Finding similar items

Algorithms for massiv datasets

Cedric Neumann (V12033)

Academic year 2024/2025

1 Introduction

The project aimed to implement an algorithm that finds similar items in different book reviews. Therefore, the Amazon Books Reviews dataset, provided by Kaggle, was used. The dataset can be found at: "https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews "(last access: 11.06.25). The report is structured as follows: Section 2 illustrates the methods used. Section 3 explains an experiment. The results of the explained experiment are shown in section 4. The report concludes with a summary in section 5.

2 Methods

This section explains how the methods were implemented. Figure 1 illustrates the programme flow. The programme must load the dataset from a CSV file using Pandas. It then has to pre-process the loaded data and create k-shingles. These k-shingles are then used to update the MinHash table, which is stored in a MinHash class. This process is carried out in parallel using the multiprocessing Python module. The MinHash signatures are then inserted into the Locality-Sensitive Hashing (LSH) class, which creates buckets based on the chosen threshold. Calling the query function compares the bucket items using Jaccard similarity and stores similar items in a list. This schematic programme flow is implemented in the *finding_similar_items(filepath: str) -> dict[str, list[str]]* function, which takes the path to a CSV dataset and creates a dictionary where each document ID is the key and the similar document IDs are the values.
This approach reduces computation time for n documents by replacing a brute-force algorithm with Jaccard similarity, which has a complexity of $\mathcal{O}(n^2)$, with an algorithm of complexity $\mathcal{O}(n)$.
A function has been defined for visualisation purposes. It takes a dataset and a directory of similar items, and prints the strings.

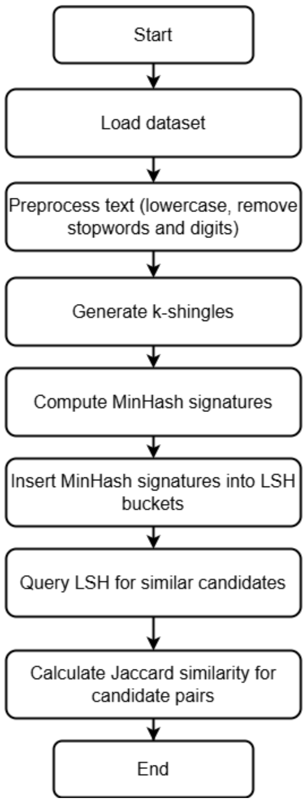
Start
Load dataset
Preprocess text (lowercase, remove stopwords and digits)
Generate k-shingles
Compute MinHash signatures
Insert MinHash signatures into LSH buckets
Query LSH for similar candidates
Calculate Jaccard similarity for candidate pairs
End

Figure 1: Schematic programme flow.

2.1 Jaccard similarity

The Jaccard similarity of two sets S and T is given by [1]:

$$J(S,T) = \frac{|S \cap T|}{|S \cup T|}. \tag{1}$$

The Jaccard similarity coefficient efficiently estimates the similarity between two sets by providing the ratio of items in both sets to the total number of items in the combined sets S and T.
There is a connection between Minhashing and Jaccard similarity. The Jaccard similarity of two sets is equal to the probability that the Minhashing function will produce the same value for a random permutation of the sets:

$$P[\min(h(S)) = \min(h(T))] = J(S,T) \tag{2}$$

The probability that S and T have the same minimum hash value is if and only if the minimum lies in the intersection of the sets S and T:

$$\min(h(S)) = \min(h(T)) \iff \arg\min_{x \in S \cup T} h(x) \in S \cap T \tag{3}$$

The permutation with different hash functions h shuffles the order of the elements in $S \cup T$. Therefore, every element has an equal probability of being the minimum. If the element $x = \arg\min_{x \in S \cup T} h(x)$ is the smallest:

$$P[\min(h(S)) = \min(h(T))] = P[x \in S \cap T] \tag{4}$$

The set $S \cup T$ contains $|S \cup T|$ elements. Each is equally likely to be the minimum, so the probability that the minimum is in $S \cap T$ is:

$$P[x \in S \cap T] = \frac{|S \cap T|}{|S \cup T|} \tag{5}$$

Equation (2) is shown.

