PRELAB #6

ENSEMBLE METHODS: Supervised learning algorithms that can improve predictive performance and model generalization. They're a class of techniques that can train multiple models and aggregate them into a single prediction

Involves combining multiple INDEPENDENT models that are predicting the same outcome.

→ Applicable to both CLASSIFICATION and REGRESSION problems.

→ WHY WE DO THIS:

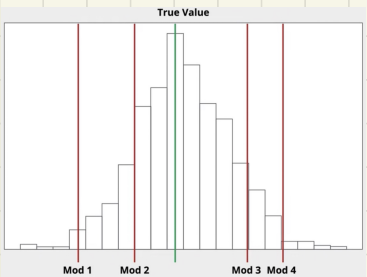
- To acheive generalization it's essential that a model has low estimation bias and variance. To acheive this through ENSEMBLE MODELING.
- Stems from model estimation and bias variance tradeoff

Model Error = Estimation bias + Estimation Variance ← consists of estimation bias & estimation variance

ESTIMATION BIAS: model rigidity that prevents adaptation to nuances of the data

ESTIMATION VARIANCE: model flexibility that causes the estimated model to be sensitive to data nuances.

- If the model has HIGHER COMPLEXITY = flexible & sensitive to nuances in the data.
- The exact contribution of each term will depend on the chosen model.
  - ↳ LOGISTIC REGRESSION has higher bias but lower variance
  - ↳ DECISION TREE has low bias but high variance
- Every algorithm will have their own estimation bias & variance tradeoff
- Different hyperparameters we choose for an algorithm will have their own estimation bias & variance tradeoff



- When we make predictions on a single example. Instead of a single prediction we get a distribution of predictions like the histogram.

GREEN LINE represents our TRUTH

RED LINE represents prediction for 4 different models.

↳ If we average the predictions of the 4 different models we'll get the CORRECT PREDICTIONS.

THE PROCESS OF BUILDING MULTIPLE MODELS WITH EACH OF IT'S OWN BIAS & VARIANCE TRADEOFF THEN AVERAGING PREDICTIONS IS THE ENSEMBLE MODELING.

→ **MODEL ESEMBLING:** The process combining different models in effort to average out the errors the individual models might exhibit.

↳ When we ensemble of models we choose models with different its own bias and variance tradeoff.

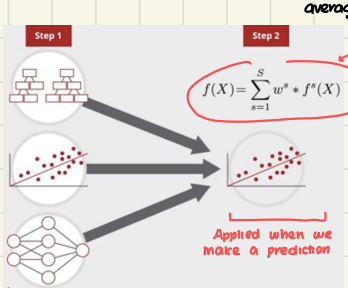
↳ ENSEMBLE MODELING TECHNIQUES:

1. **STACKING:** taking weighted combination of the predictions of a total # of K different models
   - ↳ More general procedure that doesn't have a specific supervised learning attached to it.
   - ↳ Techniques employed by specific ML algorithms that focus on building an ensemble of the same type of model, usually DT.
   - ↳ Stacking doesn't have specific algorithm implementation
   - ↳ Given a training data set we build a varied set of models. We can mix the underlying modeling algorithms s.a. using logistic regression, DT, and/or KNN.

   Aggregation method: weighted sum of the individual model predictions.

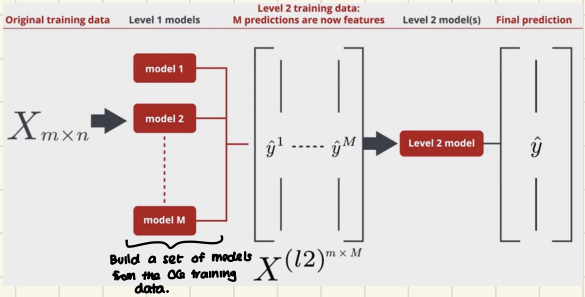
   When it comes to prediction time we run the prediction example through each model and take some aggregation of the scores.

   ↳ SIMPLIST AGGREGATION APPROACH: Average predictions than weigh each prediction separately and take the weighted average over those weighted predictions.



The nuance is in how you determine the weights.

↳ **SUPERVISED LEARNING APPROACH**: Train 2 stage model to learn the optimal weights [GETS BEST PERFORMANCE]



STAGE 1: core stacking approach

STAGE 2: Build a new training data set where the individual model predictions from STAGE 1 are the features, & the label is from the OG data set.

