DVD-HIT

EN.600.439 Computational Genomics
Richard Chen, Phani Gaddipati, Candace Liu, Andrew Tsai
December 12, 2015

Abstract

Cluster analysis is a prevalent problem in unsupervised learning and data mining. In the context of computational genomics, cluster analysis can be used to group homologous sequences in families, taxonomical classification, metagenomics, and to group reads for genome assembly. CD-HIT is a widely used program to cluster protein and nucleotide sequences. It uses short word filtering to determine sequence similarity, which is much faster than sequence alignment. For this project, we attempt to improve on the algorithm used in CD-HIT by using other string similarity metrics – namely: spaced short word filtering, Jaccard index, cosine similarity, and length of the longest common subsequence (LLCS). By testing our algorithm on 16S ribosomal RNA data, we found that although all the string similarity metrics we used were $O(n)$ runtime on average (except for LLCS), the short word filter à la CD-HIT was still the fastest; however, we did find that other string similarity metrics were better able to cluster the rRNA data into their proper genuses.

Introduction

In recent years, improvements in sequencing technology such as the advent of next generation sequencing have significantly increased the amount of biological sequencing data produced, but have also posed significant challenges in analyzing this data. Clustering is one approach to process this large amount of data. Data clustering is an analytic tool that aims to identify groups in huge datasets, in which each group contains objects that have a similarity score above a certain threshold. Clustering has been used in cancer class discovery and protein structure prediction, and has recently been applied to sequence similarity searching.

Sequence similarity searching is the process by which homologous proteins or genes are identified by comparing similarity with an excess similarity parameter - a statistically significant value that implies that they share a common ancestor. We can infer homology between two protein or genomic sequences when they are more similar than would be expected by random chance. The concept of using clustering to infer homology can be used to compare the results from different clustering programs. For example, sequencing data from 16S ribosomal RNA, which is often used to reconstruct phylogenies because of its slow rate of evolution, can be fed to different clustering algorithms and the results can be analyzed to determine how well the clusters correspond to known taxonomic categorizations.

Because the size of sequencing databases is growing rapidly, there is a need for more time and space-efficient programs to manage and analyze this data. The time complexity of many sequence analyses are $O(n^2)$, where n is the number of sequences. Clustering algorithms reduce the running time of sequencing

analysis programs based on the observation that many sequences are similar or nearly identical to known sequences. Therefore, a more efficient solution would be to group sequences into clusters based off a representative sequence of each group.

Here, we present implementations of various algorithms used in sequence clustering analysis, including short word filtering, spaced short word filtering, length of the common longest subsequence, Jaccard similarity, and cosine similarity, and compare the performance of each of these approaches.

Prior Work

CD-HIT

CD-HIT is one of the most widely used clustering tools for protein and nucleotide sequencing data developed at the Burnham Institute (now Sanford-Burnham Medical Research Institute). CD-HIT uses a greedy incremental clustering algorithm based on short word filtering. Short word filtering is used to reduce the number of pairwise alignments performed, which is the most time consuming step of the algorithm. In the CD-HIT algorithm, the set of sequences is first sorted in decreasing length and the longest sequence is taken as the representative sequence for the first cluster. The next sequence is then compared to the representative. If the similarity is above a given threshold, the sequence is added to that cluster. If under the threshold, the sequence becomes the representative for the new cluster. This is process is continued where each sequence is compared with the set of representatives and either added to an existing cluster or taken as the representative of a new cluster.

An important distinction between CD-HIT and other sequencing algorithms is that it does not use dynamic programming, which is $O(mn)$, where m and n are the lengths of the two sequences. Instead, the minimum number of identical k-mers is taken as a function of the similarity between the two sequences; therefore, the similarity of two sequences can be determined by counting without performing a sequence alignment. An example of the short word filter is shown in figure 1 for two sequences - sequence A and sequence B.

```
position:  01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
sequenceA: A  G  T  C  A  A  A  T  G  G  C  A  T  A  G  G  A  T  A  T
           |  |  x  |  |  |  |  |  |  x  |  |  |  |  |  |  |  |  |  |
sequenceB: A  G  A  C  A  A  A  T  G  A  C  A  T  A  G  G  A  T  A  T
2-mer      1        2  3  4  5  6        7  8  9  10 11 12 13 14 15
3-mer               1  2  3  4           5  6  7  8  9  10 11 12
4-mer               1  2  3              4  5  6  7  8  9  10
5-mer               1  2                 3  4  5  6  7  8
```

Figure 1: Visualization of how short word filtering works. Source: http://blog.nextgenetics.net/

The two sequences have 2 mismatches over the 20 base pair window, corresponding to a 90% identity. To be above a certain identity threshold, the number of necessary matching k-mers can be found using the following equation:

$$L - K + 1 - (1 - p) \cdot K \cdot L$$

where L is the window length, K is the size of the k-mer, and p is the threshold. Therefore, for a 20 base-pair window, there must be 15 matching 2-mers for a 90% identity score. In this way, similar sequences can be identified based on the matches of short subsequences.

