## CSE702 Spring 2025 Week 5: How the Model Works

The **basic model** uses only the move value numbers $v_1, \ldots, v_\ell$ given at the highest depth of search. The only other values that determine model outputs on-the-fly are the **main parameters** $s$, $c$, and (the restricted form $e_v$ of) the gullibility parameter $h$. We can regard the mapping $R \mapsto (a_m, b_m, a_p, b_p, uz, vz, e_v)$ itself as a giant fixed hyperparameter. Moreover, we use only the differences of the other values from $v_1$. Earlier we wrote $\delta'$ to denote a scaled difference, but from now on we will just write $\delta$ with the scaling understood.

### The Base Model Equations

For $i = 1$ to $\ell$,

$$p_i = p_1^{e^{\left(\frac{\delta_i}{s}\right)^c}}$$

and $p_1 + \cdots + p_\ell = 1$.

What a monstrosity---with three levels of exponentiation! It does give us $\ell$ equations in $\ell$ unknowns. Let's sanity-check it first:

- The values $\delta_i$, $s$, and $c$ are all nonnegative, so $\left(\frac{\delta_i}{s}\right)^c$ is nonnegative.
- Thus $e$ to that power is at least $1$. (Note that the fraction $\frac{\delta_i}{s}$ itself can be $> 1$ or $< 1$.)
- Therefore $p_1$ is raised to a power at least $1$, so $p_i \le p_1$ for each $i$.
- If $\delta_i = 0$, so that the first $i$ moves are equal-optimal, then $p_i = p_1$. Likewise, if $\delta_i = \delta_j$ then $p_i = p_j$. (We will shortly observe this to be *majorly empirically false*; patching it is the job of $e_v$ and the use of lower-depth values generally.)
- If $\delta_j > \delta_i$, then $p_j < p_i$.
- Increasing the value of the exponent lowers the value of $p_i$, which generally drives up the probability $p_1$ of finding the (or an) optimal move.
- The $s$ parameter divides out the centipawn units of $\delta_i$. Thus the fraction is dimensionless, which legitimizes raising it to an arbitrary power $c$.
- ***Lower $s$ is better***---and the effect is most notable when $\delta_i$ and $s$ are both small. Thus $s$ models sensitivity to small differences in value.
- ***Higher $c$ is better***---and for fixed $s$, has greatest impact when $\delta_i$ is large. It thus drives down the probability of a large mistake. Hence the name "consistency."
- A discovery that a move $m_k$ is better or worse than previously thought (making $\delta_k$ smaller or bigger) can impact the value of $p_i$ for move $m_i$ but not its ordinal ranking relative to any move $m_j$ other than $m_k$ itself. It does so only indirectly---through the constraint that the probabilities sum to $1$.

## Log-Linear Digression

On pain of doing a "bait-and-switch", let's look at a simpler model equation that also has these properties:

$$p_i \;=\; p_1 e^{-\left(\frac{\delta_i}{s}\right)^c}.$$

Here $p_1$ is multiplied by the utility term rather than raised to its power. The exponent is non-positive, so the multiplier is $\leq 1$. Again, lower $s$ and higher $c$ are better in terms of reducing the probabilities $p_i$ of mistakes and hence raising $p_1$. Regarding the last property, here a change in the value of $m_k$ does not affect the ratio $p_i / p_j$, thus giving a larger degree of the "independence from irrelevant alternatives" property (IIA).

This model is called "Shares" in the code and is the basic **log-linear** multinomial logit model in this context. The generic derivation is:

$$\log p_i \;=\; \alpha + \beta v_i$$

where the "$\log$" can be $\ln$ or to any base because the $\alpha$ and $\beta$ coefficients can absorb any fixed constant factor. So:

$$p_i = e^{\alpha} e^{\beta v_i}.$$

Using $p_1 + \;\cdots\; + p_\ell = 1$ means normalizing these values:

$$p_i \;=\; \frac{e^{\alpha} e^{\beta v_i}}{e^{\alpha} e^{\beta v_1} + \;\cdots\; + e^{\alpha} e^{\beta v_\ell}} \;=\; \frac{e^{\beta v_i}}{e^{\beta v_1} + \;\cdots\; + e^{\beta v_\ell}}.$$

Since the $e^{\alpha}$ terms dropped out, we equivalently can start by taking differences:

$$\ln p_1 - \ln p_i \;=\; \beta(v_1 - v_i) \;=\; \beta\delta_i,$$

so

$$e^{\ln p_1 - \ln p_i} \;=\; \frac{p_1}{p_i} \;=\; e^{\beta\delta_i}, \quad \text{so} \quad \frac{p_i}{p_1} \;=\; e^{-\beta\delta_i}$$

so

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

In place of the single fittable coefficient $\beta$ we have $s$ (which is just $1/\beta$ if we hold $c = 1$) and $c$, replacing $\beta\delta_i$ with $(\delta_i / s)^c$. Again the solution of the exact values comes by normalizing:

$$p_i = \frac{e^{-\left(\frac{\delta_i}{s}\right)^c}}{1 + e^{-\left(\frac{\delta_2}{s}\right)^c} + \cdots + e^{-\left(\frac{\delta_\ell}{s}\right)^c}}$$

The initial $1$ comes because $\delta_1 = 0$. In particular, $p_1$ would be literally a logistic curve $\frac{1}{1+e^{-\beta x}}$ if $\ell = 2$ (ignoring that we actually have some nonlinearity via $c$ that is not covered here), and the case $\ell > 2$ explains the name "multinomial logit" model. The whole vector of probabilities can be written as $\vec{p} = softmax\left\{\left(\frac{-\delta_i}{s}\right)^c\right\}$.

