TrueSkill 2: An improved Bayesian skill rating system

Tom Minka, Microsoft Research
Ryan Cleven, The Coalition
Yordan Zaykov, Microsoft Research

March 22, 2018

Abstract

Online multiplayer games, such as *Gears of War* and *Halo*, use skill-based matchmaking to give players fair and enjoyable matches. They depend on a skill rating system to infer accurate player skills from historical data. TrueSkill is a popular and effective skill rating system, working from only the winner and loser of each game. This paper presents an extension to TrueSkill that incorporates additional information that is readily available in online shooters, such as player experience, membership in a squad, the number of kills a player scored, tendency to quit, and skill in other game modes. This extension, which we call TrueSkill2, is shown to significantly improve the accuracy of skill ratings computed from *Halo 5* matches. TrueSkill2 predicts historical match outcomes with 68% accuracy, compared to 52% accuracy for TrueSkill.

1 Introduction

When a player wants to play an online multiplayer game, such as *Halo* or *Gears of War*, they join a queue of waiting players, and a matchmaking service decides who they will play with. The matchmaking service makes its decision based on several criteria, including geographic location and skill rating. Our goal is to improve the fairness of matches by improving the accuracy of the skill ratings flowing into the matchmaking service.

The skill rating of a player is an estimate of their ability to win the next match, based on the results of their previous matches. A typical match result lists the players involved, their team assignments, the length of the match, how long each player played, and the final score of each team. A skill rating system must make assumptions about how these match results relate to player skill. We follow previous work by representing these assumptions as a probabilistic generative model, that is, a process that generates random player skills and random match results from the skills. Bayesian inference in this generative model gives the optimal skill ratings under the assumptions.

A skill rating system is only as good as its underlying assumptions. This paper focuses on making skill ratings more accurate by making better assumptions. We describe the process by which we arrived at our generative model, how to evaluate the model, and how to estimate the parameters of the model.

A variety of different generative models can be found in previous work. For example, Dangauthier et al. [2008] extended TrueSkill for two-player games by giving each player an additional number describing their ability to force draws, which could evolve over time. Glickman [2001] used a model for two-player games where each player has an additional number describing their skill volatility, which could evolve over time. Chen et al. [2016] used a model where each player had a different (but unchanging) skill rating for each character that they could play in the game. Menke et al. [2006] used a model that accounted for map and server effects on the outcome of matches. Chen and Joachims [2016] used a model where 'skill rating' was a vector of numbers, to allow intransitive dominances between players.

Unfortunately, none of these models have the qualities needed by a modern game studio. After consulting with the makers of *Gears of War* and *Halo*, we have found that their top priorities are:

1. Support for team games. The system should support matches with any number of teams and any number of players on each team.

2. Changeable skill ratings. It must be possible for a player's skill rating to change, no matter how many matches they have played in the past. This ensures that players receive meaningful feedback on their performance.

3. Compatibility with an existing matchmaker. Existing matchmakers assume skill is described by a single number. Players can easily understand a single ordered skill ranking.

4. Aligned incentives. The skill rating system should create incentives that align with the spirit of the game. For example, consider a team game where players cooperate to achieve a goal. An improperly-designed skill rating system could encourage players to impede their teammates. As another example, consider a system that increased skill rating according to the number of times a player healed themselves. This creates an incentive for a player to repeatedly injure themselves so that they could be healed. In general, the more control that a player has over a quantity, the less useful it is for skill rating.

5. Minimal training data requirements. The most important time to have good matchmaking is immediately after the launch of a game. Unfortunately, this is also when there is the least amount of data available to train a model. Even if the game has a large player base, there tends to be a small amount of training data per player.

6. Low computational cost. Skill updates are done on servers hosted by the game studio, and skill ratings are stored in a database hosted by the game studio. This means that skill representations should be small and updates should be cheap.

7. Minimal tuning. Game studios generally do not have personnel with the knowledge and free time available to tune the skill rating system after launch. But they do make plenty of other changes to the game, such as adding new game modes, new player abilities, and

re-balancing game mechanics. The skill rating system needs to automatically adapt to these kinds of changes.

Our solution is to start with the TrueSkill model, which already enjoys properties 1–6, and make judicious changes that preserve these properties. A skill rating in TrueSkill2 is a single number with the same meaning (in terms of win rate) as a TrueSkill rating. Thus any matchmaker that accepts classic TrueSkill ratings also accepts TrueSkill2 ratings. TrueSkill2 only changes the way that skills are inferred from historical data. Property 7 is achieved by performing automatic parameter estimation over a batch of historical data. TrueSkill2 operates in two modes: an online mode that only propagates skill ratings forward in time, and a batch mode that infers parameters and skills over all time (also known as TrueSkill Through Time). Both modes are based on the same probabilistic generative model—they only correspond to different approximations to the Bayesian posterior.

Briefly, the classic TrueSkill model makes the following assumptions:

1. Each player has a latent skill value that represents their expected contribution to a team. A player's performance in a game is a noisy sample of their skill.

2. The performance of a team is the weighted sum of the performances of its players, where the weight is the fraction of time the player spent on the team.

3. If a team's performance is greater than the other team by a certain margin, the team wins. Otherwise, the game is a draw. When learning skills from data, only the team win/loss information is used (not scores).

4. Player skills evolve over time according to a random walk. An increase or decrease in skill is assumed equally likely.

5. A player's skill in a game mode is assumed independent of their skill in all other modes.

The TrueSkill2 model modifies the classic model in the following ways:

1. A player's latent skill is inferred from their individual statistics such as kill and death counts, in addition to team win/loss.

2. When a player quits or drops out in the middle of a game, it is treated as a surrender and their skill is updated as if they lost a game (regardless of actual outcome).

3. A player's skill in a game mode is assumed statistically correlated with their skill in other modes, so that when a player starts a new mode, their skill rating from other modes is borrowed.

4. The random walk of player skill is assumed biased toward increasing skill, with larger bias during the first matches a player plays in a game mode.

