Specification

- Program name: missingMotif.ipynb
- Input/Output: Input filename: 1st required parameter to main(), Output STDOUT

options:

- --minMotif minimum motif size to evaluate (int)
- --maxMotif maximum motif size to evaluate (int)
- --cutoff Z-score cutoff (float)

Ex: main("input.fa",["--minMotif=3", "--maxMotif=8", "--cutoff=-4."])

Background

Many prokaryotic genomes encode Restriction-Modification systems (RM) that act to protect the cell from invading DNA by cutting at specific sites (frequently short 4-6 base reverse complement palindromes). They also protect host DNA from being cut by modifying any sites within the host DNA that would be sites for the cutting enzyme. It is not unusual to find that these enzymes are adjacent to each other in the host genome. (see: https://en.wikipedia.org/wiki/Restriction_modification_system).

Meselson and Stahl (1958) showed that replication of the bacterial dsDNA genome happens in a semi-conservative fashion (see: https://en.wikipedia.org/wiki/Prokaryotic_DNA_replication). The resulting circular chromosome(s) would then be transiently hemi-methylated because the new daughter strand may not been have exposed to the protective methylation of the RM system before its cutting activity arrives. This creates a race between RM cutting and protecting activities may expose the cell to strand breaks.

Is a selective advantage incurred for the cell by allowing mutations to accumulate at these sites? If so, then would we expect the host genome to become depleted of RM sites over time? Can we use this depletion process as a signature for the existence of the specific RM systems present in the genome?

How can we find sequences that are missing in a genome?

Assignment

Using this notebook, write a proper style BME205 solution that reads a fasta file and ranks motifs based on how statistically underrepresented the specific motif is. We need only consider sequence motifs that are up to about 8 in length (this is a program option). In order to use the equation that we developed in class, the minimum size should be at least 3 (this is an option also). We will find that there are quite a few of these, so we need to specify a statistical cutoff. For this assignment, we will use Z-scores. The scores we are looking for will be negative since we are looking for underrepresented sequences (this cutoff is a program option too). Finally, we need to sort the output by motif size and then z-score within size groups and print it out.

Remember that sequences that we get from DNA sequencing are from only one strand of DNA. This means that when we see a AAGGTT, this implies that AACCTT (the reverse complement) is present on the other strand. We should count these sequences as equivalent.

Your report should be sorted by Zscore within motif-Size, such that the longest motifs are at the beginning of the report and, within motifs of a given size, print the most significant motifs first. Print the k-mer pairs in alpha order. (AAA:TTT and not TTT:AAA)

The extra credit for this assignment is particularly interesting. If you have time, this will present better results.

Here is an example (non extra credit):

In [ ]:

```
sequence: reverse     count      Expect Zscore
AAAATTTA:TAAATTTT       835     1737.66 -21.66
GATTAATA:TATTAATC       550     1326.89 -21.33
AATTAATA:TATTAATT       929     1839.72 -21.24
AATTAATC:GATTAATT       977     1861.79 -20.51
ATAATTAA:TTAATTAT       1033    1926.49 -20.36
GAAATTTA:TAAATTTC       378     1031.07 -20.34
CCGATCGC:GCGATCGG       1323    2284.74 -20.12
GCGATCGA:TCGATCGC       1221    2121.78 -19.56
ACGATCGC:GCGATCGT       1188    2035.31 -18.78
GTTTAAAA:TTTTAAAC       519     1151.17 -18.63
AAATATTA:TAATATTT       738     1444.79 -18.60
AAAATTTG:CAAATTTT       857     1591.95 -18.42
CATTAATC:GATTAATG       479     1037.49 -17.34
ATATATAA:TTATATAT       271     736.19  -17.15
ATTTAAAG:CTTTAAAT       419     941.16  -17.02
AAATATTG:CAATATTT       687     1287.74 -16.74
CAAATTTA:TAAATTTG       670     1265.24 -16.74
CTTTAAAC:GTTTAAAG       468     995.13  -16.71
```

Hints

- For **genomes that include multiple fasta records, do NOT concatenate them**. They should be reported as a single genome, though we do not want to create false kmers that would result from concatenation
- **Avoid zero divide errors!** Close examination of our null model will show that if we have a zero in the denominator, we will have a zero in the numerator. If you have pre-populated your count dictionary, then you will need to handle dividing by zero.
- Only consider kmers that are constructed from the **canonical bases {A, C, G, T}**.
- Z-score is calculated as: $\frac{s-\mu}{sd}$, where s is the specific count you are converting, and mean and standard deviation are given by μ and sd. Both μ and sd describe the null distribution. In our case, we can treat the expected distribution as a binomial where: $\mu = np, sd = \sqrt{np(1-p)}$. n = genome size (approximately), and p = probability of success, given the null distribution.
- c(x) defined as the count of x, generally the count of some substring. For example, for kmers of size 4: $K = k_1k_2k_3k_4, Pr(K) = \frac{c(K)}{N}, E(K) = \frac{c(k_1k_2k_3) \cdot c(k_2k_3k_4)}{c(k_2k_3)}$ (see below for the derivation of our null model) We can then calculate n and p, and derive μ and sd.
- To keep your scores lined up, use tabs to separate values in each line. The following format string will be of use:

print('{0:8}:{1:8}\t{2:0d}\t{3:0.2f}\t{4:0.2f}'.format(seq, rSeq, count,E,pVal))

