MinHash Genomic Fingerprinting to Estimate Edit Distance

Joanna Guo
Department of Biomedical Engineering
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
joannaguo@jhu.edu

Darius Irani
Department of Computer Science
Department of Applied Mathematics and Statistics
Johns Hopkins University
Baltimore, MD 21218
dirani2@jhu.edu

Jungmin Lee
Department of Biomedical Engineering
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
jlee581@jhu.edu

Laurent Park
Department of Computer Science
Department of Applied Mathematics and Statistics
Johns Hopkins University
Baltimore, MD 21218
lpark14@jhu.edu

Abstract

Edit distance is one of the most common similarity metrics in genomics. However, the algorithm to calculate exact edit distance between two strings, such as two DNA sequences, is infeasible to run, especially on long genomes. In this study, we implement various MinHash sketches for genomic sequences to estimate the Jaccard similarity. Using these estimated Jaccard similarities, we propose a novel method to estimate the edit distance, and state our assumptions. By evaluating our Jaccard estimate against the true Jaccard similarity, we tune two parameters: k-mer length and L hash values. The four MinHash variants implemented were L hash functions MinHash, Bottom-L MinHash, L-partitions MinHash, and Containment MinHash. In addition to these MinHash schemes, we implemented a deterministic multiple hash layer MinHash, which can be paired with any of the four base variants, as well as a striding scheme when extracting the overlapping k-mers from the reads. Our results show that Bottom-L MinHash and Containment MinHash with $k = 12$ and $L = 128$ best estimated the Jaccard similarity.

Keywords: genomics, fingerprinting, edit distance, MinHash

Acknowledgements

We are deeply indebted to Dr. Benjamin Langmead, Dr. Brad Solomon, and Michael Alonge for their help in this project.

1 Introduction

Measuring similarity between items is important in many applications across fields, from detecting similar words and correcting typos in natural language processing to refining results in search engines software to genome alignment in bioinformatics.

In genomics, one of the most common similarity metrics is edit distance, which is defined as the minimum number of edits, which are substitutions, insertions, and deletions, that are required to transform one string into another. Edit distance can be calculated using the Needleman-Wunsch algorithm [1]. This algorithm makes use of the global alignment dynamic programming matrix to determine edit distance. However, it is infeasible to scale the Needleman-Wunsch algorithm to longer read genome sequences since both memory and runtime performance scale quadratically in this setting. Considering the current era and utility of Third Generation Sequencing techniques by Pacific Biosciences (PacBio) and Oxford Nanopore Technologies (ONT), sequence lengths are now orders of magnitude longer than those obtained through previous Illumina Second Generation Sequencers. Therefore, there is a need to produce other methods that are less space intensive while accurately computing or approximating the edit distance.

In this project, we propose using MinHash fingerprinting, a common sketching technique to first approximate the Jaccard coefficient between two sequences, a metric that describes the level of similarity between two sets. MinHash returns a smaller representation of a sequence independent of read length called a fingerprint. This procedure is applied to two query sequences, and the Jaccard similarity can be approximated on these two fingerprints instead, saving time and space. The Jaccard similarity can then be used as a proxy for estimating the edit distance. Here, we explore several variations of MinHash and assess its effectiveness at calculating both the Jaccard similarities and edit distances across pairs of real and synthetic sequences.

2 Prior Work

2.1 Needleman-Wunsch Edit Distance Algorithm

The Needleman-Wunsch algorithm works by creating a 2-dimensional dynamic programming matrix D, where the dimensions are the lengths of the two input strings s_1 and s_2, whose lengths are $|s_1| = n$ and $|s_2| = m$, respectively. The entire $n \times m$ dynamic programming matrix D is then filled out in a bottom-up fashion, solving every edit distance subproblem with every possible pair of substrings of s_1 and s_2. For example, one way to fill out the dynamic programming matrix D is as follows:

1. Initialize every cell in the first row and the first column of the dynamic programming matrix D to zero.
2. Fill out the remaining cells in the dynamic programming matrix D, row-wise, starting with the second row:

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D(i-1,j-1)+\delta(x[i-1],y[j-1]) \end{cases}$$

where

$$\delta(a,b) = \begin{cases} 0 & a = b \\ 1 & a \neq b \end{cases}$$

Filling out all $m \times n$ cells in the dynamic programming matrix D is clearly an $O(mn)$ problem. At this rate, this algorithm is completely infeasible to use to determine edit distance for longer sequences and genomes. As an example, let our genome of interest be the human genome, whose length is approximately $m = 3 \times 10^9$ base pairs. Assume we are sequencing the genome at $50\times$ coverage with a Second Generation Illumina HiSeq 3000/4000 machine that reads lengths of $n = 150$ nucleotides on average. This gives us $d = 10^9$ total reads. $d = 10^9$ reads of $n = 150$ nucleotides per read is $bn = 1.5 \times 10^11$ base pairs to sequence, which will approximately be a 4-day run of the Illumina HiSeq 3000/4000 sequencer machine. Now, to determine edit distances between our reads as compared to the reference genome, we would need to fill in a $m \times d \times n = (3 \times 10^9)(10^9)(150) = 4.5 \times 10^21$ dynamic programming cells. Assuming a computer cluster with 1,000 processors, each clocked at 3 GHz, capable of completing 8 dynamic programming cell updates per clock cycle, this will take approximately 6 years, which is a totally infeasible amount of time.[1]

3 Methods and Software

The goal of our project was to apply multiple variants of MinHash to most accurately and efficiently approximate edit distance. We aim to compare these different MinHash variants by the accuracy of their approximation of edit distances

[1] Langmead, B. EN.601.447/647 Computational Genomics: Sequences, Lecture 13. Johns Hopkins University. Oct 15, 2019.

as well as space and time complexities. For each implementation, we utilize the MinHash fingerprint we obtained to approximate the Jaccard similarity metric. By comparing our estimated Jaccard similarity with the true Jaccard similarity calculated from the original set of k-mers, we then tune the parameters of our MinHash implementations: the k-mer length and the L hash values (the size of the sketch) to push our estimated Jaccard similarity close to the true Jaccard similarity. Finally, using the true Jaccard similarity, we aim to estimate the edit distance and compare this with the true edit distance as determined with the Needleman-Wunsch algorithm.