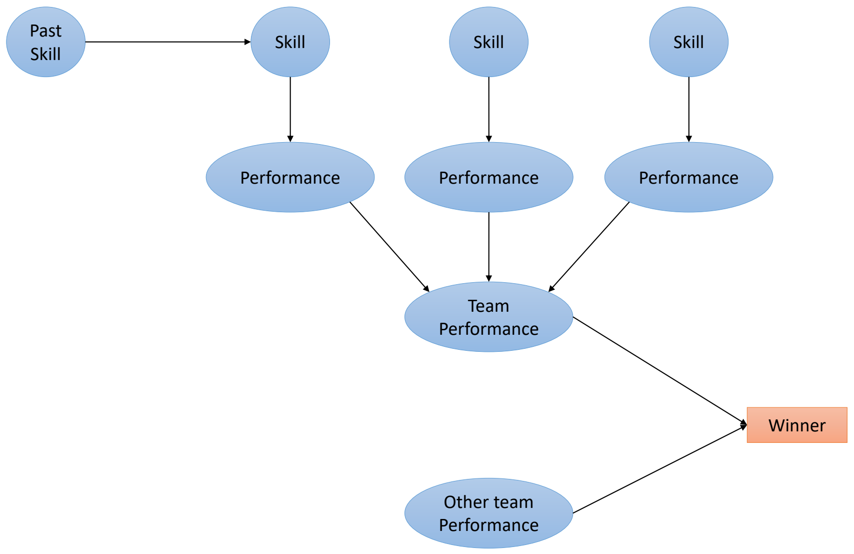
5. When a player is part of a squad, their performance is assumed to be better than normal.

2 Classic TrueSkill

This section describes the classic TrueSkill model in detail, and sets the notation used in the rest of the paper. In the classic TrueSkill model, the data is assumed to consists of a sequence of match results, each of which lists the players involved, their team assignments, the start time and length of the match, how long each player played, and the final score of each team. The team scores are only used to determine an ordering of best to worst, including ties.

To interpret this data, TrueSkill assumes a probabilistic generative model of match results. This means that, instead of experimenting with different formulas for updating a player's skill rating and hoping to stumble upon the right one, we construct an intuitive random process for generating skills and match results. This model can be checked against data and refined based on any discrepancies we encounter. When we are happy with the fit to data, we apply Bayesian inference to obtain an optimal algorithm for updating skill ratings.

The generation process is illustrated below, where boxes denote variables that are observed in the data:



The first step of the generation process is generating the player skills. Each player i is assumed to have a real-valued skill, denoted skill_i^t, at time t. The initial skill of a player is drawn from a Gaussian distribution with mean m_0 and variance v_0. This sampling process is denoted

$$\text{skill}_i^{t_0} \sim \mathcal{N}(m_0, v_0) \tag{1}$$

where t_0 is the time of the player's first match and (m_0, v_0) are tunable parameters. After each match, the player's skill changes by a random amount, also drawn from a Gaussian:

$$\text{skill}_i^{t+L} \sim \mathcal{N}(\text{skill}_i^t, \gamma^2) \qquad \text{after a match of length } L \tag{2}$$

(Note that we are talking about the player's actual skill level, not their skill rating assigned by the system.) In the original paper [Herbrich et al., 2007], this was the only way that a player's skill could change. A later paper [Dangauthier et al., 2008] made the alternative assumption (also made by Glickman [1999]) that a player's skill changed with the passage of time, rather than from playing matches:

$$\text{skill}_i^{t'} \sim \mathcal{N}(\text{skill}_i^t, \tau^2(t'-t)) \qquad \text{where } t' > t \tag{3}$$

Both of these assumptions make sense, since a player gains experience from each match they play, and their sharpness decreases over time without play. Therefore we include both types of change in the "classic" model.

The next step of the generation process is generating the match results from the player skills. TrueSkill assumes that each player has a real-valued performance in each match, drawn according to:

$$\text{perf}_i^t \sim \mathcal{N}(\text{skill}_i^t, \beta^2) \tag{4}$$

where β is a tunable parameter that reflects the amount of randomness in the game. In the following, we drop the dependence of perf_i on t since we are always referring to a specific match. In a two-player game without draws, the winner is the player with the largest performance. If draws are possible, then the model includes a parameter $\epsilon > 0$, and declares a match to be drawn if the difference between performances is less than or equal to ϵ:

$\text{perf}_1 - \text{perf}_2 > \epsilon$	player 1 wins
$\lvert\text{perf}_1 - \text{perf}_2\rvert \le \epsilon$	draw
$\text{perf}_1 - \text{perf}_2 < -\epsilon$	player 2 wins

In a game with teams, the performance of each team is defined to be the weighted sum of the player performances, where the weights are the fraction of the match time that they played on the team:

$$\text{perf}_{\text{team}} = \sum_{i \in \text{team}} \text{perf}_i \frac{\text{timePlayed}_i}{L} \tag{5}$$

The outcome between two teams is decided in the same way as above. If a game has more than two teams and no possibility of draws, then we sort the teams by their performances to get their ordering.

If a game has more than two teams and allows draws, then things get more complicated. The original paper [Herbrich et al., 2007] never gave a generative model for this situation, instead it only provided a factor graph. Here we give a generative model that captures the spirit of the factor graph. The idea is that if the performance difference between two teams is within ϵ, then the teams must have the same rank. Teams that are not forced to have the same rank according to this rule have different ranks. Given team performances $\text{perf}_1, \text{perf}_2, ..., \text{perf}_n$ in descending order, the following pseudo-code produces the rank of each team:

1. Initialize rank $\leftarrow 1$. Team 1 has rank 1.
2. For team $= 2, ..., n$:
    (a) If $\text{perf}_{\text{team}} < \text{perf}_{\text{team}-1} - \epsilon$, then rank $\leftarrow$ rank $+ 1$.
    (b) Assign rank to team.

In practice, this model for draws gives similar skill ratings as the approach in Herbrich et al. [2007]. Since the approach in Herbrich et al. [2007] is simpler to implement, we used their approach.

There is a scale ambiguity in this model, in the sense that if we scale all skill variables by 2, along with β and ϵ, then the distribution over team orderings is the same. We resolve this ambiguity by fixing $\beta = 1$. This means that a skill gap of 1 always corresponds to the same win rate. The amount of luck in a game is reflected in the size of v_0, where smaller v_0 implies more luck is involved.

In a two-player game, there is also a shift ambiguity in the model, in the sense that if we add 2 to all skills, then the distribution over outcomes is the same. Thus m_0 is irrelevant in such games. However, in a team game where teams may have different sizes, for example due to players dropping out, m_0 is relevant since it represents the value of having a teammate.

Note that this model does not generate the entire match result. It generates the ordering of teams given the players in the match, their team assignments, the length of the match, and the length of time each player played. It does not generate the set of the players that will be in a match, the team assignments, the time of the match, the length of the match, or the length of time each player played. However, most of these things are correlated with skill. For example, a person who plays in the middle of the day is more likely to be a dedicated player with higher skill. Due to skill-based matchmaking, a player on the opposite team as a skilled player is more likely to be skilled themselves. From a purely statistical perspective, including these correlations in the model would give better skill estimates. But TrueSkill does not do this, because it would create undesirable incentives for players who want to increase their skill rating. The incentive should be to play the game well, not to queue up at particular times of day or with particular teammates.

