CMPE 492

Locality Sensitive Hashing for Hierarchical Data Structures

Elif Bayraktar

Advisor:

Doğan Ulus

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Broad Impact

As we will mention in more detail later, Locality Sensitive Hashing techniques are quite popular and widely used in various forms of data mining and database applications. Some essential tasks in this area are grouping of data objects (clustering), classifying data objects, and detecting exceptional ones (outlier detection), most of which involve solutions based on similarity search, hence, making efficient similarity search in large databases of structured objects a vital research area of modern database applications.

Some application areas are Document Analysis (Plagiarism, Mirror Pages, Articles from the Same Source), Collaborative Filtering for On-Line Purchases and Movie Ratings, Social Network Analysis, Bio-informatics, Bioassay Testing, Urban Computing, and Healthcare. Our focus for this project is on the similarity of documents in JSON (and YAML) format. We search for methods to calculate the similarity of highly similar JSON objects. Our starting point is the comparison of Autonomous Driving Scenarios given in the references [1]. However, we also want to develop a technique applicable to various types of JSON and YAML documents regardless of their content.

1.2. Ethical Considerations

Since the focus of this project is on developing algorithms and library functions that can be used by developers and data scientists, rather than an end product for users, many of the ethical concerns such as user data privacy do not apply here. Nevertheless, there exists the potential risk of undetected biases in the algorithm, which may lead to favoring of some data in classification and queries, hence causing unfair situations.

2. PROJECT DEFINITION AND PLANNING

2.1. Project Definition

Locality Sensitive Hashing (LSH) is a widely popular technique used in approximate similarity search. It is used due to the "3V" (volume, velocity, and variety) of big data. Because comparing billions of data points with high and unpredictable dimensionality in current-day searches is not practically feasible. That's why we need fast approximate similarity algorithms to help with queries.

That's when Locality Sensitivity Hashing (LSH) comes into play as a powerful approach to similarity (or distance) estimation. It exploits a family of randomized hash functions to map similar data instances to the same buckets with a higher probability than dissimilar ones. Two of the most commonly used methods for LSH are Min-hashing and Random Projection, both of which effectively reduce the dimensionality of the compared sets (or vectors).

Although immensely helpful in terms of time and very much preferred due to its exchangeability, one downside of this approach is the loss of context information and semantic hierarchy. Classical LSH is applied to sets, meaning when it needs to be applied to structured data, the data is usually reduced to a flattened set, and any information regarding the hierarchy is possibly lost. That's why researchers have developed state-of-the-art Hierarchical LSH techniques to make use of the hierarchical information while benefiting from the time-saving qualities of LSH.

Our project focuses on the investigation of these hierarchical LSH techniques (along with classical ones), to come up with ways to apply them to files in JSON (or YAML) format. We start with a thorough literature survey of existing algorithms both for LSH and hierarchical LSH. We again research and compare existing implementations for similar purposes. Then we decide on a metric for determining the exact similarity

of two JSON objects, keeping in mind the hierarchical structure. We implement a few possible solutions. Finally, we incorporate our own hierarchical LSH algorithm and try to get results as close to exact similarity comparison as possible with LSH. We test the results with exact similarity scores versus candidate pairs and a comparison of exact versus approximate nearest neighbors. In the end, we report the results of our experiments and our observations.

2.2. Project Planning

2.2.1. Project Time and Resource Estimation

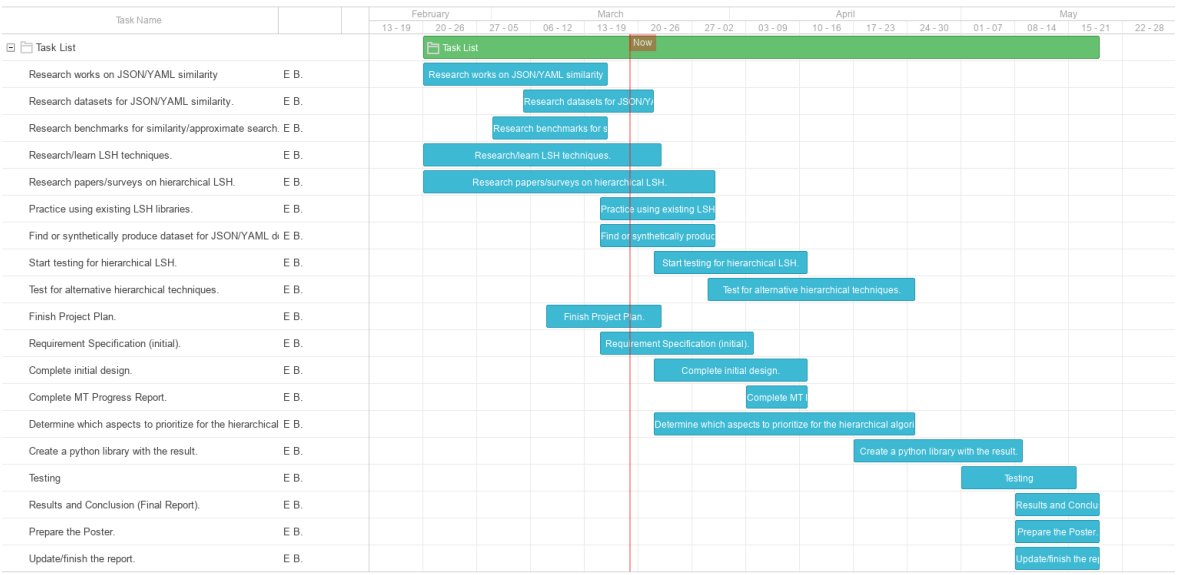


Figure 2.1: Gantt Chart

2.2.2. Success Criteria

- Researched and learned about LSH algorithms.
- Researched and learned about proposed hierarchical LSH algorithms.
- Performed a thorough literature survey of implementations with similar purposes. (LSH and/or JSON similarity)
- Developed techniques for calculating the exact similarity of two (structured) JSON files.
- Implemented the said algorithms for JSON similarity and compared/critiqued

the results.

- Developed an algorithm for approximate similarity search of JSON documents.
- Implemented the said algorithm.
- Implemented a k nearest neighbors algorithm and compared with expected results.
- Compared and critiqued the results.
- Documented the research and findings along the way.
- Prepared a well-structured report.
- Proposed issues for future development.

2.2.3. Risk Analysis

- Existing hierarchical LSH techniques for data in the form of graphs apply mainly to traditional human-readable formats like paragraphs and sentences. But in our case degree of recursion is higher, which means for applying the same technique we either need to go through more layers(repetition) or cut it off in a certain layer. The first takes more time than anticipated and the second might lead to inaccurate results due to the structure of our data explained below.
- One of our aims is to develop an algorithm that can distinguish between highly similar JSON documents. This means that the documents will follow a given schema and will have more or less the same fields/keys in the first few degrees(depth). This is different from graphs where hierarchical information is in the representations of paragraphs and sentences etc. Hence the same approach wouldn't translate well to our case and applying the cut-off method from top, most of our nodes would end up in the same bucket.
- Due to the project having a research-heavy side, estimating the timeline is hard and sometimes inaccurate.
- Due to the unique nature of the datatype and the algorithm we have developed, it is hard to work with existing libraries. For instance, once we had the code ready for producing signatures and candidate pairs, we wanted to incorporate Faiss Library of Facebook to perform approximate nearest neighbor searches. However,

Faiss [2] is commonly used with sentence or word embeddings (such as Bert), which does not serve our purposes. Although we managed to get around this by using the signatures as the vector representations, the results were inaccurate from time to time, and it proved better to implement a custom method.

3. RELATED WORK

Here we will mention some of the more significant research papers and books on the subject.

3.1. Locality Sensitive Hashing for Structured Data: A Survey

This paper [3] is one of the starting points of our project. It briefly summarizes LSH, different hashing functions, and distance measures. Then it goes on with hierarchical LSH for different data structures, namely, Trees, Sequences, and Graphs. It gives key points for the proposed algorithms and references to the original articles, which we have also investigated. Especially the ones concerning Trees, since they are the closest to our domain. The paper also mentions application areas and some possible challenges.

3.2. Mining of Massive Datasets

This [4] is a go-to book when it comes to data mining and big data applications. For our research purposes, we focused mainly on the 3rd chapter titled 'Finding Similar Items'. Here we investigated the critical components related to LSH, such as Jaccard similarity, shingling, hashing, matrix representation of sets, min-hashing, signatures, banding technique, different distance measures, locality-sensitive function families, different applications of LSH, application-specific approaches, and more. We do not summarize these methods here to save space, which are all explained in detail in Chapter 3 of MMDS.

