Week 5. Linear models & Gradient Boosting

Linear models

Online learning

Online learning – the setup when data arrives in a stream and you want to adjust the model after each new sample

Usages:

- The entire dataset doesn't fit in memory
- You need to adapt quickly to new patterns in the data

https://en.wikipedia.org/wiki/Online_machine_learning#Statistical_learning_models

In the case of linear regression

- Analytical solution:

$$w^* = (X^T X)^{-1} X^T Y$$

- Can be updated after every new sample:

$$w_0 = 0 \in \mathbb{R}^d \qquad \Gamma_i = \Gamma_{i-1} - \frac{\Gamma_{i-1} x_i x_i^T \Gamma_{i-1}}{1 + x_i^T \Gamma_{i-1} x_i}$$

$$\Gamma_0 = I \in \mathbb{R}^{d \times d} \qquad w_i = w_{i-1} - \Gamma_i x_i (x_i^T w_{i-1} - y_i)$$

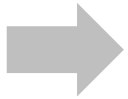
- Stochastic gradient descent (SGD):

Similar, huh?

$$w_i = w_{i-1} - \gamma_i x_i (x_i^T w_{i-1} - y_i)$$

Spam filtering: bag of words vectorization

good movie
not a good movie
did not like



good	**movie**	**not**	**a**	**did**	**like**
1	**1**	0	0	0	0
1	**1**	**1**	1	0	0
0	0	**1**	0	1	**1**

Can be classified with linear model!

Spam filtering: mapping n-grams to feature indices

If your dataset is small you can store {n-gram → feature index} in hash map.

But if you have a huge dataset that can be a problem

- Let's say we have 1 TB of texts distributed over 10 nodes
- You need to vectorize each text

- You will have to maintain {n-gram → feature index} mapping
  - May not fit in memory on one machine
  - Hard to synchronize

Spam filtering: mapping n-grams to feature indices

If your dataset is small you can store {n-gram → feature index} in hash map.

But if you have a huge dataset that can be a problem

- Let's say we have 1 TB of texts distributed over 10 nodes
- You need to vectorize each text

- An easier way is hashing: {n-gram → hash(n-gram) % 2^{22}}
  - Has collisions but works in practice
  - sklearn.feature_extraction.text.**HashingVectorizer**
  - Implemented in **vowpal wabbit** library (later on that)

Spam filtering is a huge task

- **Spam filtering proprietary dataset**
  - https://arxiv.org/pdf/0902.2206.pdf
  - 0.4 million users
  - 3.2 million letters
  - 40 million unique words

- **Let's say we map each token to index using hash function ϕ**
  - $\phi(x) = \text{hash}(x) \,\%\, 2^b$
  - For $b = 22$ we have 4 million features
  - That is a huge improvement over 40 million features
  - It turns out it doesn't hurt the quality of the model

Hashing vectorizer

- $\phi(good) = 0$
- $\phi(movie) = 1$
- $\phi(not) = 2$
- $\boldsymbol{\phi(a) = 3}$
- $\boldsymbol{\phi(did) = 3}$
- $\phi(like) = 4$

Hash collision

$$\mathbf{hash}(s) = s[0] + s[1]p^1 + \cdots + s[n]p^n$$

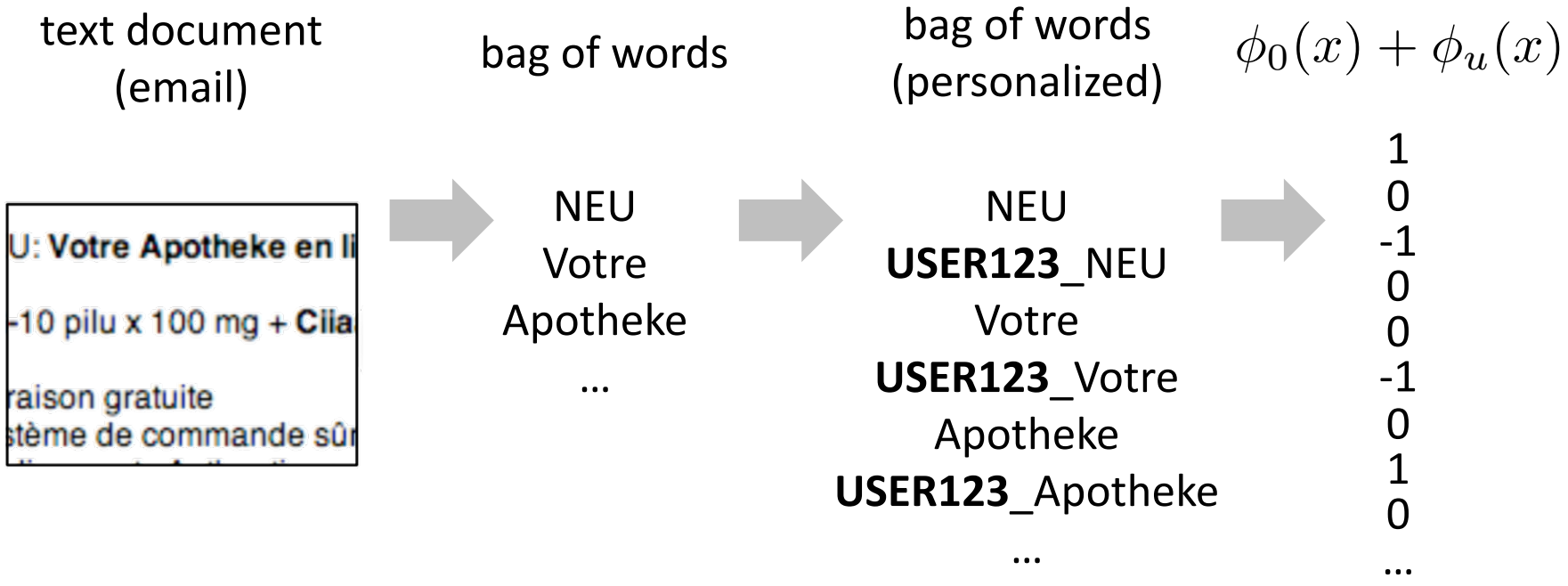
s – string
p – fixed prime number
$s[i]$ – character code

good movie
not a good movie
did not like

0	1	2	3	4
1	**1**	0	0	0
1	**1**	**1**	1	0
0	0	**1**	1	1

Trillion features with hashing

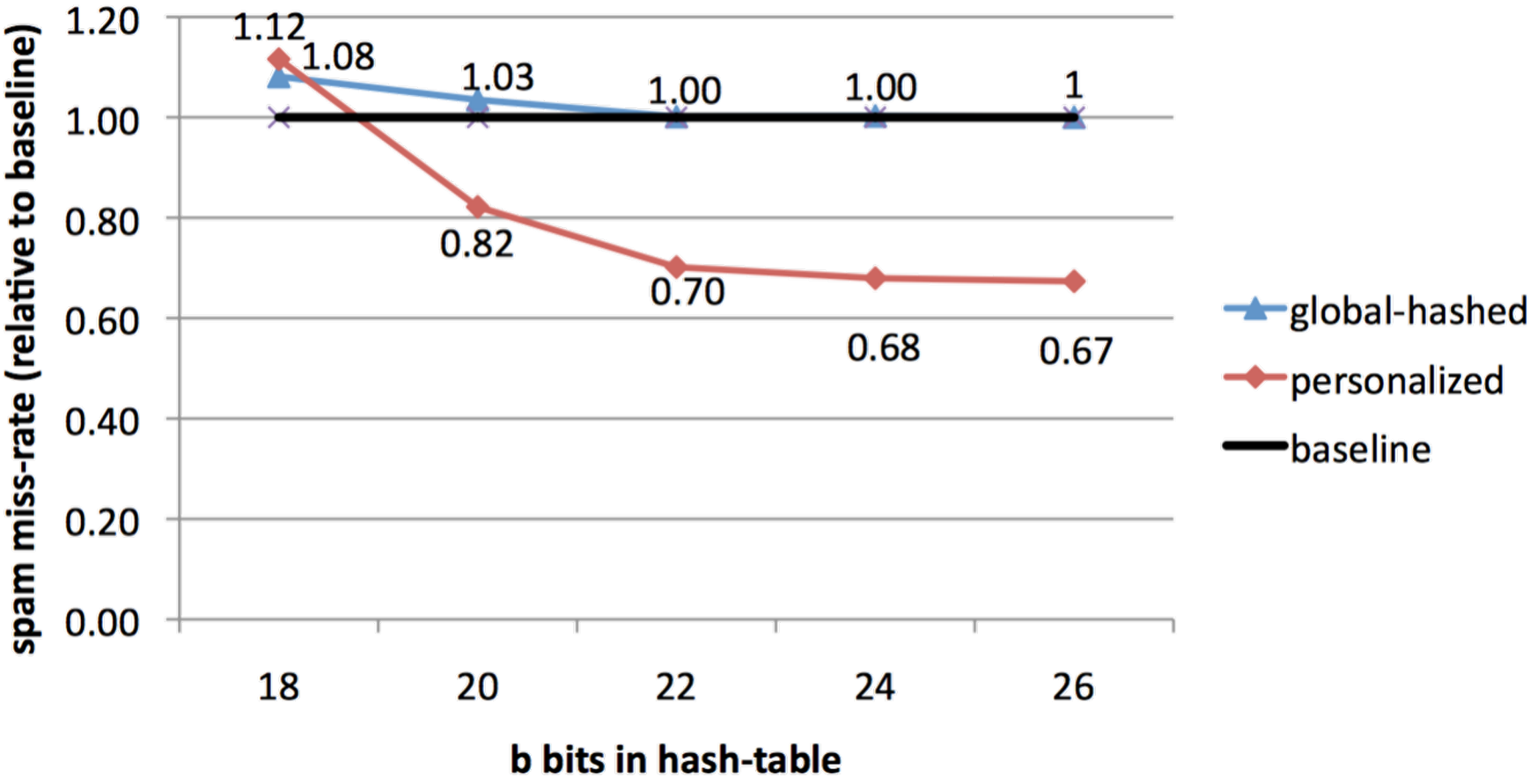
- **Personalized tokens trick**
  - $\phi_o(token) = \text{hash}(token)\ \%\ 2^b$
  - $\phi_u(token) = \text{hash}(u + "\_" + token)\ \%\ 2^b$
  - We obtain 16 trillion pairs (user, word) but still 2^b features



https://arxiv.org/pdf/0902.2206.pdf

Experimental results

- For $b = 22$ it performs just like a linear model on original tokens
- We observe that personalized tokens give a huge improvement in miss-rate!