The denominator $Q$ is positive, so $\alpha = \ln(Q)$ is well defined, and we get the equation $p_i = e^{-(\alpha+\beta\delta_i)}$ back again. Thus $\alpha$ did not completely disappear---it just absorbed the requirement that the probabilities sum to $1$. Moreover, we can unpack $\delta_i = v_1 - v_i$ (again, scaled values are intended here), so $-\delta_i = v_i - v_1$, and then multiply top and bottom by $e^{-\beta v_1}$ to recover the original form stated in terms of the (scaled) move values themselves.

Nature evidently works this way: this kind of probability distribution involving exponents was developed by ~~machine learning theorists in the 1970s~~ statistical physicists in the 1870s, in particular Ludwig Boltzmann and Josiah Gibbs. In the physics context, $Q$ is the **partition function**. In thermodynamics, the parameter $\beta$ has units of "inverse temperature"; in chess, the units are "inverse centipawns."

Chess, however, does not work this way---even when $c$ is incorporated. 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 the double-log model

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

(With double logs, I want to make absolutely sure that the outer one is not on a negative number, so I insist on writing $\log(1/p)$ instead of $-\log p$, and so on.)

$$\ln \ln\left(\frac{1}{p_i}\right) \; - \; \ln \ln\left(\frac{1}{p_1}\right) \;\; = \;\; \beta\delta_i \; .$$

This becomes:

$$\frac{\ln(1/p_i)}{\ln(1/p_1)} \;\; = \;\; e^{\beta\delta_i},$$

so

$$\ln\left(\frac{1}{p_i}\right) \;\; = \;\; \ln\left(\frac{1}{p_1}\right) e^{\beta\delta_i}.$$

Exponentiating both sides again and temporarily substituting $r_i = e^{\beta\delta_i}$ gives

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

so

$$p_i \;\; = \;\; p_1^{r_i} \;\; = \;\; p_1^{\exp(\beta\delta_i)}.$$

Raising the dimensionless $\beta\delta_i$ to the power $c$ again gives the entire model equation that I deployed from 2011 thru 2019:

$$p_i \;\; = \;\; (p_1)^{e^{\left(\frac{\delta_i}{s}\right)^c}}$$

together with $p_1 + \; \cdots \; + p_\ell = 1$ (and together with a post-hoc "fudge" of probabilities for equal-value moves). 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 the log-linear derivation gave it a negative sign.) You might wonder why the model equation can't simply be

$$p_i \;\; = \;\; (p_1)^{\left(\frac{\delta_i}{s}\right)^c}$$

without the $e$ in the middle. The flaw is that this does not enforce $p_i \leq p_1$, because $\left(\frac{\delta_i}{s}\right)^c$ 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 $\beta$---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 $p_1$.

### Three Principles and Their Violations

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

1. The probability of a move depends on its value *in relation to the values of other moves*. In particular, if a move has significantly higher value than others, it is much more likely to be found by a human player as well as by machines.
2. Besides the norm of training on separate data from (validation and) testing, it is good to base predictions on different kinds of data from that measured for assessment. I call this "separating 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."

Principle 1 is enough for good results. But it leaves the model with two properties that are empirically false:

- Moves that are tied in value---especially those tied for best with the first move---have equal probability.
- The best move always has the highest probability, even for weaker players.

[The running demo used throughout included results on tied-optimal moves.]

Thus the first principle's implication that $v_i = v_1 \implies p_i = p_1$ is overwhelmingly falsified. I did an *ad-hoc* patch by making the hyperparameter $pp$ with original fixed value 0.58, coming from the limited data I had using the Rybka 3 chess program at the time of this 2012 GLL blog article, such that any case of $v_i = v_{i+1}$ would adjust the originally-equal projected probabilities to make $p_{i+1} = 0.58p_i$. Three equal-value moves gave $p_{i+2} = 0.58p_{i+1} = 0.58^2 p_i$ and so on, hence the name stood for "patch power." Later use of versions of the Houdini, Komodo, and Stockfish engines crept $pp$ up to its current value 0.616. That is incidentally close to the (reciprocal of the) "golden ratio" $r = 0.61803398...$ , which satisfies $r^2 + r = 1$. But I also found rating dependence, hence my resort to ideas described next, ultimately roughly handled by the $e_v$ parameter.

