Assignment: Spark and LSH
Big Data Analytics
Melih Demirel
1953139

Introduction

This report details the outcomes of an assignment centered on the analysis of Stack Exchange platforms' datasets, specifically focusing on the Cryptography Stack Exchange. The primary objective is to identify near-duplicate posts within this dataset, aiming to recognize similarities such as identical questions or corresponding answers. The assignment allows flexibility to extend the analysis to the Software Engineering dataset if desired. The hardware used for implementation is a Mac M1 with 16GB of RAM.

Data Preparation

To ready the Stack Exchange dataset for analysis, we perform the following steps:

1. **XML Parsing and Extraction:**
   We parse the XML file, extracting essential attributes like Id and Body for each post.

2. **HTML Stripping:**
   Using the Bleach package, we remove HTML code from post bodies, ensuring clean and plain text. The html.unescape function is applied to convert character references to Unicode.

3. **Tokenization and Stopword Removal:**
   Employing the NLTK package, we tokenize post bodies into words and remove common English stopwords, refining the text for analysis.

4. **5-Shingling:**
   Posts are transformed into sets of word-based 5-shingles using NLTK's ngram function.

By executing these steps, we efficiently process the raw XML data into a refined format, laying the groundwork for the identification of near-duplicate posts in the Stack Exchange dataset.

Part 1: Brute Force

In this section, we follow the assignment steps:

1. **Random Sample:**
   We take a random sample of 1000 posts from the dataset.

2. **Jaccard Similarity Calculation:**
   Within this sample, we compute the Jaccard similarity between each pair of posts, leveraging the fact that each post is represented by a set of shingles.

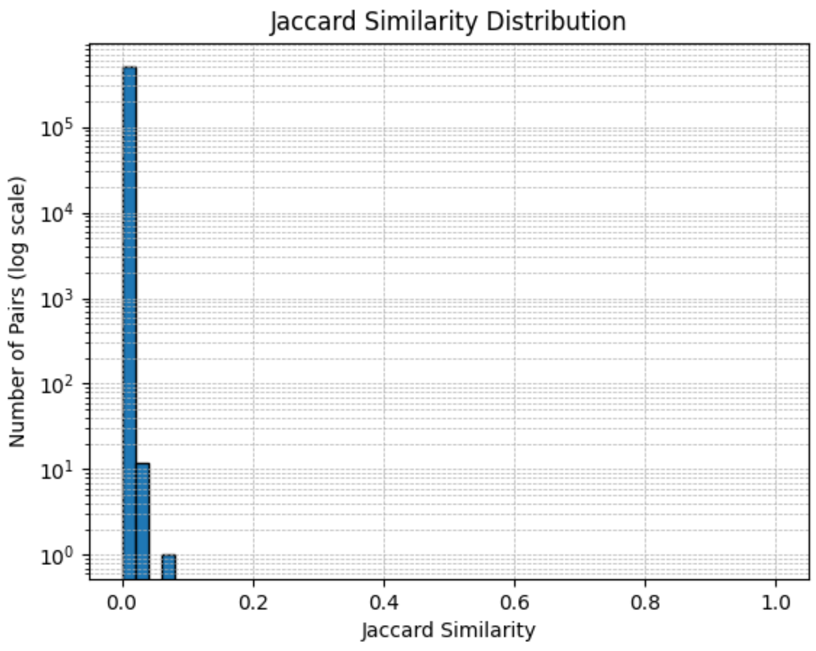
3. **Histogram Plotting:**
   The results are plotted on a histogram with similarity on the x-axis and the count (number of pairs with this similarity) on the y-axis. A reasonable bin-size of 0.02 is chosen for the x-axis, and a logarithmic scale for the y-axis is utilized, as we are particularly interested in pairs with high similarity.

4. **Threshold:**
   Based on the histogram analysis, we observe that the majority of pairs fall under 2% similarity, with only 12 pairs between 2% and 4%, and just 1 pair between 6% and 8%. Given this distribution, a reasonable threshold to identify near-duplicate posts could be set around 2% similarity. This threshold strikes a balance, capturing pairs with substantial similarity while filtering out noise.

5. **Scalability:**
   However, this brute force implementation does not scale well towards larger sets of posts. Generating combinations for pairwise comparisons becomes impractical with increased post numbers. For instance, for 1000 samples, we need 499,500 combinations to create 2 pairs, while for 10,000 samples, an immense 49,995,000 combinations would be required. The computational cost becomes prohibitive, making the implementation unfeasible for larger datasets.

