Ensemble Models: Bagging, Boosting, & Stacking

ISE – 364 / 464

Dept. of Industrial and Systems Engineering

Griffin Dean Kent

Overview of Ensemble Models

Ensemble Methods

In most machine learning models, only a single model is utilized to generate predictions for a target variable. However, a powerful technique that is often implemented is that of **ensemble models**. As opposed to training a single predictive model, ensemble models aim at training multiple models (sometimes referred to as "base learners") and then aggregating their predictions to generate a final "voted" prediction. The core idea behind this technique is that groups of models will yield better predictions over the predictions of a single model.

- There are three core types of ensemble techniques that we will discuss as well as their common implementations: **Bagging**, **Boosting**, and **Stacking**.

- Ensemble methods aid in **improving** model **generalization** by reducing the variance of high-complexity decision boundaries as well as model **accuracy** by sequentially improving upon the weaknesses of previous models.

Bagging

Bagging

Short for "**Bootstrap Aggregation**", bagging is an ensemble technique that improves the predictive accuracy and robustness of models (especially in high-complexity models that have a high variance). Bagging consists of three steps:

- **Bootstrap Sampling**: This phase of bagging involves creating $s \in \mathbb{N}$ sub-datasets $\left\{\mathcal{D}_j\right\}_{j=1}^{s}$ of the original dataset through a process called bootstrap sampling. This is a process by which one randomly selects samples from the dataset (with replacement) to generate multiple subsets (of equal size) of the original dataset.
- **Training Phase**: On **each** of the newly generated **sub-datasets**, a model $\mathcal{L}(x;\theta)$(for some vector θ) is trained independently. Thus, after this phase, one will have s models $\left\{\mathcal{L}_j\right\}_{j=1}^{s}$ trained on different datasets.
- **Aggregation**: Once all the individual base models have been trained, bagging combines their predictions into a single value via some **aggregation function** $\mathcal{A}(\cdot)$ (either by using the **average** of the predictions for regression problems or simply using the **mode** of the predictions for classification problems). Thus, the final ensembled learner is given by

$$h_\theta(x;\mathcal{L}) \coloneqq \mathcal{A}\left(\left\{\mathcal{L}_j\left(x;\theta_j\right)\right\}_{j=1}^{s}\right), \qquad \text{where } \theta_j \text{ is a vector of parameters for } \mathcal{L}_j.$$

The core idea behind bagging is that by training multiple models on slightly different variations of the original dataset and aggregating their predictions, the variance of the new "combined" model is reduced. Further, the combined model will typically be able to generalize much better than any of the individual models (hence, yielding a reduction in variance for complex models).

Illustration of Bagging

Original Dataset $\mathcal{D}$

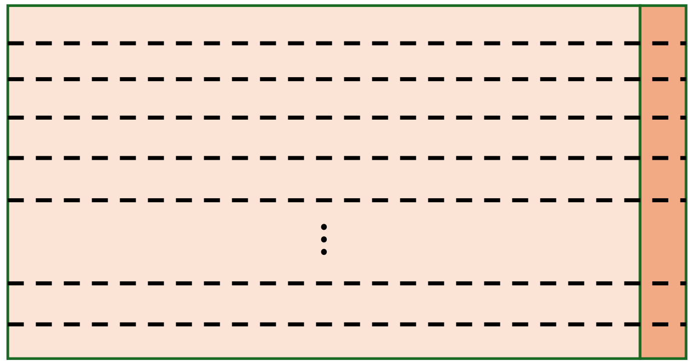


Illustration of Bagging

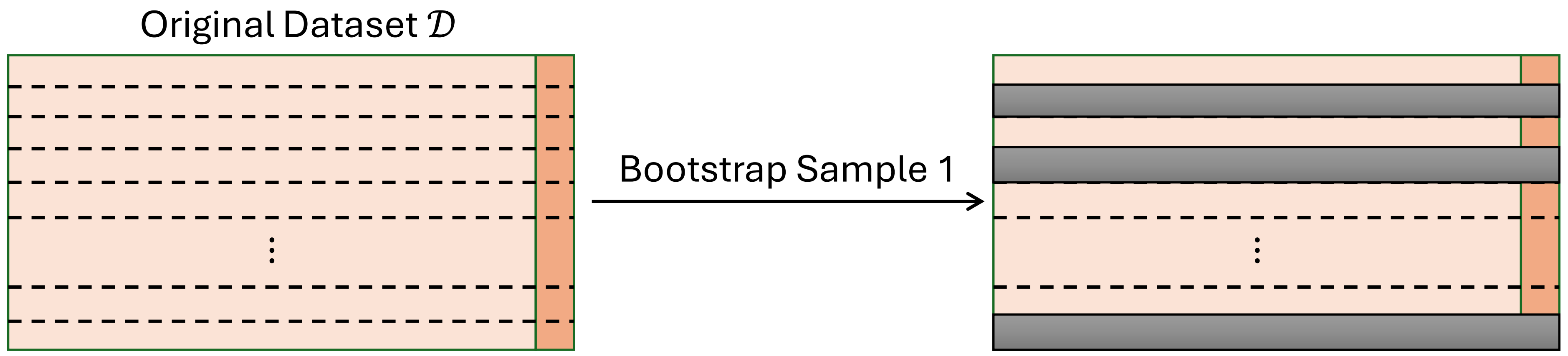


Illustration of Bagging

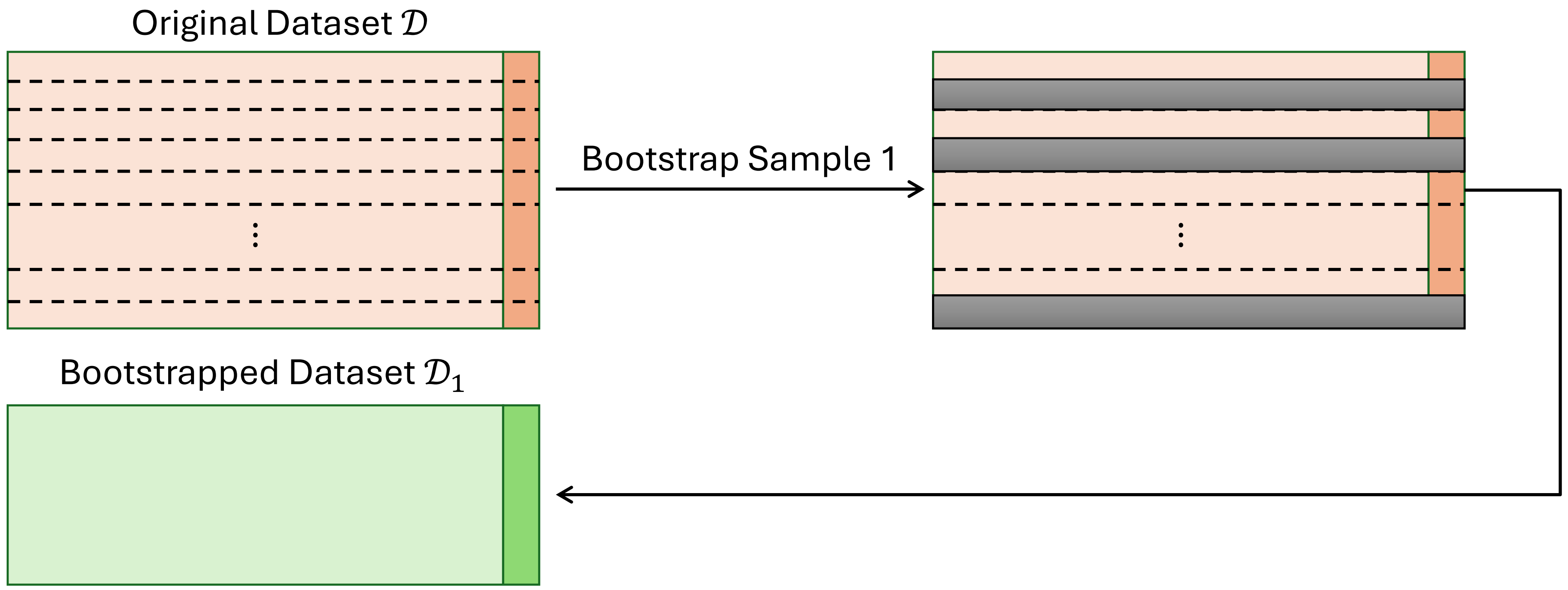

Original Dataset $\mathcal{D}$
Bootstrap Sample 1
Bootstrapped Dataset $\mathcal{D}_1$

