Lecture 11: Ensemble Methods

Olexandr Isayev

Department of Chemistry, CMU

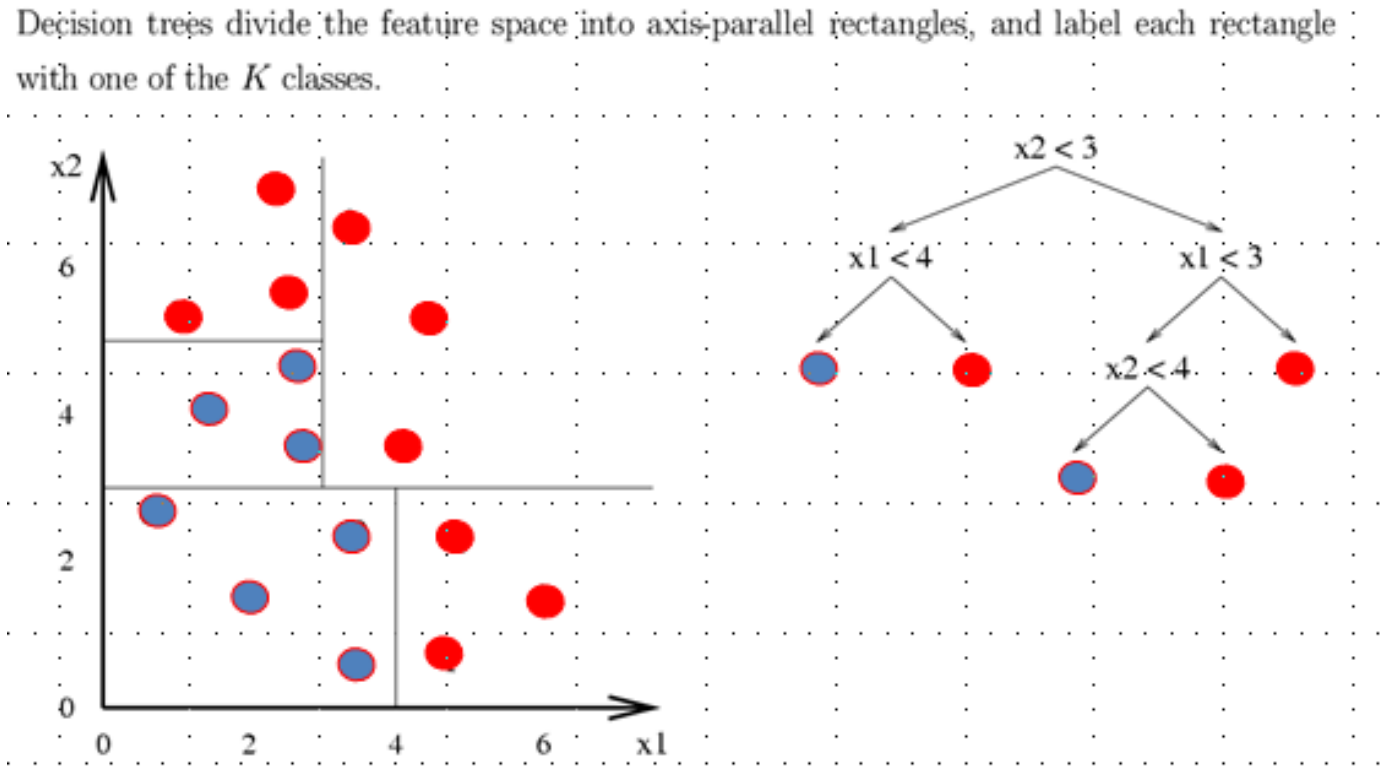
olexandr@cmu.edu

Class project Intro

- We will go over Class project intro on Thursday

Decision tree

- A flow-chart-like tree structure
- Internal node denotes a test on an attribute
- Branch represents an outcome of the test
- Leaf nodes represent class labels or class distribution



Decision Tree Induction Techniques

Decision tree induction is a top-down, recursive and divide-and-conquer approach.

The procedure is to choose an attribute and split it into from a larger training set into smaller training sets.

Different algorithms have been proposed to take a good control over

1. Choosing the best attribute to be splitted, and

2. Splitting criteria

Several algorithms have been proposed for the above tasks. In this lecture, we shall limit our discussions into three important of them

- **ID3**
- **C 4.5**
- **CART**

Notes on Decision Tree algorithms

1. **Optimal Decision Tree:** Finding an optimal decision tree is an NP-complete problem. Hence, decision tree induction algorithms employ a heuristic based approach to search for the best in a large search space. Majority of the algorithms follow a greedy, top-down recursive divide-and-conquer strategy to build decision trees.

2. **Missing data and noise:** Decision tree induction algorithms are quite robust to the data set with missing values and presence of noise. However, proper data pre-processing can be followed to nullify these discrepancies.

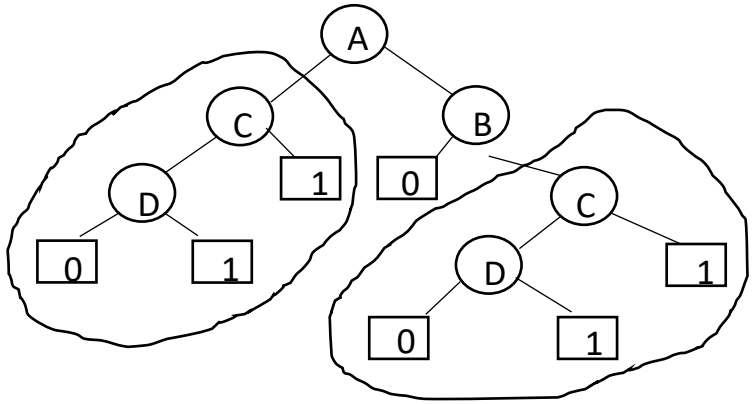
3. **Redundant Attributes:** The presence of redundant attributes does not adversely affect the accuracy of decision trees. It is observed that if an attribute is chosen for splitting, then another attribute which is redundant is unlikely to chosen for splitting.

4. **Computational complexity:** Decision tree induction algorithms are computationally inexpensive, in particular, when the sizes of training sets are large, Moreover, once a decision tree is known, classifying a test record is extremely fast, with a worst-case time complexity of O(d), where d is the maximum depth of the tree.

Notes on Decision Tree algorithms

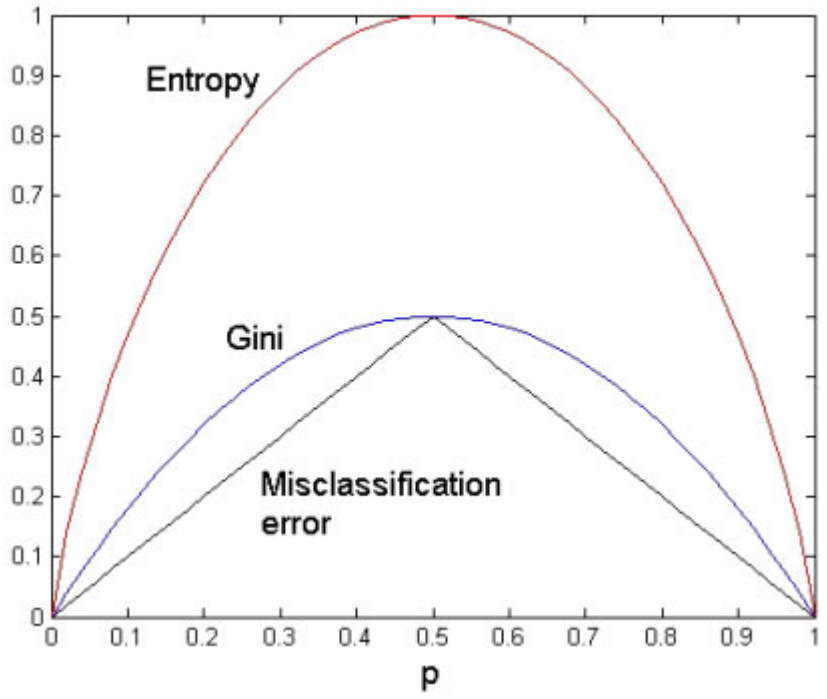
5. **Data Fragmentation Problem:** Since the decision tree algorithms employ a top-down, recursive partitioning approach, the number of tuples becomes smaller as we traverse down the tree. At a time, the number of tuples may be too small to make a decision about the class representation, such a problem is known as the data fragmentation. To deal with this problem, further splitting can be stopped when the number of records falls below a certain threshold.

6. **Tree Pruning:** A sub-tree can replicate two or more times in a decision tree (see figure below). This makes a decision tree unambiguous to classify a test record. To avoid such a sub-tree replication problem, all sub-trees except one can be pruned from the tree.



Notes on Decision Tree algorithms

7. **Decision tree equivalence:** The different splitting criteria followed in different decision tree induction algorithms have little effect on the performance of the algorithms. This is because the different heuristic measures (such as information gain (α), Gini index (γ) and Gain ratio (β) are quite consistent with each other); also see the figure below.



Summary of Decision Tree Algorithms

We have learned the building of a decision tree given a training data.

The decision tree is then used to classify a test data.

For a given training data *D*, the important task is to build the decision tree so that:

All test data can be classified accurately

The tree is balanced and with as minimum depth as possible, thus the classification can be done at a faster rate.

In order to build a decision tree, several algorithms have been proposed. These algorithms differ from the chosen splitting criteria, so that they satisfy the above mentioned objectives as well as the decision tree can be induced with minimum time complexity. We have studied three decision tree induction algorithms namely ID3, CART and C4.5.

Overfitting with Decision Trees

- You can perfectly fit to any training data
- Zero bias, high variance

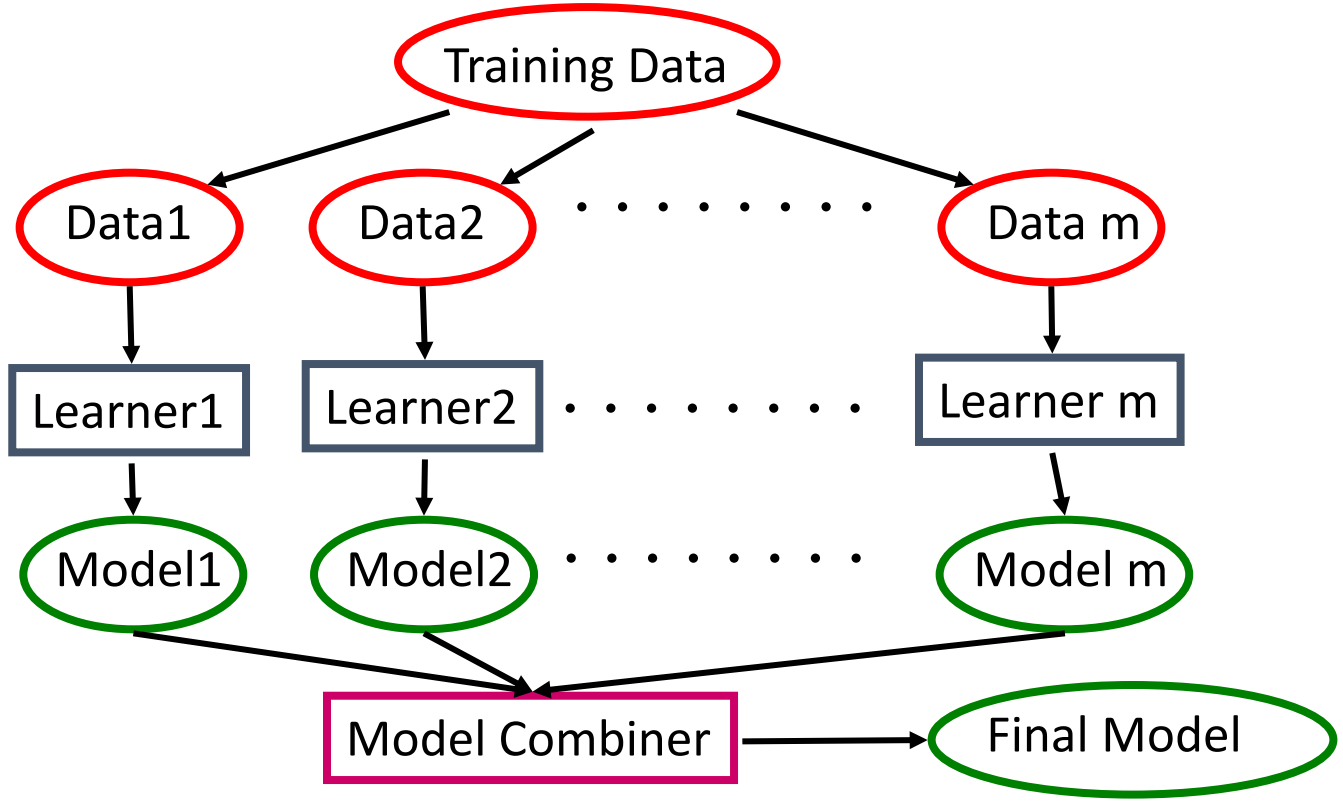
- Two approaches:
- 1. Stop growing the tree when further splitting the data does not yield an improvement
- 2. Grow a full tree, then prune the tree, by eliminating nodes.

Ensemble Learning

- So far – learning methods that learn a single hypothesis, chosen form a hypothesis space that is used to make predictions.
- Ensemble learning → select a collection (ensemble) of hypotheses and combine their predictions.
- Example 1 - generate 100 different decision trees from the same or different training set and have them vote on the best classification for a new example.
- Key motivation: reduce the error rate. Hope is that it will become much more unlikely that the ensemble will misclassify an example.

Learning Ensembles

- Learn multiple alternative definitions of a concept using different training data or different learning algorithms.
- Combine decisions of multiple definitions, e.g. using weighted voting.



Source: Ray Mooney

Value of Ensembles

- “No Free Lunch” Theorem
  - No single algorithm wins all the time!

- When combing multiple independent and diverse decisions each of which is at least more accurate than random guessing, random errors cancel each other out, correct decisions are reinforced.

- Examples: Human ensembles are demonstrably better
  - How many jelly beans in the jar?: Individual estimates vs. group average.
  - Who Wants to be a Millionaire: Audience vote.

Source: Ray Mooney

Bagging

- Bagging or *bootstrap aggregation* a technique for reducing the variance of an estimated prediction function.

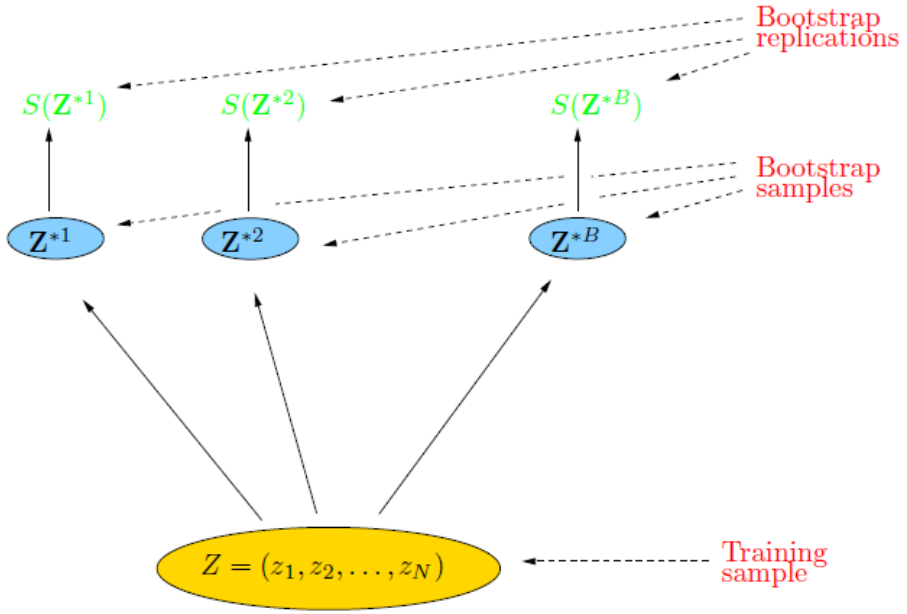
- For classification, a *committee* of trees each cast a vote for the predicted class.

