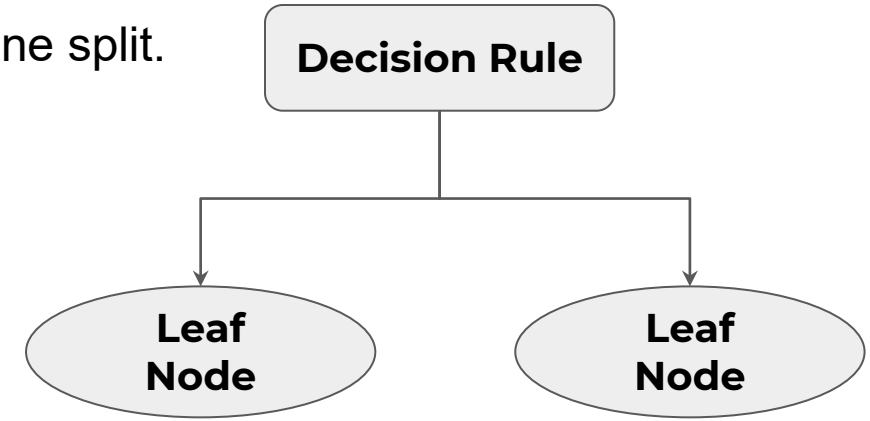
DECISION STUMP

A decision stump is a simple decision tree. You saw how decision trees work earlier. Now we're looking at a decision stump that makes a decision on one feature only.

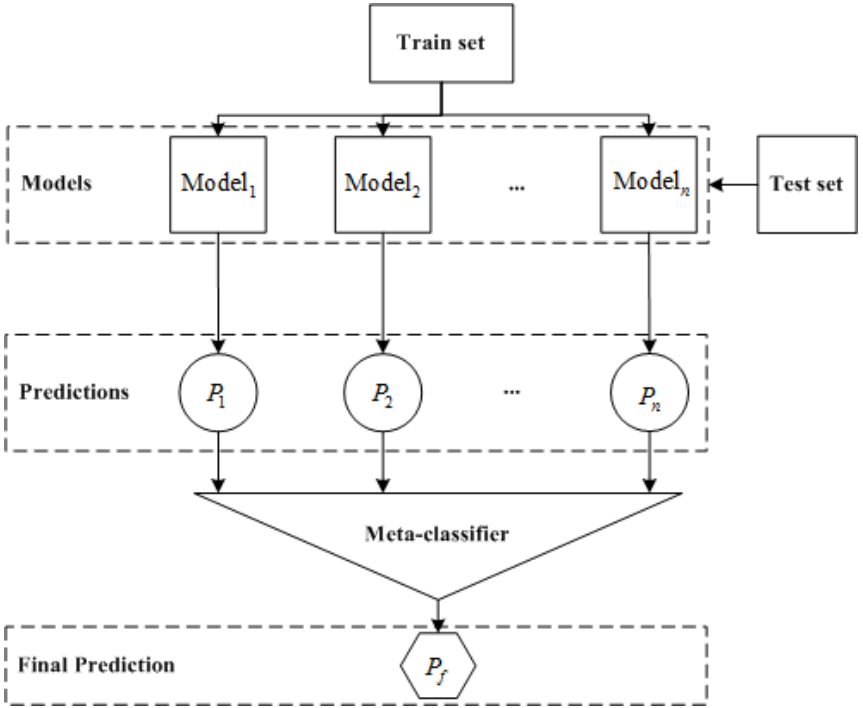
It's a tree with only one split.



