Introduction to machine learning

Master 2 Modélisation Aléatoire
Master 2 Mathématiques et Informatique pour la Data Science

Aurélie Fischer

Some supervised learning methods

- k-nn and kernel rules
- Classification and regression trees (CART)
- Bagging
- Random forests
- Boosting, adaboost, gradient boosting

General setting

We observe n independent realizations
$\mathcal{D}_n = (X_1, Y_1), (X_2, Y_2), \ldots (X_n, Y_n)$ from a generic random vector (X, Y), with values in $\mathbb{R}^d \times \{0, 1\}$ (binary classification), $\mathbb{R}^d \times \{1, \ldots, K\}$ (multi-class classification), or $\mathbb{R}^d \times \mathbb{R}$ (regression).

Nearest neighbor rule

The simplest nonparametric classification or regression rule is probably the nearest neighbor rule.

Definition (Nearest neighbors)

X_i is said to be the k-th nearest neighbor of x if the distance $\|x - X_i\|$ is the k-th smallest distance among $\|x - X_1\|, \ldots, \|x - X_n\|$.
Ties are broken by choosing the smallest index.

Classification

Definition (Nearest neighbor classifier)

The rule is defined by

$$g_n(x) = \begin{cases} 1 \text{ if } \sum_{i=1}^n w_{ni} \mathbf{1}_{\{Y_i=1\}} > \sum_{i=1}^n w_{ni} \mathbf{1}_{\{Y_i=0\}} \\ 0 \text{ sinon,} \end{cases}$$

where $w_{ni} = \frac{1}{k}$ *if* X_i *is among the* k *nearest neighbors of* x *and* $w_{ni} = 0$ *otherwise.*

This is a majority vote among the observations that are the k nearest neighbors of x.

Regression

Definition (Nearest neighbor rule for regression)

The rule is defined by

$$m_n(x) = \sum_{i=1}^{n} w_{ni} Y_i,$$

where $w_{ni} = \frac{1}{k}$ *if* X_i *is among the* k *nearest neighbors of* x *and* $w_{ni} = 0$ *otherwise.*

Model selection

This rule allows to easily illustrate the overfitting - underfitting issue : for $k = 1$, we only consider one point, the nearest neighbor of x ; if k is large, we take almost all observations into account, which cannot bring useful information for classification or regression.

Let $g^\star$ denote the best possible classifier, defined by

$$g^\star = \underset{g:\mathbb{R}^d \to \{1,\dots,K\}}{\operatorname{argmin}} \mathbb{P}(g(X) \neq Y).$$

Recall that this classifier $g^\star$ is called Bayes classifier. Denote by $L^\star = L(g^\star)$ the Bayes error, that is the lowest possible error rate.

Let L_n denote the error of the k-nn classifier. The next result ensures that the classification rule is universally consistent.

Theorem

If $k \to \infty$ and $k/n \to 0$, then for every distribution of the random pair (X, Y), we have

$$\mathbb{E}L_n \to L^\star.$$

Kernel classifier

The k-nn idea may be modified to obtain a smoother rule. Instead of specifying a number of neighbors, the neighborhood may be defined using a distance notion.

Kernel classifier

The k-nn idea may be modified to obtain a smoother rule. Instead of specifying a number of neighbors, the neighborhood may be defined using a distance notion.

The simplest such rule is the moving window rule, where every point at distance at most r of x is considered as a neighbor of x.

Let $B(x, r)$ denote the open ball with center x and radius r.

Definition (Moving window rule)

$$g_n(x) = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} \mathbf{1}_{\{Y_i=1, X_i \in B(x,r)\}} > \sum_{i=1}^{n} \mathbf{1}_{\{Y_i=0, X_i \in B(x,r)\}} \\ 0 \text{ otherwise.} \end{cases}$$

Let $B(x, r)$ denote the open ball with center x and radius r.

Definition (Moving window rule)

$$g_n(x) = \begin{cases} 1 \text{ if } \sum_{i=1}^n \mathbf{1}_{\{Y_i=1, X_i \in B(x,r)\}} > \sum_{i=1}^n \mathbf{1}_{\{Y_i=0, X_i \in B(x,r)\}} \\ 0 \text{ otherwise.} \end{cases}$$

To obtain a smoother rule, giving more weights to the closest points and less to the furthest, we may use a kernel function K, which is a nonnegative function from $\mathbb{R}^d$ to $\mathbb{R}$, such that $K(x) = L(\|x\|)$, with $x \mapsto L(x)$ nonincreasing.

Definition (Kernel classification rule)

$$g_n(x) = \begin{cases} 1 \text{ if } \sum_{i=1}^n \mathbf{1}_{\{Y_i=1\}} K\left(\frac{x-X_i}{r}\right) > \sum_{i=1}^n \mathbf{1}_{\{Y_i=0\}} K\left(\frac{x-X_i}{r}\right) \\ 0 \text{ otherwise,} \end{cases} .$$

Taking the kernel given by $K(x) = \mathbf{1}_{\{x \in B(0,1)\}}$, we get the moving window rule.

Some kernel functions :

- Gaussian, $K(x) = e^{-\|x\|^2}$,
- Cauchy, $\frac{1}{1+\|x\|^{d+1}}$,
- Epanechnikov, $(1 - \|x\|^2)\mathbf{1}_{\{\|x\|\leq 1\}}$.

Model selection : choosing the kernel function, and the window r.

Theorem

Let K be a regular kernel, which means that there exists R such that $K(x) \geq b\mathbf{1}_{B(0,R)}(x)$ and $\int \sup_{y \in B(x,R)} K(y)dx < +\infty$. If $r \to 0$ and $nr^d \to +\infty$, then, for every distribution of (X, Y), and for every $\varepsilon > 0$, there exists n_0, such that for $n > n_0$,

$$\mathbb{P}(L_n - L^\star > \varepsilon) \leq 2e^{-n\varepsilon^2/32\rho^2}.$$

Here, ρ only depends on K and the dimension d.

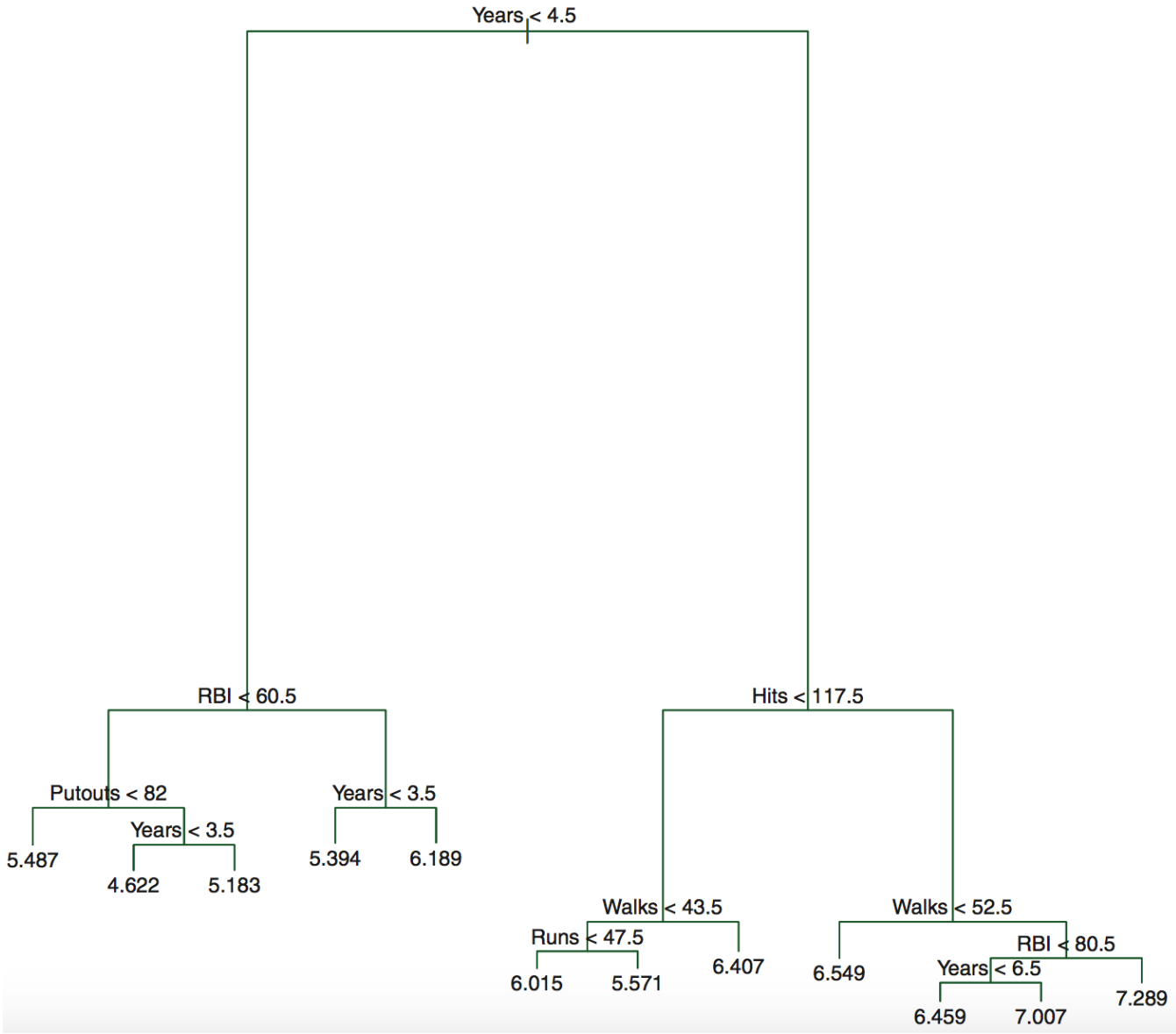
Kernel regression

Definition (Kernel regression rule)

$$m_n(x) = \frac{\sum_{i=1}^{n} K\left(\frac{x-X_i}{r}\right) Y_i}{\sum_{i=1}^{n} K\left(\frac{x-X_i}{r}\right)}.$$

