Clustering of RNA-seq reads without a reference transcriptome

Caleb Andrade, Scott Harvey, Soumadip Mukherjee, Avi Srivastava

I. Introduction

The basic idea of the paper by Brian et.al[2] is to use Latent Semantic Analysis on the problem of metagenomics. The author has exploited the fact that variation of abundance of the species across different samples directly affects covariance of the read-depth at every k-mer in that species' genome. A pooled analysis of multiple samples can capture these patterns. Inspired from this approach, we've used LSA for clustering RNA-seq reads. The motivation behind clustering RNA-seq reads clustering comes from de-novo assembly. One common method to perform de-novo assembly is by making a De Bruijn graph of all the reads and finding an eulerian walk to get the genome. However, this is very tricky when we have millions of reads and it becomes computationally intractable to build De Bruijn graph of all the reads. What we hope to have is clustering of closely related reads (possibly from same gene), so that we can build separate De Bruijn graphs on these clusters (which are magnitudes smaller than overall count of reads).

Metagenomics paper has used these steps to solve the problem:

1. Using hyperplane hashing build k-mer abundance matrix for reads across each sample.
2. Perform streaming SVD on the abundance matrix.
3. k-mer clustering and read partitioning.

Our approach for the problem is to use the same pipeline for RNA-seq clustering. As a first step of the pipeline we have to construct the k-mer abundance matrix. In our abundance matrix we have used k-mer hashing on the basis of per read instead of per sample and performed hyperplane hashing on the k-mers to map them to appropriate bins of the matrix.

As the second step they were using SVD to perform dimensionality reduction. Since, they are working on the metagenomics problem they have to reduce the dimension of the features, but in our case we feel there is no point in using SVD because we are working in computationally tractable dimensions. On the other hand, in the LSA paper, they have used variation in the k-mer across samples to really make it a basis of clustering reads, but we don't think there is any point of using this information in case of RNA-seq clustering since this variation of k-mer across different reads really doesn't give information required for this analysis.

In the paper, the authors propose that their novel hyperplane hashing method can be used for de-novo transcriptome pre-processing. First of all, we feel this method of hyperplane hashing is not novel and secondly the authors' implementation is of low quality. When using our own read sets, sometimes their code is even giving zero clusters which is very sad. So, we have implemented our own version of hyperplane hashing and used it to tweak the results, but not able to get a lot out of it. The main reason we think is that we need to play around with the variables in this method, especially the hash size and k-mer size. In the LSA paper they have used around 10GB of data and they were working with around 2^{29} as hash size, So to be in the same magnitude we have also used around 2^{10} as hash size for around 1GB of data. Also, we never moved above 2M reads since the problem becomes computationally expensive if we take more than this number of reads with both implementations.

We've observed that hash size affects the size and accuracy of the cluster, which we get. In addition, we have also observed a lot of false negatives because of the way of clustering is done.

II. Parsing Reads:

We have used standard way of generating k-mer out of reads. We didn't use Jellyfish because it returns the overall k-mer count of all the reads. For our analysis we want to maintain the read to k-mer relation but Jellyfish doesn't return the k-mer count per read. We could have run Jellyfish for each read separately but we think it would be more expensive to run Jellyfish on each read then parsing the read and maintaining the read k-mer relation by ourselves. Jellyfish is also beneficial for streaming reads and our project did not fit this use case. Thus, we felt it was best to write our own tool for reading the pair end reads. Our implementation follows the same pattern of Jellyfish k-mer-counting with the -C canonical option.

III. Locality-Sensitive Hashing (LSH):

This technique, first introduced by P. Indyk and R. Motwani[1], aims at solving the nearest neighbor problem by providing an approximate solution in an efficient way via hashing.

Given the scale of dealing with large amounts of data when it comes to transcriptomics, it is important to have an efficient way to process millions/billions of reads for a variety of purposes. In our problem, we need to cluster RNA reads according to a specified measure of similarity, that can be defined in several ways such as cosine similarity, Jaccard similarity, etc. A naive approach to find the closest pair in a data set, given a distance or similarity function, runs in quadratic time, which is too expensive for the scale of genomics.

