3011979 Practical Python for Data Sciences and Machine Learning

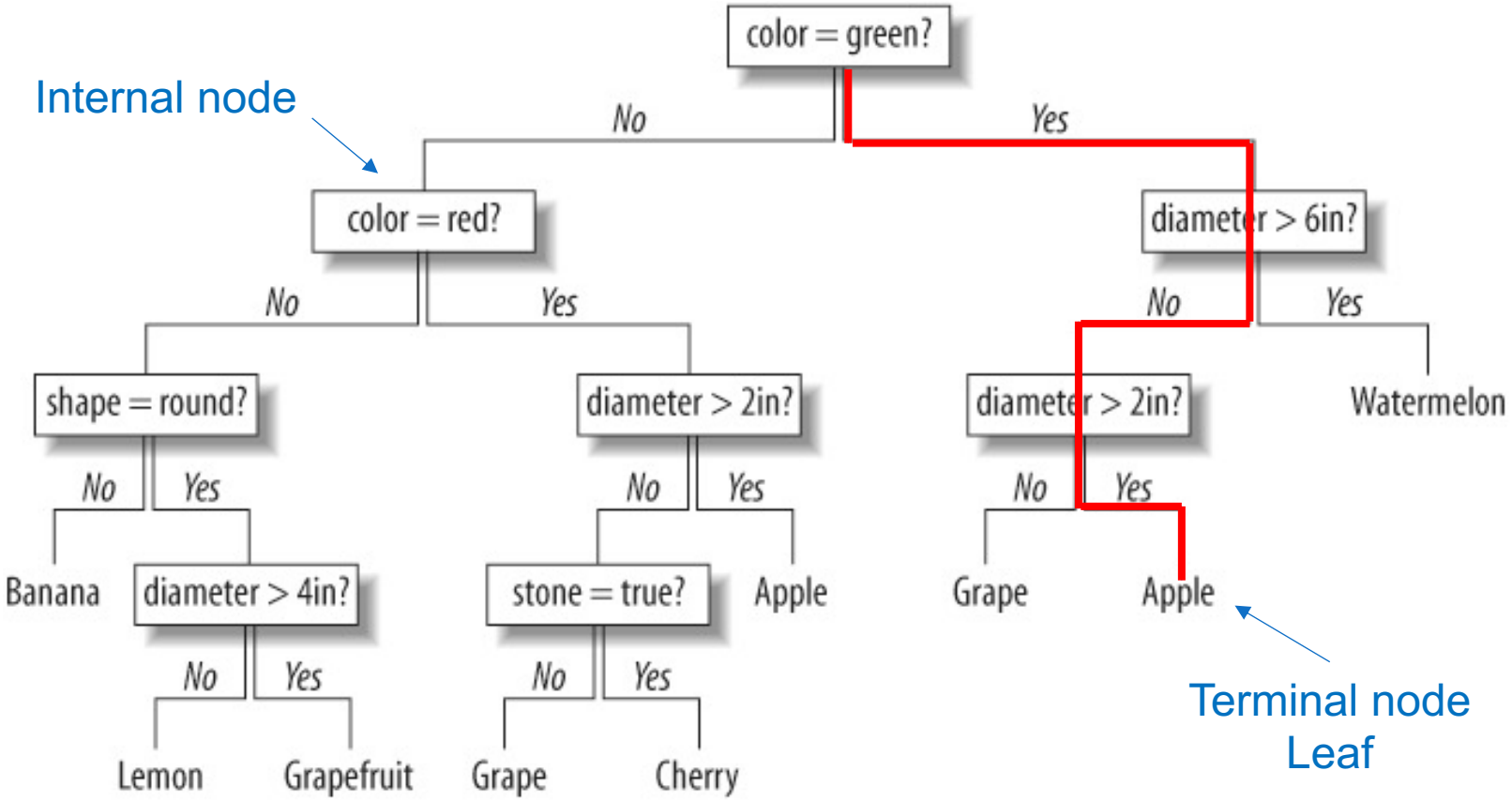
L8: Tree models

Mar 4th, 2022

Sira Sriswasdi, Ph.D.
Research Affairs, Faculty of Medicine
Chulalongkorn University