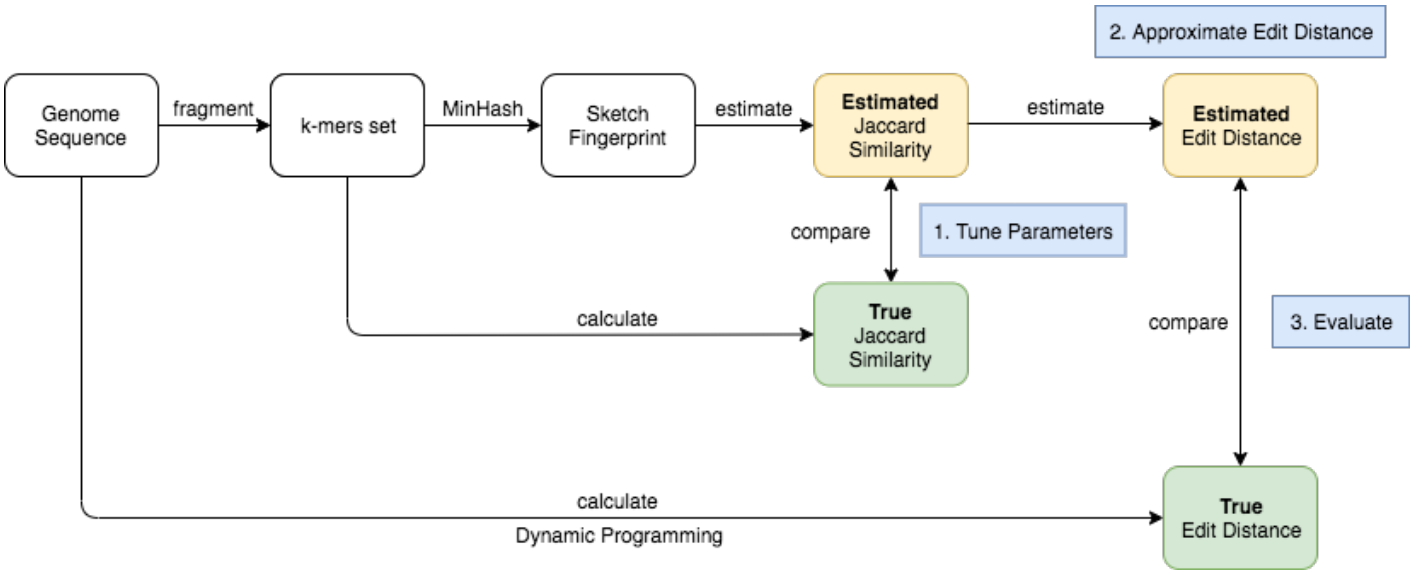


Figure 1: Methods Process

To properly assess performance and tune model parameters accordingly, we require sequence pairs where we are able to manipulate similarity manually. In addition to developing MinHash variants, we built several programs to generate synthetic sequences with controlled number of edits. In addition, we generate adversarial sequences with many repeating patterns to assess potential shortcomings of the MinHash algorithm. After studying these synthetic reads, we evaluate our algorithms on PacBio reads of E. Coli.

3.1 MinHash Implementations

MinHash is an efficient, probabilistic method for estimating Jaccard similarity, and can be used to answer queries regarding set similarity and containment. The algorithm was first introduced in [2] for applications in document similarity on the World Wide Web. The algorithm produces fingerprints, which consist of hash values and are probabilistic sketches for the complete dataset. MinHash uses locality-sensitive hashing (LSH), meaning hash functions are used to map large sets into smaller representative sets of hash values that preserve distances between elements.

LSH is a probabilistic dimensionality reduction technique ([3]), which extends the approximate set similarity to high dimensions. In LSH, the collision probability of similar objects is maximized. Applied to computational genomics, each k-mer hash may correspond to multiple k-mers in the original split dataset.

Let $\mathcal{R}^d$ denote a set of k-mers of dimension d and let H denote the family of hash functions $h : \mathcal{R}^d \to U$. LSH defines a probability distribution over H such that given two k-mers $k_1, k_2 \in \mathcal{R}^d$, then

$$Pr_{h \in H}[h(k_1) = h(k_2)] = Jaccard(k_1, k_2)$$

Typically in MinHash, H is a universal hash family, which adds the additional requirement that each hash function sampled from the family is independent of the keys being hashed ([4]). Mathematically,

$$Pr_{h \in H}[h(k_1) = h(k_2)] \leq \frac{1}{|U|}$$

Universal hash families are desirable as they reduce the variance of MinHash caused by high collision probability.

We can determine the similarity between two genomic reads by representing them as sets of distinct objects and computing the Jaccard Coefficient. Given two sets A, B, the Jaccard Coefficient is defined as intersection over union,

$$J = \frac{|A \cap B|}{|A \cup B|}$$

Jaccard takes values in the range, $0 \leq J(A,B) \leq 1$, where larger values indicate greater overlap. MinHash variants such as L hash functions, bottom-L, and L-partition each directly estimate the Jaccard Coefficient. Meanwhile, other variants like containment MinHash implement a related metric called the Containment Coefficient,

$$C = \frac{|A \cap B|}{|A|}$$

Similarly, $0 \leq C(A,B) \leq 1$, where higher values indicate high quantity of elements in A contained in B. As will be discussed in Section 3.1.4, these variants are advantageous when handling lop-sided sets, which means the reads being considered differ greatly in length.

To estimate the Jaccard similarity between two genomic reads, first each read is partitioned into a set of overlapping k-mers. Across all implementations, we iterate over k-sized substrings the reference string to generate k-mers. The index between each adjacent k-mers can be justified by setting a stride length. By default, this stride length was set to 1. There is a trade-off between computational expense, sensitivity of results and the length k of each k-mer. Low k produces short k-mers shared across many reads in a genome, while high k produces long k-mers which runs the risk of having no matches in other reads due to sequencing errors and genomic variation ([3]). Section 4 details the results of these experiments.

To obtain this fingerprint, for each hash function sampled from H, every k-mer is hashed, producing a candidate set of n hash values for each fingerprint spot. Then, for each hash function, h_i, the minimum hash value is selected from this candidate set and stored in the ith slot of F. So, for each set S of overlapping k-mers, $S = \{k_0, k_1, k_2, ..., k_{n-1}\}$ we obtain a MinHash fingerprint $F = [f_0, f_1, f_2, ...f_{L-1}]$, where L is the length of the fingerprint (e.g. the number of hash functions used in the most general k-hash approach).