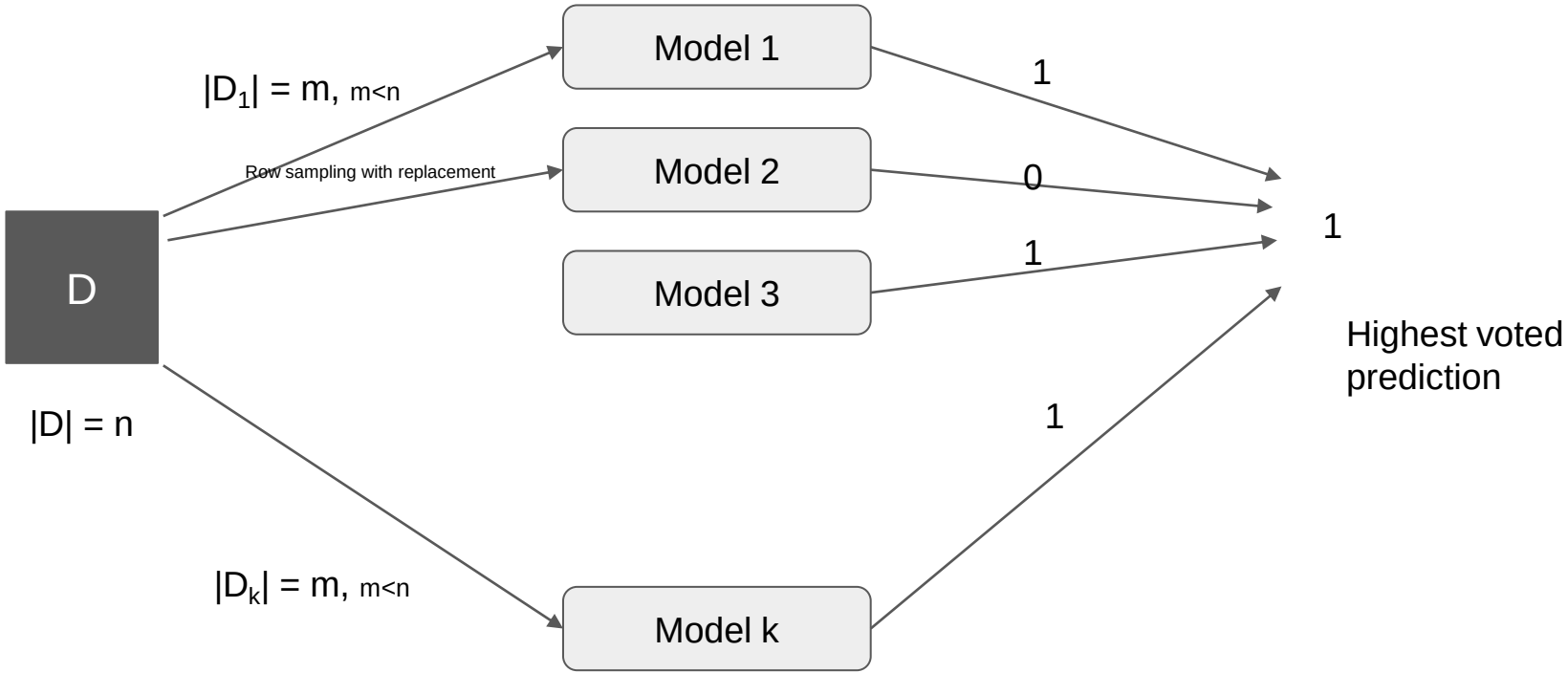
Stump

ENSEMBLE LEARNING

Ensemble Learning is the process of combining **weak classifiers** in order to create a system that eventually a **strong classifier.**

