Optimising genes with a genetic algorithm

Introduction

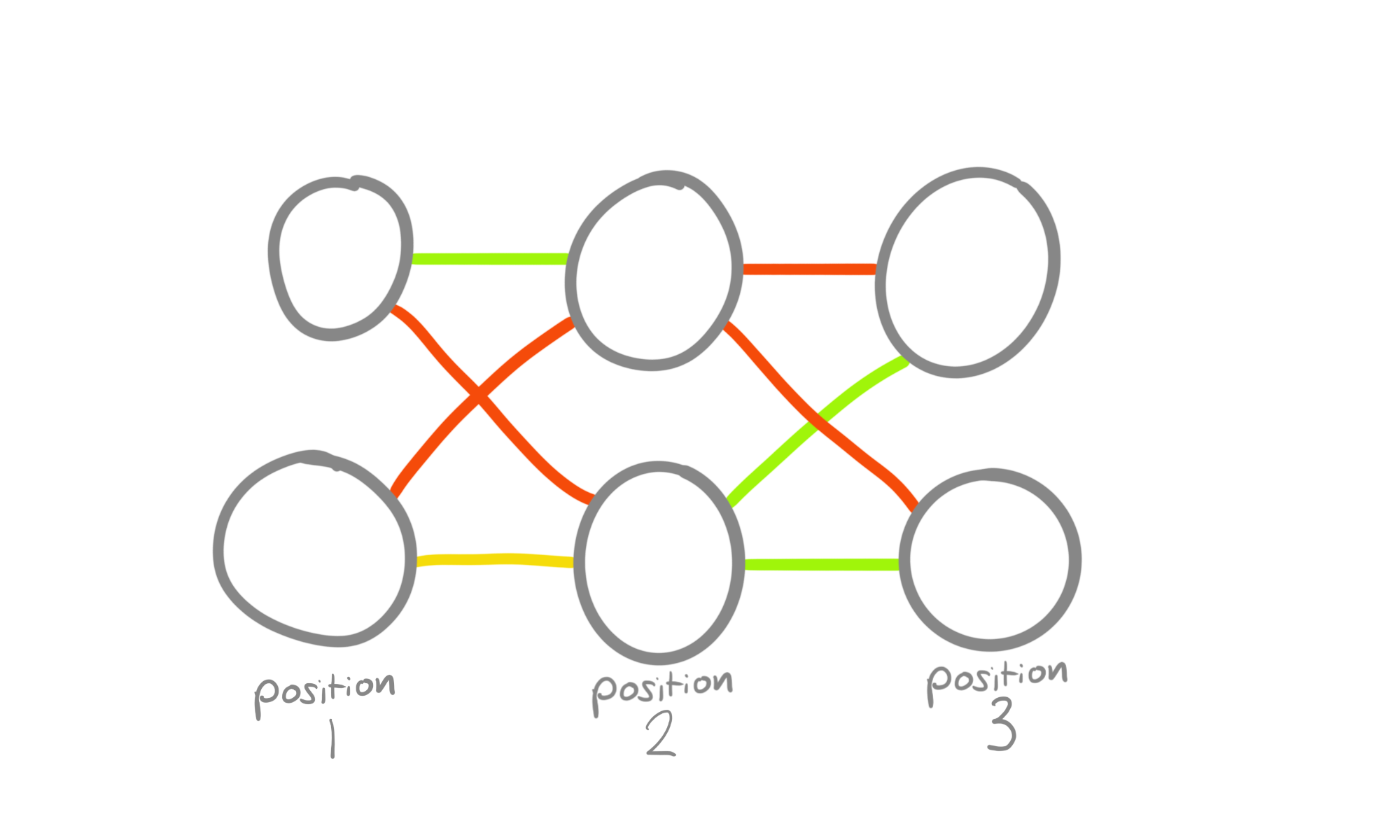
Genetic algorithms can be a great way to tackle an optimisation problem because they can reliably find a good solution, even in a complex fitness landscape. There are lost of great posts about how genetic algorithms work but this one is about how to actually use it. You just need a vague understanding of how a genetic algorithm works; they basically simulate a population where each individual is a possible “solution” and let survival of the fittest do its thing. The difficult part with using a genetic algorithm, and really this is one of the most important parts of data science in general, is how to frame the question so that the computer can answer it for you. In my case this question was how to optimise a gene and I’ll show you how to improve your solutions just by framing the question better.

All the code is available at https://github.com/DAWells/codon\_path

The problem

DNA codes for amino acids, the building blocks of proteins. Three DNA letters (A, T, G, or C) in a row code for a single amino acid and is called a *codon*. Several different codons encode the same amino acid. For example CAA and CAG both code for the amino acid Glutamine. Although several codons can do the same job, not all pairs work well together. These uncooperative pairs are far rarer than expected by chance whereas pairs that do work well are more common. These codon pairs differ in frequency even though they code for exactly the same amino acids. My aim was to use just these overly common codons to encode a given protein.

So why is this important? Optimising a gene in this way can allow you to get more of your desired protein. A protein you might need to catalyse a chemical reaction, or produce a therapeutic drug. Why is this difficult? Each codon is part of two pairs, this means you can’t select the best pairs individually. Instead you have to consider all pairs at once. A good codon pair may lock you in to a bad pair next as shown below.



