Gradient boosting and ensemble learning

Szu-Chi Chung

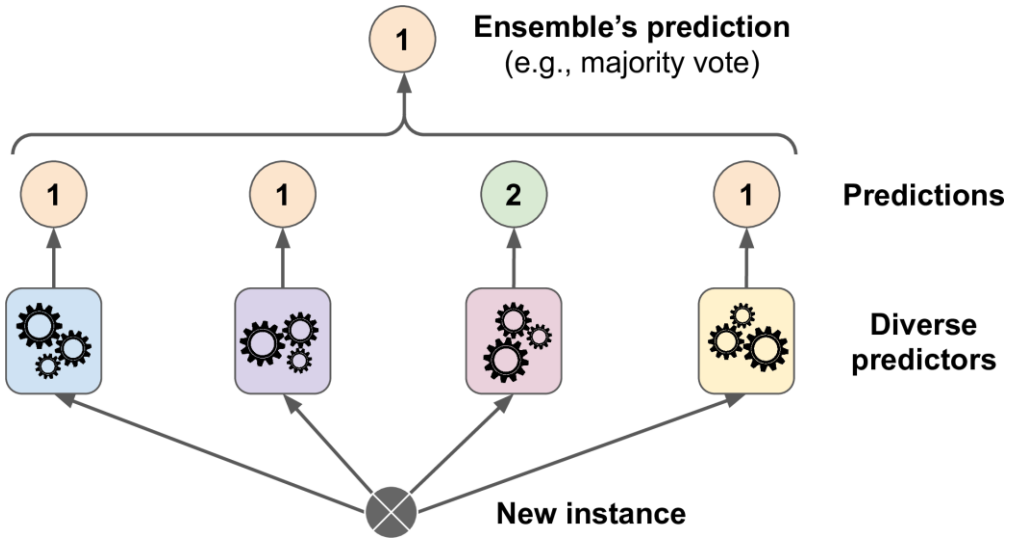
Department of Applied Mathematics, National Sun Yat-sen University

Why and when to use ensemble learning?

- Suppose you pose a complex question to thousands of random people, then aggregate their answers. In many cases you will find aggregated answer is better than an expert's answer. This is called the wisdom of the crowd
  - If you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor
  - A group of predictors is called an *ensemble*; thus, this technique is called Ensemble Learning, and an Ensemble Learning algorithm is called an Ensemble method
    - You will often use Ensemble methods *near the end of a project*, once you have already built a few good predictors, to combine them into an even better predictor

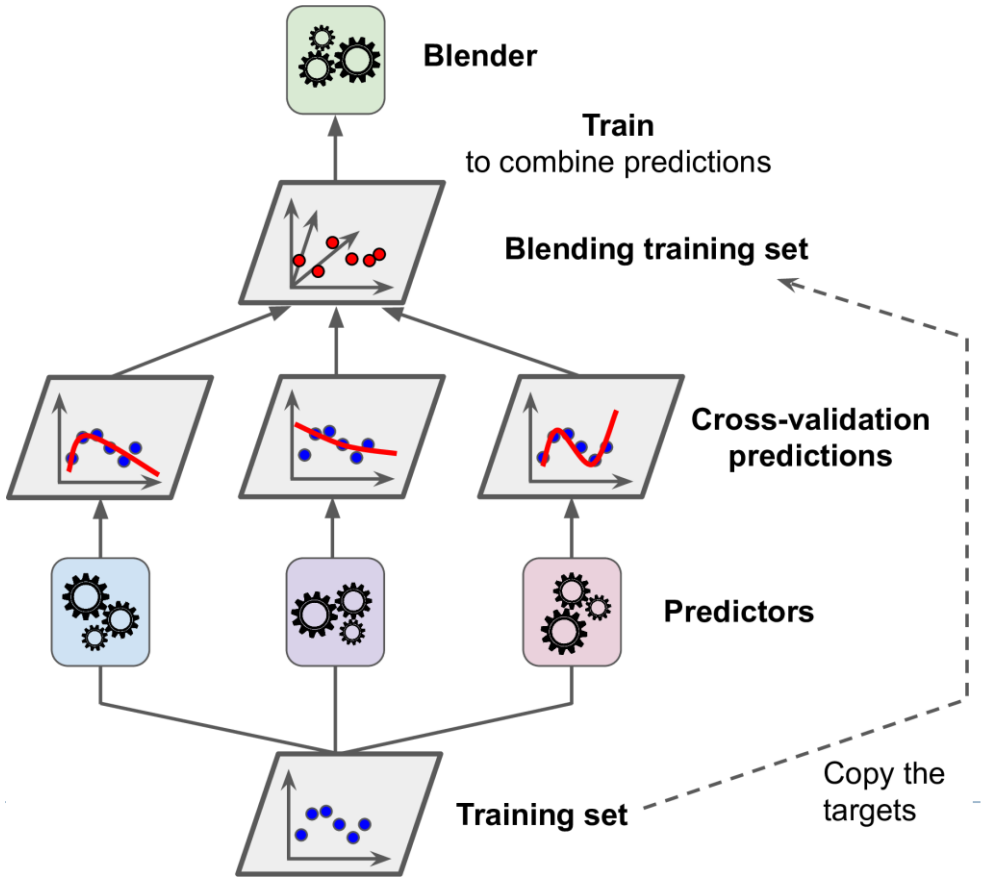
Voting classifier

- Suppose you train a few classifiers, each one achieving about 80% accuracy
  - A very simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the *most votes*. This majority-vote classifier is called a hard voting classifier
  - If you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time. If you predict the majority voted class, you can hope for up to 75% accuracy!
    - Only true if all classifiers are perfectly independent and making uncorrelated errors!
    - One way to get diverse classifiers is to train them using *very different algorithms*. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy



