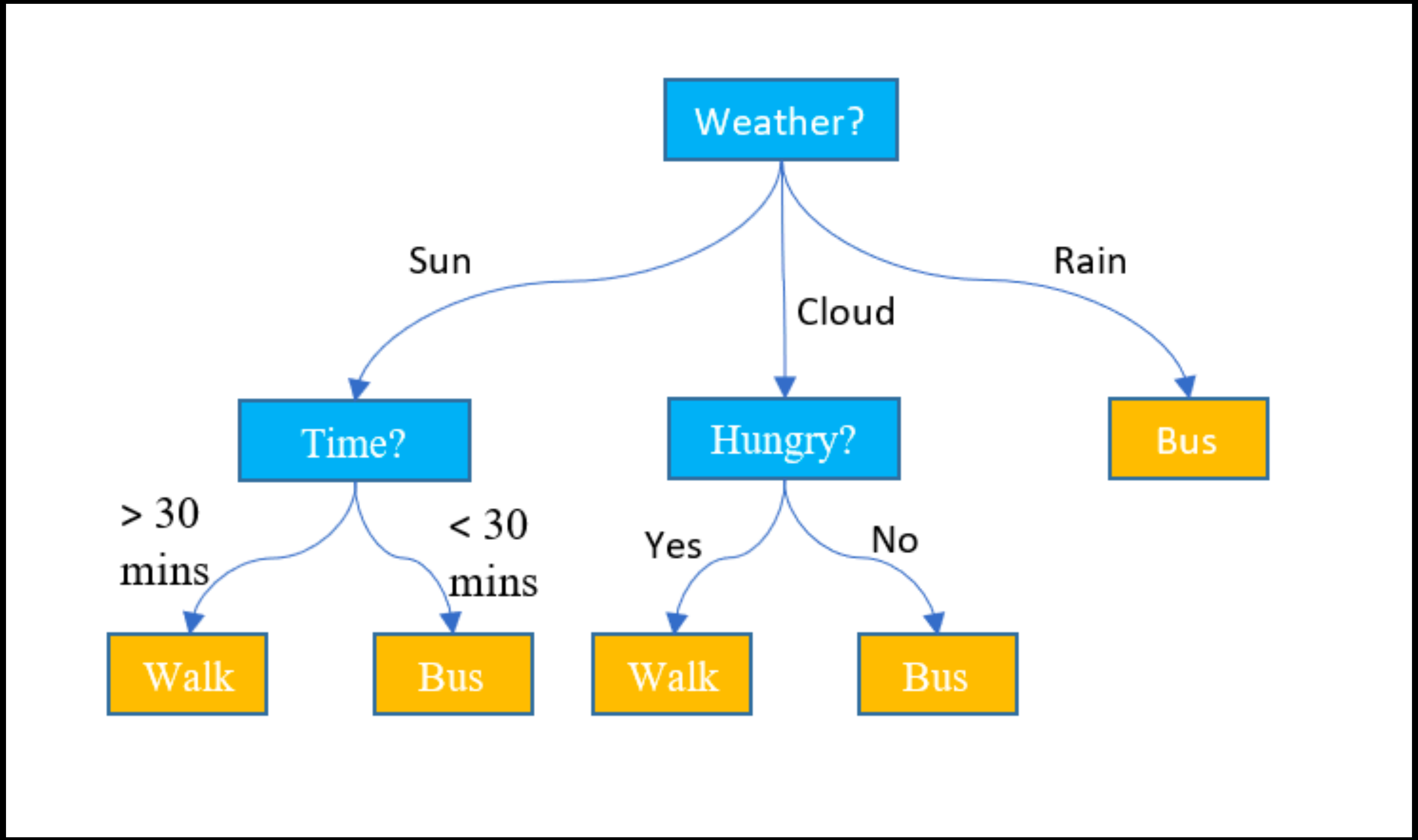
Recommendation Systems

Ranking algorithms

Eugeny Malyutin / Sergey Dudorov

Ranking algorithms

Tree reminder



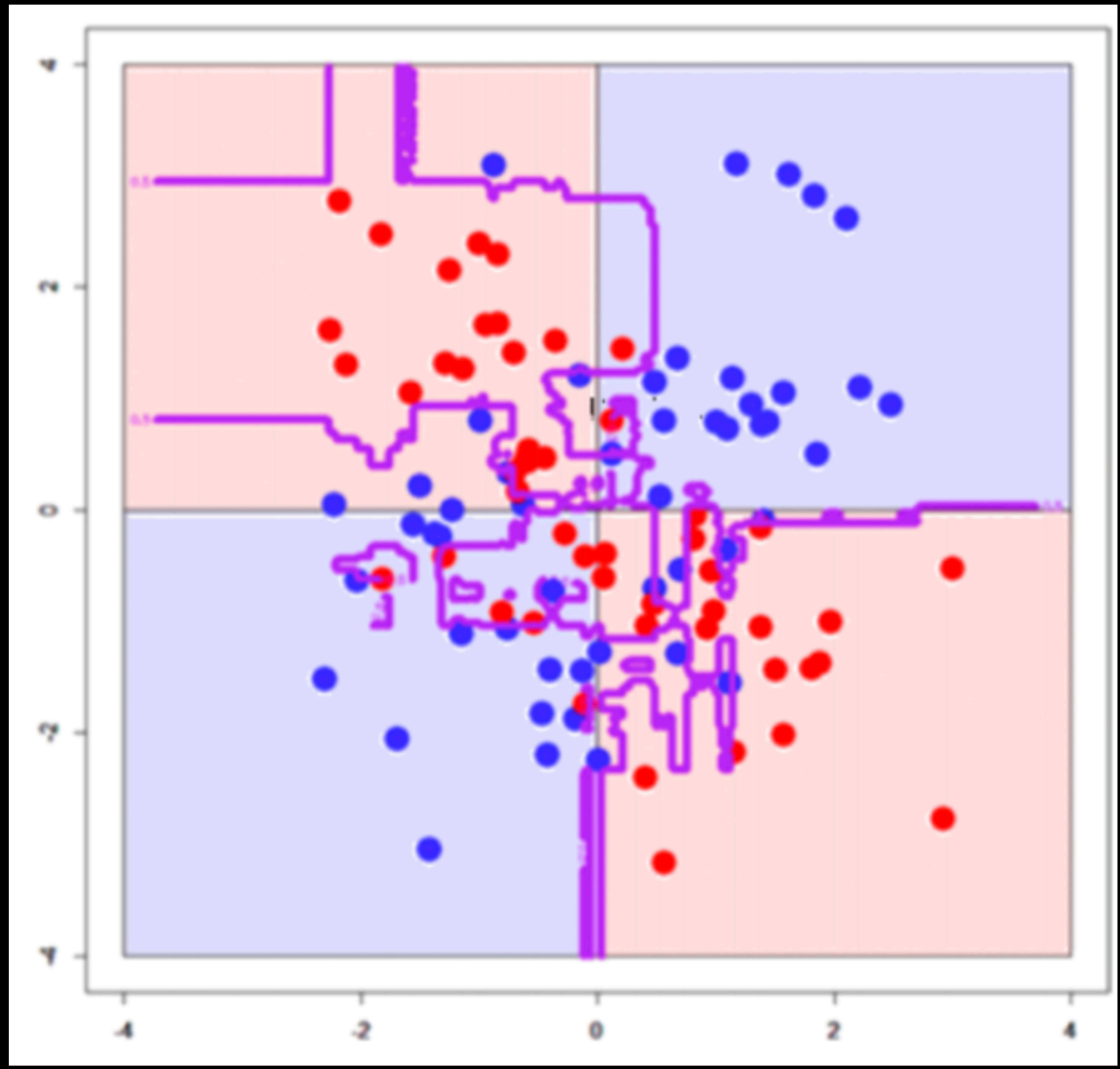
Ranking algorithms

Splitting criteria:

- **Split:** $[x_j < t]$
- **Loss criteria:** $Q(X_m, j, t) = \frac{|X_l|}{X_M} H(X_l) + \frac{|X_r|}{|X_m|} H(X_m)$
- **Information criteria:**
  - **Regression:** H(x) = $1/|X_m| \sum (y_i - \bar{y}(X))^2$
  - **Gini:** $p_k = \frac{1}{|X|} \sum_{i \in X} [y_i = k]$ and $H(x) = \sum_{k=1} p_k(1 - p_k)$
  - **Entropy:** $H(X) = \sum_{k=1}^{K} p_k ln(p_k)$

