Storey Lab Notebook

Peter Edge

July 9, 2014

June 9, 2014

Today I did not yet have Emily's code available to study or access to the cetus data, so instead I focused on preparing my workstation and reading literature related to my project. I focused on Skelly et al (http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3202289/) because it is perhaps the most relevant to what I am doing.

June 10, 2014

Today I briefly introduced myself to Emily's snp_pipepline code before spending most of my day at lab safety training.

June 11, 2014

Today I focused on reading Emily's code in the snp_pipepline.py file, and familiarizing myself with the libraries, tools, and paradigms it uses. I read through the documentation for ruffus (http://www.ruffus.org.uk) as well as reading up a bit on python iterators.

June 12, 2014

Today I finished studying the details of Emily's snp_pipepline and began trying to use pysam myself for the manipulation of sam files. I created directories RM_bowtie_test and S288C_bowtie_test in SeqSorter/sample_data and used them to align the test reads to the RM and S288C genomes using bowtie:

```
cd git/SeqSorter/sample_data
mkdir RM_bowtie_test S288C_bowtie_test
bowtie2-build RM11_1A/assembly/genome.fa RM_bowtie_test/RM_index
bowtie2-build S288C_reference_genome_R64-1-1_20110203/
    S288C_reference_sequence_R64-1-1_20110203.fsa S288C_bowtie_test/BY_index
bowtie2 -x RM_bowtie_test/RM_index -U E2a0_sample.fastq
    -S RM_bowtie_test/RM_bowtie_out.sam
bowtie2 -x S288C_bowtie_test/BY_index -U E2a0_sample.fastq
    -S S288C_bowtie_test/BY_bowtie_out.sam
```

I then started experimenting with pysam to understand how the library is used to access data from the alignment files.

June 13, 2014

Today I narrowed my focus to the method compare_mappings(infiles, outfile), since this is where the initial focus of my project will be. I familiarized myself with using pysam for navigating SAM files, using the alignment SAMs from the test dataset (created yesterday).

June 16 - 20, 2014

This week was spent coding, testing, and analyzing the initial prototype of SeqSort.py. All notes relevant to this process are in the messages appended to the github commits page and the issues tracker, while notes that pertain to code only are in the comments of SeqSort.py and testSeqSort.py:

`https://github.com/dgrtwo/SeqSorter`

Equation for the calculation of Pr(read | BY):

$$Pr(read|BY) = \prod_{i=1}^{L} \begin{cases} 1 - Pr(\text{miscall}), & \text{if } R_L \text{ matches } BY_L \\ \frac{Pr(\text{miscall})}{3}, & \text{otherwise} \end{cases}$$

Pr(read | RM) is calculated in the same fashion. Pr(BY | read) is calculated as follows, via Baiyes Rule with the law of total probability, (priors of 0.5 are assumed for BY and RM and divide out):

$$Pr(BY|read) = \frac{Pr(read|BY)}{Pr(read|BY) + Pr(read|RM)}$$

SortSeq computes the probability of BY and probability of RM for reads that bowtie maps to both.

June 23-27, 2014

This week David outlined for me a way to add a layer of sophistication for SortSeq, at least for our current experiment: handling the problem of read quality scores being an inaccurate representation of the probability of a mismatch. I will start by accumulating information about the proportion of bases for a given quality score in the RNAseq reads that are likely to be mismatches, and then ”correcting” the values associated with various quality scores.

The code for this analysis is in the file SeqSorter/estimateErrorFreq.py, and it walks through pairs of aligned_reads (RNA read aligned to BY, and same RNA read aligned to RM) in the two mappings, and builds a nested dictionary of this structure:

```
result[(BY chromosome, RM chromosome, BY position, RM position, consensus base)]
[RNA base][quality score] = count
```

That is, for reads that map to both chromosomes, if both genomes have the same base in that position, then we keep track of the specific spots the base mapped to in both BY and RM, what the base was in the RNA read, and what its quality score was. From this information, we build a set of 'likely consensus mappings' situations where both genomes agree on a base, and most aligned RNA bases agree on the base. Then, any base that disagrees with this likely consensus can be defined as random error. Then, we can tally the total amount of random errors that were associated with a base of a given quality, and therefore compute the likelihood of error from a given quality score empirically.