LCS-HIT

Namiki et al. presents a DNA sequence clustering method based on CD-HIT called LCS-HIT, that uses an algorithm based on the length of the longest common subsequence. LCS-HIT utilizes a three-tier filtering algorithm to decrease computation time. One of the major differences between LCS-HIT and CD-HIT is the addition of the longest common subsequence (LCS) filter before alignment. It is advantageous because it is stricter than short word filtering, which can be too approximate and includes dissimilar sequence pairs in the same cluster.

Figure 2 shows the algorithm used by LCS-HIT. Each sequence in the original dataset is first compared with the set of cluster representatives using short word filtering to obtain a subset of the representatives that have a similarity above a certain threshold. For each of the cluster representatives in this subset, the length of the longest common subsequence (LLCS) is found between the sequence to be compared and the representative. If the LLCS is above a threshold, the sequence may be a part of the representatives cluster. In this way, the set of representatives is narrowed even further. For the sequence and each representative in this smaller set of representatives, the optimal sequence alignment is computed using affine gap penalties and the sequence identity between the sequence and the representative. Above a threshold, the sequence is added to the representatives cluster. If the sequence is not similar enough to any existing representative, it becomes the representative for a new cluster. Finally, optimal alignment is computed to determine whether a sequence belongs to a particular cluster.

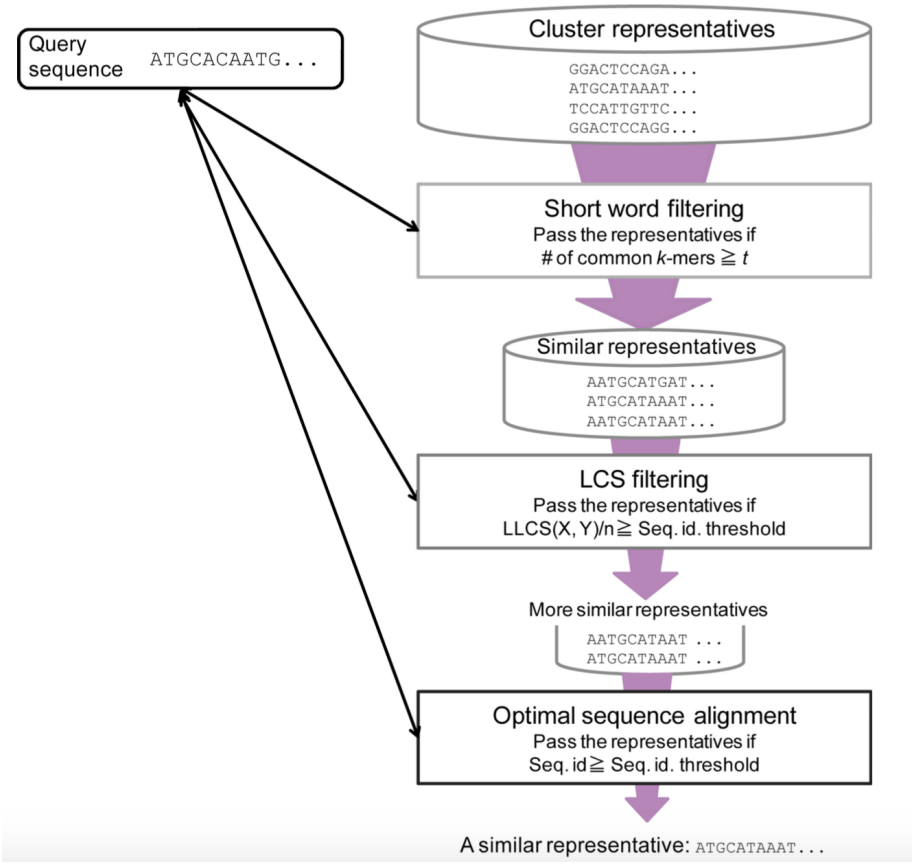


Figure 2: Greedy algorithm used in LCS-HIT. Source: Namiki *et al.*, Acceleration of sequence clustering using longest common subsequence filtering.

Although the time complexity of optimal alignment to determine whether or not a sequence belongs to a cluster is $O(mn)$, where m and n are lengths of the sequences A and B, the algorithm described above significantly reduces the number of alignments necessary by reducing the size of the cluster representatives. LCS-HIT claims to cluster 100, 150, and 400 bases 7.1, 4.4, and 2.2 times faster than CD-HIT, respectively.

The LLCS can be determined using dynamic programming. The dynamic programming algorithm for determining LCCS is very similar to the algorithm for global alignment. Let $X_i = (x_1, x_2, , x_i)$ and $Y_i = (y_1, y_2, , y_i)$, where $0 < i < m$ and $0 < j < n$. The LLCS between X and Y can be found as follows:

$$LLCS(X_i, Y_j) = \begin{cases} 0 & i = 0 \ or \ j = 0 \\ LLCS(X_{i-1}, Y_{j-1}) + 1 & x_i = y_j \\ max(LLCS(X_i, Y_{j-1}), LLCS(X_{i-1}, Y_j)) & x_i \neq y_j \end{cases}$$

The sequence identity is defined as the number of matches between two sequences that maximizes the alignment score divided by the length of the shorter sequence n. Therefore, we can write:

$$\frac{LLCS(X, Y)}{n} \leq \text{Sequence Identity}$$

Therefore, using the LLCS algorithm, we can check whether $LLCS(X,Y)/n$ is greater than a certain threshold before computing sequence identity.

