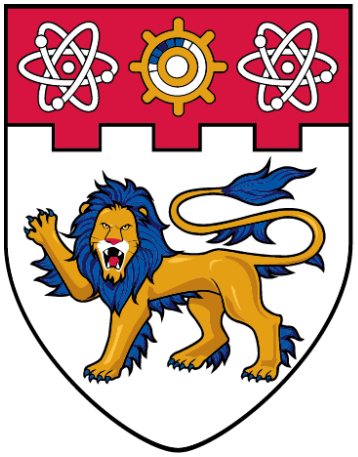


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4032 Data Analytics & Mining

Group 26 - Project 2 Report

Adrian Goh Jun Wei	U1721134D
Lim Jin Yang	U1921682L
Tan Hu Soon	U1921282A
Vivien Chew Jing Wen	U1922718D

Similarity Search

Group 26
School of Computer Science and Engineering
Nanyang Technological University
Singapore

ABSTRACT

We will be focusing on 2 algorithms, Locality Sensitive Hashing (LSH) and TF-IDF. To understand how these algorithms solve the document comparison problem we need to dig deeper into both of these algorithms, and compare how they perform in 2 key aspects: processing speed (time) and memory (space).

—

I. INTRODUCTION

Large scale data comparison has become a regular need in today's industry as data is growing by the day. These kinds of applications are usually useful to the News Agencies where they need to recognize that a group of articles are really all based on the same article about one particular story. Other usage could be Plagiarism detection, mirror page identification etc. Suppose you have 10K documents to compare similarity between them. This task will take $O(n^2)$ operations (i.e. ~ 100 million operations) if you try the brute force approach - compare each document with every other document in the corpus. Clearly you need a better approach. Our problem boils down to one fact - find the near similar documents of a query document - in other words - find the nearest neighbours.

Similarity search is a crucial task in multimedia retrieval and data mining. The recent information-based problematic issue is the rapid data growth. Text similarity measurement is a text mining approach that could overcome this problem. Finding the similarity between words is a primary stage for sentence, paragraph and document similarities.

Our goal of this paper is to analyze the similarities among large collections of documents. Experimental analysis and comparison will be done across algorithms and datasets. We will be focusing on 2 algorithms, Locality Sensitive Hashing (LSH) and Term-Frequency Document-Inverse-Frequency (TF-IDF). To understand how these algorithms solve the document comparison problem we need to dig deeper into both of these algorithms.

II. RELATED WORK

Finding similarities among large collections of documents is closely related to three other matching problems: software component matching, schema matching and finding similar web service operations.

Software component matching: Software component matching is considered important for software reuse. It formally defines the problem by examining signature (data type) matching and specification (program behaviour) matching. The techniques employed there require analysis of data types and post conditions, which are not available for text similarity measurement.

Schema matching: The database community has considered the problem of automatically matching schemas [24, 12, 13, 22]. The work in this area has developed several methods that try to capture clues about the semantics of the schemas, and suggest matches based on them. Such methods include linguistic analysis, structural analysis, the use of domain knowledge and previous matching experience.

Finding similar web service operations: In other words, this means to find operations with similar functionalities. For example, the web-service operation "GetWeather" is similar to the operation "GetTemperature". There is no formal definition for operation similarity, because, just like in other types of search, similarity depends on the specific goal in the user's mind. Intuitively, we consider operations to be similar if they take similar inputs, produce similar outputs, and the relationships between the inputs and outputs are similar.

Our work is to perform text mining to determine the similarities across documents. Most solutions to this problem of text document matching are based on term frequency analysis. In this paper, we will be discussing LSH and TF-IDF.

III. PREPROCESSING

In order to compute the similarity, we first have to make a vocabulary of unique words. We will perform removal of stop words and case folding, before doing tokenization.

