Tree-based methods - Basic

Réda Arab

1 Introduction - Decision-trees

Goal : Stratifying or segmenting the predictor space into a number of simple regions : *decision-tree methods*.

- Simple and useful for interpretation
- Not competitive with the best supervised learning approaches in terms of prediction accuracy.

We will discuss *bagging, random forests* and *boosting*. Theses methods grow multiple trees which are then combined to yield a single consensus prediction. Combining a large number of trees can often result in dramatic improvements in *prediction accuracy*.

These methods can be used both for *regression* and *classification*.

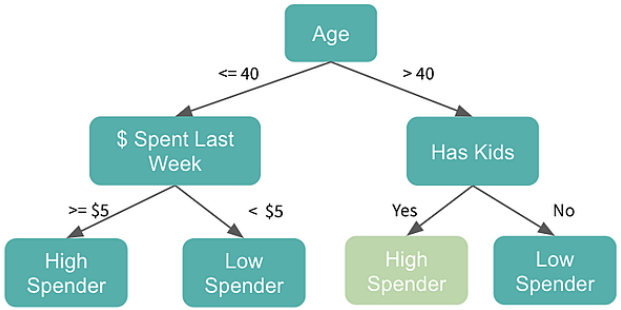

FIGURE 1 – Example of a decision tree for a classification task

2 Structure - Decision-trees

The final regions at the end are the *terminal nodes* or *leaf nodes*.

The points along the tree where the predictor space is split are referred as *internal nodes*.

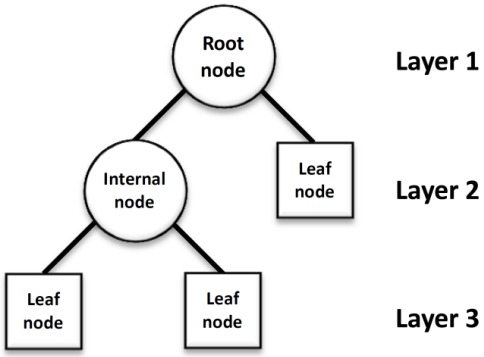

FIGURE 2 – General structure

3 Regression trees

We focus on this section on regression problems.

Process

1. We divide the predictor space into J distinct and non-overlapping regions $R_1, R_2, \ldots, R_J$
2. For every observation that falls into R_j, we make the same prediction : the mean of the response values for the training observations in R_j

In theory, the regions could have any shapes. However, we choose to **divide into high-dimension rectangles or boxes** for simplicity and interpretability.

The problem can be summarized as finding the boxes $R_1, \dots, R_J$ that minimize the RSS (Residual Sum of Squares) :

$$\sum_{j=1}^{J} \sum_{i; x_i \in R_j} (y_i - \hat{y}_{R_j})^2$$

where y_{R_j} is the mean response on the region R_j

However, it is **computationally infeasible to consider every possible partition** of the feature space into J boxes. So we take a **top-down, greedy** approach : *recursive binary splitting*.

Recursive binary splitting

General idea : We begin at the top of the tree and then successively split the predictor space. The algorithm is greedy because the best split is made at each step and we do not try to find the best global solution.

Algorithm : At each step, we try all the possible splits for each feature (if we have n datapoints, we only need $n-1$ splits per feature) and we select the predictor X_j and the cutpoint s which leads to **the greatest reduction in RSS**.

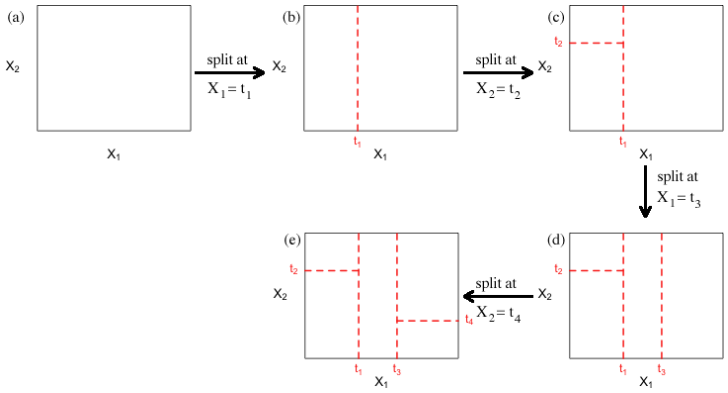

FIGURE 3 – Example of the recursive binary splitting with two features

The prediction for a given test observation is therefore the **mean response of the region in which that observation belongs**.

When do we stop growing the trees ?

If we make the tree as large as possible, we will face *overfitting*. If each region has few terminal nodes (for example 1), it tends to overfit.

