Part 2: Data Lake Discovery and Cleaning: MinHash, Similarity Metrics, and Data Quality

Berken Tekin, Radu Vasile, and Ocean Wang

October 2025

Contents

1 Part 1: MinHash - Berken Tekin

1.1 Introduction

Heuristics may help us determine the degree of similarity of two documents, or do it faster. For example, a procedure may consider only the column names of each dataset, and decide to look further only if they are similar enough. This shortcut improves execution speed at the cost of accuracy.

It is reasonable to assume two documents are similar if we knew they shared many common words and phrases. In set theory, this relation is measurable as the "Jaccard similarity" of two sets, and it is defined as $|S \cap T|/|S \cup T|$ for two sets S and T [1]. If we represent our documents as sets, we may use Jaccard Similarity as a concrete heuristic.

In order to convert documents to sets, one method is to collect all unique sequences of length k inside the document. This technique is called k-shingling. In Python, this is done by slicing a string to k characters, starting from the left and moving right one character at a time. Afterwards, the result is converted into a set to discard duplicates.

It is perfectly fine to directly calculate the Jaccard similarity afterwards. However, it may not always be the most efficient thing to do. At most $d-k+1$ shingles are produced from a document with d characters, and Python will compare the shingle sets of two documents in $O(d)$ time [2]. When a new document is added to a pool of n documents, n comparisons need to be made to rank the similarity of all pairs, bringing the total time complexity to $O(nd)$. Each document can be millions of characters long, and there can be millions of documents, which impacts the execution speed through sheer scale.

There are two techniques that can work together to mitigate this time complexity at the cost of accuracy. One of them is called MinHashing, and the other is LSH (Locally Sensitive Hashing). MinHashing can be thought to reduce the d component, and LSH the n component.

For this assignment, the main goal is to capture JOINable documents by finding individual columns with high Jaccard similarities from two documents. This will be done by creating MinHash signatures from each column, and then applying LSH for quick comparison. The impact of the condition of data lakes is evaluated by applying the same algorithms to clean and dirty data.

The content is as follows: firstly, the cleanup and preprocessing methods for the dataset is summarized. Afterwards, the algorithms for MinHash, C-MinHash and Locally Sensitive Hashing (LSH) are introduced. Lastly, the results are examined.

The standard MinHash implementation provided in the datasketch module [3] is used in this assignment. However, I have also extended the provided MinHash module, and implemented a newer algorithm called C-MinHash [4, 5] whose $(\sigma - \pi)$ version is proven to be strictly more accurate while needing less random permutations. Two variants of C-MinHash will be under consideration. I have also implemented a Locally Sensitive Hashing (LSH) algorithm that is compatible with the MinHash library.

The algorithms and implementation details are available in the corresponding Jupyter notebook.

1.2 Preparation

The dataset is read using `pandas.read_csv()`, configured to automatically find suitable delimiters, skips unreadable columns, and remove NaNs. Under these conditions, the method recognizes 16 out of the 19 tables provided.

The CSV is then flattened into an array, and converted into string. This serialization is done for the entire documents as well as each individual column. In total, 334 columns are extracted.

Afterwards, k-shingles are created for each document and column. It is recommended to choose $k = 20D$ where D is the total number of characters in the documents. For the given documents, we calculate $k \approx 6.4$ using raw document sizes, as CSV files contain minimal metadata. I settled on using $k = 6$ because data cleanup might reduce the number of characters.

All shingles are then converted to a convenient numerical form using the last 32 bits of their SHA-1 hashes. It is a rare occurrence for two different shingles to be hashed to the same value, any effect from this is assumed to be insignificant.

From this point on, the hashes are always wrapped to 32 bits if they get larger.

1.3 MinHash

If one shuffles two lists of (hashed) shingles using a random permutation, the probability of the first elements being equal is the same as the true Jaccard similarity of these two lists [1]. An estimator for Jaccard similarity can be constructed by recording the results under Q different permutations, with a variance of $J(1-J)/Q$ where J is the true Jaccard similarity [4]. For n documents, MinHashing brings down the complexity of comparing one document to the others to $O(nQ)$.

