Application of Bloom Filters in Determining Genome Variation

Randy Kuang, Eric Rong, Amritpal Singh, Dikshith Kasimahanthi

601.447 Computational Genomics, Johns Hopkins University
Instructors: Ben Langmead and Brad Solomon December 3rd, 2019

1 - Abstract

Bacteria and viruses are two of the most common causes of disease in humans and other higher forms of life. This is in part the result of their biology, which allows them to easily mutate into diverse strains. These strains make it difficult for both the immune system to fight off these organisms, and for scientists to develop methods to aid the immune system. We aim to develop tools to identify regions of DNA that are unique among the varying strains, and are thus likely the cause of the differences that prevent an effective immune response. We aim to do this by creating a data structure that stores the k-mers of a strain, of either a virus or bacterium. After we do this for many strains, we intersect and/or union these data structures, resulting in a final merged data structure that contains the common k-mers across several different strains. Then, we take a test genome which is not present in our database of training strains, and determine the overall similarity of those genomes to our database.

2 - Introduction

In a single year alone, over 3 million people die from vaccine-preventable diseases and other disease-related deaths[1]. Two of the most common causes of death are from diseases caused by bacterial and viral infections. An exploration of their biology shows why this is so.

Bacteria and Viruses come in many different strains. The bacterial immune system produces antibodies that bind to the bacteria's antigens on the surface of the cell. However, these antibodies are extremely specialized, which makes it that different strains of the same species often cannot be recognized by the same antibodies. Thus, the immune system must design new antibodies to fight off

these strains. While this process takes place, however, the bacterial cells can seriously cause damage to the human body. In the case of viruses, the immune system can use antibiotics to eradicate a virus before it infects a cell. However, antibodies are unable to do fight off the infection after a cell is infected, because viruses take control of the cell's replication machinery in order to reproduce and further spread the virus. The cell responds by releasing a protein called interferon which notifies the immune system and its neighboring cells. The immune system can then use antibodies and other immune cells to try to neutralize the virus before it infects a single cell.

One other reason that viruses and bacteria are especially harmful is because they reproduce very quickly, thereby having high mutation rates. Viruses that infect a single cell are able to reproduce and multiply thousands of times. Furthermore, these copies have a very high chance of being functionally different from the original due to mutations. There are two types of viral mutations that occur: antigenic drift, where mutations occur in the surface proteins of the virus that make it recognizable by the immune cell, and antigenic shift, in which two viruses combine genetic material. This often results in surface proteins that are a combination of both viruses. The quick replication rate of bacteria, whose populations double every 20 minutes (on average), can cause mutations to naturally form and accumulate over every bacterium's replication cycle. This high replication and mutation rate of bacteria and viruses can cause new strains to form, and these strains often inhibit the immune response in humans because the immune system is unable to recognize these new strains.

To fight off infections, antibiotics can be used to assist the immune system in recognizing bacteria, since they are antibodies themselves. Antivirals assist the immune system by making it difficult for viruses to reproduce. Both of these man-made methods require specificity, which is inhibited by the mutations which occur, so researchers need a method to assist in the creation of these medications. In order to address this, we can a method that will be able to identify regions of high similarity between the genomes of disease causing organisms. We can create a database of the most common k-mers across many different strains of bacteria and virus, and compare them to determine those critical regions.

A problem arises, however, when we see that many of these genomes are far too large to store. To address this, we can implement an approximate data structure in order to minimize the amount of space needed while still maintaining the functionality needed. Our first choice of data structure was the bloom filter. The bloom filter allows us to store k-mer information in a much smaller storage area by using many hash functions to store in a list of bits. A counting filter is an improvement on the bloom filter that uses more space, since it keeps track of how many k-mers that have hashed to the same index in the list of integers, which allows researchers to delete k-mers from the combined genome database. Finally, in order to determine the accuracy of our approximate data structures, we compared our approximate data structures to a HashSet implementation,

which preserves 100% accuracy.

