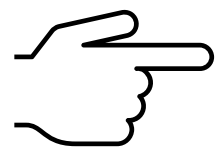


Introduction to Statistical Learning with Applications

CM8: Ensemble methods

Pedro L. C. Rodrigues



- Recap
- Ensemble methods
- Bagging
- Boosting

Discriminative approaches

There are many strategies for approximating the Bayes classifier.

$$f(\mathbf{x}_0) = \operatorname{argmax}_k \operatorname{Prob}(Y = k \mid \mathbf{X} \in \mathcal{B}_\varepsilon(\mathbf{x}_0))$$

In this course we will focus on:

- K-nearest neighbours (kNN)
- Logistic regression
- Linear and quadratic discriminant analysis
- Naïve Bayes classifier
- Random forests
- Gradient boosted trees

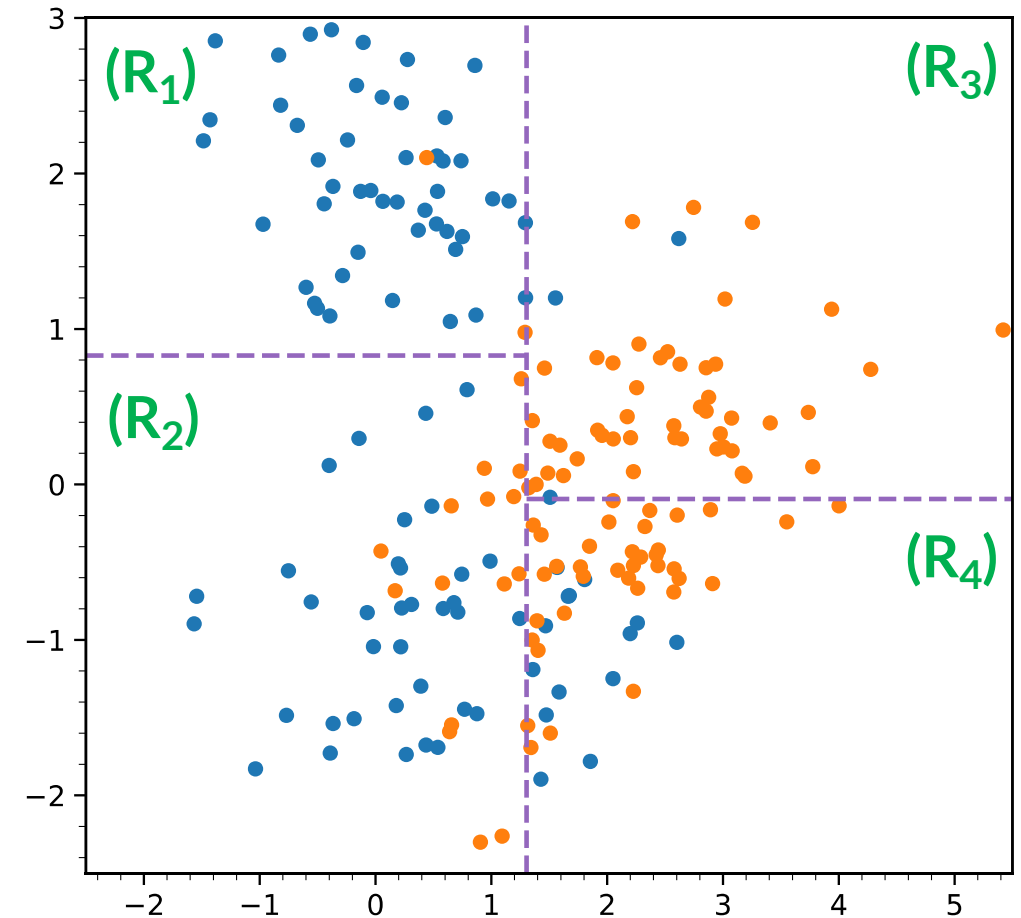
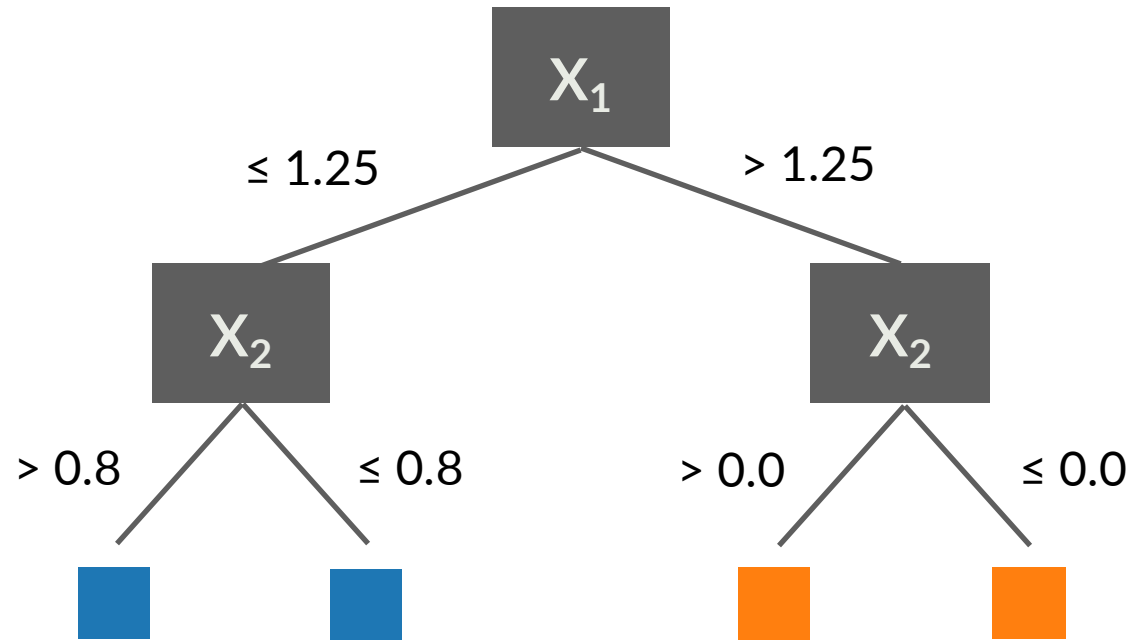
Discriminative approaches

Generative approaches

Ensembling approaches

Decision trees

Decision trees split the space according to **thresholds** learnt from the data



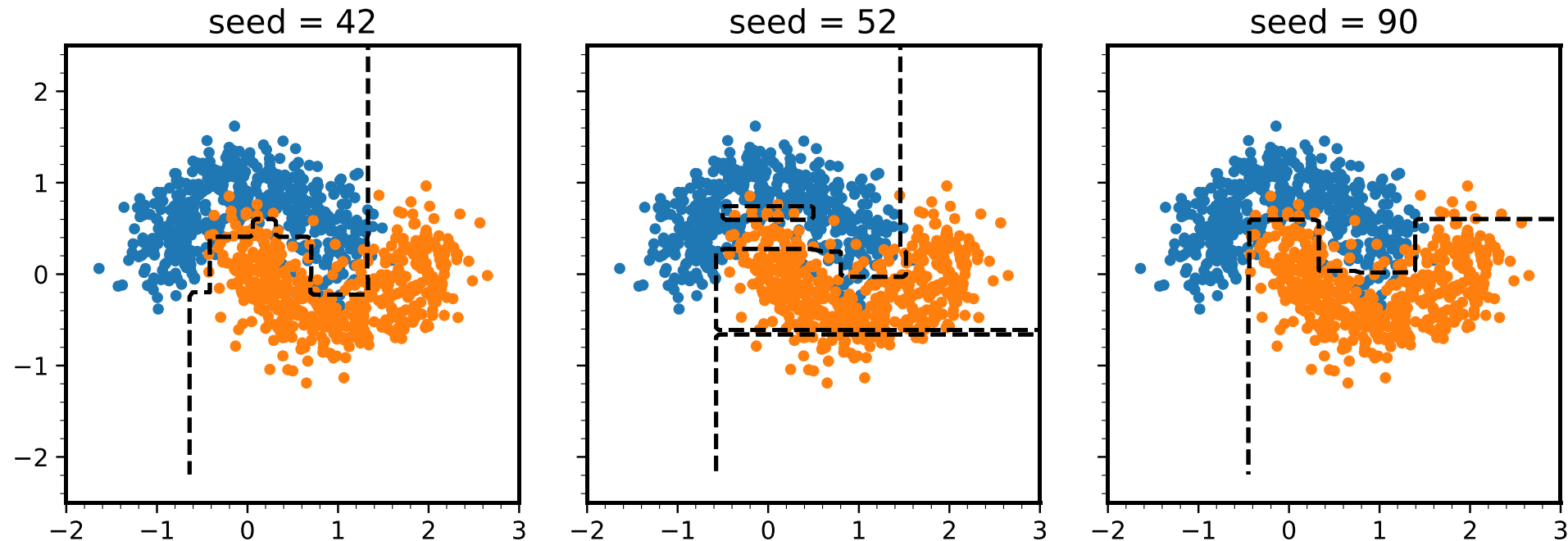
They are very **fast to build** so it is not too expensive to make several of them

Decision trees

Consider the two moons dataset.

Simulate a training dataset three times with 200 points and one test set with 1000.

The scatter dots are points from the test set and the boundary is for a **decision tree**

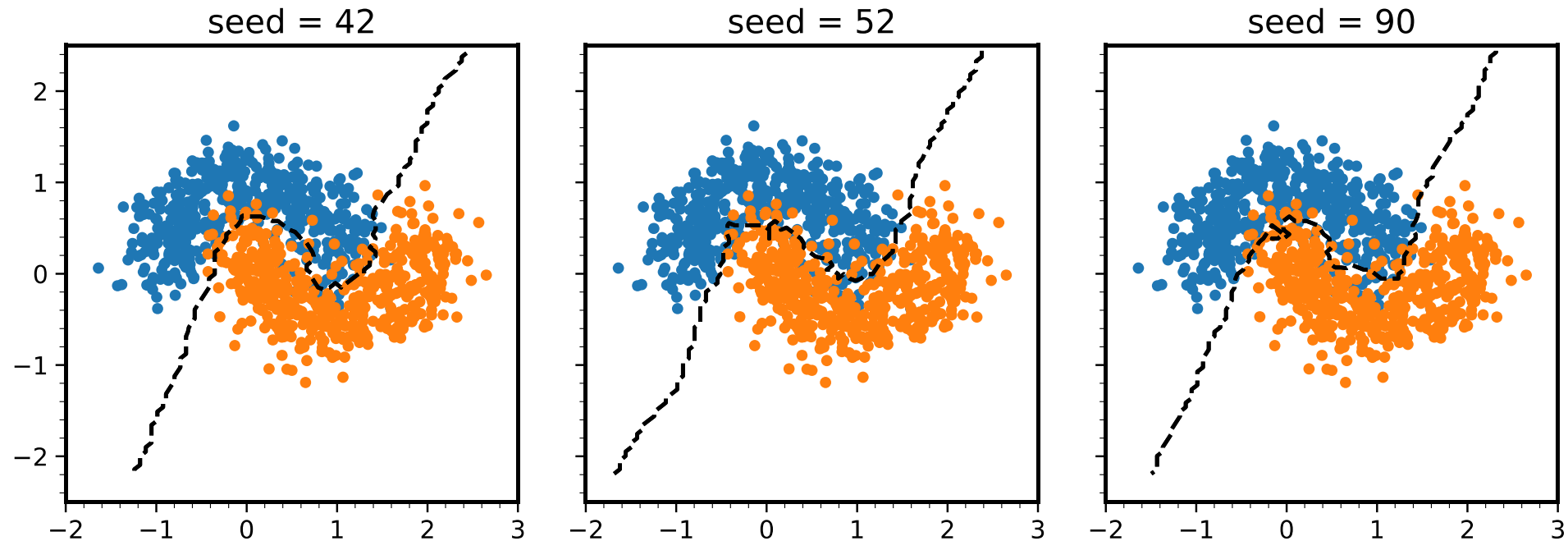


Decision trees

Consider the two moons dataset.