- Stopwords are the most common words in any natural language. In English, the most common stop words are *"the", "is", "in", "for", "where", "when", "to", "at"* etc. For the purpose of analyzing text data, these stopwords might not add much value to the meaning of the document. In other words, we can say that the removal of such words does not show any negative consequences on the model we train for our task. Removal of stop words can reduce the dataset size and thus reduces the training time due to the fewer number of tokens involved in the training.
- Case folding is the act of reducing all letters to lowercase. This is necessary because we want to group the same word in order to calculate the word frequency. Without case folding, it will result in

wrong frequency count. For example, "*JavaScript*", "*Javascript*" and "*javascript*" will be counted as 3 distinct words.

IV. METHODS

1. LOCALITY SENSITIVE HASHING (MIN HASHING)

Locality-sensitive hashing (LSH) is an algorithmic technique that hashes similar input items into the same "buckets" with high probability. As a result, data points near each other are located in the same buckets with high probability, while data points far from each other are likely to be in different buckets. This makes it easier to identify observations with various degrees of similarity. Since our goal is to find 'Near Duplicate' pairs, we can break down the LSH algorithm into 3 broad steps: Shingling, Min Hashing and Locality-sensitive hashing (also known as Band and Hash).

A. Shingling

Shingling can be thought of as tokenizing texts. However, this tokenization process differs from normal tokenization in such a way that it does not tokenize according to words but according to the number of characters defined.

Shingling is the most effective way to represent documents as sets. A shingling function was created to generate the k-word shingles of a single document and return the distinct set of them (value of k adjustable by user). Any substring of length k is defined as a k-shingle. Shingles are produced following a sliding window rule of k size.

The function was applied across all documents in order to construct for each one of them the set of -- shingles that appear at least once within that document. The Pseudocode for the shingling component is as follows:

----------**Start**----------

1. text = cleaned document text without special characters
2. shingle_set = []
3. for i in range(1 to len(text)-(k+1)):
4. shingle_set.append(sliced text of length k)
5. return set(shingle_set)

----------**End**----------

B. Min-Hashing

Minhashing is an important step in the efficient computation of document similarity. Having split the initial document into shingles, the Jaccard similarity can be calculated. However this will not scale well, as for a given document the retrieval of all other documents that are the closest/most similar to it requires heavy computations.

Min-Hashing is applied here in order to improve performance and relieve the computational burden. It allows us to convert our sparse vectors into dense vectors, reducing the memory needed while capturing the key features of our sparse vectors, retaining a good level of information through the transformation process. This method proposes that for each set of k-shingles derived from their respective documents, they are to be replaced by a number or otherwise known as a signature through a fixed transformation process. The process will be applied to all the documents we are interested in comparing.

The Pseudocode for the Min-Hashing function is as follows:

----------**Start**----------

1. for i in range(number of Min-Hash vectors):

2. hash_function = list containing int 1 to len(vocab)+1

3. hash_function = shuffle elements in hash_function

4. hashes.append(hash_function)

5. for each hash_function in hashes:

6. for j in range(number of elements in hash_function):

7. position = index of j in hash_function

8. if SparseVector[position] == 1:

9. signature.append(j)

10. break

11. Return signature

----------**End**----------

The choice of the minimum hash value is actually a rule of thumb and is equivalent to the random choice, as for each input a good hash algorithm is supposed to generate a "random" output that is equally likely to be large or small.

Regarding the parameter choice, a and b were integers sampled without replacement from a uniform distribution with range [1, N], where N stands for the number of distinct shingles across all documents. In addition, c was set equal to the first prime number greater than the total number of unique shingles.

