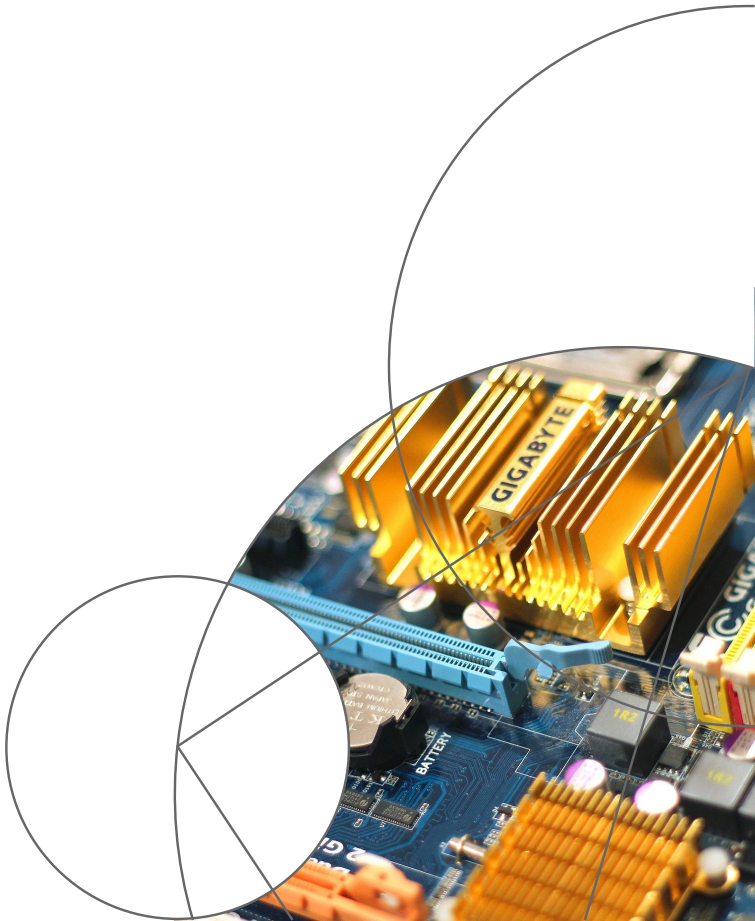
Ensembles Methods

Large-Scale Data Analysis

Fabian Gieseke, Christian Igel
Department of Computer Science



Today!

http://jmlr.csail.mit.edu/papers/volume15/delgado14a/delgado14a.pdf

Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?

Manuel Fernández-Delgado MANUEL.FERNANDEZ.DELGADO@USC.ES
Eva Cernadas EVA.CERNADAS@USC.ES
Senén Barro SENEN.BARRO@USC.ES
CITIUS: Centro de Investigación en Tecnoloxías da Información da USC
University of Santiago de Compostela
Campus Vida, 15872, Santiago de Compostela, Spain

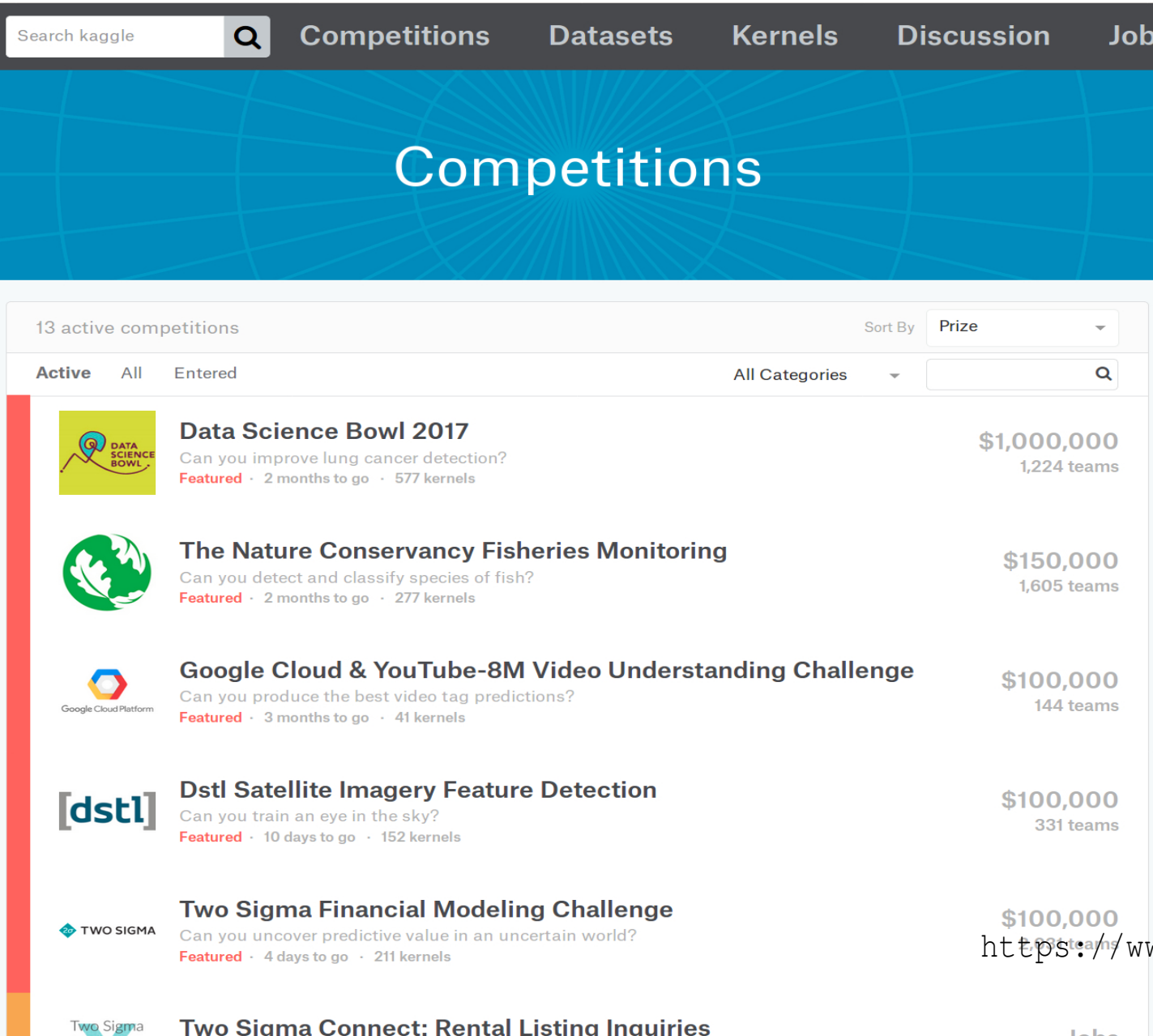
Dinani Amorim DINANIAMORIM@GMAIL.COM
Departamento de Tecnologia e Ciências Sociais- DTCS
Universidade do Estado da Bahia
Av. Edgard Chastinet S/N - São Geraldo - Juazeiro-BA, CEP: 48.305-680, Brasil

Editor: Russ Greiner

Abstract

We evaluate **179 classifiers** arising from **17 families** (discriminant analysis, Bayesian, neural networks, support vector machines, decision trees, rule-based classifiers, boosting, bagging, stacking, random forests and other ensembles, generalized linear models, nearest-neighbors, partial least squares and principal component regression, logistic and multinomial regression, multiple adaptive regression splines and other methods), implemented in Weka, R (with and without the caret package), C and Matlab, including all the relevant classifiers available today. We use **121 data sets**, which represent **the whole UCI** data base (excluding the large-scale problems) and other own real problems, in order to achieve significant conclusions about the classifier behavior, not dependent on the data set collection. **The classifiers most likely to be the bests are the random forest** (RF) versions, the best of which (implemented in R and accessed via caret) achieves 94.1% of the maximum accuracy overcoming 90% in the 84.3% of the data sets. However, the difference is not statistically significant with the second best, the SVM with Gaussian kernel

Data Analysis: Real-World Competitions



https://www.kaggle.com

Today!

https://dnc1994.com/2016/05/rank-10-percent-in-first-kaggle-competition-en/

Model Selection

When the features are set, we can start training models. Kaggle competitions usually favor **tree-based models**:

- **Gradient Boosted Trees**
- Random Forest
- Extra Randomized Trees

The following models are slightly worse in terms of general performance, but are suitable as base models in ensemble learning (will be discussed later):

- SVM
- Linear Regression
- Logistic Regression
- Neural Networks

Note that this does not apply to computer vision competitions which are pretty much dominated by neural network models.

All these models are implemented in **Sklearn**.

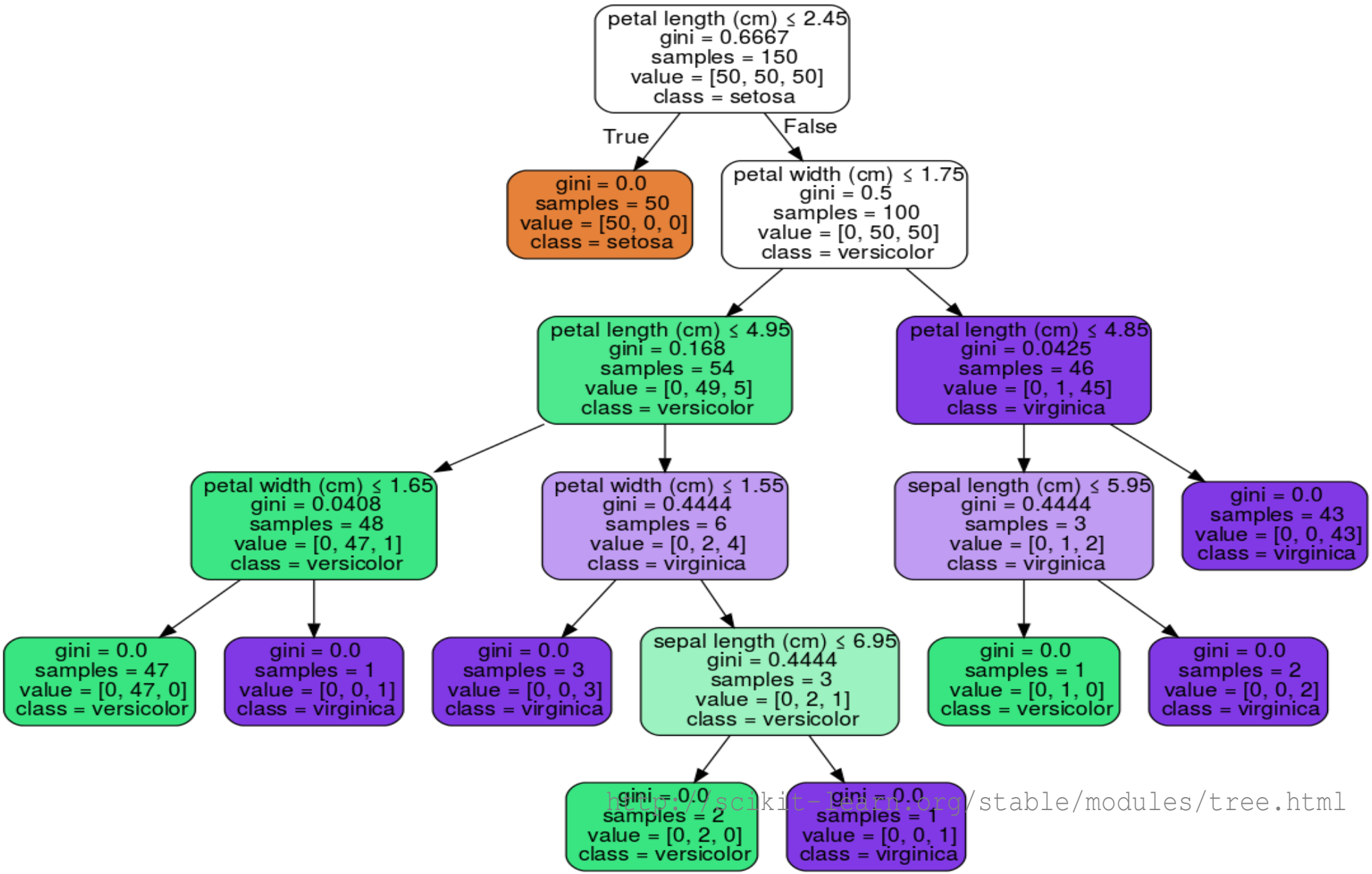
Here I want to emphasize the greatness of **Xgboost**. The outstanding performance of gradient boosted trees and Xgboost's efficient implementation makes it very popular in Kaggle competitions. Nowadays almost every winner uses Xgboost in one way or another.