- For regression: average/mean of predicted numeric values

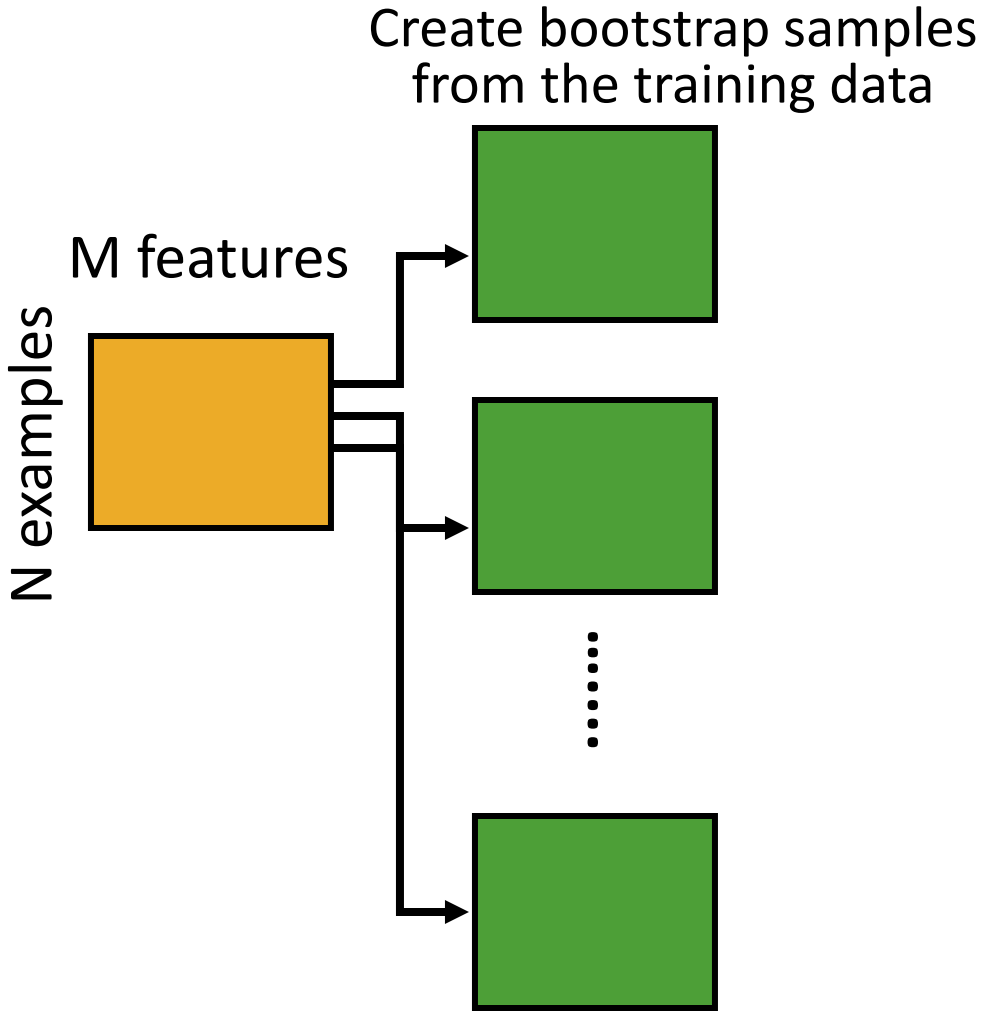
Bootstrap

A randomly draw datasets *with replacement* from the training data, all resample have the same size.

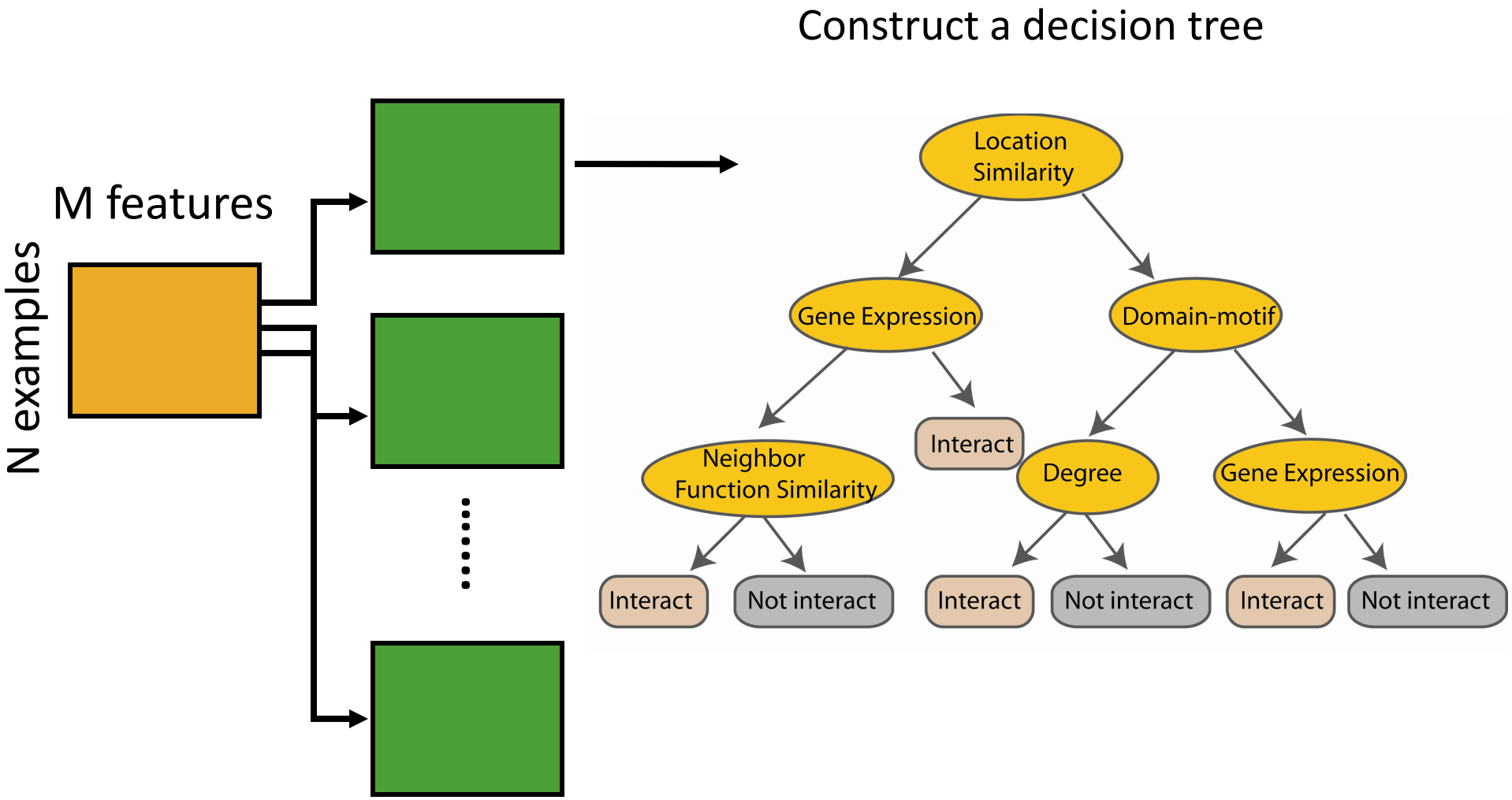
All resamples are generated independently by resampling with replacement. In each bootstrap sample about 30% of the observations are set aside for later model validation. These observations are grouped as the out of bag sample (OOBS)



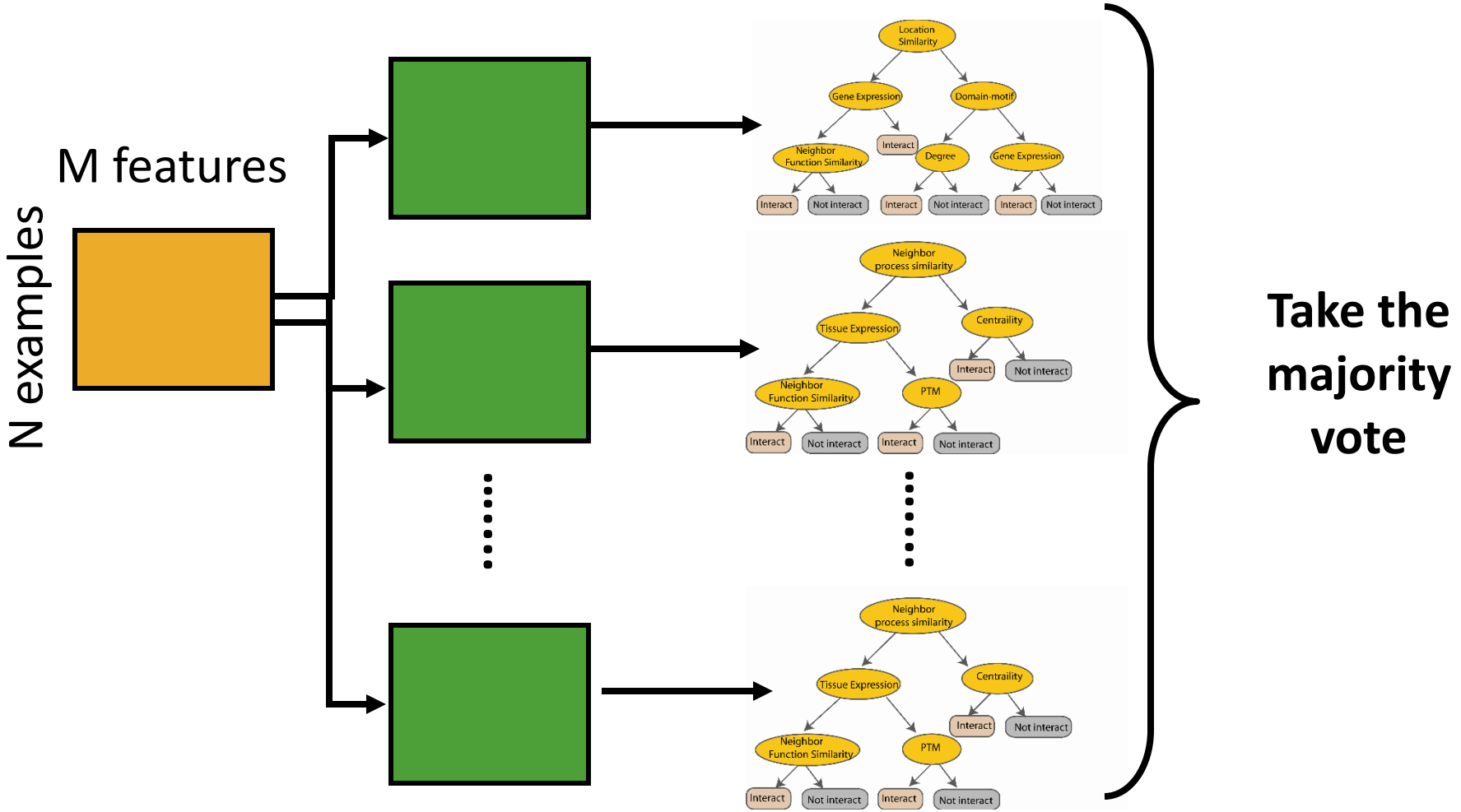
Bagging Classifier



Bagging Classifier



Bagging Classifier



Bagging

$\mathbf{Z} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$

Z^{*b} where = 1,.., B..

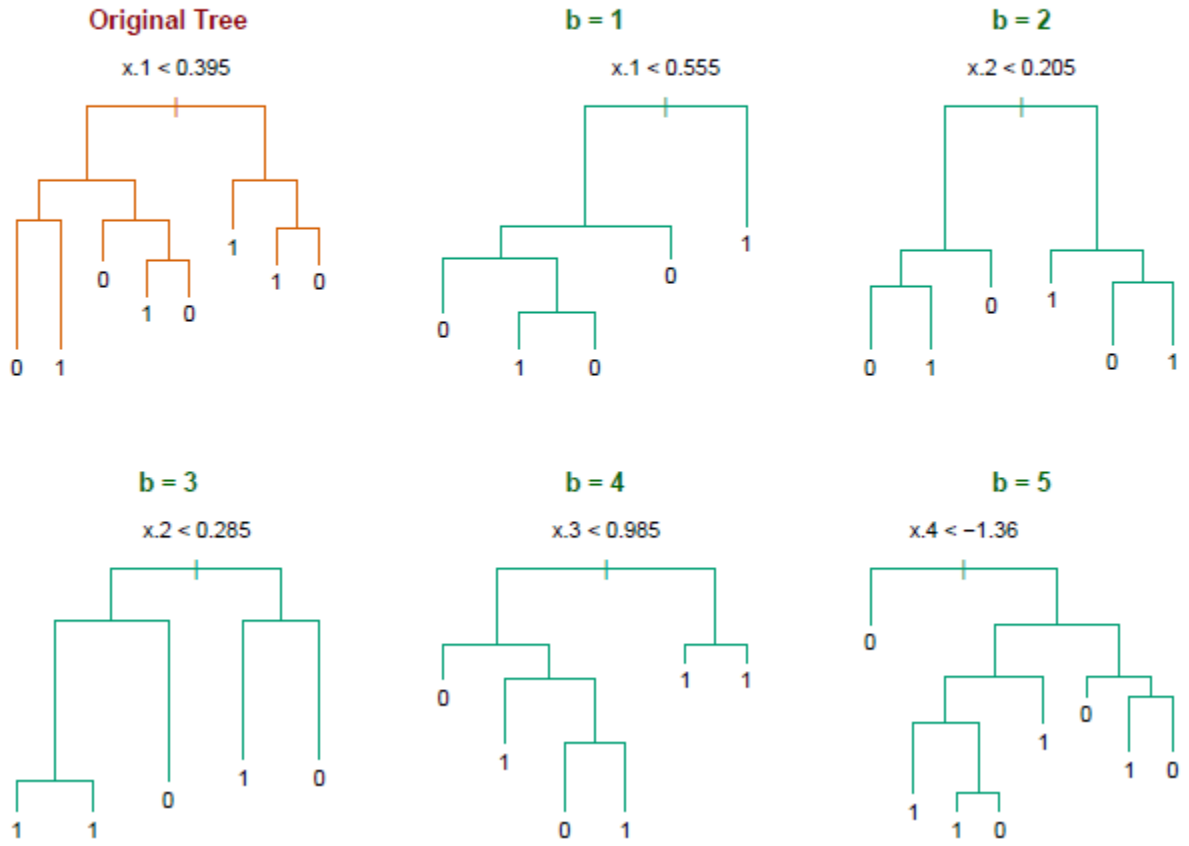
The prediction at input x when bootstrap sample *b* is used for training