3.1.1 L hash functions MinHash

The simplest MinHash setting uses L hash functions across all k-mers of the entire read. We store the minimum hash value contained over the k-mer set, and store that minimum for each function to achieve a hash fingerprint of size L.

To generate hash functions across all implementations, we utilized the xxhash[2] library. Given an integer L, this MinHash setting randomly samples L seeds in the range of 32-bit integers and instantiate `xxh32` objects with these seeds. The result is a vector of L distinct, uniformly sampled hash functions.

The time complexity of this algorithm is $O(mL)$, where L is the number of hash functions and m is the size of the k-mer set. Note that $m = O(n - k - 1)$ (n is sequence length, k is the k-mer length), since stride length is 1 in the worst case.

The space complexity is $O(L)$. We do not need to store the k-mer set in this setting, since for each hash function, we can keep track of the minimum hash as we iterate over the k-mers.

3.1.2 Bottom-L MinHash

While the number of hash functions is vastly smaller than the k-mer dimensionality, computing L hashes for each k-mer can be potentially expensive and also only considers the minimum representative across all k-mers.

The Bottom-L MinHash setting solves these issues by using a single hash function and returning the bottom L hash values as the fingerprint.

Our implementation involves first, hashing each k-mer, then sorting the hash values of all the k-mers, then finally outputting the first L hash values in the sorted list, which will be the L smallest hash values.

This implementation has a time complexity of $O(m \log m + L)$, where m is the size of the k-mer set and L is the number of hash values in the final sketch. The $m \log m$ terms is due to the overhead of sorting the hash values of all the k-mers to obtain the L minimum hash values and the L terms is due to the L values that are outputed as the hash values in the sketch.

Space complexity is $O(m + L)$, since we have to store the entire k-mer set and obtain the L minimums.

3.1.3 L-partitions MinHash

The L-partition MinHash also uses a single hash function. However, instead of taking the bottom L hash values, it partitions all hashed values into L buckets and returns the minimum hash value from each bucket as the fingerprint.

In order to deterministically partition the k-mer set into L buckets, our implementation uses the first $\log_2 L$ bits of the hash value as the bucket index of that hash value. By constraining our L value to be a power of 2, we ensure that we always have a fingerprint of size L.

[2] `https://github.com/ifduyue/python-xxhash`

The time complexity of this algorithm is $O(m)$, as we simply need to iterate over all elements in the k-mer set. For each element, we obtain its hash value, calculate its bucket index from the first $\log_2 L$ bits, and store the value if it is smaller than the value currently stored in the bucket.

The space complexity is $O(L)$. Just like the L hash MinHash, this implementation only needs to keep track of the L minimum hash values stored inside its buckets.

Another advantage of the L-partitions MinHash is its robustness to lopsided sets. Since we bin k-mers with shared hash prefixes, Each bin should contain "similar" k-mers. Thus, high or low representation of k-mers is not considered.

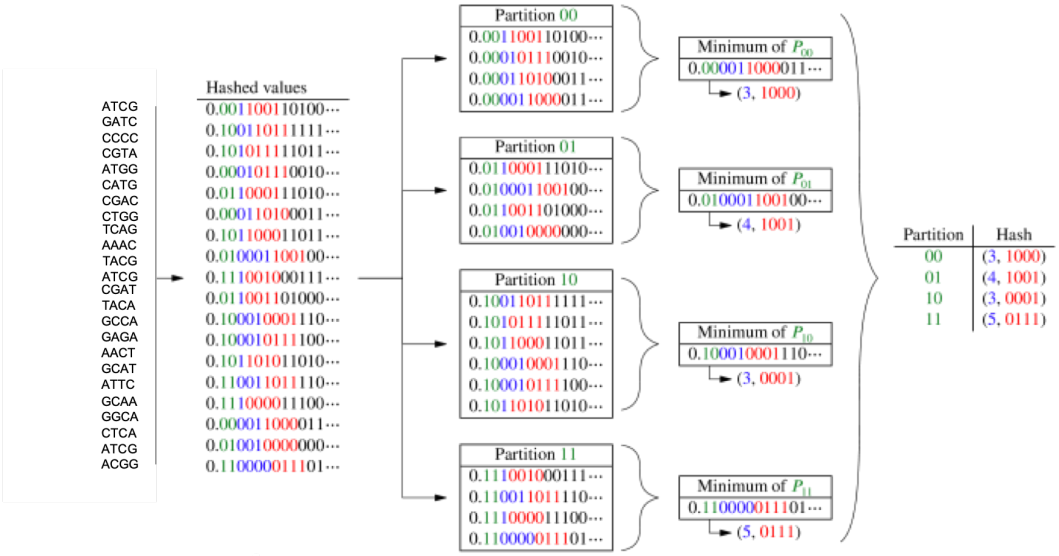


Figure 2: L-partition MinHash

Set of k-mers is hashed and deterministically binned into a subset based on its prefix (first $\log_2(L)$ bits). Minimum hash of each subset returns fingerprint. Figure adapted from
`https://www.groundai.com/project/hyperminhash-minhash-in-loglog-space`

3.1.4 Containment MinHash

Performance of conventional MinHash methods degrades when the two reads being compared differ significantly in length. Because a constant k is used, the sets of overlapping k-mers produced for each read will differ significantly, leading to a lop-sided set problem. Instead of directly estimating the Jaccard similarity between each set of k-mers, we estimate how many k-mers from the smaller set are contained within the larger set. This containment coefficient can then be used to estimate the true Jaccard similarity with better accuracy.

Since Containment MinHash is concerned with answering a set containment query, we can leverage other powerful probabilistic sketches, notably Bloom Filters. Bloom Filters enable us to perform fast set membership queries. Given a set S, a Bloom Filter $B = \{B_i\}_{i=1}^{m}$ is a bit array where $m = -\frac{|S| \log p}{(\log 2)^2}$ and p is the probability of false probability ([6]). Initially, $B_i = 0$ for each $i = 1, ..., m$. For each k-mer $s \in S$, we apply L hash functions to s, and for each hashed value, determine an index j and set $B_j = 1$. Here, $j = h_l(s) \ \% \ m$ (so that it wraps around into a valid index within the bit array). The number of hash functions is determined by $L = \frac{m}{|S|} \log 2$.

