Gradient Boosting Decision Trees

Milan Straka

December 06, 2021

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Gradient Boosting Decision Trees

The gradient boosting decision trees also train a collection of decision trees, but unlike random forests, where the trees are trained independently, in GBDT they are trained sequentially to correct the errors of the previous trees.

If we denote y_t as the prediction function of the $t^\textrm{th}$ tree, the prediction of the whole collection is then

$$y(\boldsymbol{x}_i) = \sum_{t=1}^T y_t(\boldsymbol{x}_i; \boldsymbol{W}_t),$$

where $\boldsymbol{W}_t$ is a vector of parameters (leaf values, to be concrete) of the $t^\textrm{th}$ tree.

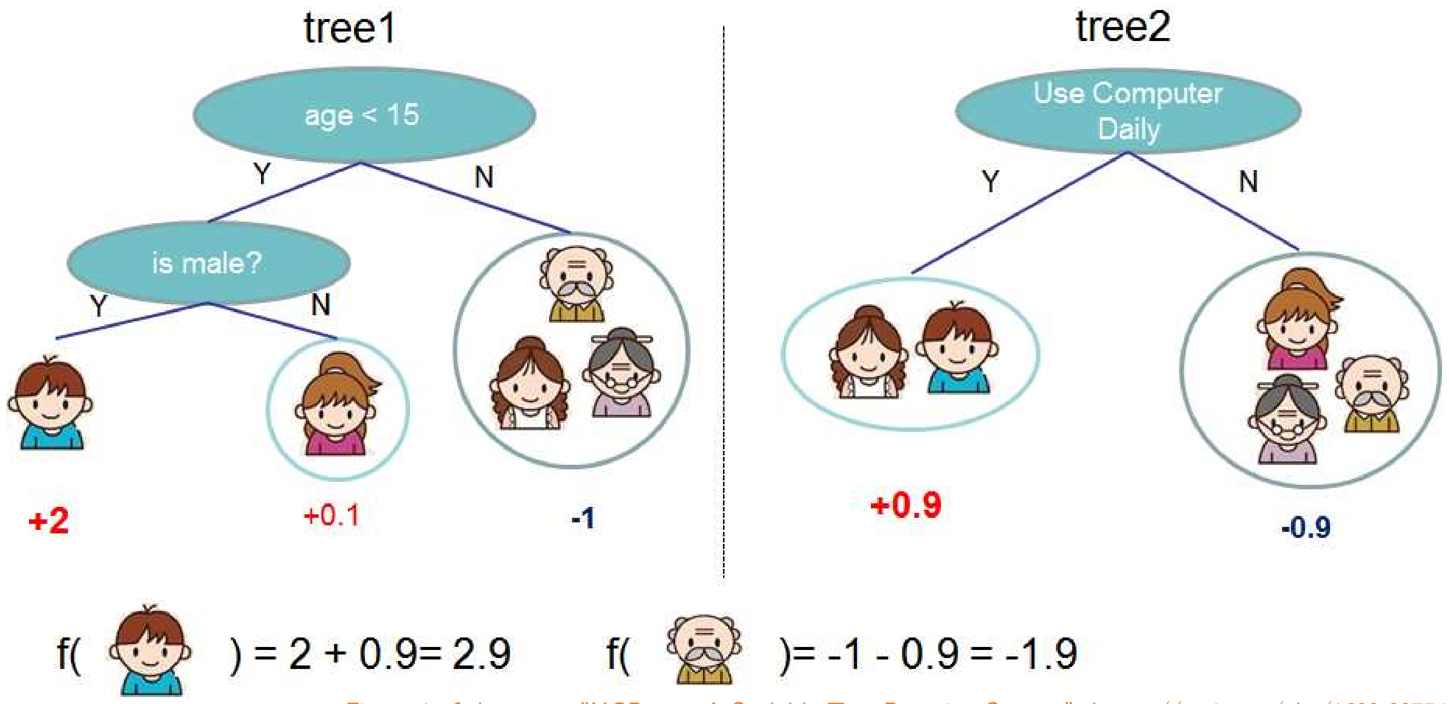


Figure 1 of the paper "XGBoost: A Scalable Tree Boosting System", https://arxiv.org/abs/1603.02754

Gradient Boosting for Regression

Considering a regression task first, we define the overall loss as

$$\mathcal{L}(\boldsymbol{W}) = \sum_i \ell\big(t_i, y(\boldsymbol{x}_i; \boldsymbol{W})\big) + \sum_{t=1}^{T} \frac{1}{2}\lambda\big\|\boldsymbol{W}_t\big\|^2,$$

where

- $\boldsymbol{W} = (\boldsymbol{W}_1, \ldots, \boldsymbol{W}_T)$ are the parameters (leaf values) of the trees;
- $\ell\big(t_i, y(\boldsymbol{x}_i; \boldsymbol{W})\big)$ is an per-example loss, $(t_i - y(\boldsymbol{x}_i; \boldsymbol{W}))^2$ for regression;
- the λ is the usual L_2 regularization strength.