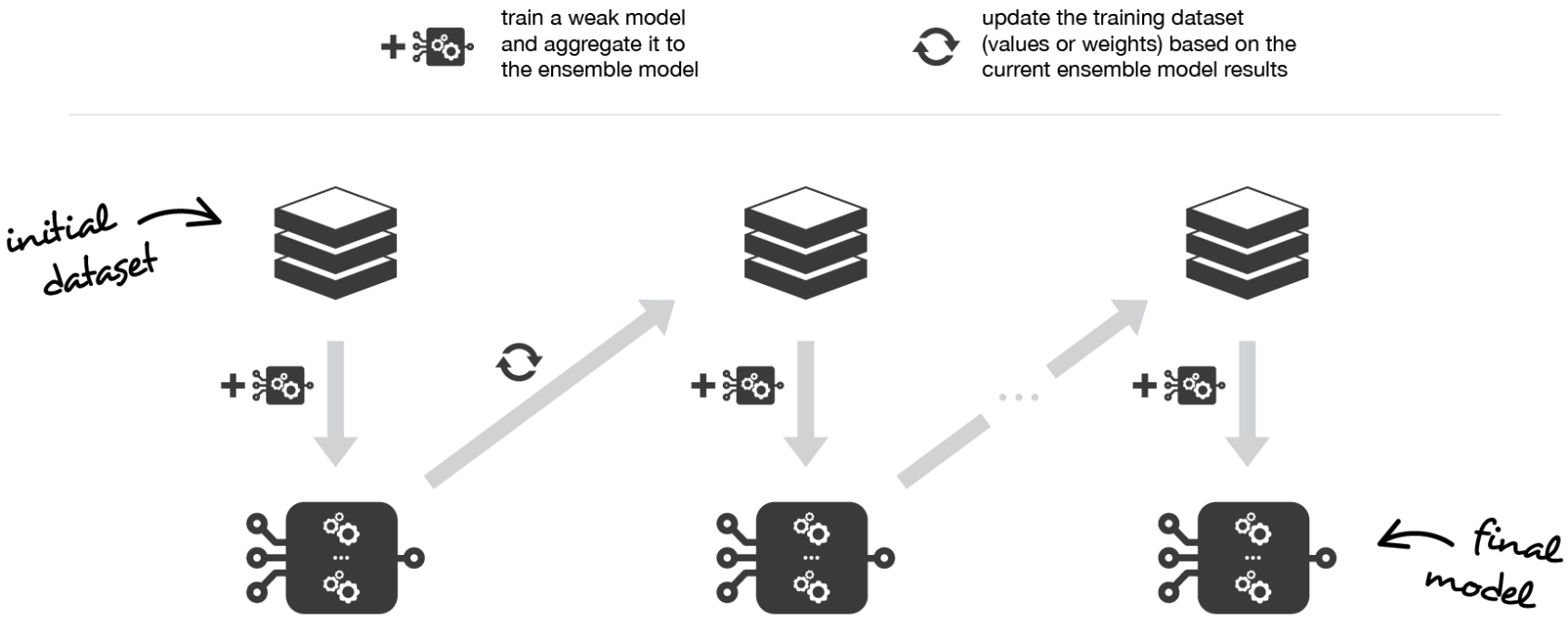


ENSEMBLE LEARNING: BAGGING



Multiple models are built independent of each other

ENSEMBLE LEARNING: BOOSTING



BOOSTING

In boosting, weights are also assigned to each training tuple. A series of k classifiers is iteratively learned. After a classifier, M_i , is learned, the weights are updated to allow the subsequent classifier,M_iC_1, to “pay more attention” to the training tuples that were misclassified by M_i. The final boosted classifier, M, combines the votes of each individual classifier, where the weight of each classifier’s vote is a function of its accuracy.

BOOSTING:

- In **sequential methods** the different combined weak models are no longer fitted independently from each others.
- The idea is to fit models **iteratively** such that the training of model at a given step depends on the models fitted at the previous steps. “Boosting” is the most famous of these approaches and it produces an ensemble model that is in general less biased than the weak learners that compose it.

BOOSTING:

- Boosting methods work in the same spirit as bagging methods: we build a family of models that are aggregated to obtain a strong learner that performs better. However, unlike bagging that mainly aims at reducing variance, boosting is a technique that consists in fitting sequentially multiple weak learners in a very adaptative way:

BOOSTING:

- each model in the sequence is fitted giving more importance to observations in the dataset that were badly handled by the previous models in the sequence. Intuitively, each new model **focus its efforts on the most difficult observations** to fit up to now, so that we obtain, at the end of the process, a strong learner with lower bias (even if we can notice that boosting can also have the effect of reducing variance)..

BOOSTING:

- Boosting, like bagging, can be used for regression as well as for classification problems

BOOSTING:

- Being **mainly focused at reducing bias**, the base models that are often considered for boosting are models with low variance but high bias.

ADABOOST

INNOMATİCS
RESEARCH LABS

ADABOOST

Adaptive Boosting (Adaboost) is a popular boosting algorithm.

Suppose we want to boost the accuracy of a learning method. We are given D, a data set of d class-labeled tuples, (X_1, y_1), (X_2, y_2), ... , (X_d, y_d), where y_i is the class label of tuple X_i.

Initially, AdaBoost assigns each training tuple an equal weight of 1=d. Generating k classifiers for the ensemble requires k rounds through the rest of the algorithm.

ADABOOST

Initially, AdaBoost assigns each training tuple an equal weight of 1=d. Generating k classifiers for the ensemble requires k rounds through the rest of the algorithm.