Smaller trees with fewer splits might lead to *lower variance* and *better interpretation* (at the cost of a little bias).

One possible strategy : grow the tree as long as the decrease in the RSS exceeds some threshold. However, a 'bad' split can be followed by a very good split. So we use another approach known as *pruning*.

Pruning

Grow a very large tere T_0, and prune it back in order to obtain a subtree. Cost complexity pruning, also known as weakest link pruning, is used to do this.

For each nonnegative parameter α, there is a subtree T such that

$$\sum_{m=1}^{|T|} \sum_{i: x_i i n R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

is minimized. |T| is the number of terminal nodes of the tree T (refers to the complexity of the tree). We choose the parameter α by *cross-validation*.

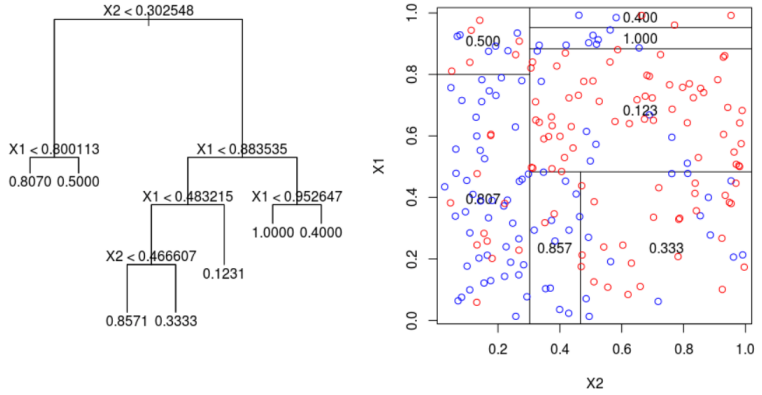

FIGURE 4 – Example of a decision tree for regression

4 Classification trees

They are similar to regression trees in the structure and we build them in the same way.

For each observation, we predict that it belongs to **the most commonly occurring class in the region** in which the observation belongs.

We do not use the RSS as a loss of function but the Gini Impurity or the Cross-entropy / Deviance.

Gini Impurity

For K classes, it is defined on the region R_m as

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

where $\hat{p}_{mk}$ is the proportion of training examples in the region m which are in the k^{th} class.

It measures the total variability accross the K classes. It takes small values when the $\hat{p}_{mk}$ are close to 0 or 1. A small value indicates that a class is predominant in the region.

Cross-entropy / Deviance

For K classes, it is defined on the region R_m as

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \cdot log(\hat{p}_{mk})$$

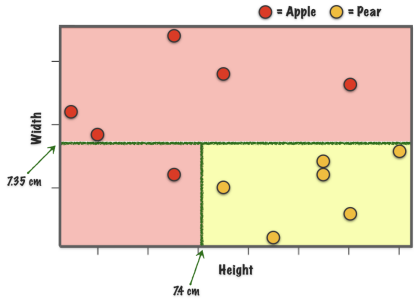

FIGURE 5 – Example of a classification tree

Trees are not always the best thing to do as we can see in the example below :

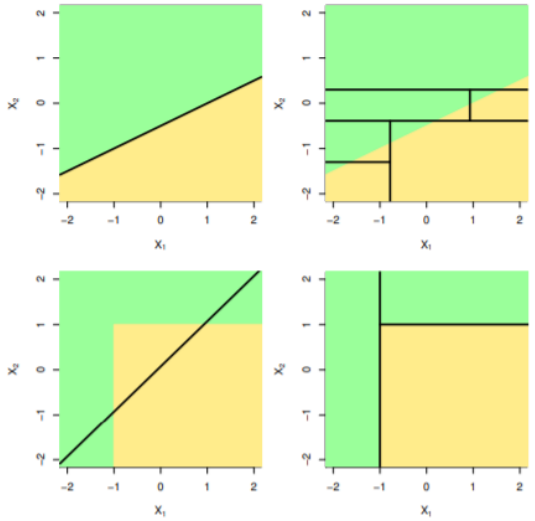

FIGURE 6 – Trees vs linear models

Decision-trees are easy to explain and interpret, can be displayed graphically, handle qualitative predictors and to some extent mimic human-decision making. However, the prediction accuracy not as good as some other methods. It is why we will see methods which consist in **aggregating many decisions trees in order to improve the prediction accuracy**.

5 Bagging

Bagging (***b**ootstrap **agg**regat**ing***) is a general procedure for **reducing the variance of a statistical learning method**. It is useful and used frequently in the context of decision trees.