Gradient Boosting for Regression

To construct the trees sequentially, we extend the definition to

$$\mathcal{L}^{(t)}(\boldsymbol{W}_t; \boldsymbol{W}_{1..t-1}) = \sum_i \Big[\ell\big(t_i, y^{(t-1)}(\boldsymbol{x}_i; \boldsymbol{W}_{1..t-1}) + y_t(\boldsymbol{x}_i; \boldsymbol{W}_t)\big)\Big] + \frac{1}{2}\lambda\big\|\boldsymbol{W}_t\big\|^2.$$

In the following text, we drop the parameters of $y^{(t-1)}$ and y_t for brevity.

The original idea of gradient boosting was to set

$$y_t(\boldsymbol{x}_i) \propto -\frac{\partial\ell\big(t_i, y^{(t-1)}(\boldsymbol{x}_i)\big)}{\partial y^{(t-1)}(\boldsymbol{x}_i)}$$

as a direction minimizing the residual loss and then finding a suitable constant γ_t, which would minimize the loss

$$\sum_i \Big[\ell\big(t_i, y^{(t-1)}(\boldsymbol{x}_i) + \gamma_t y_t(\boldsymbol{x}_i)\big)\Big] + \frac{1}{2}\lambda\big\|\boldsymbol{W}_t\big\|^2.$$

First-order and Second-order Methods

Until now, we used mostly SGD for finding a minimum, by performing

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla L(\boldsymbol{w}).$$

A disadvantage of this (so-called **first-order method**) is that we need to specify the learning rates by ourselves, usually using quite a small one and performing the update many times.

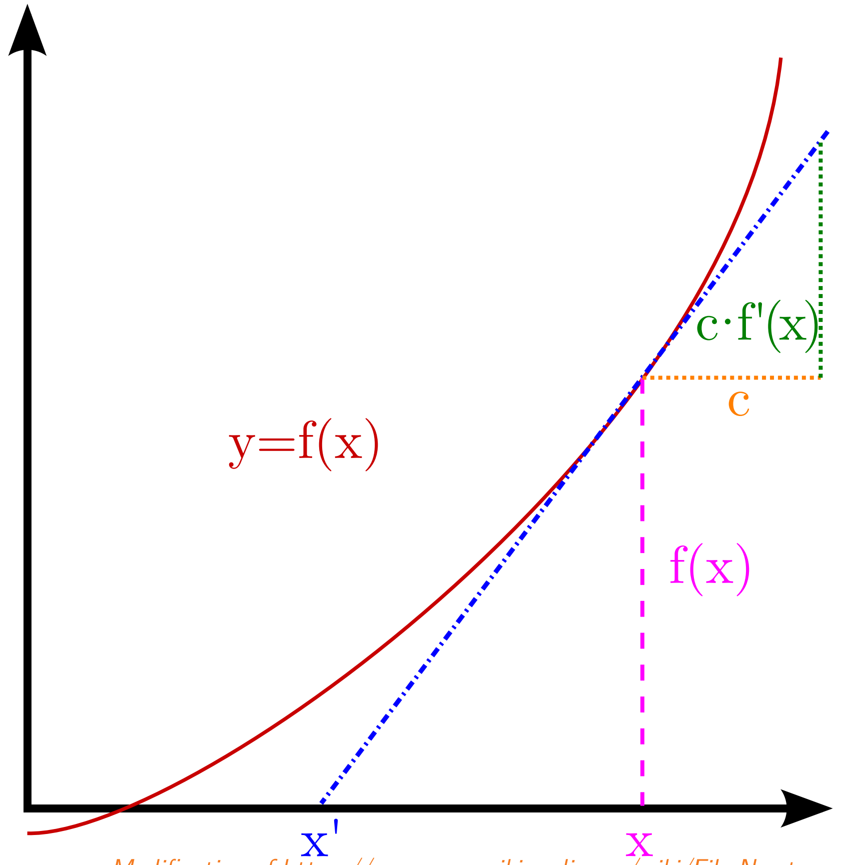
However, in some situations we can do better.

