Machine Learning

7. Ensemble learning

Yannick Le Cacheux

CentraleSupélec - Université Paris Saclay

September 2024

Outline

The bias-variance decomposition

- Assumptions:

$$y = f(\mathbf{x}) + \epsilon, \quad \text{where } \epsilon \sim \mathcal{N}(0, \sigma^2)$$
$$\hat{y} = \hat{f}_{\mathcal{D}}(\mathbf{x})$$

$f(\mathbf{x})$: *"Ground truth"* function (generally unknown)

y: Available target(s): ground truth + some noise
e.g. measurement error

$\hat{f}_{\mathcal{D}}(\mathbf{x})$: Estimated function from dataset $\mathcal{D}$

- We can then write the expected error as:

$$\mathbb{E}[(y - \hat{y})^2] = \mathbb{E}\left[((y - f) + (f - \hat{y}))^2\right]$$
$$\vdots$$
$$= \underbrace{\sigma^2}_{\substack{\text{Intrinsic} \\ \text{noise}}} + \underbrace{\mathbb{E}\left[(f - \mathbb{E}[\hat{f}])^2\right]}_{\text{Bias}^2} + \underbrace{\text{Var}[\hat{f}]}_{\text{Variance}}$$

What does it mean in practice?

- There is a trade-off between bias and variance:
  - Either we have an estimated $\hat{f}$ with low bias, but there will be high variance in the estimated $\hat{f}$
  - Or we have low variance in $\hat{f}$, but we will have higher bias

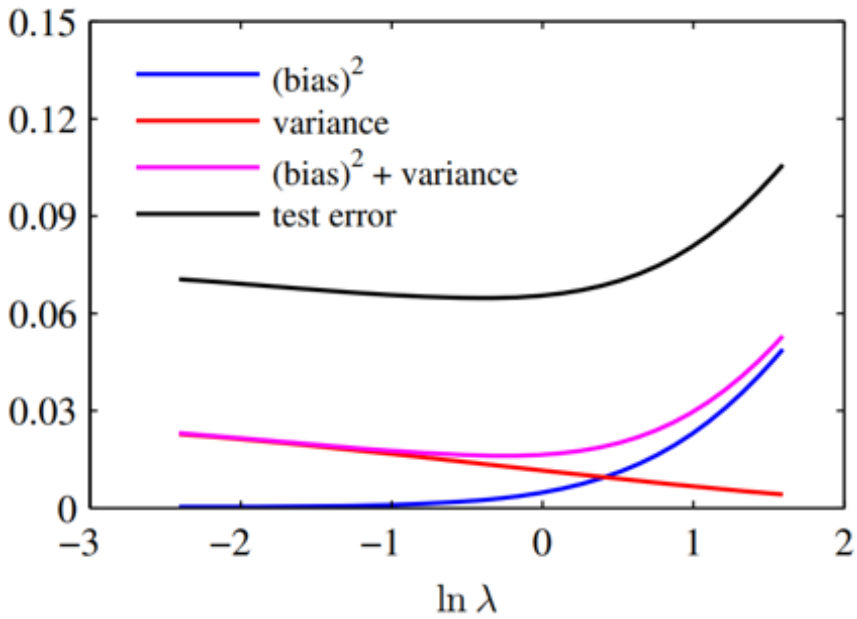
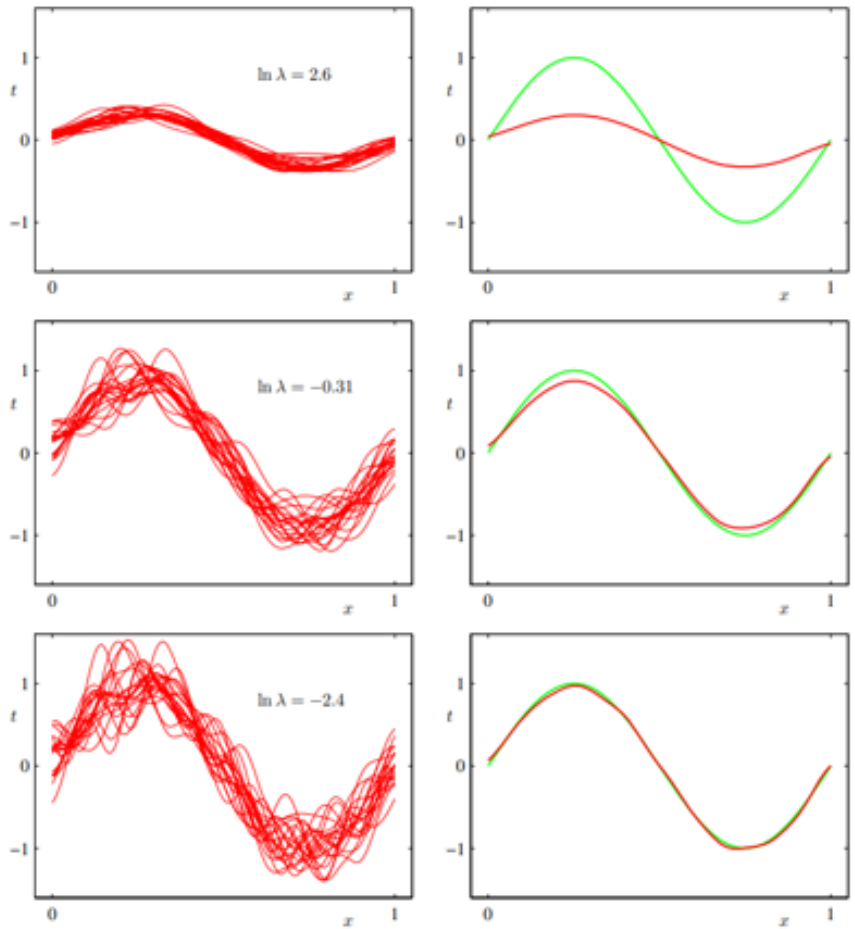