Furthermore, in order to avoid integer overflow during $a \cdot x$ multiplication, the following properties were applied on the hash functions of the form:

$$h(x) = (a \cdot x + b) \bmod c:$$

$$[(x \bmod w)+(y \bmod w)] \bmod w$$

$$= (x + y) \bmod w\ [(x \bmod w) \cdot (y \bmod w)] \bmod w$$

$$= (x \cdot y) \bmod w$$

Which resulted in the following transformation:

$$h(x)=[(a \bmod c) \cdot (x \bmod c) \bmod c + (b \bmod c)] \bmod c$$

C. Locality Sensitive Hashing (Band and Hash)

Minhashing can be used to significantly reduce the number of calculations required for the comparison of two documents. However, in order to retrieve similar documents, all-pairs computation is still necessary, which is quadratic in the number of signatures. To overcome this obstacle, the Locality Sensitive Hashing (LSH) method is used. The main idea is to create mini-signatures, i.e. b bands containing r signatures.

Each mini-signature is then hashed to a unique bucket via an appropriate hash function. Thus, candidate pairs are derived from documents mapped to the same bucket for at least one band.

The following steps were implemented across all documents:

a. For the chosen band size (further explained under part F of LSH), the document's signatures were split into separate mini-signature vectors (bands).

b. Each band vector was then transformed into a unique string, constructed from the concatenation of all members, with underscore as separator. For example, a band containing the signature values (15, 355, 89, 2) would generate the following string: "15355892"

c. Each string was hashed to a unique hexadecimal number (bucket) by using the respective hash functions for that band. Eventually, all documents were mapped to a specific bucket per band. Thus, all information necessary for the search of nearest neighbors was available at this point.

D. Strength And Weaknesses

Locality Sensitive Hashing can be applied to several problem domains, near-duplicate detection, large-scale image search, and audio/video fingerprinting. The task of finding nearest neighbours is very common. Although using brute force to check for all possible combinations will give you the exact nearest neighbour, it's not scalable at all. Therefore, approximate algorithms to accomplish this task are more utilised. Although these algorithms don't guarantee to give the exact answer, more often than not they'll provide a good approximation. These algorithms are faster and scalable. LSH is one of these algorithms.

However, LSH by design produces "false positives" hash collisions that are not similar at all. This is due to having a larger number of hash functions (bands). This means that we are increasing the probability of finding a candidate pair and that is also equivalent to taking a smaller similarity threshold. This trade-off in increasing hash functions will result in having lesser accuracy when determining similarity between documents.

To remove the false positives you do a mini brute-force search of only those candidate documents and you find your final answer. If false-negatives are a big problem you can do this "n" times having "n" hash tables, so you go to "n" buckets to pick-up candidates instead of just one bucket, this makes you check more-documents hence more-computation but reduces the chances of missing a near neighbor. So LSH is an approximation but is very fast as you only have to check the documents inside the bucket or buckets, in general this is very close to O(1).

E. Key factors that affect the performance

The performance of the Locality Sensitive Hash function can be quantified by the time and space taken by the program to finish its intended process. One of the key components that makes the algorithm efficient is Min-Hashing. By using Min-Hashing, we have optimized the space used by eliminating the sparseness and at the same time preserving the similarity.

Without the use of min-hashing, the worst-case scenario time complexity is $O(n^2)$ for a collection of n documents.
The operations on the hash table have the worst case scenario time complexity $O(n)$, thus complexity of LSH is $O(n)$. This serves to highlight that minhashing is a key factor affecting performance.

Another key factor that affects the performance would be the parameter settings as discussed in part F below. These parameter settings can be selected through certain metrics in such a way that our algorithm performs optimally as intended.

F. Analyze and discuss the parameter settings

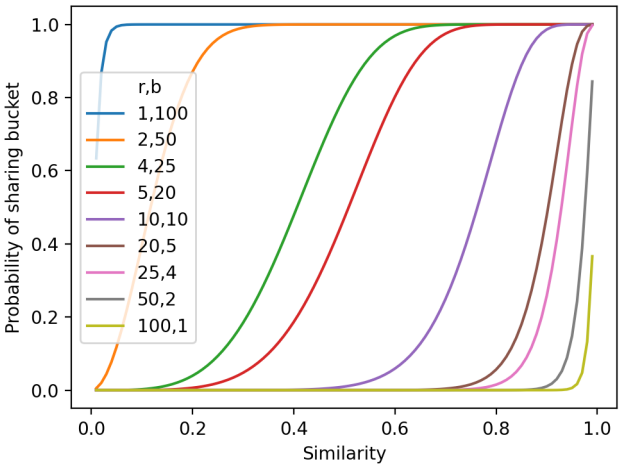
For Shingling, we pick k to represent the shingle size. The size of k depends on the size of the text documents and the length of the set of characters. k should be picked large enough that the probability of any given shingle reappearing in the documents is low.

