CSE702 Week 5B (Thu 2/29): Model Structure and Principles, continued

Now we come to the equations that model the principle that the probability of a move depends upon its value in relation to the values of alternative choices. This principle undergirds pretty much any utility-based prediction model. We first show how this is done in the vast majority of cases via a **log-linear** model, but that this approach fails in chess---at least in the straightforward implementations coded thus far. Whereas, a **loglog-linear** model works to a surprisingly and suddenly good degree for equations with only two main parameters. Does this have the force of physical law?

The Core Predictive Component

At last we come to the parts of the model that generate its projected probabilities. If we let

$$m_1, m_2, \ldots, m_\ell$$

denote the ℓ-many legal moves *in the order that the chess program lists them at its highest depth*, then we want to compute the corresponding probability vector $\vec{p} = (p_1, p_2, \ldots, p_\ell)$ with

$$p_i = \Pr[Y(s, c, \ldots) \textit{ plays } m_i]$$

for $i = 1$ to ℓ. (I have settled on the convention of numbering moves from 1.) Because this definition is stated in terms of a *virtual* player, it is mathematically well-defined. The efficacy of the model comes from how well the given $Y(s, c, \ldots)$ represents an actual human subject.

The aspect of the model being minimalist is that $\vec{p}$ for any position t depends only on:

- the main parameters used to define the virtual player $Y(s, c, \ldots)$ whose actions are projected (note that the reference rating R is determined from Y being given);
- any hyperparameters that are either completely fixed before training or are dependent on the main ones; and
- the values of each legal move at each search depth recorded by the engine---in a table such as the above (in AIF format).

Here are three main principles that govern the design of the model to generate $\vec{p}$ from these data:

1. The probability of a move depends on its value *in relation to the values of other moves*.
2. Separate performance measurement from prediction.
3. Weaker players are weaker not so much because they inherently prefer weaker moves, but because they are more likely to be "diverted by shiny objects."

I regard Property 1 as organic to any model based on a utility/risk/reward function $f(u_i)$. In our case, u_i is the loss (in ASD or points expectation) from choosing the move m_i. It tops out at 0 if m_i is an

optimal move (as judged by the engine or engines whose high-depth values are filling the model) and is negative otherwise. It actually does not matter if we flip the sign of u_i or add a fixed value to it or apply any other linear transformation to the values, so long as we do so consistently for all moves in a position and all positions in our datasets, because we will have the coefficients for a linear transformation inside our model.

Diversion: Log-Linear Model

The most prominent utility-based models are **log-linear**, meaning they have the form

$$\ln\left(\frac{1}{p_i}\right) \;=\; \alpha \;+\; \beta u_i.$$

Notes: I prefer ensuring that logarithms have nonnegative values whenever possible and reserve "$\log$" for base 2, so I avoid writing $\log(p_i)$ here. Because the left-hand side is dimensionless, α is dimensionless while β has units of "inverse utility." In our case, the utility units are *centipawns* (whether scaled down as "Pawns in Equal Positions" or not), so β has units of inverse centipawns---whatever that is. The s parameter is much the same as "β" written as a divisor, so that it expresses the idea of sensitivity in natural centipawn units---how much of a difference in assessed value you are sensitive to.

Keeping things abstract for the moment, the log-linear model is solved via:

$$\frac{1}{p_i} \;=\; e^{\alpha+\beta u_i}, \text{ so } p_i \;=\; e^{-(\alpha+\beta u_i)}, \text{ and } \sum_i p_i \;=\; 1.$$

We can write $p_i \;=\; p_1\dfrac{p_i}{p_1} \;=\; p_1\dfrac{e^{-\alpha}e^{-\beta u_i}}{e^{-\alpha}e^{-\beta u_1}} \;=\; p_1 e^{\beta(u_1-u_i)}$. Note that the term with α canceled out. If we fix $u_1 = 0$, then this simplifies further to

$$p_i \;=\; p_1 e^{-\beta u_i}.$$

Here β and u_i should have the same sign---that is, if we code the utility dropoffs to be positive rather than negative, then β will be positive. This is so that $p_i \le p_1$. If $u_i \;=\; 0$, so that the choice m_i has equal utility to the first choice m_1, then $p_i \;=\; p_1$. The probabilities all come out sorted by the values of the moves. Now we can use $\sum_i p_i \;=\; 1$ to solve this:

$$1 \;=\; p_1\left(1 \;+\; e^{-\beta u_2} \;+\; \ldots \;+\; e^{-\beta u_\ell}\right),$$

so

$$p_1 = \frac{1}{1 + e^{-\beta u_2} + \ \ldots \ + e^{-\beta u_\ell}}.$$

Note that if $\ell = 2$, meaning there are only two legal moves, this is exactly a logistic curve. Thus we can say the solution is a generalized logistic curve. The solution for $i > 1$ then becomes

$$p_i = \frac{e^{-\beta u_i}}{1 + e^{-\beta u_2} + \ \ldots \ + e^{-\beta u_\ell}}$$

