Ashwin Sekar

George MAson University  Summer 2013

Lab Journal

# **6/26/13**

First day of internship, received 3 publications from mentor on DNA clustering and the current methods that exist. My mentor also gave me a data set with 10,000 reads from an acid mine in California. The DNA Sequences come from the soil samples taken from this acid mine. This is the data set I will be using to run my preliminary experiments.

After reading the publications I concluded that MC-MinH is the current leading algorithm in terms of speed, and U-CLUST is the best in terms of accuracy. Mentor wants me to create a hybrid algorithm that will use a compression algorithm to shorten the DNA sequences before clustering them. This process will hopefully make clustering more efficient while maintaining accuracy.

# 6/29/13

After shortening my list of potential compression algorithms through my discussion with my mentor, we decided on the following list: Sequitur, Lz4 and RLE.

Sequitur – This algorithm implementation involved taking a sequence and creating a set of grammar rules to represent the sequences. Separately, a database of these rules would be stored for reference. All the sequences operate on the same set of rules and are represented by which rules they use and the order of them. The rules must be used at least twice in order to be a valid rule and must contain at least 2 or more non terminal characters.

Lz4 – Google’s Lz4 algorithm is a very fast lossless compression algorithm, providing compression speed at 300 MB/s per core, scalable with multi-cores CPU. It also features an extremely fast decoder, with speed in GB/s per core, typically reaching RAM speed limits on multi-core systems.

RLE - Run Length Encoding is a compression method which involves taking large repeating sequences and representing them with a counter on the amount of repetition.

I began by looking at the standard java implementation of Sequitur and feeding my DNA sequences into it to see the results. The algorithm generated a set of grammar rule and each sequence was represented by a permutation of these rules. However this representation did not compress the sequences at all. In all of the cases, this representation almost doubled the size of the original sequence.

# 7/1/13

After explaining the problem to mentor, he suggested altering the algorithm to use a common rule database among the entire set of sequences. I had to modify the compression method to the following:

**public** **static** **void** compress(String s) {

**for**(**int** i = 0; i < s.length(); i++){

*firstRule*.last().insertAfter(**new** terminal(s.charAt(i)));

*firstRule*.last().p.check();

}

*firstRule*.last().insertAfter(**new** guard(*firstRule*, **true**));

};

The last line inserts a guard in between the different sequences, so that the program knows not to create rules by appending the end of one sequence with the beginning of another, but it still uses the same rule database for the entire data set because it treats it as one giant rule.

I also had to modify the extraction method for getting the compressed rules out of the database:

**public** String getRule0(){

Vector<rule> rules = **new** Vector<rule>(*numRules*);

rule currentRule;

rule referedTo;

symbol sym;

**int** index;

**int** processedRules = 0;

StringBuffer text = **new** StringBuffer();

rules.addElement(**this**);

currentRule = (rule)rules.elementAt(processedRules);

text.append(' ');

**for** (sym=currentRule.first();(!sym.isGuard());sym=sym.n){

**if** (sym.isNonTerminal()){

referedTo = ((nonTerminal)sym).r;

**if** ((rules.size() > referedTo.index) &&

((rule)rules.elementAt(referedTo.index) ==

referedTo)){

index = referedTo.index;

}**else**{

index = rules.size();

referedTo.index = index;

rules.addElement(referedTo);

}

text.append(index);

}**else**{

**if** (sym.value == ' '){

text.append('\_');

}**else**{

**if** (sym.value == '\n'){

text.append("\\n");

}**else** **if**(sym.value == -1) {

text.deleteCharAt(0);

**return** text.toString();

} **else** {

text.append((**char**)sym.value);

}

}

}

text.append(' ');

}

**return** text.toString();

}

# 7/8/13

The similarity measurement for Sequitur is complete, we have decided to use the Jaccard coefficient as the final method as it most accurately depicts the similarity between two DNA sequences compressed by Sequitur.