Figure: [1] Increasing the regularization decreases the variance but increases the bias. Total error is the sum of bias, variance and noise.

[1] From Bishop.

Illustration of the trade-off

- We sample different datasets, and estimate $\hat{f}$



High bias
Low variance

Medium bias
Medium variance

Low bias
High variance

Ensemble learning

- Generic idea: train multiple volontarily *"weak" learners* h_t, and use a combination of these as the final model H.
- Each weak learner h_t has *e.g.* high variance but low bias.[2]
- By combining all the weak learners into one "stronger" learner, we keep a low bias and reduce the variance, thus decreasing the global error of the model.
- Does this mean we got rid of the bias-variance tradeoff?

[2] In the case of bagging. In boosting, it is sometimes the opposite: individual learners may have high bias but low variance, and the combination have low bias and low variance, cf in a few slides.

Ensemble learning

- Generic idea: train multiple volontarily *"weak" learners* h_t, and use a combination of these as the final model H.
- Each weak learner h_t has *e.g.* high variance but low bias.[2]
- By combining all the weak learners into one "stronger" learner, we keep a low bias and reduce the variance, thus decreasing the global error of the model.
- Does this mean we got rid of the bias-variance tradeoff?
- No, we can still overfit with ensemble learning; but we try to find a good tradeoff between bias and variance.

[2] In the case of bagging. In boosting, it is sometimes the opposite: individual learners may have high bias but low variance, and the combination have low bias and low variance, cf in a few slides.

Bagging *vs* boosting

Bagging:

- Each learner is trained on a subset of the original training dataset
- Learners are trained in parallel

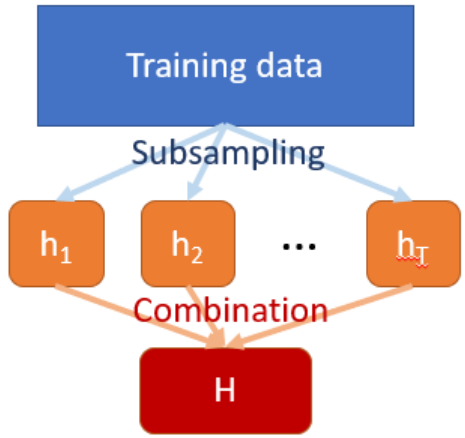


Figure: Bagging illustration

Boosting:

- Each weak learner h is (usually) trained on the whole training set
- Learners are trained sequentially

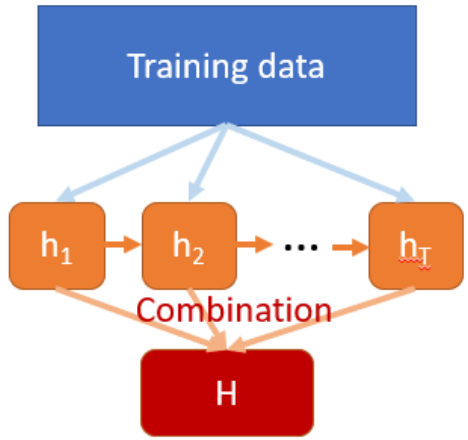


Figure: Boosting illustration

Outline

Bagging

- Each learner h_t is trained on a (randomly sampled) subset of the original "full" training dataset $\mathcal{D}$
- Learners are trained independently and can be trained in parallel
- The final model H is a combination of learners $h_1, \ldots, h_T$
  - For instance, we can use a voting system in the case of a classification task