3.3. Context-Preserving Hashing for Fast Text Classification

This paper [5] focuses on hierarchical LSH for Tree-structured data. It proposes a Recursive Min-wise Hashing which accounts for multiple levels of exchangeabilities and

probabilistic comparison of sub-sets instead of hard matching. It is mainly aimed at texts such as paragraphs and sentences and makes use of nested sets called Multi-Level Exchangeable Representations. The algorithm uses multiple levels of Min-wise hashing (recursively) with Fingerprint reorganization in between, shown in the figure below. Although the results of the experiments show significant improvement in classification

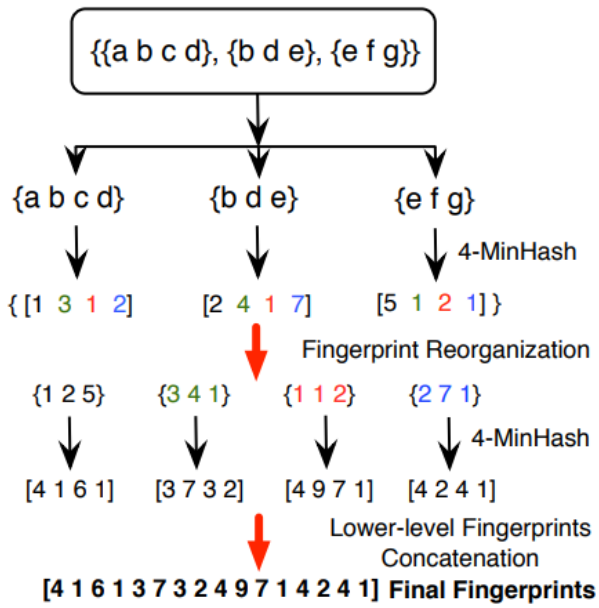


Figure 3.1: An illustration of the proposed Recursive Min-wise Hashing (RMH) algorithm on a nested set

accuracy while maintaining the same (or less) computational cost, the difference in the data structure, and increased levels of hierarchy make the algorithm not directly applicable to our case. Furthermore, even if we applied, the risk of increased complexity remains due to more hierarchic (recursive) hashing schemes.

3.4. The Power of Two Min-Hashes for Similarity Search among Hierarchical Data Objects

This [6] is another paper on the similarity search of hierarchical data (especially trees). It shows that propagating one min-hash results in poor sketch properties while propagating two min-hashes results in good sketches. It also uses Earth Mover's Distance [7] as the metric for trees.

3.5. Efficient Similarity Search in Structured Data

This [8] research focuses on efficient similarity search in large databases of structured objects. Here several efficient methods for similarity search are developed, and some existing query processing techniques are improved. Tree or graph-structured data are considered. It is found that although edit distance is an accurate and popular measure for the similarity of tree-shaped data, the computation is extremely costly; [9] hence improvements such as multi-step query processing are needed (through the use of filters). Additionally, alternative distance measures are offered, such as edge matching distance. The properties of the discussed methods are investigated via theoretical analysis and experiments.

3.6. Faiss: The Missing Manual

Facebook AI Similarity Search (Faiss) [10] is one of the best open-source options for similarity search. This e-book provides a guide on effectively using the library along with graphical, easy-to-understand explanations of basic LSH methods with examples.

3.7. Similarity Search in High Dimensions via Hashing

" The nearest- or near-neighbor query problems arise in a large variety of database applications, usually in the context of similarity searching. Of late, there has been increasing interest in building search/index structures for performing similarity search over high-dimensional data, e.g., image databases, document collections, time-series databases, and genome databases. Unfortunately, all known techniques for solving this problem fall prey to the curse of dimensionality." That is, the data structures scale poorly with data dimensionality; in fact, if the number of dimensions exceeds 10 to 20, searching in k-d trees and related structures involves the inspection of a large fraction of the database, thereby doing no better than brute-force linear search. It has been suggested that since the selection of features and the choice of a distance metric in typical applications is rather heuristic, determining an approximate nearest neighbor

should suffice for most practical purposes. In this paper, we examine a novel scheme for approximate similarity search based on hashing. The basic idea is to hash the points from the database so as to ensure that the probability of collision is much higher for objects that are close to each other than for those that are far apart. We provide experimental evidence that our method gives significant improvement in running time over other methods for searching in high dimensional spaces based on hierarchical tree decomposition. Experimental results also indicate that our scheme scales well even for a relatively large number of dimensions (more than 50)." [11]

4. METHODOLOGY

We began our project by performing a thorough literature survey of the existing methods and algorithms for both classical LSH and Hierarchical LSH. We also researched and used several libraries/packages for LSH and JSON similarity applications [12]. As per the implementation, the first step was deciding on a metric to compute the exact similarity of 2 JSON documents. - Here, we should mention that for documents in YAML format, we chose to convert them to JSON. This was relatively easy to do and protected the file's hierarchical information.

Firstly, we discussed several alternatives for exact similarity criteria and implemented 4 of them. The first is the method of flattening the JSON document to a set. Here, the elements of the set (nodes) are taken as leaf nodes of the JSON file concatenated with their hierarchical paths. This method prioritizes leaf nodes, which is desirable for our purposes since we expect compared documents to have the same schema - meaning they will have more or less the same fields(keys).

Secondly, we implemented a slightly different version of the algorithm above. This time we added (separately) the paths consisting of keys to the flattened set. This put more emphasis on the hierarchical schema (keys). This method might be useful if we expect more key/JSON schema variance.

Thirdly, we implemented a recursive algorithm that calculates the exact similarity of two JSON documents as an average of the similarity of the fields (key-value pairs). This algorithm is similar to Jaccard Similarity, yet instead of a binary value for an exact match of the nodes (key-value pairs in our case), we take the recursively calculated similarity of corresponding nodes with the same keys. And we divide the total value by the union size of the fields (nodes). Although this is a very intuitional way of calculating the similarity score, there are some possible caveats. First, when it comes to list-valued key-value pairs (called arrays in JSON), the only easy way to calculate

similarity is via exact Jaccard Distance. However, through our experiments, we have seen that there is a high risk of this leading to information loss and a low similarity score. That is due to the fact that if a significant portion of the JSON document is in another JSON object(s) that is contained in a list, and if the corresponding JSON object in the compared document is slightly different (imagine 1 line different out of 100), all of this information is disregarded, and the obtained similarity score is a binary 0. We tried different methods to avoid this issue, but they were too computationally costly or possibly inaccurate depending on the application. All of this is due to the fact that if the nodes are JSON objects themselves, then the comparison of lists (or arrays) is essentially computationally equivalent to our initial problem of comparison of a database of JSON objects. This means that there are no easy ways to compute their similarity. However, in practice, we may be able to get around this problem by making some assumptions about the sizes of the compared list and the depth of the compared objects.

Apart from that, there is also the issue of all the nodes (key-value pairs) at the same level, having the same weight in the similarity comparison. This means that a key with a single string value will have the same influence as a key with a giant JSON object at the same level. For our purposes, we decided not to perform any specific weight calculations. But this could be done either via pre-computing some information regarding the size and the type of the values or by assuming a set schema for the compared JSON object and assigning weights to its field (manually asked from the user or again computed via algorithm).

Lastly, we implemented a fourth algorithm that gathered the better working parts of the prior three. Here we almost flattened the JSON, but kept the keys and values separate. For each matching key, we compared the values, and if they were not an exact match, we computed a partial similarity score for the node just like in the recursive approach. And we added these partial scores to the final one (That is why we are calling it the partial similarity method). This way, we have avoided loss of information in the nodes with similar but not the same values. We also added a feature to extract

the range information for partial similarity calculation from maximum and minimum values of given JSON schema [13] as an alternative to default and user set methods.

After deciding on the similarity measure, the next step was to design an LSH algorithm for the approximate similarity calculation of multiple (many) JSON documents. In terms of measure, we used the fourth method mentioned above. Furthermore, for the LSH algorithm, one possible candidate was the Recursive Min-wise Hashing method mentioned in [5]. But this was not really applicable our measure of choice and the way we constructed shingles. Because the way we chose to create shingles did not allow for grouping in the form of Multi-Level Exchangeable Representations. We had the option to use the first few levels of hierarchical data, but this would not give distinct enough results since we expect similar schemas for our use case, meaning most of our data would end up in the same bucket(s). Alternatively, we could increase the depth of recursive Min-wise hashing to increase the resolution. However, this would be too computationally costly since the proposed method is only applied 2 (or at max 3) levels of recursion in the paper.