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

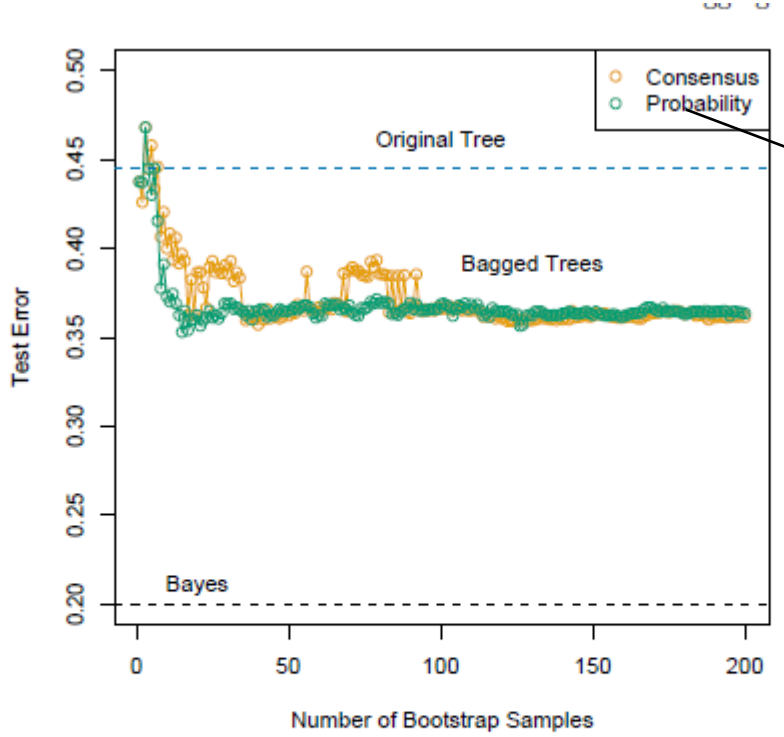
http://www-stat.stanford.edu/~hastie/Papers/ESLII.pdf (Chapter 8.7)

Bagging

Notice the bootstrap trees are different than the original tree



Bagging



Treat the voting Proportions as probabilities

FIGURE 8.10. *Error curves for the bagging example of Figure 8.9. Shown is the test error of the original tree and bagged trees as a function of the number of bootstrap samples. The orange points correspond to the consensus vote, while the green points average the probabilities.*

bagging helps under squared-error loss, in short because averaging reduces

Hastie

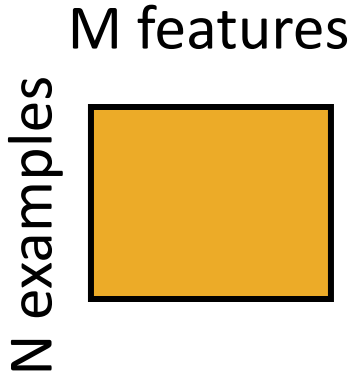
http://www-stat.stanford.edu/~hastie/Papers/ESLII.pdf Example 8.7.1

Random forest classifier

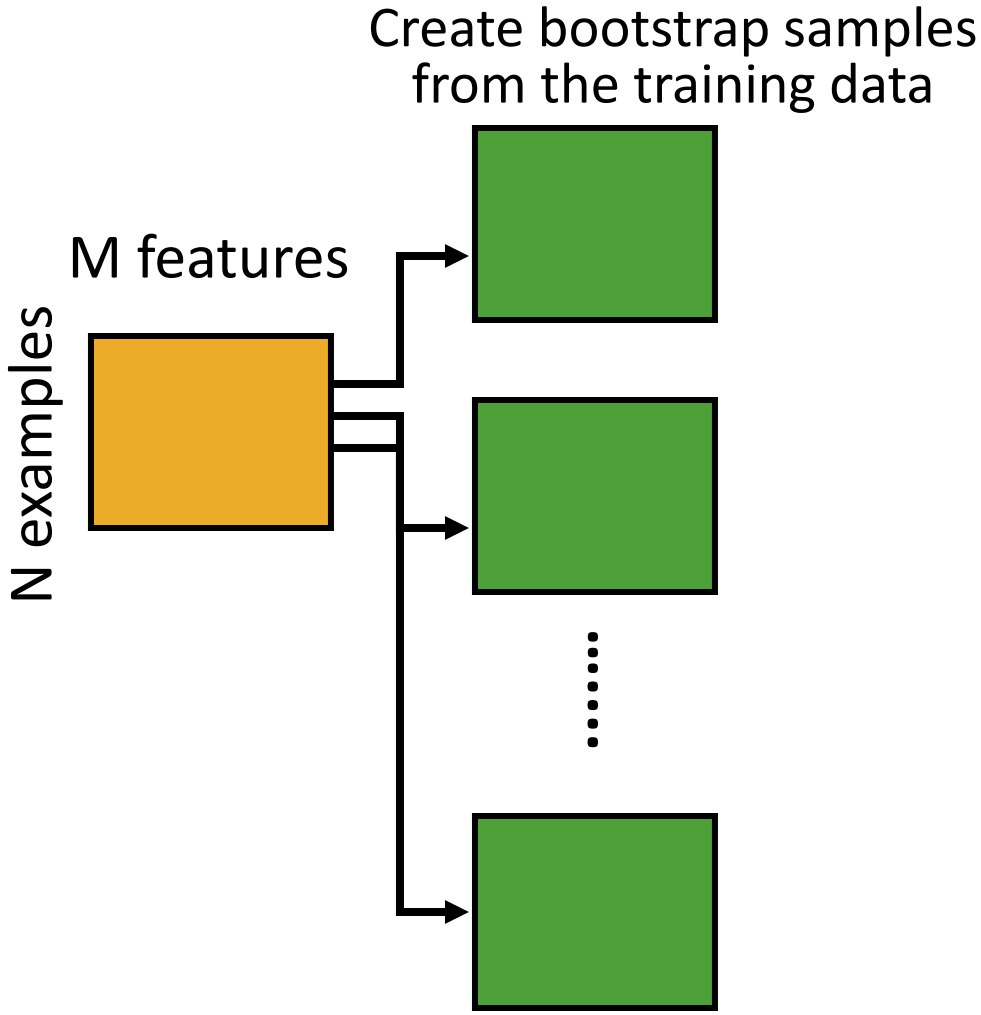
Random forest classifier, an extension to bagging which uses *de-correlated* trees.