Simulate a training dataset three times with 200 points and one test set with 1000.

The scatter dots are points from the test set and the boundary is for a **kNN (k=10)**

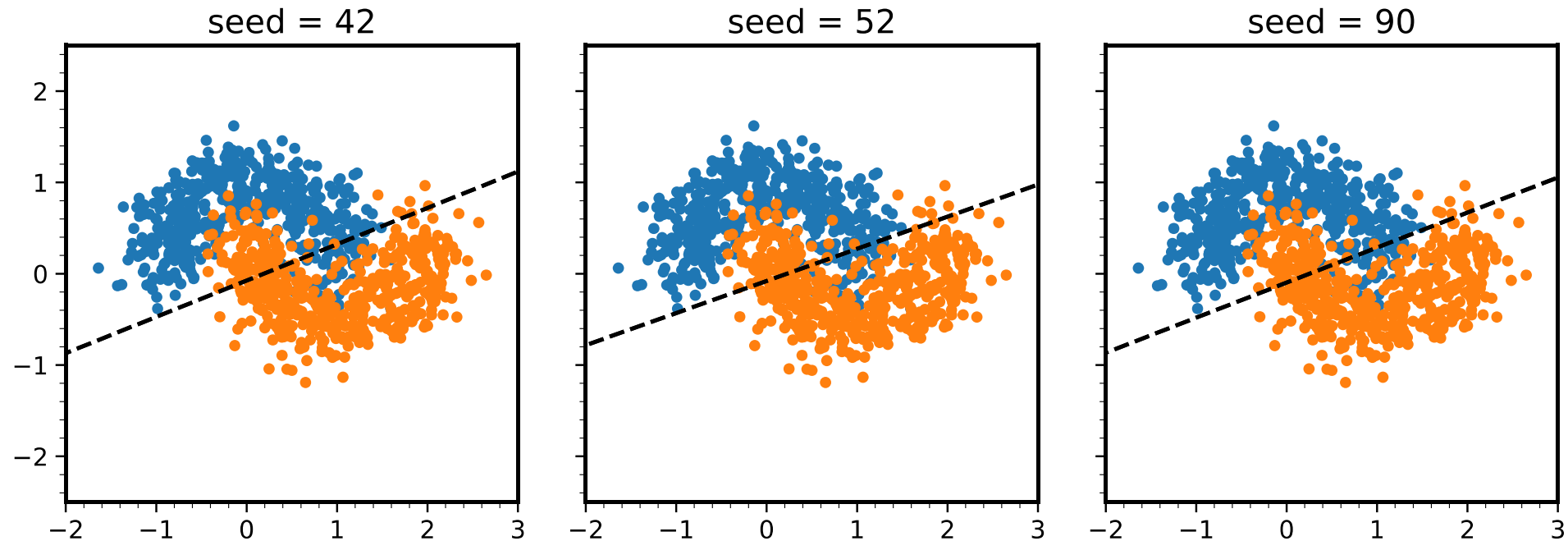


Decision trees

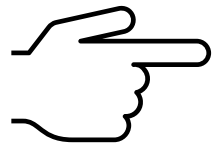
Consider the two moons dataset.

Simulate a training dataset three times with 200 points and one test set with 1000.

The scatter dots are points from the test set and the boundary is for a **logistic reg**



- Recap



- Ensemble methods

- Bagging

- Boosting

Ensemble methods

Consider the following example taken from Freund and Shapire (1999)

- A horse-racing gambler wants to maximize his winnings and creates a computer program that will accurately predict the winner of a horse race based on the usual information (number of races recently won by each horse, betting odds for each horse, etc.)

Ensemble methods

Consider the following example taken from Freund and Shapire (1999)

- A horse-racing gambler wants to maximize his winnings and creates a computer program that will accurately predict the winner of a horse race based on the usual information (number of races recently won by each horse, betting odds for each horse, etc.)
- To create such a program, he asks a highly successful expert gambler to explain his betting strategy. However, the expert is unable to articulate a grand set of rules for selecting a horse.

Ensemble methods

Consider the following example taken from Freund and Shapire (1999)

- A horse-racing gambler wants to maximize his winnings and creates a computer program that will accurately predict the winner of a horse race based on the usual information (number of races recently won by each horse, betting odds for each horse, etc.)
- To create such a program, he asks a highly successful expert gambler to explain his betting strategy. However, the expert is unable to articulate a grand set of rules for selecting a horse.
- On the other hand, when presented with the data for a specific set of races, the expert has no trouble coming up with a “rule of thumb” for that set of races (such as, “Bet on the horse that has recently won the most races” or “Bet on the horse with the most favoured odds”).

Ensemble methods

Consider the following example taken from Freund and Shapire (1999)

- A horse-racing gambler wants to maximize his winnings and creates a computer program that will accurately predict the winner of a horse race based on the usual information (number of races recently won by each horse, betting odds for each horse, etc.)
- To create such a program, he asks a highly successful expert gambler to explain his betting strategy. However, the expert is unable to articulate a grand set of rules for selecting a horse.
- On the other hand, when presented with the data for a specific set of races, the expert has no trouble coming up with a “rule of thumb” for that set of races (such as, “Bet on the horse that has recently won the most races” or “Bet on the horse with the most favoured odds”).
- Although such a rule of thumb, by itself, is obviously very rough and inaccurate, it is not unreasonable to expect it to provide predictions that are at least a little bit better than random guessing. Furthermore, by repeatedly asking the expert’s opinion on different collections of races, the gambler is able to extract many rules of thumb.

Ensemble methods

- To use these rules of thumb, there are two problems faced by the gambler:
 1. First, how should he choose the collections of races presented to the expert so as to extract rules of thumb from the expert that will be the most useful?
 2. Second, once he has collected many rules of thumb, how can they be combined into a single, highly accurate prediction rule?

The answer to these questions is of fundamental importance in machine learning and the category of algorithms that try to address them are called ensemble learning methods. Some examples of strategies are

VOTING

STACKING

BOOSTING

BAGGING

BLENDING

Etc.

Ensemble methods

Remember that the generalization error can be decomposed as

$$\text{ERROR} = \text{NOISE} + \text{BIAS} + \text{VARIANCE}$$

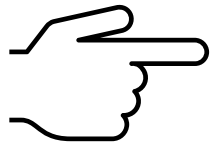
Ensemble methods reduce the bias and/or variance using several predictors

$$f(\mathbf{x}) = \sum_{k=1}^M \alpha_k f_k(\mathbf{x}) \quad \Rightarrow \quad c(\mathbf{x}) = \text{sign}\left(f(\mathbf{x})\right) \in \{-1, +1\}$$

Different techniques will have different strategies for choosing the $\{\alpha_k, f_k\}$

- The predictors are trained on different versions of the training dataset
(chosen wisely but randomly)
- In **bagging** the weights are fixed and **boosting** the α_k follow a scheduling strategy

- Recap
- Ensemble methods
- Bagging
- Boosting



Bagging

Note that the variance of a classifier among different training sets is

$$\text{Var}(f) = \mathbb{E}_{\mathcal{D}} \left[\mathbb{E}_X \left[\left(f_d(X) - \mathbb{E}_X [f_d(X)] \right)^2 \middle| \mathcal{D} = d \right] \right]$$

Note that we could try to reduce the variance by proposing a **meta-classifier** given by

$$\tilde{f}(\mathbf{x}) = \frac{1}{M} \sum_{k=1}^M f_{\mathcal{D}_k}(\mathbf{x}) \quad \Rightarrow \quad \text{Var}(\tilde{f}) \leq \text{Var}(f)$$

where the \mathcal{D}_k are IID realizations of the distribution generating the dataset

Question:

How can we do this in practice if we have a fixed training dataset?

↪ Splitting the dataset would yield IID sub-datasets, but would not work well -- Why ?

Bagging

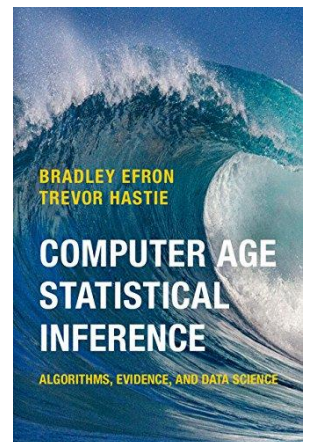
Consider the method of **cross-validation**:

- We want to estimate the test error of a classifier.
- Take **random subsets** for training, and another random set for testing.
- To get a more reliable result, we **repeat** several times and take the average.

In (computational) statistics there are many methods that follow the same principle, such as jackknife, bootstrap, conformal prediction, boosting, etc.

“Computer Age Statistical Inference”
Efron and Hastie

<https://hastie.su.domains/CASI/>



Bagging

The [Bootstrap](#) method:

- ▶ Assume you have a sample $X_1, \dots, X_n \sim p_\theta(x)$ and obtain an estimate

$$\hat{\Theta} = f(X_1, \dots, X_n)$$

You would like to know the statistical distribution of this estimate. For example, because you want to construct confidence intervals or test its statistical significance.

