Skill Issues and the Physical World: An Analysis of CS:GO Skill Rating Systems

Mikel Bober-Irizar, Naunidh Dua & Max McGuinness*
University of Cambridge

***Abstract*— The meteoric rise of online games has created a need for accurate *skill rating* systems, which can quickly determine a team's skill for the purpose of tracking improvement and fair matchmaking. Although many systems for determining skill ratings are deployed, with various theoretical foundations, less work has been done at analysing the real-world performance of these algorithms. In this paper, we perform an empirical analysis of several systems through the lens of surrogate modelling, where the model can choose which matches are played next. We look both at overall performance and data efficiency, and perform a thorough sensitivity analysis.**

I. Introduction

Counter-Strike Global Offensive (CS:GO) is a multiplayer first-person shooter game where players work in teams of 5v5 to fight over objectives, and ultimately try to win a match of up to 30 rounds. As with many modern games, gameplay is focused around a central "competitive matchmaking" mode, where two teams of five with similar skill are pitted against each other, and the outcome of each match is used to update the players' skill ratings.

With millions of competitive matches played each day [1], having accurate skill ratings for each player and team is fundamental to producing fair matchups and allowing players to progress as their skill improves. The ever-rising popularity of matchmaking in multiplayer video games has shown that this format makes for an engaging experience.

In CS:GO, the developer *Valve* uses an unpublished variant of the Glicko2 rating system [2, 3], the perceived inaccuracies of the skill rating system are a major point of contention in the community. The term "elo hell" is commonly used by players who feel that their skill rating doesn't reflect their real ability [4], and many players opt to use alternative matchmaking services such as FACEIT (which uses the Elo system [5, 6]) to form ratings instead. While the benefits of good skill ratings is clear, little comparative work has been done to understand the performance of each system in CS:GO.

In this work, we take a look at several systems available for skill ratings (including Elo and Glicko2), and apply it to a large dataset of professional CS:GO matches. As well as comparing the skill rating systems themselves, we explore the effect of different matchmaking algorithms on the accuracy of skill ratings. We achieve this through a surrogate modelling environment where the matchmaking system chooses which match to be played next (and therefore to be used for updating skill ratings before the next match), while imputing real data for the outcomes of matches. This allows us to quantify the effect of the circular dependency between skill ratings and matchmaking algorithms, as opposed to measuring the behaviour of each skill rating system on a static set of past matches.

*All authors contributed equally to the work.

II. Related Work

The history of skill rating systems for games is a rich one. The Elo system, first introduced in the 1950s, is widely used in many sports and remains the defining system the chess world accepts. The system assumes that each player has a fixed skill rating, and the probability of each player winning the match is a function of the difference in rating between the two players [5]. It was the first rating system developed that modelled the players' skill level probabilistically.

In 1995, Mark Glickman created the Glicko system, specifically to improve upon issues he saw with the Elo system [2, 7]. Glicko adds a confidence parameter *RD*, that is a measure of the system's confidence in its estimate of skill. This allows changes in skill rating to also be dependent on this confidence parameter, thus introducing an idea of "information gained" by a particular match being played. Glicko2 adds an additional parameter σ, that measures the player's fluctuation in skill.

Herbrich, *et al.* from Microsoft Research introduced TrueSkill in 2007, based on Bayesian inference [8]. TrueSkill address two concerns of online team based matchmaking:

1) Match outcomes are team-based, but a skill rating for individuals is desired.
2) Some games have multiple 'teams' playing, and the match outcomes are not binary win/loss, (e.g. "free-for-all (FFA) deathmatch" matches)

TrueSkill uses Gaussian Processes (GPs) via a Factor Graph and Message Passing approach to model the players' skill and perform matchmaking. Microsoft analysed the performance of TrueSkill on a dataset collected in the beta testing of Halo 2, and found that TrueSkill substantially outperforms Elo [8]. Microsoft published a significant upgrade, TrueSkill 2, in 2018 [9]. Trueskill 2 is designed to address deficiencies in TrueSkill by analysing game-specific metrics such as player experience, the player's individual performance in the match, tendency to quit, and skill in other game modes to assist with matchmaking.

Some evaluation of prediction performance has been previously studied. Dehpanah *et al.* analysed Elo, Glicko and TrueSkill on a set of 100,000 matches from PlayerUnknown's Battlegrounds [10]. They found that

incorporation of new players into the model proved tricky, and deteriorated the performance of these rating systems.

Makarov *et al.* analysed the performance of rating systems on two *Valve* games, focusing on Dota 2 as well as CS:GO. They analysed TrueSkill on CS:GO games and observed a 62% accuracy [11]. Both approaches analysed a static dataset of past matches, in contrast with our approach.

III. Methods

A. Framework

For the experiments in this paper, we implemented a extensible Python library called **skillbench**, which allowed us to implement each component for comparison following a consistent interface. Skill rating systems such as TrueSkill are implemented as **Emulators**, with matchmaking algorithms implemented as **Acquisition Functions**. The overall training and evaluation is governed by a **Simulator**, which simulates the playing of games (as chosen by the acquisition function) based on the match dataset, and trains and evaluates the emulator.

Figure 1 shows the overall architecture of the library. In the following sections, we talk about each component in much more detail.

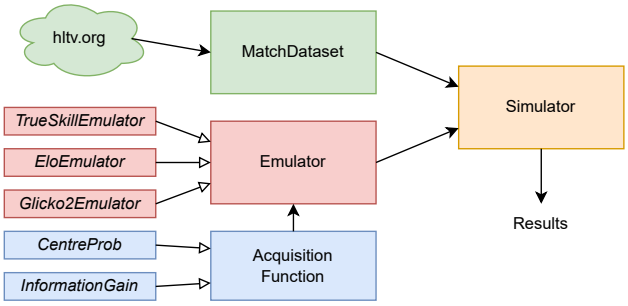


Fig. 1: The architecture of the **skillbench** library. *Emulator*s and their *Acquisition Function*s are implemented as modular components following a common interface. A *Simulator* takes an *Emulator*, trains it on a training *MatchDataset*, and evaluates it on an evaluation *MatchDataset*.