↳ SIMPLE LINEAR MODELS are favored

↳ **WHY STACKING HELPS**: (actually applies to all ensembles in general)

Taking weighted sums of individual models increases the chances that we can cancel out the error of any individual model then this would arrive at a model that generalizes better.

For this process to work:

FUNDAMENTAL RULE: Each individual model must be independent of each other!!!

For each base model:
1. Vary the algorithm
2. Vary the features
3. Vary the hyperparameters
4. Vary the training data

} This means our model errors likely would not cancel each other out.

Do this to get more independence and variety in our base models for stacking.

2. **BAGGING**: Generating multiple models from the same data by taking bootstrapped examples & averaging the individual model predictions

↳ Techniques employed by specific ML algorithms that focus on building an ensemble of the same type of model, usually DT.

↳ Bagging procedure with random forest. Bagging relies on **BOOTSTRAPPING** technique

- **BOOTSTRAPPING**: Process of taking multiple different samples from the data set, compute some quantity or statistic on each sample and then averaging them to get our final estimate.

↳ In model building we run a loop & in each we **sample with replacement** from our training data to create a new training set from the OG data.
↳ we can sample the same example multiple times

Then we build a model from the bootstrap sample.

For each iteration we'd use the same modeling algorithm. After the loop is finished we'd have a collection of models.

When making a prediction we run an example of all the models in this collection and take the average predictions across these models.

↳ BOOTSTRAP PSEUDOCODE:

Bootstrap Procedure for Random Forests:

For i in Num_Bootstraps:
1. Bootstrap data = Sample N examples randomly, with replacement
2. Build a Decision Tree on the bootstrap data
3. Add the ith Decision Tree to the ensemble ← AKA. store the result

Use the set of models to make predictions

→ First we need to define how many bootstrap iterations we'll use.

→ SAMPLE WITH REPLACEMENT means each example can be added to the bootstrap sample more than once provided you'll end up with the same training set size.

GOAL OF BAGGING: Taking the average cancels out the individual model errors to bring us closer to the actual GROUND TRUTH.

↳ Bagging helps by taking averages over different but similar models is an effective way to reduce a model's overall estimation variance.

↳ **RANDOM FOREST ALGORITHM**: Utilizes the bagging technique

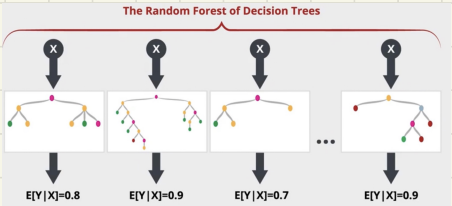
- Random forest is a set of DT.
- Each DT is expected to be different because we vary the training data by using different examples and subsets of the features.
- When training the forest we use the same exact configuration of DT hyperparameters

↳ A typical random forest can consists of dozens to hundreds of different trees:

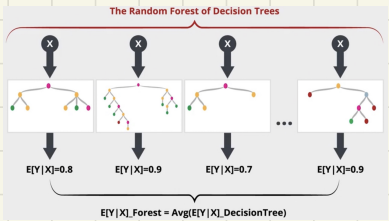
HYPERPARAMETERS: ① Total # of trees [USE STANDARD MODEL SELECTION TECHNIQUES TO FIND]

↳ Once random forest is trained the model object will store the collection of DT

- To make a **prediction**, input your feature vector into each tree into each tree to get a set of individual predictions.
  - ↳ In each individual tree the standard rules of DT apply.
    - **Classification task**: individual trees can output either class label or probability of belonging to a specific class labeled one.
    - **Regression task**: the output is the average value of the label



↳ Final prediction will be the average of individual DT predictions.



If we want a class label instead of a probability for classification problems, we can use the most common class label amongst the set of trees as predictions.

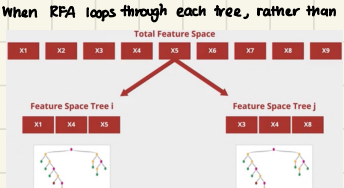
↳ For the ensemble models to be effective we need the individual models within the ensemble to be independent from each other.

- For RFA we don't vary the algorithm of the base models ∴ we need to make the individual DT more different from each other. We do this through BAGGING or BOOTSTRAPPING the data with each individual tree

The OTHER METHOD:

↳ take samples of the features that go into each tree.