3 - Prior Work

Researchers Wood et. al.[10] have developed a metagenomic analysis technique known as Kraken. In Kraken, labels are assigned to metagenomic sequences based on a k-mer's least common ancestor (LCA), which are then put into a database. The LCA is the closest ancestor which uses that specific k-mer, meaning that all of its children also contain that k-mer. When a k-mer is queried, Kraken returns the list of taxa associated with that k-mer. This method finds commonalities very quickly across all species and genomes using an exact alignment of k-mers. Our project also aims to do this, however, we want a more direct method for evaluating different strains of bacteria and viruses.

Another method we found was the Scalable Metagenomic Classification by the researchers Ames et. al[11]. In this method, the researchers used an offline computational method that which labels k-mers that are stored in a database. But in this method the researchers use an ordered list of candidate taxonomic labels based on k-mer overlap. This overlap determines the score of a k-mer which contributes to its final taxonomic label.

4 - Methods and Software

Note: In the following section, the term "merged data structure" will refer to a data structure that is the result of taking the union and/or intersection of the k-mers contained in other data structures. Furthermore, an “individual merge" refers to pairwise merging of all data structures, so that afterwards, there are half as many data structures left. For example, an individual merge on 10 data structures would result in 5 merged data structures, where the data structure at position i would be merged with the data structure at i+1, the data structures at i+2 would be merged with i+3, and so on in the same pattern until the last data structure.

The specific objective of the project was to implement various sketch data structures that support querying, use those sketches to store different strains of HIV and E. Coli, and then merge sketches together to find the most common k-mers. Note that our methods seek to determine the most common k-mers, and that we are assuming that k-mers present in the genome but not in the data structure are the ones that are most dissimilar among strains. We will empirically test this claim to verify our assumption.

4.1 - Data Collection

We chose a quantity of strains equal to a power of 2 in order to simplify the inherently binary merging process. In other words, during each merge, each strain would be merged with another strain, and we would always have $\log_2 t$ total merges, in which t is the strain count.

We therefore downloaded 32 strains of HIV from the Los Alamos National Laboratory's HIV database[2] to use in our project. The average length of our HIV genomes was 9670 nucleotides. Due to the relatively short length of the HIV genome, a few of our original 32 strains were exactly the same, so these duplicate genomes were replaced by new strains to force some amount of genetic variation. We also downloaded 5 additional strains to use for testing purposes.

Next, we downloaded 4 strains of Escherichia Coli from the University of Wisconsin-Madison's Genetics database[3] to use in our data structures. The average length of our E. Coli genomes was approximately 4.7 million nucleotides. Due the relatively long length of the E. Coli genome, we only had the computational resources to merge together 4 strains. We also downloaded 2 additional strains to use for testing purposes.

4.2 - Data Structures

To build our data structures, we started with a base Set abstract class that supports the following methods:

- A `contains(e)` method that checks whether an element is in the Set.
- An `insert(e)` method that inserts an element into the Set.
- A `union(ds)` method that unions all the k-mers in the parameter data structure into the calling data structure.
- An `intersection(ds)` method that intersects all the k-mers in the parameter data structure into the calling data structure.

The first Set implementation we created is a HashSet that uses a Python dictionary to store an exact count of each k-mer by mapping a unique k-mer to the number of times it appears in the sequence. When a HashSet A *unions* itself with another HashSet B to create a union Hashset C, every k-mer in C's value is the sum of that k-mers value in A and B. When a HashSet A *intersects* itself with another HashSet B to create an intersect Hashset C, every k-mer in C's value is the minimum of that k-mers value in A and B.

The next two Sets we implemented were sketches: a Bloom Filter and a Counting Filter. The bloom filter follows the implementation outlined in class, i.e. a k-mer x is inserted by hashing it against h hash functions to get indices in a bit array, setting those indices to a binary value of 1. A k-mer x is contained in the sketch if the indices returned by all h hash functions is 1, and is otherwise not contained. Unions and intersections are calculated by taking a the bitwise or and bitwise and between the bit arrays of two bloom filters. Recall that as mentioned in class, bloom filter intersection by bitwise and can cause additional errors due to hash collisions. Despite this downside, we still chose to intersect bloom filters by bitwise and because no better solution exists.