June 30-July 3, 2014

Lately I have mostly focused on optimization, commenting, and cleanup of SeqSorter and estimateErrorFreq. One substantial addition is the utilization of python's multiprocessing library to allow for utilization of all available cores on multiple core architectures. Since both programs process aligned RNAseq reads in a linear fashion, the general strategy is to interleave which reads are processed by which processes. For instance, while running with two cores, one core may process the evenly indexed reads while the other processes the odd indexed ones. This led to substantial speed increases for SeqSorter, around 1/3 of the processing time for a dataset of 250K alignedreads processed on cetus with 8 processes. On my dual core netbook, processing the 10K alignedread test set was accomplished in about 3/4 of the processing time as the same with the old version of SeqSorter. This should allow SeqSorter to effectively process millions of reads on more modest machines and laptops. The same strategy was implemented with estimateErrorFreq, but the overhead of repeatedly iterating over nested dictionaries to recombine them resulted in much slower runtime than the original version, so I have retired the multiprocessing version.

July 5-July 9, 2014

There are two main goals for this week:
1. find and verify a good fit for the function of mismatches per base as a function of phred score (in other words, characterize the U term which represents unreflected error):

```
Pr(mismatch) = 10^(Q/-10) + U
```

2. extend SeqSorter to use HTSeq to characterize the number of reads that map to different genes

I ran Bowtie — Seqsort on all of the RNAseq fastq files in the

```
/Genomics/grid/users/emilysn/ase/analysis/RNA-Seq/RUN_111213OB/split_fastq
```

HTSeq-count (quantify number of BY-sorted vs RM-sorted reads in each gene)
the following commands were run in the /grid/users/pedge/SeqSorter/all_sample_analysis directory:

```
for i in $(ls split_by_genotypes/*_BY_sorted.sam); do qsub -cwd -o count_out/
$(basename $i).out -e count_out/$(basename $i).err htseq-count -i ID -s no
-t gene $i ../sample_data/S288C_reference_genome_R64-1-1_20110203/
cropped_saccharomyces_cerevisiae_R64-1-1_20110208.gff; done;

for i in $(ls split_by_genotypes/*_RM_sorted.sam); do qsub -cwd -o count_out/
$(basename $i).out -e count_out/$(basename $i).err htseq-count -i ID -s no
-t gene $i ../sample_data/RM11_1A/assembly/genome.gff; done;
```

In other words, for all reads sorted by SeqSort to BY, run htseq count with the BY GFF file to assign them to genes. Do the same for RM.

Results of Bowtie | SeqSort on E1a12.fastq

```
Comparison of quality score error probs to calculated error probs:
4891463 reads : same sorting decision
5696 reads : different sorting decision
665893 unmapped by bowtie
29788 mapped by bowtie uniquely to BY
19685 mapped by bowtie uniquely to RM
3041863 double-mapped, but errors are same
562974 double-mapped, classified as BY based on errors
542341 double-mapped, classified as RM based on errors
34615 double-mapped, unclassifiable based on errors
TOTAL TIME: 3204.58062887
```

