Project Name: Locality Sensitive Hashing (LSH) via MinHash

Your Name(s): Aditya B., Gaurav G., Hitesh B.

University of Washington: CSE 312 Summer 2020

1 Application

Randomized Algorithms have been used for a long time primarily to cut down on the run-time and space complexity of a similar deterministic algorithm. As its name implies, Randomized algorithms, like the MinHash implementation of LSH, use some sort of random feature as part of its logic to solve the given problem. Locality Sensitive Hashing, otherwise known as LSH, utilizes the basic idea that the hash code will put items with a high similarity index in the same bucket with a high probability, while it will put items with a low similarity index in different buckets with a high probability.

LSH is used commonly in Near-Duplication detection with documents & text files, similar gene expression in genomes, and audio & video fingerprinting. For example, we can compare the different strains of COVID-19 that are found around the world and see if the virus has mutated. The viruses strains that are similar to each other will most likely will be placed in the same bucket, while viruses that are different from each other will placed in distinct buckets. This proves helpful to epidemiologists and Infectious Disease Specialists who can now develop strategies to attack each individual strain of the virus as opposed to one strain. It will also be helpful to know if the virus has mutated, so vaccine researchers can create a COVID-19 shot that is effective against the most prevalent strain or most strains of the virus.

2 Algorithm Pseudocode

LSH MinHash Application to DNA Strain Comparison/Tracing

```
function SHINGLING()
    data ← given list of data strings
    shingles ← set()
    shingle_length ← constant representing length of shingles
    for file in data do
        for $i = 0, ...., $len(file) - shingle_length do
            shingles.add(data_file[i: i + shingle_length)
    shingle_document ← empty list
    for Shingle in Shingles do
        document_in_current_shingle ← empty list
        for file in data do
            if shingle in file then
                document_in_current_shingle.append(1)
            else
                document_in_current_shingle.append(0)
        shingle_document.append(document_in_current_shingle)
    return shingles_document

function MIN_HASH()
    num_shingles ← length of shingles_document
    perm_indices ← random permutation of indices that fit within shingles_document
    signature_matrix ← empty list
    for $i = 0, ...., $num_permutations do
        generate random permutation of indices – shuffle
        signature_matrix_row ← $[-1] *$ num_documents
        for $j = 0, ...., $num_shingles do
            index ← index of $j$
            row ← shingles_document[index]
            for document in row do
                if signature_matrix_row[current iteration] = -1 and document = 1 then
                    signature_matrix_row[k] ← index
        signature_matrix.append(signature_matrix_row)
    return signature_matrix

function LSH()
    buckets ← empty dictionary of sets
    simiar_documents ← empty list
    for $i = 0, ...., $num_permutations, $i +$ num_rows_per_band do
        buckets ← empty dictionary of sets
        current_rows ← sig_matrix[$i : i +$ num_rows_per_band]
        for $j = 0, ...., $num_documents do
```

```
            values_in_band ← tuple of current column of band
            bucket_in ← hash(values_in_band)% num_buckets
            if bucket_in not in buckets then
                bucket_in[buckets] ← empty set
        for item in buckets.items() do
            if len(item) > 1 then
                similar_documents[index] ← .update(docs)
    return signature_matrix
```

Inputs/Outputs: Given a random permutation (π), we have that the Minhash algorithm produces $h_\pi(C) = \mathsf{min}\pi(C)$ since we are essentially determining find the first index of the permuted order in which the two documents have a common shingle. Using this, we have that $E[P(h_\pi(C_1)) = h_\pi(C_2)] = Sim(C_1, C_2)$.

Expected Runtime-MinHash: $O(p \cdot s \cdot d)$ where p is the number of permutations, s represents the length of the documents, and d is the total number of documents.

Expected Runtime-LSH: $O(p \cdot d)$ where p is the number of permutations and d is the total number of documents.

Expected Space Complexity: $O((s \cdot d) + (p \cdot d))$ where p is the number of permutations, s represents the length of the documents, and d is the total number of documents.

Our algorithm isn't deterministic and isn't guaranteed to produce the correct answer. Semantically, the LSH algorithm will say two separate documents are similar "to a high probability".