Method for regression

— From a original training set D, we generate B training sets $D_1, \ldots, D_B$ of the same size of D, sampled from D with replacement.

— Then, we train our method (decision-tree) on each new datasets and we get $\hat{f}^{*b}(x)$, the prediction at a point x, for $b = 1, \ldots, B$

— We average all the prediction to get :

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

For classification, we use **majority vote** instead of averaging among the B trees.

Out-of-bag error estimation

There is a straightforward way to **estimate the test error of a bagged model**.

Suppose $D = \{x_1, \ldots, x_n\}$. For a point x_i, the probability to be in a new dataset is $1 - \left(1 - \frac{1}{n}\right)^n$ which is near $1 - e^{-1} \approx 0.63$ as n increases (and we tend to be near this value very fast).

Therefore, on average, each observation will be in appoximately 2B/3 bagged trees. For a bagged tree, the observations which are not in the bootstrap subset of the observations are referred to as the out-of-bag (OOB) observations.

Taking all the trees in which observation i was OOB, we can predict the response by averaging (or majority vote for classification).

6 Random forests

Random forests provide an improvement over bagged trees by a small tweak that **decorrelates the trees**. This reduces the variance when we average the trees.

The difference is just the following : each time we split, **we select randomly m predictors among M** to be taken into consideration for the splitting. A typical choice of m is $\sqrt{M}$. We can also choose m via out-of-bag error.

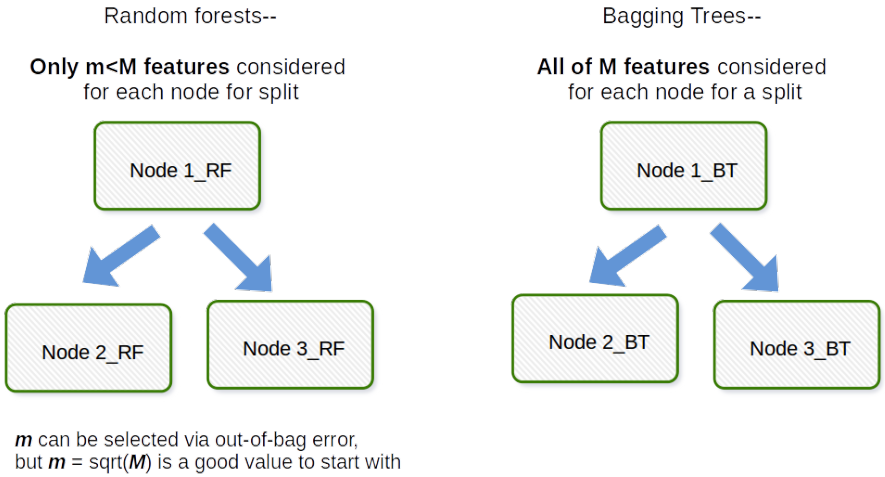

FIGURE 7 – Bagging and random forests

7 Boosting

Originally, *boosting* was born due to the question asked by Michael Kearns in 1988 : ***can we combine weak learners to generate a strong learner ?***

Robert Schapire in 1990 gives an answer : **create an ensemble classifier, in an iterative way**

$$\boxed{H_T(x) = \sum_{i=1}^{n} \alpha_t h_t(x)}$$

Difference with bagging and random forests

It is also an ensemble method which combines many learning algorithms but, comparing to bagging and random forest, we are doing it in an iterative way with each learning algorithm taking into consideration the previous ones.

One difference we have to take into account is that boosting can overfit if the number of trees is too large.

Method

We start from 0 and at each step t, we add the classifier $\alpha_t h_t(x)$ to the ensemble. We choose the new classifier in a clever way, suited for our problem.

We learn step-by-step, slowly and at the end, we evaluate all the classifiers and return the weighted sum.

Now, we can use boosting for regression problems also.

General mathematical formulation

We have a loss function l, *convex* and *differentiable*, that we want to minimize and of the form $\boxed{l(H) = \frac{1}{n}\sum_{i=1}^{n} l(H(x_i), y_i)}$.

At each step t , for a given α, we want to solve the following optimization problem :

$$\boxed{\underset{h \in H}{\operatorname{argmin}}\, l(H_t + \alpha h)}$$

Then, we update H : $H_{t+1} := H_t + \alpha h_{t+1}$. I.e we want to find which quantity/function we have to add to reduce our loss function as much as possible.