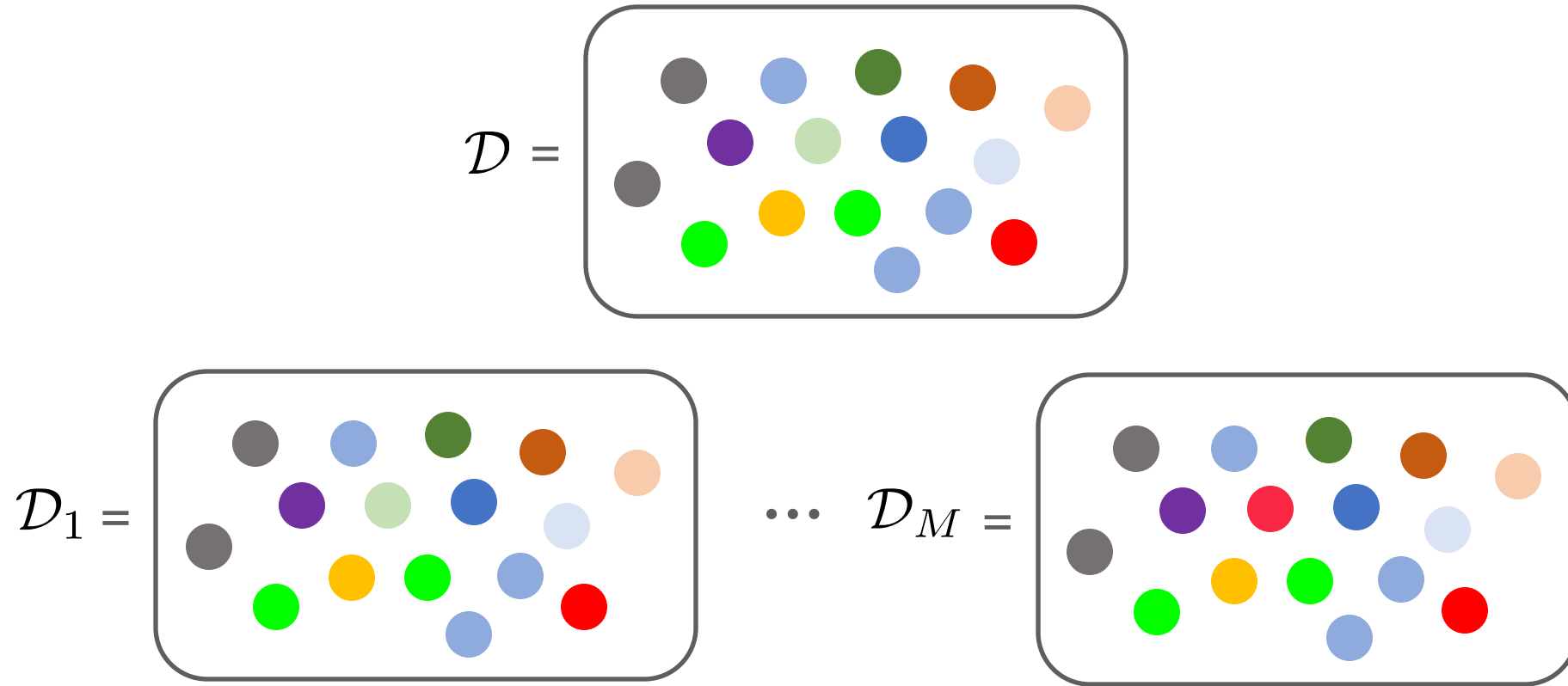
- ▶ You now draw a subsample of m points of the original sample (with or without replacement) and on this new sample you compute a new estimate of the parameter

$$\hat{\Theta}_i = f(X_1^{(i)}, \dots, X_m^{(i)})$$

- ▶ You repeat this procedure B times to obtain $\hat{\Theta}_1, \dots, \hat{\Theta}_B$ and construct statistics

Bagging

Bagging stands for “Bootstrap aggregating” and reduces **variance** of the predictions



The classifier $\tilde{f}(\mathbf{x}) = \frac{1}{M} \sum_{k=1}^M f_{\mathcal{D}_k}(\mathbf{x})$ will reduce the variance, but **slower** than IID case
(Why?)

Bagging

But how do we choose a **base classifier**?

- Since this is an expensive mechanism from a computational point of view, it makes sense to **use a reasonably simple baseline** algorithm for training
- Bagging reduces more the variance if there is **none or very little correlation** between the individual classifiers. This is what we need.
- If the classifier have a strong bias, then bagging cannot do anything about it

A standard choice is decision trees + **aggregate to make a forest**.

- ▶ Each tree is constructed randomly: on a random sample from the training dataset and by selecting the next splitting dimension from a random subset of all dimensions

Random forests

The algorithm for building a **random forest** is therefore:

For $b = 1$ to B

Draw a bootstrap sample \mathcal{D}_b of size N from the training dataset

Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating:

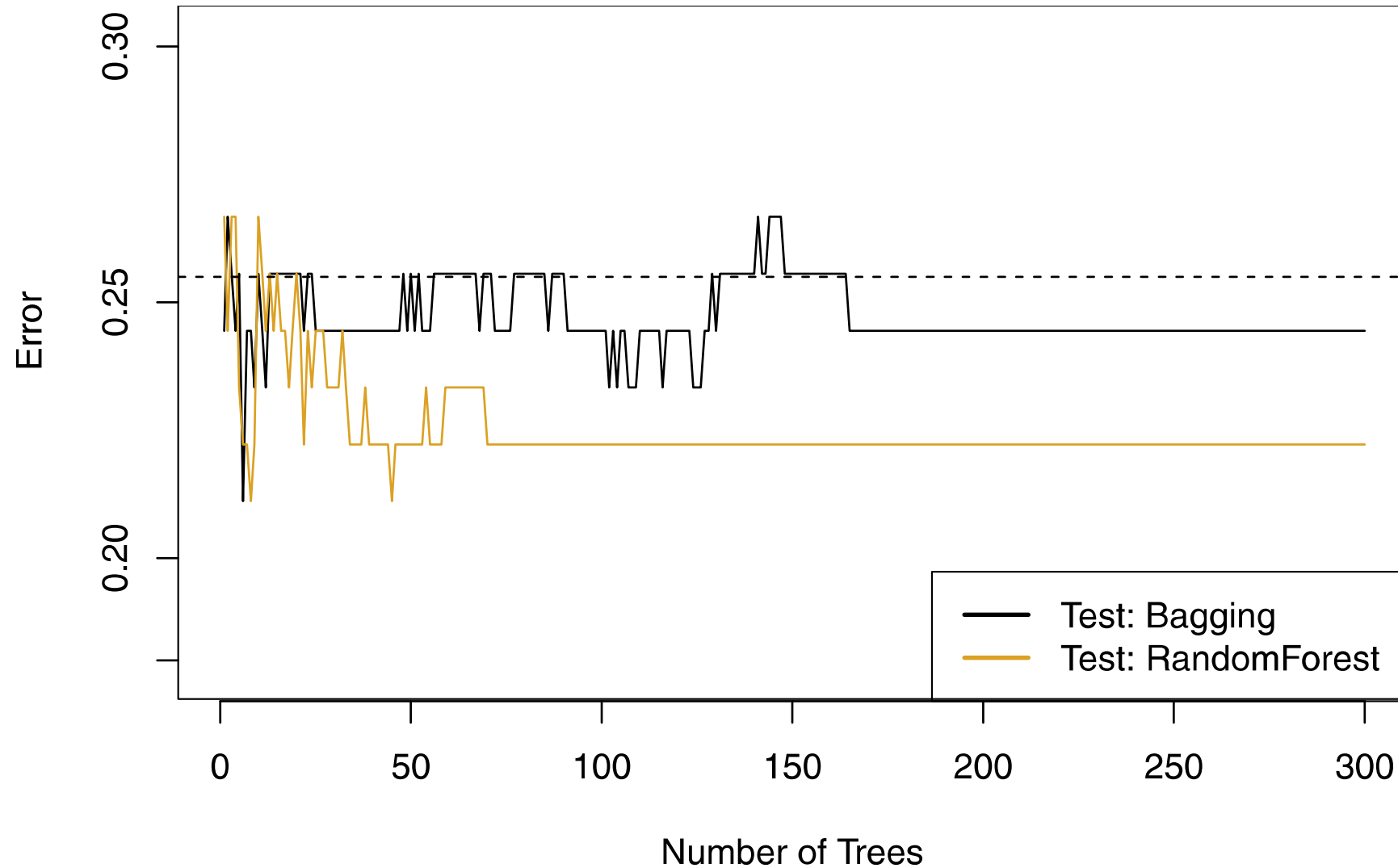
- I. Select m variables at random out of the p predictors
- II. Pick the best variable/split-point amongst the m possible choices
- III. Split the node into two daughter nodes

Output the ensemble of trees T_b with $b = 1$ to B

To make a prediction at a new point simply do
$$f_{\text{rf}}^B(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^B T_b(\mathbf{x})$$