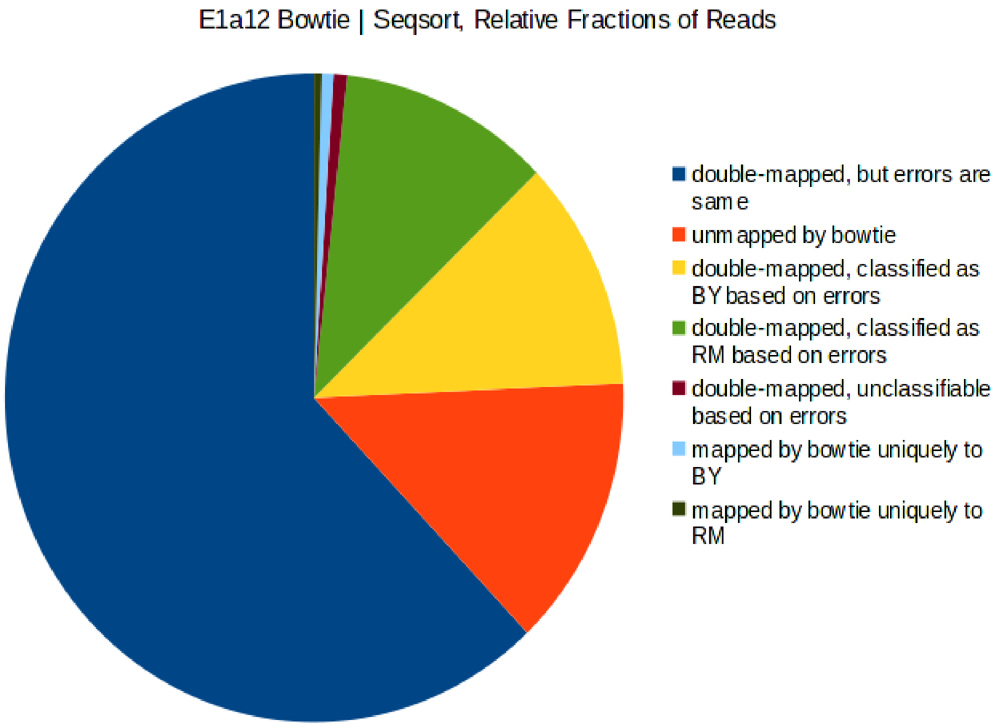

Figure 1: Pie chart of relative fractions of RNAseq reads that were mapped by bowtie or classified by SeqSorter for the E1a12 experiment.

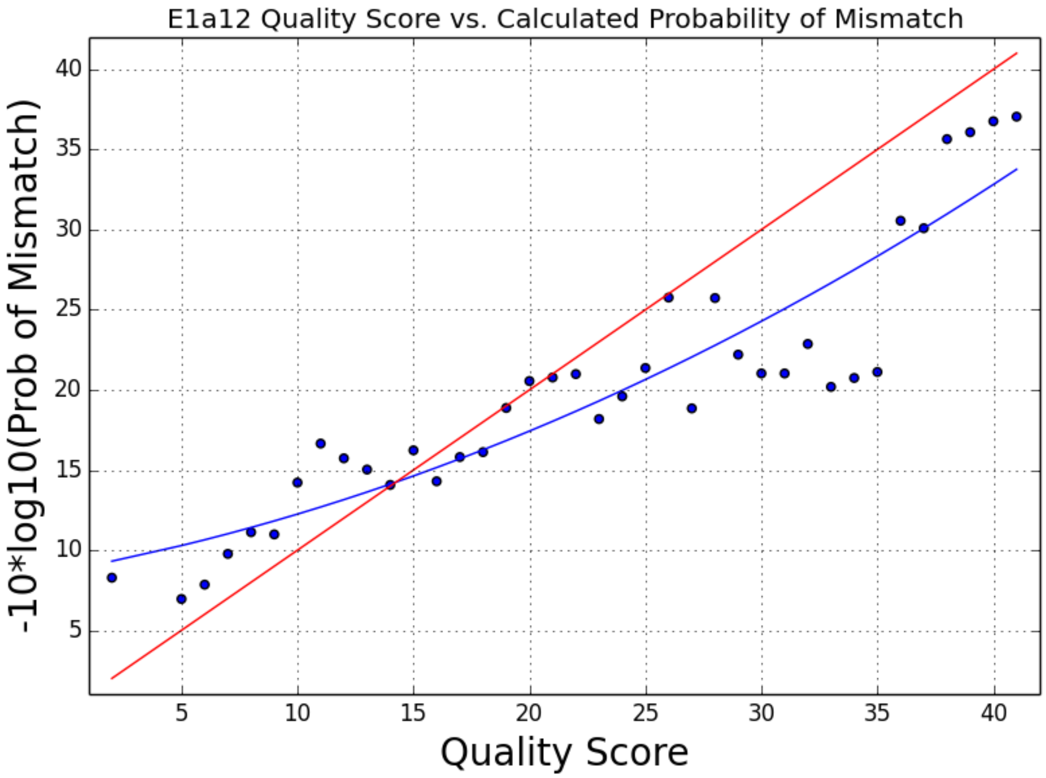

Figure 2: Best fit line for results of estimateErrorProbs.py on E1a12 data. R^2=0.841074.

Results of Bowtie | SeqSort on G1a60.fastq

```
Comparison of quality score error probs to calculated error probs:
4648137 reads : same sorting decision
4852 reads : different sorting decision
393689 unmapped by bowtie
34077 mapped by bowtie uniquely to BY
21484 mapped by bowtie uniquely to RM
3044121 double-mapped, but errors are same
587348 double-mapped, classified as BY based on errors
538265 double-mapped, classified as RM based on errors
34005 double-mapped, unclassifiable based on errors
TOTAL TIME: 3209.26929116
```

