Comparing COVID-19 Genomes Using Locality-Sensitive Hashing

Chloe Winston

1 Application

I used locality-sensitive hashing (LSH) to find similarity between the genomes of various COVID-19 strains, provided in a Kaggle dataset [1]. Specifically, I used LSH to identify "candidate pairs" of similar COVID-19 genomes. I expect that these candidate pairs correspond to where the strains originated from (e.g., Washington strains are identified as candidate pairs.) In order to do this, the sequences are processed to get sets of consecutive k-mers (essentially a subsequence of the entire sequence, each offset by one/three characters.) Then, we can create the input matrix for LSH by creating a matrix containing whether or not each k-mer is present in each sequence. Then, we can carry out LSH on the dataset to identify candidate pairs, or sequences of high similarity.

Although not covered in this project, we can compute Jaccard similarities for these candidate pairs and use these for hierarchical clustering. It would be interesting to note whether the hierarchical clustering correlates with where each strain originated from.

2 Algorithm Pseudocode

Locality Sensitive Hashing

Inputs:
input: a matrix where element at position (i, j) is 1 if the j^{th} file contains the i^{th} shingle at the i^{th} index and 0 otherwise
shingles: list of all shingles in input data
b: number of bands in signature matrix
r: number of rows per band where $b \times r$ is the size of the signature matrix

Output:
a list of tuples where each tuple (x, y) corresponds to files x and y that are candidate pairs

2.1 Time Complexity

Let us consider the analysis of N documents (with a total of s shingles) with LSH that relies on b bands and r rows/hash functions per band. LSH requires comparison only of the documents that are hashed into the same bucket. Creating the signature matrix has a time complexity of $O(Nbrs)$ because there are N columns whose signature must be computed. To compute each signature, we perform minHash by computing the minimum hash of the s shingles $b \cdot r$ times. Then to identify candidate pairs, for each band in the signature matrix we hash each segment of each column to a hash table, which results in a time complexity of $O(Nb)$. We can determine the candidate pairs from each band. Hence, the overall time complexity is $O(Nbrs)$ in addition to the optional time to compare the candidate pairs.

Now let us consider when we have hash tables for each band that contain the bands of minHashes from all considered documents. The time complexity of finding nearest neighbors for a new document is $O(br)$. This is because we need to create br minHashes and then search the b hash tables to see if there are minHash bands contained in the bucket that would correspond to the newly added document.

2.2 Space Complexity

In the locality sensitive hashing algorithm, we create a signature matrix by iterating through all columns in the input and applying a hash function on each shingle and finding the minimum hash code across all shingles. We do this $\mathsf{b} \times \mathsf{r}$ times to get $\mathsf{b} \times \mathsf{r}$ values in the signature for each column. The space complexity then of the signature matrix is $O(brn)$. Once we have the signature matrix, for each band, we add the corresponding band from each column into a hash table, then iterate through the hash table and extract candidate pairs. This part of the algorithm then takes $O(bk)$ space where we create hash tables of size k.

The space complexity of this algorithm is centered around the signature matrix since the only other significant space taken up has to do with the hash tables (which are likely relatively small.) The signature matrix takes up space with $O(nbr)$ as for each file we use br hash functions.

2.3 Correctness

This algorithm is not guaranteed to give the right answer. The whole algorithm is based on probabilities that elements of certain similarity get hashed together. Because of this probabilistic nature, there is no guarantee that the correct set of candidate pairs will be produced.

Algorithm 1 Locality Sensitive Hashing

```
1: function LSH(input, b, r, shingles)
2:     for each column i in input: do
3:         $s_i$ = new counter vector of $(b \times r)$ $\infty$ 's
4:     for each column i in input: do
5:         for $h = 1, \ldots, b \cdot r$: do
6:             for each row j in input: do
7:                 if $\text{input}_{ji} = 1$: then $s_i[h] = \min(\text{hash}_h(\text{shingles}[j]), s_i[h])$
       output ← new list
8:     for $j = 1, \ldots, b$: do
9:         table ← new hash table of k buckets
10:        for each column i in input: do
11:            band ← list of values in i from $(j-1)r$ to $jr$
12:            add file corresponding to column to bucket hash(band) in table
13:        for each bucket in table: do
14:            if more than one files in bucket: then
15:                for each file_one in bucket: do
16:                    for each file_two in bucket: do
17:                        if file_one $\neq$ file_two: then
18:                            if (file_one, file_two) and (file_two, file_one) $\notin$ output: then
19:                                add (file_one, file_two) to output
       return output
```