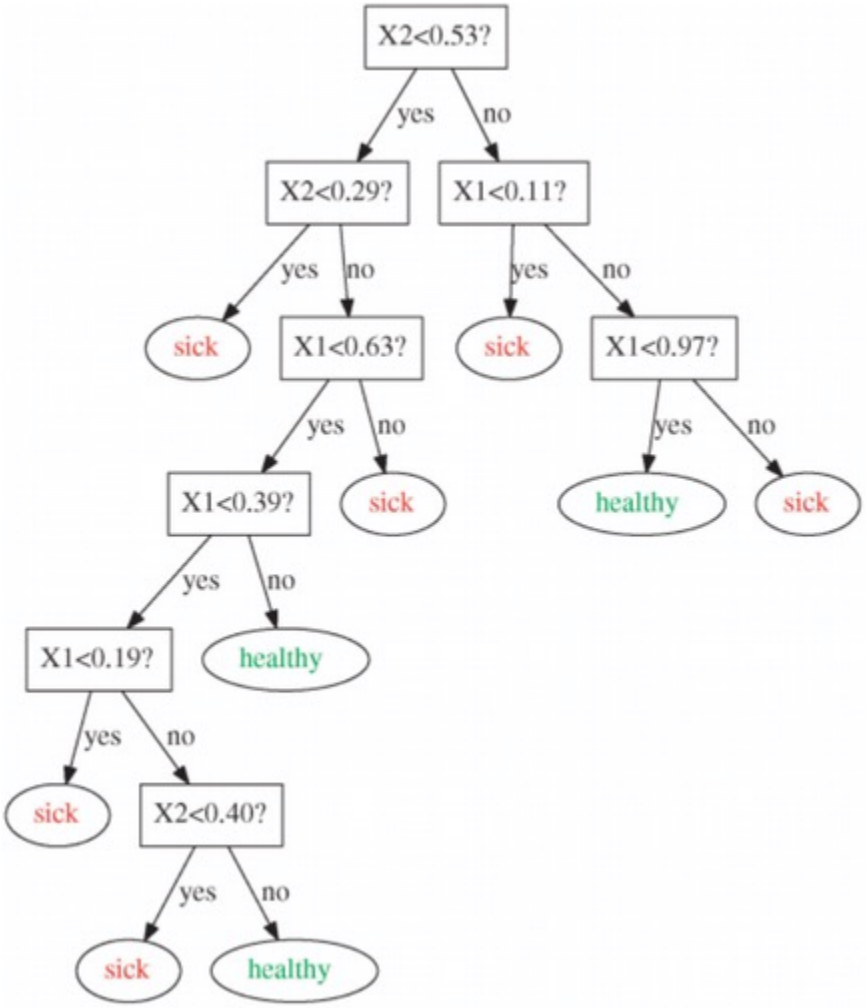
Decision tree

Decision tree



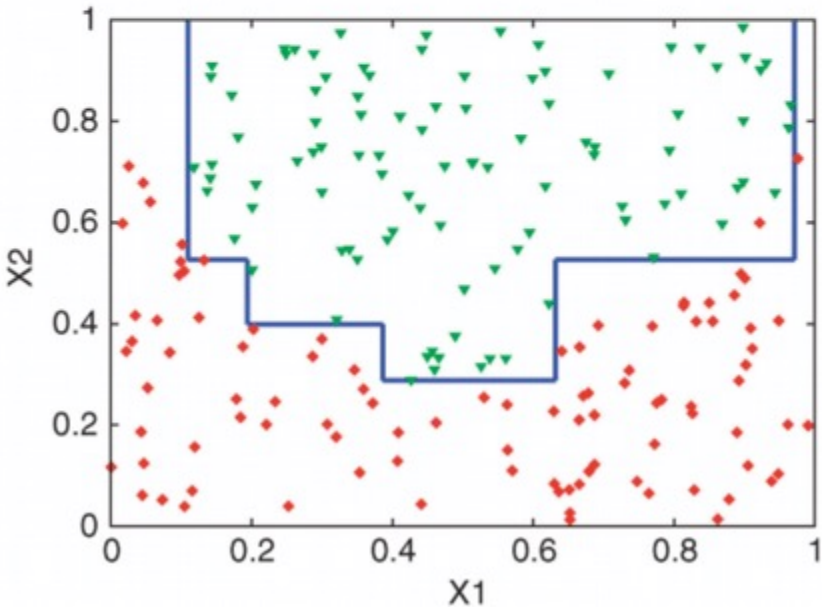
Source: Programming collective intelligence by Toby Segaran

Decision boundary



Piecewise parallel to each axis

$x_1 < 0$ → vertical line at $x_1 = 0$
$x_2 > 5$ → horizontal line at $x_2 = 5$



Miller, C. "Screening meter data: Characterization of temporal energy data from large groups of non-residential buildings"

Quality measurement of an internal node

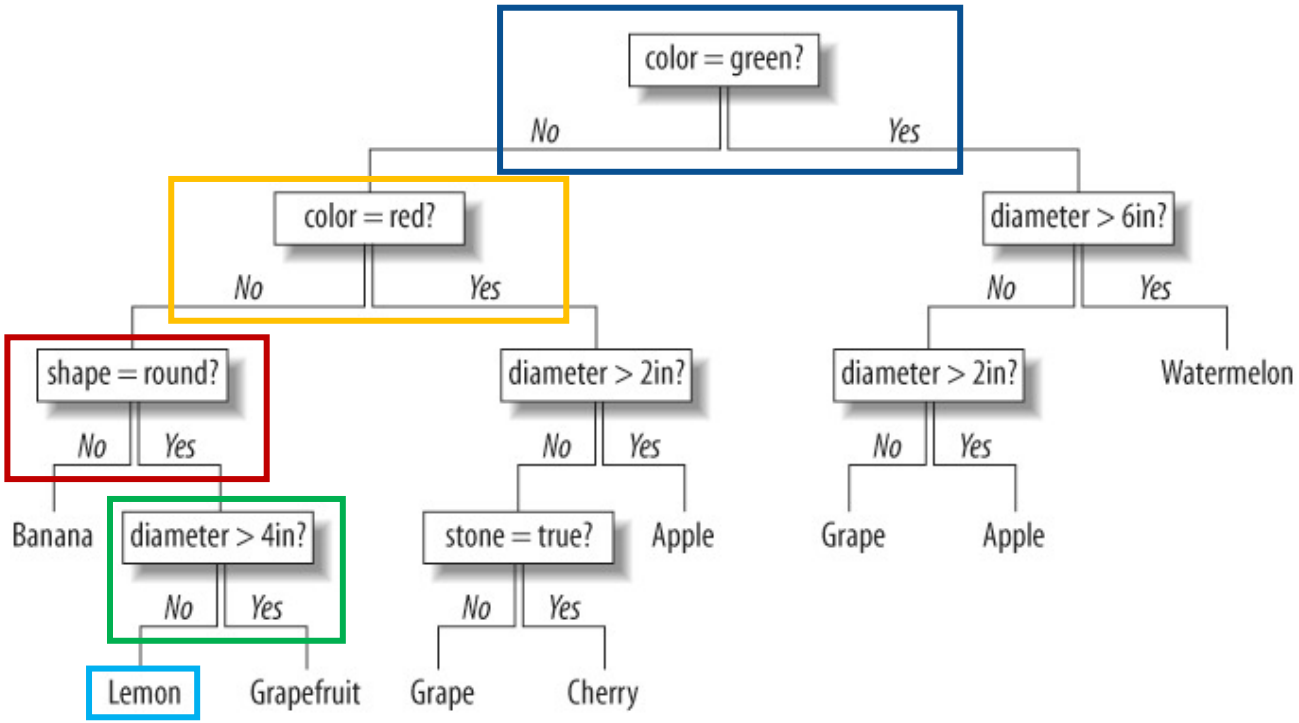
100 healthy, 100 sick

85 healthy, 5 sick

15 healthy, 95 sick

- Construction of a decision at internal node
  - Pick a feature
  - Pick a cutoff for splitting
- Evaluate the quality of samples on the left and right branches
  - How good is the 85 healthy, 5 sick and 15 healthy, 95 sick split?
    - **Gini:** $\sum p(1-p)$
    - **Entropy:** $-\sum p \log p$
  - Compare Gini or Entropy over no-split

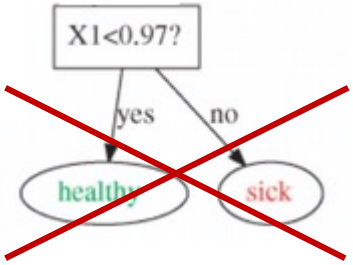
Decision tree building



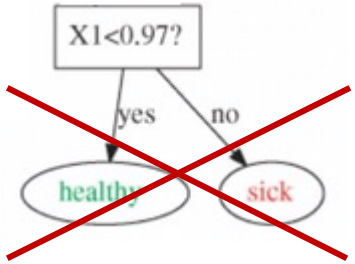
- Greedy algorithm
  - At each internal node, find the feature-cutoff combination with the lowest Gini or entropy score
    - Sample feature and cutoff
  - But using the best combination at current node may not lead to the best decision tree at the end

