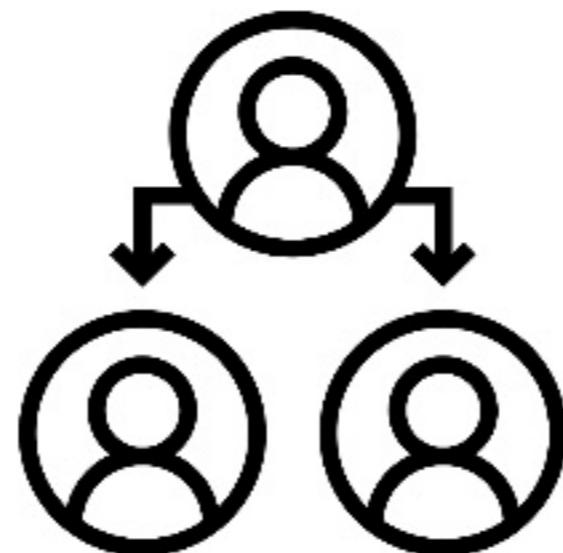


Balancing Heroes and Pokemon

Streaming/Bayesian Systems for Online Ranking



Fokko Driesprong & Vincent D. Warmerdam - GoDataDriven

we are fokko and vincent

- We **do not** work for Blizzard
- We **do not** work for a Nintendo
- We **do not** work for Microsoft [we're not demoing trueskill]
- Fokko does Engineering
- Vincent does Algorithms
- This project is a mere hobby project. Not production.

usually ML is applied as a lambda

BATCH

$$e_1 \xrightarrow{f_1} f(e_1)$$

$$e_2 \xrightarrow{f_1} f(e_2)$$

$$e_3 \xrightarrow{f_1} f(e_3)$$

$$e_4 \xrightarrow{f_1} f(e_4)$$

*f refreshes every
time we retrain f*

but it could be trained while being applied

BATCH

$$e_1 \xrightarrow{f_1} f(e_1)$$

$$e_2 \xrightarrow{f_1} f(e_2)$$

$$e_3 \xrightarrow{f_1} f(e_3)$$

$$e_4 \xrightarrow{f_1} f(e_4)$$

STREAMING

$$e_1 \xrightarrow{f_1} f_1(e_1)$$

$$e_2 \xrightarrow{f_2} f_2(e_2)$$

$$e_3 \xrightarrow{f_3} f_3(e_3)$$

$$e_4 \xrightarrow{f_4} f_4(e_4)$$

f refreshes every
time we retrain f

f_t refreshes every
time we apply it

streaming may be preferable

BATCH

$$e_1 \xrightarrow{f_1} f(e_1)$$

$$e_2 \xrightarrow{f_1} f(e_2)$$

$$e_3 \xrightarrow{f_1} f(e_3)$$

$$e_4 \xrightarrow{f_1} f(e_4)$$

STREAMING

$$e_1 \xrightarrow{f_1} f_1(e_1)$$

$$e_2 \xrightarrow{f_2} f_2(e_2)$$

$$e_3 \xrightarrow{f_3} f_3(e_3)$$

$$e_4 \xrightarrow{f_4} f_4(e_4)$$

f refreshes every
time we retrain f

f_t refreshes every
time we apply it

we will give an example in this talk

BATCH

e₁ → $f(e_1)$
e₂ → $f(e_2)$
e₃ → $f(e_3)$
e₄ → $f(e_4)$

**SOLVING STREAMING MEANS
STREAMING ALSO SOLVING
BATCH!**

f refreshes every
time we retrain f

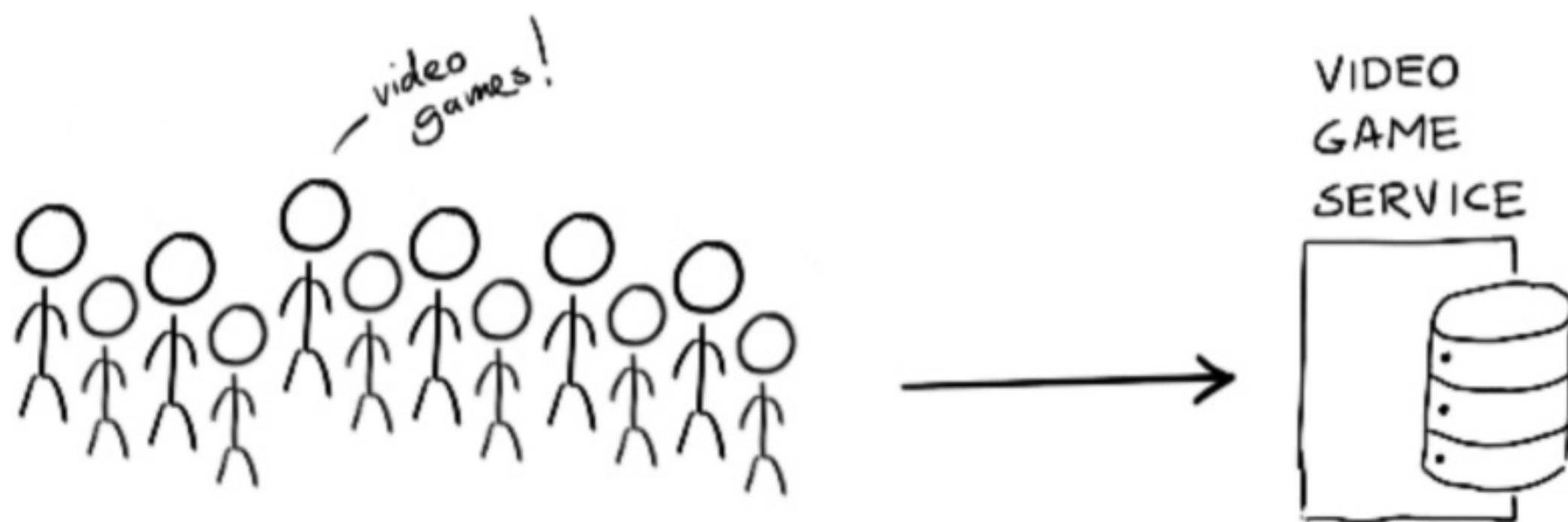
STREAMING

$e_1 \rightarrow f_1 \rightarrow f(e_1)$
 $e_2 \rightarrow f_2 \rightarrow f(e_2)$
 $e_3 \rightarrow f_3 \rightarrow f(e_3)$
 $e_4 \rightarrow f_4 \rightarrow f(e_4)$

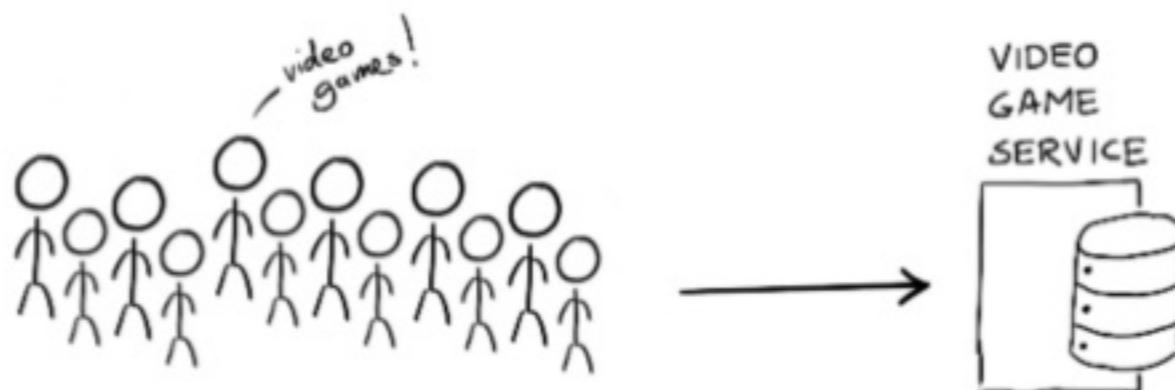
f_t refreshes every
time we apply it

problem at hand

This is our enterprise usecase.



problem at hand

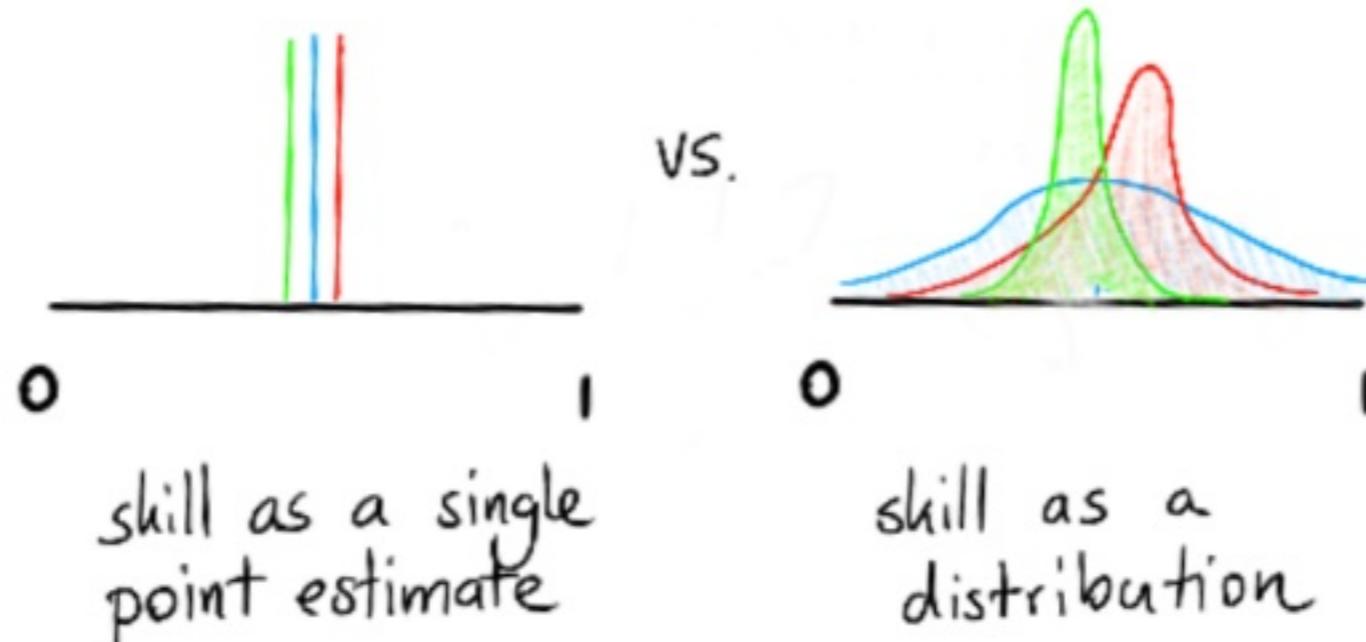


- we need to estimate player skill somehow
- we need to learn this from a stream of match outcomes
- we don't want to wait for a batch algorithm

let's talk solutions

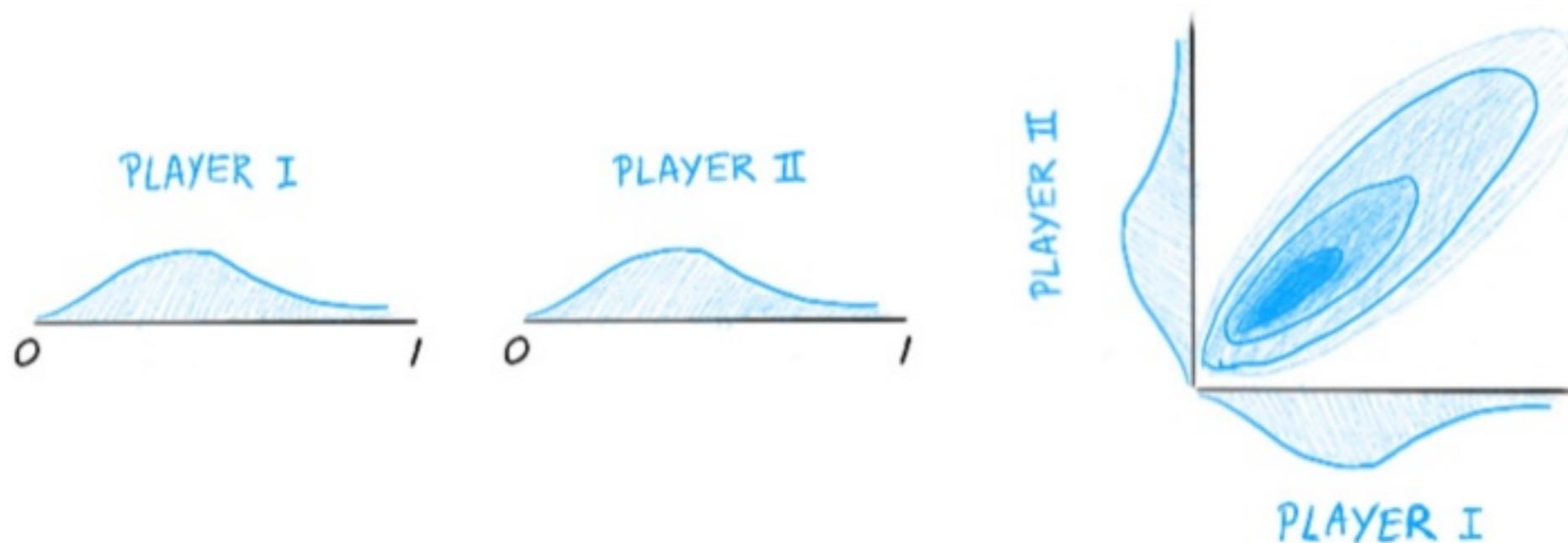
a simple solution

The way to make it work is to realize that the skill of a player is not a single number, rather a distribution of belief of the players skill.