Null Model derivation

If we assume complete independence of the any of the characters that make up a sequence K, we could approximate the expected count of K, E(K), as:

$$K = k_1k_2k_3k_4, Pr(k_i) = \frac{c(k_i)}{N}$$
$$E(K) = N * Pr(K) = N * Pr(k_1)Pr(k_2)Pr(k_3)Pr(k_4)$$

the above is a markov(0) approximation for the expected count (E) of our sequence K.

Rarely is this assumption of independence a good null model, due to many factors including codon bias which drives selection of 3-mers based on underlying amino acid frequencies. We can model this dependence using a markov model, where the probability associated with any symbol is conditioned on the preceding characters. For a kmer of size 4, we can condition our character probabilities using the preceding two bases without requiring use of the count of the specific 4-mer that we are trying to approximate(see note on information content). We need to consider edge conditions in our approximation. In general, we can use a model of order k-2, where k is the size of the kmer as long as we have sufficient data, therefore:

$$Pr(K) = \frac{E(K)}{N} = Pr(k_1)Pr(k_2|k_1)Pr(k_3|k_1k_2)Pr(k_4|k_2k_3)$$
$$= \frac{c(k_1)}{N}\frac{c(k_1k_2)}{c(k_1)}\frac{c(k_1k_2k_3)}{c(k_1k_2)}\frac{c(k_2k_3k_4)}{c(k_2k_3)}$$
$$= \frac{1}{N}\frac{c(k_1k_2k_3)c(k_2k_3k_4)}{c(k_2k_3)}$$

note: the conditional probability Pr(a | b) is read as the probability of finding sequence a, given sequence b. In this case we treat sequence b as adjacent and left of a, and ask what is the probability of finding sequence a, given that b is immediately to the left of a. This means that we should count the occurrences of string ba, among all strings that start with b (taken as the count of all sequence b).

note on information content: What would happen if we were trying to predict the frequencies of 4-mers, and our null model contained the frequency of 4-mers?

2021 Note: Make sure to review the module video on this. The derivation is a bit more elaborate.

Extra Credit

Among all kmers of size k, we have both the measured value and the expected value for that specific kMer. We are interested in knowing which of these sequences produce counts that are surprisingly small. If we were to normalize our actual counts by the expected count, we can then look to the smallest values,(<1) as those that evoke the greatest surprise. Throughout this assignment, we will refer to the actual, integer counts of a kMer as "counts". When we normalize an actual count by its expected value, we will refer to those as "normalized counts". We can then calculate a mean and standard deviation of those normalized counts within size-groups(sequence groups of the same size, for example, all of the 8-mers), to gain a quantitative measure of that surprise. We can think of this as the distribution of surprise within each size-group. kMers that yeild a normalized count near 1 are not very surprising, because the actual and expected count values are close to each other.

For a possible 5 extra points:

- add option **--kScoring** which evaluates to true only when the flag is specified by the user
- calculate Z-scores over the normalized score found for each kMer, within each size-group.
- use this Z-score for score cutoff
- include a column for the p-value associated with this Z-score.

p-Values can be calculated from the CDF of the binomial or Normal distribution. Feel free to use scipy for this. scipy.stats.norm.cdf(z). For those of you using an M1 Mac, make sure to use Anaconda

Notes

- I considered adding pseudo counts to this assignment but have deferred that to the next (4-Oct)
- Added the full derivation of the null model (4-Oct)
- clarified the use of E - the expected value, and in our case, an expected motif count. (4-Oct)
- clarified the use of the count function c(x) (4-Oct)
- clarified the use of counts of strings in conditional probabilities. (4-Oct)
- clarified the avoidance of zero divide or the use of kmers that contain non-canonical bases (4-Oct)
- **Extra Credit** To calculate a distribution over normalized counts for kmers in a particular size-group, we need to calculate the mean and sd of that size-group. This can be done by finding a normalized count of a kmer of a given size, then computing a running sum of those counts of that kmer-size, a running sum of the square of those normalized counts of that kmer-size, and the number of kmers we found of each kmer-size. These three statistics would need to be tabulated for each of the kmer-sizes that we are working with. So...lets think of these as three vectors that each contain kMax values. The three vectors contain kMax elements: $N_k, \sum normalizedCounts_k, \sum normalizedCounts^2{}_k$.

In python terms, we need:

- statsN[1..kMax],
- statsSum[1..kMax], and
- statsSum2[1..kMax].