In the end, we went with applying a version of the classical LSH method. We created shingles from partially flattened leaf nodes. Once we had the shingle sets ready, we created a vocabulary including all of the shingles. Using the vocabulary, we encrypted the shingle sets with one-hot encoding. Then, we created randomized hash functions to reorder the data. Using these functions, we created signatures by choosing the minimum indexes in reordered sets. After that, we created an LSH class incorporating the banding technique, where we separated the signatures into given b number of bands. For qualifying as a candidate pair, it suffices to match exactly in only one of the bands. Therefore, we got a higher chance of matching similar items. This technique is quite efficient, and in case one wants to narrow the interval for matching candidates further, there is always the chance of performing an exact similarity search on the filtered candidate pairs. After this, we tested the algorithm with the database and reported results together with graphs.

Furthermore, we implemented a slightly different version of the above code but added Faiss indexing [14] using our signatures. We added 2 more methods for the k approximate nearest neighbor (ANN) search and compared the results with the expected ones from exact similarity scores. Here we also added custom functionality for the exact nearest neighbors and performed extensive testing on the database.

5. REQUIREMENTS SPECIFICATION

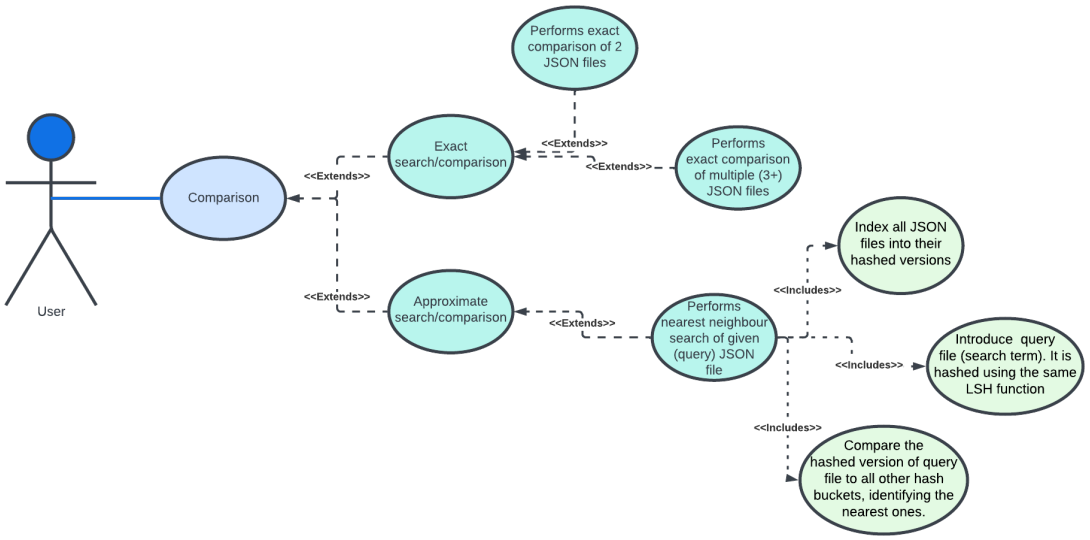


Figure 5.1: Use Case Diagram

- User should be able to perform an exact comparison of 2 JSON files.
- User should be able to perform an exact comparison of multiple (3+) JSON files.
- User should be able to perform an approximate search/comparison of JSON files.
- User should be able to index JSON files with their (LSH) hashed versions.
- User should be able to hash query JSON file with the same LSH functions.
- System should be able to build a similarity index in the linear time range ($O(n)$) (via LSH functions), instead of $O(n^2)$.
- User should be able to perform an exact k nearest neighbor query.
- User should be able to perform a k approximate nearest neighbor query.

6. DESIGN

6.1. Information Structure

ER Diagrams.

6.2. Information Flow

Activity diagrams, sequence diagrams, Business Process Modeling Notation.

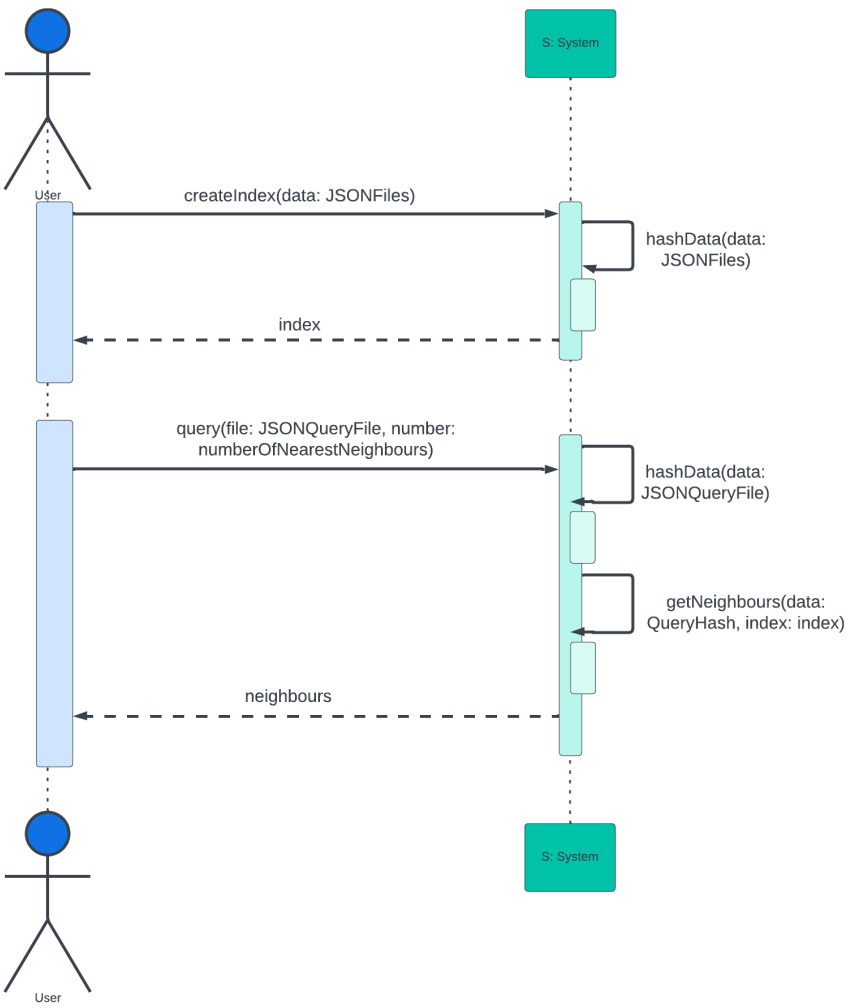


Figure 6.1: Sequence Diagram

7. IMPLEMENTATION AND TESTING

7.1. Implementation

Before continuing with the details of our implementation, we will first mention some implementations and libraries of LSH functions and JSON similarity comparators that we have researched and used.

7.1.1. JSON-Similarity-comparitor

JSON-Similarity-comparitor [15] is a command line tool for comparing JSON files by degree of similarity. It takes two files as arguments and outputs their similarity (between 0.0 and 1.0). It uses the top-down approach, which leads to the hierarchical structure of the file is one of the most significant factors of the similarity score. Here the structure is perceived as a big indicator of the purpose hence it is prioritized. However, the values themselves are also taken into account. For instance, if the same values are stored in a list in one file and in a dictionary in the other, the similarity of the values is detected but penalized because of the structure (accounting for type difference but recursively checking children for similarity).

One caveat of this method is that the top-down approach disregards any similarity between the children nodes once it detects the parents as different. For instance between BreweriesMaster.json and BreweriesSample4.json files (from examples on GitHub); we have 2 dictionaries one with the key "address" and the other "location". Even though the fields of these two are almost the same, all of this is overlooked, and the received similarity score is 2.38 % for the two JSON files. But when I applied a basic line-based set comparison, the Jaccard Similarity turned out to be $30/42 \doteq 71.4$ %. Of course, this information may or may not be desired depending on the application, but I think in many cases incorporating a certain degree of set comparison could prove useful.

Another important issue to mention is that this tool does not make use of any LSH techniques to predetermine candidate pairs. But we can certainly use a mixture of both, either by determining some candidate pairs through LSH and then applying hierarchical comparison to those or possibly by doing the reverse, through incorporating a structure comparison in the initial step and then using LSH for value checks.
I didn't find any benchmarks on this specific tool, but since it compares 2 files at a time (instead of a list like we aim to do), the comparison wouldn't have been very meaningful regardless. This one is rather an example of the usage of the hierarchical structure for comparison.

7.1.2. datasketch

Datasketch [16] is a Python package that provides probabilistic data structures that can process and search large amounts of data faster, with relatively little loss of accuracy. It provides structures like MinHash and Weighted MinHash to estimate the Jaccard similarity of sets and indexes like MinHashLSH, MinHashLSHForest and MinHashLSHEnsemble to provide sub-linear query time. There are also some documentation, example usages, and benchmarks provided with this package.