Generating k classifiers for the ensemble requires k rounds through the rest of the algorithm. In round i, the tuples from D are sampled to form a training set, D_i, of size d.

Sampling with replacement is used—the same tuple may be selected more than once. Each tuple's chance of being selected is based on its weight. A classifier model, M_i , is derived from the training tuples of D_i . Its error is then calculated using D_i as a test set. The weights of the training tuples are then adjusted according to how they were classified.

If a tuple was incorrectly classified, its weight is increased. If a tuple was correctly classified, its weight is decreased. A tuple's weight reflects how difficult it is to classify— the higher the weight, the more often it has been misclassified. These weights will be used to generate the training samples for the classifier of the next round.

ERROR RATE

To compute the error rate of model M_i , we sum the weights of each of the tuples in D_i that M_i misclassified.

$$error(M_i) = \sum_{j=1}^{d} w_j \times err(\boldsymbol{X_j})$$

$err(X_j)$ = 1 for incorrectly classified samples, 0 for correctly classified ones.

If the performance of classifier M_i is so poor that its error exceeds 0.5, then we abandon it.

If a tuple in round i was correctly classified, its weight is multiplied by error(M*i*)/(1+error(Mi))

PREDICTING THE LABEL

Unlike bagging, where each classifier was assigned an equal vote, boosting assigns a weight to each classifier's vote, based on how well the classifier performed. The lower a classifier's error rate, the more accurate it is, and therefore, the higher its weight for voting should be. The weight of classifier Mi 's vote is:

$$log \frac{1 - error(M_i)}{error(M_i)}$$

For each class, c, we sum the weights of each classifier that assigned class c to X. The class with the highest sum is the "winner" and is returned as the class prediction for tuple X.

ADABOOST ALGORITHM

Algorithm: Adaboost. A boosting algorithm—create an ensemble of classifiers. Each one gives a weighted vote.

Input:

- D, set of training tuples and their associated class labels;
- K, number of rounds (one classifier is generated per round)
- A classifier learning scheme

Output: A composite Model

Method:

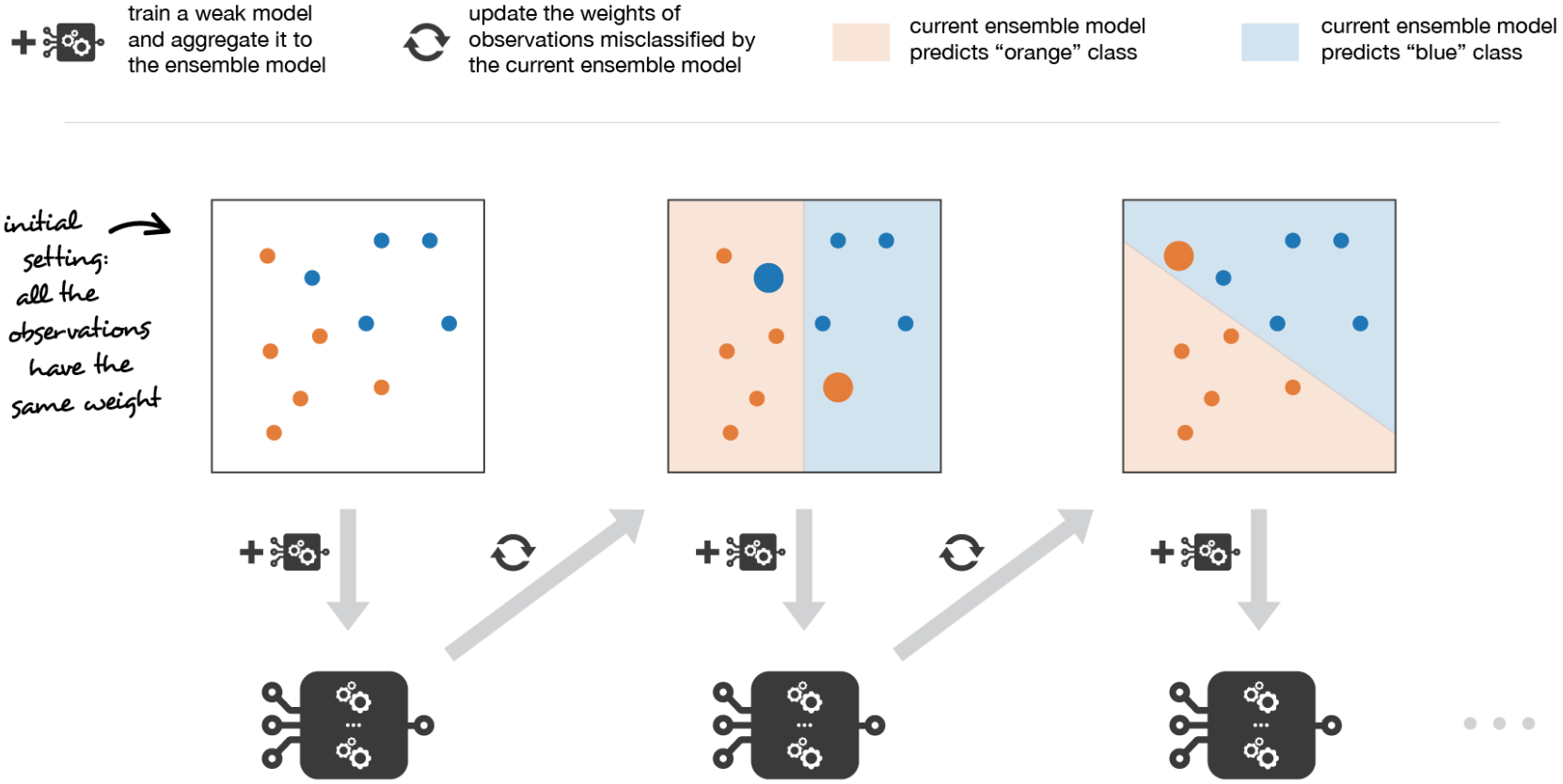
(1) initialize the weight of each tuple in D to 1=d;
(2) for i D 1 to k do // for each round:
(3) sample D with replacement according to the tuple weights to obtain Di ;
(4) use training set Di to derive a model, Mi ;
(5) compute error(M_i), the error rate of M_i
(6) if error.Mi/ > 0.5 then
(7) go back to step 3 and try again;
(8) endif
(9) for each tuple in Di that was correctly classified do
(10) multiply the weight of the tuple by error.Mi/=.1□ error.Mi//; // update weights
(11) normalize the weight of each tuple;
(12) endfor

ADABOOST ALGORITHM

Algorithm: Adaboost. A boosting algorithm—create an ensemble of classifiers. Each one gives a weighted vote.

Prediction:

(1) initialize weight of each class to 0
(2) for i = 1 to k do // for each classifier
(3) $w_i = log\frac{1-error(M_i)}{error(M_i)}$,weight of the classifier's vote
(4) c = $M_i(X)$, get class prediction for X from Mi
(5) add w_i to weight for class c
(6) endfor
(7) return the class with the largest weight;


train a weak model and aggregate it to the ensemble model
update the weights of observations misclassified by the current ensemble model
current ensemble model predicts “orange” class
current ensemble model predicts “blue” class
initial setting: all the observations have the same weight

ADABOOST ALGORITHM

- Adaboost updates weights of the observations at each iteration. Weights of well classified observations decrease relatively to weights of misclassified observations. **Models that perform better have higher weights in the final ensemble model.**

HYPER-PARAMETER TUNING:

- Let’s have a look at the hyper-parameters of the AdaBoost model. Hyper-parameters are the values that we give to a model before we start the modeling process. Let’s see all of them.

- **base_estimator**: The model to the ensemble, the default is a decision tree.
- **n_estimators**: Number of models to be built.
- **learning_rate**: shrinks the contribution of each classifier by this value.
- **random_state**: The random number seed, so that the same random numbers generated every time.

PROS

- AdaBoost is easy to implement. It iteratively corrects the mistakes of the weak classifier and improves accuracy by combining weak learners.
- You can use many base classifiers with AdaBoost.
- AdaBoost is not prone to over fitting. This can be found out via experiment results, but there is no concrete reason available.

CONS

- AdaBoost is sensitive to noise data.
- It is highly affected by outliers because it tries to fit each point perfectly.
- AdaBoost is slow.