Updated on Oct 28th, 2016: Recently Microsoft open sourced **LightGBM**, a potentially better

Outline

Outline

Example: Iris Flower Classification Tree

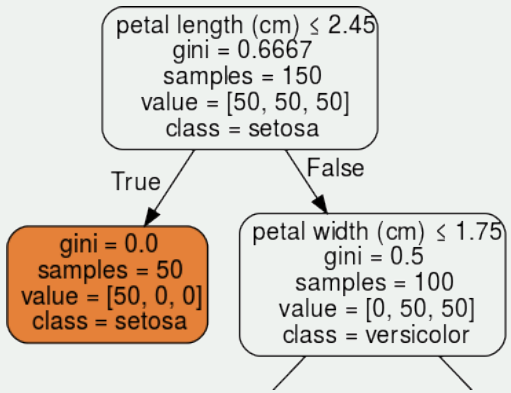


http://scikit-learn.org/stable/modules/tree.html

Classification and Regression Trees (CARTs)

Structure & Training

- Every internal node is associated with one input dimension $i \in \{1, \dots, d\}$ and a threshold θ.
- At each inner node, the training data $S = \{(\mathbf{x}_1, y_1), \dots\}$ at that node are split into



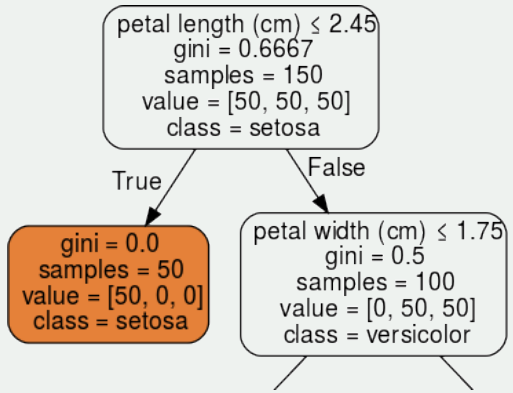
$$L_{i,\theta} = \{(\mathbf{x}, y) \in S \,|\, x_i \leq \theta\} \text{ and } R_{i,\theta} = \{(\mathbf{x}, y) \in S \,|\, x_i > \theta\},$$

  which are passed to the left/right child for further processing.
- Construction: The optimal tree cannot be found efficiently. Therefore, trees are built using a heuristic.
- The leaf nodes, indexed by $\tau = 1, \dots, M$, define regions $\mathcal{R}_\tau \subseteq \mathbb{R}^d$.

Building CARTs

Basic Idea

- Every inner node is associated with one coordinate $i \in \{1, \ldots, d\}$ and a threshold θ.
- At each inner node, the associated data $S = \{(\mathbf{x}_1, y_1), \ldots\}$ are split into $L_{i,\theta}$ and $R_{i,\theta}$ such that the information gain



$$G_{i,\theta}(S) = Q(S) - \frac{|L_{i,\theta}|}{|S|} Q(L_{i,\theta}) - \frac{|R_{i,\theta}|}{|S|} Q(R_{i,\theta})$$

  is maximized, where Q is some impurity measure.
- If the number $|S|$ of points at a node is smaller than a user-defined value or if S is pure (i.e., all point have the same label), the node becomes a leaf (further stopping rules exist).

Simplified Objective

Note that

$$\underset{i,\theta}{\text{maximize}}\ G_{i,\theta}(S) = Q(S) - \frac{|L_{i,\theta}|}{|S|}Q(L_{i,\theta}) - \frac{|R_{i,\theta}|}{|S|}Q(R_{i,\theta})$$

is equivalent to

$$\underset{i,\theta}{\text{minimize}}\ |L_{i,\theta}|Q(L_{i,\theta}) + |R_{i,\theta}|Q(R_{i,\theta})$$

Let's build a tree!

Recursive Tree Construction

Procedure: BUILDTREE(S,m)

Require: $S = \{(\mathbf{x}_1, y_1), \ldots\}$ and minimum number of leaves m.

Ensure: Tree $\mathcal{T}$ built for the input patterns in S.

1: **if** $|S| \leq m$ **then**
2: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
3: **end if**
4: **if** $y_i = y_j$ for all $(\mathbf{x}_i, y_i), (\mathbf{x}_j, y_j) \in S$ **then**
5: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
6: **end if**
7: Find $(i^*, \theta^*) = \text{argmax}_{i,\theta}\, G_{i,\theta}(S)$
8: $\mathcal{T}_l$ = BUILDTREE($L_{i,\theta}$, m)
9: $\mathcal{T}_r$ = BUILDTREE($R_{i,\theta}$, m)
10: Generate node that stores the splitting information (i^*, θ^*) and pointers to its subtrees $\mathcal{T}_l$ and $\mathcal{T}_r$. Let $\mathcal{T}$ denote the resulting tree.
11: **return** $\mathcal{T}$

Regression Trees

Construction: What are "optimal" splits?

- Impurity measure: For regression problems, a common choice is

$$Q(S) = \sum_{(\mathbf{x},y)\in S} (y - \bar{y})^2$$

with $\bar{y} = \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} y$

- Find optimal splitting dimension i^* and threshold θ^* w.r.t.

$$\underset{i,\theta}{\text{minimize}} \; |L_{i,\theta}| Q(L_{i,\theta}) + |R_{i,\theta}| Q(R_{i,\theta})$$

How to compute the optimal dimension and threshold?

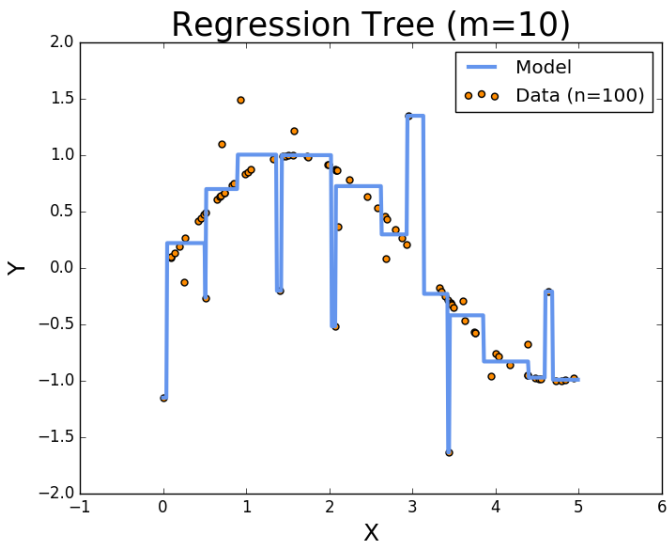
1. Naive approach, $O(d \cdot |S|^2)$ time: For each i, do:
   - Consider all possible thresholds (e.g., $\{\mathbf{x}_i | \mathbf{x} \in S\}$).
   - For each threshold θ: Compute $|L_{i,\theta}| Q(L_{i,\theta})$ and $|R_{i,\theta}| Q(R_{i,\theta})$.
2. Better approach, $O(d \cdot |S| \log |S|)$ time: For each i, do:
   - Sort all instances $(\mathbf{x}, y)$ in S according to the patterns i-th dimension.
   - Scan this sorted list and update these values incrementally.

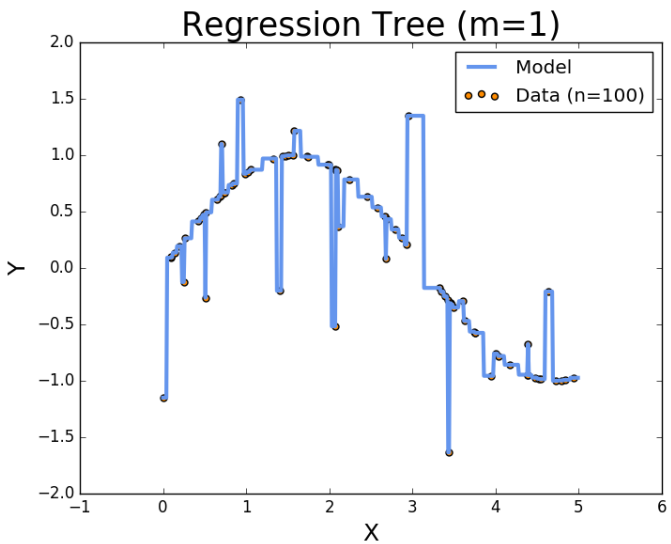
Regression Trees

Prediction

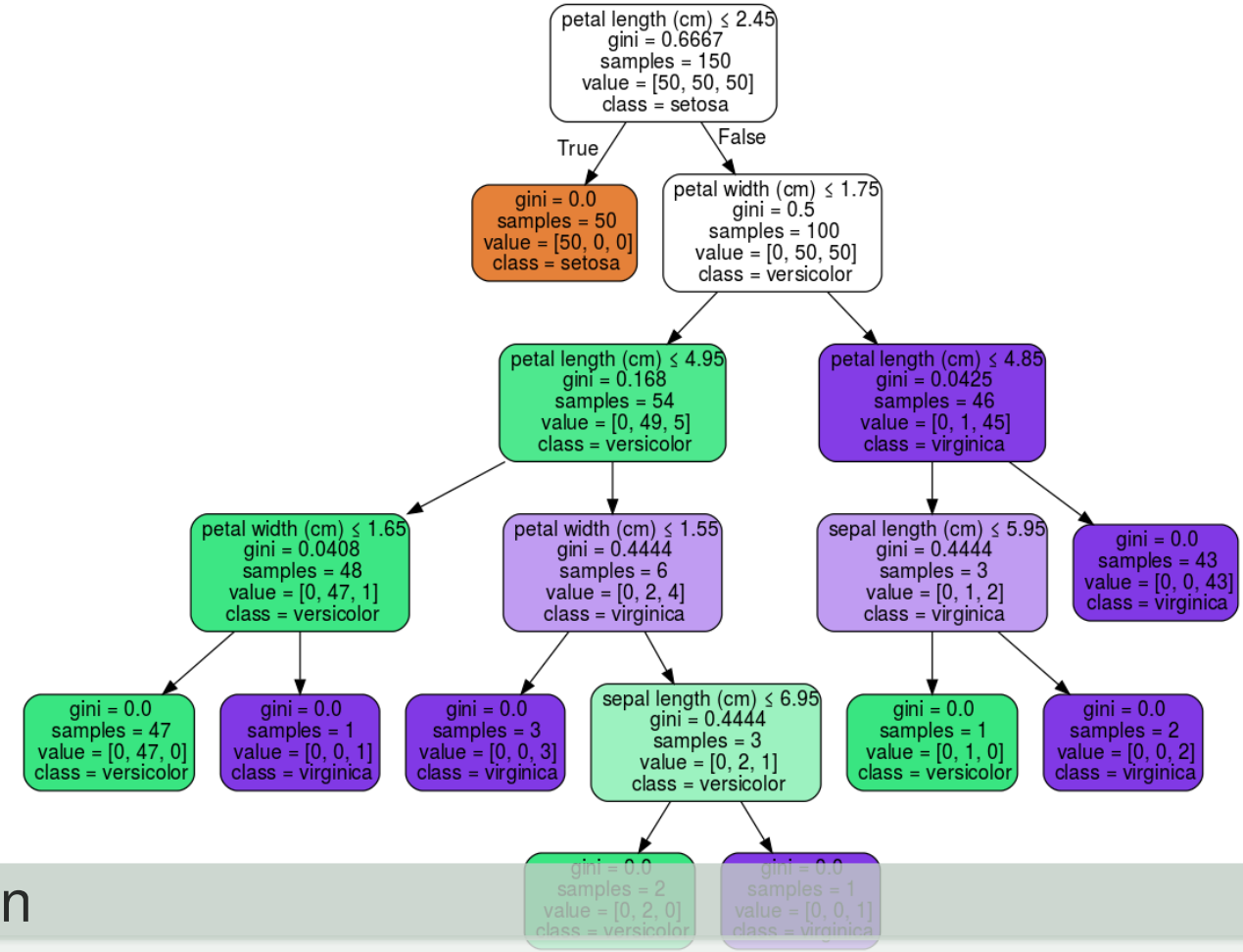
- Training data $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$
- Each leaf node $\tau = 1, \ldots, M$ of the tree $\mathcal{T}$ corresponds to a region $\mathcal{R}_\tau \subseteq \mathbb{R}^d$
- The output $\hat{y} = \mathcal{T}(\mathbf{x})$ given a new input instance $\mathbf{x}$ is

$$\mathcal{T}(\mathbf{x}) = \sum_{\tau=1}^{M} c_\tau \mathbb{I}\{\mathbf{x} \in \mathcal{R}_\tau\} \quad \text{with} \quad c_\tau = \frac{\sum_{\{\mathbf{x} \mid (\mathbf{x},y) \in S \wedge \mathbf{x} \in \mathcal{R}_\tau\}} y_j}{|\{\mathbf{x} \mid (\mathbf{x}, y) \in S \wedge \mathbf{x} \in \mathcal{R}_\tau\}|} \ .$$