3 Explanation and Intuition

The Jaccard index of sets A and B is $\frac{A \cap B}{A \cup B}$. The LSH algorithm ends up identifying candidate pairs based on the Jaccard index. When we compute the minHash on a file using a given hash function, we are essentially computing the minimum of uniformly distributed values corresponding to shingles. For a given hash function, the probability that two documents A and B will have the same minHash will be $\frac{A \cap B}{A \cup B}$ since there are $A \cup B$ total possible minHash values, out of which there are $A \cap B$ possible minHash values that would allow both A and B to have the same minHash (since both files would have the shingle that produced that hash value.) Hence, we can see that the probability that two files share the same minHash for a given hash function is identical to the Jaccard similarity. Notice that in LSH, we are taking a band of minHashes essentially. Extending this, we can see how when we take many hash functions br, the probability that files with a certain Jaccard similarity (or more) get the same minHash for at least one of the bands essentially approaches 1. The idea then is that we identify candidate pairs as pairs of documents that share the same minHash signature across at least one band. To determine this, we create hash tables for each band containing all the bands of minHashes that were obtained for all the documents. Items that hash to the same bucket in these hash tables are considered candidate pairs. This works because, as stated previously, more similar documents are more likely to produce identical minHash bands.

LSH is a Monte Carlo algorithm because its hash functions rely on random sampling for a finite number of times. Hence, we will always reach an answer, albeit not always correct.

4 Advantage(s) over Deterministic Counterpart(s)

4.1 Deterministic Counterpart

Algorithm 2 Identifying Candidate Pairs

```
 1: function findPairs(input,threshold)
 2:     for each column i in input: do
 3:         for each column j in input: do
 4:             if file_i $\neq$ file_j: then
 5:                 intersection $\leftarrow 0$
 6:                 union $\leftarrow 0$
 7:                 for each row k in input: do
 8:                     if input_ki $= 1$ or input_kj $= 1$: then
 9:                         intersection += 1
10:                     if input_ki $= 1$ and input_kj $= 1$: then
11:                         union += 1
12:                 jaccard_index = intersection / union
13:                 if jaccard_index $\geq$ threshold: then
14:                     add (file_i, file_j) to output
15:     return output
```

Algorithm 2 describes the deterministic algorithm to identify candidate pairs. Essentially, we iterate through all possible pairs in the input matrix, calculate the intersection and union between the corresponding sets of shingles and then compute the Jaccard index. If the Jaccard index is greater than a certain threshold, then we add this pair to the output list of candidate pairs. Locality-sensitive hashing has several advantages over the deterministic algorithm as described below.

4.2 Time Complexity Advantages

Suppose we are identifying similar pairs amongst N files or sequences or items, using a total of s shingles. The deterministic algorithm requires comparison of all possible pairs of items (i.e., $(N-1)N$ possibilities.) Hence, the time complexity of the deterministic algorithm is $O(sN^2)$. On the other hand, as we determined in part 2, the overall time complexity is $O(Nbrs)$ in addition to the optional time to compare the candidate pairs. Ideally, though, this additional time would be much less than $O(N^2)$. Hence, with appropriately tuned b and r, LSH has a better time complexity than its deterministic counterpart.

In addition, when we are identifying nearest neighbors given a new document, the deterministic algorithm takes $O(n)$ time (disregarding shingles) because its Jaccard index with all n documents must be compared. On the other hand, LSH allows for a time complexity of just $O(br)$.

4.3 Space Complexity Advantages

Again, suppose we are identifying similar pairs among n files, using a total of s shingles. The input matrix has N columns and s rows and so has a space complexity of $O(ns)$. This matrix, though, is a sparse matrix and is quite large. Size may increase quickly because adding a new document can have a significant number of shingles and so add a significant number of entries. The MinHash algorithm allows us to reduce the size of this input matrix to a size of $O(nbr)$ where br is the total number of hash functions we use and is smaller than s. Oftentimes, reasonable values of b and r can be found for a given problem that result in a better space complexity than the deterministic algorithm.