2.2 Preprocess and generate k-shingles

The $preprocess\_and\_shingle(text : str)$ function was defined for the purpose of preprocessing and creating k-shingles (see Listing 1). This project requires a string as an argument. In this case, the text being processed is a book review. The text of the review is converted to lowercase to make the shingles case-insensitive. Tokens are created by removing all punctuation and splitting the string into a list of words. These tokens are then filtered to exclude digits and stop words. This reduces computing time and focuses on the content. However, if the goal is to detect plagiarism, it would be better not to remove the stop words. Finally, the k-shingles are created by iterating over the tokens and joining them together as a string within a set. This set ensures that there are no duplicates. These k-shingles are then used to update the MinHash signature.

Listing 1: Preprocess and create a set of k-shingles.

```python
def preprocess_and_shingle(text: str) -> set[str]:
    # Convert the text to lowercase
    text = text.lower()
    # Find all word tokens
    tokens = re.findall(r'\b\w+\b', text)
    # Filter out tokens that are digits or are in the list of English stop words.
```

```python
        tokens = [t for t in tokens if not t.isdigit() and t not in ENGLISH_STOP_WORDS]
        # Create k-shingles of size SHINGLE_SIZE
        shingles = set([' '.join(tokens[i:i+SHINGLE_SIZE]) for i in ↩
            range(len(tokens)-SHINGLE_SIZE+1)])
        return shingles
```

2.3 Minhasing

The idea behind MinHashing is to estimate the Jaccard similarity between sets by computing multiple hash functions over the set elements (k-shingles) and storing only the minimum hash value from each function. The resulting list of minimum values forms the signature of the set. The MinHash algorithm reduces computing time to $\mathcal{O}(n)$, because similarity can be determined by checking for collisions and neighbours, which is done by LSH.
In this project, the MinHash is implemented as a class called MinHash, with arguments *num_perm* which defines the length of the signature, a longer signature leads to a more accurate Jaccard similarity approximation and *seed* which ensures that all instances use the same set of hash functions $h_i(x)$ (see Listing 2). This reproducibility is essential for making the resulting signatures comparable. The MinHash is implemented as a class, meaning that a MinHash signature can be created for every review. The first step in calculating the Minhash signature from a set of k-shingles is to generate hash functions. The function responsible for this is called *_generate_hash_functions(self)*. The number of hash functions, $h_i(x)$, needed is provided to this function, which in turn creates them in form of:

$$h_i(x) = (a_i \cdot x + b_i) \mod c. \tag{6}$$

Where a is a random number between 1 and c, and b is a random number between 0 and c. The upper boundary is set by c and also functions as a constant modulo divider. Hash functions $h_i(x)$ are created using a lambda function and stored in a list.
After the hash functions $h_i(x)$ have been generated, the *update(self, shingles: set[str])* function is called. This takes the generated k-shingles from the preprocessing stage and calculates a positive hash value. This hash value is then passed as x to all the generated hash functions $h(x)$. Only the smallest of the calculated hash values is stored in the list signature.
After minhashing the same shingles have the same hashvalue and shingles with a small difference are neighbours.

Listing 2: Implementation of MinHash.

```python
class MinHash:
    def __init__(self, num_perm=128, seed=42):
        # Number of permutations (hash functions)
        self.num_perm = num_perm
        # Maximum value for a 32-bit unsigned integer
        self.max_hash = (1 << 32) - 1
        # Initialize the signature with the maximum possible hash value for each ↩
            permutation.
        self.signature = [self.max_hash] * num_perm
        # Seed for the random number generator to ensure reproducibility of hash ↩
            functions
        self.seed = seed
        # Generate the hash functions used for creating the MinHash signature.
        self.hash_funcs = self._generate_hash_functions()

    def _generate_hash_functions(self) -> list[callable]:
        # Set random seed
        random.seed(self.seed)
        funcs = []
        # Generate 'num_perm' number of hash functions.
        for _ in range(self.num_perm):
            # Generate random parameters 'a' and 'b'
            # 'a' is chosen from 1 to max_hash, and 'b' from 0 to max_hash.
            a = random.randint(1, self.max_hash)
            b = random.randint(0, self.max_hash)
            # Append a lambda function that represents the hash function (ax + ↩
                b) % max_hash.
            funcs.append(lambda x, a=a, b=b: (a * x + b) % self.max_hash)
        return funcs

    def update(self, shingles: set[str]):
        # Iterate through each shingle in the set.
        for shingle in shingles:
            # Compute a 32-bit CRC (Cyclic Redundancy Check) hash for the shingle
            # and convert into positive value.
            shingle_hash = zlib.crc32(shingle.encode("utf-8")) & 0xffffffff
            # Iterate through each hash function and its corresponding signature ↩
                element.
            for i, func in enumerate(self.hash_funcs):
                # Apply the current hash function to the shingle's hash value.
                val = func(shingle_hash)
                # If the resulting hash value is smaller than the current ↩
                    signature element,
                # update the signature element.
                if val < self.signature[i]:
                    self.signature[i] = val
```