7.1.2.1. Set Similarity Search. This [17] is also a Python package for calculating set similarity developed by the same person who created 'datasketch'. Even though this package is an exact search algorithm (not approximate LSH like we are planning to use), it is still interesting to observe that when the set sizes are smaller exact search algorithm outperforms the approximate one. "This package includes a Python implementation of the "All-Pair-Binary" algorithm in the Scaling Up All Pairs Similarity Search paper, with additional position filter optimization. This algorithm still has the same worst-case complexity as the brute-force algorithm, however, by taking advantage of skewness in empirical distributions of set sizes and frequencies, it often runs much faster (even better than MinHash LSH)."

7.1.3. ElastiK Nearest Neighbors

Elasticsearch-Aknn (EsAknn) [18], is an Elasticsearch plugin that implements approximate K-nearest-neighbors search for dense, floating-point vectors. This allows data engineers to avoid rebuilding infrastructure for large-scale KNN and instead leverage Elasticsearch's proven distributed infrastructure. Its features include exact nearest neighbor queries for five similarity functions (L1, L2, Cosine, Jaccard, and Hamming); approximate queries using Locality Sensitive Hashing for L2, Cosine, Jaccard, and Hamming similarity; and incremental index updates, start with 1 vector or 1 million vectors and then create/update/delete documents and vectors without ever re-building the entire index.

7.1.4. ANN-Benchmarks

ANN-Benchmarks [19] is a benchmarking tool for approximate nearest neighbor algorithms. It evaluates the performance of in-memory approximate nearest neighbor algorithms. It provides a standard interface for measuring the performance and quality achieved by nearest-neighbor algorithms on different standard data sets. It supports several different ways of integrating k-NN algorithms, and its configuration system automatically tests a range of parameter settings for each algorithm. It is aimed at the data points that are described by high-dimensional vectors, usually ranging from 100 to 1000 dimensions. The following publication details the design principles behind the benchmarking framework: M. Aumüller, E. Bernhardsson, A. Faithfull: ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. Information Systems 2019.

7.1.5. Akin

This [20] is a Python library for detecting near duplicate texts in a corpus at scale using Locality Sensitive Hashing, adapted from the algorithm described in chapter three of Mining Massive Datasets. This algorithm identifies similar texts in a corpus

efficiently by estimating their Jaccard similarity with sub-linear time complexity. This can be used to detect near duplicate texts at scale or locate different versions of a document. This implementation uses the generic version of the algorithm from MMDS and does not take into account the hierarchical structure. Yet, it is a simple enough implementation and is good for practicing basic usage of the algorithm.

7.1.6. Implementation Details

As mentioned in the Methodology section, we began our project by performing a thorough literature survey of existing methods and algorithms for both classical and hierarchical LSH. We performed all of the operations on files with JSON format and chose to convert YAML files to JSON format as well, which we could do without loss of hierarchical information.

The initial step of implementation was designing and creating a function for calculating the exact similarity score of two JSON documents. For this, we discussed several different approaches and implemented 4 of them.

The first method flattens the JSON files according to leaf nodes. While doing so, it also concatenates leaf nodes with a path consisting of keys directing to it. Hence we get to incorporate the hierarchical structure information.

The second method is actually a variation of the first one. Here, we add the paths consisting of keys as nodes to the flattened set as well. By doing so, we put more emphasis on the structure (schema) of the JSON files. As expected, this resulted in higher similarity scores in the test files, since we mentioned dealing with highly similar JSON objects with the same (or similar) schema. This could be more or less desirable depending on the application domain. For our purposes, the first algorithm is more suitable as it puts more emphasis on the leaf nodes.

The third method implements a recursive algorithm that calculates the exact sim-

ilarity of two JSON documents as an average of the similarity of their fields (key-value pairs). This algorithm is similar to Jaccard Similarity, yet instead of a binary value for an exact match of the nodes (key-value pairs in our case), we take the recursively calculated similarity of corresponding nodes with the same keys. And we divide the total value by the size of the union (number of fields).

Finally, we implemented a fourth method that is a mixture of the first and the third one. This one follows the flattened approach of the first method but incorporates the partial comparison quality of the third one. That is, the fourth method takes into account the partial similarity of the fields. Say corresponding fields in 2 different JSON documents have the same keys (path of keys), but the values are not precisely the same. In this case, we calculate the partial similarity score of the values and use it in the total score. The partial similarity score is calculated as Edit (Levenshtein) Distance for strings and as the ratio of the difference between the values to the range for numbers. For this case, we have added automated minimum and maximum data pulling from the given JSON Schema(hence range calculation), as well as the option to set default values.

All of the mentioned methods have their strong and weak points. The third one leads to some data loss for fields in the form of arrays of JSON objects and requires special attention for weight distribution while calculating recursive similarity. The first two algorithms are alright in these regards, but they lead to loss of easy access to grouping information, which may be problematic when combined with the Recursive Min-wise hashing approach mentioned in [5]. These issues are discussed in detail in the Methodology section, and we will again mention them in the test results. Considering everything, we have decided to use the fourth implementation as our exact comparison of choice.

We researched several techniques for the approximate (LSH) portion of the implementation. [21] The Recursive Min-wise hashing approach mentioned here [5] was a strong contender, but as mentioned in the Methodology section, it could not be directly

applied with our distance measure of choice. Hence we went ahead with a modified version of the min-wise hashing and banding technique while retaining the structural information in the way we created shingles.

As for the implementation details of the LSH part, firstly, we created shingles from each JSON document. For this, we used both the values and the concatenated paths (separately). Then, we built a vocabulary from the shingle sets. Using the vocabulary as a guide, we built one-hot encodings for each JSON file. After that, we created min-hash signatures for each JSON file using random permutations of the one-hot encodings. Once we had signatures, we created an LSH class to store our buckets (and operation functions). For this, we also made use of the Faiss Guide [10]. Here, we incorporated the banding technique as well. This is a technique where we separate the signature into a given number of sections (bands) and compare the bands exclusively instead of the entire signature. Even though this raises the risk of false positives, we get a higher chance of matching similar texts (signatures) as candidate pairs. Moreover, we always have the option to control the effect of this technique by setting the number of bands.

At this point, we have built our index and can query the candidate pairs. In order to see the accuracy of the method, we have generated graphics with candidate pair versus exact similarity scores for some of the most commonly used similarity measures as well as our custom partial similarity score. We will analyze these results in detail in the upcoming section.

7.2. Testing

We tested our algorithms with the Autonomous Driving Scenario files [1] mentioned before. Originally, these files were in YAML format, we converted them to JSON for ease of use. Since the first milestone, we have also converted the entirety of the scenario files to test on. We also generated a schema in accordance with all of the JSON files using GenSON [22], to use while testing the pulling maximum/minimum

information from a schema feature. Below are the comparison results for 4 of these files. We have also generated a file with the exact similarity scores of all of the JSON files, but have not included here to avoid redundancy, one can find it in the GitHub repository under the name "matrixresult.txt".

- 1 = UC-001-0001-Kashiwa
- 2 = UC-001-0001-Shalun
- 3 = UC-001-0002-Kashiwa
- 4 = UC-001-0002-Shalun

Table 7.1: Similarity Scores for Different Implementations

	Flat1	**Flat2**	**Recursive**	**Jaccard**	**Partial Sim**	**Json-sc [15]**
1 & 2	0.55725	0.69295	0.31754	0.797	0.77887	7.14%
3 & 4	0.61053	0.73864	0.33369	0.825	0.82402	5.35%
1 & 3	0.54878	0.60759	0.85862	0.883	0.64204	10.89%
1 & 4	0.41436	0.49271	0.31741	0.798	0.52895	5.03%

7.2.1. Exact Similarity

As we can see from table 7.1, Flattened 1 and Flattened 2 algorithms behave in a similar way with Flattened2 having an overall higher score due to the increased influence of the structure of the schema in the second one. On the other hand, the Recursive algorithm results in an overall lover score because of ignored similarity (taken as boolean 0) of the JSON objects that are in an array. This algorithm also leads to some unexpectedly high results due to the unadjusted weights of the fields in the same hierarchical depth.

Jaccard Set Similarity gives quite high scores, as expected since it disregards any hierarchical structure information. Also, common keys are all taken as nodes which has a huge positive impact on the similarity score but takes away from the distinguishing

ability. The JSONComparitor [15] gives quite low scores, but the overall trend is consistent with the expected similarities, with slightly lower scores for files with more recursive dept. This situation is likely due to the top-down approach ignoring some lower-level similarities.