3 Explanation and Intuition

The randomness aspect of the LSH algorithm originates from the MinHash function. After we generate a matrix that represents common shingles across documents in the first step of our algorithm, we apply randomness to generate a random shuffling (permutation) of unique indices for each of the corresponding shingles. Subsequently, we then construct a matrix identified by rows of shingles and columns of documents such that a particular value of that matrix becomes the "index of the first (in the permuted order) row in which column C has value 1 (Gupta, 2018)." Those columns with the same values for each permutation therefore reflect this similarity. This works because although we generate random permutations of our indices, we notate the specific number at which a shingle does show up (in the permuted order). Since similarity is calculated based on the index of the first shingle present, we are essentially matching up document similarities in the given permuted order up until the shingle that is indeed present. We therefore have that if two documents have the same value (index n) for a particular permutation, they will therefore similarly not contain the same $n-1$ shingles but do indeed contain the same shingle that occurs at the *nth* index of the permuted order. This will be thoroughly explored using probabilities in part (7). When we perform the final step of bucket hashing, we are essentially saying two documents are similar with a high probability when they do indeed hash to the same bucket in at least one of the bands (this occurs as a result of these similar index values) and dictates whether or not the two documents are a candidate pair. The probability of two documents being matched for similarity is dictated by factors like the number of permutations, number of bands, and number of rows per band, which we will experiment with as part of our analysis in part (6).

Our LSH algorithm can be classified as a Monte Carlo Algorithm because the underlying concept of LSH is to use randomness to find similarity between two items. As we just explained, when the LSH algorithm performs its similarity computations, it only does so to a certain (high) probability, hence its Monte Carlo nature. In other words, the LSH can produce false positives, as we all as false negatives so it is a Monte Carlo algorithm.

4 Advantage(s) over Deterministic Counterpart(s)

Deterministic Counterpart to MinHash is Naive Jaccard Index

Algorithm 1 Jaccard Index Algorithm for Computing Similarity Percentage Between Two Documents:

```
1: shingles_document ← matrix representing shingles present in each document
2: intersection_count ← 0
3: union_count ← length of shingles_document
4: for $i = 0, \ldots,$ length of shingles_document do
5:     if shingles_document[i][0] = shingles_document[i][1] then
6:         intersection_count ← intersection_count + 1
   return $\frac{\text{intersection\_count}}{\text{union\_count}}$
```

Brief Explanation: Using the naive approach to the Jaccard index to compute the similarity between two documents, we can essentially use the predetermined shingles to define which shingles the two documents have in common as the intersection, and those that are in either as the union. Computing the index therefore involves diving this intersection by the union, which provides an exact similarity percentage between the provided texts.

Advantages of Randomized Algorithm: The MinHash implementation of LSH provides an effective estimate of the Jaccard index in a few ways. The MinHash implementation acts as an additional step to eliminate the sparseness of the shingles document, improving the space complexity for when we have to hash rows into buckets to determine similarity. This is because rather than mapping shingles to documents (which results in a lot of zeroes, especially with dissimilar documents), we are now mapping permutations to the documents in the new signature matrix, which no longer has to keep track of the large amount of shingles.

5 Implementation in Python

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from node import Node
from sklearn.metrics import confusion_matrix
import seaborn as sns

class LSH:
    def __init__(self, data, shingle_length, permutations, num_rows_per_band,
                 num_buckets, calc_jaccard=False):
        """
        An Locality Sensitive Hashing program that places similar documents in the
        same hash bucket with high probability and dissimilar documents
        in different buckets with high probability.

        :param data: a list of documents or genomes
        :param shingle_length: the length of each gram/shingle
                               of the document/data point
        :param permutations: the number of permutations that will be
                             used in the min-Hash Function
        :param num_rows_per_band: the number of rows in the band where we
        :param num_buckets: the number of buckets in each band
        """
        self.shingle_length = shingle_length
        self.num_permutations = permutations
        self.num_rows_per_band = num_rows_per_band
        self.num_buckets = num_buckets
        self.num_documents = len(data)
        #perform shingling
        #minhashing

        start_time = time.time()
        shingles_document = self.shingling(data)

        if calc_jaccard:
            self.jaccard_similarity = self.jaccard(shingles_document)
            self.jaccard_time = time.time()

        sig_matrix = self.min_hash(shingles_document)
        similar_documents = self.lsh(sig_matrix)
        self.sim_docs_set = set()
        self.get_set_of_sim_docs(similar_documents)
        self.time_taken = time.time() - start_time


    def get_set_of_sim_docs(self, similar_documents):
        """
        Finds the similar documents within a the original list of documents

