How To Win With Simple, Even Linear, Models

Vincent D. Warmerdam
koaning.io - fishnets88 - GoDataDriven

Why I am Talking about This

You may have heard variants of this quote;

"You should use DeepLearning[tm]"

Why I am Talking about This

You may have heard variants of this quote;

"You should use DeepLearning[tm]"
— Blogs, Reddit, HackerNews and YouTube Stars

Why I am Talking about This

"You should use DeepLearning[tm]"
— Blogs, Reddit, HackerNews and YouTube Stars

It's getting a bit worrysome. It feels like people are focussing more about the tools that they're using than the problem they are solving. I've applied deep learning in production, but I prefer the simpler models.

In this talk. I'll explain why.

Executive Summary

It's not like deep learning isn't amazing, it is, but the hype is distracting people from great ideas.

You may be doing yourself short if you pay too much attention to them. You can often win with simpler models that have properties that are much nicer in situations beyond a jupyter notebook (production).

My goal is to lure you to the domain of boring, old but ultimately beautiful simple models.

Topics of Today

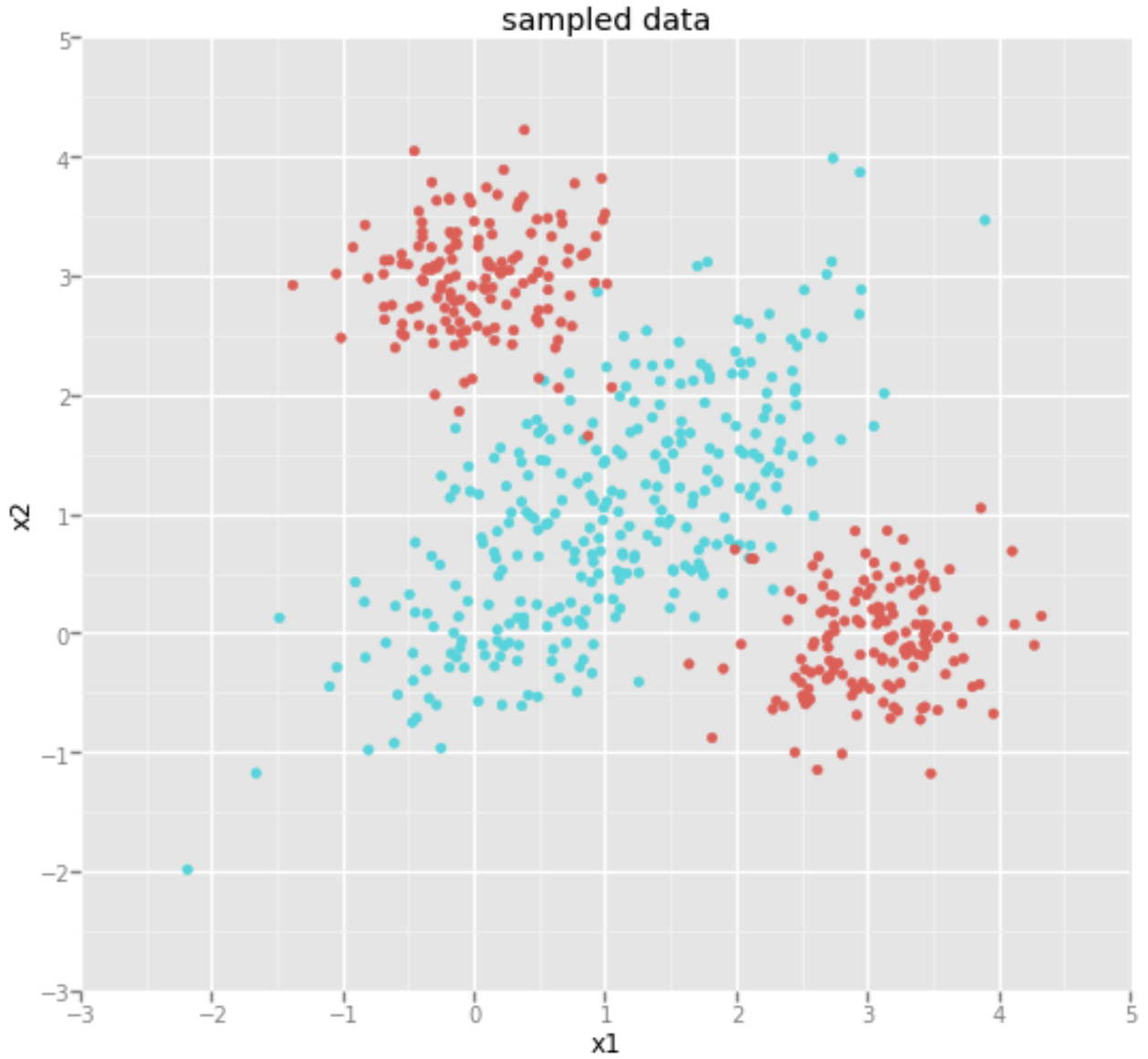
— XOR is a linear problem.
— some simple/great timeseries tricks
— neat feature generation/sklearn tricks
— streaming models
— simple content recommender
— simple video game recommender
— hierarchical domain models

The XOR Problem

One of the main arguments you hear against linear models is that they cannot deal with the XOR problem.

Linear models can only split the data into a single line, the XOR problem is a problem where the dataset is such that you cannot use a single line to split it.

Every textbook on neural networks has this example listed as a reason why NNs beat standard regression.


sampled data
x1
x2
5
4
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
4
5

Code: Logistic Regression

Let's confirm that logistic regression might be worse.

```
> from sklearn.linear_model import LogisticRegression
> y,X = patsy.dmatrices("type ~ x1 + x2", df)
> pred = LogisticRegression().fit(X, y).predict(X)
```

The confusion matrix isn't great.

```
> confusion_matrix(y,pred)
array([[223,  77],
       [118, 182]])
```

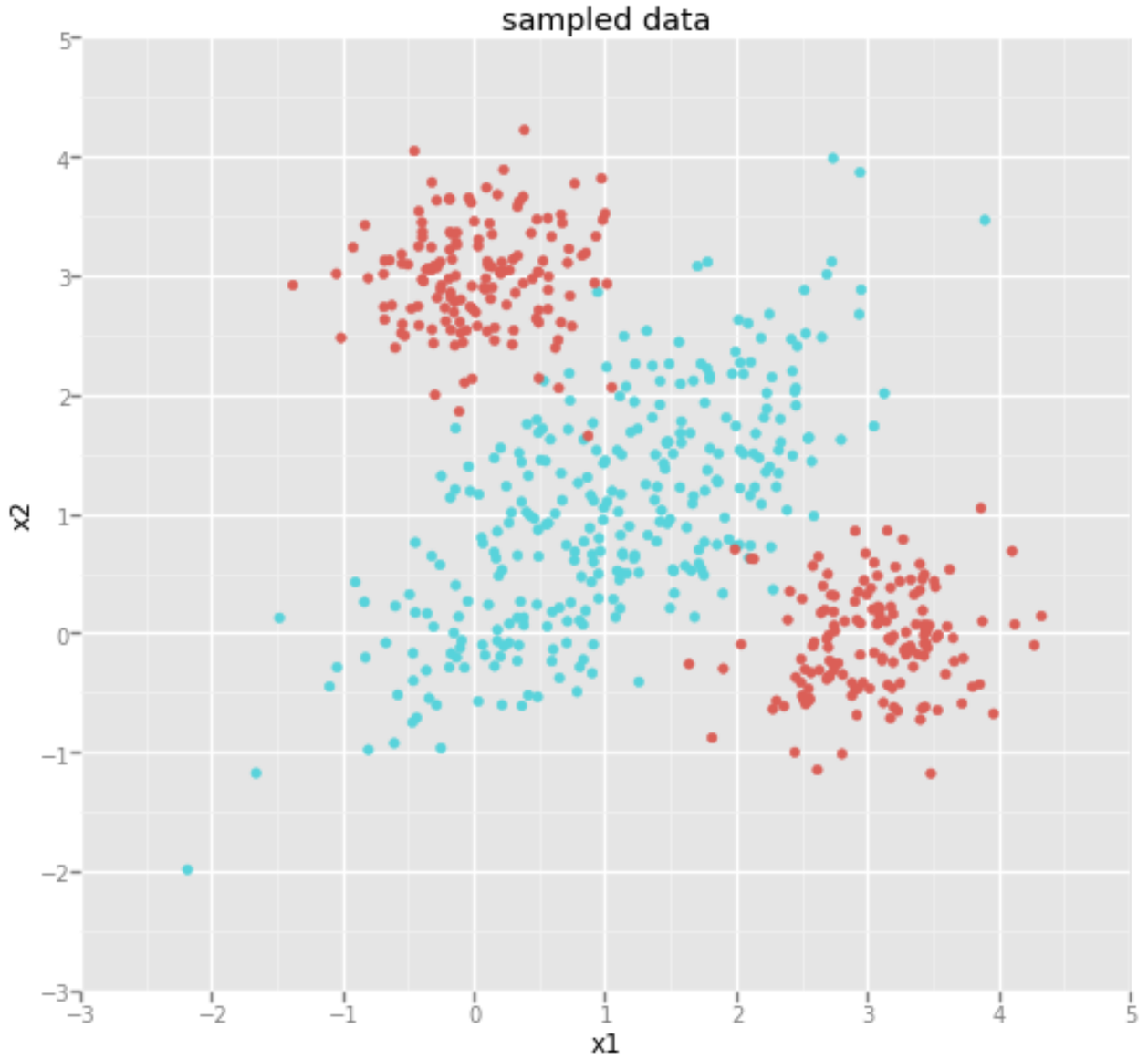
Code: Support Vector Machine

Understanding a SVM is harder than applying it.

```
> from sklearn.svm import SVC
> y,X = patsy.dmatrices("type ~ x1 + x2", df)
> pred = SVC().fit(X,y).predict(X)
```

The confusion matrix is better.

```
> confusion_matrix(y,pred)
array([[294,   6],
       [  2, 298]])
```


sampled data
x1
x2
5
4
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
4
5

Code: Feature Engineering

Feature engineering is as easy to understand/apply.

```
> df['x1x2'] = df['x1'] * df['x2']
> y,X = patsy.dmatrices("type ~ x1 + x2 + x1x2", df)
> pred = LogisticRegression().fit(X,y).predict(X)
```

The confusion matrix is better.

```
> confusion_matrix(y,pred)
array([[290,  10],
       [  3, 297]])
```

Code: Feature Engineering

We just gave a demo of a linear method solving a non-linear problem by adding a single line of code.

Because the model is linear we gain properties:

— more regularisation tricks

— better interpretability

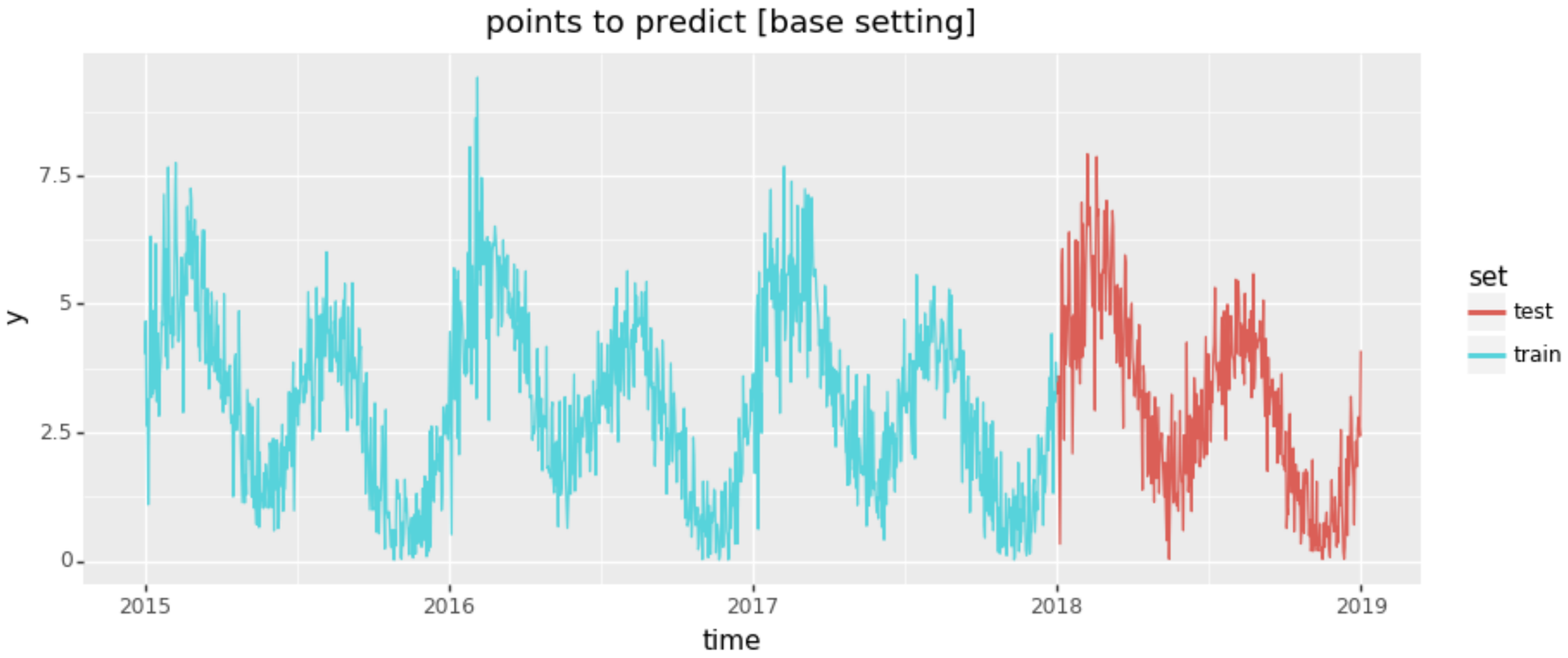
As a consultant, it's easier for me to leave a linear model at a client if they're just starting with modelling.

Next Up

I hope this example spoke to your intuition, but these feature engineering tricks are actually a life saver.

In the next part, we'll discuss feature engineering tricks that have gone to production plenty of times.

Year Ahead Timeseries

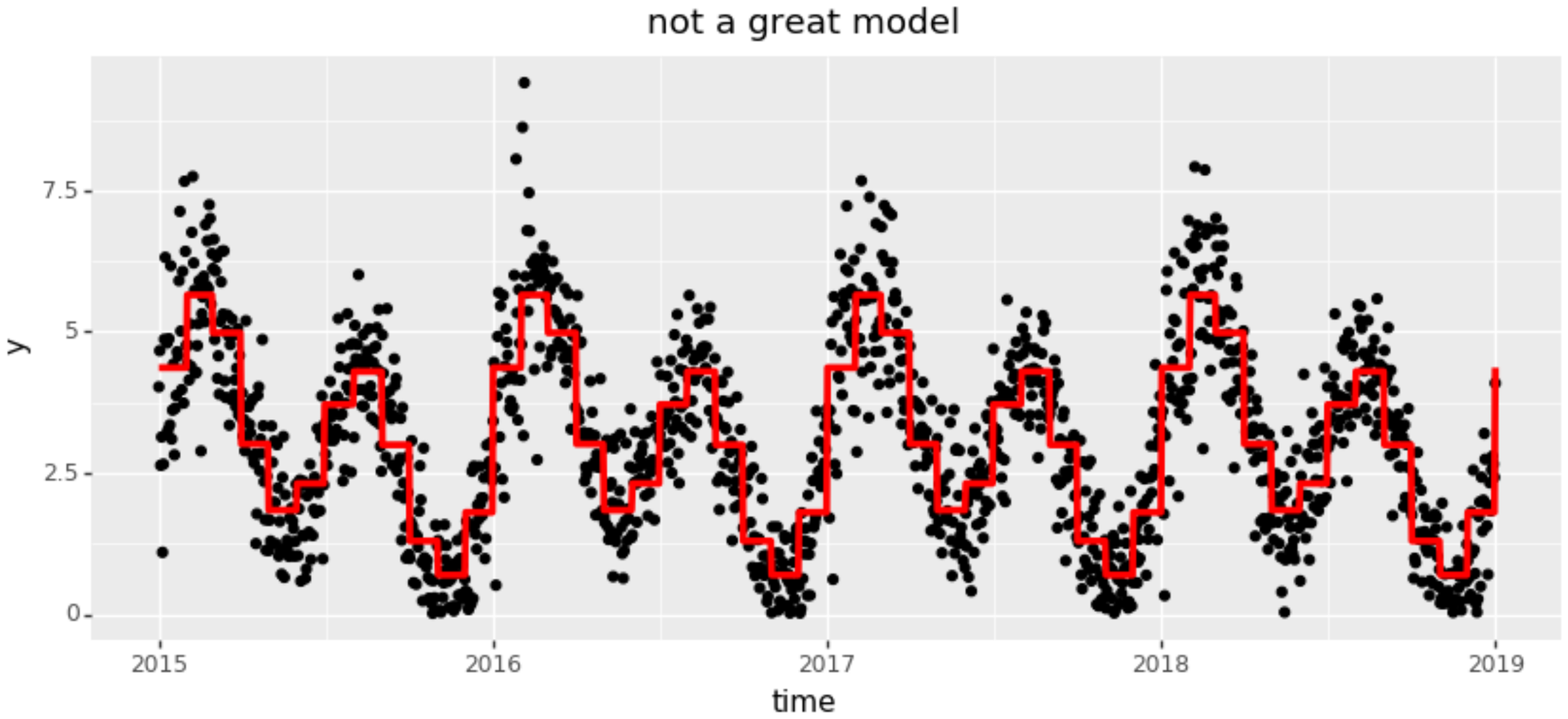


Year Ahead Timeseries

Things about this dataset;

— it is daily data and we have 4 years of data
— it is a year ahead prediction
— it is a faily common planning task
— feature engineering will play a large role

Dummy Variables Alternative



Year Ahead Timeseries

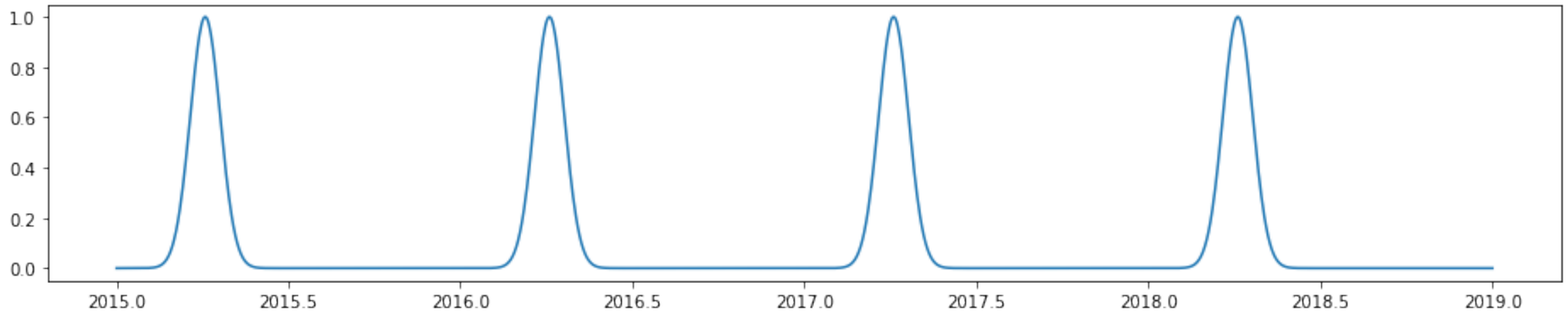
Instead of adding dummy variables, we could add smart variables. I'll call them smarties with pun intended. We want to have something that is more than a mere average and something where you can learn from **2018-04-30** to predict **2018-05-01**.

Just like with the XOR problem, the goal is to add simple features in order to deal with the non-linearity.

Radial Basis Functions

There's these functions we could apply.

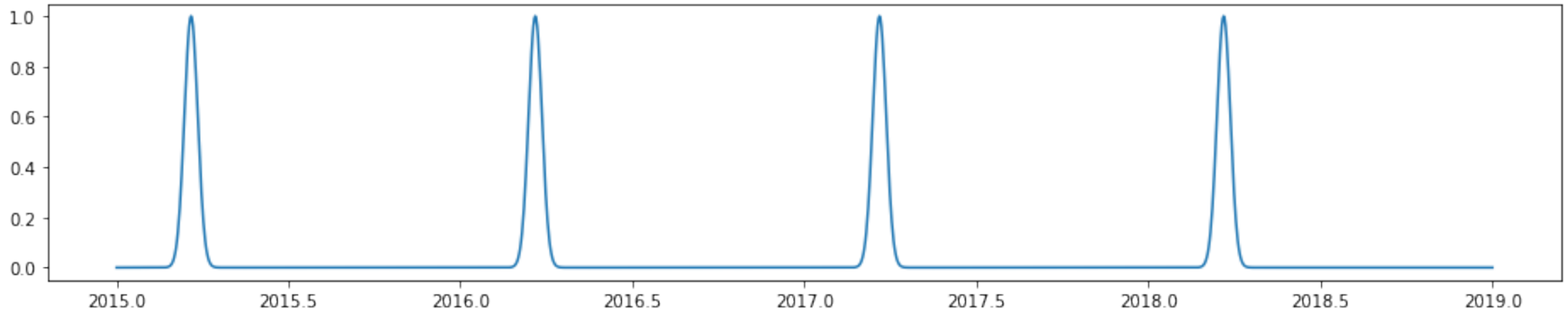
$$\phi(x_i) = \exp\left[-\frac{1}{2\alpha}(x - m_i)^2 \right] \bmod \forall \text{ year}$$



Radial Basis Functions

You can change α to change the shape.

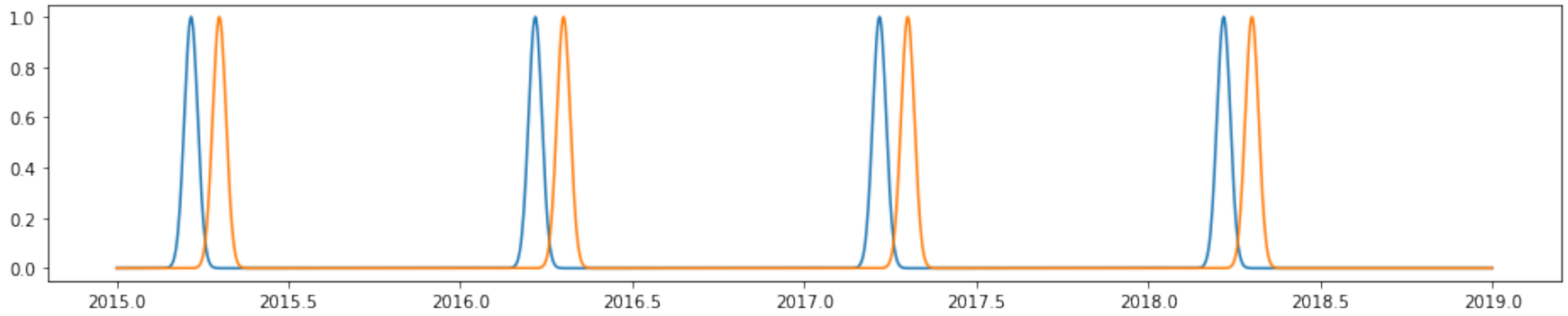
$$\phi(x_i) = \exp\left[-\frac{1}{2\alpha}(x - m_i)^2 \right] \bmod \forall \text{ year}$$



Radial Basis Functions

There's these functions we could apply.