Classification Trees



Prediction

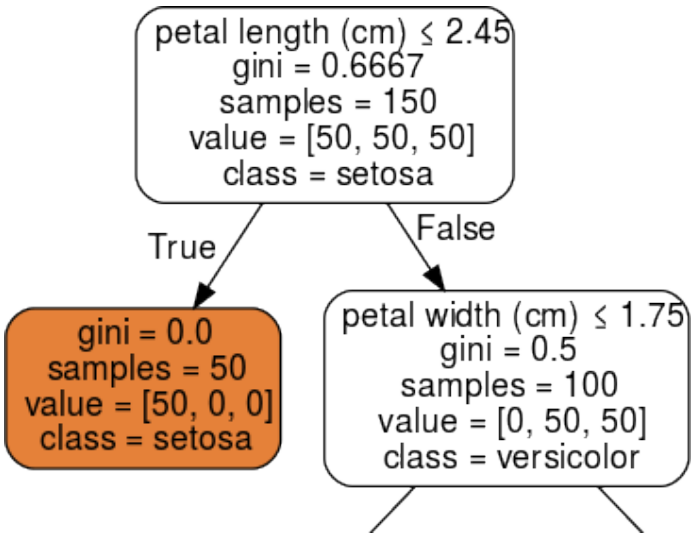
1. Traverse the tree $\mathcal{T}$ to find the leaf that contains the query $\mathbf{x}$
2. Each leaf node $\tau = 1, \ldots, M$ of the tree $\mathcal{T}$ corresponds to a region $\mathcal{R}_\tau \subset \mathbb{R}^d$
3. Use majority class in leaf as prediction $\mathcal{T}(\mathbf{x})$

Classification Trees

Construction

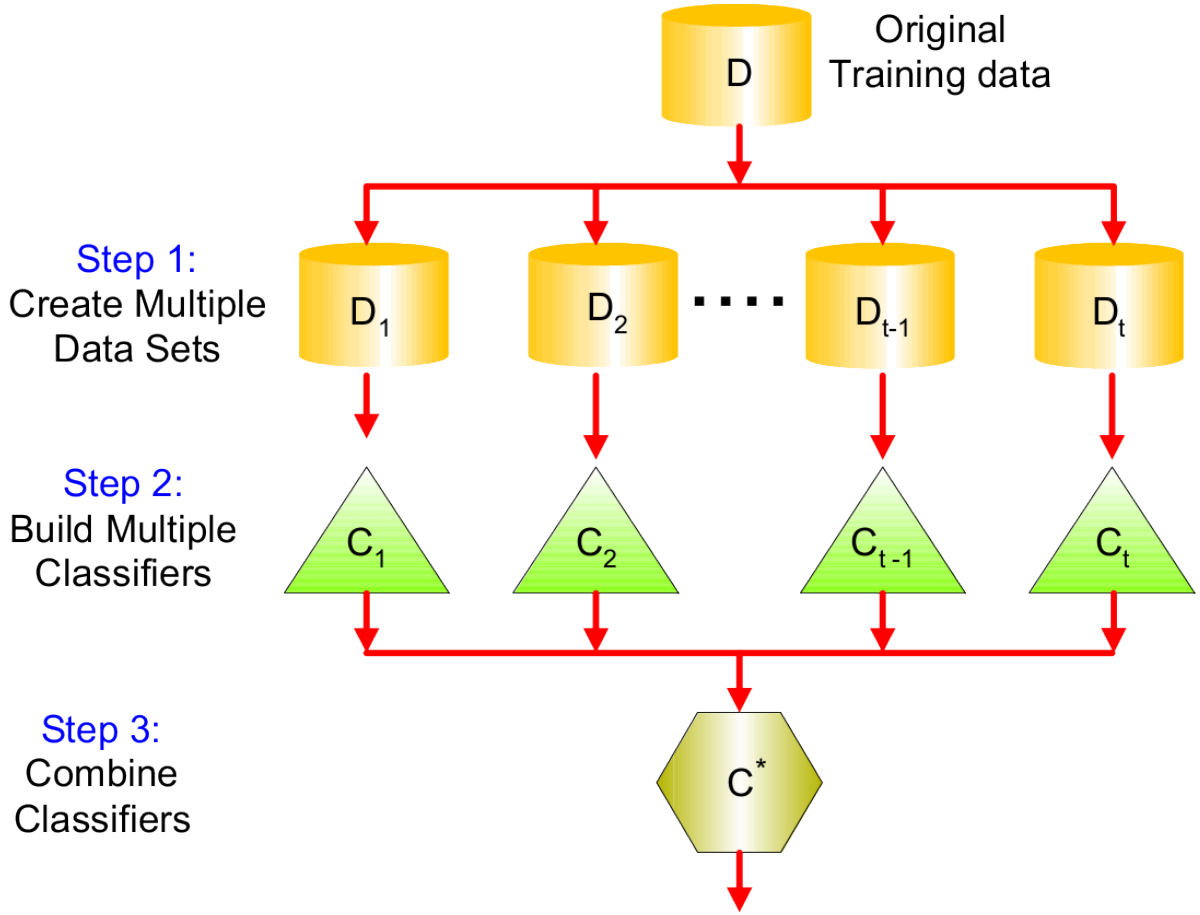
As before: We need to define an impurity measure $Q(S)$.
Let p_{Sk} be the fraction of points belonging to class k in S.

1. Misclassification error: Let $\hat{y} = \text{argmax}_k\, p_{Sk}$ be the dominant class in S, then $Q(S) = \frac{1}{|S|} \sum_{(\mathbf{x}_j, y_j) \in S} \mathbb{I}(y_j \neq \hat{y})$
2. Gini index: $Q(S) = \sum_{k=1}^{K} p_{Sk}(1 - p_{Sk})$
3. ...



Outline

General Idea



"Predict class label of previously unseen records by aggregating predictions made by multiple classifiers. Each of the classifiers is built in a slightly different way!"

http://www-users.cs.umn.edu/~kumar/dmbook

Bagging

http://www-users.cs.umn.edu/~kumar/dmbook

Original Data	1	2	3	4	5	6	7	8	9	10
Bagging (Round 1)	7	8	10	8	2	5	10	10	5	9
Bagging (Round 2)	1	4	9	1	2	3	2	7	3	2
Bagging (Round 3)	1	8	5	10	5	5	9	6	3	7

Sampling with replacement

- Generate multiple training set instances of size n by sampling with replacement. These samples are called bootstrap samples.
- Each bootstrap sample contains n instances of the data set.
- The probability for each sample to be selected at least once is $1 - \left(1 - \frac{1}{n}\right)^n \to 1 - \frac{1}{e} \approx 0.632$. Hence, a bootstrap sample only contains about 63% distinct examples.
- Samples not used for training are called out-of-bag (OOB) samples.
- Next: Build one classifier on top of each bootstrap sample.

Random Forests

Construction

Procedure: BUILDRF(S, m, d_{split}, B)

Require: $S = \{(\mathbf{x}_1, y_1), \ldots\}$, number m, number d_{split} of features to be tested per split, and number B of trees.

Ensure: Trees $\mathcal{T}_1, \ldots, \mathcal{T}_B$ for trees.

1: **for** $b = 1, \ldots, B$ **do**
2: Draw bootstrap sample S' by drawing $|S|$ elements with replacement from S.
3: $\mathcal{T}_b$ = BUILDRFTREE(S', m, d_{split})
4: **end for**
5: **return** $\mathcal{T}_1, \ldots, \mathcal{T}_B$

1. Regression: $f_{\text{RF}}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} \mathcal{T}_b(\mathbf{x})$
2. Classification: $f_{\text{RF}}(\mathbf{x}) =$ majority vote among $\mathcal{T}_1, \ldots, \mathcal{T}_B$

Breiman. Random Forests. *Machine Learning* 45, 2001

Random Forests: Tree Construction

Recursive RF Tree Construction

Procedure: BUILDRFTREE(S, m, d_{split})

Require: $S = \{(\mathbf{x}_1, y_1), \ldots\}$, number m (usually $m = 1$), and number d_{split} of features to be tested per split.

Ensure: Tree $\mathcal{T}$ built for the input patterns in S.

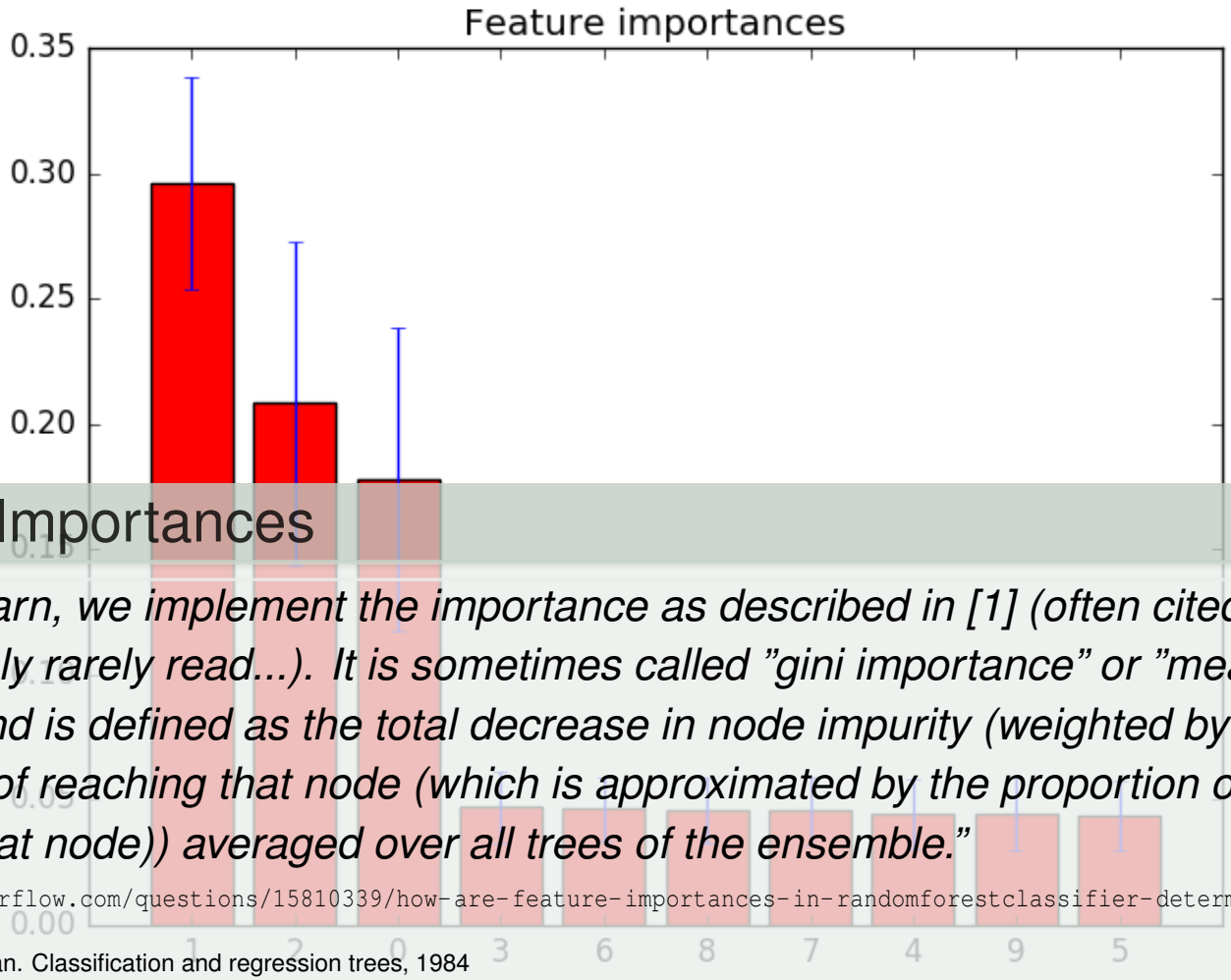
1: **if** $|S| \leq m$ **then**
2: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
3: **end if**
4: **if** $y_i = y_j$ for all $(\mathbf{x}_i, y_i), (\mathbf{x}_j, y_j) \in S$ **then**
5: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
6: **end if**
7: Find $(i^*, \theta^*) = \text{argmax}_{i,\theta}\, G_{i,\theta}(S)$ for randomly sampled $i \in \{i_1, \ldots, i_{d_{\text{split}}}\} \subseteq \{1, \ldots, d\}$ [only test d_{split} random dimensions]
8: $\mathcal{T}_l$ = BUILDRFTREE($L_{i,\theta}$, m, d_{split})
9: $\mathcal{T}_r$ = BUILDRFTREE($R_{i,\theta}$, m, d_{split})
10: Generate node ((i^*, θ^*) and pointers)
11: **return** resulting tree $\mathcal{T}$

Nice Properties of Random Forests

1. Aggregation of multiple different trees leads to a reduction of the random forest's variance.
2. Random forests are very robust w.r.t. hyperparameter choice.
3. OOB samples allow for rough estimation of generalization performance and even to compute generalization bounds.