Finally, when Bloom Filters are created, their constructor takes in a desired false positive rate as well the count of element to be inserted. Then, the filters uses the equations covered in class to calculate an optimal bit array length and an optimal number of hash functions. Specifically, for h hash functions, a bit array of length m, n items to be inserted, and a false positive rate FPR, we have the following:

$$FPR = (1 - e^{-\frac{hn}{m}})^h \qquad \textbf{Equation 1.}$$

$$h = \frac{m}{n} * \ln 2 \qquad \textbf{Equation 2.}$$

Then, we can substitute **Equation 2.** into **Equation 1.** and solve for the variable m to find the following:

$$m = -\frac{n * \ln FPR}{(\ln 2)^2} \qquad \textbf{Equation 3.}$$

Since we are given FPR and n as constructor parameters, we can use **Equation 3.** to calculate the optimal bit array length, and then use the optimal bit array length to find the optimal number of hash functions with **Equation 2.**.

Like the Bloom Filter, the counting filter also uses an array internally to sketch the k-mers it contains, so it uses the same equations above to calculate optimal array length and number of hash functions using input paramters n and FPR. When k-mer x is inserted into the counting filter, instead of flipping indices returned by the h hash functions to 1, all indices are instead incremented. Furthermore, a k-mer x is contained in the Counting Filter if all indices returned by its hash are at least 1, and not contained otherwise. Finally, unions/intersections for Counting Filters follow the strategy used for the HashSet. In other words, for every index i in the integer array for two Counting Filters A and B, $A[i]$ is added to $B[i]$ to find the union, and $min(A[i], B[i])$ is calculated to find intersection.

4.3 - Merging Algorithm

Before going into our algorithm, we will give a few notations and general notes for clarity.

Recall that an “individual merge" refers to a pairwise union or intersection of all current data structures so that afterwards, we are left with half as many. Furthermore, an “individual intersection" and “individual union" refer to pairwise intersecting/unioning all data structures once, so that afterwards we are left with half as many. We always chose to do intersections before unions, i.e. we may do 2 individual intersections, and then 3 individual unions, but never 3 individual unions and then 2 individual intersections.

For our tests on HIV strains, we first read in all 32 strains into 32 HashSets, 32 Bloom Filters, and 32 Counting Filters. Thus we had a total of 5 individual merges($2^5 = 32$) on the HIV data structures to get final merged data structures which contained information about all 32 strains. Since the goal of the HashSet was to serve as an absolute truth, as it never has errors or ambiguity, we only unioned HashSets. Thus, the final HashSet states how often each k-mer appears across all 32 strains.

The goal of the Bloom Filter and Counting Filter was to serve as a space efficient way of storing common k-mers. Thus, for these two data structures, we experimented with different amounts of individual intersections, followed by individual unions until we were left with 1 data structure. The results of these tests are described in results below.

Our methodology for the E. Coli strains was identical to what we did for HIV. However, since we only had 4 E. Coli strains for training, we only had two merging options for the Bloom/-Counting Filter: 2 individual unions, or 1 individual intersection followed by 1 individual union.

5 - Results

Note: Instructions for running any of the tests which generated the figures below are contained in the README.md file in the submitted code base folder.

One of the key properties of using Bloom Filters (and Counting Filters by extension) instead of a HashSet in storing the distinct k-mers that are present in different genomes is that Bloom Filters have significantly improved space efficiency at the cost of some accuracy. Our three data structures are evaluated on the metrics of space, time, and accuracy in the figures below.

5.1 - HIV Space Analysis

Figure 1 below shows the space required by our 3 data structures in bytes for various k-mer lengths. Note that the bloom filter and counting filter both have constant size regardless of k-mer length because their length internally is based only on the number of k-mers to be inserted, which is approximately similar as we vary k-mer length. Although it can't be seen in the figure, the counting filter takes about 32x the space as the bloom filter because it uses an integer array internally rather than a bit array, and integers use 32 bits in Python. The size of the bloom filter was 12,000 bytes and the size of the counting filter was 380,000 bytes.