In Containment MinHash, a Bloom Filter is created for the larger of the two k-mer sets. In addition to the basic Bloom Filter methods, inside of the `BloomFilter` class, we also implemented a `union` method. This method takes as input the k-mer set, and initializes a new bit array of the same size as the existing Bloom Filter. Then, for each k-mer in this sequence, we apply the same hashing scheme to add the hash values to the filter. Finally, the class member filter and this new filter are bitwise-OR'd. This output is essentially the union of the two Bloom Filters, except this cannot directly be used to calculate the union of the two sets. Instead, [7] suggests mathematical corrections to this filter to compute the union. For two sets A, B,

$$A \cup B \approx -m \log \left(1 - \frac{A^* \cup B^*}{m}\right)$$

where $A^* \cup B^*$ represents the bitwise-OR returned from our `union` method. Using the property of Mutual Exclusion, we can obtain an estimate for set intersection from this union,

$$|A \cap B| = |A| + |B| - |A \cup B|$$

Then, both the Containment Coefficient and the Jaccard Coefficient can be estimated from these quantities.

Letting L be the number of hashes functions and m_A and m_B be the number of k-mers in A and B respectively, the time complexity of this algorithm is $O(L(m_A + m_B) + m)$, since we hash Lm_A times to initialize the Bloom Filter and Lm_B times to obtain B's filter. Bitwise-OR also takes $O(m)$ to iterate over both filters.

The space complexity is given as $O(m)$, m being defined as the filter size.

3.1.5 Multi-layer MinHash

Multi-layer hash provides an extra, deterministic hash layer to any of the MinHash variations detailed above. First, each of the k-mers were passed through a specified single deterministic hash function and saved in hex. Then the sets of these hash values were passed to the other MinHash implementations. Instead of hashing the k-mers, these implementations instead were hashing these hash values. This additional hash layer is detailed in Figure3. Inspiration for this modification came from [3].

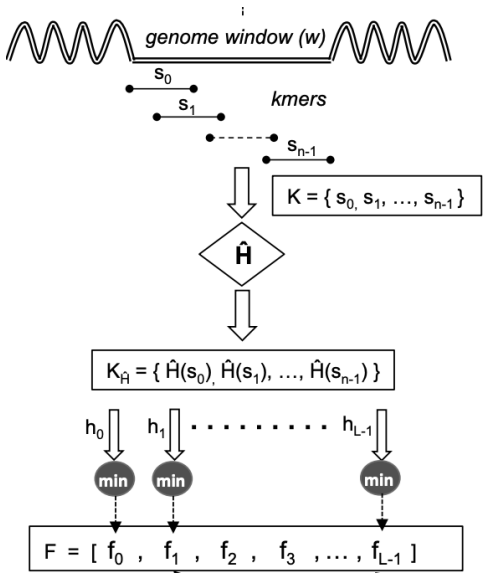


Figure 3: Deterministic Hash Layer before MinHash. Figure adapted from [3].

The time complexity is $O(mL)$, where m is the number of k-mers in the set and L is the number of hash functions.

The space complexity is $O(m + L)$, as the m hash values from the k-mers must be stored and the fingerprint size is L.

3.2 Data Usage

3.2.1 Synthetic Data

To tune and assess the effectiveness of our implementation variants, we needed to easily generate data with controlled edits, length, and other characteristics. We wrote two methods of producing synthetic data. The first simply generates random sequences, and the second inserts a user-specified number of repeating patterns with fixed length into a random sequence. The latter is used specifically for the purposes of exposing weaknesses in MinHash. For example, if two text documents had many occurrences of articles such as "the" or "and", it does not necessarily mean they are similar. However, fingerprinting schemes might not recognize this and assign high similarity.

To control edits, our program takes a sequence, sample x inserts, y deletions, and z substitutions uniformly randomly from the sequence length, and add these edits accordingly in a copy of the sequence.

In our experiments, we generated 10 "ground truth" sequences of equal lengths (8000 bp). For each sequence, we then produced five edited sequences with a different number of edits. As an example, sequence 1 generated 5 sequences with 100 edits, sequence 2 had 5 with 200 edits, 3's had 300, etc. This set of sequences then yields 60 pairs of sequences to compare, since for each of the 10 ground truth sequences, we have 5 perturbed sequences to compute similarity on.

3.2.2 PacBio E. Coli Reads

In addition to synthetic reads, we obtained third generation sequencing reads from PacBio in in FASTQ format[3]. We used 600 randomly selected sequences from the dataset and randomly paired them together to create 300 pairs.

3.3 Jaccard Similarity and Approximate Edit Distance

While it can be assumed that there must be some kind of correlation between Jaccard similarity and edit distance, the exact relationship between Jaccard similarity and edit distance is hard to determine. While Jaccard similarity could be interpreted as "content" differences, edit distances are more relevant to "structural" differences. However, as computation of the exact edit distance is expensive, we would like to explore methods where edit distance could be estimated using the more computationally inexpensive Jaccard similarity.

3.3.1 Previous Work: Bounds on Edit Distance

The only literature, that we are aware of, in which the direct relationship between Jaccard similarity and edit distance was explored was Dolev et al. [8]. One large assumption made in Dolev et al. is that they created new *representing strings* of the k-mer sets, and fond the approximate edit distance for those representing strings. These strings are created by sorting the set of k-mers (i.e. by lexicographic ordering) and concatenating them. While the representing strings are different from the original strings, Dolev et al. shows that if the frequency of the k-mers is low, the edit distance of the representing string is close to the edit distance of the original string (See Figure 4). The frequency of the k-mers can be controlled by the size of k.

