Python pipeline

Assignment

Story:

Dr. Harrington noticed a strange mold growing out of the ears on 50 of his patients. All of the mold looked identical except for the color, which ranged from black, orange, yellow and green. A swab test on all the patients showed they were infected with the same fungal species called D. gorgon.

Dr. Harrington sent the samples to a new sequencing company, Hawkins Laboratory, that offered to sequence it for free. Within a week the lab finished the sequencing and hands over the data as a single fastq file.

Dr. Harrington hands the data to you, the trustworthy hardworking unpaid intern, to build a pipeline to analyze the data and identify the mutations that resulted in the different color molds.

Inputs (found inside /home/rbif/week6/):

dgorgon_reference.fa - The wildtype reference sequence of D.gorgon

harrington_clinical_data.txt - The clinical data of the samples including a unique patient name, the color of the mold, and a sequence barcode*.

hawkins_pooled_sequences.fastq - The fastq data that contains the sequences of all the samples pooled together.

*The barcode is the first 5 basepairs of a sequence. It is a unique identifier that determines which sample that read belongs to. So if Tim's barcode was ATCCC then every sequence that begins with ATCCC is a read assigned to Tim.

There is a visual representation attached: Visual_pipeline_representation.pdf

Steps. Anything highlighted in yellow means I will provide you a python function to use and incorporate into your script.

You will not be able to use functions from BioPython for this assignment.

1. Demultiplex the pooled fastq - Create a single python script that reads in the pooled fastq* and outputs 50 new fastq files that contain only sequences belonging to that sample. This means using the barcode from the clinical data, finding the match at the start of a given read in the pooled fastq, and writing the read to its respective file. It should at the same time perform the steps **1A** and **1B** below.
script is /home/rbif/week6/necessary_scripts/parseFastq.py You just need to copy the (class) function into your own script and use the function to help you trim.

1A. Trim the beginning of the reads - Write a python function that removes the barcode and it's associated quality scores from the read.

1B. Trim the ends of the reads - Write a python function that removes the ends of the reads based on the following condition: The tail ends of the reads have degraded in quality and must be trimmed before being mapped. If the end of the read has at least two consecutive quality scores of **D** or **F**, then remove the rest of the read.

Example:

From

```
@arbitrary-seq-111
ATCGATGCACCCC
+
IIIDIIIIIDDFF
```

To

```
@arbitrary-seq-111
ATCGATGCA
+
IIIDIIIII
```

****This is any combination of D and/or F****
Everything in red would be trimmed off

```
IIIIIFDFF
Or

IIFFDII

Or
IIFFIIII
```

Final output of Step 1: 50 fastq files in a folder called **fastqs**. Each fastq inside of that directory should have the following format:

{name}_trimmed.fastq (ie tim_trimmed.fastq)

2. Perform alignment on each FASTQ to the reference sequence. Write a python function that calls the bwa command. This will transform the FASTQs to **samfiles**.
***Hint* Use the python packages os or subprocess to call a command line tool.**
Note In order to use bwa, the reference file must first be indexed. This can be accomplished with `bwa index ref.fa`
Then you can use `bwa mem ref.fa {name}_trimmed.fastq > {name}.sam`
to perform the alignment. This is taken from http://bio-bwa.sourceforge.net/bwa.shtml

The mapping_to_reference.pdf shows reads that are aligning to the reference. Bwa tries to find the best match for each individual read and maps it to a known reference.

3. Convert the samfiles to bamfiles - Use python to wrap the below commands:
Use samtools view to convert the SAM file into a BAM file. BAM is the binary format corresponding to the SAM text format. Run:
`samtools view -bS {name}.sam > {name}.bam`
Use samtools sort to convert the BAM file to a sorted BAM file.
~~`samtools sort {name}.bam {name}.sorted`~~
`samtools sort -o {name}.sorted.bam {name}.bam`

Then we need to index the sorted bam.
`samtools index {name}.sorted.bam`
We now have a sorted BAM file called {name}.sorted.bam. Sorted BAM is a useful format because the alignments are (a) compressed, which is convenient for long-term storage, and (b) sorted, which is convenient for variant discovery.
Taken from:
http://bowtie-bio.sourceforge.net/bowtie2/manual.shtml#getting-started-with-bowtie-2-lambda-phage-example

Once the sorted.bam files are created, remove the prior .sam files and the .bam files.

4. Variant Discovery - Use the python pysam pileup function to discover variants in each sorted bam file.
The script is located /home/rbif/week6/necessary_scripts/getMutations.py
Copy the function, pileup, to your own script. **This needs to be modified**. Details on where to modify it are inside the script.
You are using this script to find what position along the reference has a mutation.

In this case we are looking for a SNP (single nucleotide polymorphism). If a position contains DNA basepairs other than the wildtype, then we can consider it as a SNP and should be reported.
**You can consider the reference strain to be wildtype **

5. Create a report - Using python, create a report that outputs what nucleotide position and mutation is responsible for each color of the mold. Also print out the number of sequences that were used for each sample.
Example:
The green mold was caused by a mutation in position 23. The wildtype base was A and the mutation was C.
The black mold…

Sample Tim had a green mold, 320 reads, and had 32% of the reads at position 23 had the mutation C.
Sample Kristen ...

Final Deliverables.

A single python script with comments that performs all of the above steps called **pipeline.py**

A folder called **fastqs** that contains the demultiplexed fastqs for each sample.

A folder called **bams** that contains the **sorted.bam** file for each sample.

A text file called **report.txt** that looks like step 5.

A **readme** that tells the user how to use the script.