↳ When RFA loops through each tree, rather than using all features for each tree, the algorithm selects a subset of them.



↳ RANDOM FOREST ALGORITHM: with the addition of random feature selection

Random Forest Algorithm:

Training

For i in N_Estimators:

1. Bootstrap data = Sample N examples randomly, with replacement
2. Randomly select a subset of features
3. Build and store a decision tree

Prediction

For a given X, get prediction from all trees and average them

Run the loop N times where N is the # of trees we want in the forest

In scikit learn:

The # of trees is controlled by a hyperparameter: n_estimators

We have to tune the random forest to get the optimal out-of-sample performance

```
from sklearn.ensemble import RandomForestClassifier

RandomForestClassifier(
    #----------------------------------------------------------------
    #Forest Level Parameters
    #----------------------------------------------------------------
    n_estimators=100, #The number of trees in the forest
    max_features='auto', #The number of features to select in each tree

    #----------------------------------------------------------------
    #Individual Decision Tree Parameters
    #----------------------------------------------------------------
    max_depth=None, #Max depth of individual trees
    min_samples_split=2, #min number of examples to split in each tree
    min_samples_leaf=1, #min number of examples that can be in leaf node
)
```

n_estimator is a FOREST LEVEL HYPERPARAMETER

KEY HYPERPARAMETERS to consider can be grouped into forest level parameters and individual tree level parameters

Tree level parameters: will be the same for each tree in the forest.

GENERALLY we want to overfit the individual DT ∴ we rely on DEFAULTS.

For most hyperparameters you'll want to find a value that's neither TOO SMALL nor TOO LARGE. B/C hyperparameters usually control the balance btw over and underfitting.
↳ The n_estimator parameter increases it without leading to overfitting.

TEST DATA: random forest will severly overfit the training data.
↳ If AUC = 1 this means perfect predictions and this is usually a bad sign but its common for RF.

TAKEAWAY: Tune the n_estimator's hyperparameter to get the right balance between model performance & model scalability.

FORMALIZING RANDOM FOREST:

RECALL: The complexity of a supervised learning is controlled by its hyperparameters.
↳ Model TOO COMPLEX ⟹ OVERFITS ⟹ HIGH VARIANCE ERROR
↳ Model TOO SIMPLE ⟹ UNDERFITS ⟹ HIGH BIAS ERROR

A DT is susceptible to HIGH VARIANCE but the optimal balance of LOW BIAS + LOW VARIANCE is achieved through ENSEMBLE OF DT (AKA. RANDOM FOREST)

Number of Trees	Tree Depth	Model Estimation Error	Behavior
1	No limit	High variance, low bias	Overfitting
1	Shallow	Low variance, high bias	Underfitting
High	No limit	Low variance, low bias	Good fitting

RANDOM FOREST: the ensemble of DT that generalizes better than a individual DT.

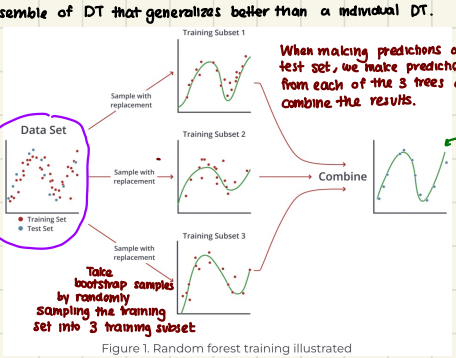


Figure 1. Random forest training illustrated

CLASSIFICATION VS. REGRESSION:

Classification: resulting outputs from each tree aggregated using majority vote, the mode of the ensemble predictions.

Regression: resulting outputs from each tree are generally aggregated using mean.

TRADEOFF: While RF does better job at generalization than single trees, the training time & prediction time are more costly than single tree.
∴ We can't have an arbitrary large # of trees.
∴ ML engineer has to find the balance between cost & performance.

3. BOOSTING: Iteratively building models by focusing on the cumulative errors from prior iterations predictions.
↳ Can be done with using a combination of any common supervised learning algorithms (S.A. logistic regression or DT)
↳ Most mathematically rigorous
↳ First we build an initial model then initiate a loop that performs the following procedures:
- Take the model predictions from the current iteration and compute the error instead of the OG label
- Then build a new model at each iteration that predicts the error instead of the OG label & we add that to the collection of models. After doing this n times we have an ensemble of models where each one aims to reduce the prediction error.