The Jaccard coefficient measures the similarity of two sequences by decomposing them into their fundamental grammar rules and comparing the percentage of shared rules between the two sequences to the appropriate lengths of the rules. It essentially takes the weighted average of the amount of rules shared by two sequences. Here is the final code for computing the Jaccard coefficient:

**public** **static** **double** jaccard(String s1, String s2) {

HashSet<String> seq1 = **new** HashSet<String>(), seq2 = **new** HashSet<String>(), intersection, union;

**for**(String s : s1.split(" ")) {

**if**(Character.*isDigit*(s.charAt(0)))

seq1.add(s);

}

**for**(String s : s2.split(" ")) {

**if**(Character.*isDigit*(s.charAt(0)))

seq2.add(s);

}

intersection = **new** HashSet<String>();

**for**(String s : seq1) {

**for**(String t : seq2) {

**if**(Math.*max*(Sequitur.*firstRule*.similarity(s,t),Sequitur.*firstRule*.similarity(t, s)) >= .4) {

intersection.add(s);

**break**;

}

}

}

union = **new** HashSet<String>(seq1);

union.addAll(seq2);

**return** ((**double**)intersection.size())/union.size();

}

In order to determine if two sequences are composed by similar grammar rules, I needed to create a similarity method that is used to populate a look up table of similarity values between all the pairs of the rules. Then if this similarity index is greater than .4, the rules are considered similar enough to be accounted for in the Jaccard Coefficient. The similarity method:

**public** **double** similarity(String rule1, String rule2) {

**int** r1, r2;

rule currentRule;

symbol sym;

**int** sum = 0, counter = 0;

r1 = Integer.*parseInt*(rule1);

r2 = Integer.*parseInt*(rule2);

**if**(Sequitur.*similarity*[r1][r2] != -1) {

**return** Sequitur.*similarity*[r1][r2];

} **else** **if**(r1 == r2) {

Sequitur.*similarity*[r1][r2] = 1;

**return** 1;

} **else** {

currentRule = Sequitur.*rules*.get(r2);

**for** (sym=currentRule.first();(!sym.isGuard());sym=sym.n){

**if**(sym.isNonTerminal()) {

sum += similarity(rule1, ((nonTerminal)sym).r.index + "") \* ((nonTerminal)sym).r.length;

counter += ((nonTerminal)sym).r.length;

} **else** {

counter++;

}

}

Sequitur.*similarity*[r1][r2] = sum/(**double**)counter;

**return** sum/(**double**)counter;

}

}

**public** **void** calculateLength() {

symbol sym;

**for**(sym = first(); (!sym.isGuard());sym=sym.n) {

**if**(sym.isNonTerminal()) {

**if**(((nonTerminal)sym).r.length == 0) ((nonTerminal)sym).r.calculateLength();

length += ((nonTerminal)sym).r.length;

} **else** {

length++;

}

}

}

The calculate length method was used to compute weighted averages.

# 7/17/13

Program finished running and produced results:

From these results for stage 2, the actual clustering, we are taking the top 3 algorithms: CDMLz4, CDMSequitur, and Lz4.

Mentor outlined the second stage, in which I have to use these 3 algorithms to compress the datasets individually and cluster these compressed datasets based on the similarity measures of the three algorithms. For CDMLz4 and CDMSequitur, this similarity measurement is the CDM, where Lz4 I have to use the edit distance.

Created the populate sequence method, the skeleton of the clustering method, which reads in the data set and creates the appropriate list of sequences:

**public** **static** **void** populateSequences(**int** n, **boolean** wsim) {

*sequences* = **new** ArrayList<String>();

String sequence;

String curLine = " ";

**try** {

*f*.readLine(); // Skips past first header

**if**(n != -1) {

**for**(**int** i = 0; i < n ; i++) {

sequence = "";

**while**((curLine = *f*.readLine()).charAt(0) != '>') {

curLine = curLine.replaceAll("n", ""); // Filter out the n's in the DNA sequence

sequence += curLine;

}

*sequences*.add(sequence);

}

} **else** {

**while**(**true**) {

sequence = "";

curLine = *f*.readLine();