Controlling the complexity

2 healthy, 3 sick



100 healthy, 10 sick

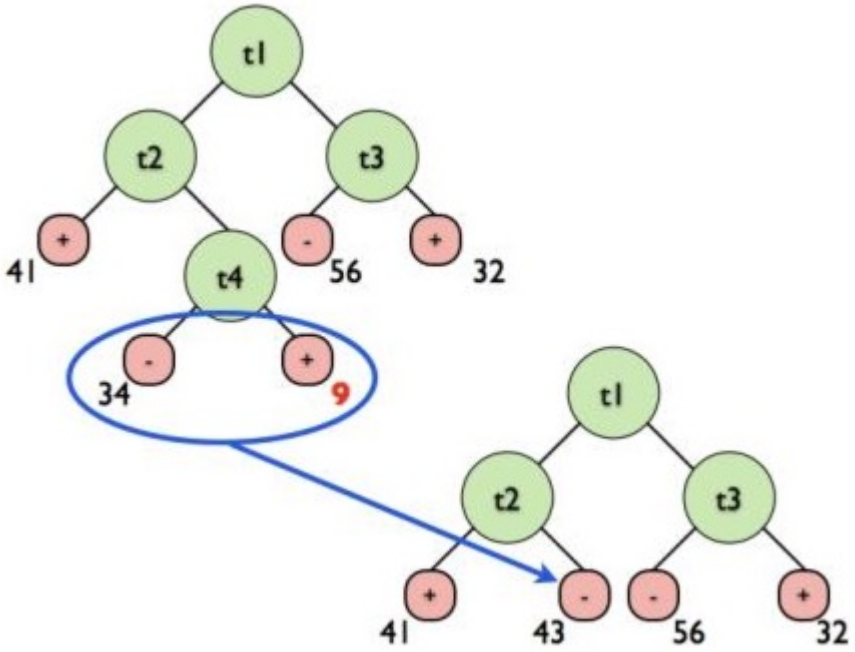


95 healthy, 5 sick

5 healthy, 5 sick

- Depth
  - Stop splitting at certain depth
- Minimum number of samples for splitting
  - Do not split if one branch contains too few samples
- Minimum impurity decrease
  - Do not split if the gain in quality is too low
- Minimum leaf size
  - Stop splitting if the current number of samples is too low
  - Prevent overfitting

Tree pruning



Patel, N. and Upadhyay, S. “Study of Various Decision Tree Pruning Methods with their Empirical Comparison in WEKA” IJCA 2012

- Model complexity = number of decisions
  - $R_\alpha(\text{Tree}) = \text{Performance}(\text{Tree}) + \alpha \cdot \text{Number of leaves}$
  - α indicates the penalty on tree complexity

Benefit of tree pruning

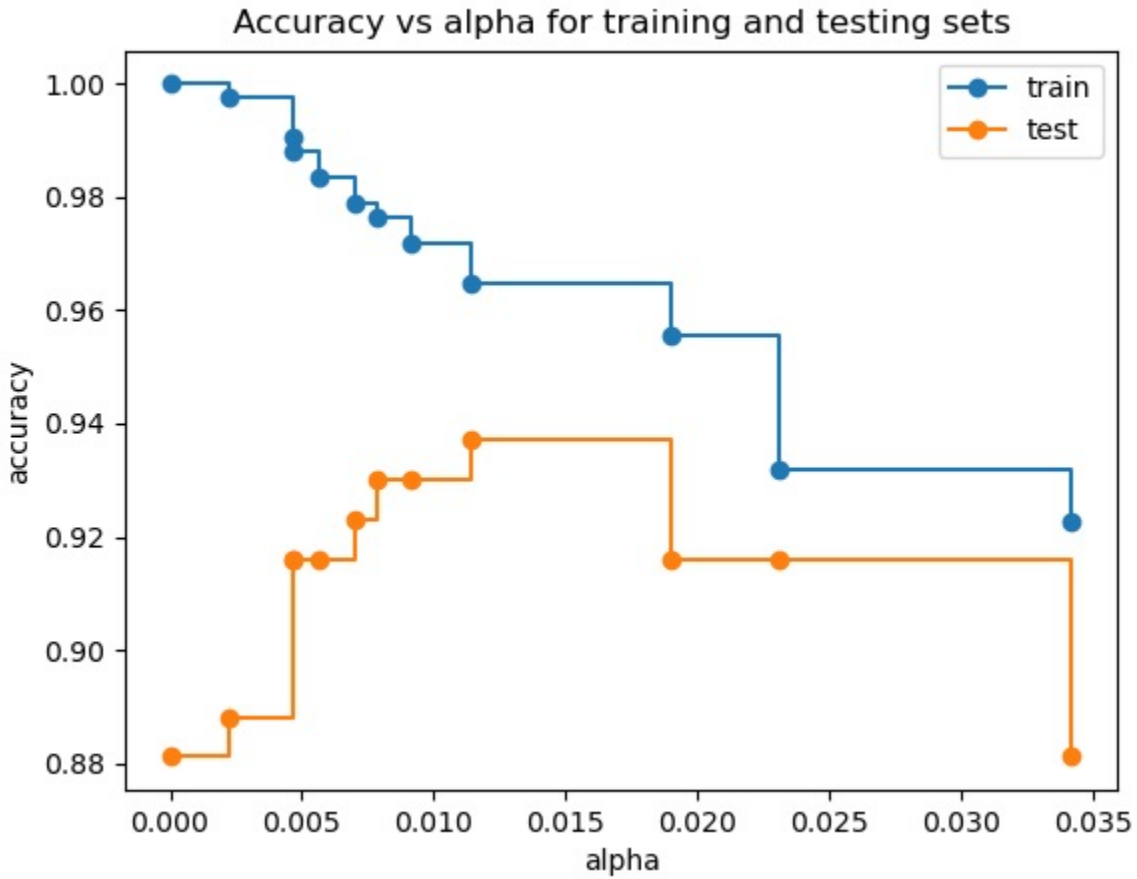
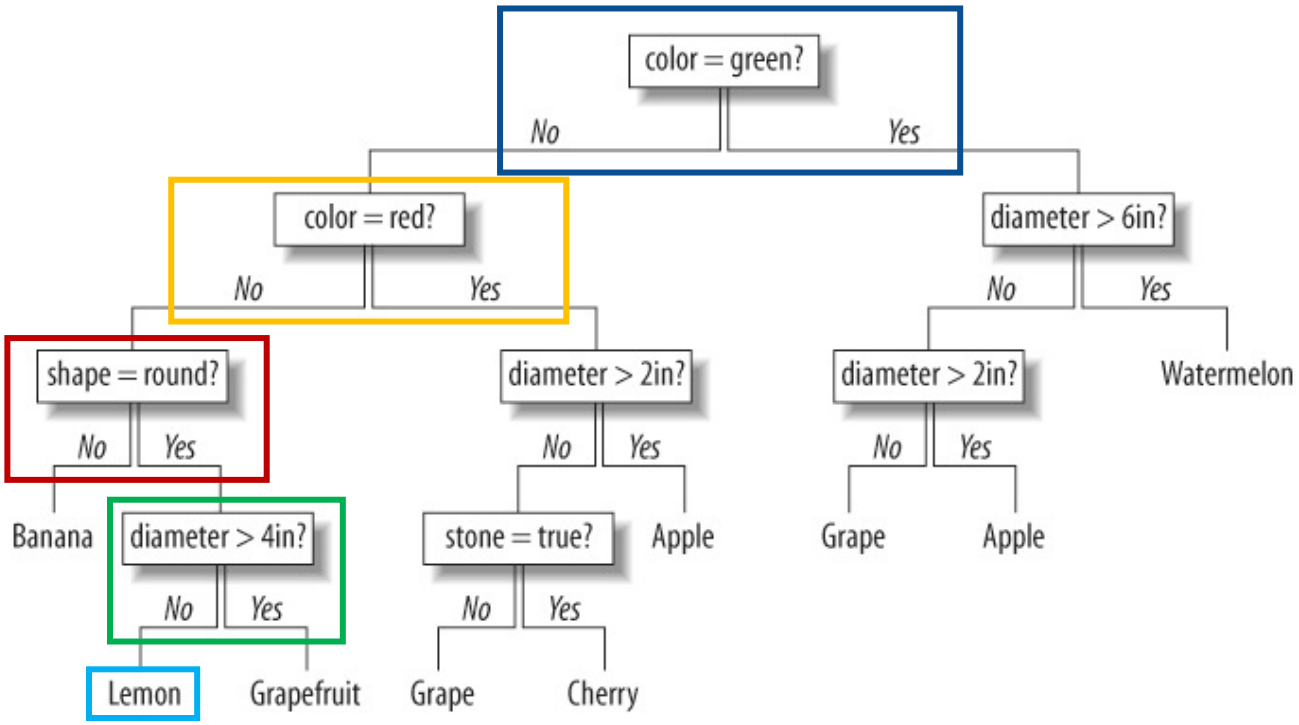