The main idea behind LSH is that if two points are actually close, then after a *projection* operation they will remain close. If points are embedded in an Euclidean space, we can generate a random collection of *n* planes that will partition our space into 2^n *bins.* Points that are close are expected to be in the same bin. Thus, we have defined a *hash* mapping from the set of points to the set of bins. This is called *hyperplane hashing*. A similar approach would be to define a random collection of *primitive projections* (a primitive projection on the *ith* coordinate of a *d*-dimensional space returns the ith coordinate of a *d*-dimensional vector), and define a hash as the concatenation of such primitive projections. Points that are very close are expected to share almost the same values in their coordinates, so after hashing, close points are expected to be in the same bin. Thus, after applying LSH we obtain a collection of *clusters* of similar points, the good news, in linear time. In our project, we explored the two mentioned LSH approaches, as pre-processing for the true clustering algorithms.

A. Reimplementation of LSA Paper's Approach

We reimplemented the LSH [2] approach. Please refer to the "Results" section to view the performance. We used our results utility, which measures the accuracy metrics of the clusters that were produced. Please refer to the section titled "Accuracy Measure" to learn more about how we measure the accuracy of the clusters.

Our implementation of LSH was done in Python. The inputs to our LSH implementation are hash-size (h), k-mer-size (k) and two pair end read files. The first stage of our implementation parses the pair ends read files. Next, h random k-mers of length k are generated. Take each of these random k-mers and build an h by k matrix (see Figure 1). In addition, we also represent the bases as the following tuples so we can evaluate weightage:

$$
\begin{aligned}
A &\rightarrow (1,0)\\
T &\rightarrow (-1,0)\\
C &\rightarrow (0,1)\\
G &\rightarrow (0,-1)
\end{aligned}
$$

We take each k-mer which has dimensions 1 by k, and take a dot-product with the random k-mer matrix. This produces an bin index where we can place a one if that k-mer is present in the read. The next stage of the

