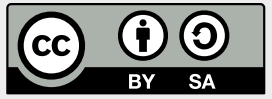
Gradient Boosted Decision Trees

Jindřich Libovický (reusing materials by Milan Straka)

December 05, 2023

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



Today's Lecture Objectives

After this lecture you should be able to

- Explain second-order optimization methods
- Implement gradient boosted decision trees for regression and classification
- Decide what supervised machine learning approach is suitable for particular problems

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^{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^{th} tree.

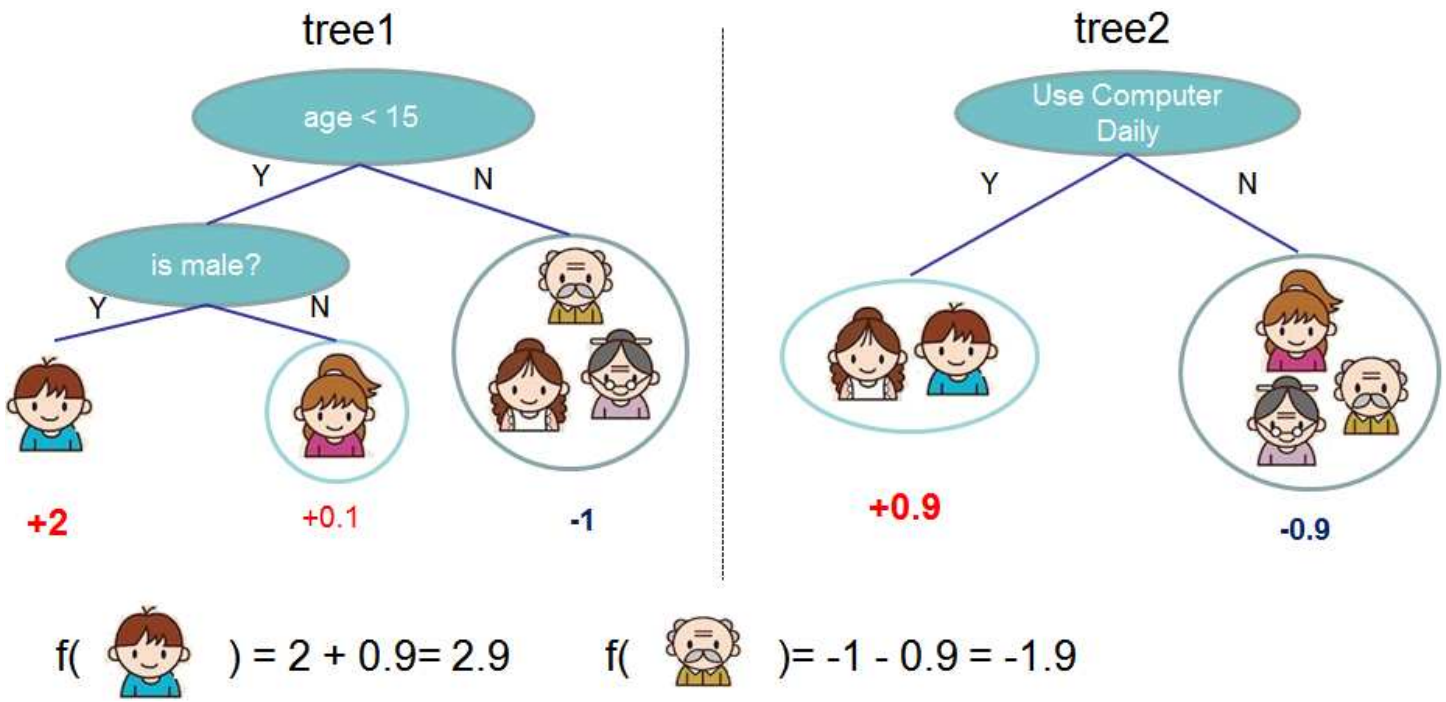


Figure 1 of "XGBoost: A Scalable Tree Boosting System", https://arxiv.org/abs/1603.02754

Gradient Boosting for Regression

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

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

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

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

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

$$y_t(\boldsymbol{x}_i) \leftarrow -\frac{\partial \ell\big(t_i, y^{(t-1)}(\boldsymbol{x}_i)\big)}{\partial y^{(t-1)}(\boldsymbol{x}_i)} = -\frac{\partial \ell\big(t_i, y\big)}{\partial y}\bigg|_{y=y^{(t-1)}(\boldsymbol{x}_i)}$$

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

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

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 perform the update many times.

