Project for "Algorithms for Massive Data"

Ettore Oteri, 34054A DSE

12 June 2025

1 Introduction

The purpose of this project is to implement a detector of pairs of similar book reviews. In this report I will explain the whole process step by step, starting from the raw dataset, to get to the final output. The code is written in Python 3 and optimized for implementation in the free to use Google colab environment, but with very slight changes to the spark configuration parameters and applying the algorithm to the entire dataset, the code is perfectly scalable. To this end, global variables have been inserted that can change the stream input by deactivating the subsampling limits dictated by the low computing power, allowing to optimize the code for the entire dataset with a few clicks. To do this I did not limit myself to parallelizing the processes using the classic libraries, but I wanted to experiment with Spark. So I developed two versions of the same project: one that mainly uses pandas and one entirely configured in Spark. Later I will discuss and compare the results and performances. To initialize an environment with multiple worker nodes would have required a paid service, but I was able to simulate a distributed architecture on a single local machine by dividing the processes on multiple cores and distributing the necessary memory. In this way, while maintaining most of the code and changing some values, the code is truly scalable and usable.

2 About the Dataset

The dataset "Amazon books reviews" is publicly available on Kaggle under the CC0 license. It contains two files but we only consider the one containing the ratings. This csv file is 2.86 GB and contains 3M rows and 10 columns. For this analysis we are ignoring all of the columns except for the one containing the text of the reviews.

3 The Jaccard Similarity

The Jaccard Similarity is a widely used metric for measuring the similarity between two sets. It is defined as the ratio of the intersection of the sets to their

union. The Jaccard Similarity ranges from 0, indicating no common elements, to 1, meaning the sets are identical. In text analysis, the Jaccard Similarity is often used to compare documents by representing them as sets of words, n-grams, or character sequences. It is particularly effective for detecting duplicate content, identifying similar documents, and filtering out irrelevant information in recommendation systems. Compared to other similarity measures, such as cosine similarity, Jaccard is more suitable when the focus is on the presence or absence of terms rather than their frequency. Due to its simple yet effective approach, Jaccard Similarity is commonly used in applications like plagiarism detection, clustering, and search result deduplication.

4 MinHash and Locality sensitive hashing (LSH)

MinHash and LSH are essential techniques for efficiently identifying similar reviews in large datasets. MinHash estimates the Jaccard similarity between two documents by comparing compact signatures generated through hash functions applied to text shingles, significantly reducing computational complexity compared to direct comparisons. LSH further optimizes the process by grouping these signatures into buckets so that similar items have a high probability of collision, enabling rapid retrieval without exhaustive pairwise checks. The combination of these methods provides a proven balance between accuracy and performance, making them ideal for tasks like review deduplication or clustering, where scalability and robustness to minor textual variations are crucial. Using these techniques allowed me to significantly enlarge the subsample of the dataset up to 10 times compared to previous versions of the code. After optimizing the code with minhash the accuracy of the model increased and the execution times decreased dramatically.

5 Code

I start with the Spark version of the code so we can deep dive into the core of the project. I load the dataset directly into the code, using the Kaggle API:

```
import os
os.environ['KAGGLE_USERNAME'] = "ettoreoteri"
os.environ['KAGGLE_KEY'] = "xxxxxxxxxxxxxxxxxxxxxxxx"
!kaggle datasets download -d mohamedbakhet/amazon-
    books-reviews
!unzip -q "*.zip" -d /content/
```