        :param similar_documents: list of similar documents or genomes
        """
        for set_docs in similar_documents:
            if len(set_docs) > 1:
                self.sim_docs_set.add(frozenset(set_docs))


    def shingling(self, data):
        """
        Creates k length shingles for all the documents in the original data file

        :param data: list of documents or genomes
        :return: a list of all the shingles in a given document
        """
        shingles = set()
        data_as_shingles = []
        for data_file in data:
            for i in range(len(data_file) - self.shingle_length):
                shingles.add(data_file[i:i+self.shingle_length])
        shingles_document = []
        for shingle in shingles:
            document_in_current_shingle = []
            for data_file in data:
                if shingle in data_file:
                    document_in_current_shingle.append(1)
                else:
                    document_in_current_shingle.append(0)
            shingles_document.append(document_in_current_shingle)
        return shingles_document


    def min_hash(self, shingles_document):
        """
        Calculates and returns the min hash signature matrix

        :param shingles_document: a matrix where the rows are the shingles and
                                  the columns are the different files
        :return signature matrix: a matrix where the rows represent different files
```

```
                                and if the columns are similar there is a high
                                probability that the documents are similar too
        """
        num_shingles = len(shingles_document)
        perm_indices = list(range(num_shingles))
        signature_matrix = []
        for i in range(self.num_permutations):
            random.shuffle(perm_indices)
            signature_matrix_row = [-1] * self.num_documents
            for j in range(len(perm_indices)):
                index = perm_indices.index(j)
                row = shingles_document[index]
                k = 0
                for document in row:
                    if signature_matrix_row[k] == -1 and document == 1:
                        signature_matrix_row[k] = index
                    k += 1
            signature_matrix.append(signature_matrix_row)
        return signature_matrix


    def lsh(self, sig_matrix):
        """
        Returns a list of sets of similar documents or genomes from the given data set

        :param sig_matrix: a matrix where the rows represent different files and
                           if the columns are similar there is a high probability
                           that the documents are similar too
        :return: a list: that contains sets of similar documents
        """
        buckets = {}
        similar_documents = []
        for i in range(self.num_buckets):
            similar_documents.append(set())
        for i in range(0,self.num_permutations,self.num_rows_per_band):
            buckets.clear()
            current_rows = sig_matrix[i:i+self.num_rows_per_band]
            for j in range(self.num_documents):
                values_in_band = tuple([row[j] for row in current_rows])
                bucket_in = hash(values_in_band) % self.num_buckets
                if bucket_in not in buckets:
                    buckets[bucket_in] = set()
                buckets[bucket_in].add(j)
            for index,docs in buckets.items():
                if len(docs) > 1:
```