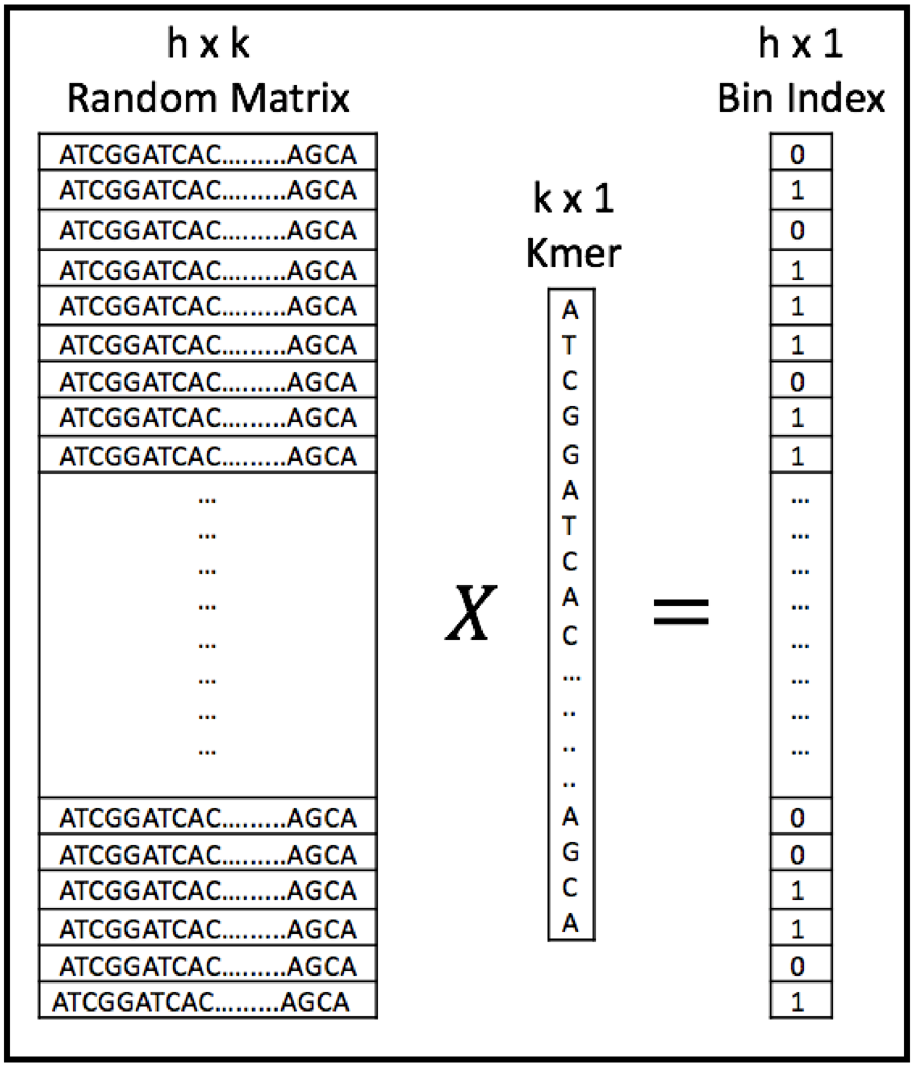
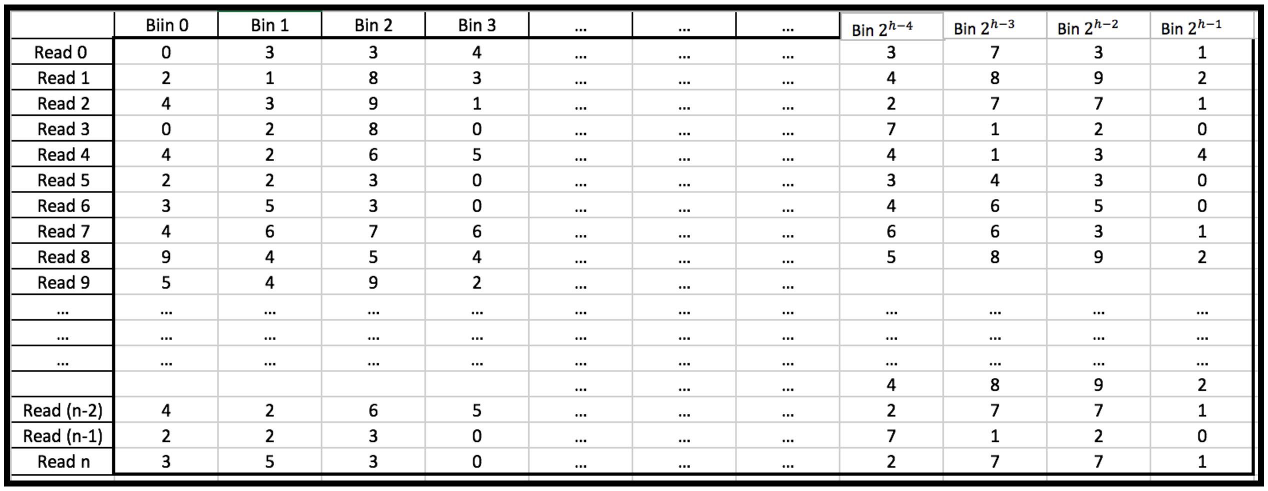


Figure 1: Random K-mer Matrix times k-mer index produces the Bin index

algorithm builds an abundance matrix which is a reads by 2^h columns. See Figure 2 below for a graphical representation of the abundance matrix.

From this stage we have the k-mer vector for each read and for each k-mer we can find where to place a one in the abundance matrix. We basically represent the read as a vector of numbers which represent the number of times a k-mer is present. This sequence of zeros and ones tells us which cluster to place the read in.