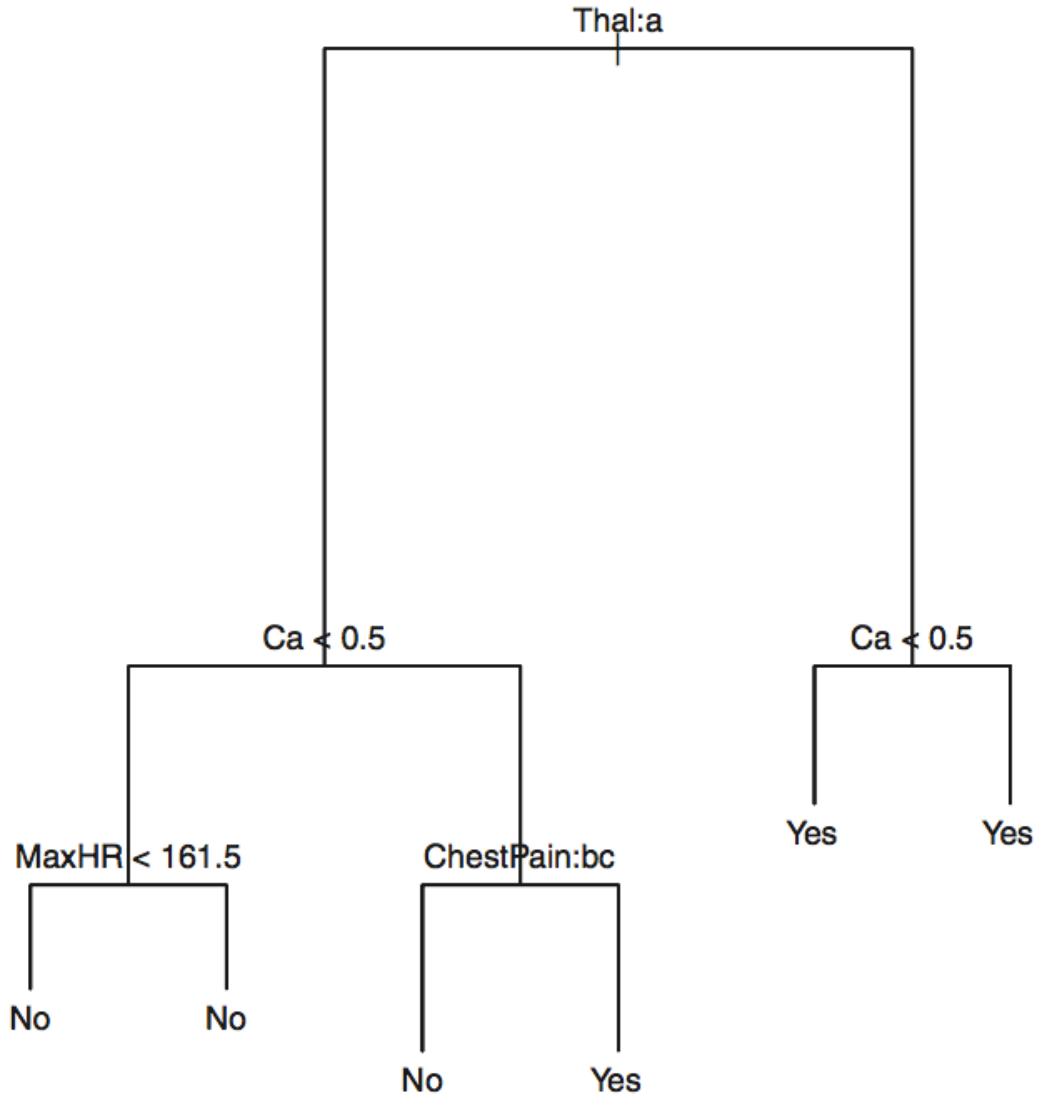
Trees

This is a regression tree

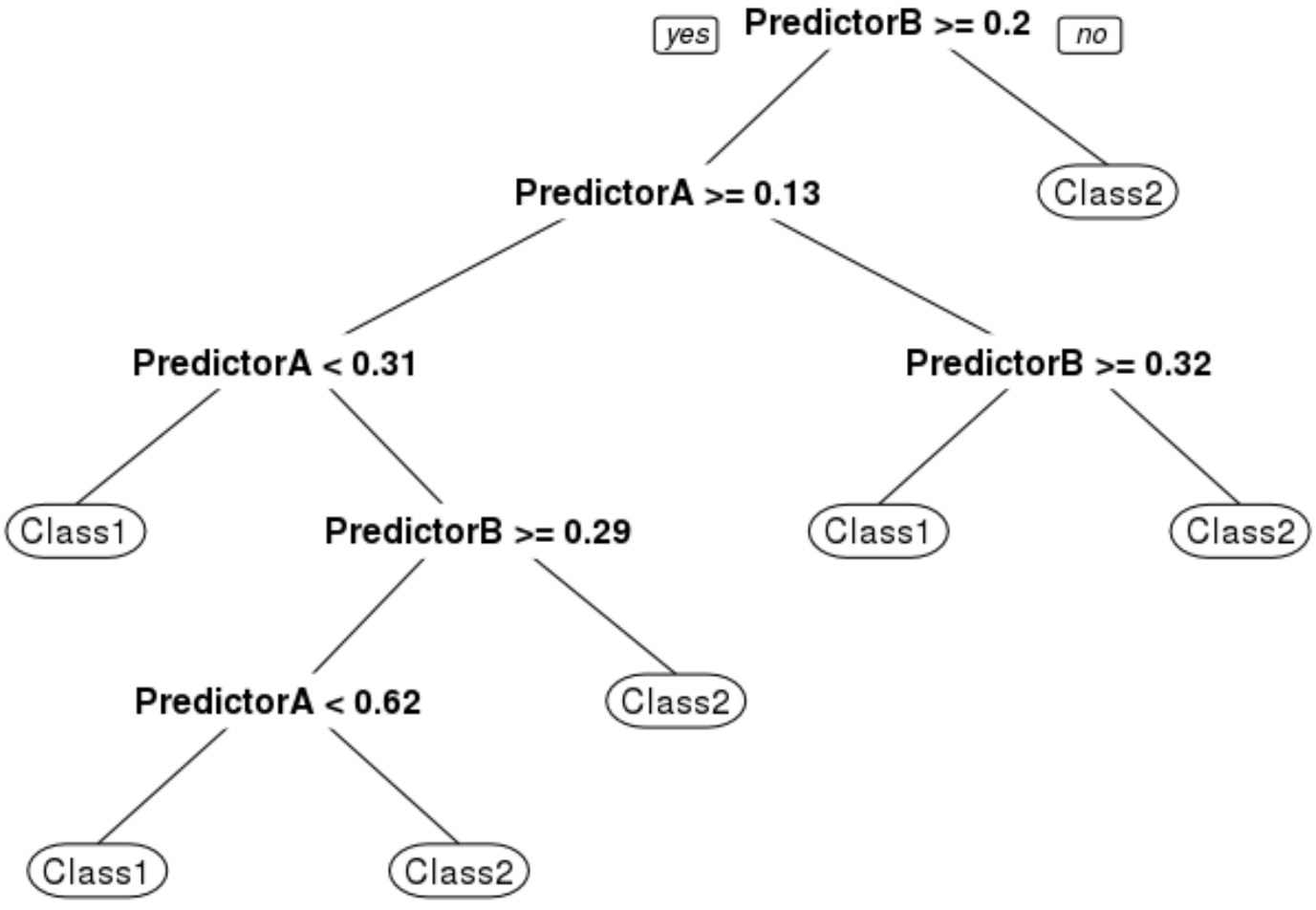


Trees

This is a classification tree



CART = Classification and Regression Trees



Tree principle

- Construct a recursive partition of rectangular domains $R_1, \ldots, R_m$ using a tree-structured set of "questions" : splits around a given value of a variable.
- Classification :
  The predicted class $\hat{y}$ is the class of the domain / leaf containing x, that is the majority class in the domain.
  It is possible to compute the class probabilities, by taking the proportion of points of class k in the domain R_ℓ :

$$\hat{p}_k(R_\ell) = \frac{1}{n_\ell} \sum_{X_i \in R_\ell} \mathbf{1}_{\{Y_i = k\}}.$$

  The predicted class is then the class which maximizes this proportion $\hat{p}_k(R_\ell)$.
- Regression : average in each domain / leaf.