```
                docs = frozenset(docs)
                similar_documents[index].update(docs)
    return similar_documents
```

6 Visual Results and Analysis

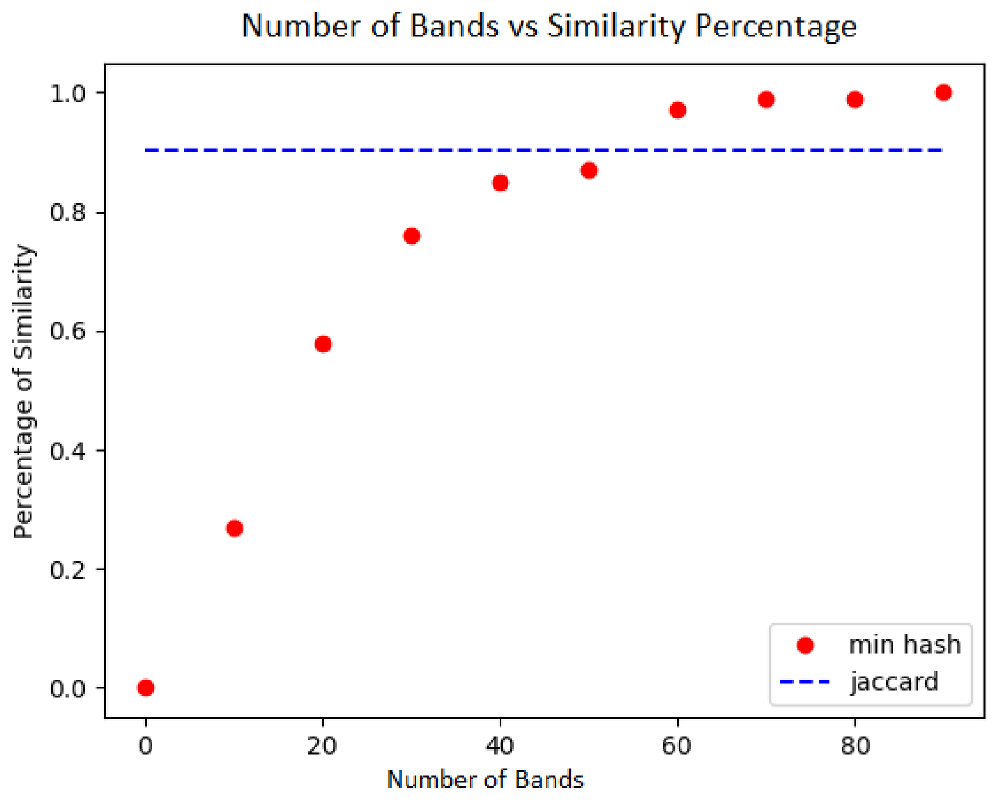

Figure 1: Plot of Number of Bands vs Percentage of Similarity Between Two Documents

This plot investigates the effects of increasing the number of bands on two documents' similarity. We see that the dashed line represents the "true" similarity, derived from the deterministic naive Jaccard approach. Since $\mathsf{num\_permutations} = \mathsf{num\_bands} \cdot \mathsf{bands\_per\_row}$, if we were to increase the total number of permutations while keeping the number of bands per each row constant, the number of total bands would naturally increase. In the case that this happens, we hypothesized that under the condition of 100 trials per each band number, we would have a greater number of trials that resulted in the two documents being similar. This is because increasing the number of bands increases the probability of matching documents because documents only need to hash to the same bucket on a single band. As you can see above, the data matched this hypothesis. Our percentage similarity actually exceeded the "true" deterministic similarity in the case of even larger numbers, which illustrates the presence of "false positives" or too many cases in which the documents are defined to be "similar". These probabilities will be explained more thoroughly in section (7).

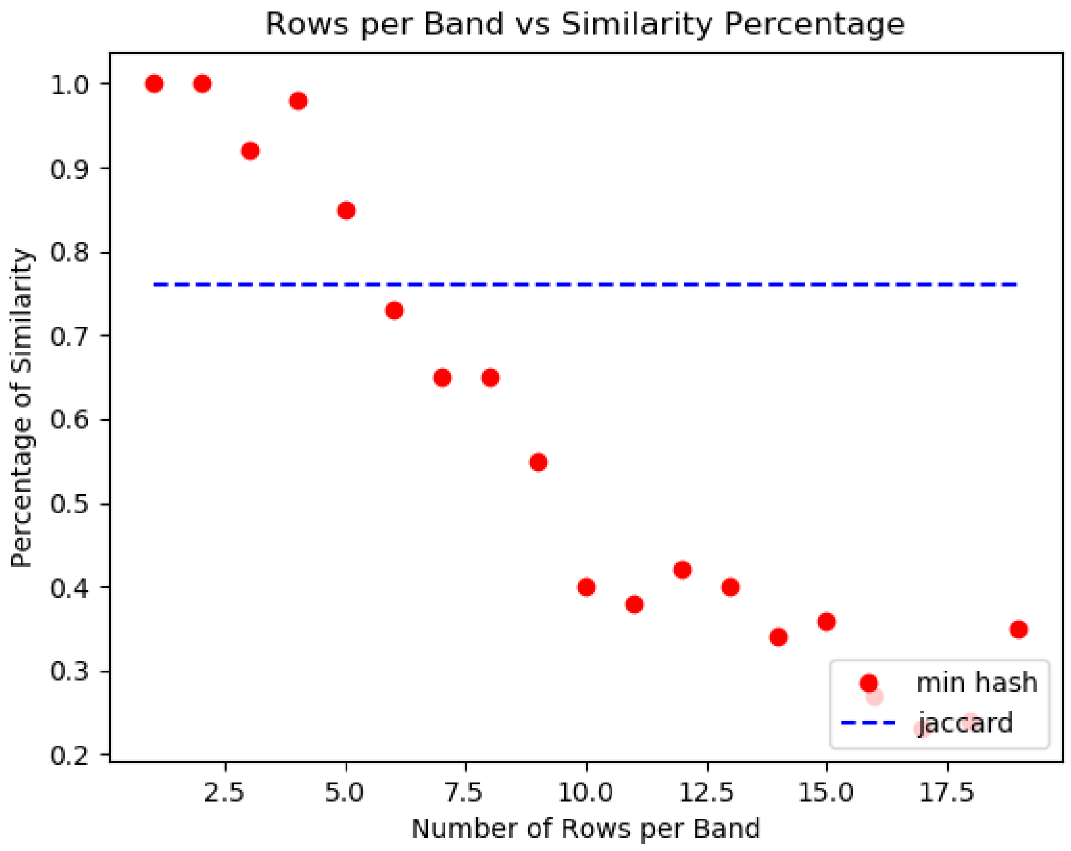

Figure 2: Plot of Number of Rows Per Band vs Percentage of Similarity Between Two Documents

This plot investigates the effects of increasing the number of rows per band on two documents' similarity. The conditions of this plot are almost identical to that of the previous. However, here we hold the number of bands constant and therefore increase the number of rows per band as the number of permutations increases. Additionally, since the two documents from which similarity is calculated are randomly chosen, we can see that our similarity is different versus the previous plot. In this case, our results once again matched our hypothesis. As the number of rows per band increased, the number of trials/100 that resulted in an outcome of similarity decreased. This most likely occurred because as when we increase the number of rows per band, we are now reducing the probability that hashes for a single band match up, resulting in an outcome of similarity. These probabilities will be explained more thoroughly in section (7).

