Decision trees and random forests

Vincent Divol

Decision trees are a family of non-parametric machine learning models that are able to handle heterogeneous data (ordered and categorical), while being easily interpretable. When combined with ensemble methods (yielding random forests), they give one of the best off-the-shelf prediction algorithms available.

1 Decision trees

Trees are a data structure of major importance in computer science, used for e.g. search algorithms, clustering tasks, or in natural language processing. Mathematically speaking, a tree is a special kind of graph. A *graph* G is given by a set of nodes V and a set of edges E, where an edge $e \in E$ consist of a set of two distinct (unordered) vertices. A *tree* T is a graph in which any two nodes are connected by exactly one path. A *rooted tree* is a tree in which one of the nodes has been designated as the *root*. A rooted tree can actually be seen as *directed* graph, by giving every edge a direction (away from the root): every edge $e \in E$ is represented as an ordered pair $e = (t_1, t_2)$. For such a directed edge, t_1 is called the *parent* of t_2 and t_2 is said to be a *child* of t_1. We say that a node is *internal* if it has one or more children, and *terminal* otherwise. Terminal nodes are also called *leaves*. A *binary tree* is rooted tree where all internal nodes have exactly two children.

We consider a regression setting, where observations $x \in \mathcal{X}$ are assigned labels $y \in \mathcal{Y}$. For decision trees, the set of labels $\mathcal{Y}$ can be arbitrary, but we will focus in these notes on binary classification ($\mathcal{Y} = \{-1, 1\}$) or regression ($\mathcal{Y} = \mathbb{R}$) for the sake of exposition. A *decision tree* is a predictor $h : \mathcal{X} \to \mathcal{Y}$ that can be represented by a binary tree $T = (V, E)$ in the following way. Each node of the tree represents a subset of $\mathcal{X}$. The rooted node t_0 represents $\mathcal{X}$, and if $\mathcal{X}_t$ is the set associated with a node $t \in V$, then the children t_1 and t_2 of t represent a partition of $\mathcal{X}_t$:

$$\mathcal{X}_t = \mathcal{X}_{t_1} \sqcup \mathcal{X}_{t_2}. \tag{1}$$

We call such a partition a *split* of $\mathcal{X}_t$ into $\mathcal{X}_{t_1}$ and $\mathcal{X}_{t_2}$. Remark that the leaves of the tree form a partition of the feature space $\mathcal{X}$. The predictor $h : \mathcal{X} :\to \mathcal{Y}$ then makes constant predictions $\hat{Y}_t$ on each set $\mathcal{X}_t$ for t a leaf of T.

There are still many issues to clarify to understand how a decisition tree works. How are the different splits chosen? How do we pick the value $\hat{Y}_t$ of the prediction h on each final split?

Let us assume that the observations $x \in \mathcal{X}$ are given as a list of features $x = (x^{(1)}, \dots, x^{(K)})$. The different features can be heterogeneous, that is some of them can be continuous while some others can be categorical. Mathematically speaking, we write $\mathcal{X} = \mathcal{X}^{(1)} \times \mathcal{X}^{(2)} \times \cdots \times \mathcal{X}^{(K)}$. Then, the split of $\mathcal{X}_t$ at the node t can be made by first selecting a feature (that is we pick k), and then splitting $\mathcal{X}_k$ in a simple way. For instance, if $\mathcal{X}^{(k)}$ represents a continuous feature in $[0, 1]$, we can split $\mathcal{X}^{(k)}$ in two by picking a threshold $s_t \in [0, 1]$, and writing

$$[0, 1] = [0, s_t) \sqcup [s_t, 1]. \tag{2}$$

Classification trees are typically built by choosing simple splits of this form in a sequential manner. That is we ask a series of questions of the form "Is the feature $x^{(k)}$ of the observation

x smaller than some threshold s?" to assign x to a certain set $\mathcal{X}_t$ for some leaf $t \in V$. Once we know that $x \in \mathcal{X}_t$, we return $h(x) = \hat{Y}_t$. Therefore, a first property of decision trees is that *they are very quick to evaluate*: the complexity only depends on the *depth* of the tree T.

2 Training with decision trees

Let $(X_1, Y_1), \dots, (X_n, Y_n)$ be a training dataset of i.i.d. observations from some law P on $\mathcal{X} \times \{-1, +1\}$. How can we fit a decision tree to this dataset?