A naive Fischer-Yates shuffle requires applying it to entire documents at once. Instead, we imagine the hashed items to be strewn across a hash table, and "shuffle" them by applying a linear permutation which moves each element into another position on the hash table. Since the mapping for each shingle is deterministic, we may process one shingle at a time, and uncover the true minimum incrementally.

We apply Q linear permutations to the shingles in the form of $\phi_q(h) = (a_q \cdot h + b_q) \bmod p$, where a_q and b_q are randomly chosen numbers for a permutation q, p is a large prime number, and h is the hashed shingle. Such permutations have a roughly equal probability of moving each element in front of other elements (to a smaller hash value), so much so that they can be treated as min-wise independent linear permutations that simulate an actual shuffle [6]. The chosen values for a and b must be the same for all documents [4]. Otherwise, two documents sharing the same MinHash would not mean them sharing their first character, as the permutations may bring different shingles to that hash value.

For each of the Q permutations, the minimum hash number encountered across all shingles is recorded in order. The resulting Q hashes together form the MinHash signature of the document, and can be used to estimate the Jaccard similarity of two documents.

1.4 C-MinHash-(σ, π)

For the classical MinHash algorithm, each of the Q permutations are created with random a and b parameters. However, there is a variant MinHash method that uses just two initial permutations instead of K ones. This algorithm achieves a strictly lower variance compared to MinHash [4,5], meaning Jaccard similarity estimations made using C-MinHash are in general closer to the true value.

With C-MinHash, only two linear permutations are needed, π and σ. The π permutation is needed to break up the internal structure within the document. It is shown that for C-MinHash, the Jaccard estimations gain a bias based on the document's form if π is not applied [5]. Afterwards, Q permutations are created by shifting a single permutation mapping right, starting from 1 up until Q times. This means modifying the permutation to assign the final position of hash h to hashes $h+1$, $h+2$, etc. in the next iterations.

In my implementation, instead of rotating the permutation function to the right, I instead subtract $1..Q$ from the hashed shingles before the permutation for the same effect. It is not likely that the shift direction matters, but a right shift is preferred to mimic the implementation on paper.

1.5 C-MinHash-($\pi - \pi$)

There exists another variant with a single function for both the initial permutation and the shifts [4]. The theory behind this variant is more convoluted, and its variance is biased based on structure in data; however, this bias is insignificant [4]. It is not clear whether removing a permutation is worth the bias, but it is implemented anyway to evaluate its real life accuracy.

1.6 Locally Sensitive Hashing (LSH)

With MinHash, we obtain Q MinHashes for each of the 343 columns (features), which means we would have to make $343 \times Q$ comparisons for each new entry in order to determine most JOINable columns. Locally Sensitive Hashing accelerates the process of calculating the Jaccard similarity between two entries by dividing each set of Q MinHashes into b bands, each containing r MinHashes. For two documents (in our case columns) the probability that all r entries match for any one of the bands is $1-(1-s^r)^b$, where s is the Jaccard similarity. All such entries are tagged as "candidate pairs". This probability sharply increases as soon as the Jaccard similarity surpasses a certain threshold. The threshold is estimated as $(1/b)^{1/r}$. In the experiment, we use $b = 4$ and $r = 64$, resulting in a threshold of $t \approx 0.98$, which means only nearly identical columns have a realistic chance of becoming candidates.

This way, MinHash comparisons can be made less redundant by stopping prematurely the moment one identical band is captured between two documents, as that is enough to add them to the set of candidate pairs.

A manual implementation for LSH is provided in the code.

1.7 Cleaning the Dataset

For subquestions (c) and (d), the group has performed a cleanup on all tables. In this section, the effects of data quality on the MinHash procedure will be discussed.

The naive way of preprocessing data is to let `read_csv()` [7] automatically determine structure in CSVs. However, a manual analysis and cleanup improves the accuracy of obtained data. For example, without a cleanup, the method (silently) fails to parse the values for Table 4 and Table 8, and yields empty shingle sets. Furthermore, Tables 9 and 12 fail to be recognized.

1.8 Results

Charts for per-file MinHash scores can be generated by the code in Parts 2 and 4.