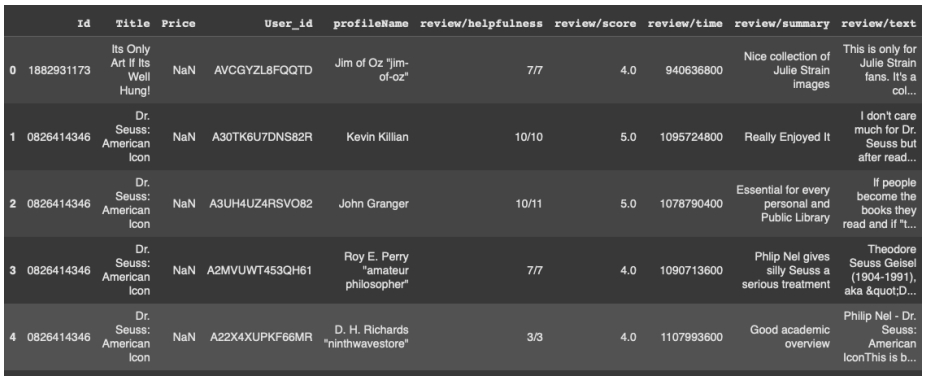
Output:

```
Dataset URL: https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews
License(s): CC0-1.0
```

I import the Pandas library and have the system read the dataset. Then, to make sure it was read correctly, I print the first few lines.

```
import pandas as pd
df1 = pd.read_csv("Books_rating.csv")
df1.head()
```

Output:



	Id	Title	Price	User_id	profileName	review/helpfulness	review/score	review/time	review/summary	review/text
0	1882931173	Its Only Art If Its Well Hung!	NaN	AVCGYZL8FQQTD	Jim of Oz "jim-of-oz"	7/7	4.0	940636800	Nice collection of Julie Strain images	This is only for Julie Strain fans. It's a col...
1	0826414346	Dr. Seuss: American Icon	NaN	A30TK6U7DNS82R	Kevin Killian	10/10	5.0	1095724800	Really Enjoyed It	I don't care much for Dr. Seuss but after read...
2	0826414346	Dr. Seuss: American Icon	NaN	A3UH4UZ4RSVO82	John Granger	10/11	5.0	1078790400	Essential for every personal and Public Library	If people become the books they read and if "t...
3	0826414346	Dr. Seuss: American Icon	NaN	A2MVUWT453QH61	Roy E. Perry "amateur philosopher"	7/7	4.0	1090713600	Phlip Nel gives silly Seuss a serious treatment	Theodore Seuss Geisel (1904-1991), aka "D...
4	0826414346	Dr. Seuss: American Icon	NaN	A22X4XUPKF66MR	D. H. Richards "ninthwavestore"	3/3	4.0	1107993600	Good academic overview	Philip Nel - Dr. Seuss: American IconThis is b...

I also check the size of the dataset to verify that it has been read in its entirety.

```
print(f"This dataset contains {df1.shape[0]} rows and
    {df1.shape[1]} columns.")
```

Output:

```
This dataset contains 3000000 rows and 10 columns.
```

I install spark, since unlike many libraries it is not pre-installed in the Colab environment we are working on.Then I load all the relevant libraries needed for the following steps and also the word lists to be removed during data cleaning.

```
!pip install pyspark nltk
!apt-get install openjdk-8-jdk-headless -qq > /dev/
    null
!pip install datasketch

from pyspark.sql import SparkSession
from pyspark import SparkContext
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import string
from itertools import combinations
import random
from datasketch import MinHash, MinHashLSH

nltk.download('all')
```

I continue by creating the spark environment and assigning the amount of memory to the different processes. After several attempts this seems to be the most balanced configuration that maximizes the computational capabilities

available with the free plan of Google Colab. This chunk of code initializes a Spark session with 8GB memory allocation for drivers/executors and 8 partitions for parallel processing, creating the core distributed computing environment.

```
# Initialize Spark configuration
spark = SparkSession.builder \
    .appName("BookReviewsJaccard") \
    .config("spark.driver.memory", "8g") \
    .config("spark.executor.memory", "8g") \
    .config("spark.sql.shuffle.partitions", "8") \
    .getOrCreate()

sc = spark.sparkContext
```

I subsample the dataset, forced by the limited computing power available, and set a global variable to activate or deactivate the subsampling function, thus using the entire dataset or not.

```
# Subsampling
SAMPLE_SIZE = 10000  # Sample size when subsampling
SEED = 42
USE_FULL_DATA = False  # Set to True to disable
    subsampling
```