Newton's Root-Finding Method

Assume we got a function $f: \mathbb{R} \rightarrow \mathbb{R}$ and we want to find its root. A SGD-like algorithm would always move "towards" zero by taking small steps.

Instead, we could consider the a linear local approximation (i.e., consider a line "touching" the function in a given point) and perform a step so that our linear local approximation has value 0:

$$x' \leftarrow x - \frac{f(x)}{f'(x)}.$$



Modification of https://commons.wikimedia.org/wiki/File:Newton–Raphson_method.svg

Finding Minima

The same method can be used to find minima, because a minimum is just a root of a derivation, resulting in:

$$x' \leftarrow x - \frac{f'(x)}{f''(x)}.$$

Newton's Method

The following update is the Newton's method of searching for extremes:

$$x' \leftarrow x - \frac{f'(x)}{f''(x)}.$$

It is a so-called **second-order** method, but it is in fact a SGD update with learning rate $\frac{1}{f''(x)}$.

Derivation from Taylor's Expansion

The same update can be derived also from the Taylor's expansion

$$f(x+\varepsilon) \approx f(x) + \varepsilon f'(x) + \frac{1}{2}\varepsilon^2 f''(x), + \mathcal{O}(\varepsilon^3)$$

which we can minimise for ε by

$$0 = \frac{\partial f(x+\varepsilon)}{\partial \varepsilon} \approx f'(x) + \varepsilon f''(x), \text{ obtaining } x + \varepsilon = x - \frac{f'(x)}{f''(x)}.$$

Training MLPs with the Newton's Method

Note that the second-order methods (methods utilizing second derivatives) are impractical when training MLPs (and GLMs) with many parameters. The problem is that there are too many second derivatives – if we consider weights $\boldsymbol{w} \in \mathbb{R}^D$,

- the gradient $\nabla L(\boldsymbol{w})$ has D elements;
- however, we have a $D \times D$ matrix with all second derivatives, called the **Hessian** H:

$$H_{i,j} \stackrel{\text{def}}{=} \frac{\partial^2 L(\boldsymbol{w})}{\partial w_i \partial w_j}.$$

For completeness, the Taylor expansion than has the following form

$$f(\boldsymbol{x} + \boldsymbol{\varepsilon}) = f(\boldsymbol{x}) + \boldsymbol{\varepsilon}^T \nabla f(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{\varepsilon}^T H \boldsymbol{\varepsilon},$$

from which we obtain the following second-order method update:

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - H^{-1} \nabla f(\boldsymbol{x}).$$

Gradient Boosting

However, a more principled approach was later suggested. Denoting

$$g_i = \frac{\partial \ell\big(t_i, y^{(t-1)}(\boldsymbol{x}_i)\big)}{\partial y^{(t-1)}(\boldsymbol{x}_i)}$$

and

$$h_i = \frac{\partial^2 \ell\big(t_i, y^{(t-1)}(\boldsymbol{x}_i)\big)}{\partial y^{(t-1)}(\boldsymbol{x}_i)^2},$$

we can expand the objective $\mathcal{L}^{(t)}$ using a second-order approximation to

$$\mathcal{L}^{(t)}(\boldsymbol{W}_t; \boldsymbol{W}_{1..t-1}) \approx \sum_i \Big[\ell\big(t_i, y^{(t-1)}(\boldsymbol{x}_i)\big) + g_i y_t(\boldsymbol{x}_i) + \frac{1}{2} h_i y_t^2(\boldsymbol{x}_i)\Big] + \frac{1}{2}\lambda \big\|\boldsymbol{W}_t\big\|^2.$$

Gradient Boosting

Recall that we denote the indices of instances belonging to a node $\mathcal{T}$ as $I_\mathcal{T}$, and let us denote the prediction for the node $\mathcal{T}$ as $w_\mathcal{T}$. Then we can rewrite

$$\begin{aligned}\mathcal{L}^{(t)}(\boldsymbol{W}_t; \boldsymbol{W}_{1..t-1}) &\approx \sum\nolimits_i \Big[g_i y_t(\boldsymbol{x}_i) + \frac{1}{2} h_i y_t^2(\boldsymbol{x}_i)\Big] + \frac{1}{2}\lambda\big\|\boldsymbol{W}_t\big\|^2 + \text{const} \\ &\approx \sum_\mathcal{T} \Bigg[\Big(\sum_{i \in I_\mathcal{T}} g_i\Big) w_\mathcal{T} + \frac{1}{2}\Big(\lambda + \sum_{i \in I_\mathcal{T}} h_i\Big) w_\mathcal{T}^2\Bigg] + \text{const}.\end{aligned}$$