Random Forest Classifier

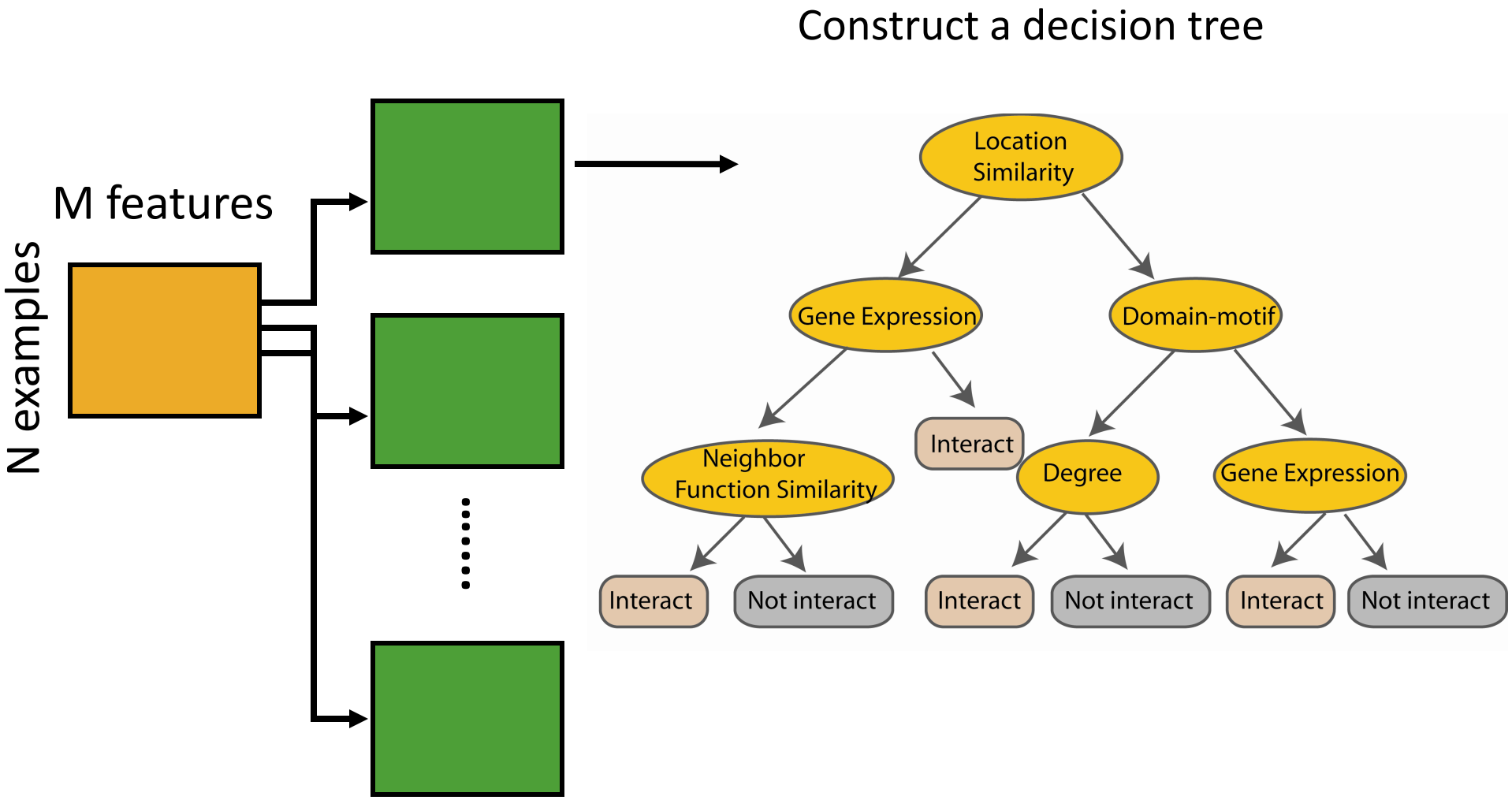
Training Data



Random Forest Classifier

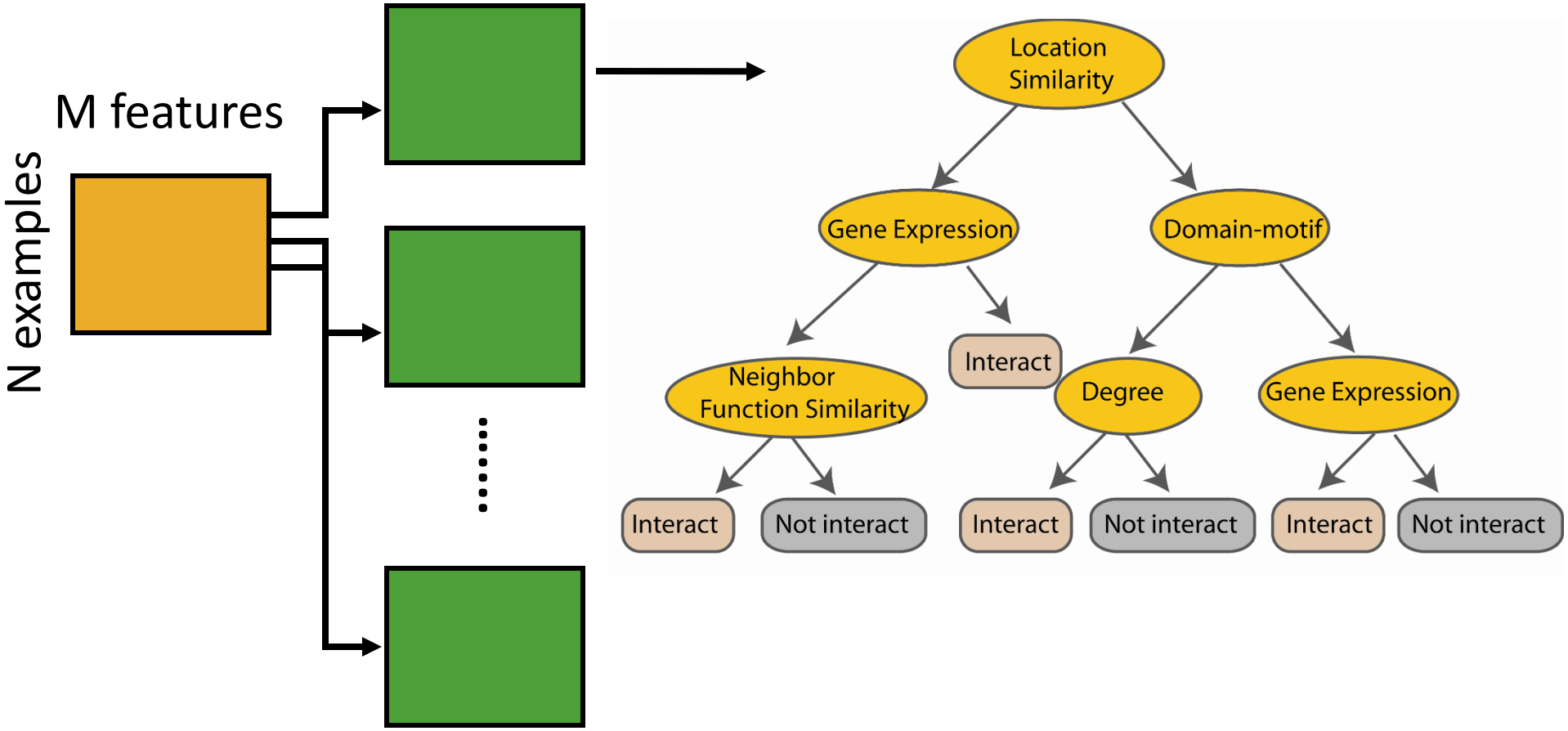


Random Forest Classifier

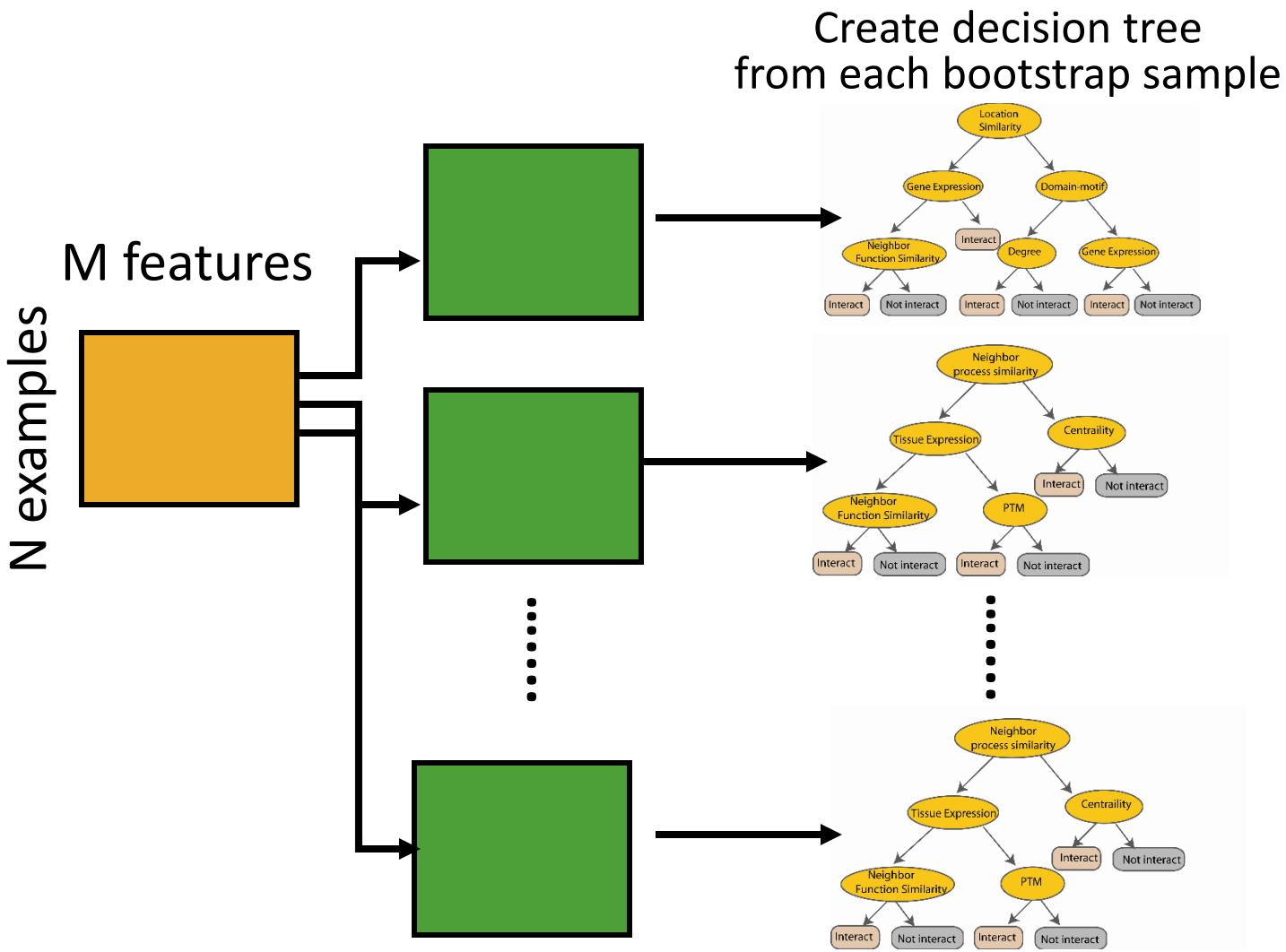


Random Forest Classifier

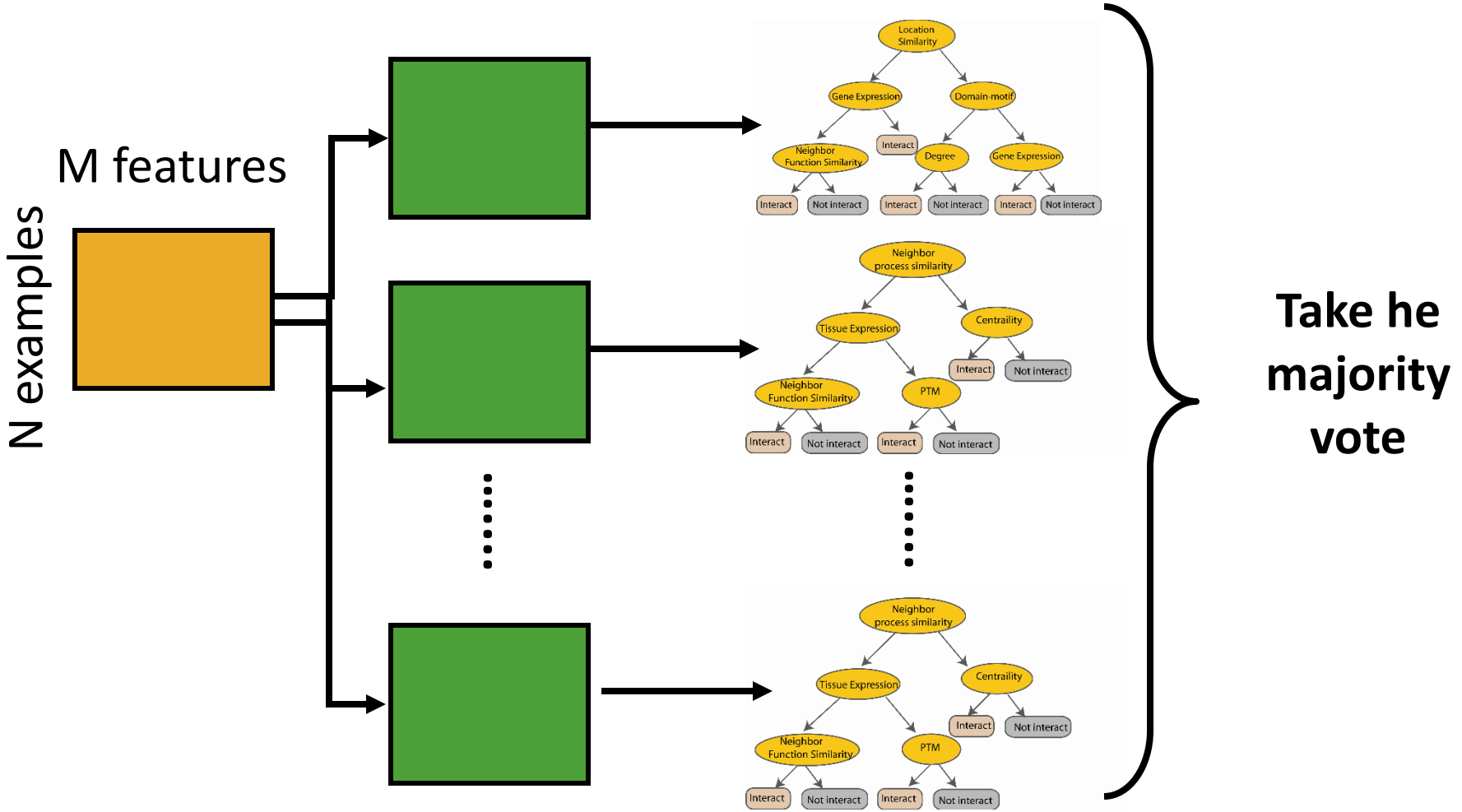
At each node in choosing the split feature
choose only among $m<M$ features



Random Forest Classifier



Random Forest Classifier



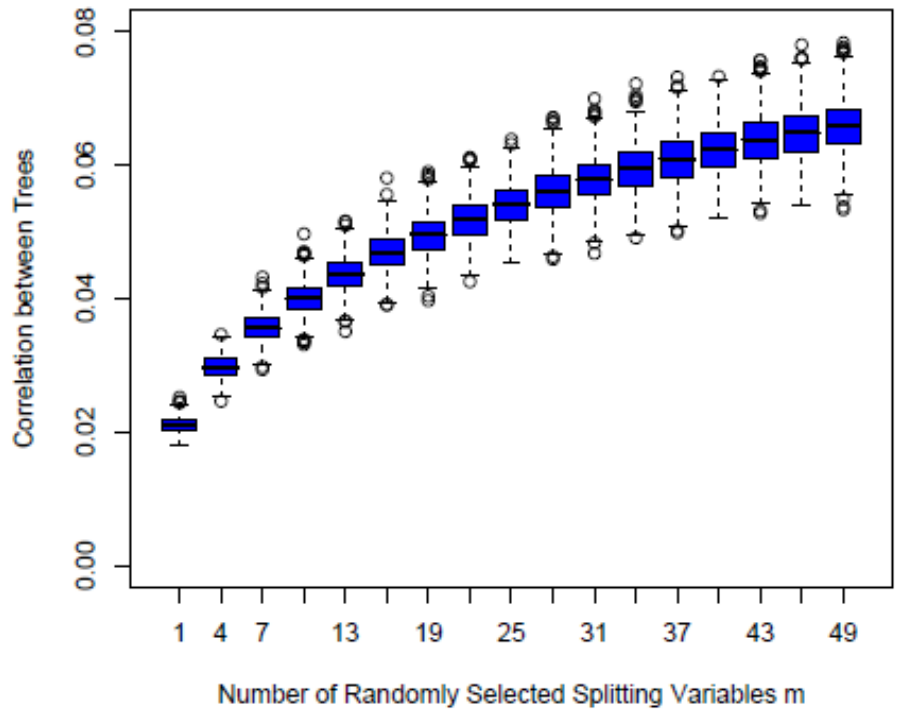


FIGURE 15.9. *Correlations between pairs of trees drawn by a random-forest regression algorithm, as a function of m. The boxplots represent the correlations at 600 randomly chosen prediction points x.*

Out of bag (OOB) score

Out of bag (OOB) score is a way of validating the Random forest model

In an ideal case, about 36.8 % of the total training data forms the OOB sample. This can be shown as follows.
If there are N rows in the training data set. Then, the probability of not picking a row in a random draw is

$$\frac{N-1}{N}$$

Using sampling-with-replacement the probability of not picking N rows in random draws is

$$\left(\frac{N-1}{N}\right)^N$$

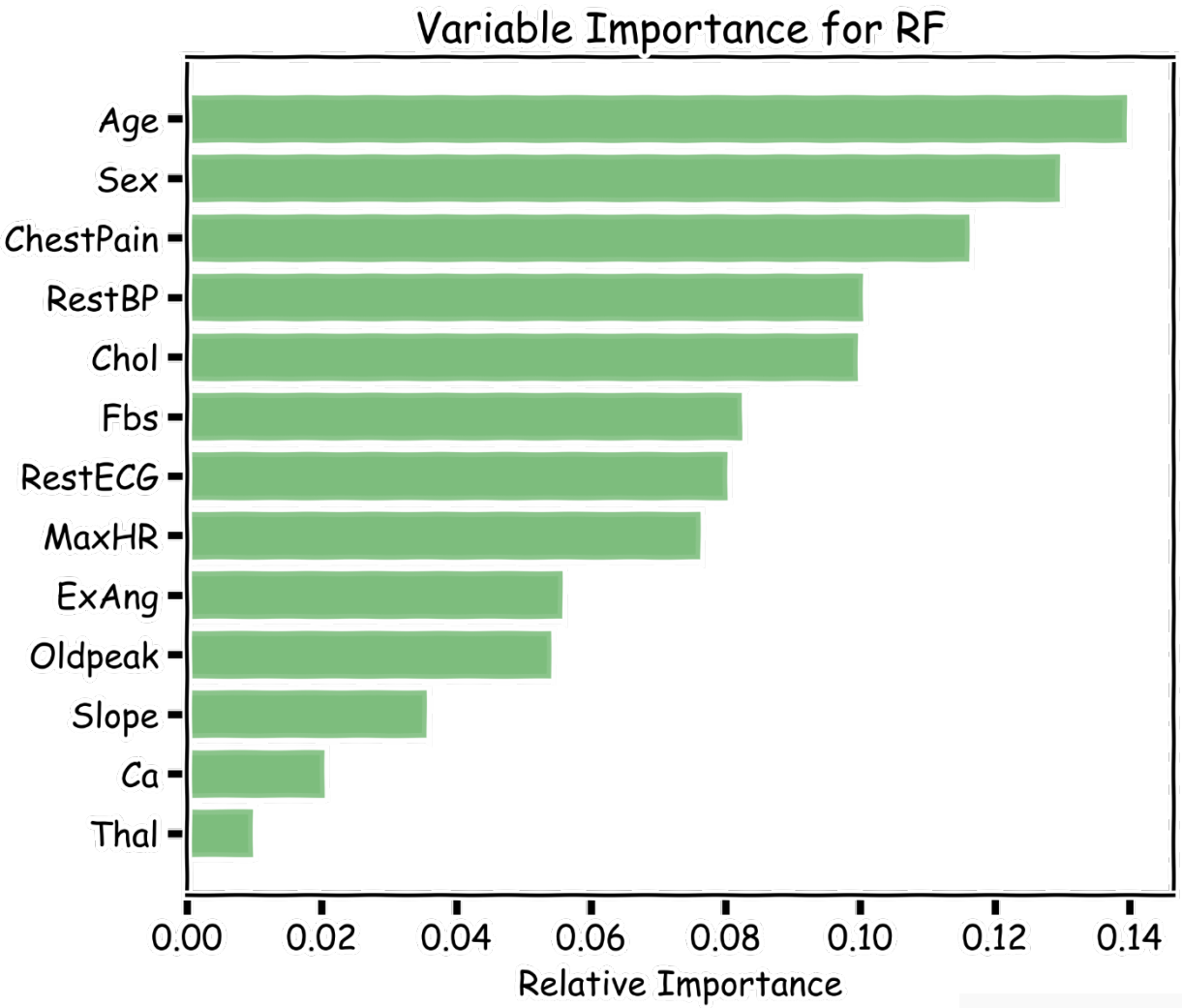
which in the limit of large N becomes equal to

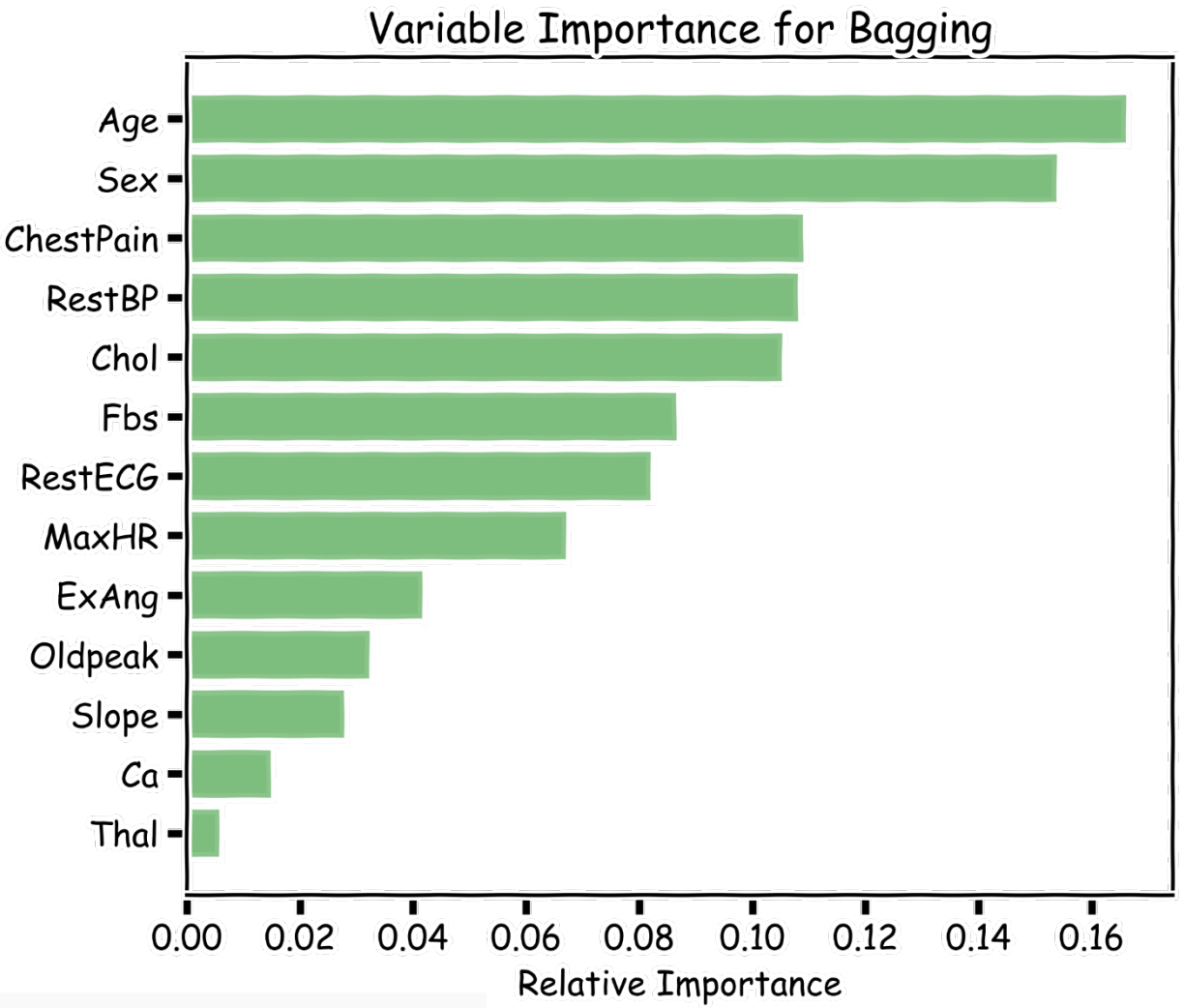
$$\lim_{N\to\infty}\left(1-\frac{1}{N}\right)^N = e^{-1} = 0.368$$

Variable Importance for RF

Calculate the total amount that the residual sum of squares (RSS, for regression) or Gini index (for classification) is decreased due to splits over a given predictor, averaged over all B trees.