Given that our corpus of documents is job descriptions, it suggests to us that it is not a large document, therefore $k = 9$ is considered safe. A small value of k will result in many repeated shingles in most of the documents. In return, this will be bad for differentiating documents.

For Min-Hashing and Band Partition, N represents the number of min-hashes, b represents the number of bands i.e the number of hash functions and r represents the number of rows per band. We will have to tune N, b and r to catch the most similar pairs but few non-similar pairs. If we take b to be a large value, then we reduce r with the following form:

$$N = b \times r$$

This represents a constant (number of rows in the signature matrix). Intuitively it means that we're increasing the probability of finding a candidate pair. Higher b implies lower similarity threshold (higher false positives) and lower b implies higher similarity threshold (higher false negatives).



The above graph illustrates the probability of candidate pairs sharing a bucket against the similarity. We would want to attempt shifting the similarity threshold to the right. With that being said, the value of *b* will decrease. When the value of *b* decreases, the number of candidate pairs returned increases. This will result in more false positives. Therefore, we need to balance out between avoiding false positives and false negatives to get the most accurate similarity.

G. Ablation study of the key components

The key components of the Locality Sensitive Hash function (LSH) implemented here includes shingling, Min-Hashing and finally the band and hash component. We will be exploring the effects of removing the Min-Hashing component from our algorithm.

The purpose of the Min-Hashing function is to compress our derived sparse vector into a dense signature vector, saving the search time for a candidate pair and space needed to carry out our process. Removing the Min-Hashing function would mean that our algorithm has to process the entire sparse vector which is significantly larger than our signature vector. The search space is expanded and the computation needed to complete the search and comparison is exponentially increased, making it inefficient in scaling our algorithm to handle bigger datasets. Without the use of min-hashing, the worst-case scenario time complexity is $O(n^2)$ for a collection of n documents. The document matrix will be a sparse matrix and storing it as it is will be a big memory overhead.

2. TF-IDF

A. Introduction

The TF-IDF (term frequency-inverse document frequency) statistic examines the relevance of a word to a document inside a collection of documents.

This is accomplished by multiplying two metrics: the number of times a word appears in a document and the word's inverse document frequency over a collection of documents.

It has a variety of applications, including automatic text analysis and scoring words in machine learning algorithms for Natural Language Processing (NLP).

The TF-IDF format was created for document search and retrieval. It works by increasing in proportion to the number of times a word appears in a document, but offset by the number of documents containing the word. As a result, words like this, what, and if, which appear frequently throughout all documents, rank low since they don't represent much to that document in particular.

However, if the word *AngularJS* appears frequently inside one document but not in others, it is likely to be very relevant. If we're trying to figure out which subjects some NPS replies belong to, the keyword *AngularJS*, for example, will almost certainly be associated with the topic of frontend engineering, because most responses including that word will be about that topic.

B. Term Frequency (TF)

The simplest method for calculating this frequency is to simply count the number of times a word appears in a document. The frequency can then be adjusted based on the length of the document or the raw frequency of the most frequently used word in the document.

C. Inverse Document Frequency (IDF)

The word's inverse document frequency over a collection of documents. This refers to how common or uncommon a word is within the entire document set. The closer a term is to zero, the more common it is. The logarithm may be determined by taking the total number of documents, dividing it by the number of documents that contain a word, and then multiplying by the total number of documents.