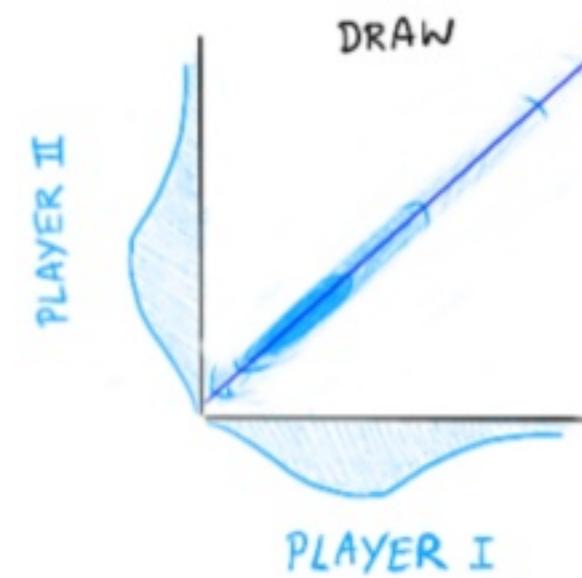
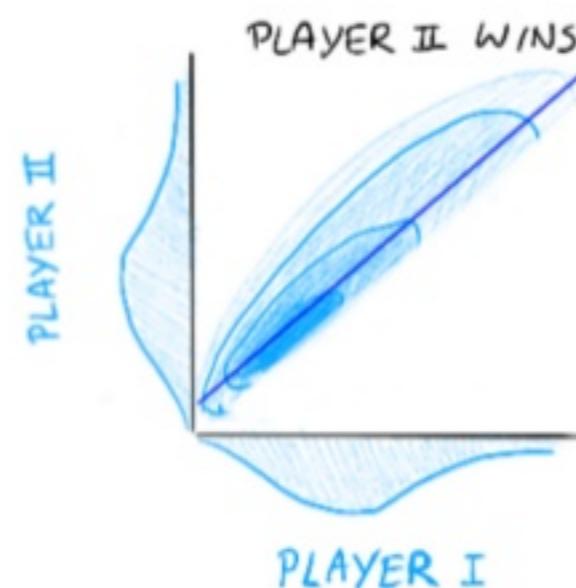
a simple solution

The next step is to realize that you can combine two one-player beliefs of skills into a one two-player belief. A prior of belief.



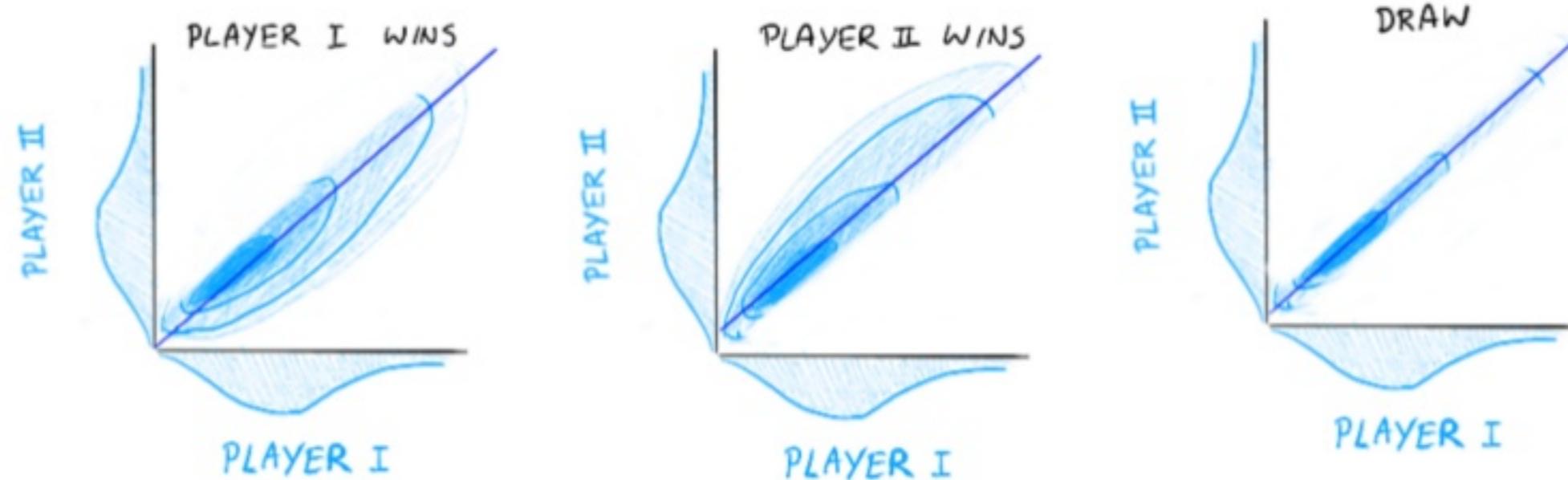
a simple solution

After a game has been played the two dimensional prior is updated depending on what the data shows us.



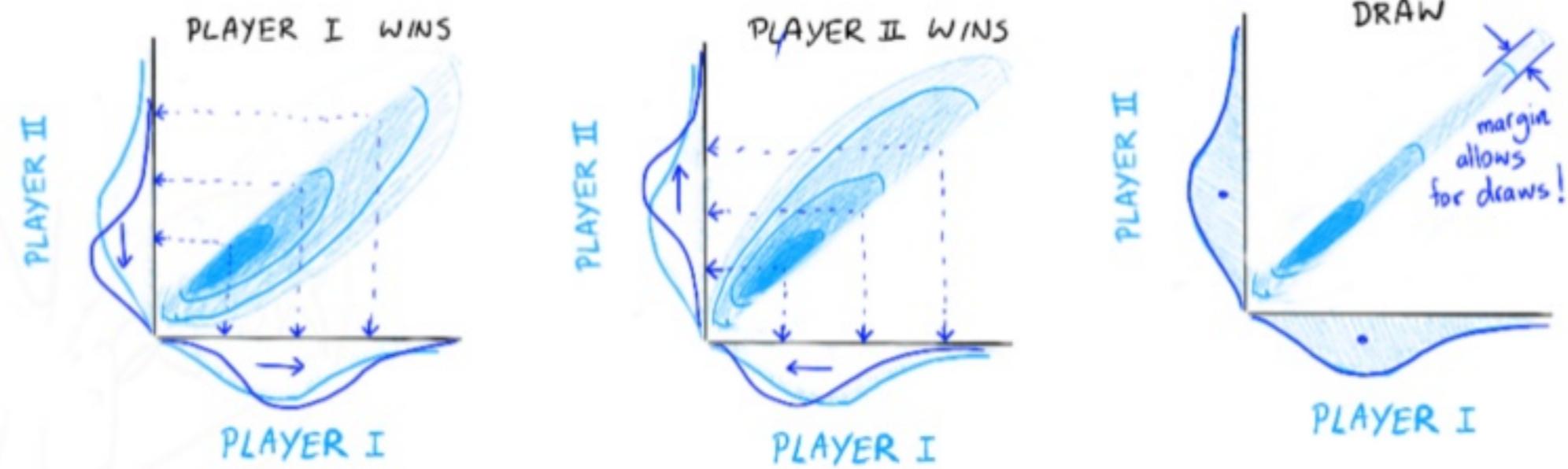
a simple solution

We take a **margin** over the diagonal and any probability mass from the region that disagrees with the match outcome.



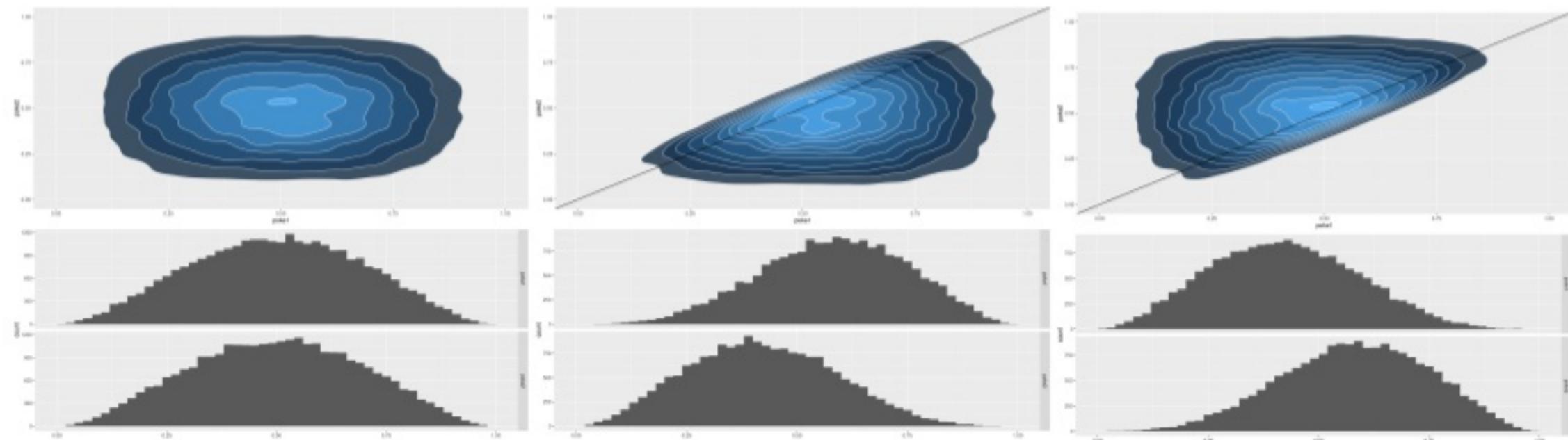
a simple solution

We map the resulting probability back to each player. These two players now have an updated belief on skill.



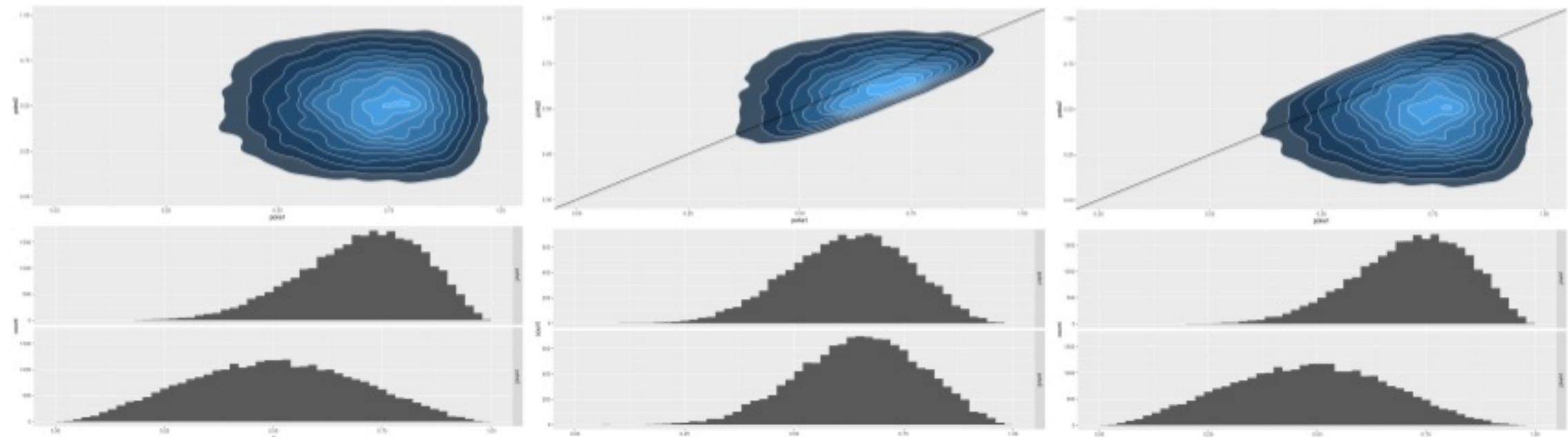
a simple simulation: two equal players

There are some benefits we get for free. Try it out [here](#).

