Final Project Report: SMASH

Nathan Roach, Patricia Walchessen, Charlie Wang, and Yue Zhang

December 8, 2017

Abstract

[1 paragraph] TO DO Final Project Abstract

1 Introduction

An intriguing and common problem that arises in computation is efficiently comparing the similiarities between high-dimensional data. The problem reaches to the far corners of computation, including with machine learning and kernels. In bioinformatics, the problem takes the form of long genomic sequences and the aim of determining the similarity between both close and disparate sequences. There are many purposes for solving the problem in bioinformatics, including evaluating evolutionary relationships, determining the organismal source of sequencing data, and comparing metagenomic data.

One class of algorithms that has been developed is Locality-Sensitive Hashing (LSH) algorithms. LSH aims to solve the problem by reducing data dimensionality and binning inputs in such a manner that similar data sets can be grouped together with high probability. There are two prevalent LSH algorithms in the wild: MinHash and SimHash.

MinHash computes the differences between two genomic sequences through

the estimation of a Jaccard Index in which a ratio is computed that evaluates the percent shared identity between the sets. SimHash, on the other hand, estimates the cosine similarity through the use of weighting of the frequency of bits and then compiling the signs from the summations.

2 Prior Work

[1 page] TO DO Some prior work include a paper published in 2016 about using MinHash to evaluate genomic sequences. In 2016, Ondov et al. used MinHash to analyze the resemblance between genomes and metagenomes[3]. After additional research, we saw no reference of using any other LSH to hash and compare genomic sequences, so we pursued a research project on the competitor, SimHash. The literature suggests that SimHash can be used for the same purpose as MinHash in analyzing resemblance.

3 Research Methods and Software

3.1 Exploration with MinHash

[1 page]. TO DO Patricia's description of what she has done with MinHash. Include the picture.

3.2 Exploration with SimHash

3.2.1 Dataset

The data that we tested on included 300+ E. coli genomes as well as multiple different yeast genomes. For the purpose of the discussion in this section, we will talk about the methods developed based on empirical testing on a dataset consisting of 1 E. coli genome and 2 yeast genomes named S. cerevisiae, S.

pombe. Additional discussion of the complete dataset will be presented in the Results section.

3.2.2 Method

To fully explore the application of SimHash in the genomics field, we commenced with surveying the existing SimHash literature and implementations. Next, we looked for existing SimHash libraries, many of which were designed for analyzing websites are texts of English words, and compiled the most reputable ones. From there, we selected two based on their reputation and fork numbers for us to base our code on and to adapt into the genomic field to explore SimHash more extensively. The first is Python-based (https://github.com/leonsim/simhash) and the second is Go-based (https://github.com/mfonda/simhash). Cite properly

3.2.3 Python-based SimHash

Upon surveying the existing code in the Python-based SimHash, we found that there were some hard-coded values in the library, such as k-mer or shingles sizes of 4. We thus took account of hard-coded values, and then created our own SimHash class based on the Python-based SimHash. The goal with our SimHash class is to enable it to fundamentally accept genomic data and build the shingles based on genomic data, not say English words. One thing to note when we explored the Python-based SimHash was that it relied on using MD5 to hash the keys, and then performed the operations of SimHash and the hamming distance on the MD5 hashes. The hashes were also of 32 bit. In all, we coded a more flexible Python-based SimHash library for running our tests of SimHash on genomic data.

3.2.4 Go-based SimHash

Our design and coding of the Go-based SimHash was similar to the Python-based SimHash. By basing our code on the Go library, we made a fundamental modification to accept genomic data. A special particularity with

the unmodified Go-based SimHash library is that though the library has the method for making shingles of words, it does not appear to be used. Also, the unmodified Go-based SimHash library very much assumes that the input is string of English words that could or could not be encoded in Unicode. Thus, we made fundamental modifications to the feature construction in the Go-based SimHash library to accept genomic data. Some of the modifications included building the features based on whether it is a substring of lowercase bases or uppercase bases, or building the features based on a k-mer size of say 15. We continued with the use of 64 bit hashes in Go.

3.2.5 Evolution of our Research

With both our Python-based SimHash and Go-based SimHash code, we fed in the genomic data to test how the core SimHash algorithm performs. Recall that the genomic data consisted of one strain of E. coli, which is about 4.5 million bases, while both of the yeast strands are about 12 million bases each. After running the SimHash algorithm in each respective language and observing the output of the distance between the genomic data, we found that the results were lackluster.

