QAA

Grace Hach

2024-09-09

QAA

This work uses demultiplexed mRNA sequencing data gathered by the 2017 cohort. In particular, this focuses on two samples with paricular barcodes: 15_3C_mbnl, and 17_3E_fox.

Part One

The first part of this work entails using the program FastQC to get a sense of the overall quality of the raw sequencing data, and comparing these results to a custom python script. The results can be seen below in figure one.

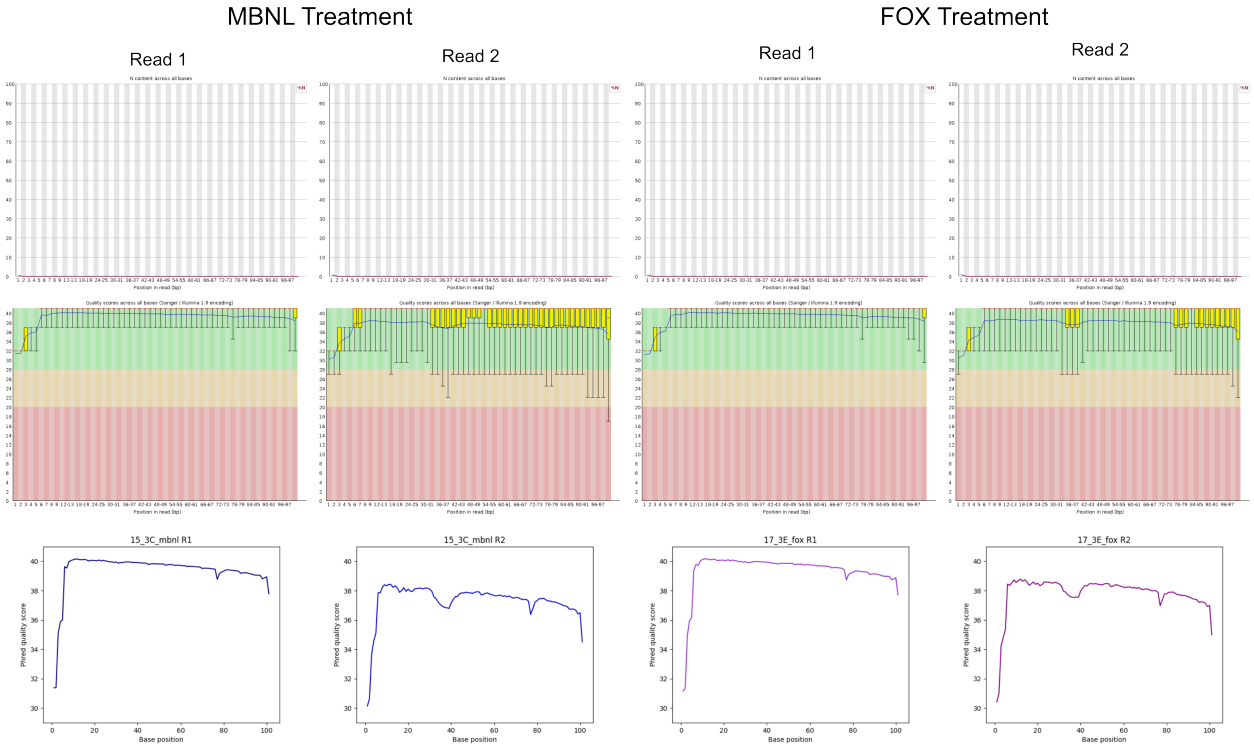

Figure One
Top Row: FastQC-generated plots of undetermined base calls as a function of base position
Middle Row: FastQC-generated plots of quality scores as a function of base position
Final Row: Python plots of quality scores as a function of base position

While the quality scores are somewhat variable, both across reads and across read positions, the N content is very small at all base positions. It is noticeably higher at the smallest base positions, it is somewhat difficult to read and fails to capture how the quality scores decrease at the higher base positions, indicating that the N content is a poor proxy for overall quality.