Finally, the Partial Similarity algorithm is another variation of the Flattened approach but just as in the Recursive one; it considers the partial similarities of fields with the same keys that are not a perfect match. Hence the overall Partial Similarity scores are higher than their Flattened counterparts. However, the trend about which pairs of files have higher similarity is the same as Flattened ones. This is to be expected since the overall approach of the algorithm is the same, just more refined with the inclusion of the partial similarities. [1]

7.2.2. LSH

As we mentioned before in Section 7.1.6, for the approximate similarity calculation part of the project we have used a variation of the min-wise hashing and banding techniques. Algorithm details are explained in detail in Section 7.1.6. We have generated some graphs to make the results more visual and easy to understand. Below we share these along with their explanations.

Figure 7.1 shows the theoretical Probability versus Similarity curves for different values of b (number of bands). The formula is calculated in the following way $p = 1 - (1 - s^r)^b$, where s is the similarity of the two documents, b is the number of bands and r is the number of rows in each band. (Assuming that the probability of the minhash signatures agreeing in one particular row of the signature is s.) This theoretical phenomena is explained in [4] in the following way; “

(i) The probability that the signatures agree in all rows of one particular band is s^r.

(ii) The probability that the signatures disagree in at least one row of a particular

[1] JSON files used for the testing results, can be found here [1] in the YAML format. The names are as mentioned before.

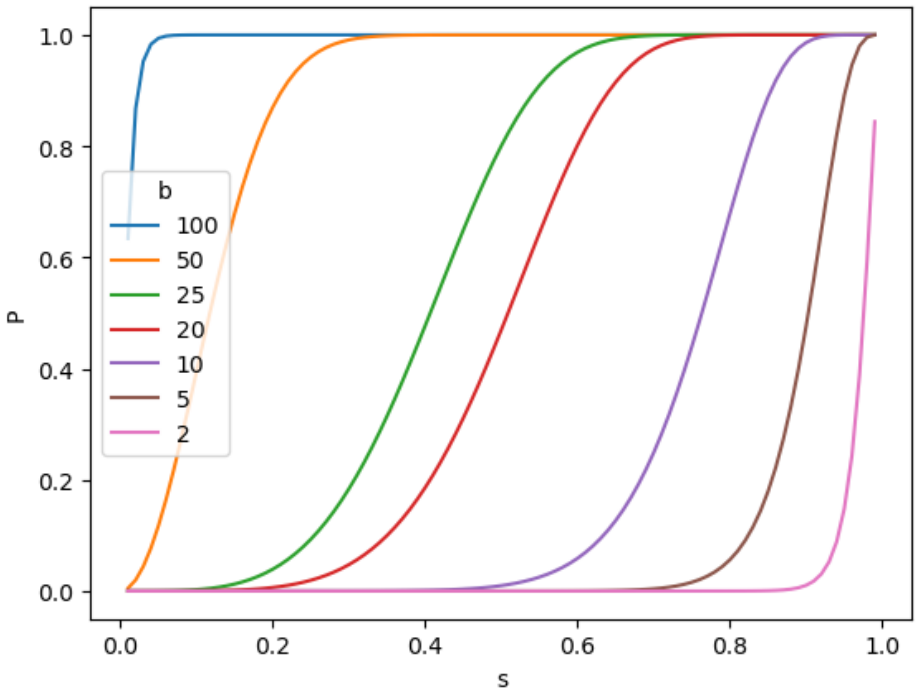


Figure 7.1: Probability (p) vs. Similarity (s) relationship for different values of b. ($p = 1 - (1 - s^r)^b$) (for signature length = 100, r = 100/b)

band is $1 - s^r$.

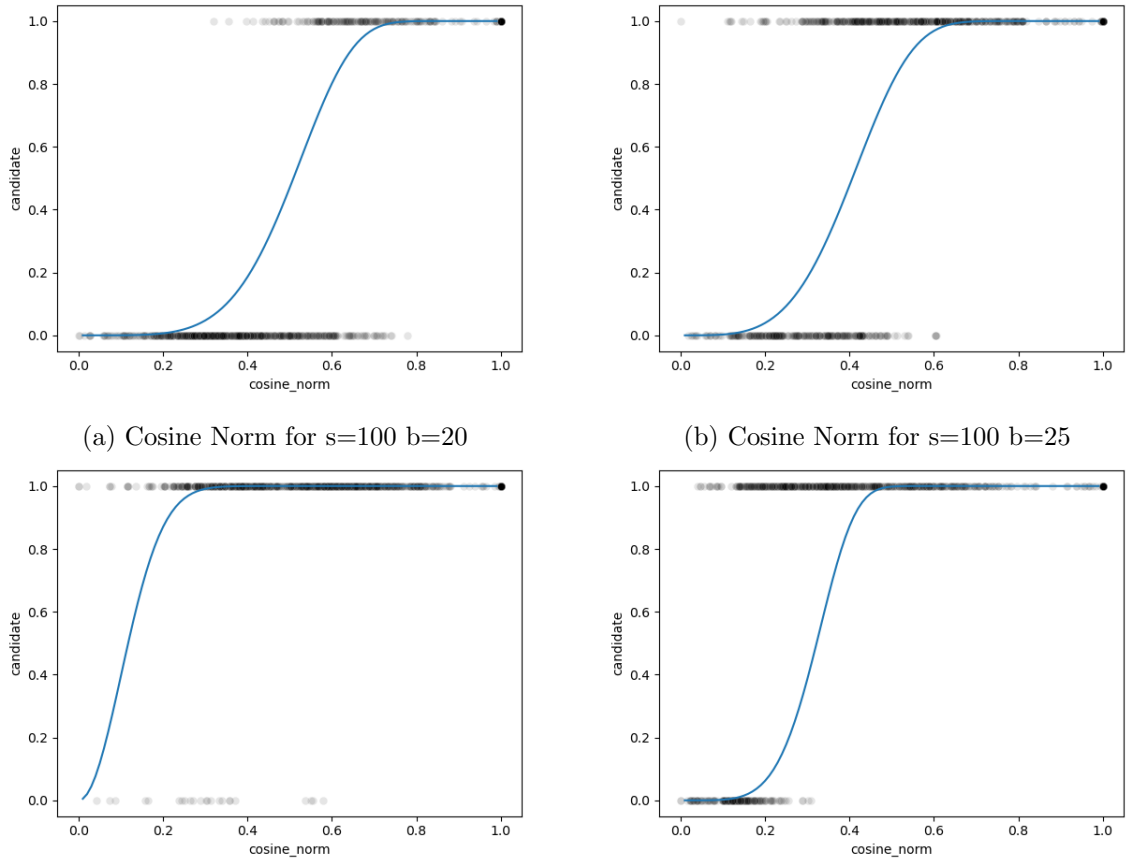
(iii) The probability that the signatures disagree in at least one row of each of the bands is $(1 - s^r)^b$.

(iv) The probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, is $1 - (1 - s^r)^b$.

It may not be obvious, but regardless of the chosen constants b and r, this function has the form of an S-curve. The threshold, that is, the value of similarity s at which the probability of becoming a candidate is 1/2, is a function of b and r. The threshold is roughly where the rise is the steepest, and for large b and r, we find that pairs with similarity above the threshold are very likely to become candidates, while those below the threshold are unlikely to become candidates – exactly the situation we want. An approximation to the threshold is $(1/b)^{1/r}$."

Below we will share some graphs showing the candidate status of the file pairs versus their similarity score. We have calculated similarity scores according to some

of the most commonly used distance measures as well as our custom one to provide a reference.



(a) Cosine Norm for s=100 b=20

(b) Cosine Norm for s=100 b=25

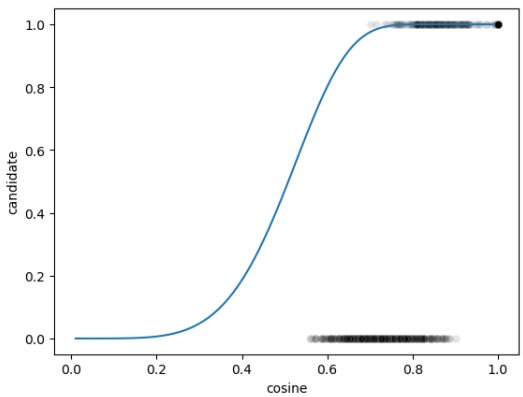
(c) Cosine Norm for s=100 b=50

(d) Cosine Norm for s=1000 b=200

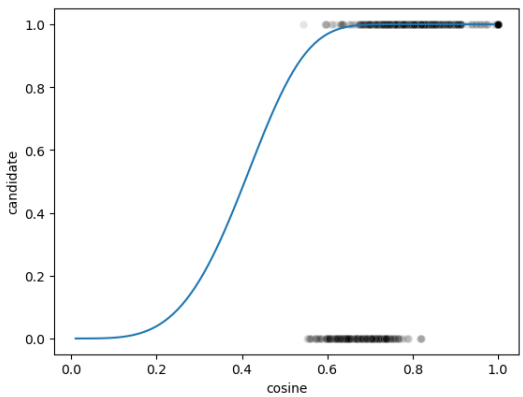
Figure 7.2: Cosine Norm Similarity vs. Candidate Pairs

Figure 7.2 displays the Cosine Norm Similarity scores of file pairs versus whether these pairs were chosen as candidate pairs by the LSH algorithm or not (black scatter plot). The blue curve is the theoretical result with respect to different b and r values. As one can see, figure 7.2 shows the results with 4 different signature length (s) and number of band (b) combinations. Increasing the signature length provides us with a more accurate representation however it also increases the memory required. Hence can be costly, especially while operating with a larger number of documents. It can also lead to an increased number of false positives while keeping the number of rows in each band the same (E.g. (a) and (d) in figure 7.2). This is simply because we have more of the same length intervals, meaning a bigger chance of finding a match.