Taylor's formula gives us, for α sufficiently small :

$$l(H + \alpha h) \approx l(H) + < \nabla l(H), \alpha h >$$

Seeing $l(H)$ as $l(H(x_1), H(x_2), \ldots, H(x_n))$, we get :

$$\underset{h \in H}{\operatorname{argmin}}\, l(H_t + \alpha h) \approx \underset{h \in H}{\operatorname{argmin}} < \nabla l(H), \alpha h >= \underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^{n} \frac{\partial l}{\partial H(x_i)} \cdot \alpha h(x_i)$$

So we can do boosting as long as we can solve :

$$\boxed{\underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^{n} \frac{\partial l}{\partial H(x_i)} \cdot h(x_i)}$$

Remark : We make progress, i.e our loss function decreases, as long as the scalar product is lower than 0.

Example If $l(H) = \frac{1}{2}\sum_{i=1}^{n}(H(x_i) - y_i)^2$, we have $\frac{\partial l}{\partial H(x_i)} = H(x_i) - y_i$. Therefore, having the dot product negative implies going nearer to y from H.

Given an algorithm $\mathbb{A}$ which can solve

$\mathbb{A}(\{(x_1, r_1), \ldots, (x_n, r_n)\}) = \underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^{n} \frac{\partial l}{\partial H(x_i)} \cdot h(x_i)$, we can write the following pseudo-code.

Generic boosting (a.k.a Anyboost)

Input: ℓ, α, $\{(\mathbf{x}_i, y_i)\}$, $\mathbb{A}$
$H_0 = 0$
for *t=0:T-1* **do**
  $\forall I : r_i = \frac{\partial \ell((H_t(\mathbf{x}_1), y_1), \ldots, (H_t(\mathbf{x}_n), y_n))}{\partial H(\mathbf{x}_i)}$
  $h_{t+1} = \mathbb{A}(\{(\mathbf{x}_1, r_1), \ldots, (\mathbf{x}_n, r_n)\}) = \operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^{n} r_i h(\mathbf{x}_i)$
  if $\sum_{i=1}^{n} r_i h_{t+1}(\mathbf{x}_i) < 0$ **then**
    $H_{t+1} = H_t + \alpha_{t+1} h_{t+1}$
  else
    return (H_t)
  end
end
return H_T

FIGURE 8 – Generic boosting in pseudo-code

Gradient Boosted Regression Tree

They are used for search engines (to predict what you are looking at).

Setting

— $y_i \in \mathbb{R}$ (regression or classification)
— h regressors, fixed depth regression trees ($d = 4, 6$)
— Step size α is fixed to a small constant

Assumptions

$$\boxed{\sum_{i=1}^{n} h(x_i)^2 = \text{cste}}$$

We are only concerned with the direction. In practice, it suffices to normalize the predictions.

Mathematical formulation

Define $r_i := \frac{\partial l}{\partial H(x_i)}$ and $t_i := -r_i$

Our problem can be reformulated as follow.

$$\operatorname*{argmin}_{h \in H} \{\textstyle\sum_{i=1}^{n} h(x_i) r_i\} = \operatorname*{argmin}_{h \in H} \{-2 \textstyle\sum_{i=1}^{n} h(x_i) t_i\}$$

$$\operatorname*{argmin}_{h \in H} \{\textstyle\sum_{i=1}^{n} h(x_i) r_i\} = \operatorname*{argmin}_{h \in H} \{\textstyle\sum_{i=1}^{n} t_i^2 - 2 t_i h(x_i) + h(x_i)^2\}$$

$$\operatorname*{argmin}_{h \in H} \{\textstyle\sum_{i=1}^{n} h(x_i) r_i\} = \boxed{\operatorname*{argmin}_{h \in H} \left\{ \sum_{i=1}^{n} (h(x_i) - t_i)^2 \right\}}$$

→ Therefore, **we can use regression trees to** $\{(x_i, t_i), i = 1, \cdot, n\}$.

Remark : If l squared loss, $t_i = y_i - H(x_i)$ (residuals).

Find the pseudo-code below :

GBRT in Pseudo Code

Input: ℓ, α, $\{(\mathbf{x}_i, y_i)\}$, $\mathbb{A}$
$H = 0$
for *t=1:T* **do**
  $\forall i : t_i = y_i - H(\mathbf{x}_i)$
  $h = \text{argmin}_{h \in \mathbb{H}} (h(\mathbf{x}_i) - t_i)^2$
  $H \leftarrow H + \alpha h$
end
return H

FIGURE 9 – GBRT in pseudo-code

AdaBoost or Adaptive Boosting

Setting

— $y_i \in \{-1, 1\}$ (classification)
— h are binary, $h(x_i) \in \{-1, 1\}$
— Step size : perform line-search after finding h
— Loss function : exponential loss $\boxed{l(H) = \sum_{i=1}^{n} e^{-y_i H(x_i)}}$

Mathematical formulation

Define $r_i := \frac{\partial l}{\partial H(x_i)} = -y_i e^{-y_i H(x_i)}$, $w_i := e^{-y_i H(x_i)}$ and $Z := \sum_{i=1}^{n} e^{-y_i H(x_i)}$.