Remarks

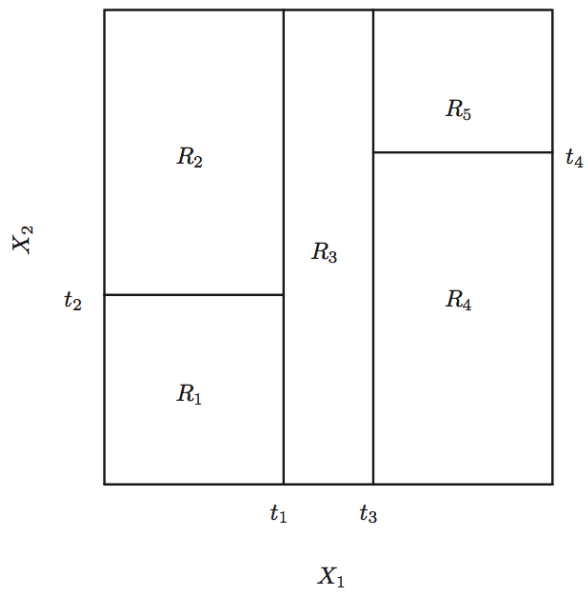
- Quality of the prediction depends on the tree / the partition.
- Issue: finding the optimal tree is hard!

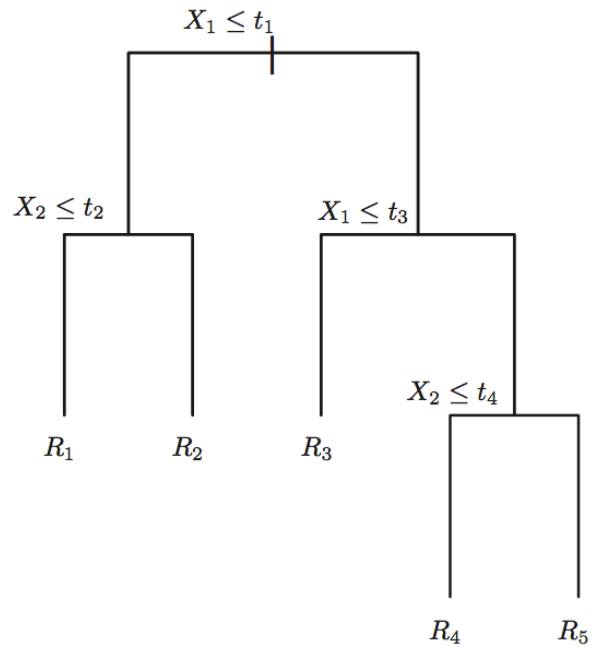
Heuristic construction of a tree

- Greedy approach = the best split point is chosen each time to create branches, in a top-down process.

Heuristic construction of a tree

- Start from a single region containing all the data (called *root*).
- Recursively splits nodes (i.e. rectangles) along a certain dimension (feature) and a certain value (threshold).
- Leads to a partition of the feature space...
- ... which is equivalent to a *binary tree*.





Remarks

- The splits, once they have appeared, remain throughout the construction of the tree.
- Heuristic: choose a split so that the two new regions are as **homogeneous** as possible...

Homogeneous means:

- For classification: class distributions in a node should be close to a Dirac mass.
- For regression: labels should be very concentrated around their mean in a node.

How to quantify **homogeneousness**?

- Variance (regression)
- Gini index, Entropy (classification)
- Among others

Splitting

- We want to split a node N into a left child node N_L and a right child node N_R.
- The childs depends on a cut: a (feature, threshold) pair denoted by (j, t).

Introduce

$$N_L(j,t) = \{x \in N : x_j < t\} \quad \text{and} \quad N_R(j,t) = \{x \in N : x_j \geq t\}.$$

Finding the best split means finding the best (j, t) pair

- Compare the *impurity* of N with the ones of $N_L(j, t)$ and $N_R(j, t)$ for all pairs (j, t)
- Using the *information gain* of each pair (j, t)

For **regression** impurity is the **variance** of the node

$$V(N) = \sum_{i:x_i \in N} (y_i - \bar{y}_N)^2 \quad \text{where} \quad \bar{y}_N = \frac{1}{|N|} \sum_{i:x_i \in N} y_i$$

with $|N| = \#\{i : x_i \in N\}$ and **information gain** is given by

$$\mathrm{IG}(j,t) = V(N) - \frac{|N_L(j,t)|}{|N|} V(N_L(j,t)) - \frac{|N_R(j,t)|}{|N|} V(N_R(j,t))$$

For **classification** with labels $y_i \in \{1, \ldots, K\}$. First, we compute the classes distribution $p_N = (p_{N,1}, \ldots, p_{N,K})$ where

$$p_{N,k} = \frac{\#\{i : x_i \in N, y_i = k\}}{|N|}.$$

Then, we can consider an impurity measure I such as:

$$G(N) = G(p_N) = \sum_{k=1}^{K} p_{N,k}(1 - p_{N,k}) \quad \text{(Gini index)}$$

$$H(N) = H(p_N) = -\sum_{k=1}^{K} p_{N,k} \log_2(p_{N,k}) \quad \text{(Entropy index)}$$

and for $I = G$ or $I = H$ we consider the information gain

$$\mathrm{IG}(j, t) = I(N) - \frac{|N_L(j,t)|}{|N|} I(N_L(j,t)) - \frac{|N_R(j,t)|}{|N|} I(N_R(j,t))$$

CART builds the partition iteratively

For each leaf N of the current tree

- Find the best (feature, threshold) pair (j, t) that maximizes $\mathrm{IG}(j, t)$
- Create the two new childs of the leaf
- Stop if some stopping criterion is met
- Otherwise continue

Stopping criterions

- Maximum depth of the tree
- All leafs have less then a chosen number of samples
- Impurity in all leafs is small enough
- Testing error is increasing
- Etc...

Remarks

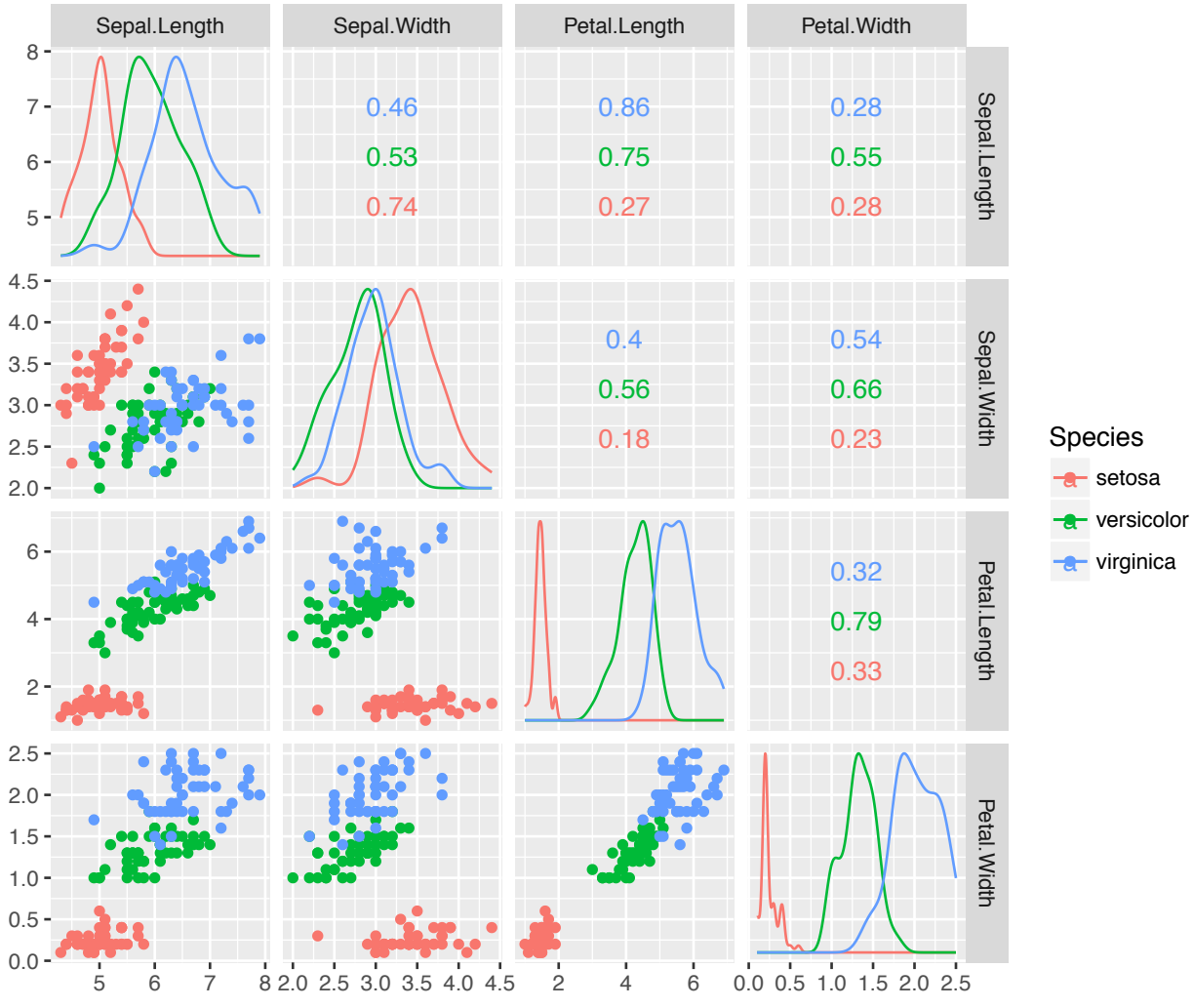
- CART with Gini is probably the most used technique
- Other criterions are possible: χ^2 homogeneity, other losses than least-squares, etc.