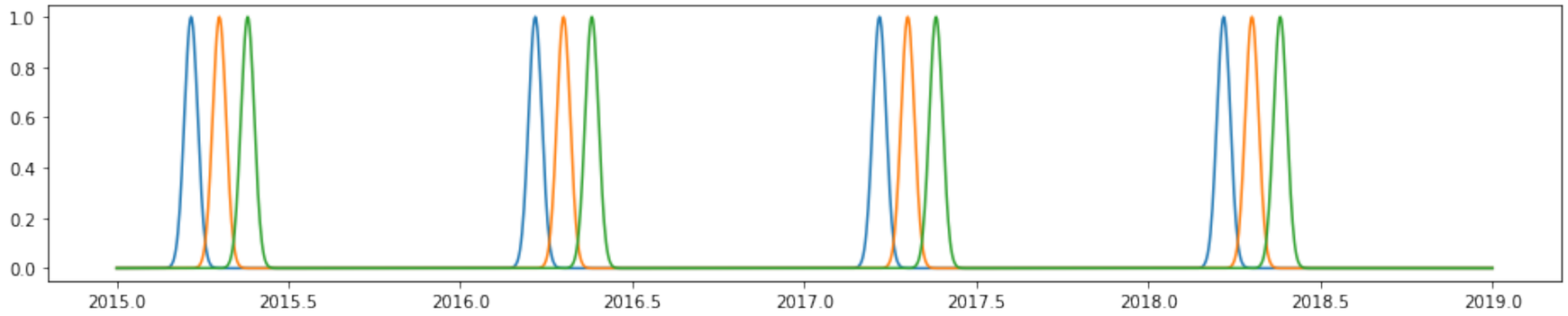
$$\phi(x_i) = \exp\left[-\frac{1}{2\alpha}(x - m_i)^2 \right] \bmod \forall \text{ year}$$



Radial Basis Functions

There's these functions we could apply.

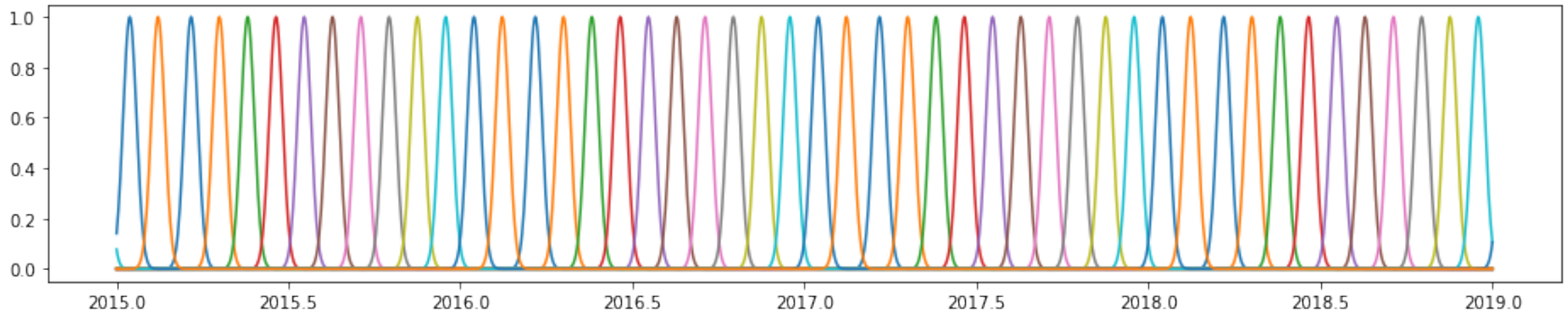
$$\phi(x_i) = \exp\left[-\frac{1}{2\alpha}(x - m_i)^2 \right] \bmod \forall \text{ year}$$



Radial Basis Functions

There's these functions we could apply.

$$\phi(x_i) = \exp\left[-\frac{1}{2\alpha}(x - m_i)^2 \right] \bmod \forall \text{ year}$$



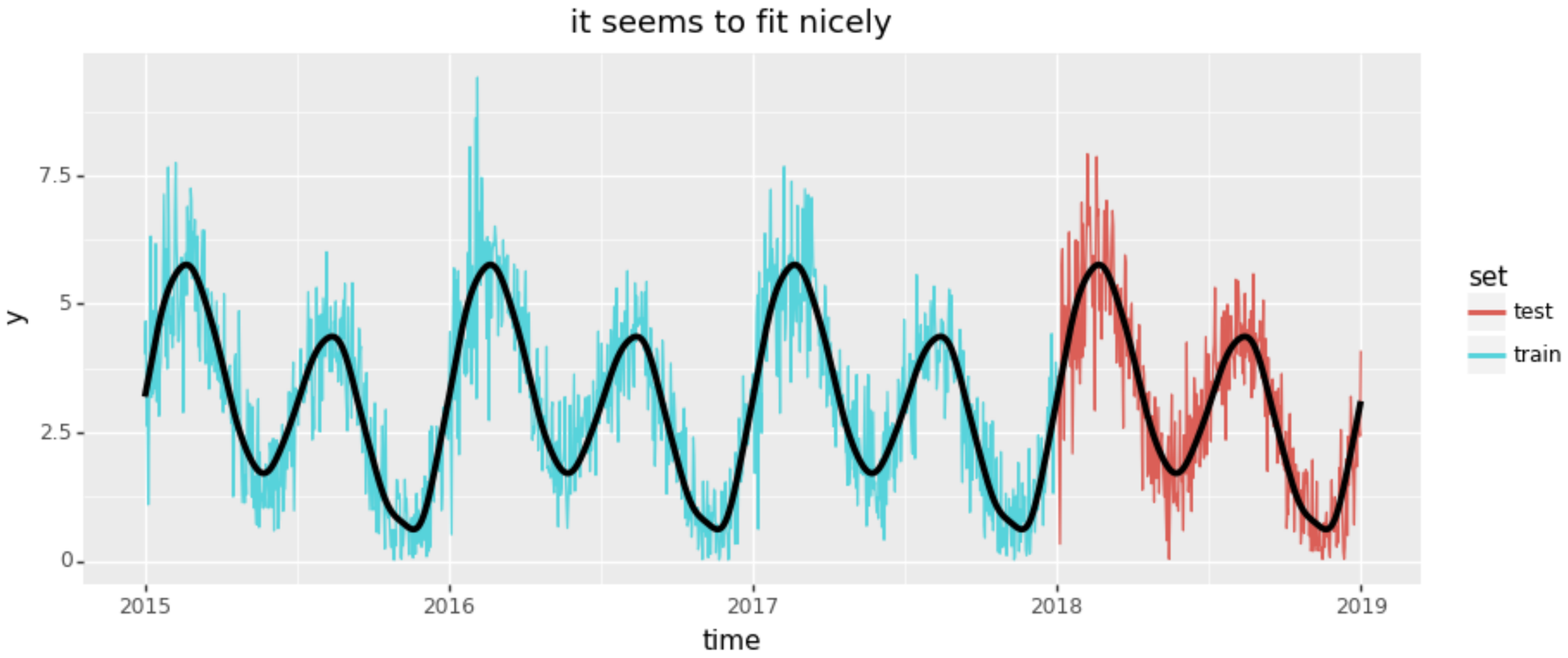
Radial Basis Functions

If you create these features, the dataframe will look like:

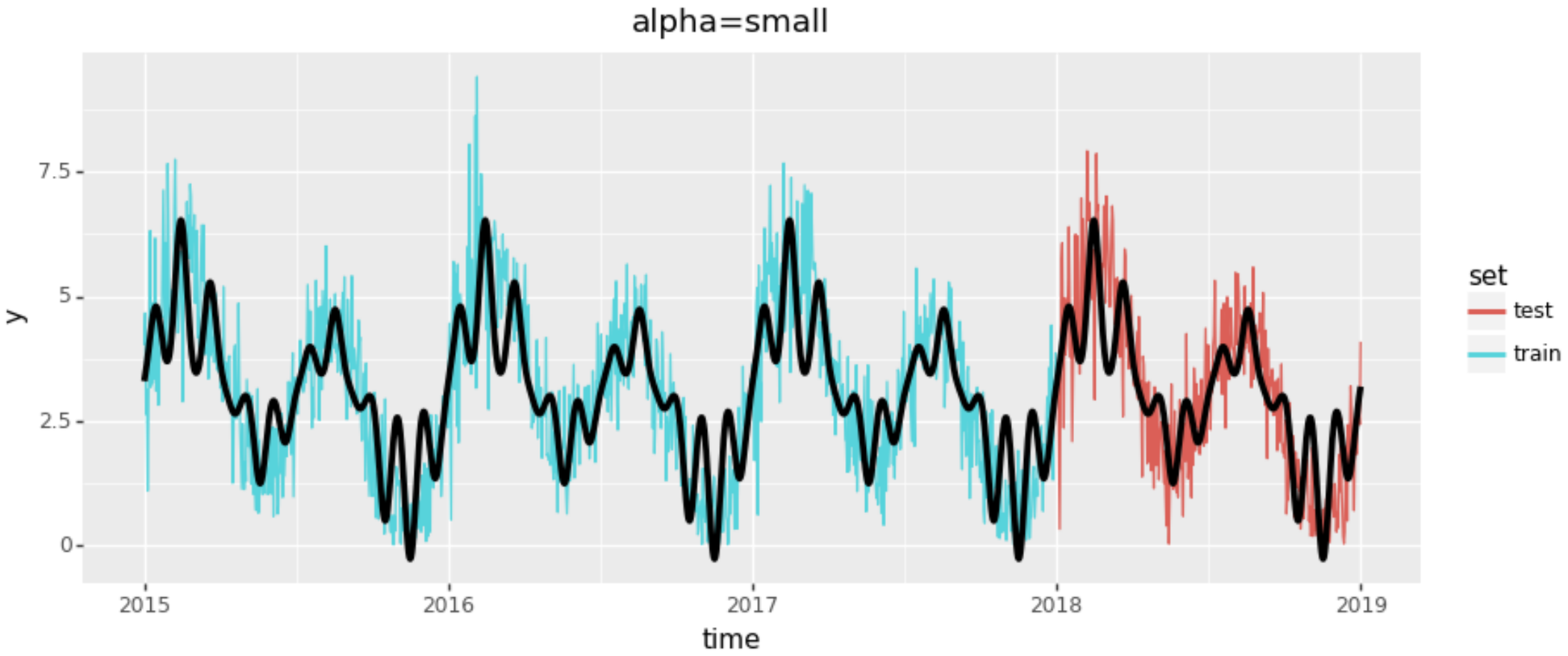
year	day	y	x01	...	x12
2016	212	4.375096	4.448581e-01	...	2.721434e-43
2016	213	4.798139	4.855369e-01	...	7.305730e-43
2016	214	3.130771	5.272924e-01	...	1.951452e-42
2016	215	5.115325	5.697828e-01	...	5.186577e-42
2016	216	4.527225	6.126264e-01	...	1.371615e-41
2016	217	3.975743	6.554063e-01	...	3.609210e-41

Put in another way, this is a feature engineering trick.

Radial Basis Functions



Effect of Hyperparam α



Since the model is linear we can use maths to figure out which columns might need to go away. Maybe like this:

```
Residuals:
    Min       1Q   Median       3Q      Max
-0.68951 -0.14282  0.00027  0.14625  0.61059

Coefficients:
    Estimate Std. Error t value Pr(>|t|)
x1   2.78156    0.05104  54.500  < 2e-16 ***
x2  -0.37352    0.10721  -3.484 0.000512 ***
x3   1.40336    0.15269   9.191  < 2e-16 ***
x4  -1.84918    0.18472 -10.011  < 2e-16 ***
x5  -0.60581    0.20559  -2.947 0.003275 **
x6  -1.58640    0.21831  -7.267 6.66e-13 ***
x7  -0.15144    0.22515  -0.673 0.501316
x8   0.11158    0.22721   0.491 0.623450
x9  -0.22219    0.22410  -0.991 0.321660
x10  0.42819    0.21271   2.013 0.044332 *
x11 -0.87599    0.18411  -4.758 2.19e-06 ***
x12  1.73673    0.11847  14.659  < 2e-16 ***
```

Since the model is linear we can use maths to figure out which columns might need to go away.

There are alternatives to feature selection;

— sklearn has a bunch of cool ones
— bayesian regression and critisism could give even more insight

Nice Properties

— This feature allows for predictions that are many timesteps into the future.

— By approaching this timeseries task with these features you can build it on anything that has linear regression (H20/SparkML/`R::lm`). If the system does not offer a timeseries module, you can still manage.

— All the magic occurs in the feature engineering step, so little mental investment is needed to learn a new domain of modelling.

Nice Properties

— You can mold this approach. You can make RBF-features for time of day and for day of year. You can create more than 12 features per year or less if your usecase allows for it.

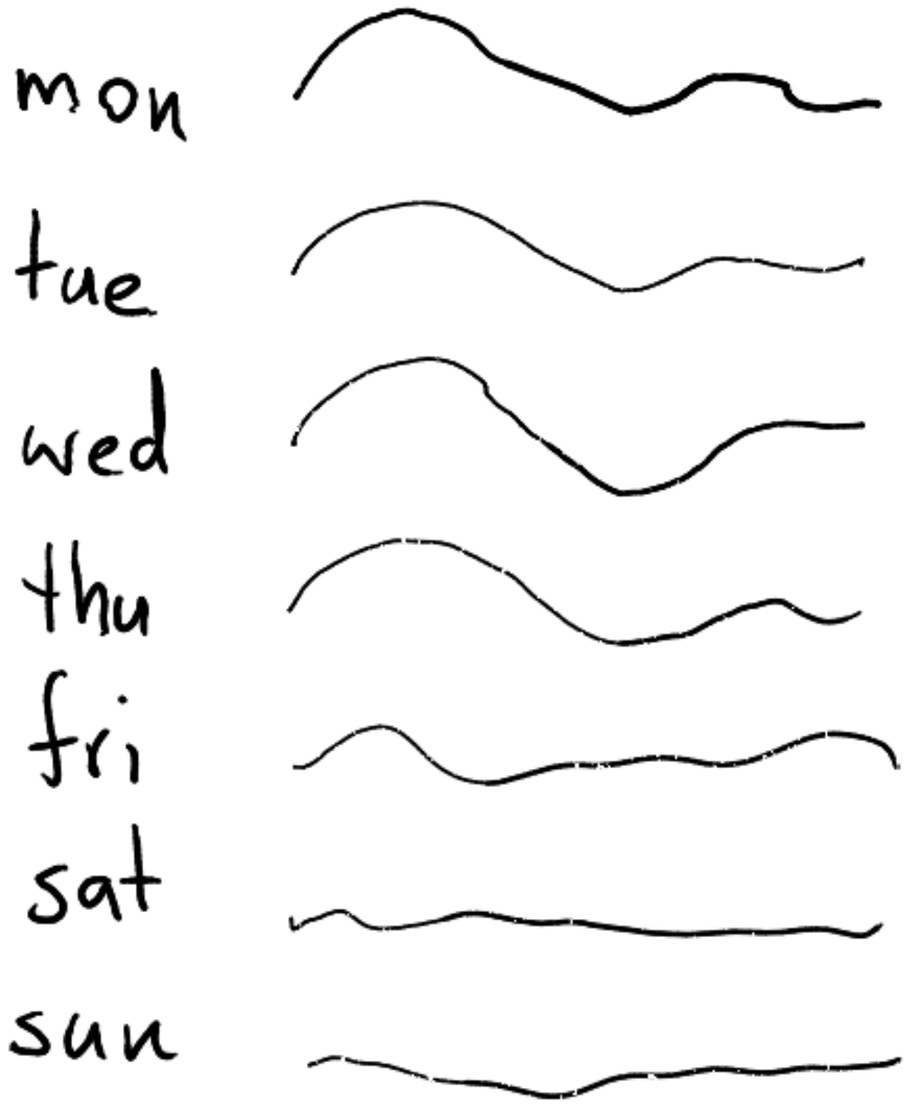
— Since everything is linear it is straightforward to use a bayesian estimator for the features. If certain months have less datapoints you can quantify the uncertainty.

— All of these steps are very flexible and very interpretable.

Small War Story

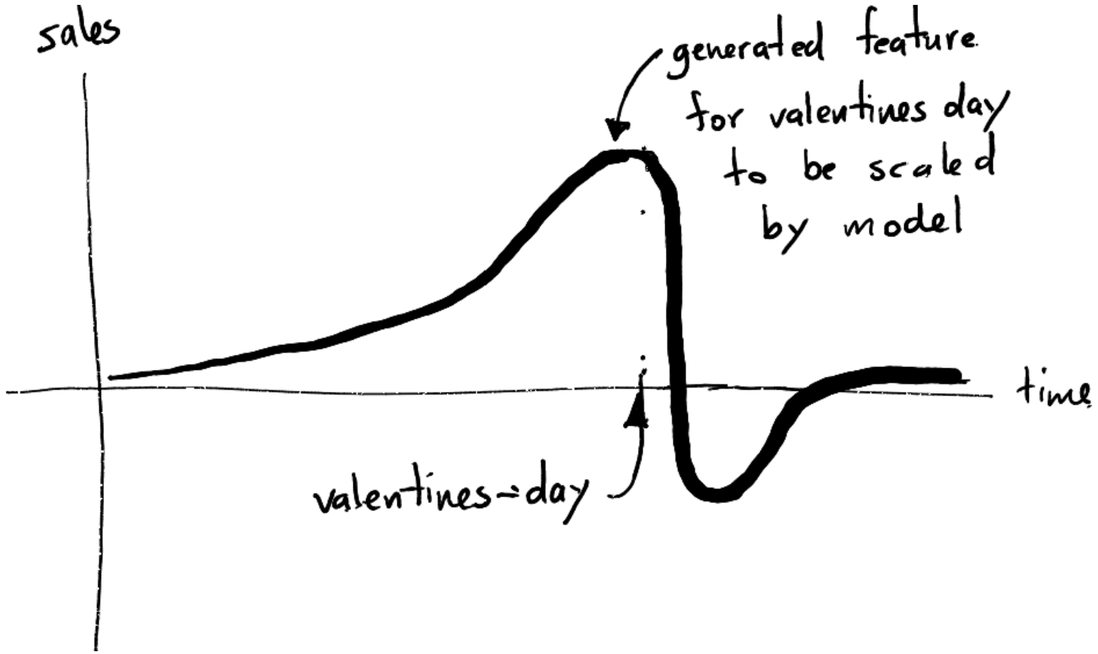
I had to make predictions for a client that was in retail, the predictions needed to be a year ahead.

One simple observation was that we might be able to improve the model by calculating these RBF features for each day of the week seperately. It made sense because the seasonality was different for day of the week.



Small War Story

An additional trick was to also generate features for holidays.



Features with Benefits!

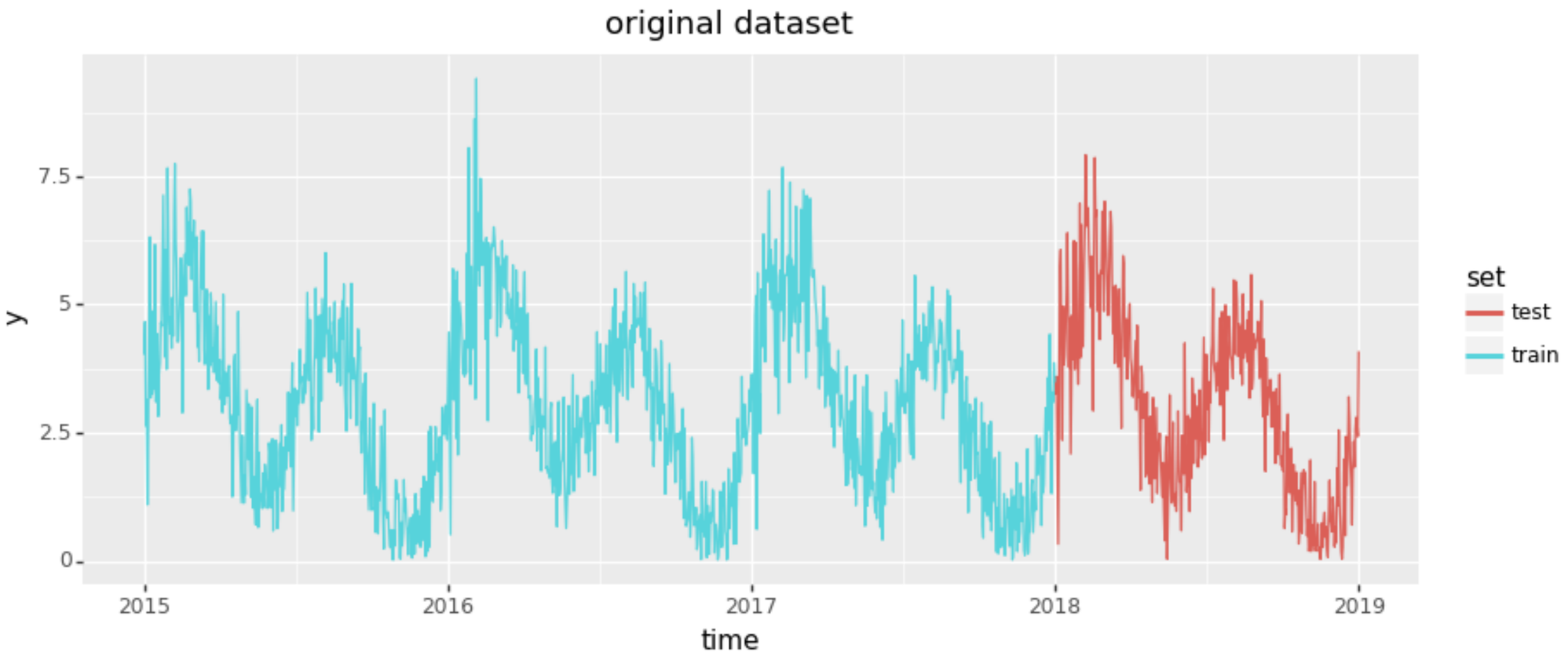
Like before, a very simple feature engineering trick made our linear model got a whole lot more powerful.

One could apply this trick for non-linear models too but you loose a bit of control over what happens to the features. You may also not realize this feature-trick if you're assuming the model solves everything.