A first remark is that, by adding sufficiently many leaves, it is always possible to find a decision tree with perfect classification on the training set. More precisely, such a task is possible with exactly n leaves (one by observation). However, such a tree will typically be very complex and leads to overfitting. As Occam's Razor principle suggests, we will favor simple trees that make few mistakes to complex trees that make zero mistakes.

A good idea therefore would be to compute the simplest tree h (say, with the minimal number of nodes) with perfect training accuracy, that is such that

$$\widehat{\mathrm{err}}(h) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{h(X_i) \neq Y_i\}. \tag{3}$$

is equal to 0. However, it has been shown that finding such a tree is a NP-complete problem [Laurent and Rivest, 1976]. Research has therefore focused on performing greedy searches for trees with good training accuracy: at each step, one looks for the best split in a given tree that will further decrease the training error.

Assume that we have access to a measure of impurity $i(t)$ of each node t. The smaller $i(t)$ is, the purer the node is and the better the prediction $\hat{Y}_t$ is on $\mathcal{X}_t$. Then, the impurity of a tree classifier h is given by

$$i(h) = \sum_{t \in \mathrm{Leaf(T)}} p_t i(t), \tag{4}$$

where p_t is the proportion of samples found in $\mathcal{X}_t$. In other words, $i(h)$ is the (weighted) average impurity of a leaf in T. We will discuss later different impurity functions used in practice (based on either the Shannon entropy or the Gini index).

Definition 2.1. *The* impurity decrease *of a binary split s that divides a note t into a left node t_L and a right node t_R is given by*

$$\Delta i(s, t) = i(t) - \frac{p_{t_L}}{p_t} i(t_L) - \frac{p_{t_R}}{p_t} i(t_R). \tag{5}$$

If a new tree $\tilde{h}$ is obtained from h after a split s_0 of a leaf t_0 of h, then we have

$$\begin{aligned} i(\tilde{h}) &= \sum_{t \in \mathrm{Leaf}(T) \setminus \{t_0\}} p_t i(t) + p_{t_L} i(t_L) + p_{t_R} i(t_R) \\ &= i(h) - p_t \Delta i(s, t). \end{aligned}$$

Therefore, for a given node t, one should pick the binary split s that maximizes the impurity decrease $\Delta i(s, t)$. The pseudoalgorithm below describes how a binary decision tree is trained.

Note that the algorithm depends upon a stopping criterion. The most basic stopping criterion consists in stopping when the node t is such that $\mathcal{X}_t$ contains only one observation X_i. Then, $\hat{Y}_t$ is simply given by Y_i and there is no use in further dividing the space $\mathcal{X}$. Such a stopping criterion yields to trees with n leaves, that are prone to overfitting. Therefore, other conditions are added for early stopping:

Algorithm 1 Greedy learning of a binary decision tree

Input: Dataset $(X_1, Y_i), \dots, (X_n, Y_n)$, stopping criterion

Initialize: Tree with root node t_0 and corresponding set $\mathcal{X}_{t_0} = \mathcal{X}$. Empty stack of open nodes S.

Add t_0 to S.

while S is not empty **do**

Choose t on top of the stack S

if the stopping criterion is met for t **then**

Decide on $\hat{Y}_t = -1$ or $+1$ by a majority vote on the X_is in $\mathcal{X}_t$.

Define $h(x) = \hat{Y}_t$ for $x \in \mathcal{X}_t$,

else

Find the split on t that maximizes impurity decrease

$$s^* = \arg\max_s \Delta i(s, t). \tag{6}$$

Partition $\mathcal{X}_t$ into $\mathcal{X}_{t_R} \sqcup \mathcal{X}_{t_L}$ according to s^*

Create a left child node t_L and a right child note t_R of t

Add t_L and t_R to the stack S.

end if

end while

Output: Final classifier h.

- Set t as a leaf if the corresponding set contains less than $n_{\min}$ samples.
- Set t as a leaf if its depth is larger than some threshold $d_{\max}$.
- Set t as a leaf if the total decrease in impurity $p_t \Delta i(s^*, t)$ is less than some threshold β.

These stopping criteria all require to tune hyperparameters to find the right trade-off, so that the tree T is neither too shallow nor too deep. If the tree is too shallow, then the model is too simple and will have large bias; whereas if the tree is too large, then there will be a large generalization eror. Note that parameter selection (typically done through cross validation) can be a computationally expensive task. These methods are called *pre-pruning*.