   Lorenzen, Igel, Seldin. On PAC-Bayesian Bounds for Random Forests. *Machine Learning* 108, 2019
4. By searching for "good" splitting features at each internal node, irrelevant features are ignored.
5. Each tree induces a hierarchical subdivision of the feature space and the model "adapts" to the induced subregions (e.g., the feature selection takes place at each internal node).
6. Due to their hierarchical structure, random forests are capable of effectively dealing with (very) unbalanced datasets.
7. They are invariant under affine transformations of individual real-valued dimensions.
8. ...

Random Forests: Feature Selection



Feature Importances

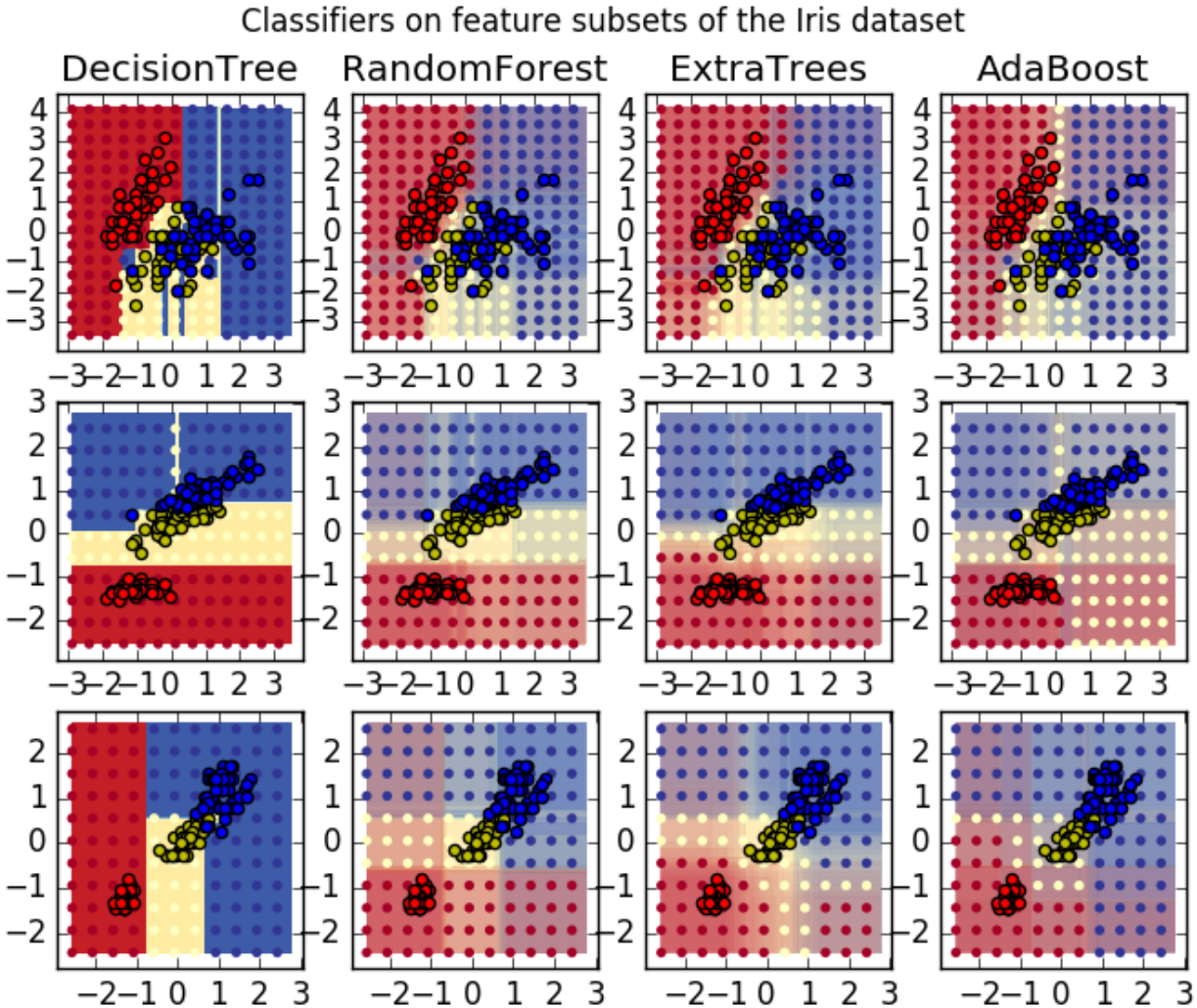
"In scikit-learn, we implement the importance as described in [1] (often cited, but unfortunately rarely read...). It is sometimes called "gini importance" or "mean decrease impurity" and is defined as the total decrease in node impurity (weighted by the probability of reaching that node (which is approximated by the proportion of samples reaching that node)) averaged over all trees of the ensemble." GillesLouppe

https://stackoverflow.com/questions/15810339/how-are-feature-importances-in-randomforestclassifier-determined

[1] Breiman, Friedman. Classification and regression trees, 1984

http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html

Tree Ensembles



http://scikit-learn.org/stable/modules/ensemble.html

Random Forest $\rightarrow$ Extra Trees

Exta Trees

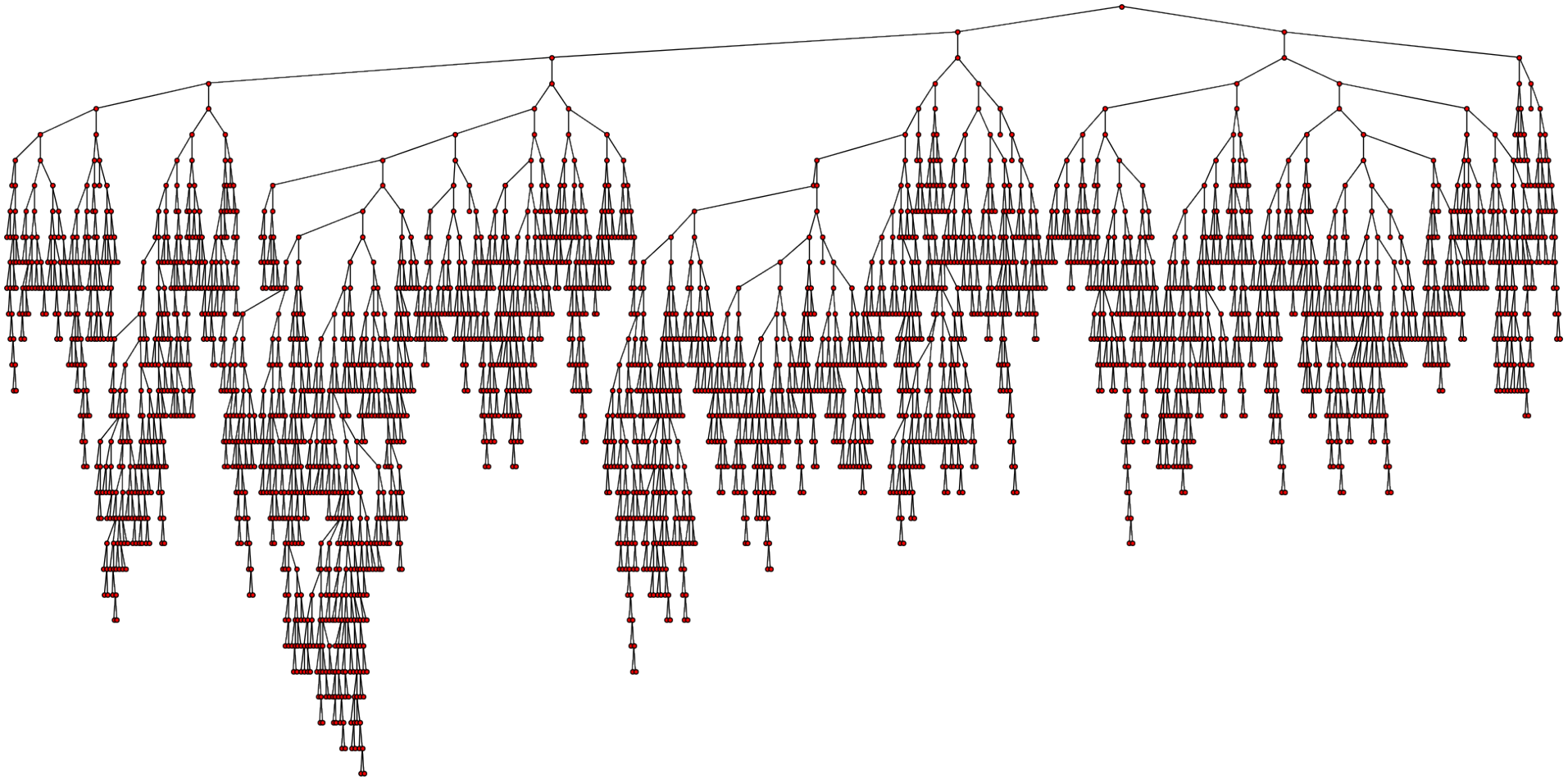
Basically the same as random forests. Two small modifications:

1. Do not draw bootstrap samples.
2. Use random threshold $\hat{\theta} \in [\min_{\mathbf{x} \in S} x_i, \max_{\mathbf{x} \in S} x_i]$ per dimension i.

Why does this still work?

Answer: We still compute the qualities of the random splits and select the best-performing one. We introduce more randomness, which can be helpful in practice.

Big Trees?



Random Forest, Extra Trees, ...

Recursive Tree Construction

Procedure: BUILDRFTREE(S,m,d_{split})

Require: $S = \{(\mathbf{x}_1, y_1), \ldots\}$, number m (usually $m = 1$), and number f of features to be tested per split.

Ensure: Tree $\mathcal{T}$ built for the input patterns in S.

1: **if** $|S| \leq m$ or $y_i = y_j$ for all $(\mathbf{x}_i, y_i), (\mathbf{x}_j, y_j) \in S$ **then**
2: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
3: **end if**
4: Find "good" splitting dimension/threshold (i^*, θ^*) checking d_{split} features
5: $\mathcal{T}_l$ = BUILDRFTREE($L_{i,\theta}$, m, d_{split})
6: $\mathcal{T}_r$ = BUILDRFTREE($R_{i,\theta}$, m, d_{split})
7: Generate node ((i^*, θ^*) and pointers). Let $\mathcal{T}$ be the resulting tree.
8: **return** $\mathcal{T}$

Computational Complexities

Single RF Tree

Let $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$ be a training set. We need

$$T(|S|) = C(|S|) + T(|L_{i,\theta}|) + T(|R_{i,\theta}|) \tag{1}$$

time to build a single tree, where $T(N)$ denotes the time to build a tree with N elements and $C(N)$ the time spent at each node wih N elements.
Naturally: $T(m) = c_1$.

1. Random forests: As shown before, one can sort the instances per split to achieve a runtime of $C(N) = O(d_{\text{split}} \cdot N \log N)$ per node split.
2. Extra trees: We only need $C(N) = O(d_{\text{split}} \cdot N)$ time per node split. Why?

The final structure of a tree, and, hence, its construction time, depends on the particular training data (we also stop in case the current set is pure).

Question: What is $T(|S|)$ in the best case for random forests?

Best case: We might be able to stop after having computed the impurity $Q(S)$ for S (root node). Hence: $T(|S|) = C(|S|) \in O(d_{split} \cdot |S| \log |S|)$. With additional check (line 1): $T(|S|) \in O(|S|)$.