While say between one of the yeast strands and the E. coli, the SimHash algorithm was able to distinguish that they were quite different, the SimHash algorithm outputted that the difference between the other yeast strand and E. coli vs. the difference between both of the yeast strands was about equivalent. Fundamentally, just from the fact that E. coli is a much shorter genome, the result showed that SimHash was losing some sort of resolution.

We first thought that it might be due to how the Python-based SimHash was using the MD5 hash and was implementing SimHash that influenced the results. However, by bringing in the Go-based SimHash which does not use a MD5 hash but the raw bit values and computes the SimHash fingerprint based on a bit by bit operation with weighting, which is a bit different than the Python-based SimHash, the fact that both were returning similarly lackluster results showed that the implementation of SimHash is not likely the factor.

We next explored whether it is because of how we are inputting the genomic data. We explored different k-mer sizes and also decided to normalize the data by converting say all the bases to lowercase letters. However, even attempts at standardizing the genomic data or exploration of different shingling sizes did not alter the lackluster results outputted from SimHash. Thus, we concluded that it is likely due to how the SimHash algorithm is designed.

We explored the literature online, and discovered that SimHash was primarily designed for finding small discrepancies between websites. Thus, the design principle of SimHash was to have a good high resolution of small amount of changes. Thus, it can be implied that SimHash is unable to accurately handle too big of changes or hamming distances. With say a hamming distance of over 20, SimHash would start losing resolution of the distance between the hashes. Thus, we needed to find a way to make up for the loss of resolution.

3.3 SMASH

Thus came the development of original research based on what we learned and observed from MinHash and SimHash. Seeing how the existing SimHash algorithm was losing resolution when the Hamming distance was too great, we went back to the drawing board and examined each part of the SimHash algorithm closely. From the examination and with additional brainstorming, we came up with SMASH, which is designed to primarily address the shortcoming of resolution loss with SimHash.

3.3.1 Design

One of the major loss of resolution with SimHash was with the following: when a E. Coli genome with some 4.5 million bases is compared to one of the yeast genomes with 12 million bases, SimHash outputs that it is about the same hamming distance apart as another comparison, which is instead between to two yeast genomes and which both of which have about 12 million bases. Thus, we sought to address the loss of frequency in the resolution.

First, we observed that the basic property of the LSHs were the consen-

sus to find a way to build a feature structure out of the dataset. In our case, we decided to build our feature structure based on frequency. The way this is handled is that for each of the genome file that is inputted, we parse it such that it becomes one complete contiguous string. Next, we chop it up into the respective k-mers. An important thing to note is that the size of the k-mer needs to be specified beforehand, and thus we test for the optimal k-mer size and describe it in the Results section. The frequency is based on how frequently the k-mers show up in the respective genome files.

Based on the order of the feature structure, we then construct vectors consisting of the frequency of the k-mers. This is then fed into the cosine similarity function, which is used in SimHash, to compute the cosine similarity. Notice that a fundamental difference between SimHash and SMASH is that while SimHash focuses on computing a weighting of frequency of bits, SMASH is instead focusing on computing a weighting of frequency of k-mers. For a finer look into what SMASH does, the pseudocode for the algorithm is accompanied in the next section.

3.3.2 Algorithm

```
file1, file2 = input 2 genome files
k = k-mer size

for each file:
    parse and format into one contiguous string

for each file:
    file1_kmers, file2_kmers = divide up file into k-mers

file1_count = dict()
file2_count = dict()

for each kmer in file1_kmers:
    file1_count[kmer] += 1

for each kmer in file2_kmers:
```

```
    file2_count[kmer] += 2

for subset of observed k-mers in file_1:
    vector1 = vector of kmer counts from file1_count

for subset of observed k-mers in file_2:
    vector2 = vector of kmer counts from file2_count

similarity_distance = 1 - cosine(vector1, vector2)
```

4 Contributions of Each Member

[1 paragraph] TO DO

5 Results

[1.5 pages] TO DO Insert results of different k-mer sizes and of different genomic sequences.

6 Conclusions

[0.5 pages] TO DO There are more ways to judge similarity between two genomic sequences than just hashing the entirety of each of the genome and comparing how close in relation both are.

References

[1] Caitlin Sadowski and Greg Levin. *SimHash: Hash-based Similarity Detection*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.473.7179&rep=rep1&type=pdf, 2007.

[2] Anshumali Shrivastava and Ping Li. *In Defense of MinHash Over SimHash*. Journal of Machine Learning, 33, 2014.

[3] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren and Adam M. Phillippy. *Mash: fast genome and metagenome distance estimation using MinHash*. Genome Biology, 17:132, 2016.