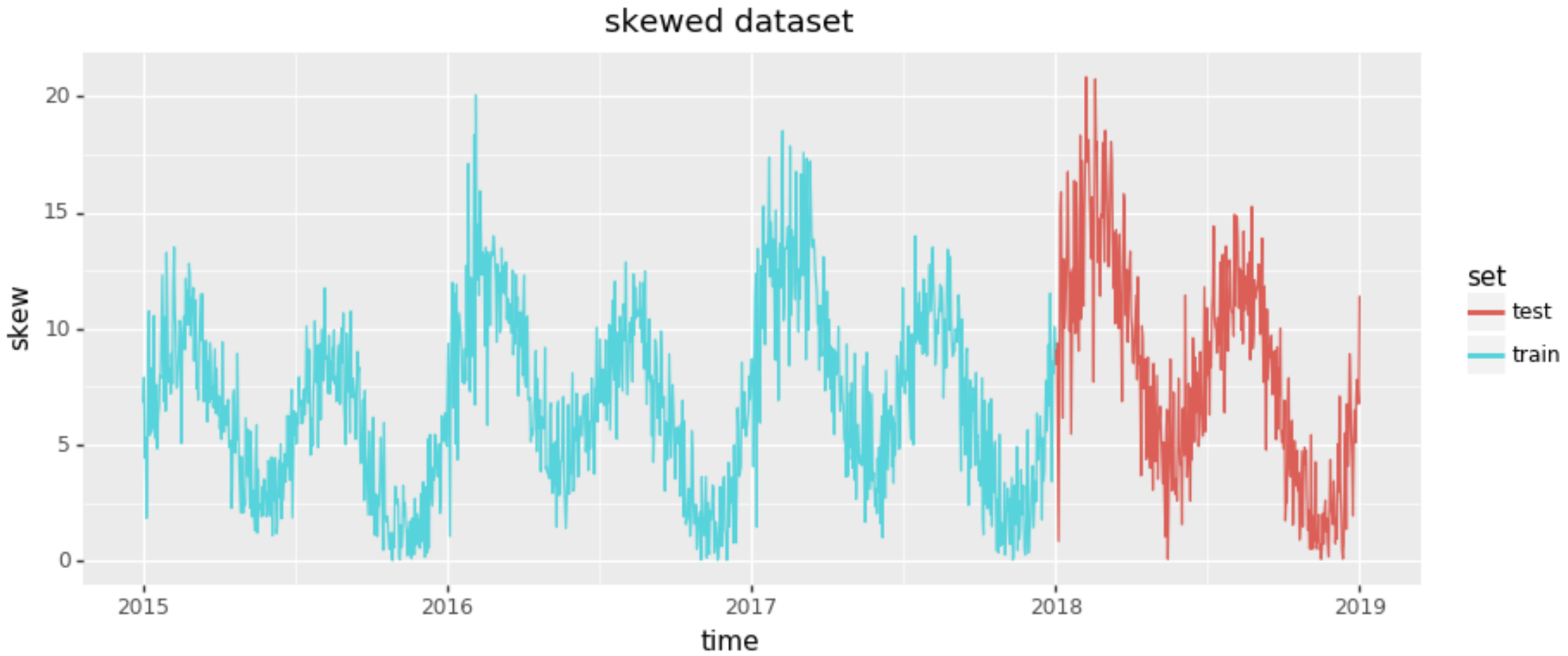
Thing to note: GBM, RF, DNN models won't automagically create such features for you.

Let's consider a variant of the same problem now.

Change of Problem: Before



Change of Problem: After



Change of Problem

This dataset resembles something common in time-series; datapoints that are more recent should have more influence than the points that are far away. It would be nice to perform a regression that is able to acknowledge this.

We can force a regression to take this into account in **sklearn**. But let's first discuss the theory behind it a little bit. I'll try to keep the maths light.

Weighted Linear Regression

Normally you want to minimize an error:

$$\mathbf{err}(\mathbf{w}) = \sum_i (y_i - \mathbf{w}x_i)^2$$

Weighted Linear Regression

Normally you want to minimize an error:

$$\mathbf{err}(\mathbf{w}) = \sum_i (y_i - \mathbf{w}x_i)^2$$

Or:

$$\mathbf{err}(\mathbf{w}) = \sum_i |y_i - \mathbf{w}x_i|$$

Weighted Linear Regression

Some people use this as a tactic to reduce overfitting.

$$\mathbf{err}(\mathbf{w}) = \sum_i (y_i - \mathbf{w}x_i)^2 + \alpha \mathbf{w}^2$$

This is called **Ridge Regression** and `sklearn` has an implementation for it.

Weighted Linear Regression

Another alternative:

$$\mathbf{err}(\mathbf{w}) = \sum_i \mathbf{s}_i (y_i - \mathbf{w} x_i)^2$$

Here s_i is some score that we come up with that determines how important we think the datapoint is.

Weighted Linear Regression: Nerd Maths

For the math geeks. We have closed form solution.

$$S = \begin{pmatrix} s_1 & 0 & \cdots & 0 \\ 0 & s_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & s_n \end{pmatrix}$$

$$\mathbf{w}_{\text{opt}} = (X^T S X)^{-1} X^T S Y$$

Weighted Linear Regression

For the rest of us, scikit learn has this implemented.

```
mod_skew = LinearRegression()
mod_skew.fit(X_train, y_train,
             sample_weight=train_df['importance'])
```

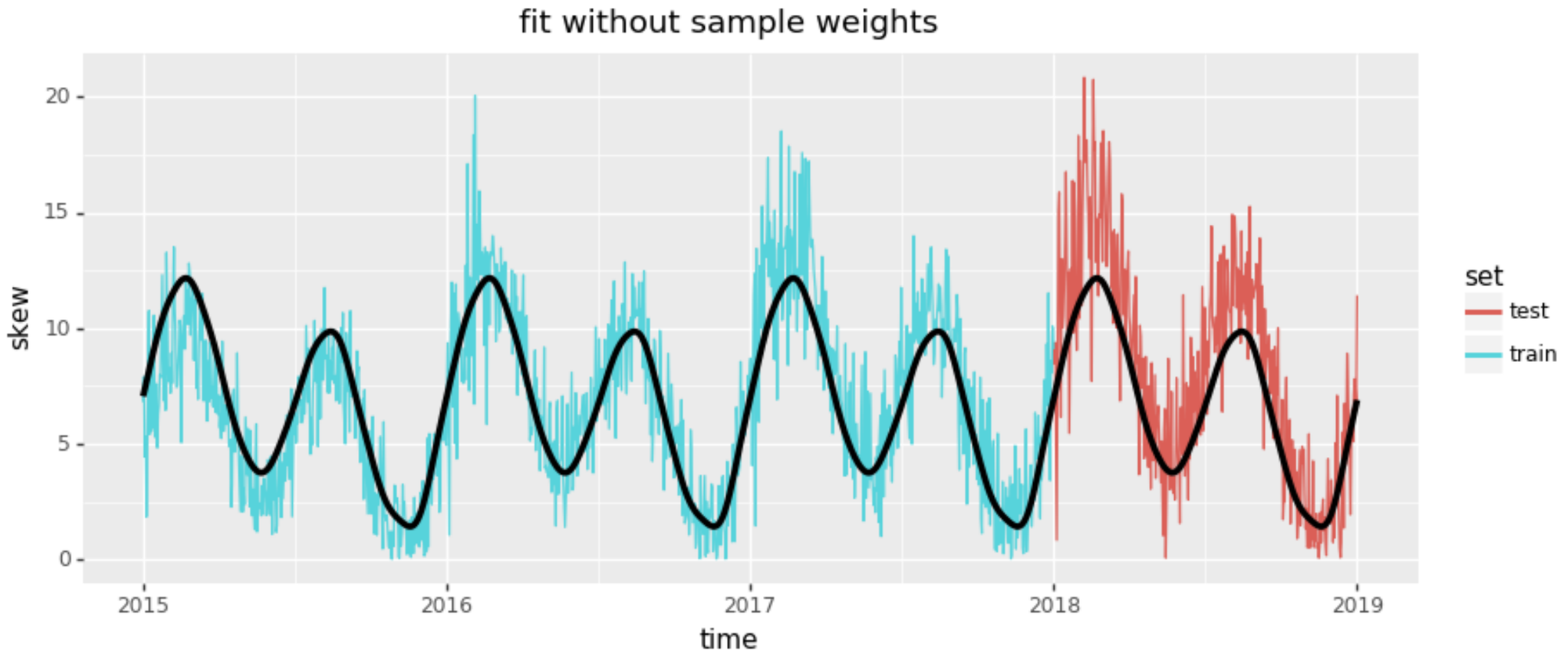
It will apply the weights to the errors automatically. Note that **Support Vector Machine** and **Logistic** models also allow you to pass in your own importance weights.

Weighted Linear Regression

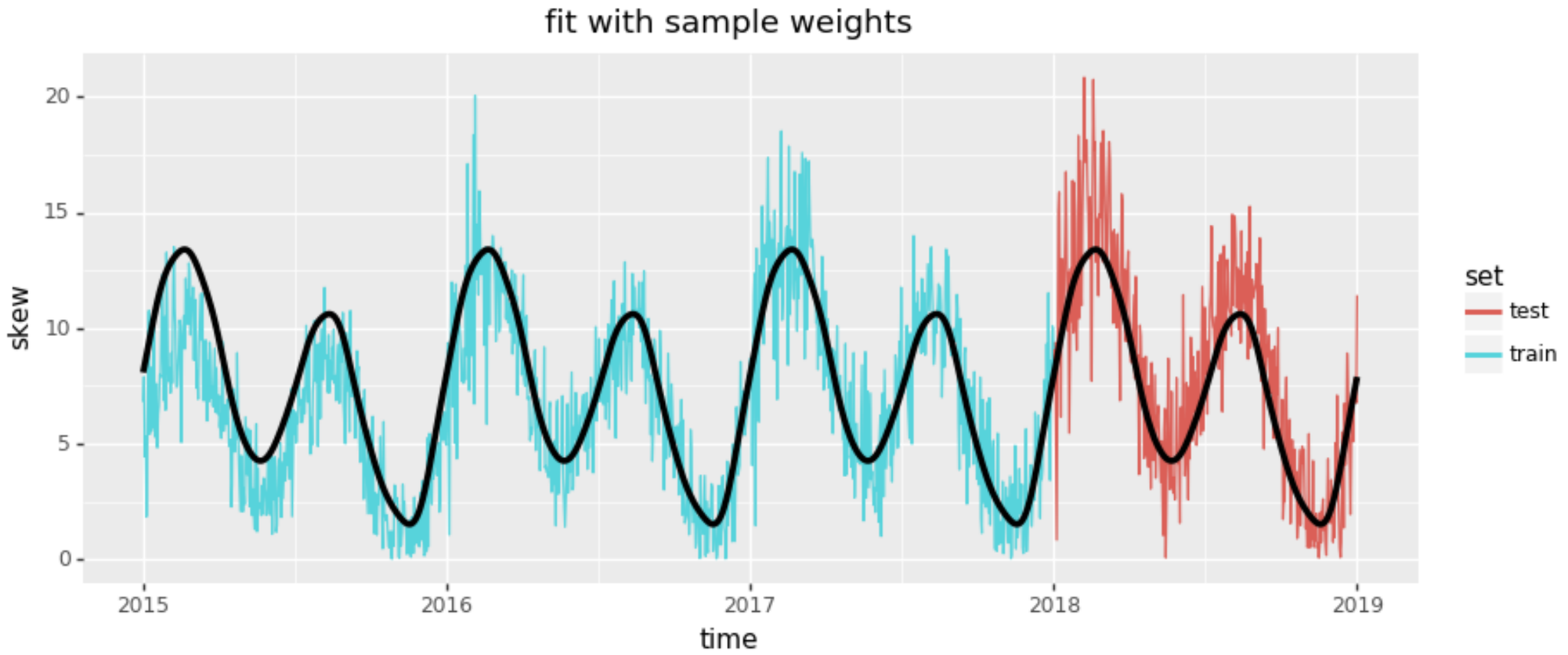
You do need to set your importance weight yourself. You could do this from **pandas** though.

year	day	y	importance
2015	1	4.026617	3.005733e-07
2015	2	4.667070	1.082064e-05
2015	3	2.636582	3.636936e-05
...			
2019	361	2.558010	1.591240e+01
2019	362	2.799675	1.593428e+01
2019	363	2.659237	1.595617e+01

Weighted Linear Regression



Weighted Linear Regression



Weighted Linear Regression

Some things to remember:

— we've used weights to tell the model we care less about the far away history and more about the recent history

— this trick may also be useful to try when you are dealing with unbalanced data

— you can tune the effect of the error as a hyperparam in a grid search if you want

Optimise Thy Hyperparameters

I like to be able to quickly play with hyperparams (**evol**!).

```
def score(alpha = 300, decay = 0.999):
    # apply pandas transformations
    ml_df = df.pipe(add_rbf_features, alpha = alpha).pipe(add_importance, decay = decay)

    # prepare data for sklearn
    radial_cols = [c for c in df.columns if 'x' in c]
    train_df, test_df = ml_df[ml_df['set'] == 'train'], ml_df[ml_df['set'] == 'test']
    X_train, X_test = train_df[radial_cols].as_matrix(), test_df[radial_cols].as_matrix()
    y_train, y_test = train_df['skew'], test_df['skew']

    # train model and return the test performance
    mod_skew = LinearRegression()
    mod_skew.fit(X_train, train_df['skew'], sample_weight=train_df['importance'])
    return np.mean(np.abs(mod_skew.predict(X_test) - y_test))
```

Weighted Linear Regression

Currently we've filtered away a seasonal trend which we assume does not change over time. We care less about the history, sure, but we haven't modelled a seasonal change.

Let's fix this, while learning a bit of R at the same time.

Interaction Terms

In R, this is how you could define column dependance.

```
y ~ x01 + x02 + x03 + x04 + x05 + x06 +
    x07 + x08 + x09 + x10 + x11 + x12
```

Interaction Terms

It is possible to define an interaction term too.

```
y ~ time*(x01 + x02 + x03 + x04 + x05 + x06 + x
          x07 + x08 + x09 + x10 + x11 + x12)
```

You will get all the RBF columns as well as all these columns multiplied by time. This is kind of like a DSL for manual feature selection.

Interaction Terms

Thanks to **patsy** you can use this trick in python too.

```
formula = "skew ~ y*(x01 + x02 + x03 + x04 + x05 +
                    x06 + x07 + x08 + x09 + x10 + x11 + x12)"
y_train, X_train = patsy.dmatrices(formula,
                                   data=ml_df[ml_df['set'] == 'train'])
```

It is very little code considering what it is all doing. Note that **patsy** automatically converts categorical/string-columns to encoded numpy arrays.

Interaction Terms

This is what `X_train` contains:

```
DesignMatrix with shape (1095, 26)
  Columns:
    ['Intercept','time','x01','x02','x03','x04',
     'x05','x06','x07','x08','x09','x10','x11',
     'x12','time:x01','time:x02','time:x03','time:x04',
     'time:x05','time:x06','time:x07','time:x08',
     'time:x09','time:x10','time:x11','time:x12']
  Terms:
    ...
  (to view full data, use np.asarray(this_obj))
```

Notes on Interaction Terms

Since we risk generating **a lot** of features this way it may be a good idea to see if we really need all these selected variables.

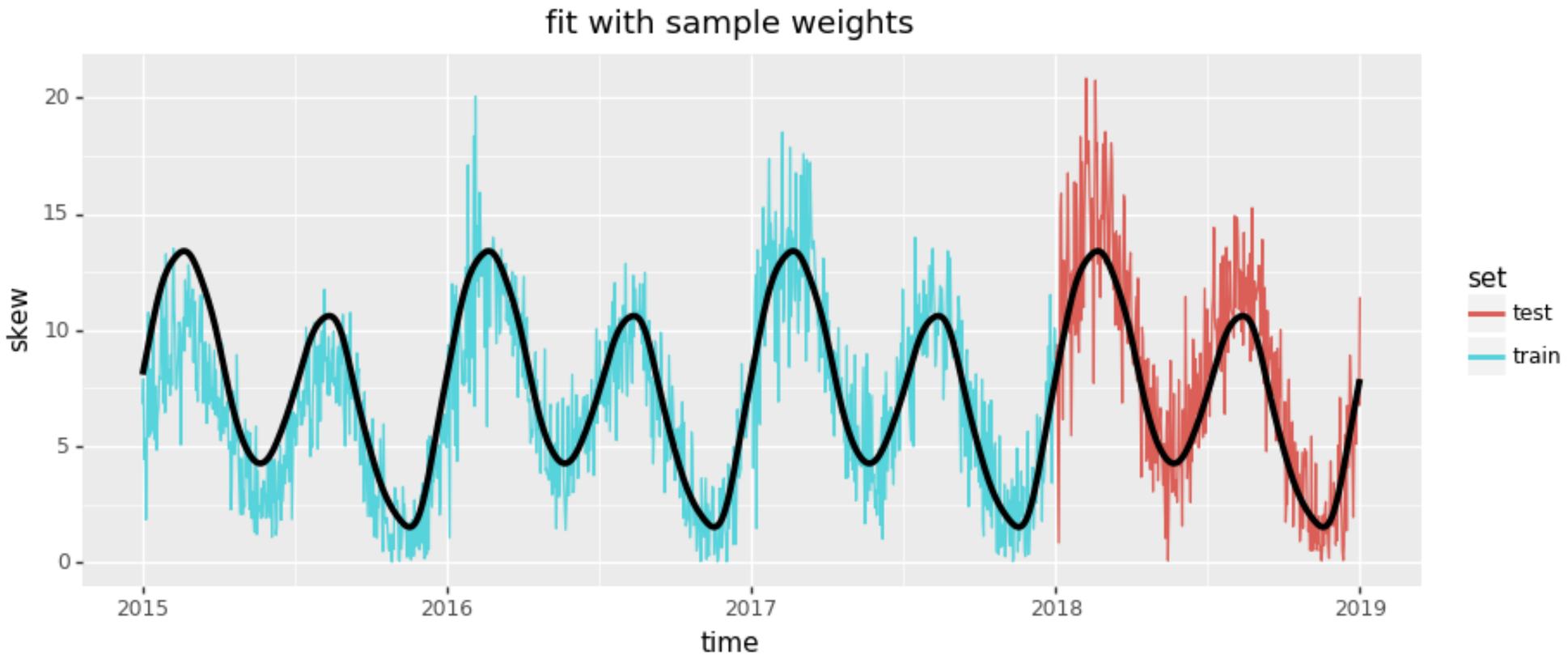
— perhaps consider a Ridge model to prevent overfitting

— you could see this as yet another hyperparameter

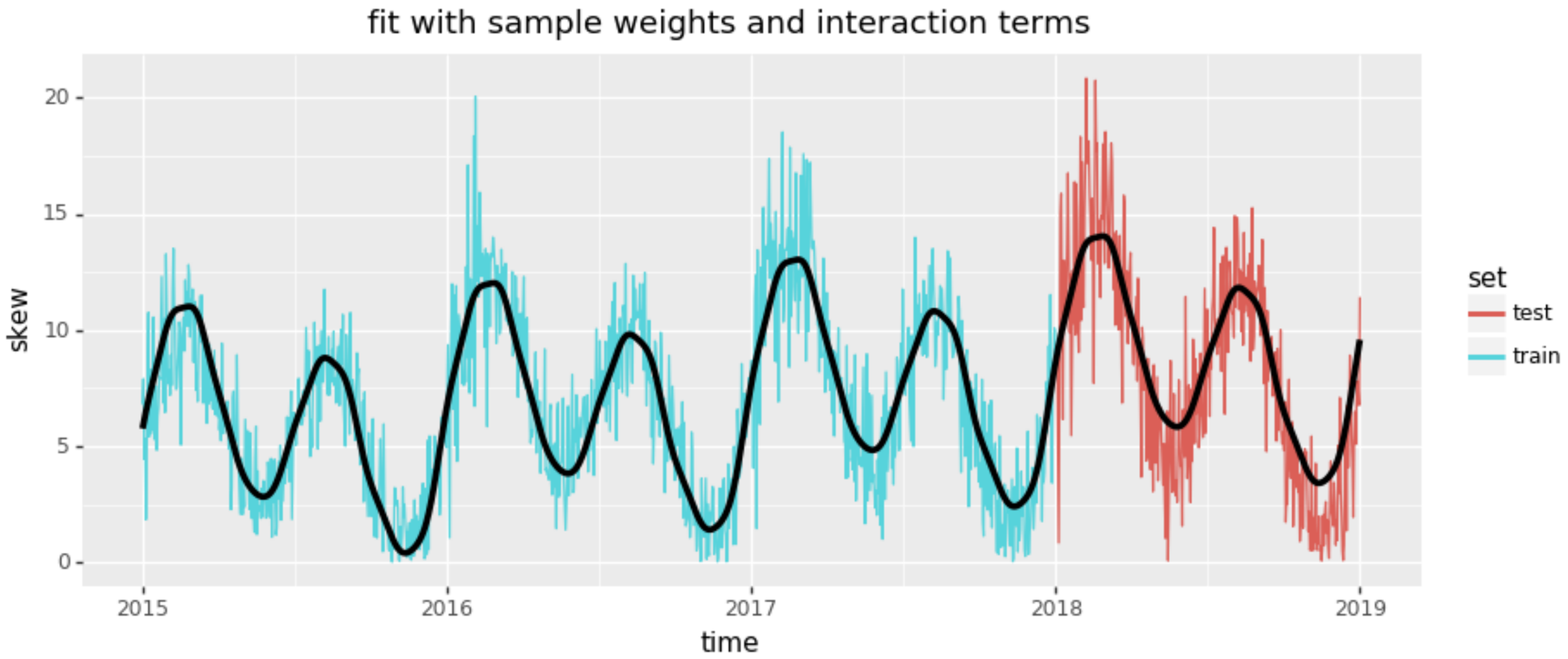
— you could apply T-tests manually (`statsmodels`)

— you could use `sklean.feature_selection`

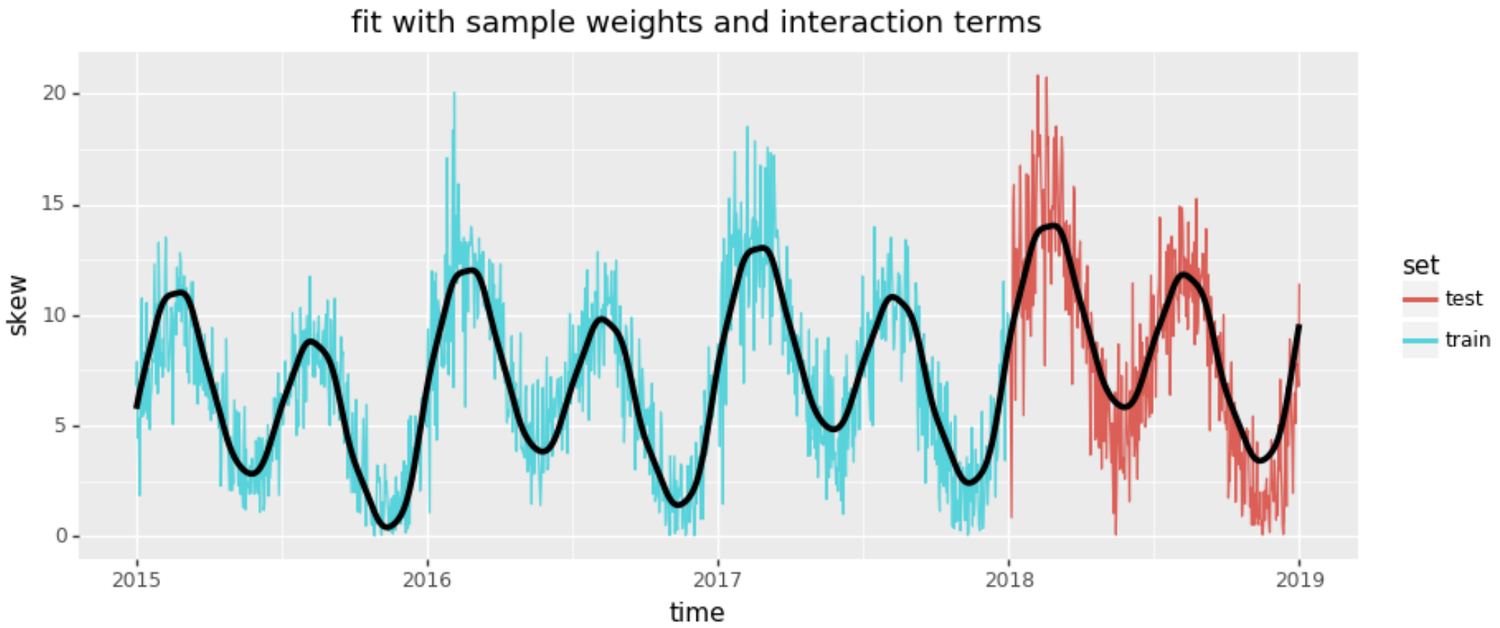
The Interaction Effect



The Interaction Effect



The Interaction Effect



Some parts are predicted better, other parts worse.