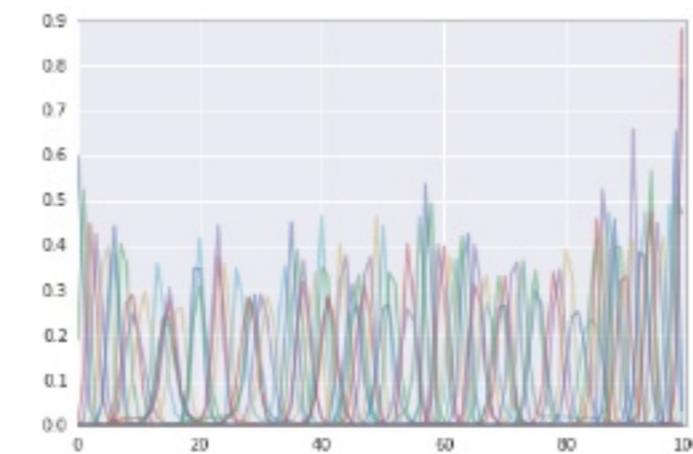
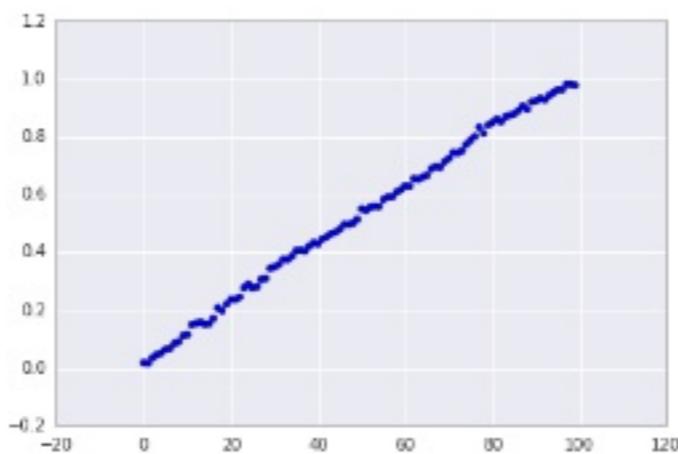
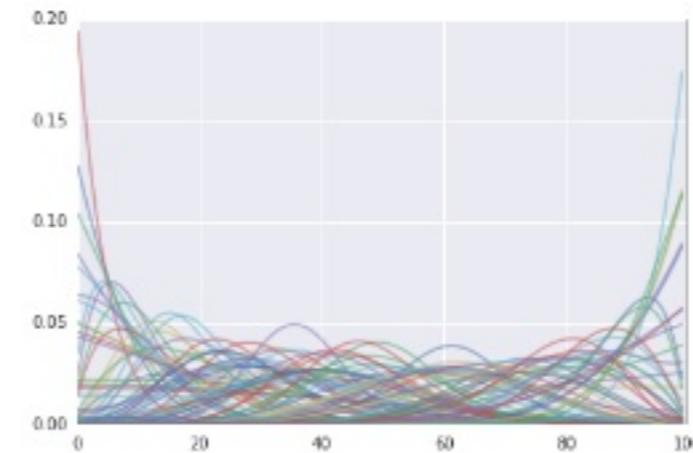
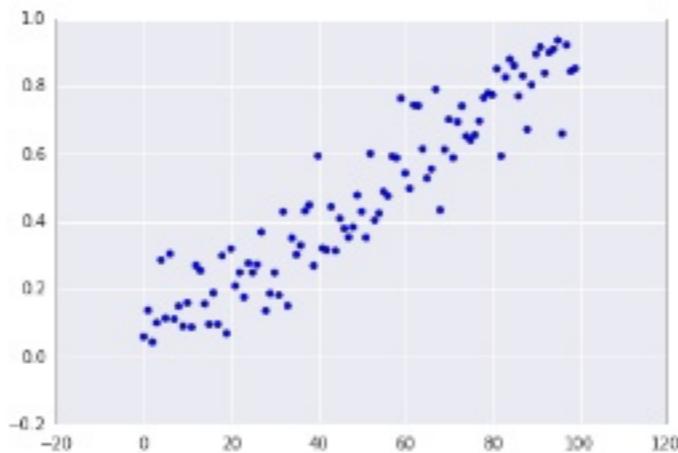


a simple simulation: two unequal players

We may learn a lot, or very little. #informationtheory



a simple simulation: many players



test of functionality: pokemon!



test of functionality: pokemon!

We got our information on pokemon from;

Pokéapi - The Pokémon RESTful API

Finally; all the Pokémon data you'll ever need, in one place,
and easily accessible through a modern RESTful API.

Over 37,503,000 API calls received!

Try it now!

Need a hint? try [pokemon/1](#) or [type/3](#) or [ability/4](#)

Nowadays you can also find the dataset on [kaggle](#).

test of functionality: pokemon!

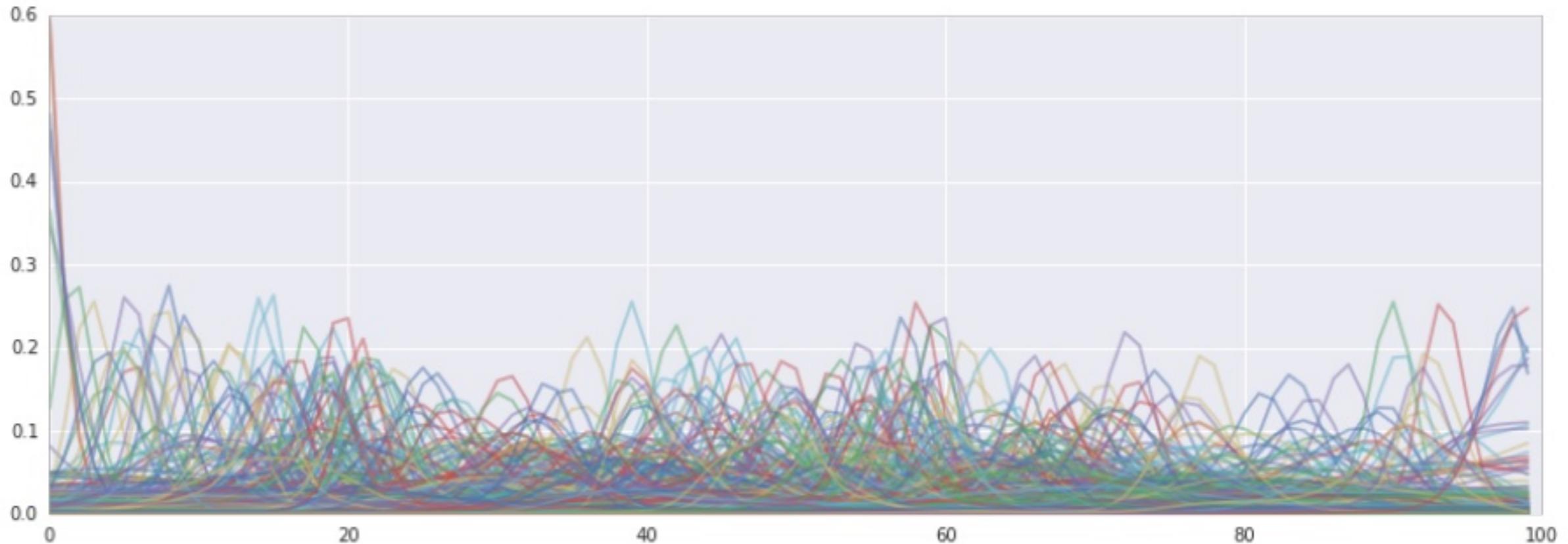
If you google around fan reditts you can find information on how many turns one pokemon can outlast the other.

$$T_{ij} = \frac{HP_i}{DMG_{ji}}$$

$$DMG_{ji} = \frac{2L_j + 10}{250} \times \frac{A_j}{D_i} \times w_{ij}$$

Simply said, we can use this to simulate game outcomes.

test of functionality: pokemon!



test of functionality: pokemon!

| | | name | mle | | | name | mle |
|-----|--|----------|----------|-----|--|------------|----------|
| 128 | | Magikarp | 0.011007 | 597 | | Ferrothorn | 0.916377 |
| 112 | | Chansey | 0.013735 | 425 | | Drifblim | 0.917240 |
| 348 | | Feebas | 0.015845 | 644 | | Zekrom | 0.919065 |
| 171 | | Pichu | 0.019108 | 537 | | Throh | 0.921175 |
| 291 | | Shedinja | 0.020473 | 482 | | Dialga | 0.928502 |
| 439 | | Happiny | 0.026763 | 149 | | Mewtwo | 0.951472 |
| 241 | | Blissey | 0.037009 | 483 | | Palkia | 0.953823 |
| 172 | | Cleffa | 0.042802 | 288 | | Slaking | 0.956629 |
| 234 | | Smeargle | 0.048575 | 375 | | Metagross | 0.957383 |
| 49 | | Diglett | 0.054010 | 492 | | Arceus | 0.957729 |

the general maths of all this

the general maths of all this

Designing the algorithm became a whole lot easier when we admitted that we want to quantify our uncertainty. This means we *must* work with distributions as our state, not mere statistics.

the general maths of all this

Designing the algorithm became a whole lot easier when we admitted that we want to quantify our uncertainty. This means we *must* work with distributions as our state, not mere statistics.

This really fits the bayesian mindset.

$$p(\theta|D) \propto p(D|\theta)p(\theta) = \prod_i p(d_i|\theta)p(\theta)$$

Bayesians to the rescue

We can estimate parameters by considering;

$$p(\theta|D) \propto p(D|\theta)p(\theta) = \prod_i p(d_i|\theta)p(\theta)$$

Let's consider three independent points of data that we've seen;
 d_1, d_2, d_3 . Then bayes rule states that

$$p(\theta|D) = p(\theta|d_1, d_2, d_3) \propto p(d_3|\theta)p(d_2|\theta)p(d_1|\theta)p(\theta)$$

Bayesians to the rescue

Everybody notice we kind of get streaming for free?

$$p(\theta|d_1, d_2, d_3) \propto p(d_3|\theta)p(d_2|\theta)\underbrace{p(d_1|\theta)}_{\text{prior for } d_2}p(\theta)$$

$$p(\theta|d_1, d_2, d_3) \propto p(d_3|\theta)\underbrace{p(d_2|\theta)p(d_1|\theta)}_{\text{prior for } d_3}p(\theta)$$

bayesians to the rescue

Any ML algorithm that can be updated via $p(\theta|D) \propto \prod_i p(d_i|\theta)p(\theta)$ is automatically a streaming algorithm for ML because it forces the recursive relation;

$$p(\theta|D_N, d_{N+1}) \propto p(d_{N+1}|\theta) \underbrace{p(D_N|\theta)p(\theta)}_{\text{prior for new datapoint}}$$

bayesians to the rescue

You can even have models that change over time appropriately.

$$p(\theta|D_N, d_{N+1}) \propto [p(d_{N+1}|\theta)]^\alpha \underbrace{[p(D_N|\theta)p(\theta)]^\beta}_{\text{prior for new datapoint}}$$

Know that this is not a free lunch though. You may still need to deal with numerics in the streaming situation, but a lot of algorithms can be streamified via this approach.