Image from scikit-learn,org

- Moderate pruning improve generalizability

Feature bagging



- At each node, different subsets of features were allowed
  - Default = All features
  - Other options are sqrt(n_feature) or log2(n_feature)
- This is a type of regularization!
  - The model cannot rely on the same features every time

Decision tree in Python

sklearn.tree.DecisionTreeClassifier

class `sklearn.tree.DecisionTreeClassifier`(*, *criterion='gini'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *class_weight=None*, *ccp_alpha=0.0*) [source]

A decision tree classifier.

Read more in the User Guide.

Parameters:

criterion : *{"gini", "entropy"}, default="gini"*
The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

splitter : *{"best", "random"}, default="best"*
The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth : *int, default=None*
The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split : *int or float, default=2*
The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Extending decision tree to regression

Predictors | Target

Outlook	Temp	Humidity	Windy	Hours Played
Rainy	Hot	High	False	26
Rainy	Hot	High	True	30
Overcast	Hot	High	False	48
Sunny	Mild	High	False	46
Sunny	Cool	Normal	False	62
Sunny	Cool	Normal	True	23
Overcast	Cool	Normal	True	43
Rainy	Mild	High	False	36
Rainy	Cool	Normal	False	38
Sunny	Mild	Normal	False	48
Rainy	Mild	Normal	True	48
Overcast	Mild	High	True	62
Overcast	Hot	Normal	False	44
Sunny	Mild	High	True	30

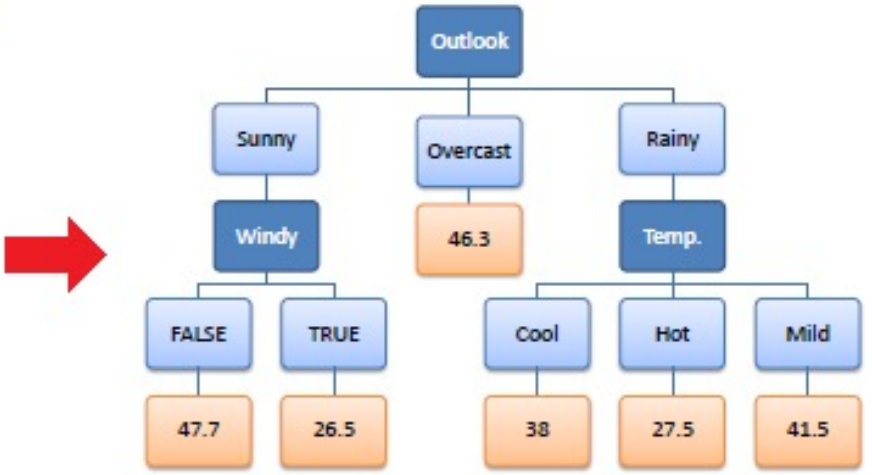


Image from saedsayad.com

- Each decision split data points with similar values (instead of coming from the same classes)
- Splitting criterion = MSE, MAE, etc.
  - Predict mean/median value of data points in each leaf