Even without assuming $u_1 = 0$, we get the solution for all $i \geq 1$ as:

$$p_i = \frac{e^{-\beta u_i}}{e^{-\beta u_1} + e^{-\beta u_2} + \ \ldots \ + e^{-\beta u_\ell}}.$$

This is exactly the **softmax function**. The denominator Q is positive, so $\alpha = \ln(Q)$ is well defined, and we get the original equation $p_i = e^{-(\alpha + \beta u_i)}$ back again. Thus α did not completely disappear---it just absorbed the requirement that the probabilities sum to 1. My way of interpreting this:

- Every option m_i has a "share" $e^{-\beta u_i}$ (of the "pot" Q) that depends only on its utility value u_i (once β is globally fitted). The probability p_i is determined by how large Q is.
- The values of other moves influence the probability only through the condition that the probabilities sum to 1.
- Each probability p_i is a **multiple** of the best-move probability p_1. The multiple is a ratio of the share of p_i and the share of p_i. When $u_1 = 0$, so that the share of p_1 is taken as 1.0, then the multiple is just the share of p_i.
- If you model $\ln\left(\frac{1}{p_i}\right) - \ln\left(\frac{1}{p_1}\right) \ = \ \alpha + \beta u_i$ instead, you get the same end result.

This kind of probability distribution involving exponents was developed by ~~machine learning theorists in the 1970s~~ physicists in the 1870s, in particular Ludwig Boltzmann and Josiah Gibbs. In the physics context, Q is the partition function. In thermodynamics, the parameter β has units of "inverse temperature"; in chess, we've already seen that the units are "inverse centipawns." That is to say, my s parameter is just $\frac{1}{\beta}$, and its presence in the model is *de rigeur*. Dividing out the units of centipawns is a matter of ontological necessity. The c parameter piggybacks on the observation that βu_i is dimensionless, so $(\beta u_i)^c$ is kosher for any power c.

In quantum circuits C, the utilities are replaced by values of a polynomial $P(y_1, \ \ldots \ , y_h)$ where h is the number of binary nondeterministic gates in the circuit. My 2018 paper on quantum circuits with Amlan Chakrabarti and my PhD graduate Chaowen Guan broadened earlier representations of these polynomials; I'm currently trying to develop a thesis that algebraic-geometric invariants of P act as

complexity measures that may explain our 30-year engineering difficulty in operating such circuits physically. But I digress...to cut to the chase:

- The log-linear model is **ordained** both in machine learning and in physics.
- Yet it fails in chess---even if you incorporate c.

I actually discovered the failure way back in 2008, in the wake of some small-scale Matlab code written by Steve Uurtamo and some others in a seminar I ran that spring. I had some other intuitions that led me to believe that a *ratio*, not *difference*, of logs was needed on the left-hand side.

What Works: The Double-Log-Linear Model

The equation that works---markedly better albeit not perfectly---in chess is:

$$\ln \ln\left(\frac{1}{p_i}\right) - \ln \ln\left(\frac{1}{p_1}\right) = \alpha + \beta u_i \, .$$

This becomes:

$$\frac{\ln(1/p_i)}{\ln(1/p_1)} = e^{(\alpha+\beta u_i)},$$

so

$$\ln\left(\frac{1}{p_i}\right) = \ln\left(\frac{1}{p_1}\right) e^{(\alpha+\beta u_i)}.$$

Exponentiating both sides again and temporarily substituting $x = e^{(\alpha+\beta u_i)}$ gives

$$\frac{1}{p_i} = e^{x \ln\left(\frac{1}{p_1}\right)} = \left(\frac{1}{p_1}\right)^x,$$

so

$$p_i = p_1^x = p_1^{\exp(\alpha+\beta u_i)}.$$

In fact, because we started with a *difference* of two double-logs, it is reasonable to suppose $\alpha = 0$. Substituting the ASD $\delta'(m_1, m_i)$ for u_i, $1/s$ for β, and raising the dimensionless βu_i to the power c again gives the entire model equation that I deployed from 2011 thru 2019:

$$p_i = (p_1)^{e^{\left(\frac{\delta'(m_1, m_i)}{s}\right)^c}}$$

together with $p_1 + \cdots + p_\ell = 1$. This has a completely mad-looking triple exponential on the right-hand side. I actually experimented with ways to substitute a gentler "curve" to use in place of $e^{(\cdots)}$ in

the exponent part, and you can find a gaggle of other "`curve`"s implemented in the C++ program, but none has worked better than exponential. (It is actually called "`invexp`" for inverse exponential in the code, because some other interpretation gives it a negative sign.) You might wonder why the model equation can't simply be

$$p_i \;=\; (p_1)^{\left(\frac{\delta'(m_1,m_i)}{s}\right)^c}$$

without the e in the middle. The flaw is that this does not enforce $p_i \le p_1$, because $\left(\frac{\delta'(m_1,m_i)}{s}\right)^c$ can be less than 1---indeed, it can be zero. The upshots now are:

- Every option m_i has a "share" $e^{(\beta u_i)^c}$ that depends only on its utility value u_i (once β---that is, s---and c are globally fitted), but now it is in the exponent of p_1. Hence I call it a "power share". There isn't a simple sum Q of shares---it's more complicated. No softmax.
- The values of other moves again influence the probability only through the condition that the probabilities sum to 1.
- Each probability p_i is now a **power** of the best-move probability p_1.

This is enough for good results. But it leaves the model with the property *that the engine's first move always has the highest projected probability*. This leads into the initial contrast of my 2019 "Predicting Chess and Horses" article. It also makes the model use only the highest-depth values for the moves, so as to form the $\delta'(m_1, m_i)$ utility values in the equation

$$\ln\ln\left(\frac{1}{p_i}\right) \;-\; \ln\ln\left(\frac{1}{p_1}\right) \;=\; \left(\frac{\delta'(m_1,m_i)}{s}\right)^c$$

for generating the projections, when those same values are used to assess performance. I had recognized the second principle as a *desideratum* in a talk I gave in 2009, but it took the whole decade to tame the vicissitudes of implementing it.

Second Principle: Separating Prediction From Performance Assessment

In many sporting areas, the same data is used for prediction and performance assessment. For instance in running foot races, the main data is your race times---previous and current, it's the same data. Projections take the form of extrapolating from scores in the (recent) past.

All forms of the model above make the same top-depth values of moves used both for judging quality and for predictive utility. It follows from the equations in particular that for any moves m_i, m_j:

$$\text{if } u_i = u_j \text{ then } p_i = p_j.$$

In particular, if the chess program lists out two moves of equal optimal value in the order m_1, m_2, then it should make no difference which one is played, no? They are moves of equal value, "peas in a pod" as far as the data and equations are concerned.

Yet the same game data (then in 2009--2011 from the **Rybka 3** program, until it was DQ-ed for plagiarism in late 2011) quickly showed that in such cases, the move m_1 was being played a massive **58%** of the time. This is detailed in my GLL article https://rjlipton.wpcomstaging.com/2012/03/30/when-is-a-law-natural/ I considered randomizing the listing order in such tied cases, but that was an ostrich reaction to a better predicition opportunity. Instead, I put in an *ad-hoc, post--hoc* fudge of the final probabilities to reflect this phenomenon.

I later verified the stable-sorting hunch stated in the article, and Tamal Biswas helped me expand this phenomenon into full efforts to realize this separation principle:

- Use the values $v_{i,d}$ over earlier depths d of search for prediction.
- Use only the top-depth values u_i for assessment.

That is to say, the values of the move m_2 in such cases are generally lower at earlier stages of the search than those of m_1. We can find many examples in the provided AIF data in **`/projects/regan/Chess/CSE712/AIF/`**, as ties for top value happen typically 8--10% of the time for chess programs.

Biswas and I tried various ways of using the lower-depth values in all cases, not just when the final values were tied. Trying to make d a fully regressable parameter for a player's "habitual depth of search" created a moonscape of bad local minima causing the regression descent to go haywire. [FYI: minimizing a polynomial $p(x_1, \dots, x_n)$ is NP-hard even when the minimum is zero, and certain forms of this are NP-hard even when p is linear---this is taught in CSE491/596.] Our more-controlled attempt involved creating a second "subjective [loss of] utility" term ρ to go with the "objective loss" term δ' and manage a third parameter h (for nautical "heave") in an extended equation like so:

$$\ln\ln\left(\frac{1}{p_i}\right) - \ln\ln\left(\frac{1}{p_1}\right) = \left(\frac{\delta'(u_1,u_i) + h\cdot\rho(---v_{i,d}---)}{s}\right)^c.$$

This is framed in such a way that whether $h < 1$ or $h > 1$ intuitively tells whether the objective or subjective values have higher influence on behavior. I expected h to be under 0.5. Instead we found it most often over 1.5. This still seemed fine as when we assembled the final model just before his dissertation defense in late July 2016, we got some tremendous three-parameter fits---which I put in my Aug. 2016 talk at the Indian Statistical Institute on my visit to Kolkata. It was on the plane home that I first tripped across unsustainable instability for fits of individual players. I told the story in this Election Day 2016 post and plumbed the issue in full the following May after Cynthia Rudin (who graduated from UB in the 1990s and visited as a Distinguished Speaker as mentioned here, also here) convinced me it was unfixable. [These old talks and blog posts are FYI; I don't necessarily want you to get down in the old modeling weeds, but feel welcome if you're curious, and some possible project ideas might use them further.]