The dynamic programing approach described above to find the LLCS is in $O(mn)$ time, which is very inefficient. However, there are more efficient methods for finding the LLCS such as bit-parallel computation algorithms. The bit-parallel algorithm for fast calculations of LLCS was first implemented by Allison & Dix et al., and later on by Crochemore et al., both of which calculated the LLCS in $O(mn/\omega)$ time, where ω is the length of a machine word. Their algorithms were approximately linear for strings that were shorter than the length of a machine word, but not suitable for DNA sequence clustering where reads can be 100 - 400 bp long. Hyrro et al. later on improved on the Crochemore algorithm that calculated the LLCS in $O([d/w]\, n)$, where d is the indel distance calculation between two strings.

Methods and Software

The goal of our project was to implement a variety of different string similarity metrics, in addition to short word and LLCS filtering, that would have potential to improve sequence clustering accuracy and runtime. The additional string similarity metrics we used were Jaccard index, cosine similarity, and spaced short word.

Clustering Algorithm

Algorithm 1 Greedy Clustering Algorithm

```
1:  Q ← [reads from .fa file]
2:  Sort Q by length, descending.
3:  R ← ()                                          ▷ To hold representative reads
4:  Map<r,()> ← {}                                  ▷ Mapping from representative read to a cluster
5:  Push the longest read from Q into R
6:  for q in Q do
7:      for r in R do
8:          if similar(q,r) == True: then
9:              Add q to r's cluster
10:             Break
11:     if q was not added to any cluster then
12:         Add q to R.                             ▷ Make q a representative read
13:         Put (q, ()) in Map
```

Greedy: Our clustering algorithm is a greedy algorithm that compares reads q with a single "representative" read r in each cluster, and if the similarity score was not greater than a threshold, it makes q a representative read and gives it its own cluster. Representative reads are chosen in the order that they appear in Q after the reads are sorted by length, with the first representative read being the longest read. Our program also accommodates applying multiple filters in a hierarchical manner, where clusters made by previous iterations of the algorithm are further clustered. This algorithm was first prototyped in Python, and then ported to Java.

K-Means: In addition to the greedy clustering algorithm, we also spent time implementing K-Means in Python to see if a more accurate clustering algorithm could be used. In Euclidean space, given k number of clusters, K-Means assigns k random coordinates from a list of coordinates as "means", and assigns each coordinate in the list to its closest "mean." K-Means then calculates new means for each cluster by averaging its coordinates, and iteratively repeats this process until the means converge. In clustering string sequences, one idea we considered was to use edit distance as a distance metric between strings, and the least common subsequence as the "average string" between strings. The idea is that within each cluster, the mean would converge to a degenerate subsequence that is only found within its respective cluster. In addition to being very slow, K-Means performed poorly, and assigned every read in our .fa file to one cluster. One of the means that was calculated with LCS after the first step was too degenerate, and thus was similar to every read in our file. A variation of K-Means called K-Mediods could have been used which clusters around mediods instead of means. In addition, we could have also implemented a stricter threshold and distance metric. Ultimately, time needed to be spent implementing different similarity filters, so this algorithm was not pursued further. Other algorithms such as Expectation Maximization and Affinity Propagation would have been experimented as well if there was more time.

Similarity Filters

In Java, the different similarity metrics implement an abstract interface that compares the similarity of two strings. Filters can be interchanged based on the program arguments.

Short Word Filtering: The short word filter that we implemented was described above in the CD-HIT section.

Spaced Short Word Filtering: We implemented a variant of the short word filter where instead of comparing the k-mers between the two sequences, we compared subsequences using every other nucleotide.

Jaccard Similarity: Jaccard similarity is a metric for comparing the similarity between two sets. It is defined as the size of the intersection divided by the size of the union of the two sets.

$$JacS(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In our implementation, in order to compare two strings A and B, we divided each string into a set of k-mers, where k is the length of the substring taken. Since the alphabet for DNA nucleotides is small, larger k's were taken to avoid completely capturing strings A and B in the intersection.

Cosine Similarity: Cosine similarity is a measure of similarity between two vectors in space. In Euclidean space, vectors that point in the same direction have a cosine similarity of 1, and thus, are perfectly similar (identical).

$$CosS(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \, \|\vec{b}\|}$$

In our implementation, we divided strings A and B into k-mers, and treated these strings as vectors holding the term frequency of each k-gram. Specifically, if there are N k-mers in the union between strings A and B, we can treat these strings as vectors in an NxN dimensional space, in which the difference in orientation is an estimate of similarity. Unlike Jaccard, frequency of each k-mer was kept track using a hashmap to account for the difference in magnitude between these vectors.

LLCS Similarity: To solve the Longest Common Subsequence Problem in Java, we implemented Hirschberg's algorithm, which improves on the time complexity of the dynamic programming solution. Instead of storing the entire dynamic programming matrix, we allocated enough memory for only one array, and recursively built the longest common subsequence in a string. The complexity of our implementation was $O(n)$ space, $O(nm)$ time.

A considerable amount of time was also spent trying to implement fast bitwise-parallel LLCS calculations to calculate LLCS in $O(n)$ time, however, the preliminary sections for Allison Dix et al., Crochemore et al., and Hyrro et al. were too dense for us to understand.

