Why log-odds?

Suppose we have a positive training set P and a negative training set Q of DNA hexamers (e.g., binding sites and non-binding sites). We want to apply "machine learning" to determine a computer program that can tell if an arbitrary given hexamer is P-like or Q-like. That is, we're considering a two-step process. First, the ideal goal is to feed P and Q into a black box that produces, as its output, a program that reads an arbitrary hexamer and reports whether the input is either P-like or Q-like. The second step is to feed as many hexamers as we like to that automatically generated program.

Stated this way, the problem is too general; we need to narrow down the class of possible computer programs that are candidates for distinguishing P-like sequences from Q-like sequences. For this discussion, let's consider a fixed program that has unspecified "program parameters". The program compares each letter of the 6-letter input with each of its 4 possible values, and increments a running total accordingly.

```
read $x_1x_2x_3x_4x_5x_6$ (a 6-letter DNA sequence)
initialize $total$ to 0
if $x_1$ is A then add $w_1(\texttt{A})$ to $total$
if $x_1$ is C then add $w_1(\texttt{C})$ to $total$
...
if $x_6$ is T then add $w_6(\texttt{T})$ to $total$
report $total$
```

The program (which should look familiar) has 24 program parameters, $w_i(j)$. Our goal is to pick those 24 numbers so the program does the best possible job of giving high totals to strings in P and low totals to strings in Q. The point is that once we decide to try a particular choice of 24 numbers, we can run the program on P and Q and look at an overall score, say the sum of the program's output for strings in P minus the sum of outputs for strings in Q. We could get smarter and use a packaged numerical optimization procedure to find a good choice for the 24-dimensional vector, $w_i(j)$..

However, for this particular program we can find a provably optimal choice of those 24 parameters with very little work. Moreover, the same ideas work for optimizing parameters in many of other kinds of computer programs. That's why we are messing with this log-odds stuff.

Another example. Suppose you perform a certain experiment on a set of 300-bp sequence. Let P be the set where the experiment succeeded, and let Q be the set where it failed. We want to automatically "learn" a program that does a good job of predicting whether an arbitrary given 300-mer will succeed or fail in the test. But what property of the sequence will we use?

Suppose we look at the sequences and note that CGs are denser in P than in Q, while TTs are denser in Q than in P. There might be other dinucleotides that provide useful information for discriminating P from Q, and we want to learn those, too, if possible. This gives us an outline for a program. Given an 300-bp sequence (or for that matter a sequence of any length) the program will start by determining frequencies in that sequence of each of the 16 dinucleotides, AA, AC, ..., TT. Intuitively, if the frequency of CG is above some threshold, we'll report a high number (to say that the sequence is P-like), and if TT is above some threshold, we'll report a low number. But what thresholds, what reported numbers, what other dinucleotides are useful, and how should the information be combined. Fortunately, all these questioned are answered by using the "log-odds stuff" on 1st-order Markov models for P and Q. Each model has 16 "unspecified constants", though they are slightly different from just the frequencies of the 16 dinucleotides.

In these two examples, the log-odds approach does two things: it gives us a rationale for picking the constants and it provides a simple and fast way to determine the constants from training data. With more complicated probabilistic models, such as hidden Markov models, the theoretical results still tell how the constants (parameters of the model) should be set, but computing those values is more like the numerical optimization approach mentioned above.