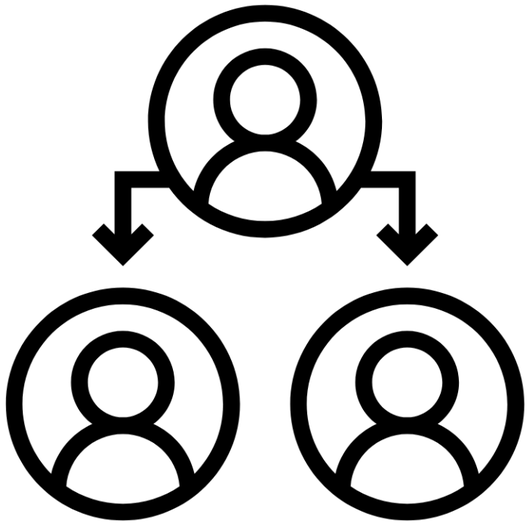
Balancing Heroes and Pokemon

Streaming/Bayesian Systems for Online Ranking

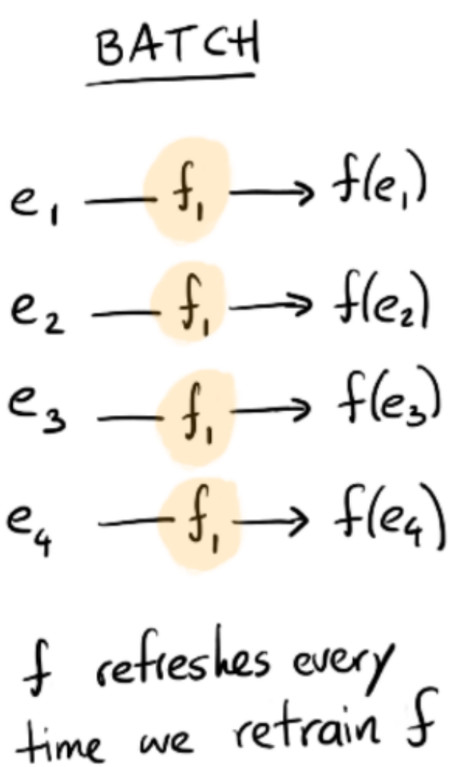


Fokko Driesprong & Vincent D. Warmerdam - GoDataDriven

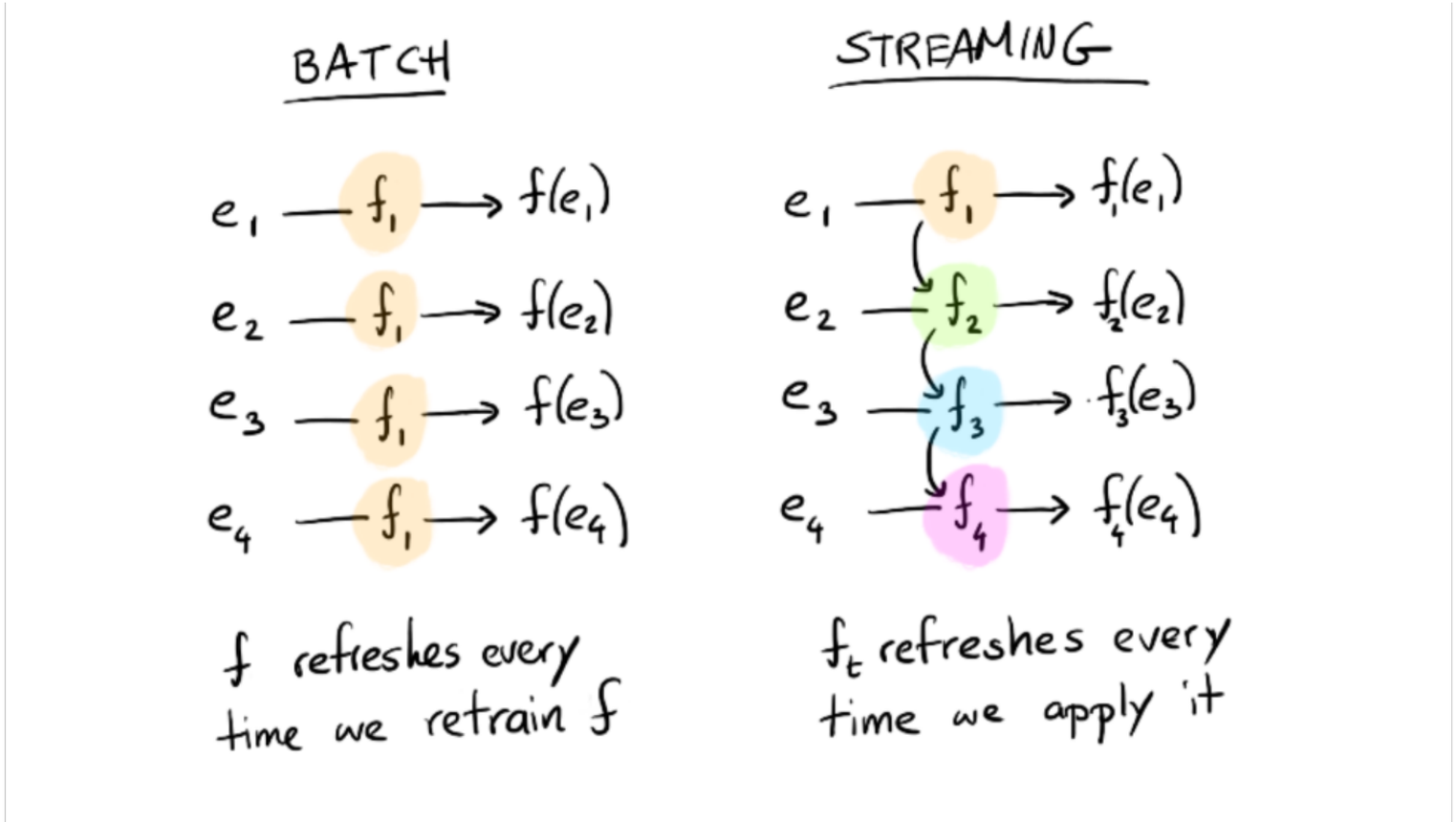
we are fokko and vincent

- 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 started out as a mere nerd snipe.

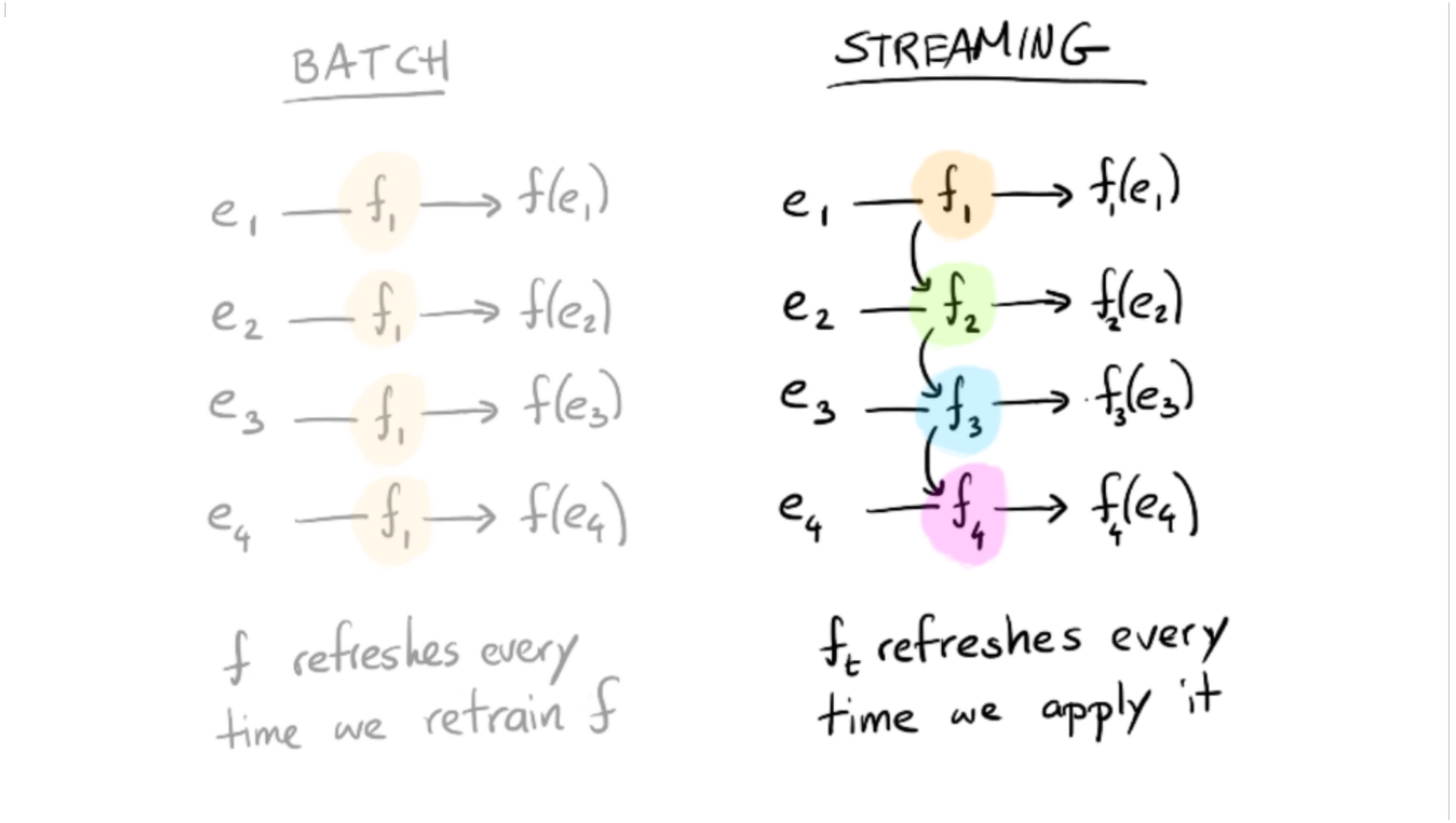
usually ML is applied as a lambda



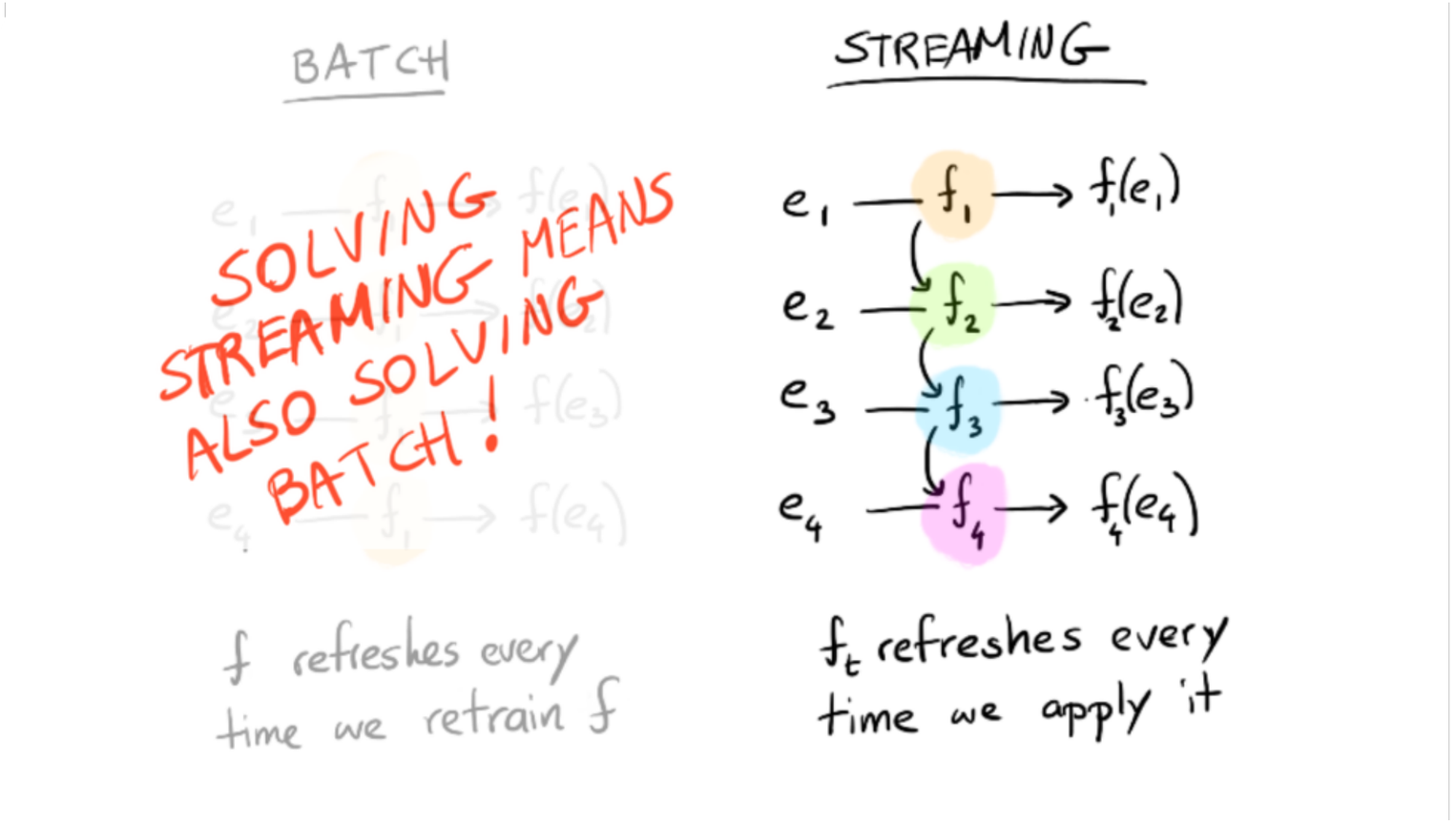
but it could be trained while being applied



streaming may be preferable

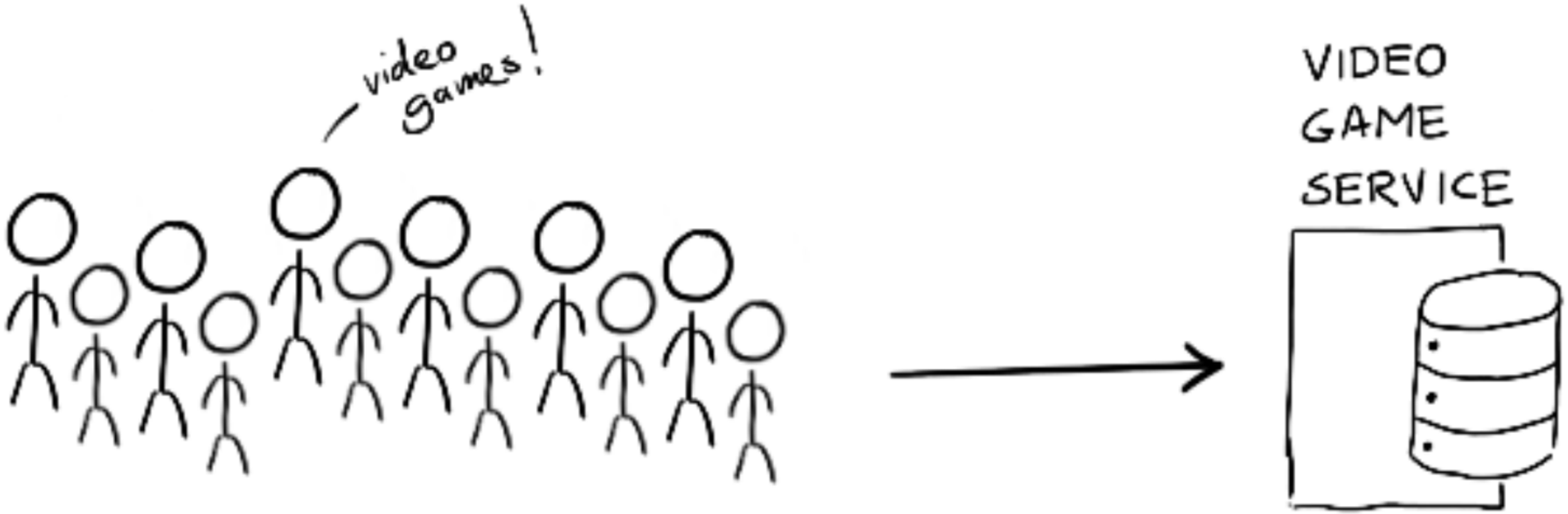


we will give an example in this talk

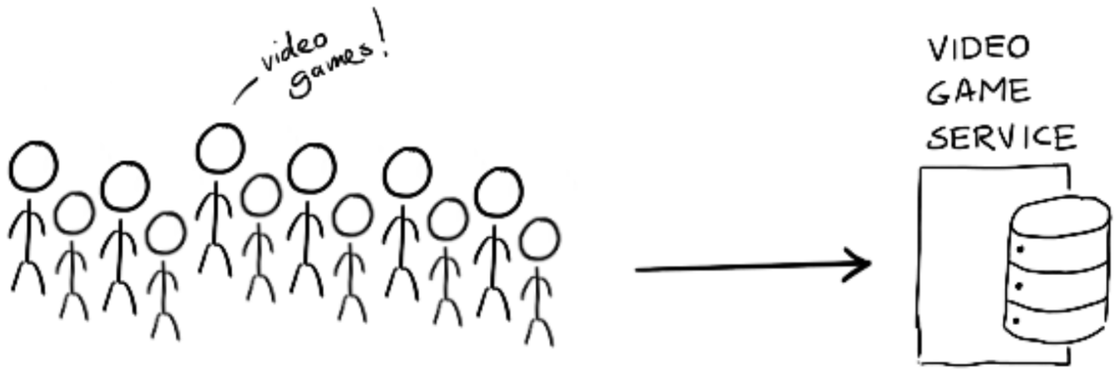


problem at hand

This is our enterprise usecase.