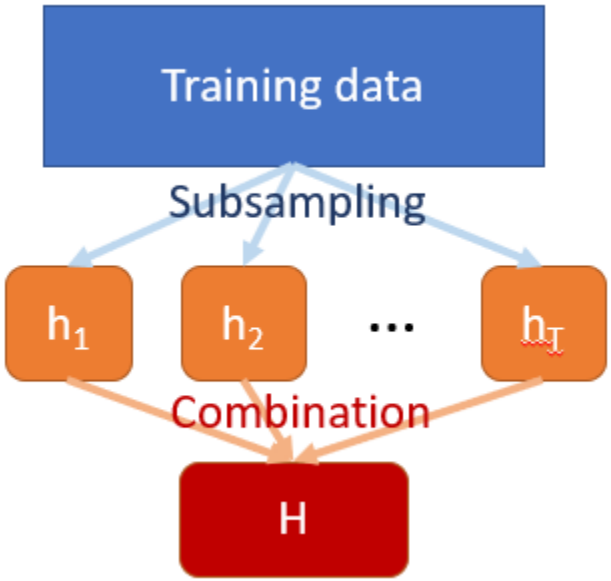


Figure: Illustration of bagging

Ensemble learning with decision trees

Bagged trees

- Straightforward application of bagging: decision trees h_t are trained in parallel on subsets of $\mathcal{D}$
- For a regression task, we can
- *Side questions*:
  - : *What are the hyper-parameters?*
- In a *random forest*, in addition to randomly sampling training examples, we randomly sample the features that we consider at each split

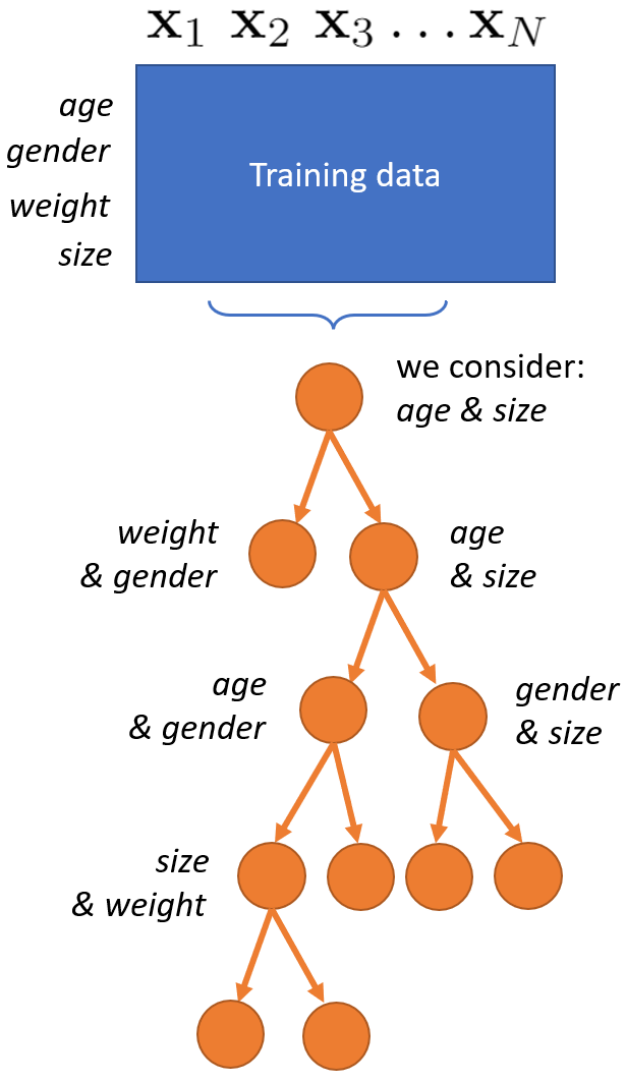


Figure: Illustration of a random forest

Scikit-learn documentation on random forest

- Which hyper-parameter(s) let use control the complexity of the model?

Parameters:

n_estimators : *int, default=100*
The number of trees in the forest.

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

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

min_samples_leaf : *int or float, default=1*
The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

max_features : *{"auto", "sqrt", "log2"}, int or float, default="auto"*
The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)` (same as "auto").

Variants of random forest

As usual, many variants exist:

- We can use different types of decision trees (entropy, gini...)
- We can add some extra randomness to obtain "Extremely Randomized Trees":
  - In a random forest, we select some features at random at each split, but still select the best feature and data partition among these features
  - In ERT, we select a subset features *and* data partition at random (but still keep the best partition among this subset).

Interpretability of random forest

- Is it possible to obtain probabilities instead of binary (or multiple) class outputs?

```
predict_proba(X)
```

- How is this computed?

Interpretability of random forest

- Is it possible to obtain probabilities instead of binary (or multiple) class outputs?

```
predict_proba(X)
```