B. Simulator

In lieu of a game environment populated by real players, we simulate the process of chosen teams competing against one another by selecting records from a dataset of historical matches (Section III-E). The key constraint to this approach is that the model's choice of matchups is limited to those present in the dataset - these limitations are discussed further in Section V-A.

In our implementation, we characterise the simulator as being responsible for the generic training process of emulators. That is to say, it is within the *Simulator* class that a matchup is chosen according to its acquisition score (computed by an *Acquisition Function*); the result of the match is then simulated (by popping a random result from the set of real matches between that pair of teams); before finally the *Emulator* is informed of the result and it is fit into its internal model.

At each iteration, the Simulator chooses 25 matches from the remaining training pool at random, and the emulator is fed whichever match has the highest score according to an *Acquisition Function*. After a given number of training matches, we evaluate the Emulator's predictions according to its accuracy against historical results. For external validity, we can split our dataset and make use of separate "training" and "testing" Simulators.

We summarise this *Simulator*-led training process below.

Algorithm 1 Simulator.Train (Emulator **E**, Acquisition Function **AF**)

Require: loaded match dataset **M**.
  for i in num_evals **do**
    $\lambda \leftarrow$ 25 random matches from M;
    m* $\leftarrow$ argmax$_{m \in \lambda}$[AF(E, m.T1, m.T2)];
    M.pop(m*);
    E.fit(m*);
  end for

Algorithm 2 Simulator.Eval (Emulator **E**)

Require: loaded match dataset **M**.
  $\mu \leftarrow 0$;
  $\nu \leftarrow 0$;
  for m in M **do**
    if m.result $\neq$ draw **then**
      p $\leftarrow$ E.predict(m.T1, m.T2);
      if p>0.5 & m.winner = m.T1 **then**
        $\mu \leftarrow \mu + 1$;
      end if
      $\nu \leftarrow \nu + 1$;
    end if
  end for
  return μ/ν

C. Emulators

We model each skill rating system as an **Emulator**, which can "emulate" (predict) the outcome of a match between two teams as a probability. Each emulator can also be updated based on the outcome of previous matches.

We implement five emulators within our framework, which are briefly described below.

*1) **WinRate**:* WinRate is a baseline naïve emulator which we introduce in this work, as a point of comparison. We keep track of each team's proportion of matches won so far, $w(T)$, and then calculate a 'probability' of A winning a match against B:

$$E[A|B] = \frac{1 + w(A) - w(B)}{2}$$

One of the shortcomings of this approach is that a team's win rate is biased by the skill of the teams they have played against; this is addressed by the other emulators.

*2) **Elo**:* Our Elo implementation is defined as follows:

Parameters: k, representing the 'k-factor' and μ, the starting rating given to new players.

$$E[A|B] = \frac{1}{1 + 10^{\frac{R_B - R_A}{400}}} \quad (1)$$

Equation 1 represents the expected score (probability of winning) of team A in a match against team B, where R_T is the rating of team T.

To update the skill ratings, we multiply the difference between the actual outcome and the predicted outcome by k, and adjust the rating by this value.

*3) **Glicko2**:* The Glicko2 implementation has the following parameters:

- μ = The default rating of player.
- ϕ = The rating deviation (RD) of a player.
- σ = The player's skill volatility.
- τ = The system constant, which dictates the volatility over time.

The exact specifics of how to calculate the rating algorithm are detailed in [2]; we provide an overview here.

1) Compute the estimated variance (v) of the team's rating based solely on the game outcome.
2) Compute the estimated improvement (Δ) in rating by comparing the current rating to the rating based on the game outcome.
3) Iteratively compute the new volatility.

$$v = \left[g(\phi')^2 E(\mu, \mu', \phi') \cdot (1 - E(\mu, \mu', \phi'))\right]^{-1} \quad (2)$$

and

$$\Delta = vg(\phi')(s - E(\mu, \mu', \phi')) \quad (3)$$

where μ' ϕ' represent the rating and RD of the opponent, and s is the actual outcome.

*4) **TrueSkill**:* The TrueSkill implementation has the following parameters [8, 12]:

- μ = The rating of player.
- σ = The player's skill volatility.
- β = The skill class width, if a player has β rating higher than another, then the player has an 80% chance to win.
- τ = The additive dynamics factor, the square of which is added to the player's variance on each skill update. This ensures that σ does not converge to 0, and thus a player's skill rating never becomes static.

TrueSkill is a Gaussian Process over teams (or players), that is a joint distribution of infinitely many Gaussians. The TrueSkill algorithm attempts to minimise the Kullback-Leibler (K-L) Divergence between a 3D truncated Gaussian (created by the performance of the two teams) and the approximation of it.

A deep dive in the maths behind TrueSkill is given in [12].

*5) **TrueSkillPlayers**:* Similar to TrueSkill, but rather than tracking a per-team skill rating, each player's skill rating is tracked individually. We take advantage of TrueSkill's unique native support for any type of match, to update all 10 skill ratings for each player in one go.

We hypothesise that this could bring benefits, as it allows players to 'bring their skill ratings with them' when they occasionally move between teams or form new teams. This is advantageous to the approach taken in other rating systems, where a team needs to retain three players (a 'core') in order to keep its rating.

D. Acquisition Functions (AFs)

One of the goals of this analysis is to determine how to select teams for matches that provide the most information to the skill rating system - i.e. how can we acquire all teams' skill ratings in as few games as possible? To do this, we use the notion of an **acquisition function** (AF) from surrogate modelling: a function which determines which data point to sample next to update the model. In this case, the data points are match outcomes between some pair of teams, and the model is the skill rating emulator. This can be thought of as an analogue to the matchmaking system which is used in practice for selecting matchups.

An acquisition function provides a heuristic quantification for how valuable any given match may be in the training of an emulator, according to:

- the emulator's internal state;
- which teams are involved in the match.