As shown in the figure, the HashSet always takes significantly more space than the bloom filter and counting filter. Even the smallest HashSet that was created with length 20 k-mers is 160x larger than the bloom filter. This is mainly due to the fact that the HashSets store entire strings instead of bits or integers. Note that the size of the HashSet increases as k-mer length increases because longer k-mers lengths result in more unique k-mers for the HashSet to store.

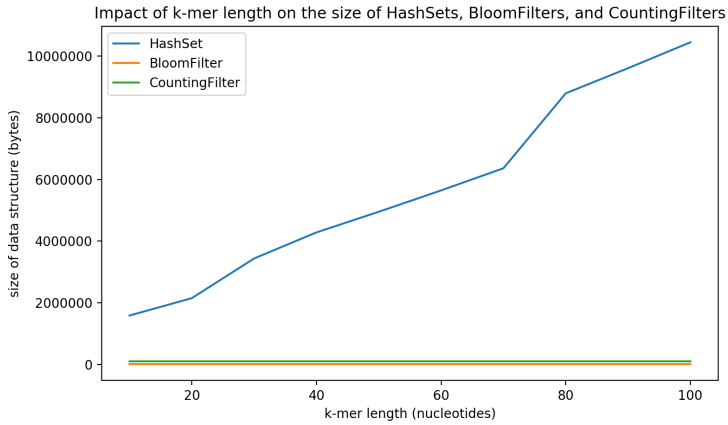
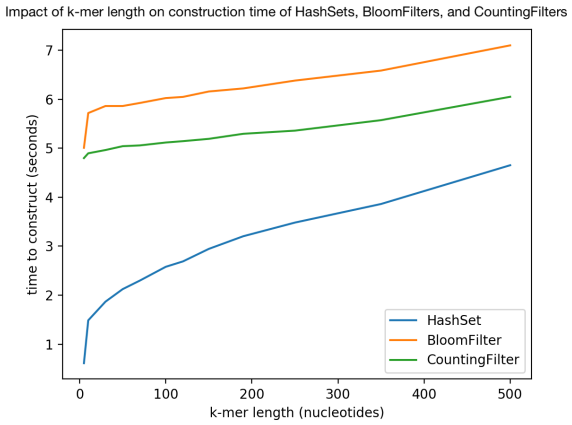


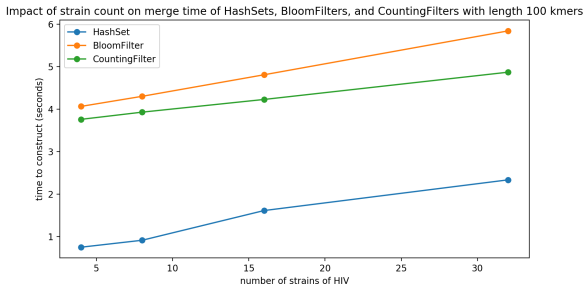
Figure 1. Size of the HashSet, Bloom Filter, and Counting Filter in bytes as k-mer size varies

5.2 - HIV Time Analysis

However, the massive savings in space using a bloom and counting filter are at the expense of time and accuracy. In **Figure 2** below we see that when altering both k-mer length and the number of strains to merge, the bloom filter takes the most time to construct, followed by the counting filter and HashSet. However, the magnitude of the difference in time usage is far smaller than the magnitude of the difference in space usage.



(a) Figure 2.1



(b) Figure 2.1

Figure 2. Time (seconds) to populate the 3 data structures with the HIV genome vs. k-mer length and number of strains

5.3 - E. Coli Time Analysis

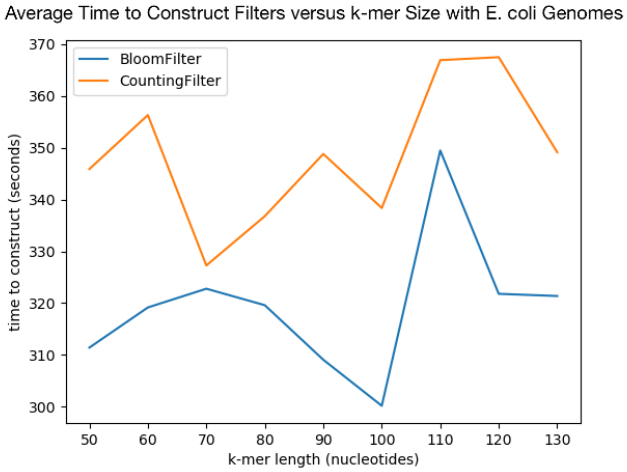


Figure 3. Size of the HashSet, Bloom Filter, and Counting Filter in bytes as k-mer size varies