By setting a derivative with respect to $w_\mathcal{T}$ to zero, we get

$$0 = \frac{\partial \mathcal{L}^{(t)}}{\partial w_\mathcal{T}} = \sum\nolimits_{i \in I_\mathcal{T}} g_i + \Big(\lambda + \sum\nolimits_{i \in I_\mathcal{T}} h_i\Big) w_\mathcal{T}.$$

Therefore, the optimal weight for a node $\mathcal{T}$ is

$$w_\mathcal{T}^* = -\frac{\sum_{i \in I_\mathcal{T}} g_i}{\lambda + \sum_{i \in I_\mathcal{T}} h_i}.$$

Gradient Boosting

Substituting the optimum weights to the loss, we get

$$\mathcal{L}^{(t)}(\boldsymbol{W}) \approx -\frac{1}{2}\sum_{\mathcal{T}} \frac{\left(\sum_{i\in I_\mathcal{T}} g_i\right)^2}{\lambda + \sum_{i\in I_\mathcal{T}} h_i} + \text{const},$$

which can be used as a splitting criterion.

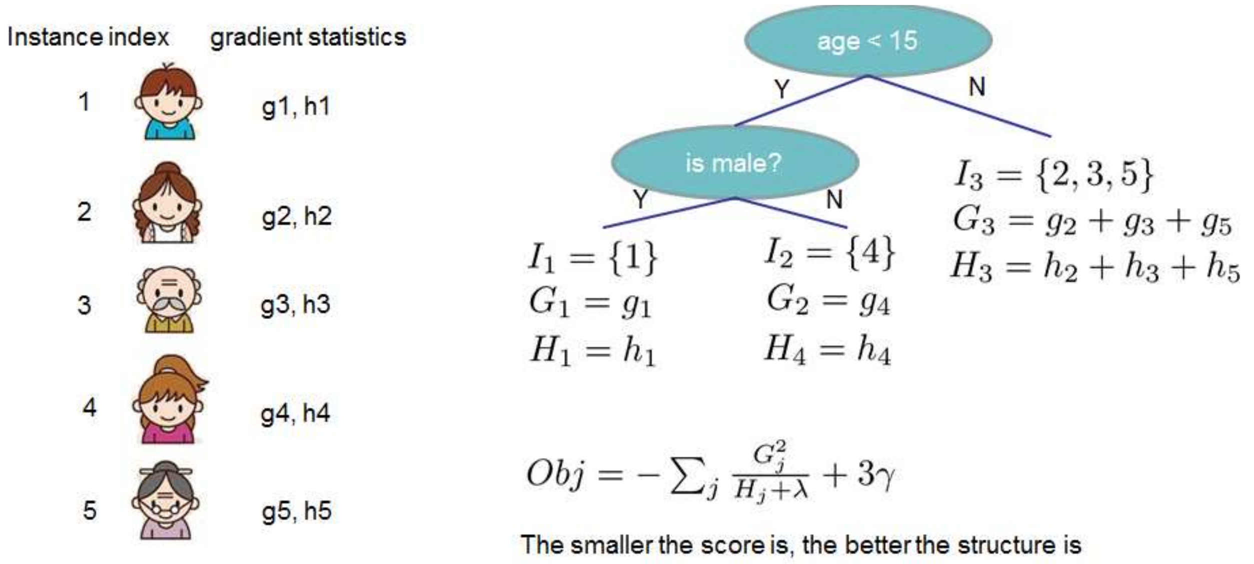


Figure 2 of the paper "XGBoost: A Scalable Tree Boosting System", https://arxiv.org/abs/1603.02754

Gradient Boosting

When splitting a node, the criterions of all possible splits can be effectively computed using the following algorithm:

```
Algorithm 1: Exact Greedy Algorithm for Split Finding
Input: $I$, instance set of current node
Input: $D$, feature dimension
$score \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i,\ H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ to $D$ do
    $G_L \leftarrow 0,\ H_L \leftarrow 0$
    for $j$ in sorted($I$, by $\mathbf{x}_{jk}$) do
        $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
        $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
        if $\mathbf{x}_{j_\textit{next}k} \neq \mathbf{x}_{jk}$ then
            $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
    end
end
Output: Split with max score
```