Each of these vectors (lists) correspond to a tabulation N = the number of kmers of some specific size(k), the sum of the the normalized-counts of specific size(k), and sum2 = the sum of the squared normalized-counts of specific size(k). Now, we can calculate mean and SD for the group of kmers of size k by:

$$\mu_k = \frac{sum_k}{N_k}$$

$$SD_k = \sqrt{\frac{sum2_k}{N_k} - \frac{(sum_k)^2}{N_k^2}}$$

2018 additions

- binned the reverse complement sequences. This produces better Evalues and a more concise report. (1-Oct)

2019 additions

- added calculation of normalized kMer counts with significance determined by their distribution within groups of size k.

2020 addition

- its good to think about our kmer counting by asking how many names does this kmer have? For example, the kmer AAAA when viewed from the opposite strand is TTTT. This bit of data has 2 names, so when we are counting we make sure that counts of this kmer by either of its names goes into the same bin. How should we handle a reverse complement palindrome like TATA ? When viewed from either strand, it has the same name.

2021 addition:

- we are using notebooks now to formalize inspections and promote Python learning. Changed the expected input.

Inspection Intro

Provide design level information for your inspection team here.Examples:

- How did you count kmers of each size without having to read the data multiple times?
- How did you organize the computing of mean and SD for each of the kmer groups ?
- How did you compute Z-scores for each kMer in a kMer group?
- How did you sort the final report to include most significant Z-scores at the top of each kmer-size group?
- How did you organize the program ?

Inspection Results

- What did your inspection team find? Which of the inspection results did you use, how did you resolve them, and which did you decide against? Who was on your inspection team?

Code that may be useful

In [13]:

```python
import sys

class FastAreader:
    def __init__(self, fname=''):
        '''contructor: saves attribute fname '''

        self.fname = fname
        self.fileH = None

    def doOpen(self):
        if self.fname == '':
            return sys.stdin
        else:
            return open(self.fname)

    def readFasta(self):

        header = ''
        sequence = ''

        with self.doOpen() as self.fileH:

            header = ''
            sequence = ''

            # skip to first fasta header
            line = self.fileH.readline()
            while not line.startswith('>'):
                line = self.fileH.readline()
            header = line[1:].rstrip()

            for line in self.fileH:
                if line.startswith('>'):
                    yield header, sequence
                    header = line[1:].rstrip()
                    sequence = ''
                else:
                    sequence += ''.join(line.rstrip().split()).upper()

        yield header, sequence
```

In [16]:

```python
class CommandLine():
    '''
    Handle the command line, usage and help requests.

    CommandLine uses argparse, now standard in 2.7 and beyond.
    it implements a standard command line argument parser with various argument options,
    a standard usage and help, and an error termination mechanism do-usage_and_die.

    attributes:
    all arguments received from the commandline using .add_argument will be
    avalable within the .args attribute of object instantiated from CommandLine.
    For example, if myCommandLine is an object of the class, and requiredbool was
    set as an option using add_argument, then myCommandLine.args.requiredbool will
    name that option.

    '''

    def __init__(self, inOpts=None):
        '''
        CommandLine constructor.
        Implements a parser to interpret the command line argv string using argparse.
        '''

        import argparse
        self.parser = argparse.ArgumentParser(
            description='Program prolog - a brief description of what this thing does',
            epilog='Program epilog - some other stuff you feel compelled to say',
            add_help=True,  # default is True
            prefix_chars='-',
            usage='%(prog)s [options] -option1[default] <input >output'
        )

        self.parser.add_argument('-l', '--minMotif', nargs='?', default=1, action='store',
                                 help='min kMer size ')
        self.parser.add_argument('-m', '--maxMotif', nargs='?', default=8, action='store',
                                 help='max kMer size ')
        self.parser.add_argument('-c', '--cutoff', nargs='?', type=float, default=.01, action='store',
                                 help='Zscore cutoff')


        self.parser.add_argument('-v', '--version', action='version', version='%(prog)s 0.1')
        if inOpts is None:
            self.args = self.parser.parse_args()
        else:
            self.args = self.parser.parse_args(inOpts)
```

In [17]:

```python
class Genome:
    """ """
    def __init__(self, minK, maxK):
        """Construct the object to count and analyze kMer frequencies"""
        pass
```

In [19]:

```python
def main (inFile=None, options = None):
    ''' Setup necessary objects, read data and print the final report.'''
    cl = CommandLine(options) # setup the command line
    sourceReader = FastAreader(inFile) # setup the Fasta reader Object
    thisGenome = Genome(cl.args.minMotif, cl.args.maxMotif) # setup a Genome object

    ###
    # Give your Genome some data and tell it to do stuff
    ###


if __name__ == "__main__":
    main(inFile="input.fa",options=[ "--minMotif=3", "--maxMotif=8", "--cutoff=-4."])
```