LatentStrainAnalysis Documentation

Release 0.1

Brian Cleary

October 17, 2016

Contents

Welcome to the Latent Strain Analysis (LSA) documentation!

LSA was developed as a pre-assembly tool for partitioning metagenomic reads. It uses a hyperplane hashing function and streaming SVD in order to find covariance relations between k-mers. The code, and the process outlined in "Analyzing Large Data Sets" in particular, have been optimized to scale to massive data sets in fixed memory with a highly distributed computing environment.

CHAPTER 1

Overview

LSA operates at the level of k-mers, and, more specifically, on hashed k-mers. In the first steps of LSA we generate a hash function that maps k-mers to columns in a matrix. The rows of this matrix will represent different samples, and we are interested in the covariance structure of k-mers across samples, with the idea that this covariance can reflect the physical linkage between k-mers found in the same genome.

We can get at the covariance information via Singular Value Decomposition, and, since our matrices might be super huge to accomodate a large diversity of k-mers, we implement a streaming SVD so that the whole thing works in fixed memory. Then, working in the eigenspace of k-mer covariation, we find a k-mer clustering that is subsequently used to partition reads into disjoint subsets.

Contents

2.1 License

2.2 Getting Started

2.2.1 Dependencies

LSA depends on the following software and libraries:

```
Python (2.7)
NumPy
SciPy
Gensim
GNU Parallel
Pyro4
```

To install GNU Parallel (http://www.gnu.org/software/parallel/):

```
(wget -O - pi.dk/3 || curl pi.dk/3/) | bash
```

If you’re having trouble with Parallel, check the version (parallel –version), and look for the following:

```
WARNING: YOU ARE USING --tollef. IF THINGS ARE ACTING WEIRD USE --gnu.
```

In this case, heed the advice and use –gnu.

Note that you will not need Pyro4 to analyze the test data as described below.

2.2.2 Test Data

To get started, make sure you have the dependencies listed above (you'll need GNU Parallel, but not Pyro4), and then download the repo (eg via git clone).

Next, unpack testData.tar.gz (into original_reads/). This folder contains a subsample of 10k reads from each of the 18 metagenomic libraries used in Sharon et. al. (SRA052203), and in the original LSA methods paper.

Each sample has been randomly spiked with 0-2,000 mock reads from a *Bacillus thuringiensis* plasmid (NG_035027.1). Note that there is one file per sample, that mate pairs are interleaved, and that files are named sample_id.*.fastq. You'll need a couple GBs of RAM to work through the test data. If something goes wrong, it's best to first check the most recently modified file in "Logs/" to track down the error.

Begin by performing some hashing tasks using 6 threads, k-mers of length 33, and a hash size of 2^22::

```
$ bash HashCounting.sh 6 33 22
```

Next, calculate the SVD and cluster the hashed k-mers using 6 threads, a hash size of 2^22 and a cluster threshold of 0.8::

```
$ bash KmerSVDClustering.sh 6 22 .8
```

Finally, partition the original reads (4 threads)::

```
$ bash ReadPartitioning.sh 4
```

Since the process includes several points of randomization, the results from each run will vary. However, when the read partitioning step completes, a report is made to indicate the specificity and completeness of spiked read partitioning::

```
Quantifying spike enrichment
Total spiked reads: 21528
Spiked read counts by partition (top 5)
partition 5: 19771 spiked reads out of 20291 total reads in partition 5 (97%)
partition 146: 1207 spiked reads out of 1557 total reads in partition 146 (77%)
partition 131: 356 spiked reads out of 670 total reads in partition 131 (53%)
partition 112: 142 spiked reads out of 492 total reads in partition 112 (28%)
partition 129: 4 spiked reads out of 506 total reads in partition 129 (0%)
```

Analyzing larger collections

LSA has been written to be highly efficient in analyzing very large collections. For data sets larger than a few Gb, significant gains in wall time can be made by running in a cluster environment. In these cases, the process is essentially the same as what is outlined above. Detailed steps and job array submission scripts can be found in LSF-Scripts/README.md.

2.3 Analyzing Large Data Sets

LSA has been optimized for the analysis of very large data sets in a highly distributed computing environment. To run this code you'll need Python (2.7), NumPy, SciPy, Gensim, and Pyro4. The following steps, which follow the same procedure as the streamlined "Getting Started" version but are more verbose, are generally one of two types: a command to create an LSF submission script, or a command to submit an LSF job.

To get started, download the LSA repo (eg via git clone). From here on, we'll assume that you've got the repo and your project housed in one location. You'll need to change the queue ("-q") argument in each *.q file to suit your environment.