Computational Complexities: Worst-Case

Single RF Tree: Worst-Case

Let $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$ be a training set. In the worst case, we reduce the problem size by only 1 per node split:

$$\begin{aligned} T(|S|) &= C(|S|) + T(1) + T(|S|-1) \\ &= C(|S|) + T(1) + C(|S|-1) + T(1) + T(|S|-2) \\ &= \ldots \\ &= \sum_{j=1}^{|S|} C(j) + |S| \cdot T(1) \end{aligned}$$

Let us consider Extra Trees: In this case, we have $T(1) \leq c_1$ and $C(j) \leq c_2 \cdot j$ for two constants c_1 and c_2. Therefore:

$$T(|S|) \leq \sum_{j=1}^{|S|} c_2 \cdot j + |S| \cdot c_1 = c_2 \frac{|S| \cdot (|S|+1)}{2} + c_1 |S|$$

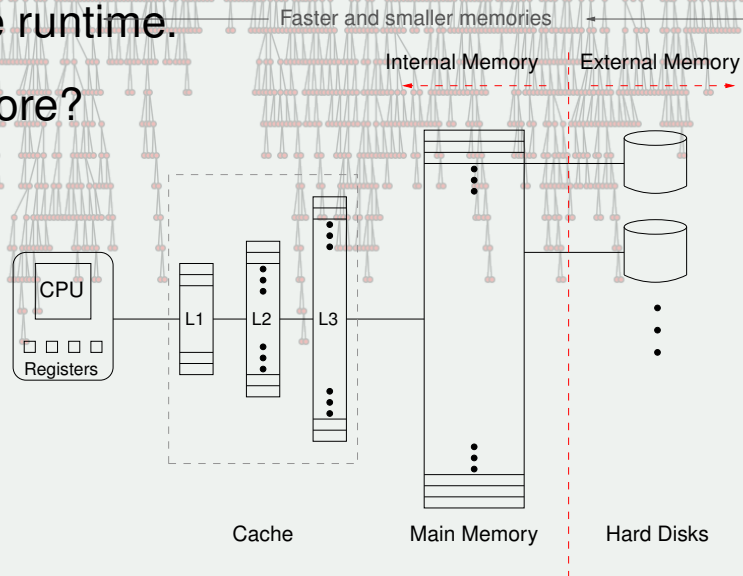
for some constant c_3. Hence, one can build a tree in $T(|S|) \in O(|S|^2)$ time.

Big Data & Trees

Practical Issues

For many tasks, it is beneficial to train forests with many trees (e.g., 1000 trees) using all the data (e.g., $n = 10^7$). This can become very tricky and is subject of ongoing research.

1. Training big trees can take a very long time.
2. One cannot easily make use of GPUs to reduce runtime.
3. Often, the data do not fit in main memory anymore?
4. At some point: The trees do not fit in main memory anymore ...
5. If the data/trees fit in main memory: The movement of the data from main memory to CPU might become the bottleneck.
   (the patterns are accessed in an arbitrary way)
6. Not easy to build forests in a distributed fashion (Hadoop/Spark).
   (building a single tree might already be an issue)
7. ...



Scikit-Learn

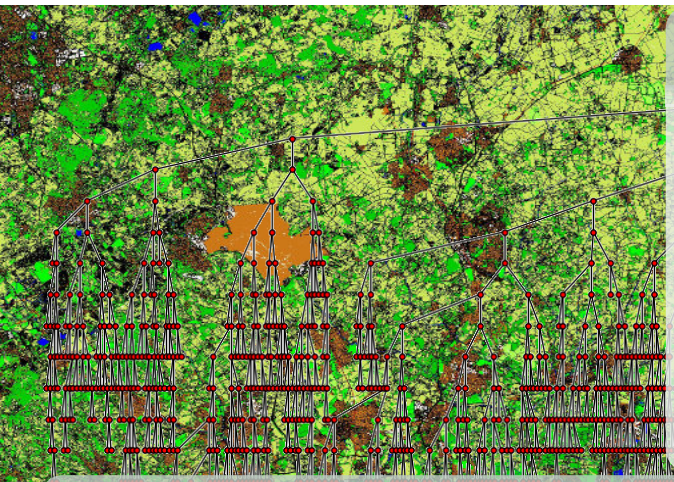
Implementation Hacks!

The Scikit-Learn implementation is extremely efficient as long as the data/trees fit in main memory. The code is written in Cython (hence: C code). Various "hacks" are used:

1. Trees & Arrays: Each tree is stored in an array (low-level, C-like)
2. Sorting: Optimized sorting algorithm (standard `qsort` $\rightarrow$ factor 2-3 slower)
3. Locally constant features: Keep track of "constant" features during construction.
4. Unique instances: If `bootstrap=True`, only use unique indices/patterns (only 63%) and assign weights to them $\rightarrow$ Speed-up of about 1.5!
5. Consecutive memory access: For evaluating $\mathcal{O}(S)$, prefetch data for dimension i
6. Sparse data: Make use of optimized computations for sparse data.
7. Random numbers: Use manual random number generator ...
8. Parallelization: For random forests, efficient parallelization over trees.
9. ...

In a nutshell: This implementation might easily be a factor of 10-100 faster than a "standard" implementation in C. Speed-up depends on the particular dataset ...

Woody: Large-Scale Random Forests



Algorithmic Workflow

1. Build top tree on small random subset.
2. Distribute all points to the leaves of the top tree.
   (store chunks on disk in a compressed manner)
3. Build (multiple) bottom trees for each of the leaf chunks.
   (constant-sized leaf chunks → linear time)

Training Output

```
13:51:20,588 - Number of training patterns:     113212922
13:51:20,588 - Dimensionality of the data:      11
13:51:20,588 - Fitting forest ...
...
13:59:52,908 - ------------------------------------------------------------------------
13:59:52,908 - (I)   Retrieving subsets:                    53.034 (s)     [10.35  %]
13:59:52,909 - (II)  Top tree constructions:                53.034 (s)     [10.35  %]
13:59:52,909 - (III) Distributing to top tree leaves:       240.208 (s)    [46.89  %]
13:59:52,909 - (IV)  Bottom trees constructions:            219.071 (s)    [42.76  %]
13:59:52,909 -                                              512.312 (s)    [100.00 %]
13:59:52,909 - ------------------------------------------------------------------------
13:51:20,588 - Training time:          512.335904
```

Outline

Boosting

Michael Kearns, 1988

Informally, [the hypothesis boosting] problem asks whether an efficient learning algorithm [. . .] that outputs a hypothesis whose performance is only slightly better than random guessing [i.e. a weak learner] implies the existence of an efficient algorithm that outputs a hypothesis of arbitrary accuracy [i.e. a strong learner].

We build the ensemble incrementally, adding a new base classifier in each step. Two popular strategies are, roughly speaking:

AdaBoost: Train the next base classifier on weighted training examples, where more weight is put on examples that have been wrongly classified so far

XGBoost: Add a base classifier such that the update can be interpreted as a gradient step minimizing the loss

Chen, Guestrin. XGBoost: A Scalable Tree Boosting System. *KDD*, 2016

Freund, Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences* 55, 1997

AdaBoost Algorithm for Binary Classification

Require: $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in (\mathbb{R}^d \times \{-1, 1\})^n$, T of boosting rounds

1: $\mathbf{w} = (\frac{1}{n}, \ldots, \frac{1}{n}) \in \mathbb{R}^n$ [initially same weight for all examples]
2: **for** $t = 1, \ldots, T$ **do**
3: Train $h^{(t)}$ to minimize $\sum_{j=1}^{n} w_j \mathbb{I}(h^{(t)}(\mathbf{x}_j) \neq y_j)$ [base classifier]
4: $\varepsilon^{(t)} \leftarrow \sum_{j=1}^{n} w_j \mathbb{I}(h^{(t)}(\mathbf{x}_j) \neq y_j)$ [compute error]
5: Ensure $\varepsilon^{(t)} < 0.5$ [e.g., invert classifier]
6: $\alpha^{(t)} \leftarrow \frac{1}{2} \ln\left(\frac{1-\varepsilon^{(t)}}{\varepsilon^{(t)}}\right)$ [determine importance of classifier]
7: **for** $i = 1, \ldots, n$ **do**
8: **if** $h^{(t)}(\mathbf{x}_i) \neq y_i$ **then**
9: $w_i^{(t+1)} \leftarrow w_i^{(t)} / \left(2\varepsilon^{(t)}\right)$ [if wrong, increase weight]
10: **else**
11: $w_i^{(t+1)} \leftarrow w_i^{(t)} / \left(2\left(1-\varepsilon^{(t)}\right)\right)$ [if correct, decrease weight]
12: **end if**
13: **end for**
14: $f^{(t)} = \sum_{i=1}^{t} \alpha^{(i)} h^{(i)}$ [decision function at step t]
15: **end for**

Final model: $h(\mathbf{x}) = \operatorname{sgn}\left(f^{(T)}(\mathbf{x})\right)$ $[f^{(T)}(\mathbf{x}) = \sum_{t=1}^{T} \alpha^{(t)} h^{(t)}(\mathbf{x})]$

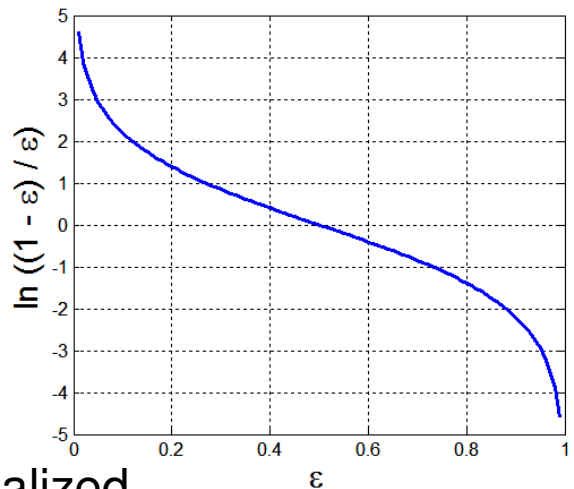
AdaBoost

- The error rate $\varepsilon^{(t)}$ for a $h^{(t)}$ is given by

$$\varepsilon^{(t)} = \sum_{j=1}^{n} w_j \mathbb{I}(h^{(t)}(\mathbf{x}_j) \neq y_j)$$

- The importance $\alpha^{(t)}$ of a classifier is defined as

$$\alpha^{(t)} = \frac{1}{2} \ln \left(\frac{1 - \varepsilon^{(t)}}{\varepsilon^{(t)}} \right)$$



- The weights are normalized

$$\sum_{i=1}^{n} w_i^{(t)} = 1$$

AdaBoost: Rewriting weight update

The weight update can be rewritten as:

$$
\begin{aligned}
w_i^{(t+1)} &= w_i^{(t)} \frac{\exp\left(-\alpha^{(t)} h^{(t)}(\mathbf{x}_i) y_i\right)}{2\sqrt{\varepsilon^{(t)}\left(1-\varepsilon^{(t)}\right)}} \\
&= \frac{\exp\left(-f^{(t)}(\mathbf{x}_i) y_i\right)}{n\prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}} \\
&= \frac{\exp\left(-f^{(t)}(\mathbf{x}_i) y_i\right)}{\sum_{l=1}^{n} \exp\left(-f^{(t)}(\mathbf{x}_l) y_l\right)}
\end{aligned}
$$

It follows:

$$
\frac{1}{n}\sum_{l=1}^{n} \exp\Big(-\underbrace{f^{(t)}(\mathbf{x}_l) y_l}_{\text{margin}}\Big) = \prod_{\tau=1}^{t} 2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}
$$

AdaBoost: Error minimization

Let's bound empirical the 0-1 loss:

$$\frac{1}{n}\sum_{j=1}^{n}\mathbb{I}\left(\operatorname{sgn}\left(f^{(T)}(\mathbf{x}_j)\right)\neq y_j\right)=\frac{1}{n}\sum_{j=1}^{n}\mathbb{I}\left(f^{(T)}(\mathbf{x}_j)y_j<0\right)$$

$$\leq\frac{1}{n}\sum_{j=1}^{n}\exp\left(-f^{(T)}(\mathbf{x}_j)y_j\right)\qquad[\mathbb{I}(x<0)\leq e^{-x}]$$

$$=\prod_{\tau=1}^{T}2\sqrt{\varepsilon^{(\tau)}\left(1-\varepsilon^{(\tau)}\right)}=\prod_{\tau=1}^{T}\sqrt{1-\left(1-2\varepsilon^{(\tau)}\right)^2}$$

assuming $\varepsilon^{(\tau)}\leq\frac{1}{2}-\delta$ for positive δ

$$\leq\left(\sqrt{1-4\delta^2}\right)^T$$

$$\leq\exp\left(-2T\delta^2\right)\qquad[1-x\leq e^{-x}]$$

After at most $\lceil\frac{\ln n}{2\delta^2}\rceil+1$ steps the bound is smaller than $\frac{1}{n}$ and thus the 0-1 loss is zero