Cluster Analysis

To test our different clustering algorithms, we used the same set of 16s ribosomal RNA sequences as input data, which we obtained from the National Center for Biotechnology Information database. 16s ribsomal RNA is offten used to reconstruct phylogenies because of its slow rate of evolution.

For each filtering algorithm, we evaluated our results by recording the number of clusters in the output as well as run time. In the input data set, there were a total of 2425 unique genuses. Ideally there would be 2425 clusters in our output, each corresponding to 1 genus. Another way we compared our results was to evaluate how well a particular implementation clustered 16s ribosomal RNA sequences into known genuses. For each cluster, we first found the most frquent genus in that cluster. The number of sequences in a cluster from that particular genus was divided by the number of total sequences in that cluster, and this was averaged across all the clusters to find the clustering score for a certain filter.

The runtime analysis of each filtering algorithm was generated by randomly sampling 1,000-18,000 reads in increments of 1,000 from the full 16s rRNA sequence file. All the filters were then tested on each one of these randomly generated files.

Results

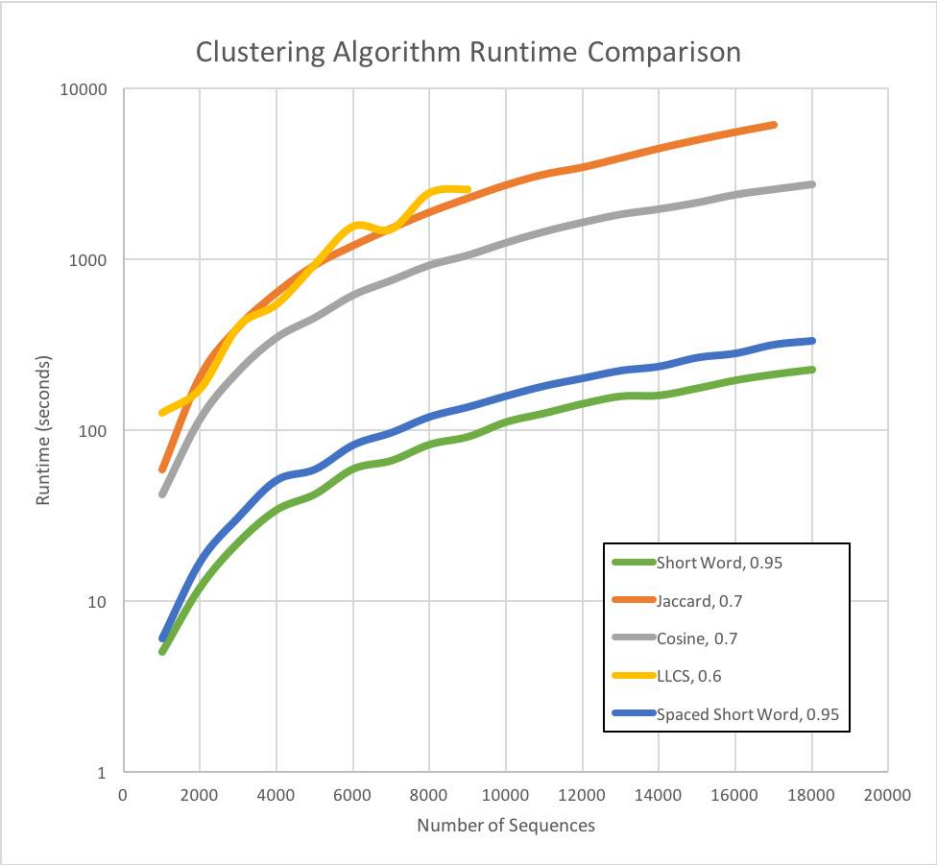


Figure 3: Log runtimes of various string similarity metric algorithms.

According to Figure 3, each filter (besides LLCS) had an approximately linear time complexity. In the clustering runtime of the different similarity metrics implemented, short word and spaced short sequence filtering were the fastest, clustering 18,000 sequences in 3 minutes 44 seconds and 5 minutes 35 seconds respectively. Cosine filtering was the slowest filter that clustered all 18,000 reads, with a runtime of 45 minutes 51 seconds. Jaccard filtering clustered 17,000 reads in 1 hour 42 minutes. Not enough data was collected for LLCS, as clustering runs with with greater than 10,000 took too long to run.

Since not enough data was collected for LLCS, we could not optimize the filter with a threshold that would give the best clustering accuracy. The reason that we used different threshold levels in comparing the runtimes is that larger thresholds for Jaccard, cosine, and LLCS took way too long to run and would not

have generated data in a timely manner.

To use the clustering score to compare our filters, we fed the entire 16S ribosomal RNA sequencing dataset to each filter. We first determined the k-mer length that gave the best results for each implementation. For each algorithm, we plotted the clustering score against the threshold for each k-mer length we tested, and determined which k-mer length resulted in the highest scores (Appendix III). We then plotted clustering score against threshold for each algorithm and best k-value, which is displayed below. Longest common subsequence was not included because it took too long to run with the entire dataset.