2.4 Locality-Sensitive Hashing (LSH)

To efficiently detect similar items using MinHash signatures, Locality-Sensitive Hashing (LSH) is applied. The main idea of LSH is to split each MinHash signature into several bands, each containing a fixed number of rows, based on the chosen number of permutations and a predefined similarity threshold.
Each band is then hashed into a bucket. If two or more signatures hash to the same bucket in at least one band, they are considered candidate pairs. Only these candidate pairs are compared using the actual Jaccard similarity of their MinHash signatures, which significantly reduces the number of required comparisons.
In this project, LSH is implemented as the MinHashLSH class with the *num_perm* and *threshold* arguments, which define the threshold at which a pair is counted as similar (see Listing refCode:lsh). The first step is to find the optimal band b and rows per band r. The probability that two documents with a similar signature list s are located in at least one band is given by $P(canidate)$ [1]:

$$P(canidate) = 1 - (1 - s^r)^b. \tag{7}$$

An good approximation of threshold is:

$$\text{threshold} \approx \left(\frac{1}{b}\right)^{\frac{1}{r}}. \tag{8}$$

This is done by the function *_optimal_lsh_params()*. It increses b from one to *num_perm* and calculates $r = \frac{num\_perm}{b}$. To ensure that all bands are filled completely, the function only considers

values where *num_perm* is divisible by *b*. The approximated threshold is calculated using the equation (8), as is the difference between the chosen threshold. The best difference is also the optimal value for the bands *b* and the rows *r*. The optimal *b* is used to create a list of dictionaries, into which a maximum of *r* reviews are later inserted.
In order to use LSH, the Minhash signature must be inserted into the buckets. This is done by the function *insert(self, doc_id: str, minhash: MinHash)*, which takes the document ID and the Minhash object. It then iterates over the different bands, calculates the upper and lower boundaries by using *b* and *r*, creates a tuple with the Minhash signatures in the range of the boundaries, and calculates the hash value of the band using the helper function *_hash_band(self, band: tuple)*. The resulting hash value is used as a key in the bucket dictionary to store the document ID. If two documents have the same band hash value, they end up in the same bucket, marking them as candidates for similarity.
Once all the documents have been inserted into the LSH structure, the function *query(self, Minhash: MinHash)* can be used to retrieve similar items. This function takes a MinHash signature as input — the one for which similar documents are to be found. It then computes the band hashes in the same way as the *insert(self, doc_id: str, minhash: MinHash)* function, checking each band for signatures with matching hash values. All matching document IDs are collected into a set of candidates for further similarity comparison. The Jaccard similarity is calculated for each candidate and compared to the threshold. If it is higher than or equal to the threshold, the document ID is added to the list of similar items.