Decision tree regression prediction

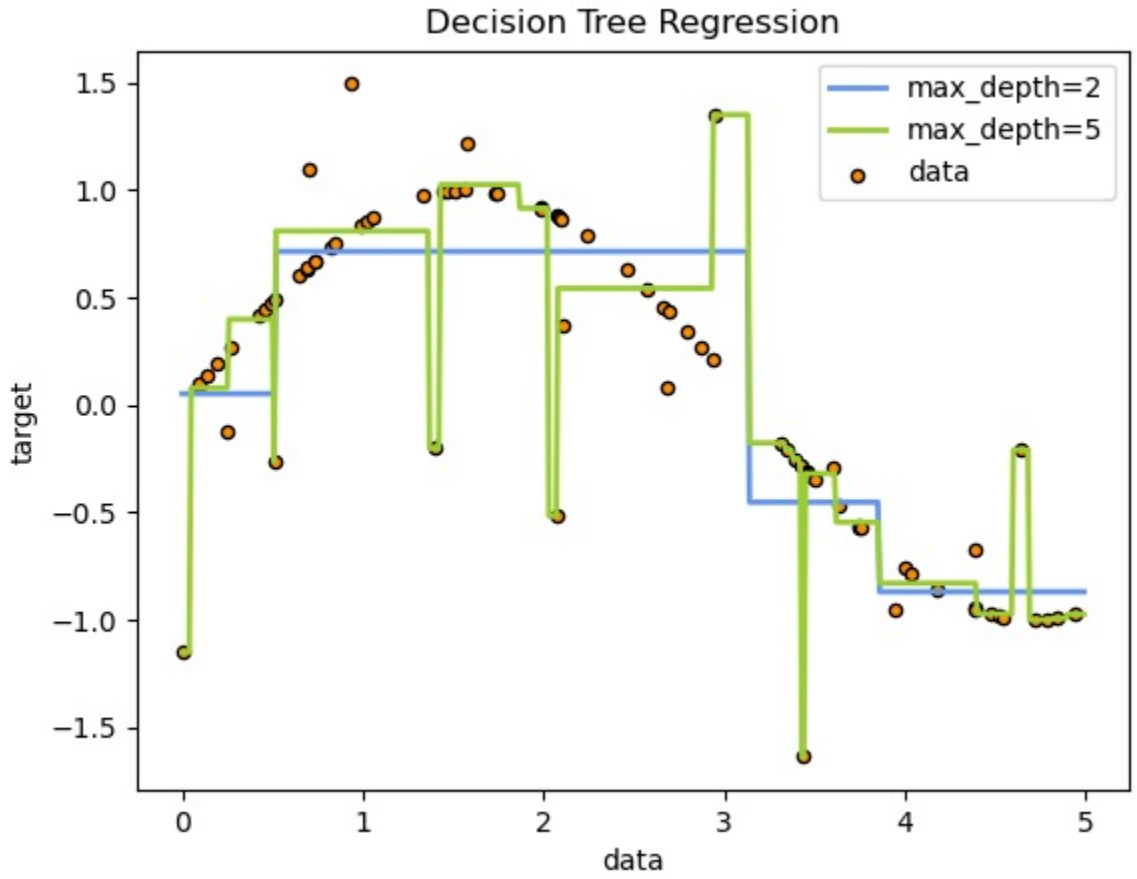


Image from scikit-learn,org

- Piece-wise constant function

Linear Tree Model = the best of both worlds

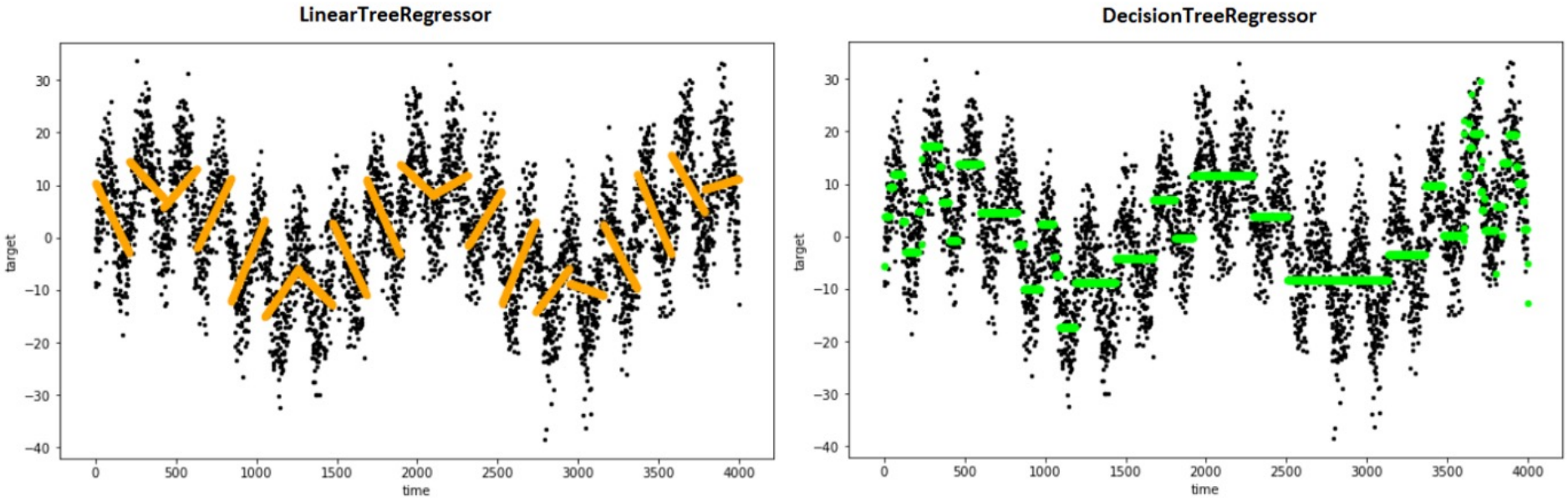


Image from towarddatascience.com

- Fit linear regression model on each leaf instead of predicting the mean values

Extending decision tree with bagging and boosting

Bagging (bootstrap aggregating)

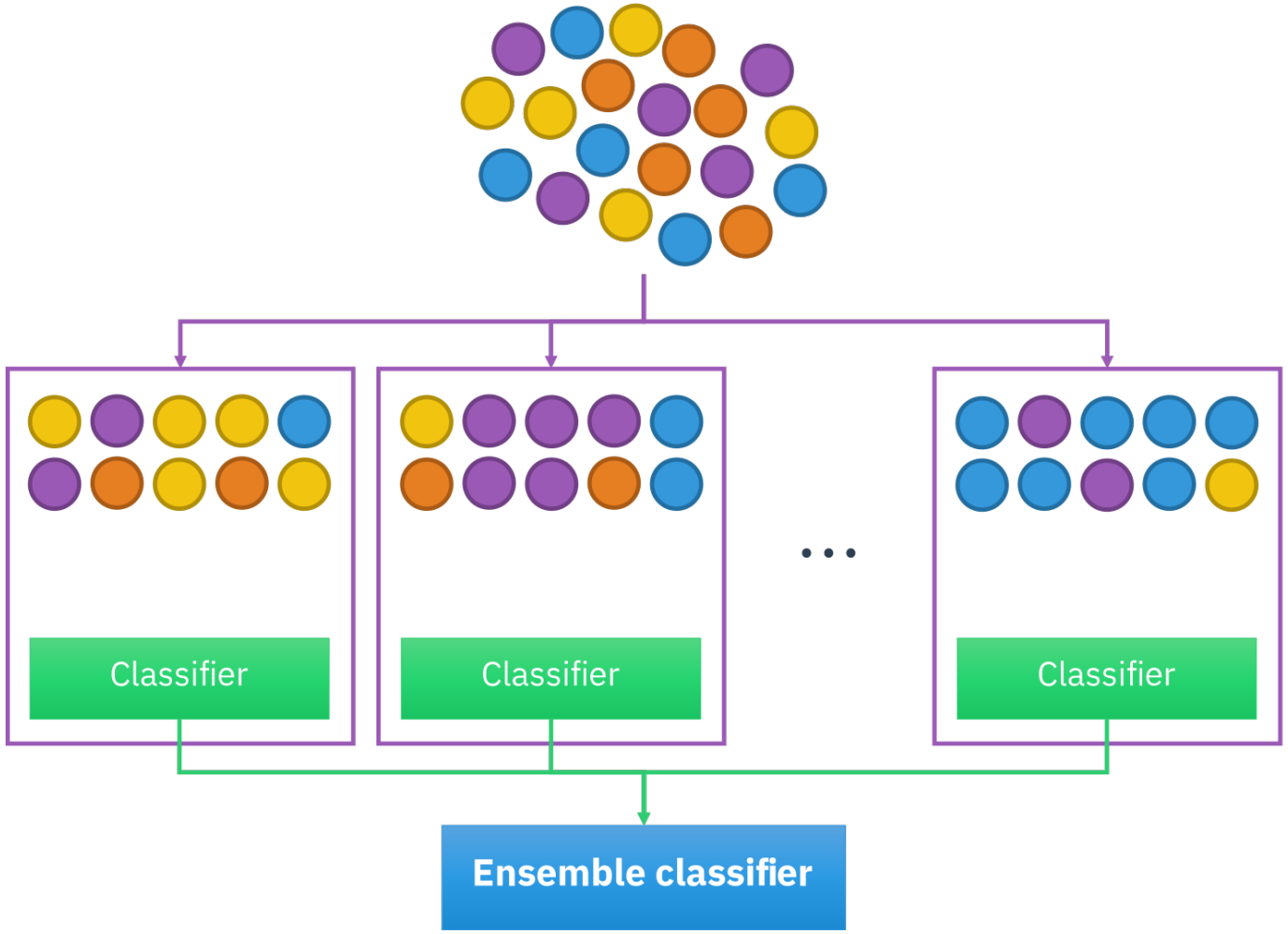
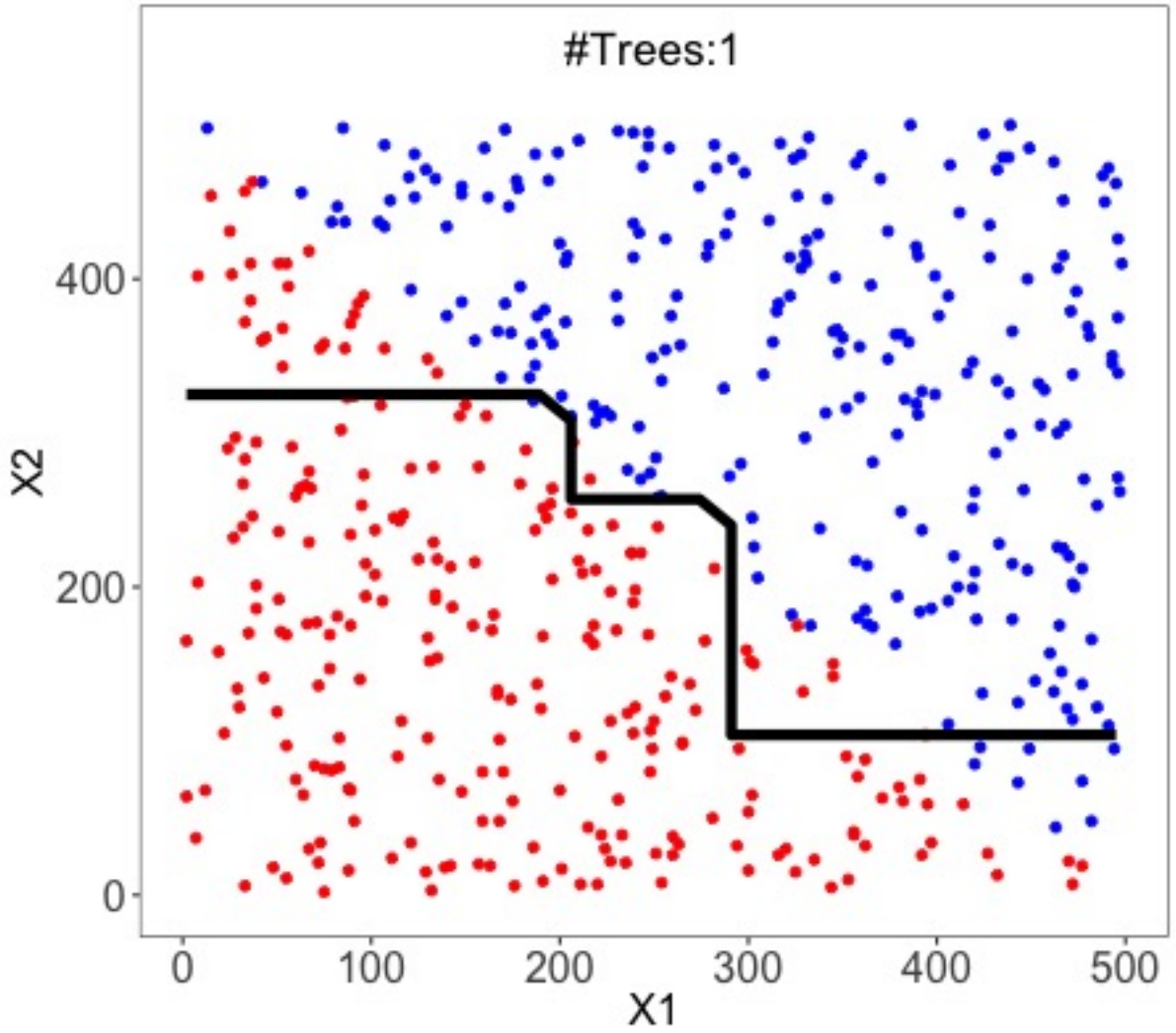