	Biin 0	Bin 1	Bin 2	Bin 3	...	...	...	Bin 2^{h-4}	Bin 2^{h-3}	Bin 2^{h-2}	Bin 2^{h-1}
Read 0	0	3	3	4	...	...	...	3	7	3	1
Read 1	2	1	8	3	...	...	...	4	8	9	2
Read 2	4	3	9	1	...	...	...	2	7	7	1
Read 3	0	2	8	0	...	...	...	7	1	2	0
Read 4	4	2	6	5	...	...	...	4	1	3	4
Read 5	2	2	3	0	...	...	...	3	4	3	0
Read 6	3	5	3	0	...	...	...	4	6	5	0
Read 7	4	6	7	6	...	...	...	6	6	3	1
Read 8	9	4	5	4	...	...	...	5	8	9	2
Read 9	5	4	9	2	...	...	...				
...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...
					...	...	...	4	8	9	2
Read (n-2)	4	2	6	5	...	...	...	2	7	7	1
Read (n-1)	2	2	3	0	...	...	...	7	1	2	0
Read n	3	5	3	0	...	...	...	2	7	7	1

Figure 2: Abundance Matrix (Vector Representation of Reads)

B. LSH in the Hamming Cube

A second LSH approach is the following [1]. Let us consider an embedding in the Hamming space by transforming each read into a binary vector, thus requiring a binary representation for each base. We define a *unary mapping f* as:

$$f(A) = 1000$$
$$f(C) = 0100$$
$$f(G) = 0010$$
$$f(T) = 0001$$

Then, an embedding from the set of reads of length *m* to the Hamming cube of dimension *4m* is as follows:

$$F(ACGT\ldots) = f(A)\,f(C)\,f(G)\,f(T)\ldots$$

The right-hand side is the concatenation of unary representations of each base. As an example:

$F(AACCTG) = 100010000100010000010010$

Now, let's define a set of indices *S* by sampling uniformly *p* coordinate positions from {1, 2, 3,...,4*m*}. We define a LSH hash *H* as the concatenation of *p* primitive projections whose coordinate positions are those in S, that is:

$$H := h_{i_1}(x)\,h_{i_2}(x)\ldots h_{i_p}(x)$$

where *x* is the read in binary representation to be hashed. As an example, if

$$S = \{1, 5, 9, 13, 17, 21\}$$

then

$$H := h_1 h_5 h_9 h_{13} h_{17} h_{21}$$

so if we apply *H* to our previous example of a read in binary representation then

$$H(AACCTG) = 110000$$

Now, If we apply the same hashing to a similar read, say *AACCTT*, we obtain

$$H(AACCTT) = 110000$$

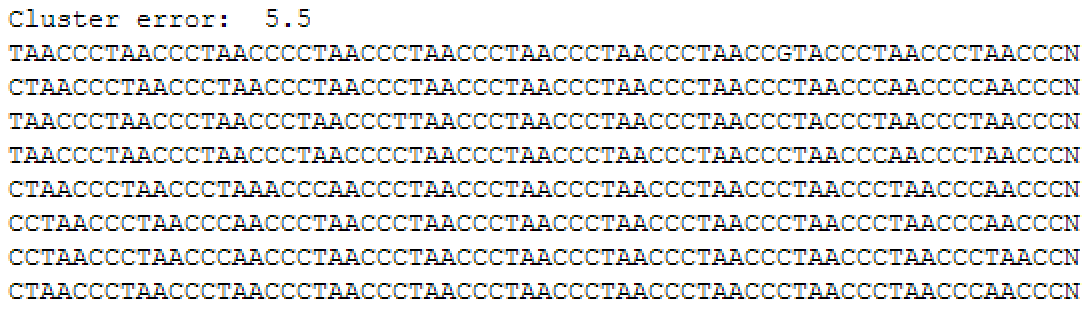
and therefore both reads would be hashed to the same bin “110000”. The more bases two reads share position-wise, the higher the probability they will be hashed to the same bin.