↳ GRADIENT BOOSTED DECISION TREES ALGORITHM (GBDT): Ensemble method consisting of individual DT
↳ STRUCTURE OF GBDT:

Weight determined by learning algorithm

$$f(X) = \sum_{i=0}^{N} v_i * T_i(X)$$

Weighted sum over N individual trees
Individual trees tuned to predict errors

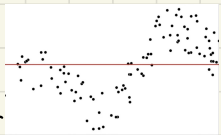
BASE MODELS: individual DT
FULL MODEL: weighted sum over the trees

In RF we'd take a simple average over individual DT
In GBDT each tree has a weight that the learning algorithm determines

↳ RFA vs. GBDT COMPARISON:

- RFA
  - ↳ Each DT uses subset features
  - ↳ Bootstrap sample of data
  - ↳ Typically overfit
- GBDT
  - ↳ Uses all features
  - ↳ Use OG data
  - ↳ Generally shallow (i.e. max_depth < 6)
  - ↳ Each tree trained on cumulative prediction error, not OG label

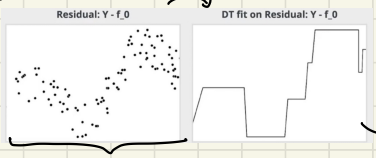
we'll control with max depth tree parameter



Start with simple model which is the RED LINE
↳ simple model is a uniform guess of the average value of the target variable or label

NEXT STAGE

next step



residual between the data and the first model which was just the average value of the target variable.

Series of steps we repeat iteratively:
↳ compute the residual or error
↳ REGRESSION: the difference between the true value of Y and the prediction up until this point.

After finding a weight for this new tree we'll add this tree to the ensemble.

AND REPEAT!
We compute the residual again, this time computing it with the latest tree added.
This process repeats until we hit our stopping criteria
↳ # of trees we specify

Increasing the # of trees always improves performance but at the cost of scalability.
This is NOT TRUE in GBDT, ↑ the # of trees can result in OVERFITTING + reduce scalability.

CHOOSING AN ENSEMBLE METHOD

- These techniques require more data ∴ if you have large data use ensemble
- These techniques tend to be slower to train & make predictions
  - ↳ If you want fast predictions maybe use something like LOGISTIC REGRESSION
- Well fit ensembles generalize better, we lose INTERPRETABILITY.
  - ↳ INTERPRETABILITY: ability to understand why a particular prediction was made given an example's features.
  - ↳ If working in an environment with strict model interpretability rules, ensemble methods are NOT appropriate.

FORMALIZING GBDT

Start off by initializing a very simple (Shallow) DT and compute error (residual) of this tree with respect to the training data set.
Then we train another tree against this residual so that this new tree represents the amount of error in this tree.
Next we use this new residual tree as "gradient" and move our OG tree towards the direction of this residual tree by a small step.
Finally we repeat this process over again until we've reached the # of trees that we desire.

GBDT visualization:

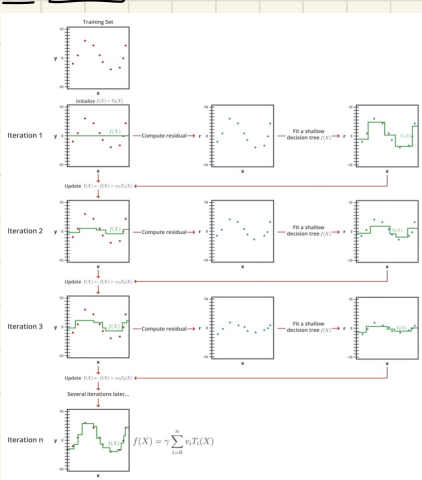


Figure 1. GBDT training illustrated for regression

In Figure 1 we start with a very simple model $f(X) = T_0(X)$, one that always produces a prediction of 0 regardless of the value of X. We subsequently calculate how much each point is off from the data, X, to get our residuals, as represented by the blue dots. We then train a shallow decision tree $T_1(X)$ against the blue dots to get our "gradient" tree. Next, we produce a new $f(X)$ by adding $\gamma \cdot v_1 \cdot T_1(X)$ to our previous version of $f(X)$, where γ is a learning rate and v_1 is an optimized weight. The above steps repeat until we reach the number of trees desired.