Image from wikipedia.com

Impact of bagging



The origin of random forest

Published: August 1996

Bagging predictors

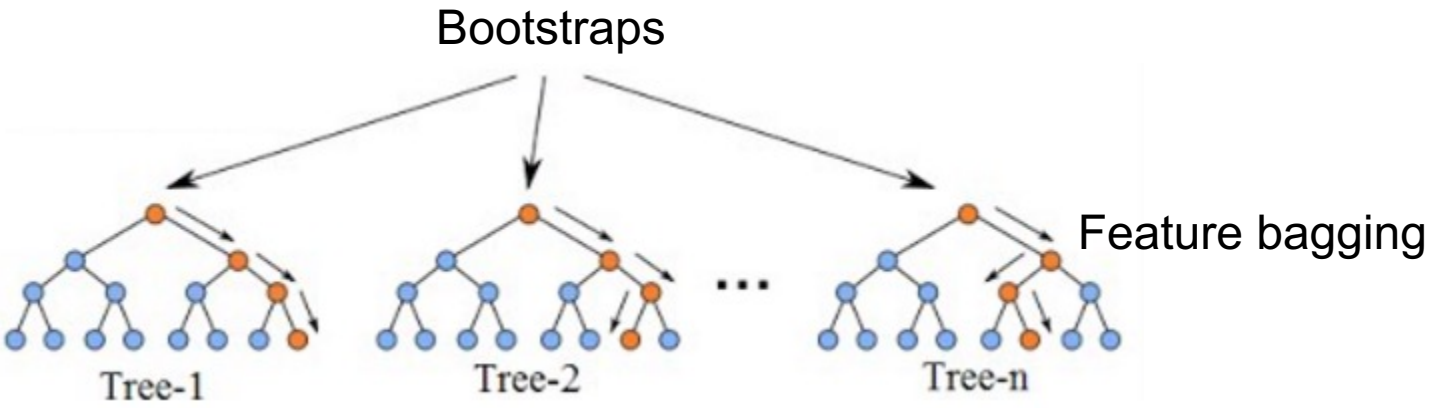
Leo Breiman

Machine Learning **24**, 123–140(1996)

Abstract

Bagging predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor. The aggregation averages over the versions when predicting a numerical outcome and does a plurality vote when predicting a class. The multiple versions are formed by making bootstrap replicates of the learning set and using these as new learning sets. Tests on real and simulated data sets using classification and regression trees and subset selection in linear regression show that bagging can give substantial gains in accuracy. The vital element is the instability of the prediction method. If perturbing the learning set can cause significant changes in the predictor constructed, then bagging can improve accuracy.

Random forest = bootstrap + feature bagging



- Bootstrap bagging prevent the model from overfitting samples
- Feature bagging prevents the model from overfitting features
- Both leads to generation of multiple slightly different models

Random forest in Python

sklearn.ensemble.RandomForestClassifier

class `sklearn.ensemble.` **`RandomForestClassifier`**(*n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None*) [source]

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the `max_samples` parameter if `bootstrap=True` (default), otherwise the whole dataset is used to build each tree.