I move on to the preprocessing and data cleaning phase to prepare the raw review texts for the next similarity calculation phase. The main function takes a text as input, verifies that it is a valid string, converts it to lowercase, removes punctuation and tokenizes the text. It then applies lemmatization and filters stopwords, keeping only words with more than 2 characters. These processes are fundamental because they focus the analysis on the essence of the text. Lemmatization unifies the different forms of words by bringing them back to a single semantic root, while the removal of stopwords eliminates linguistic noise. Together, they allow us to grasp the real affinities between the contents, going beyond simple superficial coincidences. Without this preparation of the text, the results would risk being distorted by grammatical variants or empty words, compromising the identification of authentic thematic similarities between the reviews. In hindsight, viewing the results and checking the common tokens, I realized that it would have been better to add even more stopwords for an even more truthful analysis, but doing so (given the limited sample of 10000 reviews) the similarity indices would have been too low. Having much more computing power available and therefore being able to use the entire dataset, adding stopwords to the ready list downloaded from the nltk package would result in a higher accuracy of the model. Each review is processed and only those that, after cleaning, contain at least 5 valid tokens are kept. The final result is a RDD of pairs (original text, processed text) ready for further analysis, with a report of the number of reviews actually loaded. The cache() optimizes performance for repeated processing.

```
#Preprocessing with error handling
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
```

```
4
5  def preprocess_text(text):
6      try:
7          if not isinstance(text, str):
8              return []
9          text = text.lower().translate(str.maketrans('',
               '', string.punctuation))
10         tokens = word_tokenize(text)
11         return [lemmatizer.lemmatize(w) for w in
               tokens
12                 if w not in stop_words and len(w) > 2]
13     except Exception:
14         return []
15
16 # Load function to extract only review texts
17 def load_review_texts():
18     # Read the review/text column
19     lines = sc.textFile("Books_rating.csv")
20     header = lines.first()
21
22     # Extract only the review text column
23     review_lines = lines.filter(lambda line: line !=
           header) \
24                         .map(lambda line: line.split('
                               "')[-2] if '"' in line
                               else line.split(',')[-1])
25
26     review_lines = review_lines.filter(lambda x: len(x
           ) > 10).distinct()
27
28     # Sampling logic
29     sampled_reviews = review_lines.collect() if
           USE_FULL_DATA \
30                       else review_lines.takeSample(
                             False, SAMPLE_SIZE, SEED)
31
32     print(f"Number of unique reviews considered for
           sampling: {len(sampled_reviews)}")
33
34     # Process sampled reviews
35     processed_reviews = []
36     for text in sampled_reviews:
37         processed = preprocess_text(text)
38         if len(processed) >= 5:  # Only keep reviews
               with enough tokens
39             processed_reviews.append((text, processed)
                   )
40
41     return sc.parallelize(processed_reviews, numSlices
           =8)
42
43 # Process data
44 reviews_rdd = load_review_texts().cache()
45
46 print(f"Sample size loaded: {reviews_rdd.count()}")
```