IV. Three Stage Clustering (TSC)

Another approach to tackle our clustering problem is to combine some standard clustering techniques along with LSH in a three stage process, namely

1. LSH in the Hamming cube
2. K-means Clustering
3. Hierarchical Clustering

In the first stage we follow the ideas described in section III.B, so after parsing and hashing reads we obtain a list of clusters, the bins of *H.* Let us see an example below.



```
Cluster error:  5.5
TAACCCTAACCCTAACCCCTAACCCTAACCCTAACCCTAACCCTAACCGTACCCTAACCCTAACCCN
CTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCAACCCCAACCCN
TAACCCTAACCCTAACCCTAACCCTTAACCCTAACCCTAACCCTAACCCTACCCTAACCCTAACCCN
TAACCCTAACCCTAACCCTAACCCCTAACCCTAACCCTAACCCTAACCCTAACCCAACCCTAACCCN
CTAACCCTAACCCTAAACCCAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCAACCCN
CCTAACCCTAACCCAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCAACCCN
CCTAACCCTAACCCAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCN
CTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCAACCCN
```

Figure 5: Reads clustered in a bin after LSH

After applying LSH there might be many bins with a single read, this is due to the fact that our hash only clusters reads whose bases match position-wise, so we need a more flexible approach for further clustering. To accomplish this we define another embedding that aims at encoding more read *features*, the *codon abundance* embedding. Given there are 64 different codons, the embedding is defined as

$$E : \{\text{set of reads}\} \rightarrow \mathbb{R}^{64}$$
$$E(x) = (c_1, c_2, \ldots, c_{64})$$

so each read is mapped to a 64-dimensional vector that represents the frequencies of codons in each read. With this representation, we can use the manhattan/euclidean distance to measure similarity, the smaller the distance, the more similar the reads. Also, we can compute the *centroids* of clusters as the *average* of their codon abundance vectors, which allows to quantify the notion of cluster quality in a precise way, by defining the cluster error as the standard deviation of the distance of every read in the cluster from its centroid. Now we can proceed with K-means clustering by selecting the largest k clusters' centroids, obtained during the LSH stage, as the initial centroids. But how do we define k? Because there is no way to know in advance the *true* number of clusters, we can empirically select values for k that prove to work by trial and error for a given data-set, however, this value will be dependant on the size and quality of the data. One way to address this issue is by selecting k as follows, during LSH we keep track of the size of bins in a histogram (we disregard bins of size too small and bins that are unexpectedly big), we average the histogram values and set k to be the number of bins whose size is greater than the average size. Although this might seem rather an arbitrary choice, it is an attempt to *learn* from data an approximate number of clusters. Let us consider the following example from a data-set of 100, 000 reads.

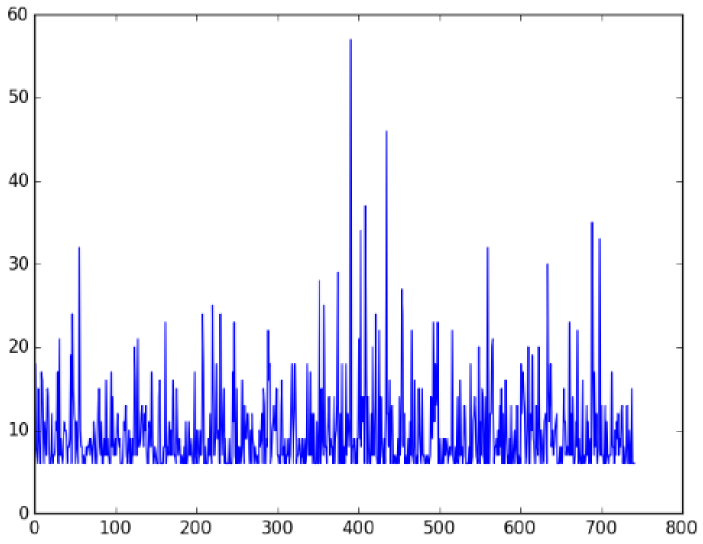


Figure 3: bin size histogram of a data set of 100,000 reads.

In Figure 3, the vertical axis represents the size of bin, the horizontal axis represents the bin's order of appearance. We set a minimum bucket size of 5. The bin average size is 9.68, the weighted average error is 0.51, the total number of bins is 68, 916 and the number of bins whose size is above 9.68 is k = 253. After applying K-means clustering we obtain 253 clusters with an increased weighted average error of 48.08. In this particular example we only ran one loop of K-means, if more loops are performed the weighted average error is expected to decrease, so the quality of clusters increases. But what if we could cluster even further without increasing significantly the current weighted average error? The answer is to continue with hierarchical clustering, and setting an error threshold as stopping criteria, that is, clustering stops if there are no clusters to merge without surpassing the error threshold.