In **Figure 3**, we plot the time needed to construct the E. Coli merged bloom filter vs. k-mer length. The first property to note about this graph is that there isn't any clear relation between construction time and k-mer length. This is most likely because the total running time was so long (8 hours) that fluctuations in the laptop running the construction changed the running time. The other notable trait is that it takes much longer to create the final merged bloom filter on E. Coli genomes than it does for HIV. From this figure, we unfortunately conclude that our method is significantly slower than other methods described in our Previous Work section. For example the authors of the Scalable Metagenomic Classification method[11] state that they can process more than a million bases per second, which is clearly faster than what our algorithm is able to do, given our algorithm needs 7+ hours merge the 5 million base E. Coli genome.

5.4 - HIV Accuracy Analysis

The most important measure of our project is the accuracy of the final merged bloom filter. In order to generate **Figure 4**, we created different merged bloom filters based on different numbers of individual intersections. For each of these bloom filters, we compared their contents against the 100% accurate merged HashSet to generate accuracy measures.

When calculating accuracy (blue line), we only queried the bloom filter for k-mers that appear in the merged HashSet at least 2^i times, where i is the number of intersections. The intuition behind this is that if we intersect our bloom filters only once, we expect only k-mers that appear at least twice (once in two different strains), to appear in the final merged bloom filter. If we intersect our bloom filters twice, then we only expect k-mers that appear at least 4 times (once in two different strains), to appear in the final merged bloom filter, and so on. We see on the graph that the merged Bloom Filter accuracy is fairly high for any number of intersections, which means that our merged bloom filter does indeed contain the most common k-mers across the 32 original strains.

We also plot a number of intersections against the false positive rate for the final merged bloom filter. The false negative line (orange) measures the percent of k-mers that shouldn't be in the final merged bloom filter, but are. To test this, we simply queried for 10,000 random strings that should not have been in the final bloom filter, and measured the amount that showed up anyway. As shown in the figure, using more individual intersections to create the merged bloom filter resulted in a lower false positive rate, which is more likely because with more intersections, the final bit array is more sparse, and thus the chance of a false positive is decreased.

As previously mentioned, **Figure 4** tells us how accurate the final merged bloom filter was. However, we also want to analyze the k-mers that were not contained in the final merged bloom filter when they should have been. **Figure 5** shows us that the average length of false negatives versus the number of intersections. There are a few important conclusions that can be drawn. Firstly, we see that for any number of intersections, not only does the merged bloom filter have high accuracy (as indicated by **Figure 4**) but the k-mers that the merged bloom filter misses are on average not the most common ones, and thus missing them is not detrimental. This means that on average, the most common k-mers across strains are present in the final merged bloom filter, which aligns with our original project goal.

The second conclusion that is worth mentioning is that the average length of false negatives slightly increases as the number of intersections increases. This is most likely because more individual intersections results in less total k-mers in the final merged bloom filter, so many k-mers are missed.

As a whole, our accuracy analysis tells us that the final merged bloom filter does a relatively good job at tracking the most common k-mers between strains, in spite of the fact that bloom filter intersection conceptually can fail due to collisions. The relatively high accuracy of the final merged bloom filter also justifies our previous assumption that the k-mers not present in the final merged bloom filter are in fact uncommon k-mers that are potential sites of mutation.