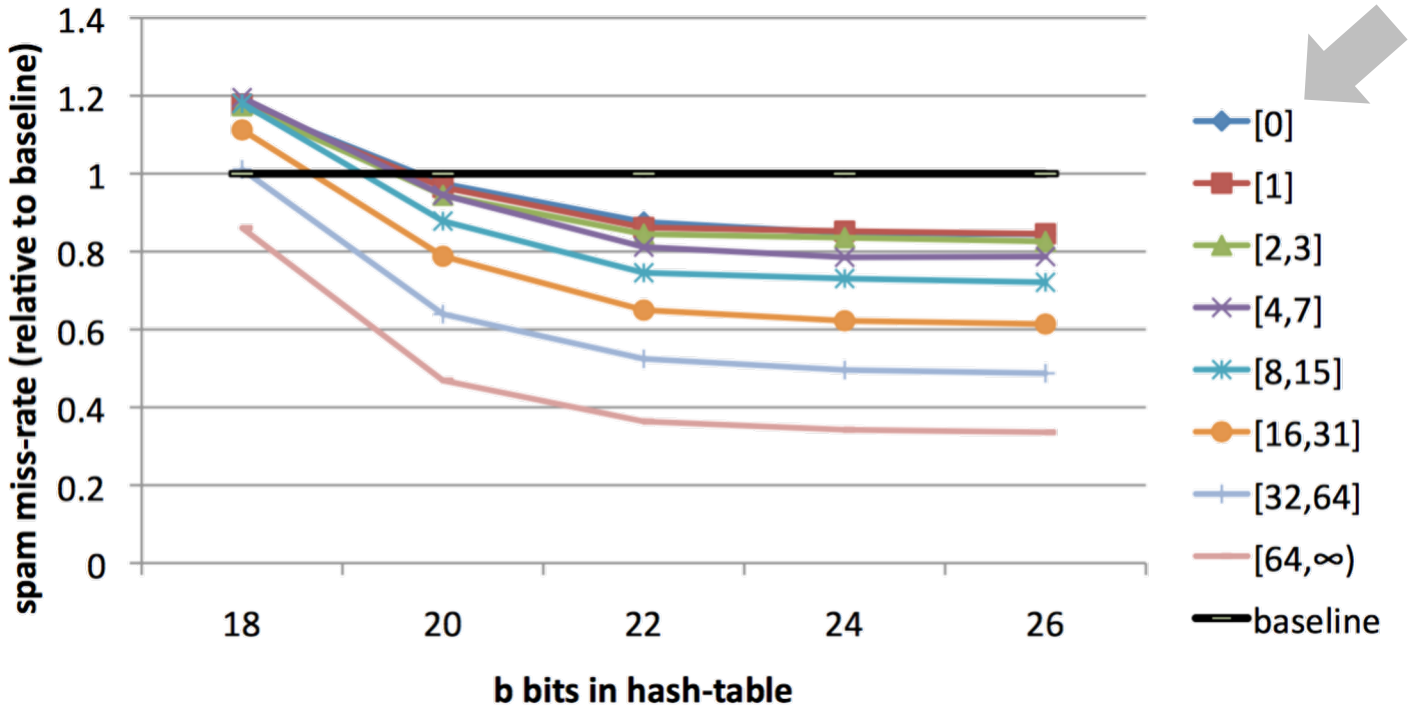


Why personalized features work

Personalized features capture "local" user-specific preference

- Some users might consider newsletters a spam but for the majority of the people they are fine

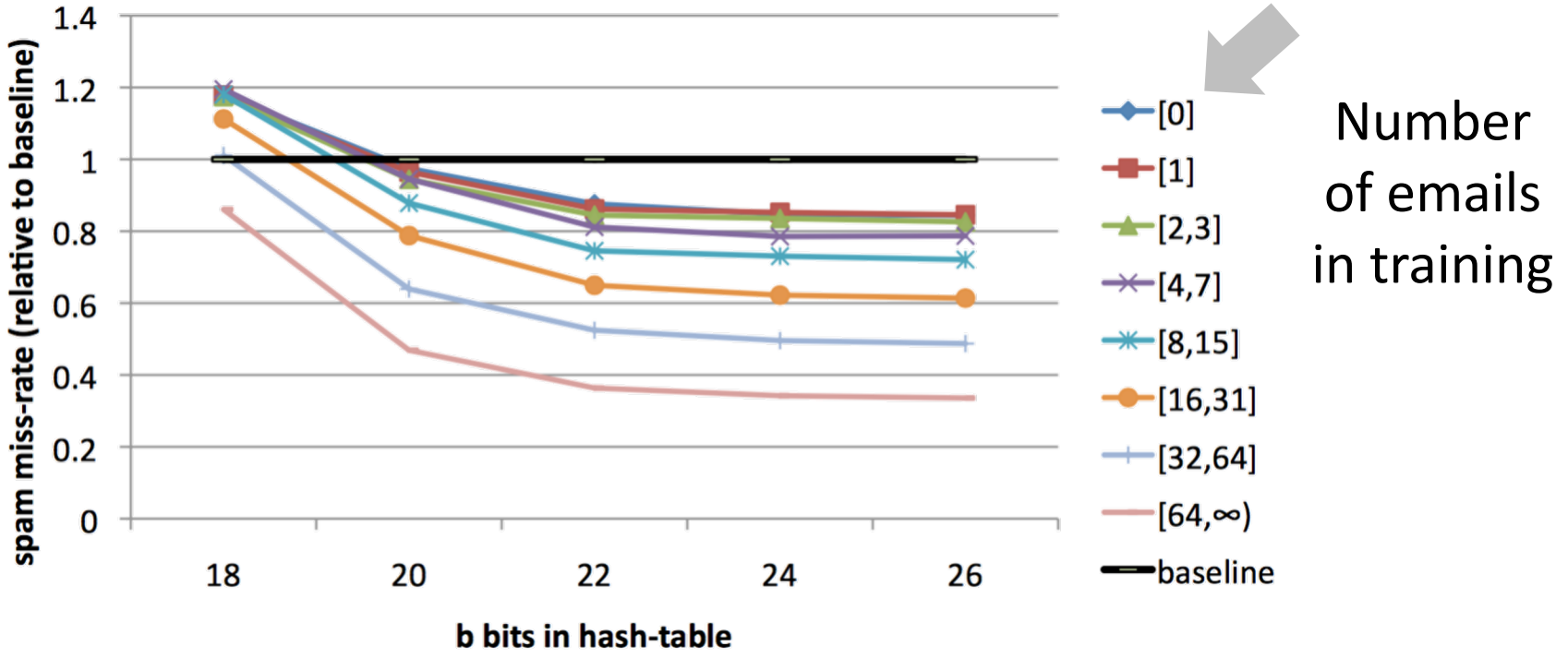
How will it work for new users?



Why personalized features work

It turns out we learn better "global" preference having personalized features which learn "local" user preference

- You can think of it as a more universal definition of spam



Vowpal Wabbit (VW)

- Popular implementation of online learning algorithms
- Not only linear models (paired interactions, neural networks)
- Lots of loss functions
- Normal line of text is a valid description of an object

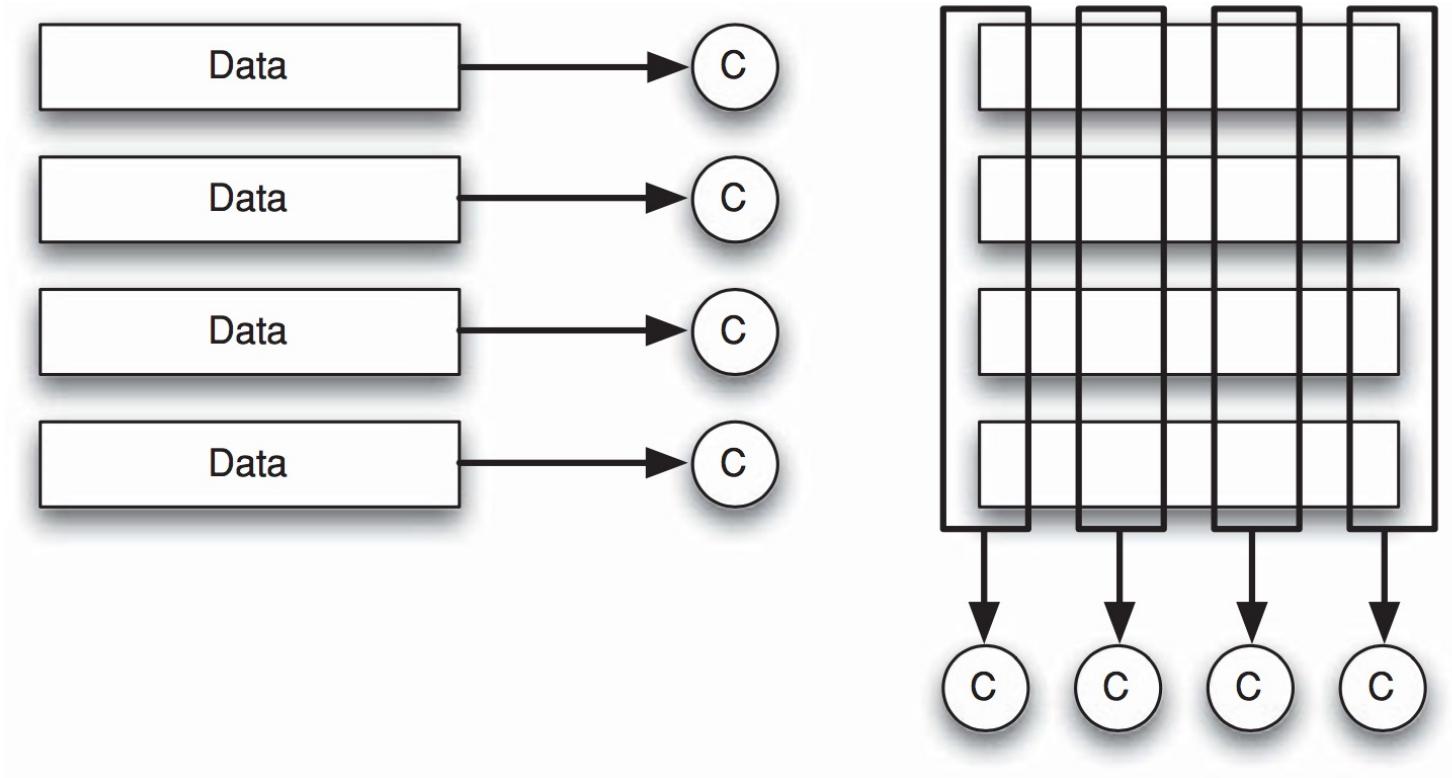
  1 | The dog ate my homework

  Every word will be hashed to get feature index

- Effectively scales to 1000 nodes using AllReduce (later on that)

https://github.com/JohnLangford/vowpal_wabbit/wiki/Tutorial http://people.cs.umass.edu/~mcgregor/stocworkshop/langford.pdf