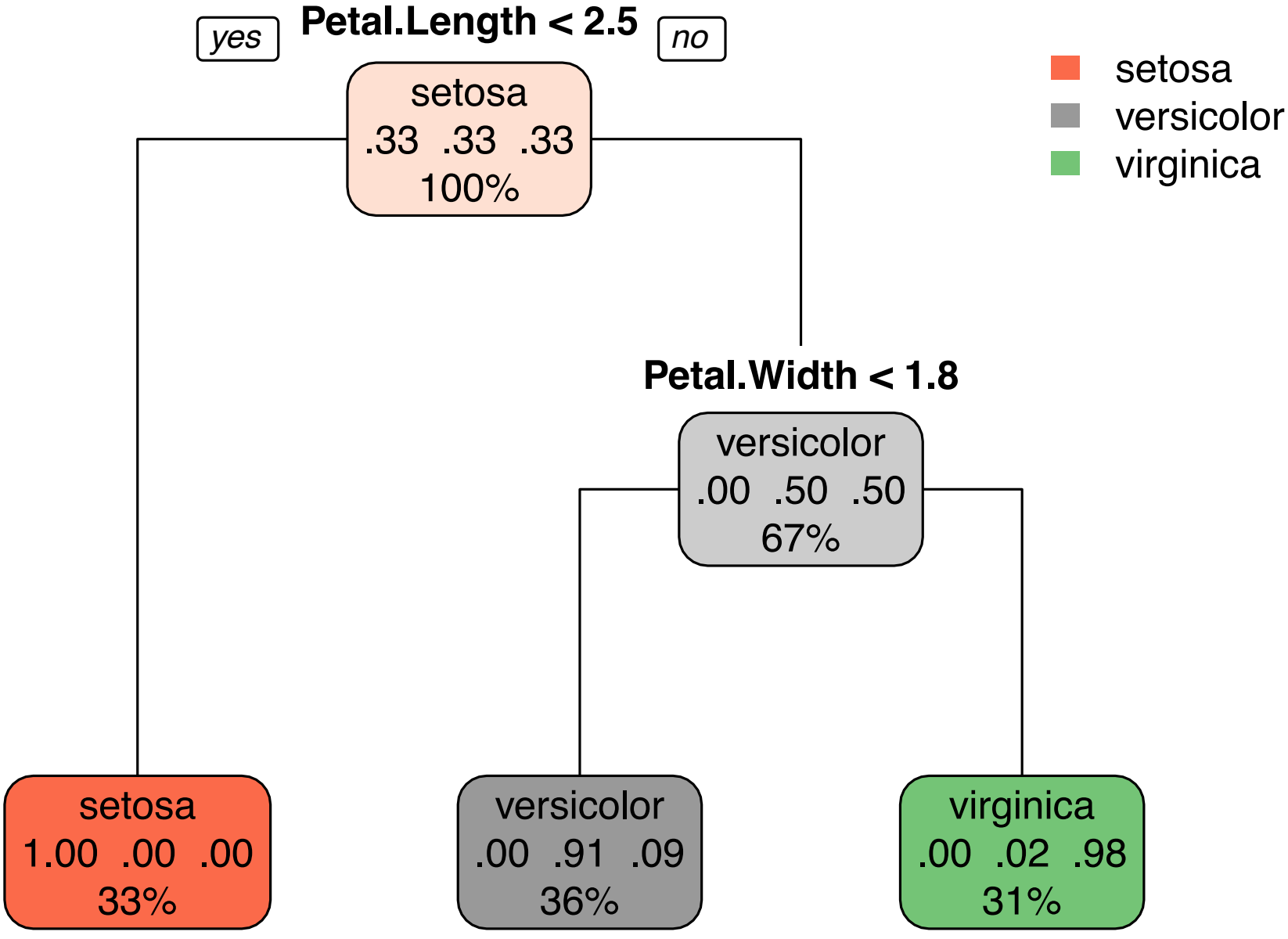
More remarks

- Usually overfits (maximally grown tree typically overfits)
- Usually leads to poor prediction
- Leads to nice interpretable results
- Is the basis of more powerful techniques: random forests, boosting (*ensemble* methods)

Example on iris dataset

- using `R` with the `rpart` and `rpart.plot` library




yes
Petal.Length < 2.5
no
setosa
.33 .33 .33
100%
Petal.Width < 1.8
versicolor
.00 .50 .50
67%
setosa
1.00 .00 .00
33%
versicolor
.00 .91 .09
36%
virginica
.00 .02 .98
31%
setosa
versicolor
virginica

Tree pruning

Tree pruning

- Bottom-up approach: try to remove leafs and all of what's below
- Number of subtrees is large, but the tree structure allows to find the best efficiently (dynamic programming)

Key idea

Given a maximally grown tree T_{max}, evaluate a subtree $T \subset T_{\text{max}}$ using

$$\sum_{\ell=1}^{|T|} \sum_{i:X_i \in R_\ell} (Y_i - \hat{Y}_{R_\ell})^2 + \alpha|T|, \sum_{\ell=1}^{|T|} (1 - \hat{p}_{C_\ell}(R_\ell)) + \alpha|T|.$$

where

- $|T|$ = number of terminal leaves in the tree
- $\alpha > 0$ is a regularization parameter

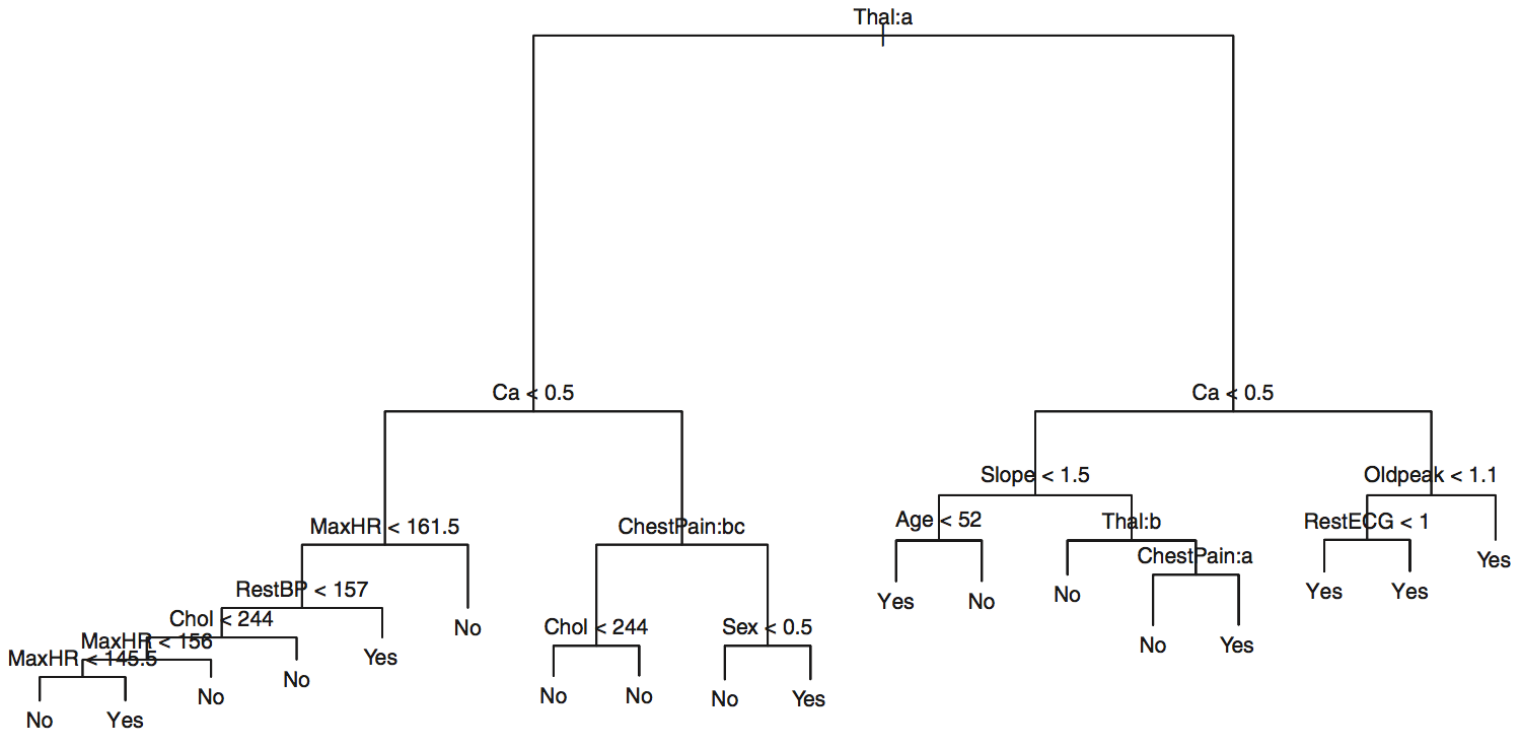
Note that $\alpha = 0$ leads to T_{max}

This means that we look for a subtree T that makes a good balance between the depth of the tree and its accuracy.