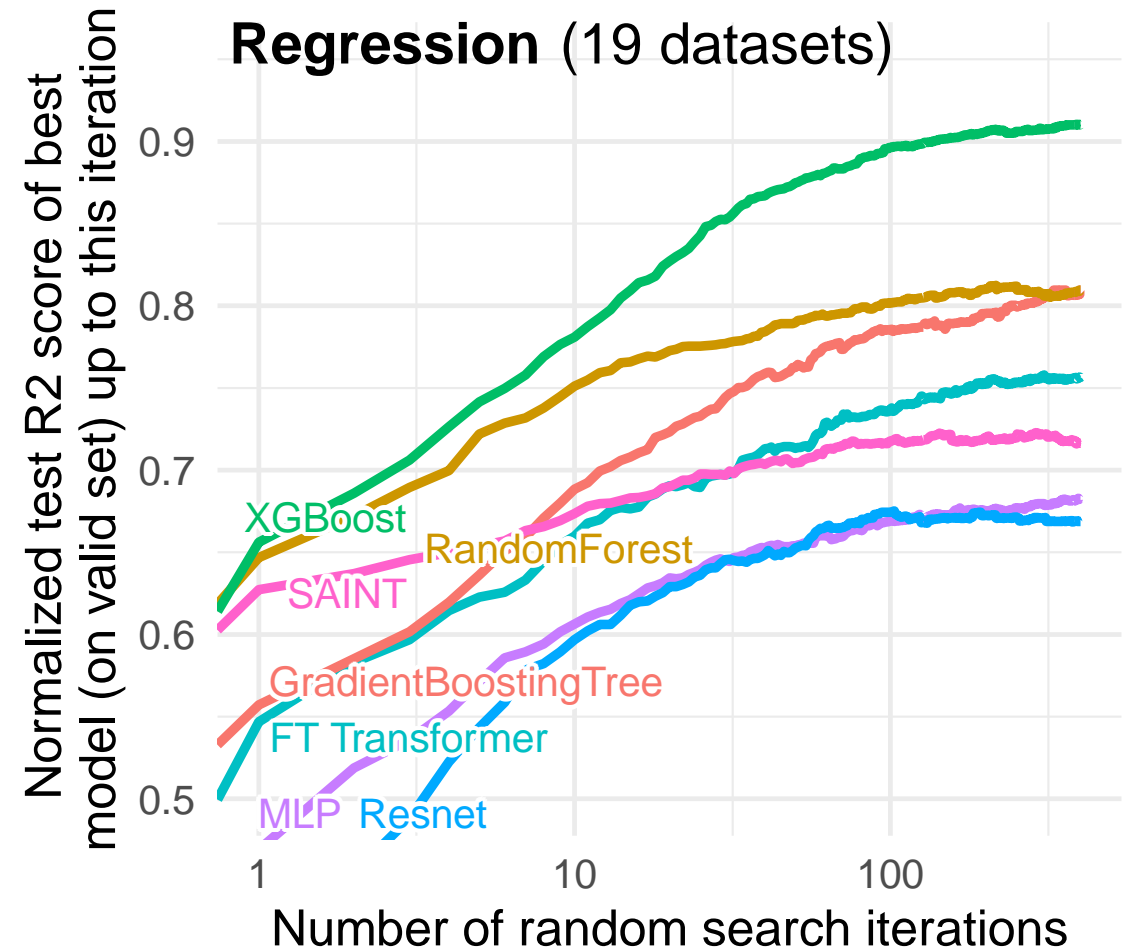
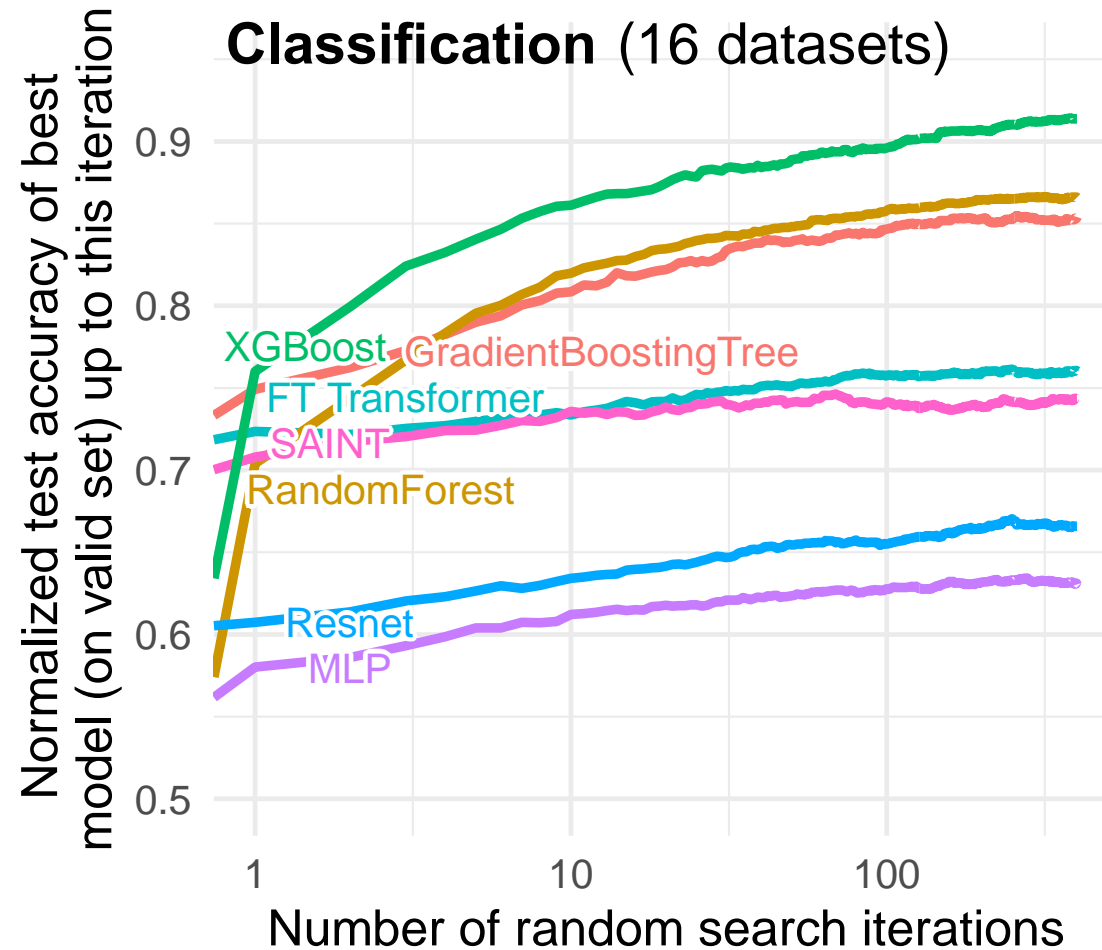
Random forests

Example 1 Classification test error on the heart dataset



Random forests

Example 2 “Why do tree-based models still outperform deep learning on tabular data?”
Grinsztajn et al. (2022)



Random forests

Regarding the parameters:

- ▶ The size of subsample should be reasonably large and can be with or without replacement (but often it is the former). The number m of dimensions of which we pick the best one is typically $p/3$. The number B of trees should be large.

However, random forests are known to be quite robust to these choices!

- ▶ Random forests have been invented by Leo Breinman in 2001
- ▶ They are a simple yet successful class of algorithms for classification and regression
- ▶ Always consider Random Forests as a baseline when you work on new ML problems

Random forests

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='sqrt', max_leaf_nodes=None,
min_impurity_decrease=0.0, 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, monotonic_cst=None)
```

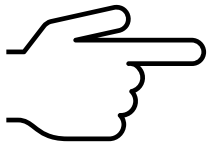
[\[source\]](#)

RandomForestRegressor