Finally, I get to the real core of the analysis: the following code implements

a sophisticated text similarity pipeline that identifies reviews with overlapping content while preserving their distinct characteristics. At its core, it uses MinHash to create probabilistic fingerprints of each review's token set, enabling efficient estimation of Jaccard similarity between documents. The LSH layer then performs approximate nearest neighbor search at scale, clustering reviews that likely share meaningful content overlaps. The final verification step against original tokens ensures semantic relevance isn't lost in the hashing process, and the ranked output allows for analyzing similarity gradients across the review corpus - from near-duplicates to conceptually related but distinct reviews. It selects only the pairs with a similarity greater than zero, takes the 20 most similar ones and displays them in detail. For each pair, it prints the original texts, the similarity score (from 0 to 1) and the precise list of words they have in common, all clearly formatted with dividing lines between one result and the other. This is all done efficiently thanks to Spark, which manages the workload by distributing it across the available resources, and finally correctly frees the memory by closing the session.

```
# Similarity calculation
def get_minhash(tokens, num_perm=128):
    m = MinHash(num_perm=num_perm)
    for token in tokens:
        m.update(token.encode('utf8'))
    return m

# Generate and process pairs
def process_pairs(rdd, num_perm=128, threshold=0.5):
    data = rdd.collect()

    minhashes = []
    for i, (original_text, tokens) in enumerate(data):
        mh = get_minhash(tokens, num_perm)
        minhashes.append((i, original_text, tokens, mh
            ))

    lsh = MinHashLSH(threshold=threshold, num_perm=
        num_perm)
    for i, _, _, mh in minhashes:
        lsh.insert(str(i), mh)

    results = []
    for i, text1, tokens1, mh1 in minhashes:
        for j in lsh.query(mh1):
            if int(j) > i:
                text2, tokens2, mh2 = minhashes[int(j)
                    ][1:4]
                set1, set2 = set(tokens1), set(tokens2
                    )
                intersection = set1 & set2
                union = set1 | set2
                jaccard = len(intersection) / len(
                    union) if union else 0.0
                if jaccard > 0:
                    results.append((text1, text2,
                        jaccard, intersection))
```

```
32
33      results.sort(key=lambda x: -x[2])
34      return results[:20]
35
36  # Get top pairs
37  top_pairs = process_pairs(reviews_rdd)
38
39  # Print results
40  print("\nTOP 20 MOST SIMILAR PAIRS:")
41  for idx, (text1, text2, sim, common_tokens) in
        enumerate(top_pairs, 1):
42      print(f"\n#{idx}: Similarity = {sim:.4f}")
43      print("\nReview 1:")
44      print(text1)
45      print("\nReview 2:")
46      print(text2)
47      print(f"\nCommon tokens ({len(common_tokens)}): {
            common_tokens}")
48      print("="*100)
49
50  spark.stop()
```

Output (even if the program generates the first 20 pairs, I will limit myself to showing the first 5 as they are already explanatory):

```
Sample size loaded: 9773

TOP 20 MOST SIMILAR PAIRS:

#1: Similarity = 0.8163

Review 1:
Fun books for their illustrations and not their writing quality, the Billy and Blaze series

Review 2:
Fun books for their illustrations and not their writing quality, the Billy and Blaze series

Common tokens (40): {'childrens', 'lover', 'dry', 'much', 'larsenreviewsorg', 'book', 'alway
====================================================================================================

#2: Similarity = 0.5556

Review 1:
Wonderful sequel to Ender's Shadow - can't wait for the next 2 books in the series.

Review 2:
,can't wait for the next book in this series.

Common tokens (5): {'next', 'cant', 'series', 'wait', 'book'}
====================================================================================================
```

```
#3: Similarity = 0.4444

Review 1:
I was very impress with this book it was also the best book i've read about wv.

Review 2:
. If you don't have this book you should get it because it's one of the best books I've read

Common tokens (4): {'ive', 'best', 'book', 'read'}
================================================================================

#4: Similarity = 0.3333

Review 1:
 and I look forward to reading more great books by this talented author.

Review 2:
 and look forward to reading it.Coleen from Kent, Wa

Common tokens (3): {'reading', 'forward', 'look'}
================================================================================

#5: Similarity = 0.3158

Review 1:
I have not had time to read this book yet. But I do know that Janette Oke is a fantastic aut

Review 2:
I've seen the movies read the series who knows how many times but reading the first book is

Common tokens (6): {'reading', 'know', 'many', 'time', 'read', 'book'}
================================================================================
```

Now let's move to the other version of the code. Here too I started by loading all the necessary libraries and downloaded the packages containing the stopwords:

```
!pip install datasketch

import string
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import nltk
from itertools import combinations
from datasketch import MinHash, MinHashLSH

# Download the required NLTK resources
```

```
nltk.download('punkt', quiet=True)
nltk.download('stopwords', quiet=True)
nltk.download('wordnet', quiet=True)
nltk.download('omw-1.4', quiet=True)
nltk.download('punkt_tab', quiet=True)
```