Pros and cons of bagging

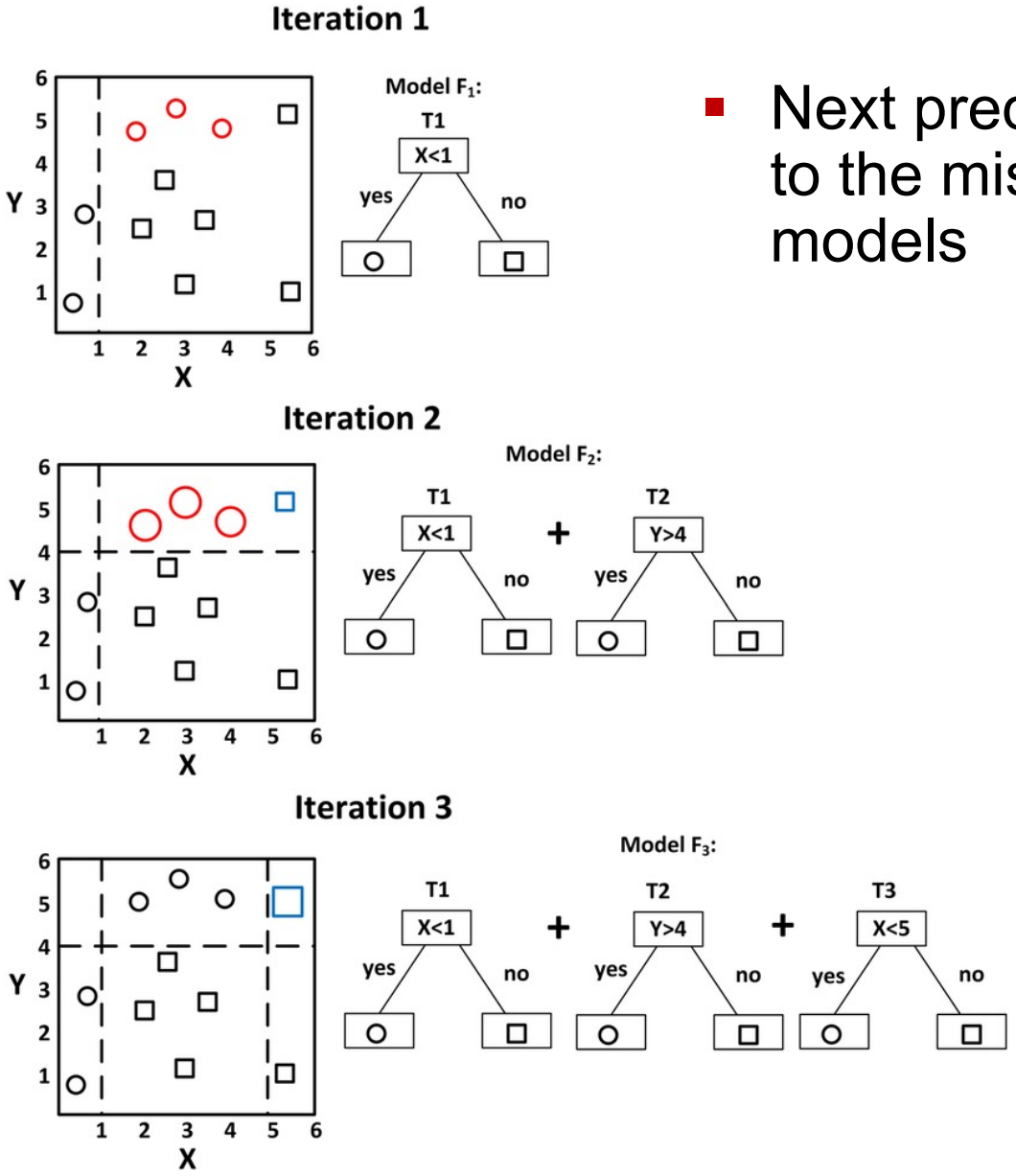
- Bagging is parallelizable
  - Fitting of each model can be done independently

- Bootstrapping and averaging reduce overfitting but does not help underfitted models

- Works well with decision tree, which tends to overfit

Boosting

Boosting with sample weights



- Next predictor gives more weights to the mistakes made by prior models

Adaptive boosting (AdaBoost)

- Ensemble predictor $= \alpha_1 f_1(x) + \alpha_2 f_2(x) + \cdots + \alpha_n f_n(x)$
  - Where $f_i(x)$'s are weak classifiers

- Given $\{\alpha_1, f_1(x), \ldots, \alpha_{n-1}, f_{n-1}(x)\}$ how do we find $\{\alpha_n, f_n(x)\}$?
  - Classification problem with $y_i \in \{+1, -1\}$
  - Define $C_{n-1}(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) + \cdots + \alpha_{n-1} f_{n-1}(x)$
    - Current model
  - Exponential loss $E = \sum_i e^{-y_i C_{n-1}(x_i)}$
    - High when the signs of y_i and $C_{n-1}(x_i)$ differ

- $f_n(x)$ minimizes $\sum_{y_i \neq f_n(x_i)} e^{-y_i C_{n-1}(x_i)}$
- $\hat{\alpha}_n = \frac{1}{2} \ln \left(\frac{\sum_{y_i = f_n(x_i)} e^{-y_i C_{n-1}(x_i)}}{\sum_{y_i \neq f_n(x_i)} e^{-y_i C_{n-1}(x_i)}} \right)$

AdaBoost in Python

sklearn.ensemble.AdaBoostClassifier

class `sklearn.ensemble.` **`AdaBoostClassifier`**(*base_estimator=None, *, n_estimators=50, learning_rate=1.0, algorithm='SAMME.R', random_state=None*) [source]

An AdaBoost classifier.

An AdaBoost [1] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

This class implements the algorithm known as AdaBoost-SAMME [2].

- AdaBoost with learning rate
  - $\hat{\alpha}_n = \gamma \frac{1}{2} \ln \left(\frac{\sum_{y_i = f_n(x_i)} e^{-y_i C_{n-1}(x_i)}}{\sum_{y_i \neq f_n(x_i)} e^{-y_i C_{n-1}(x_i)}} \right)$
  - Learning rate controls how much to trust the next classifier
- Stop after a certain number of models are made