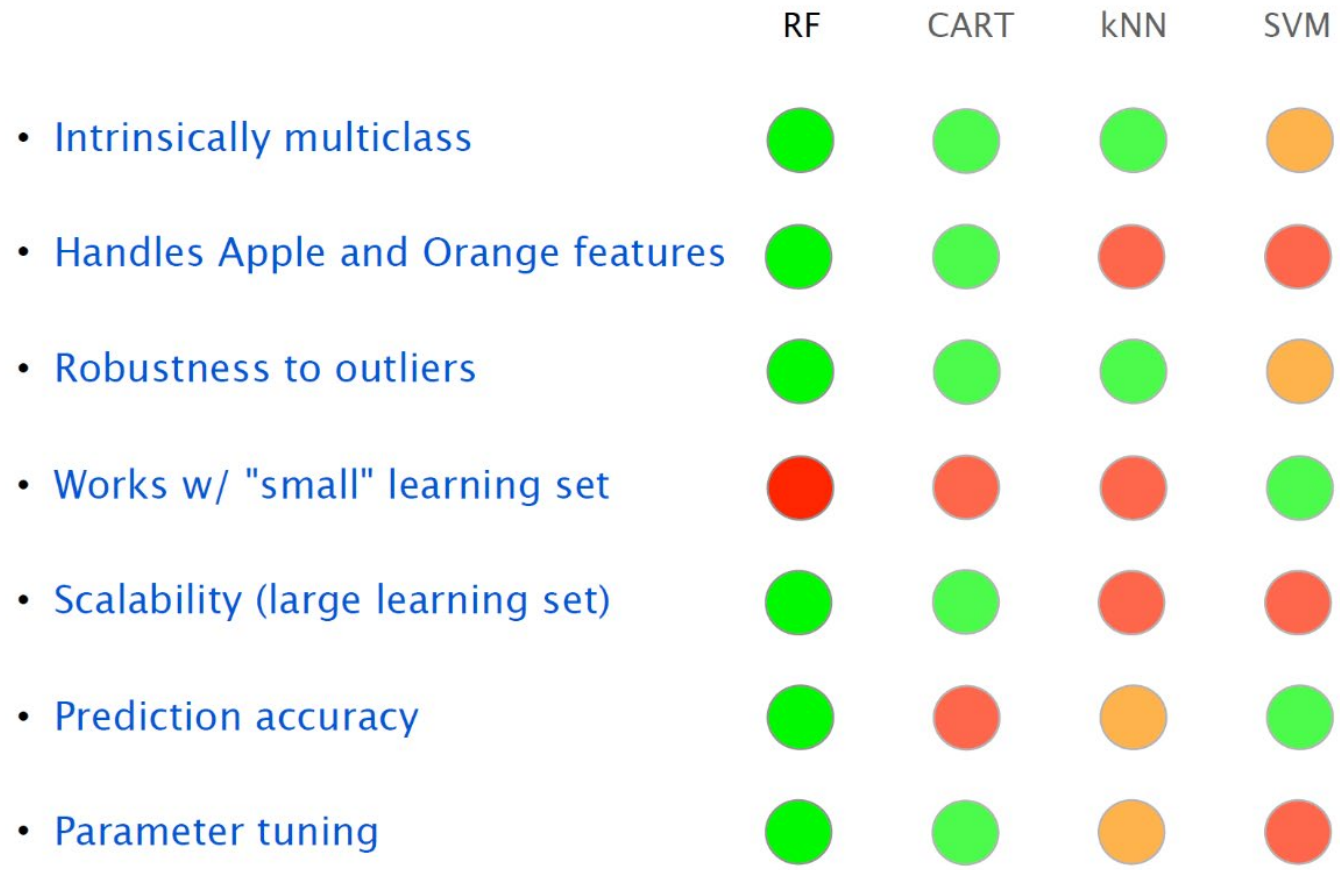
Variable Importance for RF



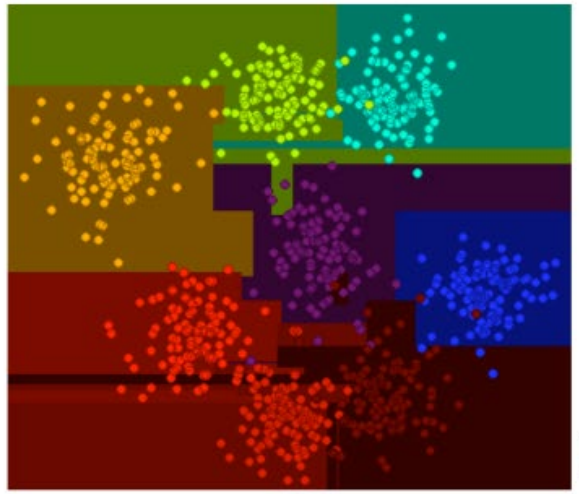


100 trees, max_depth=10

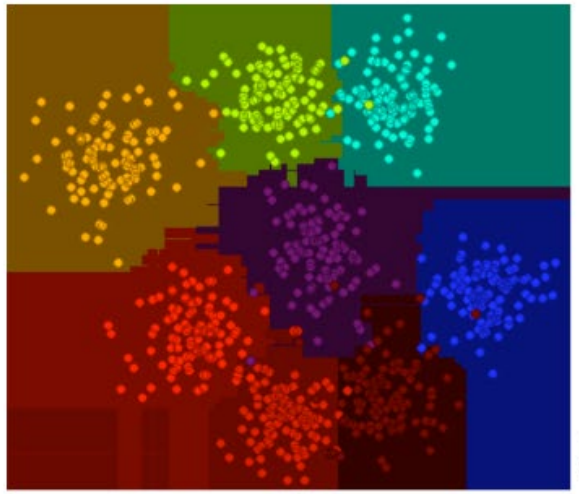
Random Forest Properties



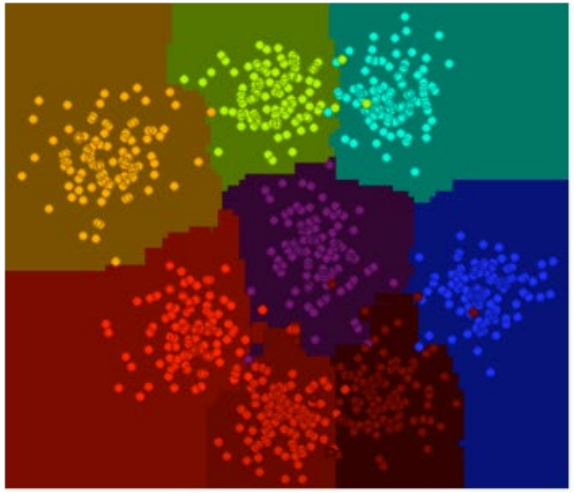
Smoother decision boundaries



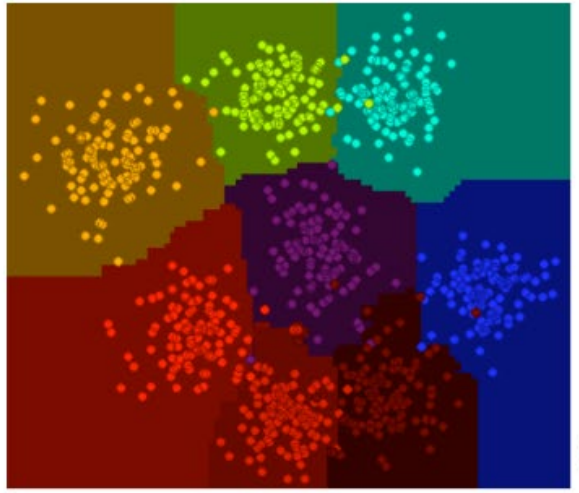
1 rCART



10 rCARTs



100 rCARTs



500 rCARTs

RF Limitations

- Can’t extrapolate
- Fundamentally discrete algorithm. Not very good for smooth & continuous outputs
- Not good for Oblique/curved frontiers
  - Staircase effect
  - Many pieces of hyperplanes

Bagging

- The bootstrap method works best when each model yielded from resampling is independent and thus these models are truly diverse.
- If all researchers in the team think in the same way, then no one is thinking.
- If the bootstrap replicates are not diverse, the result might not be as accurate as expected.
- If there is a systematic bias and the classifiers are bad, bagging these bad classifiers can make the end model worse.

Boosting

- A sequential and adaptive method
- The previous model informs the next.
- Initially, all observations are assigned the same weight. If the model fails to classify certain observations correctly, then these cases are assigned a heavier weight so that they are more likely to be selected in the next model.
- In the subsequent steps, each model is constantly revised in an attempt to classify those observations successfully.
- While bagging requires many independent models for convergence, boosting reaches a final solution after a few iterations.

AdaBoost vs Gradient Boosting Trees

AdaBoost	GradientBoost
Both AdaBoost and Gradient Boost use a base weak learner and they try to boost the performance of a weak learner by iteratively shifting the focus towards problematic observations that were difficult to predict. At the end, a strong learner is formed by addition (or weighted addition) of the weak learners.	
In AdaBoost, shift is done by up-weighting observations that were misclassified before.	Gradient boost identifies difficult observations by large residuals computed in the previous iterations.
In AdaBoost "shortcomings" are identified by high-weight data points.	In Gradientboost "shortcomings" are identified by gradients.
Exponential loss of AdaBoost gives more weights for those samples fitted worse.	Gradient boost further dissect error components to bring in more explanation.
AdaBoost is considered as a special case of Gradient boost in terms of loss function, in which exponential losses.	Concepts of gradients are more general in nature.

Gradient Boosting

The key intuition behind boosting is that one can take an ensemble of simple models $\{T_h\}_{h \in H}$ and additively combine them into a single, more complex model.

Each model T_h might be a poor fit for the data, but a linear combination of the ensemble

$$T = \sum_h \lambda_h T_H$$

- can be expressive/flexible.

Gradient Boosting: the algorithm

Gradient boosting is a method for iteratively building a complex regression model T by adding simple models. Each new simple model added to the ensemble compensates for the weaknesses of the current ensemble.

1. Fit a simple model $T^{(0)}$ on the training data

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

Set $T \leftarrow T^{(0)}$. Compute the residuals $\{r_1, \ldots, r_N\}$ for T.

2. Fit a simple model, $T^{(1)}$, to the current ***residuals***, i.e. train using

$$\{(x_1, r_1), \ldots, (x_N, r_N)\}$$

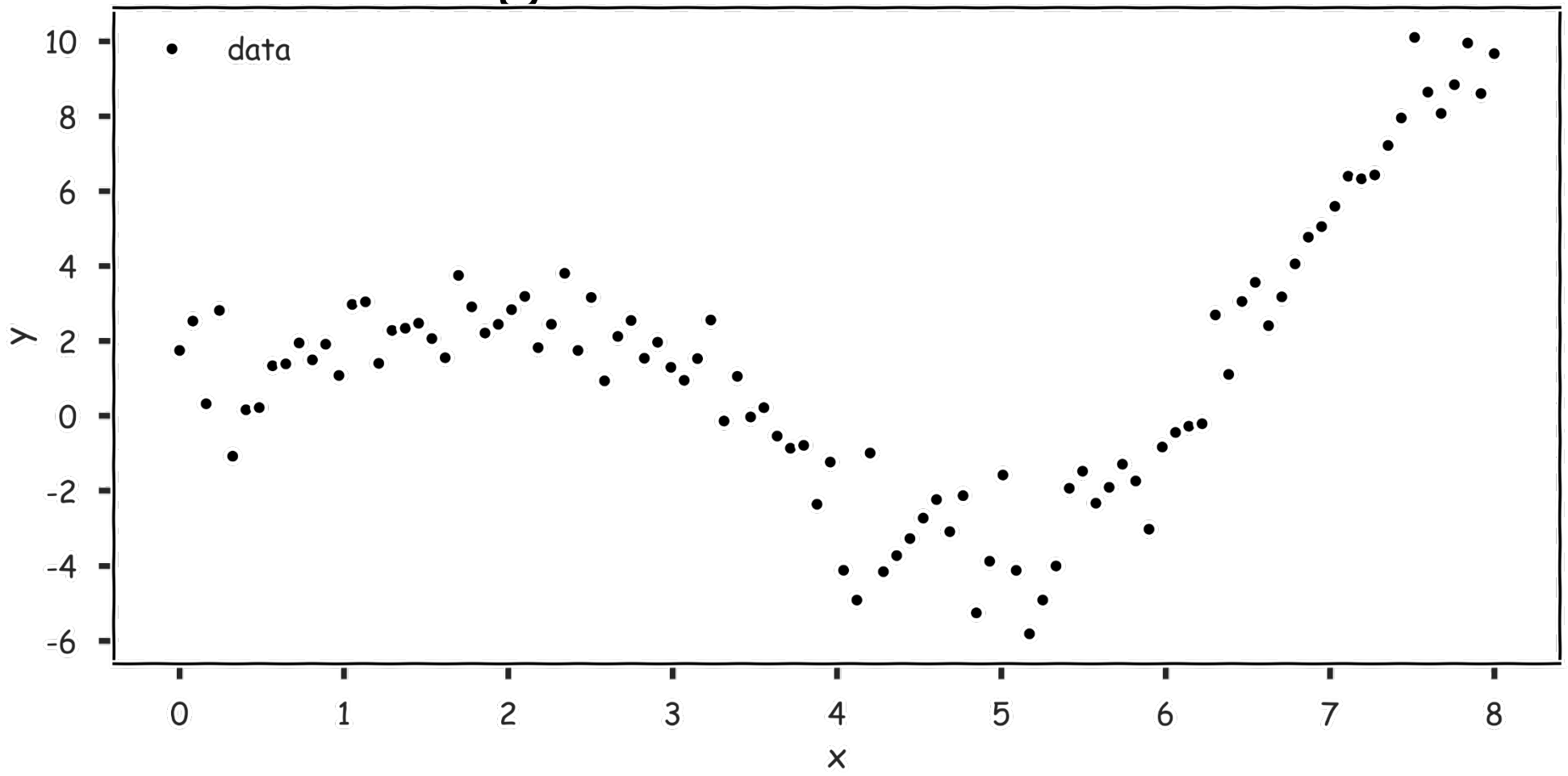
3. Set $T \leftarrow T + \lambda T^{(1)}$