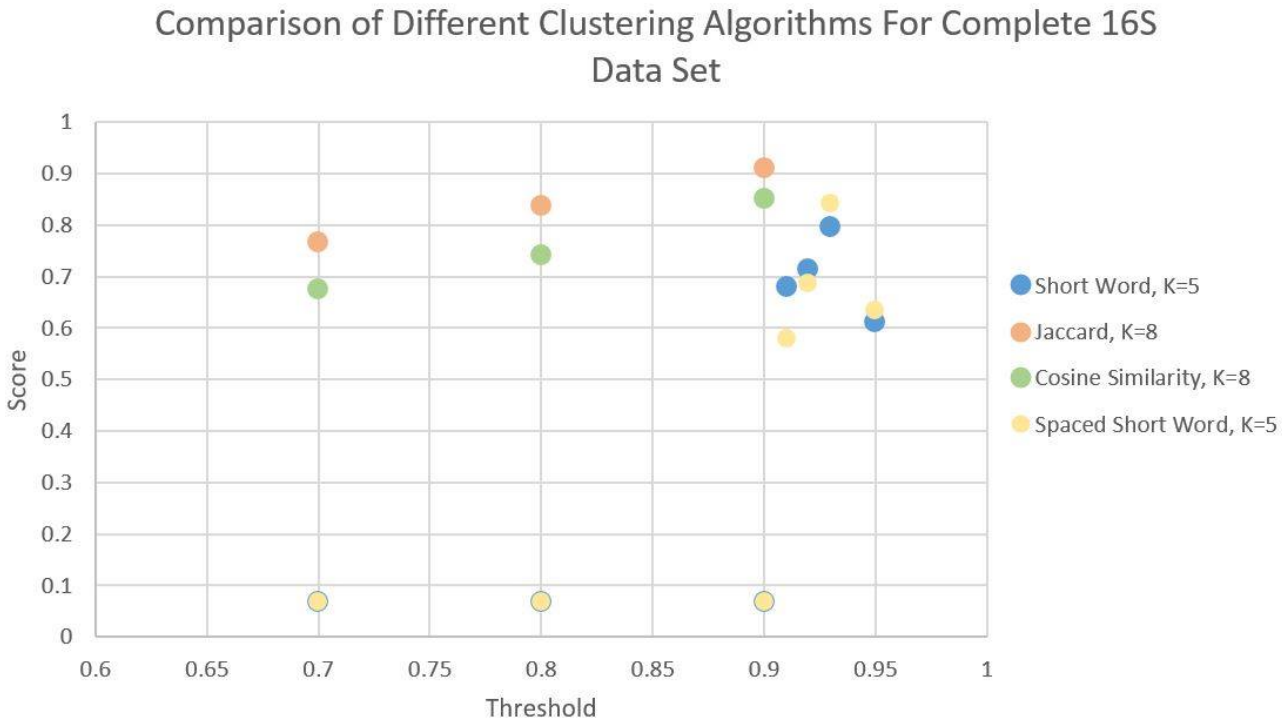


Figure 4: Comparison of cluster scores at different thresholds for various algorithms.

From the figure above, we can see that the Jaccard similarity filter resulted in the highest clustering score at a given threshold.

Conclusions

While Jaccard and cosine similarity have many uses in different fields, the results from our experiments show that they are unviable for sequence clustering in the implemented form. Both have a significant increase in runtime for very little, if any, improvement in clustering score. Of these 2 string similarity metrics, Jaccard was more promising, with some of the highest clustering scores. However, some if not all of this improvement may be attributed to a poor clustering scoring algorithm.

A potential source of improvement in the Jaccard filter is switching to a generalized Jaccard filter. In the current implementation, a k-mer is only counted as existing or not. In the generalized Jaccard index, the multiplicity of the k-mer is taken into account.

The effectiveness of the LLCS filter was not be able to be determined due to the very slow runtime. Implementing some previously demonstrated optimizations would allow for actual evaluation on the entire dataset. This remains as a possible avenue for improvement.

The spaced short word filter was very similar to the regular short word filter. There was a very small increase in clustering score but with a longer run-time. The increase in run-time is most likely a result of the increased number of clusters.

The spaced pattern used, *010101*, was arbitrarily chosen. When testing, the problem was treated as a string similarity problem with little thought on the genomic backing of the data. Alternative patterns may yield better results. For example, *001001*. The first two bases of a codon have much less variability than the last codon. Giving significance to the last base may yield a higher specificity, for the generated k-mers. However, this would be more dependent on the reading frame.

DVD-HIT was implemented with support for hierarchical clustering. However, due to the slow runtimes, we were not able to evaluate different combinations of filters. Additionally, the program does not yet support different k-mer lengths and thresholds for each level. Ideally, a lower threshold would be used for initial clustering, and be refined with a higher specificity filter.

Cluster Scoring Analysis

As mentioned above, part of the reason why our Jaccard and cosine filters have higher cluster scores is an imperfect cluster scoring algorithm. We found that there is a positive correlation between number of clusters created by DVD-HIT and the cluster score, and the Jaccard and cosine filters generate more clusters than the short word filters. Currently, our method for calculating the clustering score does not penalize the scenario in which multiple clusters have the same most frequent genus. If multiple clusters have the same most frequent genus, the clustering algorithm did not realize that those clusters were similar and thus failed to put them in the same cluster. In order to create a better genus clustering score metric, we should:

1. Award when a cluster has a large percentage of reads from the cluster's most frequent genus.
2. Penalize clusters that are more hetereogenous.
3. Penalize multiple clusters that have the same most frequent genus.