It took over two more years to craft a tightly-controlled approach to this and a third principle that stays reliably stable---usually (we have already seen a couple cases in my spreadsheets where the s parameter "crashes to nearly zero", but the outputs even then remain usable).

Third Principle: What Causes Weaker Players to Play Weaker Moves?

Common belief is that weaker players **prefer** weaker moves. In a brusque sense, that amounts to denying that the axiom

$$u_i > u_j \implies p_i > p_j$$

holds at weaker rating levels (or at all). One project idea that would really get into said weeds would be to expand the simple kind of bulk evidence I give in support of this axiom in my August 2019 "Predicting Chess and Horses" article.] Instead, the principle of using the lower-depth values for prediction can be stated as:

- Weaker players are more likely to be diverted by shiny objects.

I often feel this myself when I am playing: As soon as I learn an attractive new fact about a move, I have an impluse to play it right away---without necessarily going back to other moves I've considered to see if it is really better. Or even if it's really safe to play. This strikes me as also related to the thesis of the book Nudge by Richard Thaler and Cass Sunstein.

The implementation has a term like ρ but does not allow a separate variable h to multiply it---instead, its relative influence is globally controlled by the "gradient" hyperparameter mentioned above. The parameter e_v is allowed only to further resolve moves of equal-optimal value, and even it is a separate regressed parameter only over the large training sets. The implementation is demonstrably sub-optimal as far as prediction goes, but:

- It does give highest probability to an inferior move 10--20% of the time. That is, it often predicts players of a given rating will fall into separate traps.
- It gives 2--3 percentage points higher predictivity overall, which seems to sharpen z-scores of (cheating) players by 0.5 or so.
- The final model is highly numerically stable and passes the internal prediction accuracy checks developed here and exhibited in graphs just recently here.
- It finally reflects---in some form---all of the large-scale phenomena I know in chess play.

Two Other Features

There are two other highly technical features worth mentioning. One is that the model does not use a fixed depth of analysis---nor does the screening stage. There are bounding depths $d_1 < d_2$ and

bounding counts $c_1 < c_2$ of positions searched (called **nodes**) so that the rule for running analysis is:

- Search at least to depth d_1 and for at least c_1 nodes. When c_1 nodes are done, finish the current depth d of search. But if you finish depth d_2 before passing c_1 nodes, stop there; while if you pass c_2 nodes at any time, abort and stop with a patched-up partial rendition of the current search level d, even if $d > d_1$.

This removes any dependence on processor speed. Moreover, if only one CPU core is used, the whole search is often *reproducible*.

- Screening in Single-PV mode with Stockfish versions uses $d_1 = 20$ and $d_2 = 30$, $c_1 = 5$ million nodes and $c_2 = 50$ million. Komodo and Komodo Dragon versions use $d_1 = 18$ or 19 and $c_1 = 3$ or 4 million nodes.
- Full analysis in Multi-PV mode uses $d_1 = 20$ and $d_2 = 30$ (even for Komodo) but $c_1 = 200$ million nodes and $c_2 = 900$ million nodes.

The main point is that this allows searching to higher depths in relatively simple positions, especially endgames with few pieces on the board.

The second wrinkle is a knock-on of this motivation: The real depths $d_0 \ldots d$ are mapped onto the virtual interval $d_0 \ldots 20$. (The bottom depth d_0 is not depth 1 even though Stockfish versions display it---the reason for that and my handling of a further notorious issue with 0.00 values of chess programs are described here.) A side benefit is that the engine values---which can jerk up and down between depths as you can readily see---are usefully smoothed out. This makes "depth 20" a stationary concept in my current implementations.

Moreover, **the entire code is linearized**. Many program elements are coded as potentially being linear combinations of themselves and other elements. This is most immediately visible in the way weights of terms in the main loss function (menu option [17]) can be freely fiddled with. It is the rule of Schrödinger's Cat. This has paid some benefits in code uniformity and simplicity, for instance in how I implemented the Efron bootstrap technique.

Okaaayyy...from now on we will accept the wisdom of the modeling decisions---unless we see more ideas besides the Sonas correction for improving the model. On to applying it...