**if**(curLine == **null**) **break**;

**while**((curLine).charAt(0) != '>') {

curLine = curLine.replaceAll("n", ""); // Filter out the n's in the DNA sequence

sequence += curLine;

curLine = *f*.readLine();

**if**(curLine == **null**) **break**;

}

*sequences*.add(sequence);

}

}

} **catch** (IOException e) {

e.printStackTrace();

}

*size* = *sequences*.size();

*origSequences* = **new** ArrayList<String>(*sequences*);

*init*();

**for**(**int** i = 0; i < *sequences*.size(); i++) {

*compress*(*sequences*.get(i));

}

*sequencesSequitur* = **new** ArrayList<String>();

Sequitur.*firstRule*.getRules(*sequencesSequitur*, **false**);

*origSequencesSequitur* = **new** ArrayList<String>(*sequencesSequitur*);

*sequencesLz4* = **new** ArrayList<LZ4>();

**for**(**int** i = 0; i < *sequences*.size(); i++) {

**try** {

*sequencesLz4*.add(LZ4.*compress*(*sequences*.get(i)));

} **catch** (UnsupportedEncodingException e) {

e.printStackTrace();

}

}

*origSequencesLz4* = **new** ArrayList<LZ4>(*sequencesLz4*);

}

# 8/5/13

Finished the statistics method to collect data during the clustering process. The method tracks singletons (number of clusters with only 1 element), doubletons (number of clusters with only 2 elements), the Weighted Similarity (Average similarity between all of the sequences in a cluster) , number of clusters, runtime, memory usage, and the diversity indexes (Shannon, Chao1, and ACE. All these indexes show the diversity of the “species” represented by each cluster. Ideally clustering should separate and environment of dna samples into the species that exist in that environment, each designated by a cluster).

**public** **static** **double**[] generateStatistics(ArrayList<ArrayList<String>> clusters, **boolean** wsim) {

**double**[] stat = **new** **double**[6];

**int**[] freq = **new** **int**[*size*+1];

**double** total = 0;

**for**(ArrayList<String> c : clusters) {

**if**(c.size() == 1) stat[0]++;

**if**(c.size() == 2) stat[1]++;

freq[c.size()]++;

total += (c.size()\*(c.size()-1))/2;

}

**if**(wsim) {

stat[2] = 0;

**double** sim,den;

System.*out*.print(" ");

**for**(ArrayList<String> c : clusters) {

sim = 0;

den = 0;

**for**(**int** i = 0; i < c.size(); i++) {

**for**(**int** j = i+1; j < c.size(); j++) {

**if**(*LSA*.containsKey(c.get(i) + "\_" + c.get(j)))

sim += *LSA*.get(c.get(i) + "\_" + c.get(j));

**else** **if**(*LSA*.containsKey(c.get(j) + "\_" + c.get(i)))

sim += *LSA*.get(c.get(j) + "\_" + c.get(i));

**else** {

*LSA*.put(c.get(i) + "\_" + c.get(j), *localSeqAlignmentSimilarity*(c.get(i),c.get(j)));

sim += *LSA*.get(c.get(i) + "\_" + c.get(j));

}

den++;

System.*out*.printf("\r %.2f", den/total);

}

}

**if**(c.size() != 1)

sim = sim/den;

**else**

sim = 1.0;

stat[2] += (c.size()/((**double**) *size*)) \* sim;

}

stat[2] = stat[2]\*100;

System.*out*.println();

}

**int** obs = clusters.size();

stat[3] = obs + ((stat[0]\*(stat[0]-1.0))/(2.0\*(stat[1]+1.0)));

**for**(**int** i = 0 ; i < clusters.size(); i++) {

stat[4] += ( clusters.get(i).size()/ (**double**) *size*) \* Math.*log*(clusters.get(i).size()/ (**double**) *size*);

}

stat[4] = -stat[4];

**int** threshold = 10;

**int** sRare = 0, sAbund = 0;

**for**(ArrayList<String> s : clusters) {

**if**(s.size() > threshold) sAbund++;

**else** sRare++;

freq[s.size()]++;