The fact that TrueSkill does not model the length of time a player played is questionable. If a player tends to quit in the middle of a match, then this should arguably be reflected in their skill rating. On the other hand, if a player joins a match in progress, this shorter play time should not be reflected in their skill rating. TrueSkill treats both cases the same way. If a player quits early in the match, then timePlayed_i will be small, so the player will have a negligible change to their skill rating, which may even be positive if their team happens to win.

One way to penalize quitters is to artificially set timePlayed_i to the full length of the game. This penalizes the quitter, but also harshly penalizes players on the same team. This becomes worse when a new player joins to replace the quitter. Since the quitter is considered to have played for the full time, this team is considered to have more players than normal, and therefore expected to win. If the team doesn't win, all remaining players, including the late joiner, get large reductions in skill rating. Therefore we don't recommend this approach for team games. TrueSkill2 provides a better way to penalize quitters, as discussed in section 9.

3 Computing skill ratings

Given a probabilistic generative model for match results, skill ratings are computed by Bayesian inference. Formally, the model defines a joint distribution over player skills and match results, conditional on the unmodeled aspects of each match (which we will abbreviate as *match conditions*). The distribution over player skills comes from Bayes' rule:

$$p(\text{skills}|\text{results}, \text{conditions}) = \frac{p(\text{results}|\text{skills}, \text{conditions})p(\text{skills})}{p(\text{results}|\text{conditions})} \tag{6}$$

In practice, we don't compute the entire joint distribution over skills, but only the marginal distribution of skill_i^t, the skill of player i at time t, for all i and t. This distribution is approximated by a Gaussian, and the mean of that Gaussian is taken as the player's skill rating.

There are a variety of algorithms available for performing Bayesian inference, but this model is particularly suited to Expectation Propagation [Minka, 2001]. Herbrich et al. [2007] and Dangauthier et al. [2008] used the Expectation Propagation algorithm and gave a factor graph with message equations. In this work, we used the Infer.NET framework [Minka et al., 2014b] to generate code for Expectation Propagation automatically, based on our model description. This generated code performs message-passing on a factor graph, as described in Herbrich et al. [2007], and produces the same results as in that paper. Infer.NET generated both the online and batch versions of the inference algorithm. In fact, TrueSkill Through Time is one of the examples in the Infer.NET documentation [Minka et al., 2014a].

To generate the online TrueSkill updates, we defined a model in Infer.NET consisting of a single match. The prior distributions on the player skills, the match conditions, model parameters, and the team ranks were marked as observed variables, and the player skills were marked as variables to infer. Infer.NET then generated code where the observed variables were the input, and the inferred skill distributions were the output. This code is used to compute skill ratings in the following way:

1. Initialize all player skill distributions to a Gaussian with mean m_0 and variance v_0.
2. For each match result in order of start time:
   (a) Look up the current skill distribution for each player in the match.
   (b) Increase the variance of each skill distribution according to the amount of time elapsed since the player's last match and equation (3), to represent a possible change in skill due to their absence from the game.
   (c) Invoke the inference code on the match result and skill distributions, to get a new set of skill distributions. These represent each player's skill during the match.

(d) Increase the variance of each skill distribution according to (2), to represent a possible change in skill as a result of playing the match.

(e) Store the new skill distribution for each player in the match.

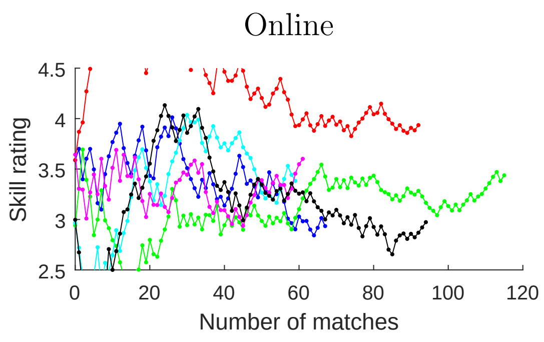
The online TrueSkill updates are useful as a fast, real-time approximation to the posterior distribution over skills. But the most accurate skill ratings come from processing many matches together as a batch. To see why, consider 5 matches between 6 players, played in this order:

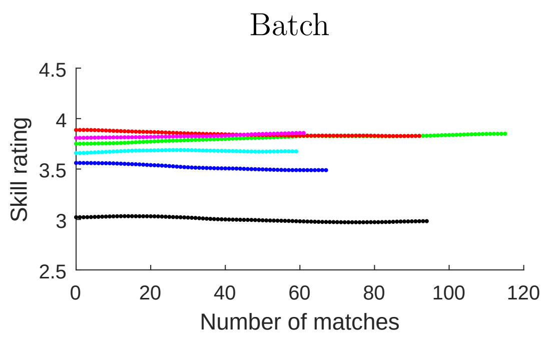
1. A beats B
2. C beats D
3. E beats F
4. B beats C
5. D beats E

All players start with the same skill distribution. After the first 3 matches, players (A,C,E) have all won 1 match against equivalent opponents, so the online updates will give them the same skill distribution. Similarly, players (B,D,F) get the same skill distribution. In the last two matches, B and D started with the same skill distribution, and beat opponents with the same skill distribution, so they end up with the same final skill distribution. Similarly, C and E end up with the same skill distribution. But this is clearly wrong. If we consider all matches together, the players should be ordered $A > B > C > D > E > F$.

To generate the batch TrueSkill algorithm, we defined a model in Infer.NET that included the entire generation process described in section 2. Each player had a skill variable skill_i^t for every match they played. This is many more variables than Dangauthier et al. [2008], who used one skill variable for each *year* of play. The generated code takes as input a list of match results and outputs a Gaussian distribution for each skill_i^t variable. The handling of model parameters is discussed in section 4.

The difference between online and batch inference is illustrated on real data below.





The data was 700k *Halo 5 Slayer* matches, and skill ratings were computed for all players. Out of this large set of players, the figures show the skill ratings of the same six players, versus the number of matches each one played. The skill ratings from batch immediately stand out as being smoother, but more importantly they give a different final ordering of players, with red below green and black far below blue.

4 Parameter estimation

The classic TrueSkill model has tunable parameters $(m_0, v_0, \gamma, \tau, \beta, \epsilon)$ for each game mode. To make the skill rating system easy to use by a game studio, these should be learned from data. Previous work tuned these parameters by sweeping over a grid, which is only practical for a small number of parameters on a small dataset. Since our goal is to extend the model with more parameters, and run on large datasets, we needed a different solution. Our approach is to treat the model parameters as random variables in the model, giving them prior distributions, and inferring them jointly with the skills. In practice, we gave the parameters broad priors, so they were mainly determined by the data.

Our initial approach was to infer the parameters with Expectation Propagation. Unfortunately, most parameters are involved in generating every match result, which means that, in the factor graph, a given parameter will be connected to millions of factors. This leads to a high memory requirement as a large number of messages need to be stored. In addition, the posterior distribution for some parameters is highly non-Gaussian, leading to poor approximations and sometimes non-convergence of the algorithm.