An alternative approach consists in computing an overfitting tree (with n leaves), and remove some leaves afterwards in a *post-pruning* process. When a single decision tree is used, post-pruning usually yields better results than pre-pruning. Anyway, we will see later that when using ensemble of decision trees (random forests), no pruning procedure at all is needed to achieve good generalization performance.

3 Impurity functions

Let us first introduce an impurity function that makes sense if our goal is to decrease the training error of the decision tree as fast as possible. Let $P(c|t)$ denote the probability that $Y = c$ conditionally on the event that $X \in \mathcal{X}_t$, that is

$$P(c|t) = P(Y = c | X \in \mathcal{X}_t) = \frac{P(Y = c, X \in \mathcal{X}_t)}{P(X \in \mathcal{X}_t)}. \tag{7}$$

Although we do not have access to $P(c|t)$, we can compute its empirical counterpart

$$\hat{P}(c|t) = \frac{1}{n_t} \sum_{i=1}^{n} \mathbf{1}\{Y_i = c,\ X_i \in \mathcal{X}_t\}, \tag{8}$$

where n_t is the number of observations X_is that fall in $\mathcal{X}_t$. The training error of a decision tree h can be written as

$$\begin{aligned}
\widehat{\mathrm{err}}(h) &= \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{h(X_i) \neq Y_i\} \\
&= \sum_{t \in \mathrm{Leaf}(T)} \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{h(X_i) \neq Y_i,\ X_i \in \mathcal{X}_t\} \\
&= \sum_{t \in \mathrm{Leaf}(T)} \frac{p_t}{n_t}\sum_{i=1}^{n} \mathbf{1}\{\hat{Y}_t \neq Y_i,\ X_i \in \mathcal{X}_t\} \\
&= \sum_{t \in \mathrm{Leaf}(T)} p_t(1 - \hat{P}(\hat{Y}_t|t)).
\end{aligned}$$

Therefore, if we define the impurity function as

$$i_R(t) = 1 - \hat{P}(\hat{Y}_t|t), \tag{9}$$

then the total impurity $i_R(h)$ is exactly equal to the training error $\widehat{\mathrm{err}}(h)$. Despite being a natural choice for an impurity function, there are some caveats with i_R. Indeed, if the majority class remains the same in the two child nodes ($\hat{Y}_t = \hat{Y}_{t_L} = \hat{Y}_{t_R}$), then $\Delta i_R(s,t)$ is zero:

$$\begin{aligned}
\Delta i_R(s,t) &= i_R(t) - \frac{p_{t_L}}{p_t} i(t_L) - \frac{p_{t_R}}{p_t} i(t_R) \\
&= 1 - \hat{P}(\hat{Y}_t|t) - \frac{p_{t_L}}{p_t}(1 - \hat{P}(\hat{Y}_t|t_L)) - \frac{p_{t_R}}{p_t}(1 - \hat{P}(\hat{Y}_t|t_R)) \\
&= 0.
\end{aligned}$$

As a consequence, there will be many different splits for which $\Delta i_R(s,t) = 0$, and all these splits are therefore considered as good. We would like to design an impurity function that pushes for splits where the new predictions are made with higher confidence.

Definition 3.1. *Let Q be a probability measure on a set $\{1, \dots, J\}$. We call the* binary entropy *or* Shannon entropy *of Q the quantity*

$$H(Q) = -\sum_{j=1}^{J} Q(j)\log_2(Q(j)). \tag{10}$$

This notion of entropy was introduced by Claude Shannon in a seminal paper that launched the field of information theory [Shannon, 1948]. Generally, if E is an event, the quantity $-\log_2(Q(E))$ is called the information content of E. If E is a very surprising event (say a certain number is a winning lottery number), then its information content will be very high, whereas if E is not surprising (it always almost happens), then we will not learn any new information if we are said that E happened. The entropy is defined as the expected information content of a given random expriment (here, the random experiment consists in picking at random a number between 1 and J with distribution Q). For instance, if the number j drawn from Q is always equal to a certain number j_0, then $H(Q) = 0$, as there is absolutely no randomness here. The other extreme is when the number j is drawn uniformly at random: in that case, the entropy is maximal, equal to $\log_2(J)$.