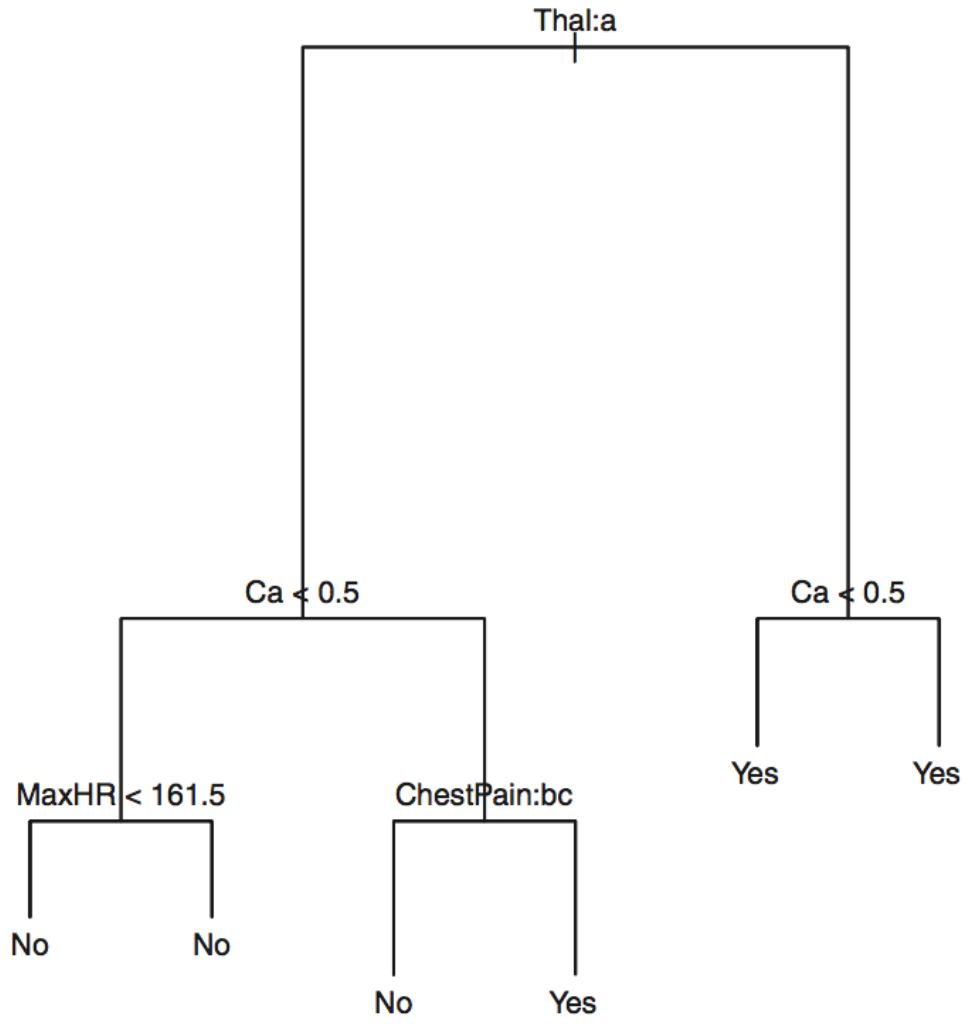
Remarks

- This allows to limit overfitting.
- α may be chosen using V-fold cross-validation.

Unpruned maximal tree



Pruned tree

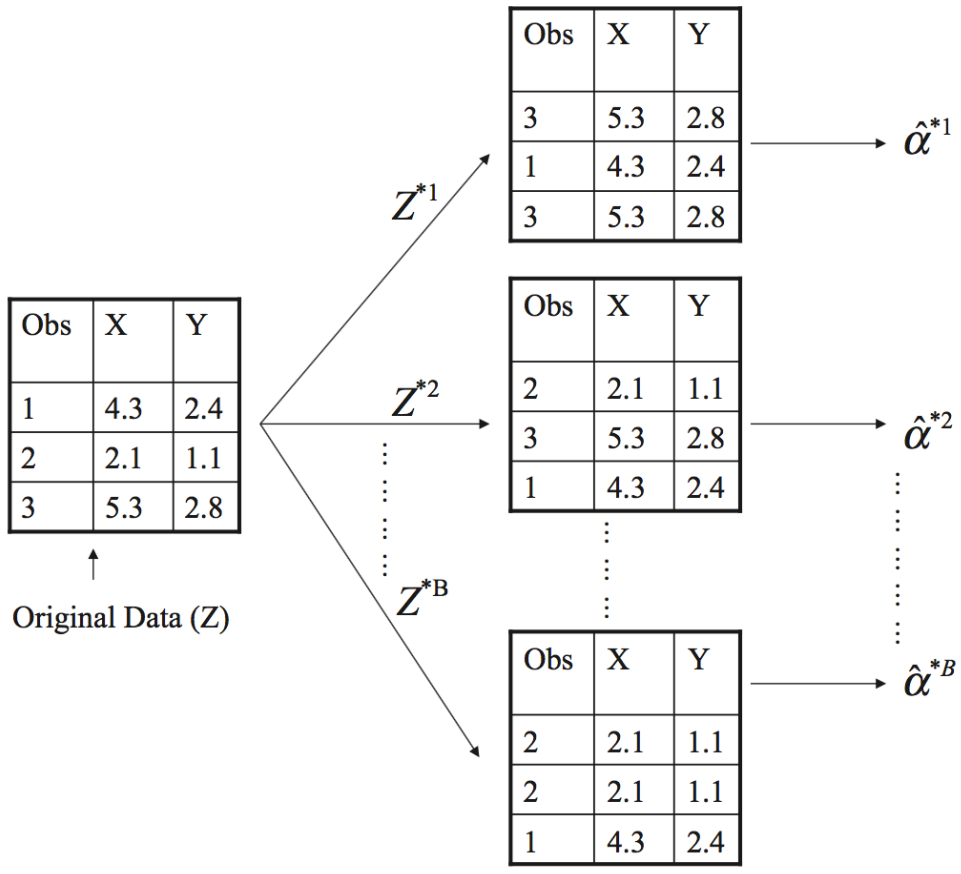


Bagging

- Decision trees suffer from high variance.
- Bagging is a general-purpose procedure to (hopefully) reduce the variance of any statistical learning method.
- Particularly useful for decision trees.
- Bagging = Bootstrap Aggregating : combine classifiers fitted on bootstrap samples.

Bootstrap

- Generation of "new" datasets



- Sampling **with replacement** from the dataset

Algorithm 1 Bagging

1: **for** $b = 1, \dots B$ **do**
2: Sample with replacement $D_n^{(b)}$ from the original dataset D_n.
3: Fit a tree $\hat{f}^{(b)}$ on $D_n^{(b)}$.
4: **end for**
5: Aggregate $\hat{f}^{(1)}, \dots, \hat{f}^{(B)}$: majority vote for classification, average for regression.

Regarding the depth, two different approaches may be used :

- the simple strategy, consisting in building trees which are potentially very large, without any pruning.
- a procedure where every tree is pruned, using the initial sample $\mathcal{D}_n$ as validation set.

How do we proceed if we want to estimate class probabilities rather than the classes themselves? One might want to consider the proportions $(p_1(x), \ldots, p_K(x))$, but they turn out not to be good estimators of class probabilities.

How do we proceed if we want to estimate class probabilities rather than the classes themselves? One might want to consider the proportions $(p_1(x), \ldots, p_K(x))$, but they turn out not to be good estimators of class probabilities.

Example

Let's consider a simple two-class example. Suppose that the true probability of class 1 is 0.75 and that the different trees all predict 1 for x. Then, $p_1(x) = 1$, which is not correct.

One solution is to directly aggregate by bagging the class probabilities rather than labels. The predicted class is the one corresponding to the highest probability.

If the class probabilities of x estimated by each of the trees are noted $\hat{p}_k^b(x)$, $k = 1, \ldots, K$, $b = 1, \ldots, B$, we calculate

$$\hat{p}_k^{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{p}_k^b(x), \quad k = 1, \ldots, K,$$

and then

$$\hat{f}^{bag}(x) = \underset{k=1,\ldots,K}{\arg\max}\, \hat{p}_k^{bag}(x).$$

In fact, this strategy is interesting for probability estimation, but also more generally, because it tends to produce classifiers with less variance, especially when B is not very large.

To understand why the technique works, consider, in a situation where there are two classes, B independent classifiers $\hat{f}^b$, $b = 1, \ldots, B$, each of which has a classification error $e = 0.4$.

Suppose, for example, that the true class of x is 1: we therefore have $\mathbb{P}(\hat{f}^b(x) = 0) = 0.4$. The number of trees B_0 voting for class 0 follows a binomial distribution $\mathcal{B}(B, 0.4)$. By definition of bagging, we have $\mathbb{P}(\hat{f}^{bag}(x) = 0) = \mathbb{P}(B_0 \geq B/2)$. As $B_0 \sim \mathcal{B}(B, 0.4)$, by the law of large numbers, B_0/B converges towards 0.4 when B tends to infinity, and thus this probability tends to 0. This means that the classifier constructed by bagging gives exact predictions when the number of bootstrap samples B tends to infinity.

On the other hand, if we consider the same context, but with a probability of error of the independent classifiers $\mathbb{P}(\hat{f}^b(x) = 0) = 0.6$, the same arguments show that $\mathbb{P}(\hat{f}^{bag}(x) = 0) = \mathbb{P}(B_0 \geq B/2)$ tends to 1, which means that the predictions of the bagging classifier become totally inaccurate when the number of bootstrap samples increases.

Warning: here, we have assumed in both cases that the individual classifiers are independent, which is not the case, since they are built on very similar samples. However, the conclusion is that a bagging strategy can improve the results of a good classifier, but can further degrade those of an already bad classifier!