problem at hand

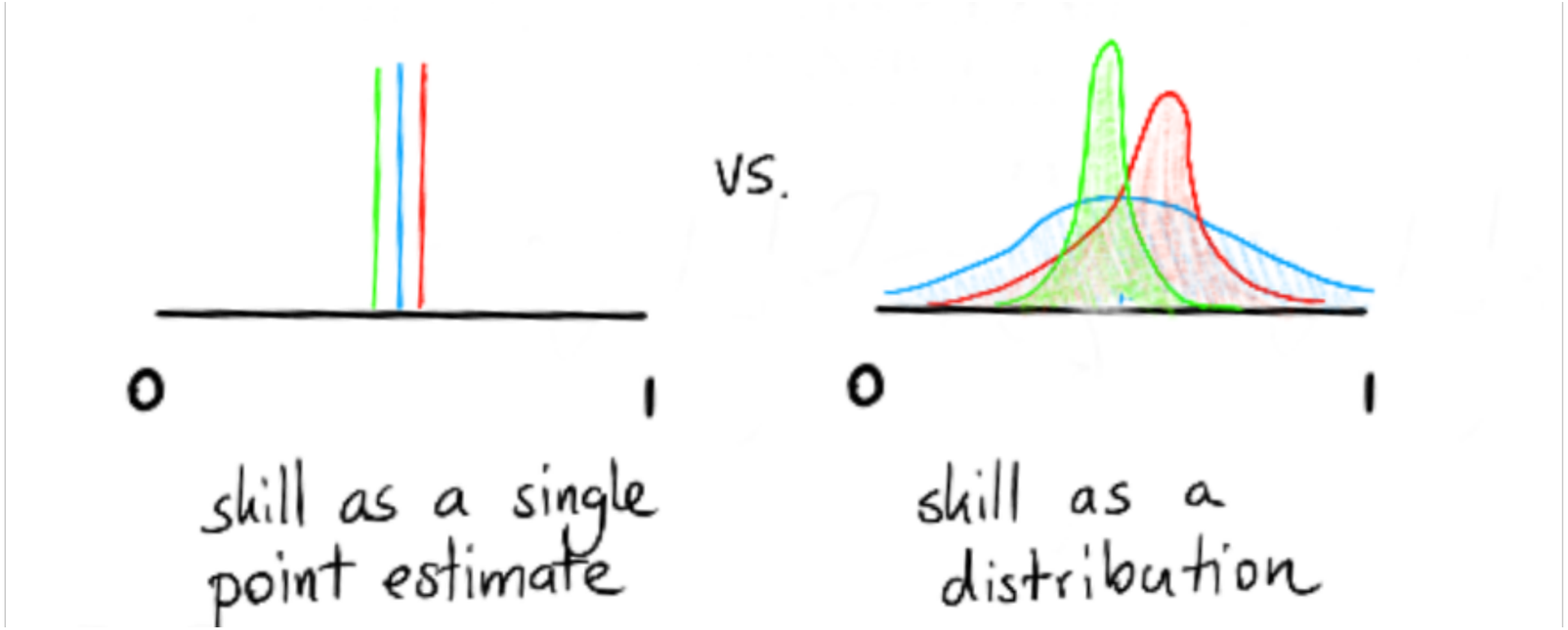


- we need to estimate player skill somehow
- we need to learn this from a stream of match outcomes
- we don't want to wait for a batch algorithm

let's talk solutions

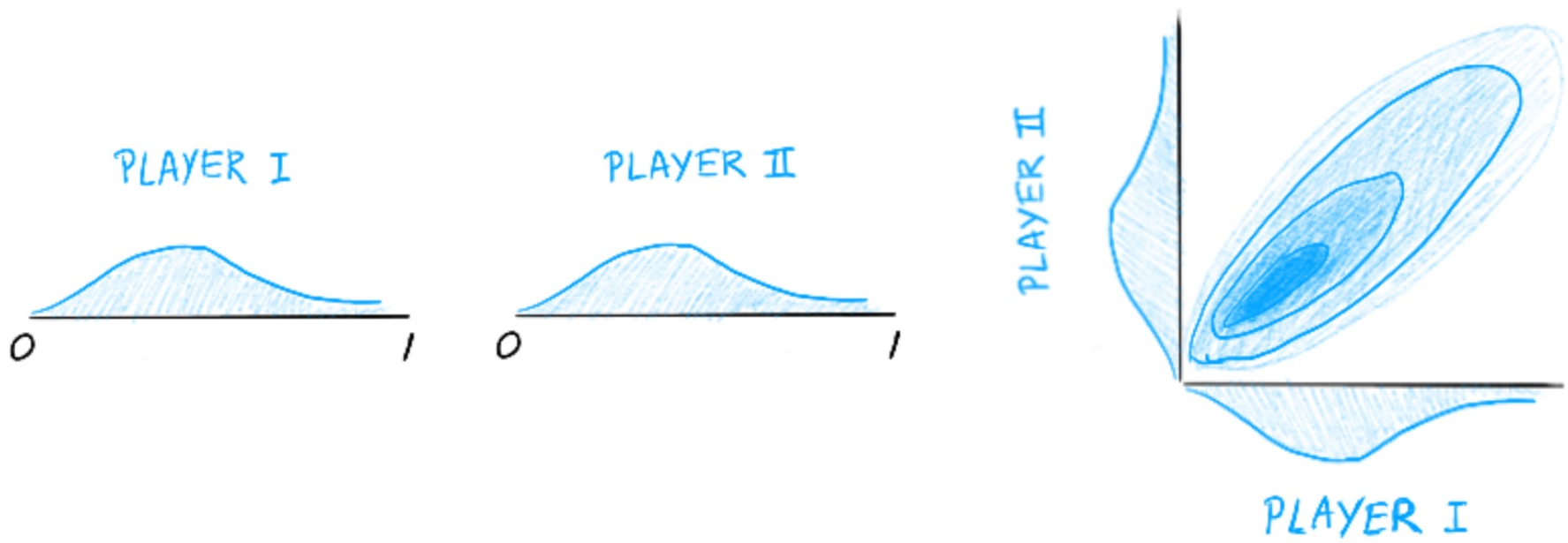
a simple solution

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.



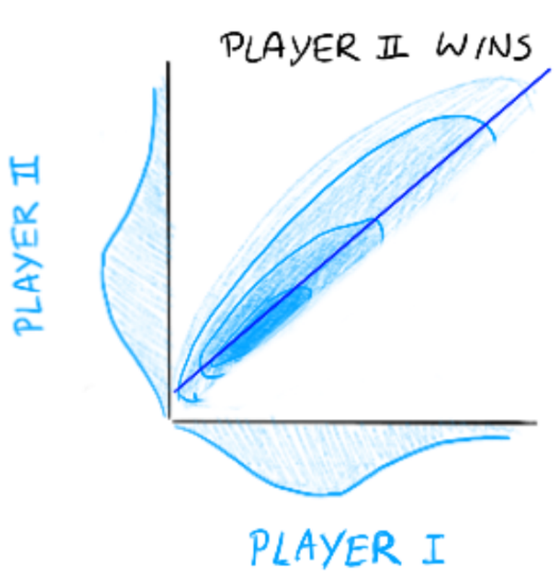
a simple solution

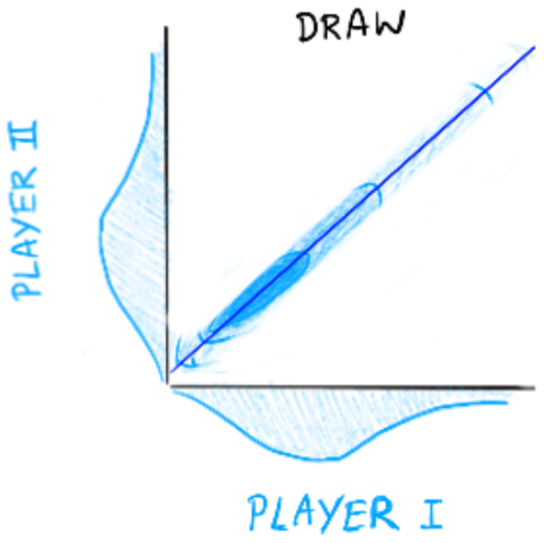
The next step is to realize that you can combine two one-player beliefs of skills into a one two-player belief. A prior of belief.



a simple solution

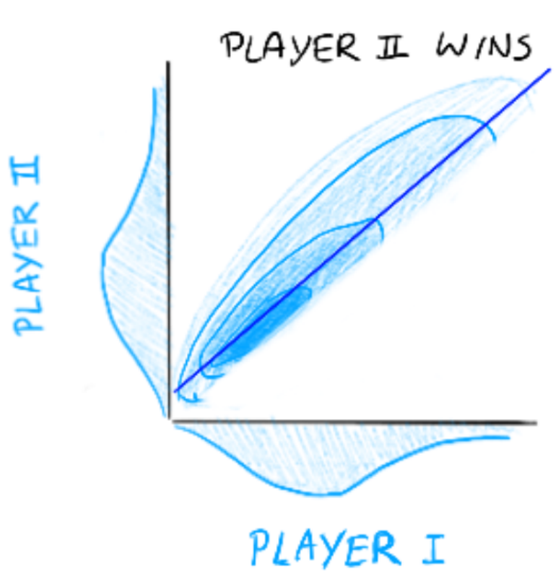
After a game has been played the two dimensional prior is updated depending on what the data shows us.

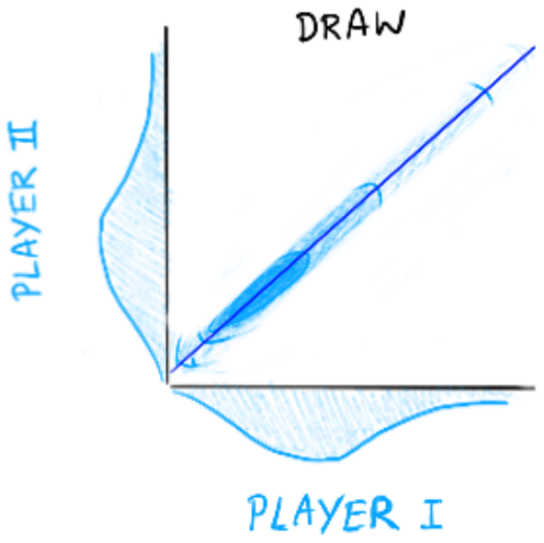




a simple solution

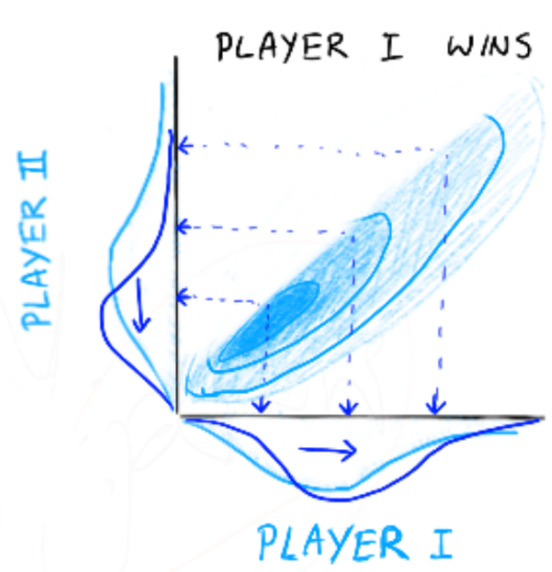
We take a **margin** over the diagonal and any probability mass from the region that disagrees with the match outcome.