More formally, we can define an acquisition function as:

$$\text{AF} : \text{Emulator} \times \text{Team} \times \text{Team} \to \mathbb{R}.$$

We will now discuss several approaches to designing an acquisition function.

*1) **Expected Improvement**:* One particularly prevalent form of acquisition is Expected Improvement (EI), which adopts a greedy strategy to locate the global minimum in a search space. It achieves this by selecting the point that has the greatest probability of being lower than our current best estimate, as predicted by a surrogate model such as a Gaussian Process (GP):

$$\mathbb{E}[u(x)|x, \mathcal{F}] = \mathbb{E}[\max(0, s_{\mathcal{F}}(x_*) - s_{\mathcal{F}}(x))|x, \mathcal{F}], \quad (4)$$

where $\mathcal{F}$ is our search space ($\mathcal{F} : X \to Y$), $s_{\mathcal{F}}$ is our surrogate model approximating $\mathcal{F}$, and x_* is our current best estimate for $\mathcal{F}$'s global minimum.

To translate this approach to the domain of CS:GO skill estimation, one could make the following correspondence:

	CS:GO Correspondence
$\mathcal{F}$	Our emulator's mean error rate at predicting match outcomes, after learning of a particular matchup (i.e. $\mathcal{F} : X \to Y$).
X	Possible matchups (i.e. Team $\times$ Team).
Y	Our emulator's mean error rate at predicting match outcomes.
x_*	The most useful matchup according to $s_{\mathcal{F}}$.

However, there is an issue with this setup. The basic formulation of EI (equation 4) models the function $\mathcal{F}$ and search domain X as static, when in reality they should depend on our emulator's internal state. In other words, over the course of training, we don't want to find the most useful single matchup, but rather the most useful

collection of matchups to condition the emulator, each iteration building on our existing collection.

This alters the premise of the Expected Improvement environment - we now have a dynamic search space, which depends on the history of matches x_{t-1} that have already been observed at that timestep:

	CS:GO Correspondence
$\mathcal{F}_{x_t}$	Our emulator's mean error rate at predicting match outcomes, after learning of a particular matchup, given match history x_{t-1} (i.e. $\mathcal{F}_{x_t} : X_{x_t} \to Y$).
X_{x_t}	Possible collections of matchups, each extending x_{t-1} by one (i.e. $\{x_{t-1}\} \times$ Team $\times$ Team).
Y	Our emulator's mean error rate at predicting match outcomes.
x_{t-1}	The most useful matchup collection according to $s_{\mathcal{F}x_{t-1}} : X_{x_{t-1}} \to Y$.

Being more explicit with regards to timesteps, we can now reformulate the Expected Improvement equation as:

$$\mathbb{E}[u(x)|x, \mathcal{F}] = \mathbb{E}[\max(0, s_{\mathcal{F}x_{t-1}}(x_{t-1}) - s_{\mathcal{F}x_t}(x))|x, \mathcal{F}]. \tag{5}$$

*2) **Cheater's AF**:* With this EI environment, we can immediately notice a hypothetically optimal (greedy) choice of acquisition function, by making our choice of $s_{\mathcal{F}}$ as inherently close to $\mathcal{F}_{x_t}$ as possible. In practical terms, this means "cheating" and computing $s_{\mathcal{F}}$ over unseen training data.

This way, the AF would be selecting its next matchup by training $|X_{x_t}|$ emulators, and taking whichever achieves the lowest error score on the remainder of our training data:

$$\text{AF}_{\text{cheat}}(E, T_1, T_2) = -\text{avg}(\text{err}(E', x) \text{ for } x \text{ in training data}), \tag{6}$$

where E' is a copy of emulator E after being trained on the outcome of a match between T_1 and T_2.

*3) **Gaussian Process**:* With our EI assumptions, we can note two factors inhibiting the usefulness of a simple GP for surrogate model $s_{\mathcal{F}}$:

- **The dynamic between X and Y changes** as t increments. A gaussian process could model this as variance in X, but doing so would fail to capture what should be a predictable dynamic (e.g. $\mathcal{F}_{(x_2,x_2)}(x_2)$ will almost certainly be higher than $\mathcal{F}_{(x_1,x_1)}(x_2)$).
- **Our model cannot directly sample Y** at each iteration, as computing Y depends on unseen future matches. Instead, the model can only sample the outcome of one match per iteration, and must *approximate* Y based on:
  - The information content of x_t.
  - An assumption about how our emulator will make use of this information content.
  - An assumption about the distribution of matchups on which the emulator would be evaluated.

We can address both of the above by focusing our acquisition function's design goal into the following:

How can we compute the information content of a particular matchup, in the context of whichever matches the emulator has seen thus far?

*4) **Information-Theoretic AF: Likeliest Draw**:* In designing an information-theoretic acquisition function, an intuitive option would be to take the matchups that our emulator believes are *most likely to be draws*, in order to settle uncertainties of skill between pairs of players. We can justify this as being the entropy of an individual matchup.

Let us suppose that R is a discrete random variable (DRV) representing the emulator's predicted results for a particular matchup, having two outcomes: w_{T1} and w_{T2}. Furthermore, suppose M is a DRV representing all matchups seen by the emulator. In this regard, $H(R|M = m)$ represents the entropy of the two possible results for a particular matchup according to our emulator.