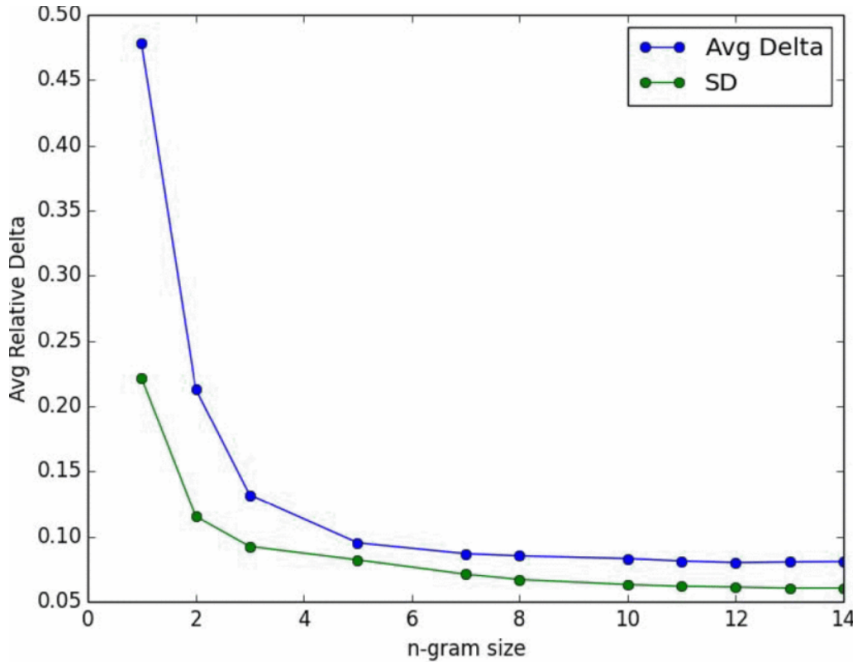


Figure 4: Average and Standard Deviation of Edit Distance Difference. The difference between the edit distances of the original and representing string decreases as n (k) increases. Figure adapted from [8]

Using this assumption, they derived that the following inequality holds for Jaccard similarity and edit distance. X and Y are the k-mer sets, and x and y are the representing strings built from X and Y. ED stands for edit distance and J is the Jaccard similarity. The exact proof can be found in [8].

$$\max(|x|,|y|) - \min(|x|,|y|) \leq ED(x,y) \leq (\max(|x|,|y|) + \min(|x|,|y|)) \frac{J(X,Y)}{2 - J(X,Y)}$$

3.3.2 Custom Jaccard Similarity to Edit Distance Conversion

While the inequality derived above holds as long as the k-mer frequencies are low, the size of the bound is often quite large. For example, if the two sequences are of equal length, the lower bound is always 0. To solve this problem, we

[3] `http://gembox.cbcb.umd.edu/mhap/raw/ecoli_p6_25x.filtered.fastq`

decided to come up with a method to provide a point estimate for the edit distance. Our two sequences are a and b, and the k-mer sets of these sequences are A and B respectively. Let's assume that $|a| \geq |b|$.

We will assume that all the differences can somehow be represented on b. Basically, we are setting a to be a sort of *ground truth* sequence and marking where b differs from a. Figure 5 shows this representation. Now, we make another

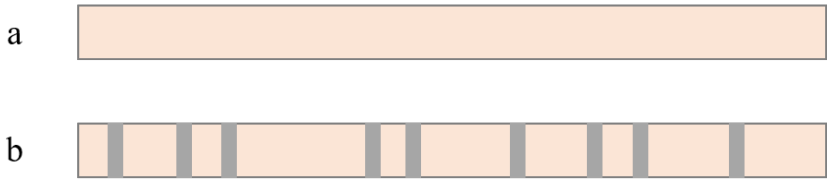


Figure 5: Representation of Mutations

assumption that these differences (mutations) are uniformly distributed. If we have x differences and $|a| = L$, the mutations will partition the sequence into $\frac{L}{x+1}$ parts. If we assume that each k-mer overlaps one mutation at most, i.e. $k \leq \frac{L}{x+1}$, then we can assume that k k-mers will be affected by a single mutation. See Figure 6.

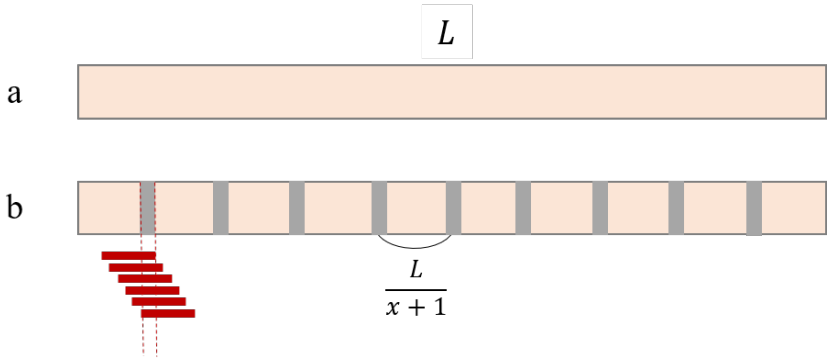


Figure 6: Representation of Mutations. Red bars indicate the k-mers affected by the corresponding mutation.

The assumption that $k \leq \frac{L}{x+1}$ will most likely hold for the genomic sequences we are focusing on. The average length of third generation sequencing reads are around 10kbp, and the k values typically used for MinHash are between 10-30. Thus, except for the more extreme cases where the normalized edit distance is greater than 30%, our assumption will hold.

Our final assumption is that the resulting "mutated" k-mers (i.e. the red bars in Figure 6) do not exist in A. This assumption will also most likely hold for our dataset. The maximum possible number of unique k-mers in a 10kbp sequence is $10000 - k + 1$. However, the total possible number of k-mers with an alphabet size of 4 (ACGT) is 4^k. Thus, assuming $k = 20$, we can see the probability of a mutated k-mer not existing inside the original sequence is 0.99999999092.

With these assumptions, we derive our point estimate for the edit distance. If the true edit distance is x, we can assume that kx k-mers were mutated and no longer match the k-mers in A. Thus, we can say that

$$|A \cap B| = |A| - kx$$

Using the inclusion exclusion principle, we can also calculate $|A \cup B|$.

$$\begin{aligned} |A \cup B| &= |A| + |B| - |A \cap B| \\ &= |A| + |B| - |A| + kx \\ &= |B| + kx \end{aligned}$$

Therefore, the relationship between the Jaccard coefficient and edit distance will be

$$J = \frac{|A \cap B|}{|A \cup B|} = \frac{|A| - kx}{|B| + kx}$$

$$x = \frac{|A| - J|B|}{k(J+1)}$$

We will evaluate the performance of this point estimation for edit distance using both simulated and real genomic sequencing data.