In our example, after applying hierarchical clustering, the number of clusters was reduced from 253 to 158 with a slightly higher weighted average error of 48.99, this is the third and last stage.

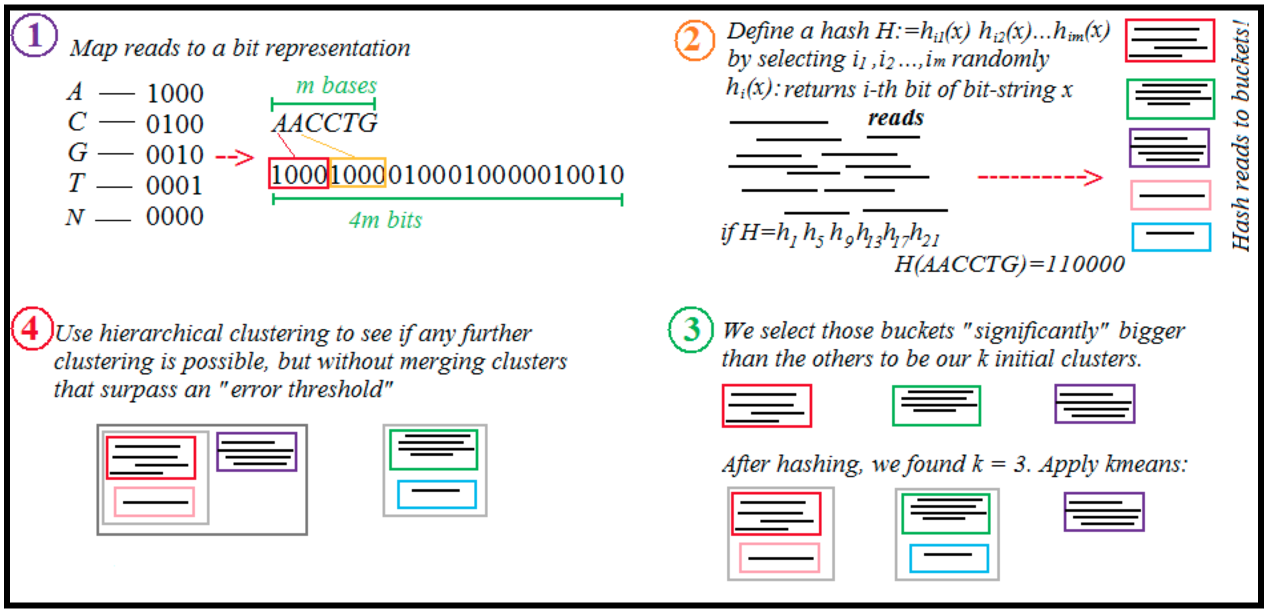

1
Map reads to a bit representation
A — 1000
C — 0100
G — 0010
T — 0001
N — 0000
m bases
AACCTG
1000100001000010000010010
4m bits
2
Define a hash H:=h_i1(x) h_i2(x)...h_im(x)
by selecting i1, i2..., im randomly
h_i(x): returns i-th bit of bit-string x
reads
if H=h1 h5 h9 h13 h17 h21
H(AACCTG)=110000
Hash reads to buckets!
3
We select those buckets "significantly" bigger than the others to be our k initial clusters.
After hashing, we found k = 3. Apply kmeans:
4
Use hierarchical clustering to see if any further clustering is possible, but without merging clusters that surpass an "error threshold"

Figure 4: TSC (Three Stage Clustering)

The motivation behind TSC is to find an efficient way to produce clusters of reads, without incurring in expensive storage and running time, but this comes with a price. During testing, the algorithm did not perform as well as desired, perhaps due to the need of more information to compare reads, that the codon abundance vector is unable to capture, or the need of a better embedding or clustering technique. We could say, at least, that this is an example of the typical tradeoff between speed and accuracy when dealing with large genomic data sets.

