Algorithms for Massive Data
Project 1: Finding Similar Items

Constantin Adrian Tanase, 46954A DSE

July 7, 2025

1 Introduction

The objective of this project is to implement a detector for pairs of similar book reviews using the Amazon Books Reviews dataset. The system must be able to identify reviews with similar content through scalable algorithmic techniques suitable for massive data analysis.

The code was developed in Python 3 and optimized for execution in the Google Colab environment. To ensure scalability, global variables have been used that allow disabling sampling and applying the algorithm to the entire dataset when greater computational power is available. The implemented approach utilizes Jaccard similarity with MinHash and LSH techniques to achieve efficient identification of similar pairs.

2 The Dataset

The "Amazon Books Reviews" dataset is available on Kaggle. It is a 2.8 GB CSV file containing 3 million reviews distributed across 10 columns. For this analysis, we exclusively consider the review/text column which contains the text of the reviews.

The original dataset contains 3,000,000 total reviews, of which 2,999,992 actually contain text after removing missing values. A preliminary check reveals the presence of exact duplicates, which are reduced to 2,062,648 unique reviews.

Due to computational limitations in the free Colab environment, a sample of 10,000 reviews was used, selected with random state=42 to ensure reproducibility of results.

3 Data Preprocessing

A tokenization-based approach was implemented using NLTK libraries, which transforms each review into a set of processed tokens.

The process begins with text normalization through lowercase conversion and punctuation removal using str.maketrans. Subsequently, tokenization is applied using NLTK's word_tokenize, which properly handles word separation even in the presence of contractions and special forms.

Each token is then processed by applying lemmatization to unify different grammatical forms of words. English stopwords are removed and words with length less than 3

characters are filtered out, as they generally do not contribute to semantic content. The final result is an ordered set of unique tokens for each review.

The preprocessing includes quality checks to maintain only meaningful reviews. After tokenization, reviews that become identical at the token level are eliminated, even if they were different in the original text. Finally, only reviews with at least 5 unique tokens are kept, ensuring sufficient content for similarity analysis.

After complete preprocessing, the final dataset contains 9,983 unique reviews with meaningful tokens, ready for similarity analysis.

4 Jaccard Similarity

Jaccard Similarity is a widely used metric for measuring similarity between two sets. It is defined as the ratio between the intersection of the sets and their union, with values ranging from 0 (no common elements) to 1 (identical sets).

In textual analysis, it is very effective when the focus is on the presence or absence of terms rather than their frequency. This helps us identify reviews that share similar content, regardless of how many times each word is repeated.

5 MinHash and Locality Sensitive Hashing

MinHash and LSH represent essential techniques for efficient identification of similar elements in large datasets. The former allows estimation of Jaccard similarity between two documents by comparing compact signatures generated through hash functions applied to token sets, greatly reducing computational complexity compared to direct comparisons.

MinHash works by creating a signature for each token set through the application of multiple hash functions. Each function is applied to all tokens in the set, and the minimum hash value is maintained. The probability that two sets have the same minimum hash value for a random function is exactly equal to their Jaccard similarity.

LSH further optimizes the process by grouping signatures into buckets so that similar elements have a high probability of ending up in the same bucket. This allows rapid retrieval of similar candidates without exhaustive checks on all possible pairs.

6 Implementation

6.1 Configuration and Data Loading

The first phase involves system configuration and dataset loading. The USE_SAMPLE parameter allows choosing whether to use the entire dataset or a sample. When set to True, the system uses only SAMPLE_SIZE reviews (10,000 in our case), while when set to False it processes the entire dataset of 3 million reviews.

Dataset loading occurs through Kaggle APIs, ensuring complete reproducibility. The system automatically downloads the compressed file, extracts it, and loads the main CSV. Immediately after loading, preliminary cleaning operations are applied: first, reviews with missing review/text fields are removed using dropna, then exact duplicates are eliminated with drop_duplicates. This latter operation is important because the original dataset contains many duplicates that could skew similarity results.

Sampling, when activated, uses random state=42 to ensure experiments are reproducible. The sample size is limited by the min function to avoid errors if a sample larger than the available dataset is requested.

6.2 Preprocessing and Tokenization

The second phase handles textual preprocessing using NLTK. The code begins by downloading necessary NLTK resources. Resources include the tokenizer, English stopwords, lemmatizer, and support models.

The safe_tokenize function implements a fallback mechanism: it first tries to use NLTK's tokenizer with word_tokenize, but if this fails (for example due to special characters) it uses a simple split on spaces. This ensures the system never blocks, even with problematic texts.