}

**double** nRare = 0;

**double** a = 0;

**for**(**int** i = 1 ; i <= threshold; i++) {

nRare += i \* freq[i];

a += i\*(i-1)\*freq[i];

}

**double** cACE = 1 - freq[1]/nRare;

**double** lamdaACE2 = Math.*max*((sRare/cACE) \* (a/(nRare\*(nRare -1)))-1, 0);

**double** lamdaACE = Math.*max*(lamdaACE2\*(1 + (nRare\*(1 - cACE)\*a)/(nRare\*(nRare-cACE))),0);

**if**(Math.*sqrt*(lamdaACE2) < .8) {

stat[5] = sAbund + (sRare/cACE) + (freq[1]/cACE)\*lamdaACE2;

} **else** {

stat[5] = sAbund + (sRare/cACE) + (freq[1]/cACE)\*lamdaACE;

}

*sequences* = **new** ArrayList<String>(*origSequences*);

*sequencesLz4* = **new** ArrayList<LZ4>(*origSequencesLz4*);

*sequencesSequitur* = **new** ArrayList<String>(*origSequencesSequitur*);

System.*out*.println();

System.*out*.println();

**return** stat;

}

Now that we have the statistic collecting part of the algorithm, I uploaded the program onto the server and started it. Hopefully it should be finished collecting the data by Friday.

# 8/9/13

Fixed the statistic generating portion of my algorithm. The Weighted Similarity calculation has been made much more efficient by precomputing the local sequence alignment values that needed to be calculated between all of the pairs of sequences. This is the reason the data collection was being so slowed down. The code for the precomputation:

**public** **class** Precompute {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

String file = "60\_Equal\_112R.fasta";

ArrayList<String> sequences = **new** ArrayList<String>();

String sequence;

String curLine = " ";

**try** {

BufferedReader f = **new** BufferedReader(**new** FileReader("LSHDIV\_DataFiles/" + file));

f.readLine(); // Skips past first header

**while**(**true**) {

sequence = "";

curLine = f.readLine();

**if**(curLine == **null**) **break**;

**while**((curLine).charAt(0) != '>') {

curLine = curLine.replaceAll("n", ""); // Filter out the n's in the DNA sequence

sequence += curLine;

curLine = f.readLine();

**if**(curLine == **null**) **break**;

}

sequences.add(sequence);

}

PrintWriter o = **new** PrintWriter(**new** File("PreComputed/" + file));

**long** total = sequences.size();

total = total \* (total-1);

total = total/ 2;

**long** counter = 0;

System.*out*.print(" ");

**for**(**int** i = 0; i < sequences.size(); i++) {

**for**(**int** j = i +1; j < sequences.size(); j++) {

o.println(sequences.get(i) + "\_" + sequences.get(j));

o.println(ClusterAll.*localSeqAlignmentSimilarity*(sequences.get(i),sequences.get(j)));

counter++;

System.*out*.print("\r" + counter + "/" + total + " " + (**double**)counter/total + "% ");

}

}

System.*out*.println();

f.close();

o.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

Reuploaded the program to the server and started the data collection, hopefully the program should finish sometime next week without problems.

# 8/12/13

The 16S simulated data sets finished running. These were the data sets that were artificially created to test clustering algorithms. Since these data sets were “created” there is a right answer for the number of clusters so it is a true test of my algorithms’ clustering ability.

From these results, an “optimal” threshold for each of these algorithms can be obtained. These thresholds are where the number of clusters closely match the leading algorithms. If my algorithms are stable, than these optimal thresholds should stay optimal for all of the environmental data sets too.

