Project for "Algorithms for Massive Data"

Ettore Oteri, 34054A DSE

2nd April 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 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	Philip 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

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

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 = 1000  # 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 1000 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'))

def preprocess_text(text):
    try:
        if not isinstance(text, str):
            return []
        text = text.lower().translate(str.maketrans('', '', string.punctuation))
        tokens = word_tokenize(text)
        return [lemmatizer.lemmatize(w) for w in tokens
                if w not in stop_words and len(w) > 2]
    except Exception:
        return []

# Load function to extract only review texts
def load_review_texts():
    # Read the review/text column
    lines = sc.textFile("Books_rating.csv")
    header = lines.first()

    # Extract only the review text column
    review_lines = lines.filter(lambda line: line != header) \
                        .map(lambda line: line.split('"')[-2] if '"' in line else
```

```
                            line.split(',')[-1])

    # Sampling logic
    sampled_reviews = review_lines.filter(lambda x:
        len(x) > 10).collect() if USE_FULL_DATA else
        review_lines.filter(lambda x: len(x) > 10).
        takeSample(False, SAMPLE_SIZE, SEED)

    # Process sampled reviews
    processed_reviews = []
    for text in sampled_reviews:
        processed = preprocess_text(text)
        if len(processed) >= 5:  # Only keep reviews
            with enough tokens
            processed_reviews.append((text, processed)
                )

    return sc.parallelize(processed_reviews, numSlices
        =8)

# Process data
reviews_rdd = load_review_texts().cache()

print(f"Sample size loaded: {reviews_rdd.count()}")
```

Finally, I get to the real core of the analysis: the following code first calculates the similarity between each possible pair of reviews using the Jaccard index, which measures how much the words of the two reviews overlap. It takes the keywords of each review (already cleaned and lemmatized) and compares how many terms they have in common compared to the total unique terms between the two. Then it organizes the work intelligently: instead of comparing all the reviews to each other in a disorderly way, it divides them into groups and makes comparisons only within each group, saving time and resources. 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 jaccard_similarity(pair):
    set1 = set(pair[0][1])  # processed tokens from
        first review
    set2 = set(pair[1][1])  # processed tokens from
        second review
    intersection = len(set1 & set2)
    union = len(set1 | set2)
    return (pair[0][0], pair[1][0], intersection /
        union if union else 0.0, set1 & set2)

# Generate and process pairs
def process_pairs(rdd):
    return rdd.zipWithIndex() \
```

```
            .map(lambda x: (x[1]//1000, x[0])) \
            .groupByKey() \
            .flatMap(lambda x: [(v1, v2) for v1, v2 in
                combinations(list(x[1]), 2)]) \
            .filter(lambda x: x[0] != x[1]) \
            .map(jaccard_similarity) \
            .filter(lambda x: x[2] > 0)

# Get top pairs
top_pairs = process_pairs(reviews_rdd).takeOrdered(20,
    key=lambda x: -x[2])

# Print results
print("\nTOP 20 MOST SIMILAR PAIRS:")
for idx, (text1, text2, sim, common_tokens) in
    enumerate(top_pairs, 1):
    print(f"\n#{idx}: Similarity = {sim:.4f}")
    print("\nReview 1:")
    print(text1)
    print("\nReview 2:")
    print(text2)
    print(f"\nCommon tokens ({len(common_tokens)}): {
        common_tokens}")
    print("="*100)

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: 969

TOP 20 MOST SIMILAR PAIRS:

#1: Similarity = 0.2609

Review 1:
This was pretty good overall. Nothing particulary ground-
shaking and I've read a lot of these things in other books
too. Still, it was well presented and helpful.Recommended.

Review 2:
It's a lot more fun to read to your kid if the book has humour
and imagination like most of these beginner books do.Not the
best but still pretty good.

Common tokens (6): {'still', 'good', 'lot', 'pretty', 'read',
'book'}
====================================================================

#2: Similarity = 0.2500
```

```
Review 1:
I read Blackout and All Clear b/f this novel. Blackout and All
Clear were great. Doomsday Book is good but not nearly as
interesting as the other two.

Review 2:
I just read it to my brother and he was happy and thought it
was a great book. I finished the book and thought that it was
a pretty good book.

Common tokens (4): {'book', 'great', 'read', 'good'}
================================================================

#3: Similarity = 0.2500

Review 1:
The book arrived within the time range allotted for shipping,
and in great condition. Would not have any problem doing
business with this seller in future.

Review 2:
Guess I didn't review this in good time. Book was in great
condition and seller was prompt in shipping. BTW if you get a
chance to read this, it's worth your while!

Common tokens (6): {'seller', 'time', 'condition', 'book',
'great', 'shipping'}
================================================================

#4: Similarity = 0.2500

Review 1:
This book is absolutely the best of Grisham. And I've read
them all! The ending has a twist like no other !

Review 2:
Much more information in this book than in any other book I've
read on the subject

Common tokens (3): {'book', 'read', 'ive'}
================================================================

#5: Similarity = 0.2353

Review 1:
This book is absolutely the best of Grisham. And I've read
```

```
them all! The ending has a twist like no other !

Review 2:
I just wanna say that this book IS THE BEST OF ALL THE CLAN
NOVELS I'VE READ SO FAR... SIMPLY AMAZING, bye

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

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:

