**Project Documentation**

The high level goal of my project is to annotate genomic data, with data from different databases, and to do it fast enough to keep up with the AI model training speed. The required runtime is around 100,000 samples per second.

The first step of the project was to annotate information about genes. The goal was to determine for each fragment whether it falls within a gene or not, based on a database of human genes. My first approach was to create a sorted dataset of all the human genes and then use a format of binary search on a given fragment. The runtime was around 40sec per 100,000 samples. My second approach was to preprocess the genes database and create an index that maps each coordinate in the genome. On the index, each coordinate is marked as 0/1 (within a gene or not). Therefore, at the beginning of the run, we have to load the index into memory and then we can retrieve data from it in O(1). The runtime with this approach is around 0.3sec per 100,000 samples so it seems to be a good solution for now.

The current stage of the project is to verify the correctness of the solution mentioned above, by comparing statistics of "inside-gene/outside-gene" between my algorithm and a dedicated bioinformatic tool which maps reads to genes.

The next steps of the project which Artem mentioned are related to assimilating additional information into the model:

* Information about methylation.
* Information about the cell type.
* Information about spatial DNA interactions
* Information about the twist of the samples.

1. Given a sample (chromosome, start position, end position), return whether it is within a gene or not.

First attempt- I created a sorted dataset of all the hg37 genome (sorted by: chromosome and start position). I used pandas to filter it from the whole genome file. Than, I used binary search on the given interval. It wasn't fast enough- took ~ 40sec per 100,000 samples.

Second attempt- preprocessing the genes database so later we can retrieve the relevant data in O(1).   
So I created an index (using json package) that maps each chromosome to a bitarray of it's length (based on the lengths Artem sent me), where each bit represents a coordinate in the genome and marked as 1 if it is inside a gene and 0 if not. Then given a sample check the index to see if there are 1's in it. The initial index was 19GB so I compressed it using zlib package (lossless compression) to only ~620KB (Memory efficient).

Then I added support on strand specificity to input format of : (chromosome, start position, end position, BAM flag)

The way to extract strand information from BAM flag:bitwise AND based on the information:

BAM flag: 0x10 SEQ being reverse complemented

<https://davetang.org/muse/2014/03/06/understanding-bam-flags/>

With this approach the **running time of 100,000 was ~0.3sec.**

**Tradeoff between running time and space complexity.**

1. Check the correctness of the first step.

Statistics using pysam + my algorithm:

* Total 77678 reads

204 reads unmapped

18328 reads inside genes (from mapped reads)

23.59% from all reads

Statistics using bedtools + my algorithm:

* Total 77838 read

18328 reads inside genes (from mapped reads)

Statistics from bedtools.intersect:

* Total 77838 read

18327 reads inside genes (from mapped reads)

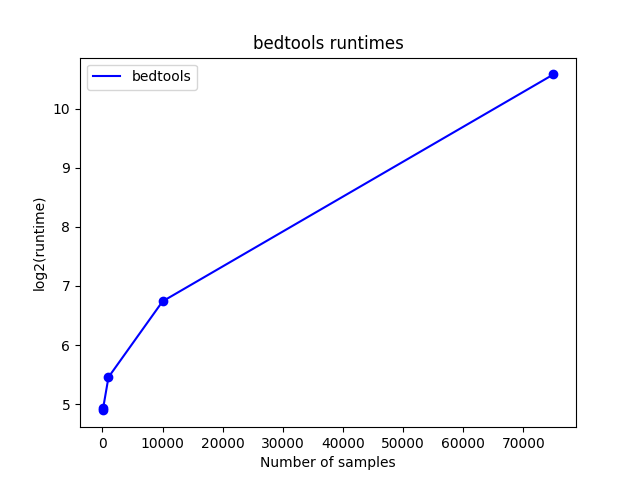
Results:

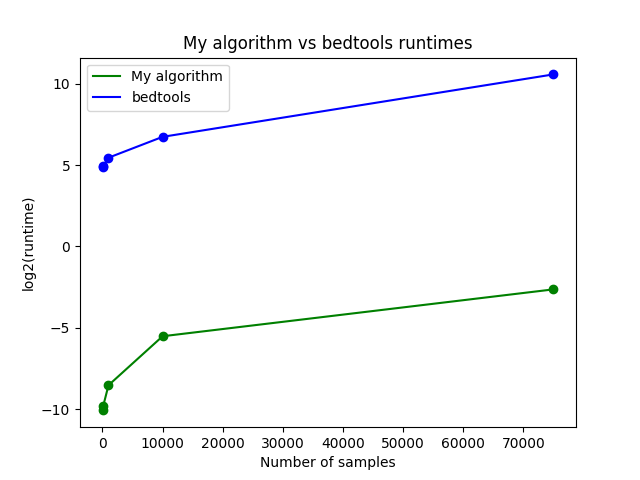
* My function returns True for all the reads which were returned in the intersection by bedtools.intersect method.
* There is a single sample for which my function returns True but wasn't appear in the intersection found by the intersect method. So I investigated this sample - (1, 171086959,171087109) and found in the genes file that there is a gene on chromosome 1 forward strand (+) from position 171060018 to position 171086959. That is the last coordinate of the sample is actually within the gene.