```
class sklearn.ensemble.RandomForestRegressor(n_estimators=100, *,
criterion='squared_error', max_depth=None, min_samples_split=2,
min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=1.0,
max_leaf_nodes=None, min_impurity_decrease=0.0, bootstrap=True,
oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False,
ccp_alpha=0.0, max_samples=None, monotonic_cst=None)
```

[\[source\]](#)

- Recap
- Ensemble methods
- Bagging
- Boosting



Boosting

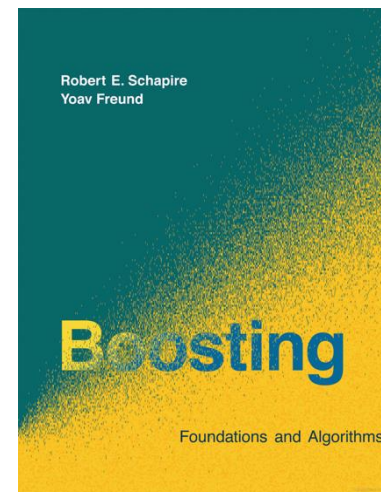
The goal of machine learning is to build **strong learners**. In the context of classification, this means having a small generalization error.

However, they are often hard to construct.

A **weak learner** is an algorithm that is just slightly better than random guessing. For example, in a binary classification problem this would mean a test error of $0.5 - \epsilon$.

► The idea of **boosting** is to combine many weak learners to obtain a strong learner.

“Boosting: Foundations and algorithms”
Schapire and Freund



Boosting

The outline of a classical boosting algorithm for classification (AdaBoost) is :

Given a training set of n points (x_i, y_i) the algorithm proceeds in $T+1$ rounds

Training points have weights that change from round to round. They add up to one.

In each round, we train a weak classifier on the training points with the current weights

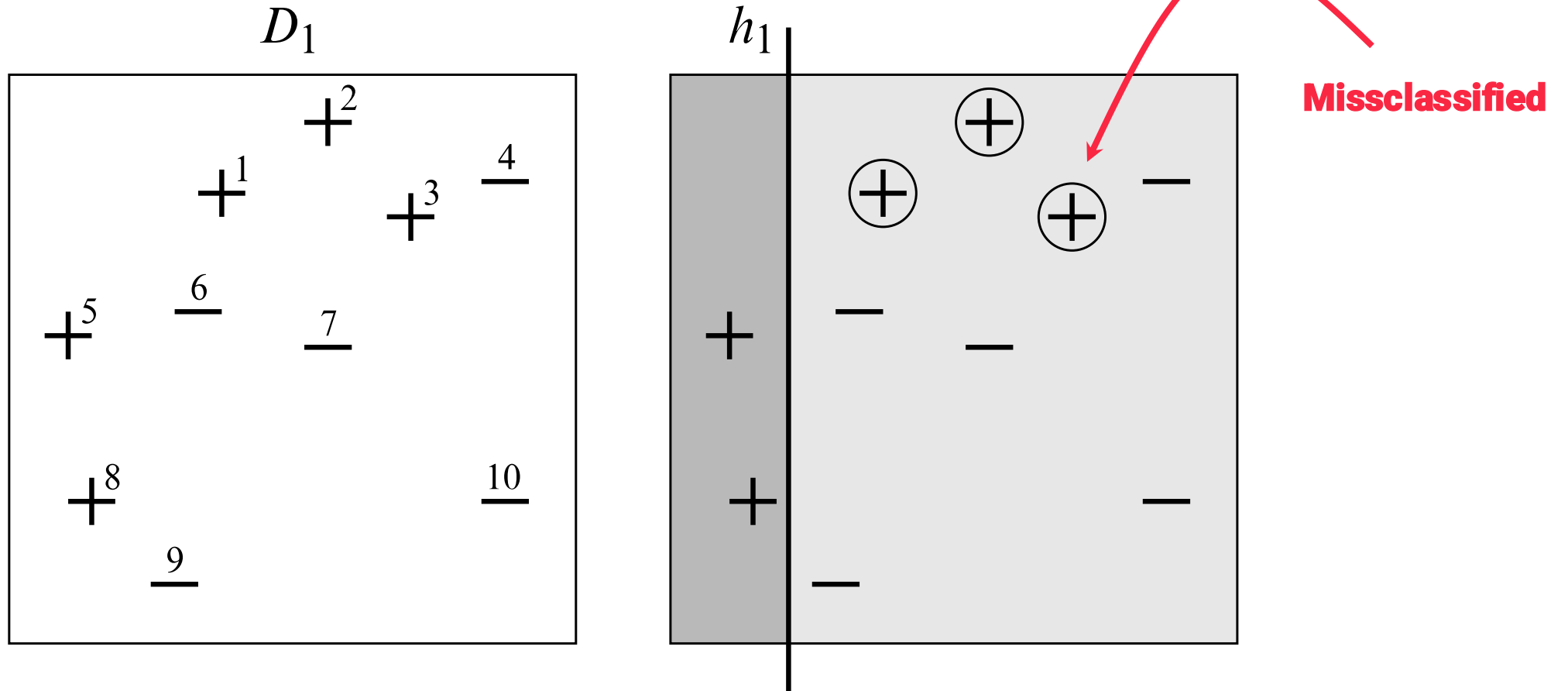
- For point x_i that was **missclassified**, we **increase** its weight w_i
- For point x_i that was **correctly classified**, we **decrease** its weight w_i

The final classifier uses a weighted sum of the weak classifiers of each round

$$f_{\text{boost}}(\mathbf{x}) = \text{sign} \left(\sum_{t=0}^T \alpha_t f_t(\mathbf{x}) \right)$$

Boosting

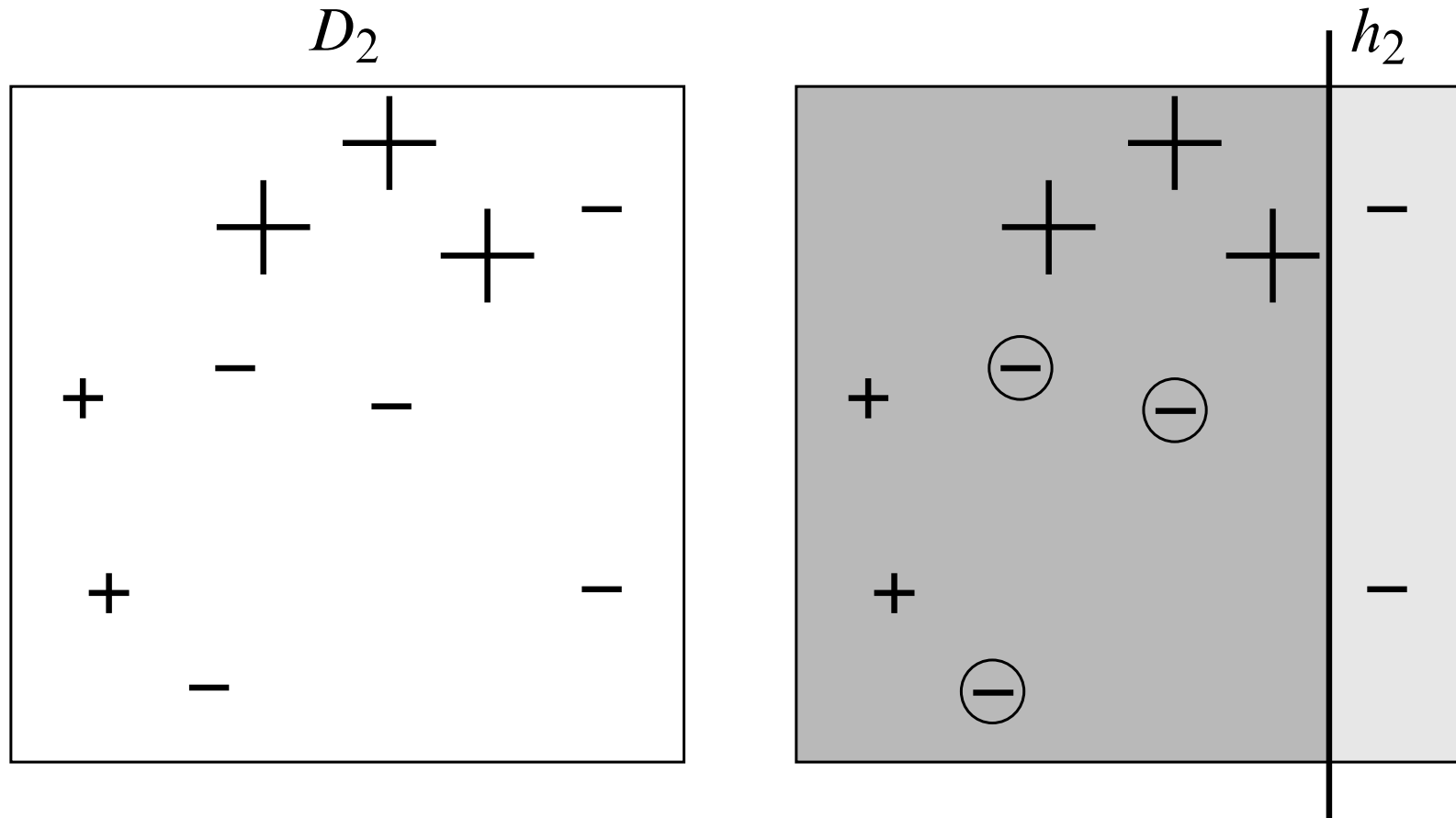
Example: Consider a toy model with 10 examples labelled as + or -



Our base classifier is a decision stump on the vertical or horizontal axis

Boosting

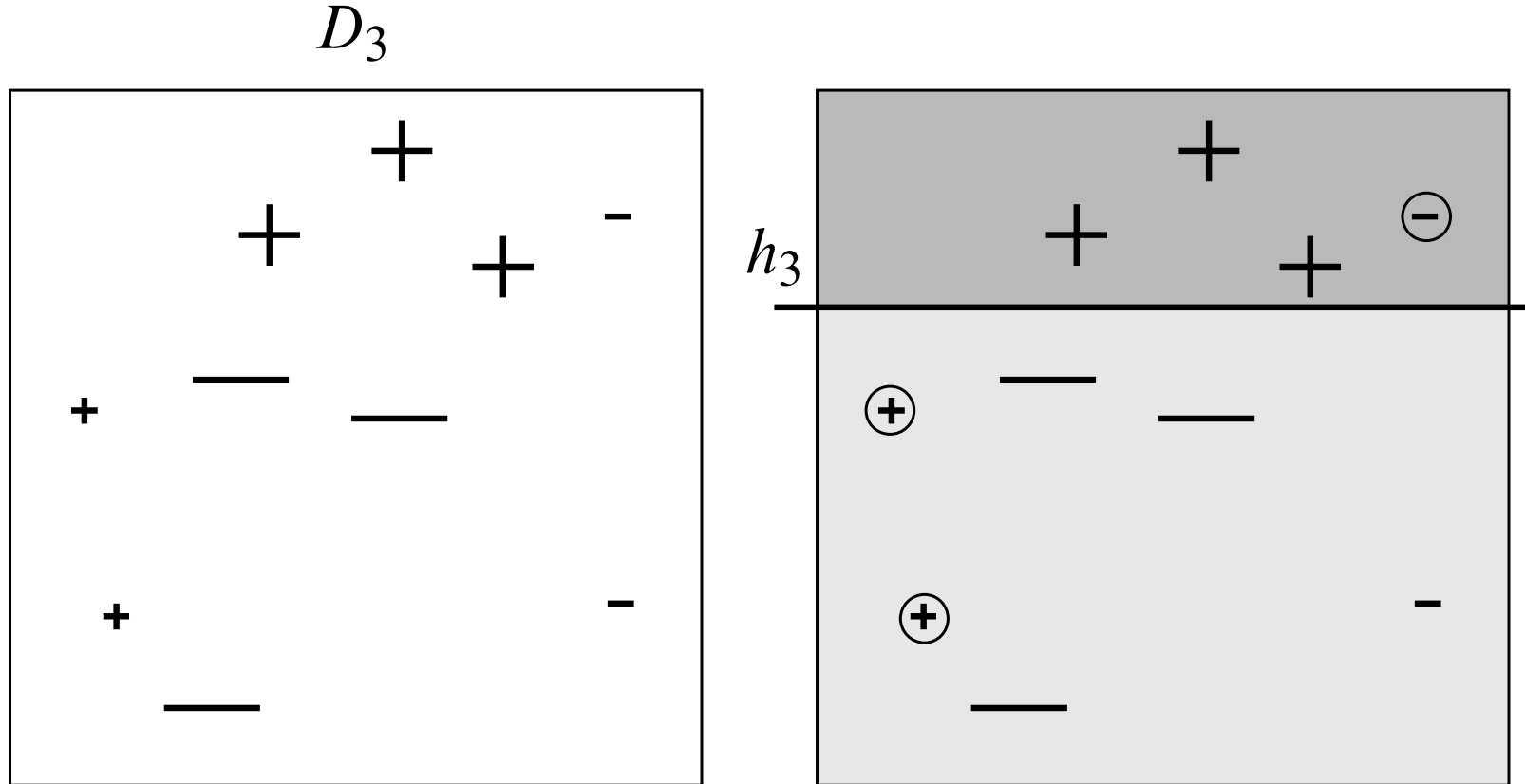
Example: Consider a toy model with 10 examples labelled as + or -



Our base classifier is a decision stump on the vertical or horizontal axis

Boosting

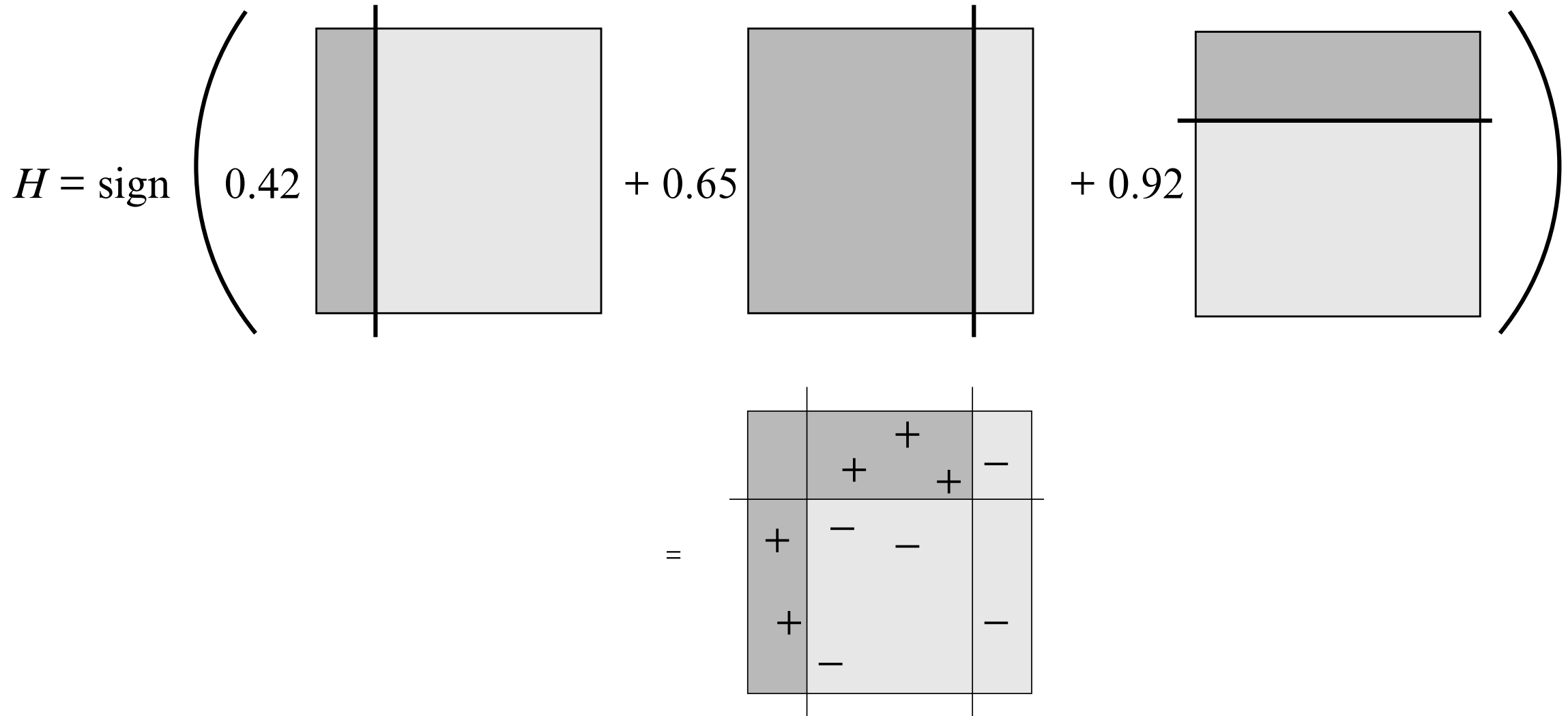
Example: Consider a toy model with 10 examples labelled as + or -



Our base classifier is a decision stump on the vertical or horizontal axis

Boosting

Example: Consider a toy model with 10 examples labelled as + or -



Boosting

The **AdaBoost** algorithm

Given: $(x_1, y_1), \dots, (x_m, y_m)$ where $x_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$.

Initialize: $D_1(i) = 1/m$ for $i = 1, \dots, m$.

For $t = 1, \dots, T$:

- Train weak learner using distribution D_t .
- Get weak hypothesis $h_t : \mathcal{X} \rightarrow \{-1, +1\}$.
- Aim: select h_t to minimize the weighted error:

$$\epsilon_t \doteq \Pr_{i \sim D_t}[h_t(x_i) \neq y_i].$$

- Choose $\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$.
- Update, for $i = 1, \dots, m$:

$$\begin{aligned} D_{t+1}(i) &= \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases} \\ &= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}, \end{aligned}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(x) \right).$$

Boosting

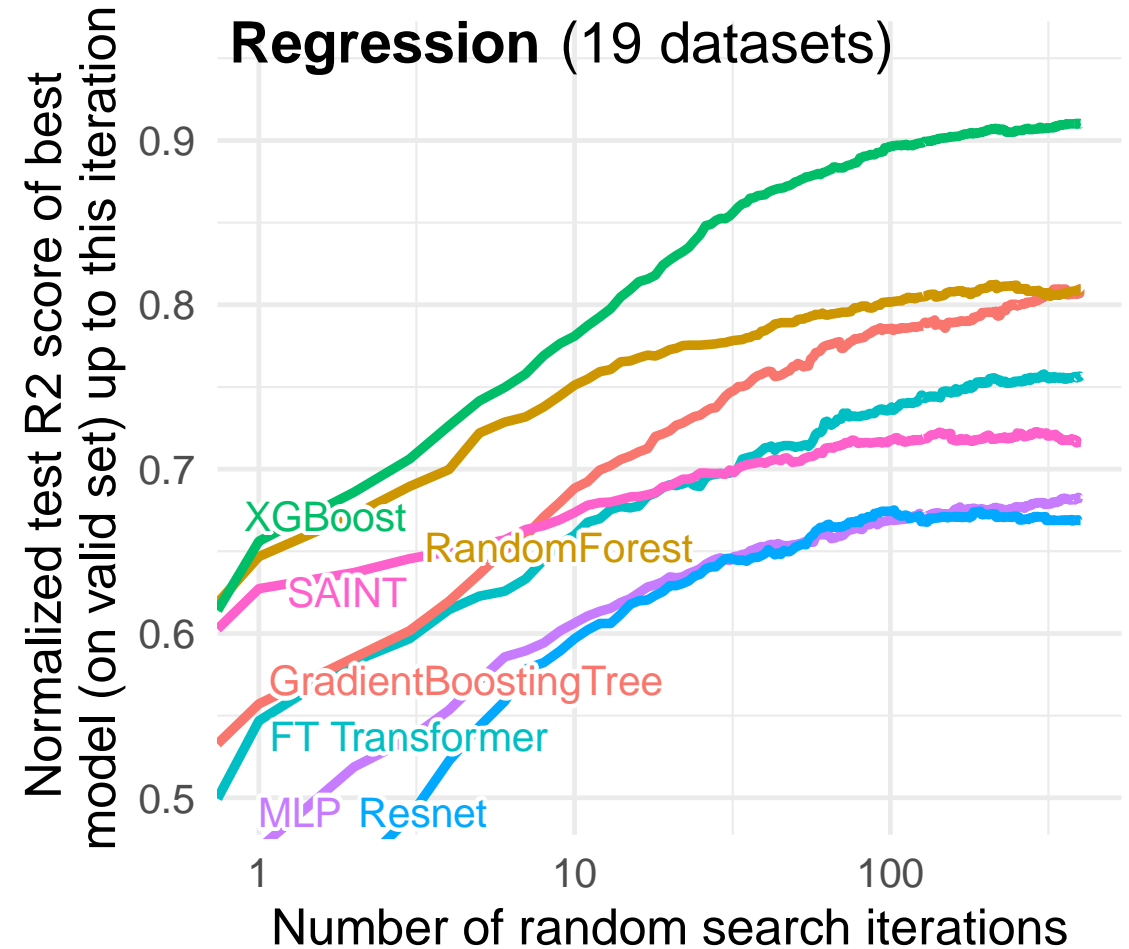
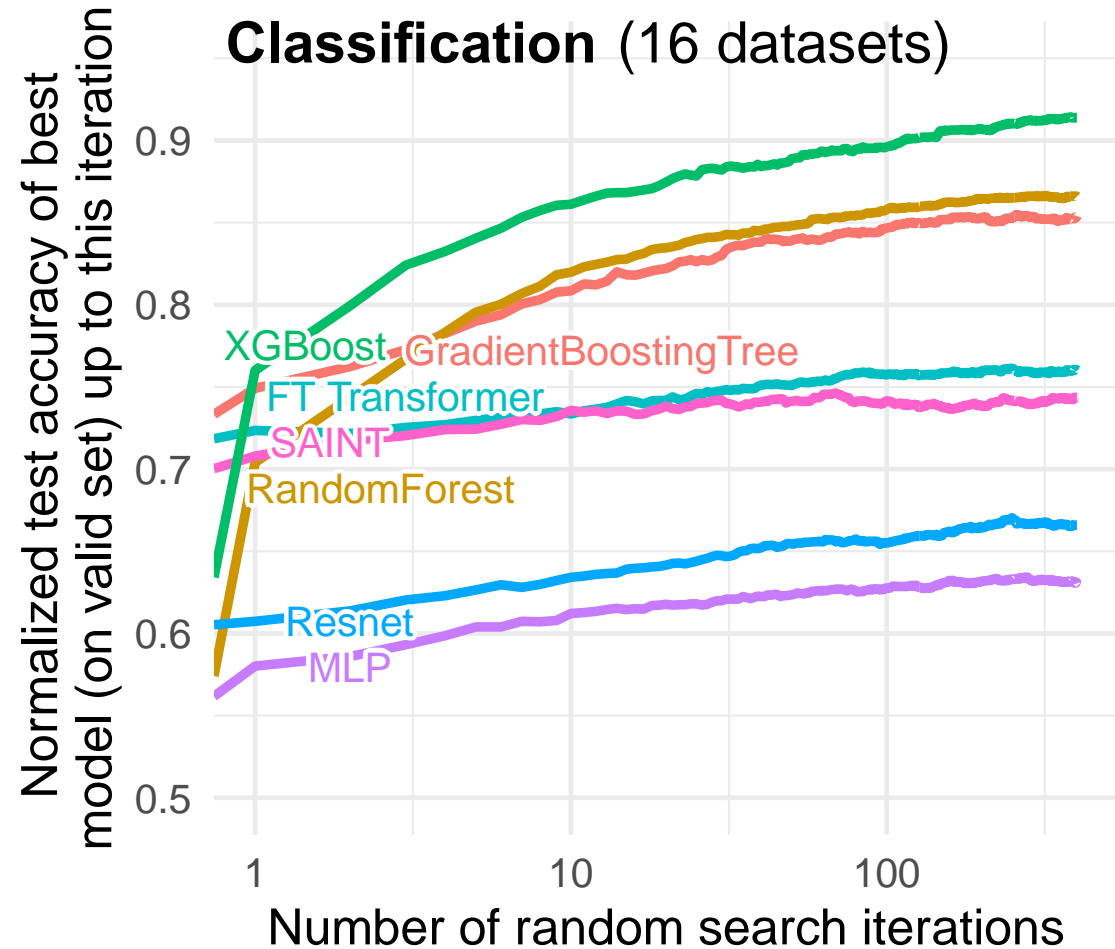
The **AdaBoost** algorithm... but why would it be plausible for it to work?

- Assume the final classifier makes an error on some training point x
- Because it is a weighted majority vote over all weak classifiers, it can only get x wrong if most of the weak classifiers classified x wrongly themselves
- This means that the weight of x has been increased very often, so it must be still large after the final round of boosting
- But there can only be a few points with large weights, since they all add to 1