Literature Cited

1. Allison L. and Dix T.L. A bit-string longest common subsequence algorithm. Information Processing Letters, 23:305310, 1986

2. Bergroth L., Hakonen H., and Raita T.. A survey of longest common subsequence algorithms. In Proc. 7th International Symposium on String Processing and Information Retrieval (SPIRE00), pages 3948, 2000.

3. Crochemore M., Iliopoulos C.S., Pinzon Y.J., and Reid J.F. A fast and practical bitvector algorithm for the longest common subsequence problem. Information Processing Letters, 80:279285, 2001. Appeared also in proc. AWOCA 2000.

4. Do C.B., Batzoglou S. What is the expectation maximization algorithm?. Nat Biotechnol 2008, 26:897-899

5. Frey, B.J. Dueck, D. Clustering by passing messages between data points. Science 315, 972976 (2007)

6. Grzegorz K., N-gram similarity and distance, Proceedings of the 12th international conference on String Processing and Information Retrieval, November 02-04, 2005, Buenos Aires, Argentina

7. Hirschberg D.S. A linear space algorithm for computing maximal common subsequences. Information Processing Letters, 7(1):4041, 1978

8. Hirschberg D.S. An information theoretic lower bound for the longest common subsequence problem. Communications of the ACM, 18(6):341343, 1975.

9. Li W., Godzik A. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. Bioinformatics. 2006;22:16581659.

10. Li W., Jaroszewski L, Godzik A. Clustering of highly homologous sequences to reduce the size of large protein databases. Bioinformatics. 2001;17:2823. doi: 10.1093/bioinformatics/17.3.282.

11. Namiki Y., Ishida T., Akiyama Y. Acceleration of sequence clustering using longest common subsequence filtering. BMC Bioinformatics. 2013;14(Suppl 8):S7. doi:10.1186/1471-2105-14-S8-S7.

12. Huang A. Similarity Measures for Text Document Clustering. Department of Computer Science, The University of Waikato, Hamilton, New Zealand. April 2008.

13. Pearson W.R. An Introduction to Sequence Similarity (Homology) Searching.Current protocols in bioinformatics / editoral board, Andreas D Baxevanis, 2013; doi:10.1002/0471250953.bi0301s42.

14. Fred A (2002) Similarity measures and clustering of string patterns. In: Chen D, Cheng X (eds) Pattern recognition and string matching. Springer-Verlag, New York, pp 155194

Appendix I: Runtime Expiriment Data

# Sequences	Runtime (s)				
	Short Word k = 8, t = .95	Jaccard k = 8, t = .7	Cosine k = 8, t = .7	LLCS k = 8, t = .6	Spaced Short Word k = 8, t = .95
1000	5	59	42	126	6
2000	12	208	117	175	17
3000	22	404	222	405	31
4000	34	642	349	546	51
5000	42	929	457	934	59
6000	59	1208	618	1559	82
7000	66	1525	755	1502	97
8000	82	1895	925	2452	120
9000	91	2287	1059	2571	137
10000	111	2732	1253		159
11000	125	3154	1455		182
12000	142	3468	1647		202
13000	157	3932	1836		224
14000	159	4470	1972		237
15000	175	5014	2152		267
16000	195	5574	2394		283
17000	211	6145	2573		318
18000	225		2751		335

Appendix II: Filter Expiriment Data

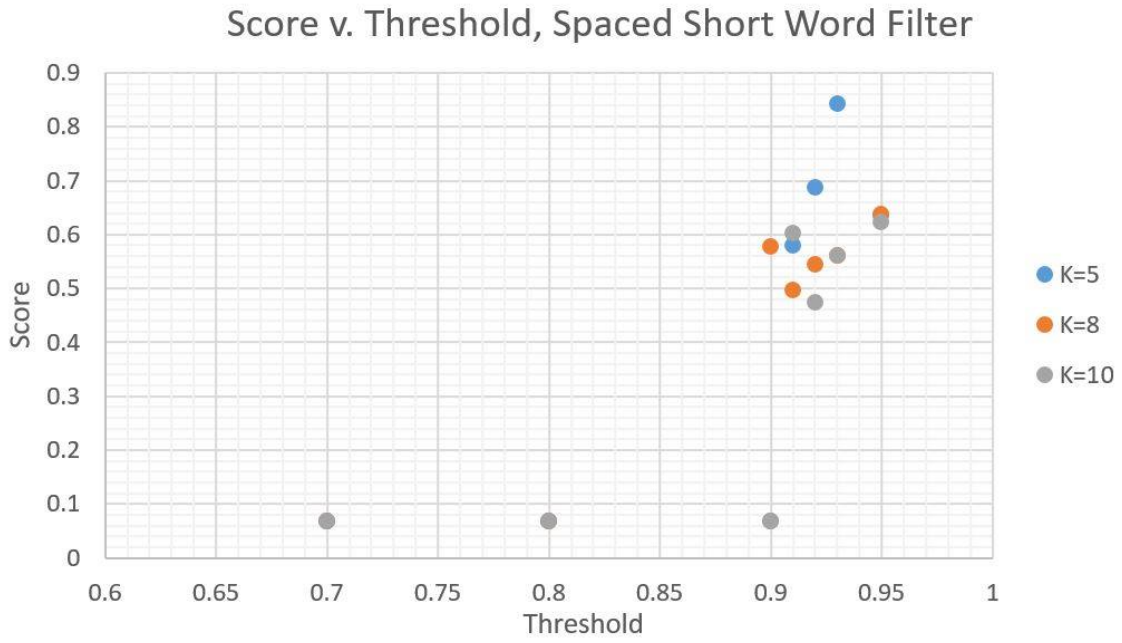
Short Word				
K	Threshold	Score	# Clusters	Run Time (s)
5	0.7	0.067	1	3
5	0.8	0.067	1	4
5	0.9	0.067	1	5
5	0.91	0.680	5	5
5	0.92	0.713	14	4
5	0.93	0.795	35	4
5	0.95	0.612	465	66
8	0.7	0.067	1	3
8	0.8	0.067	1	2
8	0.9	0.620	21	7
8	0.91	0.554	82	12
8	0.92	0.515	160	32
8	0.93	0.553	380	102
8	0.95	0.618	1214	485
10	0.7	0.067	1	2
10	0.8	0.067	1	2
10	0.9	0.067	1	2
10	0.91	0.520	15	4
10	0.92	0.489	62	9
10	0.93	0.521	187	33
10	0.95	0.590	946	321