Our final approach exploits the fact that each parameter connects to a million factors. In such a model, the parameter will tend to have a sharp posterior distribution. Therefore we chose to drop the uncertainty in the parameter posteriors, and approximate them as point masses during each iteration. This not only saves memory on messages, but also saves computation. At the end of each iteration, we update each parameter using Rprop [Riedmiller and Braun, 1993], where the gradient is computed by accumulating the Expectation Propagation messages into that parameter. The incoming messages are computed once and discarded. This procedure reaches reasonable parameter estimates in about 100 iterations, i.e. 100 forward and backward sweeps through the data. Note that this approximation breaks down for game modes with less than 1000 matches. For such modes, we used the parameters estimated from the most similar popular mode.

For the *Halo 5* data, typical parameter values are:

- $m_0 = 3$ (teammates are very important)
- $v_0 = 1.6$ (skills of new players only vary by 1 from the mean)
- $\gamma = 10^{-3}$ (skills change slowly)
- $\tau = 10^{-8}$ per minute, so that $\gamma/\tau = 10^5$ minutes of non-play are equivalent to one match
- $\beta = 1$ (by design)
- $\epsilon = 10^{-3}$ (draws are rare)

5 Metric-driven modelling

This section describes the approach that was used to develop the TrueSkill2 model. The motivation behind it is that a skill rating system should use the simplest model that meets the needs of a game studio, and we shouldn't have to spend large amounts of time searching for that model. The basic principle is that any change to the model must be motivated by a metric that demonstrates a gap between the model and reality. The metric must be meaningful to the application, and specific enough that it is clear where the model needs to be changed in order to improve the metric. We call this approach "metric-driven modelling," taking inspiration from test-driven development of computer software. The metrics encode a deep understanding of the application, while modelling is in principle a mechanical search for ways to improve the metrics.

A *metric* is a function averaged over a subset of test data. It can refer to the data, inferences from the model, or the difference between an inference and the data. Data subsets are defined using *features*. The metric designer's job is to determine the relevant subsets and comparisons. In principle, this does not require knowledge of the model. Metrics should be in data space, or more generally the external interface of the model, not in the parameter space.

For a generative model of match results, the key metric is how well it predicts the final ordering of teams. We call this the *predictive accuracy*. For matches with two teams, there are only 3 possible orderings. With more teams, the number of possible orderings grows exponentially. To get a meaningful and easily computable metric, we only ask the model to predict the identity of the winning team, or predict "draw" when there is a tie for first place. Thus the number of options is the number of teams plus 1.

Accuracy is useful for comparing models, but it doesn't tell you why errors were made. Is the model favoring certain kinds of teams or players? Does it penalize quitters too harshly, or not enough? Does it overestimate the skill of new players, or underestimate them? To answer these questions, we look at how often the model thinks a particular kind of team or player would win, compared to how often they actually win.

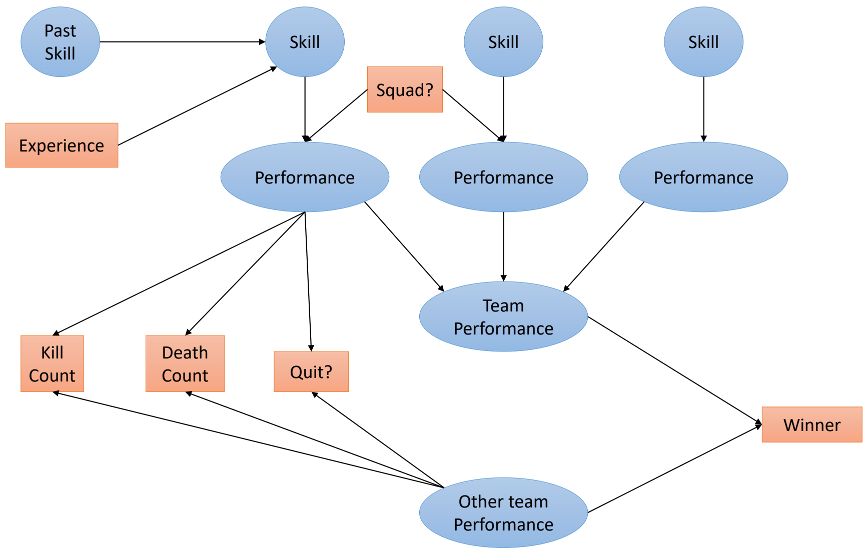
The dataset used to develop TrueSkill2 was the set of all matches played in Halo 5 since its release in 2015, provided to us by 343 Industries. Halo 5 has a variety of game modes, which span various numbers of teams and team sizes. This dataset contains millions of match results and millions of unique players. For the results shown in this paper, the training set consisted of 23 million games played just after release. The test set consisted of the next 3 million games played after the end of the training period. Because of the large size of this dataset, there is no need to report error bars in our summary statistics since they are tiny.

All metrics, including predictive accuracy, were measured in the following way. For each model, the parameters were tuned on the training set. Then the model was asked to predict the outcomes in the test set. The player skills were reset between the train and test phase. Resetting the player skills is not essential but made the implementation simpler.

The matches in the test set were sorted by start time and processed once in order. As each match was processed, we asked the model to predict the winning team and give a probability for this event. When making this prediction, only the team composition and squad membership are used—the length of the game, time each player played, completion status, and kill/death counts are not used. Then we updated the skills using the online updater with all of the information available at the end of the match. The percentage of correct predictions is the predictive accuracy, and the percentage of matches where a particular kind of team or player was predicted to win is its expected win rate.

Many of our metrics involve the expected win rate of a subpopulation of players. This is computed by recording the prediction for all matches, visiting each player in each match, and testing if the player meets the criteria. If so, the prediction for that player's team is added to the tally. The final metric is an average of the predictions for all players that met the criteria. Thus the subpopulation that we define only changes the subset of predictions that go into the average. It does not change how we make predictions or update skills. Skills are always updated using all matches.

After tuning model parameters on the training data, classic TrueSkill achieved a predictive accuracy of 52% on the test set. TrueSkill2 achieved a predictive accuracy of 68% on the test set. The next sections explain how this was accomplished. The figure below illustrates the new generative model in TrueSkill2, where boxes denote variables that are observed in the data (compare to section 2).