Drawbacks :

- The loss of interpretability, since the resulting classifier is not necessarily a tree any more.
- Computational complexity, since we build B times a tree, maybe with pruning, with validation on the sample $\mathcal{D}_n$.

Random forests

Random forests are an alternative version of bagging. The ingredients are:

Regression or classification trees. In the random forest algorithm, these trees can be maximally grown.

Bagging: each tree is trained over a sampling with replacement of the dataset.

- Instead of a single tree, it uses an average of the predictions given by several trees: reduces noise and variance.
- Note that this strategy works all the better as the trees are less correlated.

Feature subsampling

Feature subsampling step: the particularity of random forests is that only a random subset of p features is tried each time when looking for a split.
This reduces the correlation between the trees.

Recommendations from the authors:

- Classification : default value for p is $\lfloor \sqrt{d} \rfloor$ and minimum node size is 1.
- Regression : default value for p is $\lfloor d/3 \rfloor$ and minimum node size is 5.

In practice, best values depend on the problem : tuning parameters.

Out of bag samples

For each observation (X_i, Y_i), on may construct the random forest predictor corresponding to the average of the trees built thanks to bootstrap samples in which this observation does not appear.

Computing the OOB error is similar to performing cross validation and allows to tune the parameters. This is a very interesting feature of random forests.

Variable importance

Regarding interpretability, once again, the resulting estimator is not a tree. Nevertheless, random forests come along with variable importance plots, yielding very useful information.

To compute variable importance:

- At each split in each tree, the improvement in the split-criterion is attributed to the splitting variable, and then, for each variable, the values are accumulated over all the trees in the forest.
- A different variable importance measure, based on the OOB samples.
  When the b-th tree is grown, the prediction accuracy is compared with the prediction accuracy when the values for the j-th variable are randomly permuted in the OOB samples. The decrease in accuracy due to this permuting is averaged over all trees, and is used as a measure of the importance of variable j in the random forest.

Some remarks on the method

- Usually, trees are not pruned in random forests. Indeed, the gain of pruning does not seem crucial, and hence, this choice allows to save a tuning parameter.
- When the number of variables is large, but the fraction of relevant variables small, random forests do not perform very well for small p, since, at each split, the chance that the relevant variables will be selected is small.

Random forest algorithm

1. For $b = 1, \ldots, B$,
   1.1 Draw a bootstrap sample $\mathcal{D}_m^b$ of size m from the sample $\mathcal{D}_n$.
   1.2 Build a tree $\hat{f}^b$ on $\mathcal{D}_m^b$ as follows :
   - Draw at random p variables among d.
   - Select the best split among these p variables.
2. Combine the predictions of all B trees by a majority vote or an average.

A variant of random forests, ExtraTrees

ExtraTrees: extremely randomized trees.

Here, trees are built on the initial sample $\mathcal{D}_n$, no bootstrap.

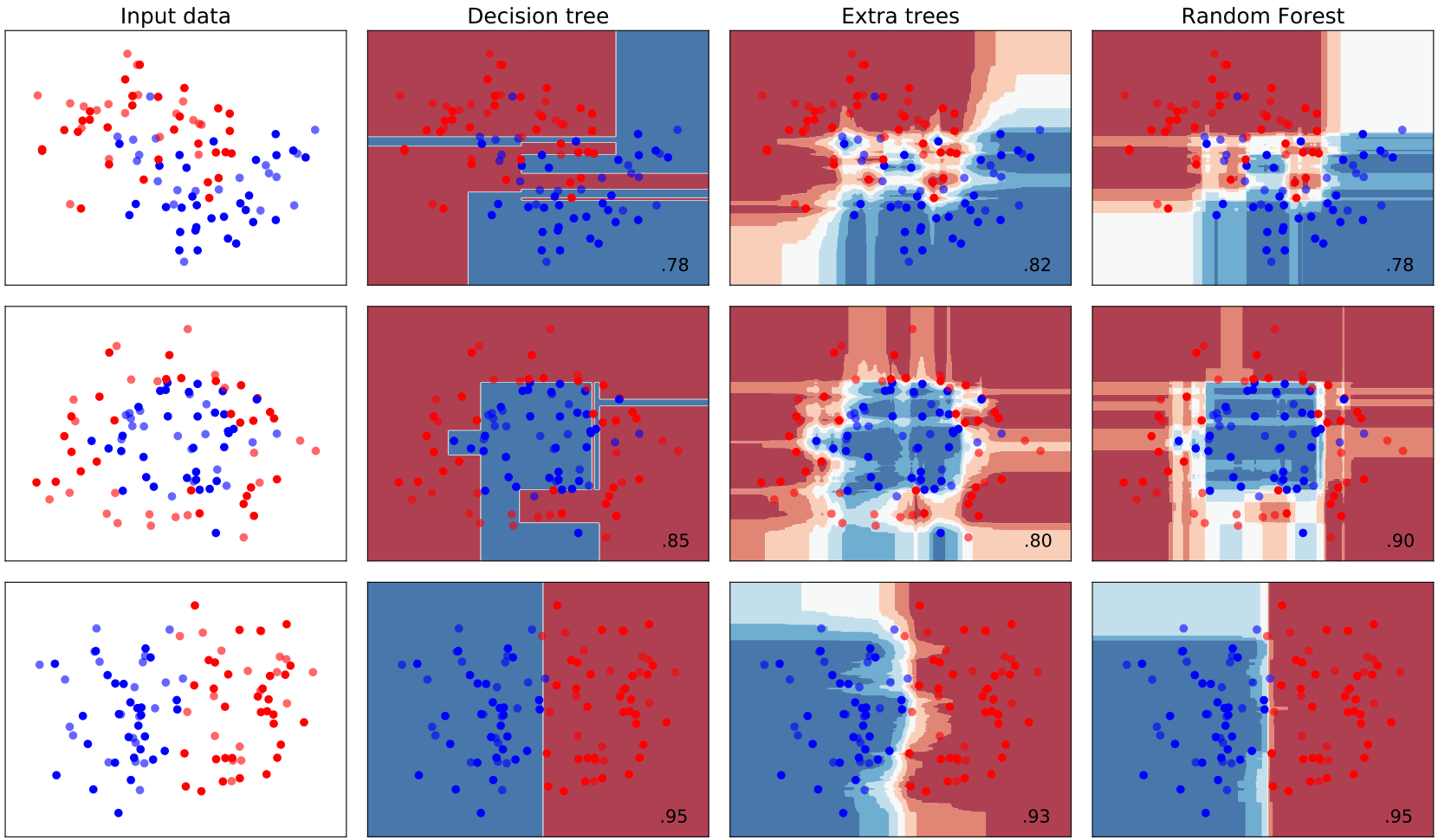
- A random subset of features is drawn as for random forests.
- A few number of thresholds is then selected uniformly at random in the feature range.
- Finally, the best split among these thresholds and features is selected using some impurity criterion (Gini, Entropy).

Remark

- Due to this random choice, the algorithm is faster.
- Random thresholds lead to more diversified trees, and thus, are some form of regularization.
- Since trees are combined and averaged, not so bad to select the thresholds at random...

Using `scikit-learn`: decision functions of

- `tree.DecisionTreeClassifier`
- `ensemble.{ExtraTreesClassifier,RandomForestClassifier}`



Boosting

Boosting is another ensemble method, which differs from bagging and random forests by the fact that, this time, the estimators to be combined are different in nature : they are not any more "equivalent" objects built thanks to bootstrap.

- Features $X_i \in \mathbb{R}^d$.
- Labels $Y_i \in \{-1, 1\}$ for binary classification and $Y_i \in \mathbb{R}$ for regression.

- Finite set of "weak estimators", very simple.
- Classification : slightly better than a coin toss (classification error $< 1/2$).
- Regression : slightly better than average of all Y_i's.

Examples:

- **Trees:** One-level decision tree, also called decision stump : there is only one node, the root, which is immediately connected to the leaves. The prediction is made based on the value of a single input feature.
- **Other estimators:** linear model (regression) or logistic regression (classification), using one or two features.

A stump

Boosting principle

The most widely used algorithm is AdaBoost: ADAptive BOOSTing.

AdaBoost.M1 or discrete AdaBoost for binary classification.

General idea : produce iteratively a sequence of weak estimators, to be combined linearly in order to get the final prediction

$$f^{boost}(x) = \text{sign}\left(\sum_{b=1}^{B} \beta_b f^b(x)\right).$$

Here, the weights $\beta_1, \ldots, \beta_B$ are computed by the boosting algorithm, and weight the contribution of the $f^b(x)$. The goal is to give higher influence to the more accurate classifiers in the sequence.

How to choose the weights ? How to build adaptively the weak estimators ?

The data modifications at each boosting step consist of applying weights $w_1, \ldots, w_n$ to the training observations (X_i, Y_i), $i = 1, 2, \ldots, n$.

- Initially, all weights are set to $1/n$.
- Then, for $b = 2, 3, \ldots, B$ the observation weights are individually modified and the classification algorithm is reapplied to the weighted observations.
- At step b, the observations that were misclassified at the previous step have their weights increased, whereas the weights are decreased for observations that were classified correctly. Hence, progressively, observations that are difficult to classify correctly receive ever-increasing influence.