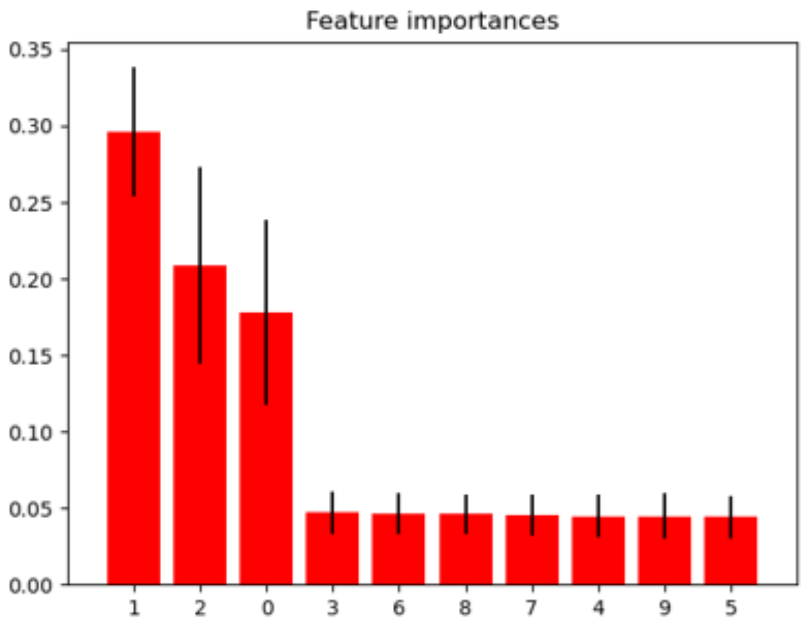
- How is this computed?
- Here, the probability associated with tree h_t is the ratio of 1s in the corresponding leaf
- The probability of the forest is the mean probabilities of the trees

Interpretability of random forest

- Scikit-learn also provides an easy way to display the importance of each feature in a random forest:

```
forest = ExtraTreesClassifier(n_estimators=250,
                              random_state=0)

forest.fit(X, y)
importances = forest.feature_importances_
```



- How do we interpret this?

Interpretability of random forest

- Here, the importance refers to (normalized) reduction in the splitting criterion (entropy, gini impurity...)
- However, it does not tell us whether there is a positive or negative correlation between the feature and the target
  - Since random forests are non linear, there may be zero correlation between an important feature and the output

Be careful

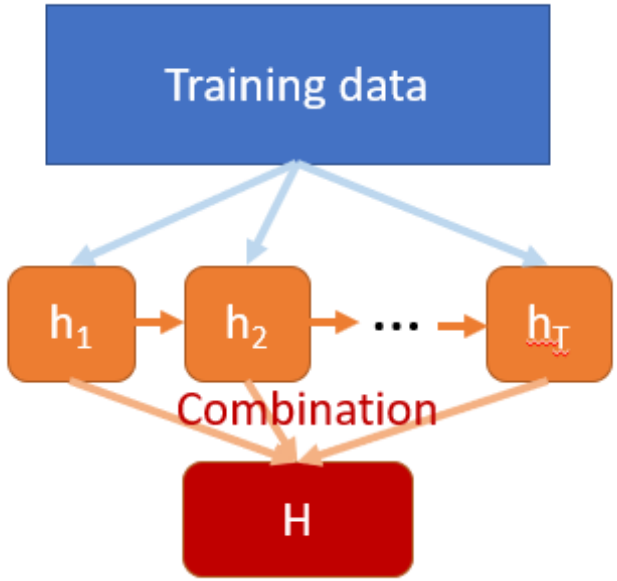
If the model is overfitting, feature importance is totally meaningless

Outline

Boosting

General idea: learners h_t are trained sequentially

- The 1st weak learner h_1 makes a first basic estimation, and makes some errors
- The 2nd learner h_2 tries to correct these errors
- ...
- The final combination H of all learners should benefit from all these corrections by having low error



Adaboost

- One of the first proposed boosting algorithms (Freund *et al.*, 1997)
- Also a fairly simple one

Main idea:

- We train a first basic classifier h_1
- The importance v in the loss function L associated with misclassified samples is increased, the importance of correctly classified samples is decreased
- This process is repeated for steps 1, ..., T
- Each successive classifier h_t is thus incentivized to correct the errors made by its predecessors h_{t-1}, h_{t-2}...

Adaboost initialization

- We have N labels $y_n \in \{-1, 1\}$
- Classification function at time t:

$$h_t(\mathbf{x}_n) \in \{-1, 1\}$$

- "Importance" weight associated with n^{th} sample at step t:

$$v_n^t \in [0, 1]$$

- Initially, we set

$$v_n^0 = \frac{1}{N} \quad \forall n$$

Adaboost training

- We train classifier h_t for time step t on the weighted dataset $\mathcal{D}_t$
- For instance, using the logistic loss with labels $y_n \in \{0, 1\}$, estimated probabilites at time t $\hat{p}_n^t \in [0, 1]$ and importance weights $v_n^t \in [0, 1]$:

$$J = -\sum_{n=1}^{N} v_n^t (y_n \log(\hat{p}_n) + (1 - y_n) \log(1 - \hat{p}_n))$$

- We compute the (weighted) error rate ϵ_t of time t from estimated labels $\hat{y}_n^t \in \{0, 1\}$:

$$\epsilon_t = \frac{\sum_{n=1}^{N} v_n^t \mathbb{1}[y_n = \hat{y}_n^t]}{N}$$

Adaboost update

- We update each importance weight v_n^t based on whether $\mathbf{x}_n$ was correctly classified:

$$v_n^{t+1} = \frac{v_n^t}{Z_t} \exp(-\alpha_t y_n h_t(\mathbf{x}_n))$$

where

$$\exp(-\alpha_t y_n h_t(\mathbf{x}_n)) = \begin{cases} e^{\alpha_t} & \text{if } h_t(\mathbf{x}_n) \neq y_n \\ e^{-\alpha_t} & \text{if } h_t(\mathbf{x}_n) = y_n \end{cases}$$

with

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \qquad Z_t = 2\sqrt{\epsilon_t (1 - \epsilon_t)}$$