GRADIENT
BOOSTING

INNOMATİCS
RESEARCH LABS

GRADIENT BOOSTING

Gradient Boosting = Gradient Descent + Boosting

- Fit an additive model (ensemble) in a forward stage-wise manner
- In each stage, introduce a weak learner to compensate the shortcomings of existing weak learners
- Shortcomings are identified by gradients
- In adaboost, shortcomings are identified by high weight data points
- Both high weight data points and the gradients tell us how to improve the model

GRADIENT BOOSTING ALGORITHM

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

GRADIENT BOOSTING ALGORITHM

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

GRADIENT BOOSTING ALGORITHM

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

GRADIENT BOOSTING ALGORITHM

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:

    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$

GRADIENT BOOSTING ALGORITHM

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For $m = 1$ to M:

   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$

   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

GRADIENT BOOSTING ALGORITHM

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

GRADIENT BOOSTING ALGORITHM

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (or weak learner, e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

WHY GRADIENT?

Target outcomes for each case are set based on the gradient of the error with respect to the prediction. Each new model takes a step in the direction that minimizes prediction error, in the space of possible predictions for each training case.

Suppose you have a true value y and a predicted value y'. The predicted value is constructed from some existing trees. Then you are trying to construct the next tree which gives a prediction z. Then your final prediction will be y'+z. The correct choice of z is z=y−y'. Therefore, you are now constructing trees to predict y−y'.

WHY GRADIENT?

Here, loss function is $L = \frac{1}{2}(y - \hat{y})^2$
and your prediction target for this new tree is the gradient of this loss function as

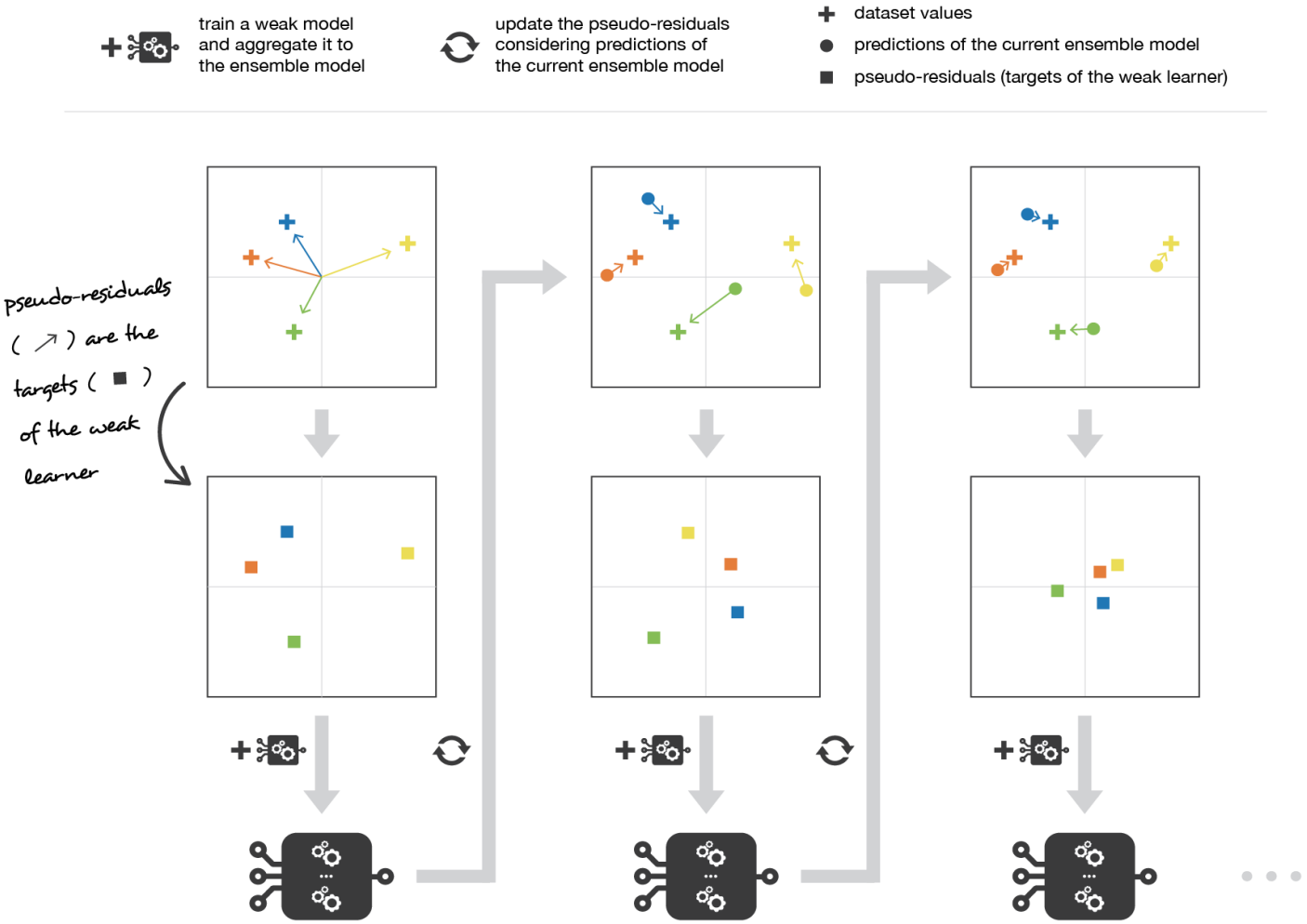
$$y - \hat{y} = -\frac{\partial L}{\partial \hat{y}}$$