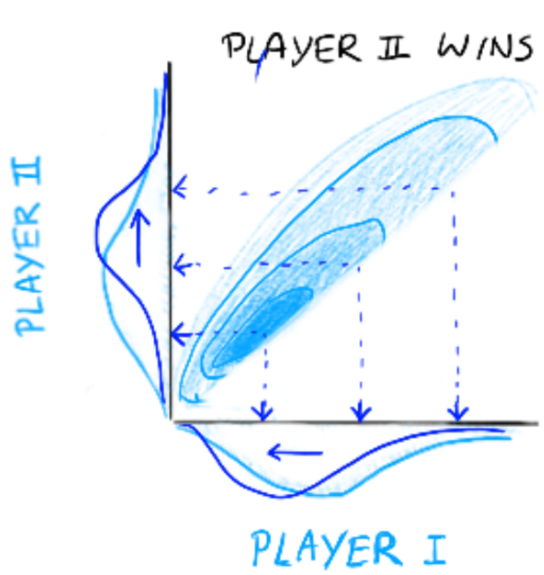


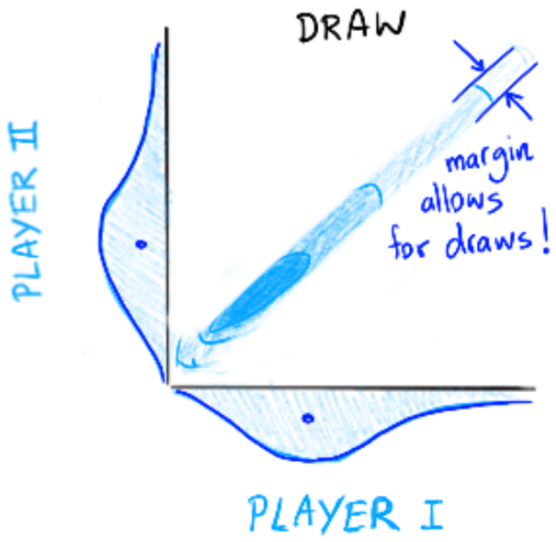


a simple solution

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

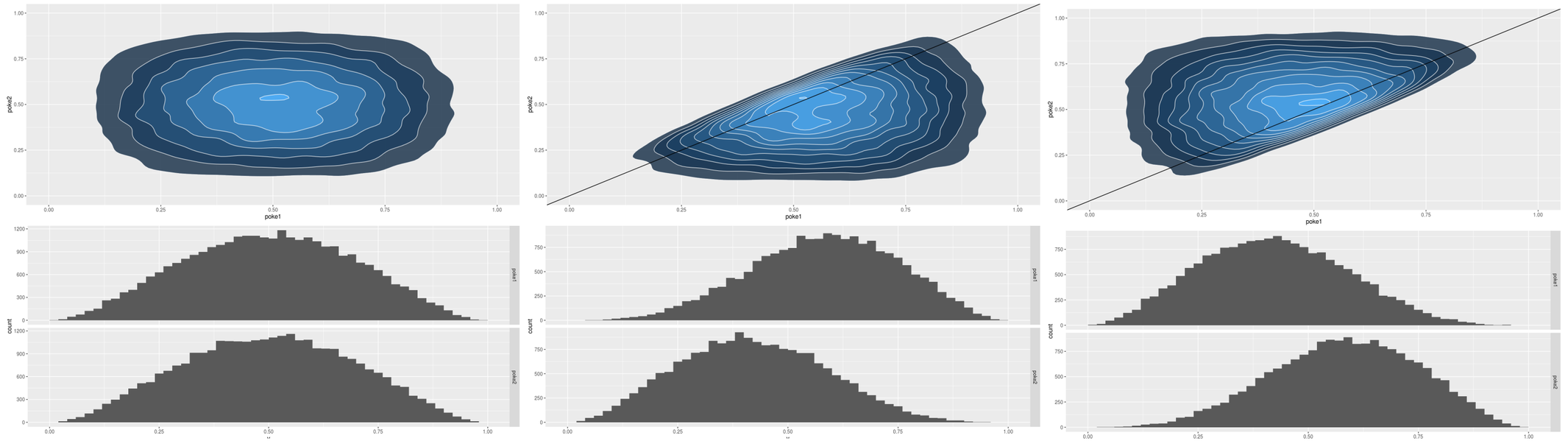






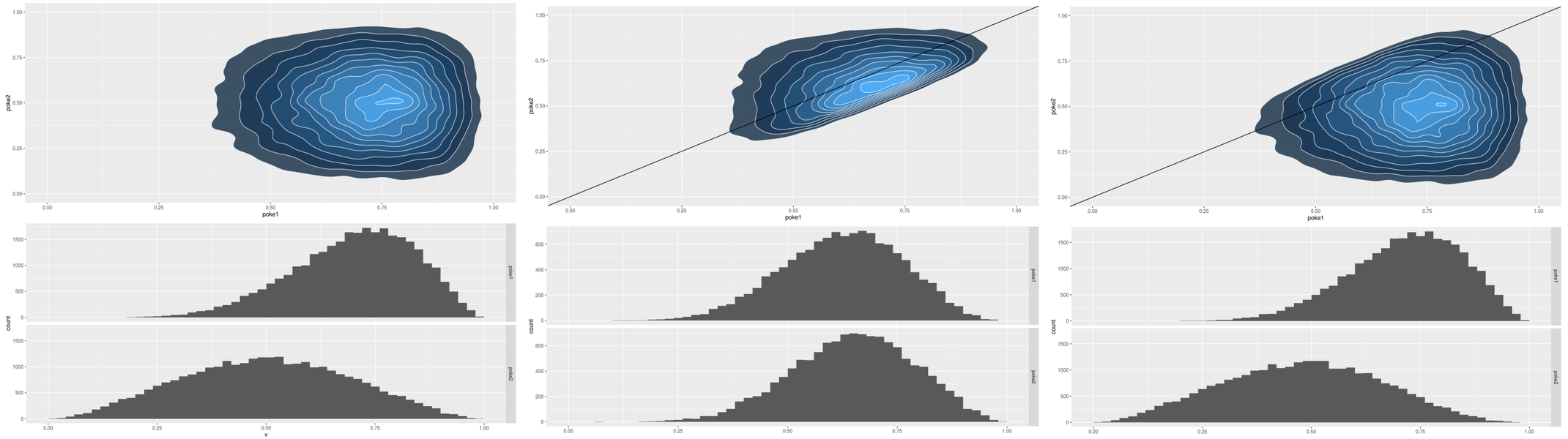
a simple simulation: two equal players

There are some benefits we get for free. Try it out here.

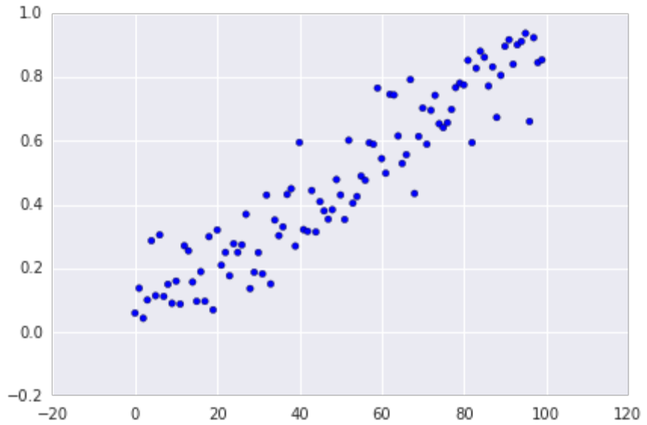


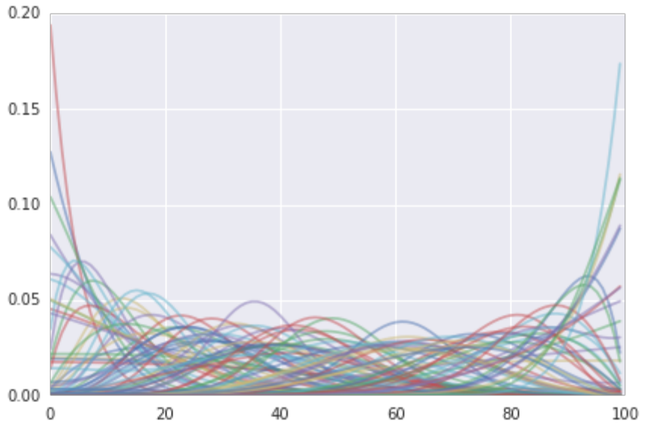
a simple simulation: two unequal players

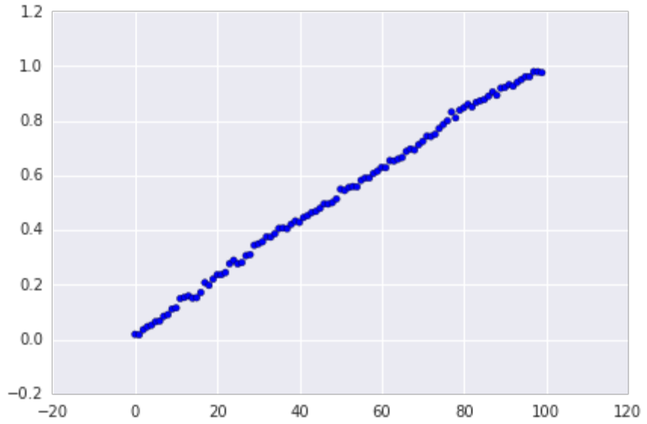
We may learn a lot, or very little. `#informationtheory`

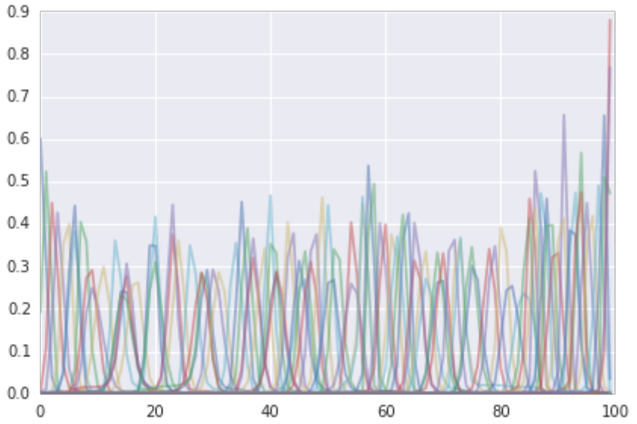


a simple simulation: many players





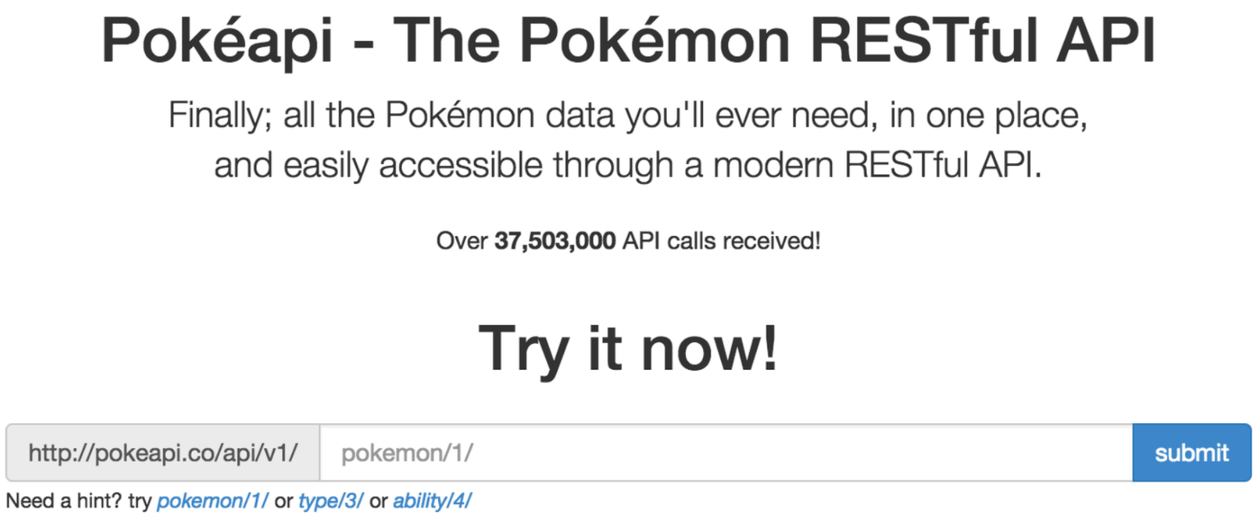




test of functionality: pokemon!

test of functionality: pokemon!

We got our infromation on pokemon from;



Nowadays you can also find the dataset on kaggle.

test of functionality: pokemon!

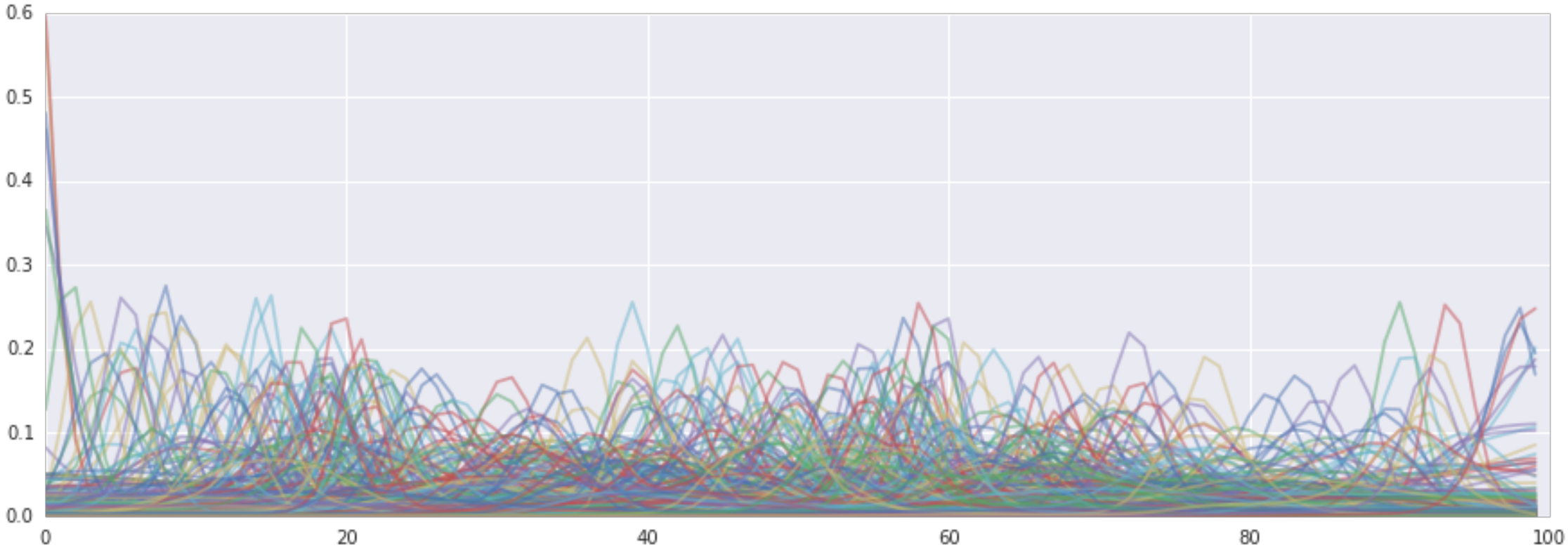
If you google around fan reddits you can find information on how many turns one pokemon can outlast the other.

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

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

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

test of functionality: pokemon!



test of functionality: pokemon!

	name	mle
128	Magikarp	0.011007
112	Chansey	0.013735
348	Feebas	0.015845
171	Pichu	0.019108
291	Shedinja	0.020473
439	Happiny	0.026763
241	Blissey	0.037009
172	Cleffa	0.042802
234	Smeargle	0.048575
49	Diglett	0.054010

	name	mle
597	Ferrothorn	0.916377
425	Drifblim	0.917240
644	Zekrom	0.919065
537	Throh	0.921175
482	Dialga	0.928502
149	Mewtwo	0.951472
483	Palkia	0.953823
288	Slaking	0.956629
375	Metagross	0.957383
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.

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

Bayesians to the rescue

We can estimate parameters by considering;

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

Let's consider three independent points of data that we've seen; $d1, d2, d3$. 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?