### 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. 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. Here are two ways to embody it:

(a) Use a "panel" of programs to generate the move projections $p_i$, but measure actual T1 and ASD (etc.) against a given strong program. The "panel" programs need not be as strong---the closer they are to "human play", the better.
(b) Use lower-depth values for the predictions, but use only the highest-depth values to judge quality of the results.

Option (a) brings up the idea of **model averaging**. This can always be done by running separate instances of the model (for different chess programs) and then averaging their outputs. I had hoped to save considerable effort by *combining inputs*---ideally by putting their evaluation functions on a common scale governed by the notion of points expectation ("win probability"). Alas, as we've seen:

- The points expectation scale is rating-dependent.
- Chess programs can---and do---post-process their evaluations in various ways that frustrate bringing them onto a common scale.

The latter point makes the task become one of *building a model of the particular program*. A final operational issue: In principle, the model built exclusively with Stockfish 11 still can be applied to test conformance to Komodo 13---or to the latest Stockfish 17. But that would require a separate validation run and possible $z$-score adjustment for each pair of source and target. It is easiest in practice to keep the source and target the same.

Option (b) remains on the table. The unequal actual frequencies of equal-value moves clued me into the use of lower-depth values. Tamal Biswas discovered in his thesis work that the overall effect of lower-depth values was far greater than I'd suspected. This is detailed in this 2015 GLL blog article. This led to a full articulation of (b) as a separation principle:

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

Biswas and I tried various ways of using the lower-depth values in all cases, not just when the final values were tied.

1. The most general and ambitious idea was to make a third player-specific skill parameter $\boldsymbol{d}$ for "habitual depth of thinking." It came with a fourth parameter $\boldsymbol{v}$ for the "variance" of this depth. The plausibility of $d$ growing with rating was shown by the GIF animation in the 2015 article, showing the *depth at which a player's major mistakes are seen by the engine*. Yes, it is possible to judge your chess rating by looking only at positions where you screw up! (For the most part, though, mistakes are already refuted by the computer at low depth.)

However, the minimization landscape for the resulting model became a moonscape of bad local minima causing the regression descent to go haywire. [FYI: minimizing a polynomial $p(x_1, \ldots, x_n)$ 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.]

2. Our second attempt involved creating a second "subjective [loss of] utility" term $\rho$ to go with the "objective loss" term $\delta$ and manage only a third parameter $h$ (for nautical "heave") in an extended equation like so:

$$\ln\ln\left(\frac{1}{p_i}\right) - \ln\ln\left(\frac{1}{p_1}\right) = \left(\frac{\delta(v_1,v_i) + h\cdot\rho(---v_{i,d}---)}{s}\right)^c.$$

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). [OK, those words with "already seen" were from last year, but we saw it happen live in today's demo. Plus you can find more cases in the newly-posted spreadsheets. It is fortunate---maybe even strange---that the model retains its coherence even when $s$ is tiny and $c$ is small to match it.]

### 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 use of lower-depth values embodies this principle:

- 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 $\rho$ but does not allow a separate variable $h$ to multiply it---instead, its relative influence is globally controlled by the $v$ hyperparameter mentioned above. Since it applies only to the maximum depth, it is more of a "gradient" down on its lower side than a "variance." 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 (little) 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.

[The demo from Isle of Wight 2025 included showing that on several hundred positions faced by sub-1550 players---positions with at least 5 moves within 0.20 of the best move---those players still favored the computer's best move with a clear plurality over any other move. A case on larger data with even weaker players and positions with ten close-to-optimal moves is detailed in my pivotal 2019 "Predicting Chess and Horses" article. This shows my grounds for believing that a large number of sub-1550 rated players, voting on moves while isolated from each other, could compete in a competition like 1999's Kasparov Versus the World. One other point is, let's say a weak player has a 5% chance of making a losing blunder on any move. Over 30 or so moves of a game, that player will probably trip on such a case and lose the game. But if 1,000 such players are voting on any move, 5% = 50 of them will screw up but be rescued in the vote by the 950 who don't---and plurality over 950 will be alm,ost as clear as over 1,000. It would take a lot of resources to conduct such an experiment, however.]

### 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 furthers this motivation: The real depths $d_0 \ldots d$ are mapped onto the virtual interval $d_0$ ... $20$. (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 regime of Schrödinger's Cat. This has paid some benefits in code uniformity and simplicity, for instance in how I implemented the Efron bootstrap technique: the code does sampling-with-replacement as randomly-assigned integer weights on each position in the sample.

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...

[The demos showed an overall moderate under-preformance, from 1725 down to about 1500, on the positions with at least 5 close-to-optimal moves. Those positions had higher "unscaled evaluation entropy", 2.94 versus the overall 2.49. They also had a moderate increase in average thinking time, 183 rather than the overall 153 seconds used per move. When we switched to a sample of positions that saw at least 300 seconds of thinking time, the performance on those positions plummeted to under 1100. Whereas, when we ran positions where the player used at most 8 seconds, the performance zoomed over 2800---by these under-2000 players!]