The main part of preprocessing is the preprocess_text function which transforms each review into an ordered tuple of unique tokens. The process starts by checking that the input is actually a string, returning an empty tuple otherwise. Subsequently, it converts the text to lowercase to normalize capitalization differences and removes all punctuation using str.maketrans. Tokenization is performed with the safe function defined previously.

The final phase of preprocessing applies NLTK's lemmatizer to unify different grammatical forms (for example, "running" and "ran" both become "run"). Standard English stopwords and words that are too short (less than 3 characters) are filtered out, as they generally do not contribute to semantic content. The result is sorted and deduplicated with set to create a set of unique tokens, then converted to a tuple to make it hashable.

After applying preprocessing to all reviews, the code removes a second type of duplicates: reviews that, while being different in the original text, become identical after tokenization. This can happen when two reviews differ only in punctuation or stopwords. Finally, only reviews with at least 5 unique tokens are kept to ensure sufficient content for similarity analysis.

6.3 Signature Creation and Index Construction

The third phase implements MinHash signature creation and LSH index construction. The create_minhash function creates a compact signature for each token set. It initializes a MinHash object with the specified number of permutations (128 in our case), then iterates over all tokens in the set. Each token is converted to bytes with encode('utf8') before being passed to the hashing algorithm, which internally applies hash functions and maintains minimum values.

Signature creation occurs in a loop that processes all filtered reviews, using tqdm to show a progress bar. This is useful because with large datasets the operation can take several minutes. Each signature is added to the minhashes list which maintains the order corresponding to the reviews dataframe.

Index construction uses the MinHashLSH class from the datasketch library. The threshold=0.5 parameter determines the minimum estimated similarity threshold for a pair to be considered a candidate. A lower threshold generates more candidates but requires more subsequent calculations, while a higher threshold is more selective but risks missing similar pairs. The value 0.5 represents a compromise that captures enough candidates without overloading the subsequent phase.

Signature insertion into the index occurs through a second loop that associates each signature with a string identifier corresponding to the position in the dataframe. The LSH index internally builds an optimized data structure that allows rapid retrieval of all signatures similar to a given query.

6.4 Similar Pair Identification

The final phase handles similar pair identification through a two-level strategy. The jaccard_similarity function implements exact Jaccard similarity calculation by converting token tuples to Python sets and calculating the ratio between intersection size and union size. Handling the union == 0 case prevents division by zero errors.

The main loop iterates over all previously created signatures. For each one, the index is queried with lsh.query which rapidly returns all identifiers of signatures considered similar according to the set threshold. This avoids quadratic calculation of all possible pairs, drastically reducing computational complexity.

For each candidate pair returned by the index, the code verifies that indices are in ascending order (j ¿ i) to avoid duplicates and self-comparison. Subsequently, it calculates exact Jaccard similarity using original tokens from the dataframe. Only pairs that have similarity greater than 0 are added to results. This means we include all candidate pairs that have at least one token in common.

Results are sorted in descending order of similarity and the top 20 pairs are shown. For each pair, the code extracts the original review text and truncates it to 150 characters for display, maintaining output readability without taking up too much space.

7 Experiment Description

The experiment was conducted in the Google Colab environment using available free resources. The setup was designed to test the effectiveness of the implemented approach while maintaining reasonable execution times, given the limited environment.

The experiment uses a sample of 10,000 reviews randomly extracted from the complete dataset with random state=42 to ensure reproducibility. Parameters were configured with 128 hash functions for MinHash and an LSH threshold of 0.5. No minimum Jaccard similarity threshold was applied in the final filtering, allowing capture of even pairs with low similarity.

The methodology consists of sequentially applying all pipeline phases, monitoring intermediate results. The effectiveness of preprocessing is first measured by counting how many reviews are maintained after each filter. Subsequently, indexing efficiency is evaluated by measuring how many candidates are generated compared to the total number of possible pairs. Finally, the found pairs are qualitatively analyzed to verify that they actually represent similar content.

The main metrics used include the number of successfully processed reviews, the number of candidates generated by the index, execution time of different phases, and the distribution of final Jaccard similarities. To evaluate scalability, the ratio between examined candidates and total theoretical pairs is calculated, which provides an indication of the approach's efficiency. Qualitative analysis of results focuses on verifying that high-similarity pairs actually share relevant semantic content rather than being simple duplicates.

8 Results

System execution on a sample of 10,000 reviews produced results that demonstrate the effectiveness of the implemented approach. After complete preprocessing, 9,983 unique reviews were maintained, confirming the quality of the data cleaning process.