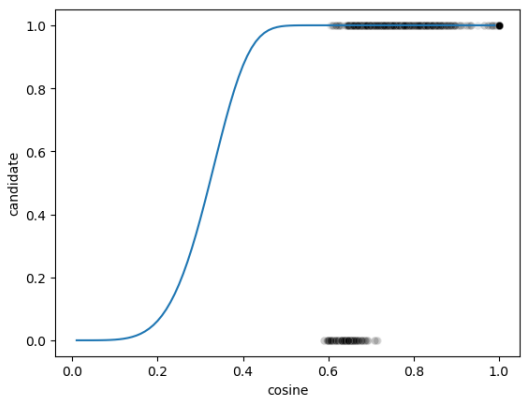
Increasing the number of bands, on the other hand, means smaller number of rows that need to match exactly to be considered as a candidate. While this increases the probability of catching similar files as candidates, it can also increase the number of false positives (meaning a left shift of the theoretical curve). This is a trade-off that one must consider depending on the application domain and the desired result. In our implementation, these changes can be easily made by adjusting the b and signature length parameters.



(a) Cosine for s=100 b=20



(b) Cosine for s=100 b=25

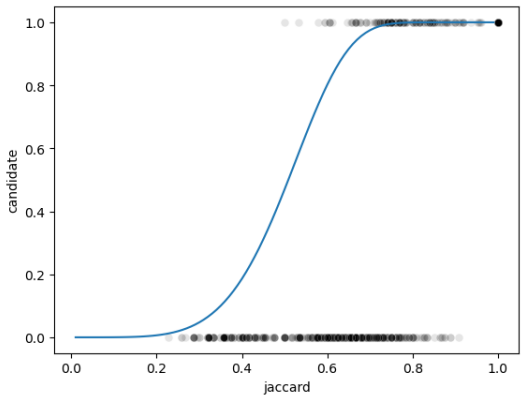


(c) Cosine for s=1000 b=200

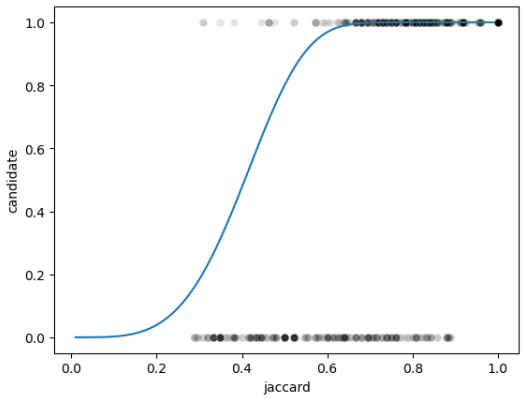
Figure 7.3: Cosine Similarity vs. Candidate Pairs

Figure 7.3 displays the cosine similarity versus candidate status of the pairs of documents. This is similar to the previous figure except this time we have not normalized the similarity scores. This allows us to make the following observation; In almost all cases pairs of files have more than 50% similarity. But this is expected with the similarity of measure (cosine) since we are using a highly similar data set for our experiments. This is one of the main reasons why we have decided to develop our own

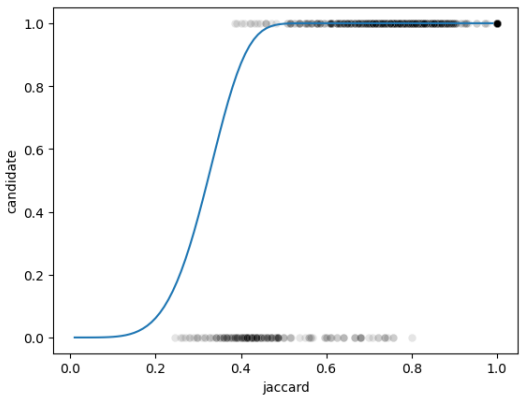
measure of similarity.



(a) Jaccard for s=100 b=20



(b) Jaccard for s=100 b=25

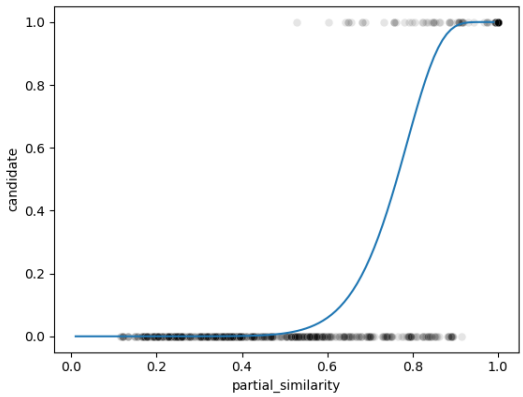


(c) Jaccard for s=1000 b=200

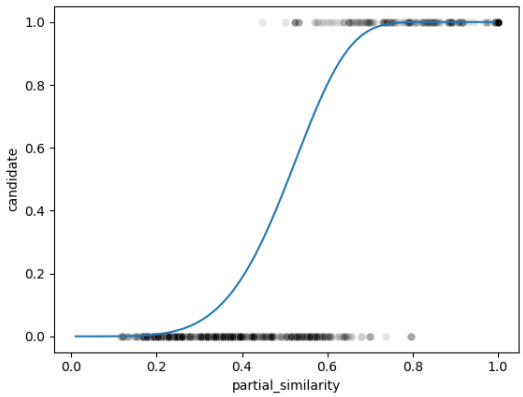
Figure 7.4: Jaccard Similarity vs. Candidate Pairs

Similarly, Figure 7.4 depicts Jaccard Similarity versus candidate status of the pairs of documents. Our analysis about changing the b value apply here as well. Furthermore, the statement about increased signature length (hence resolution) leading to more accurate representation is more clear here. We can see that, from the decreased rate of false negatives in (c) of figure 7.4 compared to (a). (No rejections after 0.8 similarity in (c) while there exist several such points in (a)).

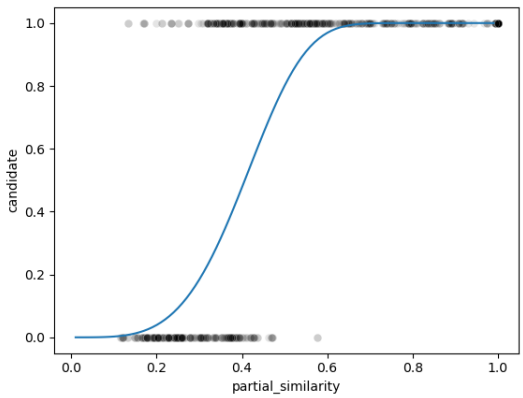
Finally, Figure 7.5 displays the Custom Partial Similarity scores of pairs of documents versus their candidate status. This set of graphics is the most significant for us since this is our similarity measure of choice. We have used the other more commonly used ones to explain the overall trends and to make comparisons. Yet, our overall goal was to distinguish the documents as accurately as possible (within a reasonable time



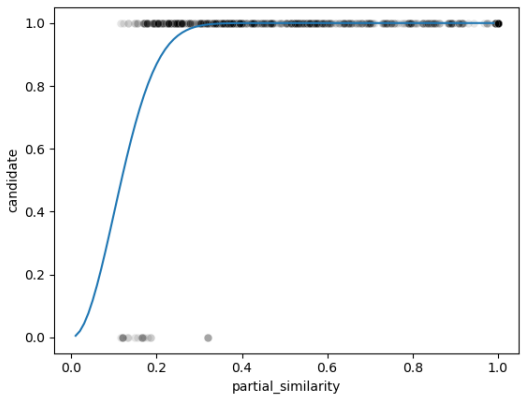
(a) Partial for s=100 b=10



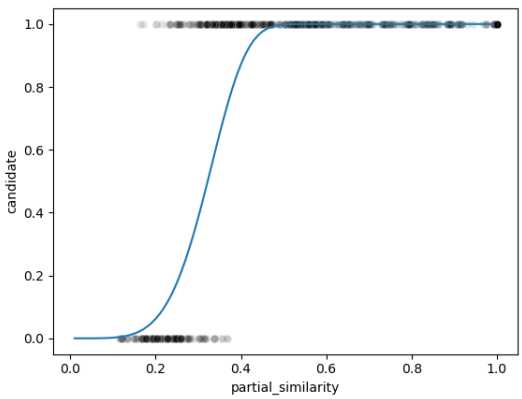
(b) Partial for s=100 b=20



(c) Partial for s=100 b=25



(d) Partial for s=100 b=50



(e) Partial for s=1000 b=200

Figure 7.5: Custom Partial Similarity vs. Candidate Pairs

frame) according to this one.