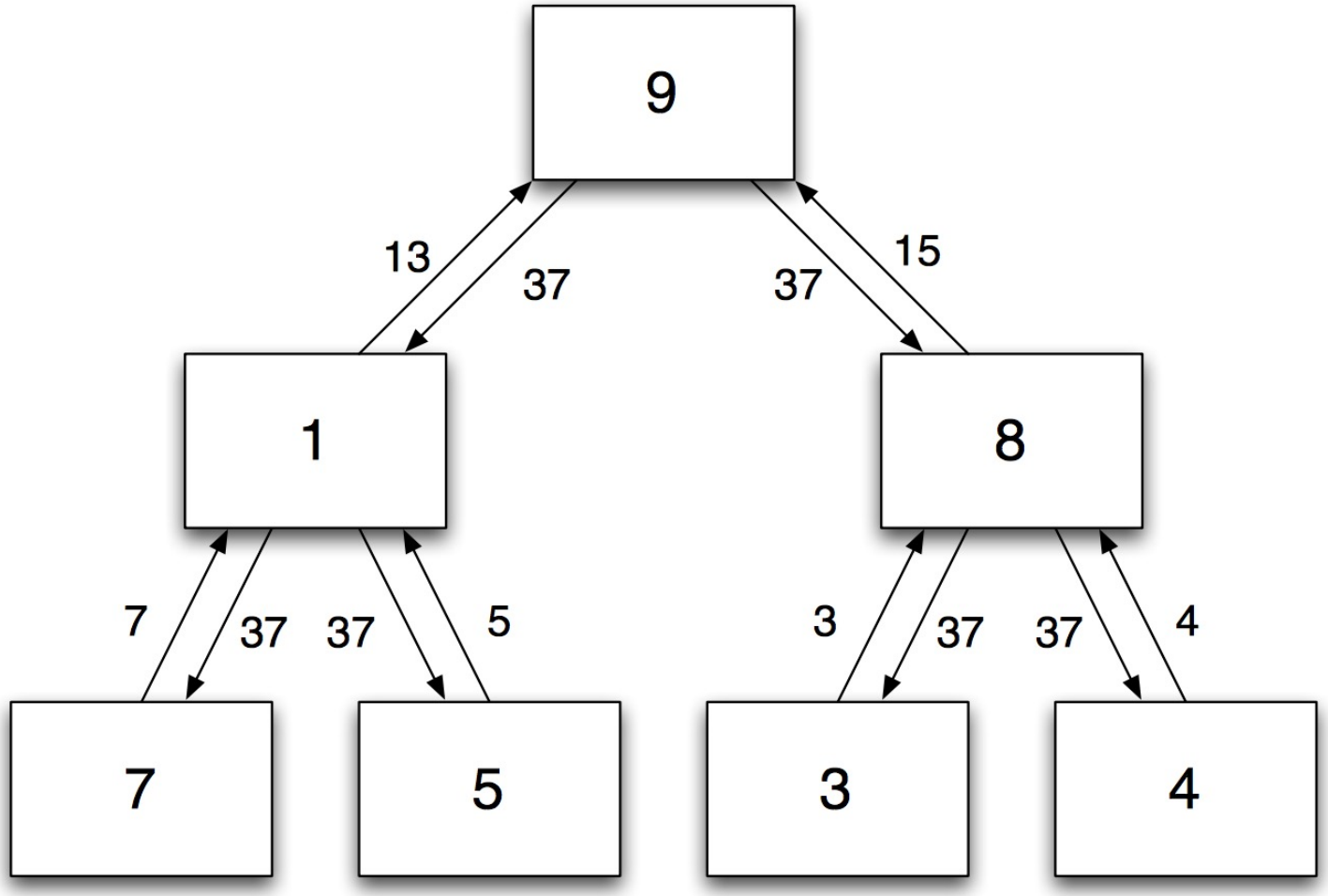
How to scale VW



Divide objects

Divide features

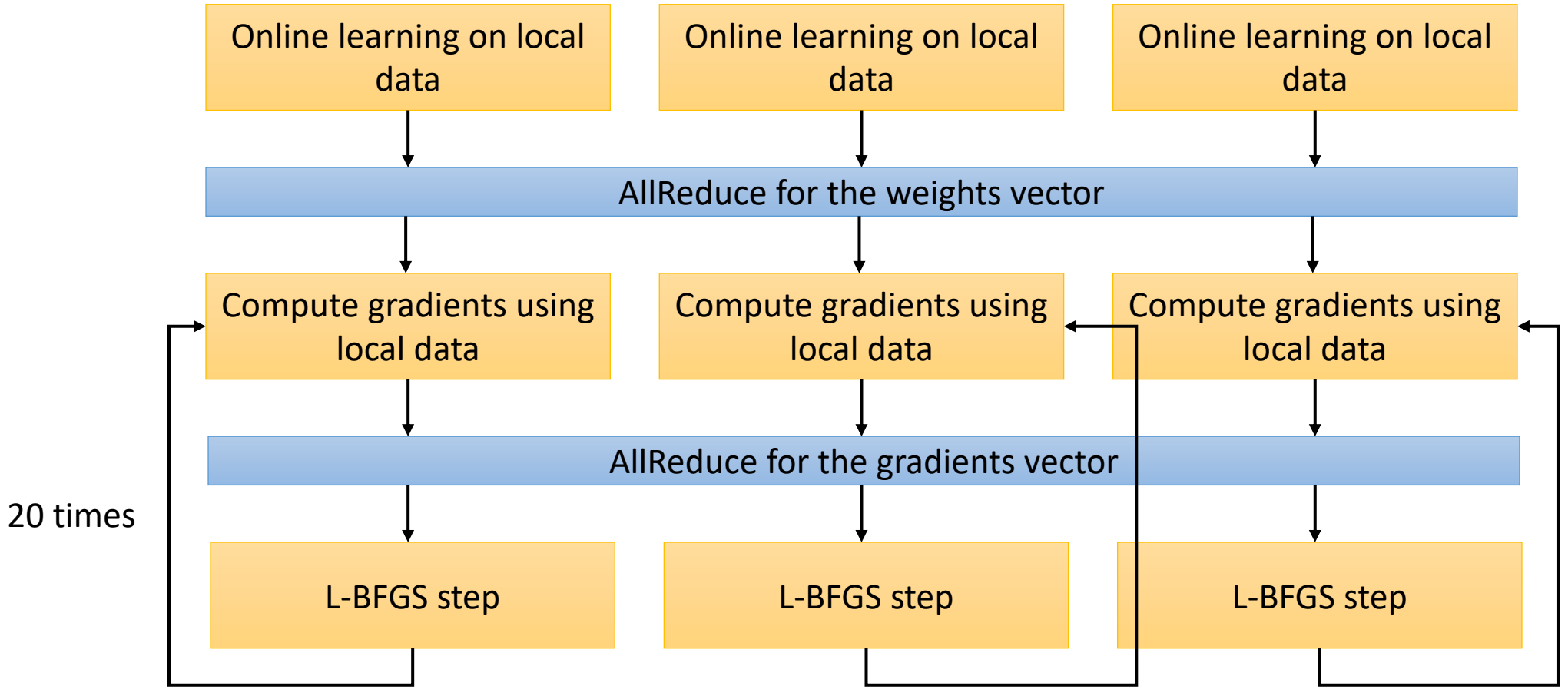
AllReduce in VW



- Each node computes a vector (gradients or weights) using local data
- Every node wants to get the sum of these vectors from all the nodes
- Good network utilization due to pipelining

https://arxiv.org/pdf/1110.4198.pdf

Full-batch training on 3 nodes



http://people.cs.umass.edu/~mcgregor/stocworkshop/langford.pdf https://arxiv.org/pdf/1110.4198.pdf

Online learning on local data

- Map tasks in Hadoop
- SGD with adaptive gradient update:

Require: Invariance update function s
(see Karampatziakis and Langford, 2011)
$\mathbf{w} = \mathbf{0}$, $\mathbf{G} = \mathbf{I}$
for all $(\mathbf{x}, y)$ in training set **do**
$\mathbf{g} \leftarrow \nabla_{\mathbf{w}} \ell(\mathbf{w}^\top \mathbf{x}; y)$
$\mathbf{w} \leftarrow \mathbf{w} - s(\mathbf{w}, \mathbf{x}, y)\mathbf{G}^{-1/2}\mathbf{g}$
$G_{jj} \leftarrow G_{jj} + g_j^2$ for all $j = 1, \ldots, d$
end for

AllReduce for the weights vector

- Better way to aggregate weights:

$$\bar{\mathbf{w}} = \left(\sum_{k=1}^{m} \mathbf{G}^k \right)^{-1} \left(\sum_{k=1}^{m} \mathbf{G}^k \mathbf{w}^k \right)$$

- G^k — diagonal matrix (diagonal of the same size as w^k)
- Requires 2 AllReduce operations, works much faster than MapReduce:

	Full size	10% sample
MapReduce	1690	1322
AllReduce	670	59

Why the size matters

Why do we need such huge datasets?