6 Squad offset

Many online multiplayer games give friends the option of queuing up together. We call such a grouping a *squad*. The matchmaker is expected to find a fair match where the entire squad is on one team. This is typically done by assuming that the skill rating of a squad is the sum of skill ratings of its players. We can test this assumption by measuring an appropriate metric. The metric we chose partitions players according to whether they queued up in a squad versus solo, and measures their actual win rate compared to their predicted win rate under the assumption. This was done regardless of game mode. The results on the test set are collected below:

Player Squad Size	Win Rate (%)	Expected by TrueSkill
1	47	47
2	50	50
3	50	50
4	57	52
5	53	49
6	56	48
7	62	49
8	68	49
9	70	54
≥10	89	67

We see that the assumption about squads is false. The skill rating of a squad should be larger than the sum of its players, and the effect grows with the size of the squad. This is a well-known phenomenon in online gaming. In fact, some playlists in *Halo 5* and *Gears of War 4* prohibit large squads in an attempt to provide fairer matches, even though this lowers the enjoyment of people who like to play with their friends. Other games address this by forcing squads to match with other squads. This has the problem of shrinking the matchmaking pool.

Instead of prohibiting squads or constraining matchmaking, TrueSkill2 changes the assumption about the skill rating of squads. This new assumption should be used during matchmaking as well as after a match, when updating skill ratings. There are a variety of different models that one could imagine, but, in keeping with metric-driven modelling, we chose the simplest model that fixes the metrics in the above table. This model simply adds an offset to the skill rating of a player, depending on the size of their squad, when generating the player's performance. Therefore equation (4) is replaced by:

$$\text{perf}_i^t \sim \mathcal{N}(\text{skill}_i^t + \text{squadOffset}(\text{size of squad}), \beta^2) \tag{7}$$

where squadOffset is an array of tunable parameters, one for each possible squad size, for each game mode. We fix squadOffset(1) = 0.

To incorporate this change, we updated the model description for Infer.NET, generated the new inference code automatically, and ran batch inference on the training set to infer the model parameters. On the same test set as before, we get the following metric values:

Player Squad Size	Win Rate (%)	Expected by TrueSkill2
1	47	46
2	50	51
3	50	50
4	57	57
5	53	53
6	55	54
7	62	59
8	68	61
9	70	66
≥10	89	87

We cannot change the matchmaking in historical data, so the actual win rates are unchanged. But the model now predicts close to the correct win rates, implying that its skill ratings for squads are more accurate.

7 Experience effects

A key priority of any game studio is ensuring that new players have an enjoyable first time with the game. Part of this is putting new players into fair matches. This can only happen if the skill ratings assigned to new players are accurate. We measure this by partitioning players according to the number of matches that they have previously played in any game mode (we call this their *experience*), and comparing their actual win rate to that expected under the TrueSkill model.

Player Experience	Win Rate (%)	Expected by TrueSkill
0	45	49
1	46	49
2	45	48
3	45	48
4	45	48
5	46	48
6	46	48
7	46	48
8	46	48
9	47	48
≥10	49	48

We see that TrueSkill is not predicting the correct win rate for new players. These results suggest that players tend to increase in skill as they play the game, with the largest increases happening at the beginning. It makes sense intuitively that skill is correlated with experience. However, the classic TrueSkill model assumes that player skill changes according to a random walk, where upward and downward changes are equally likely. This suggests a simple change to the model, to make the random walk biased. Let experience_i^t be the number of matches played in the current game mode before time t. In practice, we cap this at 200 to limit the number of parameters. Instead of equation (2), we use

$$\text{skill}_i^{t+L} \sim \mathcal{N}(\text{skill}_i^t + \text{experienceOffset}(\min(\text{experience}_i^t, 200)), \gamma^2) \quad (8)$$

where experienceOffset is an array of 200 tunable parameters per game mode.

Operationally, this means that each player receives a small increment to their skill rating after playing a match, regardless of the outcome. These increments are small relative to the difference between winning and losing a match, so skill rating will still go down if a player loses repeatedly. The table below shows the effect of adding these increments for players in the test set:

Player Experience	Win Rate (%)	Expected by TrueSkill2
0	45	43
1	46	44
2	45	44
3	45	44
4	45	45
5	46	45
6	46	46
7	46	46
8	46	46
9	47	47
≥10	49	49

TrueSkill2 does a much better job of tracking win rates. However, it is consistently underestimating the win rate. This is because many of the players considered "new" in our test set are not actually new players.

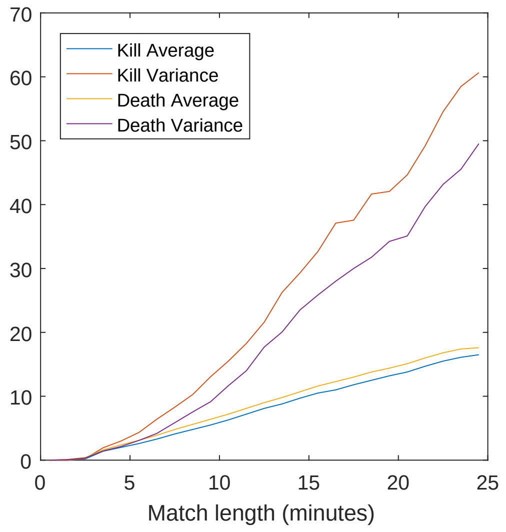
8 Individual statistics

In the Halo 5 data, each match result lists statistics for each player, such as the number of kills they scored and the number of times they were killed (deaths). Since these numbers are likely to be correlated with skill, it is worth trying to include this data in the model.

As with any model change, the first step is to define metrics that demonstrate the usefulness of this data for estimating skill ratings. One approach is to partition players according to the number of kills they score per minute of play (their *kill rate*), and look for discrepancies between their actual and predicted win rate. In doing so, it is important to exclude the number of kills in the match whose outcome is being predicted, because there is an obvious correlation between scoring many kills in a match and winning that match. The table below partitions players in the test set according to their kill rate in their previous match. There are significant differences between their actual and expected win rate under the classic TrueSkill model.

Player Previous Kill Rate	Win Rate (%)	Expected by TrueSkill
(0.0 - 0.4)	43	48
(0.4 - 0.8)	46	48
(0.8 - 1.2)	49	48
(1.2 - 1.6)	50	48
(1.6 - 2.0)	51	48
(2.0 - 2.4)	52	48
(2.4 - 2.8)	53	49
(2.8 - 3.2)	54	49
(3.2 - 3.6)	56	50
(3.6 - 4.0)	59	50

The remaining task is to choose an appropriate probabilistic generative model for a player's kill and death counts. A common model for sports scores is a Poisson distribution whose mean depends on a team's scoring ability minus their opponent's defensive ability [Baio and Blangiardo, 2010, Guo et al., 2012, Ruiz and Perez-Cruz, 2015]. This assumes that scoring events occur with a constant rate and independently of the time since the last event. This in turn implies that scores should scale with the length of the match, yet none of the cited papers include this effect in their model. To test this theory, we partitioned matches by length, computed the average number of kills and deaths per player, and the variance around this average. To eliminate variations due to skill, we only considered matches where all players had a near-average skill rating. To eliminate variations due to game mode, we only considered matches in the *Warzone* mode. Here are the results:



The plot shows that both the mean and variance of the counts scale linearly with match length, as would be expected by the independence assumption. However, the variance does not equal the mean, as required by a Poisson distribution. Therefore we chose to use a Truncated Gaussian distribution for the counts.