Ranking algorithms

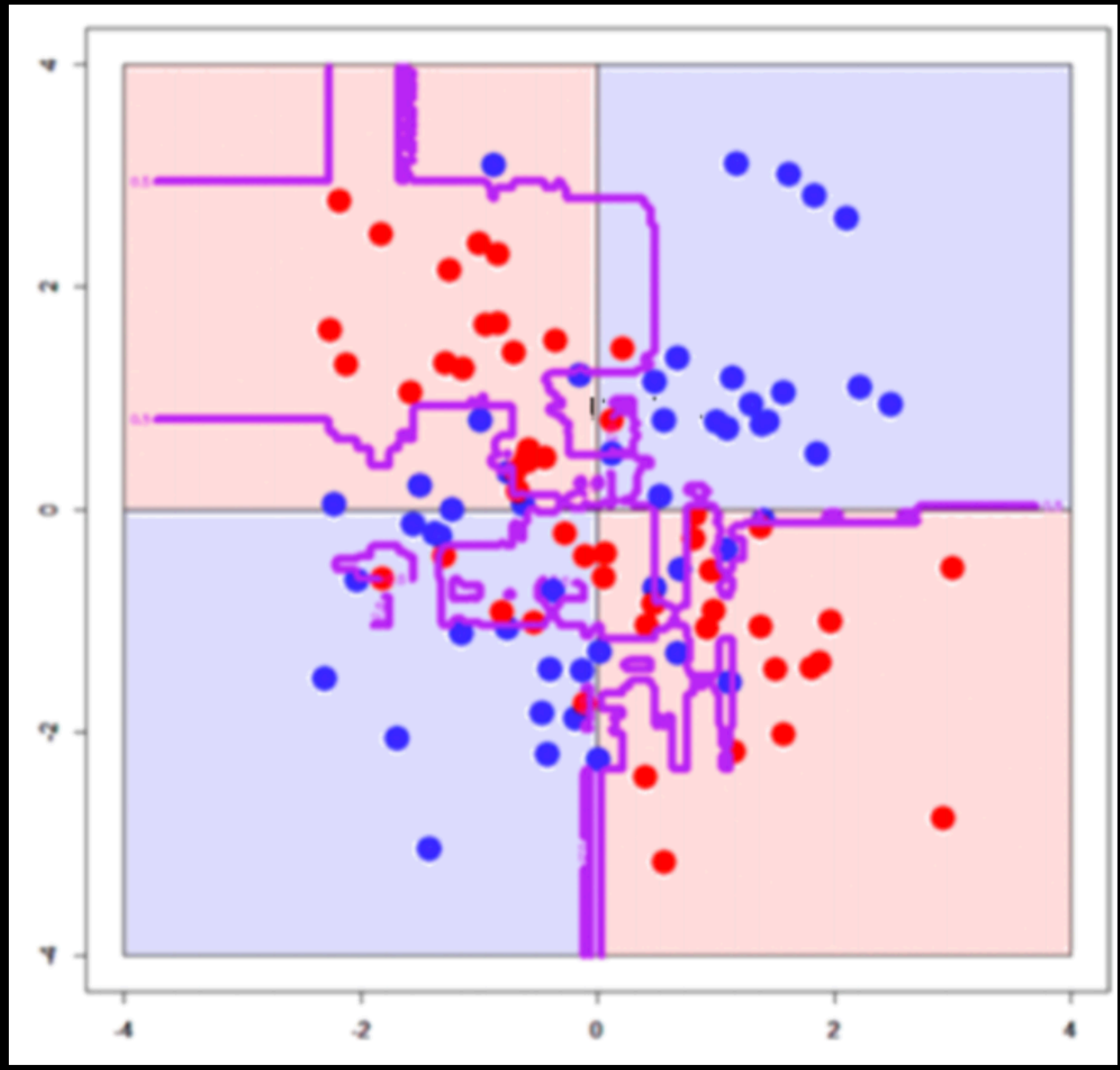
What’s went wrong?