Runtimes:

|  |  |  |
| --- | --- | --- |
| # samples | My algorithm | bedtools |
| 10 | 0.0009 | 29.7574 |
| 100 | 0.0010 | 30.4373 |
| 1000 | 0.0027 | 43.9430 |
| 10000 | 0.0218 | 106.9054 |
| 75000 | 0.1608 | 1528.7368 |

תמונה שמכילה טבלה

התיאור נוצר באופן אוטומטיTime performances of match gene plots:



1. Given a sample, return a list with 0/1 for each type of classification.

There are 33 different types of segments in the GFF file:  
*{'RNA', 'snoRNA\_gene', 'lincRNA', 'VD\_gene\_segment', 'NMD\_transcript\_variant', 'exon', 'miRNA', 'biological\_region', 'mRNA', 'snRNA', 'rRNA\_gene', 'miRNA\_gene', 'gene', 'rRNA', 'nc\_primary\_transcript', 'CDS', 'snoRNA', 'processed\_pseudogene', 'V\_gene\_segment', 'three\_prime\_UTR', 'mt\_gene', 'processed\_transcript', 'pseudogene', 'snRNA\_gene', 'lincRNA\_gene', 'pseudogenic\_transcript', 'J\_gene\_segment', 'supercontig', 'C\_gene\_segment', 'aberrant\_processed\_transcript', 'chromosome', 'transcript', 'five\_prime\_UTR'}*

After removing 'chromosome ' we left with 32.

-First try: create 32 different indexes as before (a specific one for each classification) and use the same mechanism. Failed due to too big memory requirements (the program crashed after loading 5 indexes).

-Second try:

Step 1:

Create a DB that stores for each nucleotide on each strand a 32-bit array. Each bit in this array represents a classification (there are 32 different classifications in the GFF file) and is marked with 0/1 based on its classifications from the GFF file. The problem is that this DB requires ~25GB:

3101804739 \* 2 \* 32 = 1.98\*10^11 bits = 2.48\*10^10 bytes = 24.8GB

(The genome's length is 3101804739, then multiply by 2 (for each strand) then multiply by 32 (for each classification)).

So, creating a file and loading it to Python won't work with this large file. I consulted my brother who is a Data Engineer, and he thinks that a key-value store can work here, specifically, he mentioned Redis DB.

Step 2:

On runtime, the algorithm will break each sample into its nucleotides and query each of those against the Redis DB, then run bitwise OR to determine which of the classifications are active in the sample.

I tried an initial Redis setup but unfortunately, it wasn't fast enough (It took 2 hours to upload only ~7,000,000 keys to it (out of ~6,000,000,000 nucleotides in total) and was very slow when querying it.

So before  I delve into trying to improve it I wanted to check some other options.

-Third try:

I calculated how many informative nucleotides we have in the GFF file, (by uninformative I mean a nucleotide that doesn't appear in any segment in the GFF file) and I found out there are only 1670845956 from a total of

6203609478 nucleotides (stranded nucleotides- some segments miss the strand information. If we count also the unstranded nucleotides we have 1687920575 informative). That is only 26.93% (or 27.2%) from the genome.

 I realized that saving for each informative nucleotide a Python int which is represented by 32 bits instead of a 32 bits bitarray is more efficient in terms of memory (32 bytes instead of 84 using sys.getsizeof). The logic will be the same- add powers of 2, when each unique power will represent a classification, and then go over all the nucleotides within a sample and calculate bitwise OR between them to get the list of 0's and 1's.

In this case, we'll need to create a dict where keys are (chromosome, strand, position) and values are int between 0 and 2\*\*31. We have 1670845956 keys so the memory requirements are ~   1670845956 \* 32 = 53467070592 bits = 6.68GB.

That dictionary returns 0 for keys that don't exist.  
Much better but my laptop still can't support it.

I wanted to check the running time of the algorithm assuming it is possible to save such a dictionary on the cluster, so I generated a dictionary of size 10,000,000 and ran it on a ~75,000 sample file which I have from before. The approximate runtime of 100,000 on my computer will be around 15 seconds.  
Then,  I ran the algorithm with OR between only the first and last nucleotide of each read and the approximate runtime of 100,000 on my computer will be around 1 second on my computer (different runs: 0.76, 1.21, 1.10, 0.83, 0.78).