1.8.1 Dataset cleanup

Cleaning the dataset augmented true similarities, and helped mitigate false positives. The impact of dataset cleanup can be listed as follows:

The most prominent example is the similarity between Table 4 and Table 8. In the unclean dataset, due to `read_csv()` silently failing to read the columns, the MinHash function declared a false positive by counting the Jaccard similarity between empty sets as 1. However, the cleaning process fixes this issue, and all tables are read properly.

1.8.2 Variants of MinHash

The Jaccard estimations computed from MinHash, C-MinHash-(σ, π) and C-MinHash-$(\pi - \pi)$ never significantly diverge from each other, or the actual Jaccard similarity. C-MinHash should still be preferred over MinHash due to theoretical increase in accuracy without increasing time complexity.

1.8.3 JOINable Columns

An output that lists all candidate pairs per column (if found) can be printed by running the code (Parts 2 and 4).

LSH captures many columns that are nearly identical to each other in real life. However it is not always an indicator of actual JOINability. For example, through the algorithm we found that the "TYPE_TERRAIN" column in Table 8 takes on values between 0 and 3, and so does the "PEAK_PARKING" column in Table 14. However, the column labels indicate different topics.

1.8.4 GPT Disclaimer

The method that prints out the tables for Jaccard similarities across whole datasets is generated by GPT.

2 Part 2: Discovery Methods - Radu Vasile

2.1 Introduction

In this part, I aim to identify similarities between pairs of columns of both datasets, to have an overview of whether the datasets can be joined on the respective columns. In my analysis, I am looking at two possible relations between columns: Similarity between column values and Similarity between column names, while implementing two different methods for each relation.

All code for this analysis has been written in Python, using the Pandas library to read the provided CSV files. I have noticed that some of them did not use the comma as a separator, but the underscore character, which was causing errors in the code that read the files. Therefore, I had to adapt the method to also recognise the underscore character as a separator.

2.2 Similarity between column values - JOIN

One reliable method to identify similarities between columns is to compare the values. For a meaningful comparison, we have to consider all column values, which usually leads to very high computation times and large complexity, however it comes with improved accuracy, meaning they should be able to capture all relations.

2.2.1 Method 1 - Set Containment

For the first method, I have opted to measure the Set Containment between all pairs of columns of all tables. The Set Containment metric of sets X and Y represents a ratio of how many values from set X are also present in set Y. Large values mean that most of the values from column X also appear in column Y, giving us a strong indication that they might be related.

One issue with this method arises when set X consists of a very small set of values, while set Y contains many. This may cause accidental overlaps between the sets, yielding a high value for Set Containment, even when the columns are not related. For instance, consider a column in dataset X which represents a boolean, having only values in $\{0, 1\}$, and a column in dataset Y which represents a real number in $[0, 1000]$. If the second set is sufficiently large, it may happen that 0 and 1 appear in Y, leading to a set containment value of 1. Therefore, while successfully identifying all true positives, this method can lead to a high number of false positives as well. To mitigate this effect, I have set the threshold of set containment to 0.8, which is a sufficiently high number to ignore a large part of false positives, while also capturing most of the true positives.

I have implemented this method by iterating over all pairs of columns of the datasets, calculating the set containment and adding them to the result if the value exceeds the 0.8 threshold. The results appear to align with my expectations, in the sense where it appears to capture many similarities between tables 0, 1 and 2, which seem to have similar data at a first glance, while also having different column names. It also manages to capture similarities between various columns of tables 17 and 18, where the column name appears to be intentionally scrambled, thus very different.

2.2.2 Method 2 - JOSIE

JOSIE is an algorithm that constructs a top-k of most similar columns to a given query column. It makes use of posting lists to store values, then uses those lists to find matches in other columns, for each value in the query column. The use of these indexes makes the algorithm very fast, however it brings a huge memory overload due to the need of storing all (value, column) pairs. I have implemented these lists in python using a dict structure, which is most similar to a map in other programming languages. The keys represent unique values found throughout all columns, while the values represent (column, table) pairs, a unique identifier that allows us to find all columns that contain the key. In my analysis, I have chosen the value $k = 3$, considering a top-3 similar column enough to properly interpret the results. After constructing the list, I create a set of values for each column in each dataset. Then, for each value in the set, I look in the posting lists and create a counter to keep track of how many times each column appears. I then consider the top-3, if it exists, and add it to the result. If no other column is found that shares the same values, then I exclude the current column from the results.