- It turns out you can learn better models using the same simple linear classifier

Ad click prediction

- https://arxiv.org/pdf/1110.4198.pdf
- Trillions of features, billions of training examples
- Data sampling hurts the model

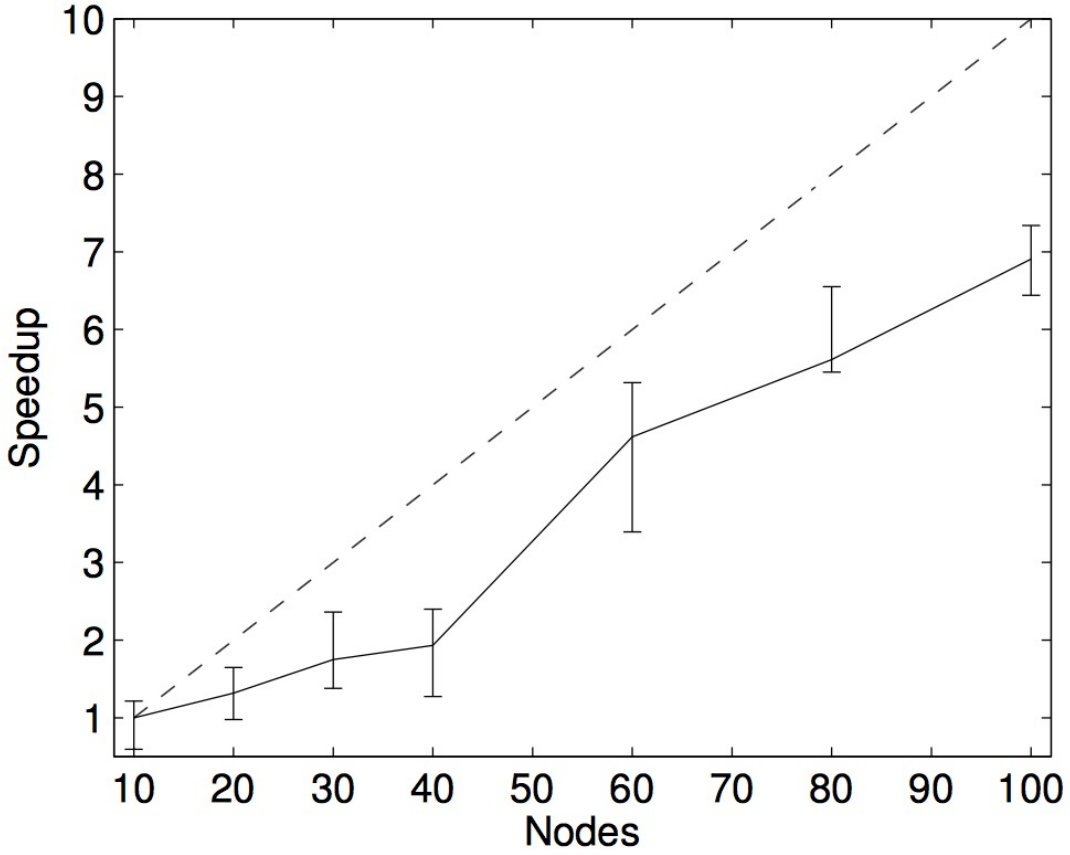
	1%	10%	100%
auROC	0.8178	0.8301	0.8344
auPRC	0.4505	0.4753	0.4856
NLL	0.2654	0.2582	0.2554

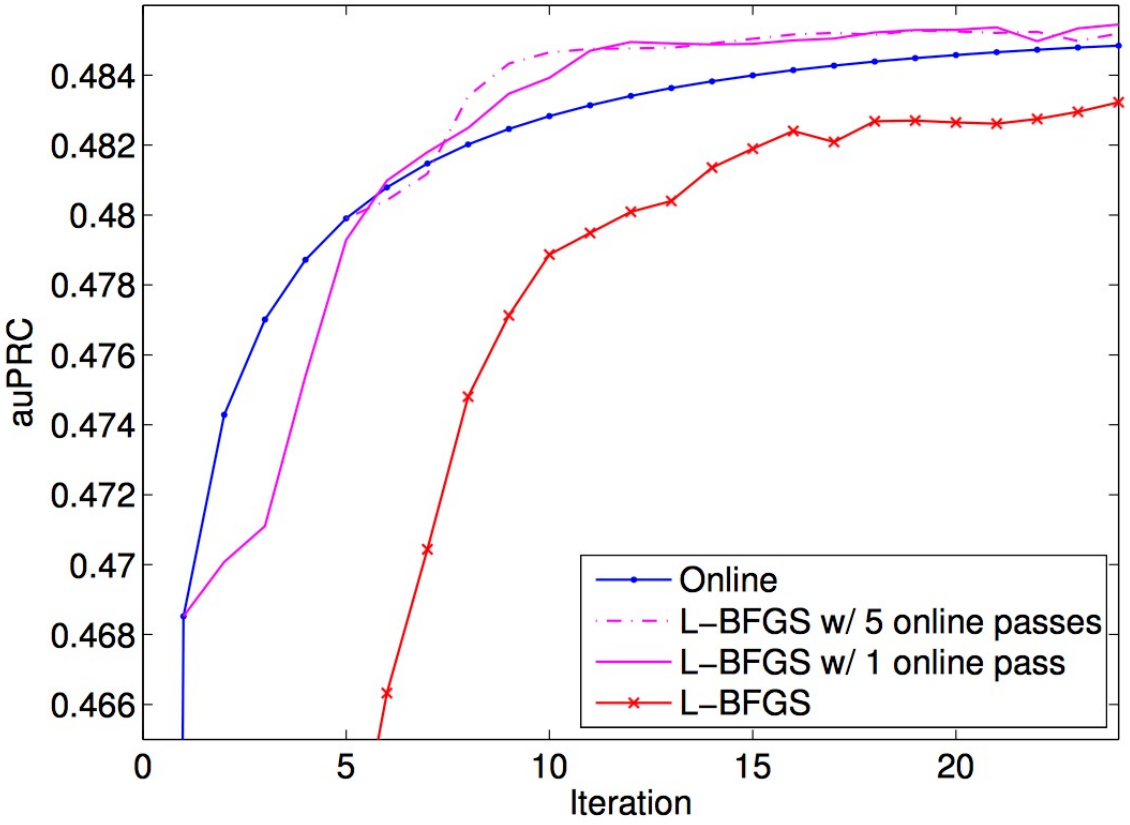
Sampling rate

Ad click prediction

2.1T features, **17B** samples, **16M** parameters (24 hash bits)

Optimal linear classifier in 70 minutes on 1000 nodes





http://fastml.com/vowpal-wabbit-eats-big-data-from-the-criteo-competition-for-breakfast/

https://arxiv.org/pdf/1110.4198.pdf

VW model error reporting

- Standard way: training on train set, testing on test set

- With single pass over data you can have **progressive validation**:
  - For each new sample an error is computed before the model update
  - The average (exponentially weighted) of such errors is displayed
  - It's been proved to be close to the error on the test set

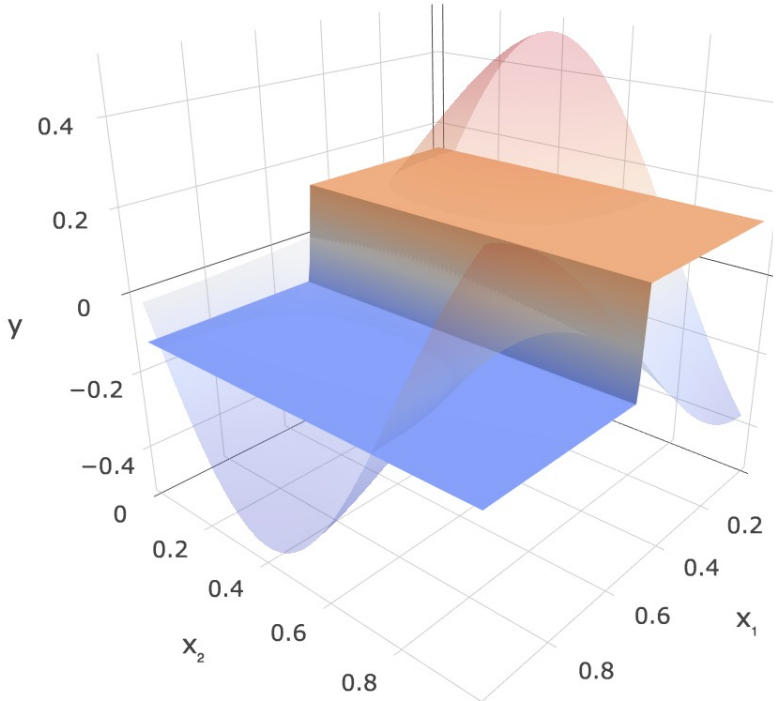
- With the second pass over data it won't make sense
  - So VW holds out 10% of data if you configure more than one pass over data
  - You can turn it off with --holdout_off

http://cilvr.cs.nyu.edu/diglib/lsml/lecture01-online-linear.pdf http://hunch.net/~jl/projects/prediction_bounds/progressive_validation/coltfinal.pdf