D. TF-IDF weighting

The TF-IDF is the product of two statistics, term frequency and inverse document frequency - a formula that aims to define the importance of a keyword or phrase within a document.

$$TF \times IDF$$

A higher weightage will mean that the document is more relevant, and vice-versa.

E. Strength And Weaknesses

TF-IDF is still widely used owing to its simplicity and ease of computation. We also have some basic metric to extract the most descriptive terms in a document

It also has an advantage over Bags of Words (BoW) because it can be used on bigger data. In addition, for TF-IDF, every repetition of words with the number of the sentence has its own frequency which tells about its importance.

The first weakness of TF-IDF is that it is based on the bag-of-words (BoW) model, but it cannot necessarily derive the contexts of the words and understand importance that way. Thus, it does not capture position in text, semantics, co-occurrences in different documents, etc. As such, the scoring might be inaccurate and documents will be ranked incorrectly in the following cases.

a. Phrases will not be considered as a “single unit”. What this means is that when comparing a document with the phrase “*Nanyang Technological University*” and another document with the same 3 words in different places and context, the model will score them equally, ceteris paribus.
b. The above also extends to situations like negation. For example, “*not good*” and “*good*” have opposite meanings, but the model does not take that into consideration when scoring the similarity.
c. The lack of capability to capture and understand word ordering might give rise to inaccurate results too. For example, document A with the phrase “*school of computer science and engineering*” will receive the same scoring as document B with the phrase “*engineering computer of science school*”.

d. Also, the model cannot recognize the difference between words with different meanings. For example, it cannot differentiate between the tech company *Apple* from the fruit *Apple*.
e. Lastly, TF-IDF also does not take synonyms into consideration. For example, "*funny*", "*humorous*", "*hilarious*" have similar meanings but they are treated as entirely different unique words.

The second weakness is that document frequency only takes into account the number of documents the word appears in, but not the number of times it appears within each document and in totality (collection frequency). This may incorrectly suggest that the word isn't particularly significant in any given document. While this may be sensible in many cases, there are instances where this poses significant obstacles to calculating an accurate value, and where the assigned value is incorrectly low.

To understand how this can be an issue, let us consider the following example. Assuming we have 10 documents: 1 of them is 50% made up of the term "computer" and the other 9 contain the term "*computer*" only once (less than 1%). Since the term appears in every document, the model will assign a zero value to the IDF for the word, thus suggesting that "*computer*" is not uniquely important to every document, even though it clearly is for the first document.

F. Analyze and discuss the parameter settings

The term frequency tf(t,d) of term t in document d is defined as the number of times that t occurs in d.

However, raw frequency is not what we want as relevance does not increase proportionally with term frequency. A document with 10 occurrences of the term is more relevant than a document with 1 occurrence, but not 10 times more relevant.

As such, in our implementation, we will be using the log frequency of the term instead of the actual frequency. Log frequency can be calculated using $1 + log(tf(t,d))$ when tf(t,d) is more than 0, otherwise it will be 0.

On the other hand, df(t) is the document frequency of t: the number of documents that contain t.

Frequent terms are less informative than rare terms, so we will define idf(t) with the formulae $log(N/df(t))$, where N is the number of documents.

Both values will undergo cosine normalization by multiplying them with $1/\sqrt{(w_1)^2 + (w_2)^2 + ... + (w_k)^2}$ whereby w is the weighting for each value in term frequency or inverse document frequency.

G. Key factors that affect the performance

In order to calculate the TF-IDF value for a search, we will need to do pre-processing and calculating of the respective TF and IDF as mentioned above. These processes will affect the performance in three main aspects: accuracy of similarity, processing speed and memory required.

If case folding is omitted during the data preprocessing state, the similarity will be reduced. For example, a user searching for "javascript" will not find any match if all the documents are spelled using camelcase "JavaScript". This is very likely going to be the case, given that users have a tendency to not type in proper writing style, such as when using a search engine.