In conclusion, the histogram analysis informs us about an effective similarity threshold for near-duplicate post identification, but the scalability concerns highlight the need for more efficient algorithms to handle larger datasets.

Histogram



Part 2: LSH and Spark

In this part, we provide a comprehensive overview of the Locality-Sensitive Hashing (LSH) solution based on MinHashing implemented on Apache Spark, as per the assignment requirements.

1. **Spark Setup and Data Loading:**
   The initial steps involve setting up Spark and loading the dataset into memory. The extraction of posts from XML is achieved using a code snippet. Leveraging Spark allows for efficient parallel processing, crucial for handling large-scale datasets. The data is parallelized into Resilient Distributed Datasets (RDDs), providing a distributed and fault-tolerant data structure for subsequent processing.

2. **Data Transformation:**
   The RDD containing posts undergoes transformation using the '**map'** function to extract post IDs and shingles for each post. This mapping operation is inherently parallelized across partitions, thanks to the RDD structure, enhancing the scalability of the solution.

   Hashing is then applied to the shingles using the SHA-1 hash function (mentioned in the assignment), creating a set of hashed shingles. However, detailed discussion about the generation of hash functions is deferred until later in the report.

3. **MinHashing**:
   The determination of the number of bands, rows, and hash functions is a crucial step in achieving the desired similarity threshold. Since we are working with the cryptography dataset, the assignment specifies aiming for a threshold around 60%. Using 60 hash functions and 12 bands is calculated to achieve this threshold, as per the formula:

   *THRESHOLD = (1 / BANDS) ** (1 / (NUMBER_HASHFUNCTIONS // BANDS))*

   It's important to note that these parameters (60 hash functions, 12 bands) are not the only possible choices. In order to find suitable parameters for a specific threshold, we developed a script named 'bandAndSignatureSizePicker.py'. This script calculates parameters based on the desired threshold. The discussion of parameter choices is deferred later in the report.

   3.1 **Hash Function generation:**
   To obtain the hash functions, the formula '***ℏ(x)=(ax+b) mod p'*** is used, where ***a*** and ***b*** are random 32-bit integers, and ***p*** is a large prime, specifically the Mersenne prime number. A unique pair of random integers ***a*** and ***b*** is generated for each hash function. These 60 hash functions are then broadcasted using '**sc.broadcast'** in Spark, ensuring efficient distribution across nodes in the cluster.

3.2 **MinHash Transformation**:

Subsequently, MinHashing is applied to each post. A transformation is performed on the hashed shingles, iterating over each hash function for every post. The result is a list of MinHashed signatures for each post, with each signature representing the minimum value obtained from applying the corresponding hash function to the hashed shingles.

4. **Locality-Sensitive Hashing (LSH) Implementation:**

The MinHashed signatures are employed to detect duplicates using LSH. The next steps involve:

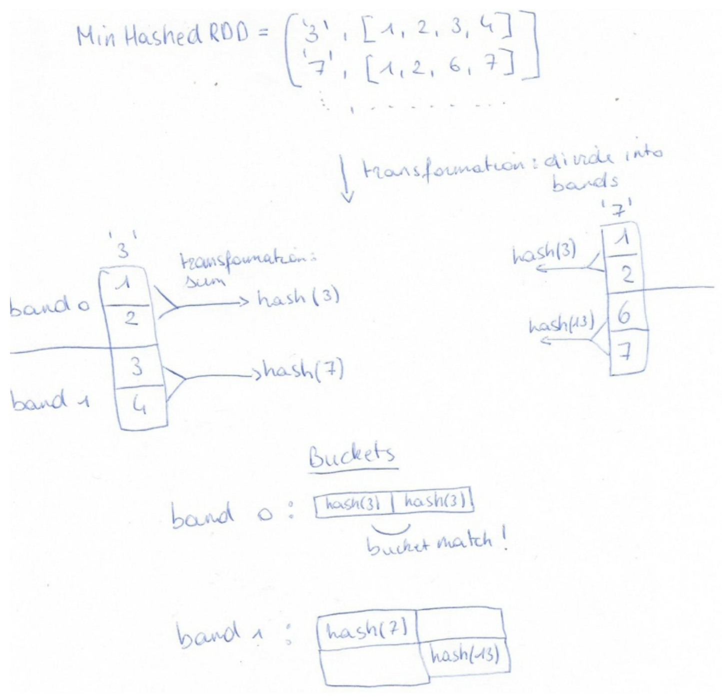
4.1 **Band Division**:

The signatures are divided into bands, each band comprising multiple rows. The '**flatMap'** transformation is utilized for this division, ensuring that the subsequent operations are parallelized efficiently.