Boosting with gradient

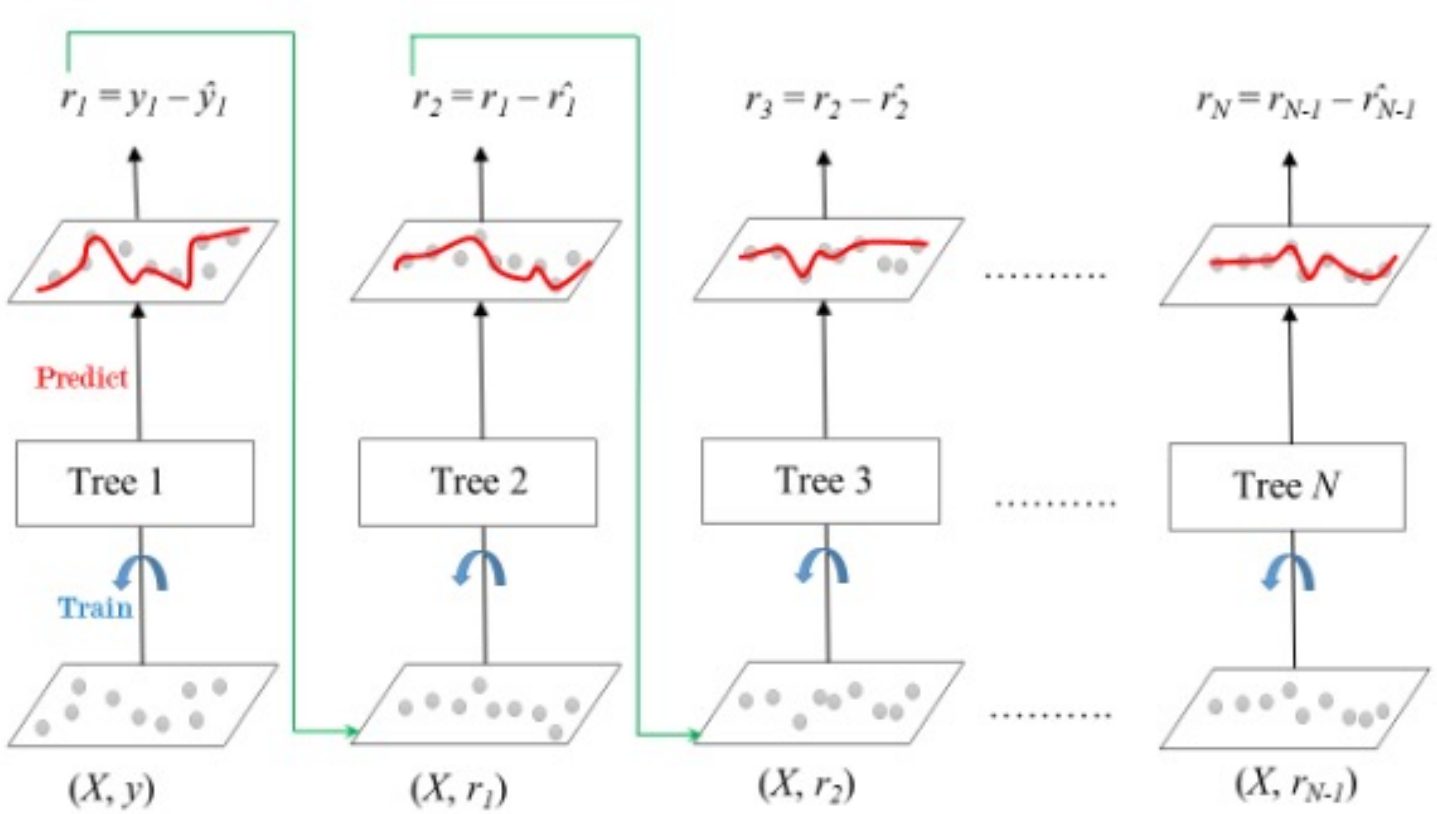


Image from geeksforgeeks.org

- Next predictor fits the residual, or regression error, made by the prior models

Gradient boosting in Python

sklearn.ensemble.GradientBoostingClassifier

class `sklearn.ensemble.GradientBoostingClassifier`(*, *loss='deviance', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, min_impurity_split=None, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0*) [source]

Use decision tree by default

Gradient Boosting for classification.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes_` regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

Read more in the User Guide.

Parameters:

loss : *{'deviance', 'exponential'}, default='deviance'*
The loss function to be optimized. 'deviance' refers to deviance (= logistic regression) for classification with probabilistic outputs. For loss 'exponential' gradient boosting recovers the AdaBoost algorithm.

learning_rate : *float, default=0.1*
Learning rate shrinks the contribution of each tree by `learning_rate`. There is a trade-off between learning_rate and n_estimators.

n_estimators : *int, default=100*
The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

subsample : *float, default=1.0*
The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. `subsample` interacts with the parameter `n_estimators`. Choosing `subsample < 1.0` leads to a reduction of variance and an increase in bias.

criterion : *{'friedman_mse', 'mse', 'mae'}, default='friedman_mse'*
The function to measure the quality of a split. Supported criteria are 'friedman_mse' for the mean squared error with improvement score by Friedman, 'mse' for mean squared error, and 'mae' for the mean absolute error. The default value of 'friedman_mse' is generally the best as it can provide a better approximation in

Early stopping

sklearn.ensemble.GradientBoostingClassifier

class `sklearn.ensemble.GradientBoostingClassifier`(*, *loss='deviance', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, min_impurity_split=None, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0*) [source]

validation_fraction : *float, default=0.1*

The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if `n_iter_no_change` is set to an integer.

New in version 0.20.

n_iter_no_change : *int, default=None*

`n_iter_no_change` is used to decide if early stopping will be used to terminate training when validation score is not improving. By default it is set to None to disable early stopping. If set to a number, it will set aside `validation_fraction` size of the training data as validation and terminate training when validation score is not improving in all of the previous `n_iter_no_change` numbers of iterations. The split is stratified.

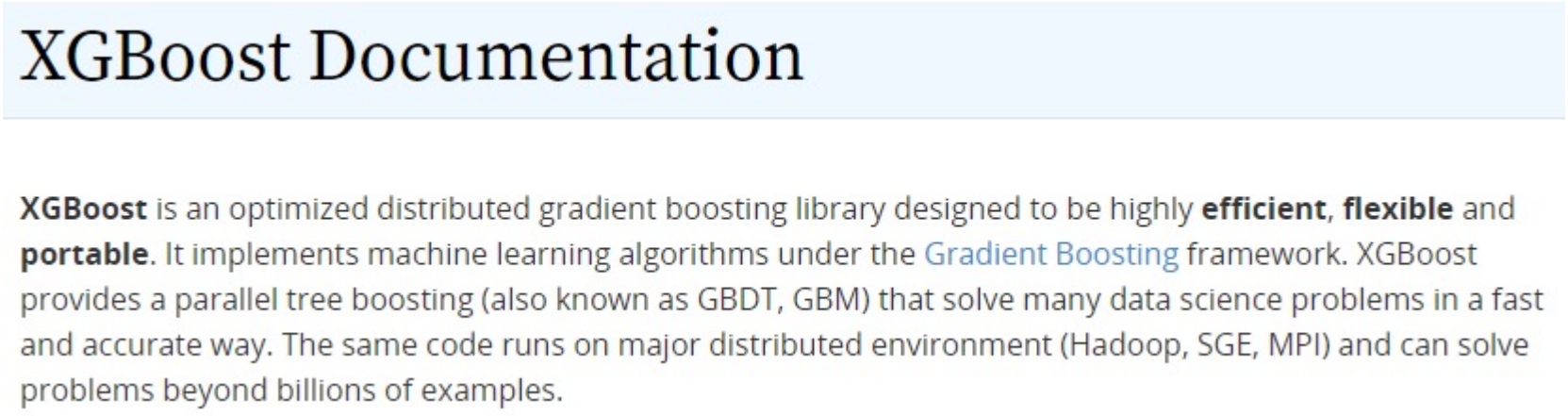
New in version 0.20.

- Stop training if performance on **validation set** does not improve for a certain number of steps
  - The model is overfitting to the training data

Pros and cons of boosting

- Build additional predictors iteratively
  - Cannot be parallelized
- Address the errors of current predictors directly
  - Help with underfitting
- Sensitive to misclassified outliers
  - Multiple models will try to fit them
- Will overfit to the training set
  - Prevented by using small learning rate and early stopping on validation
  - Validation performance will overestimate true performance

Next week: XGBoost



- Optimized implementation of gradient boosting
- Can handle missing data
- Can handle monotonicity constraint
  - "Prediction must increase if feature X increases"
- pip install xgboost

Any question?