Illustration of Bagging

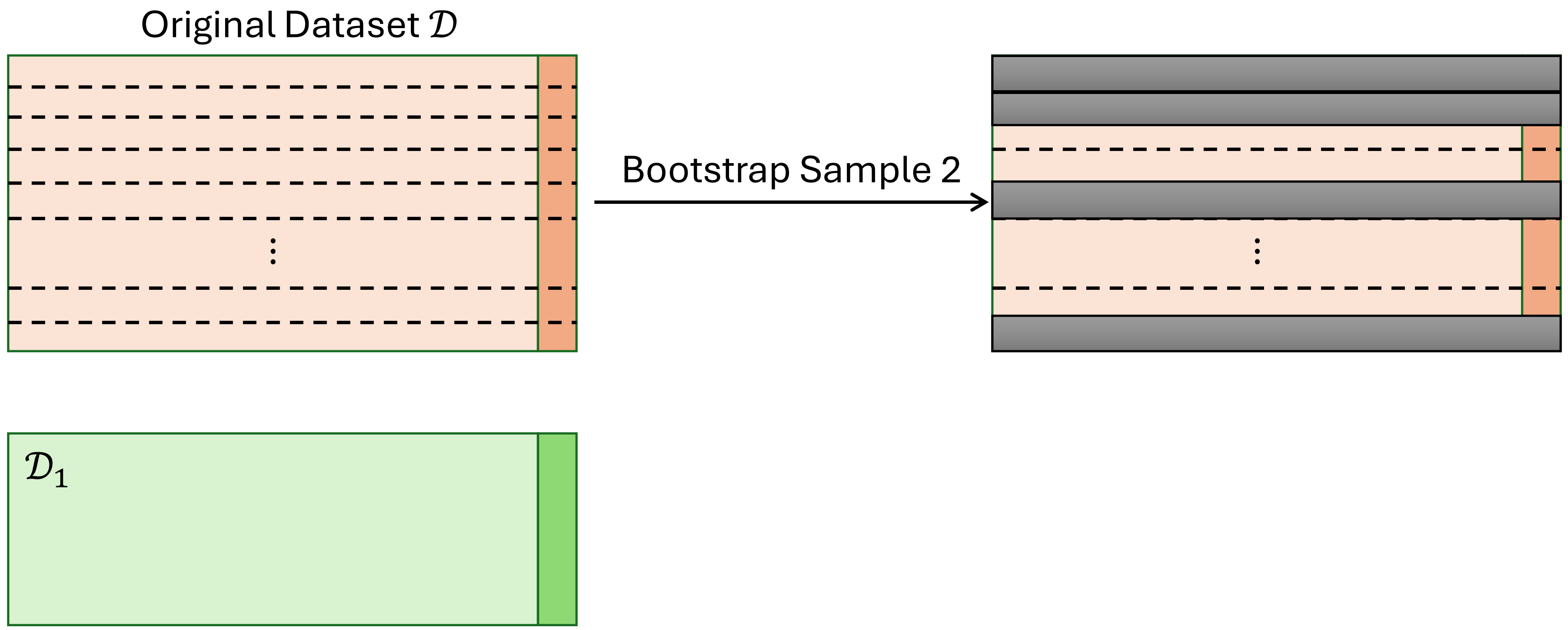


Illustration of Bagging

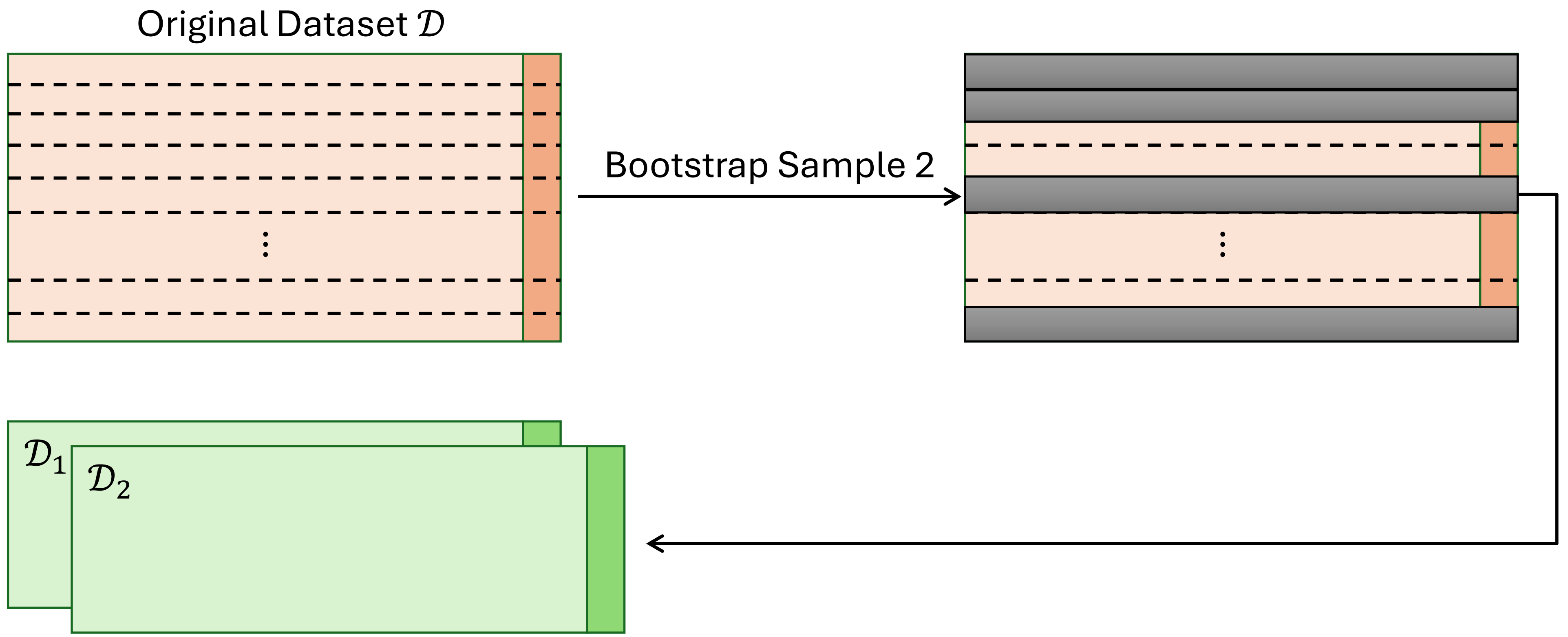


Illustration of Bagging

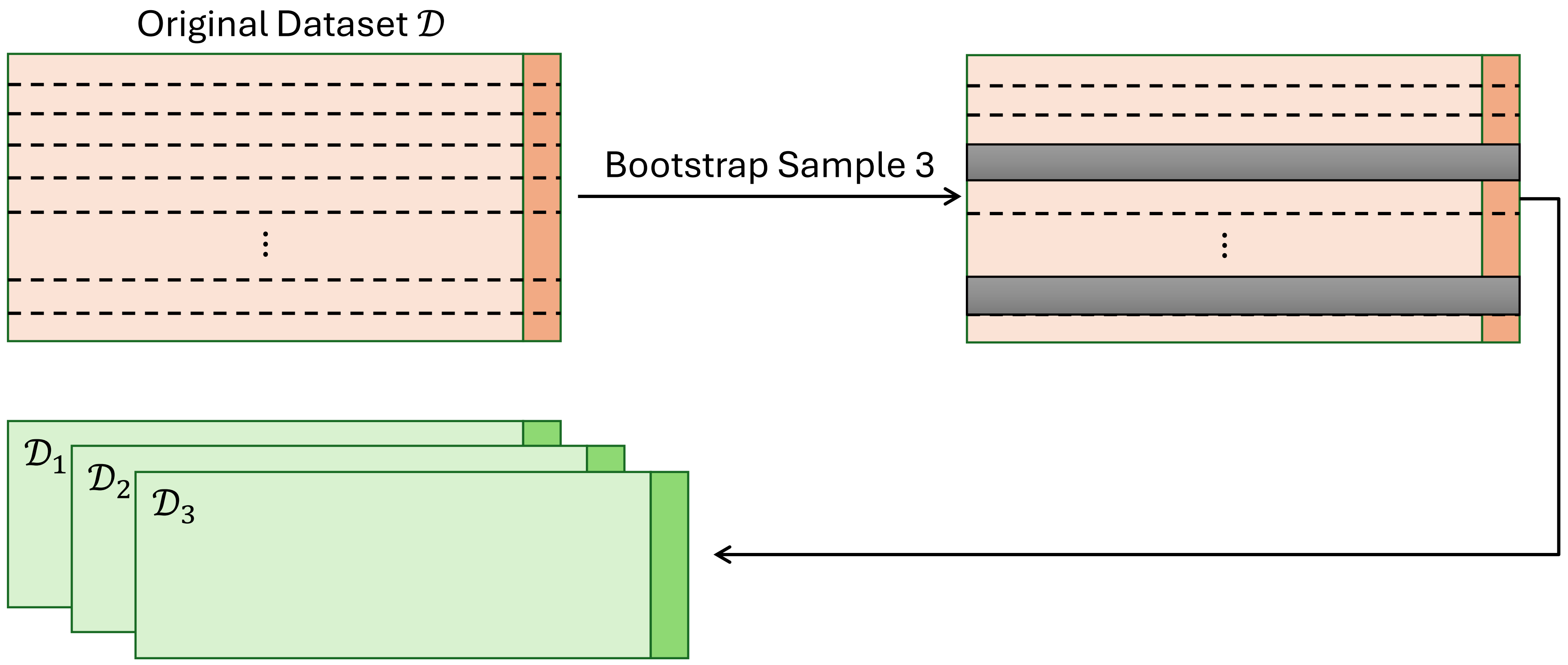


Illustration of Bagging

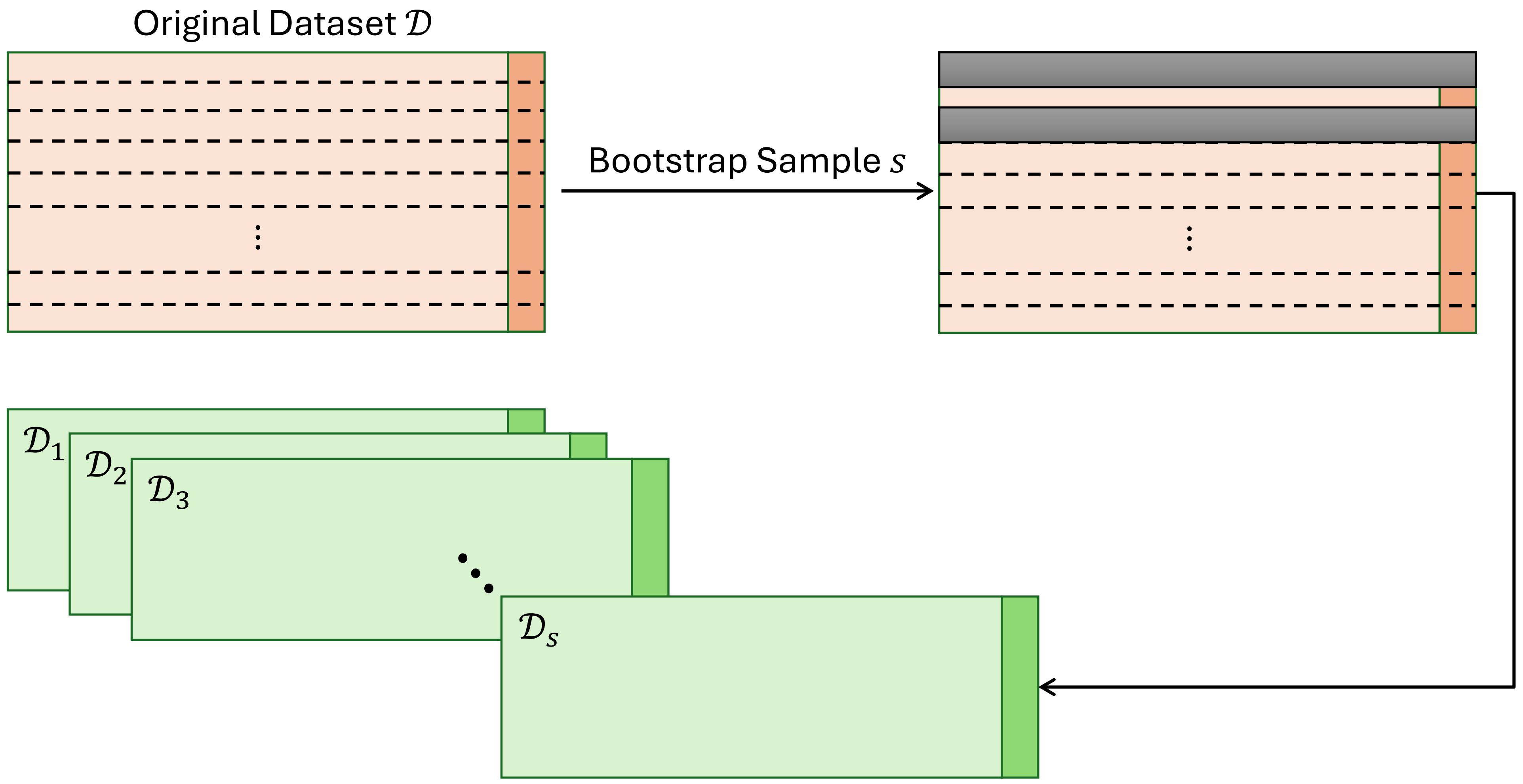


Illustration of Bagging

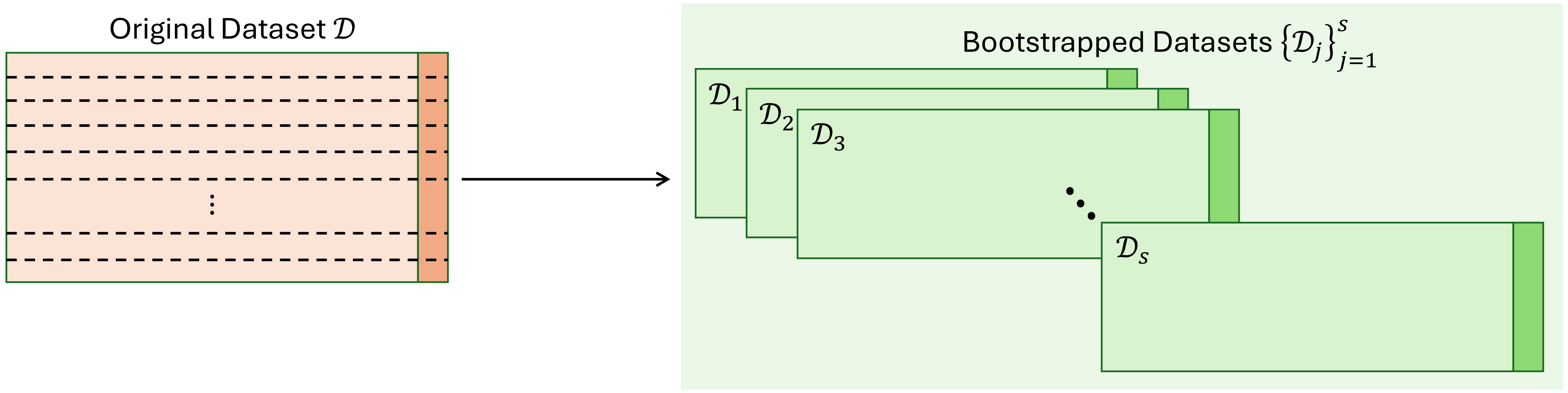

Original Dataset $\mathcal{D}$
Bootstrapped Datasets $\{\mathcal{D}_j\}_{j=1}^{s}$
$\mathcal{D}_1$
$\mathcal{D}_2$
$\mathcal{D}_3$
$\mathcal{D}_s$