The solution

The first thing you need for a genetic algorithm is a score function, some way to measure the fitness of possible solutions to your problem. This is used to decide which solutions get to “reproduce”. Deciding how to calculate this number is a vital step to any optimisation problem and is really a major part of data science. You have to understand the problem well enough to capture all the nuance in this one value. Fortunately for me, Coleman et al 2008 calculated how much more common each pair is than expected for all 3,721 codon pairs. So for any DNA sequence I can add up this value for each consecutive pair of codons in my sequence and calculate a score; the higher this score, the better a solution it is for my problem.

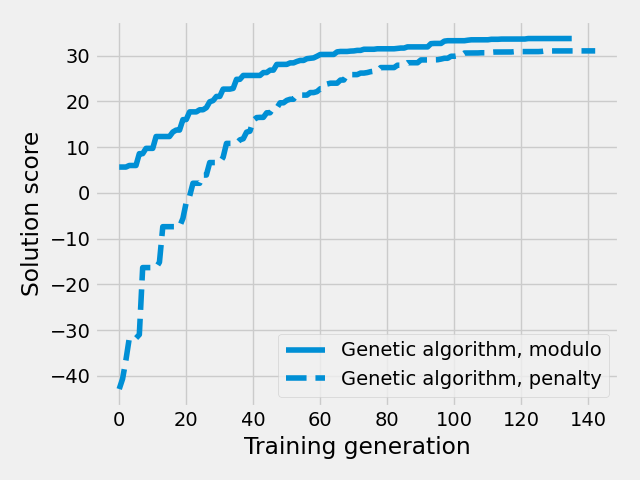
Next we need to frame the question for the genetic algorithm, what form will each possible solution take while the algorithm optimises it? The most obvious representation for specifying codons would be a vector where each element is a codon, represented by a number 1-61 (for each of the 61 codons). But this is too much freedom for our algorithm because it can swap in a codon that produces the wrong amino acid.

A better formulation is that each element represents one of the codons that encodes the correct amino acid. Figuring out what “the correct amino acid” is can be abstracted away to the scoring function, which means our genetic algorithm can’t breaking anything. Any single amino acid can be coded for by up to 6 different codons, so our solution vector is a sting of 1-6s and is as long as the string of amino acids we’re optimising (or a third as long as the DNA gene).

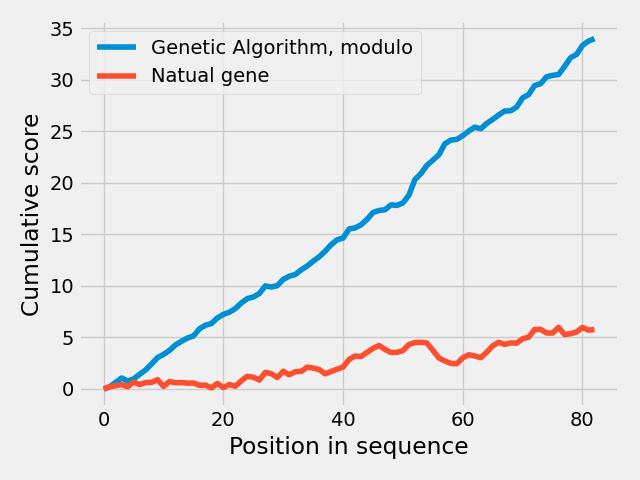
Another key benefit of making our solution vectors from 1-6 rather than 1-61 is a massive reduction in problem space. If L is the length of our vector there are 6L rather than 61L possible solutions to explore. A smaller problem space means a faster answer, which is particularly important with genetic algorithms. Although they find good solutions, they do it slowly. Another formulation I tried was 1-36, each element was correct codon pair (6x6), but this was so slow it didn’t have time to find a good solution. If I hadn’t thought to reformat to a smaller problem space I wouldn’t have been able to optimise genes.

There is a issue with using 1-6 to represent codons though, not all amino acids have 6 codons e.g. Lysine only has 2. The genetic algorithm might use the 5th Lysine codon (which doesn’t exist); what then? You could have the score function deduct points every time an invalid codon is used and hope that this pushes you towards good and valid solutions but this is not using our problem space efficiently. Every possible solution with an invalid codon in it is definitely not the best solution. This means lots of tried solutions in our 6L space (which is still an awful lot) are wasted effort.

Instead of a penalty, a better solution is modulo remainders. Whenever the genetic algorithm asked for an codon outside of the list of real codons I looped it back to the start of the list. So if it asked for the 5th codon when there were only 3, it looked back to the start and counted the remaining 2. This means that any solution explored by the genetic algorithm could potentially be the optimal solution. Because of this, we should get better answers, faster. In the plot below can see that the penalty algorithm is always behind the modulo algorithm and takes 30 generations just to get to the same start point.



Fortunately this allowed me to get a really good solution, much better than the naturally occurring gene. Below I’ve plotted the cumulative fitness over the length of the natural and optimised genes, common codon pairs increase the fitness, rare pairs reduce it. The natural gene is generally positive indicating that common pairs are generally preferred but not by much. In our optimised gene the slope is much steeper and consistently positive, indicating a much better use of codon pairs. In all, our optimised gene is 6 times better than the natural gene.



Conclusions

To get the most out of your genetic algorithms you need to frame your question such that problem space is as small and efficiently explored as possible. You also need a fitness function that accurately captures the essence of what you’re trying to achieve. So much of data science is just this, translating real world problems into numbers so the computer can help you. With practice you can frame the same question in multiple ways. Hopefully this post has show that the same question, if framed properly, can get you a better answer.