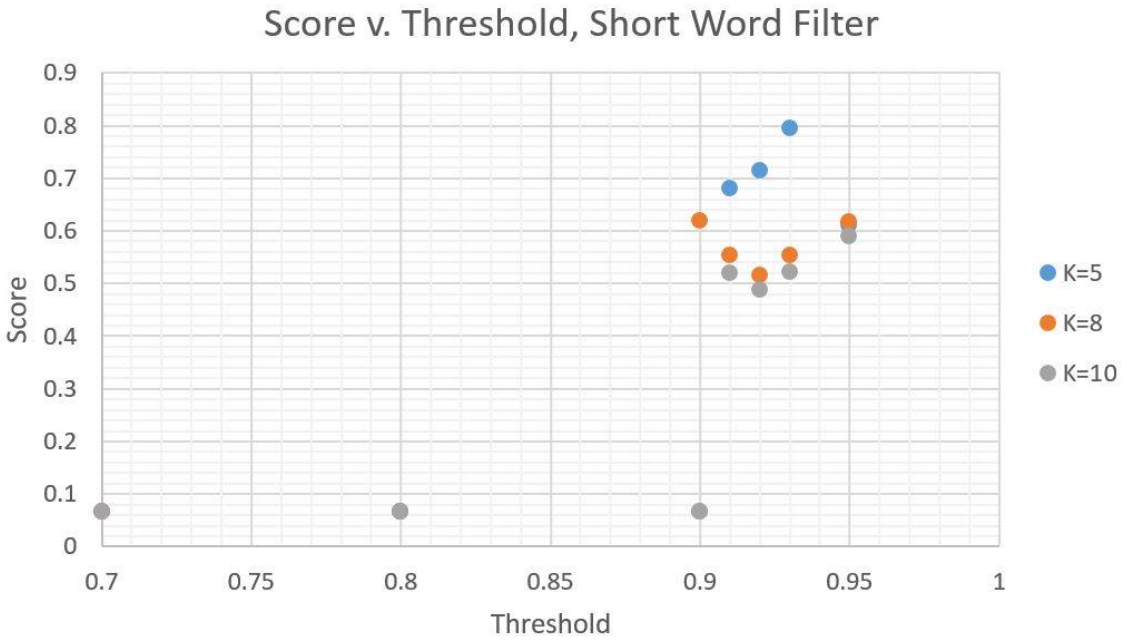
Jaccard				
K	Threshold	Score	# Clusters	Run Time (s)
5	0.7	0.738	553	142
5	0.8	0.700	2062	1752
5	0.9	0.800	6809	3919
8	0.7	0.766	5869	6600
8	0.8	0.839	8746	11745
8	0.9	0.912	12543	19177