we just gave an example of this

BATCH

$$e_1 \xrightarrow{f_1} f(e_1)$$

$$e_2 \xrightarrow{f_1} f(e_2)$$

$$e_3 \xrightarrow{f_1} f(e_3)$$

$$e_4 \xrightarrow{f_1} f(e_4)$$

STREAMING

$$e_1 \xrightarrow{f_1} f_1(e_1)$$

$$e_2 \xrightarrow{f_2} f_2(e_2)$$

$$e_3 \xrightarrow{f_3} f_3(e_3)$$

$$e_4 \xrightarrow{f_4} f_4(e_4)$$

f refreshes every
time we retrain f

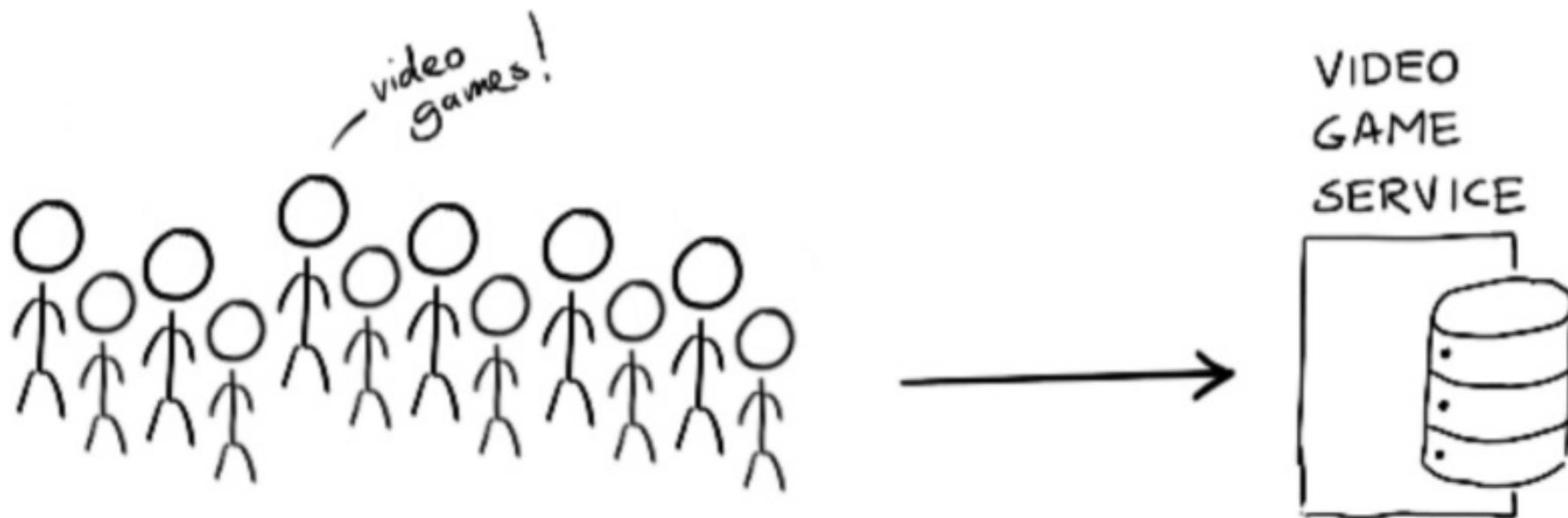
f_t refreshes every
time we apply it

a harder problem: heroes of the storm



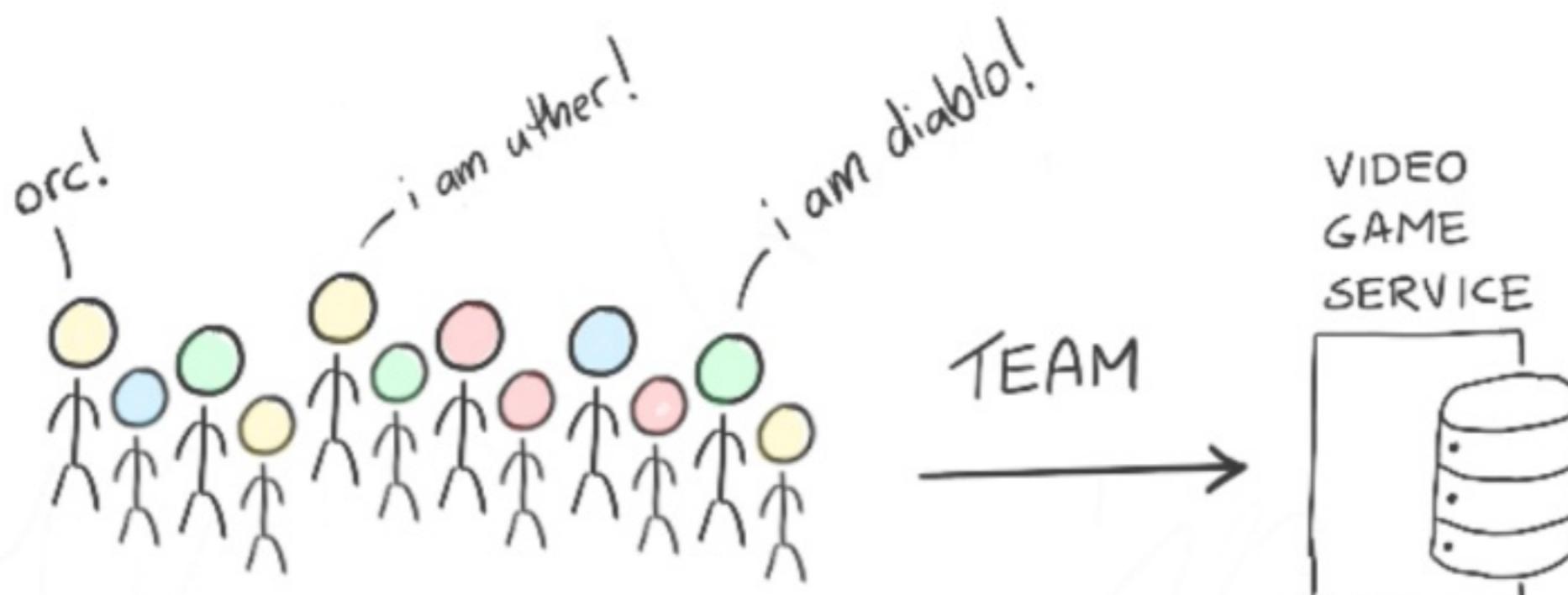
a harder problem: heroes of the storm

This was our enterprise usecase before.



a harder problem: heroes of the storm

This is our even more enterprise usecase now.

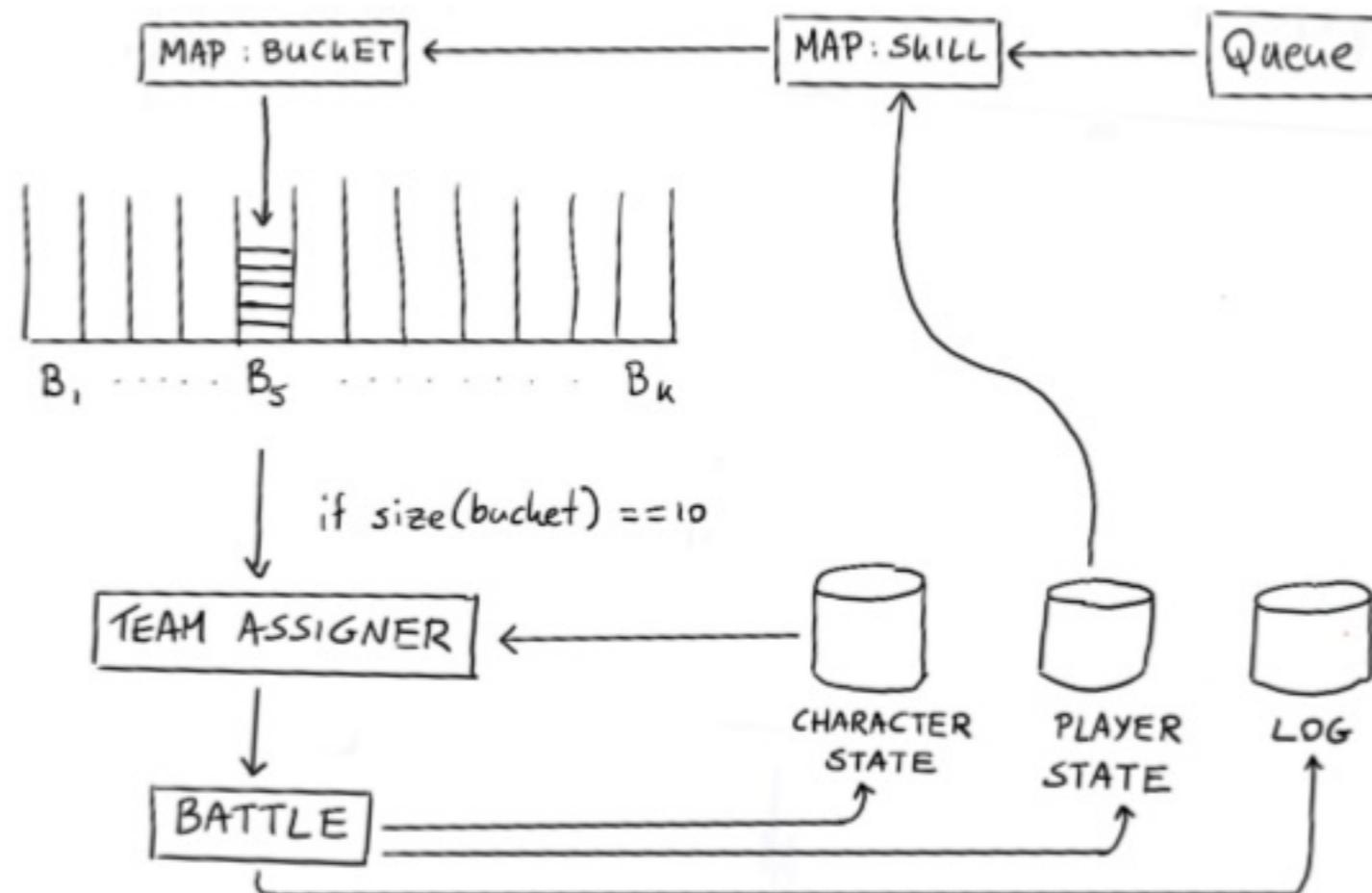


a harder problem: heroes of the storm



- we need to update player skills after a team match
- we now also need to assign teams as well
- we need to worry about character imbalance as well

proposal solution architecture

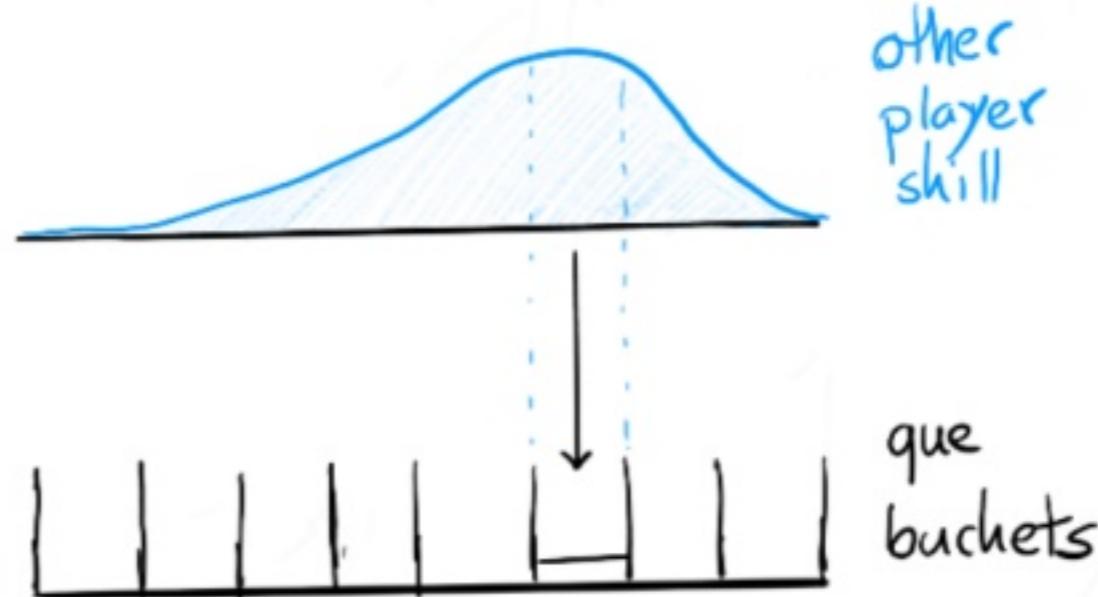
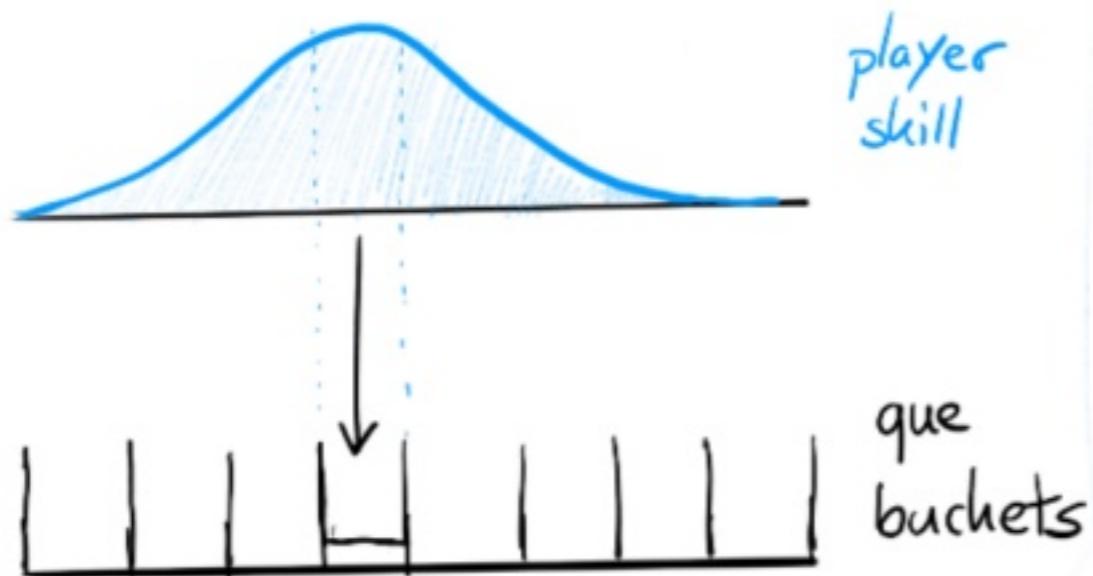


proposal solution architecture

There are three main steps to consider now;

- **how to map a skill to a que**
- **how to assign teams from a que**
- **how to update individual skill after a team fight**

teamfights: skill to que



teamfights: assigning teams

Predicting team imbalance with ML is tricky. There are lots of combinations of heroes ($n = 67$) that are possible.

$$C = \binom{n}{5} \times \binom{n}{5} \approx 9.3e13$$

This makes it hard to make some sort of machine learning model that can predict what team might win. Is there an easy hack?

teamfights: assigning teams

We have some domain knowledge; types of heroes.