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

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

bayesians to the rescue

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

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

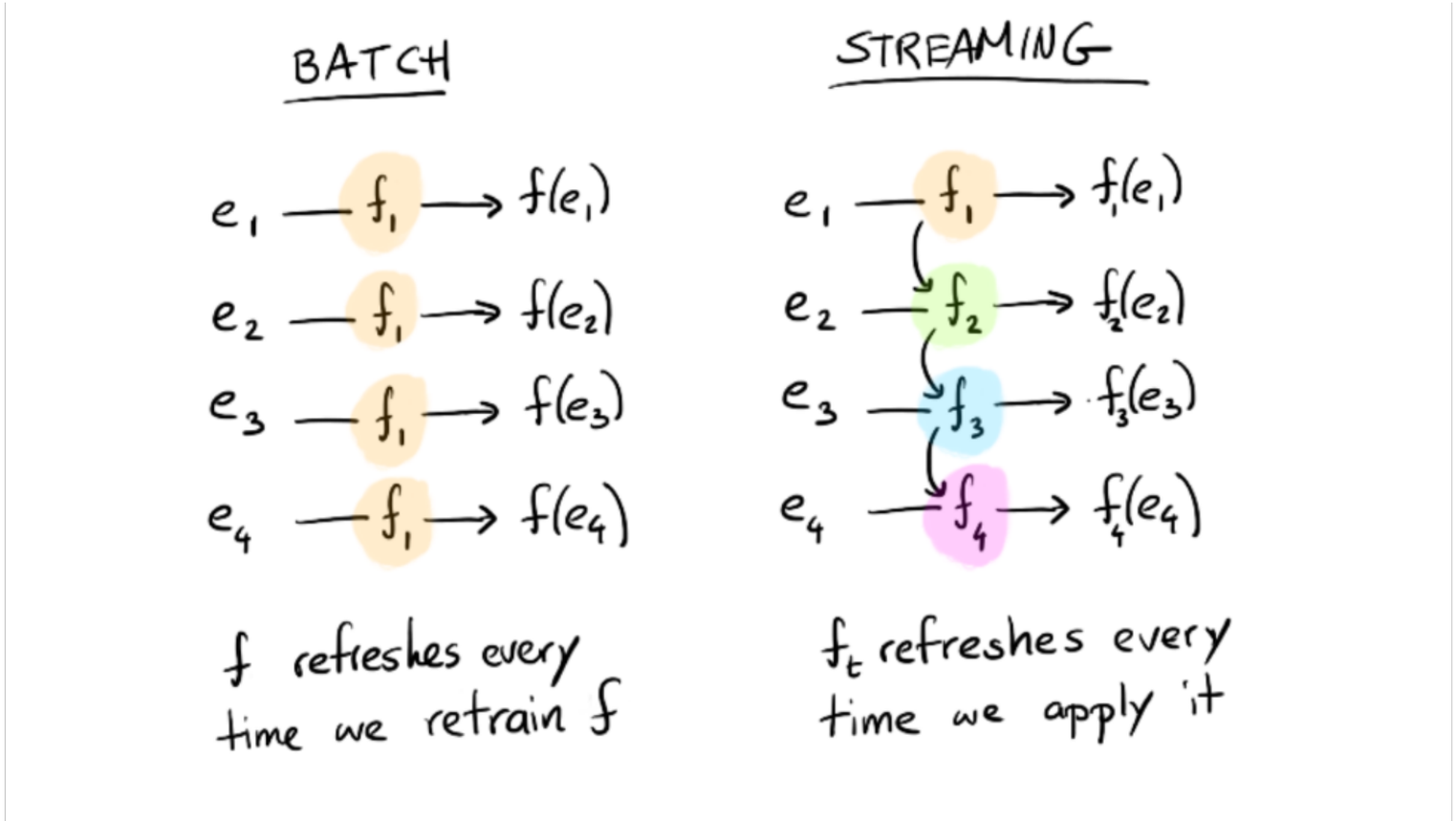
bayesians to the rescue

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.

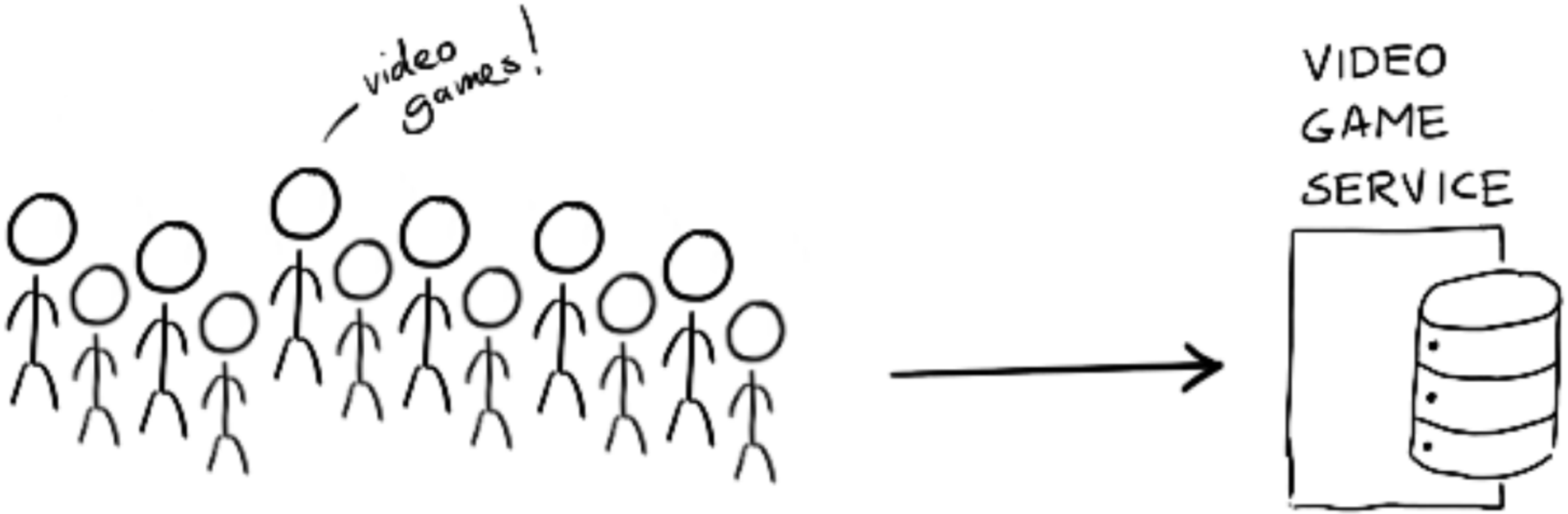
we just gave an example of this



a harder problem: heroes of the storm

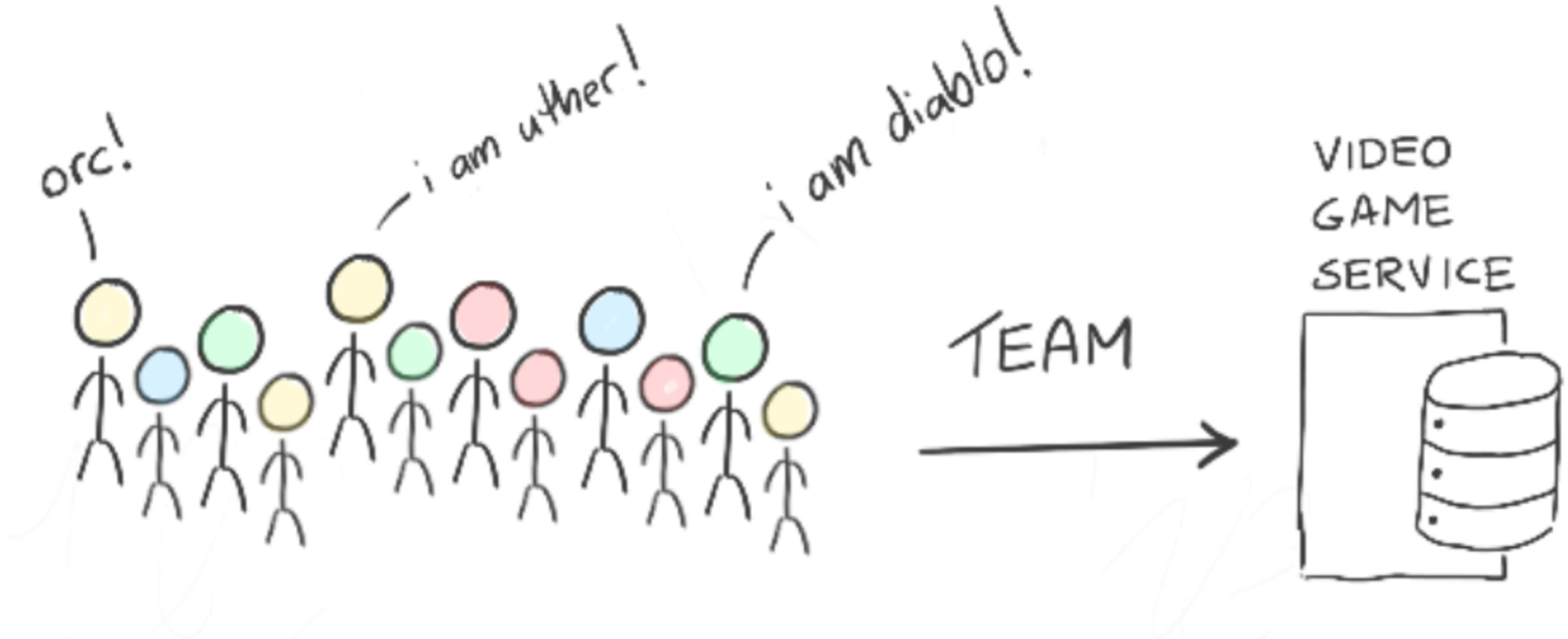
a harder problem: heroes of the storm

This was our enterprise usecase before.

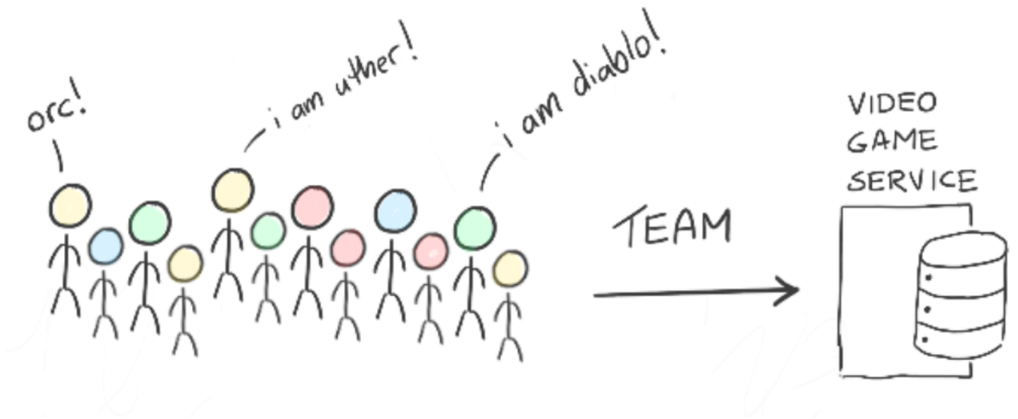


a harder problem: heroes of the storm

This is our even more enterprise usecase now.