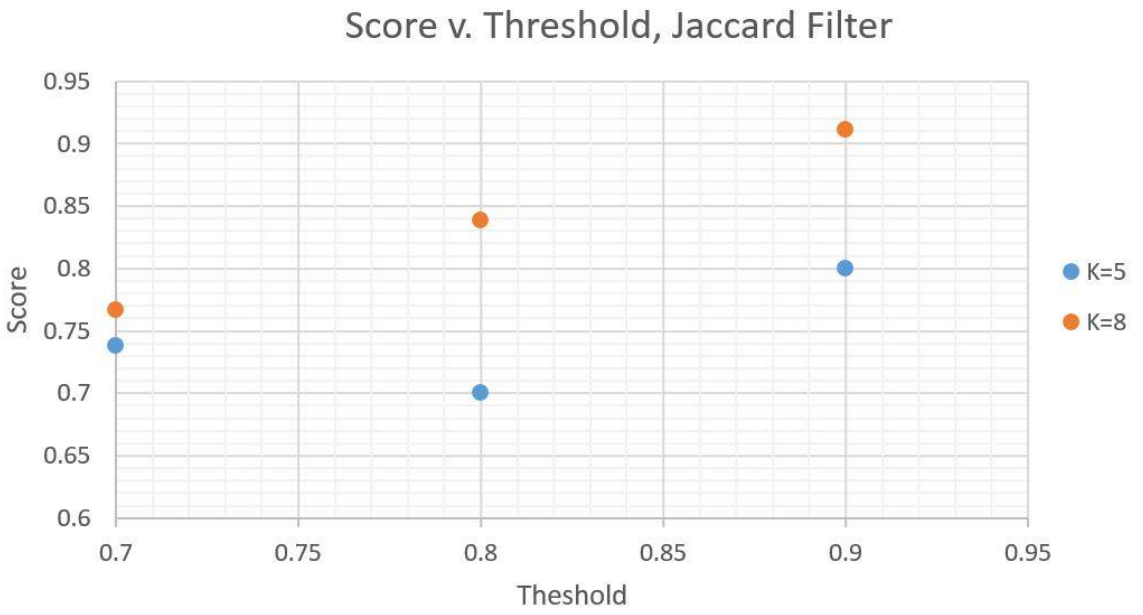
Spaced Short Word				
K	Threshold	Score	# Clusters	Run Time (s)
5	0.7	0.067	1	3
5	0.8	0.067	1	3
5	0.9	0.067	1	4
5	0.91	0.580	5	4
5	0.92	0.688	11	4
5	0.93	0.842	48	4
5	0.95	0.635	676	230
8	0.7	0.067	1	3
8	0.8	0.067	1	3
8	0.9	0.577	43	9
8	0.91	0.496	112	27
8	0.92	0.545	261	68
8	0.93	0.561	543	175
8	0.95	0.636	1607	853
10	0.7	0.067	1	3
10	0.8	0.067	1	3
10	0.9	0.067	1	2
10	0.91	0.601	28	5
10	0.92	0.472	97	16
10	0.93	0.561	290	57
10	0.95	0.622	1272	483

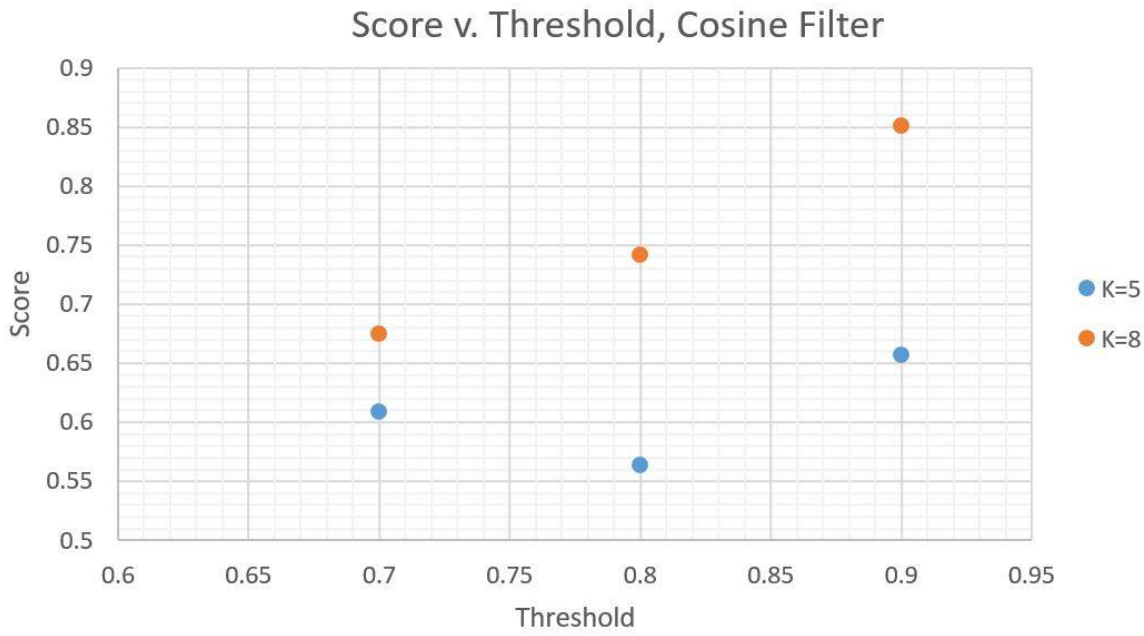
Cosine				
K	Threshold	Score	# Clusters	Run Time (s)
5	0.7	0.609	47	34
5	0.8	0.563	283	334
5	0.9	0.657	2071	5145
8	0.7	0.675	2617	2813
8	0.8	0.741	4942	6425
8	0.9	0.851	9309	15531

Appendix III: Comparing Score and Threshold for Each Algorithm






Score v. Threshold, Jaccard Filter
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
Score
0.7
0.75
0.8
0.85
0.9
0.95
Theshold
K=5
K=8


Score v. Threshold, Cosine Filter
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Score
0.6
0.65
0.7
0.75
0.8
0.85
0.9
0.95
Threshold
K=5
K=8