Stacking

- Stacking is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation?
  - To train the blender, you first need to build the blending training set
  - You can use cross-validation on every estimator in the ensemble to get out-of-sample predictions for each instance in the original training set
    - These can be used as the input features to train the blender, and the targets can be simply be copied from the original training set



Ensemble method - Bagging

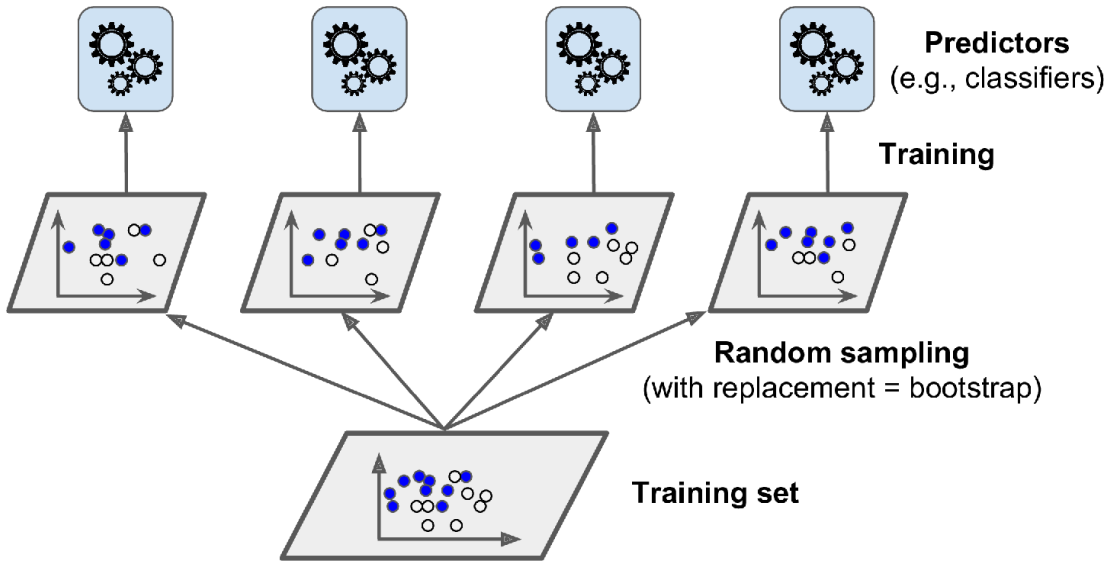
- Another approach is to use the same training algorithm for every predictor but train them on different dataset
  - Bootstrap aggregation, or bagging, is a general-purpose procedure for reducing the variance of a statistical learning method
  - Recall that given a set of n independent observations $Z_1, \ldots, Z_n$, each with variance σ^2, the variance of the mean $\bar{Z}$ of the observations is given by σ^2/n
  - In other words, averaging a set of observations reduces variance. Of course, this is not practical because we generally do not have access to multiple training sets
    - Instead, we choose random subsets of the training set. When sampling is performed with replacement, this method is called bagging. When sampling is performed without replacement, it is called pasting

Bagging— continued

- In this approach we generate B different bootstrapped training data sets
  - We then train our method on the bth bootstrapped training set in order to get $\hat{f}^{*b}(x)$, the prediction at a point x. We then average all the predictions to obtain

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x)$$

  - For classification, we take a majority vote among the B predictions
  - It scales well because the predictors can all be trained in parallel and the predictions can be made in parallel, too
  - Feature sampling is also possible which is called random subspaces methods
    - When combining both, it is call random patches

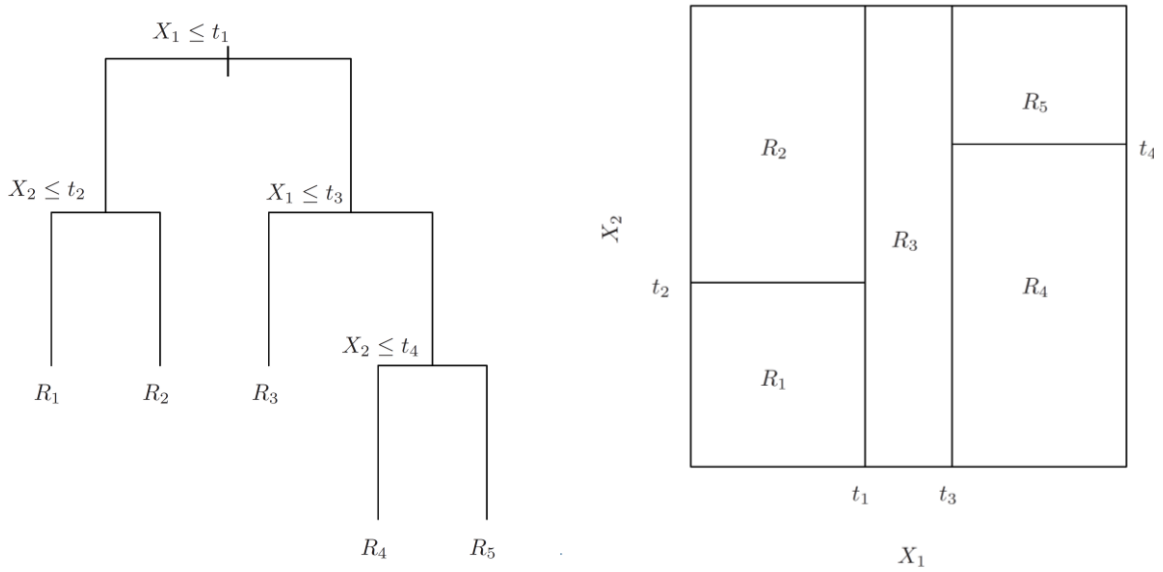


Out-of-Bag Error Estimation

- It turns out that there is a very straightforward way to estimate the test error of a bagged model
  - Recall that the key to bagging is that estimators are repeatedly fit to bootstrapped subsets of the observations. One can show that on average, each bagged tree makes use of around two-thirds of the observations (the exact number of having jth observation is: $1 - (1 - \frac{1}{m})^m$)
    - The remaining one-third of the observations not used to fit a given bagged tree are referred to as the out-of-bag (OOB) observations
    - We can predict the response for the ith observation using each of the trees in which that observation was OOB. This will yield around $B/3$ predictions for the ith observation, which we average (or vote)
    - The resulting OOB error is a valid estimate of the test error for the bagged model, since the response for each observation is predicted using only the predictors that were not fit using that observation