We can derive a formula for the entropy in terms of the emulator's predictions $p(w_{T1}|m)$ and $p(w_{T2}|m)$:

$$\begin{aligned}\theta = H(R|M = m) = &-p(w_{T1}|m)\log(p(w_{T1}|m)) \\ &-p(w_{T2}|m)\log(p(w_{T2}|m)).\end{aligned}$$

Taking $p(w_{T2}|m) = 1 - p(w_{T1}|m)$:

$$\begin{aligned}\theta = H(R|M = m) = &-p(w_{T1}|m)\log(p(w_{T1}|m)) \\ &-(1 - p(w_{T1}|m))\log(1 - p(w_{T1}|m)).\end{aligned}$$

We can then show that the entropy is maximized when $p(w_{T1}|m) = 0.5$ by taking the derivative:

$$\begin{aligned}\frac{d\theta}{dp} &= log(1 - p(w_{T1}|m)) - log(p(w_{T1}|m) \\ 0 &= log(1 - p(w_{T1}|m)) - log(p(w_{T1}|m) \\ 1 &= \frac{1 - p(w_{T1}|m)}{p(w_{T1}|m)} \\ 0.5 &= p(w_{T1}|m).\end{aligned}$$

Thus, we define acquisition function:

$$\begin{aligned}\text{AF}_{\text{draw}}(E, T_1, T_2) = &-p(w_{T1}|m)\log(p(w_{T1}|m)) \\ &-(1 - p(w_{T1}|m))\log(1 - p(w_{T1}|m)).\end{aligned} \tag{7}$$

*5) **Information-Theoretic AF: Cross-Entropy**:* The likeliest-draw approach has a key limitation: it neglects to consider the number of prior encounters that an emulator has had with a specific matchup.

The failure here is to assume that the entropy within a match is reflective of the entropy within our emulator's understanding of the world. In actual fact, some teams will naturally have a new-draw winrate against one another, and sampling these matchups many times over won't be informative to the emulator as time goes on.

To rectify this, we propose reformulating the entropy calculation to instead compute the surprisal of each result *in the context of all results that the emulator has previously seen*. The assumption here is that more unexpected results will be more informative to the emulator.

This is equivalent to the Cross-Entropy (CE) between the emulator's predicted distribution of results for a particular matchup $(R|M = m)$, and its predicted distribution of results across all known matchups $(R|M)$, derived as follows:

$$\begin{aligned}&CE((R|M=m),(R|M))\\&=-\mathbb{E}_{(R|M=m)}[\log(R|M)]\\&=-p(w_{T_1}|m)*\log(p(w_{T_1}))-p(w_{T_2}|m)*\log(p(w_{T_2})).\end{aligned}$$

By Bayes' theorem, we take $p(w_{T_1}) = p(w_{T_1}|m)p(m)$:

$$\begin{aligned}&CE((R|M=m),(R|M))\\&=-p(w_{T_1}|m)\log(p(w_{T_1}|m)p(m))\\&\quad-p(w_{T_2}|m)\log(p(w_{T_2}|m)p(m))\\&=-p(w_{T_1}|m)\log(p(w_{T_1}|m)p(m))\\&\quad-(1-p(w_{T_1}|m))\log((1-p(w_{T_1}|m))p(m)).\end{aligned}$$

Here, $p(m)$ represents the probability of matchup m occurring according to the emulator. By assuming matchups are independently distributed between teams, we can find $p(m) = \frac{c(T1)}{\sum_{T\in Team} c(T)} \cdot \frac{c(T_2)}{\sum_{T\in Team} c(T)}$, where $c(T_1)$ is the number of times that the emulator has observed team T_1. This gives us a computable expression:

$$\begin{aligned}&\text{AF}_{\text{CE}}(E,T_1,T_2) = CE((R|M=m),(R|M))\\&=-p(w_{T1}|m)\log\left(p(w_{T_1}|m)\frac{c(T_1)}{\sum c(T)}\frac{c(T_2)}{\sum c(T)}\right)\\&-(1-p(w_{T_1}|m))\log\left((1-p(w_{T_1}|m))\frac{c(T_1)}{\sum c(T)}\frac{c(T_2)}{\sum c(T)}\right).\end{aligned} \tag{8}$$

*6) **Weighted AF**:* One could think of AF_{CE} as a function of two factors: the estimated likelihood of a draw between two teams, and the number of times our emulator has seen those teams before. However, it is not clear how to parameterise these factors within AF_{CE}.

Instead, we propose a weighted acquisition function which models the two factors (draw probability, no. of times teams seen) explicitly:

$$\begin{aligned}&\text{AF}_{\text{weighted}}(E,T_1,T_2) = \alpha\cdot\text{draw\_factor}+\beta\cdot\text{seen\_factor}\\&=\alpha\cdot(1-|p(w_{T_1}|m)-p(w_{T_2}|m)|)\\&+\beta\cdot\left(\left(\frac{1}{c(T_1)}-\frac{1}{c(T_1)+1}\right)+\left(\frac{1}{c(T_2)}-\frac{1}{c(T_2)+1}\right)\right).\end{aligned} \tag{9}$$

*7) **Other Acquisition Functions**:* For the sake of comparison and experimentation, we implemented several other simple acquisition functions:

- **LeastSeen**: simply sum the number of times an emulator has seen each team (or, in the case of *TSPlayers*, players) and perform a negative logarithm on each result:

$$\text{AF}_{\text{unseen}} = -\sum_{T\in m}\log(c(T)). \tag{10}$$

  This is another information-theoretic approach that equates observations with bits of information, though neglects to take into account result predictions.

- **MostSeen**: take the negative of LeastSeen. *We expect this AF to perform poorly, by the logic that more expected matchups will tend to have lesser information content for the Emulator.*
- **LikeliestWin**: take the negative of LikeliestDraw (7). *We expect this AF to perform poorly, by the logic that more "obvious" outcomes will also convey lesser information.*
- **TSQuality**: take TrueSkill's built-in `quality()` metric. The TrueSkill paper describes this as "*the draw probability relative to the highest possible draw probability in the limit* $\epsilon \to 0$", which in practice can be thought of as the expectation of draw probability when treating team skill as a Gaussian Process. In other words, a trade-off between an Emulator's confidence in player skill and its confidence in match outcome.

E. Data

For our experiments, we scraped a large database of professional and semi-professional CS:GO matches from hltv.org. For the analysis, we used all matches with a greater than or equal to "1 star" team rating between 2017-2022, for a total of 9,929 matches.