P₁₀₂

DIABLO → TANK

P₁₀₃

RAYNOR → ASSASSIN

P₁₀₅

MURKY → SPECIALIST

P₁₀₆

PROBIUS → SPECIALIST

P₁₀₇

CENARIUS → HEALER

teamfights: assigning teams

| | | | |
|------------|-----|---|---|
| BRIGHTWING | → H | A | I |
| MALTHAEL | → A | A | 2 |
| PROBIUS | → S | A | I |
| GUL'DAN | → A | A | 2 |
| SAMURO | → A | T | I |
| RAYNOR | → A | H | 2 |
| DEHAKA | → T | H | I |
| SYLVANNAS | → S | S | 2 |
| LILI | → H | S | I |
| ZAGARA | → S | S | 2 |

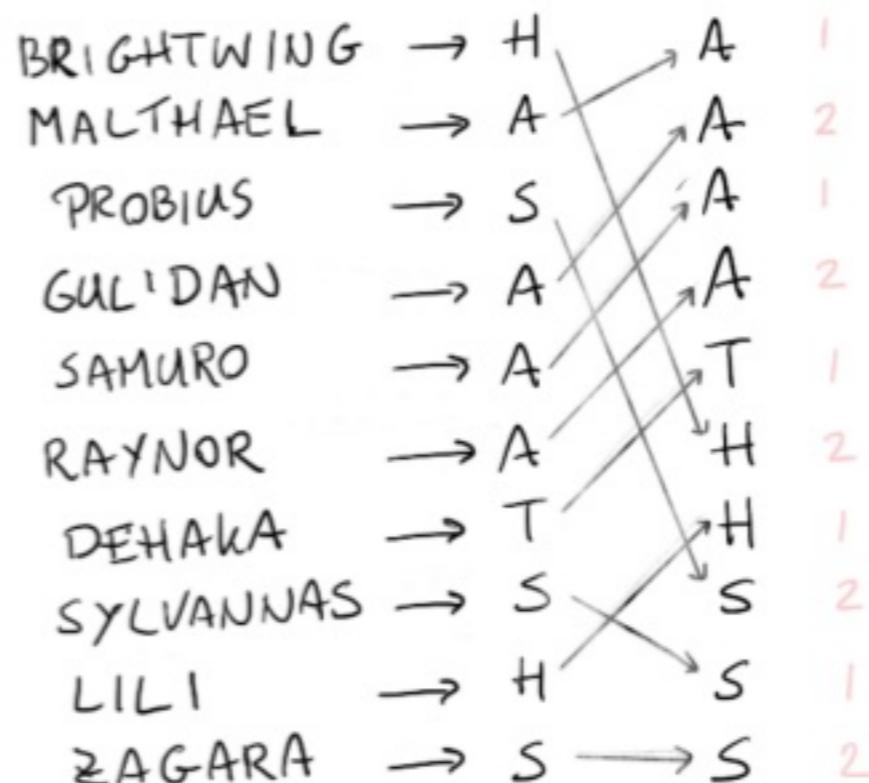
TEAM I

MALTHAEL
PROBIUS
DEHAKA
LILI
SYLVANNAS

TEAM 2

GUL'DAN
RAYNOR
BRIGHTWING
PROBIUS
ZAGARA

teamfights: assigning teams



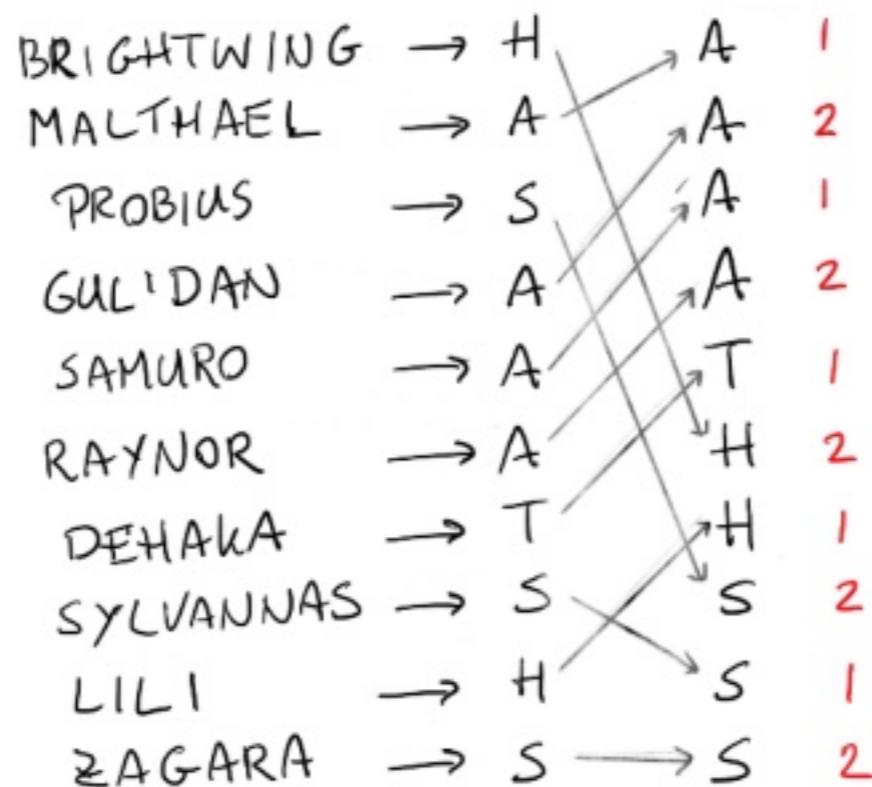
TEAM I

MALTHAEL
PROBIUS
DEHAKA
LILI
SYLVANNAS

TEAM 2

GUL'DAN
RAYNOR
BRIGHTWING
PROBIUS
ZAGARA

teamfights: assigning teams



TEAM I

MALTHAEL
PROBIUS
DEHAKA
LILI
SYLVANNAS

TEAM 2

GUL'DAN
RAYNOR
BRIGHTWING
PROBIUS
ZAGARA

teamfights: updating skill

- Each 5v5 game is a single ten-dimensional distribution of skill.
We remove prob mass where appropriate → **heavy** calculation

teamfights: updating skill

- Each 5v5 game is a single ten-dimensional distribution of skill. We remove prob mass where appropriate → **heavy** calculation
- Cheat a bit by summarising all opponent histogram into a single histogram and pretend that the player was battling this person

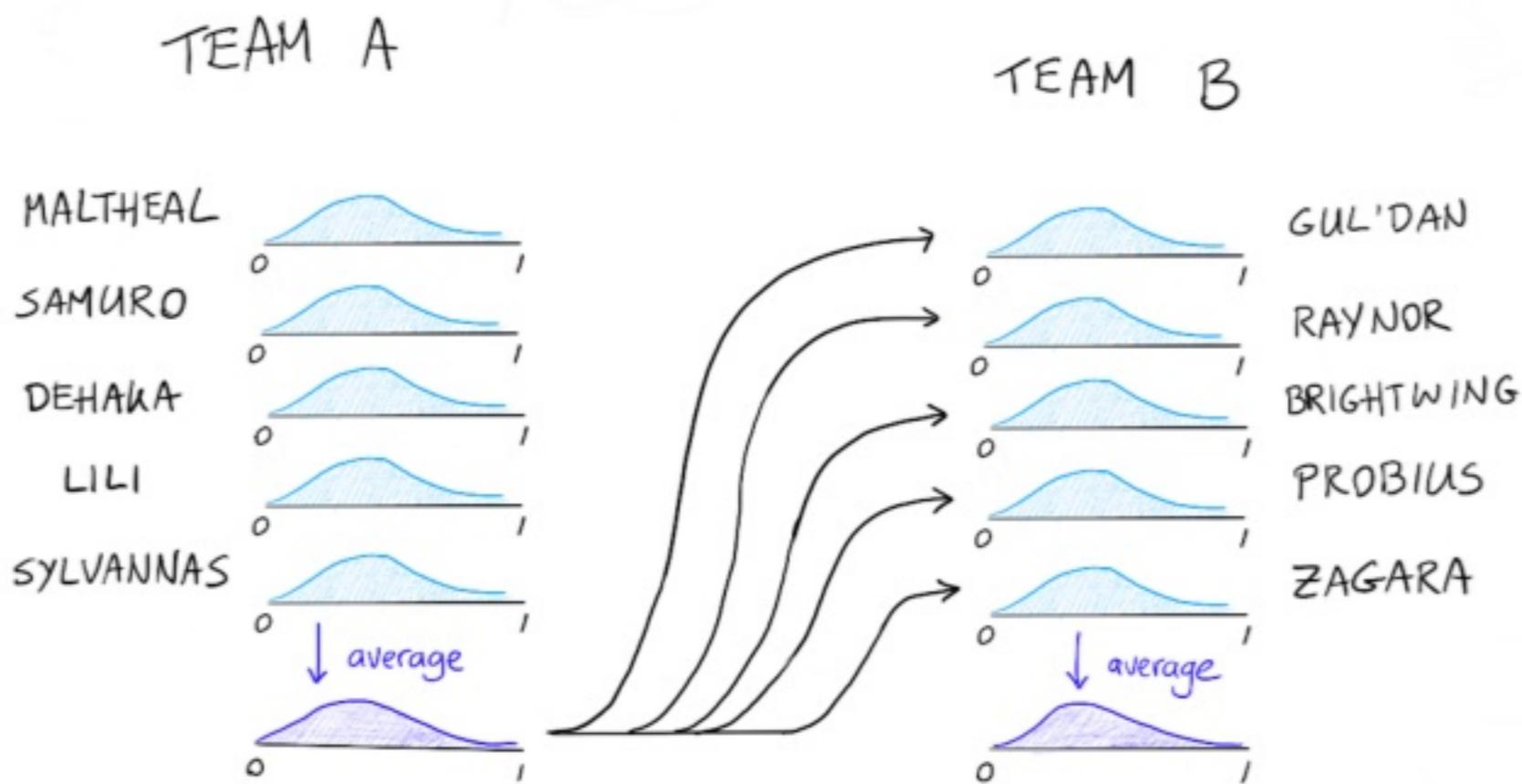
teamfights: updating skill

- Each 5v5 game is a single ten-dimensional distribution of skill. We remove prob mass where appropriate → **heavy** calculation
- Cheat a bit by summarising all opponent histogram into a single histogram and pretend that the player was battling this person
- Have the data scientist apply maths to make an update rule which involves proper distributions instead of histograms → mathematical complexity → maintenance **risk**