Listing 3: Implementation of Locality-Sensitive Hashing.

```python
class MinHashLSH:
    def __init__(self, threshold: float = 0.8, num_perm: int = 128):
        # Similarity threshold above which two documents are considered similar
        self.threshold = threshold
        # Number of permutations = length of the MinHash signature
        self.num_perm = num_perm
        # Determine the optimal number of bands and rows per band
        self.bands, self.rows_per_band = self._optimal_lsh_params()
        # Create one bucket dictionary per band to group similar signatures
        self.buckets = [defaultdict(list) for _ in range(self.bands)]
        # Store all inserted MinHash objects for later comparison
        self.data = {}

    def _optimal_lsh_params(self) -> tuple[int, int]:
        # Goal: choose b (bands) and r (rows per band) to best approximate the ↩
            threshold
        best_diff = float('inf')
        best_b, best_r = 1, self.num_perm

        for b in range(1, self.num_perm + 1):
            # b needs to be a whole divider
            if self.num_perm % b != 0:
                continue
            r = self.num_perm // b
            # Approximate threshold
            approx_thresh = (1 / b) ** (1 / r)
            # calculate diffreance of approximate threshold and given threshold
            diff = abs(approx_thresh - self.threshold)
            # store if the diff is smaler than the previost ones
            if diff < best_diff:
                best_diff = diff
                best_b, best_r = b, r

        return best_b, best_r

    def _hash_band(self, band: tuple) -> int:
        # Convert the band tuple into bytes and apply a hash function (CRC32)
        band_bytes = str(band).encode('utf-8')
        return zlib.crc32(band_bytes) & 0xffffffff

    def insert(self, doc_id: str, minhash: MinHash):
        # Store the MinHash object for future comparison
        self.data[doc_id] = minhash

        # Go through all bands and insert band hashes into corresponding buckets
        for band_idx in range(self.bands):
            # Calculate the start and end index for this band in the minhash signature
            start = band_idx * self.rows_per_band
            end = start + self.rows_per_band

            # Extract the part of the signature corresponding to this band
            band = tuple(minhash.signature[start:end])

            # Hash the band to get a bucket ID
            hash_val = self._hash_band(band)

            # Add the document ID to the appropriate bucket
            self.buckets[band_idx][hash_val].append(doc_id)

    def query(self, minhash: MinHash) -> list[str]:
        # Find all candidate documents that share at least one band with the input
        candidates = set()
        for band_idx in range(self.bands):
            # Calculate the start and end index for this band in the minhash signature
            start = band_idx * self.rows_per_band
            end = start + self.rows_per_band
            # Extract the part of the signature corresponding to this band
            band = tuple(minhash.signature[start:end])
            # Hash the band to get a bucket ID
            hash_val = self._hash_band(band)

            # Add all documents in the same bucket to the candidates
            candidates.update(self.buckets[band_idx].get(hash_val, []))

        # Compare actual similarity with all candidates using Jaccard similarity
        similar_items = []
        for doc_id in candidates:
            other = self.data[doc_id]
            if self._jaccard(minhash.signature, other.signature) >= self.threshold:
                similar_items.append(doc_id)

        return similar_items

    def _jaccard(self, sig1: list[int], sig2: list[int]) -> float:
        # Calculate Jaccard similarity between two MinHash signatures
        assert len(sig1) == len(sig2)
        return sum(1 for a, b in zip(sig1, sig2) if a == b) / len(sig1)
```

3 Experiments

For the experiment, a benchmark function was defined. It takes a list of sample sizes and the file path of the dataset as input, and measures the execution time using Python's `time` library. To provide a comparison, the function *finding_similar_items(filepath: str) -> dict[str, list[str]]* was used with MinHash and LSH replaced by their counterparts from the `datasketch` library. Each sample size is tested three times, and the average execution time is calculated and stored in a dictionary.
Measurements were taken for sample sizes of 100, 500, 1000, 5000, 10000, and 20000, and the results were plotted using `matplotlib.pyplot`.
A second experiment was conducted to visualise the impact of the k-shingle size and the similarity threshold. To achieve this, thresholds of 0.3, 0.5, 0.7 and 0.9 were tested in combination with k-shingle sizes of 3, 5 and 7, using a sample size of 1,000 and a permutation of 128.

4 Results

Figure 2 shows the benchmark times. It is clear that the execution time follows the order of magnitude of, $\mathcal{O}(n)$. This makes the implemented algorithm scalable for even larger datasets. It also shows that the imported libraries are significantly faster. One reason for this could be better optimization, as well as the switch to NumPy, which is written in C.