Illustration of Bagging

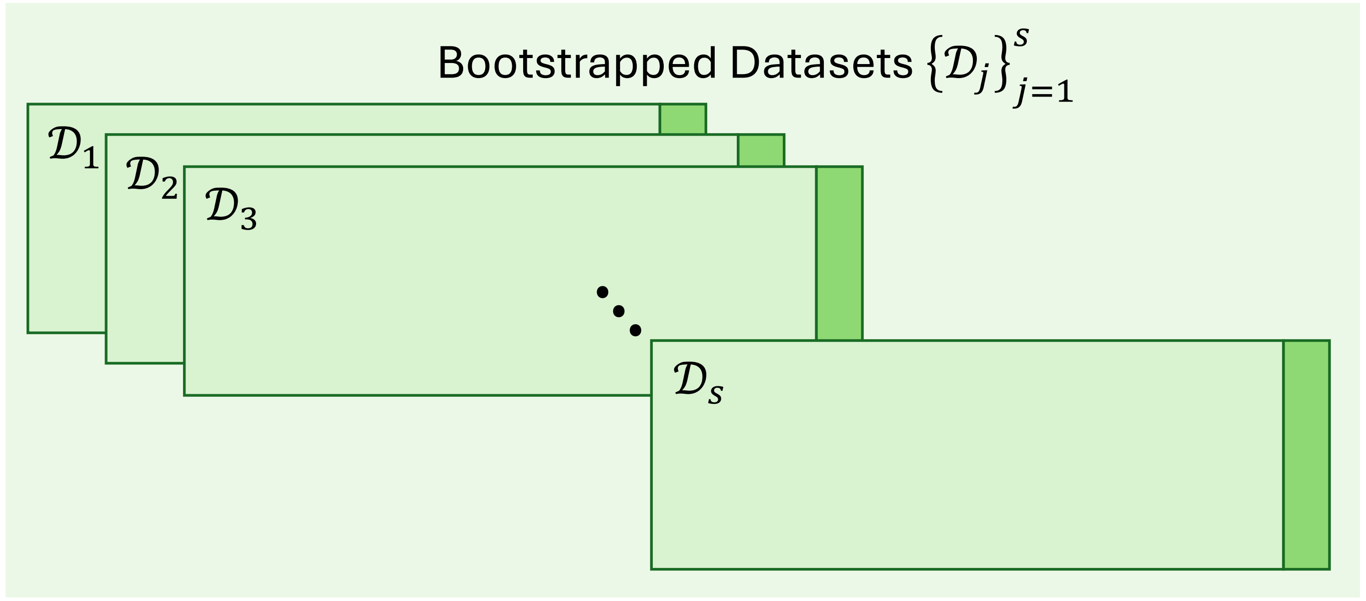


Illustration of Bagging

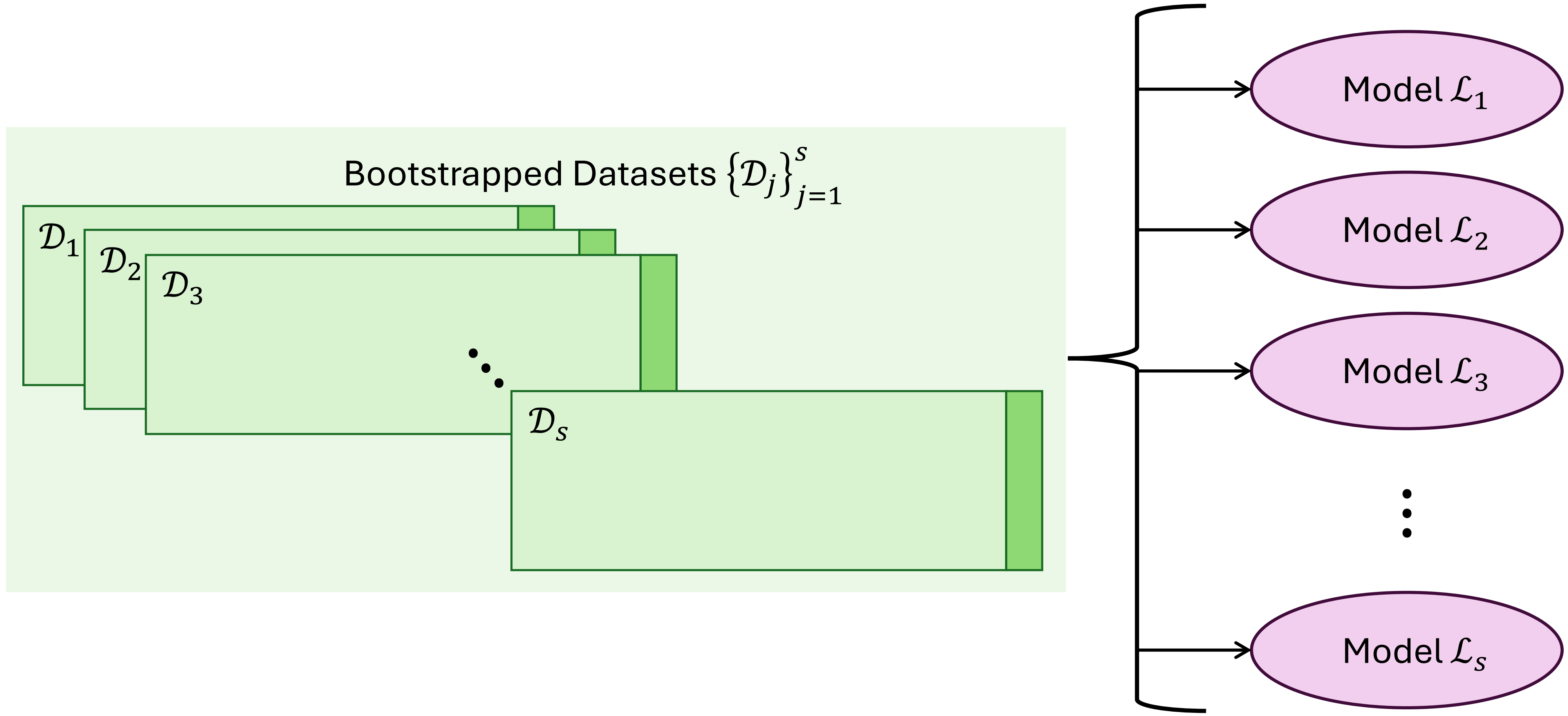
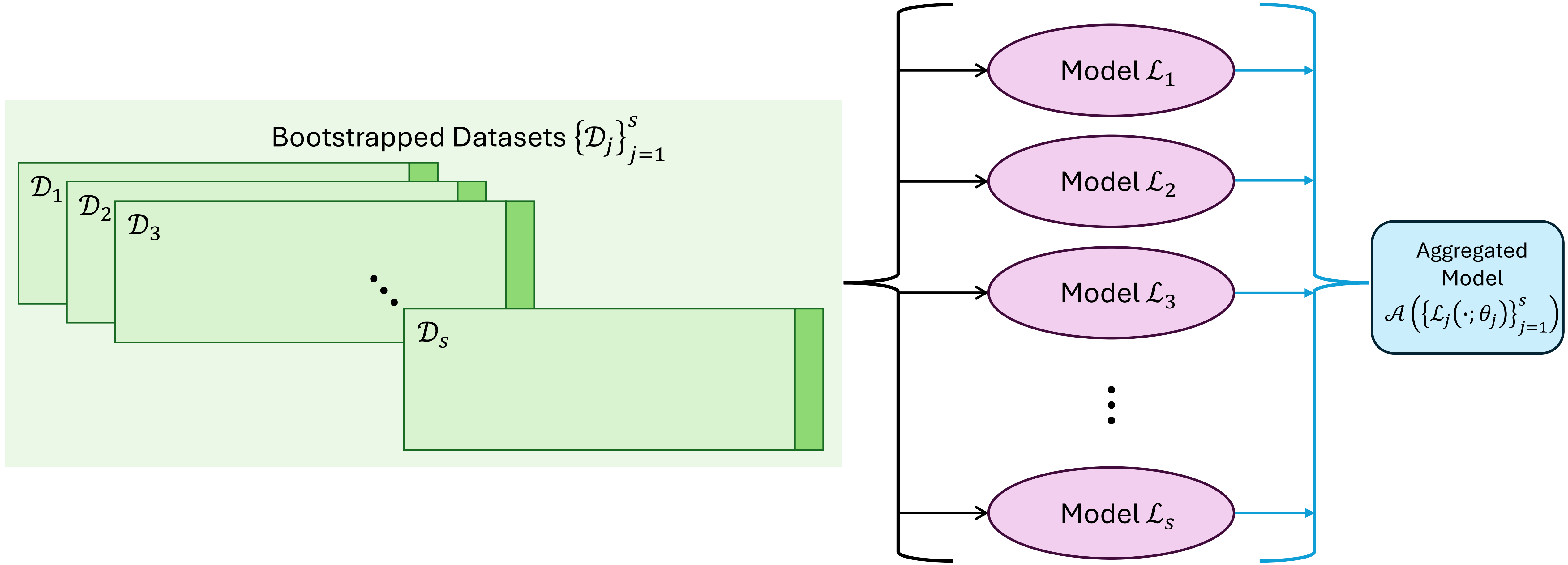
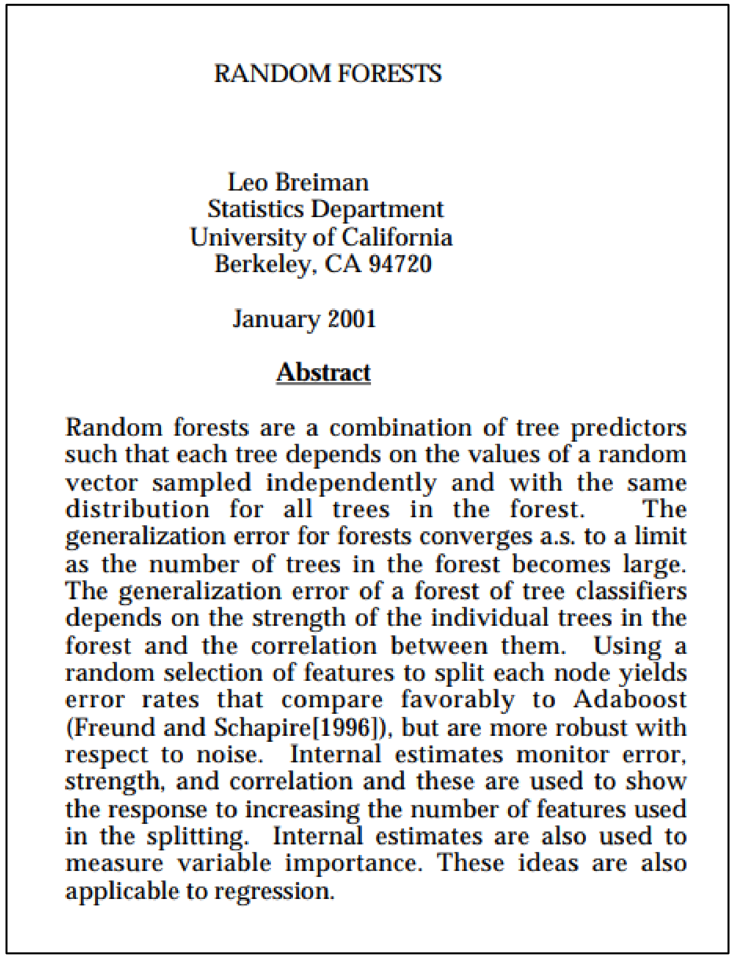


Illustration of Bagging



Random Forests


RANDOM FORESTS

Leo Breiman
Statistics Department
University of California
Berkeley, CA 94720

January 2001

Abstract

Random forests are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. The generalization error for forests converges a.s. to a limit as the number of trees in the forest becomes large. The generalization error of a forest of tree classifiers depends on the strength of the individual trees in the forest and the correlation between them. Using a random selection of features to split each node yields error rates that compare favorably to Adaboost (Freund and Schapire[1996]), but are more robust with respect to noise. Internal estimates monitor error, strength, and correlation and these are used to show the response to increasing the number of features used in the splitting. Internal estimates are also used to measure variable importance. These ideas are also applicable to regression.

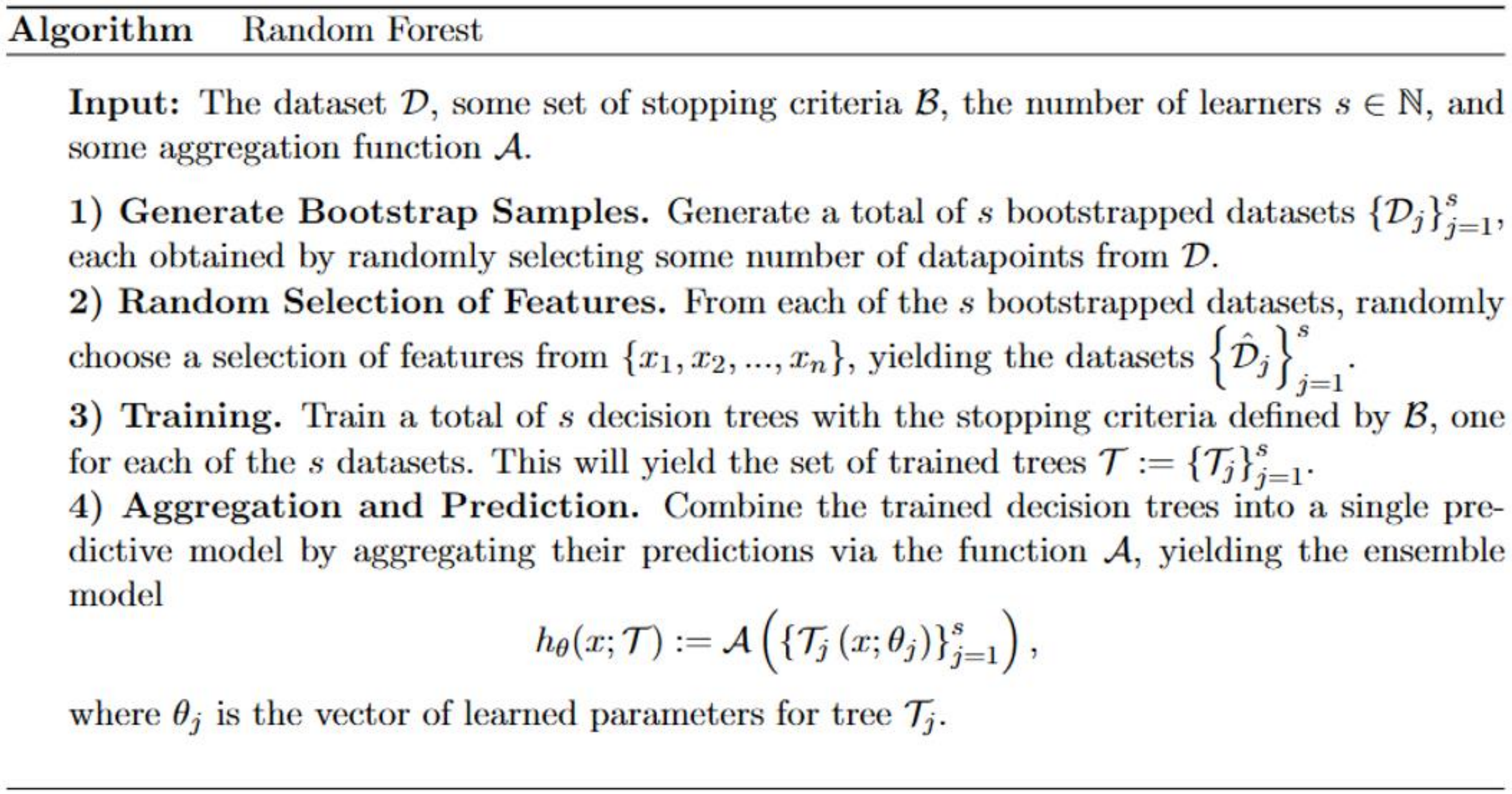
- In 2001, Leo Brieman developed the idea of the Random Forest machine learning algorithm.

