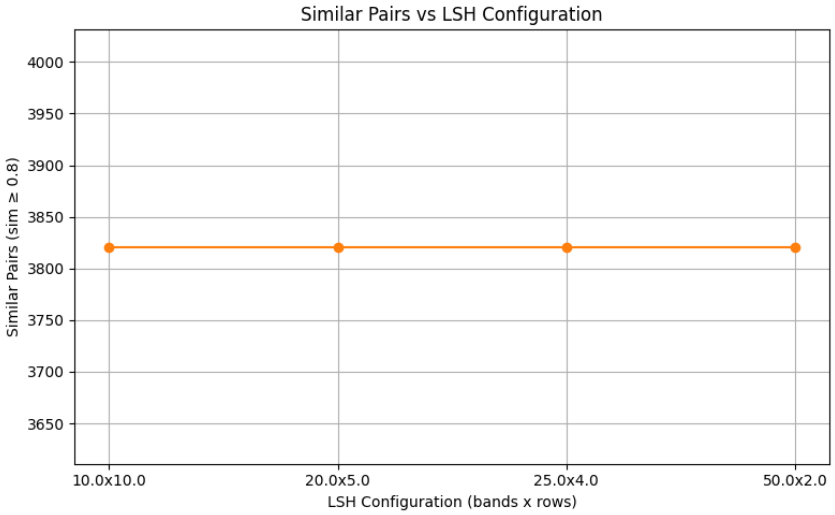


Figure 5.2: Similar Pairs vs LSH Configuration

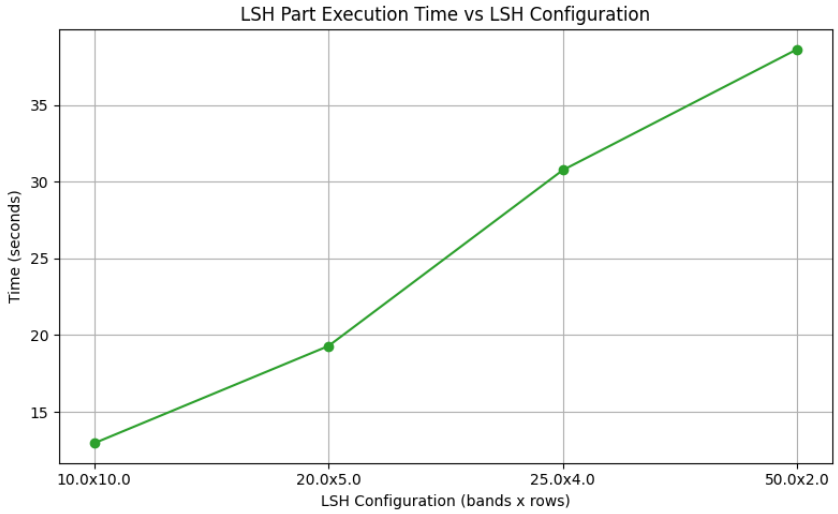


Figure 5.3: Execution Time vs LSH Configuration

### Observations

- As the number of bands increases, the number of candidate pairs also increases—rising from 3,886 ($10 \times 10$) to 4,467 ($50 \times 2$). This is consistent with the S-curve behavior of LSH, where higher band counts lead to a more permissive candidate selection strategy.
- Interestingly, the number of similar pairs (3,821) remained the same across all configurations. This indicates that the review overlap and threshold were strong enough to be captured at all tested sensitivities.
- Execution time increased significantly with more bands and fewer rows per band: from 11.4 seconds ($10 \times 10$) to 37.3 seconds ($50 \times 2$). This is due to an increase in the number of candidate pairs that require similarity evaluation.

These results are consistent with the theoretical expectations of the LSH S-curve (Figure 2.1): as the number of bands increases, the probability of selecting a candidate increases even for moderately similar reviews. In this dataset, the overlapping content among reviews was sufficient to yield stable similar pair detection across configurations, but the efficiency and candidate volume varied as expected.

## 5.3 Effect of Shingle Length ($k$)

In this experiment, we analyzed the effects of changing the shingle length $k$ on the number of candidate pairs, final similar pairs, and overall execution time. The shingle length $k$ determines how many consecutive words are grouped into a unit during feature extraction. Smaller $k$ values may lead to excessive overlap, increasing false positives, while larger values may miss meaningful similarity, increasing false negatives.

## Configurations Tested

- $k = 3$
- $k = 5$
- $k = 7$

All experiments used a sample of 100,000 reviews, with 100 hash functions and an LSH configuration of $b = 20$, $r = 5$. Prefix filtering was enabled with $m = 5$ to accelerate comparison by skipping pairs with dissimilar early signature segments. To ensure all signatures were meaningful, shingles were preprocessed to remove those unable to produce valid shingles.