Similar to the previous cases, we observe the trend of increased rates of candidate status with the increased number of bands (b). We also see that rate of false negatives decreases significantly, as we increase the signature length (s). However, one must

keep in mind that on many occasions LSH -or approximate similarity measures in general- are used for refining (filtering) the data, followed by exact similarity calculation performed on candidate pairs. In this regard, it might be useful to select a number of bands (b) that allows for pairs with less than desired similarity scores to be chosen as candidates. This way, we will have a higher chance of the desired ones being selected. Then, we can perform exact similarity calculation and eliminate the ones we don't want. Nonetheless, these decisions highly depend on the application domain and our aims, meaning an approach that works well in one application might not perform as well in another. So, it is important to understand the purpose of each variable and customize it accordingly.

8. RESULTS

As discussed in the previous chapter, we have performed a thorough literature survey for classical and hierarchical LSH methods and tools. We have also designed and implemented some algorithms for the exact comparison of JSON files. Different algorithms perform better depending on the application's needs. We have also developed some algorithms for approximate (LSH) similarity comparison and implemented them. We have also implemented the k nearest neighbor search using 3 different approaches. Below are the results.

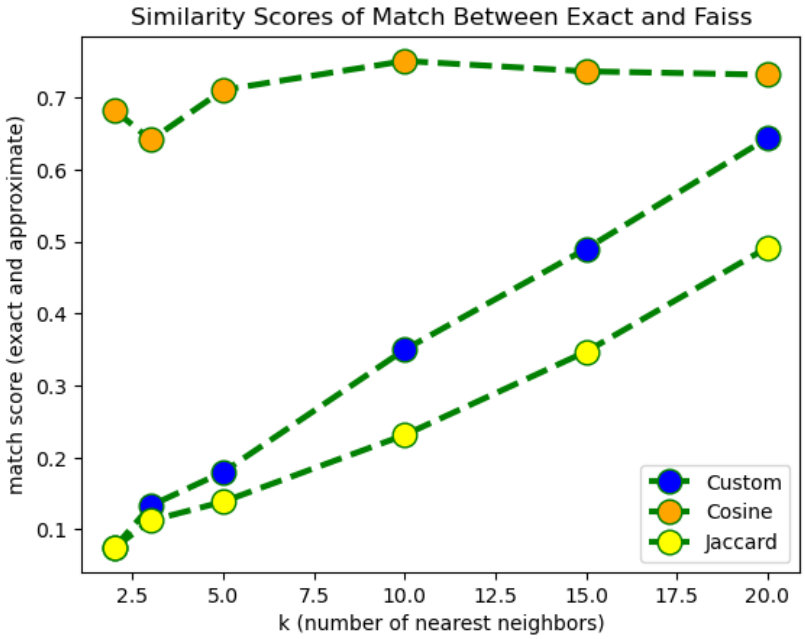


Figure 8.1: Similarity scores of the approximate and exact k nearest neighbor queries

Figure 8.1 displays the similarity scores of the approximate and the exact k nearest neighbor (NN) queries for different values of k. Here approximate nearest neighbor query is performed using a Faiss index created from signatures of the JSON files. Unfortunately, this method did not prove to be very accurate (as one can see from overall low match scores). This is due to the fact that the Faiss index used, measures similarity from vectors. And the difference in the values of minhash signatures rows is not really representative of the similarity. That is, [1,2] is not necessarily more similar to [3,2] than [81, 49].

Another point to mention about figure 8.1 is that there are 3 different measures of similarity used to calculate the similarity (match) score between the approximate and the exact k nearest neighbor query result. These are Custom, Cosine, and Jaccard; color-coded in blue, orange, and yellow. As we will also observe in the upcoming graphs, Cosine similarity is not an accurate measure for comparing NN results, since it gives very high results even for very dissimilar NN comparisons. From manual testing as well as the batch results, we saw that it gives more than %50 similarity for two lists with barely anything in common. The Jaccard similarity is pretty accurate to expectations, yet it is known to give relatively low scores with more emphasis on the uncommon (not intersection) part of the lists. That's why we have implemented a custom similarity measure inspired by Jaccard to compare NN results. In the Custom similarity measure, we use the maximum of the list lengths as the denominator instead of the union, which is in fact equal to any one of the list lengths since they are always equal (k).

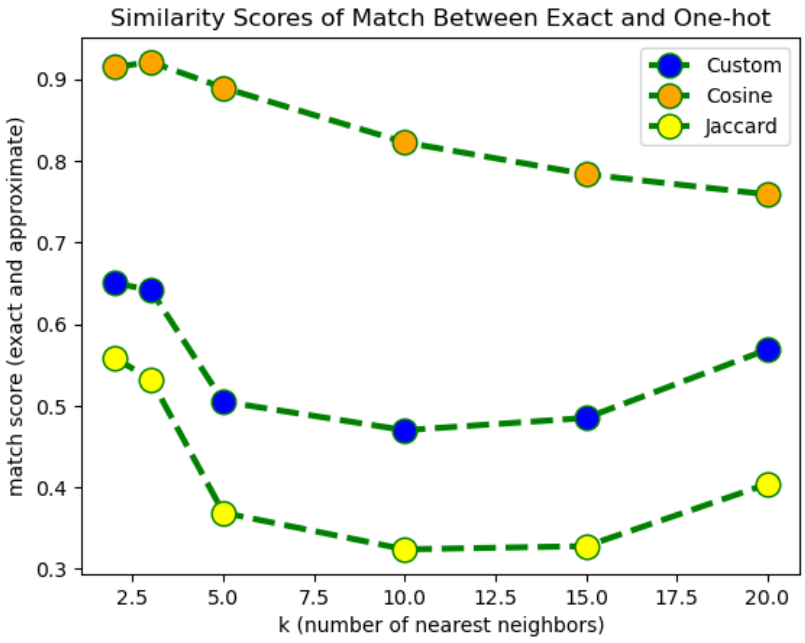


Figure 8.2: Similarity scores of the approximate and exact k nearest neighbor queries

Figure 8.2 shows an overall trend of increasing match score with k, the initial high scores are likely the cause of query vector finding itself in the index. The graph is similar to 8.1, but this time instead of the minhash signatures themselves we have used a signature consisting of boolean values indicating whether the one-hot encoding of the

JSON file in the given random minhash location (the first one) was filled or not. We did this to avoid the distance calculation inaccuracy mentioned in the previous case, and although this solved one problem, it created another. The signatures created this way turned out to be quite sparse, meaning they contained very little distinguishing information. This was actually eye-opening to the fact of how useful the classical minhash algorithm was in representing the information accurately in small dimensions. In order to achieve the same level of accuracy with the current method, we would have to increase the signature length quite a bit. This would be very counterproductive since one of the main motivations of this whole ordeal is reducing dimensionality.

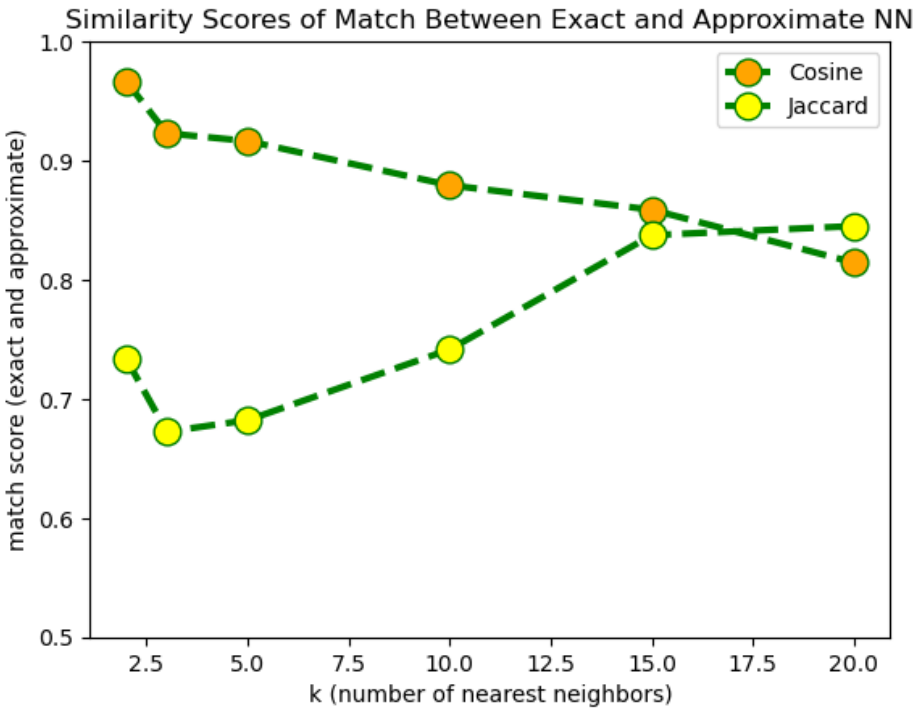


Figure 8.3: Similarity scores of the approximate and exact k nearest neighbor queries