2.3.1 Initialization

Initialize your project with a few directories ("-i" takes the location of your reads, and "-n" takes the number of samples):

```
$ python LSFScripts/setupDirs.py -i /input/reads/ -n 50
```

2.3.2 Splitting the input files

Begin by splitting the original reads (from many samples) into many small files:

```
$ bsub < LSFScripts/SplitInput_ArrayJob.q
```

The purpose of this is to create many small files that can each be operated on by a single task in a distributed environment. The size of many job arrays downstream from this point are set by the number of chunks created in this step. Note that this code assumes the files are named sample_id.*.fastq.1 and sample_id.*.fastq.2 for paired reads. If you used some other naming convention, this needs to be reflected in line 26 in array_merge.py:

WARNING

If your input files are significantly different from paired fastq files separated into 2 parts (.fastq.1 and .fastq.2) plus a singleton file (.single.fastq.1), then you will either need to modify these python files, or just take it upon yourself to split your files into chunks containing ~1million reads each, and named like: sample_id.fastq.xxx, where ".xxx" is the chunk number (eg '.021')

2.3.3 Generate the hash function

Create a k-mer hash function by drawing a bunch of random hyperplanes. If you want to adjust the k-mer length or hash size, alter the "-k" or "-s" arguments in the create_hash.py command of CreateHash_Job.q.:

```
$ python LSFScripts/create_jobs.py -j CreateHash -i ./
$ bsub < LSFScripts/CreateHash_Job.q
```

Look at the log files for this when it's done. Also, the hash function should be stored in hashed_reads/Wheels.txt

2.3.4 Hash the reads

Hashing all the reads:

```
$ python LSFScripts/create_jobs.py -j HashReads -i ./
$ bsub < LSFScripts/HashReads_ArrayJob.q
```

Failure of a small fraction of these jobs is tolerable in as much as it is not necessary to hash every read in order to continue (these reads will then be missed in the remainder of the analysis).

2.3.5 Hashed k-mer counting

Tabulating k-mer counts in 1/5th of each sample:

First, make sure that there is just one *.fastq file per sample in original_reads/. The reason this is important is that the number of *.fastq files will be used to determine the array size. (The *.fastq.* files are no longer needed, so you can remove those as well if you want).:

```
$ python LSFScripts/create_jobs.py -j MergeHash -i ./
$ bsub < LSFScripts/MergeHash_ArrayJob.q
```

You don't really want any of these tasks to fail. So take a look at the logs when it's done and resubmit anything that died.

Merging the 5 count files for each sample:

```
$ python LSFScripts/create_jobs.py -j CombineFractions -i ./
$ bsub < LSFScripts/CombineFractions_ArrayJob.q
```

If any of these fail, they need to be run again.

2.3.6 Create the abundance matrix

Global (k-mer) conditioning:

```
$ python LSFScripts/create_jobs.py -j GlobalWeights -i ./
$ bsub < LSFScripts/GlobalWeights_Job.q
```

This launches a single job that must succeed to continue. Should produce cluster_vectors/global_weights.npy

Writing martix rows to separate files, and computing local (sample) conditioning:

```
$ python LSFScripts/create_jobs.py -j KmerCorpus -i ./
$ bsub < LSFScripts/KmerCorpus_ArrayJob.q
```

These must all complete to continue. Relaunch any that failed. This job should produce one hashed_reads/*.conditioned file per sample.

2.3.7 Streaming SVD

Calculating the SVD (streaming!):

```
$ python LSFScripts/create_jobs.py -j KmerLSI -i ./
$ bsub < LSFScripts/KmerLSI_Job.q
```

For very large matrices, this one will probably take a couple days to complete. Will produce cluster_vectors/kmer_lsi.gensim.

2.3.8 K-mer clustering

Create the cluster index:

```
$ python LSFScripts/create_jobs.py -j KmerClusterIndex -i ./
$ bsub < LSFScripts/KmerClusterIndex_Job.q
```

This step will set the k-mer cluster seeds, and the number of these seeds ultimately affects the resolution of partitioning. It is highly recommended that you check cluster_vectors/numClusters.txt for the number of clusters. If the resolution is markedly different from the expected / desired resolution, this job should be re-run with a different "-t" value in the submission script. Roughly speaking, we've found the following values to work for different scale datasets: 0.5-0.65 for large scale (Tb), 0.6-0.8 for medium scale (100Gb), >0.75 for small scale (10Gb). See misc/parameters.xlsx for more info.

Cluster blocks of k-mers:

```
$ bsub < LSFScripts/KmerClusterParts_ArrayJob.q
```

Merge cluster blocks:

```
$ bsub < LSFScripts/KmerClusterMerge_ArrayJob.q
```

Arrange k-mer clusters on disk:

```
$ python LSFScripts/create_jobs.py -j KmerClusterCols -i ./
$ bsub < LSFScripts/KmerClusterCols_Job.q
```

This should produce (among other things) a file cluster_vectors/kmer_cluster_sizes.npy

2.3.9 Read Partitioning

Partition all the read chunks:

```
$ python LSFScripts/create_jobs.py -j ReadPartitions -i ./
```

You'll need to modify ReadPartitions_ArrayJob.q to contain your tmp directory of choice.

```
$ sed 's/TMPDIR/\/your\/tmp\/dir/g' < LSFScripts/ReadPartitions_ArrayJob.q | bsub
```

If a few of these fail, it's not super critical, but if a large number fail you'll want to resubmit them.

Merge the partition chunks:

```
$ python LSFScripts/create_jobs.py -j MergeIntermediatePartitions -i ./
$ bsub < LSFScripts/MergeIntermediatePartitions_ArrayJob.q
```

If any of these jobs fail you'll need to resubmit them.

If you've made it this far...good job! Your reads are now partitioned. Have at em'!