The FastQC plots are very informative, although they cannot be customized. I preferred to manually scale my axes to directly compare several data sets over a more narrow range. However, the FastQC plots are far more informative, because they contain the median (red bar), 25th and 75th percentiles (yellow boxes) and 10th and 90th percentiles (whiskers). I originally plotted maxes and mins along with mean scores in my demux project, and combined the individual plots into a figure with multiple subplots, but this took a staggering 36 hours to run, so the plotting here is paired down by necessity.

Even using the simplest plotting script I could think of (albeit without iteratively optimizing), each individual plot took about 300 to 500 seconds to generate. FastQC also took just over 100 seconds to run, and each run processed one entire library. The peak memory utilization for the python script was on the order of 60,000 kb. For FastQC, this number was closer to 180,000 kb. FastQC is far, far more efficient at generating a dense and informative output, likely because it is written in Java, which generally runs much faster than python. FastQC has also likely been extensively optimized by its creators, and may be especially suited to run on a HPC.

While the data are, on the whole, fairly high quality, FastQC raises one major flag, which is that much of the data appear to be duplicated reads. The output gives both a distribution of duplicate numbers, and an estimate of how much of the dataset would remain if de-duplicated. On the low end, Read two of library 15 3C HBML would be reduced to about 67% of its original size. On the high end, Read one of library 17 3E FOX would be reduced to just 49% of its original size. It is therefore recommended to de-duplicate the data before further analysis.

Part Two

The FastQC output has a plot of adapter content as a function of base position. This indicates that the adapters used here are Illumina universal adapters, and they only found toward the end of the sequences. This makes sense, as the adapters are only sequenced when the inserts are too short and the read runs over into the adapter region. I confirmed the adapters by grepping the exact sequences, and found them almost exclusively at the very end.

While I didn't find the adapters with bash, It could be done using the following command, which starts by zcat-ing the entire file to standard out, then extracts the sequence lines only. It uses the substring function within awk to grab the last 33 characters of each line. These are sorted, and then filtered for unique sequences, with the counts preceding them. The second call to sort sorts them according to their numerical order, in descending order. This gives the most common last 33 characters in each file, which are the adapters.

```
zcat <filename> | sed -n '2~4'p | awk '{ print substr($0, length($0)-32) }'| sort
| uniq -c | sort -nr | head
```

Initially, I expected the two files (R1 and R2) for each library to be trimmed the same way. Since both reads come from one insert, the adapter sequence should be present in the same number of bases on each side. However, the reads are also quality trimmed here, and in both libraries, the quality of read two is far lower than read one, as seen in figure one. Thus, in both libraries, read two is trimmed much more aggressively. The length distributions of both reads after quality and adapter trimming can be seen in figure two.