4. Compute residuals, set $r_n \leftarrow r_n - \lambda T^i(x_n),\ n = 1, \ldots, N$

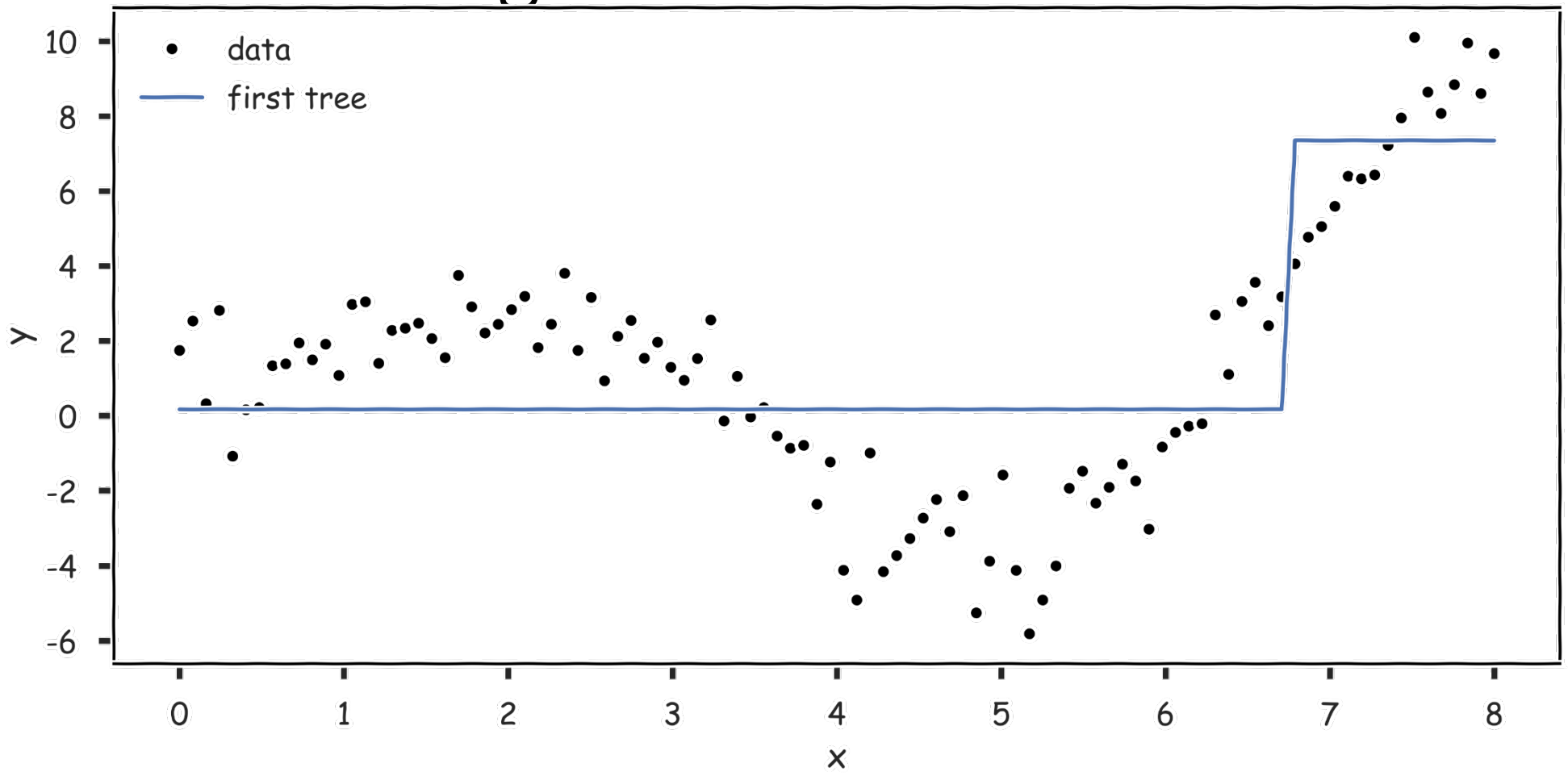
5. Repeat steps 2-4 until ***stopping*** condition met.

where λ is a constant called the ***learning rate***.