The total size of the collection (sum of the number of documents and the size of the individual documents) are proportional to the processing speed, as well as additional memory required. This is because:

a. A larger collection size means more data preprocessing is needed
b. A larger collection size is likely to have more unique terms, which will therefore increase the size of the matrix that is storing the TF and IDF values
c. A larger matrix will increase the lookup time required to find the TF

All of the processes involved are almost linearly proportional to the number of documents in the collection.

H. Ablation study of the key components

For the ablation study of the importance of case folding in TF-IDF, we ran the query "*DOCTOR*" (in full uppercase) amongst a collection of 10,000 documents to observe the number of records that could be at least relevant.

- With case folding, 26 documents were returned
- We then proceed to disable case folding and not surprisingly, 0 documents were returned, even though a number of documents contain the word "*doctor*"

As we can see, not doing case folding could severely handicap the similarity accuracy of TF-IDF. In addition, further normalization can also improve the accuracy of the model, such as deleting periods to form a term ("*U.S.A.*" and "*USA*") and deleting unnecessary whitespace ("*java script*" and "*javascript*"). However, they are not implemented as they are quite domain specific, and failure to normalize properly will backfire and reduce the similarity accuracy.

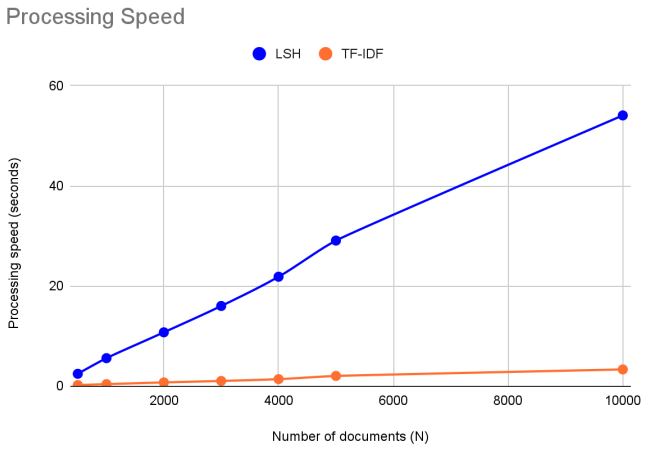
V. Experimental Analysis and Comparison

A. Processing Speed

We will be comparing the time required for the different methods across records of different sizes (N)

	LSH		TF-IDF	
N	**Time (s)**	**Gradient**	**Time (s)**	**Gradient**
500	2.4422	-	0.1875	-
1000	5.5711	0.00626	0.3750	0.00038
2000	10.7356	0.00517	0.7188	0.00034

3000	15.9993	0.00526	1.0156	0.00030
4000	21.8483	0.00585	1.3750	0.00036
5000	29.0737	0.00723	2.0156	0.00064
10000	54.0880	0.00500	3.3125	0.00026



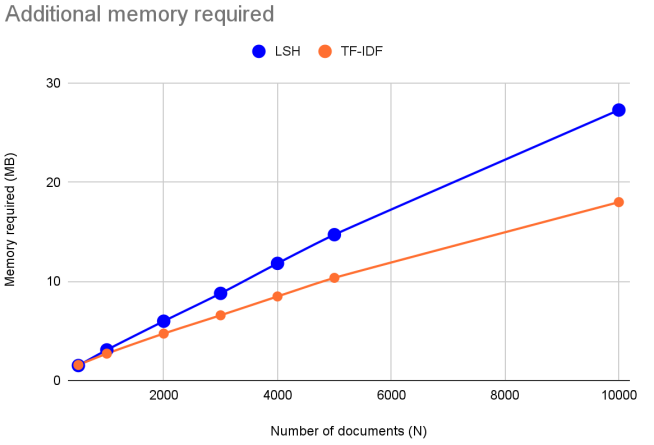
We can see from the table and graph above that LSH takes longer to run than TF-IDF. LSH features more processing layers than TF-IDF, such as shingling and min-hashing. TF-IDF utilizes tokenization to break down the query depending on the number of words. On the other hand, LSH's shingling stage breaks down the text into k-length strings. This means that LSH has a larger number of substrings than TF-IDF. Therefore, LSH takes a longer time to run than TF-IDF during text mining.