Figure 8.3 again shows the similarity scores of the approximate and the exact k nearest neighbor (NN) queries. However, this time instead of using a Faiss index we have created a custom function to compute k nearest neighbors of the query item using minhash signatures. We have seen that this method turned out to be the most accurate out of all 3 (according to all 3 measures). The graph 8.3 shows the similarity (match) scores according to Cosine and Jaccard measures, per usual, the Cosine score is quite high while Jaccard is lower. But they seem to get close relative soon this time.

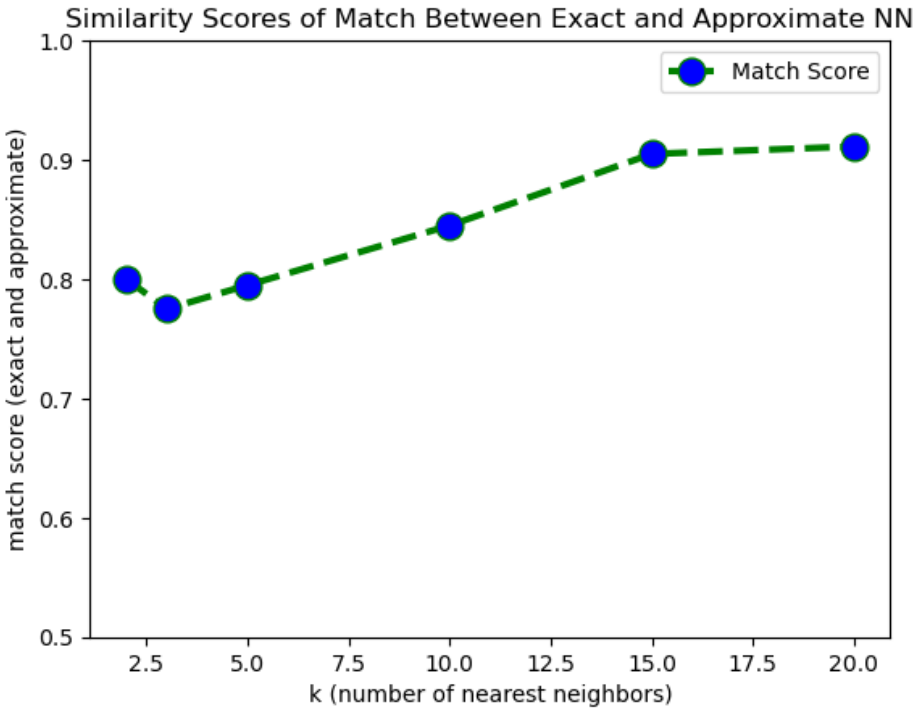


Figure 8.4: Similarity scores of the approximate and exact k nearest neighbor queries

Finally, figure 8.4 displays the similarity scores of the approximate and the exact k nearest neighbor (NN) queries, using the Custom method for the approximate case. And the measure of choice is the custom one. As we have explained through previous graphs this measure is the most accurate one And as one can judge by the high similarity scores the custom method of calculating ANNs is the most successful. Hence this graph represents the results of the LSH algorithm most accurately. The score increases with the number of neighbors k since the effect of small local differences in the order is compensated this way. Overall, the algorithm is successful with a match score above 0.8, meaning the exact and the approximate algorithms returned at least 8 out of 10 the same results. This method can be further refined by increasing the resolution (signature length), but this comes with an increase in computation and memory cost.

9. CONCLUSION & FUTURE WORK

LSH is an indispensable tool for many data mining applications in the age of big data. Hierarchical LSH is a natural extension of LSH for making use of the information embedded in the text structure (or in our case JSON Schema). We have investigated many state-of-the-art techniques and developed some of our own according to our application needs. We have implemented several exact search algorithms and an approximate (LSH) one. We have also implemented a few ANN (Approximate Nearest Neighbor) search algorithms and an exact one to compare. The LSH algorithm was quite successful at detecting high-similarity pairs as candidates with a few low-similarity ones to spare. And the Custom ANN algorithm was successful at matching exact nearest neighbors with rate = 0.8 for lower values of k (number of NN) and 0.9 for the higher values.

The subject of LSH has a vast application domain and great potential for new, creative methods. One can always come up with new ways of calculating (approximate) similarity depending on application needs (such as stop-word shingling for news articles). In the context of our project, the algorithms can be generalized to cover more hierarchical datatypes, and using this generalized version, one can test on bigger datasets, and compare with other hierarchical LSH algorithms. Another experiment could be done on the shingling. One can come up with alternative ways to create shingles, an example would be creating mini trees from each node and its immediate children, these nodes could be used as shingles to represent the JSON file. In theory, this approach protects the structural information while allowing for more elasticity than taking entire paths as shingles. However, it also has the risk of putting too much emphasis on the schema rather than leaves, which is undesired in our application domain. One can never make sure without trying. As we have said before, possibilities are endless!

REFERENCES

1. "Autonomous Driving Scenarios", `https://gitlab.com/autowarefoundation/operational-design-domains/-/tree/master`, accessed: 2023-03-27.

2. "Facebook AI Similarity Search", `https://github.com/facebookresearch/faiss`, accessed: 2023.

3. Wu, W. and B. Li, "Locality Sensitive Hashing for Structured Data: A Survey", , 2023.

4. Leskovec, J., A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*, Cambridge University Press, second edn., 2014, `http://mmds.org`.

5. Chi, L., B. Li and X. Zhu, "Context-Preserving Hashing for Fast Text Classification", pp. 100–108, 04 2014.

6. Gollapudi, S. and R. Panigrahy, "The power of two min-hashes for similarity search among hierarchical data objects", pp. 211–220, 06 2008.

7. Cohen, S., "The Earth Mover's Distance (EMD)", , 1999.

8. Schönauer, S., "Efficient Similarity Search in Structured Data", , 02 2004.

9. Broder, A., "On the resemblance and containment of documents", , 1997.

10. "Faiss: The Missing Manual", `https://www.pinecone.io/learn/faiss/`, accessed: 2023-03-30.

11. Gionis, A., P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing", *Vldb*, Vol. 99, pp. 518–529, 1999.

12. Wahyudi, E., S. Sfenrianto, M. J. Hakim, R. Subandi, O. R. Sulaeman and R. Setiyawan, "Information Retrieval System for Searching JSON Files with Vector Space Model Method", *2019 International Conference of Artificial Intelligence and Information Technology (ICAIIT)*, pp. 260–265, 2019.

13. Machado, F. T., D. de Brum Saccol, E. Piveta, R. Padilha and E. Ribeiro, "A Text Similarity-based Process for Extracting JSON Conceptual Schemas.", *ICEIS (1)*, pp. 264–271, 2021.

14. "How to Build Your First Similarity Search", `https://medium.com/loopio-tech/how-to-use-faiss-to-build-your-first-similarity-search-bf0f708aa772`, accessed: 2023.

15. "JSON-Similarity-comparitor", `https://github.com/Geo3ngel/JSON-Similarity-comparitor`, accessed: 2023.

16. "DataSketch Library", `https://github.com/ekzhu/datasketch`, accessed: 2023-03-30.

17. "Set Similarity Search", `https://github.com/ekzhu/SetSimilaritySearch`, accessed: 2023.

18. "ElastiK Nearest Neighbors", `https://github.com/alexklibisz/elastiknn`, accessed: 2023.

19. "ANN-Benchmarks", `https://github.com/erikbern/ann-benchmarks`, accessed: 2023.

20. "Akin", `https://github.com/justinbt1/Akin`, accessed: 2023.

21. "Semantic Search", `https://www.pinecone.io/learn/semantic-search/`, accessed: 2023.

22. “GenSON”, https://github.com/wolverdude/GenSON, accessed: 2023.

APPENDIX A: APPENDIX

GitHub Repository : https://github.com/BElifb/CMPE492_PROJECT