(normalization constant so that $\sum_n v_n^t = 1$)

Adaboost prediction

- We repeat the training and update steps T times
- The resulting classifier is a weighted combination of successive classifiers:

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

- Classifiers with fewer errors have a higher weight α_t

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

Adaboost example

- Example: we use a "stump" function h as a weak learner (equivalent to a decision tree with depth 1)

$$h(\mathbf{x}) = \text{sign}(w_d x_d + x_0) \quad \text{for some } d \in [1, D]$$

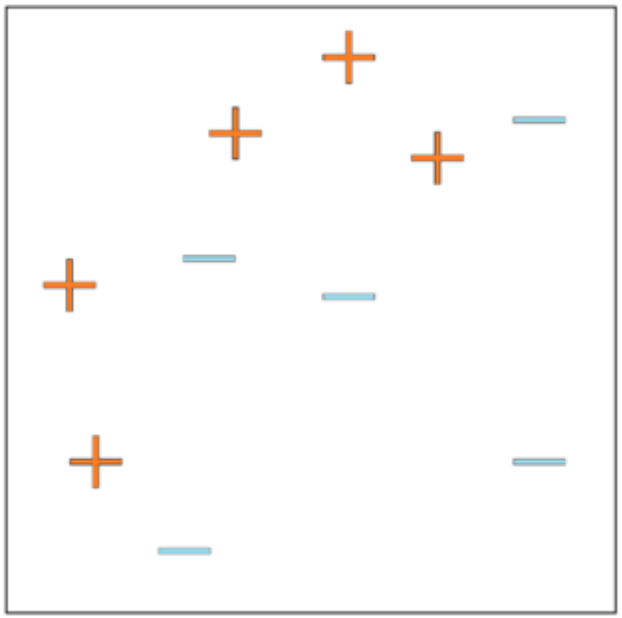


Figure: Training data

Adaboost example

- 1st iteration: we choose a linear boundary, make some errors and increase the weights of corresponding training samples