Tree-based Methods

- Here we describe tree-based methods for regression and classification
  - These involve firstly stratifying or segmenting the predictor space into a number of simple regions
  - And use the mean or the mode response value for the training observations in the region to which it belongs for inference
- Since the set of splitting rules used to segment the predictor space can be summarized in a tree, these types of approaches are known as decision-tree methods



Details of the tree-building process

1. We divide the predictor space — that is, the set of possible values for $X_1, X_2, \ldots, X_p$— into J distinct and non-overlapping regions, $R_1, R_2, \ldots, R_J$
2. For every observation that falls into the region R_j , we make the same prediction, which is simply the mean of the response values for the training observations in R_j

- In theory, the regions could have any shape. However, we choose to divide the predictor space into high-dimensional rectangles, or boxes, for simplicity and for ease of interpretation of the resulting predictive model
  - The goal is to find boxes $R_1, R_2, \ldots, R_J$ that minimize the RSS, given by

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

$\hat{y}_{R_j}$ is the mean response for the training observations within the jth box

More details of the tree-building process

- Unfortunately, it is computationally infeasible to consider every possible partition of the feature space into J boxes
  - For this reason, we take a top-down, greedy approach that is known as *recursive binary splitting*
  - The approach is *top-down* because it begins at the top of the tree and then successively splits the predictor space; each split is indicated via two new branches further down on the tree
  - It is *greedy* because at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step

More details of the tree-building process

- We first select the predictor X_j and the cutpoint s such that splitting the predictor space into the regions $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ leads to the greatest possible reduction in RSS (choosing j and s to minimize)

$$\sum_{i:x_i \epsilon R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \epsilon R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

- Next, we looking for the best predictor and best cutpoint in order to split the data further so as to minimize the RSS within each of the resulting regions
  - Instead of splitting the entire predictor space, we split one of the two previously identified regions. We now have three regions
  - Again, we look to split one of these three regions further, so as to minimize the RSS. The process continues until a stopping criterion is reached; for instance, we may continue until no region contains more than five observations

Regularization

- The process described above may produce good predictions on the training set, but is likely to overfit the data, leading to poor test set performance
  - A simple way to limit a tree's size is to directly regulate its depth, the size of its terminal nodes (training observation belongs to them), or both
- A smaller tree with fewer splits (that is, fewer regions $R_1, R_2, \ldots, R_J$) might lead to lower variance and better interpretation at the cost of a little bias
  - One possible alternative to the process described above is to grow the tree only so long as the decrease in the RSS due to each split exceeds some (high) threshold
  - This strategy will result in smaller trees, but is too short-sighted: a seemingly worthless split early on in the tree might be followed by a very good split — that is, a split that leads to a large reduction in RSS later on

Pruning a tree

- A better strategy is to grow a very large tree T_0, and then prune it back in order to obtain a subtree
  - Cost complexity pruning — also known as weakest link pruning — is used to do this
  - We consider a sequence of trees indexed by a nonnegative tuning parameter α. For each value of α there corresponds a subtree $T \subset T_0$ such that

$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha|T|$$

    is as small as possible. Here $|T|$ indicates the number of terminal nodes of the tree T, R_m is the rectangle (i.e. the subset of predictor space) corresponding to the mth terminal node, and $\hat{y}_{R_m}$ is the mean of the training observations in R_m
- The tuning parameter α controls a trade-off between the subtree's complexity and its fit to the training data

Algorithm 8.1 *Building a Regression Tree*

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.

2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.

3. Use K-fold cross-validation to choose α. That is, divide the training observations into K folds. For each $k = 1, \ldots, K$:

   (a) Repeat Steps 1 and 2 on all but the kth fold of the training data.

   (b) Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.

   Average the results for each value of α, and pick α to minimize the average error.

4. Return the subtree from Step 2 that corresponds to the chosen value of α.

Advantages and Disadvantages of Trees

- Pros
  - Trees are very easy to explain to people. In fact, they are even easier to explain than linear regression!
  - Some people believe that decision trees more closely mirror human decision-making than do the regression and classification approaches seen in previous chapters
  - Trees can be displayed graphically, and are easily interpreted even by a non-expert (especially if they are small)
  - Trees can easily handle qualitative predictors without the need to create dummy variables
- Cons
  - Unfortunately, trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches
  - However, by aggregating many decision trees, the predictive performance of trees can be substantially improved!