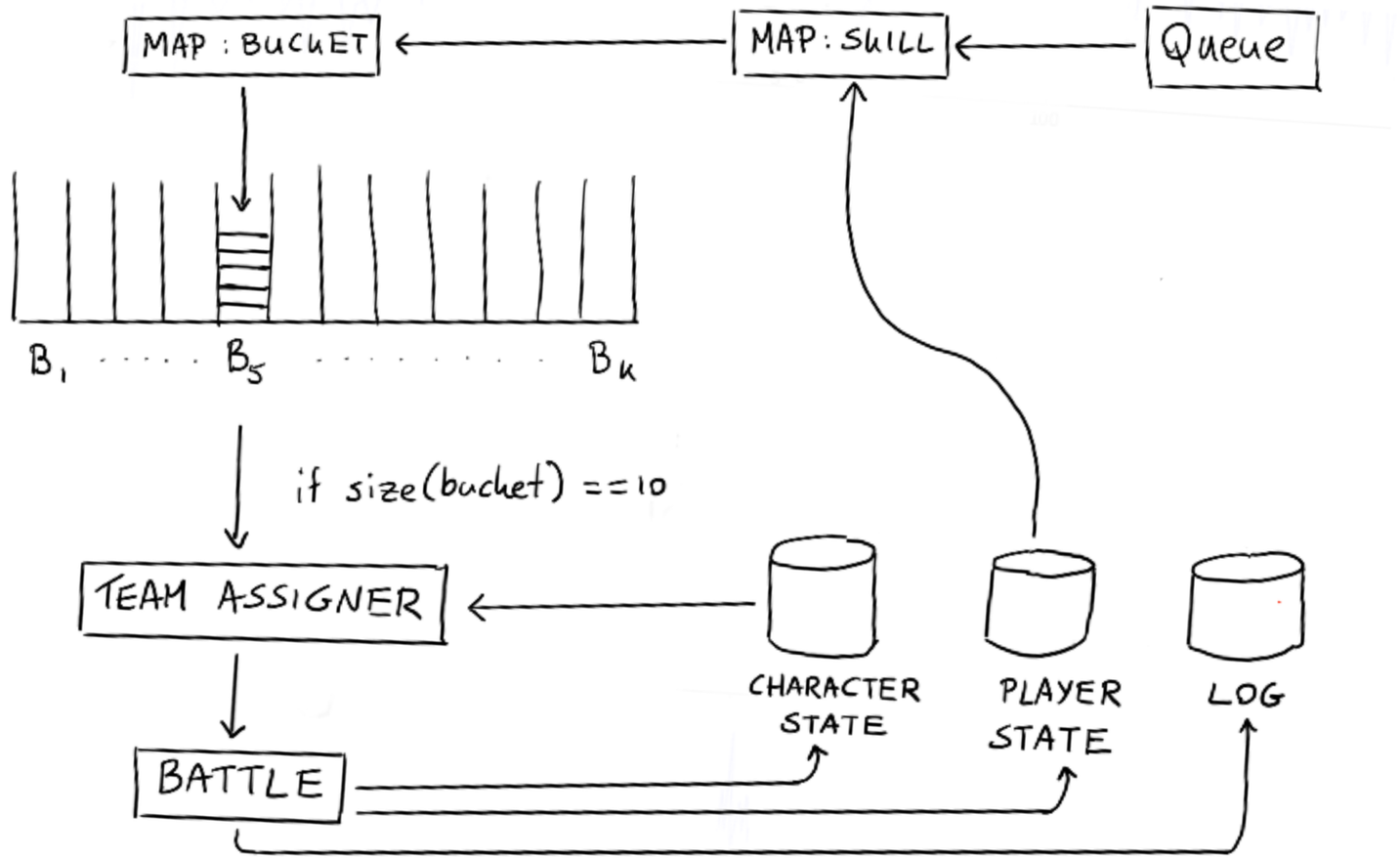


a harder problem: heroes of the storm



- 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

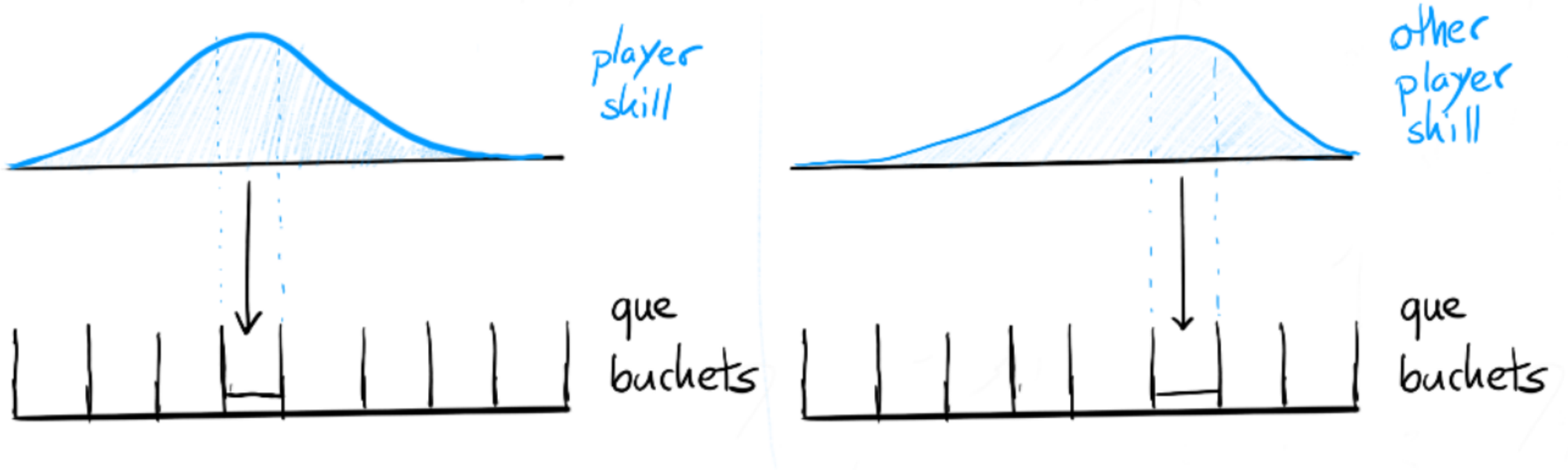


proposal solution architecture

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

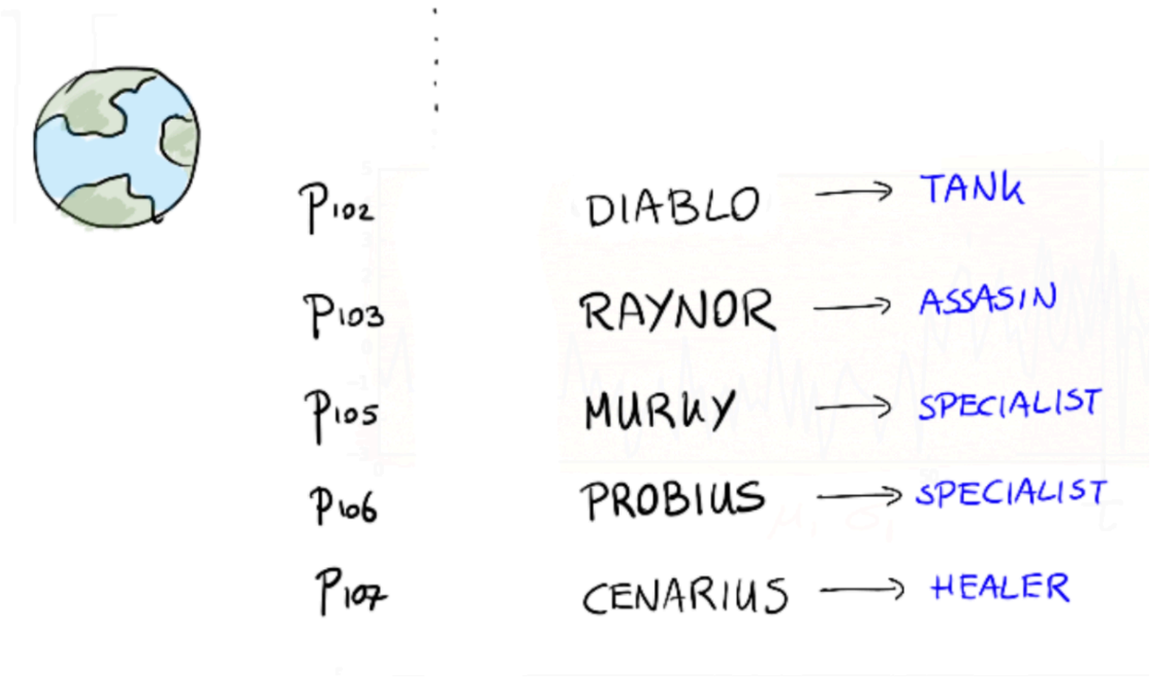
Predicting team imbalance with ML is tricky. There are lot's of combinations of heroes ($n = 67$) that are possible.

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

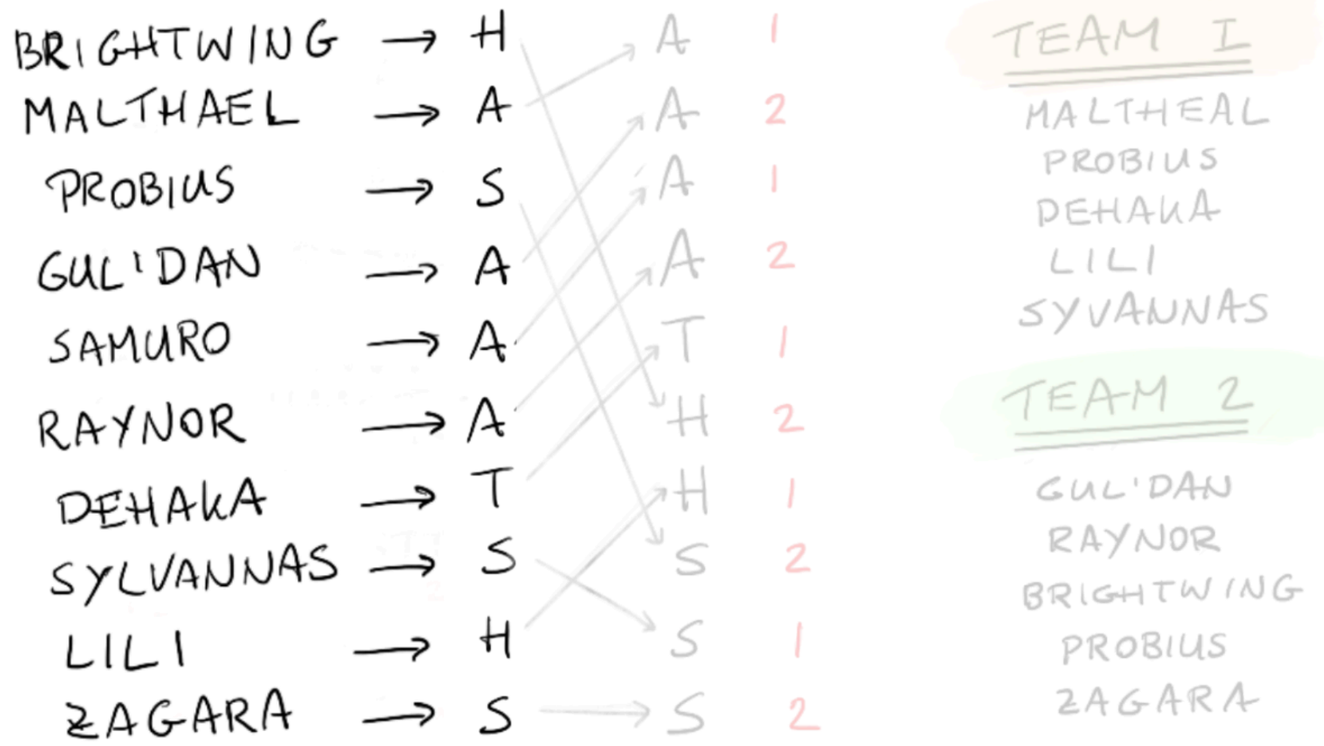
This makes it hard to make some sort of machine learning model that can predict what team might win. Is there an easy hack?

teamfights: assigning teams

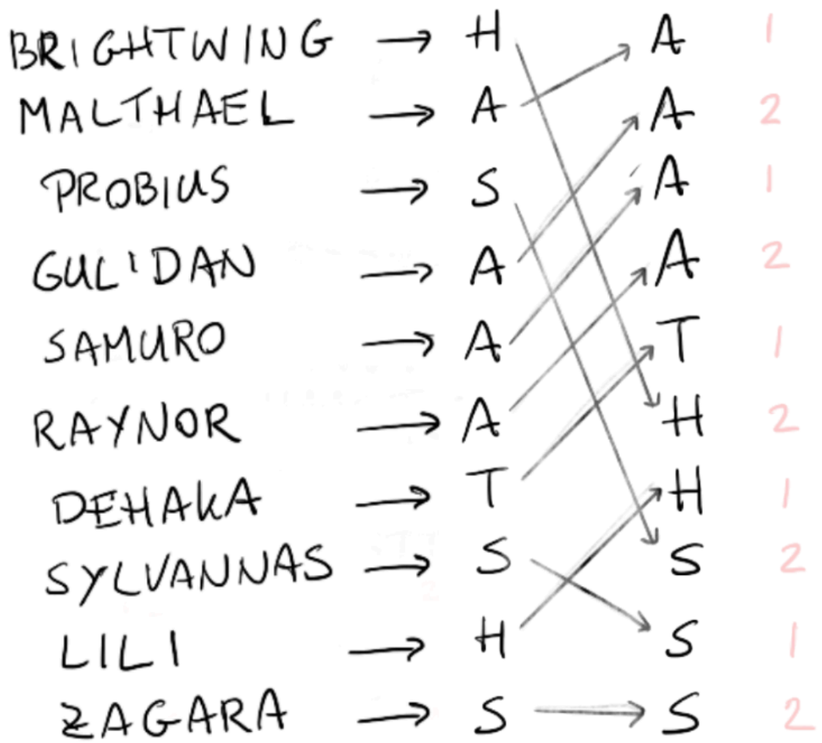
We have some domain knowledge; types of heroes.



teamfights: assigning teams



teamfights: assigning teams



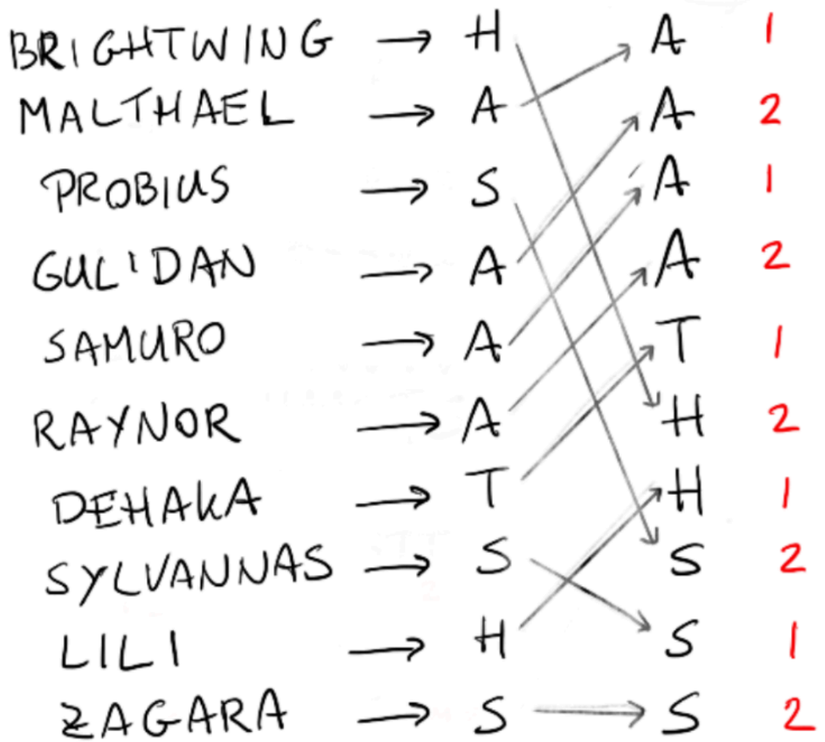
TEAM 1

MALTHEAL
PROBIUS
DEHAKA
LILI
SYVANNAS

TEAM 2

GUL'DAN
RAYNOR
BRIGHTWING
PROBIUS
ZAGARA

teamfights: assigning teams



TEAM 1

MALTHEAL
PROBIUS
DEHAKA
LILI
SYVANNAS

TEAM 2

GUL'DAN
RAYNOR
BRIGHTWING
PROBIUS
ZAGARA

teamfights: updating skill

- Each game is a single ten-dimensional distribution of skill. We can cut where appropriate $\rightarrow$ problematic when our internal state is a histogram.
- 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 $\rightarrow$ mathematical complexity $\rightarrow$ maintenance risk

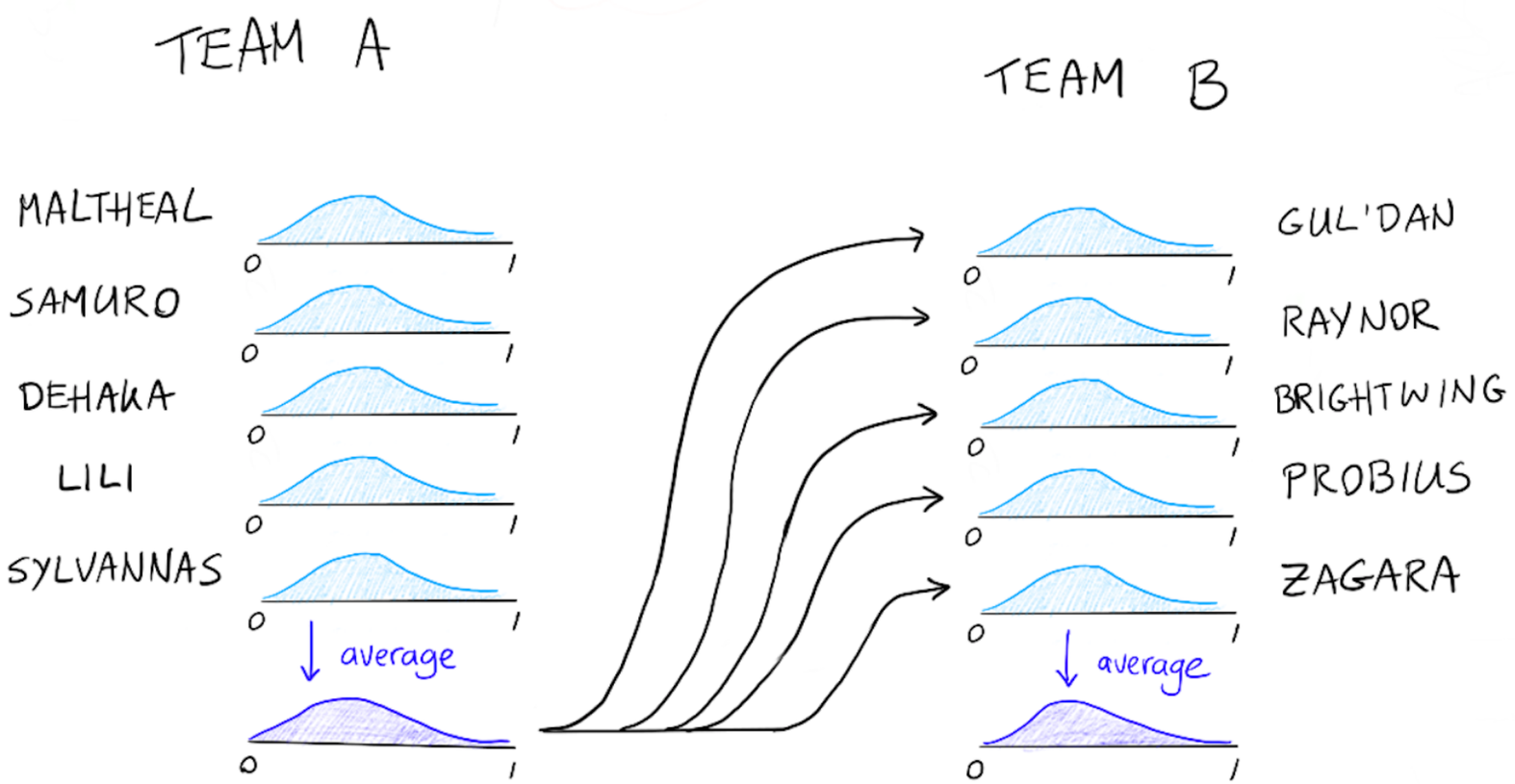
teamfights: updating skill

~~- Each game is a single ten-dimensional distribution of skill. We can cut where appropriate → problematic when our internal state is a histogram.~~