As the modelling of skill ratings over time is out of scope of this work, we split the dataset into a training and evaluation set using a random 50/50 split. Matches used for fitting by the AFs are taken from the training set, and evaluation is always done by analysing emulator performance on the entire validation set.

F. Trueskill Sensitivity Analysis

Algorithms such as TrueSkill naturally have several parameters that control their behaviour and thus performance (in the case of Trueskill, there are four primary parameters as described above: μ, σ, β, τ). While the defaults are generally considered to "lead to reasonable dynamics" [12], there is no current literature exploring the selection of these parameters.

Hence, we here perform a sensitivity analysis against our dataset to determine which parameters have the largest effect on performance, and whether the defaults provide reasonable results on our dataset. Since all parameters are set relative to μ, we perform a logarithmic grid search across each pair of parameters in σ, β, τ, ± 1 order of magnitude from the default.

Since all other parameters are defined in terms of μ, we leave the default $\mu = 25$, and vary two parameters at a time out of σ, β, τ in the form of a logarithmic grid search. We then use the **GPy** Python library [13], to fit a Gaussian Process with an RBF kernel to the results:

$$k(x,x') = \frac{1}{1000}\exp\left(-\frac{||x-x'||^2}{0.5}\right)$$

with a mean performance prior of 60%. Using a GP allows us to smooth the noisy results without needing to repeat every run many times to get an average.

Training	Emulator	Random	MostSeen (III-D.7)	LeastSeen (10)	LikeliestWin (III-D.7)	LikeliestDraw (7)	CrossEntropy (8)	Weighted (9)	TSQuality (III-D.7)
500 matches	**Random**	50.1%	50.0%	50.0%	49.9%	**50.1%**	50.0%	50.0%	-
	WinRate	58.8%	54.8%	58.9%	58.1%	58.3%	**59.0%**	59.0%	-
	Elo	59.3%	55.4%	59.7%	56.1%	**59.8%**	59.4%	59.5%	-
	Glicko2	60.1%	58.3%	59.4%	58.8%	60.5%	60.3%	**61.2%**	-
	TrueSkill	59.1%	57.4%	59.0%	58.2%	59.2%	58.9%	**60.4%**	56.5%
	TSPlayers	59.6%	56.3%	60.6%	56.5%	60.9%	61.0%	**62.1%**	57.8%
	Average	59.4%	56.4%	59.5%	57.5%	59.7%	59.7%	**60.4%**	-
1000 matches	**Random**	50.1%	50.1%	**50.1%**	49.9%	49.9%	50.0%	50.0%	-
	WinRate	60.5%	55.9%	60.3%	59.9%	59.4%	**60.8%**	60.7%	-
	Elo	61.0%	56.6%	60.9%	57.8%	61.4%	61.0%	**62.2%**	-
	Glicko2	61.4%	60.1%	60.3%	59.5%	62.0%	61.2%	**62.4%**	-
	TrueSkill	60.8%	59.1%	59.7%	59.5%	60.9%	60.1%	**61.8%**	58.2%
	TSPlayers	60.7%	57.7%	60.8%	57.5%	62.8%	61.8%	**63.2%**	60.1%
	Average	60.9%	57.9%	60.4%	58.9%	61.3%	61.0%	**62.0%**	-
2000 matches	**Random**	**50.1%**	50.1%	50.0%	50.0%	50.0%	50.1%	50.0%	-
	WinRate	61.9%	57.0%	62.0%	61.3%	60.2%	**62.3%**	61.3%	-
	Elo	62.3%	58.0%	62.0%	59.8%	62.3%	62.2%	**62.8%**	-
	Glicko2	62.6%	61.4%	62.2%	60.5%	63.0%	63.1%	**63.1%**	-
	TrueSkill	62.2%	60.8%	61.9%	60.9%	62.2%	61.8%	**62.9%**	60.4%
	TSPlayers	61.8%	59.2%	60.7%	59.4%	63.9%	62.6%	**64.1%**	62.9%
	Average	62.2%	59.3%	61.8%	60.4%	62.3%	62.4%	**62.8%**	-

TABLE I: Evaluation accuracy after 500 to 2000 training matches (roughly 1-4 matches per team), for each Emulator and Aquisition Function combination. Each point is an average of 100 runs, with a epistemic confidence interval $< 0.1\%$. The best acquisition function for each emulator is bolded. Average values exclude the Random Emulator.

.

IV. Results & Analysis

Table I summarises the results across each combination of emulator and acquisition function. As expected, we see that as more matches are selected to fit the emulator, the overall performance of that emulator increases.

The Weighted acquisition function (parameterised with $\alpha = 1, \beta = 1$) introduced in this work produces the best overall performance for all emulators based on existing rating systems, with the biggest gains seen after only 500 matches of training. The Random AF provides a baseline to which we can compare the other acquisition functions; we see that the MostSeen and LikeliestWin AFs provide significantly worse performance for the same number of training matches. This is inline with theory, as we are essentially feeding the emulator the results of matches it is already sure about. However, this highlights the importance of choosing a good acquisition function when minimal data is available.