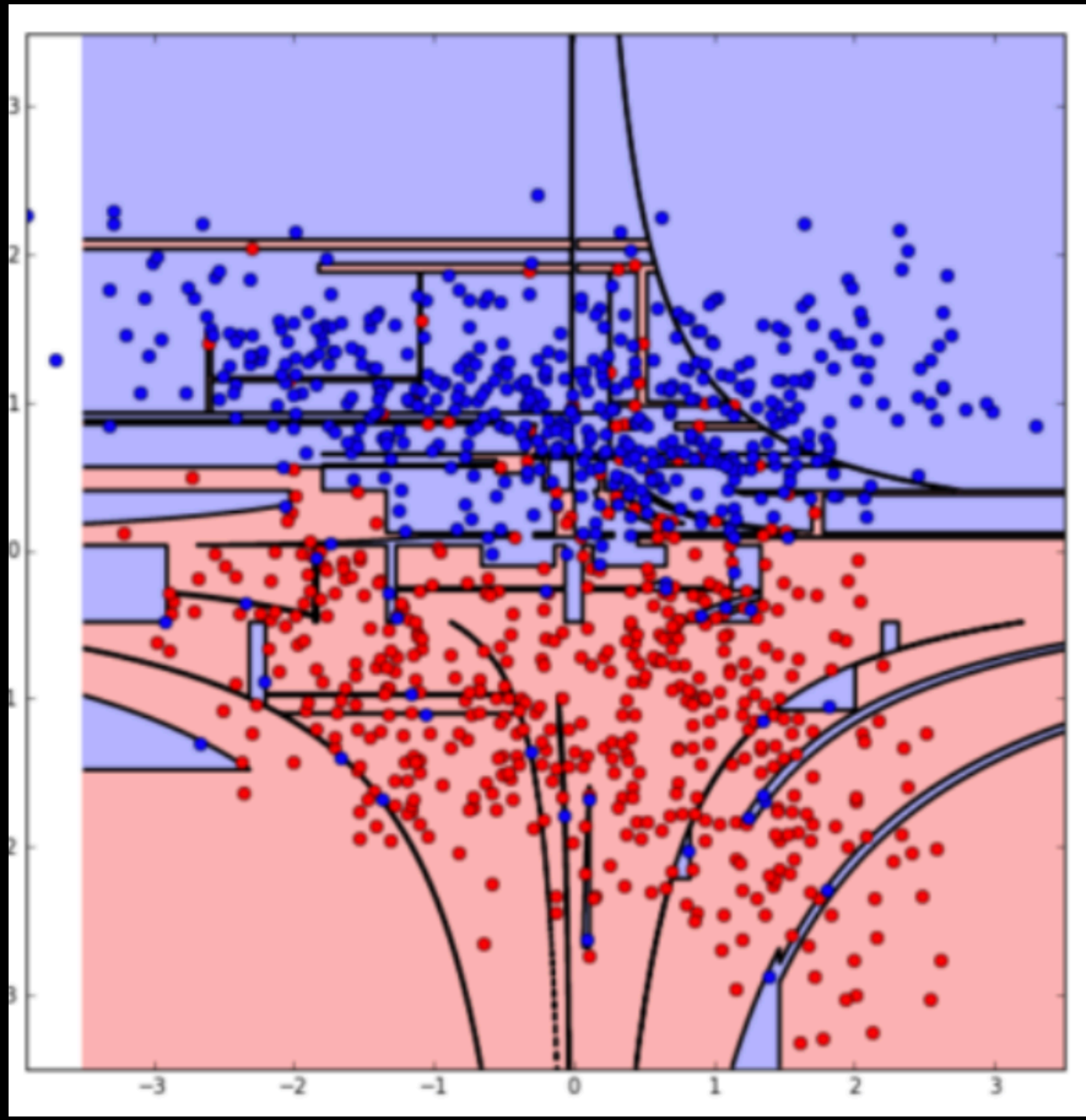
Ranking algorithms

What’s went wrong?

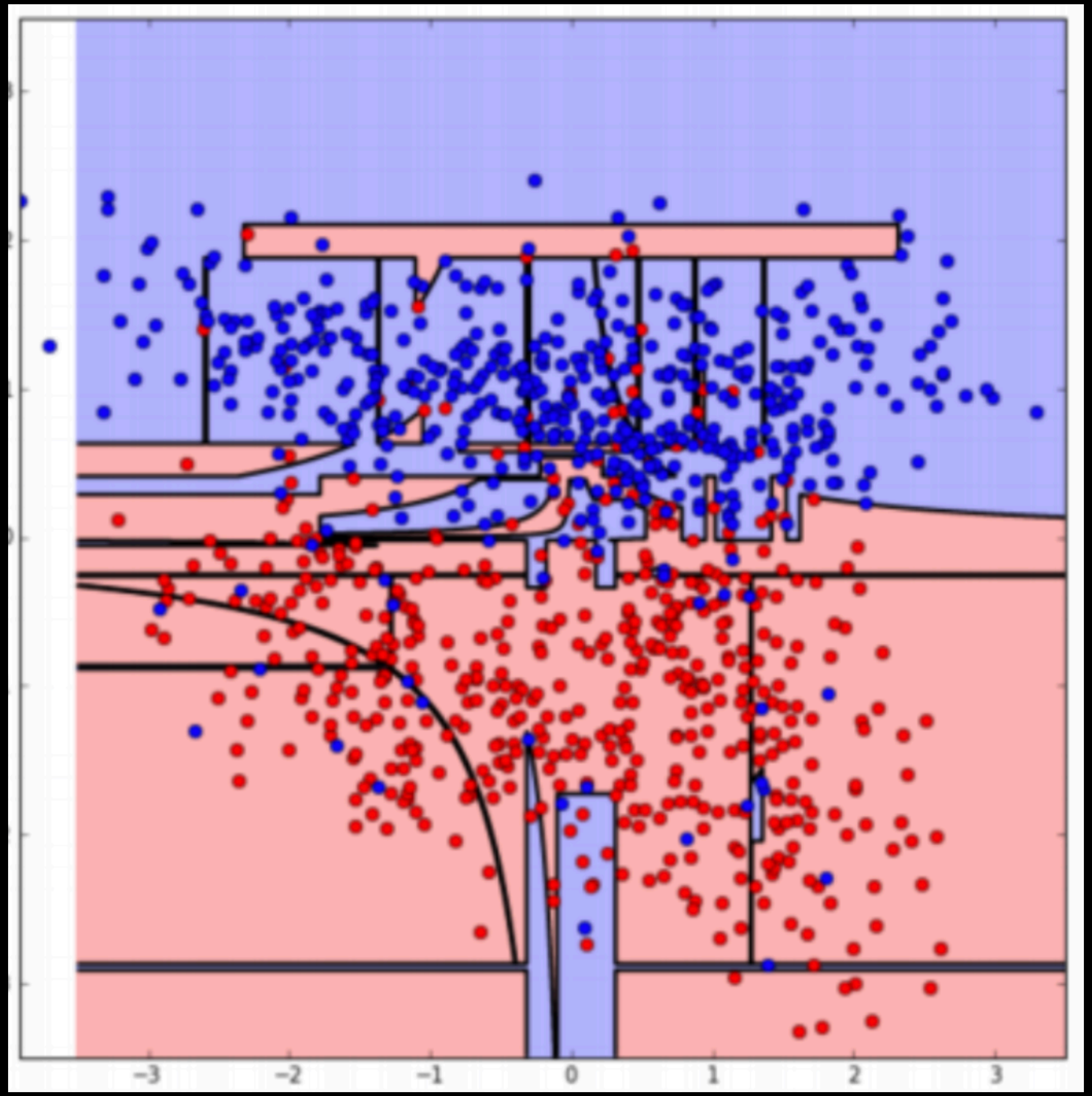
- Stop when:
  - Max length
  - Min elements
  - Pruning



Ranking algorithms
What’s went wrong?