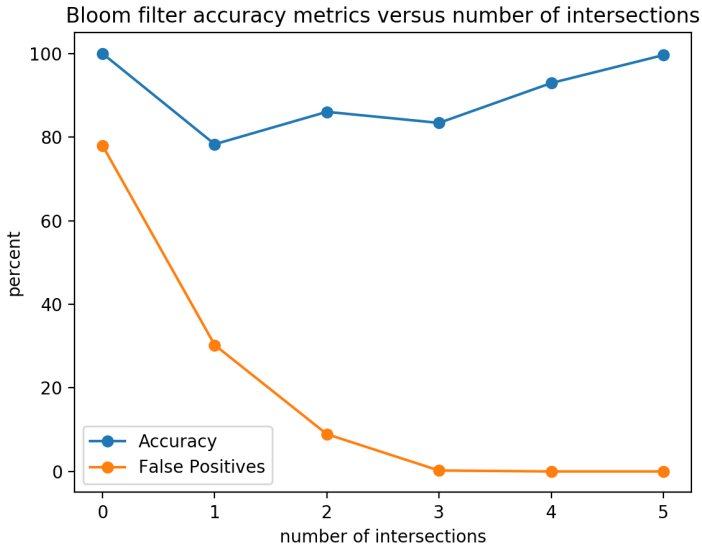


Figure 4. The accuracy, false positive rate, and false negative rate of Bloom Filters vs. the number intersections

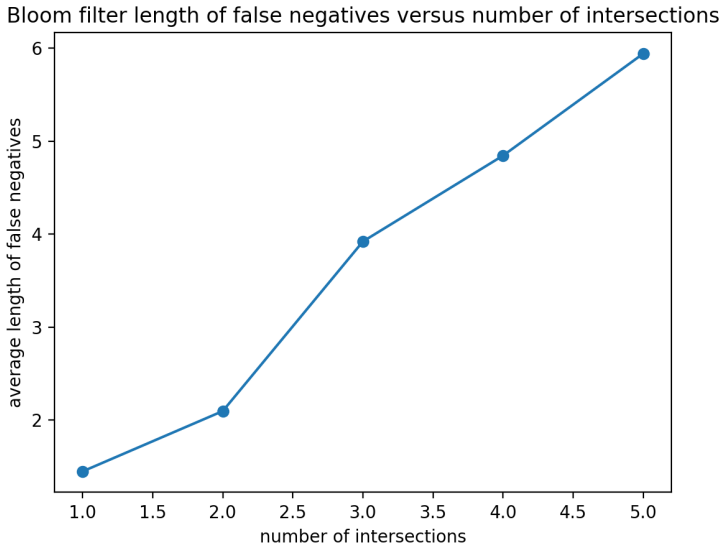


Figure 5. The average length of k-mers that should have been in the merged bloom filter but were not

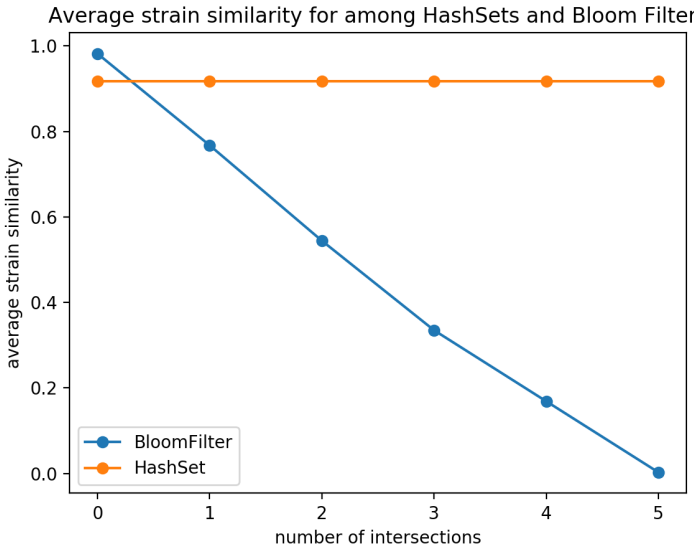


Figure 6. Strain similarity between final merged bloom filter and test strains

5.5 - HIV Strain Similarity

As shown in **Figure 6**, as the number of intersections on the bloom filter increases, the set similarity between the final merged bloom filter and test HIV strains that were not part of the 32 strains used to construct the bloom filter, decreases. This relationship can be explained by the fact that increasing the number of intersections decreases the number of k-mers contained in the final bloom filter, which will reduce set similarity.