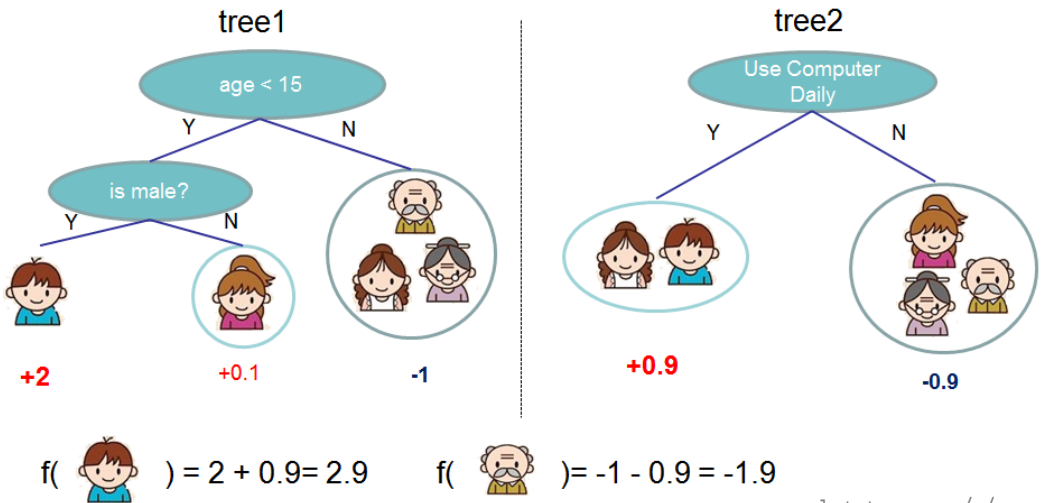
XGBoost

Task

- Training data $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$
- The final model $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ is based on K trees $f_k \in \mathcal{F}$, where $\mathcal{F}$ is the space of decision trees.
- Learning goal: Find K such trees that minimize the objective

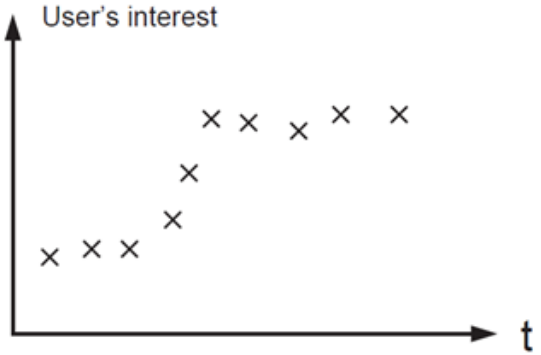
$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

where $\mathcal{L}$ is a loss function and $\Omega(f_k)$ a measure for the complexity of a tree.

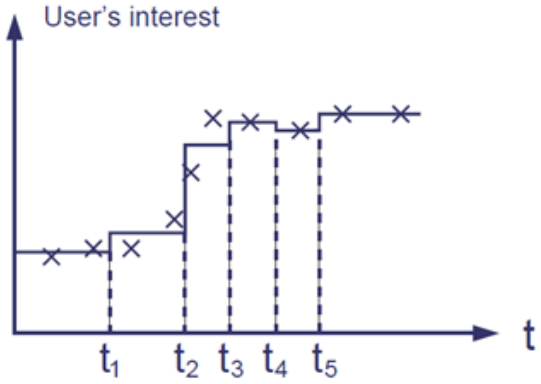


https://xgboost.readthedocs.io

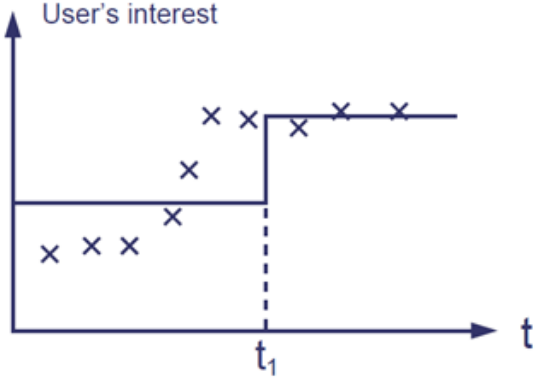
Trade-Off: Small Loss vs. Complexity



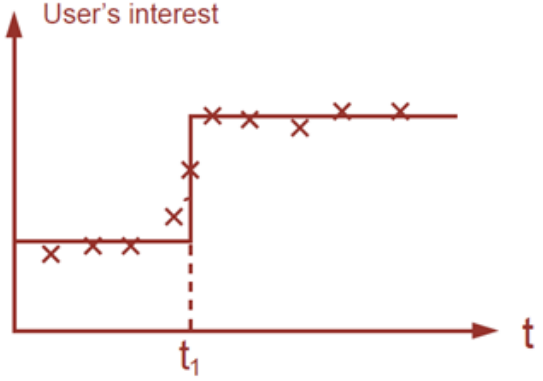
Observed user's interest on topic k against time t



☒ Too many splits, $\Omega(f)$ is high



☒ Wrong split point, $L(f)$ is high



☑ Good balance of $\Omega(f)$ and $L(f)$

https://xgboost.readthedocs.io

XGBoost: Goal & Training

Objective

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_k)$ specifying the complexity of the tree f_k.

Comment: It is, in general, difficult to find a "global" optimal solution.

Additive Training (Boosting): Start with constant predictions and add a new function (tree) in each iteration:

1. $\hat{y}_i^{(0)} = h^{(0)}(\mathbf{x}_i) = 0$
2. $\hat{y}_i^{(1)} = h^{(1)}(\mathbf{x}_i) = f_1(\mathbf{x}_i) = \hat{y}_i^{(0)} + f_1(\mathbf{x}_i)$
3. $\hat{y}_i^{(2)} = h^{(2)}(\mathbf{x}_i) = f_1(\mathbf{x}_i) + f_2(\mathbf{x}_i) = \hat{y}_i^{(1)} + f_2(\mathbf{x}_i)$
4. ...
5. $\hat{y}_i^{(t)} = h^{(t)}(\mathbf{x}_i) = \sum_{k=1}^{t} f_k(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)$

Gradient Boosting I

XGBoost is a gradient boosting algorithm. Let's discuss general gradient boosting in the following (ignoring Ω for simplicity). Under the assumption that the predictions are independent, optimizing the predictions directly

$$\text{argmin}_{\hat{y}_1,\ldots,\hat{y}_n} \sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) \qquad \text{[we ignore that the solution is obvious]}$$

is the same as solving

$$\text{argmin}_{\hat{y}_i} \mathcal{L}(y_i, \hat{y}_i)$$

for each $i = 1, \ldots, n$.

Now assume we do so by iterative steepest descent. If we do a gradient step with learning rate η, we get for the update in iteration t:

$$\hat{y}_i^{(t+1)} = \hat{y}_i^{(t)} - \eta \nabla \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right)$$

with $\nabla \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right) = \frac{\partial \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right)}{\partial \hat{y}_i^{(t)}}$.

Gradient Boosting II

Now write

$$\hat{y}_i^{(t+1)} = \hat{y}_i^{(t)} - \eta \nabla \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right)$$

as

$$\hat{y}_i^{(t+1)} = \underbrace{h^{(t)}(\mathbf{x}_i)}_{\sum_{k=1}^{t} f_k(\mathbf{x}_i)} + \eta \underbrace{f^{(t+1)}(\mathbf{x}_i)}_{-\nabla \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right)} .$$

Thus, we want $f^{(t+1)}(\mathbf{x}_i)$ to be $-\nabla \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right)$ for all i.

Thus, we train the new $f^{(t+1)}$ on the training data set

$$\left\{\left(\mathbf{x}_1, -\nabla \mathcal{L}\left(y_1, \hat{y}_1^{(t)}\right)\right), \ldots, \left(\mathbf{x}_n, -\nabla \mathcal{L}\left(y_n, \hat{y}_n^{(t)}\right)\right)\right\} .$$

The targets $-\nabla \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right)$ are also called pseudo-residuals.

XGBoost: In a Nutshell

Objective and Parameters

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$, where T is the number of leaves .

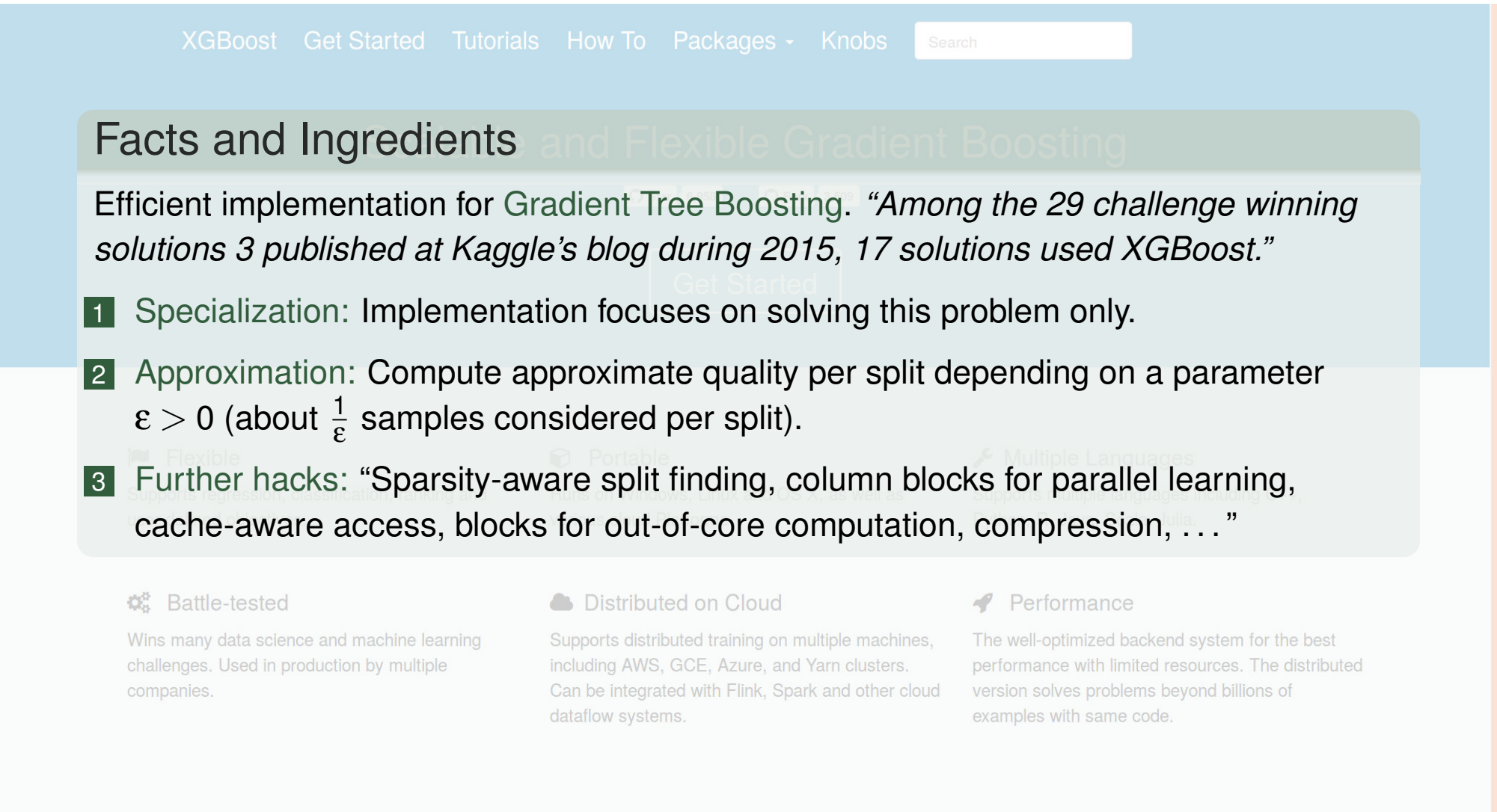
- K specifies the number of trees to be fitted (e.g., 100).
- One can restrict the trees to have a maximum depths (e.g., 5).
- One usually specifies a learning rate η (default $\eta = 0.3$):

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta \cdot f_t(\mathbf{x}_i)$$

- One can adapt γ and λ (see above)
- One can consider column-subsampling (proportion of features to be checked per tree (e.g., 0.3).
- ...

https://xgboost.readthedocs.io/en/latest/python/python_api.html

XGBoost: Implementation



Facts and Ingredients

Efficient implementation for Gradient Tree Boosting. *"Among the 29 challenge winning solutions 3 published at Kaggle's blog during 2015, 17 solutions used XGBoost."*

1. Specialization: Implementation focuses on solving this problem only.
2. Approximation: Compute approximate quality per split depending on a parameter $\varepsilon > 0$ (about $\frac{1}{\varepsilon}$ samples considered per split).
3. Further hacks: "Sparsity-aware split finding, column blocks for parallel learning, cache-aware access, blocks for out-of-core computation, compression, . . . "

Tianqi Chen and Carlos Guestrin. *XGBoost: A Scalable Tree Boosting System*, KDD, 2016.