Ensemble method - Random Forests

- Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees which reduces the variance when averaging
  - As in bagging with decision tree estimator, we build a number of decision trees on bootstrapped training samples
  - But when building these decision trees, each time a split in a tree is considered, a random selection of m predictors is chosen as split candidates from the full set of p predictors
    - The split is allowed to use only one of those m predictors
  - A fresh selection of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$ — that is, the number of predictors considered at each split is approximately equal to the square root of the total number of predictors
    - The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model

Ensemble method - Random Forests

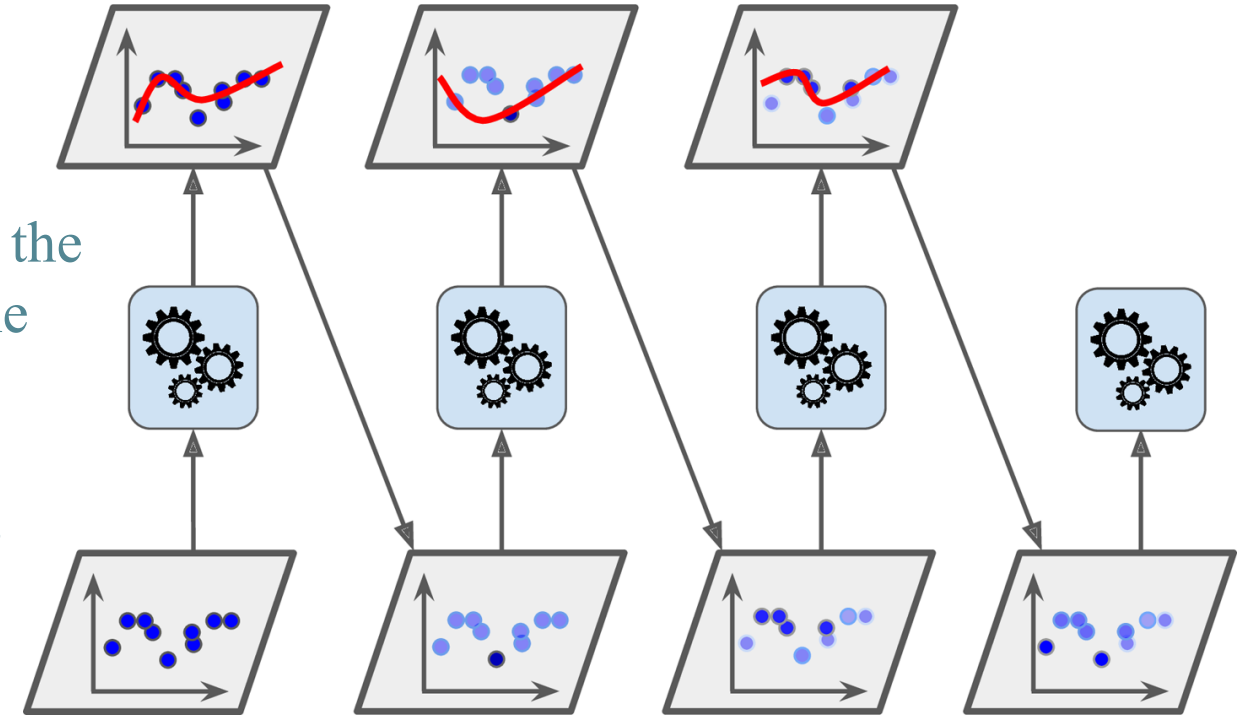
- Suppose that there is one very strong predictor in the data set, along with a number of other moderately strong predictors. Then in the collection of bagged trees, most or all of the trees will use this strong predictor in the top split
  - Consequently, all of the bagged trees will look quite similar to each other. Averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities
  - Random forests overcome this problem by forcing each split to consider only a subset of the predictors. Using a small value of m in building a random forest will typically be helpful when we have a large number of correlated predictors
  - It is possible to make trees even more random by also using *random thresholds* for each feature rather than searching for the best possible thresholds
    - This is call Extremely Randomized Trees ensemble (or Extra-Trees)

Ensemble method - Boosting

- Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification
- Recall that bagging involves creating multiple copies of the original training data set using the bootstrap, fitting a separate decision tree to each copy, and then combining all of the trees in order to create a single predictive model
  - Notably, each tree is built on a bootstrap data set, *independent* of the other trees
- Boosting works in a similar way, except that the trees are grown *sequentially:* each tree is grown using information from previously grown trees
  - Boosting does not involve bootstrap sampling; instead each tree is fit on a modified version of the original data set