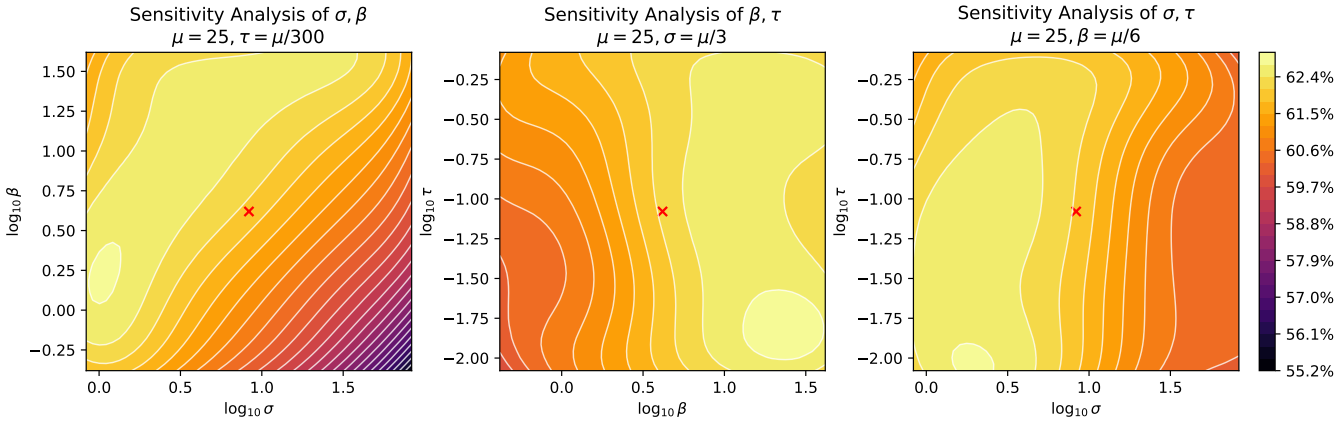
The LikeliestDraw and CrossEntropy AFs select matchups which the emulator is unsure about, and this yields an improvement of 1-1.5% over random matchups from the dataset. However, we see that while the Weighted AF consistently provides the best performance for Elo, The more basic WinRate emulator favours the CrossEntropy AF.

The TSQuality AF, which uses TrueSkill's own internal match quality for selection, underperforms in comparison to other AFs, even when compared to random sampling.

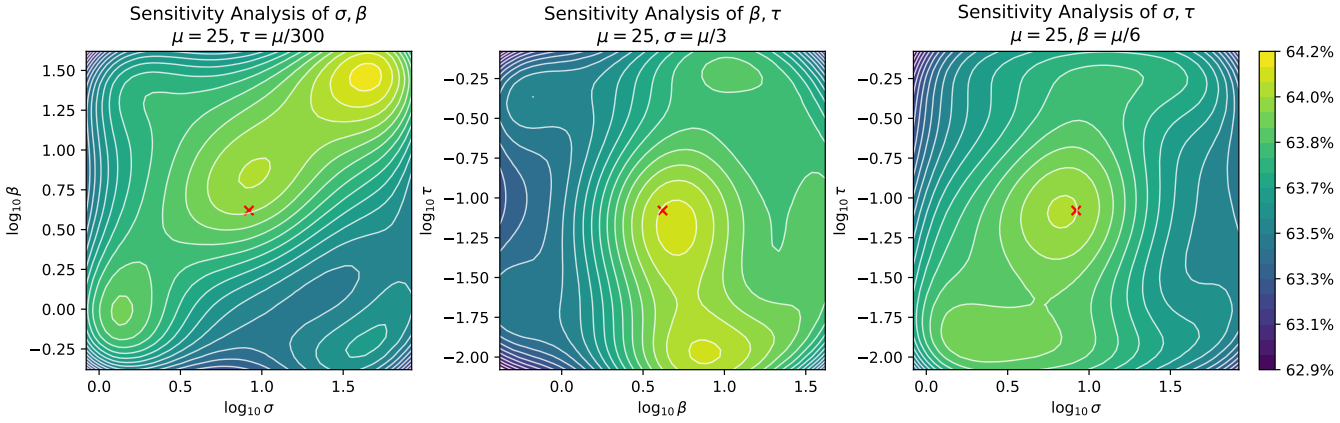
We can also analyse the comparative behaviour of different Emulators. We find that among the team-based skill rating systems, Glicko2 provides better performance across the board, with greater gains with fewer training matches. This is surprising, as TrueSkill was introduced to address the shortcomings of Glicko2.

However, one of the main benefits that TrueSkill brings is the ability to generalise beyond 1v1 matchups, and the benefit of this is seen when we use 5v5 matchups in TrueSkill with per-player skill ratings. The TSPlayers emulator beats all other approaches by around 1%, giving us the best achieved average accuracy of 64.1%.

Overall, we see that the choice of acquisition function and emulator can be largely decoupled: player-based TrueSkill excels as an emulator throughout, and the Weighted acquisition function provides the best results across emulators.



(a) TrueSkillEmulator



(b) TrueSkillPlayersEmulator

Fig. 2: Sensitivity analysis for our TrueSkill emulators, after 2000 training matches selected by the **LikeliestDraw** (7) acquisition function. The red × shows the default TrueSkill parameters, with the plot being ± 1 order of magnitude. Note that the scale in (a) is a much larger range than in (b).

A. *Trueskill Sensitivity Analysis*

Figure 2 shows us the performance of each of our TrueSkill-emulators, as the algorithm parameters are varied. We can draw a number of interesting conclusions from this:

1) β and σ seem like the most important parameters, with τ having a much smaller effect on performance. We confirm that the ratio between β and σ is very important ($\beta = \sigma/2$ is the default) - performance remains similar if the ratio is maintained. We find that the optimal ratio is $\beta = \sigma/0.5$ and $\beta = \sigma/1.6$ for the two emulators.
2) In **TrueSkillEmulator** (with a single skill rating for a given team), the combination of a high σ and low β dramatically reduces the performance of the algorithm. This effect is also seen (less drastically) in **TrueSkillPlayersEmulator**, which maintains a separate skill rating for each of the team's 5 players.
3) The **TrueSkillPlayersEmulator** is much more robust to changes in parameters (a 1.3% range versus 7.5%). This suggests that the TrueSkill algorithm is more stable with 5v5 matches rather than 1v1 matches.
4) That being said, we see that the default values given achieve close to optimal performance in both cases! We can make slight gains in the per-team emulator, but the cost of modifying parameters in the wrong direction is much greater.

B. *Accuracy over training*

Figure 3 shows the performance of emulators using the Random and Weighted acquisition function, and the emulators are trained until they run out of matches.