## Results

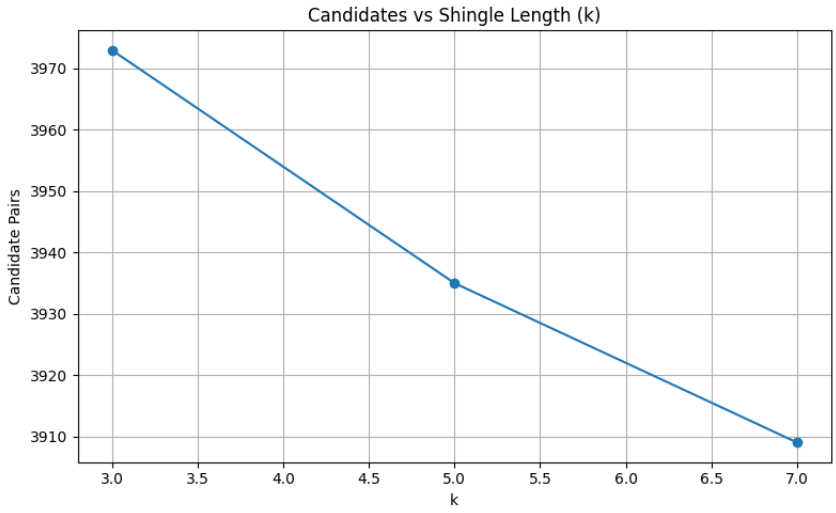


Figure 5.4: Candidate Pairs vs Shingle Length

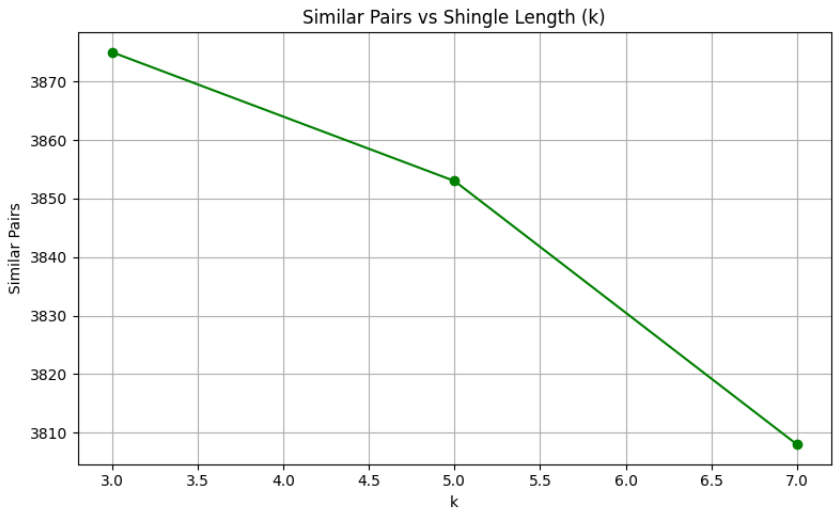


Figure 5.5: Similar Pairs vs Shingle Length

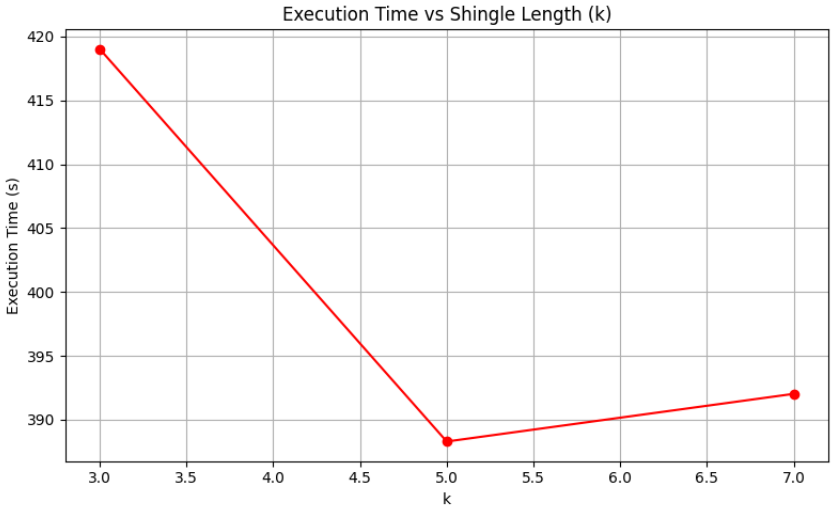


Figure 5.6: Execution Time vs Shingle Length

## Observations

- As the parameter $k$ increases from 3 to 7, the number of candidate pairs decreases slightly (from 3,973 to 3,909), indicating that higher-order shingles reduce overlap and the likelihood of collisions during LSH banding. Shingles were preprocessed to remove those unable to produce valid $k$-grams, ensuring that all signatures were meaningful.
- Similarly, the number of actual similar pairs drops from 3875 to 3808, which is expected since larger $k$ values generate more specific features, narrowing the match potential.
- Execution time slightly fluctuated across $k$ values, with a dip at $k = 5$ and a rise at $k = 7$. These variations are relatively small and likely influenced by both the number of candidates and runtime dynamics of the Google Colab environment.

In summary, while the differences are subtle, $k = 5$ appears to offer a strong balance between sensitivity and performance. It produces a competitive number of similar pairs without unnecessary inflation.

# Chapter 6

# Conclusion

This project successfully demonstrates an efficient and scalable approach for detecting similar book reviews using MinHash and Locality Sensitive Hashing (LSH). The system efficiently reduced high-dimensional review data into compact representations through modular implementation and targeted preprocessing.

Experiments confirmed that the pipeline scales linearly with input size and responds predictably to changes in parameters such as band configuration and shingle length. The quality of candidate selection and execution time remained consistent across various settings, validating the theoretical performance of LSH.

In conclusion, the project delivers a reusable and well-structured solution for approximate similarity detection in large textual datasets. It also combines theoretical soundness with practical performance.

# Chapter 7

# References

[1] J. Leskovec, A. Rajaraman, and J. Ullman, *Mining of Massive Datasets*, 2nd ed., Cambridge University Press, 2014. Available at: http://www.mmds.org

# Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. No generative AI tool has been used to write the code or the report content.