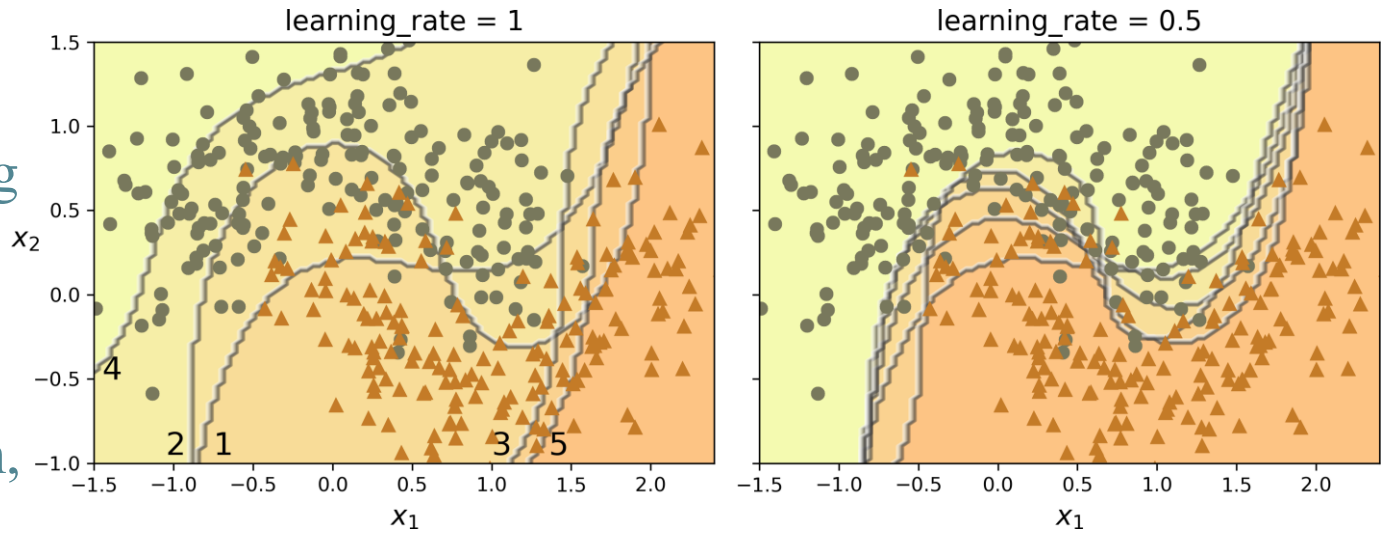
AdaBoost

- One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted
  - This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost
  - For example, when training an AdaBoost classifier, the algorithm first trains a base classifier and uses it to make predictions on the training set. The algorithm then increases the relative weight of *misclassified* training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on



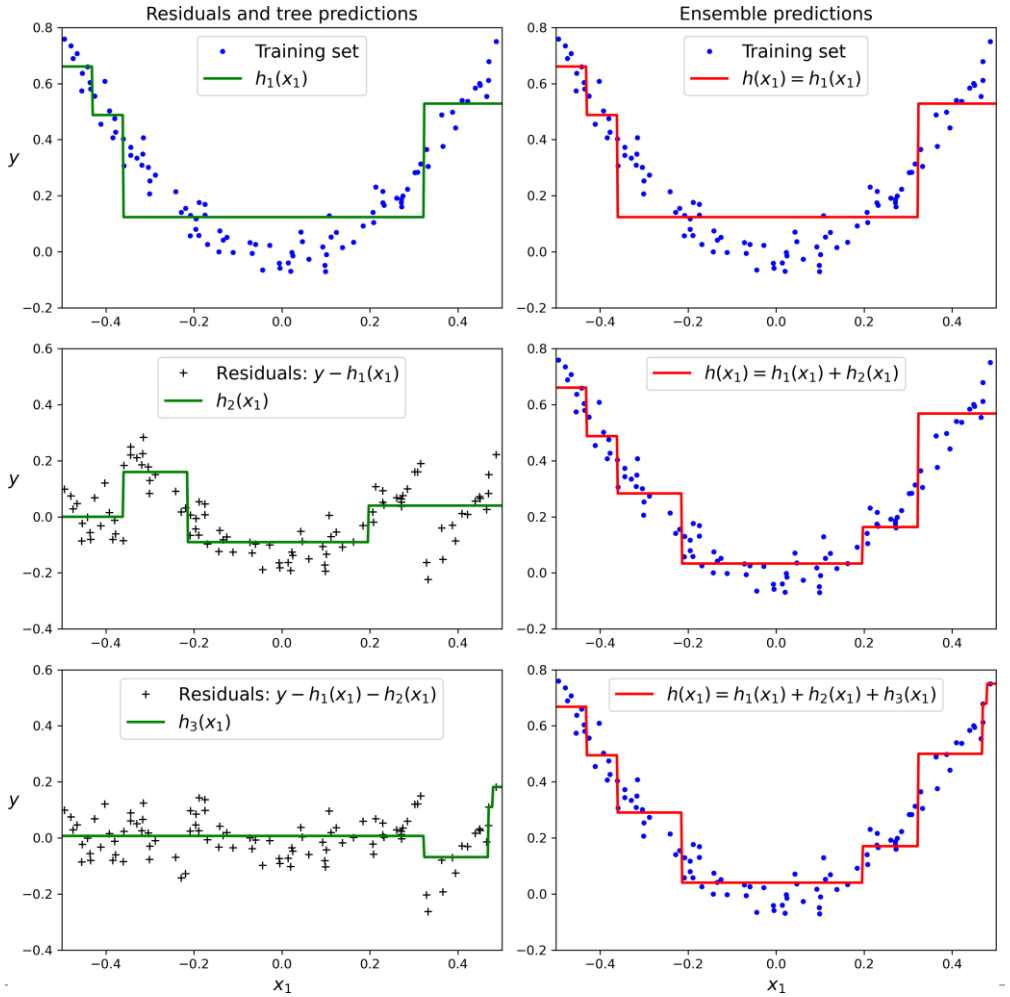
AdaBoost

- The following shows the decision boundaries of five consecutive predictors on the moons dataset
  - The first classifier gets many instances wrong, so their weights get boosted. The second classifier therefore does a better job on these instances, and so on
  - The plot on the right represents the same sequence of predictors, except that the learning rate is halved
  - As you can see, this sequential learning technique has some similarities with Gradient Descent, except that instead of tweaking a single predictor's parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better



Gradient Boosting

- Another very popular boosting algorithm is Gradient Boosting
  - Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to *fit the new predictor to the residual errors* made by the previous predictor



Boosting algorithm for regression trees

Algorithm 8.2 *Boosting for Regression Trees*

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.

2. For $b = 1, 2, \ldots, B$, repeat:

   (a) Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).

   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:

   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{8.10}$$

   (c) Update the residuals,

   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{8.11}$$

3. Output the boosted model,

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{8.12}$$

What is the idea behind this procedure?