- So there can only be few points that get classified wrong by the final classifier

$$f_{\text{boost}}(\mathbf{x}) = \text{sign} \left(\sum_{t=0}^T \alpha_t f_t(\mathbf{x}) \right)$$

Boosting

But what about **gradient** boosting?



Boosting

Boosting is a strategy for answering the question :

“Can we combine a set of weak learners so to generate a strong learner with low bias?”

The answer is **yes** and the procedure goes as follows:

- Suppose we have trained a learner f (i.e. a classifier or regressor) to minimize

$$f_0 = \operatorname{argmin}_{h \in \mathbb{H}} \mathcal{L}(h, \mathcal{D}) \quad \text{with} \quad \mathcal{L}(h, \mathcal{D}) = \sum_{i=1}^N \ell(h(\mathbf{x}_i), y_i)$$


- We could try to improve predictions by training a new learner so that

$$f_1 = \operatorname{argmin}_{h \in \mathbb{H}} \mathcal{L}(f_0 + h, \mathcal{D}) \quad \Rightarrow \quad f(\mathbf{x}) = f_0(\mathbf{x}) + f_1(\mathbf{x})$$

- Boosting sequentially adds several learners so to get a final strong predictor

Boosting

In **boosting** we approximate the change in the loss function for a small perturbation as

$$\mathcal{L}(f_0 + h, \mathcal{D}) \approx \mathcal{L}(f_0, \mathcal{D}) + \langle \nabla \mathcal{L}(f_0, \mathcal{D}), h \rangle$$


1st order Taylor expansion

so that the minimization problem becomes

$$\operatorname{argmin}_{h \in \mathbb{H}} \mathcal{L}(f_0 + h, \mathcal{D}) = \operatorname{argmin}_{h \in \mathbb{H}} \langle \nabla \mathcal{L}(f_0, \mathcal{D}), h \rangle$$

The gradient of the functional looks scary, but note that

$$\mathcal{L}(f, \mathcal{D}) = \sum_{i=1}^N \ell(f(\mathbf{x}_i), y_i) \Rightarrow \nabla \mathcal{L}(f, \mathcal{D}) = \begin{bmatrix} \vdots \\ t_i \\ \vdots \end{bmatrix} \in \mathbb{R}^N$$

where $t_i = \frac{\partial \ell(f(\mathbf{x}_i), y_i)}{\partial f(\mathbf{x}_i)}$

Boosting

Generic boosting (aka AnyBoost):

- Start with dataset $\mathcal{D}_0 = \left\{ (\mathbf{x}_i, y_i) \right\}_{i=1}^{i=N}$ and train $f_0 = \operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^N \ell(h(\mathbf{x}_i), y_i)$ $\mathcal{L}(h, \mathcal{D}_0)$
 - Form new dataset $\mathcal{D}_1 = \left\{ (\mathbf{x}_i, t_i^{(1)}) \right\}$ with $t_i^{(1)} = \frac{\partial \mathcal{L}(f_0, \mathcal{D}_0)}{\partial f_0(\mathbf{x}_i)} = \frac{\partial \ell(f_0(\mathbf{x}_i), y_i)}{\partial f_0(\mathbf{x}_i)}$
 - Train new predictor $f_1 = \operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^N t_i^{(1)} h(\mathbf{x}_i)$
 - Combine the predictors to get $f(\mathbf{x}_i) = f_0(\mathbf{x}_i) + \alpha_1 f_1(\mathbf{x}_i)$
- Continue iterating to get more and more predictors in the sum

Boosting

If we're dealing with regression using MSE we get a very nice interpretation...

$$\mathcal{L}(f, \mathcal{D}) = \sum_{i=1}^N \left(f(\mathbf{x}_i) - y_i \right)^2 \Rightarrow \langle \nabla \mathcal{L}(f, \mathcal{D}), h \rangle = \sum_{i=1}^N \left(h(\mathbf{x}_i) - r_i \right)^2$$

with $r_i = y_i - f(\mathbf{x}_i)$

So that

$$f_1 = \operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^N \left(h(\mathbf{x}_i) - r_i^{(1)} \right)^2 \quad \text{with} \quad r_i^{(1)} = -t_i^{(1)} = y_i - f_0(\mathbf{x}_i)$$

residuals

Conclusion:

Learner f_1 is trained using the residuals of the previous learner f_0

This means that the new regressors are trained on the errors from the previous round

PS: We will see in TD how to derive these results and particularize it to decision trees

Boosting

Boosting algorithm for regression:

○ Start with dataset $\mathcal{D}_0 = \left\{ (\mathbf{x}_i, y_i) \right\}_{i=1}^{i=N}$ and train $f_0 = \operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^N \left(h(\mathbf{x}_i) - y_i \right)^2$

↑ ○ Form new dataset $\mathcal{D}_1 = \left\{ (\mathbf{x}_i, r_i^{(1)}) \right\}$ with $r_i^{(1)} = y_i - f_0(\mathbf{x}_i)$