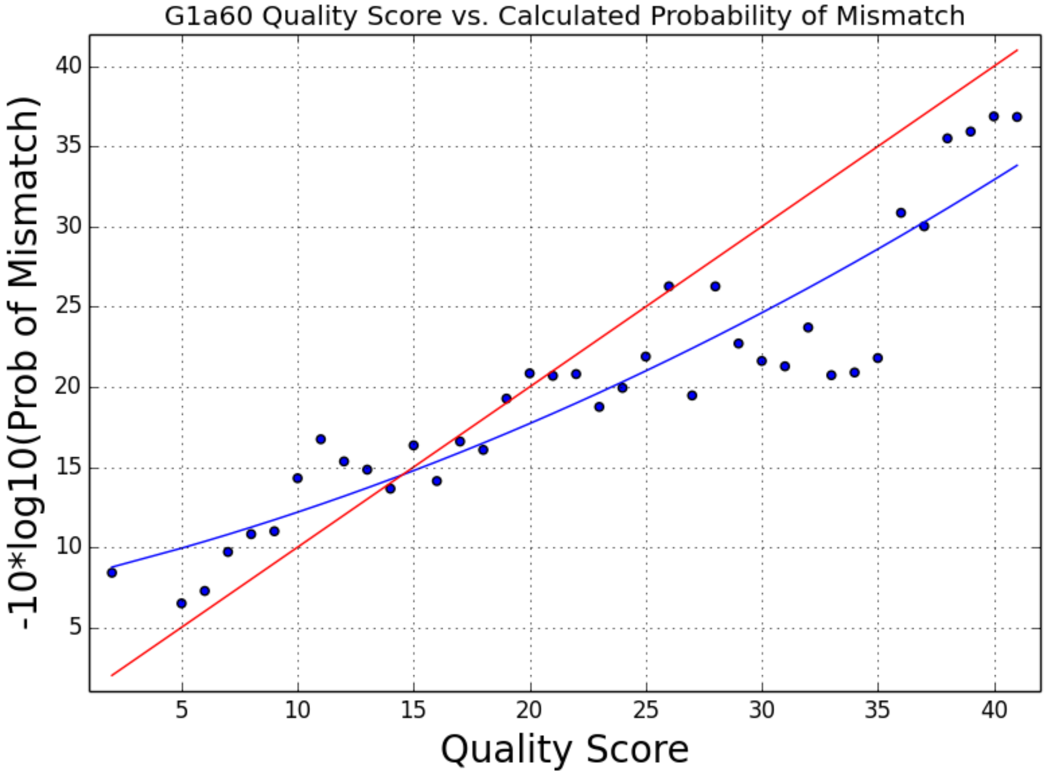

Figure 3: Best fit line for results of estimateErrorProbs.py on G1a60 data. R^2=0.852710185136.

Miscellaneous Remarks:

Python Multiprocessing

The hardest, most devious issue I encountered is that the multiprocessing module does not play well with pysam Samfile objects, presumably because of the C bindings that are abstracted by them. When I would pass Samfile objects as an argument for the method that is passed into pool.apply_async() or pool.map(), the child process would terminate without warning upon the first access of a contained aligned_read object, without any indication of a problem. The way to work around this is to NEVER pass a pre-existing Samfile object (or presumably, any other object that is a wrapper for C data structures) as an argument into pool.apply_async, map, etc. My workaround for this was to pass in the filename of the samfiles to each process, have each process open its own Samfile with pysam.Samfile(filename), and then have them process and pass back the results as appropriate.

Python2/3 compatibility

What I have done so far to allow my code to work seamlessly on python 2 and 3:

1. Change all print statements to print functions (add parens) and add

```
from __future__ import print_function
```

2. Python 2 uses itertools.izip but in python 3, zip() returns a view (like an iterator) so itertools.izip is gone. The solution is to try to import itertools.izip as izip, but except the importerror (in the case of python3) in which case just define izip as zip. This is implemented in my file compatibility.py, so you can just use

```
from compatibility import izip
```

and replace all occurences of itertools.izip with izip
3. Change all instances of someiterator.next() to next(someiterator)
4. If you use the method dict.iteritems(), this is gone in python3 because dict.items() does the same thing (returns view, like iterator). My solution was to make compatibility_dict, a subclass of collections.defaultdict which defines items() as iteritems() if the version is python 2 or as items() if the version is python 3. To use this, simply add

```
from compatibility import compatibility_dict
```

and replace any instance of collections.defaultdict with compatibility_dict. If one wanted, the exact same thing could be done to make a type of compatibility_dict that is a subclass of the basic dict rather than defaultdict.
5. In python 2, pysam alignedreads have a qual that is a string, whereas in python 3 it is a bytearray. My solution was to explicitly cast alignedread.qual to a byte array before use ensuring that whether the version was python 2 or 3, I could just treat it as a bytearray (indexing in returns 70 instead of 'F' if the quality is F in that spot, same as if ord() was called)