- A **Random Forest** model is an **ensemble** of s **decision trees** $\{\mathcal{T}_j\}_{j=1}^{s}$ that utilizes a specific type of bagging procedure.

- Although each of the decision tree learners $\{\mathcal{T}_j\}_{j=1}^{s}$ will be trained on their own bootstrapped dataset, the random forest algorithm adds one more layer of randomness by also selecting random subsets of the features x_k, for $k \in \{1,2,\ldots,n\}$, to be used to train each of the trees.

- Thus, the bootstrapped datasets $\{\mathcal{D}_j\}_{j=1}^{s}$ will not only have randomly selected rows of the original $\mathcal{D}$, they will also have randomly selected columns.

Random Forest Algorithm



Algorithm Random Forest

Input: The dataset $\mathcal{D}$, some set of stopping criteria $\mathcal{B}$, the number of learners $s \in \mathbb{N}$, and some aggregation function $\mathcal{A}$.

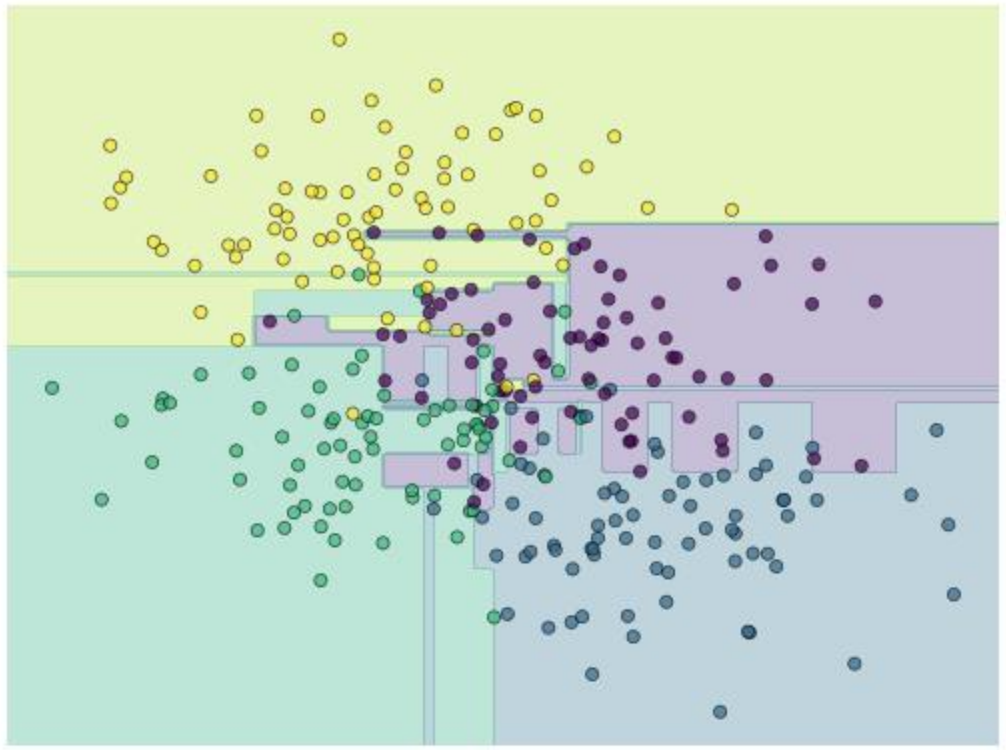
1) Generate Bootstrap Samples. Generate a total of s bootstrapped datasets $\{\mathcal{D}_j\}_{j=1}^{s}$, each obtained by randomly selecting some number of datapoints from $\mathcal{D}$.
2) Random Selection of Features. From each of the s bootstrapped datasets, randomly choose a selection of features from $\{x_1, x_2, ..., x_n\}$, yielding the datasets $\left\{\hat{\mathcal{D}}_j\right\}_{j=1}^{s}$.
3) Training. Train a total of s decision trees with the stopping criteria defined by $\mathcal{B}$, one for each of the s datasets. This will yield the set of trained trees $\mathcal{T} := \{\mathcal{T}_j\}_{j=1}^{s}$.
4) Aggregation and Prediction. Combine the trained decision trees into a single predictive model by aggregating their predictions via the function $\mathcal{A}$, yielding the ensemble model

$$h_\theta(x; \mathcal{T}) := \mathcal{A}\left(\{\mathcal{T}_j\left(x; \theta_j\right)\}_{j=1}^{s}\right),$$

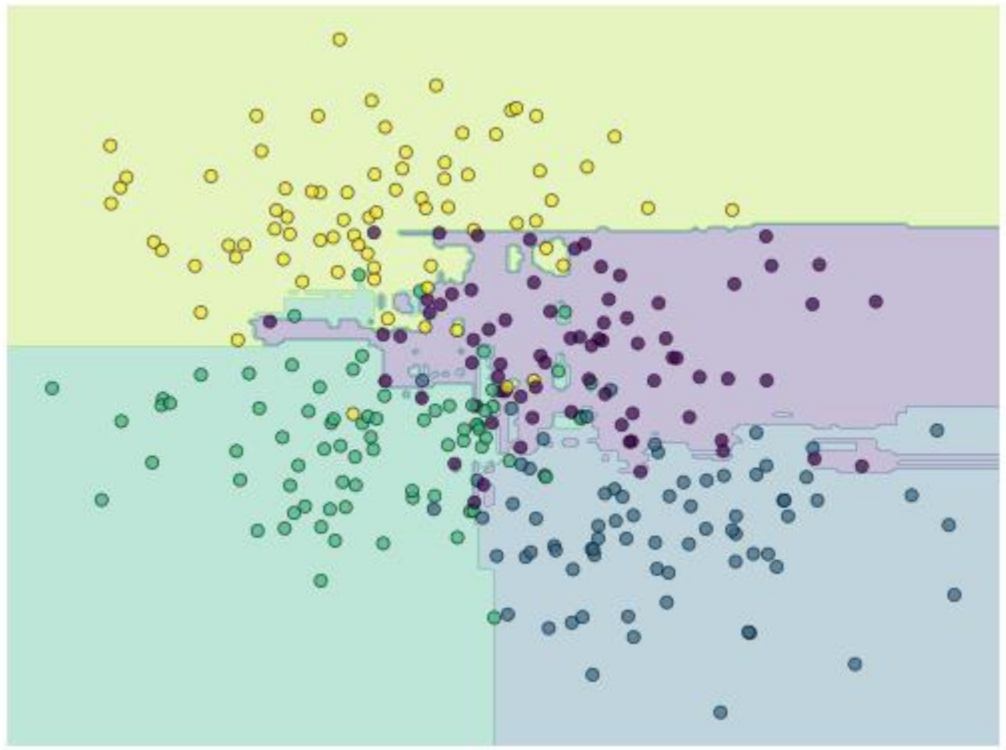
where θ_j is the vector of learned parameters for tree $\mathcal{T}_j$.

Decision Space Obtained by a Random Forest

A Single Decision Tree (Decision Space)



Random Forest (Decision Space)



See References

Boosting

Boosting

This is an ensemble technique that combines the predictions of several base (weak) learners sequentially to create a strong predictive model. Unlike bagging, which builds models independently of one another, boosting builds models sequentially, where each subsequent model aims to correct the predictive errors made by the previous models.

Typically, boosting is a process that follows the four following phases:

- **Base Model**: Boosting starts by training a base model $\mathcal{L}_1$ on the entire dataset $\mathcal{D}_1 \coloneqq \mathcal{D}$, where each datapoint will have equal weights (importance) assigned to them.
- **Weight Adjustment**: After the first model is trained, the weights of the incorrectly predicted datapoints are increased, making those points more influential in the subsequent trainings, yielding dataset $\mathcal{D}_2$.
- **Sequential Learning**: Subsequent models $\mathcal{L}_j$ (for $j \in \{1,2,\dots,s\}$) are built which focus more on the misclassified datapoints from the previous models. These models are trained on a modified version of the dataset $\mathcal{D}_j$ where the weights of these misclassified points are adjusted to emphasize their importance.
- **Combining Learners**: Boosting then combines the predictions of all these sequential models using a weighted aggregation function $\mathcal{A}_w(\cdot)$ (each sequential learner is weighted differently) to generate the final prediction. Thus, the final ensembled learner is given by

$$h_\theta(x; w, \mathcal{L}) \coloneqq \mathcal{A}_w\left(\left\{\mathcal{L}_j(x; \theta_j)\right\}_{j=1}^{s}\right), \qquad \text{where } \theta_j \text{ is a vector of parameters for } \mathcal{L}_j \text{ and } w \in \mathbb{R}^s.$$

Boosting methods can be more sensitive to overfitting compared to bagging but can often achieve higher accuracy.

Illustration of Boosting

Original Dataset $\mathcal{D}_1$

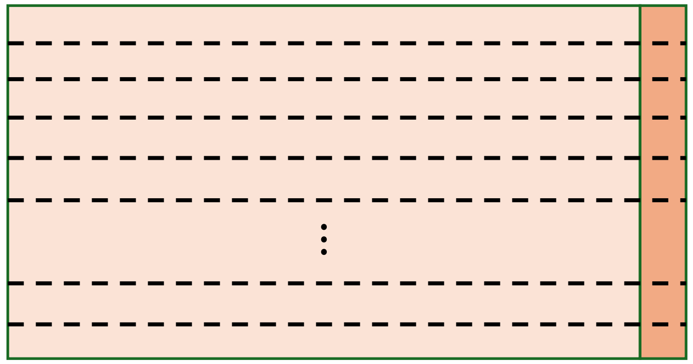
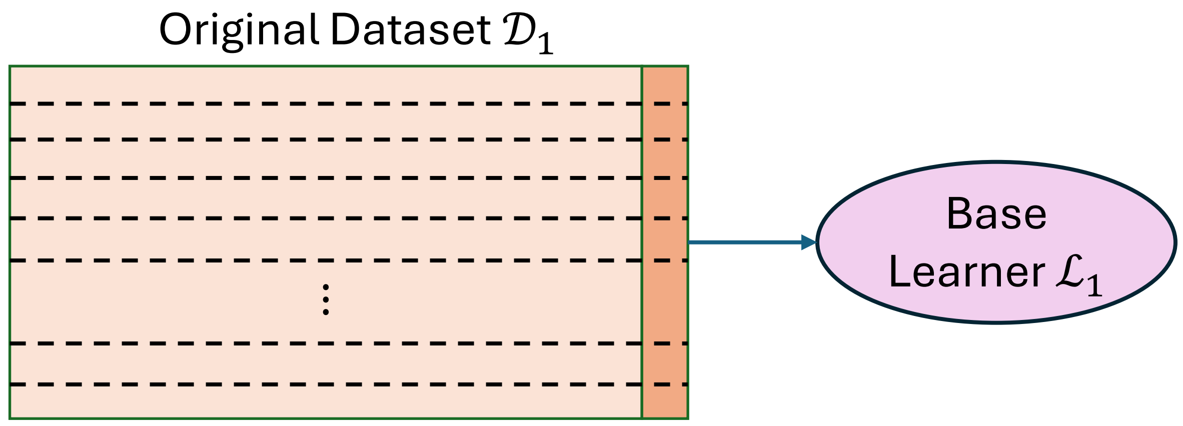