Outline

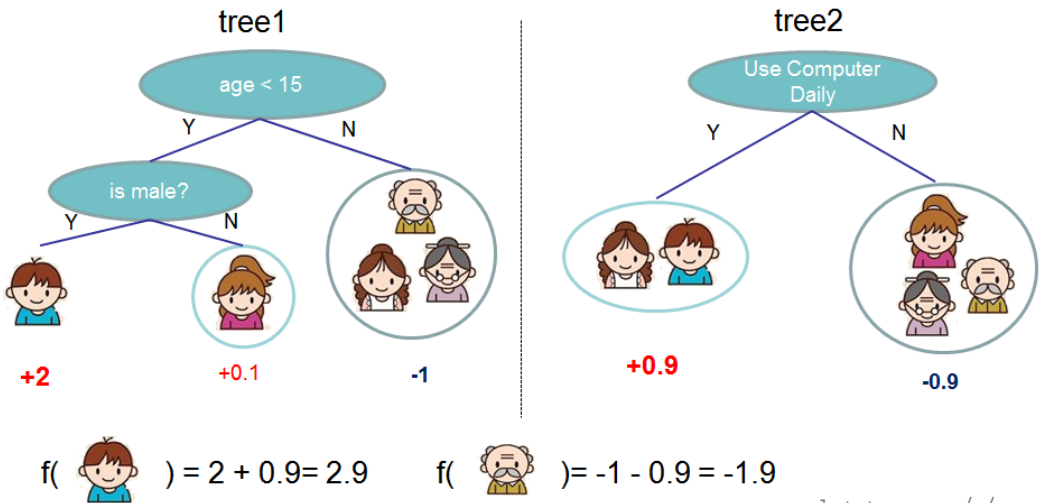
Recall: XGBoost

Task

- Training data $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$
- The final model $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ is based on K trees $f_k \in \mathcal{F}$, where $\mathcal{F}$ is the space of decision trees.
- Learning goal: Find K such trees that minimize the objective

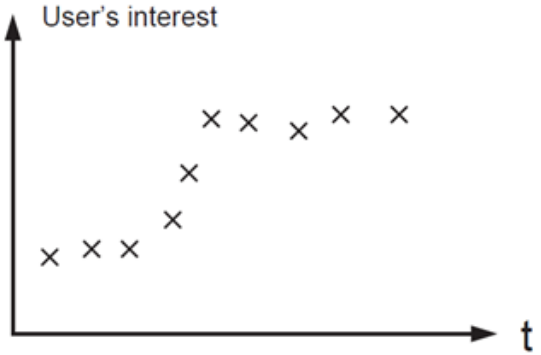
$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

where $\mathcal{L}$ is a loss function and $\Omega(f_k)$ a measure for the complexity of a tree.

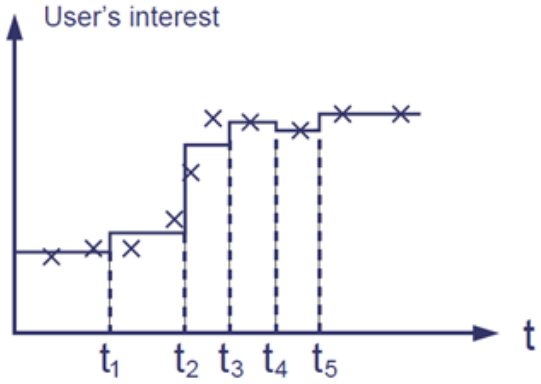


https://xgboost.readthedocs.io

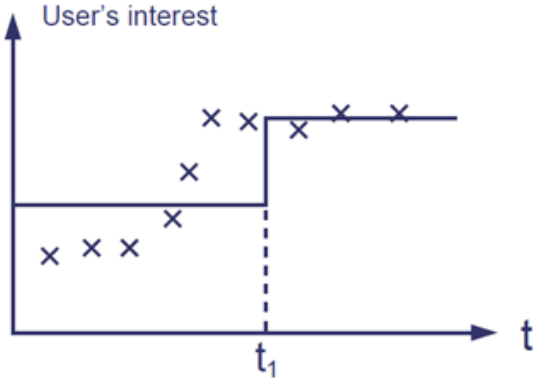
Trade-Off: Small Loss vs. Complexity



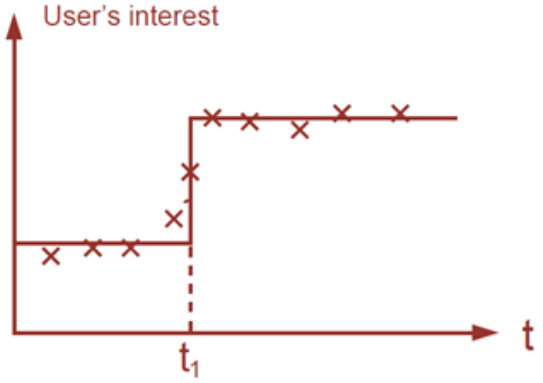
Observed user's interest on topic k against time t



☒ Too many splits, $\Omega(f)$ is high



☒ Wrong split point, $L(f)$ is high



☑ Good balance of $\Omega(f)$ and $L(f)$

https://xgboost.readthedocs.io

Recall: Goal & Training of XGBoost

Objective

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_k)$ specifying the complexity of the tree f_k.

Comment: It is, in general, difficult to find a "global" optimal solution.

Additive Training (Boosting): Start with constant predictions and add a new function (tree) in each iteration:

1. $\hat{y}_i^{(0)} = 0$
2. $\hat{y}_i^{(1)} = f_1(\mathbf{x}_i) = \hat{y}_i^{(0)} + f_1(\mathbf{x}_i)$
3. $\hat{y}_i^{(2)} = f_1(\mathbf{x}_i) + f_2(\mathbf{x}_i) = \hat{y}_i^{(1)} + f_2(\mathbf{x}_i)$
4. ...
5. $\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)$

Simplifying the Objective I

- At each step t, we have $\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)$
  - Keep functions $\hat{y}_i^{(t-1)}$ that were added in the previous rounds
  - Find new function $f_t(\mathbf{x}_i)$ that minimizes the objective
- We want to minimize

$$\begin{aligned}\mathcal{E}^{(t)} &= \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{k=1}^{t} \Omega(f_k) \\ &= \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)\right) + \Omega(f_t) + \text{constant}\end{aligned}$$

- Let us start with the square loss $\mathcal{L}(y, y') = (y - y')^2$. This yields:

$$\begin{aligned}\mathcal{E}^{(t)} &= \sum_{i=1}^{n} \left(y_i - (\hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i))\right)^2 + \Omega(f_t) + \text{constant} \\ &= \sum_{i=1}^{n} \left[2(\hat{y}_i^{(t-1)} - y_i) f_t(\mathbf{x}_i) + f_t(\mathbf{x}_i)^2\right] + \Omega(f_t) + \text{constant}\end{aligned}$$

Simplifying the Objective II

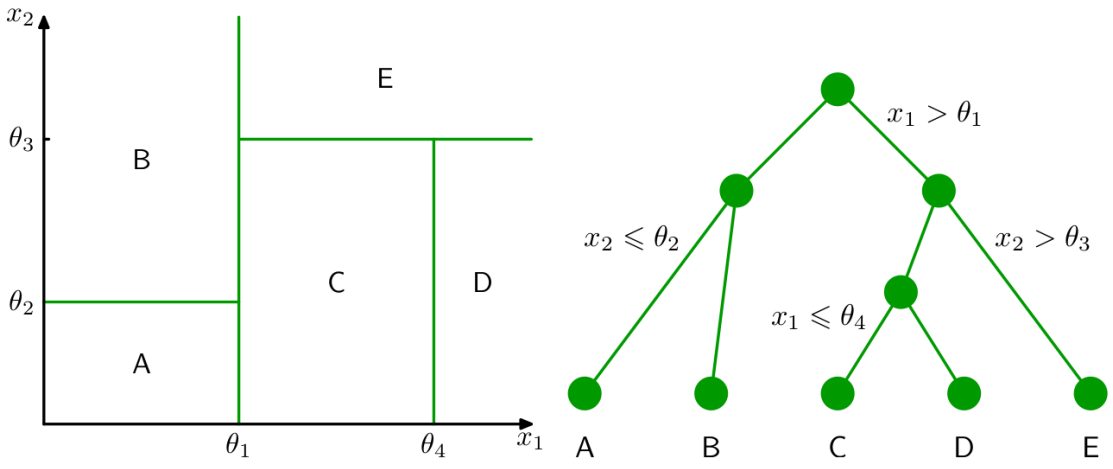
- Omitting the constants, we have the following objective to be minimized

$$\mathcal{E}^{(t)} = \sum_{i=1}^{n} \left[2(\hat{y}_i^{(t-1)} - y_i) f_t(\mathbf{x}_i) + f_t(\mathbf{x}_i)^2 \right] + \Omega(f_t)$$

- Let us define $g_i = 2(\hat{y}_i^{(t-1)} - y_i)$ and $h_i = 2$
- We can then rewrite the objective in the following way:

$$\mathcal{E}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t(\mathbf{x}_i)^2 \right] + \Omega(f_t)$$

Tree Representation



Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006

Structure and Weights

Each tree $f_t(\mathbf{x})$ with T leaves can be represented as

$$f_t(\mathbf{x}) = w_{q(\mathbf{x})}$$

with $q : \mathbb{R}^d \to \{1, 2, \ldots, T\}$ defining the partition induced by the tree and with $\mathbf{w} \in \mathbb{R}^T$ defining the leaf weights.

Simplifying the Objective III

- We can now regroup the objective by the leaves of the tree:

$$\begin{aligned}
\mathcal{E}^{(t)} &= \sum_{i=1}^{n} \left[g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t(\mathbf{x}_i)^2 \right] + \Omega(f_t) \\
&= \sum_{i=1}^{n} \left[g_i w_{q(\mathbf{x}_i)} + \frac{1}{2} h_i w_{q(\mathbf{x}_i)}^2 \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \\
&= \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T
\end{aligned}$$

  where we used $\Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2$.
  (other complexity measure are possible as well)

- Thus, the objective can be decomposed into a sum of T independent quadratic functions.