Modified from Algorithm 1 of the paper "XGBoost: A Scalable Tree Boosting System", https://arxiv.org/abs/1603.02754

Gradient Boosting

Furthermore, gradient boosting trees frequently use:

- data subsampling: either bagging, or (even more commonly) utilize only a fraction of the original training data for training a single tree (with 0.5 being a common value),
- feature subsampling;
- shrinkage: multiply each trained tree by a learning rate α, which reduces influence of each individual tree and leaves space for future optimization.

Binary Classification with Gradient Boosting Decision Trees

To perform classification, we train the trees to perform the linear part of a generalized linear model.

Specifically, for a binary classification, we perform prediction by

$$\sigma\big(y(\boldsymbol{x}_i)\big) = \sigma\left(\sum_{t=1}^{T} y_t(\boldsymbol{x}_i; \boldsymbol{W}_t)\right),$$

and the per-example loss is defined as

$$\ell\big(t_i, y(\boldsymbol{x}_i)\big) = -\log\Big[\sigma\big(y(\boldsymbol{x}_i)\big)^{t_i}\big(1 - \sigma\big(y(\boldsymbol{x}_i)\big)\big)^{1-t_i}\Big].$$

Multiclass Classification with Gradient Boosting Decision Trees

For multiclass classification, we need to model the full categorical output distribution. Therefore, for each "timestep" t, we train K trees $\boldsymbol W_{t,k}$, each predicting a single value of the linear part of a generalized linear model.

Then, we perform prediction by

$$\operatorname{softmax}\big(\boldsymbol y(\boldsymbol x_i)\big) = \operatorname{softmax}\left(\sum\nolimits_{t=1}^T y_{t,1}(\boldsymbol x_i; \boldsymbol W_{t,1}), \ldots, \sum\nolimits_{t=1}^T y_{t,K}(\boldsymbol x_i; \boldsymbol W_{t,K})\right),$$

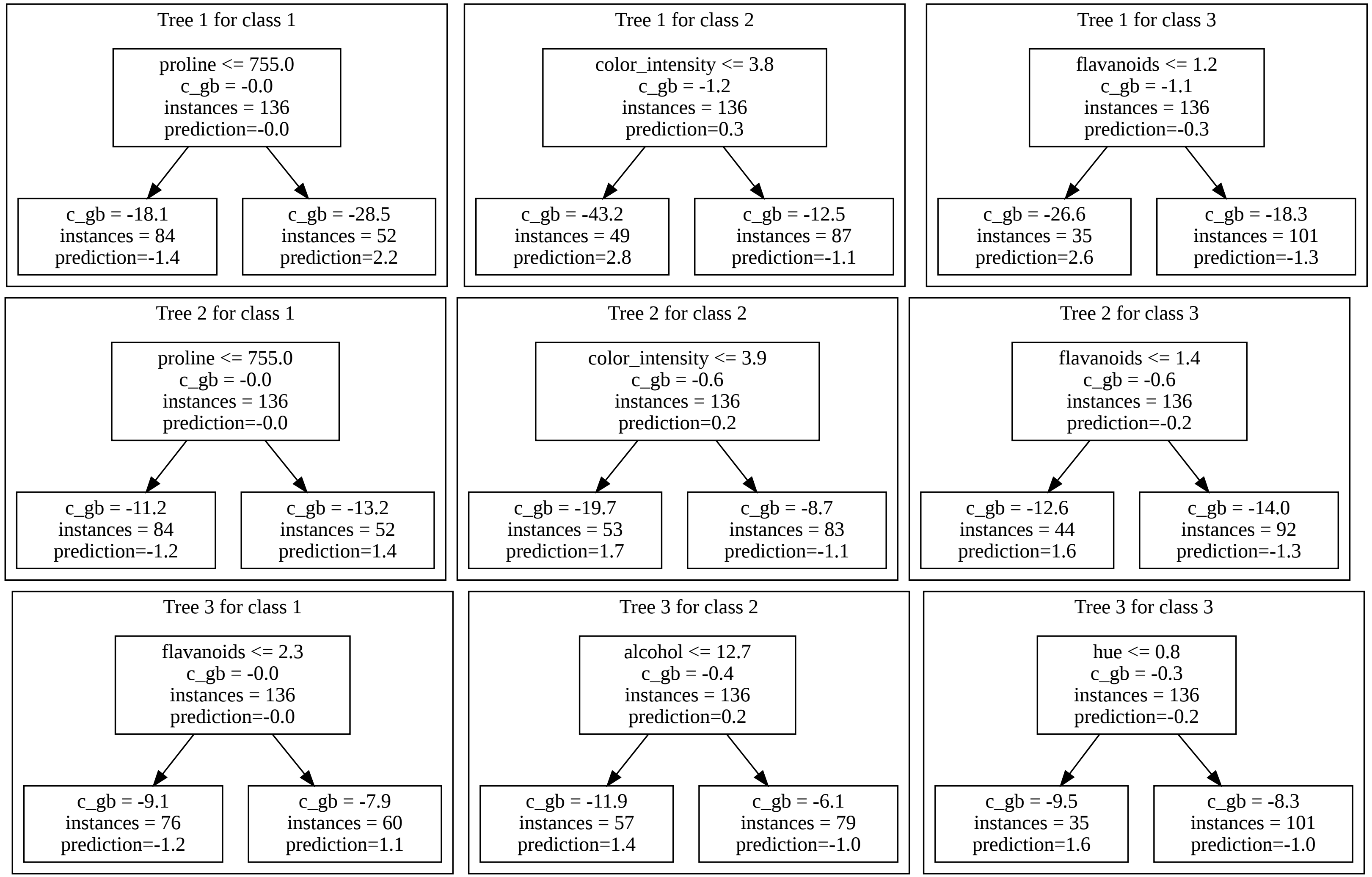
and the per-example loss for all K trees is defined analogously as