Illustration of Boosting



Original Dataset $\mathcal{D}_1$
⋮
Base Learner $\mathcal{L}_1$

Illustration of Boosting

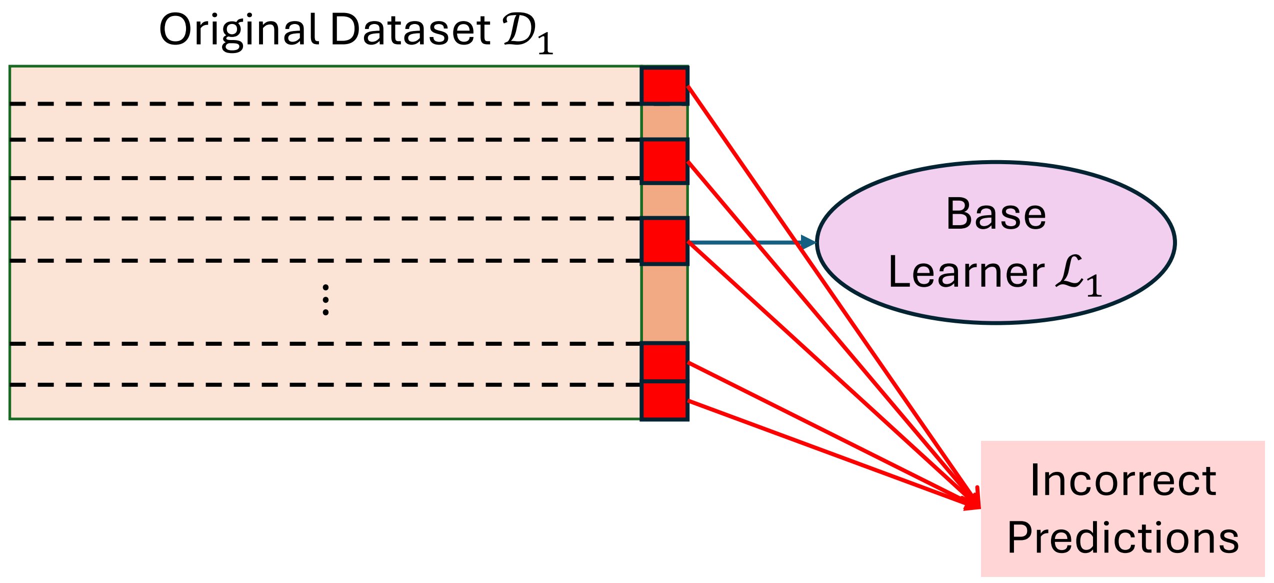

Original Dataset $\mathcal{D}_1$
⋮
Base Learner $\mathcal{L}_1$
Incorrect Predictions

Illustration of Boosting

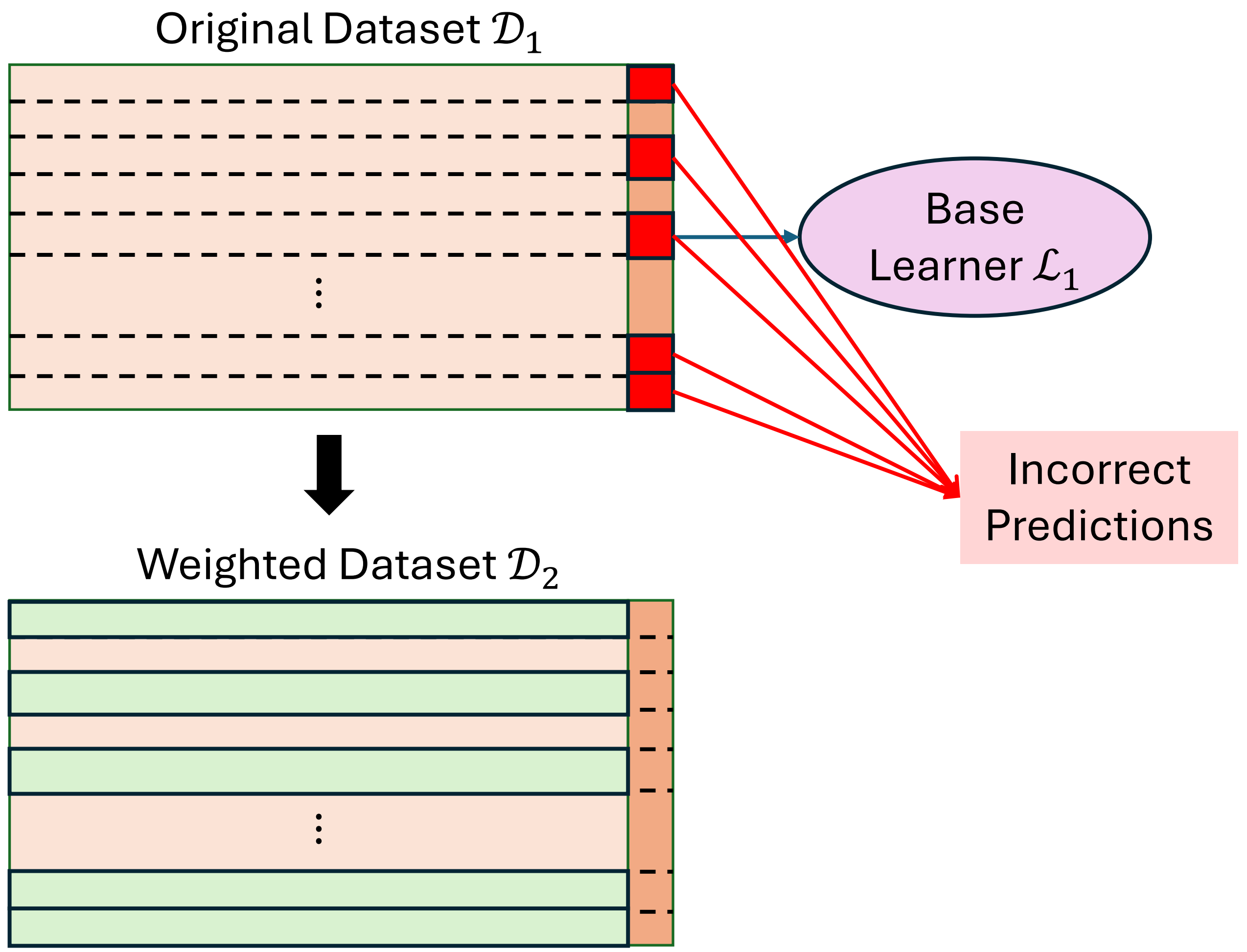

Original Dataset $\mathcal{D}_1$
Base Learner $\mathcal{L}_1$
Incorrect Predictions
Weighted Dataset $\mathcal{D}_2$

Illustration of Boosting

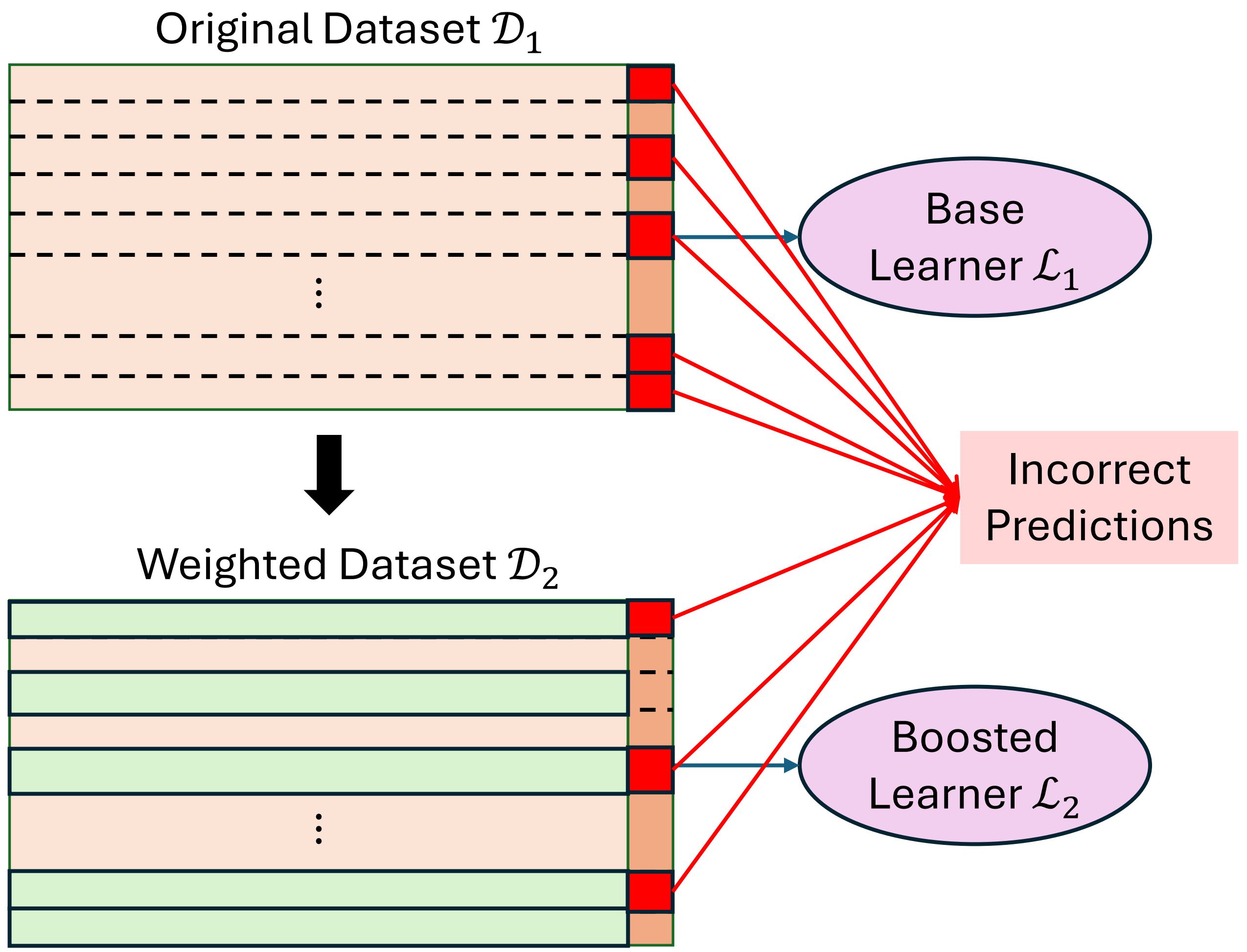
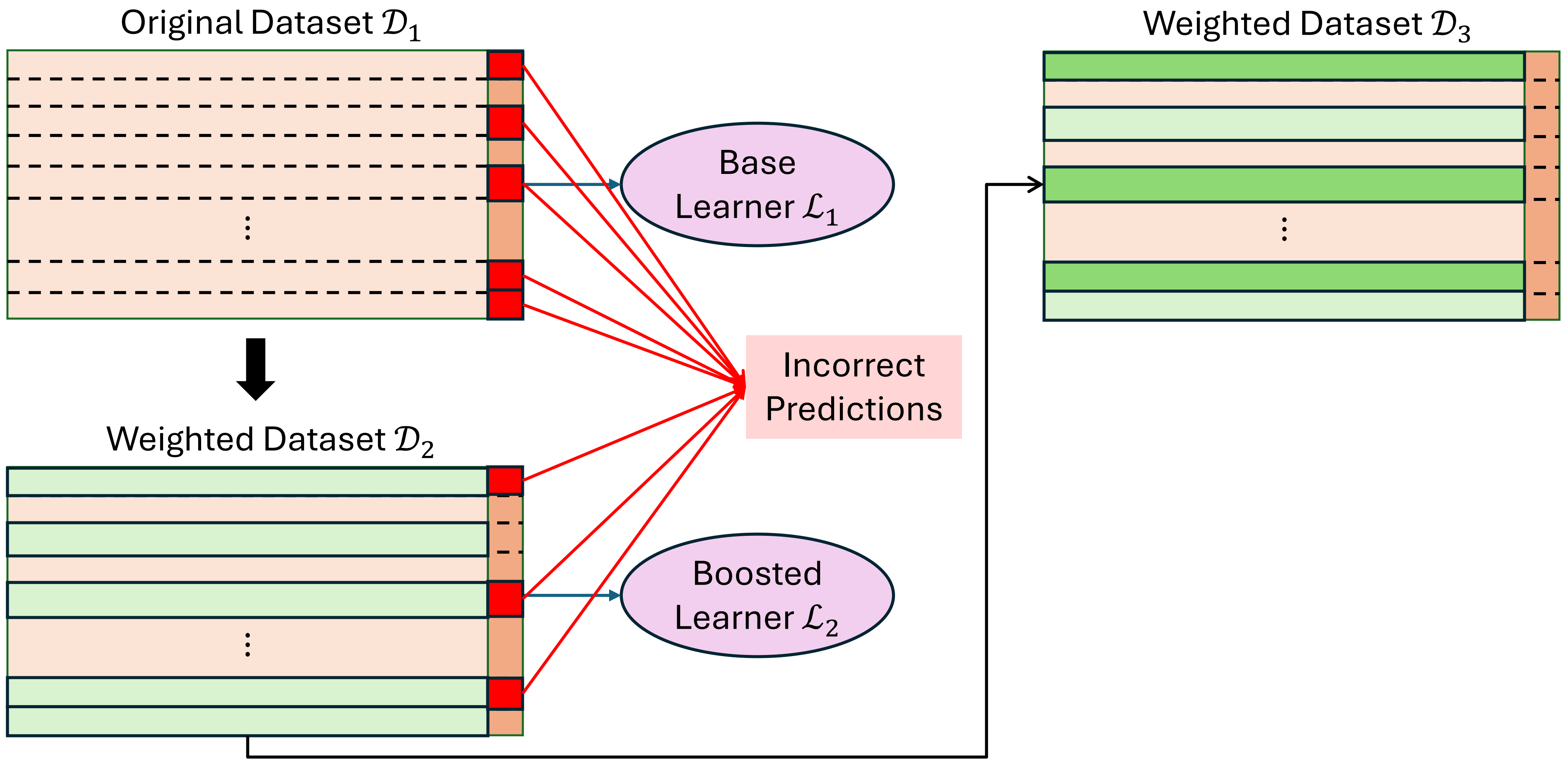