We find that the engineered AFs reach a plateau of performance much faster than picking matches at random: matches are picked that try to maximise the speed at which the emulators can learn. In the case of the Weighted AF, we actually see overall accuracy actually decreases towards the end of training. This is because a good acquisition function selects "high-quality" matches (for some notion of high quality), which inevitably means that the 'low-quality' matches are all fed to the emulator at the end; potentially an unrepresentative sample that skews ratings. We see that this behaviour is much more prominent in the TrueSkillPlayers emulator.

Note that by intentionally training on an AF-selected subset of the dataset, we avoid this behaviour in Table I, as we report results there only after training on a subset of the dataset. This effect is a limitation of our training method, which limits matchups to a subset of actually

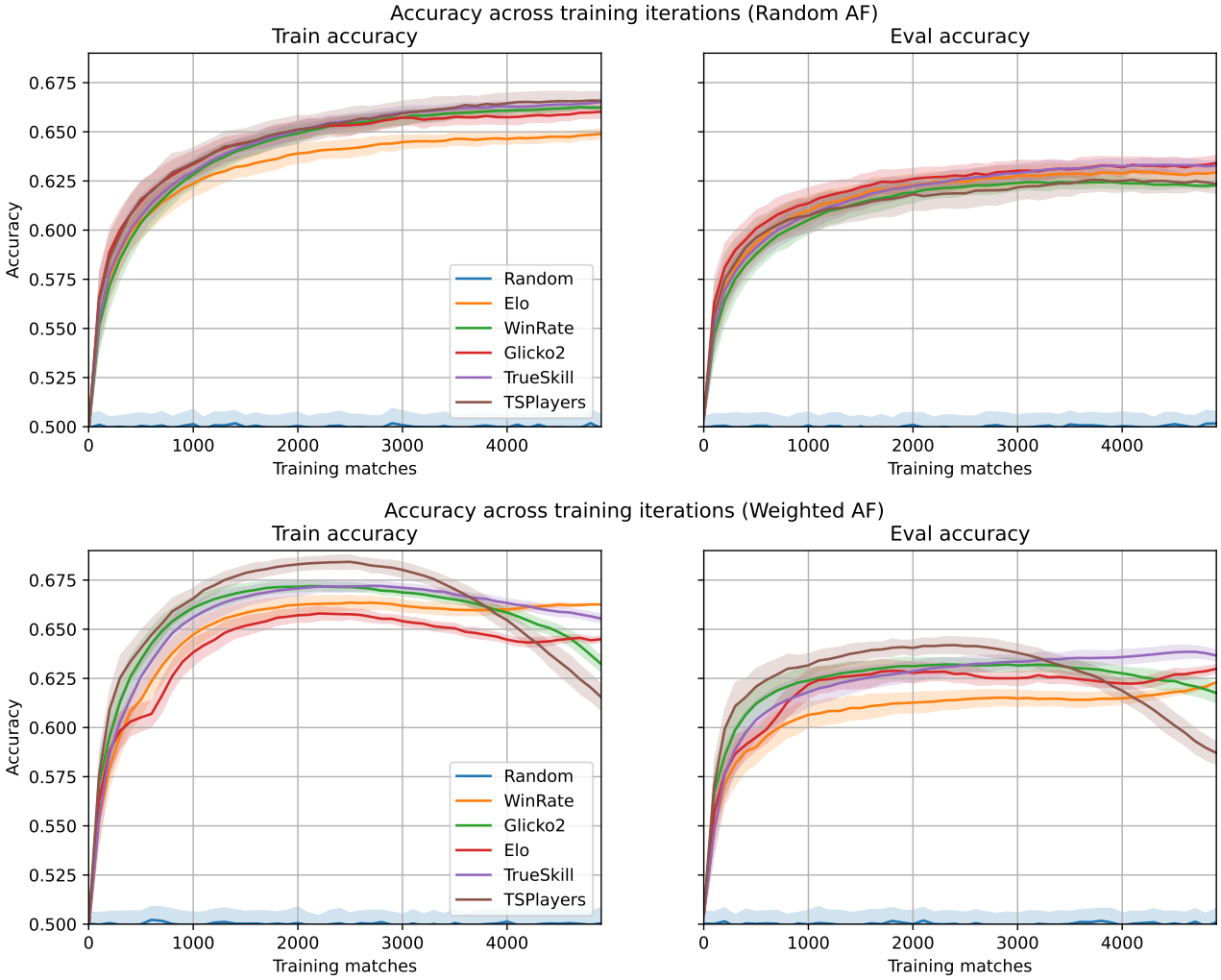


Fig. 3: The training and evaluation accuracy across emulators, using a Random and Weighted 9 AF. As the dataset is exhausted, both acquisition functions train on all matches, in a different order. Error bars show $\pm 1\sigma$ of **aleatoric** uncertainty; the variance between individual runs of the Simulator.

observed matchups (rather than letting the model choose any possible matchup), as opposed to a fundamental property of the emulators.

Overall, the amount of variance observed run-to-run was a surprising result. This means that on any given run (especially with the Random AF), any two skill rating systems could achieve comparable performance - however, there are significant differences between *average* performance across many runs.

V. Discussion

A. Limitations

While the goal of this work was to accurately model each method's performance within an environment where it can choose which match is played next, there are several limitations to our work:

- Not every matchup is selectable by the emulator during training - it can only choose matchups that actually occurred in the dataset.
- Matches are not presented to the algorithm in *time-order*, meaning that the ability of each emulator to model *time-varying* skill was not tested. This was done to avoid dramatically limiting match choice.
- We test only on a single dataset, based on professional CS:GO matches. Therefore, it is unclear how well the results transfer to amateur matches, or other video games which rely on similar matchmaking systems.
- We have only evaluated Emulators in their accuracy at predicting non-draw outcomes. It may be insightful to compute the log-loss of each emulator to reward stronger beliefs & to factor in drawed games.
- Acquisition Functions are evaluated on their ability to produce accurate skill ratings (as measured by match prediction accuracy), but there are many other metrics. For example, the TrueSkill Quality metric [8] (which did not perform well in this work) is designed to try to maximise the probability of a draw, under the heuristic that evenly matched games are the most fun; the best match for players may not be the match that allows the skill rating system to learn the most.