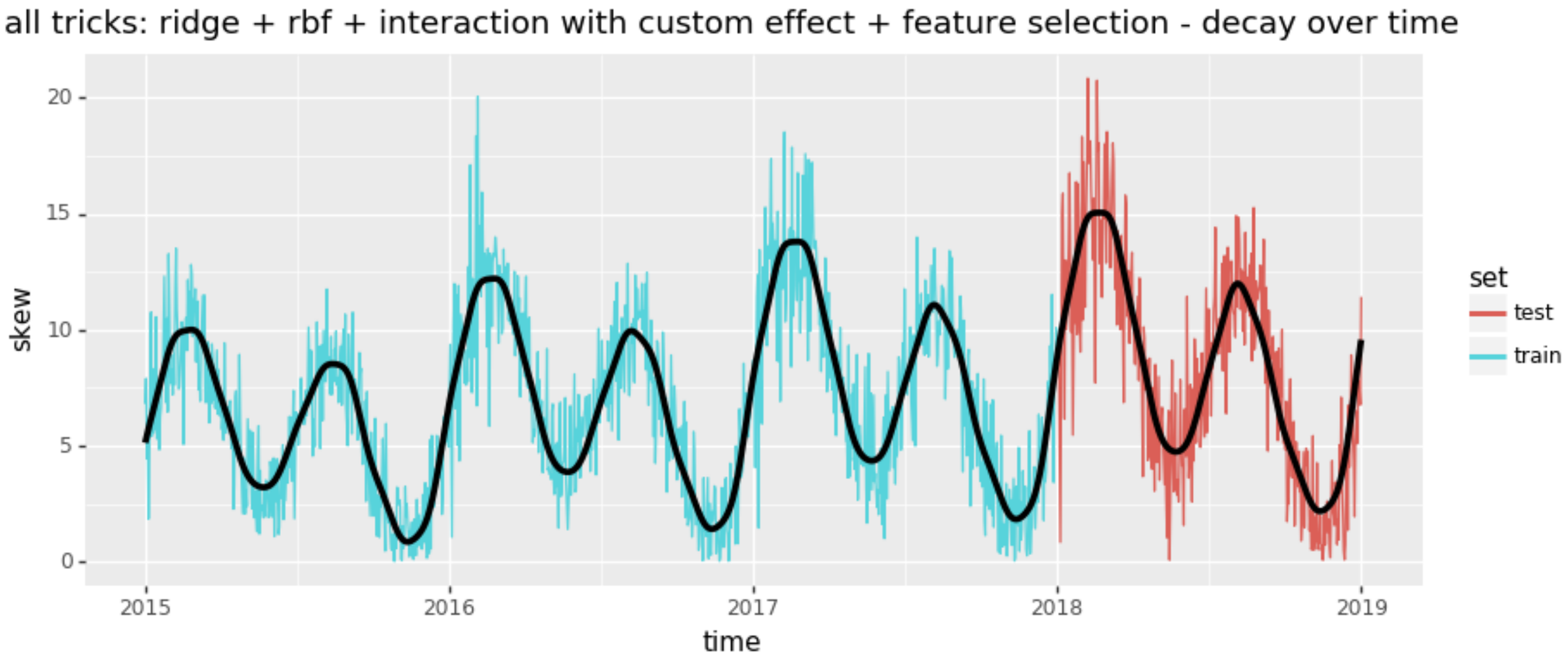
A Bit of Trial And Error Ensues

After some trial and error I decided to drop the importance weights per row and I switched from linear regression to a Ridge regression.

I was afraid about overfitting so I decided to use a feature selector from **sklearn**.

```
from sklearn.feature_selection import RFE
rfe_mod = Ridge(alpha=0.00001)
mod_feature_cv = RFE(rfe_mod, step=5).fit(X_train, y_train)
```

A Bit of Trial And Error Ensues



Why Not DeepLearning[tm]?

Our example had 3 years of data (≈ 1100 datapoints).
This is not a whole lot of data for a deep learning model, you want to have much more to train all the weights.

Why Not DeepLearning[tm]?

Our example had 3 years of data (≈ 1100 datapoints).
This is not a whole lot of data for a deep learning model, you want to have much more to train all the weights.

Linear models are easy to maintain and debug.

Why Not DeepLearning[tm]?

Our example had 3 years of data (≈ 1100 datapoints).
This is not a whole lot of data for a deep learning model, you want to have much more to train all the weights.

Linear models are easy to maintain and debug.

Linear models actually train kind of fast, which is great.

Why Not DeepLearning[tm]?

Our example had 3 years of data (≈ 1100 datapoints). This is not a whole lot of data for a deep learning model, you want to have much more to train all the weights.

Linear models are easy to maintain and debug.

Linear models actually train kind of fast, which is great.

Linear models are easy to explain to humans of management.

Why Not DeepLearning[tm]?

Here comes my favorite reason.

Linear models are convex! This means that math tells us the optimiser will always converge to the maximum fit. Tensorboard is a cool tool but it's even cooler if you don't need it.

Why Not DeepLearning[tm]?

Don't get me wrong. I've put deep learning systems into production and I like the algorithms. They solve problems I couldn't solve with other algorithms.

But production is dangerous. We have to code up checks to confirm the new data didn't cause the optimiser to get stuck in a wrong optimum.

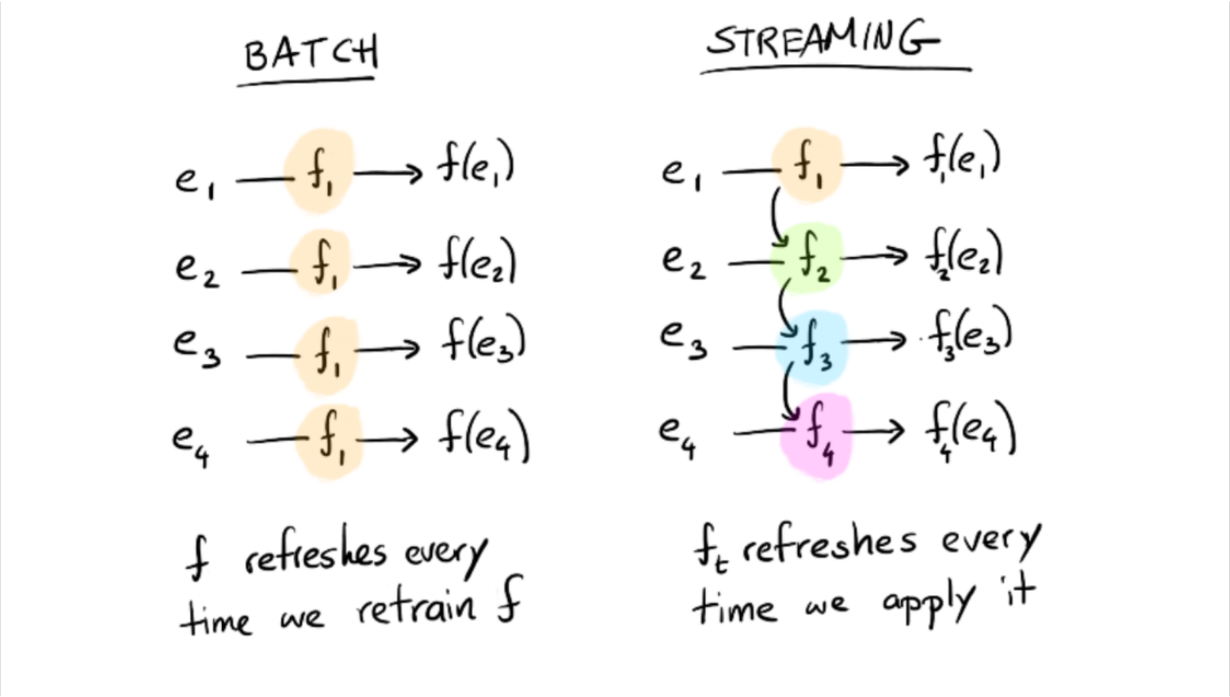
Speaking of Production

Next Up; Streaming

Instead of handling data in a batch setting we sometimes need to deal with models in a stream setting. Preferably we have models that can adapt and update very quickly.

Passive Agressive Algorithms

There's a cool trick about linear models: streaming!



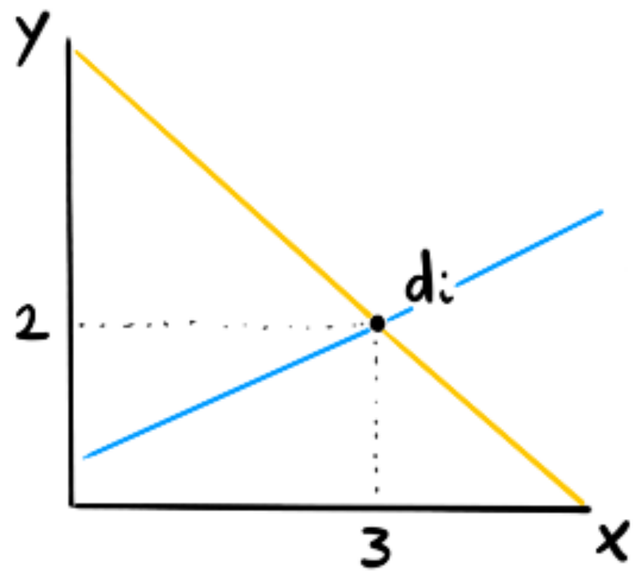
Passive Agressive Algorithms

There's multiple ways to do this. I'll briefly discuss PA.

There is an implementation in sklearn, but you could also imagine an easy implementation for apache flink or spark streaming.

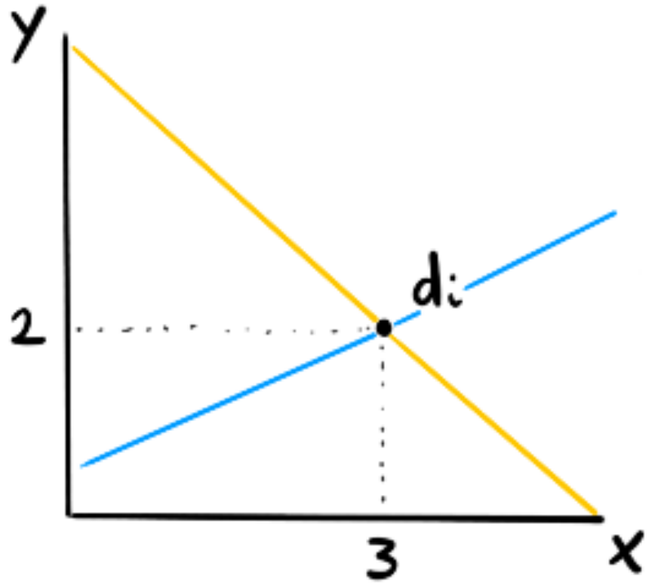
Passive Agressive Algorithms

Supppose we are doing a regression for point d_i.



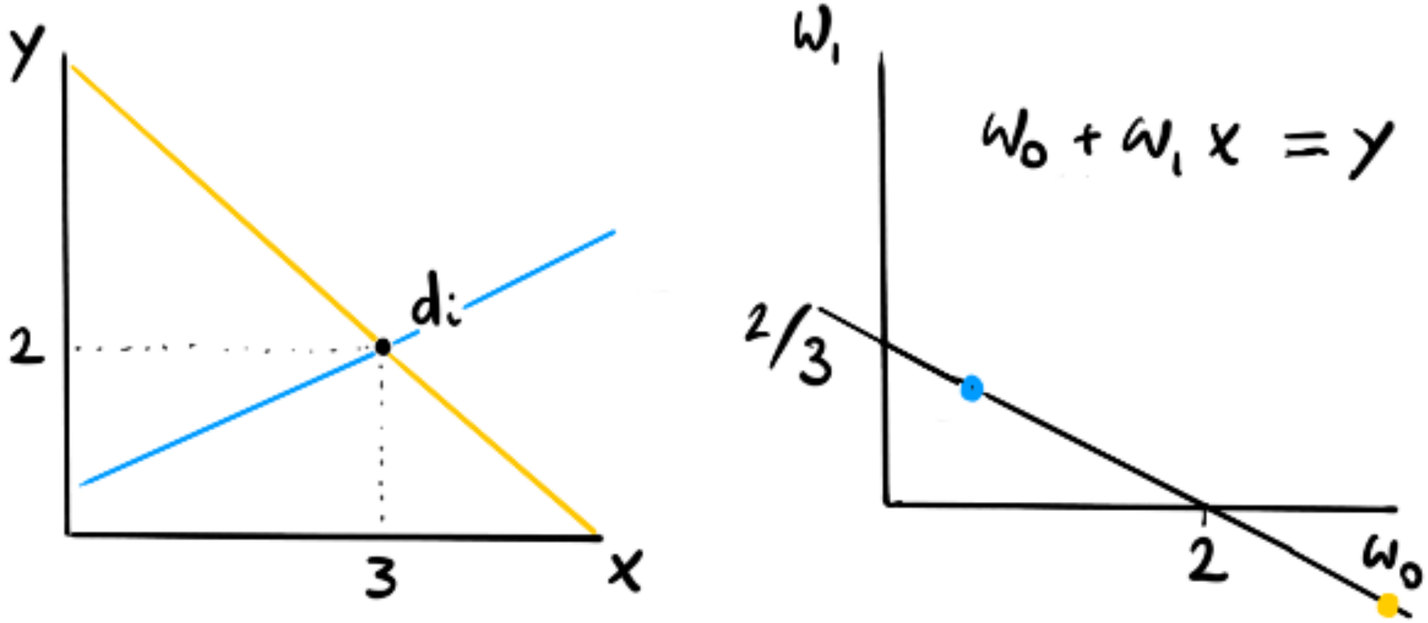
Passive Agressive Algorithms

For 1 datapoint, the blue and yellow line are equal in fit.



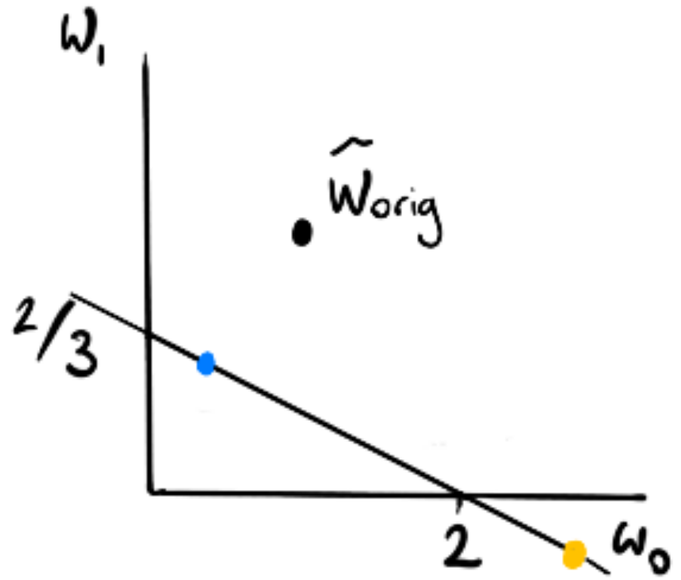
Passive Agressive Algorithms

We can also look at the point in weight space.



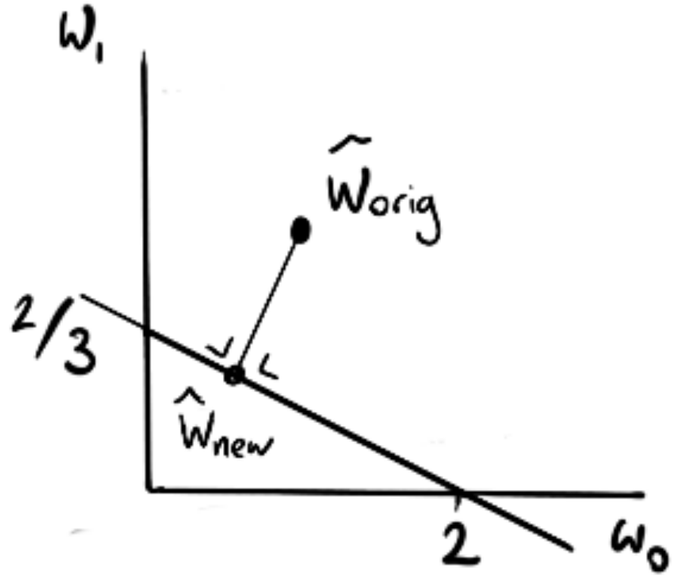
Passive Agressive Algorithms

Suppose that we had weights from before d_i.



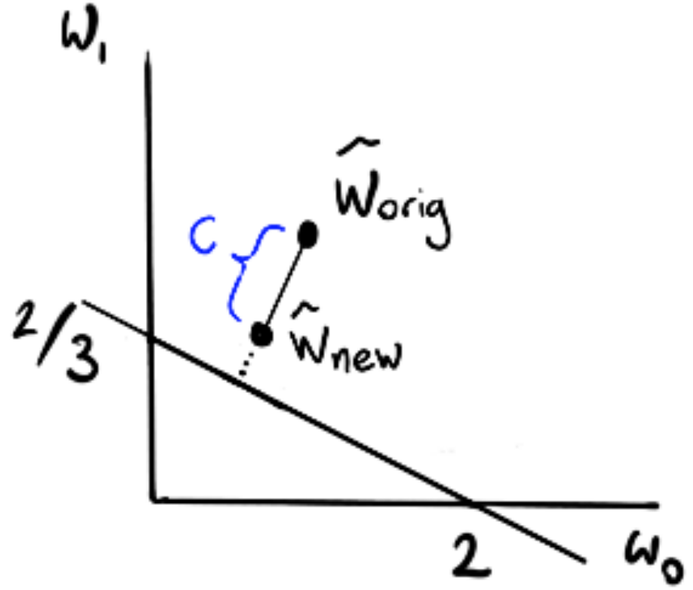
Passive Agressive Algorithms

We know the shortest path to make d_i fit perfectly.



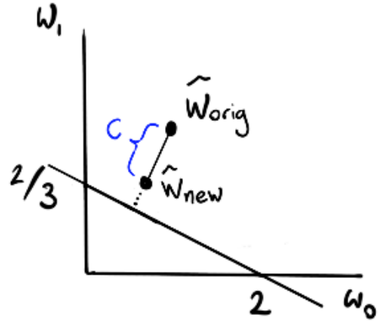
Passive Agressive Algorithms

Let's never update more than a certain stepsize C.



Passive Agressive Algorithms

Let's only update if $\text{dist}(w_{\text{orig}}, w_{\text{new}}) > e$



That way the algorithm either does not update (passive) or it does a large update (agressive).

Passive Agressive Algorithms

Nice. We now have an algorithm to update weights of a regression in a stream. Turns out that **sklearn** has an implementation of this (both for regression and classification).

Note that this streaming approach is interesting when you run your algorithm in batch too. The memory needed for a streaming approach is much smaller because you don't need the entire dataset in memory.