Original Dataset $\mathcal{D}_1$
Base Learner $\mathcal{L}_1$
Incorrect Predictions
Weighted Dataset $\mathcal{D}_2$
Boosted Learner $\mathcal{L}_2$

Illustration of Boosting



Original Dataset $\mathcal{D}_1$
Base Learner $\mathcal{L}_1$
Incorrect Predictions
Weighted Dataset $\mathcal{D}_2$
Boosted Learner $\mathcal{L}_2$
Weighted Dataset $\mathcal{D}_3$

Illustration of Boosting

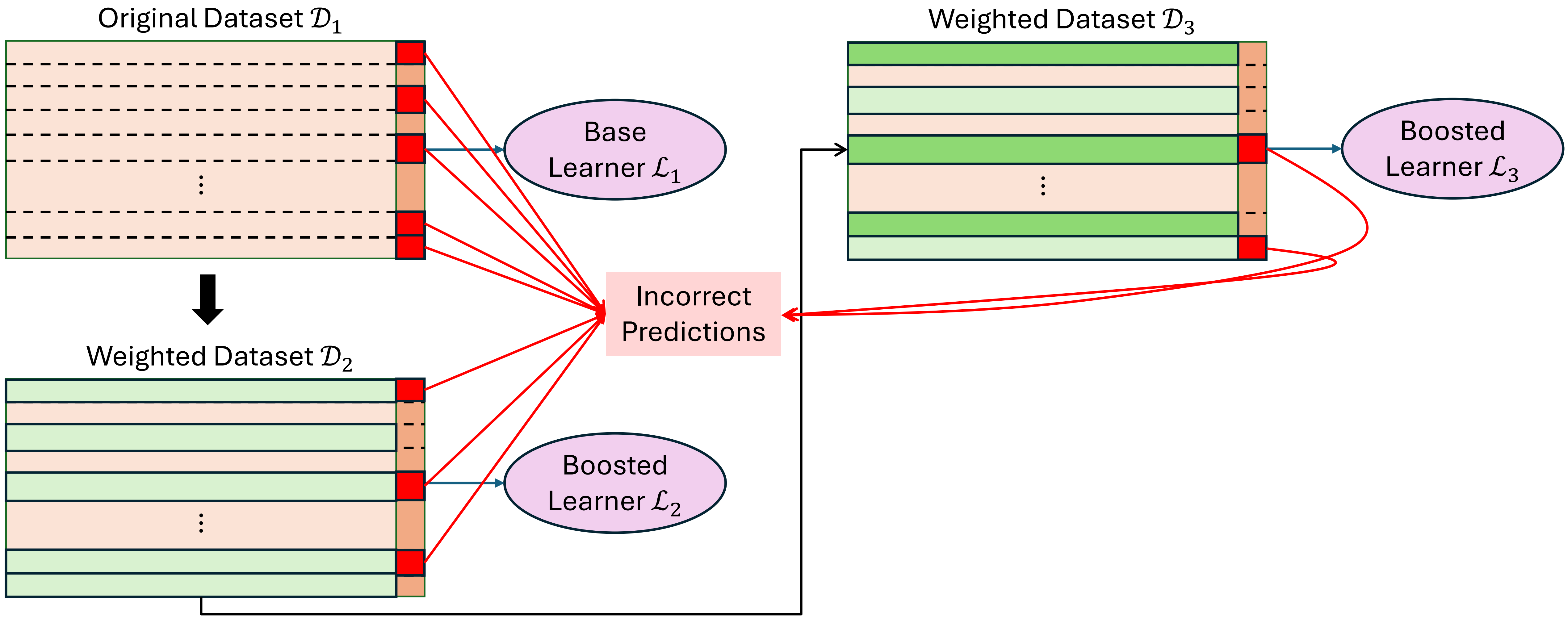

Original Dataset $\mathcal{D}_1$
Base Learner $\mathcal{L}_1$
Incorrect Predictions
Weighted Dataset $\mathcal{D}_2$
Boosted Learner $\mathcal{L}_2$
Weighted Dataset $\mathcal{D}_3$
Boosted Learner $\mathcal{L}_3$

Illustration of Boosting

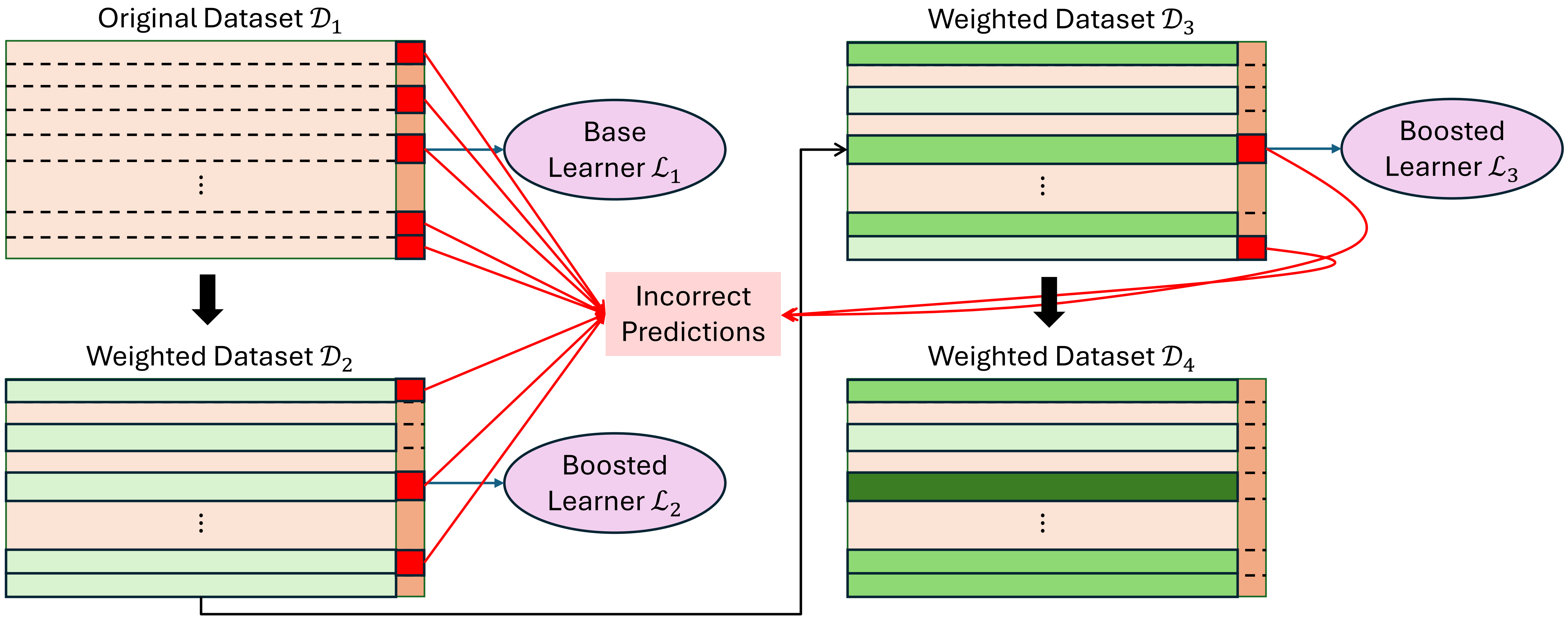

Original Dataset $\mathcal{D}_1$
Base Learner $\mathcal{L}_1$
Incorrect Predictions
Weighted Dataset $\mathcal{D}_2$
Boosted Learner $\mathcal{L}_2$
Weighted Dataset $\mathcal{D}_3$
Boosted Learner $\mathcal{L}_3$
Weighted Dataset $\mathcal{D}_4$

Illustration of Boosting

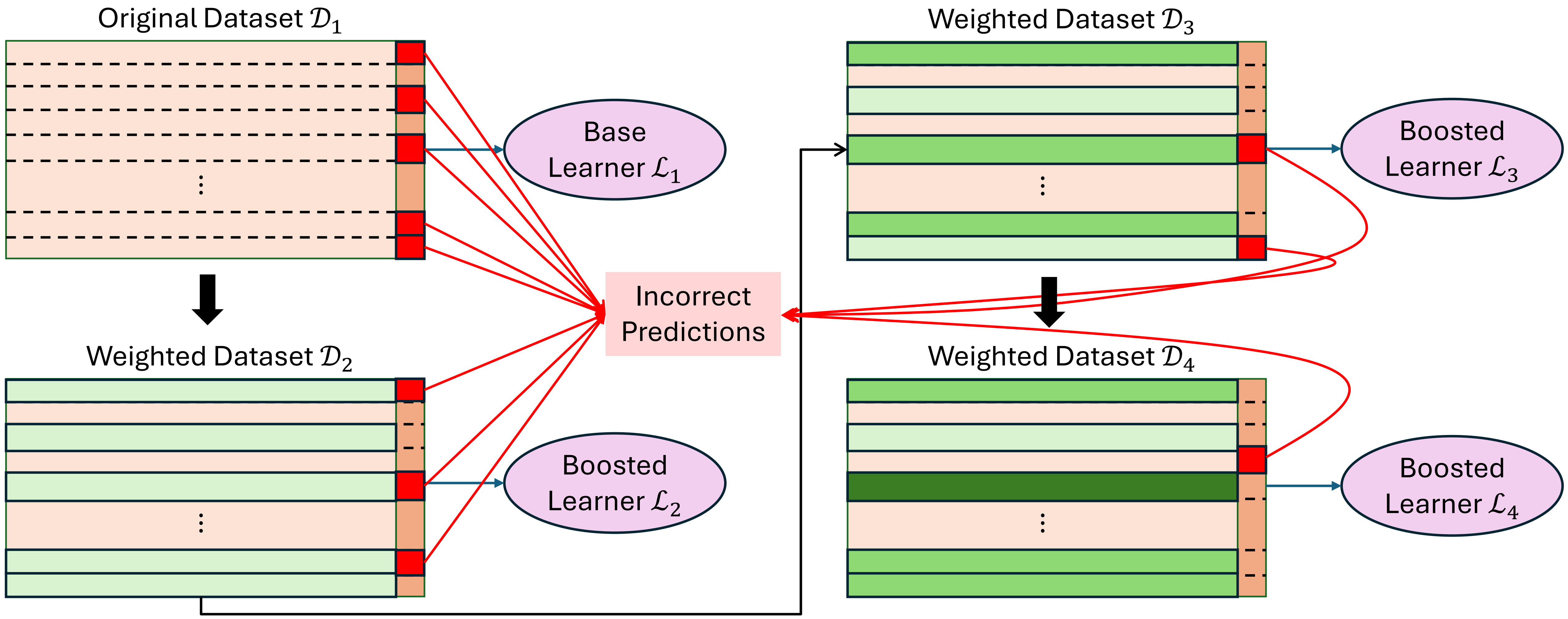
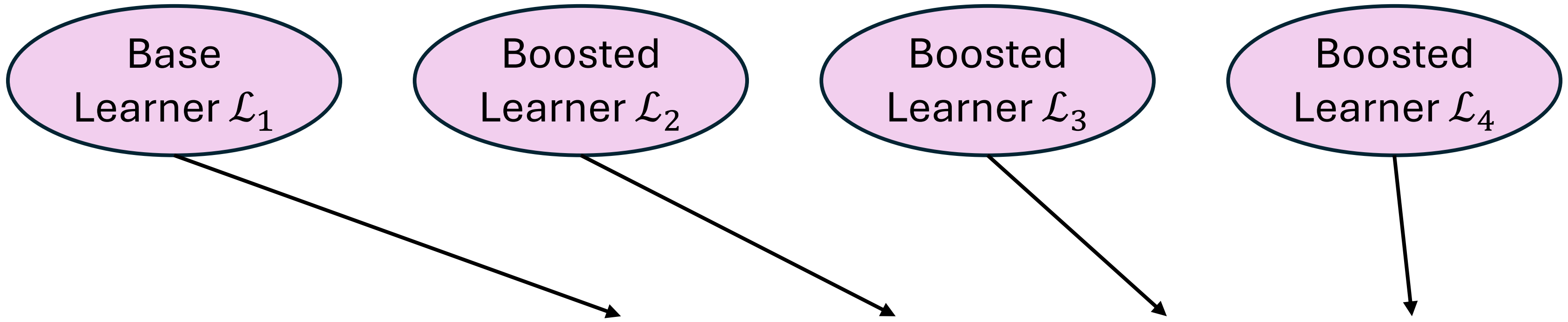