# 8/16/13

Last day of internship, all of the data finished being collected by the program. This morning I averaged the optimal thresholds for each of the individual data sets to come up with the optimal thresholds. I averaged them twice, once based on the # of clusters metric, and once on the weighted similarity metric:

**Optimal Thresholds by W. Sim comparison**

|  |  |
| --- | --- |
| Algorithm | Threshold |
| CDMLz4 | .76 |
| CDMSequitur | .66 |
| Lz4 | .3 |
|  |  |

**Optimal Thresholds by # Clu comparison**

|  |  |
| --- | --- |
| Algorithm | Threshold |
| CDMLz4 | .64 |
| CDMSequitur | .34 |
| Lz4 | .18 |

With these two thresholds I extrapolated the data in order to create a comparison chart:

**Clustering Results on 16S Environmental sets using Threshold #1**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Algorithm | Metric | | 53R | 55R | 112R | 115R | 137 | 138 |
| CDMLz4 | # Clu | | 320 | 288 | 507 | 324 | 306 | 308 |
| W.Sim | | 94.30 | 93.35 | 92.01 | 94.30 | 93.81 | 93.91 |
| Time(s) | | 6.9 | 5.1 | 5.5 | 5.3 | 9.2 | 6.4 |
| CDMSequitur | # Clu | 346 | | 292 | 528 | 322 | 306 | 319 |
| W.Sim | 93.93 | | 93.07 | 92.41 | 92.42 | 93.79 | 93.99 |
| Time(s) | 59.4 | | 44.9 | 59.5 | 333.0 | 71.8 | 58.3 |
| Lz4 | # Clu | 517 | | 448 | 775 | 509 | 456 | 465 |
| W.Sim | 96.63 | | 95.39 | 96.00 | 95.73 | 96.08 | 95.91 |
| Time(s) | 21.5 | | 14.3 | 20.8 | 168.9 | 24.0 | 19.2 |
| MC-MinH | # Clu | 1165 | | 1077 | 1634 | 1156 | 1020 | 1042 |
| W.Sim | 96.90 | | 92.45 | 91.18 | 93.33 | 95.86 | 93.10 |
| Time(s) | 2.5 | | 2.1 | 3.3 | 3.0 | 2.7 | 2.5 |
| MC-LSH | #Clu | 1172 | | 1199 | 1795 | 1205 | 1041 | 1072 |
| W.Sim | 96.90 | | 93.12 | 91.33 | 93.50 | 95.86 | 93.10 |
| Time(s) | 161.0 | | 183.0 | 317.0 | 188.0 | 172.0 | 175.0 |
| UCLUST | #Clu | 1062 | | 992 | 1561 | 1071 | 900 | 923 |
| W.Sim | 96.67 | | 91.67 | 91.02 | 93.33 | 93.50 | 92.82 |
| Time(s) | 2.0 | | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| CD-HIT | #Clu | 824 | | 716 | 1196 | 820 | 712 | 725 |
| W.Sim | 92.56 | | 90.80 | 90.61 | 93.33 | 91.82 | 90.16 |
| Time(s) | 3.6 | | 3.1 | 3.9 | 3.8 | 3.2 | 3.1 |
| ESPRIT | #Clu | 940 | | 859 | 1361 | 970 | 818 | 832 |
| W.Sim | 93.12 | | 91.35 | 90.88 | 93.33 | 91.82 | 90.16 |
| Time(s) | 283.0 | | 266.0 | 537.0 | 348.0 | 280.0 | 296.0 |
| DOTUR | #Clu | 1241 | | 1258 | 1854 | 1279 | 1096 | 1121 |
| W.Sim | 96.95 | | 94.06 | 91.33 | 93.50 | 95.86 | 93.10 |
| Time(s) | 5129.0 | | 3511.0 | 5567.0 | 9237.0 | 6563.0 | 5618.0 |
| Mothur | #Clu | 1238 | | 1256 | 1853 | 1278 | 1094 | 1119 |
| W.Sim | 96.95 | | 94.06 | 91.33 | 93.50 | 95.86 | 93.10 |
| Time(s) | 10130.0 | | 5940.0 | 12303.0 | 13501.0 | 12861.0 | 12310.0 |