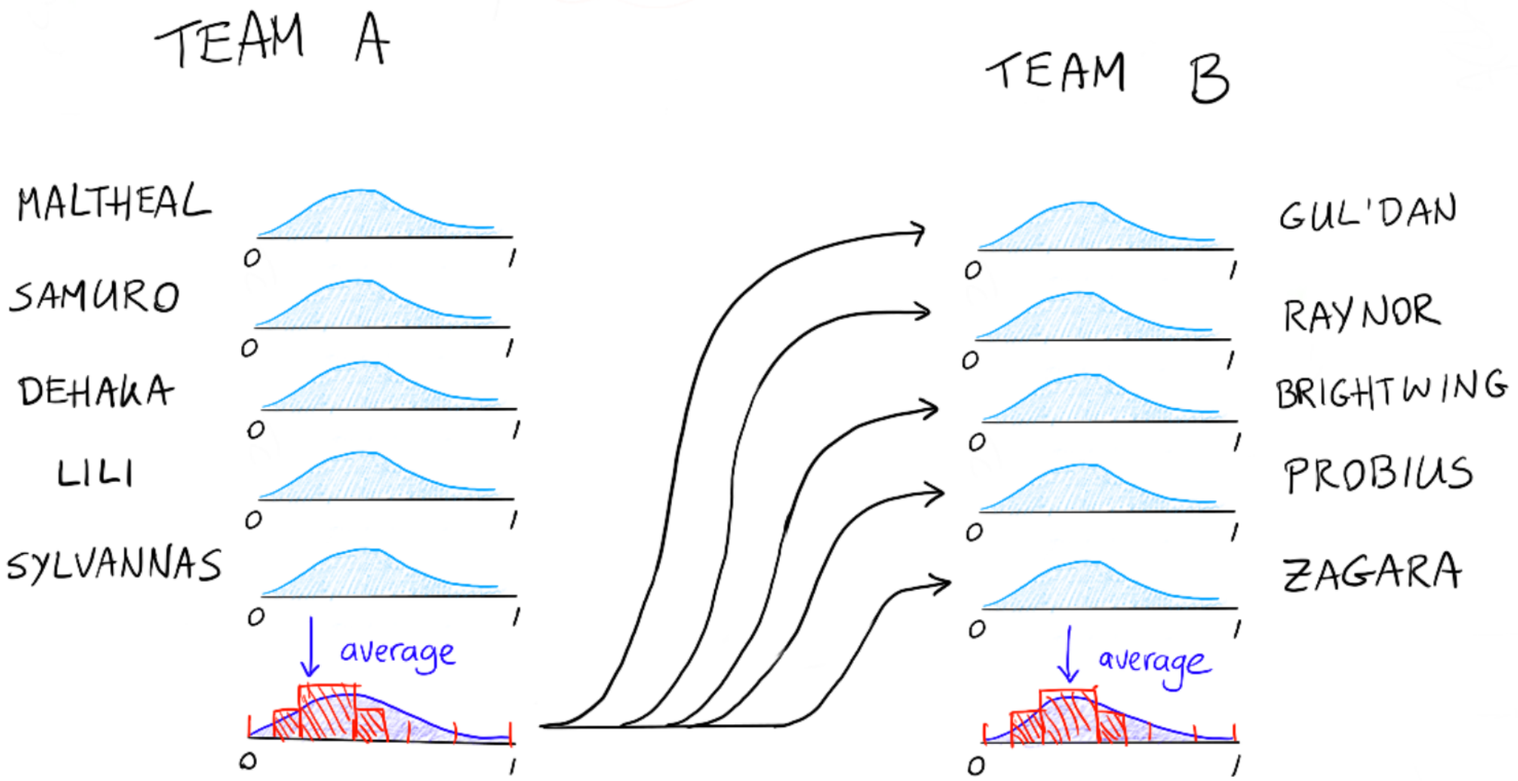
- Cheat a bit by summarising all opponent histogram into a single histogram and pretend that the player was battling this person

~~- Have the data scientist apply maths to make an update rule which involves proper distributions instead of histograms → mathematical complexity → maintenance risk~~

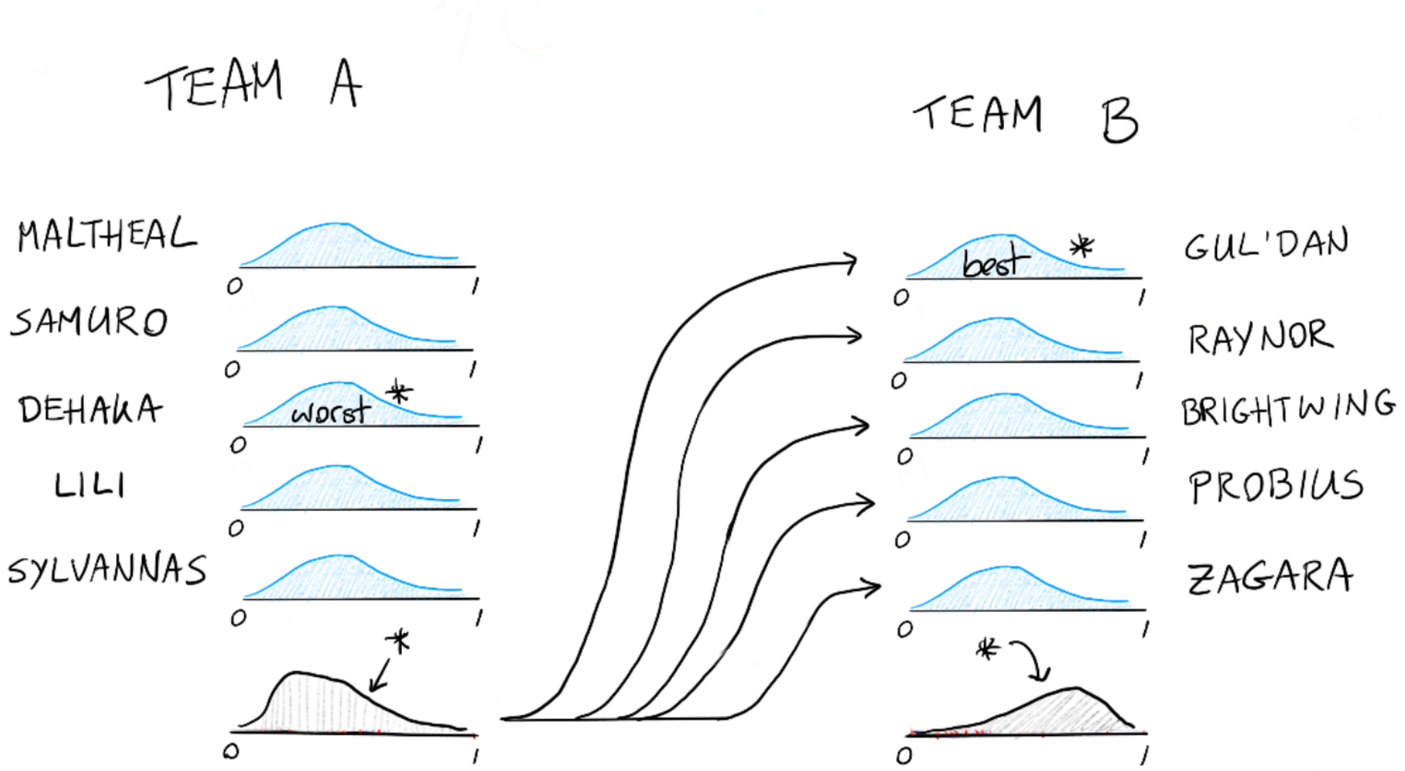
teamfights: updating skill



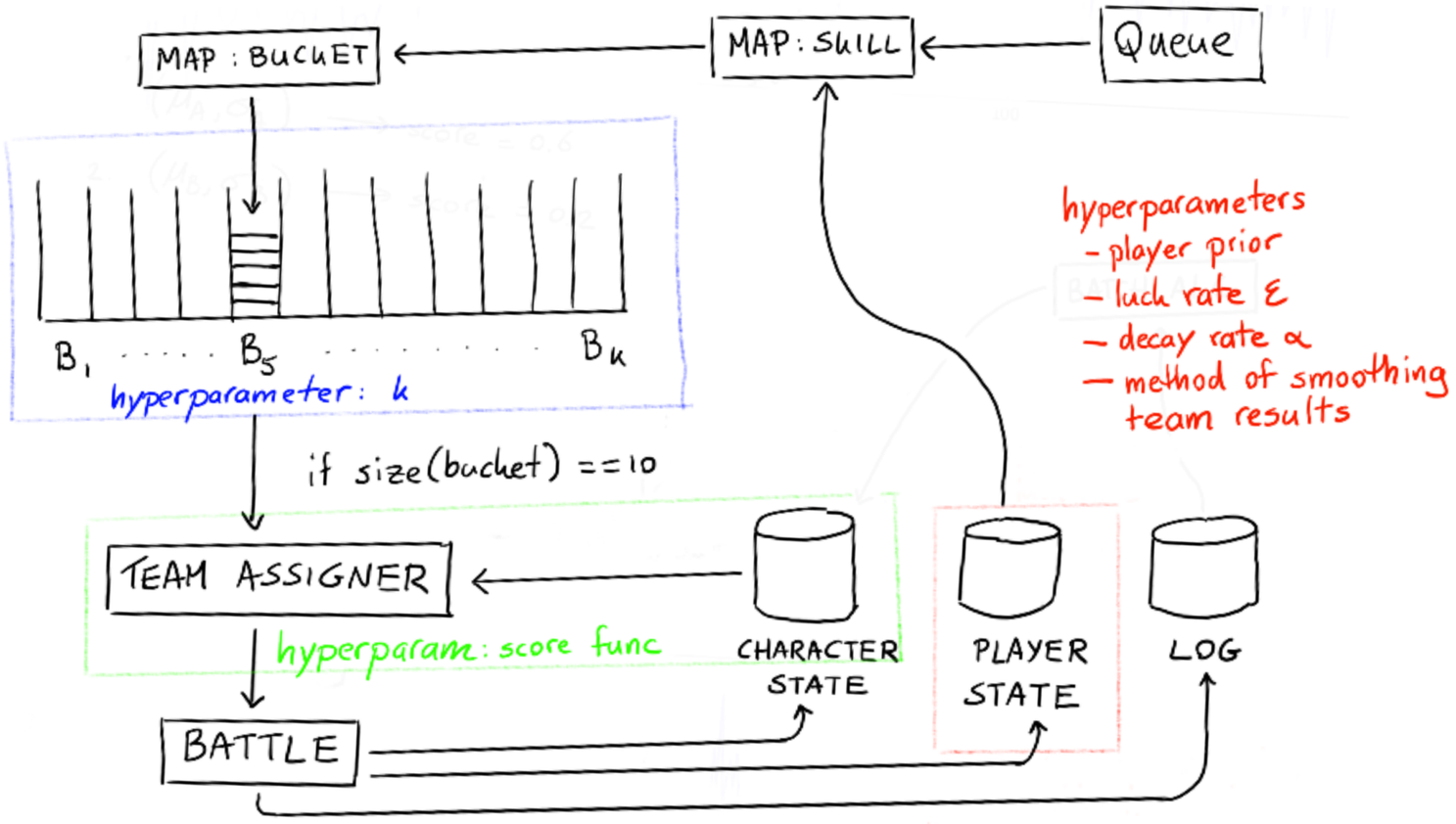
teamfights: updating skill



teamfights: updating skill



total architecture



things to like...

things we like

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

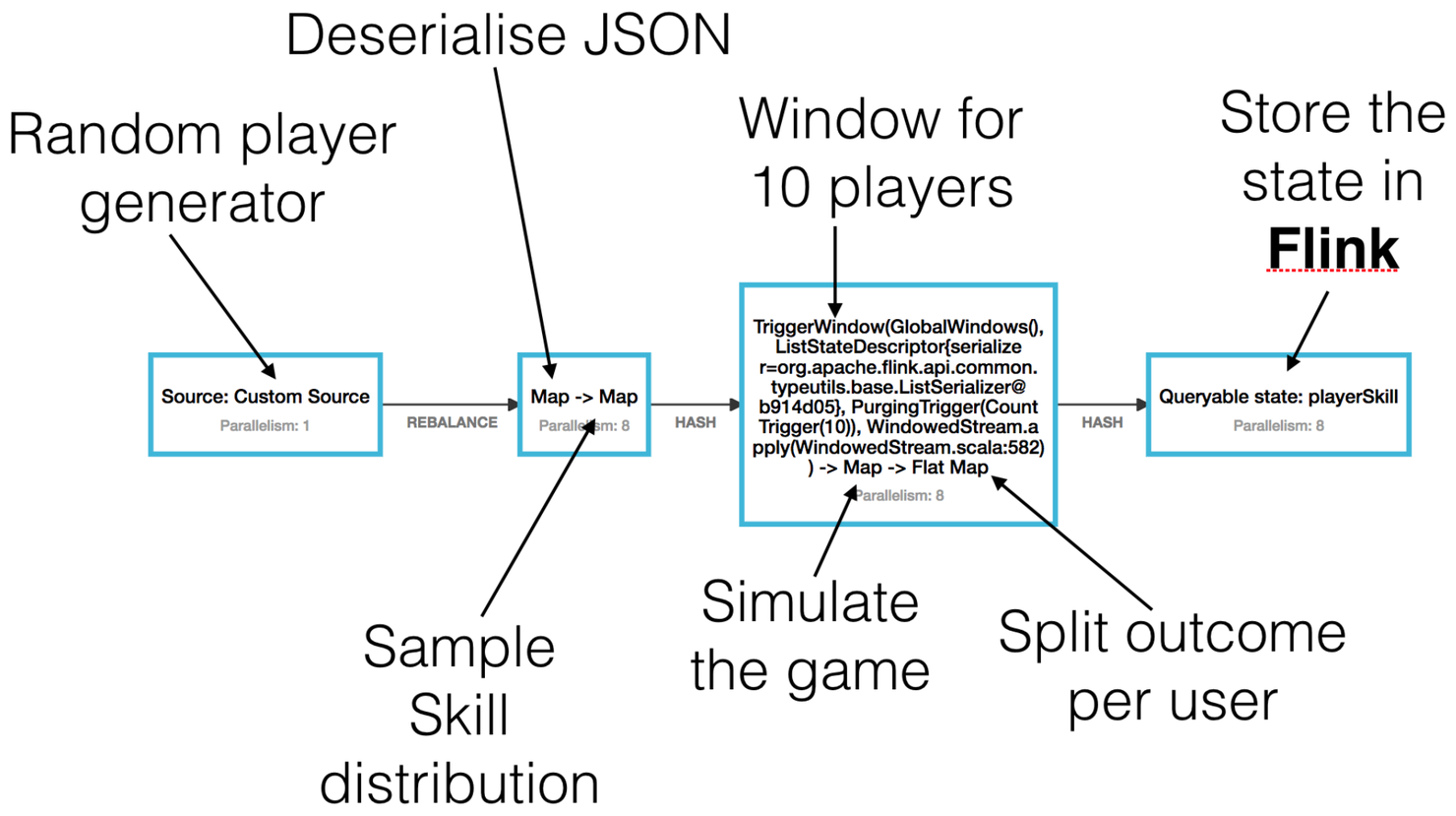
things we like

- We have **separate** functions in our architecture. The team selection step needs no notion of how many queues there are.
- There are a lot of hyperparameters we can tune. This is nice because we know our model to not be perfect. Having degrees of freedom in hyperparameters means we can do tuning. This keeps the algorithm flexible but also requires us to perform some form of grid search first.

