Bayesian Statistics: Sequence Matching

Mitchell Kwock

Abstract

In Communications and Biology, matching two sequences with an unknown separation is a constant problem that always needs to be solved. In signals, there is often data that affects itself, dispersing its information in random and varied ways. In Biology, we find that DNA also has sequences that are important to understand the nature of. However, in order to get a good hold of the sequence's meaning, we have to know what the sequence is in the first place.

1. The Project

In order to obtain the readings of a strand of DNA, biologists use PCR machines to copy DNA en masse and use the wonderful process of forward and reverse reading of DNA strands. What these biologists get is something that looks like the following without the line up:

Forward: ATTACGGTAAATCGGATCCCCTGAA.....

Reverse: .....TCGGATCCCCTGAAAAAATCCGAGGATC

However, there are often errors in the reading due to the nature of using small enzymes to pick up on these strands. In reality, we see that as the strand is read, the values are more likely to become incorrect. They become less distinct, more random, and increasingly harder to work with. This is where this Bayesian Genome matcher comes in. By taking the two strands and applying Bayesian updates to each possibility, the program finds the most probable separation distances and attempts to reconstruct the strand.

2. Modeling the Genome

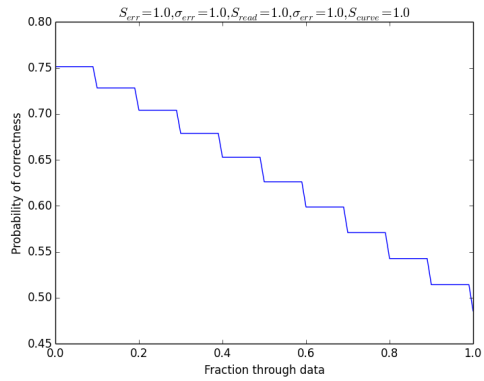
Each genome is made by putting together a sequence of about 10,000 base pairs (A, T, C, G) in equal weightings using a weighted choice function. These rates can be changed, but for these purposes, it will be kept to a uniform value. A forward and reverse read are then chosen between a random length and have a guaranteed area of overlap so the program has something to match. Error is modeled by the following algorithm:

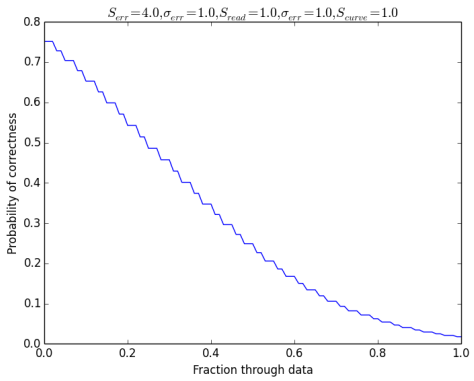
$$\mu_{err} = strength * portion^{shape}$$

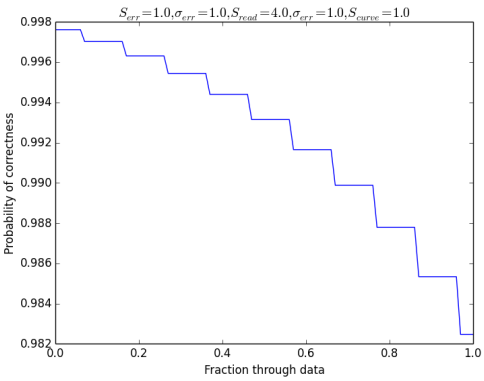
If a gaussian random with center μ_{error} is larger than a gaussian random of constant predictability: Return a random read (A, T, C, G of equal weights), otherwise: Read correctly.

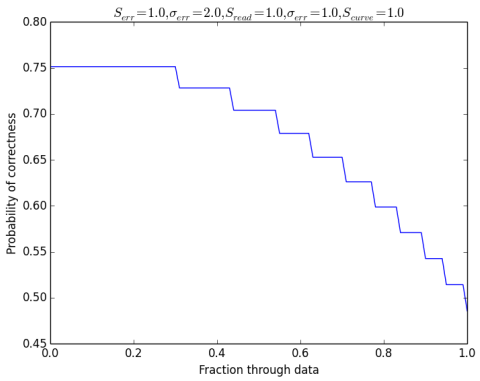
This model was chosen because it would be a straightforward way of making sure that the end conditions and the internal conditions were met:

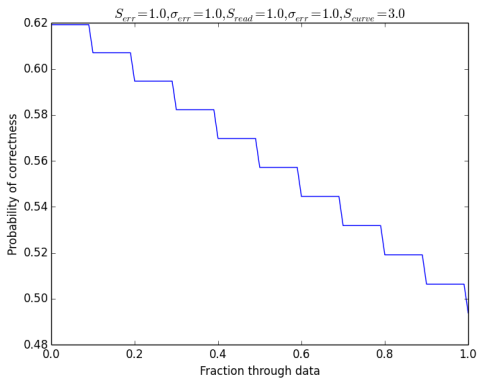
- Least wrong in the beginning
- Consistently erroring near the end
- $\frac{\delta_{Err}}{\delta_{Index}} >= 0$ for all Indices











3. Modeling the Bayesian Update

It'd be no fun if these were hand-matched! That's where Bayesian comes in. Given each base-pair, we find a lovely and frightening problem. Bayesian updates require us to know one thing to compare our hypotheses: the relative likeliness of the hypothesis based on the data. In this case, we need to find how likely a given sequence shift occurs if a base pair matches or doesn't match.

Hypothesis: The sequences are shifted N base-pairs

Data: At index I, the base pairs either match or don't match.

P(H) begins as a uniform distribution and is constantly updated while P(D H) is the hardest part!

In order to solve this, the problem must be split into two possibilities: they originally either match or don't match.

Given that the data matches, let's assume they're both AA. The forward read will have a p chance of being A while the reverse will have a k chance of being A. (This is the same no matter what the match)

Forward: Chances for each base

A	T	C	G
p	$\frac{1-p}{3}$	$\frac{1-p}{3}$	$\frac{1-p}{3}$

Reverse: Chances for each base

A	T	C	G
k	$\frac{1-k}{3}$	$\frac{1-k}{3}$	$\frac{1-k}{3}$

This means that the chances for being positively matched under these circumstances is the dot product of the vectors (That is, the chance that a AA is read plus to the chance that TT is read, and so on):

$$\text{Positive: } p*k+\frac{(1-p)(1-k)}{3} \quad \text{False Negative: } 1-[p*k+\frac{(1-p)(1-k)}{3}]$$

The other case occurs when the original data doesn't match. Perhaps the original bases with these shifts are A and T for forward and reverse respectively. How will this be factored into the final Bayesian update? Time to find out

Forward: Chances for each base

A	T	C	G
p	$\frac{1-p}{3}$	$\frac{1-p}{3}$	$\frac{1-p}{3}$

Reverse: Chances for each base

A	T	C	G
$\frac{1-k}{3}$	k	$\frac{1-k}{3}$	$\frac{1-k}{3}$

Here we get a less pretty, but still workable set of equations:

$$\text{False Positive: } \frac{p+k-2pk}{3} + \frac{2(1-p)(1-k)}{9} \text{ Negative: } 1 - [\frac{p+k-2pk}{3} + \frac{2(1-p)(1-k)}{9}]$$

Given this information, if we read a match, what is the likelihood that the hypothesis is correct? Well, it's the possibility that it was a match and a match was read divided by the chance that there was no match and there was a match read. That is:

$$\frac{Positive}{Positive+FalsePositive}$$

And lastly, if there is no match, what is the likelihood that the hypothesis is correct? The answer is, of course, based on the chance that it is falsely negative divided by the chance that it is read falsely:

$$\frac{FalseNegative}{Negative+FalseNegative}$$

Now we have the beginnings necessary to begin the code's Likelihood function and make it match sequences for us.

4. Execution

This is personally my favorite part of the project: actually getting something to work! Here is the python code described to fit the requirements back in *Modeling the Genome*

```
def Prob_True(portion):
    '''
    Given a certain portion through the sequence,
    returns the probability that a correct read is obtained.
    '''
        try:
                loss = s.Get('error_max') * portion**s.Get('curve_strength')
        except ValueError:
                loss = 0
        pmf_true = MakeNormalPmf(s.Get('read_strength'),
                    s.Get('sigma_correct'), 5, n=100)
        pmf_false = MakeNormalPmf(loss,
                    s.Get('sigma_wrong'), 5, n=100)
        return pmf_true > pmf_false
```

Just a quick peek at the Likelihood function that our thinkbayes2 library needs us to define:

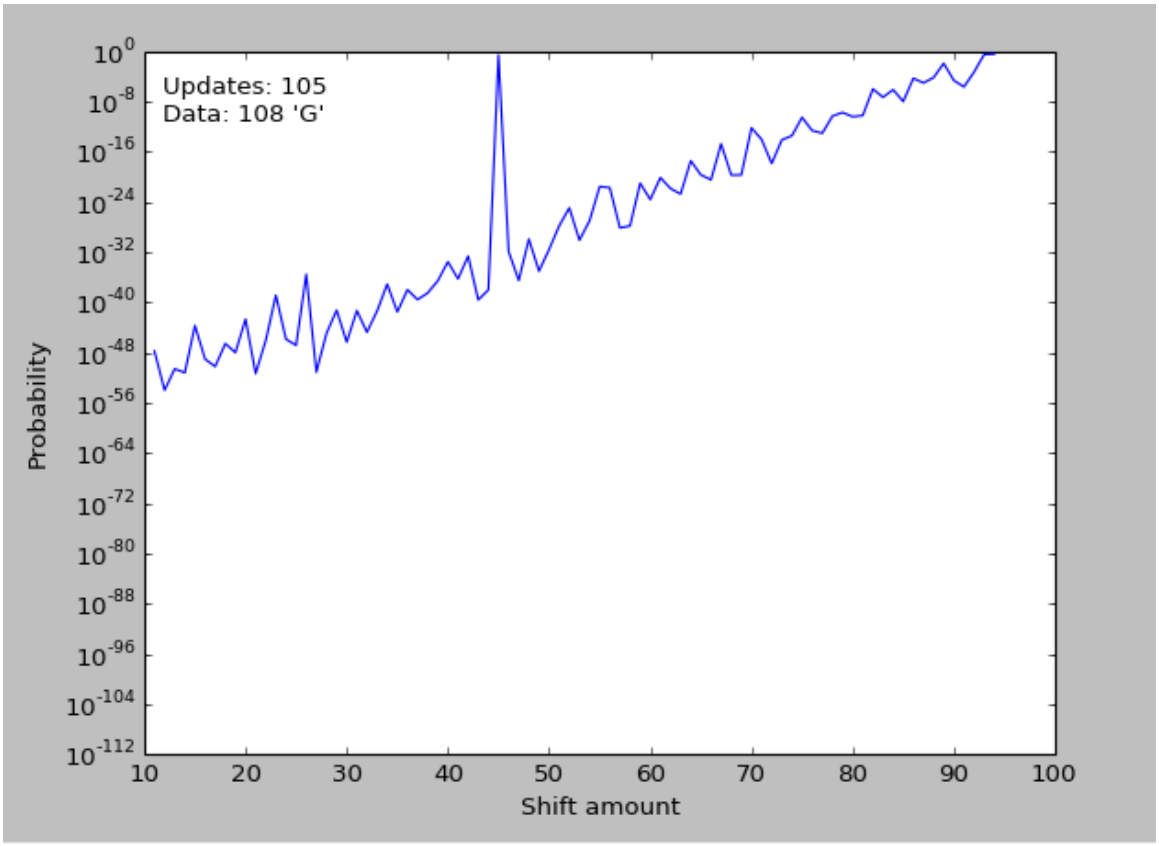
```
class Genome_Matcher(Suite):
        def Likelihood(self, data, hypo):
                '''
                hypo: Starting index of the intended string
                data: [Index, reverse_base] The read Base at Index

                Returned probability is the probability
                        that they were matched

                Needs to account for:
                Increasing error in forward string as
                        well as decreasing in reverse
                '''
                index = data[0]
                index_match = hypo + index

                # Deals with the no-match case
                if(index_match == self.forward_length):
                        # Most probably true at later values
                        return(1.0 * index / self.min_length)
                if(index_match > self.forward_length):
```

```
                return 1

        forward_base = self.Get_At(index_match)
        reverse_base = data[1]

        if(forward_base == reverse_base):
                # Match Found, determine chance of match
                #   accounting for False Positive

                # Accounts for random chance of reading true
                # Forward Probability of reading correctly

                f_portion = 1.0 * index_match / self.forward_length
                p = (1 + 3 * Prob_True(f_portion))/4
                # Reverse Probability of reading correctly
                r_portion = 1 - 1.0 * index_match / self.reverse_length
                k = (1 + 3 * Prob_True(r_portion))/4

                positive = p * k + (1-p)*(1-k)/3
                false_positive = (p + k - 2*p*k)/3 + (1-p)*(1-k) * 2/9
                return positive / (positive + false_positive)
        else:
                # Forward Probability of reading correctly
                f_portion = 1.0 * index_match / self.forward_length
                p = (1 + 3 * Prob_True(f_portion))/4
                # Reverse Probability of reading correctly
                r_portion = 1 - 1.0 * index_match / self.reverse_length
                k = (1 + 3 * Prob_True(r_portion))/4

                false_negative = 1 - (p * k + (1-p)*(1-k)/3)
                negative =          1 - ((p + k - 2*p*k)/3 +
                        (1-p)*(1-k) * 2/9)
                return false_negative / (negative + false_negative)
```

Notice that there is the case handling for the cases. The possibility that the index is too long is actually based on the chance of said data not being seen. The chance that the 100^{th} index in a sequence of 101 is not seen is actually very high, however, the chance of the 10^{th} index in a sequence in 101 is fairly low, lowering that hypothesis's chance of being correct.