The preprocessing process is also similar: here tokenization, lemmatization and removal of stopwords occur in a more linear way, without the typical Spark functions, but beyond the performances the final result is identical.

```

# Tokenization function
def safe_tokenize(text):
    try:
        return word_tokenize(text)
    except:
        return text.split()  # Simple fallback

# Text preprocessing setup
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
translator = str.maketrans('', '', string.punctuation)

def preprocess_text(text):
    if not isinstance(text, str):
        return []

    # Text normalization
    text = text.lower().translate(translator)
    tokens = safe_tokenize(text)
    # Return sorted tuple of unique lemmatized tokens
    return tuple(sorted(set(lemmatizer.lemmatize(word)
         for word in tokens
                     if word not in stop_words and
                         len(word) > 2)))

# Data loading and cleaning
print("Loading and cleaning data...")
df = pd.read_csv("Books_rating.csv")

# Remove exact duplicates
df = df.drop_duplicates(subset=['review/text'], keep='
    first')

# Sample 10,000 reviews and preprocess
df = df.sample(10000, random_state=36)
df['tokens'] = df['review/text'].apply(preprocess_text
    )

# Remove duplicates after preprocessing
df = df.drop_duplicates(subset=['tokens'], keep='first
    ')

# Filter reviews with at least 5 unique tokens
df = df[df['tokens'].apply(len) >= 5]
reviews = df['tokens'].tolist()
```

The similarity calculation is performed the same way as the previous code,

but with minor changes in the code and without distributing the computation with Spark.

```
# Similarity calculation using MinHash + LSH
print("Calculating similarities with MinHash...")

minhashes = []
num_perm = 128
for tokens in reviews:
    m = MinHash(num_perm=num_perm)
    for token in tokens:
        m.update(token.encode('utf8'))
    minhashes.append(m)

lsh = MinHashLSH(threshold=0.5, num_perm=num_perm)
for i, m in enumerate(minhashes):
    lsh.insert(str(i), m)

results = []
for i, mh1 in enumerate(minhashes):
    for j_str in lsh.query(mh1):
        j = int(j_str)
        if j > i:
            set1 = set(reviews[i])
            set2 = set(reviews[j])
            intersection = set1 & set2
            union = set1 | set2
            jaccard = len(intersection) / len(union)
                if union else 0.0
            if jaccard > 0:
                results.append((jaccard, i, j))

# Sort and select top 20 pairs
results.sort(reverse=True, key=lambda x: x[0])
top_20 = results[:20]

# Display results
print("\nTOP 20 MOST SIMILAR PAIRS:")
for rank, (sim, i, j) in enumerate(top_20, 1):
    orig_text_i = df.iloc[i]['review/text'][:100] + ('
        ...' if len(df.iloc[i]['review/text']) > 100
        else '')
    orig_text_j = df.iloc[j]['review/text'][:100] + ('
        ...' if len(df.iloc[j]['review/text']) > 100
        else '')

    print(f"\n#{rank}: Similarity = {sim:.4f}")
    print(f"Review {i}: {orig_text_i}")
    print(f"Review {j}: {orig_text_j}")
    print(f"Common tokens: {set(reviews[i]) & set(
        reviews[j])}")
```

Here I also added a function that also allows you to save the results directly to a csv file if you want

```
# Save results
results_df = pd.DataFrame(top_20, columns=['similarity
    ', 'index1', 'index2'])
```

```
results_df['text1'] = results_df['index1'].apply(
    lambda x: df.iloc[x]['review/text'])
results_df['text2'] = results_df['index2'].apply(
    lambda x: df.iloc[x]['review/text'])
results_df.to_csv('top_pairs_no_duplicates.csv', index
    =False)
```