6 - Conclusions

Overall, our project was able to accomplish most of our main goals, although we failed to hit our reach goals. We found that it took too much time to construct our data structures for the E. Coli genome. As a result, our method isn't feasible in practice, nor is it better than any existing, more efficient methods, such as Scalable metagenomic taxonomy classification[11]. With regards to our other HIV strain data, however, we obtained reasonable accuracy, space, and similarity measures. A significant goal that we weren't able to accomplish was generating data for accuracy using E. Coli genomes and bloom/counting filters. Unfortunately, we were unable to test how accurate these data structures were because we couldn't compare them to a HashSet, as inserting all k-mers of an E. Coli genome into a HashSet occupied more memory than we could handle with our machines. We would've liked to do this to see if our method worked with much larger genome sizes.

Although our strain similarity results may seem counter intuitive, since we identified few of the

test strain k-mers in the final merged bloom filter, it makes sense for this to occur on the HIV genomes. HIV genomes are approximately 10,000 nucleotides in length and we used a k-mer length of 100. Matching 100 consecutive nucleotides in such a small genome across only 32 strains will be understandably rare. We were unable to test this across more k-mer lengths because we were attempting to run 100 k-mer lengths on E.Coli strains. This too, however, failed because of memory constraints. Unfortunately, we did not have the computational resources to demonstrate our results on the E. coli data, in addition to the HIV data.

We also wanted to mention that our code, after establishing a data structure to hold the combined k-mers, is capable of returning whether a specific k-mer is common or uncommon across many strains. This is exactly what our original goal stated in the abstract was - being able to tell whether a specific k-mer is uncommon across many strands can give researchers information about what regions of a strain are important. We included instructions in the README.md file found in the base folder of our code submission for running this code.

Finally, note that our counting filter is essentially a bloom filter variant that supports extra methods, but takes up more memory. Two benefits of a counting filter are that k-mers can be deleted, and specific k-mer counts can be returned. In the future, we would like to investigate the usefulness of these additional features supported by the counting filter.

References

1. Global Immunization: Worldwide Disease Incidence. Children's Hospital of Philadelphia, The Children's Hospital of Philadelphia, 1 Dec. 2014, https://www.chop.edu/centers-programs/vaccine-education-center/global-immunization/diseases-and-vaccines-world-view.
2. HIV Sequence Database. (n.d.). Retrieved from https://www.hiv.lanl.gov/content/sequence/HIV/mainpage.html.
3. A Systematic Annotation Package for Community Analysis of Genomes. (n.d.). Retrieved from https://asap.genetics.wisc.edu/asap/download_Source.php?LocationID=&SequenceVersionID=&GenomeID=.
4. Bloom filter. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Bloom_filter.
5. Broder, A., & Mitzenmacher, M. (n.d.). Network Applications of Bloom Filters: A Survey. Retrieved from https://www.cs.rochester.edu/~stefanko/Teaching/13CS484/BM.pdf.
6. Solomon, B., & Kingsford, C. (n.d.). Large-Scale Search of Transcriptomic Read Sets with Sequence Bloom Trees. Retrieved from https://www.biorxiv.org/content/10.1101/017087v1.
7. Bender, M. A., Zadok, E., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B. C., Spillane, R. P. (2012). Dont thrash. Proceedings of the VLDB Endowment, 5(11), doi: 10.14778/2350229.2350275
8. Pandey, P., Bender, M. A., Johnson, R., & Patro, R. (2017). A General-Purpose Counting Filter. Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD 17. doi: 10.1145/3035918.3035963
9. Jackman, S., Vandervalk, B., Mohamadi, H., Chu, J., Yeo, S., & Hammond, S. A. (n.d.). ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5411771/.
10. Wood, D. E., & Salzberg, S. L. (n.d.). Kraken: ultrafast metagenomic sequence classification using exact alignments. Retrieved from https://genomebiology.biomedcentral.com/articles/10.1186/gb-2014-15-3-r46.
11. Ames, S. K., Hysom, D. A., Gardner, S. N., Lloyd, G. S., Gokhale, M. B., & Allen, J. E. (n.d.). Scalable metagenomic taxonomy classification using a reference genome database. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/23828782.