Original Dataset $\mathcal{D}_1$
Base Learner $\mathcal{L}_1$
Weighted Dataset $\mathcal{D}_2$
Boosted Learner $\mathcal{L}_2$
Incorrect Predictions
Weighted Dataset $\mathcal{D}_3$
Boosted Learner $\mathcal{L}_3$
Weighted Dataset $\mathcal{D}_4$
Boosted Learner $\mathcal{L}_4$

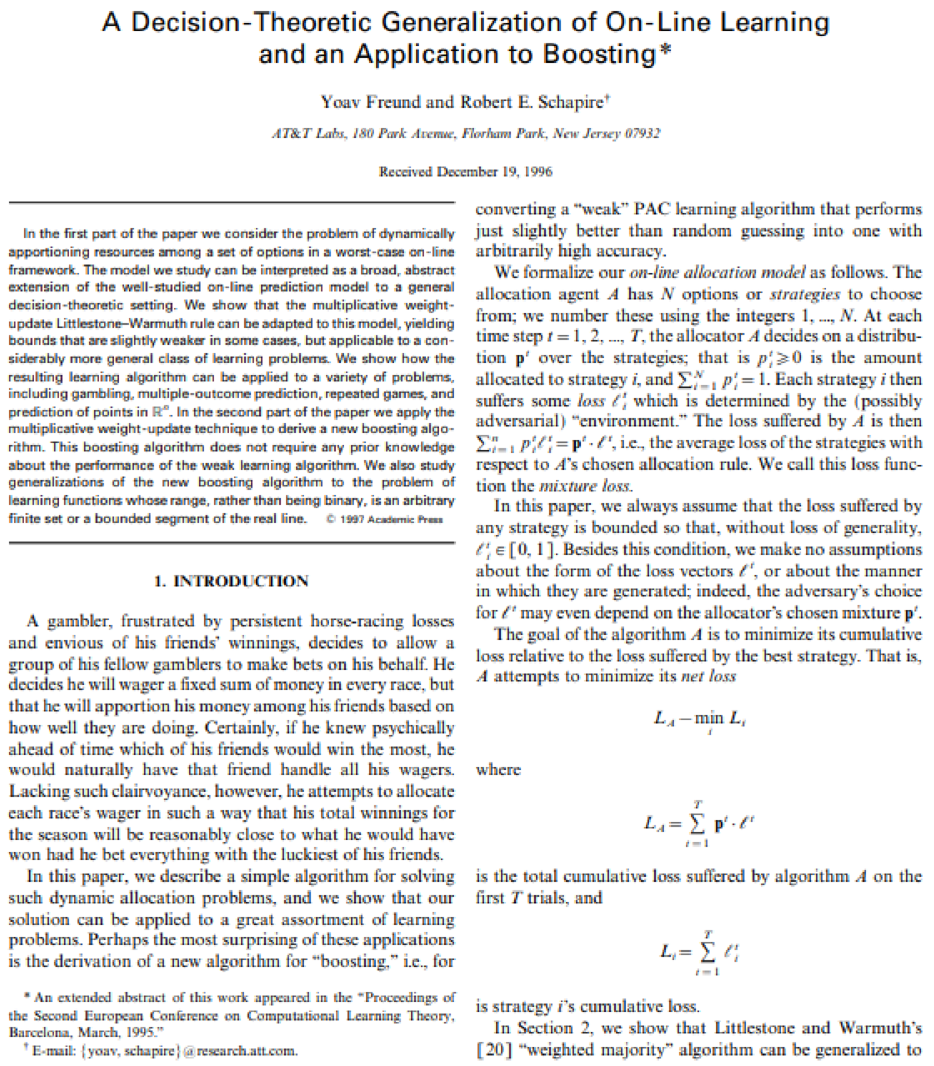
Illustration of Boosting


Base Learner $\mathcal{L}_1$
Boosted Learner $\mathcal{L}_2$
Boosted Learner $\mathcal{L}_3$
Boosted Learner $\mathcal{L}_4$

$$h_\theta(x; w, \mathcal{L}) \coloneqq \mathcal{A}_w(\{\mathcal{L}_1(x;\theta_1), \mathcal{L}_2(x;\theta_2), \mathcal{L}_3(x;\theta_3), \mathcal{L}_4(x;\theta_4)\})$$

(Final Boosted Model)

Adaptive Boosting



A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting*

Yoav Freund and Robert E. Schapire†

AT&T Labs, 180 Park Avenue, Florham Park, New Jersey 07932

Received December 19, 1996

In the first part of the paper we consider the problem of dynamically apportioning resources among a set of options in a worst-case on-line framework. The model we study can be interpreted as a broad, abstract extension of the well-studied on-line prediction model to a general decision-theoretic setting. We show that the multiplicative weight-update Littlestone–Warmuth rule can be adapted to this model, yielding bounds that are slightly weaker in some cases, but applicable to a considerably more general class of learning problems. We show how the resulting learning algorithm can be applied to a variety of problems, including gambling, multiple-outcome prediction, repeated games, and prediction of points in $\mathbb{R}^n$. In the second part of the paper we apply the multiplicative weight-update technique to derive a new boosting algorithm. This boosting algorithm does not require any prior knowledge about the performance of the weak learning algorithm. We also study generalizations of the new boosting algorithm to the problem of learning functions whose range, rather than being binary, is an arbitrary finite set or a bounded segment of the real line. © 1997 Academic Press

1. INTRODUCTION

A gambler, frustrated by persistent horse-racing losses and envious of his friends' winnings, decides to allow a group of his fellow gamblers to make bets on his behalf. He decides he will wager a fixed sum of money in every race, but that he will apportion his money among his friends based on how well they are doing. Certainly, if he knew psychically ahead of time which of his friends would win the most, he would naturally have that friend handle all his wagers. Lacking such clairvoyance, however, he attempts to allocate each race's wager in such a way that his total winnings for the season will be reasonably close to what he would have won had he bet everything with the luckiest of his friends.

In this paper, we describe a simple algorithm for solving such dynamic allocation problems, and we show that our solution can be applied to a great assortment of learning problems. Perhaps the most surprising of these applications is the derivation of a new algorithm for "boosting," i.e., for converting a "weak" PAC learning algorithm that performs just slightly better than random guessing into one with arbitrarily high accuracy.

We formalize our *on-line allocation model* as follows. The allocation agent A has N options or *strategies* to choose from; we number these using the integers $1, ..., N$. At each time step $t = 1, 2, ..., T$, the allocator A decides on a distribution $\mathbf{p}^t$ over the strategies; that is $p_i^t \geqslant 0$ is the amount allocated to strategy i, and $\sum_{i=1}^N p_i^t = 1$. Each strategy i then suffers some *loss* ℓ_i^t which is determined by the (possibly adversarial) "environment." The loss suffered by A is then $\sum_{i=1}^n p_i^t \ell_i^t = \mathbf{p}^t \cdot \boldsymbol{\ell}^t$, i.e., the average loss of the strategies with respect to A's chosen allocation rule. We call this loss function the *mixture loss*.

In this paper, we always assume that the loss suffered by any strategy is bounded so that, without loss of generality, $\ell_i^t \in [0, 1]$. Besides this condition, we make no assumptions about the form of the loss vectors $\boldsymbol{\ell}^t$, or about the manner in which they are generated; indeed, the adversary's choice for $\boldsymbol{\ell}^t$ may even depend on the allocator's chosen mixture $\mathbf{p}^t$.

The goal of the algorithm A is to minimize its cumulative loss relative to the loss suffered by the best strategy. That is, A attempts to minimize its *net loss*

$$L_A - \min_i L_i$$

where

$$L_A = \sum_{t=1}^T \mathbf{p}^t \cdot \boldsymbol{\ell}^t$$

is the total cumulative loss suffered by algorithm A on the first T trials, and

$$L_i = \sum_{t=1}^T \ell_i^t$$

is strategy i's cumulative loss.

In Section 2, we show that Littlestone and Warmuth's [20] "weighted majority" algorithm can be generalized to

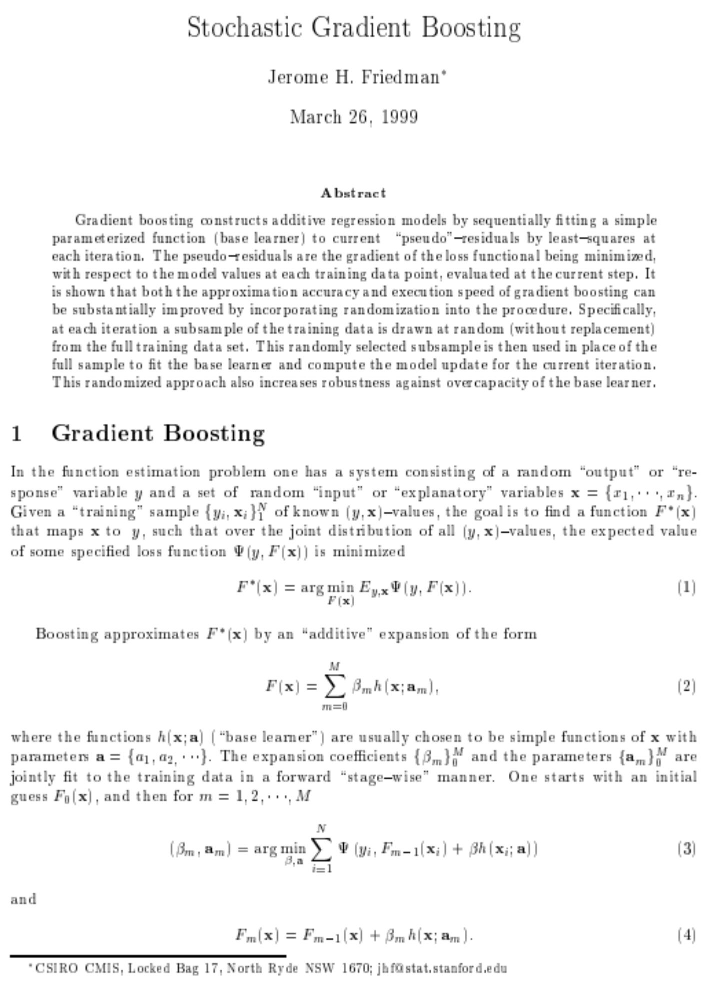
* An extended abstract of this work appeared in the "Proceedings of the Second European Conference on Computational Learning Theory, Barcelona, March, 1995."

† E-mail: {yoav, schapire}@research.att.com.

Adaptive Boosting (AdaBoost)

AdaBoost is the original direct implementation of the general boosting methodology to tree-based models. The models that are used as iterative learners are "**simple**" (or weak) decision trees (these are **very shallow decision trees**, typically with only **one layer** deep). Each new tree is built on a modified version of the original dataset where the misclassified datapoints of the previous tree are granted more weight in the loss function than the correctly classified points. Lastly, when "voting" on a new prediction, the outputs of each tree are aggregated by weighting each according to its accuracy.

Gradient Boosting



Stochastic Gradient Boosting

Jerome H. Friedman*

March 26, 1999

Abstract

Gradient boosting constructs additive regression models by sequentially fitting a simple parameterized function (base learner) to current "pseudo"–residuals by least–squares at each iteration. The pseudo–residuals are the gradient of the loss functional being minimized, with respect to the model values at each training data point, evaluated at the current step. It is shown that both the approximation accuracy and execution speed of gradient boosting can be substantially improved by incorporating randomization into the procedure. Specifically, at each iteration a subsample of the training data is drawn at random (without replacement) from the full training data set. This randomly selected subsample is then used in place of the full sample to fit the base learner and compute the model update for the current iteration. This randomized approach also increases robustness against overcapacity of the base learner.

1 Gradient Boosting

In the function estimation problem one has a system consisting of a random "output" or "response" variable y and a set of random "input" or "explanatory" variables $\mathbf{x} = \{x_1, \cdots, x_n\}$. Given a "training" sample $\{y_i, \mathbf{x}_i\}_1^N$ of known $(y, \mathbf{x})$–values, the goal is to find a function $F^*(\mathbf{x})$ that maps $\mathbf{x}$ to y, such that over the joint distribution of all $(y, \mathbf{x})$–values, the expected value of some specified loss function $\Psi(y, F(\mathbf{x}))$ is minimized

$$F^*(\mathbf{x}) = \arg\min_{F(\mathbf{x})} E_{y,\mathbf{x}} \Psi(y, F(\mathbf{x})). \tag{1}$$

Boosting approximates $F^*(\mathbf{x})$ by an "additive" expansion of the form

$$F(\mathbf{x}) = \sum_{m=0}^{M} \beta_m h(\mathbf{x}; \mathbf{a}_m), \tag{2}$$

where the functions $h(\mathbf{x}; \mathbf{a})$ ("base learner") are usually chosen to be simple functions of $\mathbf{x}$ with parameters $\mathbf{a} = \{a_1, a_2, \cdots\}$. The expansion coefficients $\{\beta_m\}_0^M$ and the parameters $\{\mathbf{a}_m\}_0^M$ are jointly fit to the training data in a forward "stage–wise" manner. One starts with an initial guess $F_0(\mathbf{x})$, and then for $m = 1, 2, \cdots, M$

$$(\beta_m, \mathbf{a}_m) = \arg\min_{\beta, \mathbf{a}} \sum_{i=1}^{N} \Psi\left(y_i, F_{m-1}(\mathbf{x}_i) + \beta h(\mathbf{x}_i; \mathbf{a})\right) \tag{3}$$

and

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \beta_m h(\mathbf{x}; \mathbf{a}_m). \tag{4}$$

*CSIRO CMIS, Locked Bag 17, North Ryde NSW 1670; jhf@stat.stanford.edu

Gradient Boosting

The method of gradient boosting is very similar to AdaBoost, but with a few caveats.