The system identified 269 candidate pairs through LSH, which represents a drastic reduction compared to approximately 50 million possible pairs. Of these, 2 pairs showed a Jaccard similarity of 0.5, 5 pairs reached values between 0.4 and 0.5, and 7 pairs showed similarity between 0.3 and 0.4. The complete distribution shows 115 pairs in the 0.1-0.2 range and 106 pairs with similarity below 0.1.

The qualitative results are interesting. The top five pairs with highest similarity show significant semantic patterns:

```
Rank 1: Jaccard Similarity = 0.5000
Review 4719: Absolutely wonderful series of books. I can't
wait to read the next one and then I start all over again!
Review 4942: I really got into this book and can't wait for
the next one. It was a wonderful story from start to finish.
------------------------------------------------------------

Rank 2: Jaccard Similarity = 0.5000
Review 20: I really enjoyed this book; it's a must read
Review 3357: My son is 7 and I read this book aloud to him.
He really enjoyed the adventure!Great read!
------------------------------------------------------------

Rank 3: Jaccard Similarity = 0.4545
Review 5546: Good book. great its free. It took me no time
at all to get hooked. This is highly recommended. Great!
Review 7188: It took some time for me to get into this book,
but when I did...it was GREAT!!
------------------------------------------------------------

Rank 4: Jaccard Similarity = 0.4231
Review 3953: I have been reading Danielle Steel for many
years and I truly enjoyed this book. It was a wonderful
love story. I read the reviews and decided to read...
Review 9613: I found this book to be a wonderful love story.
I loved it! I had read some of the reviews, but decided to
read it for myself and boy was I glad I did...
------------------------------------------------------------

Rank 5: Jaccard Similarity = 0.4118
Review 914: This book series by Lynn Austin was recommended
to me by a friend. Once I've started reading these books
it so hard to put them down. Each book is bet...
Review 9793: This was a very good book hard to put down once
I started reading it.. Excellent reading. I highly recommend it
------------------------------------------------------------
```

Performance analysis shows that indexing generated an average of 1.1 candidates per query, a very efficient number that demonstrates the selectivity of LSH filtering. The final similarity range from 0.033 to 0.5 confirms that the system identifies a broad spectrum of similarities, from the weakest to the strongest.

9 Scalability and Performance

The implementation was designed to be scalable through the use of appropriate algorithmic techniques and design choices that facilitate expansion to larger datasets. The use of MinHash drastically reduces the memory space required to represent each review, moving from variable-size token sets to fixed-size signatures.

LSH provides additional advantages in terms of execution time, avoiding Jaccard similarity calculation on all possible pairs. With 10,000 reviews, the number of direct comparisons would have been approximately 50 million, while indexing allowed limiting calculations to only 269 candidates, representing only 0.0005% of possible pairs.

Implementation choices include the use of configurable parameters that can be optimized for datasets of different sizes. The index threshold of 0.5 represents a good compromise between precision and recall, while the absence of filters on minimum similarity allows complete exploratory analysis.

The token-based approach rather than character shingles proved more effective for this type of analysis, providing semantically more meaningful results and reducing noise due to minor orthographic variations.

10 Conclusions

This project successfully implemented a system for identifying similar reviews in the Amazon Books Reviews dataset using Jaccard similarity, MinHash, and LSH techniques. The developed implementation demonstrated the ability to efficiently process a sample of 10,000 reviews, identifying 269 similar pairs with various degrees of similarity.

The approach based on tokenization and preprocessing with NLTK proved adequate for this type of textual analysis. The implemented pipeline - which combines data cleaning, MinHash signature creation, and LSH indexing - reduced the number of necessary comparisons from 50 million to less than 300, demonstrating the effectiveness of approximate nearest neighbor search techniques.

The obtained results show that the system is capable of identifying different types of textual similarity. The identified pairs actually present semantically correlated content.

The code was structured to be easily scalable through the use of global configuration variables. The implementation can be applied to the entire dataset of 3 million reviews by simply modifying the USE_SAMPLE flag, although this would require greater computational resources and possible memory management optimizations.