Another key aspect of these papers is that they use latent variables whose sole purpose is to model the scoring ability of a team. In TrueSkill2, the goal is to correlate kill/death counts with the existing player skill variable. In game modes where the objective to score the most kills, then we expect this correlation to be high. In game modes where the objective is to capture territory or simply stay alive as long as possible, we expect this correlation to be low. Even in modes where the objective is to score kills, there may be teamwork effects where players can help their team win without scoring kills themselves. We ultimately want player skill to reflect a player's ability to win, not their ability to score kills.

Our solution to this problem is to predict individual statistics from the performance variables perf_i. As explained in section 2, the performance variables determine the winner of the match. Therefore the model cannot choose skill ratings that only predict individual statistics. They must predict the match winner first, and secondarily predict the individual statistics. We expect that the most benefit from including individual statistics will come in team games. In such games, the players on a team can have different performance values, as long as their total performance is consistent with the match result. TrueSkill2 uses this flexibility to give more credit for a win to players with the best individual scores, something that classic TrueSkill cannot do.

To determine how the individual statistics depend on performance, we partitioned players by skill, the average skill of their teammates, and the average skill of their opponents. We found that the statistics were linear in all three, with teammates having a very small effect. Therefore to keep the model simple, we dropped the dependence on teammates. Putting all of the above together gives the following model for a count of type c ($c \in \{\text{kill}, \text{death}\}$):

$$\text{count}_i^c \sim \max(0, \mathcal{N}((w_p^c \text{perf}_i + w_o^c \text{perf}_i^o)\text{timePlayed}_i, v^c \text{timePlayed}_i)) \tag{9}$$

where (w_p^c, w_o^c, v^c) are tunable parameters for each count type in each game mode. These parameters determine the correlation between individual statistics and skill rating. For two teams of equal size where everyone played for the full time, perf_i^o is the average performance of the opposing team. Typically, w_p and w_o have different signs. For kill count, $w_p > 0$ and $w_o < 0$. For death count, $w_p < 0$ and $w_o > 0$.

If the player's team is larger than the opposing team, then we need a different formula for perf_i^o, to account for the divided attention of each opposing player. For each player i, let opposing_i be the total number of opposing players, weighted by their play time (L is the length of the match):

$$\text{opposing}_i = \sum_{j \mid \text{team}(j) \neq \text{team}(i)} \frac{\text{timePlayed}_j}{L} \tag{10}$$

We assume that each player's attention is divided by this number. Therefore

$$\text{perf}_i^o = \sum_{j \mid \text{team}(j) \neq \text{team}(i)} \frac{\text{timePlayed}_j}{L} \frac{\text{perf}_j}{\text{opposing}_j} \tag{11}$$

This formula works for any number of teams of any sizes. Thus there are three key differences between TrueSkill2's model of individual statistics and previous work on sports scores:

1. TrueSkill2 incorporates the effect of match length.
2. TrueSkill2 models the correlation between individual statistics and a player's ability to win.
3. TrueSkill2 incorporates the effect of having many teams of different sizes.

The linear model here could be replaced with a more flexible model, if the data warranted it. However, it is important that the model is constrained to be monotonic. A monotonic relationship incentivizes players to maximize their kill count and minimize their death count. A non-monotonic relationship would give players misaligned incentives, such as stopping when they reach a certain number of kills.

The table below shows the effect of these model changes on the test set:

Player Previous Kill Rate	Win Rate (%)	Expected by TrueSkill2
(0.0 - 0.4)	43	41
(0.4 - 0.8)	46	45
(0.8 - 1.2)	49	48
(1.2 - 1.6)	50	50
(1.6 - 2.0)	51	51
(2.0 - 2.4)	52	52
(2.4 - 2.8)	53	53
(2.8 - 3.2)	54	54
(3.2 - 3.6)	56	56
(3.6 - 4.0)	59	59

There is a much better agreement between actual and expected win rate, therefore the skills estimated from individual statistics are more accurate.

9 Quit penalty

As discussed in section 2, classic TrueSkill does not penalize players who quit. TrueSkill2 solves this problem by including a player's completion status ("completed" versus "quit") in the generative model.

The first step, as always, is to define metrics that measure the relationship between completion status and skill. In section 8, we looked at a player's behavior in the previous match. The same approach works here. Specifically, we can partition players according to their completion status in the previous match and look for discrepancies between their actual and predicted win rate.

Player Previous Outcome	Win Rate (%)	Expected by TrueSkill
Win Completed	51	50
Win Quit	45	49
Loss Completed	47	46
Loss Quit	44	47
Draw Completed	49	48
Draw Quit	46	47

This table shows that quitters tend to have lower skill (for the same match outcome), but their TrueSkill rating does not reflect this.

Digging deeper into the data, we found that players who quit have worse individual statistics than players who complete, even for the same amount of time played. This suggests that players quit more when their performance is low. But low relative to what? According to section 8, individual statistics are determined by the player's performance minus the opposing team's performance. This suggests the following probabilistic model for whether a player is under-performing:

$$\text{under}_i \sim (\mathcal{N}(\text{perf}_i - \text{perf}_i^o - m_q, v_q) < 0) \tag{12}$$

where perf_o is defined the same way as in section 8, and (m_q, v_q) are tunable parameters (per game mode) that allow for variation in the degree that a player underperforms. We force $m_q \leq 0$ during parameter estimation. Of course, a player may not quit when they underperform, and players may quit for other reasons. We model these effects by random Boolean variables:

$$\text{quit}_i = \text{unrelated}_i \text{ OR } (\text{related}_i \text{ AND } \text{under}_i) \tag{13}$$

where unrelated_i is true with probability p_u and related_i is true with probability p_r, both tunable parameters in each game mode.

The net effect of this model is that it will infer lower performance values for players who quit compared to players who do not quit. A surprising side effect is that, in order to penalize players that quit, the model must reward players that complete. It is counter-intuitive for a player's skill rating to increase simply for completing a match, especially if they played for zero seconds. But in order to penalize players that quit with zero seconds played, the model must reward players that complete with zero seconds played. Note that this model also adds a small amount of additional computation to process each match, even if no player ever quits.