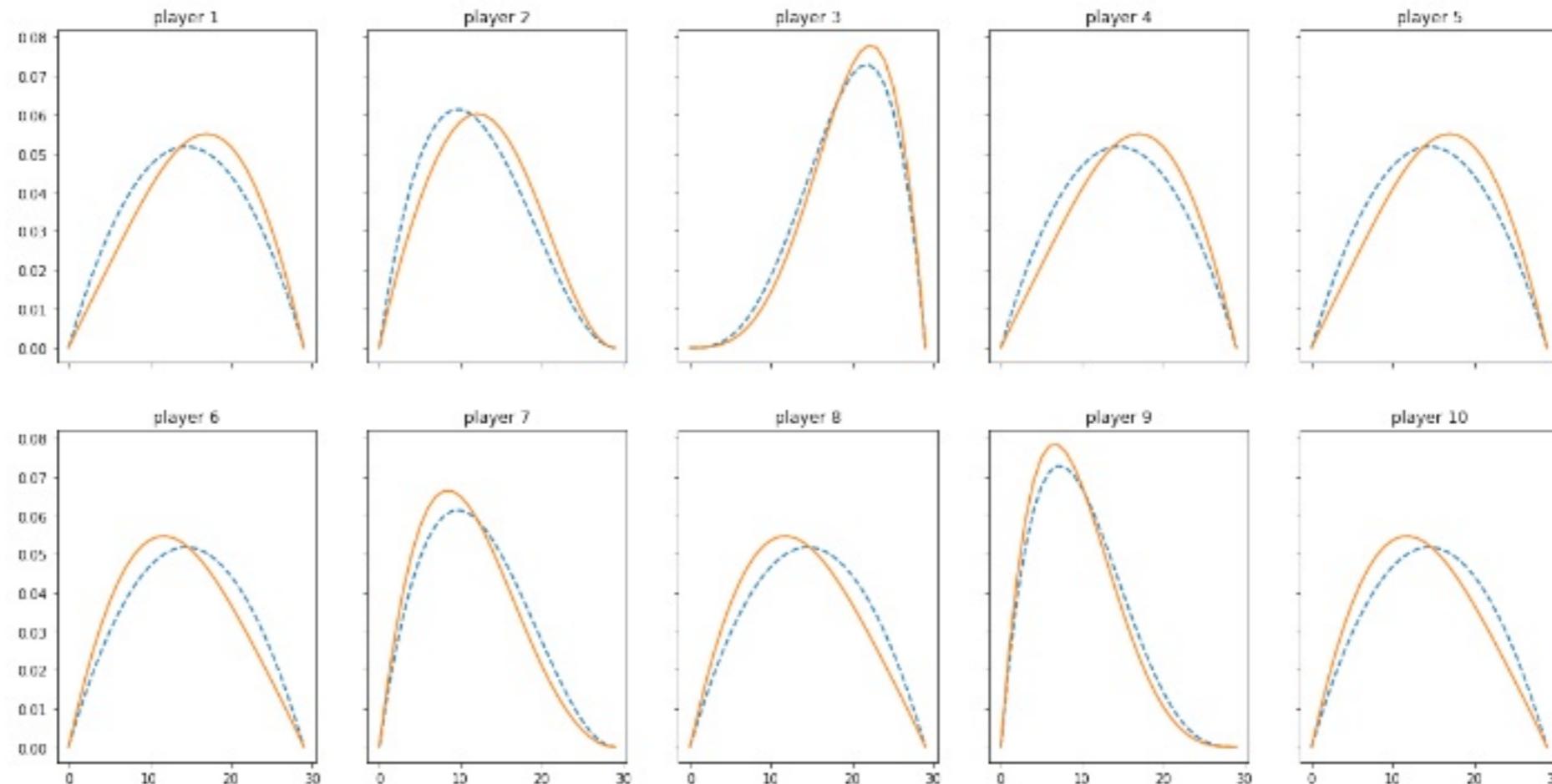
teamfights: updating skill

- ~~Each 5v5 game is a single ten-dimensional distribution of skill. We remove prob mass where appropriate → heavy calculation~~
- Cheat a bit by summarising all opponent histogram into a single histogram and pretend that the player was battling this person
- ~~Have the data scientist apply maths to make an update rule which involves proper distributions instead of histograms → mathematical complexity → maintenance risk~~

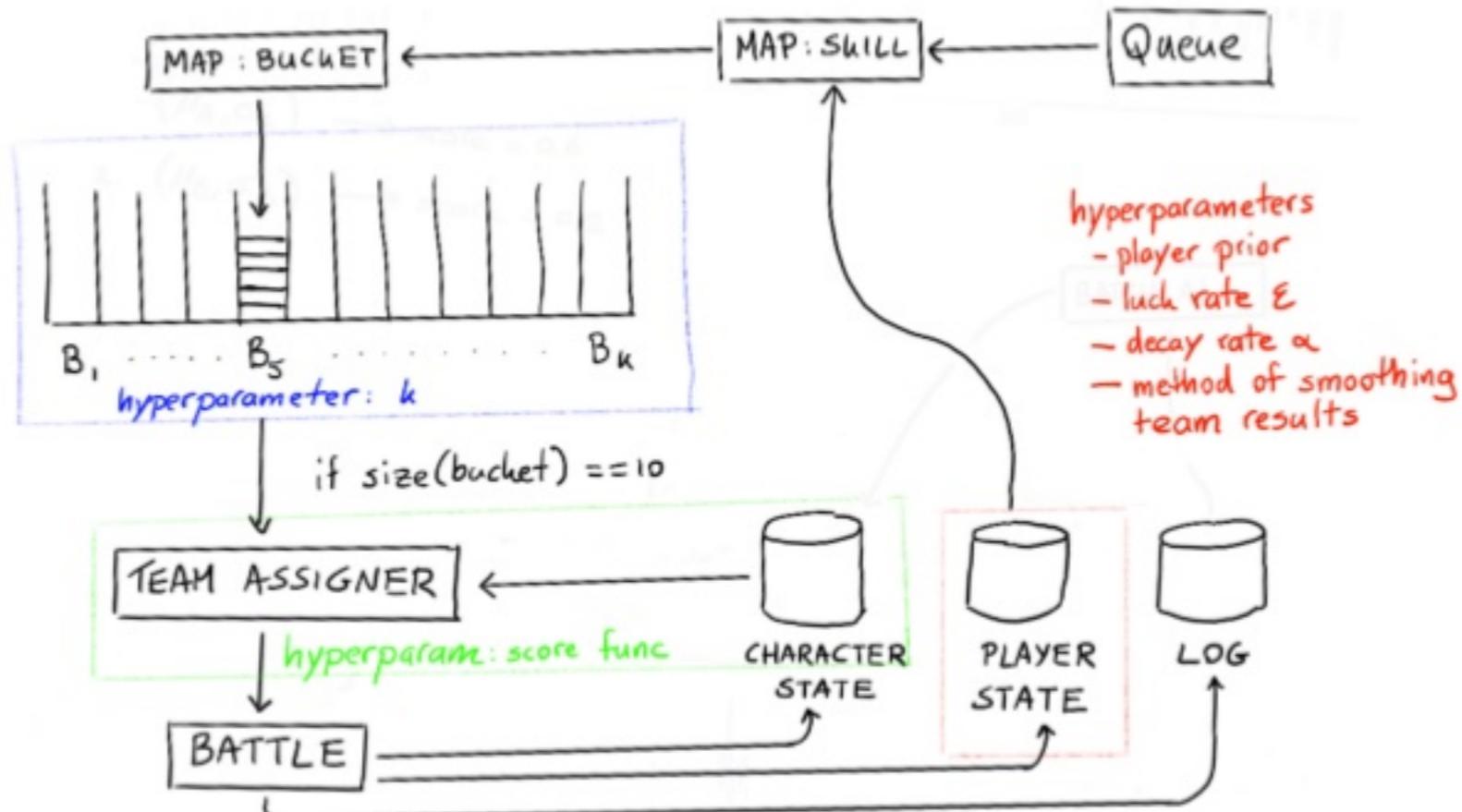
teamfights: updating skill



teamfights: updating skill



total architecture



things we like

- We have **separate** functions in our architecture. The team selection step needs no notion of how many queues there are.

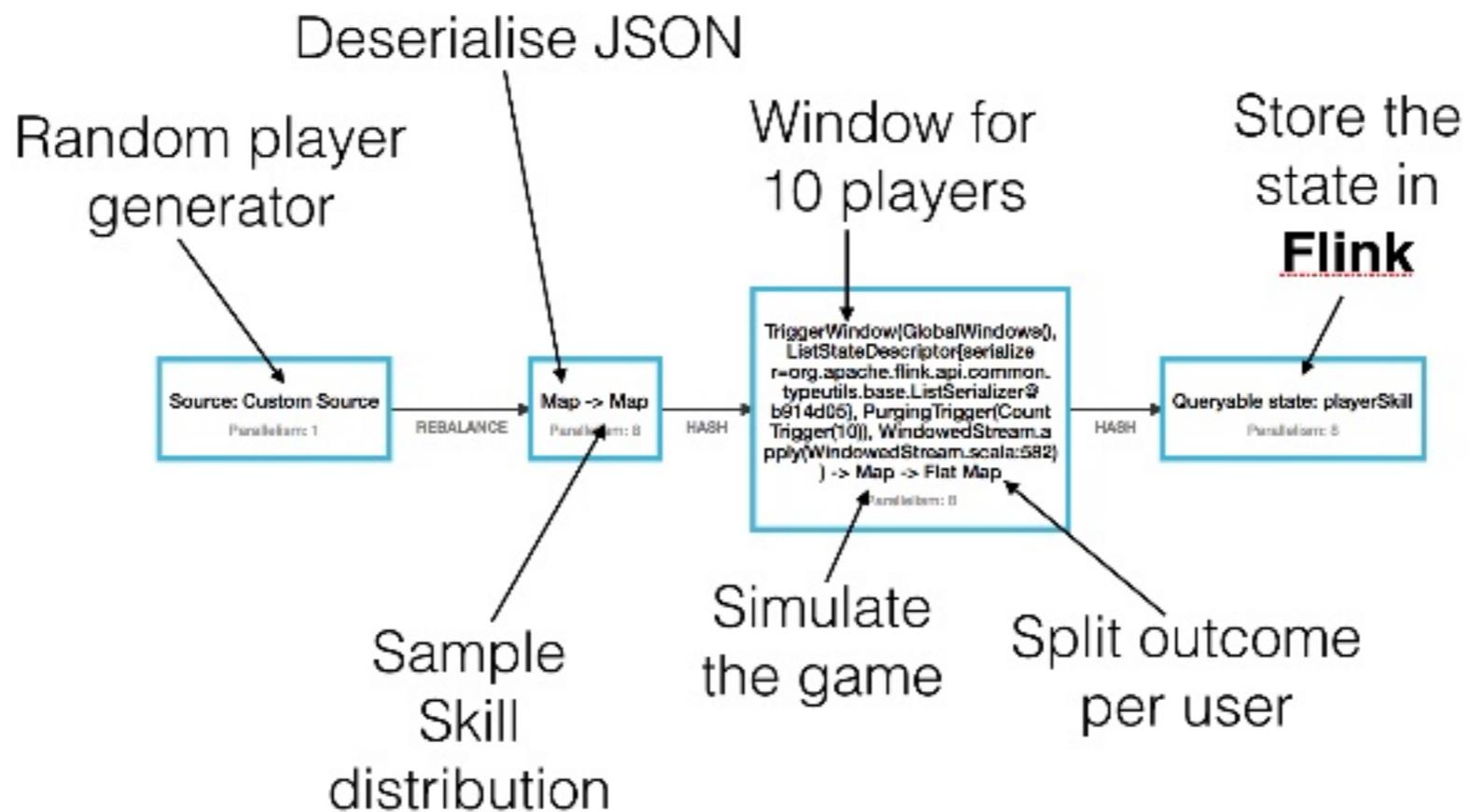
things we like

- We have **separate** functions in our architecture. The team selection step needs no notion of how many queues there are.
- We have interpretable hyperparameters. If we need to debug the system we can create meaningful visualisations.

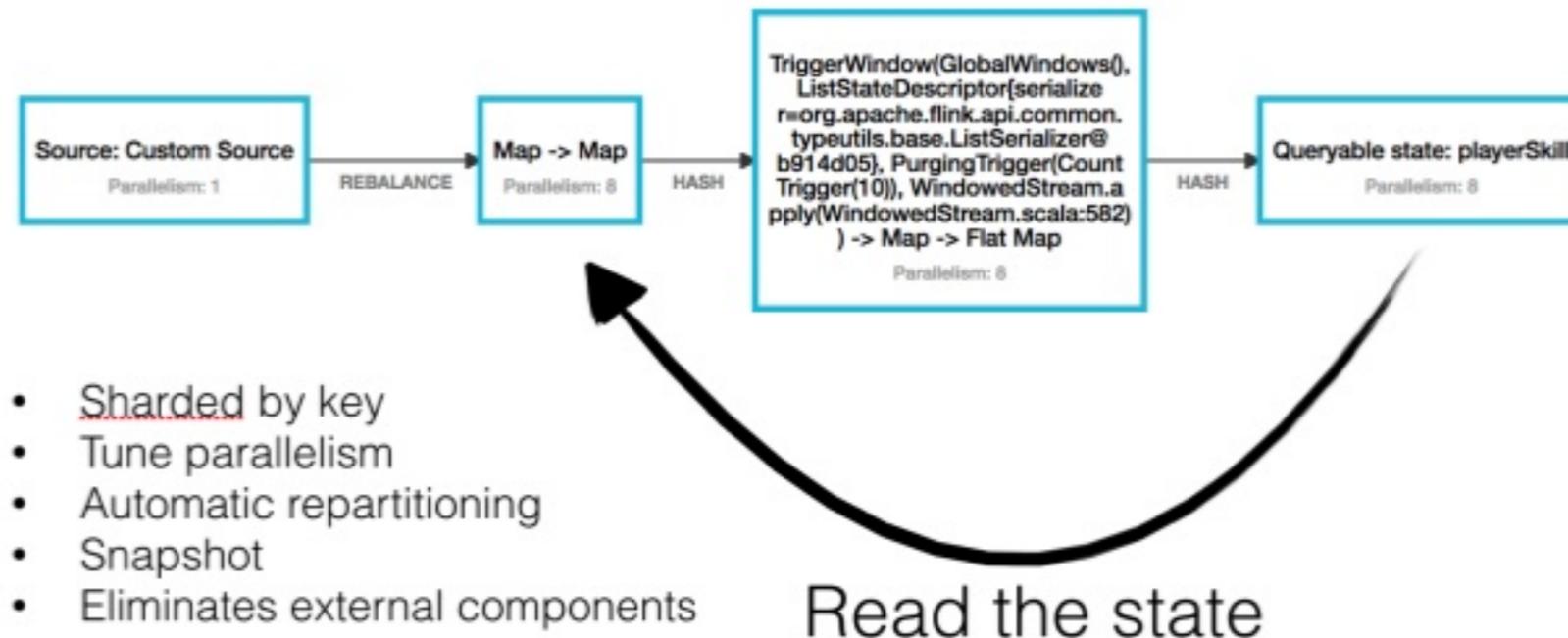
things we like

- We have **separate** functions in our architecture. The team selection step needs no notion of how many queues there are.
- We have interpretable hyperparameters. If we need to debug the system we can create meaningful visualisations.
- We can scale, should we get more users playing our game.