AdaBoost Algorithm

1. The weights w_i are intialized to $w_i^1 = 1/n$ for every i.
2. Fo every $b = 1, \ldots, B$:
   - 2.1 Build a classification tree f^b for the data with the weights $w_1^{(}b), \ldots, w_n^{(}b)$.
   - 2.2 Compute the weighted classification error

$$e_b = \frac{\sum_{i=1}^n w_i^b \mathbf{1}_{\{Y_i \neq f^b(X_i)\}}}{\sum_{i=1}^n w_i^b}.$$

   - 2.3 Define β_b by

$$\beta_b = \ln\left(\frac{1 - e_b}{e_b}\right).$$

   - 2.4 Update of the weights : for every i,

$$w_i^{(b+1)} \leftarrow w_i^{(b)} \exp(\beta_b \mathbf{1}_{\{Y_i \neq f^b(X_i)\}}).$$

3. Compute the boosting classifier

$$f^{boost}(x) = \text{sign}\left(\sum_{b=1}^{B} \beta_b f^b(x)\right).$$

Example to get insight

Let us illustrate that AdaBoost is able to increase the performance of even a very weak classifier.

Consider $X_1, \ldots, X_{10}$ standard independent Gaussian, and

$$Y = \begin{cases} 1 \text{ if } \sum_{j=1}^{10} X_j^2 > \chi^2_{10}(0.5) \\ -1 \text{ otherwise.} \end{cases}$$

Here, $\chi^2_{10}(0.5) = 9.34$ is the median of a chi-squared random variable with 10 degrees of freedom.

Consider 2000 training observations, with approximately 1000 in each class, and 10 000 test observations.

As weak classifier, we use a stump. Applying this classifier alone to the training data set yields a very poor test set error : 45.8%, so almost as bad as random guessing.

Thanks to boosting iterations, the error rate decreases, reaching 5.8% after 400 iterations.

It also outperforms a single large classification tree (error rate 24.7%).

General idea

In fact, AdaBoost may be seen as a particular case of a bunch of procedures called gradient boosting, based on functional gradient descent.

The overall operation consists in looking for $f^1, \ldots, f^B \in \mathscr{F}$ and $\alpha_1, \ldots, \alpha_B \in \mathbb{R}$ such that

$$f^{boost}(x) = \sum_{b=1}^{B} \alpha_b f^b(x)$$

minimizes an empirical risk

$$\frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f^{boost}(X_i)),$$

where ℓ is a loss (least-square, logistic, etc.).

Numerical optimization via Gradient Boosting

At step $b+1$, look for f_{b+1} and α_{b+1} such that

$$(\alpha_{b+1}, f_{b+1}) = \underset{\alpha \in \mathbb{R}, f \in \mathscr{F}}{\operatorname{argmin}} \sum_{i=1}^{n} \ell(Y_i, f_b^{boost}(X_i) + \alpha f(X_i))$$

and put

$$f_{b+1}^{boost} = f_b^{boost} + \alpha_{b+1} f_{b+1}.$$

- Exact minimization at each step is too hard.
- Replace exact minimization by a gradient step.

Compute the negative gradient $-\frac{\partial}{\partial f}\ell(Y, f)$ and evaluate at $f_b^{boost}(X_i)$:

$$\hat{\delta}_{b,i} = -\frac{\partial \ell(Y_i, f(X_i))}{\partial f(X_i)}\bigg|_{f(X_i)=f_b^{boost}(X_i)}, \quad i = 1, \ldots, n.$$

If $\ell(y, z) = \frac{1}{2}(y - z)^2$, this is given by

$$\hat{\delta}_b = -\Big(f_b^{boost}(X_1) - Y_1, \ldots, f_b^{boost}(X_n) - Y_n\Big).$$

Minimization is approximated by a step in the negative gradient direction.

These gradients might not be included in the constraint space $\mathscr{F}$ (in particular, sum of trees), and this steps dot not provide functions able to generalize: the next task is to approximate the negative gradients using elements of $\mathscr{F}$.

We look for a weak learner whose predictions are as close as possible to the negative gradient :

$$(f, \nu) = \underset{f \in \mathscr{F}, \nu \in \mathbb{R}}{\operatorname{argmin}} \sum_{i=1}^{n} \left(\nu f(X_i) - \hat{\delta}_{b,i} \right)^2.$$

For least-squares, this means

$$(f, \nu) = \underset{f, \nu}{\operatorname{argmin}} \sum_{i=1}^{n} \left(\nu f(X_i) - f_b^{boost}(X_i) + Y_i \right)^2,$$

meaning that we look for a weak learner that corrects the current residuals obtained from the current f_b^{boost}.

Gradient boosting algorithm

1. Initialize $f_0^{boost} = \text{argmin}_{\gamma \in \mathbb{R}} \sum_{i=1}^n \ell(Y_i, \gamma)$.
2. For $b = 1, \ldots, B$
   - $\hat{\delta}_b \leftarrow -\frac{\partial \ell(Y_i, f(X_i))}{\partial f(X_i)}\Big|_{f(X_i)=f_b^{boost}(X_i)}, i = 1, \ldots, n$
   - $f^{b+1} \leftarrow f$ with

$$(f, \nu) = \underset{f,\nu}{\text{argmin}} \sum_{i=1}^n \left(\nu f(X_i) - \hat{\delta}_{b,i}\right)^2.$$

   - $\alpha_{b+1} \leftarrow \text{argmin}_\alpha \sum_{i=1}^n \ell(Y_i, f_b^{boost}(X_i) + \alpha f^{b+1}(X_i))$

   Return a boosting learner $f^{boost}(x) = \sum_{b=1}^B \alpha_b f^b(x)$.

Remark

In practice, the step size in the gradient descent may be fixed.

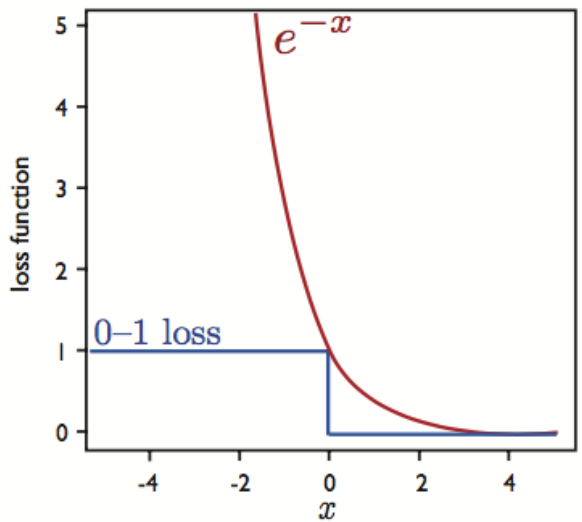
Back to AdaBoost

- Gradient boosting with the exponential loss $\ell(y, z) = e^{-yz}$ (binary classification).
- Construct recursively

$$f_{b+1}^{boost}(x) = f_b^{boost}(x) + \alpha_{b+1} f^{b+1}(x)$$

where

$$(f^{b+1}, \alpha_{b+1}) = \underset{f,\alpha}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} e^{-Y_i(f_b^{boost}(X_i) + \alpha f(X_i))}$$



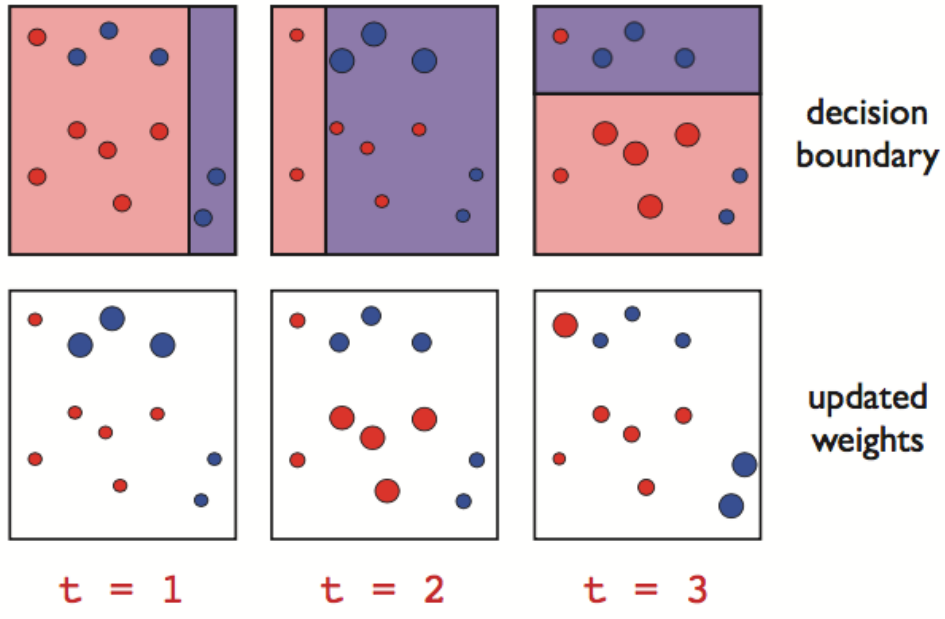
As already mentioned, AdaBoost was originally constructed in order to over-weights miss-classified samples.