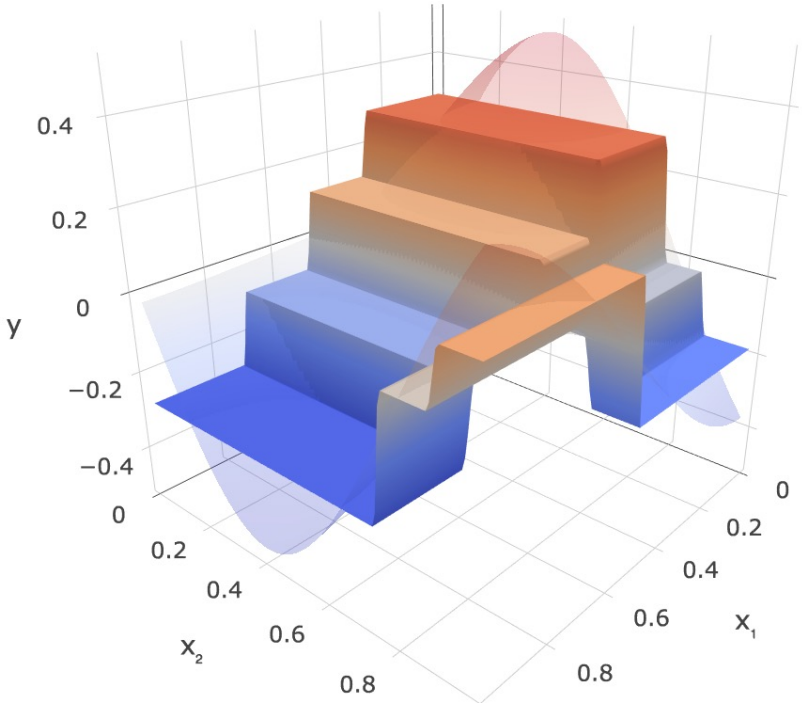
Gradient Boosting

One decision tree

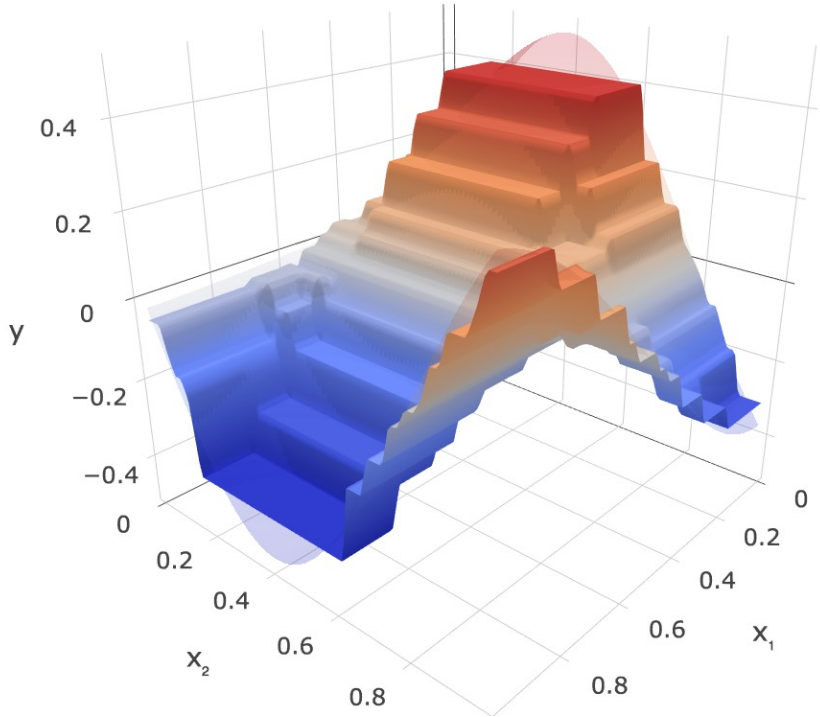
Depth 1



Depth 3



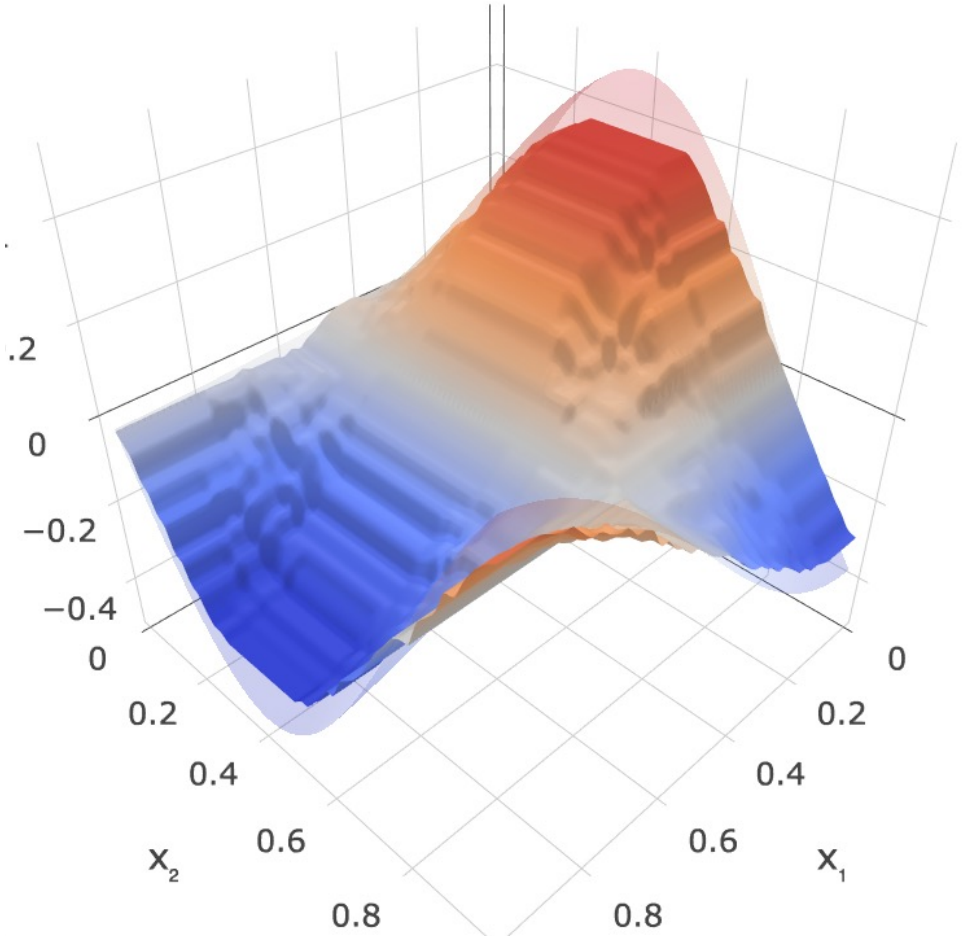
Depth 6



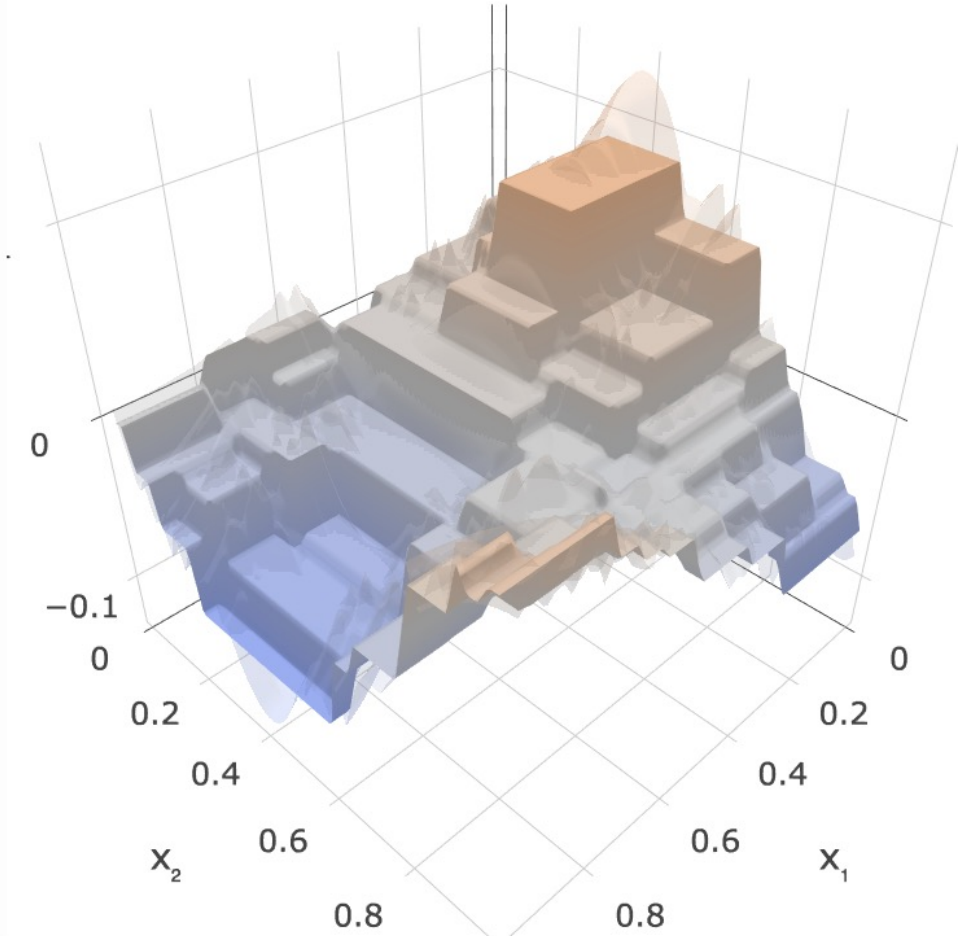
http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html