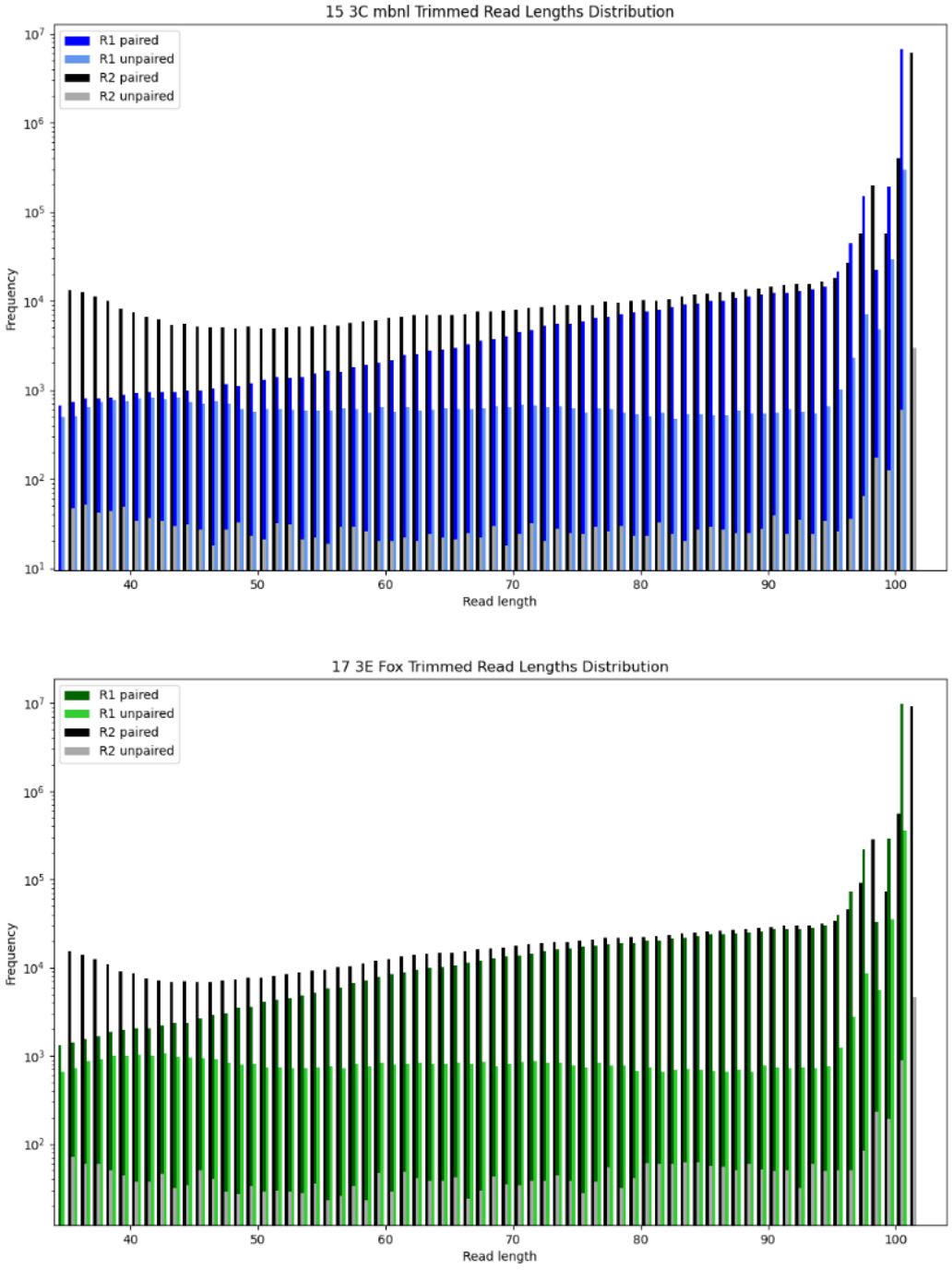

Figure Two
Length distribution of reads from both libraries after adapter and quality trimming.

The overall much higher rate of trimming in read two is also reflected in the greater number of unpaired read ones. The unpaired reads are created when the a read itself has decent quality scores, but its mate has such a poor overall quality that it is discarded.

Part Three

The results of aligning the remaining paired reads to a mouse reference genome are summarized in the table below.

	15 3C MBL	17 3E FOX
Number of mapped reads	14,436,372	21,532,811
Percentage mapped	98.6%	97.8%
Number of unmapped reads	207,582	483,272
Percentage unmapped	1.42%	2.20%

To determine the strand specificity, I ran htseq-count in paired end mode with both forward and reverse orientations. Based on these results, both libraries used a stranded prep. When run with argument `--stranded=yes`, virtually all reads don't map to a feature. When run with `--stranded=reverse`, virtually all of them do. For a tabular summary of the unmapped reads given the two different htseq-count parameters, see below.

	17 3E Fox		15 3C MBNL	
	stranded=yes	stranded=reverse	stranded=yes	stranded=reverse
___no_feature	**87.13%**	**9.75%**	**89.14%**	**8.10%**
___ambiguous	0.09%	1.52%	0.08%	1.63%
___too_low_aQual	0.15%	0.15%	0.19%	0.19%
___not_aligned	4.14%	4.14%	2.60%	2.60%
___alignment_not_unique	4.80%	4.80%	4.38%	4.38%