Computing the Structure Score

- For a function of the form $h(v) = Gv + \frac{1}{2}(H+\lambda)v^2$ with $H > 0$, we have

$$\operatorname*{argmin}_v h(v) = -\frac{G}{H+\lambda} \quad \text{and} \quad \min_v h(v) = -\frac{1}{2}\frac{G^2}{H+\lambda}$$

- By defininining $G_j := \sum_{i \in I_j} g_i$ and $H_j := \sum_{i \in I_j} h_i$, we obtain

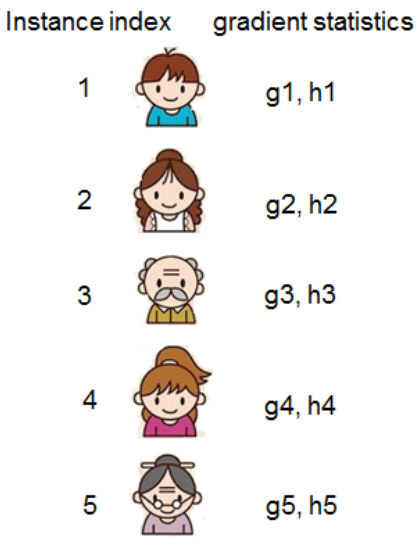
$$\mathcal{E}^{(t)} = \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2 \right] + \gamma T$$

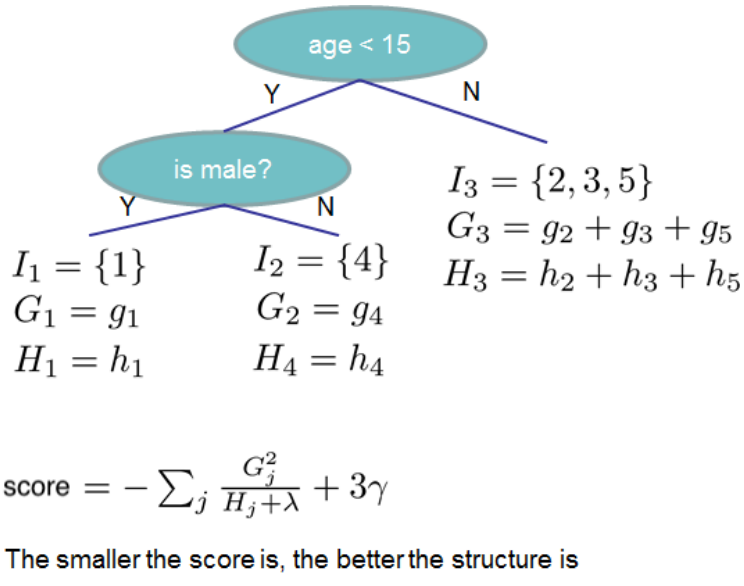
- Assuming that the structure of the tree is fixed, one can compute the optimal weight in each leaf as well as the overall optimal objective value:

Weights and Objective

$$w_j^* = -\frac{G_j}{H_j+\lambda} \qquad \mathcal{E}^{(t)} = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda} + \gamma T$$

XGBoost: Structure Score





$$\text{score} = -\sum_j \frac{G_j^2}{H_j + \lambda} + 3\gamma$$

The smaller the score is, the better the structure is

We now have a way to assess the quality of a given tree structure!

https://xgboost.readthedocs.io

Finding the Tree Structure: Approach I

- Idea: Enumerate all possible tree structures q!
- For a particular structure: Compute the objective ("quality") via

$$\mathcal{E}^{(t)} = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda} + \gamma T$$

- Keep track of the structure q with lowest objective ...
- Problem: In general, there are too many tree structures to check all of them.

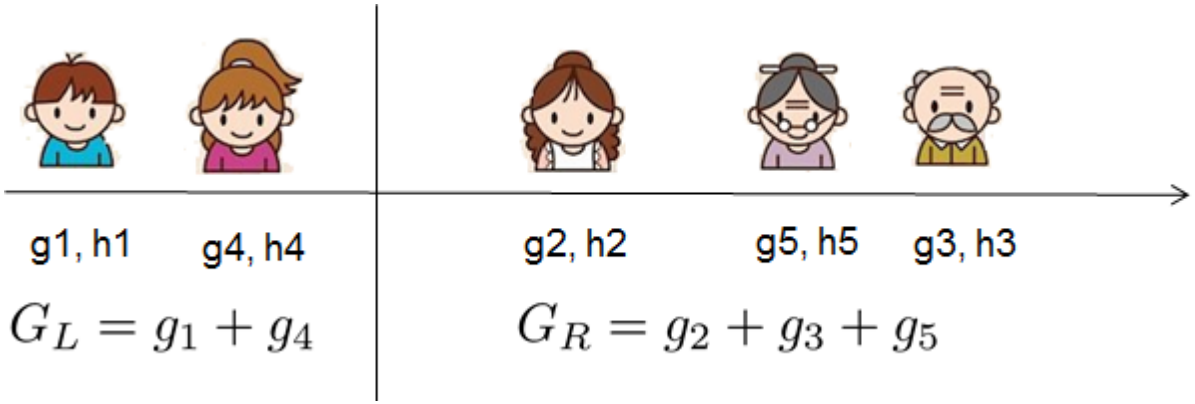
Finding the Tree Structure: Approach II

- New idea: Grow the tree in a greedy fashion:
  1. Start with a tree having depth 0
  2. At each leaf node, try to split. Use the following "gain" to assess the potential:

$$\underbrace{\frac{G_L^2}{H_L+\lambda}}_{\text{Score of left child}} + \underbrace{\frac{G_R^2}{H_R+\lambda}}_{\text{Score of right child}} - \underbrace{\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}}_{\text{Score of "no split"}} - \underbrace{\gamma}_{\text{Cost for +1 leaf}}$$

- Note: The gain can become negative.
- Stop if the best split has negative gain.
- Question: How do we find the best split efficiently?

Finding the Best Split



- What is the gain of doing a split of the form $x_j < a$?
- We need to compute the sum of all g and h values on each side ...
- Question: How can one compute all possible split gains efficiently?
- Observation: The scores do not change "in between" two data points.

1. Pre-sort the instances w.r.t. the j-th feature.
2. Scan the sorted list from left to right. While scanning, update G_L, H_L, G_R, H_R, and the induced gain.

https://xgboost.readthedocs.io

Other Loss Functions?

- At each step t, we have $\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)$
  - Keep functions that were added in the previous rounds: $\hat{y}_i^{(t-1)}$
  - Find new function $f_t(\mathbf{x}_i)$ that minimizes the objective
- We want to minimize

$$\begin{aligned} \mathcal{E}^{(t)} &= \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{k=1}^{t} \Omega(f_k) \\ &= \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)\right) + \Omega(f_t) + \text{constant} \end{aligned}$$

- We have used the square loss $\mathcal{L}(y, y') = (y - y')^2$ so far, which yielded:

$$\mathcal{E}^{(t)} = \sum_{i=1}^{n} \left[2(\hat{y}_i^{(t-1)} - y_i) f_t(\mathbf{x}_i) + f_t(\mathbf{x}_i)^2\right] + \Omega(f_t) + \text{constant}$$

- Problem: Other loss functions might not yield such a “nice” form.

Other Loss Functions?

- Objective: $\mathcal{E}^{(t)} = \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)\right) + \Omega(f_t) + \text{constant}$
- Idea: Use Taylor expansion of the objective!
  - Recall: $f(x+\Delta x) \approx f(x) + f'(x)\Delta x + \frac{1}{2} f''(x)\Delta x^2$
  - Define the first and second derivatives:

$$\begin{aligned} g_i &= \partial_{\hat{y}_i^{(t-1)}} \mathcal{L}(y_i, \hat{y}_i^{(t-1)}) \\ h_i &= \partial^2_{\hat{y}_i^{(t-1)}} \mathcal{L}(y_i, \hat{y}_i^{(t-1)}) \end{aligned}$$

  - We can then approximate the objective via

$$\begin{aligned} \mathcal{E}^{(t)} &\approx \sum_{i=1}^{n} \left[\mathcal{L}\left(y_i, \hat{y}_i^{(t-1)}\right) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t(\mathbf{x}_i)^2\right] + \Omega(f_t) + \text{constant} \\ &= \sum_{i=1}^{n} \left[g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t(\mathbf{x}_i)^2\right] + \Omega(f_t) + \text{constant} \end{aligned}$$

- Remaining steps: As before

Notation as in the paper. At this step, the $\hat{y}_i^{(t-1)}$ are constants (e.g., set $\hat{y}_i^{(t-1)} = F_i$).

XGBoost: In a Nutshell

Objective and Parameters

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

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

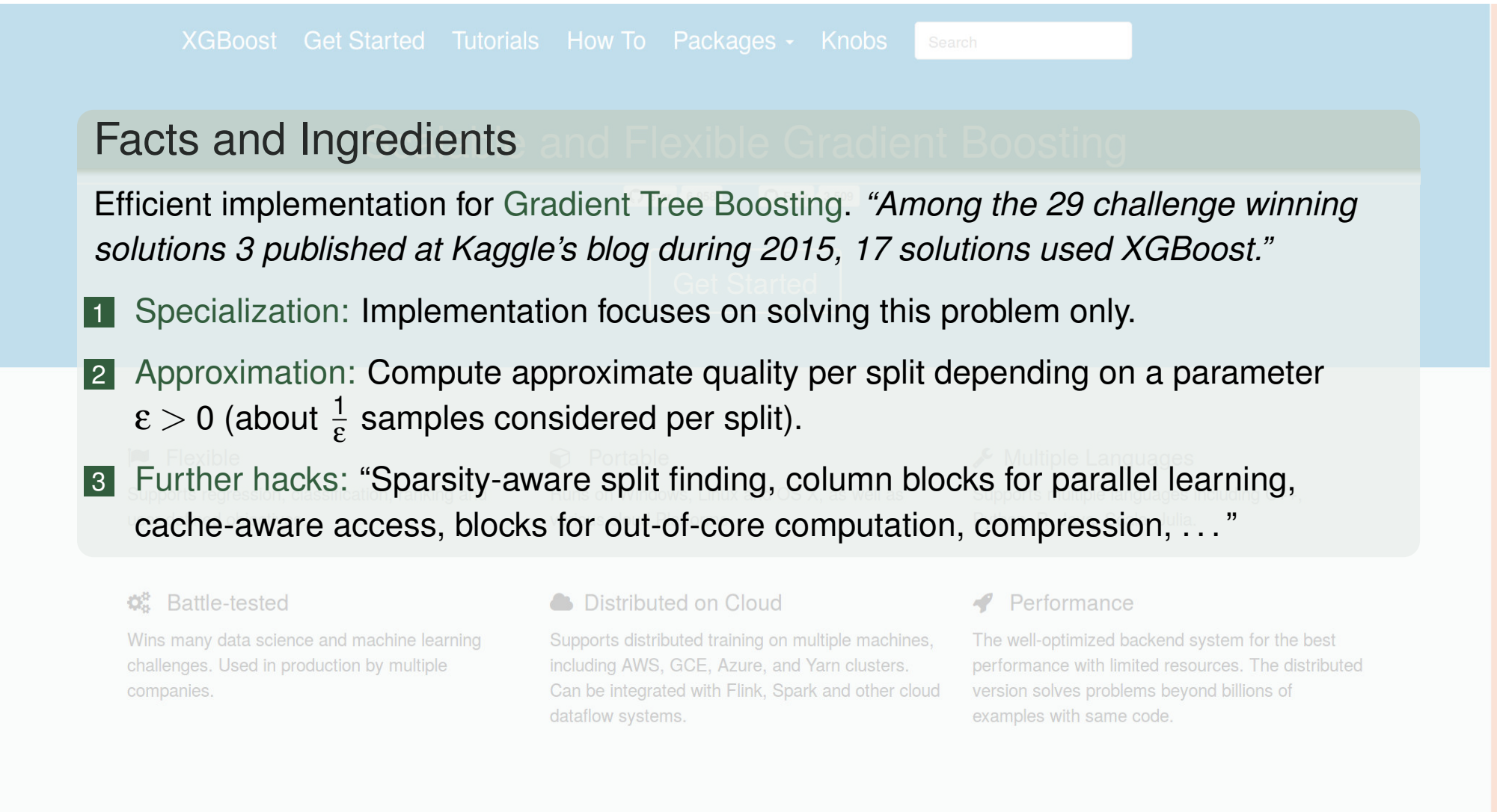
- K specifies the number of trees to be fitted (e.g., 100).
- One can restrict the trees to have a maximum depths (e.g., 5).
- One usually specifies a learning rate η (default $\eta = 0.3$):

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta \cdot f_t(\mathbf{x}_i)$$

- One can adapt γ and λ (see above)
- One can consider column-subsampling (proportion of features to be checked per tree (e.g., 0.3).
- ...

https://xgboost.readthedocs.io/en/latest/python/python_api.html

XGBoost: Implementation



Facts and Ingredients

Efficient implementation for Gradient Tree Boosting. *"Among the 29 challenge winning solutions 3 published at Kaggle's blog during 2015, 17 solutions used XGBoost."*

1. Specialization: Implementation focuses on solving this problem only.
2. Approximation: Compute approximate quality per split depending on a parameter $\varepsilon > 0$ (about $\frac{1}{\varepsilon}$ samples considered per split).
3. Further hacks: "Sparsity-aware split finding, column blocks for parallel learning, cache-aware access, blocks for out-of-core computation, compression, . . . "

Tianqi Chen and Carlos Guestrin. *XGBoost: A Scalable Tree Boosting System*, KDD, 2016.

Outline

Summary

Random forests and boosting are powerful methods, in particular for data without spatial or temporal structure.