let's talk flink



managed state



parse the json to a case class

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_.id)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

if exists, get skill of the player

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_.id)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

grab appropriate queue bucket as the key

stream

```
.map(line => JsonUtil.parseJson[Player](line))
.map(new SamplePlayerSkill)
.keyBy(_.id)
.countWindow(LocalConfig.playersPerTeam * 2)
.apply(new DetermineTeam)
.map(new PlayGame)
.flatMap(new ComputeNewPlayerSkill)
.keyBy(_.player.id)
.asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

wait for bucket to have 10 players

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_.id)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

determine the teams of these players

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_.id)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

simulate the game

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_.id)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

map the game outcome to the player

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_.id)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

group player id; hash to the correct node

stream

```
.map(line => JsonUtil.parseJson[Player](line))
.map(new SamplePlayerSkill)
.keyBy(_.id)
.countWindow(LocalConfig.playersPerTeam * 2)
.apply(new DetermineTeam)
.map(new PlayGame)
.flatMap(new ComputeNewPlayerSkill)
.keyBy(_.player.id)
.asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

fold the new state into the current state

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_.id)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

code where most things happen

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_.id)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

time for demo

workflow

- You may have noticed our workflow relies partly on jupyter. If we did not visualise our logging we would have been left in the dark.

workflow

- You may have noticed our workflow relies partly on jupyter. If we did not visualise our logging we would have been left in the dark.
- We prefer logging to a file and loading that into jupyter to grafana. We need a flexible plotting engine and grafana is a bit limited.

workflow

- You may have noticed our workflow relies partly on jupyter. If we did not visualise our logging we would have been left in the dark.
- We prefer logging to a file and loading that into jupyter to grafana. We need a flexible plotting engine and grafana is a bit limited.
- Without flexible visualisation you're going to get stuck understanding your numerical system.

workflow

- It is hard to understand bugs. There may be an error in the code/system, in the maths or in a hyperparameter.

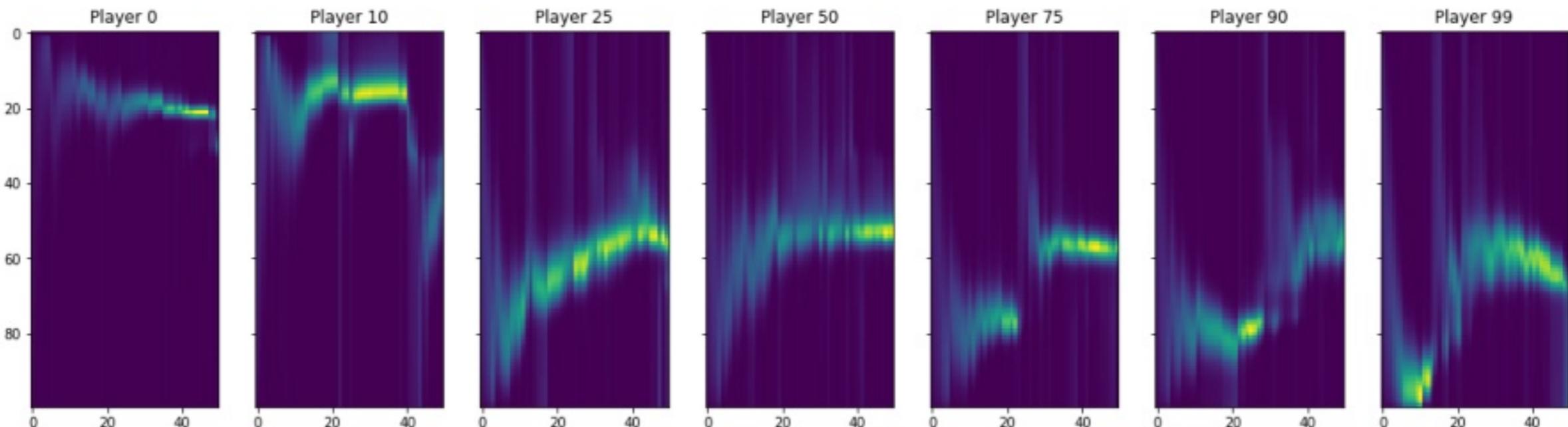
workflow

- It is hard to understand bugs. There may be an error in the code/system, in the maths or in a hyperparameter.
- For example. We did some smoothing to ensure that our posterior histogram never has a 0 value. You really want this because otherwise a user can never have a change in their skill level. Turns out that the smoothing parameter has a huge effect.

workflow

- It is hard to understand bugs. There may be an error in the code/system, in the maths or in a hyperparameter.
- For example. We did some smoothing to ensure that our posterior histogram never has a 0 value. You really want this because otherwise a user can never have a change in their skill level. Turns out that the smoothing parameter has a huge effect.
- We can demonstrate this with our player simulations.

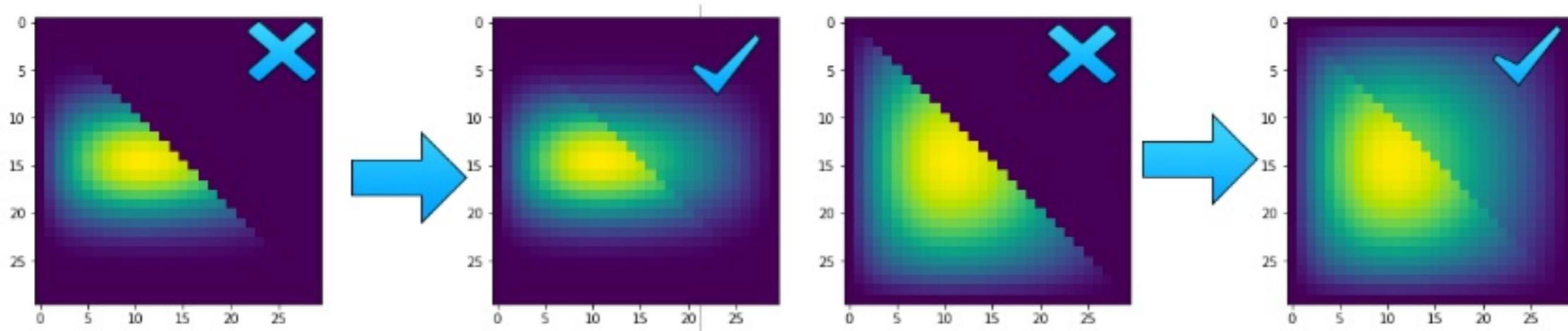
main issue: numeric stability of teams



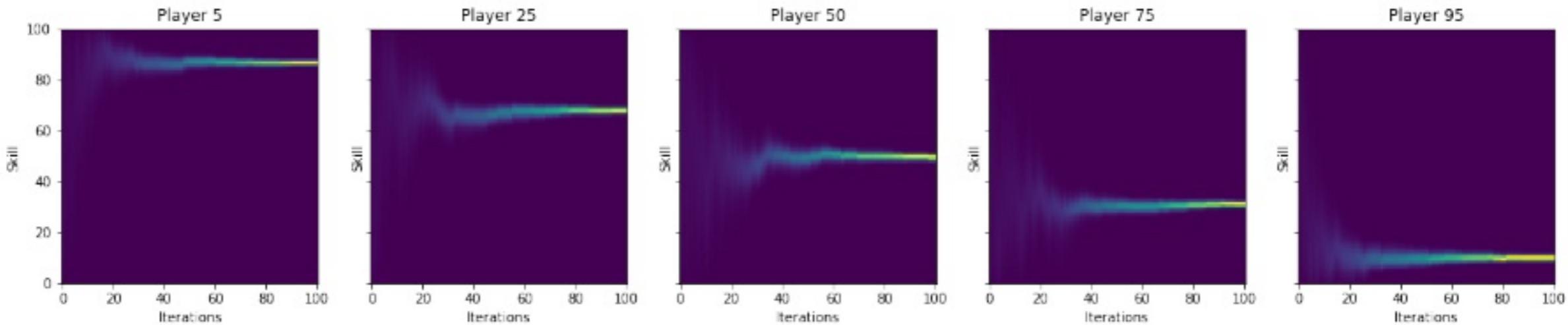
This is the 5v5 convergence. Ai caramba.

numerical debugging

We could quickly iterate on numerical scenarios very quickly with python. It really helps to have plotting available. We solved this smoothing issue by introducing a different smoothing method all together.

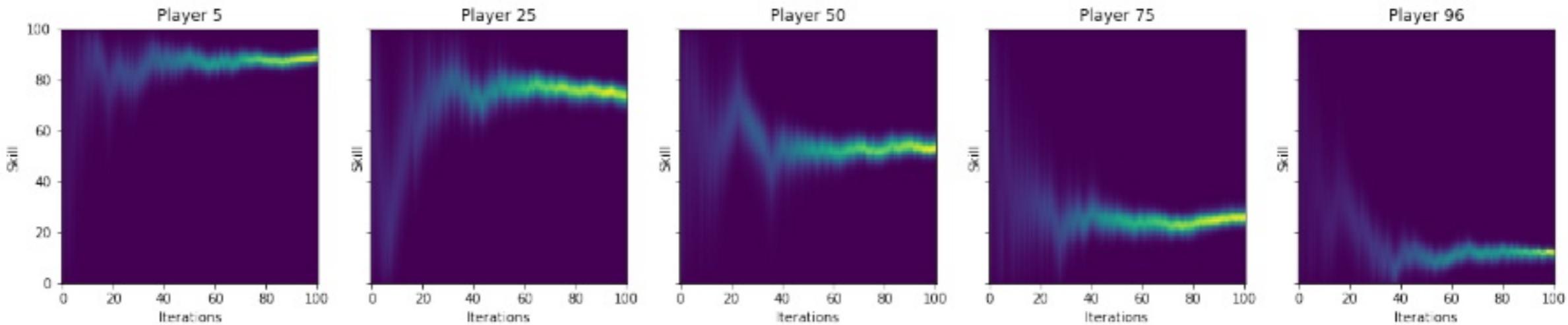


numeric stability of individuals



Above is the 1v1 convergence, exactly the same as pokémon.

numeric stability of teams

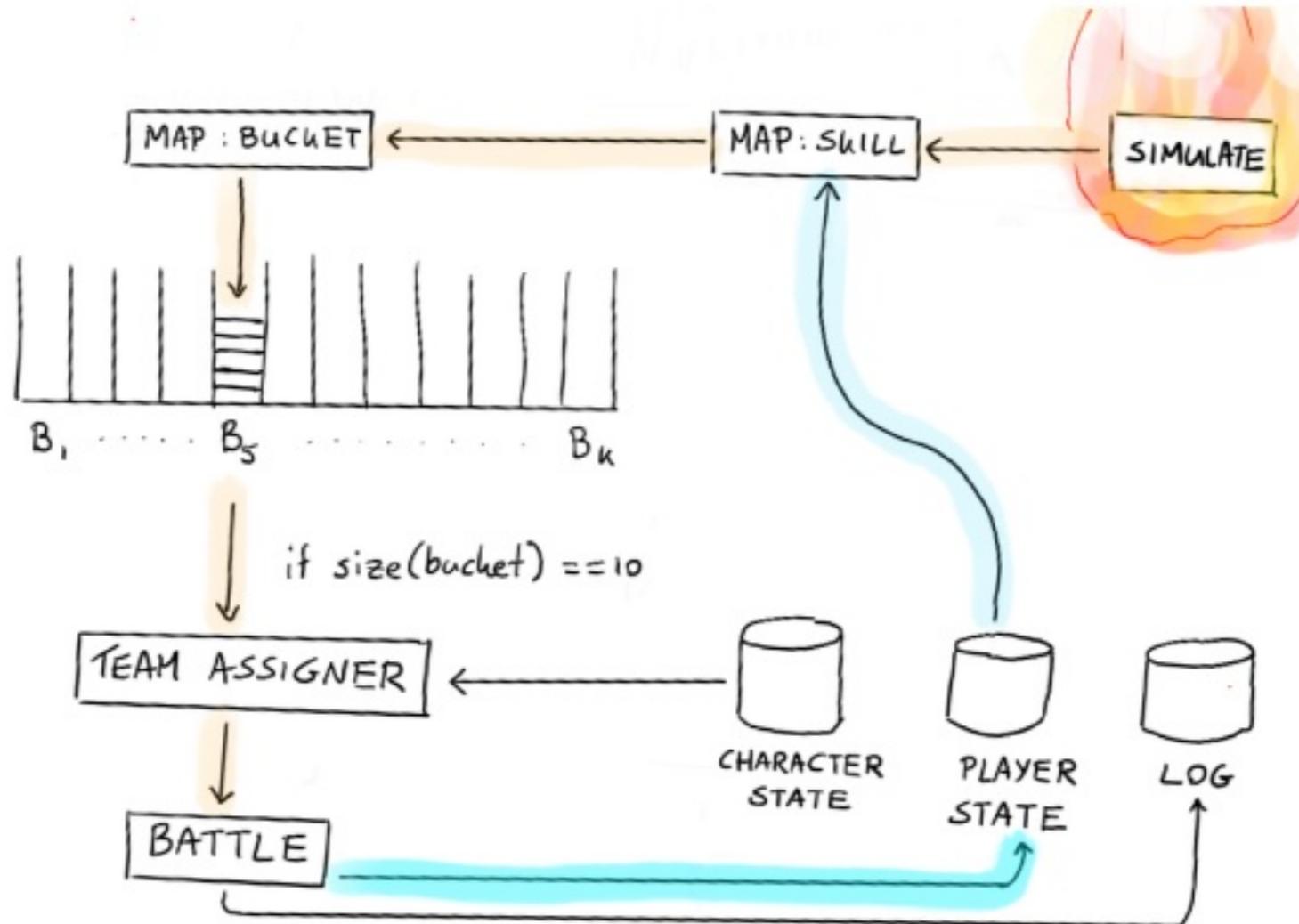


This is the 5v5 convergence, more jitter, but converges.

best practices

It is great to have degrees of freedom, but the parameter tuning is a hard requirement with this system and things definitely can go wrong if you're not careful.