This method is great at figuring out which columns share the most values, which is a relevant metric for identifying joinable tables. For instance, it manages to identify relations between tables 0, 1 and 2, as well as 17 and 18, which appear to be the same after data cleaning. The number of overlaps between the corresponding columns appears to be very high, which is a good indication that they are indeed related. However, since the algorithm only constructs a ranking of most similar columns, it may be the case that some of them are unrelated to others, and it happens, by chance, that some values overlap. This is usually indicated through a low value of the Set Overlap metric used by JOSIE. Therefore, this method works best if the results are also validated through the use of other methods or metrics.

2.3 Similarity between column names - UNION

It is also meaningful to consider similarities between the column names to identify columns that may refer to the same data. It is especially useful in the situation where the values from the columns are different, but the names point to the same classification criteria. For example, if we have two datasets representing company reports, and each of them has a date column, containing values from different years, these tables could be unioned. This relation would not have been captured by any of the methods described above.

2.3.1 Method 1 - Levenshtein Distance

Levenshtein Distance is a good metric to identify similar strings. It is also known as "Edit Distance" and it calculates the number of "edits" to turn one string into another. An edit can be any of the following operations: insertion, deletion or substitution. This metric is calculated using a dynamic programming approach.

However, it returns an integer, which is not very meaningful by itself. Therefore, in order to set a threshold, I will consider the ratio of the distance divided by the length of the longest string, in order to obtain a value between 0 and 1. We only want to consider similar strings, therefore I chose to include in the result the column names which have a ratio smaller than 0.2.

One advantage of this approach is the fact that it is more lenient towards larger strings, where it is more common for spelling mistakes or punctuation differences to appear. However, this may also be an issue in some situations. For instance, if we compare two strings of length 4, even one different character makes the string to be excluded. However, if we compare two strings of length 30, we allow for up to 6 differences in characters, which is sufficient for a change in meaning. Another disadvantage of this method is that it will probably not work with abbreviations, as they are usually much shorter than the full word, leading to a large value for the edit distance algorithm.

A discussion on results will follow in the next subsection, since both methods share the same output file.

2.3.2 Method 2 - Jaccard Similarity on Shingles

This method considers all k-shingles of the column names and compares them using Jaccard Similarity. I have chosen to consider $k \in \{2, 3, 4, 5\}$, to see how longer substrings impact the results. The aim of this method is, again, to identify similarities between names, however having a better chance to also identify abbreviations. Since we are comparing all substrings of length k, shingles of abbreviated versions of the same name will also be present in that subset. However, if the name is different, this will result in different substrings, giving a lower value.

I have decided to include both methods in the same output file, filling up the results with "null" if the value falls below the threshold. I believe it will be interesting to see for which strings Jaccard Similarity works better and for which it does not.

The results confirm the expectations, meaning that Levenshtein distance performs better when the names only feature a few typing errors. No data has resulted from columns with abbreviations, meaning it is possible they were not included in the dataset, or the abbreviations shortened too many characters from the original version. Most relations identified were captured by both methods, therefore leading to similar results.

Meaningful results are also impacted by the fact that the datasets have not been previously cleaned. For instance, some datasets are missing column names, while others contain intentional spelling mistakes, or even scrambled names, which makes it hard to capture all relations. I expect very different results from this method after cleaning the datasets.

All outputs from the above methods can be found in the attached "outputs" folder.

2.4 Results after cleaning

After data cleaning, we notice slightly different results. Firstly, the Set Containment method returns even more matches, with most of the new candidates appearing from tables which were unable to be read before cleaning due to parsing errors. However, all data that appears in the uncleaned results is also featured in the new analysis, having the same values for set containment. This indicates that the discovery method was able to successfully capture all relations where the data was correctly formatted and displayed.

The JOSIE method also does display minor differences, keeping similar top-3 results in most cases, however it is able to capture with more accuracy the similarity between tables 0, 1 and 2.

A major difference is observed for the performance of the algorithms identifying the union relation. After the cleaning of the datasets, which implied removal of symbols such as emojis, square brackets, quotation marks, as well as unscrambling names, we notice these methods return many more results than before. This is because the algorithms no longer take into account those extra characters in the analysis, which would have resulted in a larger edit distance or smaller Jaccard similarity.