WHY BOOSTING?

GRADIENT BOOSTING CLASSIFICATION

Target outcomes for each case are set based on the gradient of the error with respect to the prediction. Each new model takes a step in the direction that minimizes prediction error, in the space of possible predictions for each training case.

Suppose you have a true value y and a predicted value y'. The predicted value is constructed from some existing trees. Then you are trying to construct the next tree which gives a prediction z. Then your final prediction will be y'+z. The correct choice of z is z=y−y'. Therefore, you are now constructing trees to predict y−y'.


train a weak model and aggregate it to the ensemble model
update the pseudo-residuals considering predictions of the current ensemble model
dataset values
predictions of the current ensemble model
pseudo-residuals (targets of the weak learner)
pseudo-residuals (↗) are the targets (■) of the weak learner

GRADIENT BOOSTING ALGORITHM

- Gradient boosting updates values of the observations at each iteration. Weak learners are trained to fit the pseudo-residuals that indicate in which direction to correct the current ensemble model predictions to lower the error.

GRADIENT BOOSTING PARAMETERS

- The overall parameters of this ensemble model can be divided into 3 categories:

- **Tree-Specific Parameters:**
  These affect each individual tree in the model.
- **Boosting Parameters:**
  These affect the boosting operation in the model.
- **Miscellaneous Parameters:**
  Other parameters for overall functioning.

GB HYPER PARAMETERS:

1. **min_samples_split:**

- Defines the minimum number of samples (or observations) which are required in a node to be considered for splitting.
- 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.

2. **min_samples_leaf:**

- Defines the minimum samples (or observations) required in a terminal node or leaf.
- Used to control over-fitting similar to min_samples_split.
- Generally lower values should be chosen for imbalanced class problems because the regions in which the minority class will be in majority will be very small.

GB HYPER PARAMETERS:

3. min_weight_fraction_leaf:

- Similar to min_samples_leaf but defined as a fraction of the total number of observations instead of an integer.
- Only one of #2 and #3 should be defined.

4. max_depth:

- The maximum depth of a tree.
- 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.

GB HYPER PARAMETERS:

5. 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.

6. max_features:

- The number of features to consider while searching for a best split. These will be randomly selected.
- As a thumb-rule, square root of the total number of features works great but we
- should check upto 30-40% of the total number of features.
- Higher values can lead to over-fitting but depends on case to case.

PROS & CONS OF GRADIENT BOOSTING:

Pros:

- It is extremely powerful machine learning classifier.
- Accepts various types of inputs that make it more flexible.
- It can be used for both regression and classification.
- It gives you features important for the output.

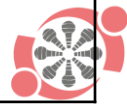
Cons:

- It takes longer time to train as it can’t be parallelized.
- More likely to over fit as it obsessed with the wrong output as it learns from past mistakes.
- In some cases, Tuning is very hard as it has many parameters to tune.

Adaboost	Gradient Boost
1. An additive model where shortcomings of previous models are identified by high-weight data points.	1. An additive model where shortcomings of previous models are identified by the gradient.