A property that we would like is that, if no player quits, then the computational cost and final results should be identical to classic TrueSkill. This means we cannot use a model that generates completion status. Instead, we treat a quit as an additional observation that only appears when a player quits. We interpret a quit as the outcome of a mini-game between the player and (a rescaled version of) the enemy team. Unfortunately, since this model is unnormalized, we cannot easily learn its parameters. Our solution is to learn parameters using the normalized model, then switch to the unnormalized model for online updating, with the same parameters. In experiments, this switch actually gives a slight improvement in predictive accuracy, compared to using the normalized model throughout.

The table below shows the effect of these model changes, on the same test data as above:

Player Previous Outcome	Win Rate (%)	Expected by TrueSkill2
Win Completed	51	52
Win Quit	45	43
Loss Completed	47	46
Loss Quit	44	42
Draw Completed	49	48
Draw Quit	46	43

We see that TrueSkill2 lowers a player's skill rating after a quit, but a bit more than it should.

10 Features that did not need to be added

The model changes so far have all had orthogonal metrics. That is, the metrics that motivated each change could not be fixed by the other changes. For example, the metrics that motivated squad offsets are not fixed by modeling experience or individual statistics. But in other cases, metrics that seem to indicate a model change are automatically fixed by earlier changes.

For example, when we partition players according to the length of time since their last match ("lapse"), we get the following table:

Lapse in days	Win Rate (%)	Expected by TrueSkill
(0 - 1)	49	48
(1 - 2)	46	47
(2 - 3)	46	48
(3 - 4)	45	48
(4 - 5)	45	48
(5 - 6)	45	48
(6 - 7)	44	48
(7 - 8)	43	48
(8 - 9)	43	48
≥ 9	45	48

Win rate decreases with lapse, and classic TrueSkill does not capture this. This seems to indicate that player lapse should be added to the model. However, once the experience effect was added in TrueSkill2, the correct trend with lapse emerged automatically:

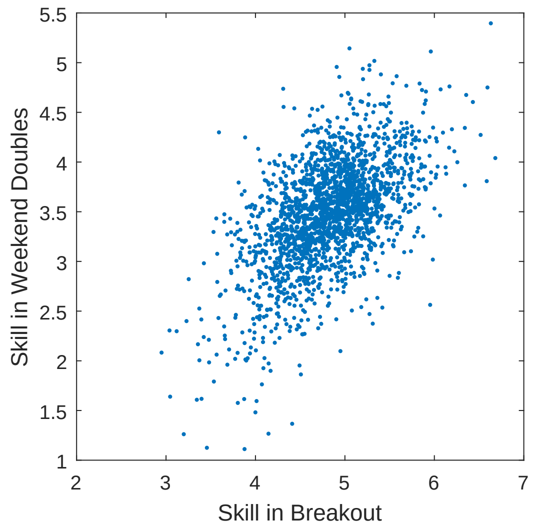
Lapse in days	Win Rate (%)	Expected by TrueSkill2
(0 - 1)	49	48
(1 - 2)	46	46
(2 - 3)	46	45
(3 - 4)	45	44
(4 - 5)	45	44
(5 - 6)	45	43
(6 - 7)	44	43
(7 - 8)	43	42
(8 - 9)	43	42
≥ 9	45	42

A related occurrence was features that seemed like they should be useful to add to the model, but turned out to be unnecessary. These were caught early by looking at the right metrics. For example, player score seems like it should be relevant since it incorporates mode-specific objectives. The raw score reported by the game includes kills, so we subtracted these off to get a number we call "Extra score". As in section 8, we partitioned players by their "Extra score" in the previous match, and compared their actual and expected win rate. With kills and deaths already included in the model, these numbers matched. So player score did not need to be added.

11 Mode correlation

Most online multiplayer games have multiple game modes, which change the rules and/or objectives of the game. A player typically queues up to play a particular mode, and expects the matchmaker to construct a fair match for that mode. Thus the matchmaker needs to know a player's skill rating in a particular mode. In some games, such as Halo 5, a mode is chosen after matchmaking. Even in these cases, it can be useful to have a skill rating per mode since it leads to faster convergence and more accurate skill ratings overall. For example, suppose player A has high skill in mode 1 and low skill in mode 2. When playing against a mid level opponent, we expect A to win in mode 1 and lose in mode 2. So if A wins in mode 1, then the opponent's skill rating should decrease slightly, if at all. But if A wins in mode 2, then the opponent's skill rating should decrease significantly. To construct a single skill rating for matchmaking, take a weighted average of the player's rating in each mode, where the weight is the probability of playing that mode.

The traditional approach to handling multiple gameplay modes in TrueSkill is to have a separate skill distribution for each mode. This is equivalent to assuming that a person's skill in one mode is independent of their skill in another mode. We can easily test this assumption by comparing the skill ratings computed by TrueSkill for different gameplay modes. For two of the modes in the Halo 5 data, we plot the skill ratings of players with many games played in both modes:



The skill ratings are correlated, with correlation coefficient = 0.6. Repeating this experiment for all pairs of gameplay modes gives similar results. A player's skill in any gameplay mode is positively correlated with their skill in all other modes. This makes sense since many of the game mechanics, such as the controls, are the same between modes.

This suggests that we can improve skill estimation by borrowing information from other gameplay modes. Since the prior distribution (1) on a player's skill is Gaussian, a natural way to correlate skills is to use a multivariate Gaussian distribution with a full covariance matrix between modes. However, estimating the correlation between all pairs of modes requires a significant amount of training data. Typically, a game has a fewer popular modes, and a long tail of short-lived or niche modes. Therefore, in order to keep training data requirements low, we opted for a simpler approach where skills are correlated along a single dimension, which we call the player's *base skill*. The base skill evolves over time, and the base skill for player i at time t is denoted base_i^t. The skill of player i in mode d at time t is denoted skill_{id}^t and is equal to the base skill plus an offset.

The generative process for skills is changed to the following. The base skill is drawn from a Gaussian prior distribution:

$$\text{base}_i^{t_0} \sim \mathcal{N}(0, v_b) \tag{14}$$

where t_0 is the time of the first game, and v_b is a tunable parameter. The base skill changes with each match played, as well as between matches:

$$\text{base}_i^{t+L} \sim \mathcal{N}(\text{base}_i^t, \gamma_{\text{base}}^2) \qquad \text{after a match of length } L \tag{15}$$

$$\text{base}_i^{t'} \sim \mathcal{N}(\text{base}_i^t, \tau_{\text{base}}^2(t'-t)) \qquad \text{between matches} \tag{16}$$

The offset for a mode is drawn from a Gaussian prior distribution:

$$\text{offset}_{id}^{t_0} \sim \mathcal{N}(m_d, v_d) \tag{17}$$

where (m_d, v_d) are tunable parameters. The offset changes with each match played, as well as between matches:

$$\text{offset}_{id}^{t+L} \sim \mathcal{N}(\text{offset}_{id}^t + \text{experienceOffset}(\min(\text{experience}_{id}^t, 200)), \gamma_d^2) \tag{18}$$

$$\text{after a match of length } L$$

$$\text{offset}_{id}^{t'} \sim \mathcal{N}(\text{offset}_{id}^t, \tau_d^2(t'-t)) \tag{19}$$

$$\text{between matches}$$

Finally, skill is the weighted sum of base and offset:

$$\text{skill}_{id}^t = w_d \text{base}_i^t + \text{offset}_{id}^t \tag{20}$$

where w_d is a tunable parameter that governs the correlation between modes. If w_d is zero, mode d has no correlation with any other mode. To avoid bad local optima, we force $w_d \geq 0$ during parameter estimation.