Next tree approximates the residuals

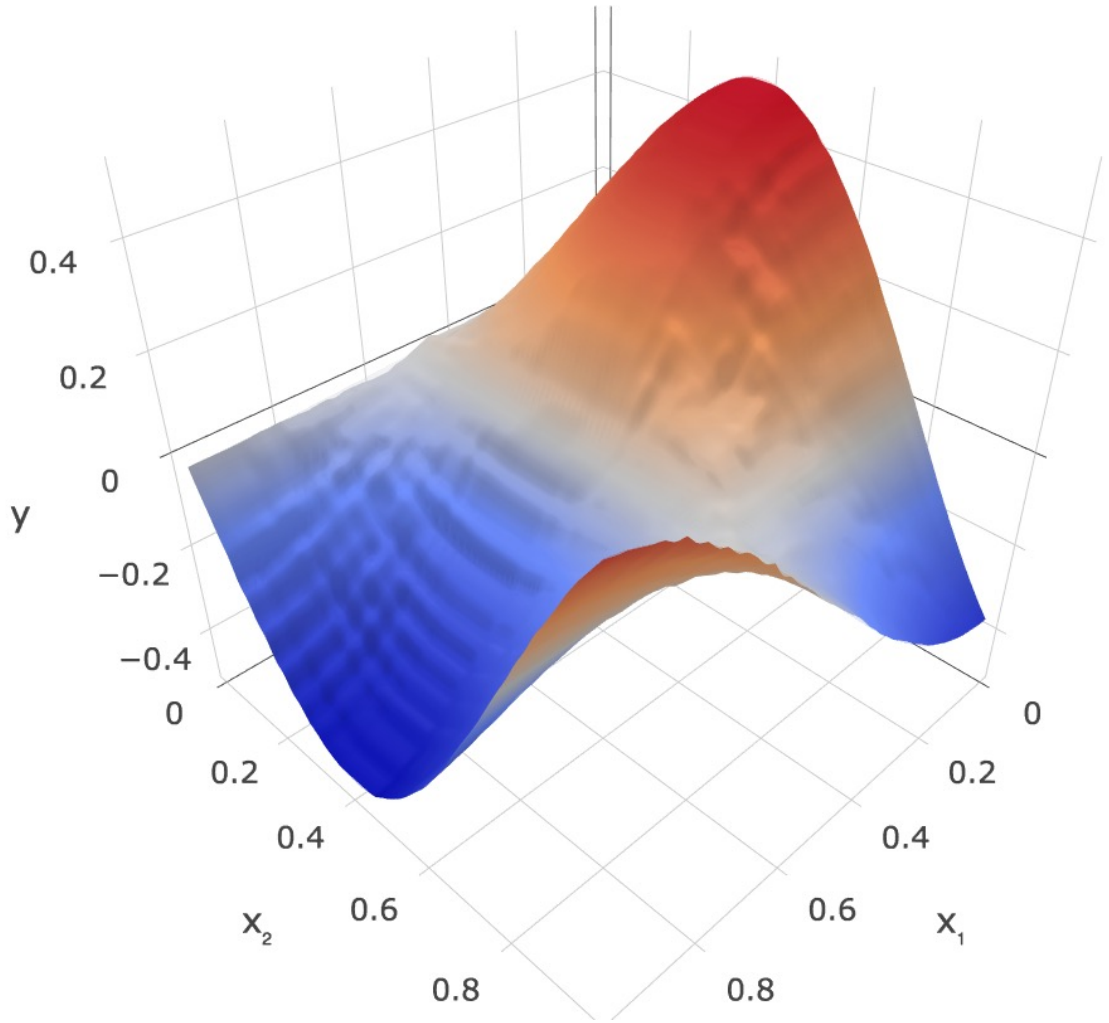
Current ensemble



Residuals

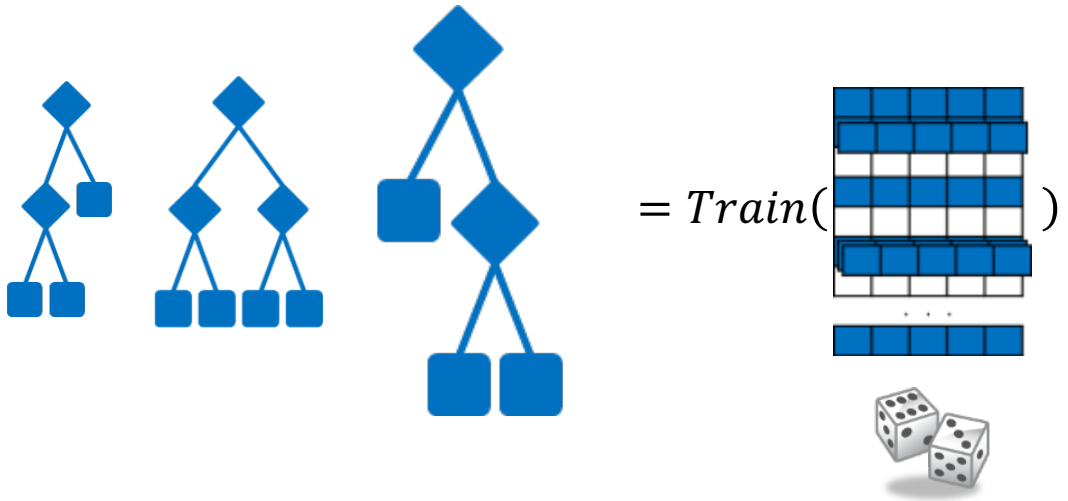


Gradient Boosting: 100 trees

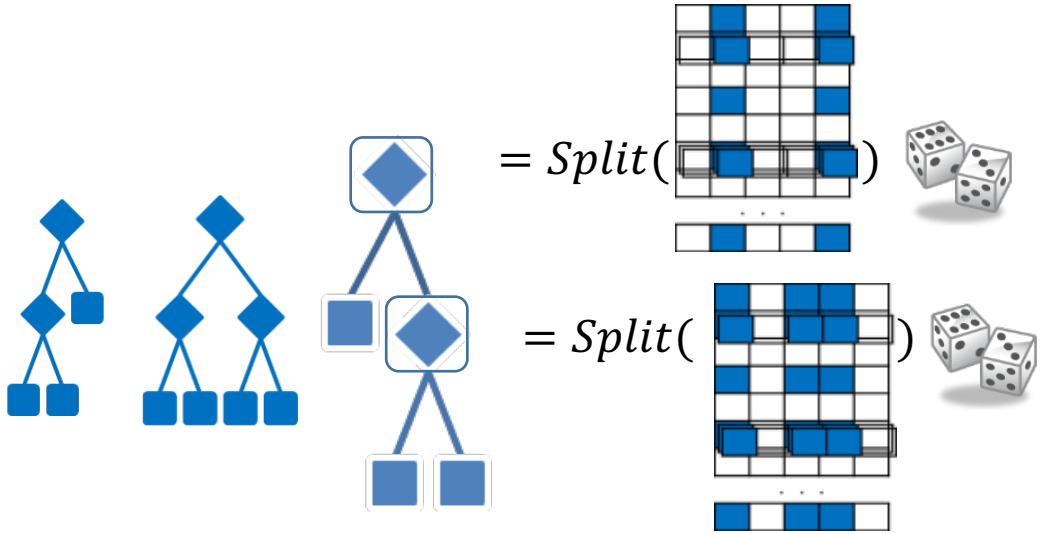


Easy way: Bagging & Random Forest

- **Bagging** – each tree is trained on bootstrap sample
- **RF** – bootstrap + features sampling on every split
- Embarrassingly parallel



Bagging



RF

http://hunch.net/~large_scale_survey/

Split criterion for regression

$$\sum_{j \in N} (y_j - c)^2 - \sum_{j \in N_1} (y_j - c_1)^2 - \sum_{j \in N_2} (y_j - c_2)^2 \to max$$

$$c_1 = \frac{Y_1}{N_1} \quad \text{(mean)}$$

$$\sum_{j \in N_1} (y_j - c_1)^2 = \sum_{j \in N_1} y_j^2 - 2c_1 \sum_{j \in N_1} y_j + \sum_{j \in N_1} c_1^2$$

$$= \sum_{j \in N_1} y_j^2 - 2c_1^2 N_1 + c_1^2 N_1$$

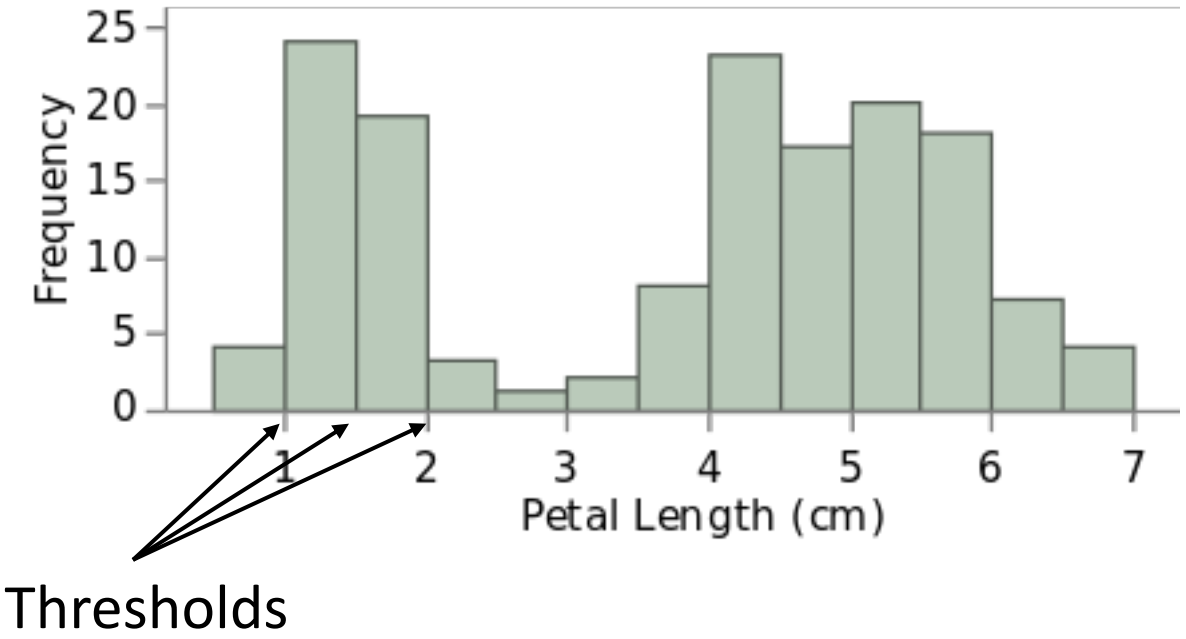
$$= \sum_{j \in N_1} y_j^2 - \left(\frac{Y_1}{N_1}\right)^2 N_1$$

$$= \sum_{j \in N_1} y_j^2 - \frac{Y_1^2}{N_1}$$

$$-\frac{Y_1^2}{N_1} - \frac{Y_2^2}{N_2} \to min$$

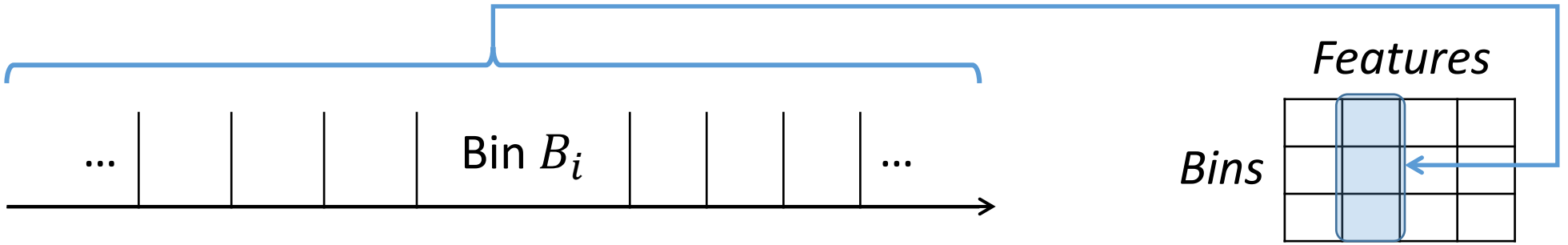
Gradient Boosting

- Boosting is iterative – we need to scale the work for a single tree
- On each split we go over all possible thresholds for every feature
  - We can discretize real valued features (binning)