○ Train new predictor $f_1 = \operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^N \left(h(\mathbf{x}_i) - r_i^{(1)} \right)^2$

○ Combine the predictors to get $f(\mathbf{x}_i) = f_0(\mathbf{x}_i) + \alpha_1 f_1(\mathbf{x}_i)$ └ Note this new parameter!

Continue iterating to get more and more predictors in the sum

Boosting

Put very simply, every boosting algorithm does the following:

- ▶ At round T fit an additive model
$$h_T(\mathbf{x}) = \sum_{t=0}^T \alpha_t f_t(\mathbf{x})$$
- ▶ In each new round $T+1$, train a weak learner to **compensate shortcomings** of h_T

In AdaBoost, shortcomings are identified by high-weight data points

In gradient boosting they are identified by gradients

- Invent AdaBoost, the first successful boosting algorithm (Freund 1996)
- Formulate AdaBoost as a gradient descent with a special loss function (Breiman 1998)
- Generalize AdaBoost to Gradient Boosting to handle other loss functions (Friedman 2000)

Boosting

Gradient boosting has a story of much success with some famous implementations

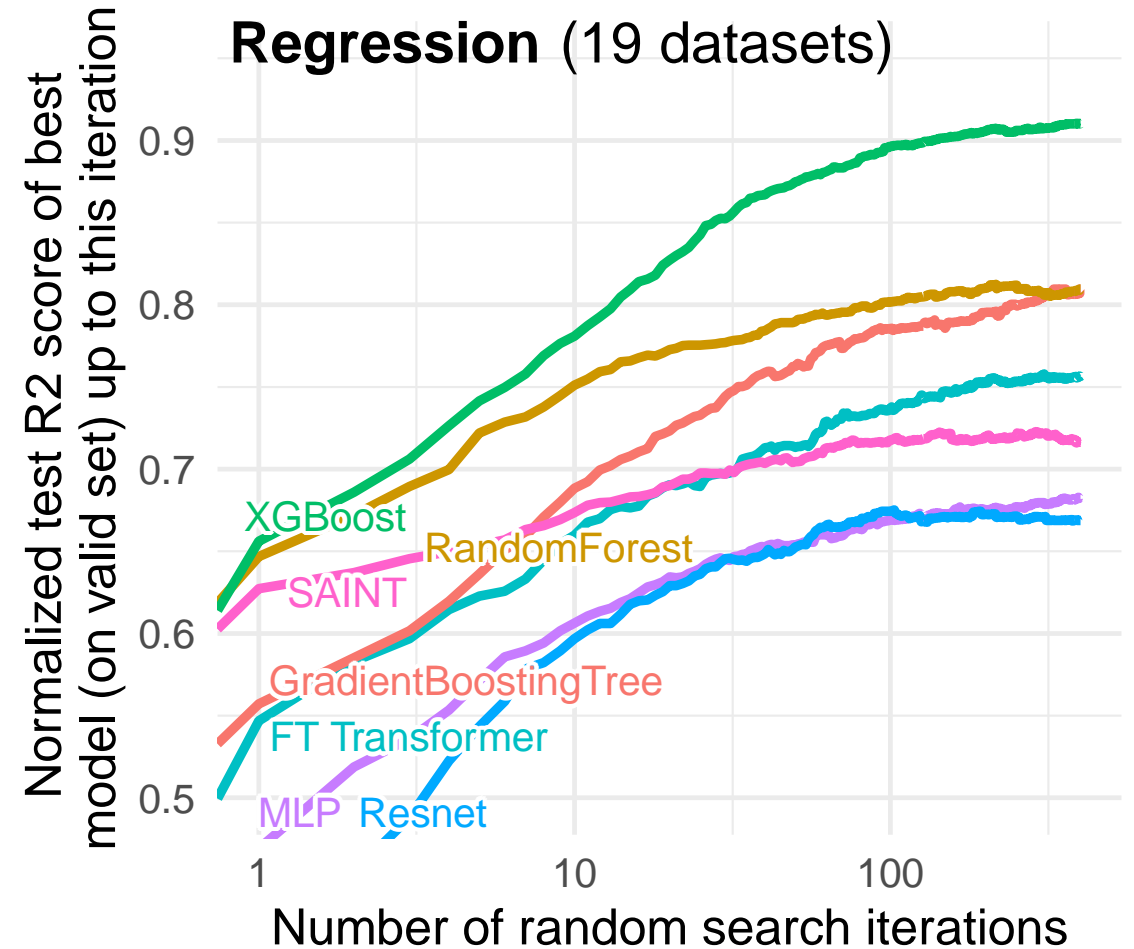
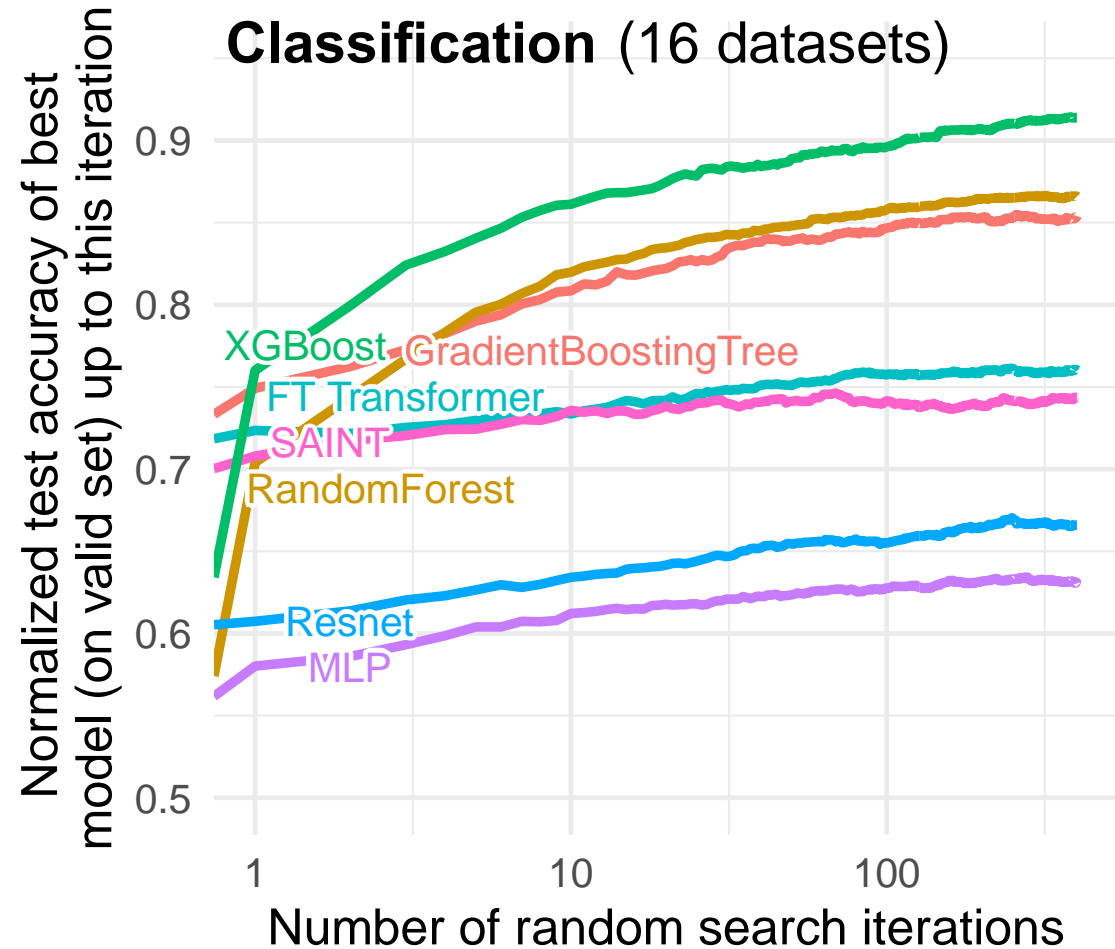
- XGBoost [Chen & Guestrin, Arxiv 2016] (w. Apple, NVidia)
- LightGBM [Ke et al., Proc. NIPS 2017] (by Microsoft)
- CatBoost [Prokhorenkova et al. Arxiv 2017] (by Yandex)
- `sklearn.ensemble.HistGradientBoostingClassifier` (v0.21)

You can check this video with some nice extra practical explanations:


<https://www.youtube.com/watch?v=o6seqpMJSTI>

Boosting

We see some of these implementations in the curves



XGBoost: eXtreme gradient boosting

- Proposed in Chen and Guestrin (2016) – arxiv.org/abs/1603.02754
- XGBoost uses a 2nd Order Taylor approximation of the cost function
- XGBoost has a regularization term to control the size of the trees
- Several tricks to make the algorithm scale to very large datasets
 - **Approximate splitting via feature binning** 
 - Parallel implementation
 - Cache-aware access

dmlc
XGBoost

XGBoost: Approximate splitting via feature binning

Remember that when growing a tree, we had to do for each of P features:

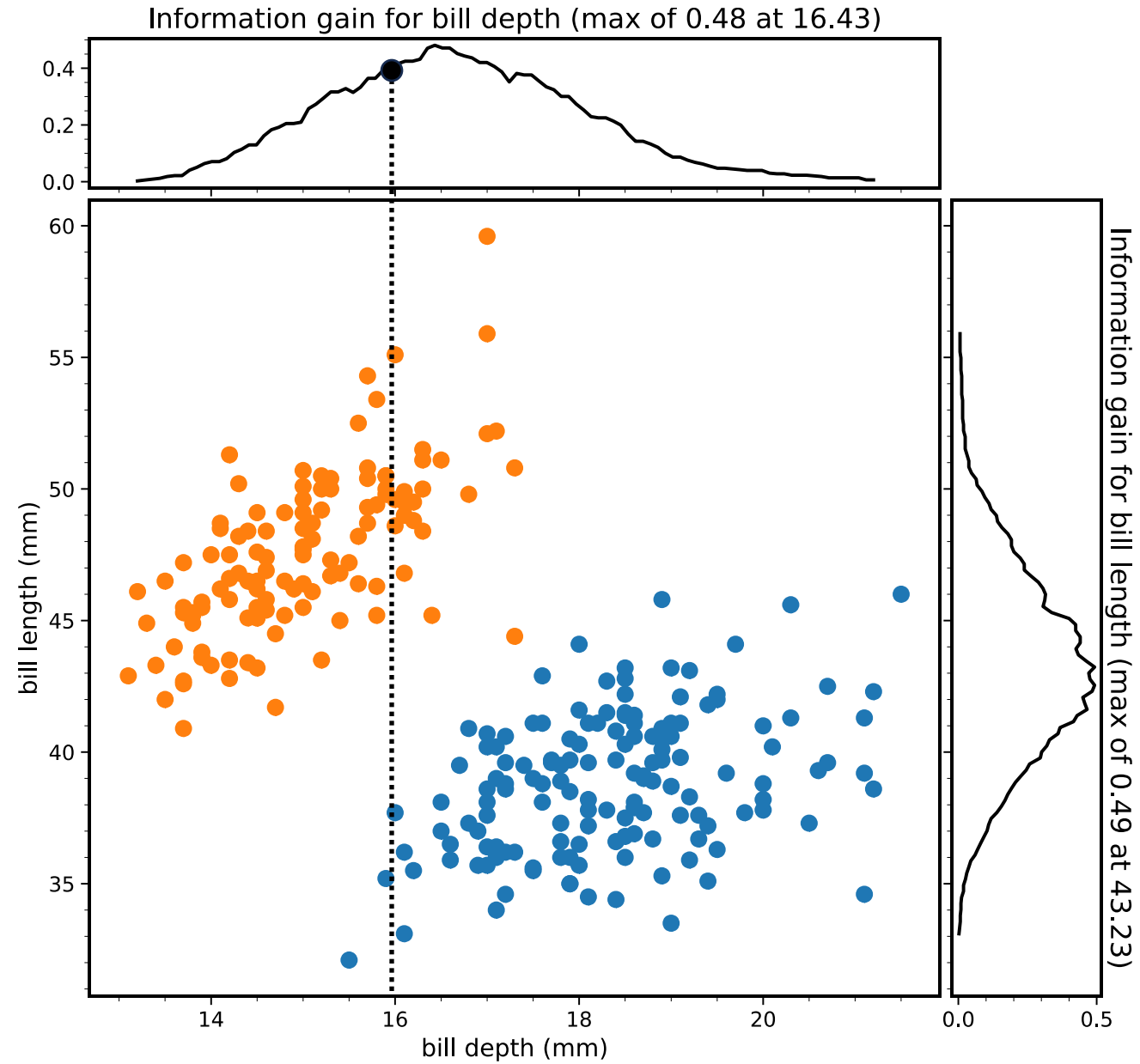
- 1) Sort the values of the data points on that feature **$O(n \log n)$**
- 2) For every possible split point **$O(n)$**
 - 2.1) Evaluate the split
- 3) Pick the best split

Final complexity : **$O(D \cdot n \log n + n)$**

-- Q: What if we could avoid having to sort?

dmlc
XGBoost

Boosting



XGBoost: Approximate splitting via feature binning

Remember that when growing a tree, we had to do for each of P features:

- 1) Sort the values of the data points on that feature **$O(n \log n)$**
- 2) For every possible split point **$O(n)$**
 - 2.1) Evaluate the split
- 3) Pick the best split

Final complexity : **$O(D \cdot n \log n + n)$**

-- Q: What if we could avoid having to sort?

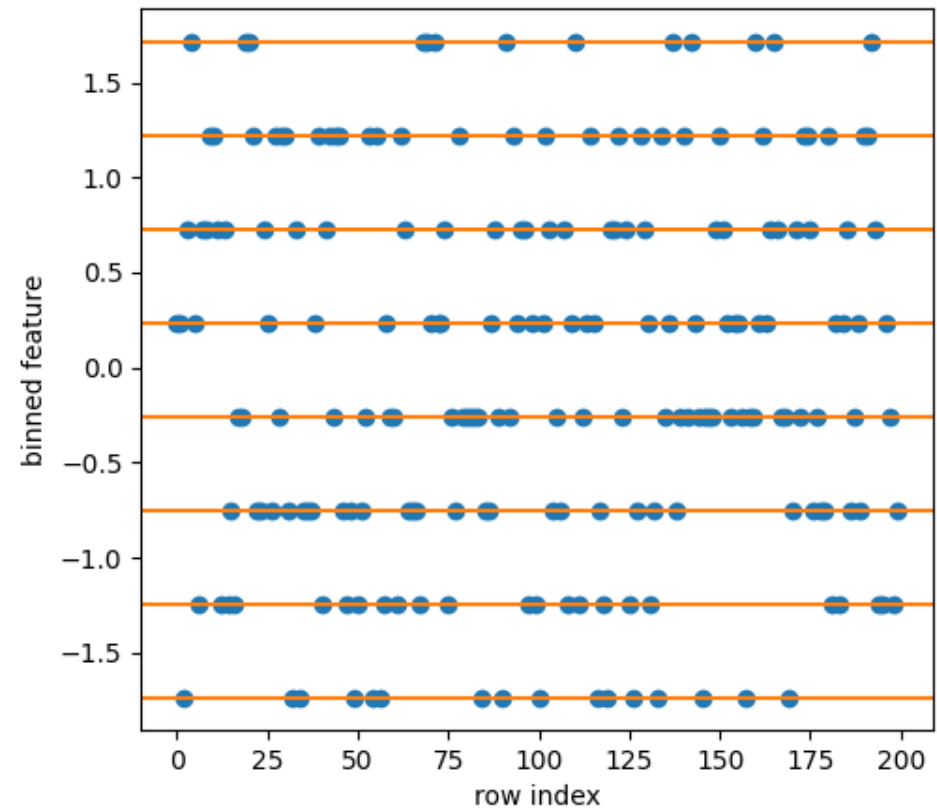
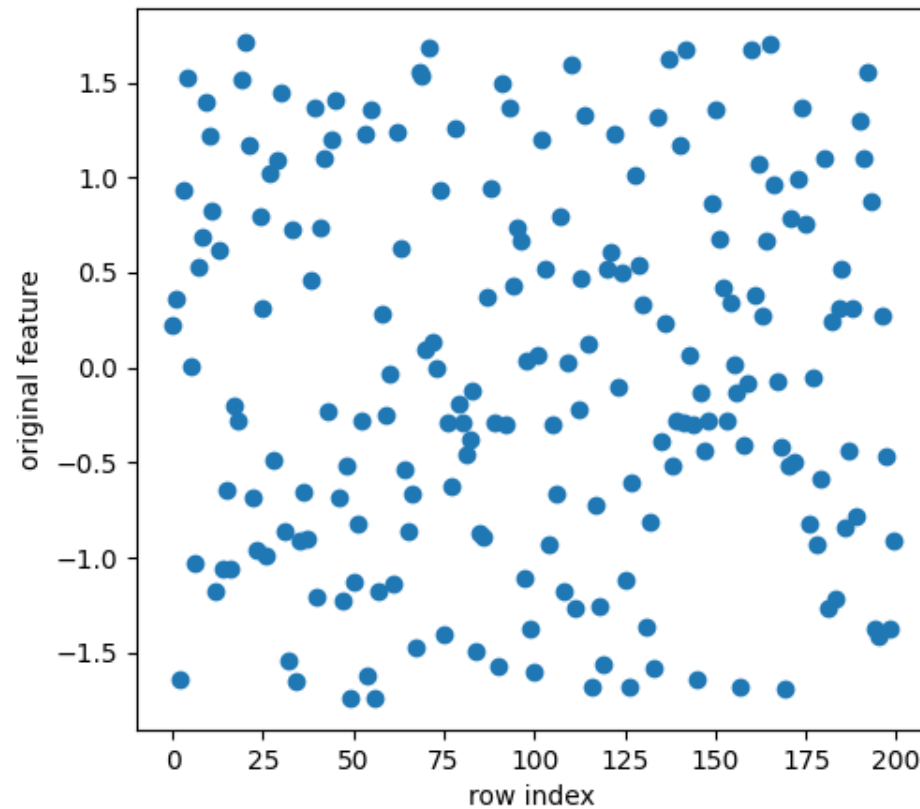
dmlc
XGBoost

Boosting

XGBoost: Approximate splitting via feature binning

-- Q: What if we could avoid having to sort?

-- A: We can bin the feature values and create a histogram!



XGBoost: Approximate splitting via feature binning

Remember that when growing a tree, we had to do for each of P features:

- 1) Construct a histogram $O(n)$
- 2) For every possible split point $O(n_bins)$
 - 2.1) Evaluate the split
- 3) Pick the best split

Final complexity : $O(D \cdot n_bins + n)$



[sklearn.decomposition](#) ▾[sklearn.discriminant_analysis](#) ▾[sklearn.dummy](#) ▾[sklearn.ensemble](#) ▴[AdaBoostClassifier](#)[AdaBoostRegressor](#)[BaggingClassifier](#)[BaggingRegressor](#)[ExtraTreesClassifier](#)[ExtraTreesRegressor](#)[GradientBoostingClassifier](#)[GradientBoostingRegressor](#)[HistGradientBoostingClassifier](#)[HistGradientBoostingRegressor](#)[IsolationForest](#)[RandomForestClassifier](#)[RandomForestRegressor](#)

HistGradientBoostingClassifier