5 Implementation in Python

```
# Takes sequence_to_kmers, a dictionary mapping from an id
# to a set of shingles (or kmers), and num_hashes, number
# of hash functions for MinHash to create signature matrix.
# Returns signature matrix.
def compute_signature_matrix(sequence_to_kmers, num_hashes):
    signatures = []

    for sequence,kmers in sequence_to_kmers.items():
        signature_list = []

        for index in range(num_hashes):
            minimum = sys.maxsize
            for kmer in kmers:
                hashed = hash_kmer(kmer,index)
                if hashed < minimum:
                    minimum = hashed
            signature_list.append(minimum)

        signatures.append(signature_list)

    return signatures


# Takes sequence_to_kmers a dictionary mapping from an id
# to a set of shingles (or kmers), b the number of bands in
# the signature matrix, and r the number of rows per band.
# Returns the set of candidate pairs found and a list of b
# hash tables generated for each band.
```

```python
def lsh_candidate_pairs(sequence_to_kmers, b, r):
    sequence_ids = list(sequence_to_kmers.keys())

    hash_tables = []
    candidate_pairs = set()

    for band in range(1, b + 1):
        hash_table = {}
        for col in range(len(sequence_ids)):
            band_sig = signatures[col][(band - 1) * r : band * r]

            hash_value = mmh3.hash(str(band_sig), 42, signed=False)
            bucket_num = hash_value % num_buckets

            if bucket_num not in hash_table:
                hash_table[bucket_num] = set()

            hash_table[bucket_num].add(sequence_ids[col])

        for bucket_num, sequences in hash_table.items():
            for id1 in sequences:
                for id2 in sequences:
                    if id1 != id2:
                        candidate_pairs.add(tuple(sorted((id1, id2))))

        hash_tables.append(hash_table)

    return candidate_pairs, hash_tables
```

6 Visual Results and Analysis

We first generate kmers for each of the DNA sequences. The following function takes a pandas dataframe with loaded data from a csv file [1] and the length of each kmer as k.

```python
def generate_kmers(sequences, k):
    sequence_to_kmers = {}  # map from NCBI_ID to set of k-mers in that sequence
    all_kmers = set()

    for index, row in sequences.iterrows():
        ncbi_id = index
        sequence = row["Sequence"]
        kmers = []
        # Create k-mers
        i = 0
        while i + k < len(sequence):
            kmer = sequence[i : i + k]
            kmers.append(kmer)
            all_kmers.add(kmer)
            i += 3
        sequence_to_kmers[ncbi_id] = kmers

    print("Done generating", str(len(all_kmers)), "kmers")

    return sequence_to_kmers, all_kmers
```

Next, we compute the signature matrix and candidate pairs using LSH, using the functions defined in the previous section. We can now use the following functions to calculate the Jaccard similarities of all the candidate pairs.

```
# Takes two sets/lists of kmers and calculates the Jaccard
# similarity of them.
# Returns the Jaccard similarity.
def jaccard_index(kmers1, kmers2):
    kmers1 = set(kmers1)
    kmers2 = set(kmers2)

    intersection = len(kmers1.intersection(kmers2))
    union = len(kmers1.union(kmers2))

    return intersection / union

# Takes a list of candidate pairs and a dictionary mapping
# from id to set of kmers and computes the Jaccard Similarities
# of all the candidate pairs.
# Returns list of Jaccard similarities.
def compute_jaccard_similarities(candidate_pairs, sequence_to_kmers):
    sims = []

    for pair in candidate_pairs:
        sims.append(jaccard_index(sequence_to_kmers[pair[0]], sequence_to_kmers[pair[1]]

    return sims
```