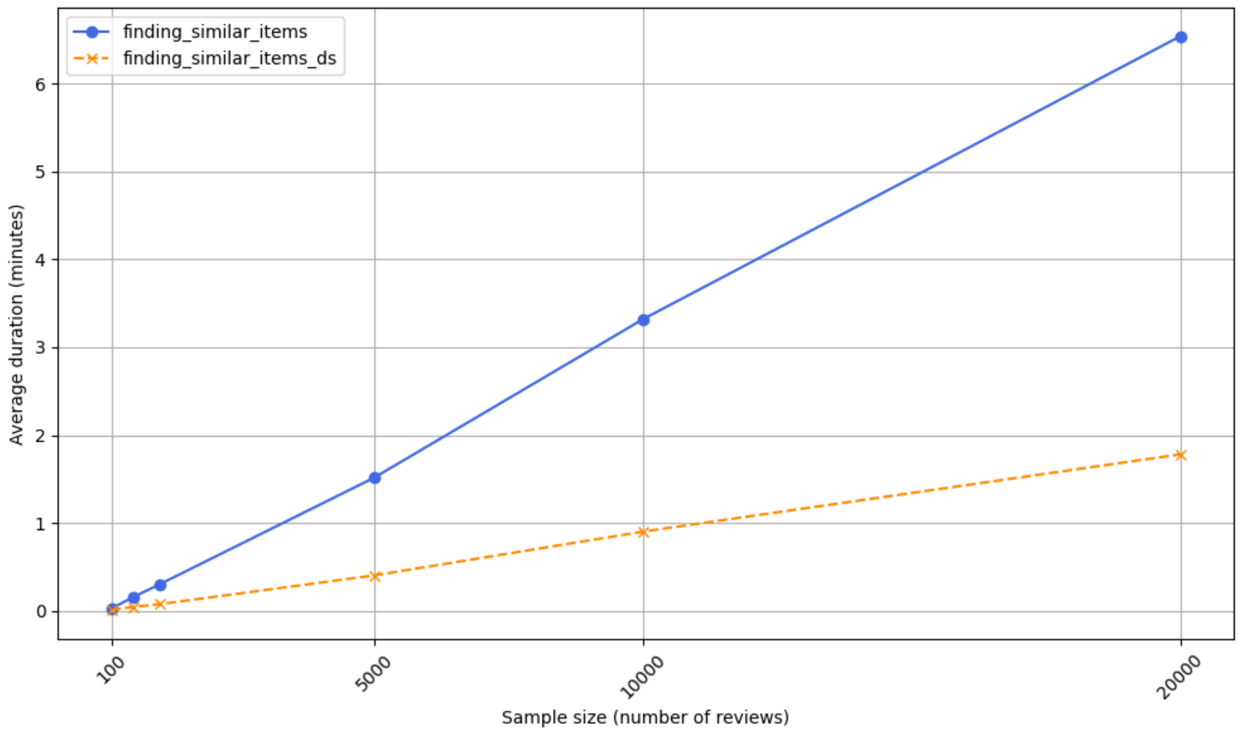

Figure 2: Execute time by different sample sizes.

Figure 3 shows the number of documents with similar reviews. It shows that a lower threshold and a larger shingle size result in more similar reviews being found. It also appears that 0.7 is sufficient for similar items because the same number of pairs are found with a reasonable k-shingle size (lower 7). This behaviour could change drastically with a different dataset. There is also a slight difference in the number of pairs found between the imported and self-written versions. One possible reason for this is the use of different random seeds for the hash functions.