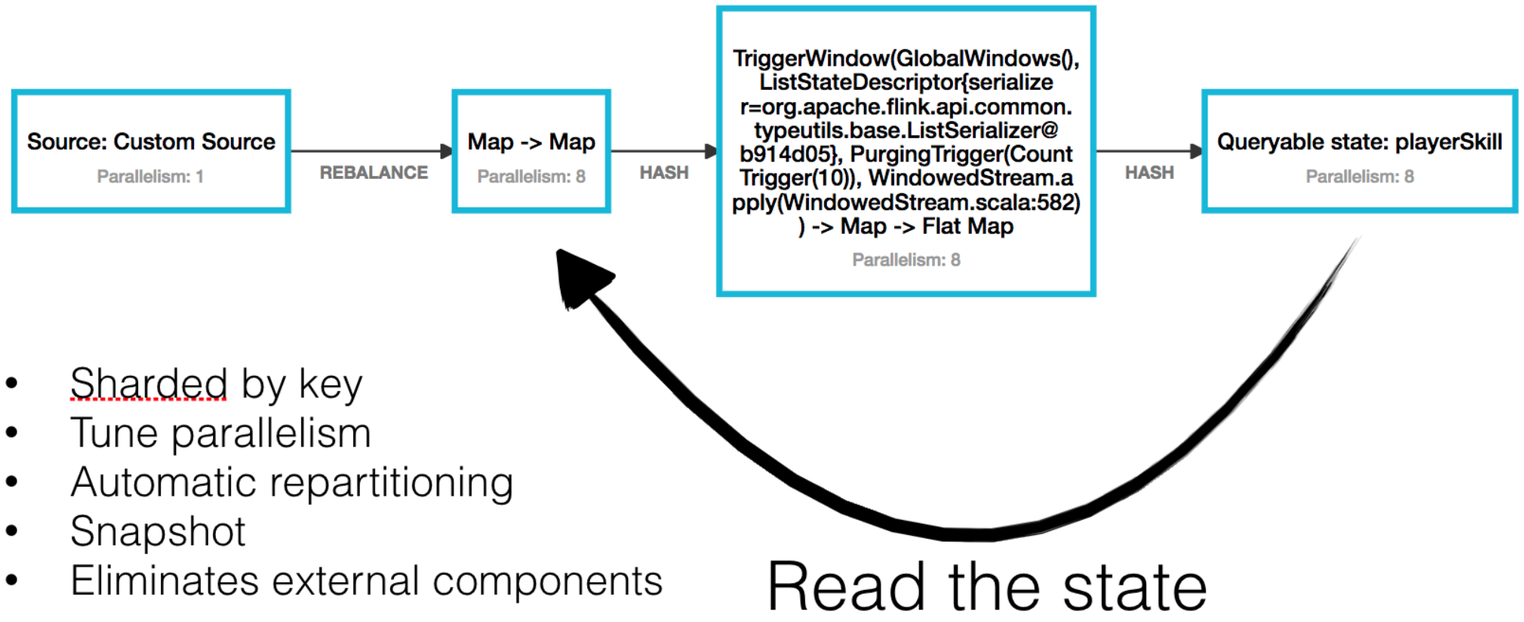
things we like

- We have **separate** functions in our architecture. The team selection step needs no notion of how many queues there are.
- There are a lot of hyperparameters we can tune. This is nice because we know our model to not be perfect. Having degrees of freedom in hyperparameters means we can do tuning. This keeps the algorithm flexible but also requires us to perform some form of grid search first.
- We can scale this, should we get more users playing our game.

let's talk flink



managed state



- Sharded by key
- Tune parallelism
- Automatic repartitioning
- Snapshot
- Eliminates external components

parse the json to a case class

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_._1)
  .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(_._1)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

grab appropriate que bucket as the key

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_._1)
  .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(_._1)
  .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(_._1)
  .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(_._1)
  .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(_._1)
  .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(_._1)
  .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

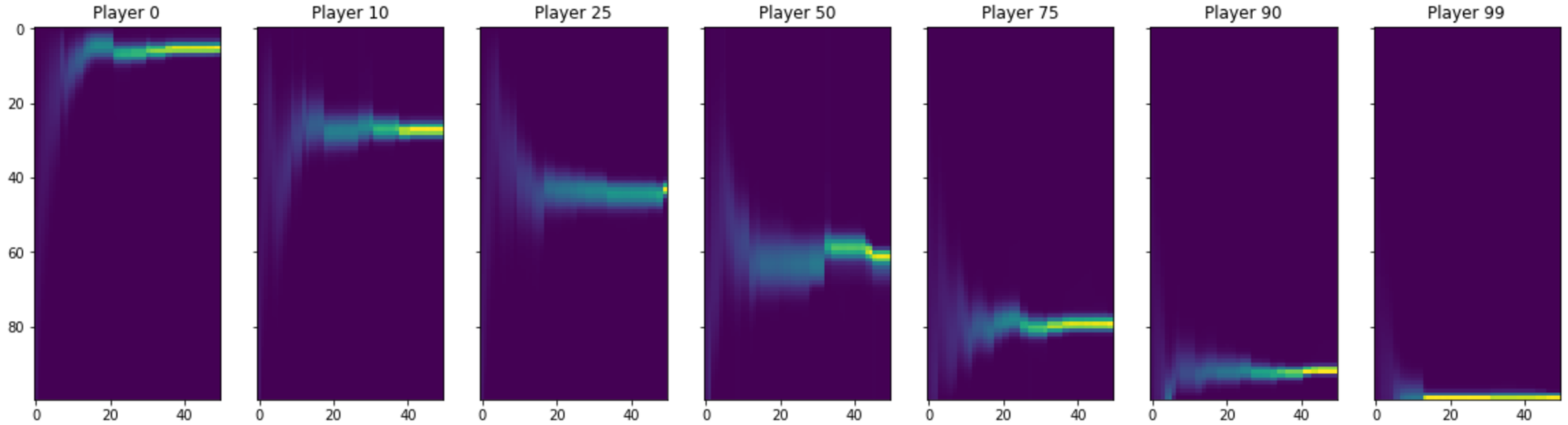
```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_._1)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

code where most things happen