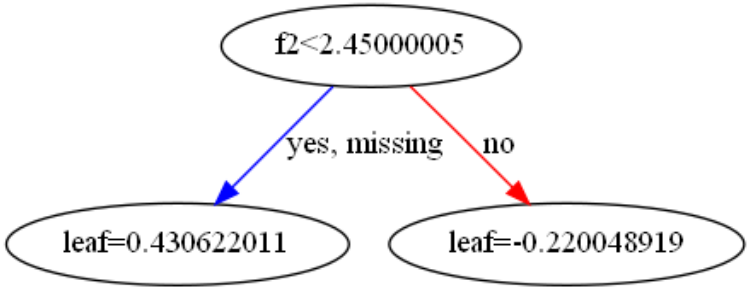
- Unlike fitting a single large decision tree to the data, which amounts to fitting the data hard and potentially overfitting, the boosting approach instead *learns slowly*
  - Given the current model, we fit a decision tree to the residuals from the model. We then add this new decision tree into the fitted function in order to update the residuals
  - Each of these trees can be rather small, with just a few terminal nodes, determined by the parameter d in the algorithm
  - By fitting small trees to the residuals, we slowly improve $\hat{f}$ in areas where it does not perform well. The shrinkage parameter λ slows the process down even further, allowing more and different shaped trees to attack the residuals

Boosting for classification

- Boosting has three tuning parameters
  - The number of trees B (Choose by CV)
  - The shrinkage parameter λ (Typical values are 0.01 or 0.001)
  - The number d of splits in each tree (Often $d = 1$ works well, in which case each tree is a stump, consisting of a single split and resulting in an additive model)
- Boosting for classification is similar in spirit to boosting for regression, but is a bit more complex. We will not go into detail here
  - Can learn about the details in Elements of Statistical Learning, chapter 10

XGBoost

- XGBoost, short for Extreme Gradient Boosting is a form of gradient boosting included built-in regularization and impressive gains in speed
  - The need for faster algorithms is evident when dealing with big data
- XGBoost or Gradient Boosting Decision Tree (GBDT)
  - Apply target encoding
  - Don’t need to perform scaling (Only the relative size matters)
  - When given a missing data point, XGBoost treat missing value as a feature and scores different split options and chooses the one with the best results
  - Can handle sparse matrix (Sparsity-aware split finding)
- XGBoot presents
  - Approximate split-finding algorithm
  - Parallel computing
  - Cache-aware access – improve cache performance

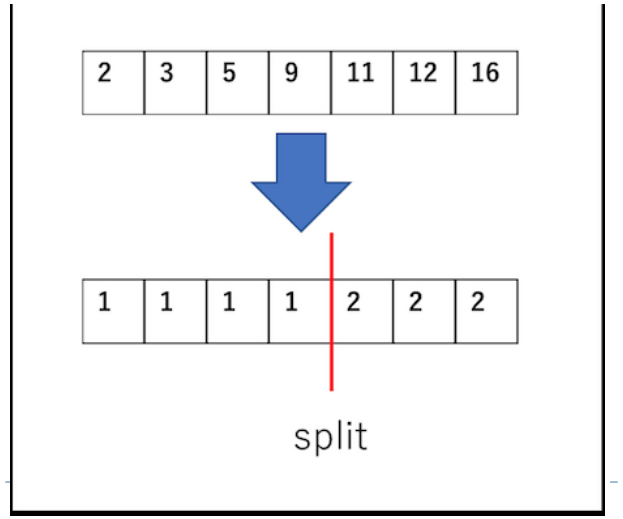


XGBoost

- Performance gain
  - XGBoost adds built-in regularization to achieve accuracy gains beyond gradient boosting. XGBoost includes regularization as part of the learning objective, as contrasted with gradient boosting and random forests. The regularized parameters penalize complexity and smooth out the final weights to prevent overfitting. XGBoost is a regularized version of gradient boosting
    - For more information about the objective function, please refer to here or here
    - In addition to the regularization term, it used an approximation similar to Newton's Method which is more accurate than naïve gradient boosting. An in depth discussion can be found here
- Take a look for how to handle categorical variables and missing value
  - Encode categorical variable before entering the algorithm
  - Missing value can be automatically handled

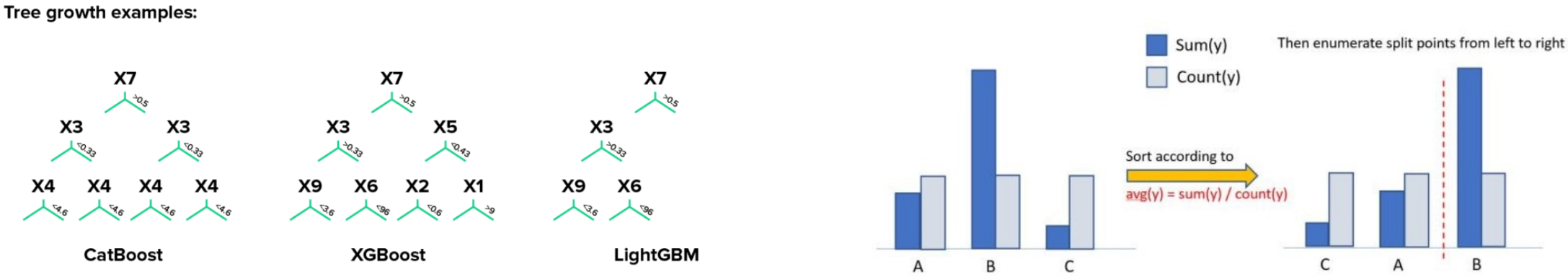
LightGBM

- For speed, use Histogram-based Gradient Boosting (HGB)
  - It works by binning the inputs features, replacing them with integers. The number of bins is controlled defaults to 255 and cannot be set any higher than this. The way the bins are built ($O(n)$) removes the need for sorting ($O(nlog(n))$) the features when training each tree
    - The complexity of split a single node reduce from $O(n_{features} \times nlog(n))$ to $O(n_{features} \times n)$
  - Binning can enormously reduce the number of possible thresholds that the training algorithm needs to evaluate. Moreover, working with integers makes it possible to use faster and more *memory-efficient* data structures