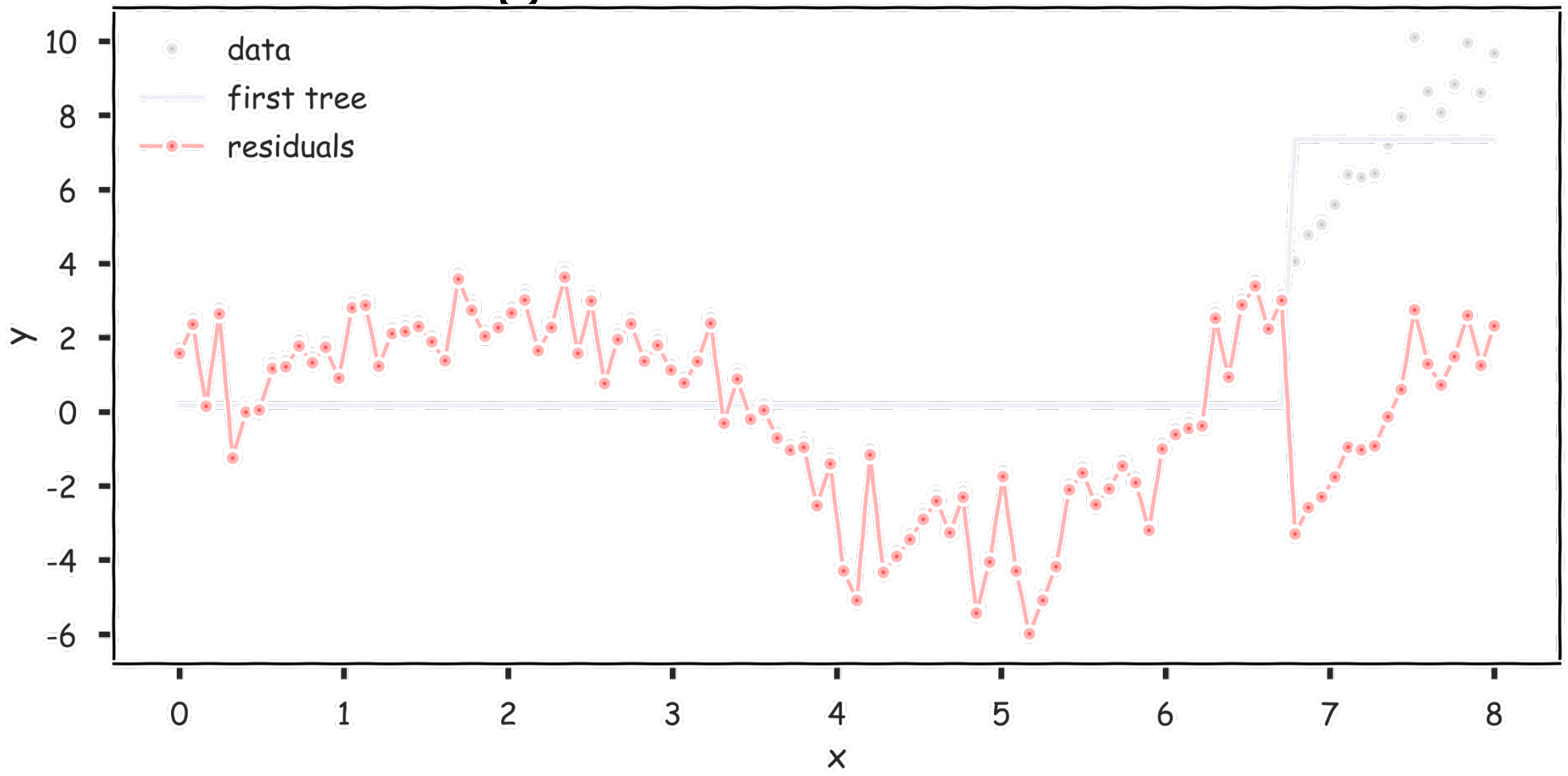
Gradient Boosting: illustration



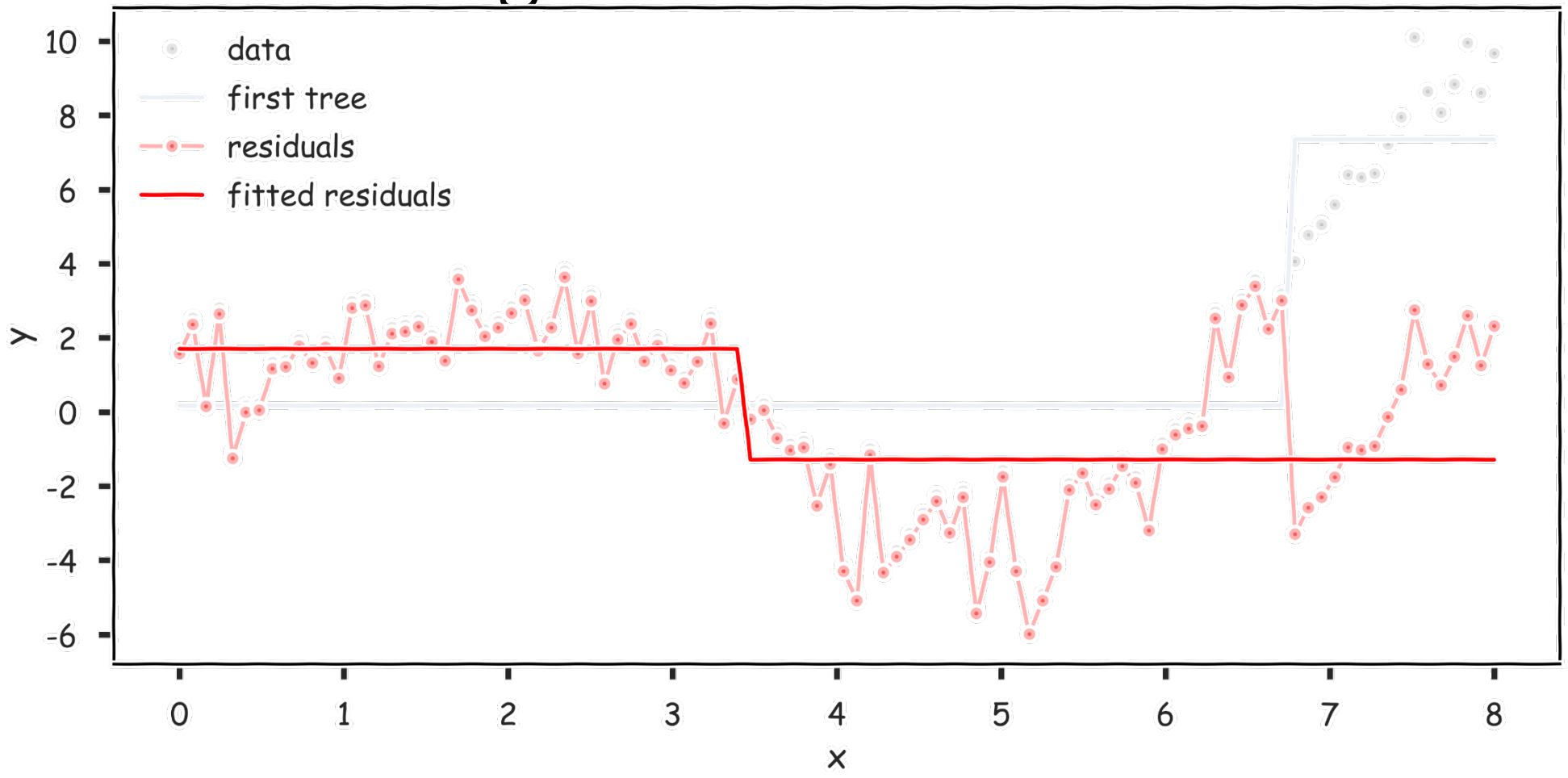
Gradient Boosting: illustration



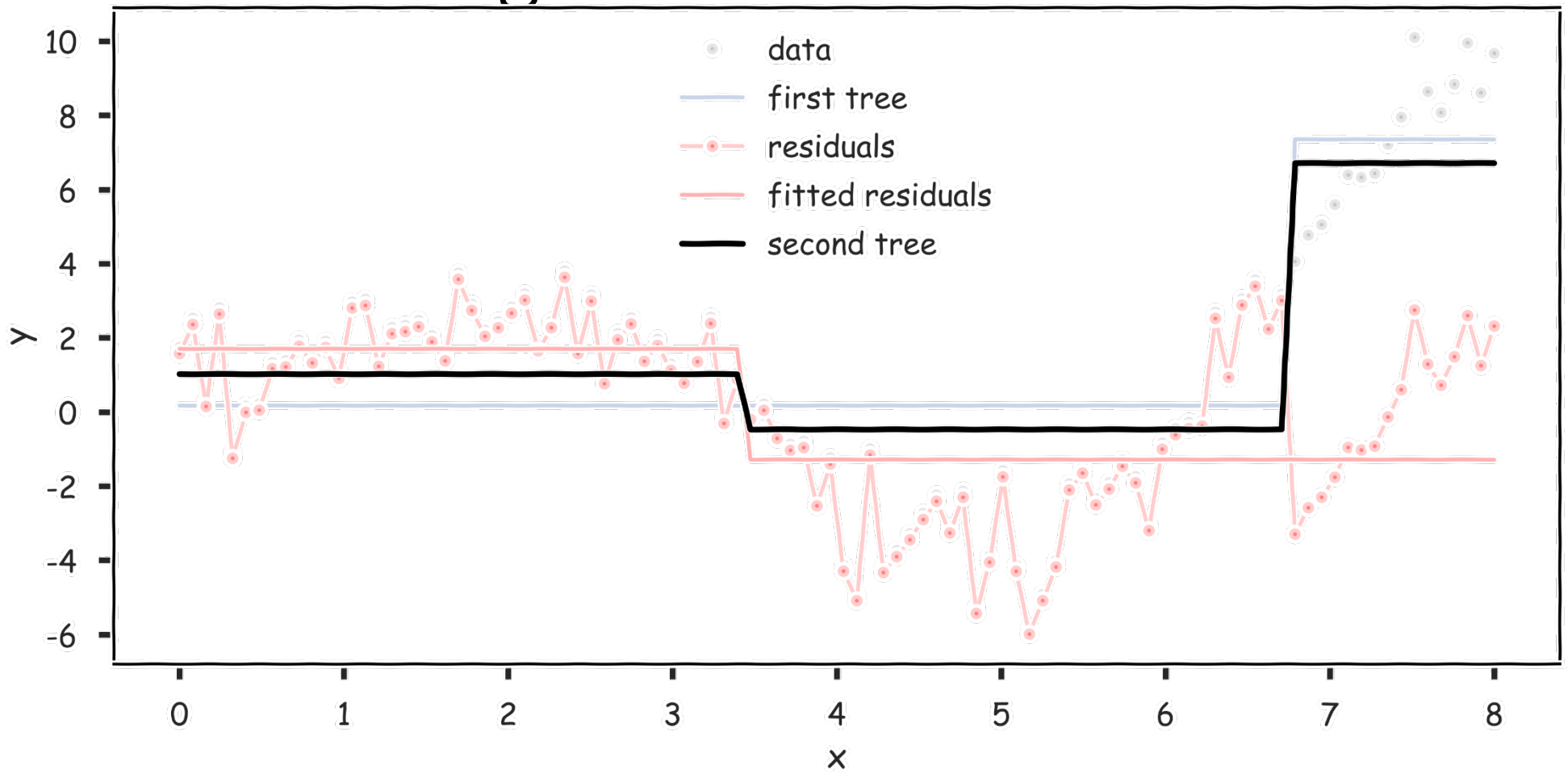
Gradient Boosting: illustration



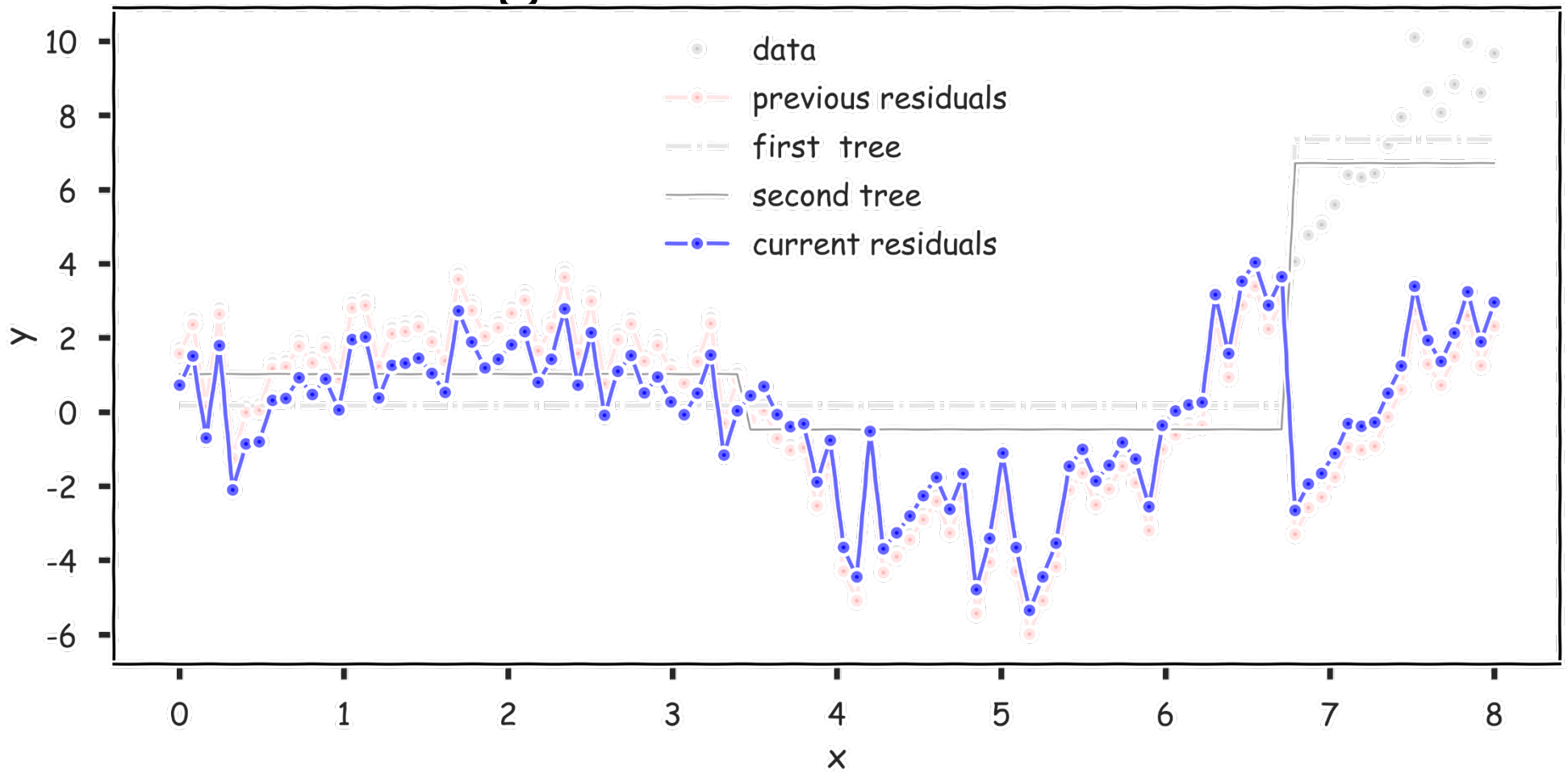
Gradient Boosting: illustration



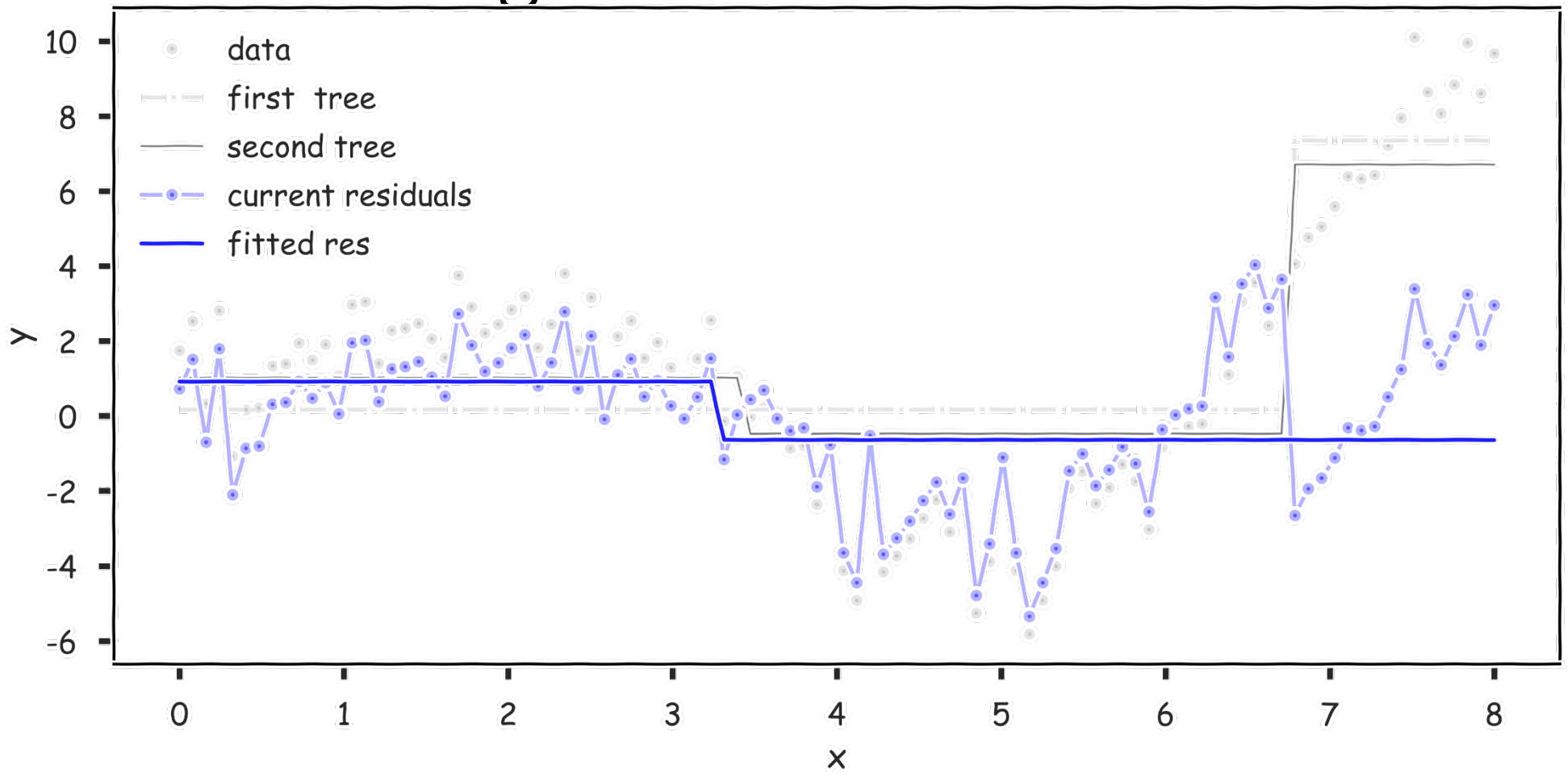
Gradient Boosting: illustration



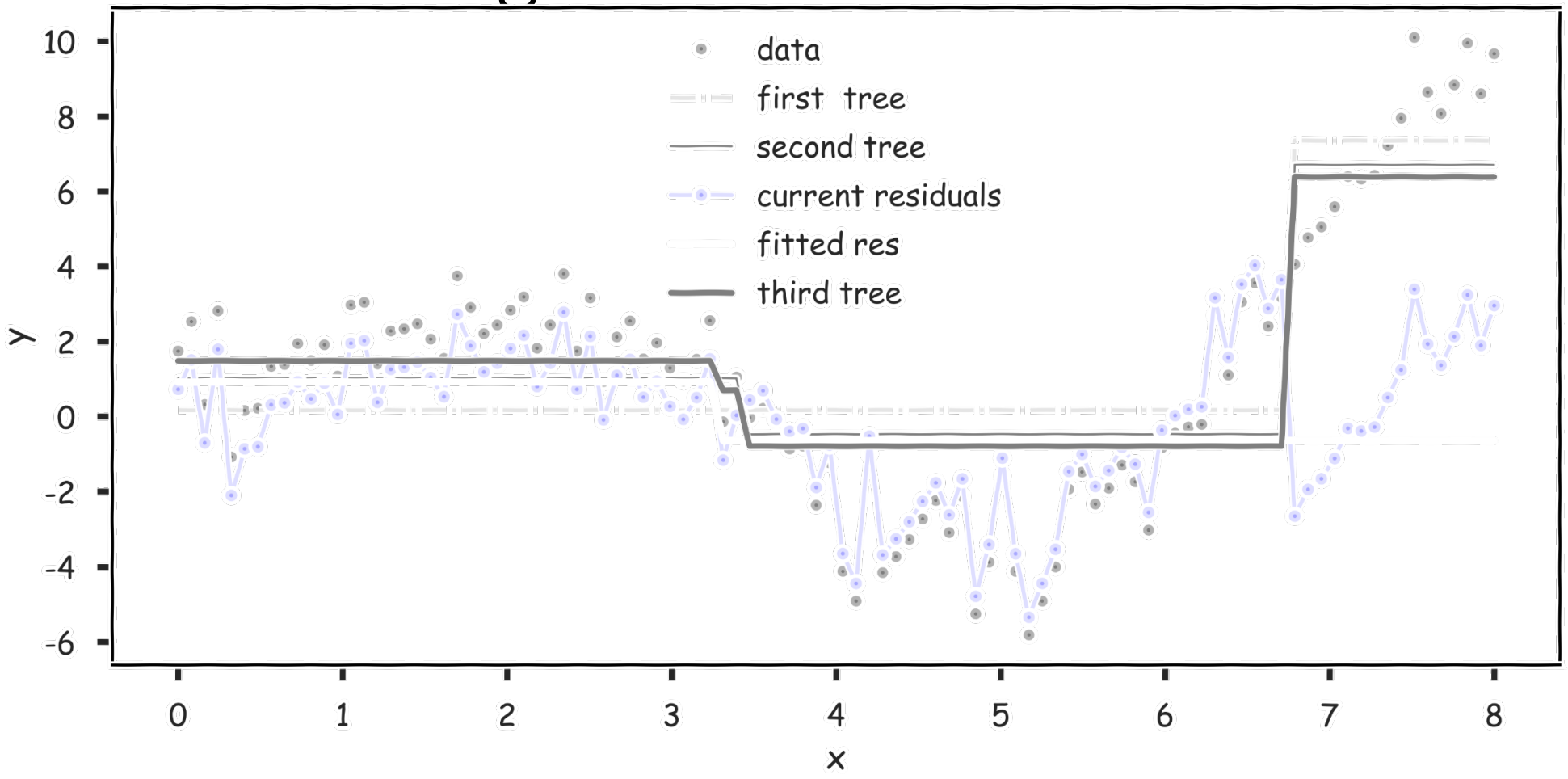
Gradient Boosting: illustration



Gradient Boosting: illustration



Gradient Boosting: illustration



Why Does Gradient Boosting Work?

Intuitively, each simple model $T^{(i)}$ we add to our ensemble model T, models the errors of T.

Thus, with each addition of $T^{(i)}$, the residual is reduced

$$r_n - \lambda T^{(i)}(x_n)$$

Note that gradient boosting has a tuning parameter, λ.

If we want to easily reason about how to choose λ and investigate the effect of λ on the model T, we need a bit more mathematical formalism.

In particular, how can we effectively descend through this optimization via an iterative algorithm?

We need to formulate gradient boosting as a type of ***gradient descent***.

A Brief Sketch of Gradient Descent

In optimization, when we wish to minimize a function, called the ***objective function***, over a set of variables, we compute the partial derivatives of this function with respect to the variables.

If the partial derivatives are sufficiently simple, one can analytically find a common root - i.e. a point at which all the partial derivatives vanish; this is called a ***stationary point.***

If the objective function has the property of being ***convex***, then the stationary point is precisely the min.

A Brief Sketch of Gradient Descent

In practice, our objective functions are complicated and analytically find the stationary point is intractable.

Instead, we use an iterative method called ***gradient descent***:

Initialize the variables at any value:

$$x = [x_1, \ldots, x_J]$$

Take the gradient of the objective function at the current variable values:

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_J}(x)\right]$$

Adjust the variables values by some negative multiple of the gradient:

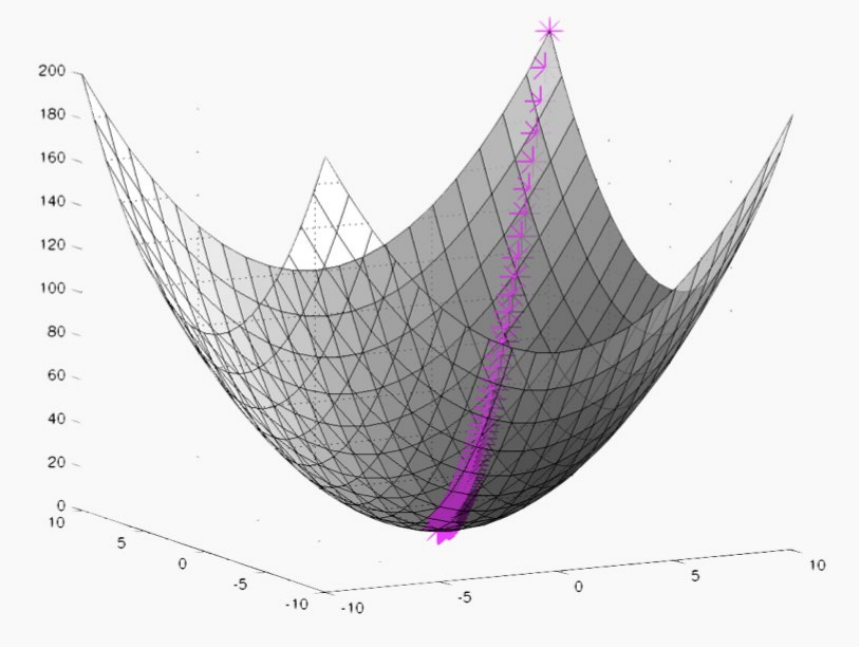
$$x \leftarrow x - \lambda \nabla f(x)$$

The factor λ is often called the learning rate.