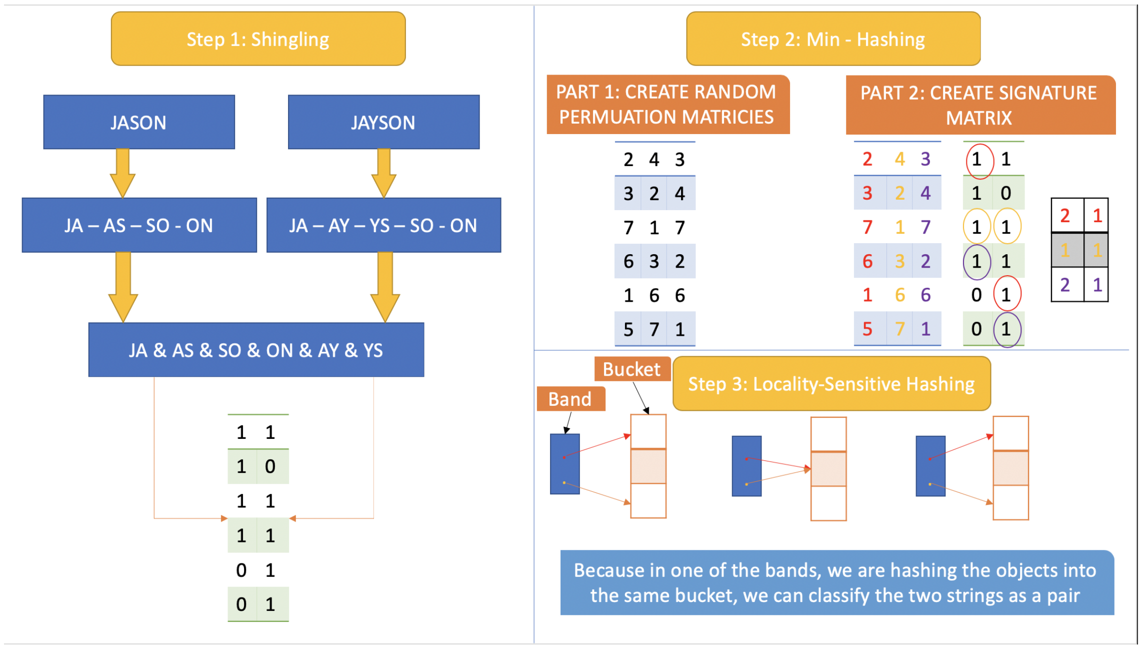
Step 1: Shingling
JASON
JAYSON
JA – AS – SO - ON
JA – AY – YS – SO - ON
JA & AS & SO & ON & AY & YS
1 1
1 0
1 1
1 1
0 1
0 1
Step 2: Min - Hashing
PART 1: CREATE RANDOM PERMUATION MATRICIES
2 4 3
3 2 4
7 1 7
6 3 2
1 6 6
5 7 1
PART 2: CREATE SIGNATURE MATRIX
2 1
1 1
2 1
Bucket
Band
Step 3: Locality-Sensitive Hashing
Because in one of the bands, we are hashing the objects into the same bucket, we can classify the two strings as a pair

Figure 3: Diagram of Process of LSH MinHashing for Two-Document Setup

Constants:
Shingle Length = 2
Number of Permutations = 3
Number of Bands = 3
Number of Rows Per Band = 1

7 Probabilistic Analysis

Let's find the probability of accurately finding two documents to be similar given that they are similar with percentage s, number of bands b, and number of rows per band r. Let's pretend our threshold for similarity (t) is 80%. We therefore also know that $s \geq 0.80$ if the documents are already classified as similar.

Let's define X to be the event that two documents, d_1, d_2 are accurately found to be similar. Therefore, we can define the event that this occurs to be a "success". As we explored in our analysis in the previous section, the probability of getting such a "success" is influenced both by the number of bands, b, and the number of rows per band r. Given a similarity percentage s, we have that in order for two documents to be found similar, they must be hashed into the same bucket for at least one of the b bands.

First, the probability of two documents being hashed into the same particular bucket, or $P(d_1, d_2 \text{ hash to same bucket}) = s^r$ because they must hash to the same bucket for each row in a band. Then this means that by complements, the probability of the two documents not hashing to a particular bucket is $1 - s^r$. We want this to be true for all b bands, so the probability of two documents not being hashed to a single bucket together, or $P(d_1, d_2 \text{ hash to no same buckets}) = (1 - s^r)^b$. So, for the two documents to be found similar, this means that they land in at least one bucket together, or the complement of our previous probability. Therefore, for two documents d_1, d_2 that have a similarity percentage s, the probability of them being considered similar (hashing to at least one bucket together) is $1 - (1 - s^r)^b$.