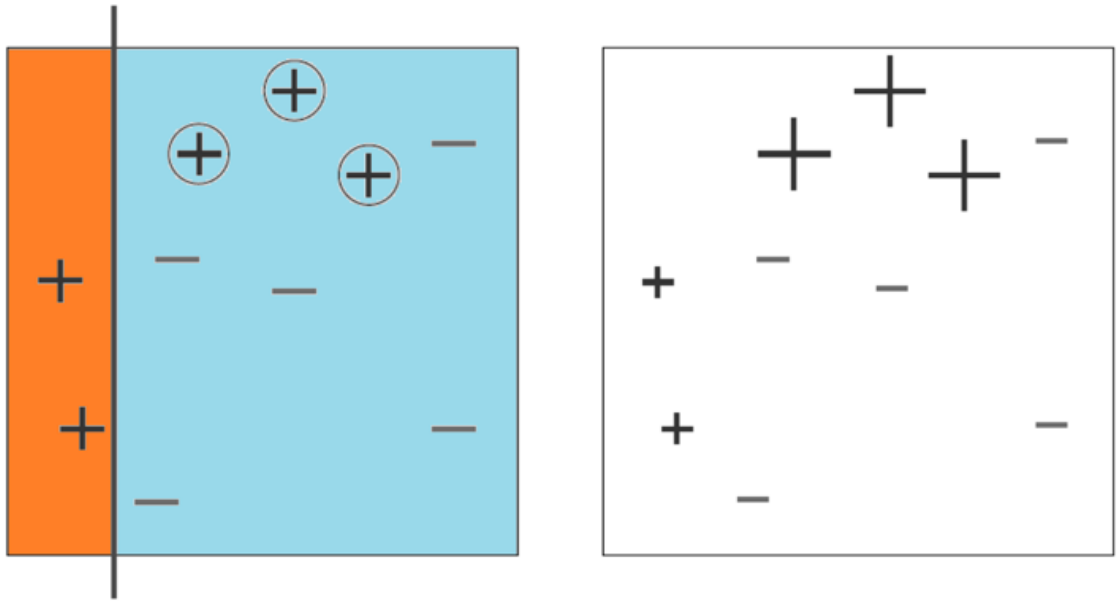


Figure: Result after 1 iteration: $\epsilon_1 = 0.3$, $\alpha_1 = 0.42$

Adaboost example

- 2nd iteration

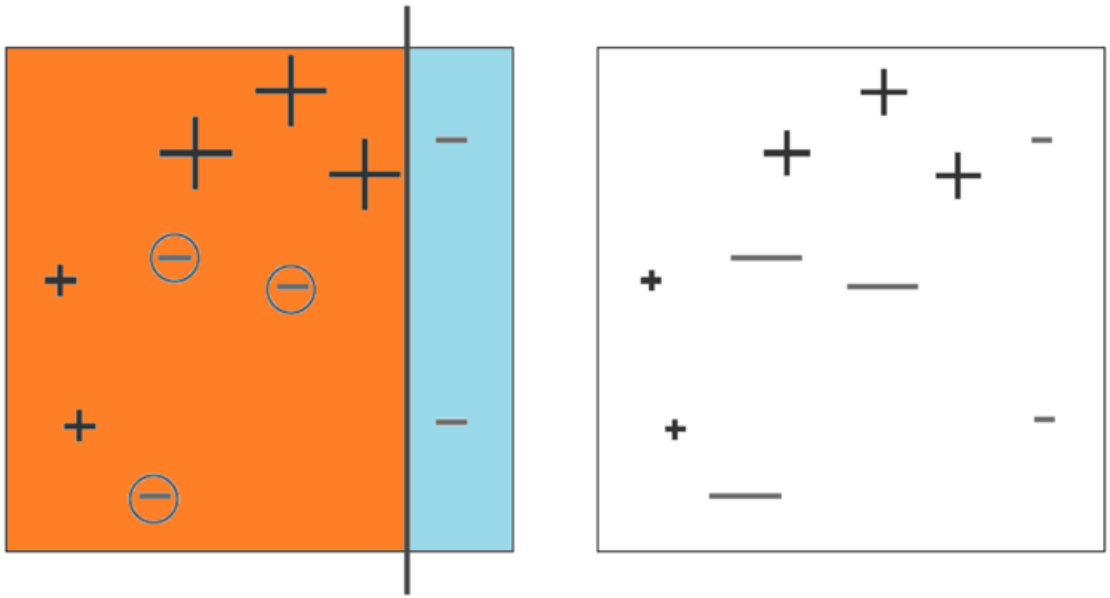
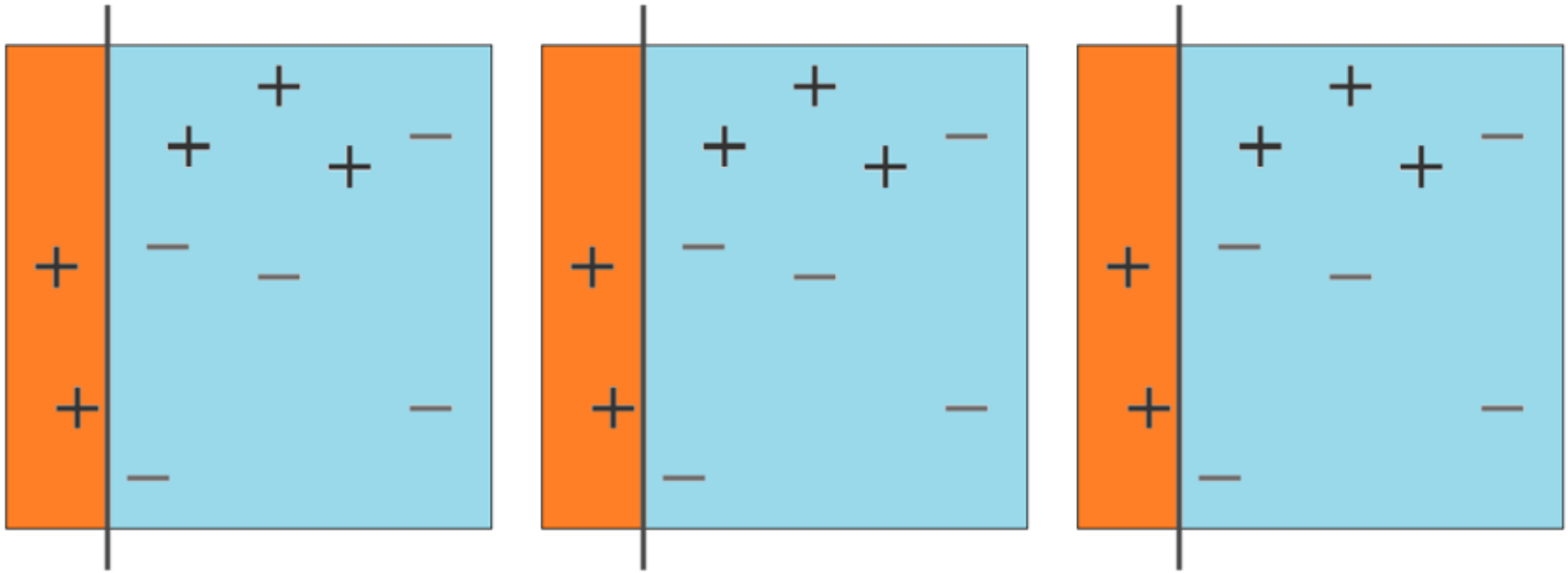