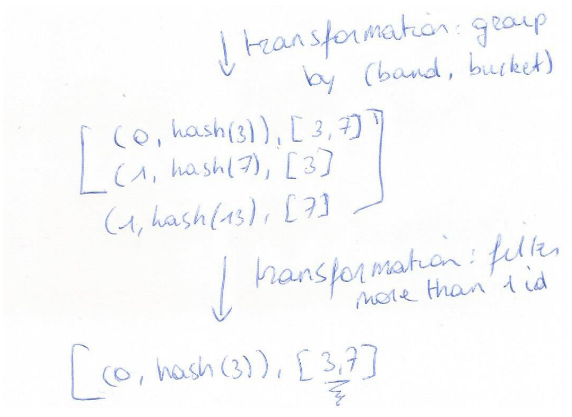
4.2 **Signature Assignment to Buckets:**

Within each band, a hash function is applied to assign signatures to specific bucket. The resulting RDD contains elements in the form: *((band index, bucket), post ID*). This mapping process is executed using the '**map'** transformation.

In this process, we take note that our input comprises a list of hashed signatures for each post within a band. To assign these signatures to specific buckets, a common approach involves summing all the hashed values within each post. Following the summation, a hash function is applied to the summed value to determine the bucket. However, alternate methods, such as concatenation, exist for combining hashed values.



5. **Duplicate Detection**:
To identify potential duplicates, the ‘**groupByKey’** transformation is employed, grouping post IDs within the same band and bucket. Subsequently, the ‘**filter’** transformation is applied to isolate buckets with more than one post ID, indicating potential duplicates.



Finally, the ‘**collect**’ action is employed to gather the identified duplicate post pairs along with their associated buckets, providing a comprehensive list of potential duplicates.

The runtime of the ‘**collect’** action depends on the chosen parameters of bands and hash functions. For instance, with the parameters we mentioned before, the ‘**collect**’action took approximately a minute.

5.1 **Duplicate Examples:**

Candidates ids: 3409, 3410
3409 : The process of encrypting individual files on a storage medium and permitting access to the encrypted data only after proper authentication is provided.
3410 : The process of encrypting individual files on a storage medium and permitting access to the encrypted data only after proper authentication is provided.
Shingle Similarity: 1.0
Body   Similarity: 1.0

Candidates ids: 63188, 63189
63188 : Requests to decrypt a specific message are off-topic, as the results are rarely useful to anyone else and/or would be too long for this site.
63189 : PLEASE NOTE: requests to decrypt a specific message are OFF-TOPIC, as the results are rarely useful to anyone else and/or would be too long for this site.
Shingle Similarity: 0.8
Body   Similarity: 0.6666666666666666

6. **Parameter Choices and Performance:**
Different combinations of bands, rows, and hash functions are explored, each influencing the threshold, true positives, and false positives. A thorough analysis reveals the trade-offs between computational efficiency and accuracy.

6.1 Parameter Impact Analysis:
To illustrate the impact of different parameters on the implementation, the following table showcases how variations in bands, rows, and hash functions influence key performance metrics:

Hash Functions	Bands	Rows	Threshold	True Positives	False Positives	Computational Time (s)
60	12	5	0.608	104	14227	+- 45
126	21	6	0.602	105	15547	+- 70
245	35	7	0.601	110	16436	+- 140

This table succinctly communicates how adjusting parameters impacts the performance of the implementation. Higher values generally lead to increased accuracy but at the cost of longer computational times. Striking the right balance is crucial for optimizing the solution for a specific dataset. The 'Computational Time' column represents the approximate time taken for the 'collect' action, which is the final action performed after all transformations are completed. The collect action gathers the final results for further analysis or output.

7. **Conclusion:**
In conclusion, the LSH-based MinHashing approach implemented on Spark proves to be highly effective for large-scale duplicate detection. Spark's parallelization capabilities significantly enhance efficiency, making it a suitable choice for processing extensive datasets. The comprehensive analysis presented in this report underscores the superiority of LSH over brute-force methods for scalable and accurate duplicate detection. Detailed discussions on parameter choices are deferred to later sections of the report.