However, our algorithm utilizes Min Hashing to estimate the Jaccard index which represents the true similarity percentage s. We claim that this MinHash procedure is a proper estimate of this similarity percentage. Let's take a single permutation for example and examine the procedure of the MinHash implementation. Using the data provided from shingling, let's define x to be rows (shingles) in which both documents are present, and y to be rows in which only a single document is present. We can define the true Jaccard similarity as follows. To find this probability, we must find $\frac{d_1 \cap d_2}{d_1 \cup d_2}$. Since x represents only shingles that are contained in both documents, it can be defined as the intersection. The total number of shingles defines the union, which we have as $x + y$. We therefore get a result for the true similarity to be $\frac{x}{x+y}$. Under this interpretation, we have that the MinHash implementation will find the first index of the permuted order in which the two documents have a common shingle. The means we are essentially finding the probability that the first index we come across in the permuted order is at a shingle that is present in both documents, or the probability of finding x before y. This is how our estimate becomes $\frac{x}{x+y}$. If we were to increase the number of permutations, we would be increasing the number of "trials" and therefore improving the accuracy of our Jaccard estimate. We have therefore met our claim that the MinHash procedure does indeed provide a good estimate of the true Jaccard index.

References

Gupta, Shikhar. "Locality Sensitive Hashing" *Towards Data Science*, 29 Jun. 2018, https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134. Accessed 16 July 2020.

"3 2 Minhashing 25 18". *Mining Massive Datasets*, Leskovec (Stanford University Professor), https://www.youtube.com/watch?v=96WOGPUgMfw

"Applications of Nearest Neighbor Search". *Finding Similar Items*, Ullman (Stanford University Professor), http://infolab.stanford.edu/~ullman/mmds/ch3.pdf