Figure: Result after 1 iteration: $\epsilon_2 = 0.21$, $\alpha_2 = 0.65$

Adaboost example

- After 3 iterations, we have 3 classifiers h_1, h_2 and h_3



(a) h_1
$\epsilon_1 = 0.3,\ \alpha_1 = 0.42$

(b) h_2
$\epsilon_2 = 0.21,\ \alpha_2 = 0.65$

(c) h_3
$\epsilon_3 = 0.14,\ \alpha_3 = 0.92$

Adaboost example

- We combine h_1, h_2 and h_3 to obtain H

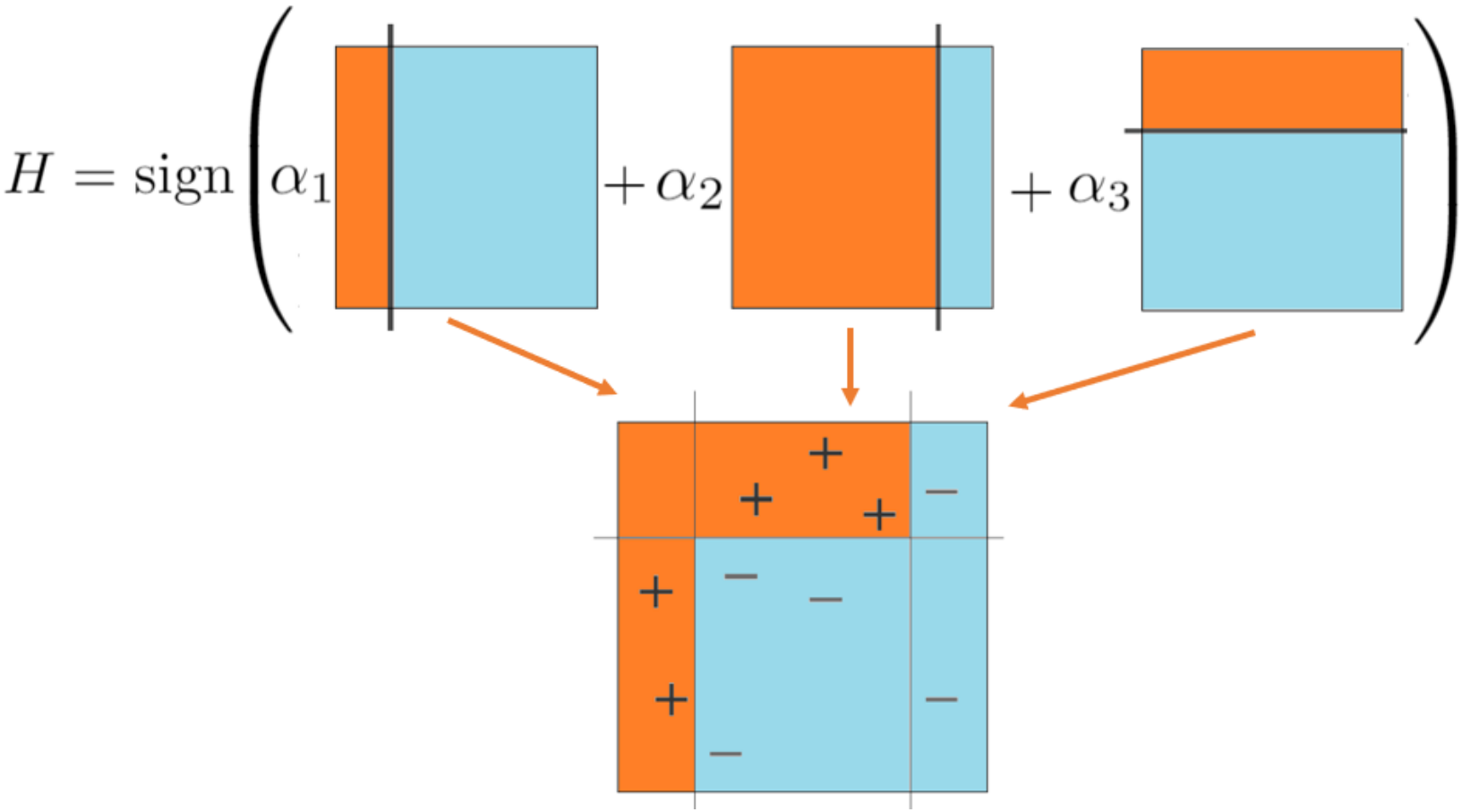


Figure: $\alpha_1 = 0.42$, $\alpha_2 = 0.65$, $\alpha_3 = 0.92$

Gradient boosting

- Let's consider a regression task where we have samples $\mathbf{x}_n$ and labels y_n.
- We first fit a learner h_1 to approximate $\mathbf{x}$ from y

$$H_1(\mathbf{x}_n) = h_1(\mathbf{x}_n) \approx y_n \quad \forall n$$

(we write $H_t(\mathbf{x}) = \sum_{i=1}^{t} \alpha_i h_i(\mathbf{x})$)