I show the first 5 results:

```
TOP 20 MOST SIMILAR PAIRS:

#1: Similarity = 0.5000
Review 6795: One of the best books of photojournalism that I've ever seen. Highly recommende
Review 9946: One of the best books I've ever read!.
Common tokens: {'one', 'ever', 'best', 'ive', 'book'}

#2: Similarity = 0.4444
Review 5170: I received this book and it was in perfect condition, it came very quickly.
Review 5546: The book was in perfect condition when I received it. It is an interesting reac
Common tokens: {'received', 'book', 'perfect', 'condition'}

#3: Similarity = 0.4000
Review 6224: this is the best book ever. it is very funny. I have and have read all the book
Review 9946: One of the best books I've ever read!.
Common tokens: {'read', 'ever', 'best', 'book'}

#4: Similarity = 0.3636
Review 8672: Wonderful service. Arrived in 6 days. Book in condition as promised.
Review 9713: The book arrived in a great condition. Also, it arrived in about 2 days time. r
Common tokens: {'arrived', 'book', 'condition', 'day'}

#5: Similarity = 0.3529
Review 8122: This was my 1st Evanovich book ever read and I was hooked, went out and bought
Review 9157: Read this book (and the series) over and over and it gets better each time! Car
Common tokens: {'series', 'enough', 'cant', 'read', 'book', 'get'}
```

6 Conclusions

This project demonstrated how text similarity algorithms can be used to analyze article reviews automatically and effectively. The Jaccard index approach proved to be particularly suitable for identifying pairs of reviews with similar content, highlighting recurring themes and common language between texts. Lemmatization and stopword removal allowed extracting the most significant words, improving the accuracy of comparisons. The analysis revealed that reviews with high similarity scores often share specific terms related to the plot, the author's style or the emotions conveyed. This method could be extended

for example to detect fake reviews based on anomalous similarities or recommend books with similar content. It was not easy to experiment and test such algorithms and tools for big data dealing with limited computational resources, but I'm really satisfied with the result.

6.1 About the Output

Comparing the outputs of the two models does not make sense for several reasons: first of all because they both use the same dataset, the same algorithm and the exact same preprocessing technique; secondly because they cannot use the entire dataset and limit the analysis to the randomized subsample, the result will simply be dictated by the randomness with which some reviews are chosen compared to others and how similar they are to each other. On the contrary, using the entire dataset the two models should return exactly the same result, differing only in runtime. However, limiting myself to commenting on the results obtained by changing the randomness seed several times, both models have found similar similarity indices for the top positions in the ranking, taking into account that this is purely dictated by chance.

6.2 Performances

In terms of performance, the non-Spark model was faster, contrary to expectations. This is because Spark is slower than Pandas with 10,000 reviews because it is designed for large distributed datasets, not small data. Starting the distributed structure (Java processes, scheduler, executor) takes longer than the processing itself on such a small dataset. Pandas, on the other hand, operates directly in the memory of the single computer, without this overhead. Spark's strength - the ability to split and process data in parallel across multiple machines - becomes a liability with small data, because the time spent organizing the distribution outweighs the benefits. This architecture only becomes useful when the data is so large that it does not fit into the memory of a single computer, typically above 50,000-100,000 reviews. In these cases, the ability to process in parallel justifies the initial overhead.

7 Code Sources

In addition to my previous knowledge of python, due to my passion for programming applied to numerous university and personal projects, and the teaching material studied during this course, the main sources from which I extracted and customized the code are:

- `https://github.com/shiivangii/NLP_Tutorial/blob/master/1_Text_preprocessing.ipynb` for the preprocessing part. I voluntarily chose to use only some of the techniques illustrated because, as previously said, an exaggeration in filtering and manipulating the texts would have reduced the similarity indices too much.
- `https://spark.apache.org/docs/latest/index.html` Spark official documentation, of course.
- `https://github.com/TheAlgorithms/Python/blob/master/maths/jaccard_similarity.py` with major changes to fit my code.
- `https://github.com/ekzhu/datasketch` for MinHash and LSH implementation.
- Some Stack Overflow useful past questions to adapt Spark in the right way.

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.