LightGBM

- Gradient-based One-Side Sampling (GOSS) which adjust the sampling strategy
  - Keeps all data instances with large gradients and performs random sampling for data instances with small gradients. Data points with larger gradients have higher errors and would be important for finding the optimal split point
- Optimization in Accuracy
  - Leaf-wise (Best-first) tree growth instead of fixed ordered, see discussion here
  - *Optimal Split* for Categorical Features
    - Use strategy similar to target encoding



https://medium.com/riskified-technology/xgboost-lightgbm-or-catboost-which-boosting-algorithm-should-i-use-e7fda7bb36bc

CatBoost

- Symmetric trees
  - CatBoost builds symmetric (balanced) trees, unlike XGBoost and LightGBM. In every step, leaves from the previous tree are split using the same condition. The feature-split pair that accounts for the lowest loss is selected and used *for all the level's nodes*
    - This balanced tree architecture aids in efficient CPU implementation, decreases prediction time and controls overfitting as the structure serves as regularization.
- Ordered boosting
  - When calculating the gradient estimate of a data instance, classic algorithms use the same data that the model was built with. CatBoost, on the other hand, uses the concept of ordered boosting, a permutation-driven approach to train model on a subset of data while calculating residuals on another subset, thus preventing target leakage and overfitting.

CatBoost

- Sampling techniques
  - MVS can be considered as an improved version of the GOSS, and provide lower variance
- CatBoost adds native supports all kinds of features be it numeric, categorical, or text and saves time and effort of preprocessing
  - Take a look at how to deal with categorical features at here
    - First it uses Borders method with one target border. The obtained float encodings are further discretized into 15 different values. Three values of Prior parameter are used to create 3 three different encodings: Prior=0/1:Prior=0.5/1:Prior=1/1
    - Also for each categorical feature, it create an encoding with Counter method. The number of categorical encoding value borders is also equal to 15, and only one value of Prior=0/1 is used

Hyperparameters

1. For Faster Speed
   - Setting bagging fraction ratio to randomly choose instances
   - Use feature sub-sampling (random subspace) by setting the fraction of features
   - Use smaller number of bins for Histogram-based Gradient Boosting
2. For better accuracy
   - Use smaller learning rate with larger number of iterations (number of estimators)
   - Use larger number of bins for Histogram-based Gradient Boosting
   - Try different categorical encoding methods
3. Prevent overfitting
   - Use larger value of number of data in leaf to avoid splitting
   - Use smaller number of depth to avoid growing deeper tree
   - Use DART (Like dropout in neural network)
   - Try to adjust regularization strength in the objective function

Hyperparameters

	XGBoost	LightGBM	CatBoost
Speed	colsample_bytree subsample n_estimator	feature_fraction bagging_fraction num_iterations	rsm iterations
Control overfitting	learning_rate (0.01~0.2) max_depth min_child_weight	learning_rate max_depth, num_leaves min_data_in_leaf	learning_rate Depth L2-leaf-reg
Categorical variable	Experimental	categorical_feature	cat_features one_hot_max_szie

Conclusion

- In conclusion, ensemble learning is versatile, powerful, and fairly simple to use. Random Forests, AdaBoost and GBRT are among the first models you should test on most Machine Learning tasks, and they particularly shine with heterogeneous tabular data. Moreover, as they require very little preprocessing, they're great to get a prototype up and running quickly. Lastly, ensemble methods like voting classifiers and stacking classifiers can help push your system's performance to its limits
- About the framework
  - XGBoost have largest community and provide sufficient support for production
  - LightGBM may be a better choice when considering the speed and accuracy
  - CatBoost is a choice when the dataset is small or when the categorical variables are important in the model

References

[1] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition
Chapter 6,7

[2] An Introduction to Statistical Learning, second edition

[3] Hands-On Gradient Boosting with XGBoost and scikit-learn

Appendix

Resources and libraries

- Gradient boosting
  - https://www.kaggle.com/code/alexisbcook/xgboost
  - https://neptune.ai/blog/when-to-choose-catboost-over-xgboost-or-lightgbm
  - StaQuest about Gradient Boosting
  - StaQuest about XGBoost
- Libraries for gradient boosting
  - XGBoost
  - LightGBM
  - CatBoost
  - https://www.tensorflow.org/decision_forests

Resources and libraries

- Dynamic selection or dynamic ensemble
  - https://github.com/Menelau/DESlib
- Ensemble for neural network
  - https://ensemble-pytorch.readthedocs.io/en/latest/

Classification Trees

- Very similar to a regression tree, except that it is used to predict a qualitative response rather than a quantitative one
  - For a classification tree, we predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs
  - Just as in the regression setting, we use recursive binary splitting to grow a tree
  - A natural alternative to RSS is the classification error rate. This is simply the fraction of the training observations in that region that do not belong to the most common class

$$E = 1 - \max_k(\hat{p}_{mk})$$

  Here $\hat{p}_{mk}$ represents the proportion of training observations in the mth region that are from the kth class

- However classification error is not sufficiently sensitive for tree-growing, and in practice two other measures are preferable

Gini index and Deviance

- The Gini index is defined by

$$G = \sum_{i=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

  - A measure of total variance across the K classes. The Gini index takes on a small value if all of the $\hat{p}_{mk}$'s are close to zero or one
  - For this reason the Gini index is referred to as a measure of node purity — a small value indicates that a node contains predominantly observations from a single class