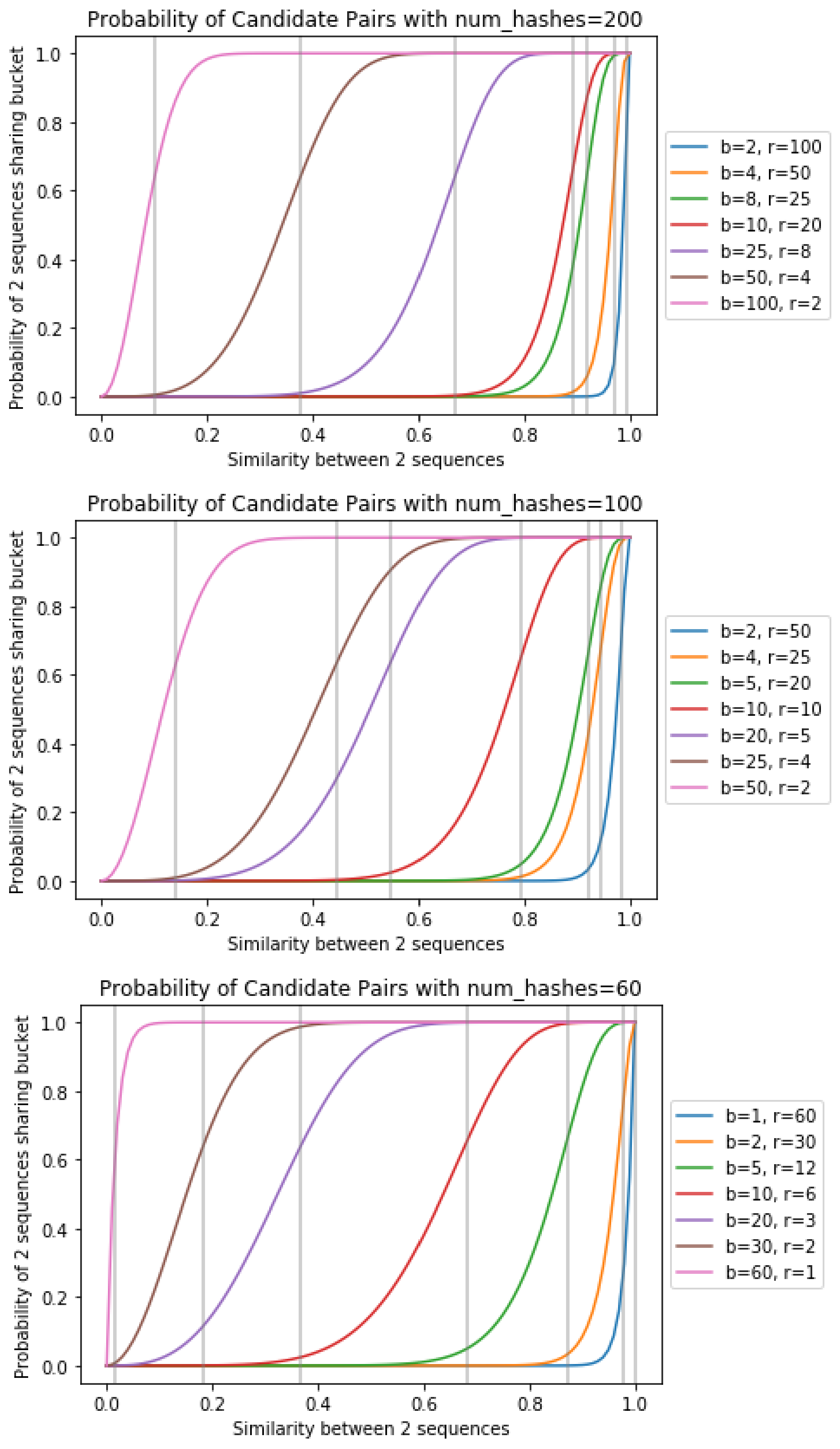


Figure 1: These three plots demonstrate the trend that the probability of two sequences sharing the same bucket takes over their Jaccard similarity for different parameters.

(continued from Figure 1) Each plot assumes different total numbers of hashes and explores various options of b and r for that given total number of hashes. Each plot contains vertical lines that show what would be considered the "similarity" threshold that those parameters give rise to but they basically mark the midpoints of these S-curves. As we can see, as b increases (and r decreases), the threshold decreases, and this makes sense because the probability for a given band being similar between two documents increases. It is interesting to note as well that as the total number of hashes increases, the threshold for a given b increases.