Why Does Gradient Descent Work?

Subtracting a λ multiple of the gradient from x, moves x in the ***opposite*** direction of the gradient (hence towards the steepest decline) by a step of size λ.

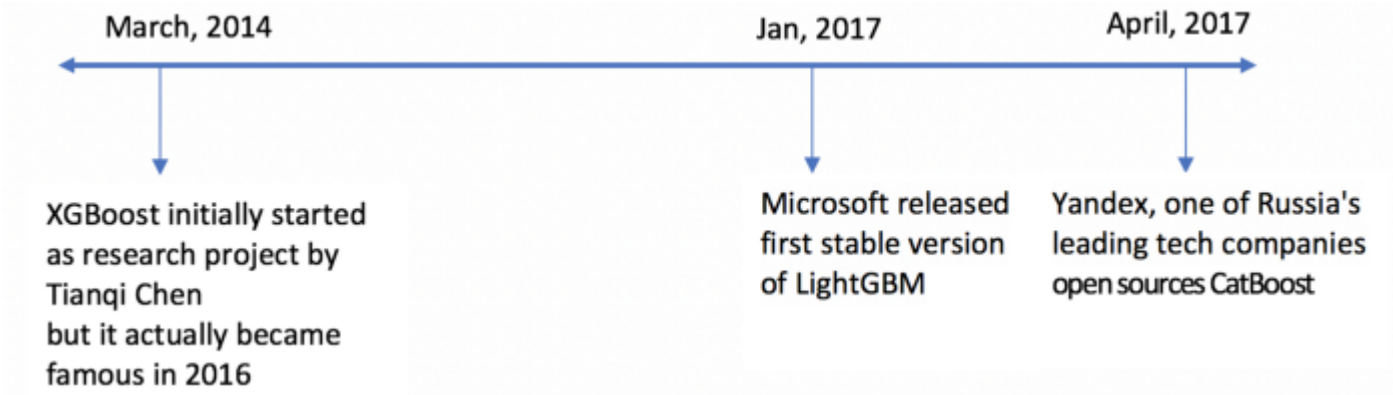
If *f* is convex, and we keep taking steps descending on the graph of *f*, we will eventually reach the minimum.



Comparing bagging and boosting

	Bagging	Boosting
Sequent	Two-step	Sequential
Partitioning data into subsets	Random	Give misclassified cases a heavier weight
Sampling method	Sampling with replacement	Sampling without replacement
Relations between models	Parallel ensemble: Each model is independent	Previous models inform subsequent models
Goal to achieve	Minimize variance	Minimize bias, improve predictive power
Method to combine models	Weighted average or majority vote	Majority vote
Requirement of computing resources	Highly computing intensive	Less computing intensive

Development of Boosting Machines



Gradient Boosting Machine (GBM)
GBM combines predictions from multiple decision trees, and all the weak learners are decision trees. The key idea with this algorithm is that every node of those trees takes a different subset of features to select the best split. As it's a Boosting algorithm, each new tree learns from the errors made in the previous ones.

Light Gradient Boosting Machine (LightGBM)
LightGBM can handle huge amounts of data. It's one of the fastest algorithms for both training and prediction. It generalizes well, meaning that it can be used to solve similar problems. It scales well to large numbers of cores and has an open-source code so you can use it in your projects for free.

Categorical Boosting (CatBoost)
This particular set of Gradient Boosting variants has specific abilities to handle categorical variables and data in general. The CatBoost object can handle categorical variables or numeric variables, as well as datasets with mixed types. That's not all. It can also use unlabeled examples and explore the effect of kernel size on speed during training.

e**X**treme **G**radient **Boost**(XGBoost)

Some important features of XGBoost are:

•***Parallelization:*** The model is implemented to train with multiple CPU cores and GPU.

•***Regularization:*** XGBoost includes different regularization penalties to avoid overfitting. Penalty regularizations produce successful training so the model can generalize adequately.

•***Non-linearity:*** XGBoost can detect and learn from non-linear data patterns.

•***Cross-validation:*** Built-in and comes out-of-the-box.

•***Scalability:*** XGBoost can run distributed thanks to distributed servers and clusters like Hadoop and Spark, so you can process enormous amounts of data. It's also available for many programming languages like C++, JAVA, Python, and Julia.

Advantages

- Gradient Boosting comes with an easy to read and interpret algorithm, making most of its predictions easy to handle.
- Boosting is a resilient and robust method that prevents and cubs over-fitting quite easily
- XGBoost performs very well on medium, small, data with subgroups and structured datasets with not too many features.
- It is a great approach to go for because the large majority of real-world problems involve classification and regression, two tasks where XGBoost is the reigning king.

Disadvantages

- XGBoost does not perform so well on sparse and unstructured data.
- A common thing often forgotten is that Gradient Boosting is very sensitive to outliers since every classifier is forced to fix the errors in the predecessor learners.
- The overall method is hardly scalable. This is because the estimators base their correctness on previous predictors, hence the procedure involves a lot of struggle to streamline.

eta [default=0.3]

- Analogous to learning rate in GBM
- Makes the model more robust by shrinking the weights on each step
- Typical final values to be used: 0.01-0.2

min_child_weight [default=1]

- Defines the minimum sum of weights of all observations required in a child.
- This is similar to **min_child_leaf** in GBM but not exactly. This refers to min "sum of weights" of observations while GBM has min "number of observations".
- Used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
- Too high values can lead to under-fitting hence, it should be tuned using CV.

max_depth [default=6]

- The maximum depth of a tree, same as GBM.
- Used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
- Should be tuned using CV.
- Typical values: 3-10

max_leaf_nodes

- The maximum number of terminal nodes or leaves in a tree.
- Can be defined in place of max_depth. Since binary trees are created, a depth of 'n' would produce a maximum of 2^n leaves.
- If this is defined, GBM will ignore max_depth.

gamma [default=0]

- A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.
- Makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.

max_delta_step [default=0]

- In maximum delta step we allow each tree's weight estimation to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative.
- Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced.
- This is generally not used but you can explore further if you wish.

subsample [default=1]

- Same as the subsample of GBM. Denotes the fraction of observations to be randomly samples for each tree.
- Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
- Typical values: 0.5-1

colsample_bytree [default=1]

- Similar to max_features in GBM. Denotes the fraction of columns to be randomly samples for each tree.
- Typical values: 0.5-1

colsample_bylevel [default=1]

- Denotes the subsample ratio of columns for each split, in each level.
- I don't use this often because subsample and colsample_bytree will do the job for you. but you can explore further if you feel so.

lambda [default=1]

- L2 regularization term on weights (analogous to Ridge regression)
- This used to handle the regularization part of XGBoost. Though many data scientists don't use it often, it should be explored to reduce overfitting.

alpha [default=0]

- L1 regularization term on weight (analogous to Lasso regression)
- Can be used in case of very high dimensionality so that the algorithm runs faster when implemented

scale_pos_weight [default=1]

- A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.

LightGBM

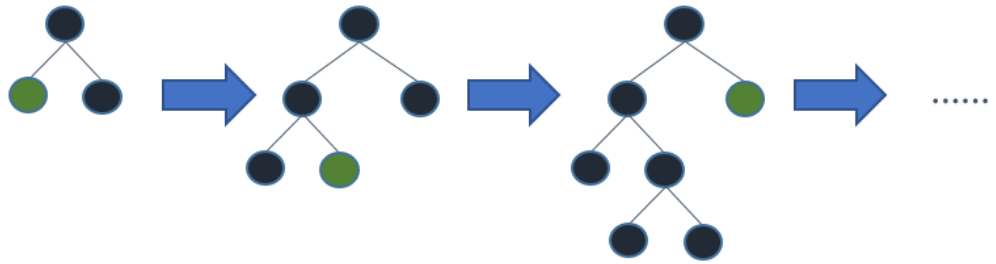
Advantages of LightGBM

Faster training speed and higher efficiency: LightGBM use histogram-based algorithm i.e it buckets continuous feature values into discrete bins which fasten the training procedure.

Lower memory usage: Replaces continuous values to discrete bins which result in lower memory usage.

Better accuracy than any other boosting algorithm: It produces much more complex trees by following leaf wise split approach rather than a level-wise approach which is the main factor in achieving higher accuracy. However, it can sometimes lead to overfitting which can be avoided by setting the max_depth parameter.

Compatibility with Large Datasets: It is capable of performing equally good with large datasets with a significant reduction in training time as compared to XGBOOST.



Leaf-wise tree growth

LightGBM is different from other gradient boosting frameworks because it uses a leaf-wise tree growth algorithm. Leaf-wise tree growth algorithms are known to converge faster than depth-wise growth algorithms. However, they're more prone to overfitting. overfitting.

Stacking

Motivation for Stacking

For each of our ensemble methods so far we have:

- Fit the base model on the same type (regression trees, for example).
- Combined the predictions in a naïve way.

Stacking is a way to generalize the ensembling approach to combine outputs of various types of model, and improves on the combination as well.

Motivation for Stacking

- Recall that in boosting, the final model T, we learn is a weighted sum of simple models, T_h,

$$T = \sum_h \lambda_h T_H$$

- where λ_h is the learning rate. In AdaBoost for example, we can analytically determine the optimal values of λ_h for each simple model T_h.

- On the other hand, we can also determine the final model T implicitly by ***learning any model, called meta-learner, that transforms the outputs of*** T_h ***into a prediction***.

Stacked Generalization

The framework for ***stacked generalization*** or ***stacking*** (Wolpert 1992) is:

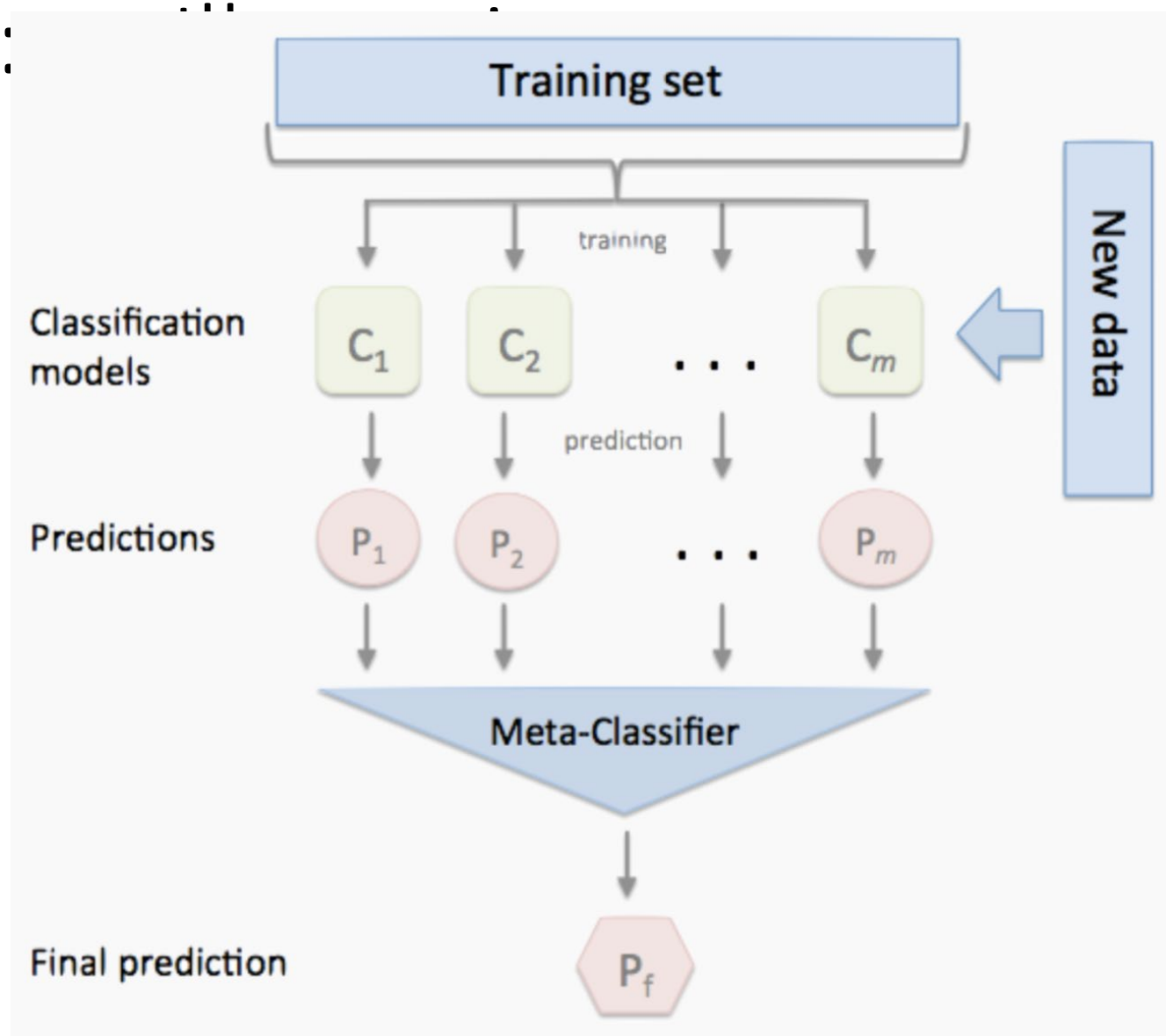
- train L number of models, T_l on the training data

$$\{(x_1, y_1), ..., (x_N, y_N)\}$$

- train a meta-learner $\tilde{T}$ on the predictions of the ensemble of models, i.e. train using the data

$$\{(T_1(x_1), ..., T_L(x_1), y_1), ..., (T_1(x_N), ..., T_L(x_N), y_N)\}$$

Stacking:


Training set
training
New data
Classification models
C_1
C_2
. . .
C_m
prediction
Predictions
P_1
P_2
. . .
P_m
Meta-Classifier
Final prediction
P_f

Stacking: General Guidelines

- The flexibility of stacking makes it widely applicable but difficult to analyze theoretically. Some general rules have been found through empirical studies:
- models in the ensemble should be diverse, i.e. their errors should not be correlated.
- for binary classification, each model in the ensemble should have error rate < 1/2.
- if models in the ensemble outputs probabilities, it's better to train the meta-learner on probabilities rather than predictions.
- apply regularization to the meta-learner to avoid overfitting.

Stacking in sklearn

Scikit-Learn implemented stacking algorithms implemented for you:

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html

We can also set it up by ‘manually’ fitting several base models, take the outputs of those models, and fitting the meta model on the outputs of those base models.

- It’s a model on models!