Adaboost	Gradient Boost
1. An additive model where shortcomings of previous models are identified by high-weight data points.	1. An additive model where shortcomings of previous models are identified by the gradient.
2. The weak learners incase of adaptive boosting are a very basic form of decision tree known as stumps.	2. Weak learners are decision trees constructed in a greedy manner with split points based on purity scores (i.e., Gini, minimize loss). Thus, larger trees can be used with around 4 to 8 levels. Learners should still remain weak and so they should be constrained (i.e., the maximum number of layers, nodes, splits, leaf nodes)

Adaboost	Gradient Boost
1. An additive model where shortcomings of previous models are identified by high-weight data points.	1. An additive model where shortcomings of previous models are identified by the gradient.
2. The weak learners incase of adaptive boosting are a very basic form of decision tree known as stumps.	2. Weak learners are decision trees constructed in a greedy manner with split points based on purity scores (i.e., Gini, minimize loss). Thus, larger trees can be used with around 4 to 8 levels. Learners should still remain weak and so they should be constrained (i.e., the maximum number of layers, nodes, splits, leaf nodes)
3. Each classifier has different weights assigned to the final prediction based on its performance. The final prediction is based on a majority vote of the weak learners' predictions weighted by their individual accuracy.	3. All the learners have equal weights in the case of gradient boosting. The weight is usually set as the learning rate which is small in magnitude



XGBOOST

INNOMATICS
RESEARCH LABS

XGBOOST

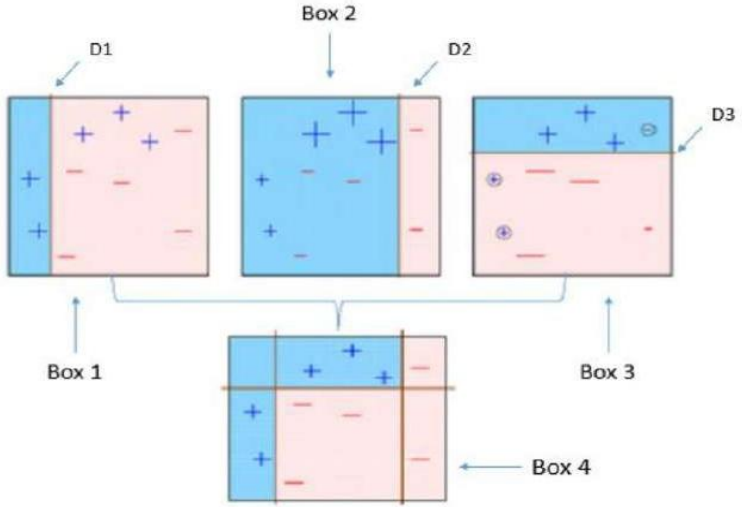
- XGBoost (Extreme Gradient Boosting) is an **optimized distributed** gradient boosting library.
- It uses Gradient Boosting (GBM) framework at core.
- Gradient boosting is **supervised learning algorithm**, which attempts to accurately predict a target variable by combining the **estimates of a set of simpler, weaker models.**

WHY IS IT SO GOOD?

- **Parallel Computing:** It is enabled with parallel processing (Using OpenMP)
  i.e., when you run XGBoost, by default, It would use all the cores of your laptop/machine.
- **Regularization:** Regularization is a technique used to avoid over fitting in linear and tree-based models.
- **Enabled Cross Validation:** XGBoost is enabled with internal CV function.
- **Missing Values:** XGBoost is designed to handle missing values internally.

HOW DOES XGBOOST WORK ?

- Boosting is a sequential process; i.e., trees are grown using the information from a previously grown tree one after the other. This process slowly learns from data and tries to improve its prediction in subsequent iterations.



XGBOOST TUNING PARAMETERS

- Choose relatively high learning rate to start with.
- Tune tree-specific parameters (max_depth, min_child_weight, gamma, subsample, colsample_bytree) for decided learning rate and number of trees
- Tune regularization parameters (lambda, alpha) for xgboost which can help reduce model complexity and enhance performance.
- Lower the learning rate and decide the optimal parameters

REFERENCES:

- Statistics and Machine Learning in Python by –
  Edouard Duchesnay, Tommy Lofstedt, Feki Younes

THAT'S ALL FOLKS