```
stream
  .map(line => JsonUtil.parseJson[Player](line))
  .map(new SamplePlayerSkill)
  .keyBy(_._1)
  .countWindow(LocalConfig.playersPerTeam * 2)
  .apply(new DetermineTeam)
  .map(new PlayGame)
  .flatMap(new ComputeNewPlayerSkill)
  .keyBy(_.player.id)
  .asQueryableState(LocalConfig.keyStateName, reduceStateDescriptor)
```

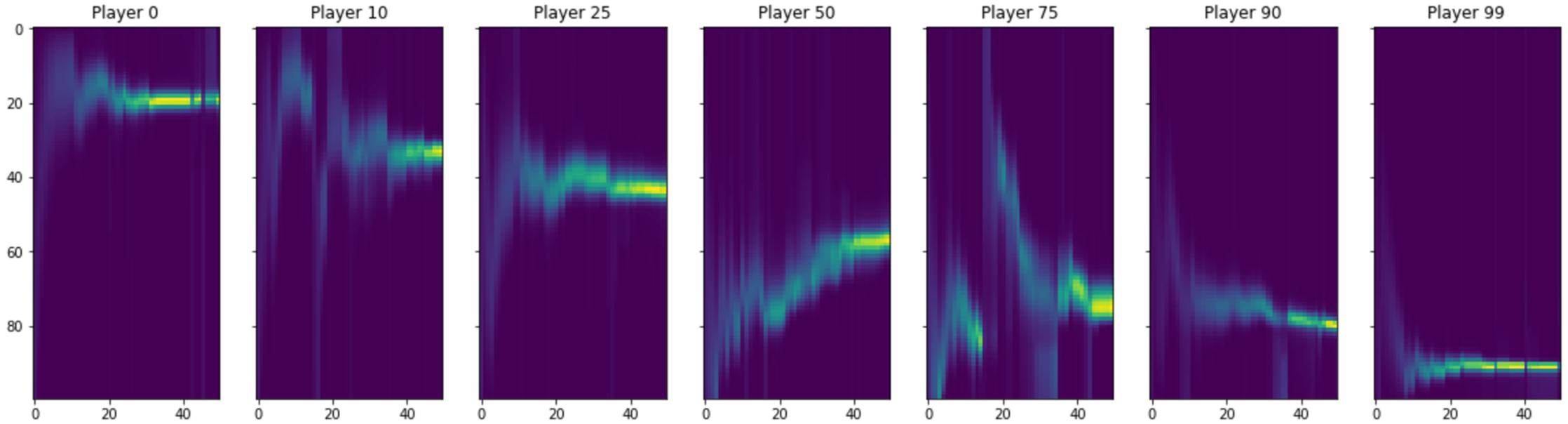
time for demo

main issue: numeric stability of teams



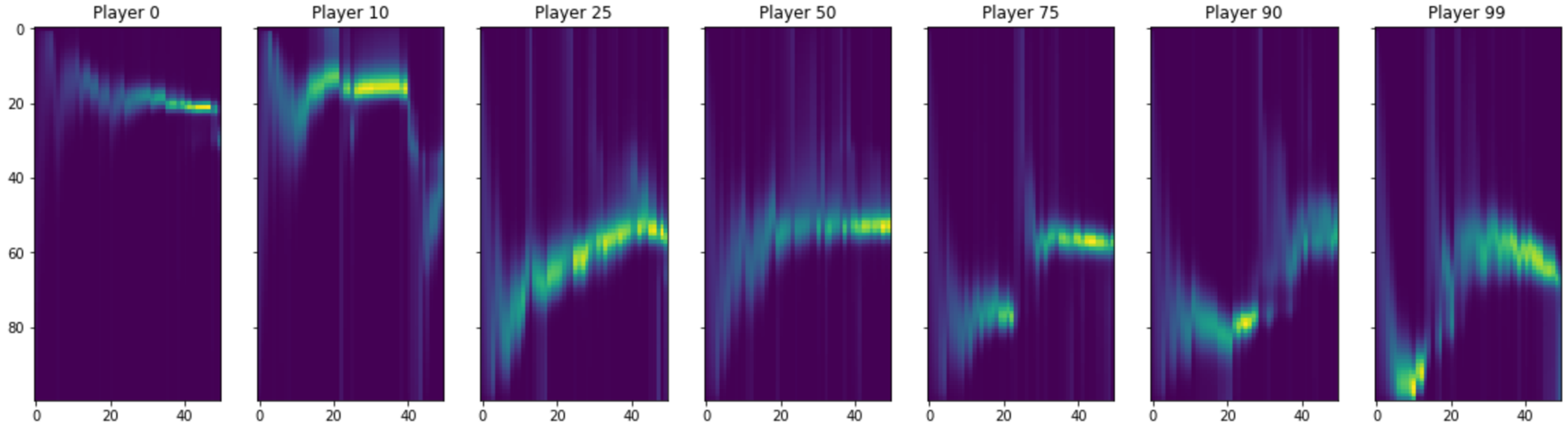
Above is the 1v1 convergence, exactly the same as pokemon.

main issue: numeric stability of teams



This is the 2v2 convergence, more jitter, but converges.

main issue: numeric stability of teams



This is the 5v5 convergence. Ai caramba.

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 elasticsearch + kibana. We need a flexible plotting engine and kibana is a bit limited.

Without flexible visualisation you're going to get stuck understanding your numerical system.

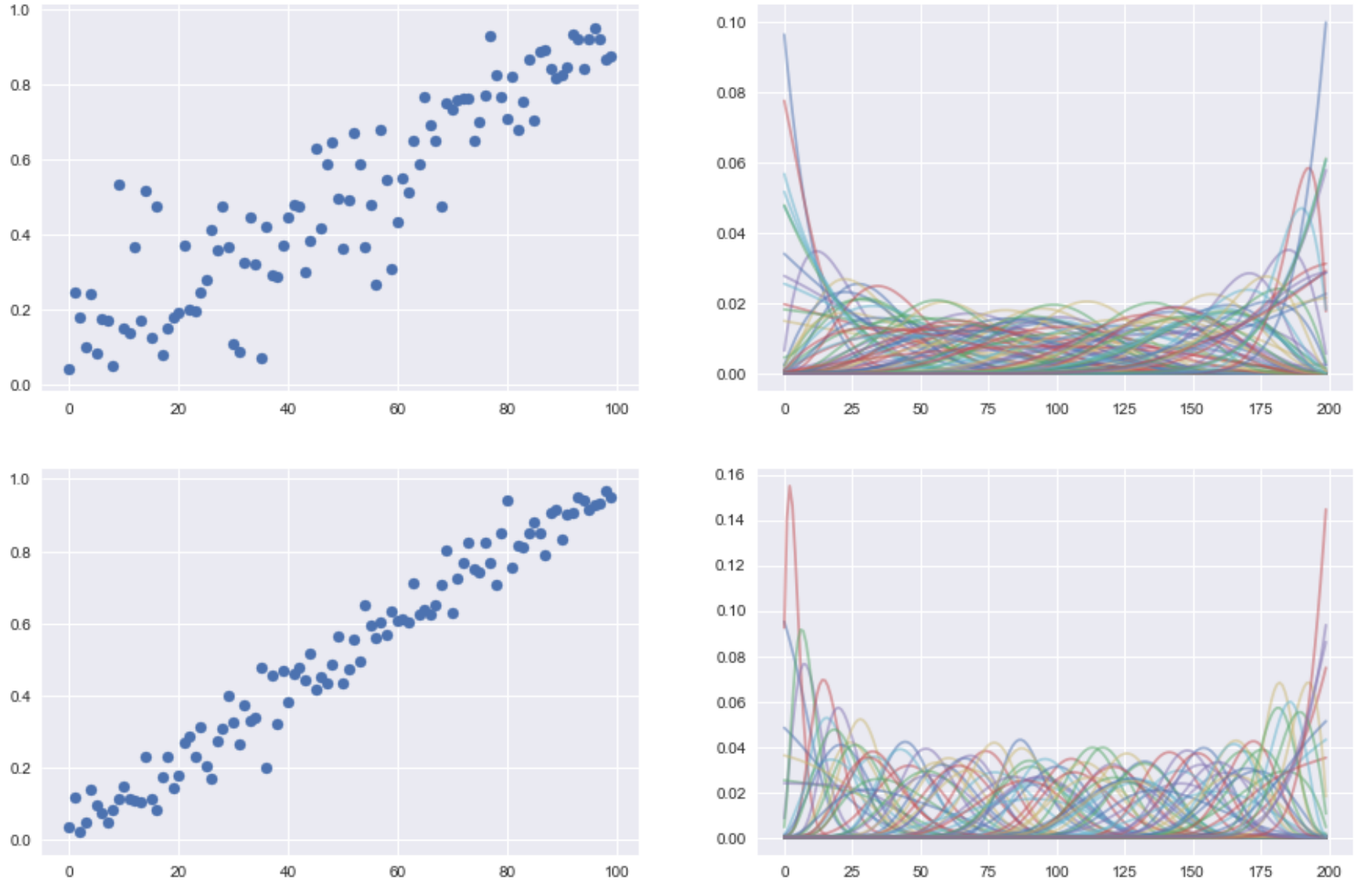
workflow

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

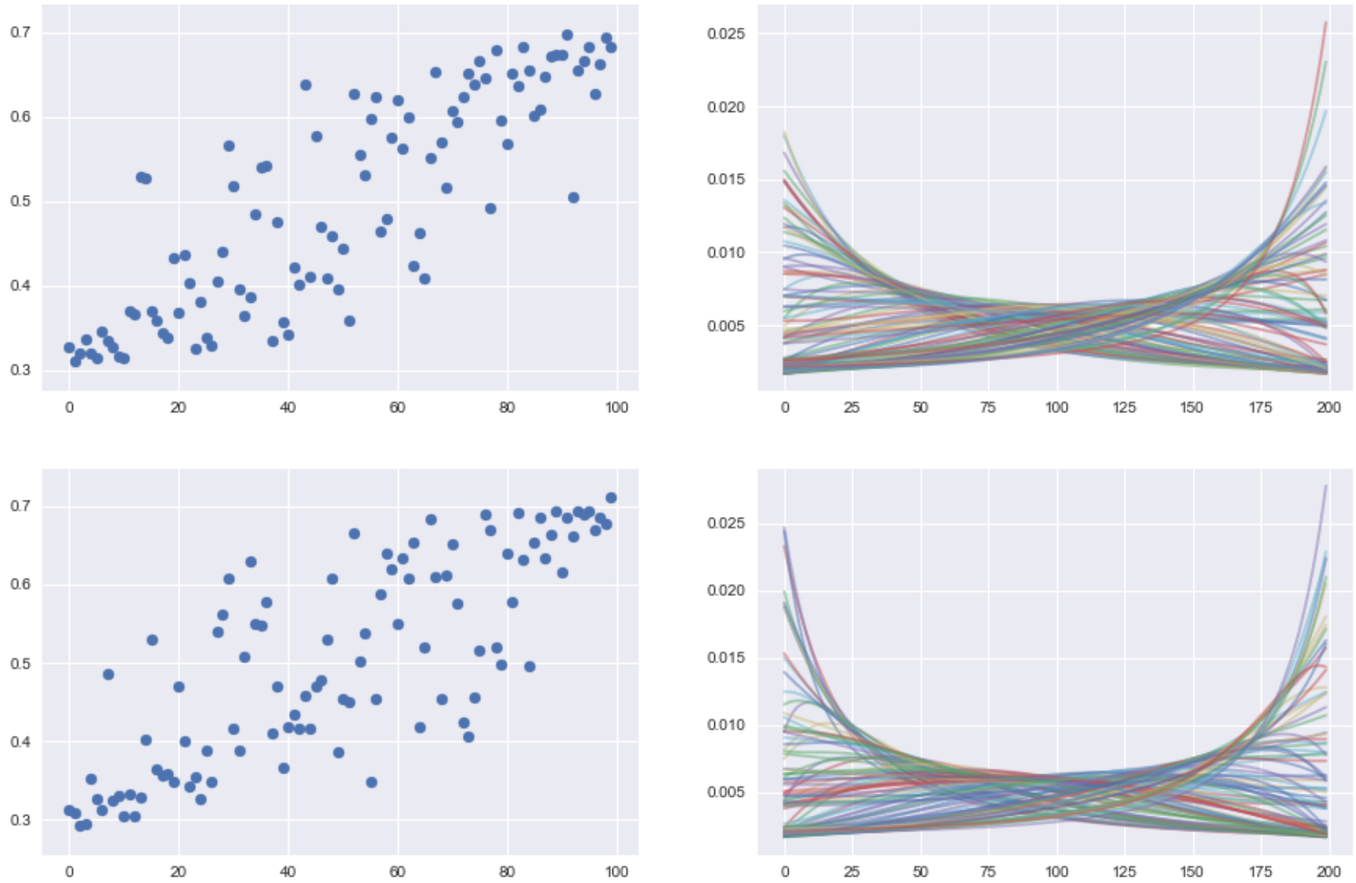
For example. We do some smoothing to ensure that our posterior histogram never has a 0 value. You really want this because otherwise a user can never have a change in their skill level. Turns out that the smoothing parameter has a huge effect.

We can demonstrate this with our player simulations.

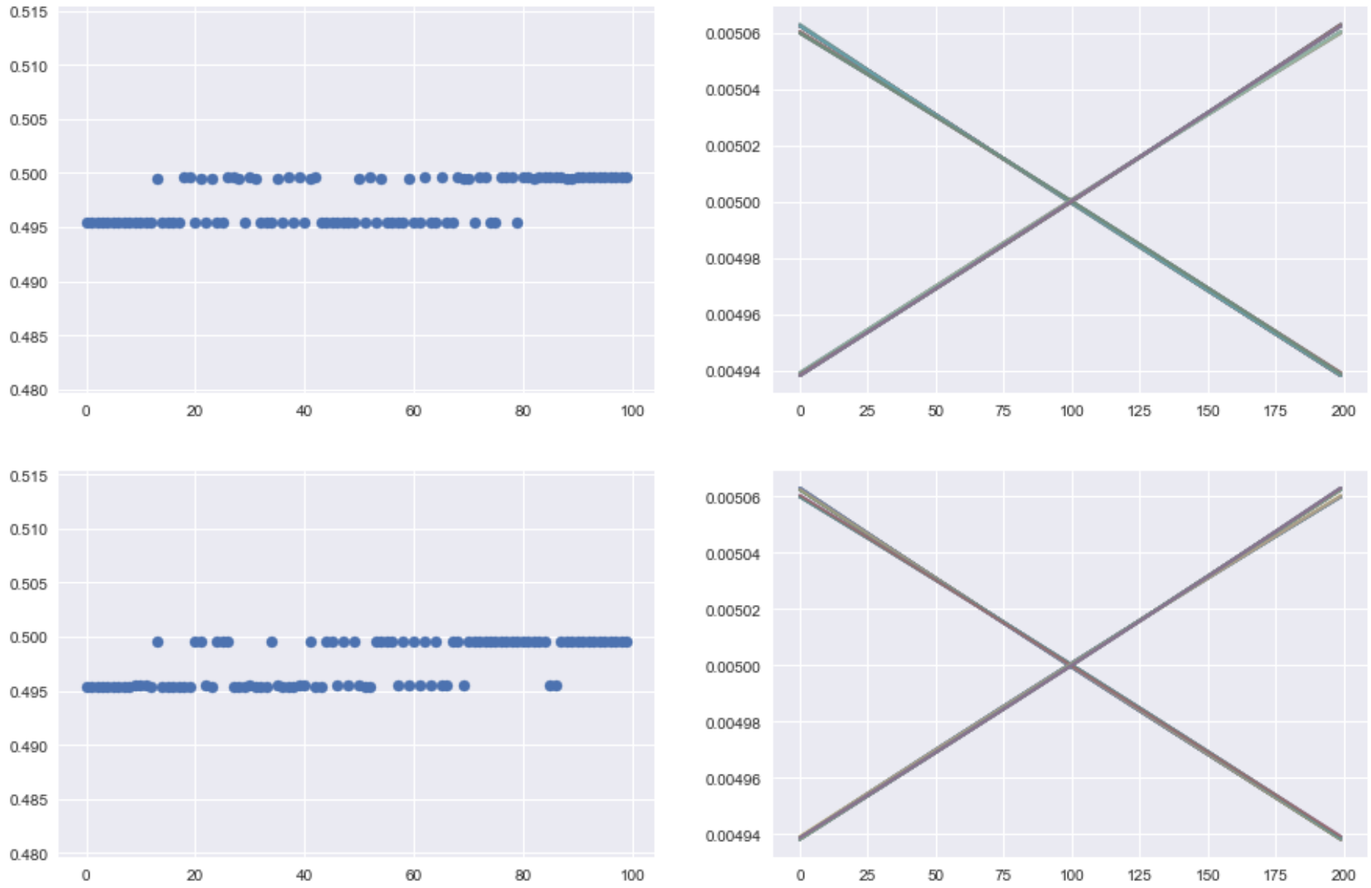
smoothing factor = 0.00001



smoothing factor = 0.0001



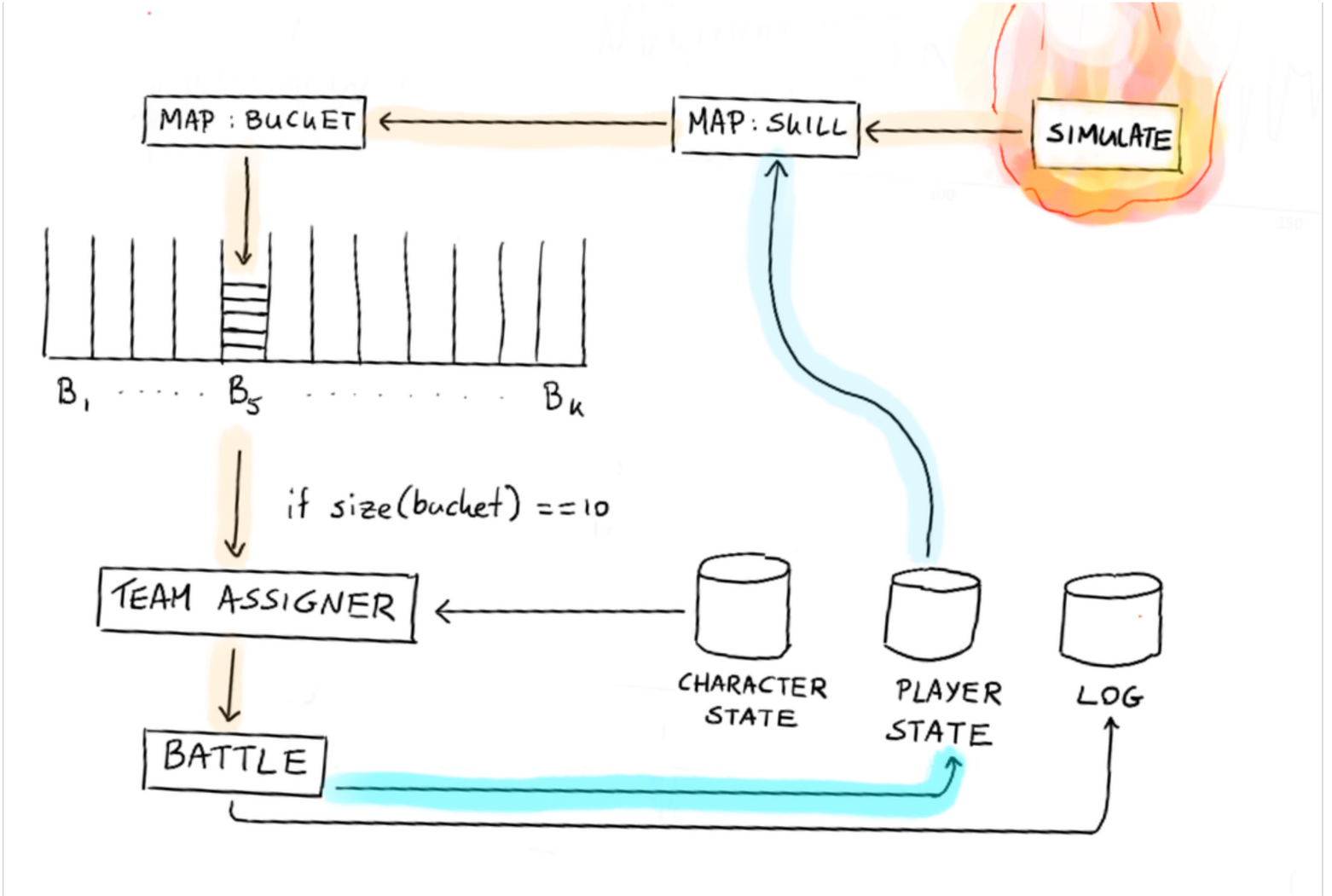
smoothing factor = 0.001



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.


MAP : BUCKET
MAP : SKILL
SIMULATE
B1
B5
Bk
if size(bucket) == 10
TEAM ASSIGNER
CHARACTER STATE
PLAYER STATE
LOG
BATTLE

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.

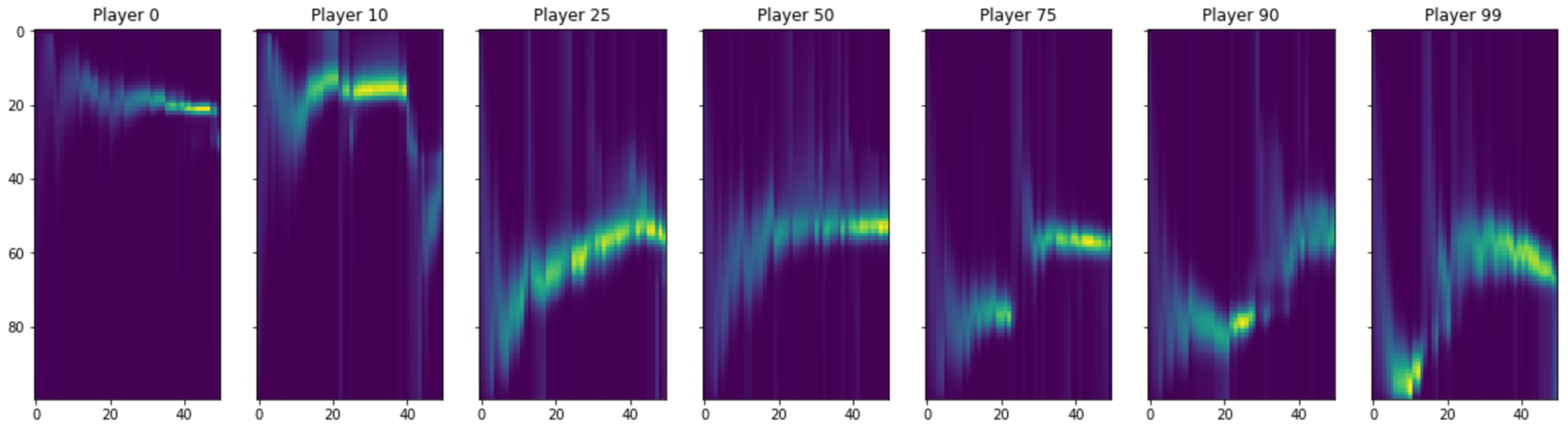
best practices

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.

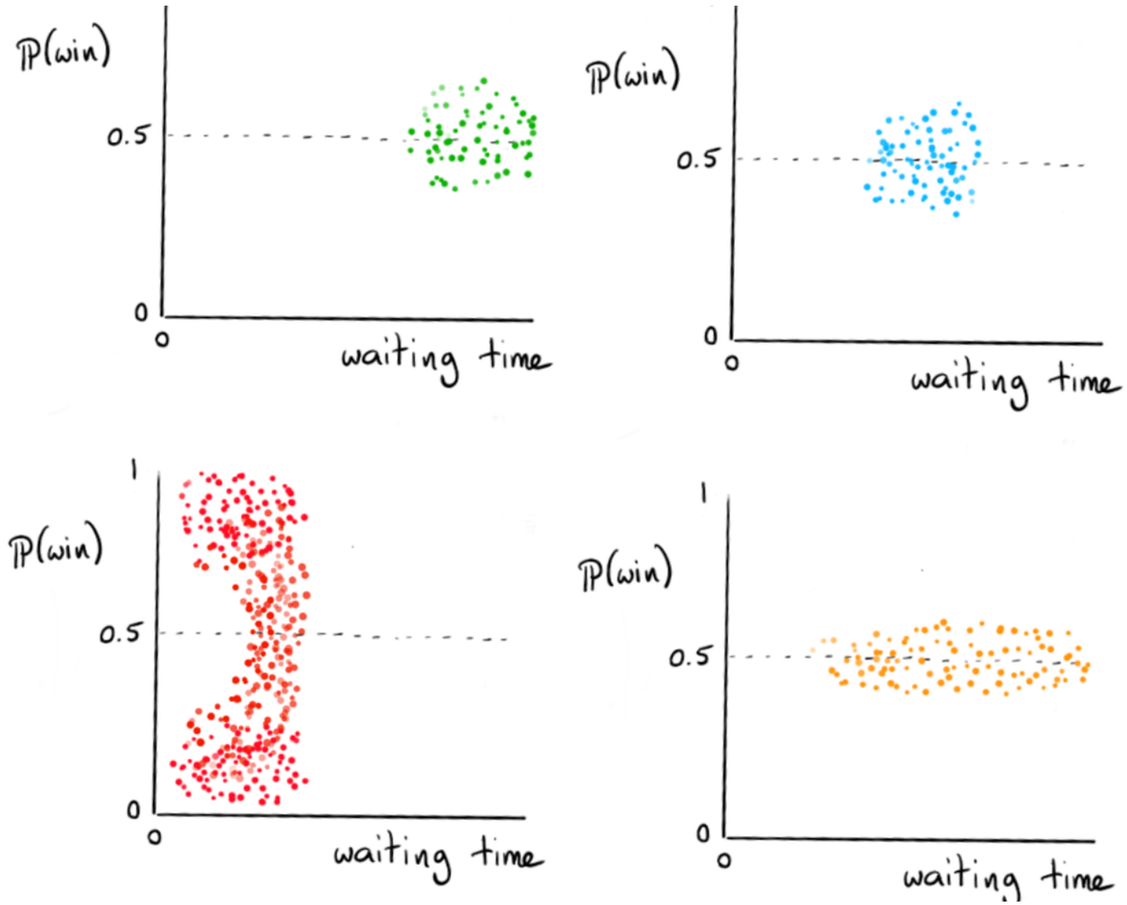
Confirming that 1v1 teams and 2v2 teams worked helped us understand how to improve 5v5 games.

end result



Our final solution, after tweaking hyperparameters by hand showed this behaviour. Good enough for our demonstration purposes because this is a hobby project. Still needs work.

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. You need to be mindful of numerics though.

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

When making such a system, please consider prototyping assumptions and heuristics. Our next iteration will focus on this.

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.