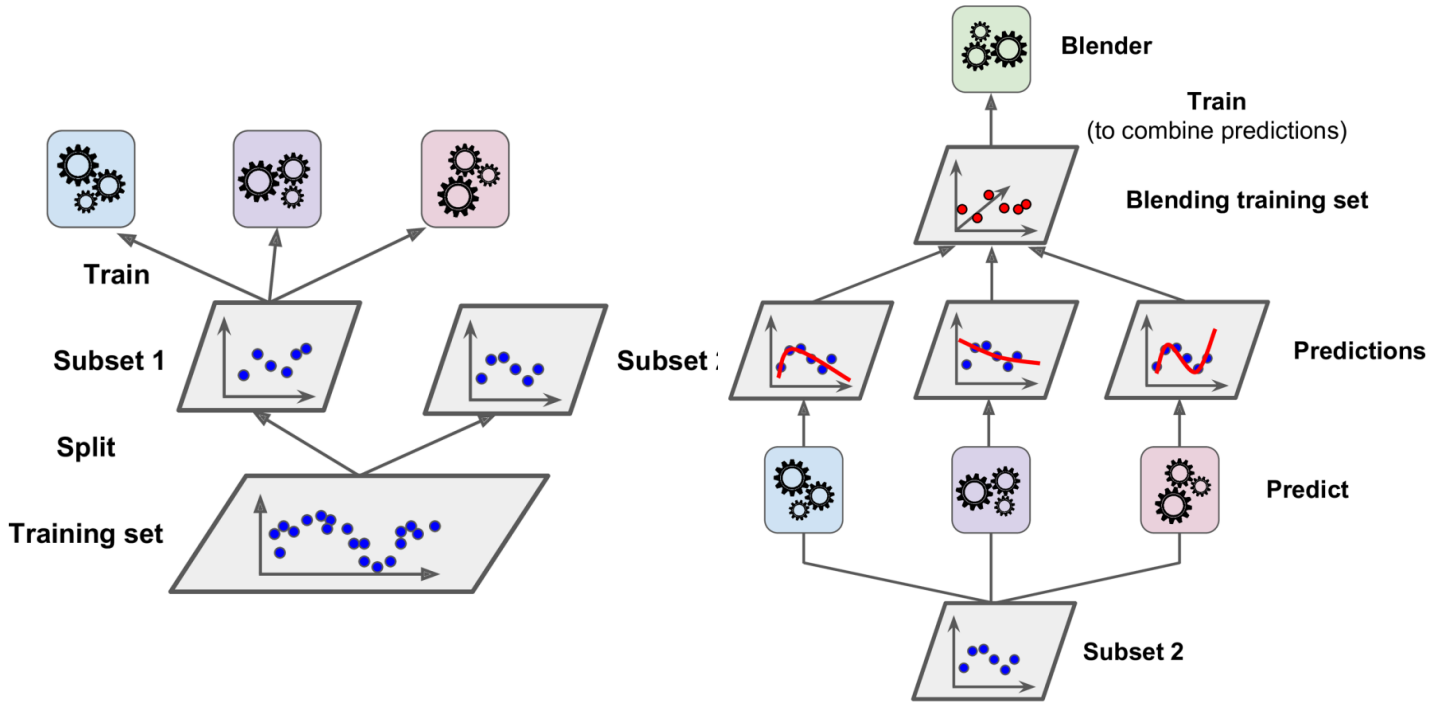
- An alternative to the Gini index is cross-entropy, given by

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

- It turns out that the Gini index and the cross-entropy are very similar numerically (differentiable)

Another way to train stacking

- To ensures that the predictions are “clean”
  - The first subset is used to train the predictors in the first layer. Next, the first layer’s predictors are used to make predictions on the second (held-out) set (since the predictors never saw these instances during training)
  - For each instance in the hold-out set, there are three predicted values. We can create a new training set using these predicted values as input features, and keeping the target values
  - The blender is trained on this new training set, so it learns to predict the target value, given the first layer’s predictions



AdaBoost

1. Initialize the weights with $w_n^1 = \frac{1}{N}$ for $n = 1, 2, \ldots, N$.
2. For $t = 1, 2, \ldots, T$ or while $\bar{L}^t$, as defined below, is less than or equal to 0.5,
   - Draw a sample of size N from the training data with replacement and with probability w_n^t for $n = 1, 2, \ldots, N$.
   - Fit weak learner t to the resampled data and calculate the fitted values on the original dataset. Denote these fitted values with $f^t(\mathbf{x}_n)$ for $n = 1, 2, \ldots, N$.
   - Calculate the observation error L_n^t for $n = 1, 2, \ldots, N$:

$$D^t = \max_n \{|y_n - f^t(\mathbf{x}_n)|\}$$
$$L_n^t = \frac{|y_n - f^t(\mathbf{x}_n)|}{D^t}$$

   - Calculate the model error $\bar{L}^t$:

$$\bar{L}^t = \sum_{n=1}^{N} L_n^t w_n^t$$

     If $\bar{L}^t \geq 0.5$, end iteration and set T equal to $t - 1$.
   - Let $\beta^t = \frac{\bar{L}^t}{1 - \bar{L}^t}$. The lower β^t, the greater our confidence in the model.
   - Let $Z^t = \sum_{n=1}^{N} w_n^t (\beta^t)^{1 - L_n^t}$ be a normalizing constant and update the model weights with

$$w_n^{t+1} = \frac{w_n^t (\beta^t)^{1 - L_n^t}}{Z^t},$$

     which increases the weight for observations with a greater error L_n^t.
3. Set the overall fitted value for observation n equal to the weighted median of $f^t(\mathbf{x}_n)$ for $t = 1, 2, \ldots, T$ using weights $\log(1/\beta^t)$ for model t.

ESL 10.9 – Boosting Trees

Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$\lvert y_i - f(x_i)\rvert$	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $\lvert y_i - f(x_i)\rvert \le \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $\lvert y_i - f(x_i)\rvert > \delta_m$ where $\delta_m = \alpha$th-quantile$\{\lvert y_i - f(x_i)\rvert\}$
Classification	Deviance	kth component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^N L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.