- H_2 tries to correct errors of H_1 so that

$$H_2(\mathbf{x}_n) = H_1(\mathbf{x}_n) + h_2(\mathbf{x}_n) \approx y_n$$

- Or more generally

$$h_t(\mathbf{x}_n) \approx \underbrace{y_n - H_{t-1}(\mathbf{x}_n)}_{\text{residual}}$$

Gradient boosting

- Let us write the cost function

$$J = \frac{1}{2N} \sum_{n=1}^{N} (y_n - H_{t-1}(\mathbf{x}_n))^2$$

- We notice that

$$-\frac{\partial J}{\partial H_{t-1}(\mathbf{x}_n)} = y_n - H_{t-1}(\mathbf{x}_n)$$

- So minimizing $y - H_{t-1}(\mathbf{x})$ is equivalent to doing gradient descent

Gradient boosting

In general, in gradient boosting, we make the residual update formula equal to the gradient of the cost function.

[2] The Adaboost update formula from earlier actual corresponds to gradient boosting if we use the exponential loss as a loss function.

Important note on cost functions

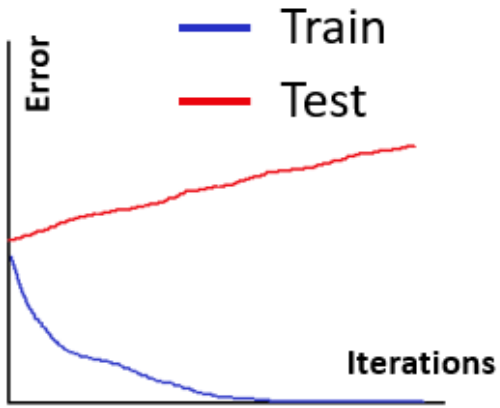
- Since each learner tries to correct errors made by previous learners, this method is very sensitive to outliers
- The choice of the cost function is therefore very important

$$J_1 = |y - \hat{y}| \qquad (\ell 1 \text{ loss})$$

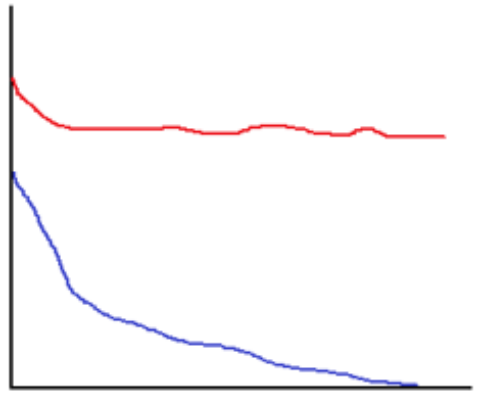
$$J_2 = (y - \hat{y})^2 \qquad (\ell 2 \text{ loss})$$

$$J^\delta = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta |y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \qquad \text{(Huber loss)}$$

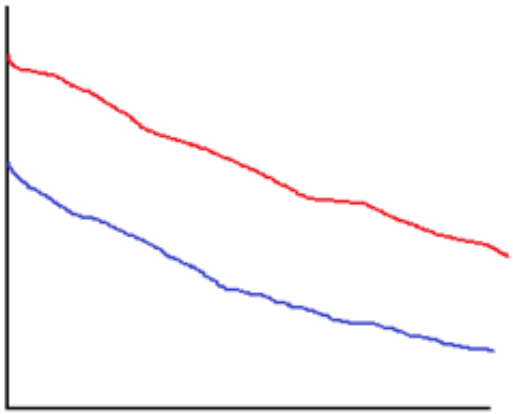
Gradient boosted trees in practice



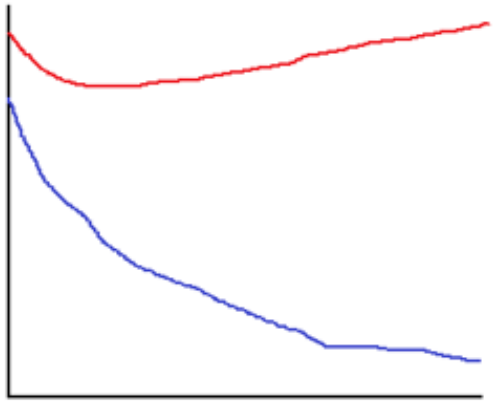
(a) Decrease max depth



(b) Decrease learning rate



(c) Increase number of iterations



(d) ↘ iterations, ↘ learn rate

Figure: Adapted from Biernat & Lutz, "Data Science: fondamentaux et études de cas"

Gradient boosting libraries

- Lightgbm: easy to use and fast to train. Default hyper-parameters often give fairly good results. Useful as a first model to get a quick, first idea of performance reachable on given dataset.
- Xgboost: often enable to get really good performance. Winner of many data science challenges.
- ...

Gradient boosting

Gradient boosting is very often used in practice, as it enables to fairly easily obtain good performance.[3]

[3] This is likely an approach you may want to try if you are working on a challenge or something that requires good performance *(*wink wink*)*