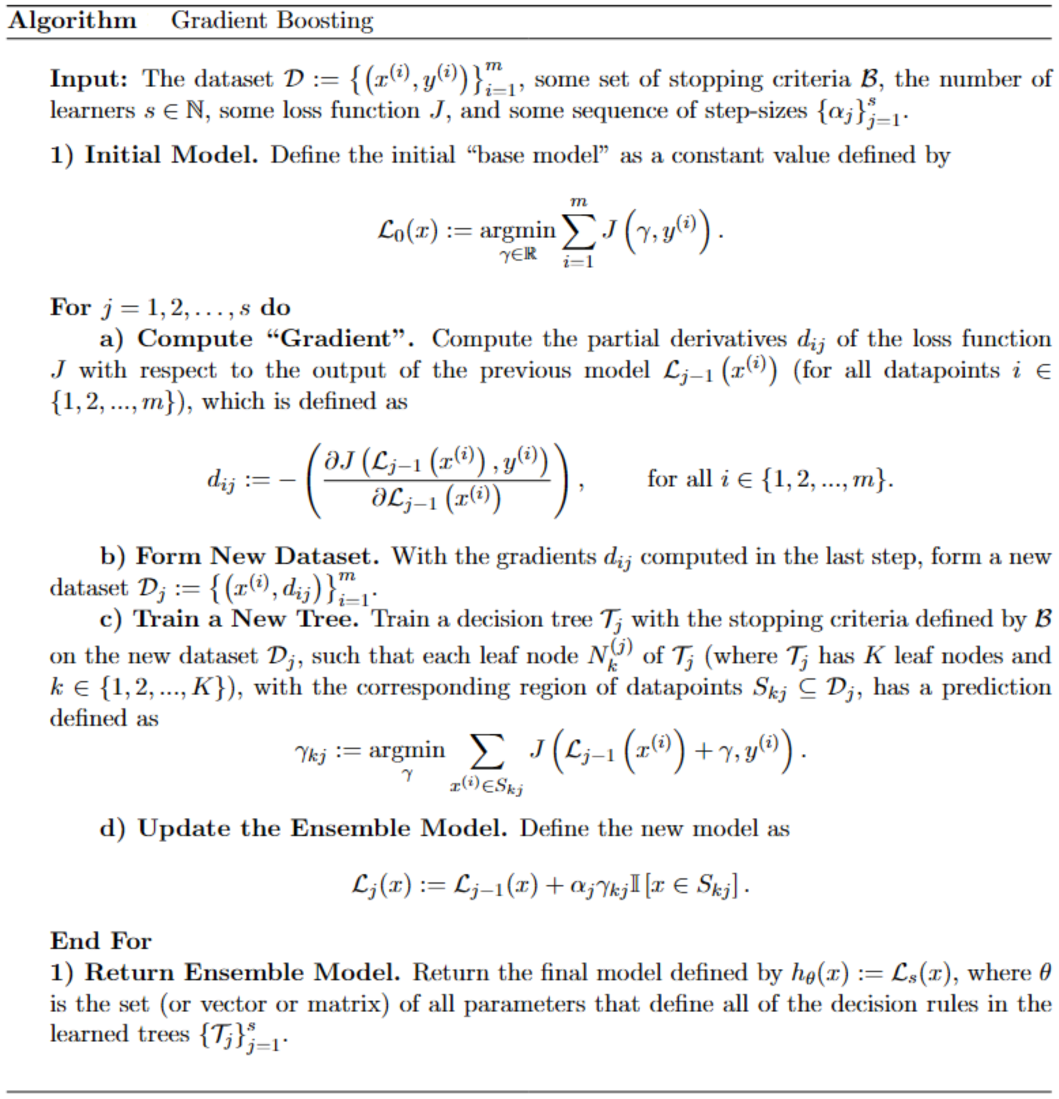
- The initial learner $\mathcal{L}_0$ is simply the most likely value the target feature will take on (either the mean or the mode depending on regression or classification).
- The individual decision trees that are sequentially learned are deeper than those used in AdaBoost.
- The sequential trees are built to predict the residuals (or the gradients) of the outputs of the previous trees instead of the target feature itself.
- The predictions of all $s + 1$ models will be combined with a weighted sum, where the weights $\{\alpha_j\}_{j=1}^{s}$ can be thought of as step sizes.
- In this way, one can view each consecutive tree that is trained and added to the total sum as applying a single iteration of gradient descent.

Extreme Gradient Boosting (XGBoost)

XGBoost is the most common implementation of gradient boosting as it is essentially just a computationally optimized and enhanced version of the gradient boosting method.

- XGBoost includes regularization, parallelization, sophisticated pruning, and is overall a more sophisticated and efficient method.
- XGBoost is typically the most dominant algorithm in ML competitions, typically only rivaled by deep neural networks in more specialized tasks.

Gradient Boosting Algorithm



Algorithm Gradient Boosting

Input: The dataset $\mathcal{D} := \left\{ \left(x^{(i)}, y^{(i)}\right) \right\}_{i=1}^{m}$, some set of stopping criteria $\mathcal{B}$, the number of learners $s \in \mathbb{N}$, some loss function J, and some sequence of step-sizes $\{\alpha_j\}_{j=1}^{s}$.

1) Initial Model. Define the initial "base model" as a constant value defined by

$$\mathcal{L}_0(x) := \operatorname*{argmin}_{\gamma \in \mathbb{R}} \sum_{i=1}^{m} J\left(\gamma, y^{(i)}\right).$$

For $j = 1, 2, \ldots, s$ **do**

a) Compute "Gradient". Compute the partial derivatives d_{ij} of the loss function J with respect to the output of the previous model $\mathcal{L}_{j-1}\left(x^{(i)}\right)$ (for all datapoints $i \in \{1, 2, ..., m\}$), which is defined as

$$d_{ij} := -\left(\frac{\partial J\left(\mathcal{L}_{j-1}\left(x^{(i)}\right), y^{(i)}\right)}{\partial \mathcal{L}_{j-1}\left(x^{(i)}\right)} \right), \qquad \text{for all } i \in \{1, 2, ..., m\}.$$

b) Form New Dataset. With the gradients d_{ij} computed in the last step, form a new dataset $\mathcal{D}_j := \left\{ \left(x^{(i)}, d_{ij}\right) \right\}_{i=1}^{m}$.

c) Train a New Tree. Train a decision tree $\mathcal{T}_j$ with the stopping criteria defined by $\mathcal{B}$ on the new dataset $\mathcal{D}_j$, such that each leaf node $N_k^{(j)}$ of $\mathcal{T}_j$ (where $\mathcal{T}_j$ has K leaf nodes and $k \in \{1, 2, ..., K\}$), with the corresponding region of datapoints $S_{kj} \subseteq \mathcal{D}_j$, has a prediction defined as

$$\gamma_{kj} := \operatorname*{argmin}_{\gamma} \sum_{x^{(i)} \in S_{kj}} J\left(\mathcal{L}_{j-1}\left(x^{(i)}\right) + \gamma, y^{(i)}\right).$$

d) Update the Ensemble Model. Define the new model as

$$\mathcal{L}_j(x) := \mathcal{L}_{j-1}(x) + \alpha_j \gamma_{kj} \mathbb{I}\left[x \in S_{kj}\right].$$

End For

1) Return Ensemble Model. Return the final model defined by $h_\theta(x) := \mathcal{L}_s(x)$, where θ is the set (or vector or matrix) of all parameters that define all of the decision rules in the learned trees $\{\mathcal{T}_j\}_{j=1}^{s}$.

Stacking

Neural Networks, Vol. 5, pp. 241–259, 1992
Printed in the USA. All rights reserved.

0893-6080/92 $5.00 + .00
Copyright © 1992 Pergamon Press Ltd.

ORIGINAL CONTRIBUTION

Stacked Generalization

DAVID H. WOLPERT

Complex Systems Group, Theoretical Division, and Center for Non-linear Studies

(*Received* 16 *January* 1991; *revised and accepted* 29 *July* 1991)

Abstract—*This paper introduces stacked generalization, a scheme for minimizing the generalization error rate of one or more generalizers. Stacked generalization works by deducing the biases of the generalizer(s) with respect to a provided learning set. This deduction proceeds by generalizing in a second space whose inputs are (for example) the guesses of the original generalizers when taught with part of the learning set and trying to guess the rest of it, and whose output is (for example) the correct guess. When used with multiple generalizers, stacked generalization can be seen as a more sophisticated version of cross-validation, exploiting a strategy more sophisticated than cross-validation's crude winner-takes-all for combining the individual generalizers. When used with a single generalizer, stacked generalization is a scheme for estimating (and then correcting for) the error of a generalizer which has been trained on a particular learning set and then asked a particular question. After introducing stacked generalization and justifying its use, this paper presents two numerical experiments. The first demonstrates how stacked generalization improves upon a set of separate generalizers for the NETtalk task of translating text to phonemes. The second demonstrates how stacked generalization improves the performance of a single surface-fitter. With the other experimental evidence in the literature, the usual arguments supporting cross-validation, and the abstract justifications presented in this paper, the conclusion is that for almost any real-world generalization problem one should use some version of stacked generalization to minimize the generalization error rate. This paper ends by discussing some of the variations of stacked generalization, and how it touches on other fields like chaos theory.*

Keywords—Generalization and induction, Combining generalizers, Learning set preprocessing, cross-validation, Error estimation and correction.

1. INTRODUCTION

This paper concerns the problem of inferring a function from a subset of $\mathbf{R}^n$ to a subset of $\mathbf{R}^p$ (the *parent function*) given a set of m samples of that function (the *learning set*). The subset of $\mathbf{R}^n$ is the *input space*, and the subset of $\mathbf{R}^p$ is the *output space*. A *question* is an input space (vector) value. An algorithm which guesses a parent function, basing the guess only on a learning set of m $\mathbf{R}^{n+p}$ vectors read off of that parent function, is called a *generalizer*. A generalizer guesses an appropriate output for a question via the parent function it infers from the learning set. For simplicity, although the analysis of this paper holds for any positive integer p, unless explicitly indicated otherwise I will always take $p = 1$.

In this paper I am usually assuming noiseless data. This means that the best guesses for the inputs of elements of the learning set are already known to us—they are provided by the learning set. Building a system to guess properly for those elements (i.e., "learning that learning set") is trivial, and can be achieved simply by building a look-up table. (Difficulties only arise when one insists that the look-up table be implemented in an odd way (e.g., as a feed-forward neural net).) Therefore, in this paper the questions of interest are almost always outside of the learning set.

Some examples of generalizers are back-propagated neural nets (Rumelhart & McClelland, 1986), Holland's (1975) classifier system, and Rissanen's (1986) minimum description length principle (which, along with all other schemes which attempt to exploit Occam's razor, is analyzed in (Wolpert, 1990a)). Other important examples are memory-based reasoning schemes (Stanfill & Waltz, 1986), regularization theory (Poggio et al., 1988), and similar schemes for overt surface fitting of a parent function to the learning set (Farmer & Sidorowich, 1988; Omohundro, 1987; Wolpert, 1989; Wolpert 1990a; Wolpert, 1990b).

This work was performed under the auspices of the Department of Energy.

Requests for reprints should be sent to David H. Wolpert, Complex Systems Group, Theoretical Division, and Center for Non-linear Studies, MS B213, LANL, Los Alamos, NM 87545.

Stacked Ensembles

The idea of stacking models is something that all the algorithms in this series of lecture slides on ensemble methods is doing in some form or another. In its simplest form, stacking models are simply some form of aggregation of the predictions of other trained models. In this way, given a series of different learned models $\{\mathcal{L}_j\}_{j=1}^{S}$ on some dataset $\mathcal{D}$, a stacked model can simply be defined as

$$h_\theta(x;\mathcal{L}) \coloneqq \mathcal{A}\left(\left\{\mathcal{L}_j(x;\theta_j)\right\}_{j=1}^{S}\right),$$

where θ_j is a vector of parameters for $\mathcal{L}_j$ and $\mathcal{A}$ is some aggregation function. The only difference between this and the models we have discussed earlier is that stacking models are general, and one can choose any type of models for the learners $\mathcal{L}_j$.

- For example, one could choose to stack the outputs of a linear regression model, a polynomial regression model, a K-NN regression model, a random forest, and a neural network, all by computing a weighted sum of their outputs.

References

- https://keeeto.github.io/ebook-data-analysis/lecture-5-decision-trees-classification.html