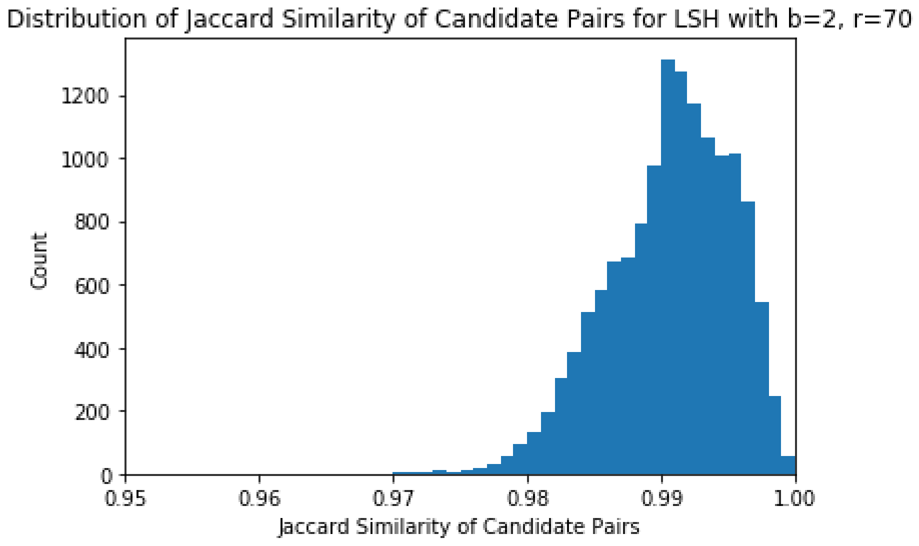
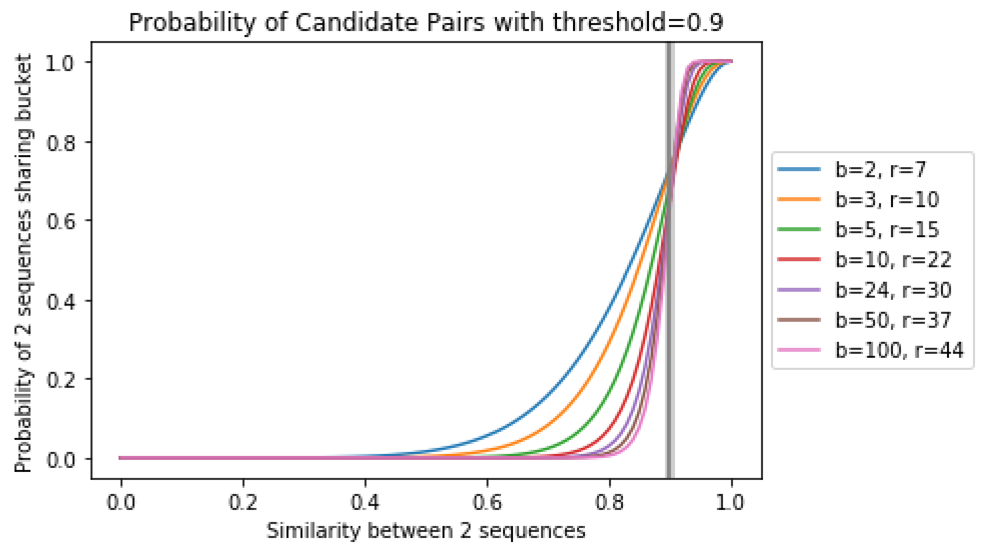
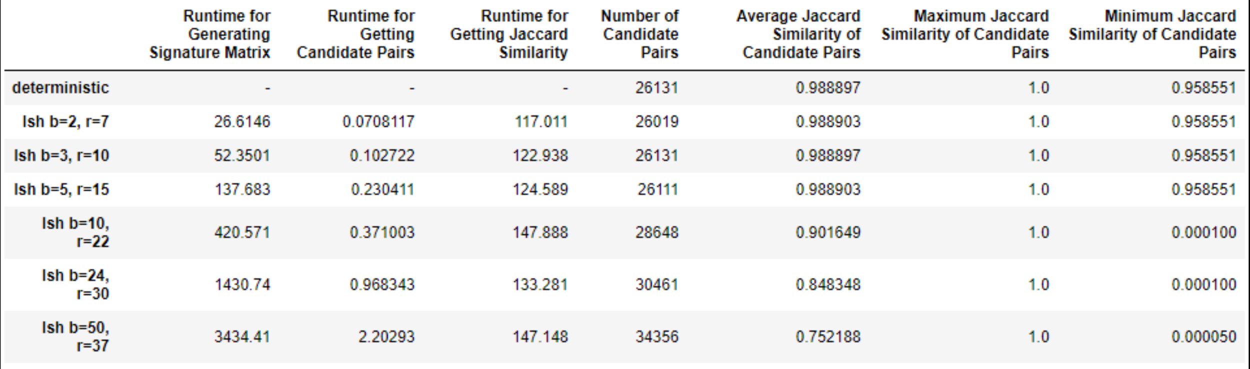


Figure 2: This histogram demonstrates the Jaccard similarities of all candidate pairs outputted by the LSH algorithm run on our set of sequences with parameters $b = 2$ and $r = 70$.