Overfitted



Overfitted and unstable

Ranking algorithms

Error decomposition

- What is?

Ranking algorithms
Error decomposition

- **Error**:
  - **Noise:** world's imperfectness measure, exists even for ideal model on ideal data
  - **Bias:** deviation of concrete model from ideal one (averaged over all data sets)
  - **Variance:** dispersion of models answers caused by different datasets

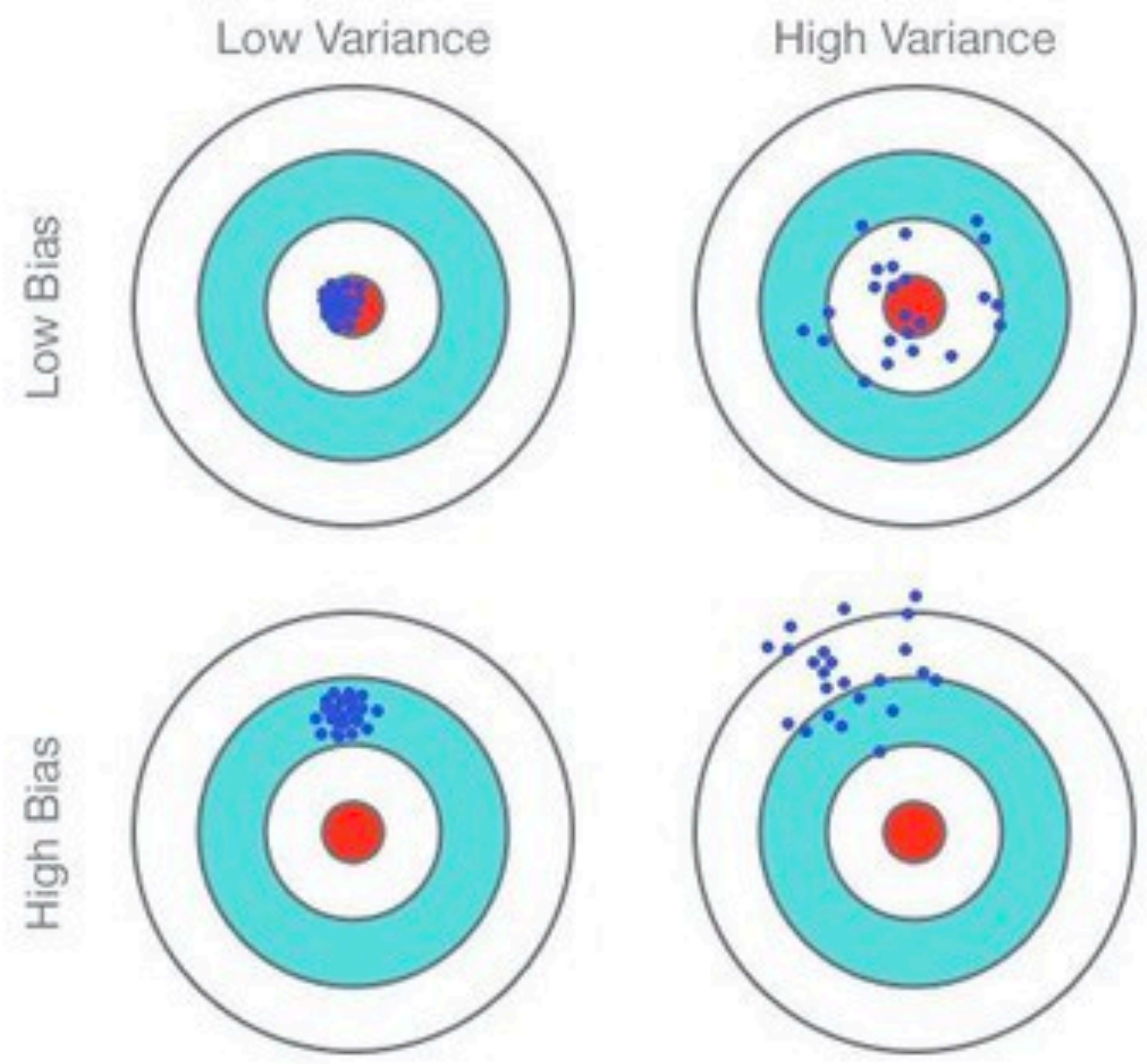


Fig. 1: Graphical Illustration of bias-variance trade-off , Source: Scott Fortmann-Roe., Understanding Bias-Variance Trade-off

Ranking algorithms

Composition idea

- Takes N different trees
- **Averages** answers:
  - **Regression:** $a(x) = \sum_{1}^{N} b_n(x)$
  - **Classification:** most popular answer
- **PRBLMS?**