It is now also more obvious where each method performs best: Levenshtein Distance outperforms Jaccard in situations where the column names differ through a few characters in the middle of the word, for instance: "MSC_ID" in `table_9.csv` and "MSC ID" in `table_11.csv`. On the other hand, Jaccard performs best in case of word repetition, such as: "SCALAR_ID" in `table_3.csv` and "SCALAR_ID SCALAR_ID" in `table_4.csv`. In normal situations, both methods manage to identify similar titles, with relatively high scores.

3 Data Lake Cleaning - Ocean Wang

The lake33 dataset contains 19 CSV files with corruption issues that would hinder discovery algorithms in finding relations between datasets if left unaddressed. The most severe problems are BOM character corruption that mangles first column names (tables 3-5), missing headers entirely (tables 6, 10, 11), and scrambled headers (tables 7, 18) that break column-to-data mapping. Secondary issues include emoji contamination across 11 tables, multilingual headers mixing English and translations, excessive quotation marks, and duplicate header tokens.

The cleaning methodology uses configurations, where each table's corruption patterns are encoded as tuples containing the delimiter and a set of flags indicating which cleaning operations to apply. The `datacleaner` class processes tables in two passes, independent tables that have their own headers run first, followed by dependent tables that inherit headers from already-cleaned sources. The `load_table` method reads the CSV with the correct separator and handles special cases like missing headers (loads without header row), malformed headers (skips corrupted first line), or custom formats. The `clean_col` method strips BOM characters by removing BOM markers and filters non-printable characters, extracts English portions from multilingual headers by splitting on newline delimiters, removes emojis through ASCII filtering, strips quotes and brackets with regex, and removes duplicate tokens when a header repeats the same word multiple times. Text data gets similar treatment through `clean_text`, which normalizes whitespace, removes non-ASCII characters, and detects when a value consists of the same word repeated. Header inheritance works by checking if a table's config specifies a source table in position 4 of its tuple, then copying that source's already-cleaned column names if the column counts match. The `process` method first loads with appropriate setting for table, apply cleaning or header inheritance, fixes specific column names if configured, clean text data in object columns, remove duplicate rows, and then writes the results. Table 7's scrambled headers and table 6's missing headers are not cleaned as tables with similar structure could not be found. A solution would be to look on the internet for a copy of table 6 that does contain the headers, and for table 7 bruteforcing to unscramble each header name is also a solution.

After cleaning, there should be a folder named "lake33c" containing the clean tables. The discovery algorithms should perform better now in showing relations between the tables. Without cleaning, wrong column names cause joins to fail, emoji-contaminated text breaks matching and aggregation, missing headers force analysts to guess field meanings, and multilingual content leads to inconsistent grouping. The discussion for the results of the discovery algorithms with cleaned data can be found on the pages "P2_ab_Discovery".

References

[1] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*. Cambridge University Press, 2014.

[2] "TimeComplexity - Python Wiki," Accessed: Sept. 29, 2025. [Online]. Available: `https://wiki.python.org/moin/TimeComplexity`

[3] "datasketch/datasketch/lsh.py at master · ekzhu/datasketch," GitHub. Accessed: Sept. 29, 2025. [Online]. Available: `https://github.com/ekzhu/datasketch/blob/master/datasketch/lsh.py`

[4] X. Li and P. Li, "C-MinHash: Practically Reducing Two Permutations to Just One," arXiv:2109.04595, Sept. 10, 2021. doi: 10.48550/arXiv.2109.04595.

[5] X. Li and P. Li, "C-MinHash: Improving Minwise Hashing with Circulant Permutation," in *Proceedings of the 39th International Conference on Machine Learning*, PMLR, June 2022, pp. 12857–12887. [Online]. Available: `https://proceedings.mlr.press/v162/li22m.html`

[6] T. Bohman, C. Cooper, and A. Frieze, "Min-Wise Independent Linear Permutations," *Electron. J. Comb.*, vol. 7, no. 1, p. R26, Apr. 2000. doi: 10.37236/1504.

[7] "pandas.read_csv — pandas 2.3.3 documentation," Accessed: Oct. 01, 2025. [Online]. Available: `https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html`