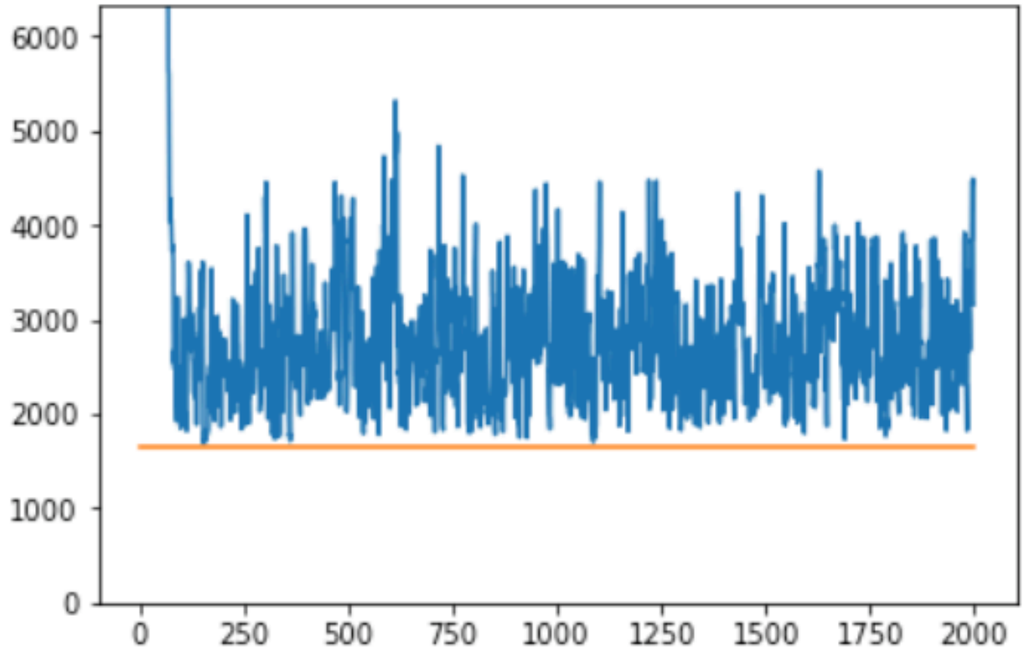
Passive Agressive Algorithms

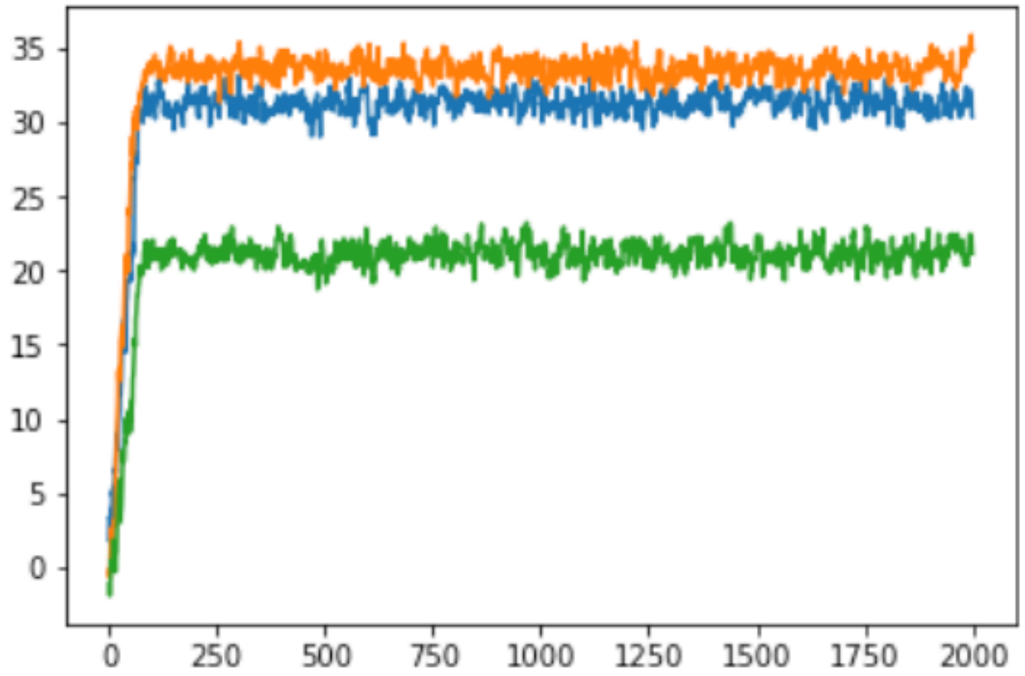
In practice you probably want to introduce a stepsize for when the algorithm just started and when it is hot.

I've started a small experiment:

— use the **sklearn** randomdata
— after 30 datapoints use C_{hot}, before C_{cold}
— compare to a normal regression on entire dataset
— `sklearn.linear_model.PassiveAggressiveRegressor`

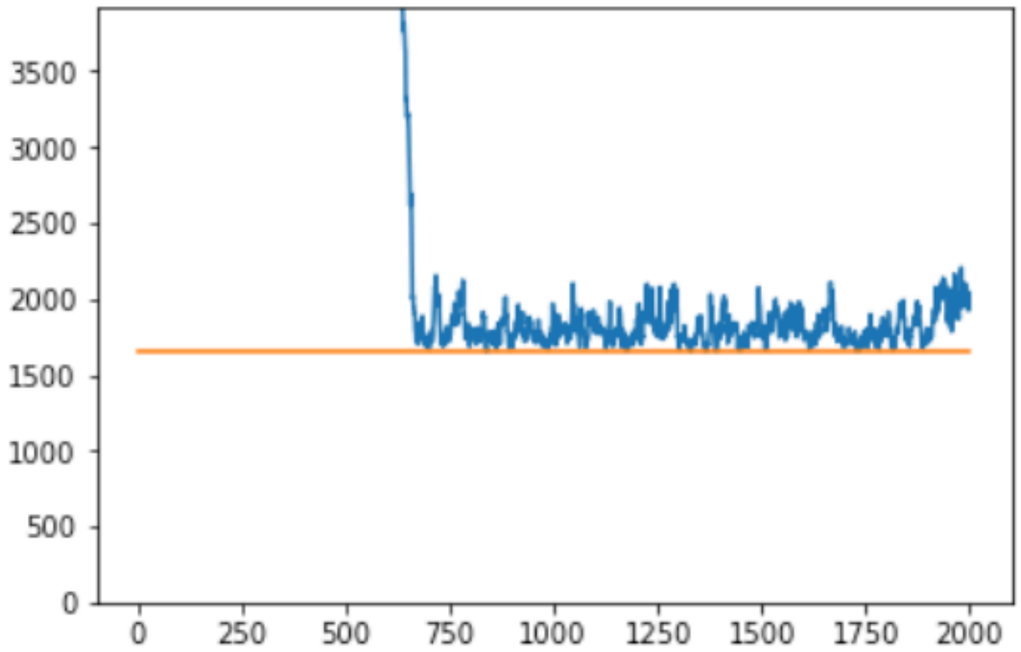
$C_{\text{cold}} = 1$ and $C_{\text{warm}} = 1$

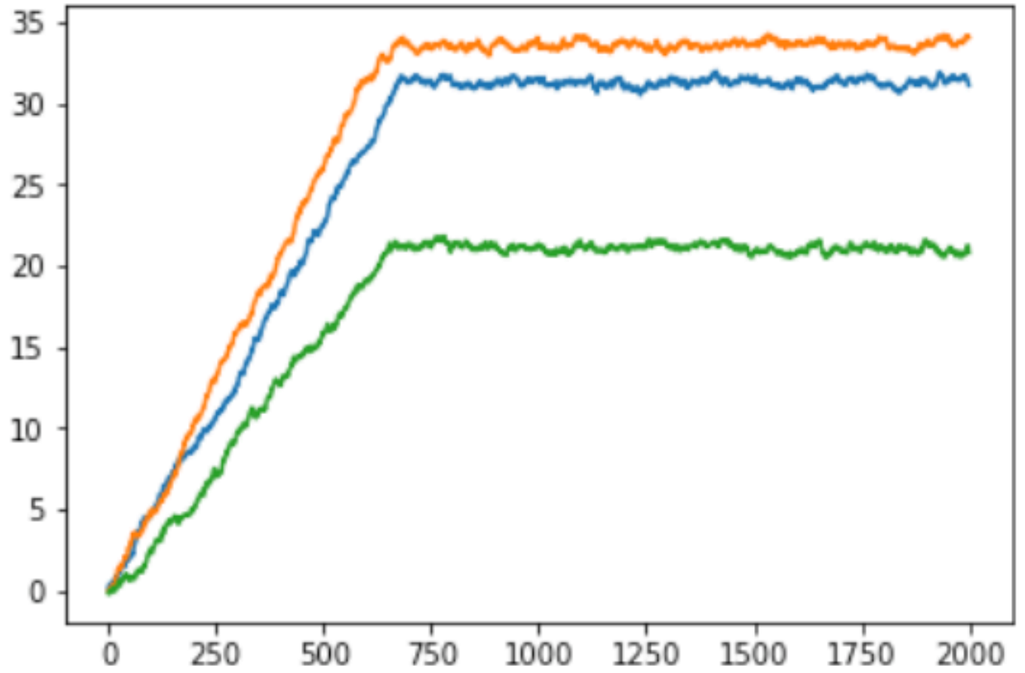




The first plot shows the mean squared error over the entire set after the regression has seen more data. The orange line demonstrates the baseline performance of the batch algorithm. The second plot demonstrates how the weights change over time. In this case you can confirm that the MSE fluctuates quite a bit.

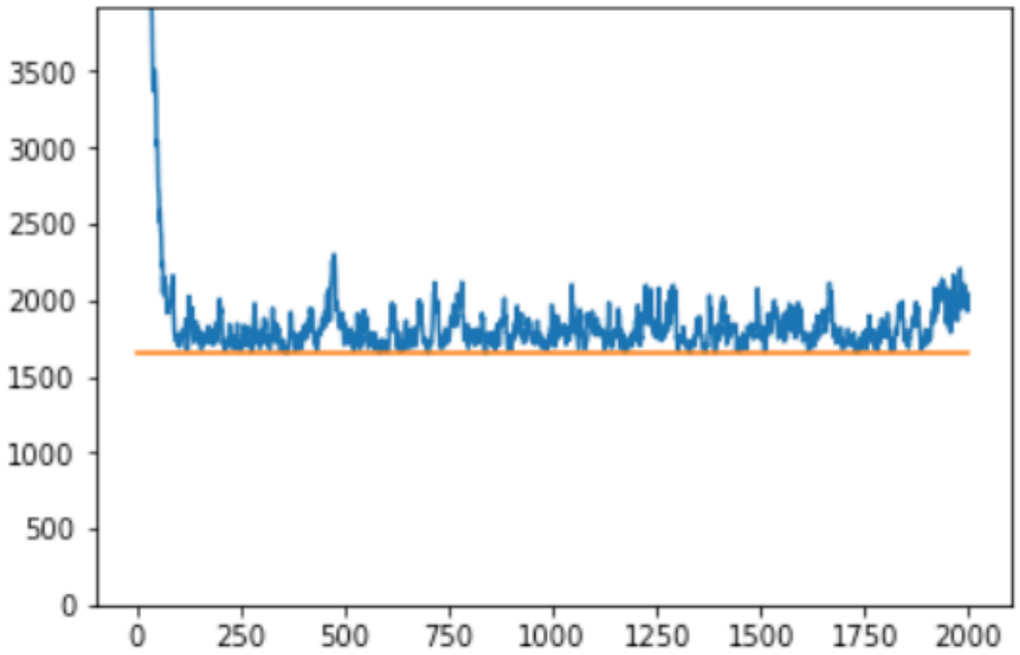
$C_{\text{cold}} = 0.1$ and $C_{\text{warm}} = 0.1$

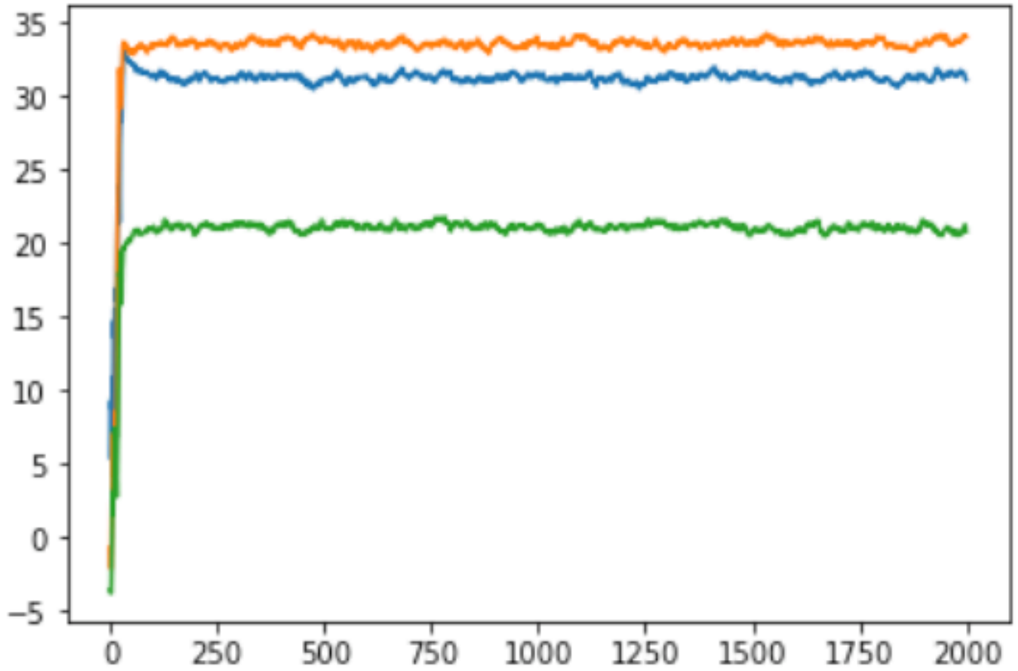




The fluctuations are small, but the algorithm seems to need a lot of data before the regression starts to become sensible.

$C_{\text{cold}} = 3$ and $C_{\text{warm}} = 0.1$





You can now see that the fluctuations are still very small but the large steps that are allowed in the first few iterations ensure that the algorithm can converge a bit globally before it starts to limit itself to only local changes.

Passive Agressive Algorithms

Note that this approach on a live system is especially useful when:

— you have labels that come in during your stream

— you want near realtime updating of the weights

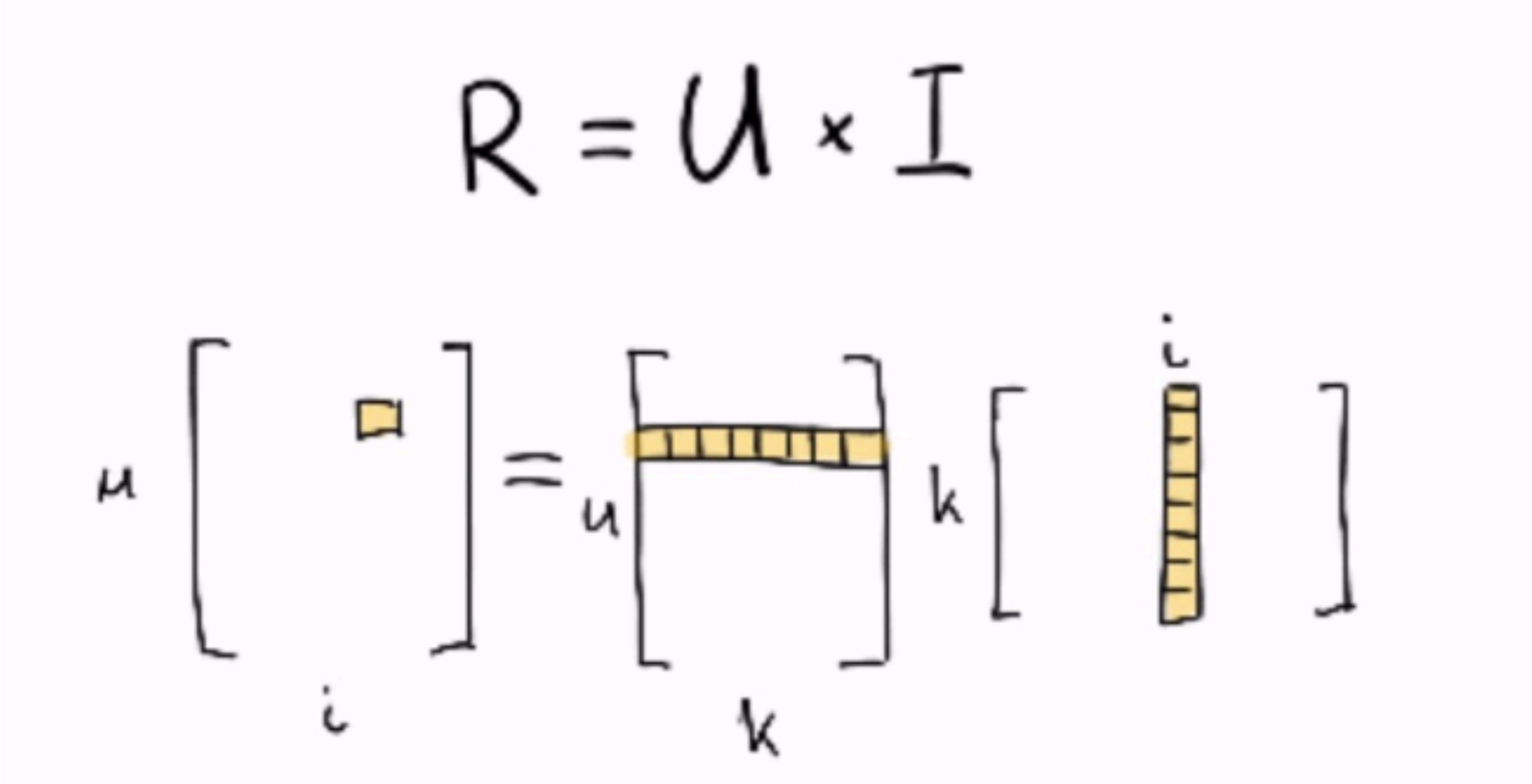
— you might have a world that changes over time, this algorithm favors recent datapoints

More details/maths on blog or in the original paper.

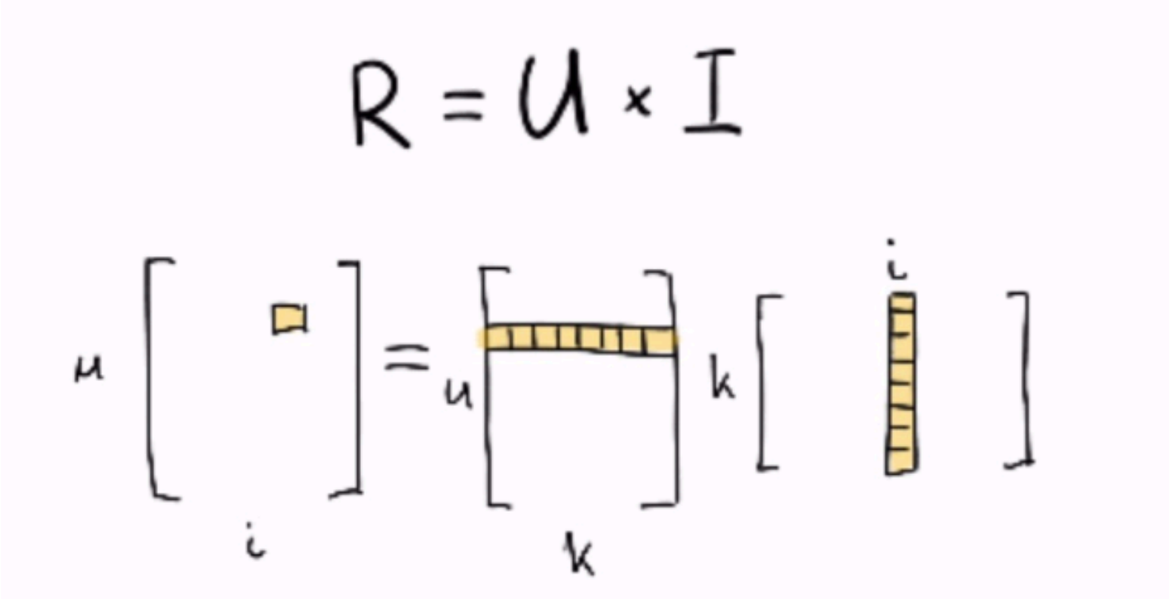
"But, when do you have labels that come in a stream?"

— The Audience

Let's talk Recommender Systems

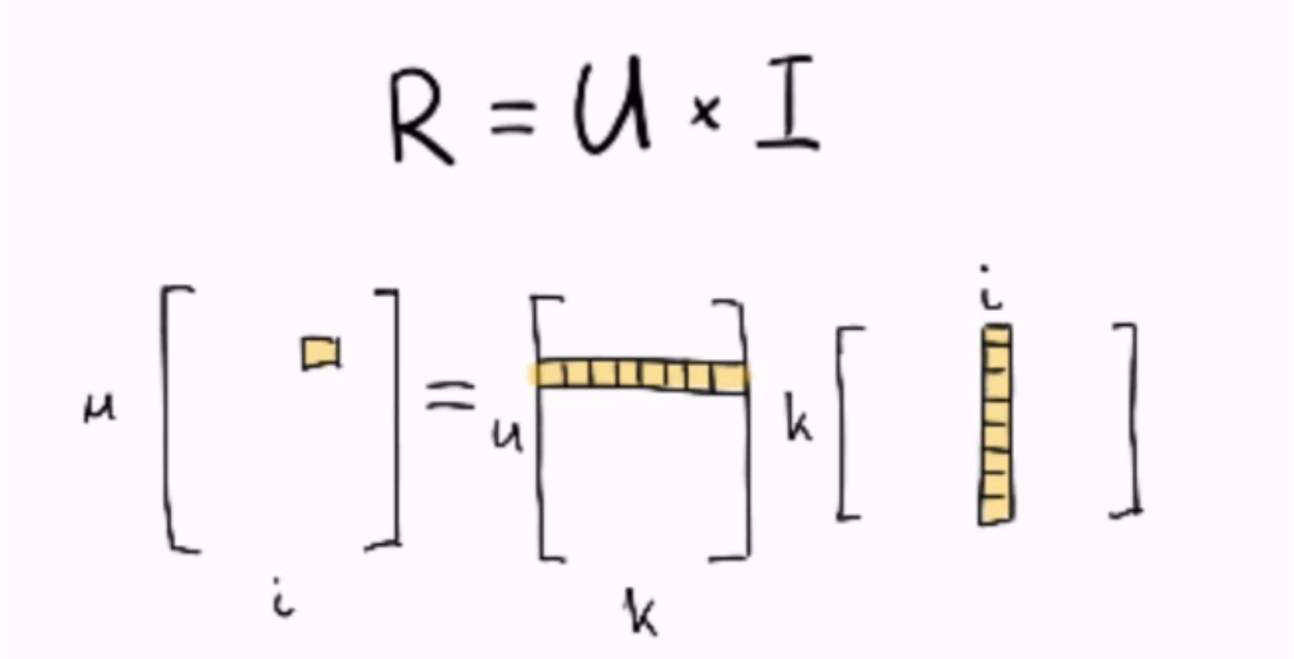


Let's talk Recommender Systems



The items features won't change, but the user features we might want to update per click, ASAP!

Can you spot the regression?



$$\text{click}(u, i) \approx \text{sigmoid}(u_0 i_0 + \ldots + u_k i_k)$$

Speaking of Recommenders

It could make sense to use a deep learning method instead (latent space for an item-feature vector is very sensible). But which is more important:

— to have a more accurate algorithm that can be updated once per day

— to have an algorithm that can update it's belief at every mouseclick

Don't just think ML, think about system design.

Speaking of Recommenders

I wouldn't recommend this collaborative/neural approach if you're just starting out though.

There's a much simpler algorithm that I find to work a lot better.

Speaking of Recommenders

Let's pretend we're about to build a recommender at the dutch BBC. We could calculate what is popular.

$$p(\text{content } i \rightarrow \text{content } j)$$

Speaking of Recommenders

Let's pretend we're about to build a recommender at the BBC. We could calculate what is popular.

$$p(\text{content } i \rightarrow \text{content } j)$$

It would be better instead to calculate.

$$\frac{p(\text{content } i \rightarrow \text{content } j)}{p(\text{not content } i \rightarrow \text{content } j)} \approx \frac{p(\text{content } i \rightarrow \text{content } j)}{p(\text{content } j)}$$

Speaking of Recommenders

$$\frac{p(\text{content } i \rightarrow \text{content } j)}{p(\text{content } j)}$$

series_i	series_j	prob_i_and_j	prob_j	rating_score
content_a	content_b	0.2	0.1	2
content_a	content_c	0.1	0.01	10
content_a	content_d	0.05	0.02	2.5
content_a	content_e	0.4	0.3	1.3333
...				