	Runtime for Generating Signature Matrix	Runtime for Getting Candidate Pairs	Runtime for Getting Jaccard Similarity	Number of Candidate Pairs	Average Jaccard Similarity of Candidate Pairs	Maximum Jaccard Similarity of Candidate Pairs	Minimum Jaccard Similarity of Candidate Pairs
deterministic	-	-	-	26131	0.988897	1.0	0.958551
lsh b=2, r=7	26.6146	0.0708117	117.011	26019	0.988903	1.0	0.958551
lsh b=3, r=10	52.3501	0.102722	122.938	26131	0.988897	1.0	0.958551
lsh b=5, r=15	137.683	0.230411	124.589	26111	0.988903	1.0	0.958551
lsh b=10, r=22	420.571	0.371003	147.888	28648	0.901649	1.0	0.000100
lsh b=24, r=30	1430.74	0.968343	133.281	30461	0.848348	1.0	0.000100
lsh b=50, r=37	3434.41	2.20293	147.148	34356	0.752188	1.0	0.000050

Figure 3: This plot demonstrates the trend that the probability of two sequences sharing the same bucket in the LSH algorithm over Jaccard similarity for different parameters of b and r with the same threshold. The table demonstrates the runtime for generating the signature matrix, for identifying the candidate pairs, and for getting the Jaccard similarity for these various parameters. It also demonstrates the number of candidate pairs and the average, minimum, and maximum Jaccard similarity of the candidate pairs outputted by the LSH algorithm for each set of parameters, in addition to the deterministic algorithm. It appears from this that the parameters of $b = 3$ and $r = 10$ give rise to the largest average Jaccard similarity among the outputted candidate pairs.

7 Probabilistic Analysis

Set-Up

Suppose we have n files with a total of s input shingles and we are running a LSH with b bands and r rows per band. We consider two documents with Jaccard index of J.

Probability of Success

Let us start by calculating the probability of success (i.e., that they will be output as a candidate pair with LSH) for two documents A and B with a Jaccard similarity of J.

First, let us consider the probability that all elements in an arbitrary band of the signature matrix are identical between documents associated with files A and B. This is identical to calculating the probability r hash functions result in identical minHashes for documents A and B. For a single hash function, the probability that minHash(A) = minHash(B) is the probability $P(\min_{x \in a} h(x) = \min_{y \in b} h(y))$ where h is the hash function being used, a is the set of shingles in document A, and b is the set of shingles in document B. Notice that for $\min_{x \in a} h(x) = \min_{y \in b} h(y)$, $\arg\min_{x \in a} h(x) = \arg\min_{y \in b} h(y) = \arg\min_{z \in a \cap b} h(z)$ as well. Hence, the probability that the two minHashes are identical for this single hash function is the probability that the shingle that gives the minimum hash code among all the shingles used by A and B is in both a and b. Because the hash function is supposedly uniform, the probability that this happens is just the number of shingles that both files share (i.e., the number of hashes that are identical between both) divided by the total number of shingles. Notice that this is just Notice that because each hash function is supposedly uniform, this probability is simply $\frac{|a \cap b|}{|a \cup b|} = J$. Now, for all r minHashes in an arbitrary band to be identical between the signatures ofr A and B, we can apply the product rule and find the probability to be J^r. [2]

From this, we can find that the probability that a given band in the signature matrix is different between columns A and B is $1 - J^r$ by complement. Hence, again applying the product rule, we can find that the probability that two columns differ completely in all b bands is $(1 - J^r)^b$ since each hash function is independent, and so the probability of each band differing is also independent from each other. Therefore, using complement, the probability that two columns in the signature matrix are identical in at least one band is $1 - (1 - J^r)^b$. Because A and B will be marked a candidate pair if they are identical in at least one band, **the probability that A and B will be identified as candidate pairs will be $1 - (1 - J^r)^b$.** [2]

References

[1] https://www.kaggle.com/ritamenezes/covid19-complete-genomes

[2] https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134