4 Results

4.1 Jaccard Similarity Comparison & Tuning MinHash Parameters

We compared the absolute value difference between the true Jaccard similarity, which was calculated from the set of k-mers, and the estimated Jaccard similarity from each of the MinHash implementations. The following figures plot the Jaccard Similarity Error versus a varying number of hash functions and varying k-mer lengths for each MinHash implementation.

Based on Figures 7, 8, 9, and 10 below, for each implementation of MinHash and for all numbers of hash functions used, as the length of k-mers increases, the error in the Jaccard similarity decreases. Although this may seem like the largest k-mer length is optimal k-mer, it is important to note that the Jaccard similarity error is not the only metric to consider here. As the k-mer length increases, the chance that the k-mer will overlap an edit in the genome edit transcript increases to a point where there will not be any k-mers that do not overlap an edit. This results in no matching k-mers between the sketches of the two sequences being compared, so therefore an intersection of 0 between the two k-mer sets, giving both a true Jaccard similarity of 0 and an estimated Jaccard similarity of 0, and therefore a Jaccard similarity error of 0. So the idea of a declining error with an increasing k-mer length is misleading. On the opposite extreme, if k-mer lengths are to small, there will be too many coincident matches just by chance of the short k-mer occuring, which will result in a lot of matching k-mers between the sketches of the two sequences being compared, so therefore an intersection close to that of the union of the two k-mer sets, giving both a true Jaccard similarity of close to 1 and an estimated Jaccard similarity of close to 1, and therefore a Jaccard similarity error of close to 0. So, the idea here is to find a k-mer length approximately in-between the two extremes. Based on our plots, we chose the optimal k-mer $= 12$, as it is at approximately the midpoint in the declining error and in our data resulted in the fewest extreme Jaccard similarity values of 0 and 1.

The number of hash functions L seems to be generally constant across the board, not significantly and consistently affecting the Jaccard similarity error.

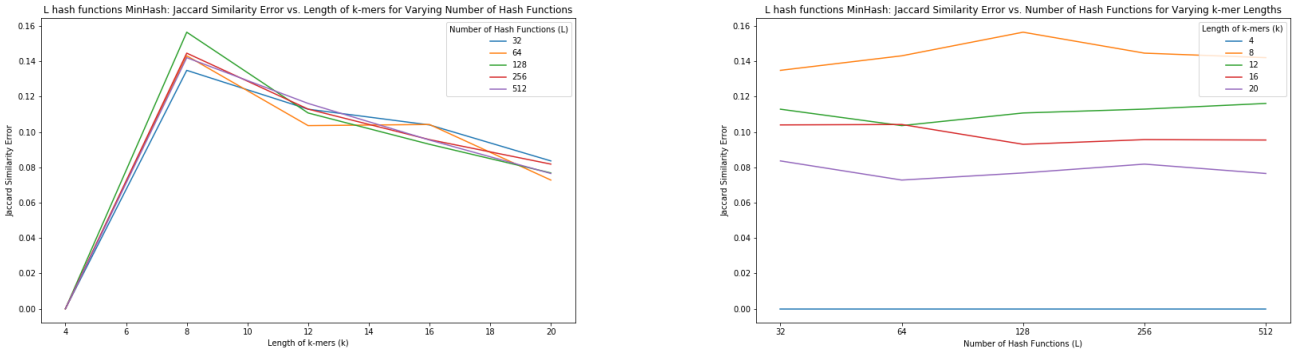
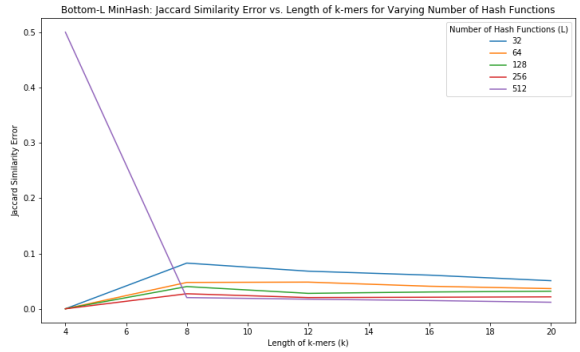


Figure 7: L hash functions MinHash Jaccard Similarity Error

While it is difficult obtain a fidelic approximation of edit distance from Jaccard Similarity, we infer that generally as edit distance increases, there should be fewer similarities, and thus the Jaccard Similarity should also decrease. If the k-mer lengths are too small, then Jaccard Similarity will stay at 1, since it is likely two sequences that have small edits relative to their lengths should share similar fingerprints. Similarily, large k values will cause Jaccard to go to 0, since it is less likely for long k-mers to have exact matches in the presence of edits. These extremes, $J = 0$ and $J = 1$, cannot yield any information about our edit distance.

To establish a robust choice for k-mer length and avoid these extremes, we calculated the true Jaccard Similarity across k-mer sets retrieved from 8000 bp synthetic reads, as seen in Figure 11.

We can see that if $k = 4$ the true Jaccard similarity always evaluates to 1, thus rendering the Jaccard similarity useless for edit distance estimation. On the opposite end of the spectrum, we can see that for k values of 16 and 20, the Jaccard similarity is quite close to 0 for normalized edit distance of only 0.12. Therefore, we decided to use $k = 12$ as our optimal k-mer size.



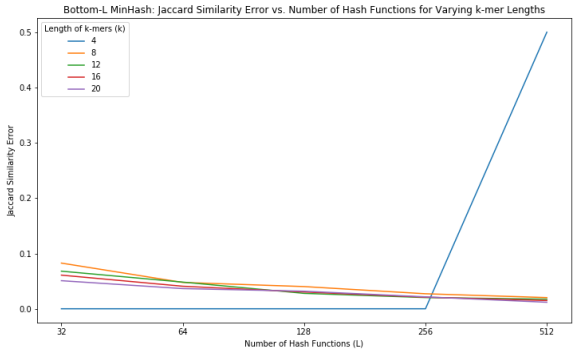
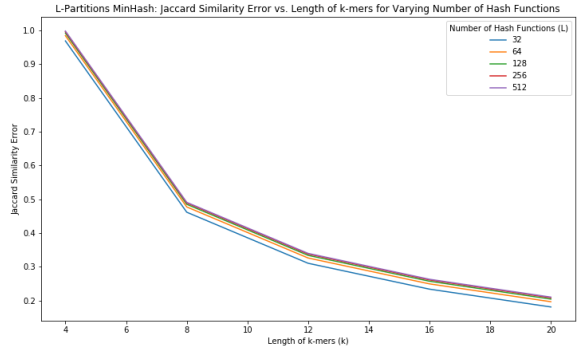