-Forth try: Remove uninformative classifications- classifications that has the same value for >99% of locations. I prepared the next table: (with each classification and the num of active nucleotides, percent of active nucleotides (from the sum of the lengths of the two strands), the longest interval, the shortest interval)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Classification** | **Number of active nucleotides** | **Percent of active nucleotides** | **Shortest interval** | **Longest Interval** |
| RNA | 427686 | 0.007 | 62 | 518 |
| snoRNA\_gene | 160694 | 0.003 | 43 | 330 |
| lincRNA | 200949792 | 3.239 | 87 | 1375317 |
| VD\_gene\_segment | 851 | 0 | 11 | 37 |
| NMD\_transcript\_variant | 330829101 | 5.333 | 207 | 1619643 |
| exon | 124285442 | 2.003 | 1 | 91667 |
| miRNA | 281018 | 0.005 | 41 | 195 |
| biological\_region | 25036992 | 0.404 | 1 | 57607 |
| mRNA | 1244771298 | 20.065 | 59 | 2304638 |
| snRNA | 209891 | 0.003 | 39 | 230 |
| rRNA\_gene | 67049 | 0.001 | 35 | 5070 |
| miRNA\_gene | 275331 | 0.004 | 41 | 195 |
| gene | 1403330149 | 22.621 | 8 | 2304638 |
| rRNA | 67538 | 0.001 | 35 | 5070 |
| nc\_primary\_transcript | 166100 | 0.003 | 474 | 50817 |
| CDS | 35482252 | 0.572 | 1 | 21693 |
| snoRNA | 169059 | 0.003 | 43 | 420 |
| processed\_pseudogene | 9718475 | 0.157 | 23 | 259171 |
| V\_gene\_segment | 56974 | 0.001 | 429 | 1845 |
| three\_prime\_UTR | 39167441 | 0.631 | 1 | 22552 |
| mt\_gene | 1507 | 0 | 59 | 75 |
| processed\_transcript | 460085408 | 7.416 | 101 | 1548835 |
| pseudogene | 50811324 | 0.819 | 23 | 586570 |
| snRNA\_gene | 208899 | 0.003 | 39 | 230 |
| lincRNA\_gene | 194862911 | 3.141 | 87 | 1375317 |
| pseudogenic\_transcript | 12123795 | 0.195 | 63 | 249600 |
| J\_gene\_segment | 4366 | 0 | 46 | 69 |
| supercontig | 0 | 0 | 4262 | 547496 |
| C\_gene\_segment | 59624 | 0.001 | 306 | 9993 |
| aberrant\_processed\_transcript | 232241164 | 3.744 | 3 | 1292210 |
| transcript | 129027749 | 2.08 | 8 | 874666 |
| five\_prime\_UTR | 10855752 | 0.175 | 1 | 14960 |

From these results we thought to remove only 5 classifications because all others look either common enough to be important, or they have biological significance. Therefore, we decided to keep all 32 classifications.  
Then I started to work on the cluster. Because in practice, Python assigns each int 32 bytes, not bits, that means that the whole DB will require ~200GB which is very bad.

Fourth try:using Numpy

Building the DB as a single numpy array, in which the chromosomes are chained one after the other (based on the partial sums, first I save all the + strand and after the – strand). The values data type is np.uintc32. Same as before, each nucleotide has a value between 0 to 2\*\*32-1 (the idea is the same as before: each bit represents a classification). To run the function, we first need to load to the program this DB, and then the function receives as an input (DB, chromosome, start position, end position, BAM flag), runs a bitwise OR between all the nucleotides within the sample and return a list of 0's and 1's.  
The compressed file size is 0.0234516GB and the RAM when loading it into the program is 24.8GB.  
Gh37 has 32 classifications. Then we want to run the analysis on Gh38. This reference has 25 classifications so the function returns a 25-long list. The classifications order:  
['scaffold', 'pseudogene', 'lnc\_RNA', 'ncRNA', 'unconfirmed\_transcript', 'V\_gene\_segment', 'biological\_region', 'snRNA', 'D\_gene\_segment', 'five\_prime\_UTR',  
'pseudogenic\_transcript', 'gene', 'mRNA', 'scRNA', 'snoRNA', 'tRNA', 'J\_gene\_segment', 'ncRNA\_gene', 'exon', 'rRNA', 'miRNA', 'three\_prime\_UTR', 'transcript',  
'C\_gene\_segment', 'CDS']

1. Given a sample, return a list with 0/1 for each type of regulatory region.

I did the same as the classification using numpy. We build the DB based on the chromosomes lengths using partial sums, and then each nucleotide has a value between 0 to 2\*\*32-1 (the idea is the same as before: each bit represents a regulation type). In Gh38 we have 5 types of regulations so the function returns a 5-long list of 0's / 1's. The regulations order is:  
['CTCF\_binding\_site', 'promoter', 'open\_chromatin\_region', 'TF\_binding\_site', 'enhancer']

1. Given a sample, return whether it is methylated.
2. Given a sample, return it's cell type (from ~150 cell types DB)
3. Annotate spatial interactions information of a sample (for example, what other intervals a sample interact with)
4. Given a sample, return how it is twisted (based on DB of known twisting forms)