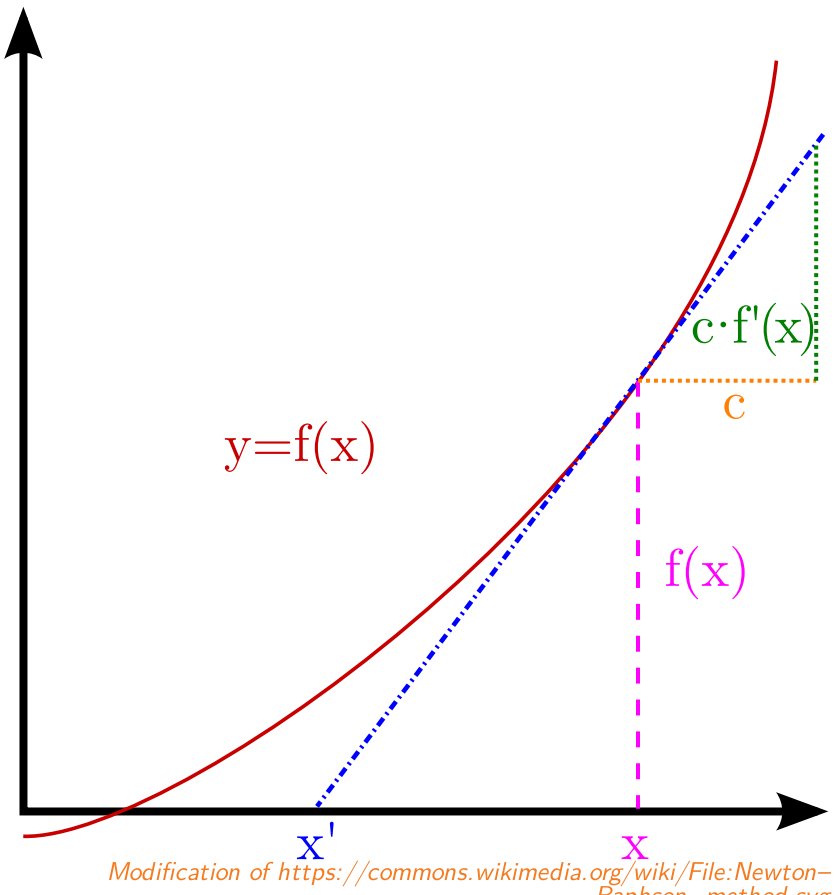
However, in some situations, we can do better.

Newton's Root-Finding Method

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

Instead, we could consider the 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 a 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 derivative, resulting in:

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

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 just an SGD update with a learning rate $\frac{1}{f''(x)}$.

Derivation from Taylor's Expansion

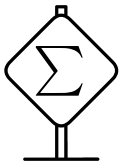
The same update can be derived also from the Taylor's expansion (x is a fixed point and ϵ is now the variable that moves)

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

which we can minimize for ε by (i.e., the minimum of the approximation)

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

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

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

For completeness, the Taylor expansion of a multivariate function then has the following form:

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

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

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

Gradient Boosting

Returning to the gradient boosting decision trees, instead of using a first-order method, it was later suggested that a second-order method could be used. Denoting

$$g_i = \frac{\partial \ell\left(t_i, y^{(t-1)}(\boldsymbol{x}_i)\right)}{\partial y^{(t-1)}(\boldsymbol{x}_i)} = \left.\frac{\partial \ell\left(t_i, y\right)}{\partial y}\right|_{y=y^{(t-1)}(\boldsymbol{x}_i)}$$

and

$$h_i = \frac{\partial^2 \ell\left(t_i, y^{(t-1)}(\boldsymbol{x}_i)\right)}{\partial y^{(t-1)}(\boldsymbol{x}_i)^2} = \left.\frac{\partial^2 \ell\left(t_i, y\right)}{\partial y^2}\right|_{y=y^{(t-1)}(\boldsymbol{x}_i)},$$

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

$$E^{(t)}(\boldsymbol{w}_t; \boldsymbol{w}_{1..t-1}) \approx \sum_i \left[\ell\left(t_i, y^{(t-1)}(\boldsymbol{x}_i)\right) + g_i y_t(\boldsymbol{x}_i) + \frac{1}{2} h_i y_t^2(\boldsymbol{x}_i)\right] + \frac{1}{2}\lambda \left\|\boldsymbol{w}_t\right\|^2.$$