Ranking algorithms
Error decomposition

- **Trees:**
  - High variance
  - Low bias
- **Linear algorithms:**
  - Low variance
  - High bias

- **Composition:**
  - Same bias
  - Variance = 1/N (algorithm_variance) + **correlation**
  - Decrease variance **N times** in case of **independent algorithms** — that's why **we need randomisation**

Ranking algorithms

Randomisation

- **Bagging (bootstrap):** learn algorithms on random subsamples from train set. Less subsamples — more randomised trees.
- **Random sub-spaces**: random sub-set of features for each tree
- **Extreme randomised trees:** choose random subset of features on each split

Ranking algorithms

Summary

- **Cons:**
  - Powerful
  - Easy to parallel
  - With bootstrap 1/3 of data samples (for concrete tree) wasn’t in train set — we can estimate test metrics on them (Out-of-bag score)
- **Cons:**
  - More trees -> more computational resources
  - Undirected search

Ranking algorithms

Boosting idea

- $b_0(x)$ — initial algorithm (zero, mean class, mean value)
- $a_m(x) = \sum_{i=1}^{m} b_i(x)$ — step of composition
- $F = \sum_{i=1}^{N} L(y_i, a_{m-1}(x_i) + b_i(x_i)) \to min_b$ — optimisation
- $s = (s_1, s_2 \dots s_N)$ — displacements vector then
  $F = \sum_{i=1}^{N} L(y_i, a_{m-1}(x_i) + s_i) \to min_s$
- Optimal shift $- \nabla F = [\frac{dF}{da_{m-1}}(x_i)]_{i=1}^{N} = [\frac{\sum_{i=1}^{N} L(y_i, a_{m-1}(x_i))}{da_{m-1}}]_{i=1}^{N} = [\frac{dL(y_i, a_{m-1})}{da_{m-1}} x_i]_{i=1}^{N}$

Ranking algorithms

Boosting idea

- $a_m(x) = \sum_{i=1}^{m} b_i(x)$ – step of composition, $F = \sum_{i=1}^{N} L(y_i, a_{m-1}(x_i) + b_i(x_i)) \to min_b$ – optimisation

- Optimal shift

$$s_i = -\nabla F = [\frac{dF}{da_{m-1}}(x_i)]_{i=1}^{N} = [\frac{\sum_{i=1}^{N} L(y_i, a_{m-1}(x_i))}{da_{m-1}}(x_i)]_{i=1}^{N} = [\frac{dL(y_i, a_{m-1})}{da_{m-1}}(x_i)]_{i=1}^{N}$$

- But wait, s_i is not an algorithm, it's a vector of numbers!

- Yep, but we can learn out algorithm $b_m(x_i) \to s_i$

http://www.machinelearning.ru/wiki/images/9/9a/fonarev.overview_of_boosting_methods.pdf

Ranking algorithms

Summary

- Powerful (extremely powerful)
- Allows all this tricks of gradient algorithms: learning rate, decay, …
- Hard to interpret (~)
- Works mostly with weak algorithms
- Allows you to set various range of functions(?) as loss

http://www.machinelearning.ru/wiki/images/9/9a/fonarev.overview_of_boosting_methods.pdf

XGboost

XGBoost

$$\hat{y}_i^{(0)} = 0$$
$$\hat{y}_i^{(1)} = f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i)$$
$$\hat{y}_i^{(2)} = f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i)$$
$$\dots$$
$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)$$

$$\longrightarrow$$

$$\text{obj}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega(f_i)$$
$$= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + \text{constant}$$

Composition structure

Objective function

$\Omega(f_i)$ **— regularisation**

NB! — xgboost about regression trees only,
but possible to solve classification

https://xgboost.readthedocs.io/en/latest/tutorials/model.html

XGboost

XGBoost

$$\text{obj}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega(f_i)$$
$$= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + \text{constant}$$

Objective function

$$\text{obj}^{(t)} = \sum_{i=1}^{n} (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \sum_{i=1}^{t} \Omega(f_i)$$
$$= \sum_{i=1}^{n} [2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2] + \Omega(f_t) + \text{constant}$$

MSE example

$$\text{obj}^{(t)} = \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + \text{constant}$$

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$
$$h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

2nd order Taylor's approximation

https://xgboost.readthedocs.io/en/latest/tutorials/model.html

XGboost

Regularisation

$$f_t(x) = w_{q(x)}, w \in R^T, q : R^d \to \{1,2,\cdots,T\} .$$

Formal tree

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Complexity

- w — vector of scores on j-th index
- q — function to assign object to leaf
- T — number of leaves

https://xgboost.readthedocs.io/en/latest/tutorials/model.html

https://stats.stackexchange.com/questions/202858/xgboost-loss-function-approximation-with-taylor-expansion

XGboost

Regularisation in learning

$$\text{obj}^{(t)} \approx \sum_{i=1}^{n} [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$
$$= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T$$