**Clustering Results on 16S Environmental sets using Threshold #2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Algorithm | Metric | | 53R | 55R | 112R | 115R | 137 | 138 |
| CDMLz4 | # Clu | | 1009 | 868 | 1419 | 1056 | 849 | 858 |
| W.Sim | | 99.00 | 98.69 | 98.72 | 99.03 | 99.21 | 98.90 |
| Time(s) | | 34 | 24.2 | 23.9 | 24.1 | 34.7 | 27.8 |
| CDMSequitur | # Clu | 827 | | 745 | 1236 | 870 | 803 | 748 |
| W.Sim | 98.27 | | 98.14 | 97.94 | 97.94 | 98.74 | 98.49 |
| Time(s) | 187.7 | | 154.6 | 201.9 | 1032.5 | 244.3 | 186.7 |
| Lz4 | # Clu | 922 | | 777 | 1302 | 921 | 754 | 789 |
| W.Sim | 98.64 | | 98.26 | 98.50 | 98.52 | 98.66 | 98.66 |
| Time(s) | 43.7 | | 31.9 | 43.6 | 384.4 | 47.4 | 37.6 |
| MC-MinH | # Clu | 1165 | | 1077 | 1634 | 1156 | 1020 | 1042 |
| W.Sim | 96.90 | | 92.45 | 91.18 | 93.33 | 95.86 | 93.10 |
| Time(s) | 2.5 | | 2.1 | 3.3 | 3.0 | 2.7 | 2.5 |
| MC-LSH | #Clu | 1172 | | 1199 | 1795 | 1205 | 1041 | 1072 |
| W.Sim | 96.90 | | 93.12 | 91.33 | 93.50 | 95.86 | 93.10 |
| Time(s) | 161.0 | | 183.0 | 317.0 | 188.0 | 172.0 | 175.0 |
| UCLUST | #Clu | 1062 | | 992 | 1561 | 1071 | 900 | 923 |
| W.Sim | 96.67 | | 91.67 | 91.02 | 93.33 | 93.50 | 92.82 |
| Time(s) | 2.0 | | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| CD-HIT | #Clu | 824 | | 716 | 1196 | 820 | 712 | 725 |
| W.Sim | 92.56 | | 90.80 | 90.61 | 93.33 | 91.82 | 90.16 |
| Time(s) | 3.6 | | 3.1 | 3.9 | 3.8 | 3.2 | 3.1 |
| ESPRIT | #Clu | 940 | | 859 | 1361 | 970 | 818 | 832 |
| W.Sim | 93.12 | | 91.35 | 90.88 | 93.33 | 91.82 | 90.16 |
| Time(s) | 283.0 | | 266.0 | 537.0 | 348.0 | 280.0 | 296.0 |
| DOTUR | #Clu | 1241 | | 1258 | 1854 | 1279 | 1096 | 1121 |
| W.Sim | 96.95 | | 94.06 | 91.33 | 93.50 | 95.86 | 93.10 |
| Time(s) | 5129.0 | | 3511.0 | 5567.0 | 9237.0 | 6563.0 | 5618.0 |
| Mothur | #Clu | 1238 | | 1256 | 1853 | 1278 | 1094 | 1119 |
| W.Sim | 96.95 | | 94.06 | 91.33 | 93.50 | 95.86 | 93.10 |
| Time(s) | 10130.0 | | 5940.0 | 12303.0 | 13501.0 | 12861.0 | 12310.0 |

The results show a definite increase in the diversity of the clusters with the thresholds chosen by number of clusters versus weighted similarity. CDMLz4 has the fastest runtime indicating it is the most efficient of the algorithms. It outperforms all of the algorithms except MC-MinH in terms of speed. In terms of accuracy, it has a much higher weighted similarity compared to MC-MinH, and comparable species diversity statistics. Weighted similarity is calculated based on how similar the sequences within a cluster are; however CDMLz4 and MC-MinH both use slightly different similarity metrics so the W. Sim comparison is not precise. Differences of 2-3% aren’t indicative of major accuracy differences, especially because the two algorithms have very similar diversity metrics.

Although CDMSequitur and Lz4 are comparable to CDMLz4 in terms of accuracy metrics, the run time of those two algorithms are much greater, ranking them in the 5th-8th place range for most of the data sets. Therefore of the three algorithms, CDMLz4 is the most practical for future use.