Figure 8: Bottom-L MinHash functions MinHash Jaccard Similarity Error



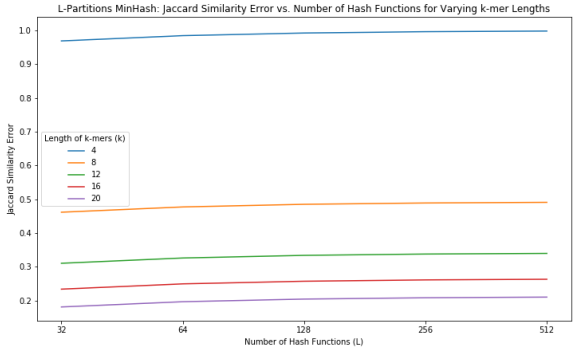
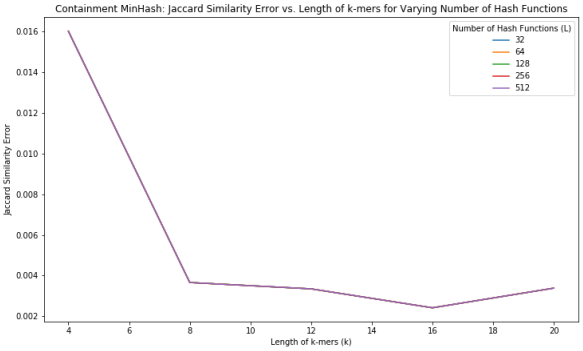


Figure 9: L-Partitions MinHash Jaccard Similarity Error



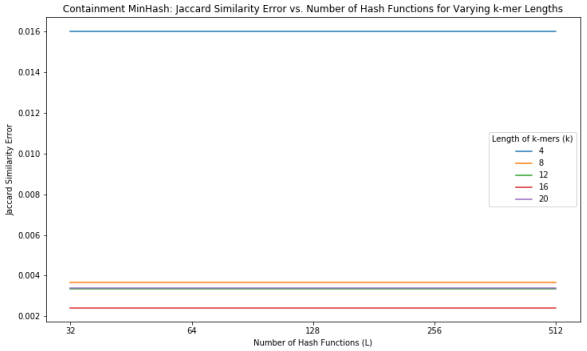


Figure 10: Containment MinHash Jaccard Similarity Error

4.2 Edit Distance Comparison

To evaluate our point estimate of the edit distance, we computed the true edit distance of the sequence pairs in the synthetic dataset using Needleman-Wunsch. We then calculated the estimate of the edit distance using our formula using the Jaccard similarities derived by the four different MinHash implementations. Figure 12 shows that our formula is able to generate a good estimate for the true edit distance. The purple line is $y = x$, and we can see that our estimates from bottom-L, L-partitions, and containment MinHash are all distributed close to this line. We did however noticed

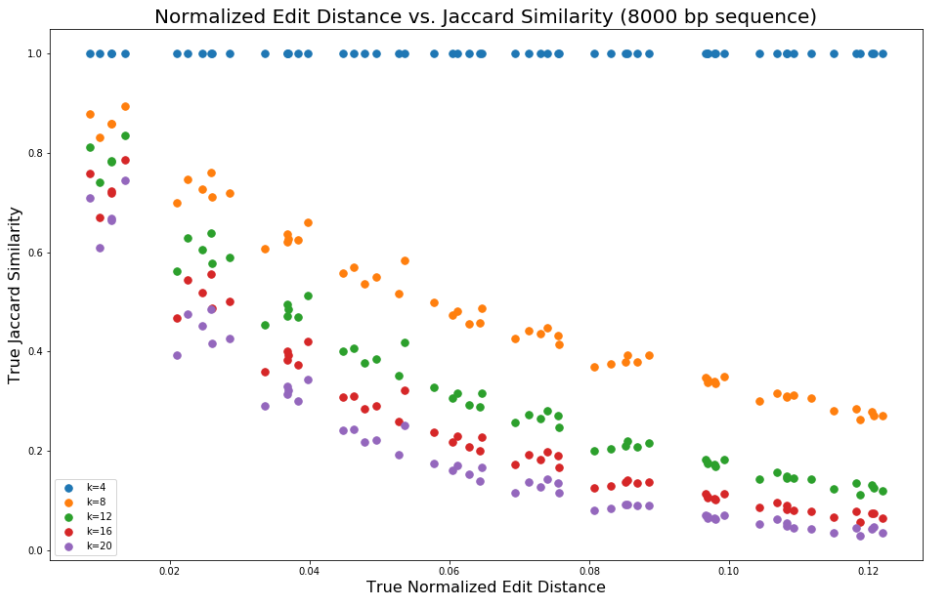


Figure 11: $k = 4$ yields Jaccard of 1, which shows no correlation with increasing edits.

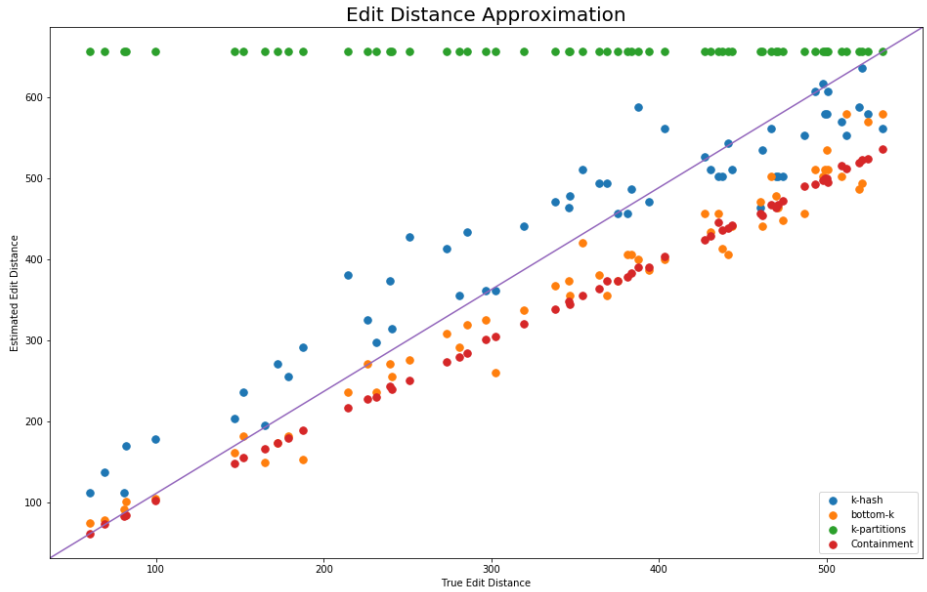


Figure 12: Approximate Edit Distance for synthetic data calculated with $k = 16$, $L = 128$ across MinHash variants

that our implementation of L-partition always seemed to output a constant value, which indicates that there is some error with our implementation. Thus we will exclude that from our analysis. Overall, we can conclude that our point estimate formula performs well for our synthetic dataset.