Speaking of Recommenders

$$\frac{p(\text{content } i \rightarrow \text{content } j)}{p(\text{content } j)}$$

Note that this item-item recommender;

— so simple, you could write this algorithm in SQL
— this figure is easily calculated on a stream of data
— all the parts of the algorithm are interpretable

Speaking of Recommenders

$$R_{i \to j} = \frac{p(\text{content } i \to \text{content } j)}{p(\text{content } j)}$$

Note that it is easy to turn into a personal one too.

$$R_{\{k,l,m\} \to j} = R_{k \to j} \times R_{l \to j} \times R_{m \to j}$$

Speaking of Recommenders

I've seen this algorithm go to production a bunch and it was pretty hard to beat.

There's only one algorithm that beat it when I was at a video content company. Hint; the algorithm was even simpler.

Speaking of Recommenders

I've seen this algorithm go to production a bunch and it was pretty hard to beat.

There's only one algorithm that beat it when I was at a video content company. Hint; the algorithm was even simpler.

You can also just recommender the next episode.

Speaking of Recommenders

It would be awkward if the video service:

— never tried recommending next episode first

Speaking of Recommenders

It would be awkward if the video service:

— never tried recommending next episode first
— never implemented A/A or A/random baselines

Speaking of Recommenders

It would be awkward if the video service:

— never tried recommending next episode first
— never implemented A/A or A/random baselines
— was thinking about using DeepLearning[tm] instead

Speaking of Recommenders

It would be awkward if the video service:

— never tried recommending next episode first
— never implemented A/A or A/random baselines
— was thinking about using DeepLearning[tm] instead

Speaking of Recommenders

It would be awkward if the video service:

— never tried recommending next episode first
— never implemented A/A or A/random baselines
— was thinking about using DeepLearning[tm] instead
— was not thinking beyond a notebook

Focus on hyped algorithms can be **dangerous**. Please start with the simplest end to end pipeline before

Two More Examples

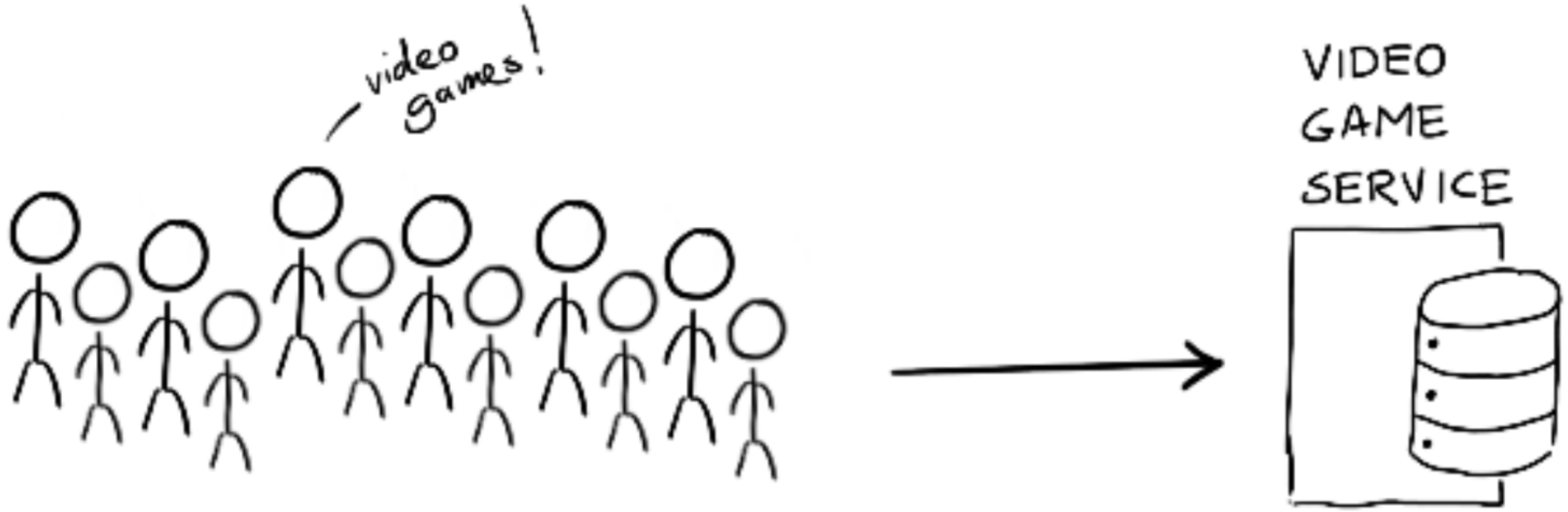
I'll conclude with two more examples that hopefully will convince you even more to try modelling things yourself. I'll even give you a glimpse of what ML programming might look like in the (hopefully) nearby future.

— one example is about video games

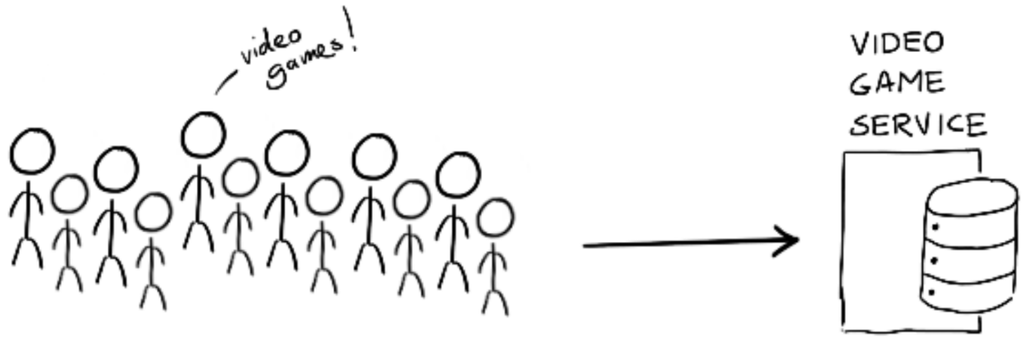
— one example is about chickens

video games go first

This is our enterprise usecase.



video games go first



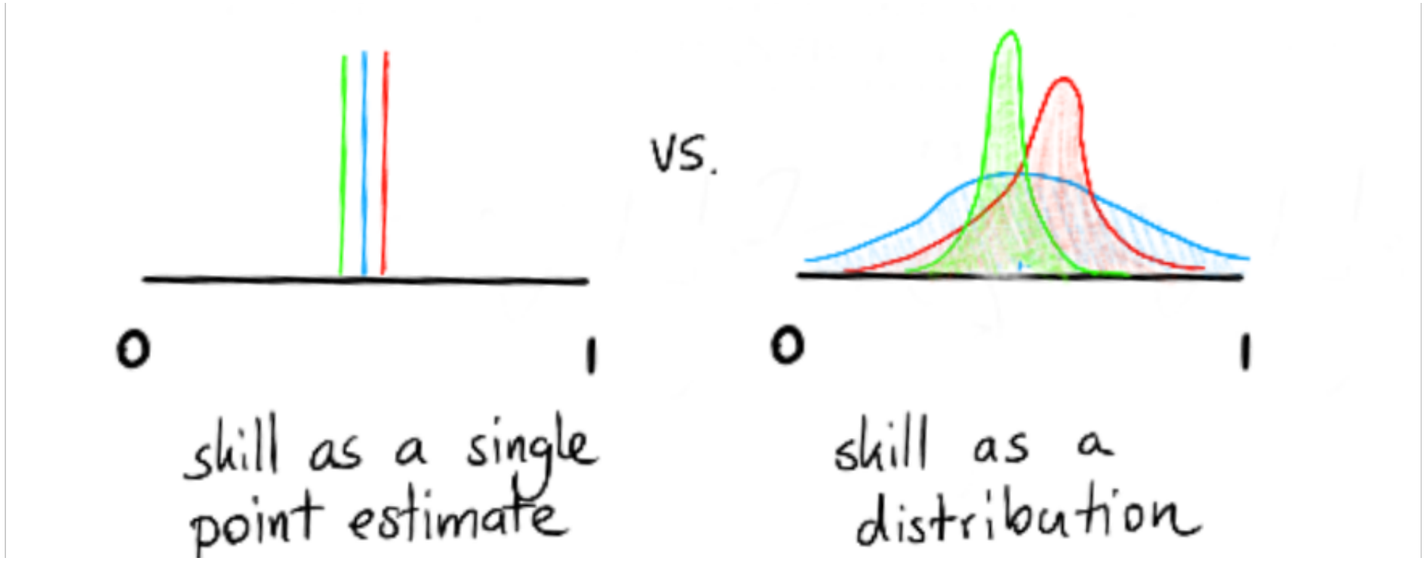
— we need to estimate player skill somehow

— we need to learn this from a stream of match outcomes

— we don't want to wait for a batch algorithm

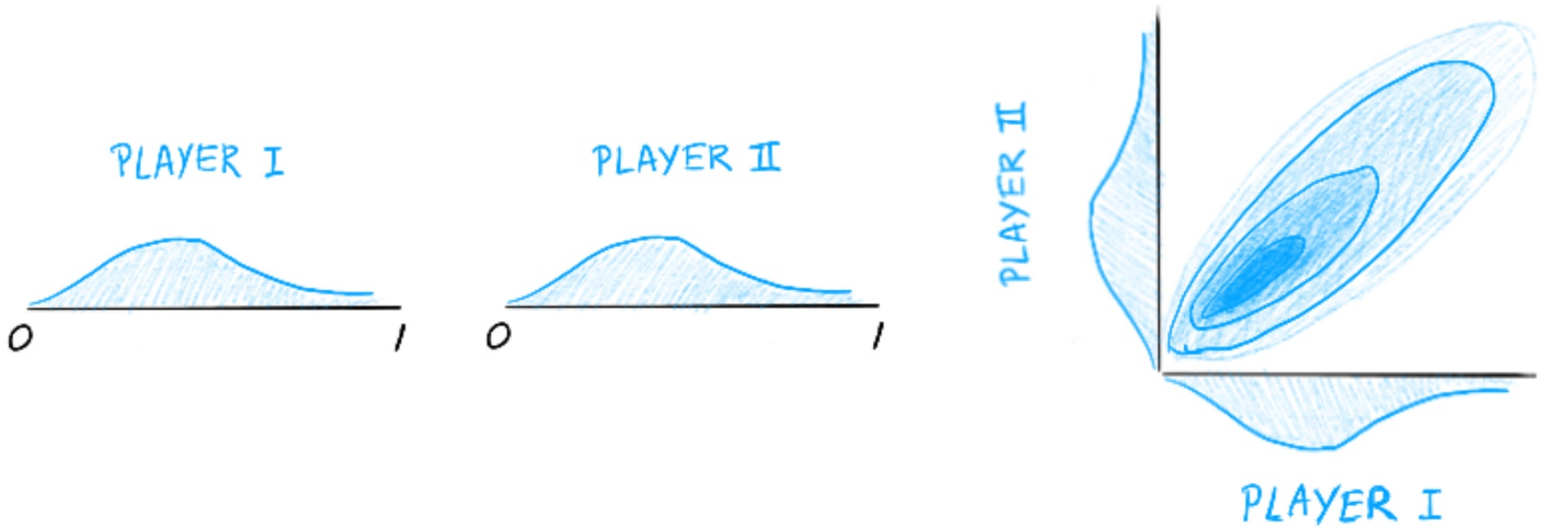
a simple solution

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.



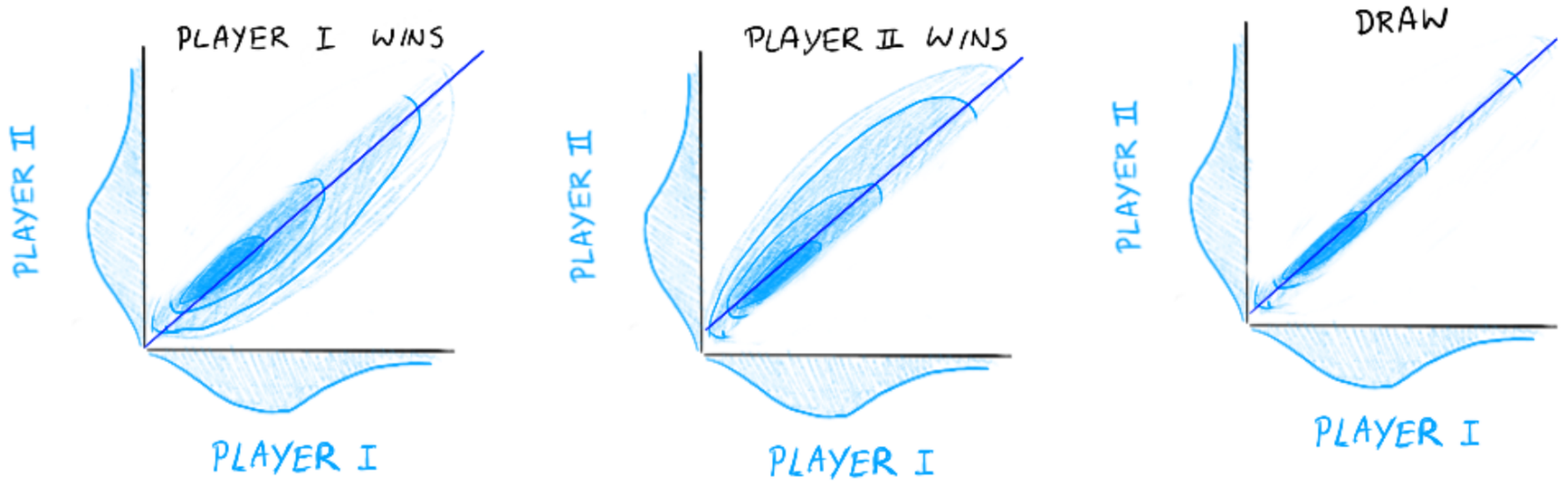
a simple solution

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.



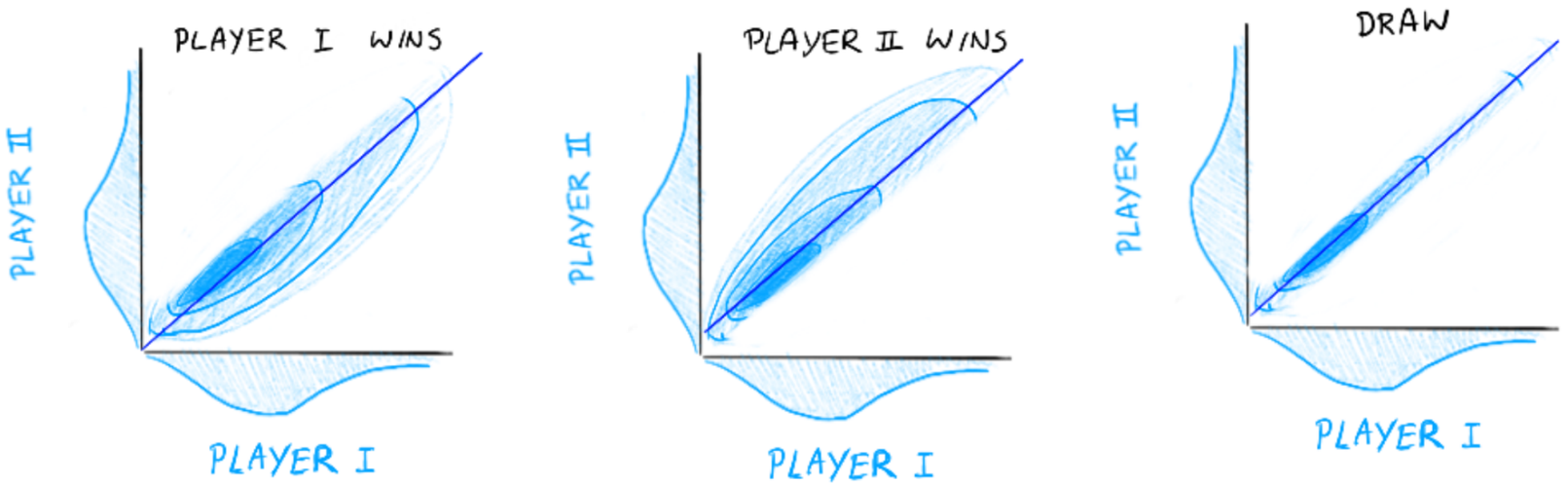
a simple solution

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



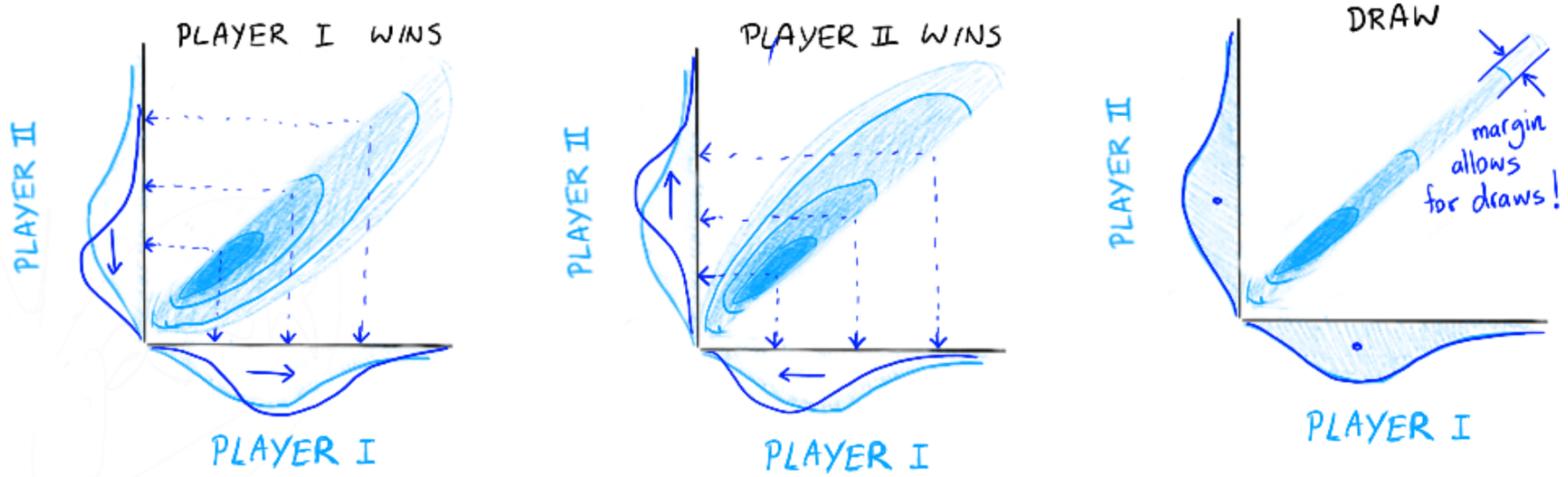
a simple solution

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



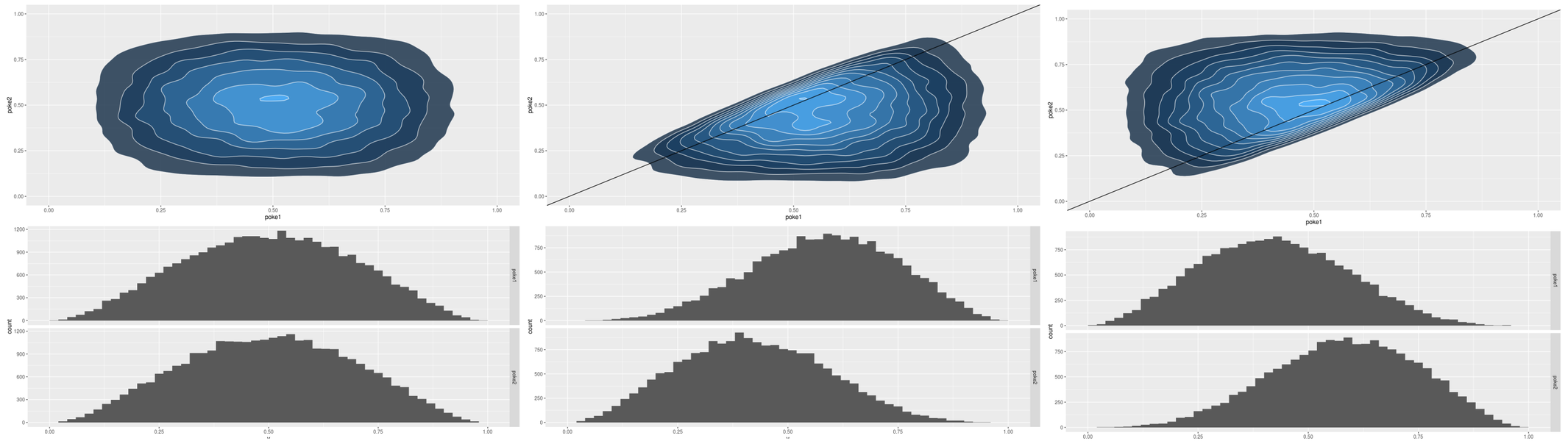
a simple solution

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



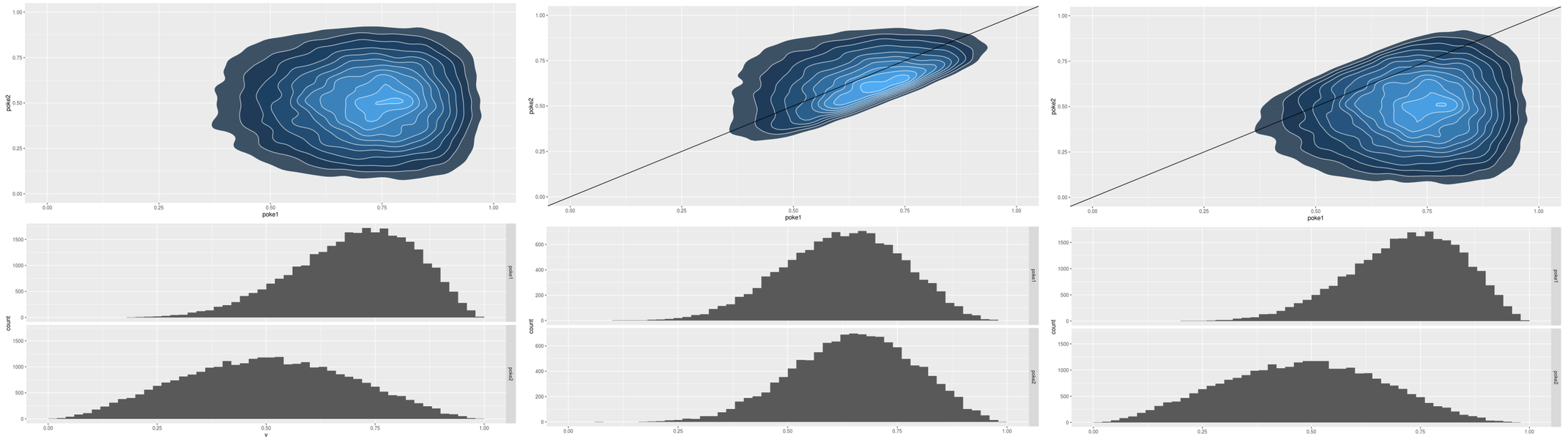
a simple simulation: two equal players

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

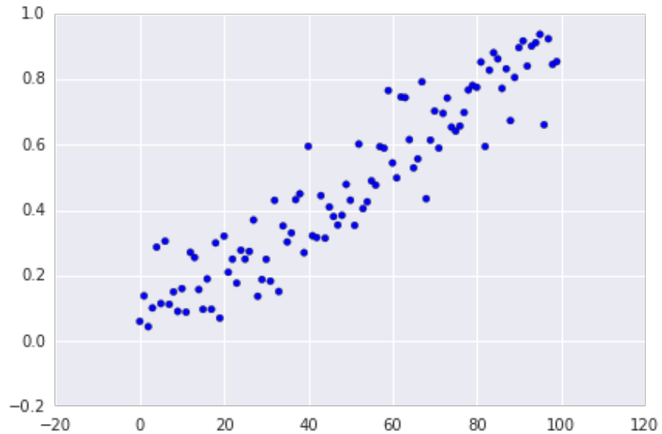


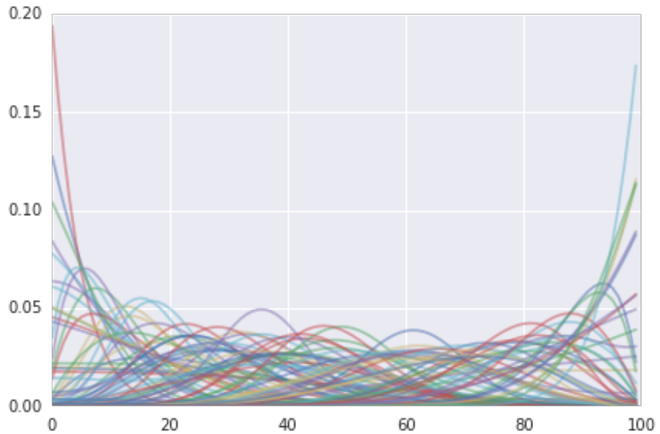
a simple simulation: two unequal players