11 Implemented Code

11.1 Cell 1: Setup and Data Loading

```
USE_SAMPLE = True
SAMPLE_SIZE = 10000
```

```
3
4  import os
5  import pandas as pd
6  import re
7  from datasketch import MinHash, MinHashLSH
8  from tqdm import tqdm
9
10 # Kaggle API setup
11 os.environ['KAGGLE_USERNAME'] = "xxxxxx"
12 os.environ['KAGGLE_KEY'] = "xxxxxx"
13
14 # Download dataset
15 print("Downloading Amazon Books Reviews dataset...")
16 !kaggle datasets download -d mohamedbakhet/amazon-books-reviews
17 !unzip -q "*.zip" -d /content/
18
19 # Load data
20 print("Loading data...")
21 df = pd.read_csv('Books_rating.csv')
22 print(f"Total reviews: {len(df)}")
23
24 # Remove missing reviews
25 df = df.dropna(subset=['review/text'])
26 print(f"Reviews after removing missing: {len(df)}")
27
28 # Remove exact duplicates
29 df = df.drop_duplicates(subset=['review/text'], keep='first')
30 print(f"Reviews after removing exact duplicates: {len(df)}")
31
32 # Sample data if specified
33 if USE_SAMPLE:
34     df = df.sample(n=min(SAMPLE_SIZE, len(df)), random_state=42)
35 print(f"Using sample of {len(df)} reviews")
```

11.2 Cell 2: Text Preprocessing

```
1  import nltk
2  from nltk.tokenize import word_tokenize
3  from nltk.corpus import stopwords
4  from nltk.stem import WordNetLemmatizer
5  import string
6
7  # Download NLTK resources
8  nltk.download('punkt', quiet=True)
9  nltk.download('stopwords', quiet=True)
10 nltk.download('wordnet', quiet=True)
11 nltk.download('omw-1.4', quiet=True)
12
13 def safe_tokenize(text):
14     try:
15         return word_tokenize(text)
16     except:
17         return text.split()
18
19 lemmatizer = WordNetLemmatizer()
20 stop_words = set(stopwords.words('english'))
```

```
translator = str.maketrans('', '', string.punctuation)

def preprocess_text(text):
    if not isinstance(text, str):
        return tuple()
    text = text.lower().translate(translator)
    tokens = safe_tokenize(text)
    return tuple(sorted(set(lemmatizer.lemmatize(word) for word in
       tokens
                            if word not in stop_words and len(word) > 2)))

print("Preprocessing reviews...")
df['tokens'] = df['review/text'].apply(preprocess_text)
df = df.drop_duplicates(subset=['tokens'], keep='first')
df = df[df['tokens'].apply(len) >= 5].reset_index(drop=True)
print(f"Final reviews: {len(df)}")
```

11.3 Cell 3: MinHashing and LSH

```
def create_minhash(tokens, num_perm=128):
    """Create MinHash signature for a tuple of tokens"""
    minhash = MinHash(num_perm=num_perm)
    for token in tokens:
        minhash.update(token.encode('utf8'))
    return minhash

print("Creating MinHash signatures...")
minhashes = []
for tokens in tqdm(df['tokens'], desc="Computing MinHashes"):
    minhash = create_minhash(tokens, 128)
    minhashes.append(minhash)

print("Building LSH index...")
lsh = MinHashLSH(threshold=0.5, num_perm=128)
for i, minhash in enumerate(tqdm(minhashes, desc="Building LSH")):
    lsh.insert(str(i), minhash)
```

11.4 Cell 4: Similar Pair Identification

```
def jaccard_similarity(tuple1, tuple2):
    """Calculate exact Jaccard similarity between two tuples"""
    set1 = set(tuple1)
    set2 = set(tuple2)
    intersection = len(set1.intersection(set2))
    union = len(set1.union(set2))
    return intersection / union if union > 0 else 0.0

print("Finding similar pairs...")
similar_pairs = []

for i, minhash in enumerate(tqdm(minhashes, desc="Finding candidates")):
    candidates = lsh.query(minhash)
    for candidate_str in candidates:
        j = int(candidate_str)
        if j > i:
```

```
            similarity = jaccard_similarity(df.iloc[i]['tokens'],
                                            df.iloc[j]['tokens'])
            if similarity > 0.0:
                similar_pairs.append((similarity, i, j))

# Sort by similarity and get top pairs
similar_pairs.sort(reverse=True)
top_similar_pairs = similar_pairs[:TOP_PAIRS]

print(f"\nTop 20 Most Similar Review Pairs:")
print("=" * 80)

for rank, (similarity, i, j) in enumerate(top_similar_pairs, 1):
    review1 = df.iloc[i]['review/text']
    review2 = df.iloc[j]['review/text']

    review1_display = review1[:150] + "..." if len(review1) > 150 else
        review1
    review2_display = review2[:150] + "..." if len(review2) > 150 else
        review2

    print(f"\nRank {rank}: Jaccard Similarity = {similarity:.4f}")
    print(f"Review {i}: {review1_display}")
    print(f"Review {j}: {review2_display}")
    print("-" * 80)
```

I/We declare that this material, which I/We now submit for assessment, is entirely my/our 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/our work, and including any code produced using generative AI systems. I/We 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/us or any other person for assessment on this or any other course of study.