Feature Binning

- An example for **regression**, binning into k bins



- **First step**: collecting statistics

$$N_i = \sum 1 \qquad Y_i = \sum y \qquad N = \sum N_i \qquad Y = \sum Y_i$$

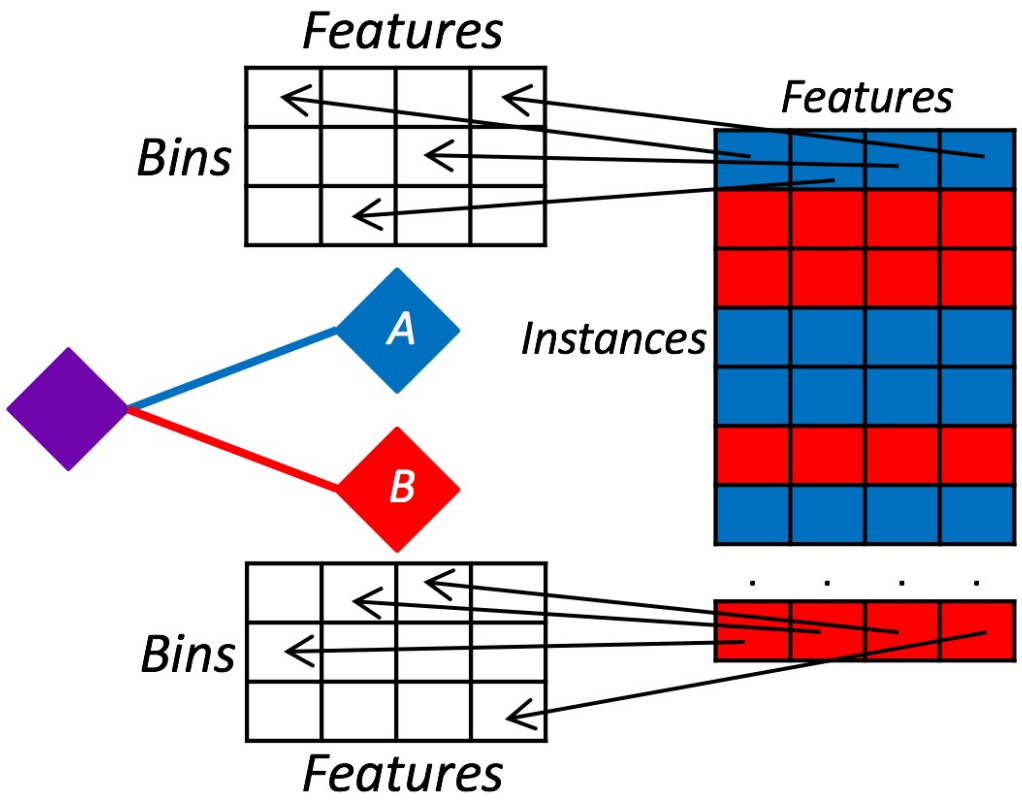
- **Second step**: calculating split criterion for a right border of bin B_i

$$-\frac{Y_{0:i}^2}{N_{0:i}} - \frac{(Y - Y_{0:i})^2}{N - N_{0:i}}$$

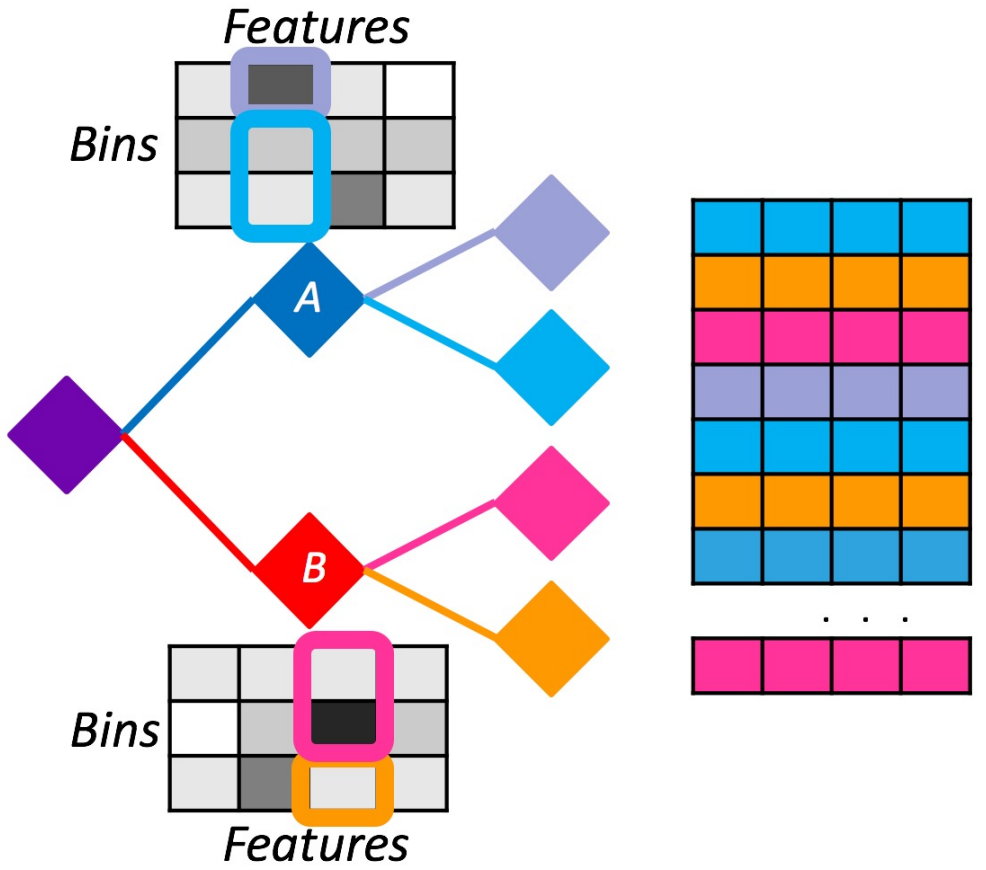
$$Y_{0:i} = \sum_{k=0}^{i} Y_k$$

(notation)

Feature Binning

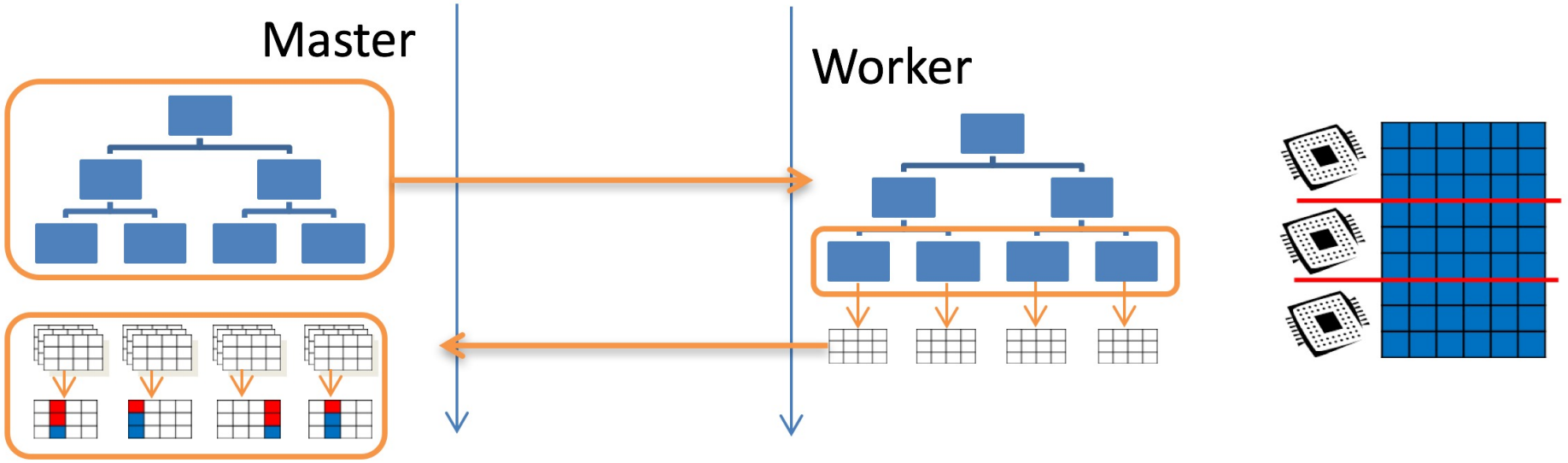


First step: collecting statistics



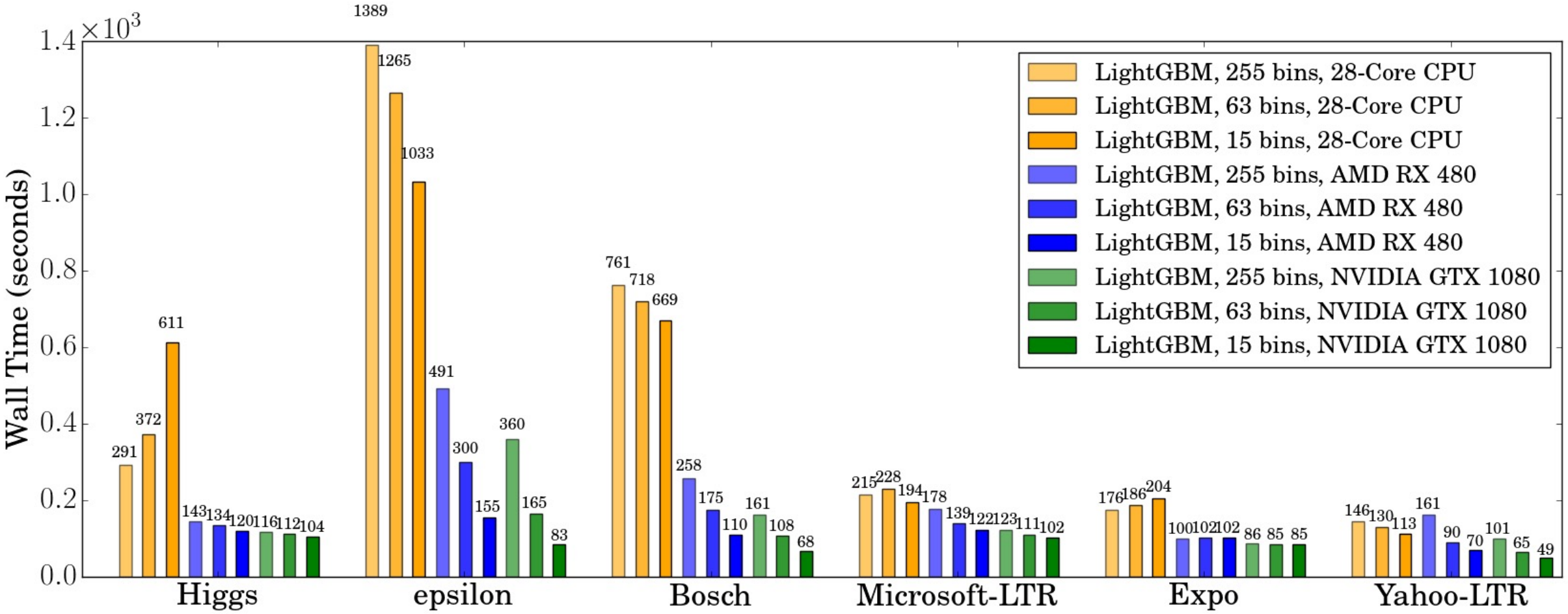
Second step: choosing the split

Final scheme



- Master
  1. Sends current model to workers
  3. Aggregates histograms (Features-Bins) from workers and chooses the best split
- Worker
  2. Goes over local data partition and fills out histograms (Features-Bins)