Another way to think about the entropy is the following. Assume that one wants to send messages drawn from Q to a receiver. Of course, they can simply send the number j each time,

requiring $\log_2(J)$ bits in the process (to write the number j in binary). Assume for instance the word we are sending is taken at random from the dictionnary $\{\text{kiwi}, \text{banana}, \text{zebra}, \text{apple}\}$. Then, we can use the code of length $2 = \log_2(4)$:

$$\begin{aligned} \text{kiwi} &\to 00 \\ \text{banana} &\to 01 \\ \text{zebra} &\to 10 \\ \text{apple} &\to 11. \end{aligned}$$

However, if both the sender and the receiver know Q in advance, there might be more succinct ways of sending the code. Assume that we pick zebra 90% of the time, and one fruit uniformly at random 10% of the time. Then, a better encoding is

$$\begin{aligned} \text{zebra} &\to 0 \\ \text{kiwi} &\to 100 \\ \text{banana} &\to 101 \\ \text{apple} &\to 110. \end{aligned}$$

With this scheme, under Q, the average length of the code sent will be $1 \cdot 0.9 + 3 \cdot 0.1 = 1.2 \leq 2$. What is the minimal average length that we can reach? Shannon proved that the minimal average length reachable by a code is between $H(Q)$ and $H(Q) + 1$. To put it another way, *the Shannon entropy is the average length of i.i.d. words from Q encoded in the optimal way.*

A commonly used impurity index is given by the Shannon impurity $i_H(t)$ given by the entropy of the probability measure $\hat{P}(\cdot|t)$ over the label set $\mathcal{Y}$. Another measure of "uncertainty" of a distribution is given by its variance. For binary classification, the *Gini index* i_G is based on this measure, with $i_G(t)$ being the variance of the probability measure $\hat{P}(\cdot|t)$ (seen as a probability measure on $\{0, 1\}$).

4 Random forests

We already encountered an exemple of ensemble methods with boosting: they consist in making many different classifiers take part in a vote to obtain a classifier with way better accuracy than any of those taking part in the vote. For boosting, the idea was to make weak classifiers (with large bias, but small generalization error) vote to improve the complexity of the model, therefore reducing the bias. In random forests, the aggregation will have an opposite effect. Many different trees without pruning (with small bias, but high variance) will vote to obtain a classifier with reduced variance.

To sum up, ensemble methods often work better than single models for two different main reasons:

- **Variance reduction:** when the number of observations is small and single models are too complex, classifiers will have large variance. When aggregating several such models, if the different predictions are sufficiently uncorrelated, one can hope to have an averaging effect that reduces the variance.

- **Bias reduction:** if the models are too simple, the optimal classifier cannot be represented by one of the candidate models. However, combining several such simple models can increase their representational capacity.

Mathematically speaking, an ensemble method is obtained by calling repeatedly randomized algorithms on our set of observations $D_n = ((X_1, Y_1), \dots, (X_n, Y_n))$. The randomness may come from different sources:

- **Bagging:** Bagging (for Bootstrap AGGregatING) consists in training a classifier on a set of N observations drawn with replacement from D_n. When $N = n$, there are approximately $1/e \approx 37\%$ duplicates in the sample, the others being unique observations. By drawing M such samples in a sequential way, we obtain a list of classifiers $h_1, \dots, h_M$, that can then take part in a majority vote. When the classifier is unstable (that is small changes in the data set can cause large changes in the learned models), bagging will generate very different models, so that we are likely to benefit from the averaging process. However, this also leads to a larger bias, as each individual model is trained on a data set of a smaller effective size (roughly of size $N(1 - 1/e)$), leading to simpler trees.

- **Random variable selection:** to split the node t, one looks for the split s of some feature $x^{(k)}$ with $\Delta i(s, t)$ maximal. When the data are high dimensional, one can instead look for the best splits among K different features drawn at random among all the possible features. For high dimensional dataset, this will lead to individual classifiers with different structures, although each of them will still have very small training error. Once again, we can hope to have a smoothing effect when averaging these structurally different trees.

Random forests [Breiman, 2001] consist in combining these two ideas, and work surprisingly well as an off-the-shelf method, giving result that are competitive with boosting algorithm such as AdaBoost.

Contrary to boosting, for which we were able to develop mathematical heuristics, giving a theoretical analysis of random forests is a delicate question, and is out of the scope of these lecture notes [Scornet et al., 2015].

References

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine learning*, 45:5–32.

[Laurent and Rivest, 1976] Laurent, H. and Rivest, R. L. (1976). Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17.

[Scornet et al., 2015] Scornet, E., Biau, G., and Vert, J.-P. (2015). Consistency of random forests. *The Annals of Statistics*, 43(4):1716 – 1741.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423.