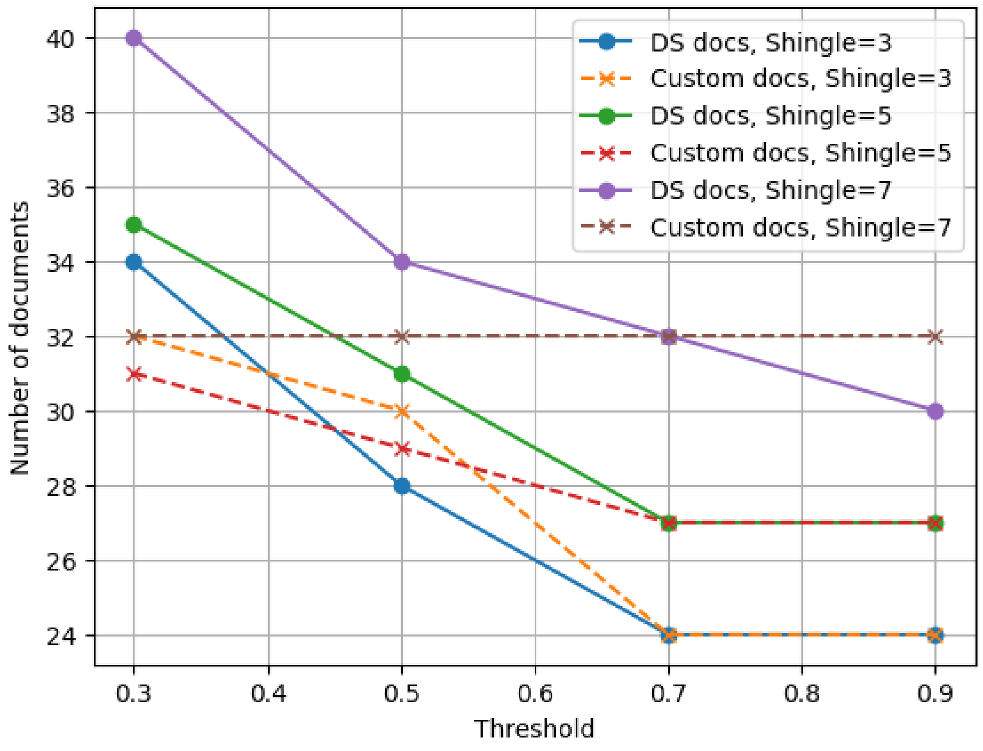

Figure 3: Number of documents with a found pair (samples = 1000, number of permutation = 128).

Figure 4 shows the same behaviour, displaying the total number of pairs in relation to the k-shingle size and threshold. It can also be seen that the highest k-shingles and the lowest threshold result in the greatest number of pairs. The figure also shows that a threshold higher than 0.7 seems to have no effect. But this is highly influence by the dataset.

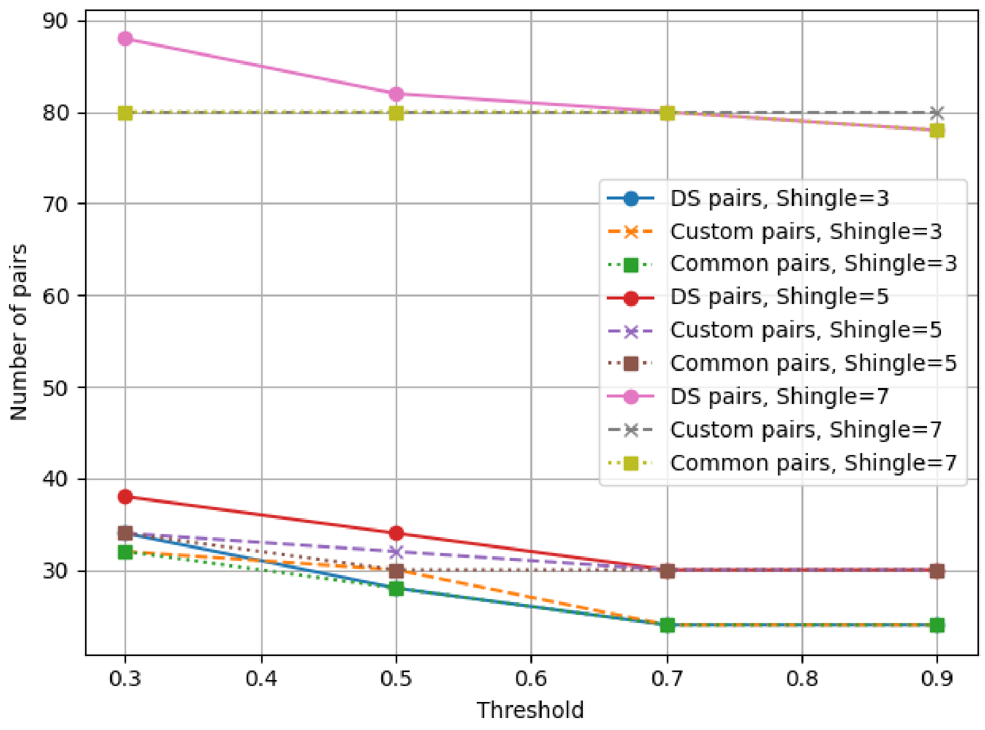

Figure 4: Number of total pairs (samples = 1000, number of permutation = 128)

5 Summary

This report describes the implementation of an algorithm for identifying similar items within the Amazon Books Reviews dataset. The algorithm uses Minhashing and Locality-Sensitive Hashing (LSH), as well as a Jaccard similarity score, to check for similarities. Benchmarking verified that the execution time was $\mathcal{O}(n)$. An imported version from Datasketch was also used for reference. The effect of different k-shingle sizes and thresholds on the number of similar items identified was also demonstrated.

References

[1] A. Rajaraman, J. Ullman, Mining of Massive Datasets. Cambridge University Press, 2011.