```python
import pandas as pd
import string
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import nltk
from itertools import combinations

# 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.

```python

# 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 1,000 reviews and preprocess
df = df.sample(1000, 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 done in the most explicit form of the Jaccard theorem, using operations between sets and the results of the algorithm are also shown in a very similar way.

```
# Similarity calculation with duplicate checks
print("Calculating similarities...")

top_pairs = []
checked_pairs = set()

for i, j in combinations(range(len(reviews)), 2):
    if reviews[i] == reviews[j]:  # Skip identical
        reviews
        continue

    pair_key = frozenset({i,j})
    if pair_key in checked_pairs:  # Avoid duplicate
        comparisons
        continue

    checked_pairs.add(pair_key)
    set1 = set(reviews[i])
    set2 = set(reviews[j])
    sim = len(set1 & set2) / len(set1 | set2) if (set1
         | set2) else 0
    top_pairs.append((sim, i, j))

# Sort and select top 20 pairs
top_pairs.sort(reverse=True, key=lambda x: x[0])
top_20 = top_pairs[: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 = 0.2857
Review 759: If you are into books about Jesus and his life
then you must get this. I have read it over and over ...
Review 933: I have read the book and realy enjoyed it. I also
have the DVD of it. You realy must get both.
Common tokens: {'must', 'get', 'read', 'book'}

#2: Similarity = 0.2727
Review 118: This was an incredible book I can't wait for the
seaquel
Review 916: I have yet to read the book, but I have heard good
things about it. Can't wait to start it!
Common tokens: {'wait', 'book', 'cant'}

#3: Similarity = 0.2632
Review 168: This book is on the Kumon required reading list.
Wonderful illustrations, amusing, and easy to read....
Review 425: It's not hype, it truly is a great read. Wonderful
story, easy reading. Great book to just kick back...
Common tokens: {'easy', 'wonderful', 'reading', 'read', 'book'}

#4: Similarity = 0.2500
```

```
Review 75: This is an awesome book about grieving for all
ages. I highly recommend it to everyone.
Review 699: I was amazed at the emotion that this book was
able to gather ... I would highly recommend this book...
Common tokens: {'highly', 'everyone', 'recommend', 'book'}

#5: Similarity = 0.2500
Review 389: Jane Eyer is a great book, have read it before,
will surely read it again, one of the great books
Review 869: This book is great!! It has always been one of my
childhood favorties and now I am sharing it with m...
Common tokens: {'great', 'one', 'book'}
```

5 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.

5.1 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.

5.2 Performances

In terms of performance, the non-Spark model was faster, contrary to expectations. This is because Spark is slower than Pandas with 1,000 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.

6 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.
- 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.