B. Future Work

Perhaps the primary way in which our work could be extended is through the use of larger datasets (such as amateur matchmaking data if such data were obtainable), which would allow for greatly reduced aleatoric uncertainty in results. Another obvious extension would be to analyse several different games and determine whether the choice of system is game-specific or if e.g. TrueSkill is always superior to Elo.

Other avenues include:

- **Parameter-tuned acquisition functions.** To further improve the performance of our Weighted AF (9), we could consider tuning its parameters *per emulator*, such as through Bayesian optimisation. Furthermore, we could explore the use of highly parameterized AFs that make deeper assumptions about the emulator and data, such as modeling the impact of network effects on different teams; or even the relationship between factors like draw probability and team count as a Gaussian process to achieve closer to optimal results.
- **Exploring non-tournament matchmaking settings.** To provide a more realistic matchmaking experience, we could simulate a setting whereby the matchmaking function must balance the usefulness of the emulator, fairness of the match, and queue waiting times, rather than assuming that all teams are always available for matches. This approach would offer a more dynamic and practical solution.
- **Next-generation emulators**. Microsoft recently published TrueSkill2 [9], which claims to improve match prediction accuracy on Halo 5 from 52% to 68%. This was out of scope in this paper as no open-source implementations currently exist. Alternatives that could be evaluated include OpenSkill. [14].

VI. Conclusion

In this work, we performed an thorough analysis on the performance of different skill rating systems and matchmaking algorithms when applied to a real-world dataset of professional CS:GO matches. The core novelty of our work is the use of a surrogate modelling architecture to evaluate a skill rating system's effect on matchmaking and its recursive effect on future skill ratings, while retaining the use of real-world data. Our architecture is released as a Python library to allow for future extension and evaluation.

We draw several conclusions from our analysis: that the default parameters of TrueSkill yield close to optimal results, that a selection of emulator and AF can be made largely independently, that a 5v5 TrueSkill and Weighted AF provide the best results, and that optimal performance can be reached with surprisingly few matches, with only marginal gains between skill rating systems. Larger datasets and an evaluation of generalisation across games could allow for more robust future analysis.

VII. Source Code

Our **skillbench** framework, including datasets and all the emulators used in this work is available at `https://github.com/mgm52/skillbench`.

References

[1] "Steamcharts: Counter-strike: Global offensive." [Online]. Available: https://steamcharts.com/app/730

[2] M. E. Glickman, "Example of the glicko-2 system," 2022. [Online]. Available: https://www.glicko.net/glicko/glicko2.pdf

[3] "r/globaloffensive - comment by u/vitaliy_valve on "the ultimate guide to csgo ranking"." [Online]. Available: https://www.reddit.com/r/GlobalOffensive/comments/2g3r4c/the_ultimate_guide_to_csgo_ranking/ckfhfir/

[4] P. Mackey, "The summoner's guidebook: Getting out of elo hell," *Engadget*, 2012. [Online]. Available: https://www.engadget.com/2012-05-10-the-summoners-guidebook-getting-out-of-elo-hell.html

[5] A. E. Elo, "The proposed uscf rating system," *Chess Life*, vol. August 1967. [Online]. Available: http://uscf1-nyc1.aodhosting.com/CL-AND-CR-ALL/CL-ALL/1967/1967_08.pdf

[6] Ollipasanen, "An analysis of the rating system in faceit (cs:go)." [Online]. Available: https://github.com/ollipasanen/faceit-rating-system-analysis

[7] M. E. Glickman, "The glicko system," 1995.

[8] R. Herbrich, T. Minka, and T. Graepel, "Trueskill^{tm}: A bayesian skill rating system," in *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, B. Schölkopf, J. C. Platt, and T. Hofmann, Eds. MIT Press, 2006, pp. 569–576. [Online]. Available: https://proceedings.neurips.cc/paper/2006/hash/f44ee263952e65b3610b8ba51229d1f9-Abstract.html

[9] T. Minka, R. Cleven, and Y. Zaykov, "Trueskill 2: An improved bayesian skill rating system," Microsoft, Tech. Rep. MSR-TR-2018-8, March 2018. [Online]. Available: https://www.microsoft.com/en-us/research/publication/trueskill-2-improved-bayesian-skill-rating-system/

[10] A. Dehpanah, M. F. Ghori, J. Gemmell, and B. Mobasher, "The evaluation of rating systems in online free-for-all games," *CoRR*, vol. abs/2008.06787, 2020. [Online]. Available: https://arxiv.org/abs/2008.06787

[11] I. Makarov, D. Savostyanov, B. Litvyakov, and D. I. Ignatov, "Predicting winning team and probabilistic ratings in "dota 2" and "counter-strike: Global offensive" video games," in *Analysis of Images, Social Networks and Texts - 6th International Conference, AIST 2017, Moscow, Russia, July 27-29, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, W. M. P. van der Aalst, D. I. Ignatov, M. Y. Khachay, S. O. Kuznetsov, V. S. Lempitsky, I. A. Lomazova, N. V. Loukachevitch, A. Napoli, A. Panchenko, P. M. Pardalos, A. V. Savchenko, and S. Wasserman, Eds., vol. 10716. Springer, 2017, pp. 183–196. [Online]. Available: https://doi.org/10.1007/978-3-319-73013-4_17

[12] J. Mozer, "The math behind trueskill," 2011. [Online]. Available: https://www.moserware.com/assets/computing-your-skill/The%20Math%20Behind%20TrueSkill.pdf

[13] GPy, "GPy: A gaussian process framework in python," http://github.com/SheffieldML/GPy, since 2012.

[14] R. C. Weng and C.-J. Lin, "A bayesian approximation method for online ranking," *Journal of Machine Learning Research*, vol. 12, no. 9, pp. 267–300, 2011. [Online]. Available: http://jmlr.org/papers/v12/weng11a.html