As you can see, the residual graph gradually flattens out as the error of $f(X)$ is reduced. Also, observe that our ensemble model $f(X)$ becomes more and more fitted against the training set as the number of iterations increases.

GBDT PSEUDOCODE:

Initialize $f(X) = T_0(X) = 0$, the average of y.

For i from 1 to N (the total number of boosted trees desired):

1. Compute the residual.
2. Train a shallow tree $T_i(X)$ against the residuals.
3. Identify the optimal weight v_i.
4. Update $f(X) \leftarrow f(X) + \gamma \cdot v_i \cdot T_i(X)$.

RF vs. GBDT:

Random Forest	GBDT
Consists of deep trees	Consists of shallow trees
Trains on randomly sampled data	Trains on residuals
Uses a subset of features	Uses all features

Table 1. Comparison between random forest and GBDT

Common:

- Both ensemble methods consist of multiple DT
- Both are excellent in reducing error due to bias & variance

GBDT OPTIMIZATION

- GBDT is susceptible to overfitting
- To mitigate overfitting risk run a thorough hyperparameter search to find the middle ground between over & underfitting.
- Too many hyperparameters & GBDT involves several trees, hyperparameter optimization will take too long!
- Scikit-learn to build GBDT to avoid overfitting:
  - ↳ Hyperparameters to consider:

Hyperparameter	Description	Recommended Range
n_estimators	Number of trees in the ensemble	{100,200,300}
learning_rate	The amount by which each tree's predictions are reduced	{0.05, 0.1}
max_depth	The max depth of individual trees	{4,5,6}

→ reduces or shrinks the weight of each DT in the ensemble

Ensemble level hyperparameters

← For individual trees

The ↑ the it == more complex your model == more likely to overfit.

For tree level hyperparameter each individual tree will have the same parameters applied to them.

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import roc_auc_score

for ne in [100, 200, 300]:
    for lr in [0.05, 0.1]:
        for md in [4,5,6]:

            model = GradientBoostingClassifier(n_estimators = ne,
                                               learning_rate = lr,
                                               max_depth =  md)

            model.fit(df_train[X], df_train[y])
            test_predictions = model.predict_proba(df_test[X])[:,1]