With an increase in the number of documents, we can see that the distance between the two graphs or the difference in their processing speed also increases. Since LSH will have to spend a longer time during text mining when the number of documents increases, this is the reason for the difference in processing speed increasing with an increased number of documents.

B. Memory

We will be comparing the additional memory required for the different methods across records of different sizes (N)

	LSH		TF-IDF	
N	**Memory (MB)**	**Gradient**	**Memory (MB)**	**Gradient**
500	1.49	-	1.548	-
1000	3.07	0.00316	2.693	0.00229
2000	5.97	0.00290	4.718	0.00203
3000	8.78	0.00281	6.566	0.00185
4000	11.81	0.00303	8.470	0.00190
5000	14.710	0.00290	10.348	0.00188
10000	27.30	0.00252	17.996	0.00153



We can see from the table and graph above that LSH requires more memory to run than TF-IDF. As explained under the processing speed section, LSH has a larger number of substrings than TF-IDF due to the difference in the way text mining is done. LSH uses shingling while TF-IDF uses tokenization.

With an increase in the number of documents, we can see that the distance between the two graphs or the difference in their memory required also increases. This phenomenon happens because for TF-IDF, it follows a logarithmic shape where there will be fewer unique words appearing each time more documents are added. For example, increasing the number of documents from 10 to 100 might increase the number of unique words by 10 times, but increasing it from 100,000 to 1,000,000 is less likely to introduce a large number of unique words. On the other hand, LSH will create even more unique k-length substrings when more documents are added due to shingling.

VI. CONCLUSION

While TF-IDF is a more popular and familiar model for working with text similarity; if dealing with a dataset that scales, LSH is a more convenient model.

It is also noteworthy that the accuracy of a model is limited by the data's cleanliness and usability. As such, to improve the accuracy of the similarity computation, data preprocessing is very crucial. As seen from our research, even skipping a seemingly simple step such as case folding could result in vastly inaccurate similarity scoring.

Furthermore, domain-specific preprocessing can also be done to improve the model. For example, if we are dealing with data pertaining to technological framework, we should not remove fullstops from words such as "*.NET*" or "*web3.js*" or "*react.js*" as it might change the meaning of the word and lead to further false negatives.

Besides the two methods discussed in this paper, there are also other similarity calculation and search methods, such as N-grams, KNN (K-Nearest Neighbour), Product Quantization, Inverted FIle Index etc. As every approach has its strengths and weaknesses, there is no one-size-fits-all approach.

REFERENCES

[1] Locality Sensitive Hashing, Towards Data Science, n.d. accessed on Nov 05, 2021.
https://towardsdatascience.com/locality-sensitive-hashing-how-to-find-similar-items-in-a-large-set-with-precision-d907c52b05fc

[2] Locality Sensitive Hashing, Pinecone, n.d. accessed on Nov 07, 2021.
https://www.pinecone.io/learn/locality-sensitive-hashing/

[3] MinHash, Wikipedia, n.d. accessed on Nov 07, 2021.
https://en.wikipedia.org/wiki/MinHash

[4] Understanding Locality Sensitive Hashing, Towards Data Science, n.d. accessed on Nov 07, 2021.
https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134

[5] Locality Sensitive Hashing -- LSH Explained, Medium, n.d. accessed on Nov 08, 2021.'
https://medium.com/@hubert_46043/locality-sensitive-hashing-explained-304eb39291e4

[6] Min-Hashing and Locality Sensitive Hashing, Cran-r Project, n.d. accessed on Nov 08, 2021.'
https://cran.r-project.org/web/packages/textreuse/vignettes/textreuse-minhash.html