$$\ell\big(t_i, \boldsymbol y(\boldsymbol x_i)\big) = -\log\Big(\operatorname{softmax}\big(\boldsymbol y(\boldsymbol x_i)\big)_{t_i}\Big),$$

so that for a tree k at time t,

$$\frac{\partial \ell\big(t_i, \boldsymbol y^{(t-1)}(\boldsymbol x_i)\big)}{\partial \boldsymbol y^{(t-1)}(\boldsymbol x_i)_k} = \Big(\operatorname{softmax}\big(\boldsymbol y^{(t-1)}(\boldsymbol x_i)\big) - \boldsymbol 1_{t_i}\Big)_k.$$

Multiclass Classification with Gradient Boosting Decision Trees



Gradient Boosting Demo and Implementations

Playground

You can explore the Gradient Boosting Trees playground and Gradient Boosting Trees explained.

Implementations

Scikit-learn offers an implementation of gradient boosting decision trees, `sklearn.ensemble.GradientBoostingClassifier` for classification and `sklearn.ensemble.GradientBoostingRegressor` for regression.

- Furthermore, `sklearn.ensemble.HistGradientBoosting{Classifier/Regressor}` provide histogram-based splitting (which can be much faster for larger datasets – tens of thousands of examples and more) and efficient categorical feature splitting.

There are additional efficient implementations, capable of distributed processing of data larger than available memory (both offering also scikit-learn interface):

- XGBoost,
- LightGBM (which is the inspiration for the `HistGradientBoosting*` implementation).

Supervised Machine Learning

This concludes the **supervised machine learning** part of our course.

We have encountered:

- parametric models
  - generalized linear models: perceptron algorithm, linear regression, logistic regression, multinomial (softmax) logistic regression, Poisson regression
    - linear models, but manual feature engineering allows solving non-linear problems
  - multilayer perceptron: non-linear, perfect approximator – Universal approx. theorem
- non-parametric models
  - k-nearest neighbors
  - kernelized linear regression
  - support vector machines
- decision trees
  - can be both parametric or non-parametric depending on the constraints
- generative models
  - naive Bayes

Supervised Machine Learning

When training a model for a new dataset, I start by evaluating two models:

- an **MLP** with one/two hidden layers
  - works best for high-dimensional data (images, speech, text), where an individual single dimension (feature) does not convey much meaning;
- **gradient boosted decision tree**
  - works best for lower-dimensional data, where the input features have interpretation on their own.

However, if the amount of training examples is not too large (tens/hundreds of thousands at most) and there are lot of features, **SVM** with **RBF** kernel might offer best performance.

Furthermore, if there are only a few training examples with a lot of features, **naive Bayes** might also work well.

Finally, if your goal is to reach the highest possible performance and you have a lot of resources, definitely use **ensembling**.