            test_auc = roc_auc_score(df_test[y], test_predictions)
```

Loop through different combinations

Loop through the combinations and perform out of sample validation

GBDT specific model setup

Method specific HYPERPARAMETERS

Standard fit & out of sample evaluation

ASK THE EXPERT VIDEO:

- Data scientists
  - ↳ Role is a combination of statistics, ML, and programming with domain expertise in operating space.
  - ↳ Build models that help businesses move foward & draw insights from their vast amount of data they have.
  - ↳ Build model prototype
  - ↳ Create model pipeline
- ML engineers
  - ↳ Intersection of data science & ML & software engineering
  - ↳ Goal is to deploy & manage ML models
  - ↳ Transform that data science & deploy it in production, scale it, and monitor & debug to ensure it's not causing wrong predictions or harm.
  - ↳ Use tools to scale & monitor & debug model pipelines
- Data engineering
  - ↳ Set up infrastructure on which the data scientists & ML engineers collaborate & do their work
  - ↳ Responsible for data storage, data transportation.
  - ↳ Primarily software engineers but specialize in data pipeline
  - ↳ Ensures systems have what they need to deploy them.

CHEAT SHEET

Model Debugging

If a model is not performing well, there are several ways to improve its performance. To determine which of the many techniques to use, the first step is to identify the root of the problem.

	High Variance	High Bias
Description	Models with high variance are capable of memorizing many more properties of the training data and do not do well on unseen data, resulting in low training error but high test error.	Models with high bias are too simplistic or make ill-suited assumptions and therefore cannot even achieve low error on the training data set.
Symptoms	Training error is much lower than test error.	Training error is higher than a desired error threshold.
Remedies	• Add more training data • Reduce model complexity (complex models are prone to high variance) • Bagging	• Use a more complex model (or use nonlinear models) • Add features • Boosting

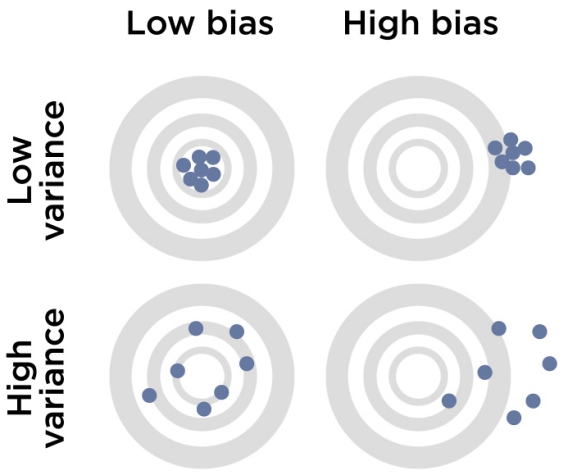
Visualize Variance and Bias

The graph below exemplifies data with high/low variance and high/low bias as "darts" thrown at a target. The bullseye at the center of the target is the location of the perfect classifier on the testing data. The blue dots illustrate the darts, which represent classifiers trained on different training data sets.

High variance/low bias models perform well when performance is averaged over large data sets. In expectation they are close to the bullseye but can perform very differently on any two particular data sets. We say that these models are overfit; that they have learned to predict meaningless noise patterns in the data.

High bias/low variance settings lead to models that are very similar across different training data sets (the blue dots are close together); however, they are systematically off-target (i.e., they make wrong assumptions).

The worst case is **high bias and high variance**. The goal is to achieve a model with low bias and low variance.



CHEAT SHEET

Random Forest

Algorithm Name	Random forest
Description	Random forest combines decision trees into an ensemble. The models are trained independently (possibly in parallel) on data bootstraps with one small modification: For each split in each tree, a small subset of k features is randomly sampled and all other features are ignored. This procedure reduces the variance of the final model as training trees on random features results in them being uncorrelated further.
Applicability	Classification and regression problems
Assumptions	Data set is not too high dimensional; similar inputs have similar labels.
Underlying Mathematical Principles	Full depth trees have high variance. Random forests combine many high-variance models to create a low-variance ensemble.
Hyperparameters	• Number of trees N to average. • Number of features k to sub-sample. Random forests are famously insensitive to hyperparameters. Default choices: N large enough until the predictions converge (N=100, or N =1000) $k = \sqrt{d}$, where d is the dimensionality of the input data
Setting	Regression or classification. While **regression** trees return continuous values, we can still use them to solve **classification** problems with discrete labels. For example, we can return the sign of the output of the tree.
Out-of-Bag Error	Because each tree is not learned on the full data but only a sub-sample, each tree has its own out-of-bag subset of the training data on which it was not trained. One can estimate the test error of a random forest classifier by computing the classification error of each data point obtained by averaging the predictions of those ensemble members that were **not** trained on this data point.
Ensemble	Combination of many decision trees

Prediction Confidence	Random forests naturally provide prediction confidences. For example, in classification settings, you can output the percentage of trees that predicted the most common label as a statistic of confidence. In regression settings, you can output the variance of the predictions of all ensemble members.
Feature Importance	Random forests can naturally provide a measure of feature importance. For example, for each feature, compute the average (or total) reduction in loss obtained on the training set through splitting on this particular feature.
Strengths	Random forests are particularly well suited for applications with heterogeneous features. The big advantages of random forests are that they tend to not overfit to the training data, are amazingly insensitive to hyperparameters, work naturally for regression and classification settings, provide feature importances, output prediction confidences, and provide an unbiased estimate of the testing error directly from the training set.
Weakness	Random forests tend to excel in lower-dimensional feature spaces (<1000 dimensions) and are often not well suited for very high and sparse feature spaces. For very large data sets (n in the millions), random forests can become slow to evaluate, as the trees become too large.

CHEAT SHEET

Gradient Boosted Decision Trees (GBDT)

Algorithm Name	GBDT (an acronym for gradient boosted decision trees)
Description	GBDT is an iterative model that progressively refines its predictions by combining multiple decision trees. The first "tree" simply predicts the average of the output variable then each successive tree is trained on the residuals from the predictions at the previous level.
Applicability	Classification and regression problems
Assumptions	Independent and identically distributed data
Underlying Mathematical Principles	Adds trees to the ensemble, reduces the residual error in each iteration until there is none
Hyperparameters	Number of trees to add, depth of the trees, and learning rate
Setting	Note that while **regression** trees return continuous values, we can still use them to solve classification problems with discrete labels. For example, we can return the sign of the output of the tree.
Loss Function	For regression, squared loss; for classification, log loss