MinHash Scheme	Time (function calls in total seconds)	Memory (MiB)
L Hash Functions	10221380 fc in 2.080 s	44.8 MiB
Bottom-L	79903 fc in 0.019 s	45.3 MiB
L-Partition	95873 fc in 0.026 s	44.9 MiB
Containment	199649 fc in 0.092 s	47.0 MiB

Table 1: Each MinHash scheme was run using on two sequences of 8001 nucleotide base pairs, with a true edit distance of 98. The parameters used were: k-mer length 16, 128 hash functions, and stride length 1.

4.3 MinHash Performance Benchmarking

From the benchmark results summarized in Table 1, bottom-L and L-partition had comparable number of function calls and total time execution. These results are reasonable since both variants make use of a single hash faction and compute a fingerprint by taking the bottom L hash value or minimum hash value from L partitions, respectively. Containment MinHash had about twice as many function calls and required almost twice the time to complete execution. This can be explained by the extra overhead required to initialize and populate the Bloom Filter. Most of this overhead comes from the hash function within the `BloomFilter` class, which was called 95820 times and required 0.055 seconds. Finally, L hash functions had the worst performance of the base MinHash implementations, calling a staggering 10221380 functions in 2.080 seconds. This massive overhead came from having to generate the 128 distinct hash functions. For example, there were 2044160 function calls each `xxhash.xxh32`'s `update`, `intdigest`, and `reset` methods, which took 0.231, 0.166, and 0.116 seconds, respectively. Clearly, using L distinct hash functions is computationally expensive, and there was no positive payoff to this expense as was seen in the previous section. These time benchmarks were run using Python's `cProfile` profiler.

From a memory usage standpoint, each method was significantly more practical than actually storing the complete sequence of k-mers. The L hash functions, bottom-L, and L-partitions each were comparable in their memory usage, requiring about 45 MiB of space. This is reasonable since each of these variants followed the same framework, and did not require auxilary data structures. In contrast, Containment MinHash required 47 MiB of space, which is explained by populating and storing a Bloom Filter. These memory benchmarks were run using Python's `memory_profiler`.

5 Conclusion

We conclude that MinHash is indeed a good way to estimate Jaccard similarity and edit distance. As previously discussed, MinHash is much more conservative with memory and runtime than the naive Needleman-Wunsch algorithm for finding edit distance.

5.1 Future Work

We made several goals for this project that fell short because of time constraints. Further areas of study are discussed. We were also not able to include our results from the real E.coli data due to the large size of the dataset. Our scripts are still currently running, and we hope to have results for that by the time of our final presentation.

5.1.1 Adversarial Performance

Generating the ground truth label for edit distance across all pairs of sequences took exceptionally long; all sequences were at least 8000 nucleotides long, and true edit distance was retrieved using the dynamic programming global alignment approach. Both space and time complexities are bounded by $O(n^2)$, where n is the sequence length. To find edit distance on random sequences took several hours, and we could not simulate on sequences with repeating patterns.

With more time, we would have generated adversarial patterns similar in length to our optimal k-mer length. This data setting could potentially cause MinHash to perform poorly if there are many edits contained outside these patterns.

5.2 Contributions

Each member contributed significantly to the project. We each implemented one of the hashing schemes: Laurent implemented L hash functions, Joanna implemented bottom-L and the Needleman-Wunsch dynamic programming edit distance algorithm, JungMin implemented L-partition, and Darius implemented Containment MinHash and the multi-layer hashing modification. Laurent created the pipeline to generate synthetic data and randomly insert edits into a given sequence. JungMin downloaded and processed the E Coli. data and computed the true edit distances and the hash value datasets. Joanna created the method process figure. JungMin and Darius ran the experiments. Joanna plotted all

the Jaccard similarity comparison results to the tune all the MinHash parameters, and Laurent and Darius visualized the Edit Distance vs. Jaccard Similarity results. Darius did the benchmarking, and Laurent wrote asymptotic analyses of our implementations.

We all learned a lot in the process and were able to help each other with troubleshooting issues.

References

[1] Needleman, Saul B. Wunsch, Christian D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology. 48 (3): 443–53. doi:10.1016/0022-2836(70)90057-4. PMID 5420325.

[2] A. Z. Broder. On the resemblance and containment of documents, in Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171), Salerno, Italy, 1998, pp. 21–29.

[3] V. Popic and S. Batzoglou, "Privacy-Preserving Read Mapping Using Locality Sensitive Hashing and Secure Kmer Voting," Bioinformatics, preprint, Apr. 2016.

[4] Cormen, Thomas H, and Thomas H. Cormen. Introduction to Algorithms. Cambridge, Mass: MIT Press, 2001. Print.

[5] Lander, E., Linton, L., Birren, B. et al. Initial sequencing and analysis of the human genome. Nature 409, 860–921 (2001) doi:10.1038/35057062

[6] D. Koslicki and H. Zabeti, "Improving Min Hash Via The Containment Index With Applications To Metagenomic Analysis," Bioinformatics, preprint, Sep. 2017.

[7] S. J. Swamidass and P. Baldi, "Mathematical Correction for Fingerprint Similarity Measures to Improve Chemical Retrieval," J. Chem. Inf. Model., vol. 47, no. 3, pp. 952–964, May 2007.

[8] S. Dolev, M. Ghanayim, A. Binun, S. Frenkel, and Y. S. Sun, "Relationship of Jaccard and edit distance in malware clustering and online identification (Extended abstract)," 2017 IEEE 16th International Symposium on Network Computing and Applications (NCA), 2017.