```
class sklearn.ensemble.HistGradientBoostingClassifier(loss='log_loss', *,  
learning_rate=0.1, max_iter=100, max_leaf_nodes=31, max_depth=None,  
min_samples_leaf=20, l2_regularization=0.0, max_features=1.0, max_bins=255,  
categorical_features='from_dtype', monotonic_cst=None, interaction_cst=None,  
warm_start=False, early_stopping='auto', scoring='loss',  
validation_fraction=0.1, n_iter_no_change=10, tol=1e-07, verbose=0,  
random_state=None, class_weight=None)
```

[\[source\]](#)

Histogram-based Gradient Boosting Classification Tree.

This estimator is much faster than [GradientBoostingClassifier](#) for big datasets ($n_{\text{samples}} \geq 10\,000$).

This estimator has native support for missing values (NaNs). During training, the tree grower learns at each split point whether samples with missing values should go to the left or right child, based on the potential gain. When predicting, samples with missing values are assigned to the left or right child consequently. If no missing values were encountered for a given feature during training, then samples with missing values are mapped to whichever child has the most samples.

Boosting

Overview of the gradient boosting with regression trees

- 1) If large-scale dataset : bin data for each feature
- 2) Make initial predictions with f_0 – it could be simply constants
- 3) Calculate gradients of the loss $\mathcal{L}(f_0, \mathcal{D})$
- 4) Grow trees for boosting:
 - 4.1) Find best splits
 - 4.2) Add tree to predictors
 - 4.3) Update de gradients and hessians

Conclusion

In summary, the two main methods that we've seen today can be compared as below

| BAGGING | | BOOSTING |
|---------------------------------------|---|------------------------------------|
| Resamples data points | ↔ | Reweight data points |
| Weight of each classifier is the same | ↔ | Weight is dependent of performance |
| Only variance reduction | ↔ | Both variance and bias are reduced |