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

$$E^{(t)}(\boldsymbol{w}_t; \boldsymbol{w}_{1..t-1}) \approx \sum\nolimits_i \left[g_i y_t(\boldsymbol{x}_i) + \frac{1}{2} h_i y_t^2(\boldsymbol{x}_i) \right] + \frac{1}{2}\lambda \left\| \boldsymbol{w}_t \right\|^2 + \text{const}$$
$$\approx \sum_\mathcal{T} \left[\Big(\sum_{i \in I_\mathcal{T}} g_i \Big) w_\mathcal{T} + \frac{1}{2} \Big(\lambda + \sum_{i \in I_\mathcal{T}} h_i \Big) w_\mathcal{T}^2 \right] + \text{const}.$$

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

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

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

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

Substituting the optimum weights to the loss, we get

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

which can be used as a *splitting criterion.*

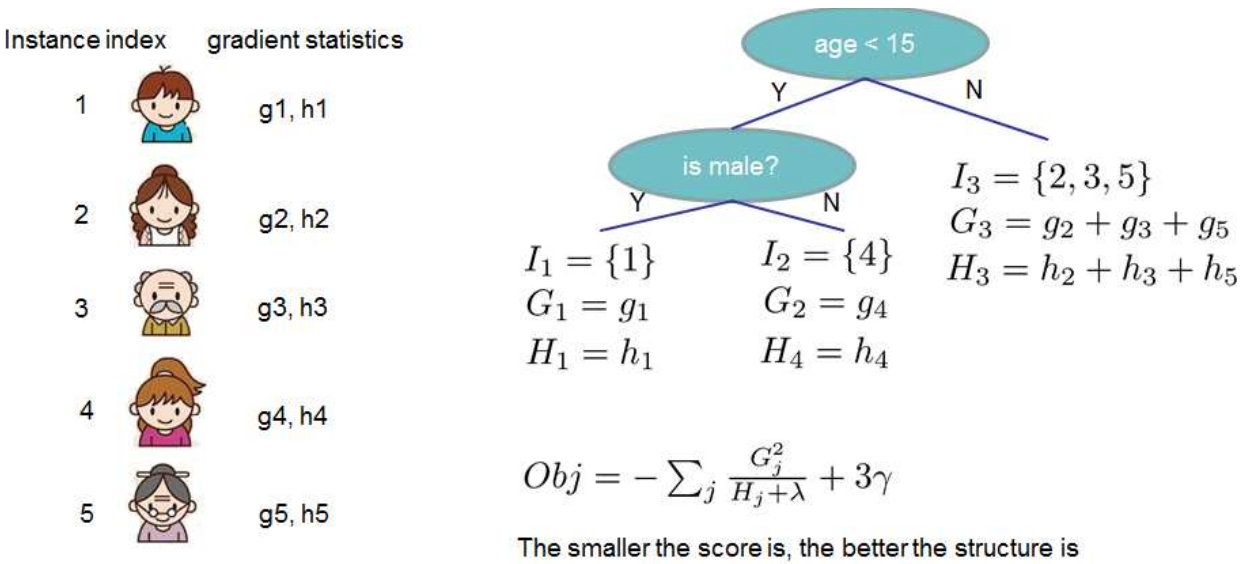


Figure 2 of "XGBoost: A Scalable Tree Boosting System", https://arxiv.org/abs/1603.02754

When splitting a node, the criteria 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
$score \leftarrow -\infty$
$G \leftarrow \sum_{i \in I} g_i,\ H \leftarrow \sum_{i \in I} h_i$
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 $\mathbf{x}_{j_{next}\,k} \neq \mathbf{x}_{jk}$ **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 "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) only a fraction of the original training data is utilized for training of a single tree (with 0.5 being a common value),
- feature subsampling;
- shrinkage: multiply each trained tree by a learning rate α, which reduces the 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

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

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

$$\operatorname{softmax}\big(\boldsymbol{y}(\boldsymbol{x}_i)\big) = \operatorname{softmax}\left(\sum_{t=1}^{T} y_{t,1}(\boldsymbol{x}_i; \boldsymbol{w}_{t,1}), \ldots, \sum_{t=1}^{T} y_{t,K}(\boldsymbol{x}_i; \boldsymbol{w}_{t,K})\right),$$