Example: LightGBM from Microsoft



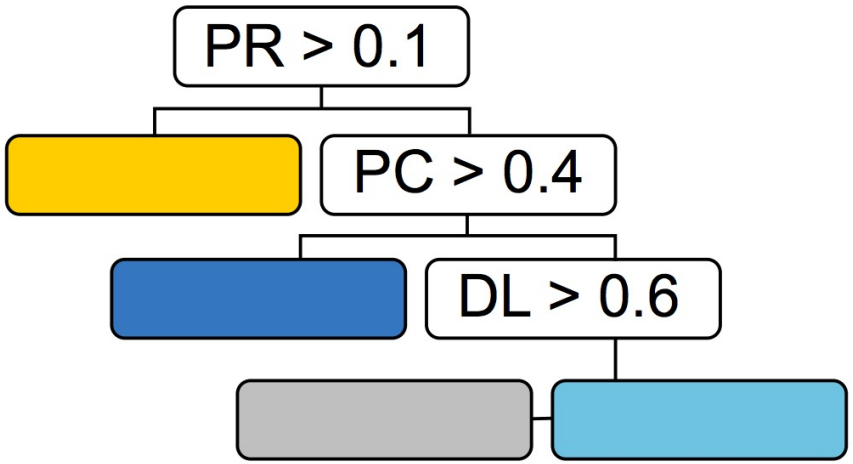
https://github.com/Microsoft/LightGBM/blob/master/docs/GPU-Performance.md

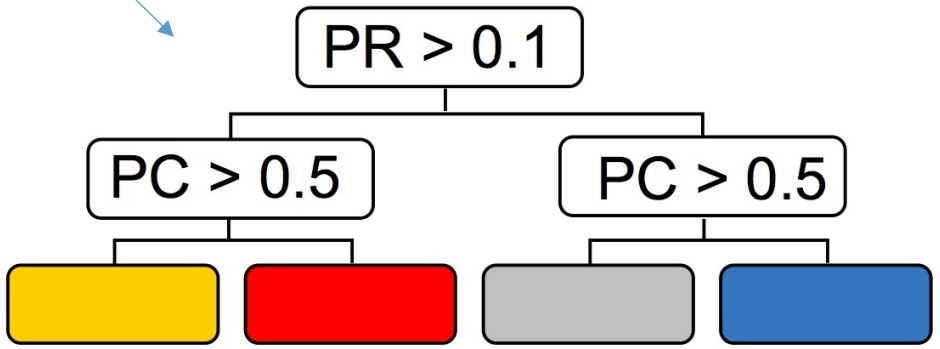
Example: Catboost from Yandex

	Catboost	Azure Boosted DT	XGBoost	LightGBM
Pol	0,994	0,922 ↓0,14%	0,991 ↓0,23%	0,991 ↓0,23%
2dplanes	0,9476	0,9474 ↓0,02%	0,9474 ↓0,02%	0,9474 ↓0,01%
Elevator	0,915	0,909 ↓0,67%	0,9 ↓1,54%	0,908 ↓0,74%
Ailerons	0,86	0,856 ↓0,45%	0,837 ↓2,67%	0,856 ↓0,55%
Fried	0,957	0,955 ↓0,22%	0,954 ↓0,32%	0,955 ↓0,17%
House	0,677	0,68 ↑0,51%	0,658 ↓2,72%	0,661 ↓2,23%

https://yadi.sk/i/3nzyqx063EAYmi

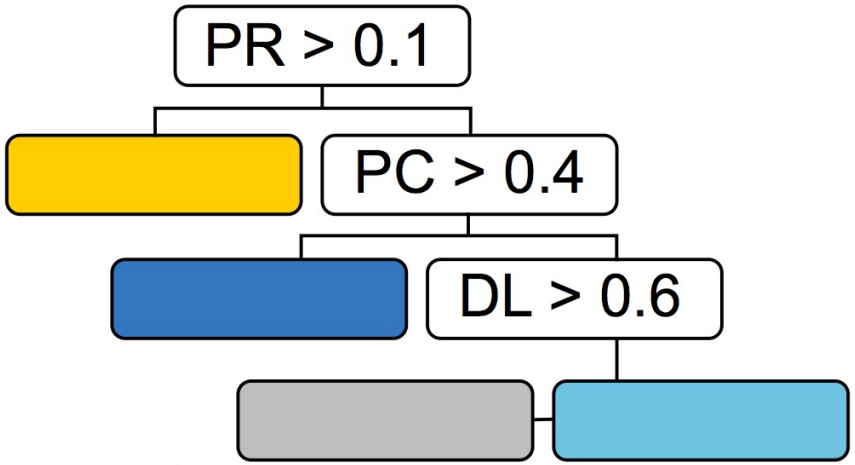
Oblivious trees in Catboost





https://yadi.sk/i/3nzyqx063EAYmi

Oblivious trees in Catboost

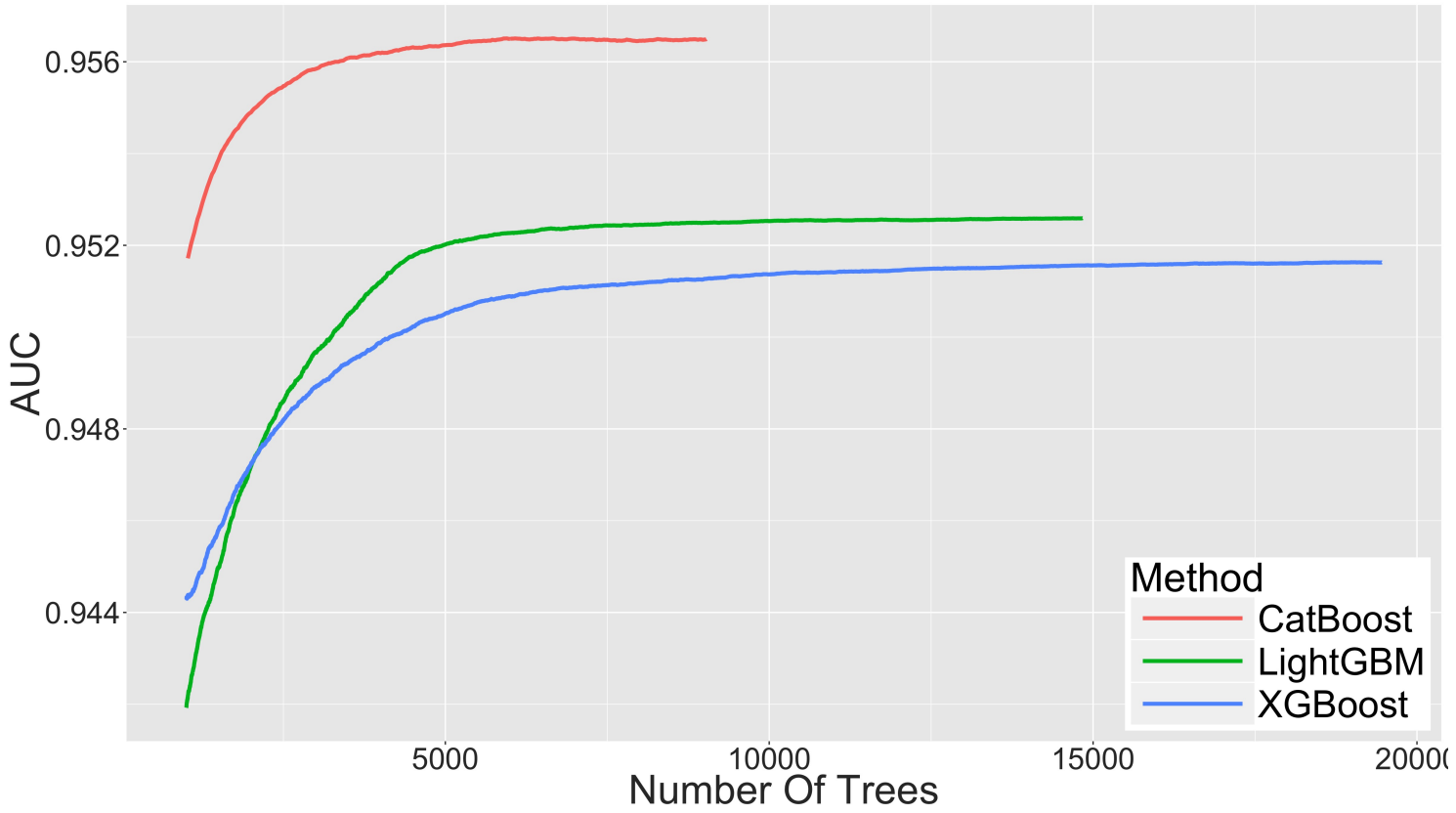


	PR ≤ 0.1	PR > 0.1
PC > 0.5		
PC ≤ 0.5		

Catboost on GPU

- https://www.kaggle.com/c/criteo-display-ad-challenge
- first 36M samples, 26 categorical, 13 numerical features

	128 bins
CPU 32 cores	1060 (1.0)
K40	373 (2.84)
GTX 1080	285 (3.7)
P40	123 (8.6)
GTX 1080Ti	301 (3.5)
P100-PCI	82 (12.9)
V100-PCI	69.8 (15)



https://github.com/catboost/benchmarks/tree/master/gpu_training

Links

- https://arxiv.org/pdf/1110.4198.pdf
- http://cilvr.cs.nyu.edu/diglib/lsml/lecture01-online-linear.pdf
- https://github.com/JohnLangford/vowpal_wabbit/wiki/Tutorial
- http://www.zinkov.com/posts/2013-08-13-vowpal-tutorial/
- http://mlwave.com/predicting-click-through-rates-with-online-machine-learning/
- http://aria42.com/blog/2014/12/understanding-lbfgs
- http://www.kdd.org/kdd2016/papers/files/rfp0697-chenAemb.pdf
- https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf
- https://github.com/catboost/benchmarks/tree/master/gpu_training
- https://github.com/Microsoft/LightGBM/blob/master/docs/GPU-Performance.md