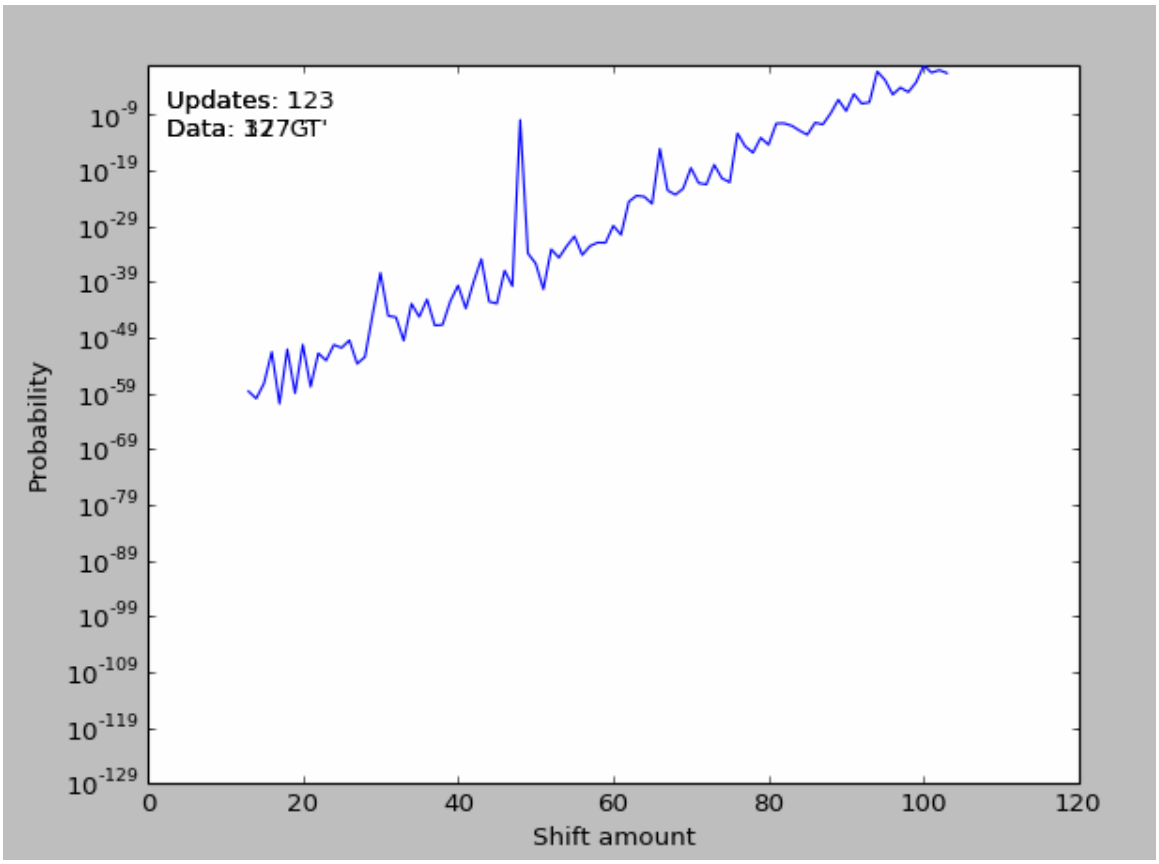
5. Results

The results are actually fairly useful and brings to light things that I did not expect!



By iterating through the data, we learn some other fun things about what Bayesian probability tells us: Lack of information can sometimes point us in a better direction. We see a single spike, the actual data, but the general idea is that with more wrong data, the end piece is more and more likely compared to everything else (Fewer mistakes by virtue of being at the end). If, for instance, there were sufficient incorrect matches, a later value will be more likely because there are fewer WRONG values

Let's look at another execution, this time executing falsely:



The correct shift is 44, as we can see in the above, but because there were a sufficient number of incorrect bases, it is actually much lower (it's *about* 10^{-10}) than the outputted "best" outcome of 100. Graphing this on a Log Scale gives us much more insight as to the outcome of the probability. Even though I know the outcome is wrong, the probabilities given to the engine are all correct. This tells me that based on the sheer unlikelihood of that many bases being correct outweighs the matches sufficiently.

6. Looking Forward

This was a fun exercise in Bayesian Statistics and a reminder to the the mind-blowing nature of Bayesian updates. The fact simply is, given the probabilities of things, coming later is more likely than the incorrectness of the various bases. I feel the need to optimize the code to pick out the *correct* shift more often, but the simple fact is, if I want to make the proper choice, I must believe in the math.

I would extend this code by adjusting the probability function to something more realistic and add different weightings to the different bases. Perhaps a "smart chooser" that implements a chooser that finds these relative peaks and return them as possible outcomes as well.