At each iteration, we want to

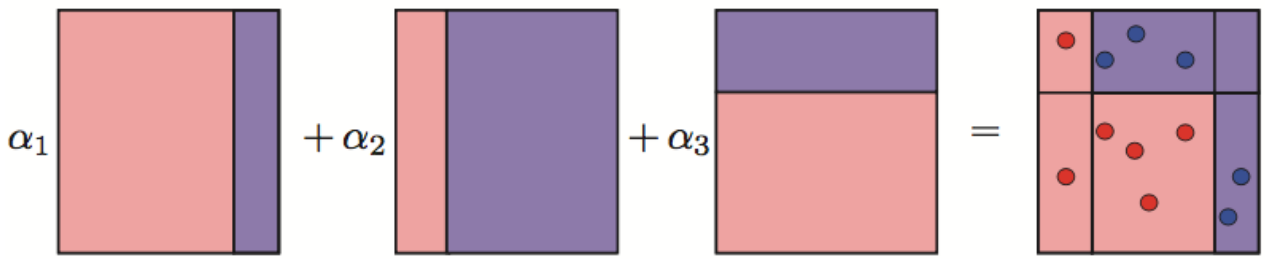
- Select a weak learner that improves the current fit.
- Focus on sample points that are not well-fitted.
- Combine each step in a meaningful way.

A mental picture

When weak learners are 1-split decision trees (stumps)



(a)



(b)

The problem

$$(f^{b+1}, \alpha_{b+1}) = \underset{f,\alpha}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} e^{-Y_i(f_b^{boost}(X_i)+\alpha f(X_i))}$$

may be rewritten

$$(f^{b+1}, \alpha_{b+1}) = \underset{f,\alpha}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} w_i^{(b)} e^{-Y_i \alpha f(X_i)},$$

where $w_i^{(b)} = \exp\left(-Y_i f_b^{boost}(X_i)\right)$, $1, \ldots, n$. These weights do not depend on α or f.

Note that

$$\sum_{i=1}^{n} w_i^{(b)} e^{-Y_i \alpha f(X_i)} = e^{-\alpha} \sum_{Y_i = F(X_i)} w_i^{(b)} + e^{\alpha} \sum_{Y_i \neq F(X_i)} w_i^{(b)}$$

$$= (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{n} w_i^{(b)} \mathbf{1}_{\{Y_i \neq f(X_i)\}} + e^{-\alpha} \sum_{i=1}^{n} w_i^{(b)},$$

where only the quantity $\sum_{i=1}^{n} w_i^{(b)} \mathbf{1}_{\{Y_i \neq f(X_i)\}}$ depends on f.

Differentiating in α leads to

$$(e^{\alpha} + e^{-\alpha}) \sum_{i=1}^{n} w_i^{(b)} \mathbf{1}_{\{Y_i \neq f(X_i)\}} - e^{-\alpha} \sum_{i=1}^{n} w_i^{(b)}.$$

Then

$$(e^{\alpha} + e^{-\alpha}) \sum_{i=1}^{n} w_i^{(b)} \mathbf{1}_{\{Y_i \neq f(X_i)\}} = e^{-\alpha} \sum_{i=1}^{n} w_i^{(b)}$$

is equivalent to

$$\frac{e^{\alpha} + e^{-\alpha}}{e^{-\alpha}} = \frac{\sum_{i=1}^{n} w_i^{(b)}}{\sum_{i=1}^{n} w_i^{(b)} \mathbf{1}_{\{Y_i \neq f(X_i)\}}},$$

i.e.

$$e^{2\alpha} = \frac{\sum_{i=1}^{n} w_i^{(b)}}{\sum_{i=1}^{n} w_i^{(b)} \mathbf{1}_{\{Y_i \neq f(X_i)\}}} - 1.$$

2 steps

- Hence, for any value of $\alpha > 0$, we may optimize the above quantity in f by minimizing

$$\sum_{i=1}^{n} w_i^{(b)} \mathbf{1}_{\{Y_i \neq f(X_i)\}}.$$

  In other words, we look for the classifier minimizing the weighted error rate in predicting y.

- Optimization in α then leads to

$$\alpha_b = \frac{1}{2} \ln \left(\frac{1 - e_b}{e_b} \right),$$

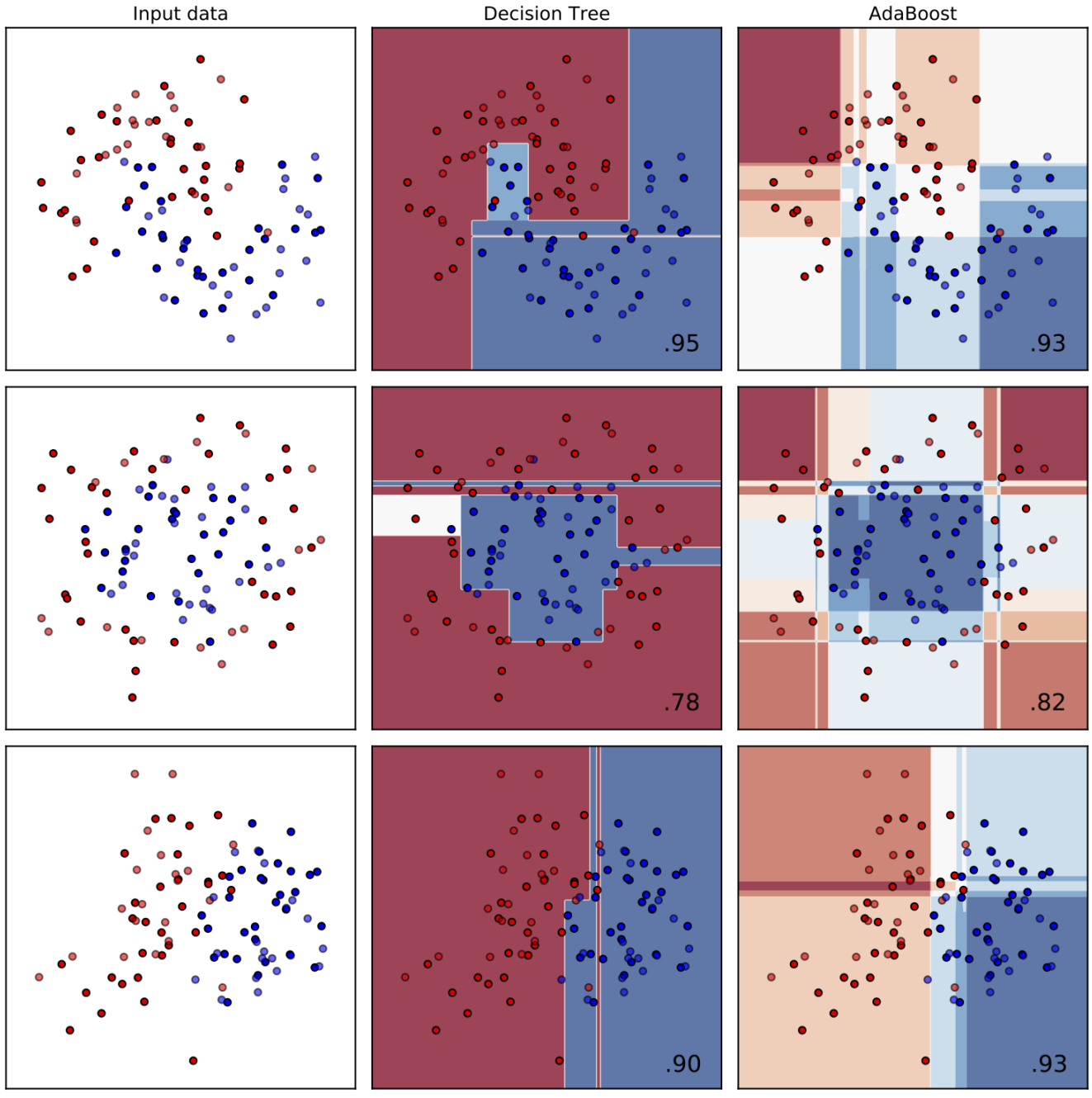
  where

$$e_b = \frac{\sum_{i=1}^{n} w_i^{(b)} \mathbf{1}_{\{Y_i \neq f^b(X_i)\}}}{\sum_{i=1}^{n} w_i^{(b)}}.$$

Consequently, updating $f_{b+1}^{boost}(x) = f_b^{boost}(x) + \alpha_{b+1} f^{b+1}(x)$, we get for the weights

$$\begin{aligned} w_i^{(b+1)} &= w_i^{(b)} e^{-\alpha_b Y_i f^b(X_i)} \\ &= w_i^{(b)} \exp(\beta_b \mathbf{1}_{\{Y_i \neq f^b(X_i)\}}) e^{-\alpha_b}, \end{aligned}$$

since $-Y_i f^b(X_i) = 2\mathbf{1}_{\{Y_i \neq f^b(X_i)\}} - 1$.
Note that the factor multiplies all weights by the same value, so that it has not effect. So, we finally obtain the AdaBoost algorithm.


Input data
Decision Tree
AdaBoost
.95
.93
.78
.82
.90
.93

Choice of hyper-parameters

- For trees, choose their depth (usually 1, 2 or 3).
- For generalized linear models (logistic regression), select the number of features (usually one or two).
- Number of iterations B (usually a few hundreds, few thousands, until test error does not improve).

Regularization

- Add a regularization factor

$$f_{b+1}^{boost} = f_b^{boost} + \beta\alpha_{b+1}f^{b+1}$$

  where β chosen using cross-validation.

Check `XGBoost`'s documentation for more details on the hyper-parameters

State-of-the-art implementations of Gradient Boosting

- With many extra tricks...

XGBoost

- `https://xgboost.readthedocs.io/en/latest/`
- `https://github.com/dmlc/xgboost`

LightGBM

- `https://lightgbm.readthedocs.io/en/latest/`
- `https://github.com/Microsoft/LightGBM`