V. Accuracy Measure

We have used pair-wise comparison of reads as an accuracy measure. Since the reads are simulated we know which transcripts they came from, to avoid the alternative splicing events we have used a transcript to gene map and have done gene level clustering. A toy example of the measure will look like as follows:

$$\begin{aligned} R &= [r_1, r_2, r_3, r_4] \\ C &= [(r_1, r_2), (r_3, r_4)] \\ T &= [(r_1, r_2, r_3), (r_4)] \end{aligned}$$

here R is the set of reads, C is the cluster reported by method (each tuple represent one cluster) and T represent true cluster for reads.

$$\begin{aligned} TP &= \quad (r_1, r_2) \\ FP &= \quad (r_1, r_3), (r_2, r_3) \\ FN &= \quad (r_3, r_4) \\ TN &= \quad (r_1, r_4), (r_2, r_4) \end{aligned}$$

where TP, FP, FN, TN represent True positive, False positive, False negative, True negative.

VI. Results and Conclusion

We've observed that running the LSA paper's code with our datasets gives 0 clusters (regardless of the hash size). This is a problem of their implementation.

We've done testing on two datasets for reads. First dataset is with 5k reads from 1 gene, second dataset has 10k reads from 6 genes. In our testing we are working with k-mer size of 21 and hash size of 2^{10}. With 5k reads we get high precision but almost no recall:

```
tp : 29254, fp : 567, tn : 2942526, fn : 9525153
prec: 0.9809865531, recall: 0.0030618331415, f1: 0.00610461270329
```

Since we have only 1 gene, all reads must go into 1 cluster but we see a lot of clusters making our recall almost 0.

So, we've tried with more genes but even this doesn't solve the problem. Instead, it has made it worse:

```
tp : 1683, fp : 7300, tn : 45433872, fn : 4552145
prec: 0.187353890682, recall: 0.000369579176025, f1: 0.000737703139578
```

We also tried reducing the hash size to 2^6 but it didn't made much difference.

```
tp : 2070, fp : 9684, tn : 45431488, fn : 4551758
prec: 0.176110260337, recall: 0.000454562622919, f1: 0.000906784720984
```

We think real problem was that we were being too strict in comparing two keys so we tried relaxing the problem by taking sum of the bin id as the clustering measure, but it didn't worked out either.

```
tp : 11650, fp : 100439, tn : 45340733, fn : 4542178
prec: 0.103935265726, recall: 0.0025582872256, f1: 0.00499365933856
```

The real bottleneck is the number of clusters which is almost 20-50% of the total reads.

We see increase in recall by greatly reducing the size of the bins(2^4), but that is really not very intelligent, because we never will know that in the real word problem.

```
tp : 2667832, fp : 27345647, tn : 18095525, fn : 1885996
prec: 0.0888877960466, recall: 0.585843821945, f1: 0.154355790574
```

In our view using this approach for RNA-seq experiment is not very fruitful thing to do. Coding done by authors[2] is really not appreciable. The programs sometimes go into infinite loops (one such bug was fixed). They claim that running their code in a distributed environment greatly improves performance and scalability, however these claims could not be investigated. Another project could explore reimplementing their clustering pipeline for the metagenomics problem.

VII. Code

The code for our implementations:
https://github.com/keyavi/LRA.
The code for the LSA paper is in a forked repo (their code is not of high quality): https://github.com/keyavi/LatentStrainAnalysis
The gitter discussion forum for the project:
https://gitter.im/keyavi/LRA

Reference:

[1] “Approximate Nearest Neighbor. Towards Removing the Curse of Dimensionality.” (Proceedings of the 30th Symposium on Theory of Computing, 1998, pp. 604-613)
[2] “Detection of low-abundance bacterial strains in metagenomic datasets by eigengenome partitioning” by Brian Cleary, Ilana Lauren Brito, Katherine Huang, Dirk Gevers, Terrance Shea, Sarah Young and Eric J Alm