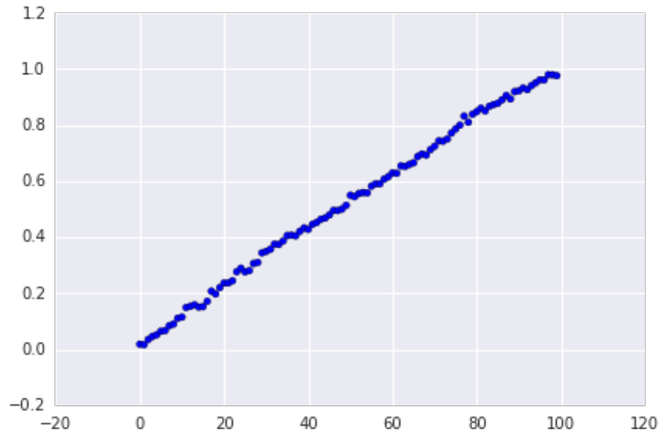
We may learn a lot, or very little. #informationtheory

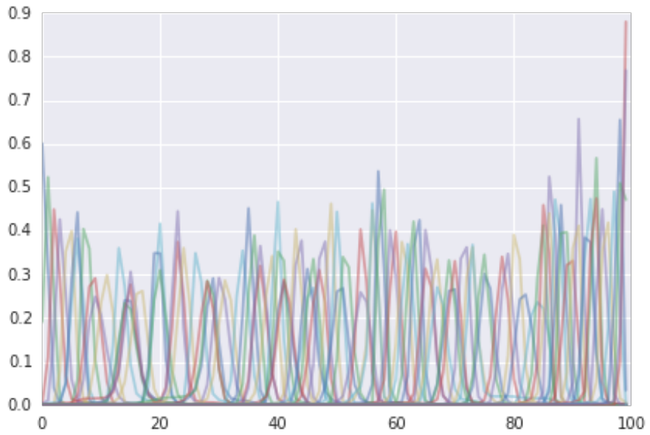


a simple simulation: many players





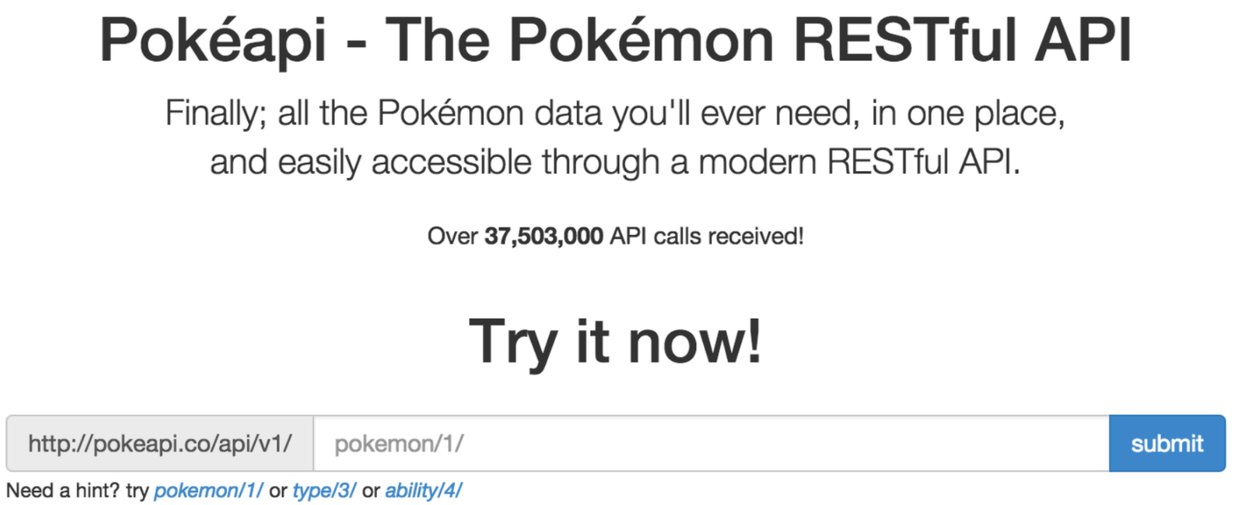




test of functionality: pokemon!

test of functionality: pokemon!

We got our infromation on pokemon from;



Nowadays you can also find the dataset on kaggle.

test of functionality: pokemon!

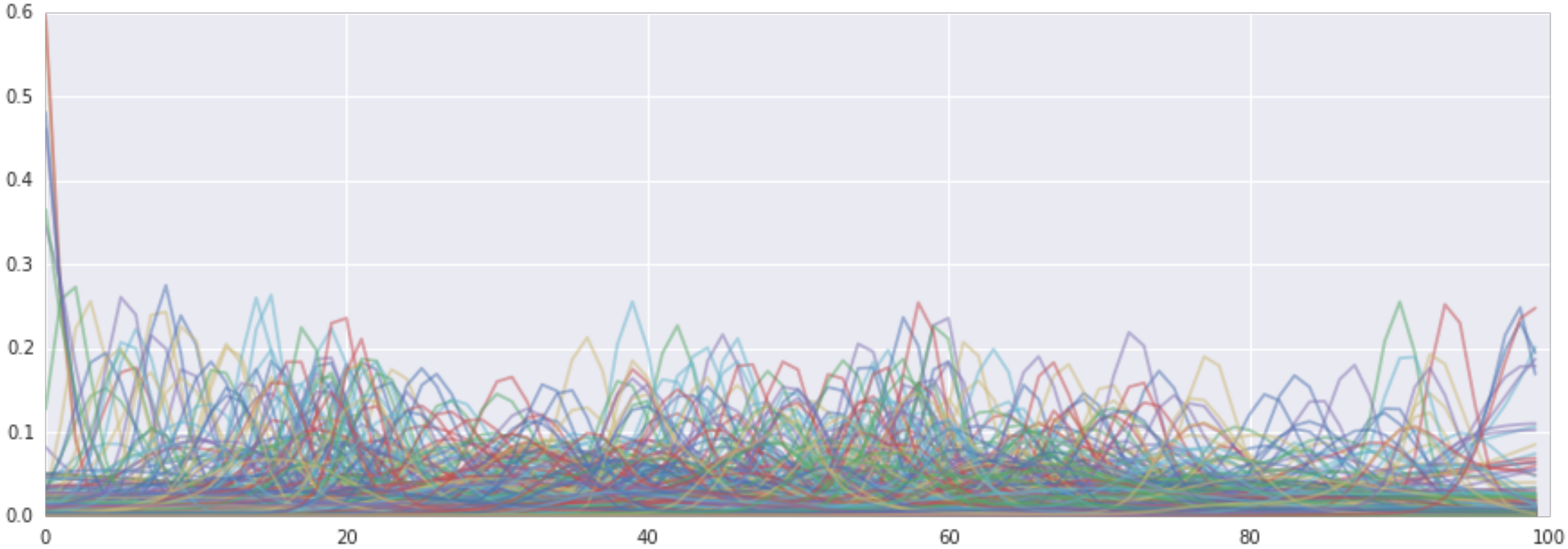
If you google around fan reddits you can find information on how many turns one pokemon can outlast the other.

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

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

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

Designing the algorithm became a whole lot easier when we admitted that we want to quantify our uncertainty. Using distributions as our state, not mere statistics, made the algorith rather simple but very smart.

the general maths of all this

Designing the algorithm became a whole lot easier when we admitted that we want to quantify our uncertainty. Using distributions as our state, not mere statistics, made the algorith rather simple but very smart.

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

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

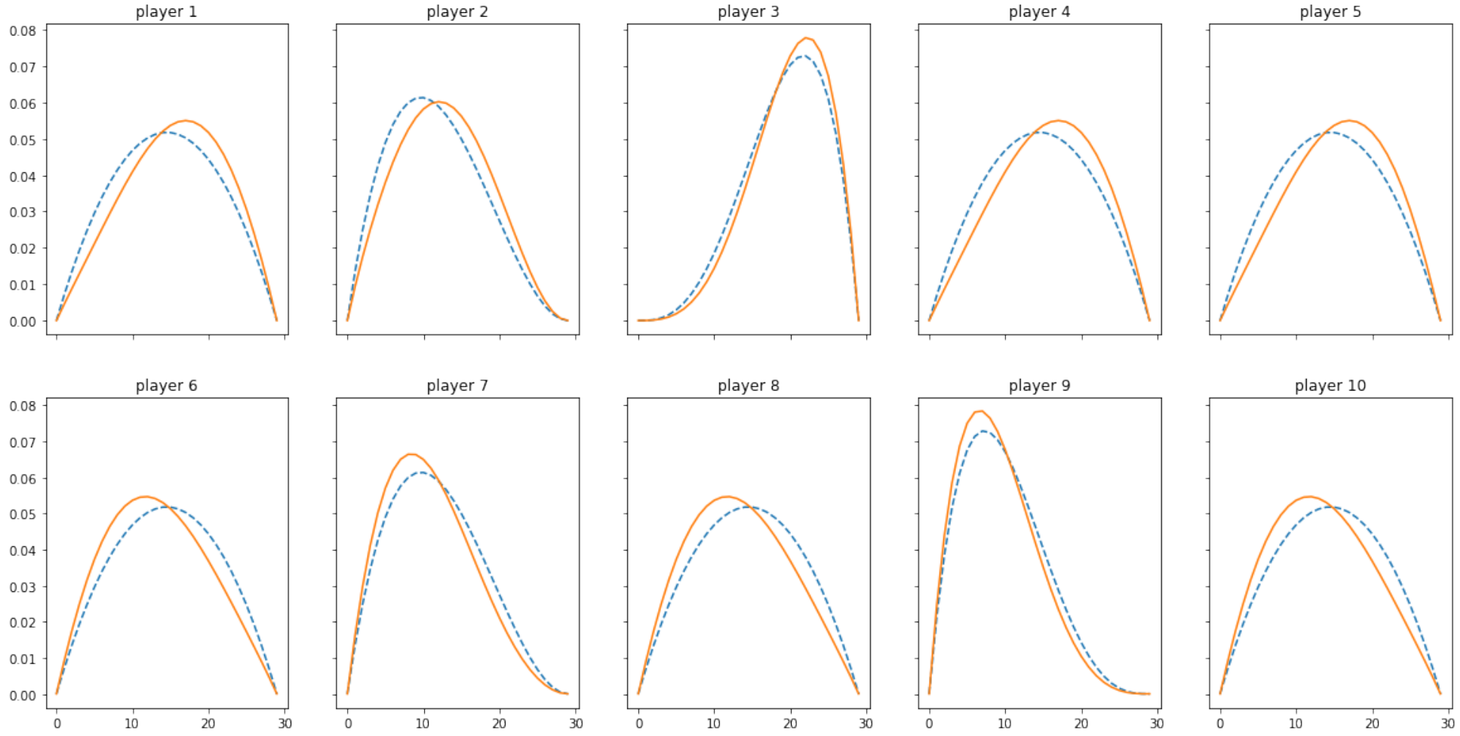
Any ML algorithm that can be updated via $p(\theta|D) \propto \Pi_i p(d_i|\theta)p(\theta)$ is automatically a streaming algorithm for ML because it forces the recursive relation;

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

laymans terms: come up with a sensible update rule for a distribution and 'yer done!

bayesians to the rescue

You could even update it for teamplay.



bayesians to the rescue

Here's the thing: **the model is a mere histogram**.

And, oh the benefits:

— we quantify our uncertainty
— we can apply the model in a streaming setting
— it's very easy to deploy/understand/debug/test

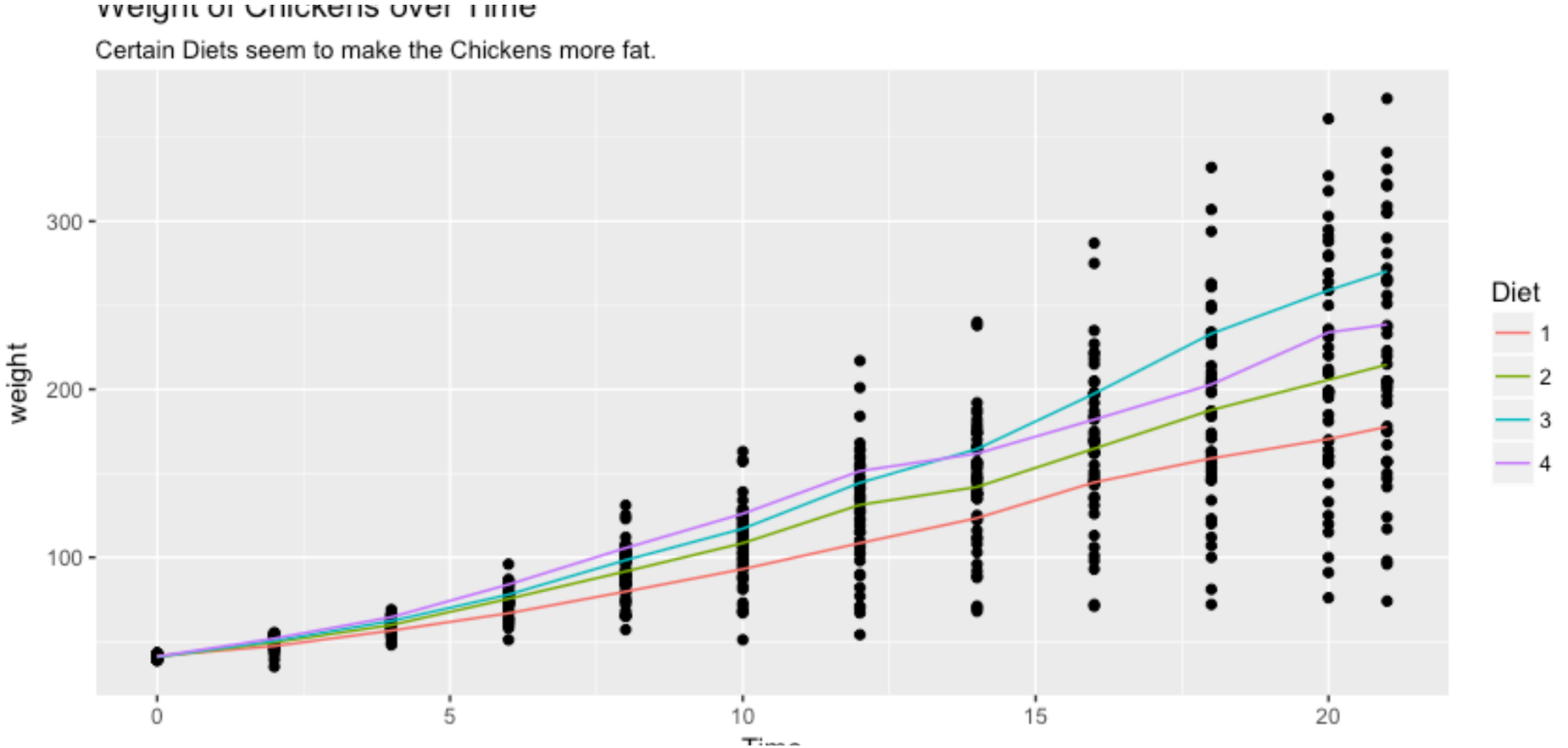
If you want to come up with such a model, taking a step back from hype can really help.

One Last Example: Chickens

Suppose that I have a dataset with chickens.

	weight	Time	Chick	Diet
1	42	0	1	1
2	51	2	1	1
3	59	4	1	1
...				
576	234	18	50	4
577	264	20	50	4
578	264	21	50	4

One Last Example: Chickens

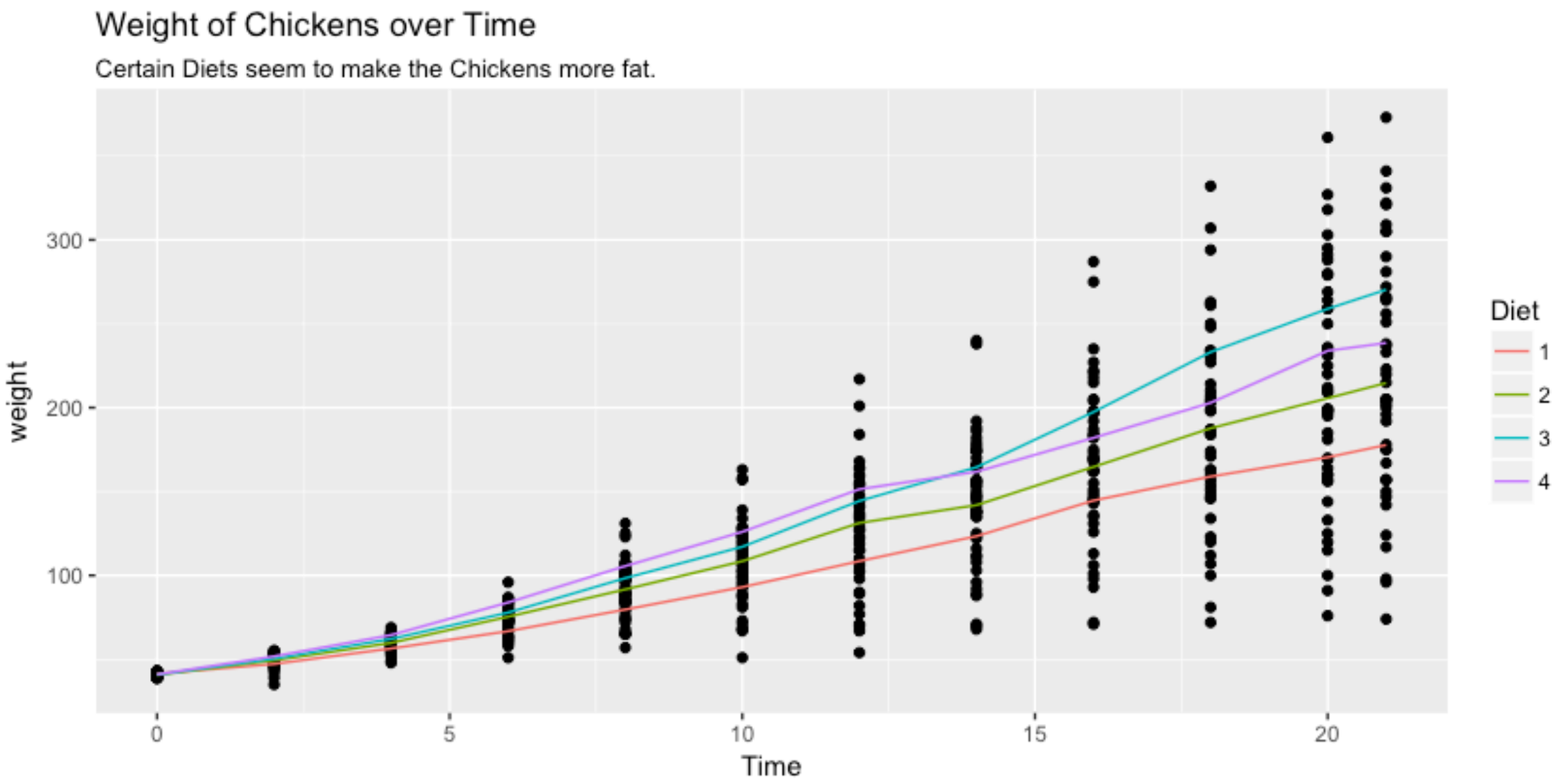


Model 1: Base Regression

We could model it with a linear regression (R).

```
> model <- lm(weight ~ Time + Diet, data=chickweight)
> model %>% summary()
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  10.9244     3.3607   3.251  0.00122 **
Time          8.7505     0.2218  39.451  < 2e-16 ***
Diet2        16.1661     4.0858   3.957 8.56e-05 ***
Diet3        36.4994     4.0858   8.933  < 2e-16 ***
Diet4        30.2335     4.1075   7.361 6.39e-13 ***
```

No matter what backend you use, the model is **all wrong**.


Weight of Chickens over Time
Certain Diets seem to make the Chickens more fat.
weight
300
200
100
0
5
10
15
20
Time
Diet
1
2
3
4

Model 2: R-trick: Nested Regression

```
> chickweight %>%
    group_by(Diet) %>%
    nest() %>%
    mutate(mod = data %>% map(~ lm(weight ~ Time, data=.)))

  Diet              data      mod
<fctr>            <list>   <list>
1      1 <tibble [220 x 3]> <S3: lm>
2      2 <tibble [120 x 3]> <S3: lm>
3      3 <tibble [120 x 3]> <S3: lm>
4      4 <tibble [118 x 3]> <S3: lm>
```

Better, but this is still **wrong**.

The problem

In Machine Learning if feels like we can pour data into a predefined model but it doesn't feel like we can define the model much.

We're usually constrained to perhaps feature engineering and hyperparam tuning (which granted, is good enough for lots of problems).