The weights w_i represent **the contribution of a training point** (x_i, y_i) **towards the overall loss**.

i) First step : find h

Recall that we want to find h such that $h = \underset{h \in H}{\text{argmin}}\, l(H_t + \alpha h)$
$h \approx \underset{h \in H}{\text{argmin}} \sum_{i=1}^{n} r_i \cdot h(x_i)$

In our case, our problem can be rewritten as :

$$\underset{h \in H}{\text{argmin}} \left\{ -\sum_{i=1}^{n} w_i y_i h(x_i) \right\} = \underset{h \in H}{\text{argmin}} \left\{ \sum_{i:h(x_i) \neq y_i} w_i - \sum_{i:h(x_i) = y_i} w_i \right\}$$

$$\boxed{\underset{h \in H}{\text{argmin}} \sum_{i:h(x_i) \neq y_i} w_i}$$

It is the *weighted classification error*.

Denote $\epsilon := \sum_{i:h(x_i) \neq y_i} w_i$.

Note that to make a progression, i.e decrease l, we only need to have the $\epsilon < 0.5$ (cf Taylor approximation).

ii) Second step : find α

Now that h is found, we want to solve to following problem :

$$\alpha = \underset{\alpha}{\text{argmin}} l(H + \alpha h)$$

We can easily show that :

$$\boxed{\alpha = \frac{1}{2} ln(\frac{1-\epsilon}{\epsilon})}$$

AdaBoost Pseudo-code

Input: ℓ, α, $\{(\mathbf{x}_i, y_i)\}$, $\mathbb{A}$
$H_0 = 0$
$\forall i : w_i = \frac{1}{n}$
for *t=0:T-1* **do**
  $h = \text{argmin}_h \sum_{i:h(\mathbf{x}_i) \neq y_i} w_i \qquad [h = \mathbb{A}\left((w_1, \mathbf{x}_1, y_1), ..., (w_n, \mathbf{x}_n, y_n)\right)]$
  $\epsilon = \sum_{i:h(\mathbf{x}_i) \neq y_i} w_i$
  if $\epsilon < \frac{1}{2}$ **then**
    $\alpha = \frac{1}{2} \ln(\frac{1-\epsilon}{\epsilon})$
    $H_{t+1} = H_t + \alpha h$
    $\forall i : \quad w_i \leftarrow \frac{w_i e^{-\alpha h(\mathbf{x}_i) y_i}}{2\sqrt{\epsilon(1-\epsilon)}}$
  else
    return (H_t)
  end
end
return (H_T)

FIGURE 10 – GBRT in pseudo-code

8 Lexique

Decision-tree, CART, Regression trees, Classification trees, bagging, bootstrapping, random forest, boosting, greedy algorithm, adaboost, anyboost, gradient boosted regressoin tree

9 Sources

Click on the number of the figure to be redirected.

- Figure 1

- Figure 2

- Figure 3

- Figure 4

- Figure 6, *The elements of statistical learning*

-Figure 8,9,10 : Course notes of Kilian Weinberger on boosting methods

10 Learning resources

- The course Statistical Learning available on the plateform Stanford Online (`https://online.stanford.edu/courses/sohs-ystatslearning-statistical-learning`).
You can also find it on Youtube (`https://www.youtube.com/watch?v=XvdLKUOldkE&list=PLOg0ngHtcqbPTlZzRHA2ocQZqB1D_qZ5V&index=48`).

- You can also refer to the textbook *An Introduction to Statistical Learning* (`https://web.stanford.edu/~hastie/ISLR2/ISLRv2_website.pdf`) or *Elements of Statistical Learning*(https ://web.stanford.edu/ hastie/Papers/ESLII.pdf)

- The videos of Kilian Weinberger on his Youtube channel (`https://www.youtube.com/channel/UC7p_I0qxYZP94vhesuLAWNA/videos`). Lectures 28 to 34.

- Course notes of Kilian Weinberger on his website `http://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote17.html`, 17 to 19.

- A discussion about the number of trees needed for random forests `https://www.researchgate.net/post/How_to_determine_the_number_of_trees_to_be_generated_in_Random_Forest_algorithm`