The online updating algorithm from section 3 changes in the following way. Instead of maintaining a skill distribution per game mode, we maintain one base skill distribution plus an offset distribution per game mode. When processing a match result, we first increase the variance of the base distribution and offset distribution for the game mode. Since these distributions are all Gaussian, the distribution over skill defined by (20) is also Gaussian. This skill distribution is updated by the per-match model, which can be interpreted as a message from the match result to the skill variable. This message propagates to the base and offset variables in the usual manner of Expectation Propagation, to determine their posterior distributions. In practice, all we have to do is add (20) to the model description for Infer.NET and re-generate the skill updating code, which will output the posterior distribution for base and offset directly. During matchmaking, when we want a player's skill rating in a particular mode, we look up the base skill distribution and offset skill distribution (if it exists), then apply (20).

The net effect is that, when a player's skill is updated in one mode, it will also be updated in other modes. However, the mathematics of the update ensures that these changes will be negligible once the player has played a sufficient number of matches in the other mode. A player's skill rating in a mode where many matches have already been played will not be affected by playing other modes.

In some games, such as *League of Legends* and *Dota 2*, players can play as different characters with different abilities. Using the same mechanism as above, a player can be given a different skill rating with each character, and these can be correlated. Even if players choose their character after matchmaking, maintaining separate skill ratings increases the convergence rate and overall accuracy of the skill rating system, for the same reason as given earlier for game modes. If you beat someone who is playing a character for the first time, you should not get the same rating change as beating their best character. To construct a single skill rating for matchmaking, a simple approach is to take a weighted average of the player's skill rating with each character, where the weight is their probability of choosing that character.

12 Gears of War 4

TrueSkill2 has been used successfully as the skill rating system in *Gears of War 4* since its launch in 2016. Skills are updated in real-time using the online updates. Before each new "season" of ranked play, all skills and model parameters are recomputed using batch inference (TrueSkill Through Time) on the data from all previous seasons.

Unlike *Halo 5*, *Gears of War 4* has AI-controlled bots in social and co-op game modes. Each bot has a difficulty level. We include bots into the skill rating system by treating each difficulty level as a unique player. However, while human skills are assumed to change over time, a bot's skill is assumed constant over time. This means that the updates (8) and (3) are not applied to bot skills. (A bot's skill rating may change as we learn about the bot, but eventually this will converge.) Thus bots provide a fixed baseline against which all human skills can be measured. The inferred skill ratings for the bots can also provide feedback to the game designers about the true difficulty level of each bot.

13 Summary

This paper has presented TrueSkill2, a collection of model changes to TrueSkill as well as a new system for estimating model parameters. TrueSkill2 gives significantly more accurate skill ratings than TrueSkill, measured along a variety of axes important to a game studio.

Acknowledgements

Thanks to Alan Lawrence for help implementing the evaluation framework, and Vijay Sharma for help implementing the experimentation infrastructure. Thanks to The Coalition for providing development support and testing, and Clayton Vaught for his help with the Gears of War matchmaking designs. Thanks to 343 Industries for providing the data, and Joshua Menke and Brian Richards for helpful discussions.

References

Gianluca Baio and Marta A. Blangiardo. Bayesian hierarchical model for the prediction of football results. *Journal of Applied Statistics*, 37(2):253–264, 2010. URL http://www.statistica.it/gianluca/Research/BaioBlangiardo.pdf.

Shuo Chen and Thorsten Joachims. Modeling intransitivity in matchup and comparison data. In *WSDM*, 2016. URL http://www.cs.cornell.edu/~shuochen/pubs/wsdm16_chen.pdf.

Zhengxing Chen, Yizhou Sun, Magy Seif El-nasr, and Truong-Huy D. Nguyen. Player skill decomposition in multiplayer online battle arenas. In *Meaningful Play*, 2016. URL https://arxiv.org/abs/1702.06253.

Pierre Dangauthier, Ralf Herbrich, Tom Minka, and Thore Graepel. TrueSkill Through Time: Revisiting the History of Chess. In *Advances in Neural Information Processing Systems*, pages 931–938. MIT Press, January 2008. URL https://www.microsoft.com/en-us/research/publication/trueskill-through-time-revisiting-the-history-of-chess/.

Mark E. Glickman. Parameter estimation in large dynamic paired comparison experiments. *Applied Statistics*, 48:377–394, 1999. URL http://www.glicko.net/research/glicko.pdf.

Mark E. Glickman. Dynamic paired comparison models with stochastic variances. *Journal of Applied Statistics*, 28:673–689, 2001. URL http://www.glicko.net/research/dpcmsv.pdf.

Shengbo Guo, Scott Sanner, Thore Graepel, and Wray Buntine. Score-based bayesian skill learning. In *ECML*, 2012. URL http://users.rsise.anu.edu.au/~ssanner/Papers/sbsl_ecml2012.pdf.

Ralf Herbrich, Tom Minka, and Thore Graepel. TrueSkill(TM): A Bayesian Skill Rating System. In *Advances in Neural Information Processing Systems*, pages 569–576. MIT Press, January 2007. URL https://www.microsoft.com/en-us/research/publication/trueskilltm-a-bayesian-skill-rating-system/.

Joshua E. Menke, Shane Reese, and Tony R. Martinez. Hierarchical models for estimating individual ratings from group competitions. 2006. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.533.1843.

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Chess Analysis example. Infer.NET documentation, 2014a. URL http://research.microsoft.com/infernet/docs/Chess%20Analysis.aspx.

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6, 2014b. URL http://research.microsoft.com/infernet. Microsoft Research Cambridge.

Thomas P. Minka. Expectation propagation for approximate bayesian inference. In *Uncertainty in AI*, pages 362–369, 2001. URL https://tminka.github.io/papers/ep/.

Martin Riedmiller and Heinrich Braun. A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. In *IEEE International Conference on neural networks*, 1993.

Francisco J. R. Ruiz and Fernando Perez-Cruz. A generative model for predicting outcomes in college basketball. *Journal of Quantitative Analysis in Sports*, 11(1), 2015.