The problem

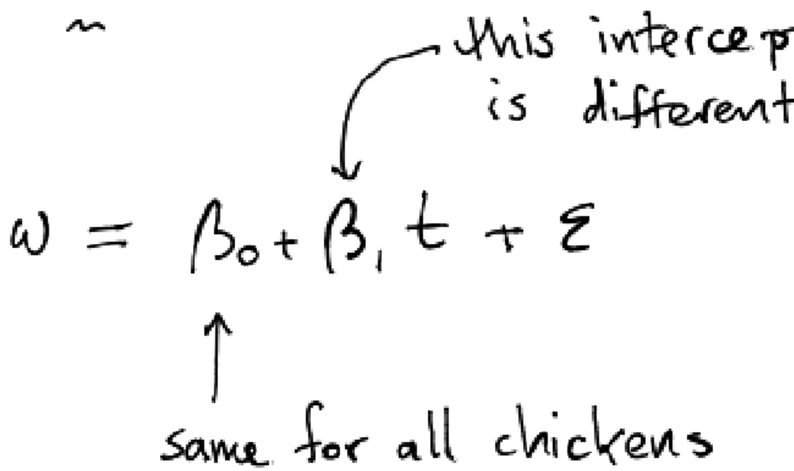
In Machine Learning if feels like we can pour data into a predefined model but it doesn't feel like we can define the model much.

We're usually constrained to perhaps feature engineering and hyperparam tuning (which granted, is good enough for lots of problems).

Popular ML libraries don't offer a real DSL for models.

Model 3: Domain Model

I wrote what I want on a piece of paper:

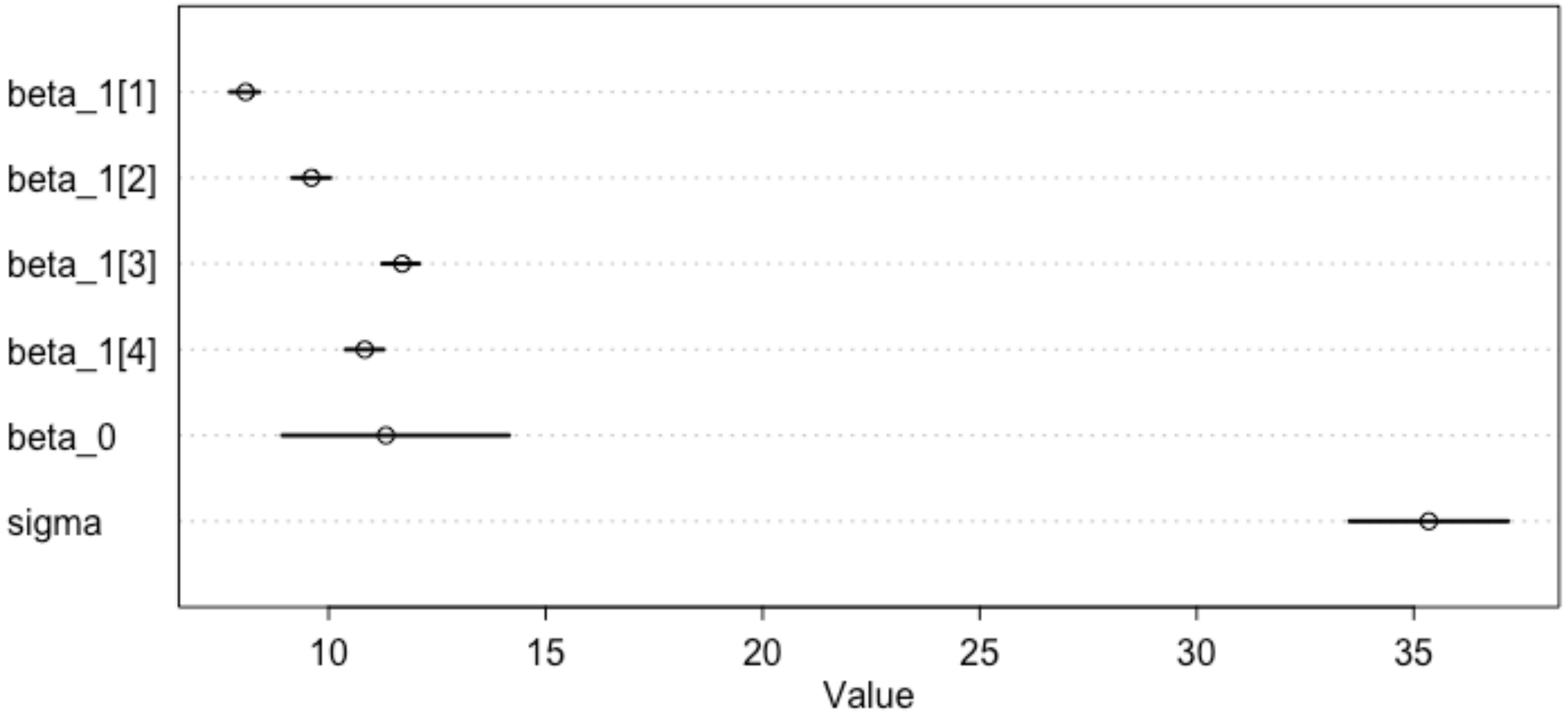


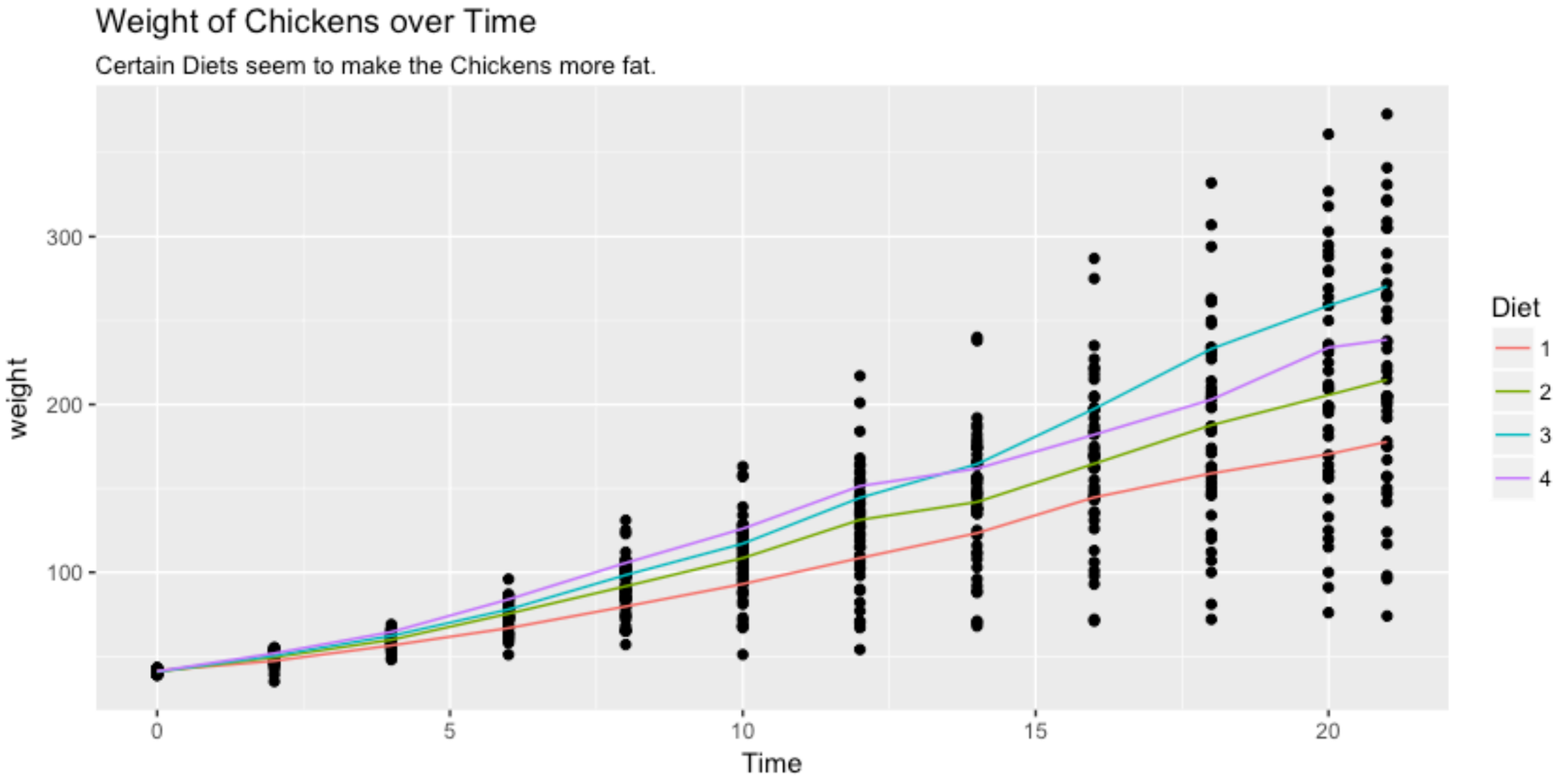
I want to basically try this, in a few lines of code.

Model 3: Domain Model `rethinking`

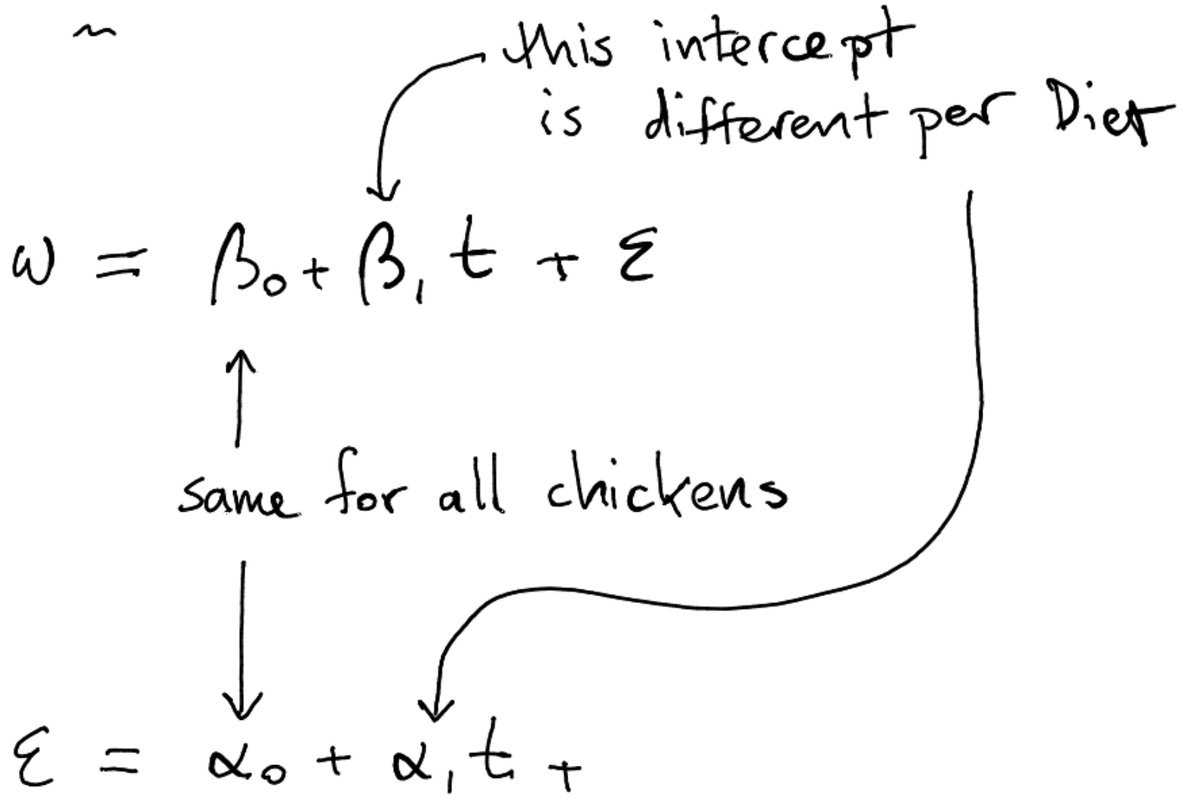
```
mod <- map2stan(
    alist(
        weight ~ dnorm(mu, sigma),
        mu <- intercept + slope[Diet]*Time,
        slope[Diet] ~ dnorm(0, 2),
        intercept ~ dnorm(0, 2),
        sigma ~ dunif(0, 10)
    ), data = ml_df, warmup = 500)
```

Model 3: Domain Model `rethinking`




Weight of Chickens over Time
Certain Diets seem to make the Chickens more fat.
weight
300
200
100
0
5
10
15
20
Time
Diet
1
2
3
4

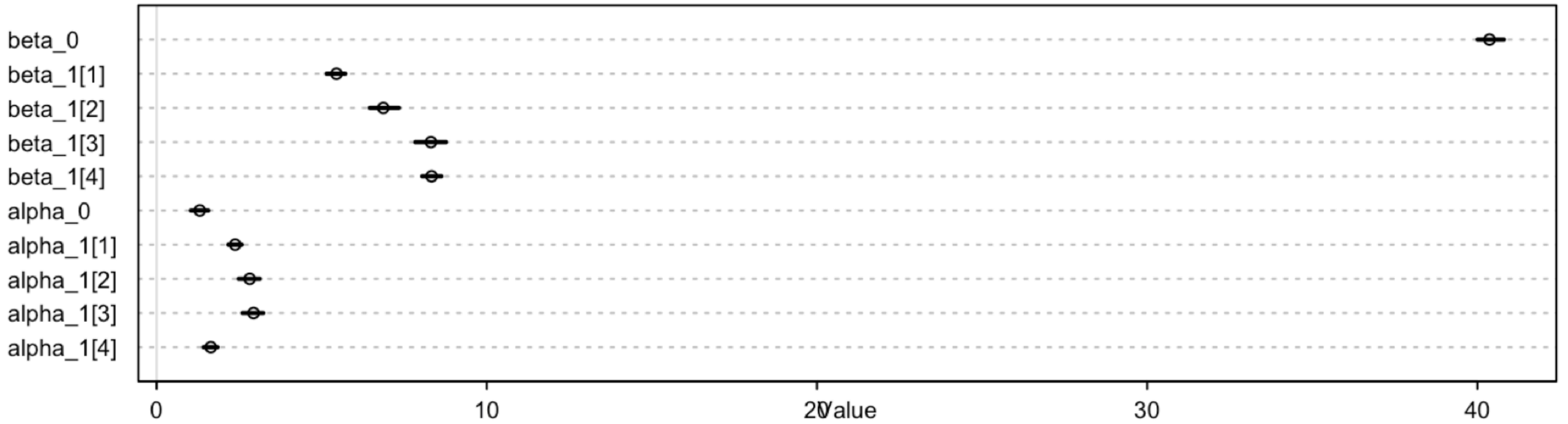
Model 3: Domain Model `rethinking`



Model 3: Domain Model `rethinking`

```
mod <- map2stan(
    alist(
        weight ~ dnorm(mu, sigma),
        mu <- beta_0 + beta_1[Diet]*Time,
        beta_0 ~ dnorm(0, 2),
        beta_1[Diet] ~ dnorm(0, 2),
        sigma <- alpha_0 + alpha_1[Diet]*Time,
        alpha_0 ~ dunif(0, 10),
        alpha_1[Diet] ~ dunif(0, 10)
    ), data = ml_df, warmup = 500)
```

Model 3: Domain Model `rethinking`

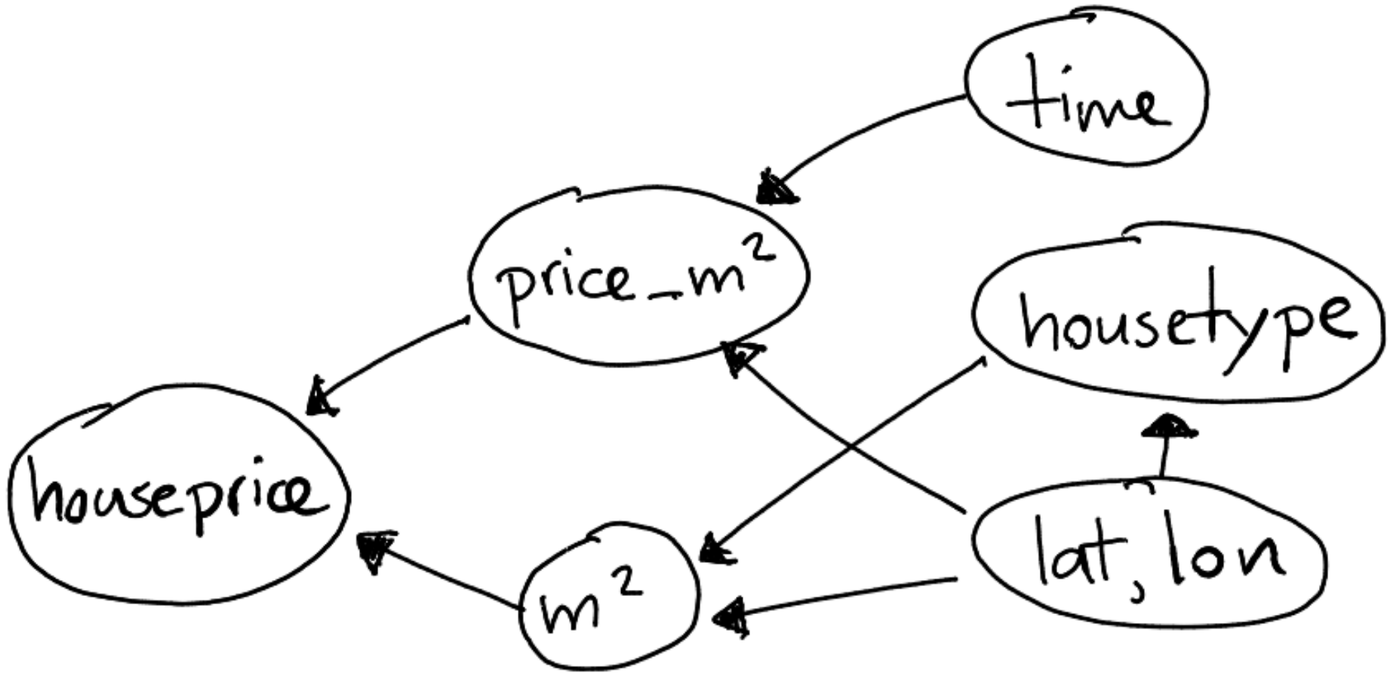


Freedom

There's something very precious happening here. Instead of modelling in the feature space we can also keep models simple by modelling in the model space. All sorts of features are automatically generated by this DSL that make creative modelling rather convenient.

The model I've just defined can assign (un)certainty to each prediction. And since it is generative I could also input the weight of the chicken and infer the diet!

It's great to be able to model the model generatively.



Freedom

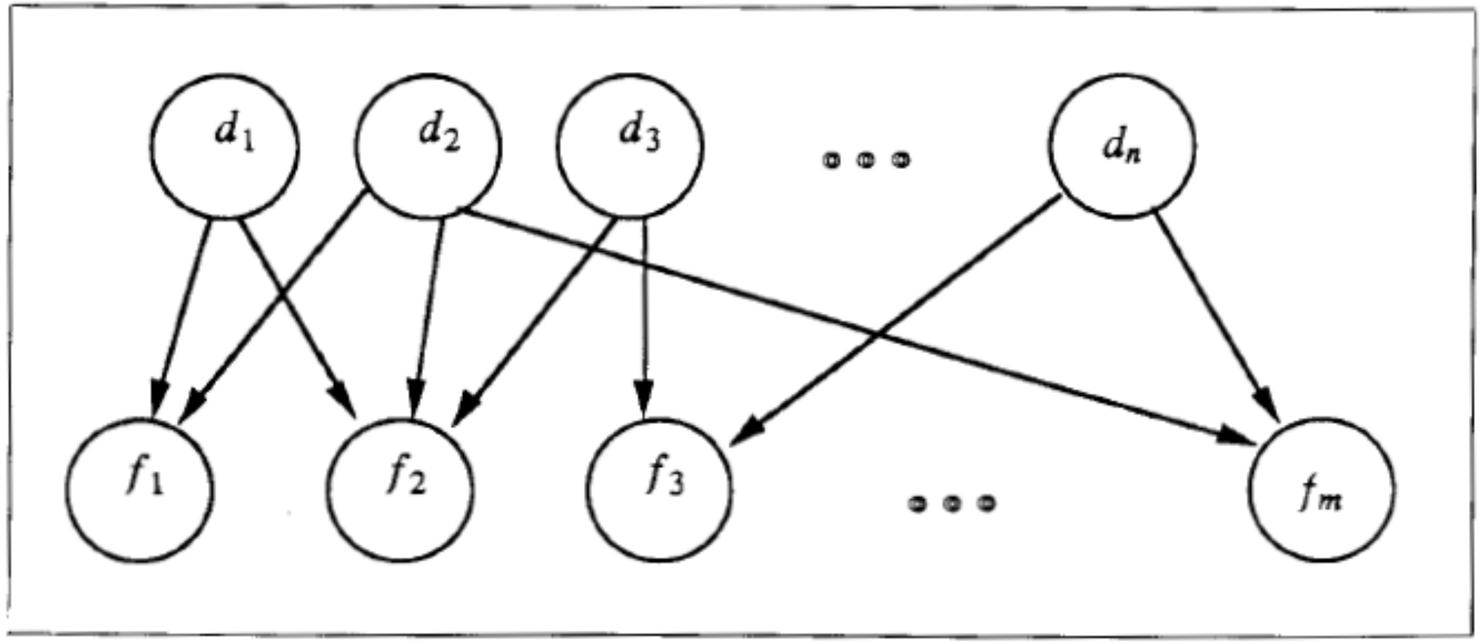


Fig. 1 The two-level belief-network representation of the current QMR-DT KB. The disease nodes are labeled $d_1, \ldots, d_n$ and the finding nodes are labeled $f_1, \ldots, f_m$. The probabilistic dependencies between diseases and findings are specified with directed arcs between nodes, where an arc points in the causal direction that we assume; that is, we assume that diseases cause findings.

I could do a whole talk on just this topic but I'll keep it brief

It's great to be able to model the model instead of pooring the data into a standardized cast-mold.

Python tools like **pomegrenate**, **pymc3** and **edward** may be a nice place to invest some knowledge in if these sorts of models sound like things you'd like to play with.

If this fancies your interest, there's a cool book being written by Bishop, preview here.

Conclusion Time

Conclusion

We saw how;

— feature engineering can still save the day
— it makes sense to come up with systems instead merely applying algorithms
— simple algorithms can have properties that complex algorithms are missing out on
— we could care about more than just "error in test set" if we want to provide a service

Conclusion

Simple models can actually be rather advanced/smart, complex models can be rather inarticulate/dumb.

Unfortuntely there seems to be a fear of missing out and I see people looking for excuses to use **DeepModels[tm]**. This might be a dangerous pursuit.

There are some problems that require deep learning/ forest methods. Most problems should be tackled with simpler models first though. Let's celebrate this!

Thanks for letting me speak!

Simple models are easier to;

— understand
— explain
— debug
— maintain
— adapt

Let's use 'em to solve some problems. Questions?