Obj function with Taylor and formal tree

$\longrightarrow$

$$\text{obj}^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2] + \gamma T$$

$$G_j = \sum_{i \in I_j} g_i \qquad H_j = \sum_{i \in I_j} h_i$$

«Simplification»

https://xgboost.readthedocs.io/en/latest/tutorials/model.html

https://stats.stackexchange.com/questions/202858/xgboost-loss-function-approximation-with-taylor-expansion

XGboost

...

$$\text{obj}^{(t)} \approx \sum_{i=1}^{n} [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$
$$= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T$$

Obj function with Taylor and formal tree

$$\longrightarrow$$

$$\text{obj}^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2] + \gamma T$$

$$G_j = \sum_{i \in I_j} g_i \qquad H_j = \sum_{i \in I_j} h_i$$

«Simplification»

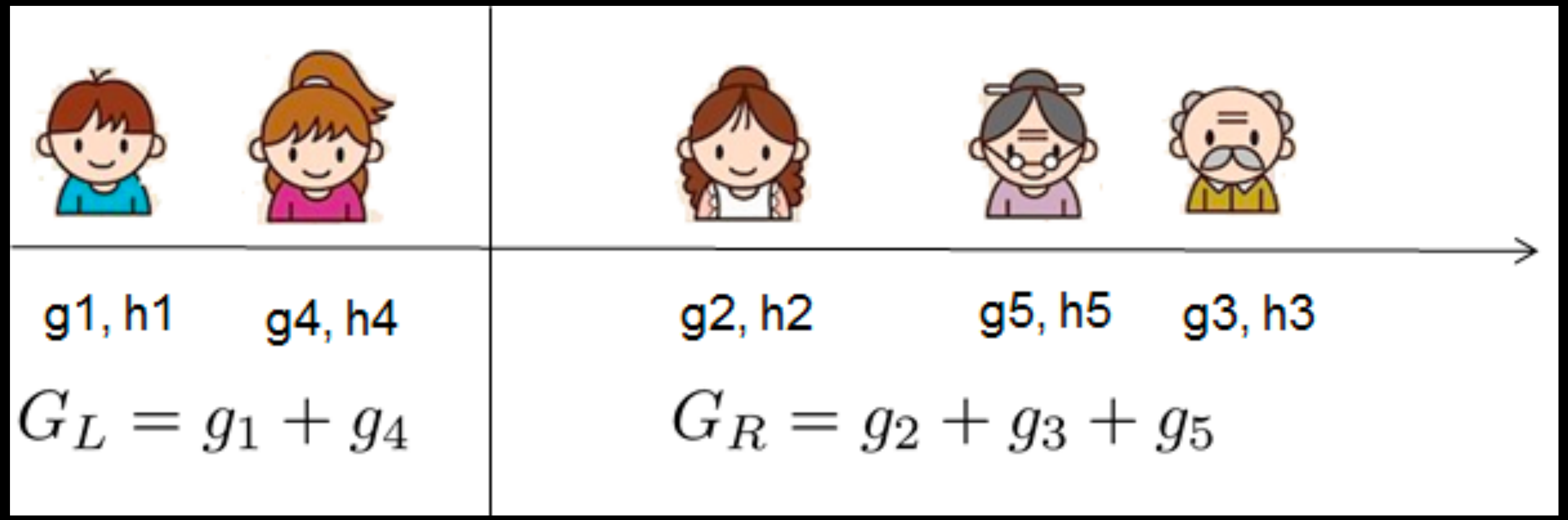
https://xgboost.readthedocs.io/en/latest/tutorials/model.html

https://stats.stackexchange.com/questions/202858/xgboost-loss-function-approximation-with-taylor-expansion

XGboost:
Sorting schema:

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$$

«Pruning»



https://xgboost.readthedocs.io/en/latest/tutorials/model.html

https://stats.stackexchange.com/questions/202858/xgboost-loss-function-approximation-with-taylor-expansion

XGboost

Summing up:

- Decomposes loss-function in Taylor row
- Allows arbitrary set of loss functions
- Zip regularisation inside learning process
- Scan all dataset ones to find best split

https://xgboost.readthedocs.io/en/latest/tutorials/model.html

https://stats.stackexchange.com/questions/202858/xgboost-loss-function-approximation-with-taylor-expansion

XGboost
And his friends:

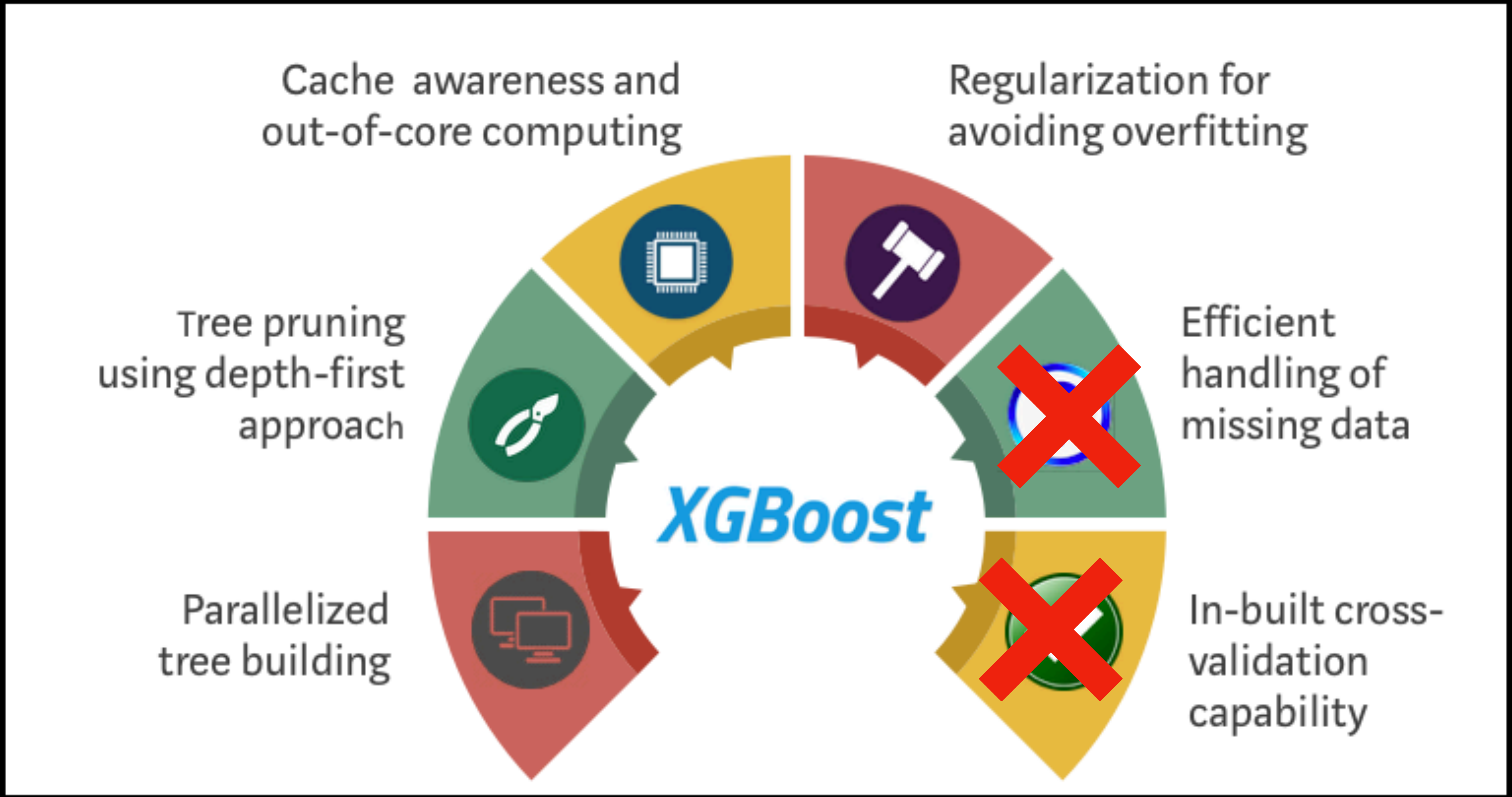
Pros:

- Well-known (est. 2016)
- Available on ton different platforms (as cpp binding, mostly)
- Allows distribute training (spark, for example)

Cons:

- Not so fancy comparing to lGBM, catBoost
- Not so powerful (~)
- Very strange sparsity

XGBoost:


Cache awareness and out-of-core computing
Regularization for avoiding overfitting
Tree pruning using depth-first approach
Efficient handling of missing data
XGBoost
Parallelized tree building
In-built cross-validation capability

XGBoost notes:

- XGBoost internal feature significance sucks, **use SHAP values**.
- XGBoost objectives:
  - `binary:logistic`: logistic regression for binary classification, output probability
  - **reg:squarederror**: regression with squared loss. Practically better on classification oO
  - **rank:{pairwise/map/ndcg}**: lambda mart ranking. Sometimes works significantly better then regr losses.
- Hyperparameters matters, tune them wisely (look refs).
- High gamma -> feature selection -> low gamma and features removed

https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

Learning to rank

«Lambda»-smth

$$w = w + \eta \frac{\sigma}{1 + exp(\sigma \langle x_i - x_j, w \rangle)} (x_i - x_j)$$

Gradient step optimising logit function on linear algorithm

NDCG delta
by changing x_i and x_j

$$w = w + \eta \frac{\sigma}{1 + exp(\sigma \langle x_i - x_j, w \rangle)} |\Delta NDCG_{ij}| (x_i - x_j)$$

Gradient step optimising NDCG function on linear algorithm

https://www.microsoft.com/en-us/research/uploads/prod/2016/02/MSR-TR-2010-82.pdf

Learning to rank

Summary

- Learning to rank - specific task with specific metrics and tricks.
- There are: point- pair- and list-wise algorithms.
- Best pair-wise algorithms looks as pointwise at runtime.
- There are tons ranking algorithms. Most powerful and/or used — GBDT
- XGBoost — allows arbitrary set of loss functions, encorporates regularisation into training, use Taylor 2nd order to estimate loss function.
- It’s possible to optimise ranking metrics directly using lambda- techniques