Not all errors are numeric though. Some errors exist because of the fact that distributed streaming needs to be approached by a different mindset.



best practices

When debugging, maybe focus on **logging** from scala such that you can easily review this data in a jupyter notebook. Proper **visualisation** will be a saving grace. You may need more than simple timeseries charts though.

best practices

When debugging, maybe focus on **logging** from scala such that you can easily review this data in a jupyter notebook. Proper **visualisation** will be a saving grace. You may need more than simple timeseries charts though.

When debugging, be extremely scientific. Eliminate variables with unit tests until you're absolutely sure **what** is going wrong.

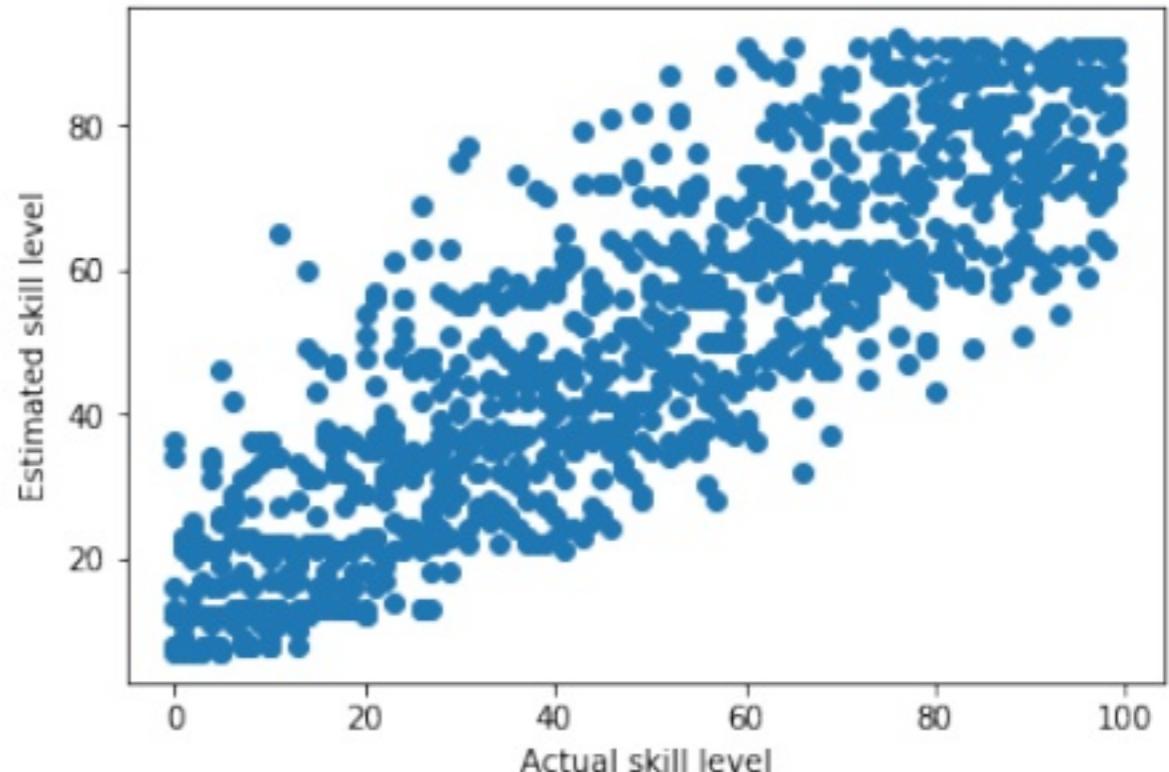
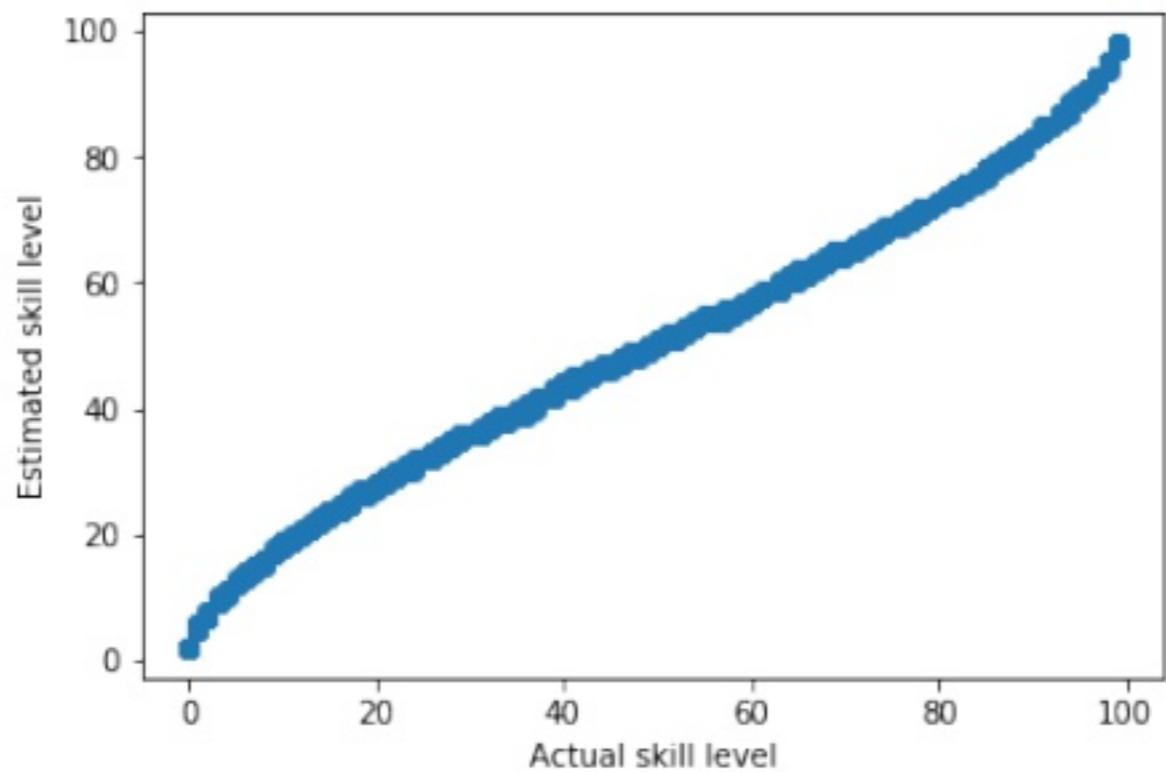
best practices

When debugging, maybe focus on **logging** from scala such that you can easily review this data in a jupyter notebook. Proper **visualisation** will be a saving grace. You may need more than simple timeseries charts though.

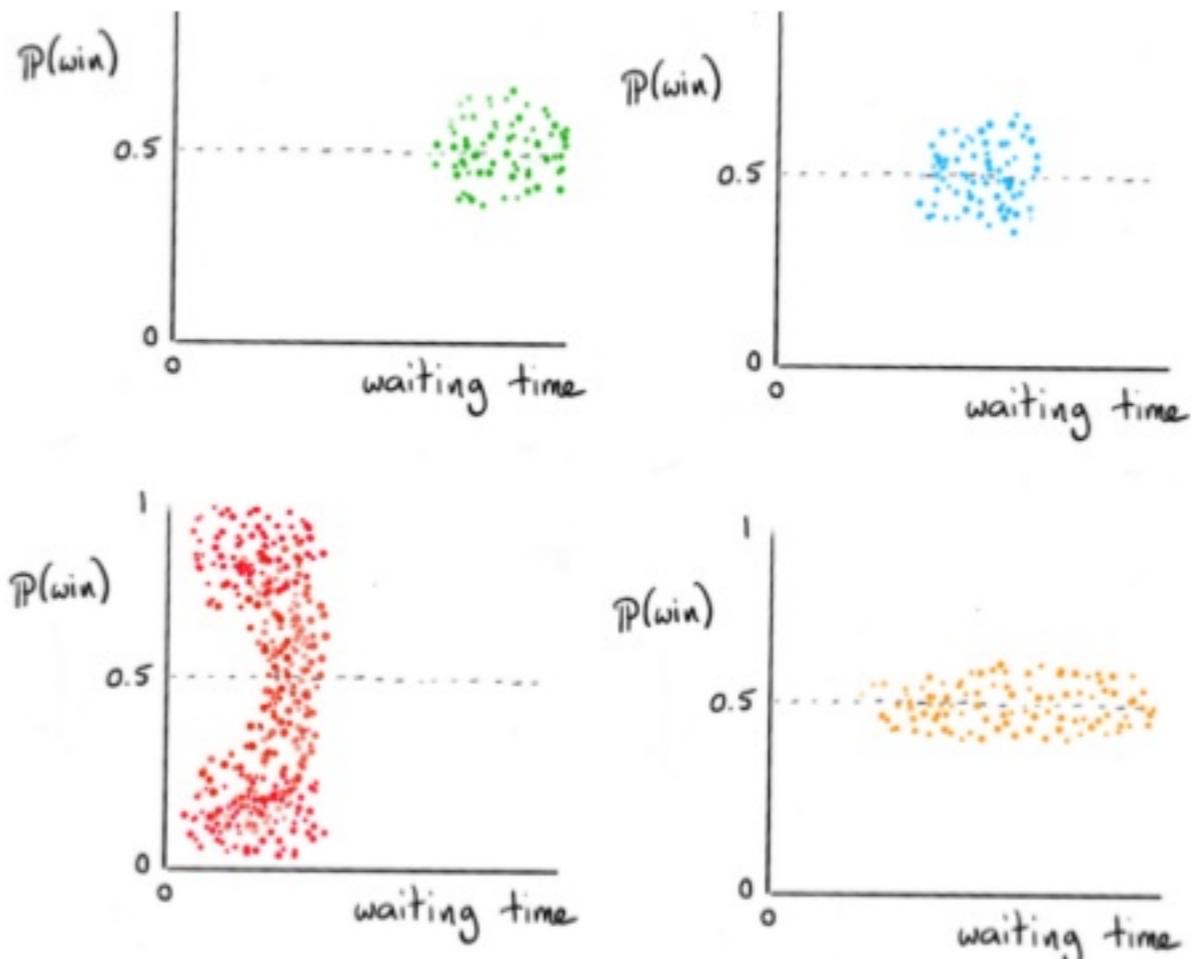
When debugging, be extremely scientific. Eliminate variables with unit tests until you're absolutely sure **what** is going wrong.

Confirming that 1v1 teams and 2v2 teams worked helped us understand how to improve 5v5 games.

end result: 1v1 vs 10v10 convergence



main problem: no typical scheduling problem



conclusions

You can have machine learning models that learn in a streaming setting if you appropriately apply bayes rule.

Having distributed state in Flink is great, it allows for a stack with less moving parts and helps you scale.

We assume that all players are new, in practice new plays will flow in the already converged system.

When making such a system, please consider prototyping assumptions and heuristics

thanks for listening

Code of this project is on [github](#).

Slides will also become available.

Feel free to reach out to us via twitter: [fokko](#) & [vincent](#).