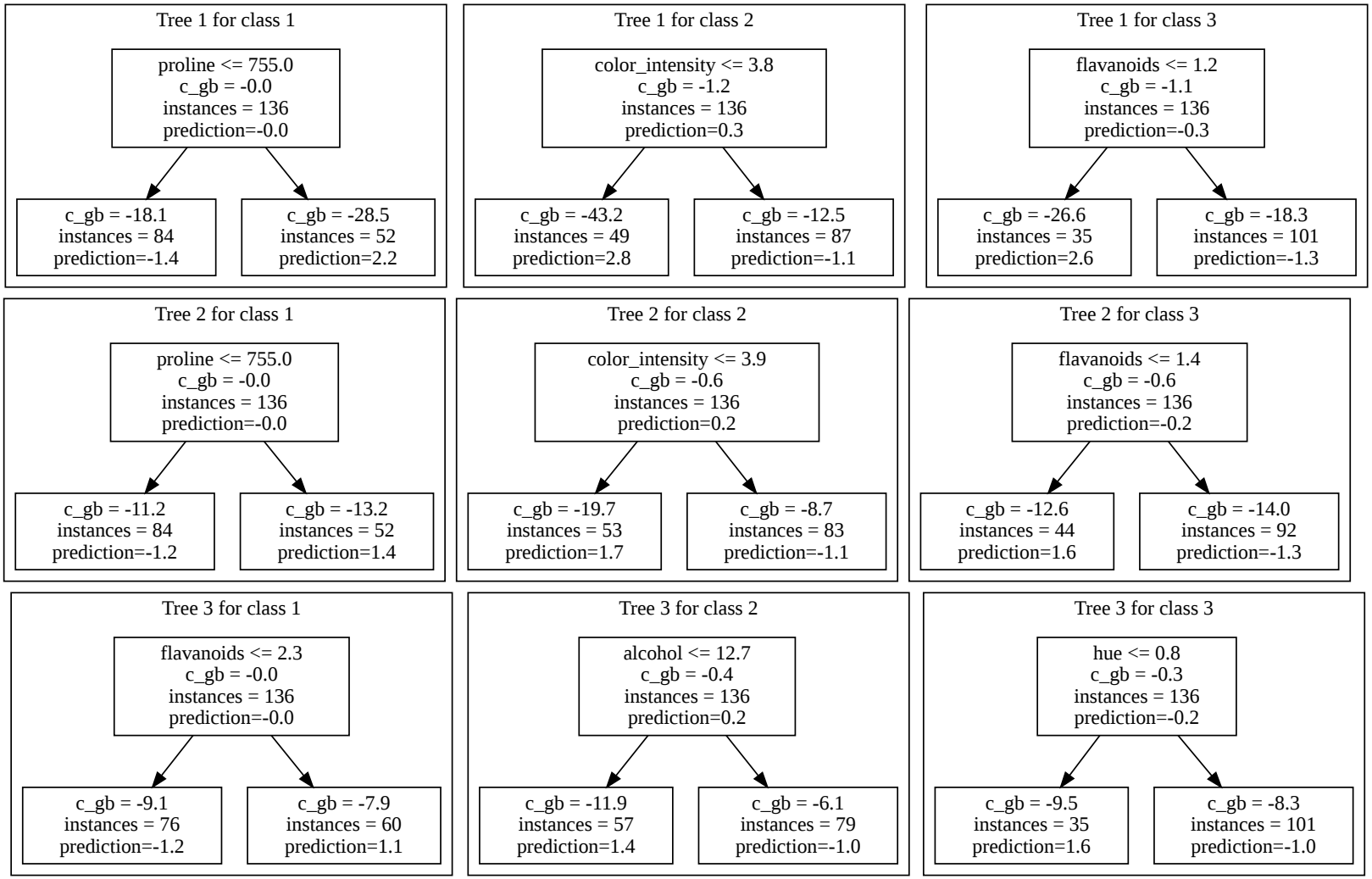
and the per-example loss for all K trees is defined analogously as

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

so that for a tree k at time t,

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

Multiclass Classification with Gradient Boosting Decision Trees



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
    - linear models, but manual feature engineering allows solving nonlinear problems
  - multilayer perceptron: nonlinear, perfect approximator – Universal approx. theorem
- nonparametric models
  - k-nearest neighbors
  - support vector machines not in this course, but in the state exam
- decision trees
  - can be both parametric or nonparametric depending on the constraints
- generative models
  - naive Bayes

Supervised Machine Learning

When training a model for a new dataset, a good start is 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; use pre-trained representation if possible;
- **gradient boosted decision tree**
  - works best for lower-dimensional data ("tabular data"), where the input features have interpretations on their own.

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**.

Today's Lecture Objectives

After this lecture you should be able to

- Explain second-order optimization methods
- Implement gradient boosted decision trees for regression and classification
- Decide what supervised machine learning approach is suitable for particular problems