Robust CART - Dealing with errors in the data

Thibaud Rahier

A thesis submitted in partial satisfaction of the Master of Arts in Statistics of University of California, Berkeley

Committee in charge:
Noureddine El Karoui, Chair
Haiyan Huang
Laurent El Ghaoui

Spring 2015
I want to convey my sincere gratitude to my advisor and thesis chair, Noureddine El Karoui, for his continued support. My thanks also go to my thesis committee, Laurent El Ghaoui and Haiyan Huang. The theoretical setting presented in the first chapter is the result of discussions between Noureddine El Karoui, James Long and John Rice, and none of my work could have been possible without their ideas. I am very grateful to have joined such an interesting project.

I also want to thank Joey Kogan for providing me with his work on the project, even if his explorations of the rpart package were not finally the way I chose to implement the algorithms.
Introduction

Upon the wide range of supervised learning methods, decision trees show many advantages: they are very easy to display graphically, to interpret, to explain to even non-expert people and are also believed to mirror closely human decision-making processes. However, they are very unstable: they are not robust to errors in the data they are built from in the sense that their structure can dramatically change if there is only a little variation in the training data. This lack of robustness raises two main questions.

1. If a tree is very unstable, it is also less interpretable. How can we quantify the stability, and therefore the interpretability of trees?

2. How can we use trees in areas of science where there is systematic uncertainty in measures (such as Astronomy)?

This report focuses on studying the stability of trees (and therefore their interpretability), by defining an adapted version of the Classification and Regression Trees (CART) algorithm that is able to build trees when the features of the given training data are located in confidence intervals, and are not punctual as for the regular CART algorithm. This new algorithm will enable us to measure the stability of trees by studying how their structure changes for different sizes of confidence intervals, giving an element of response for the first question. Moreover, it will enable us to build trees from data that has given uncertainty as is commonly the case in Astronomy.

In the first part, I will focus on presenting the theoretical setting for this algorithm, and how it can be expressed as a slightly modified version of the regular CART algorithm. I will then explain how I implemented this algorithm in R with the help of the package `rpart`. After this, I will discuss an alternative approach on pruning, and a rule to implement it in an approximate (faster) way. Finally, I will test those methods on both synthetic and real data and give some answers to the two previously asked questions.
Chapter 1
Theoretical setting

In this first chapter, we will introduce the robust cart algorithm with the following steps:

• First, we will make a quick review of the usual CART algorithm, explaining the main ideas behind building trees and predicting from already built trees.

• Then we will see two ways to adapt this algorithm to data that contains uncertainty (where each observation is in a box and not simply a point).
  – The distribution approach, which is only feasible when the distribution of the features is known or can be estimated.
  – The worst-case scenario, or min-max approach, which enables us to build robust trees, but which is computationally too heavy to tackle upfront for large data. The idea that robust optimization is good way to control uncertainty is a very important concept explained in [1].

• We will then rewrite the min-max approach in a much simpler way, using a relaxation trick and Kakutani’s theorem (Lemma 1).

• Finally, we use this lemma to show a key theorem (Theorem 1) that enables us to see the min-max approach as a slight modification of the regular CART algorithm with a different convex loss function, thus solving the computational problem raised by this method.

We will conclude this chapter by stating how to tackle pruning and prediction in the case of this adapted algorithm.

The work that I present in this chapter was done by Noureddine El Karoui and James Long before I began my research at UC Berkeley. Most of my work consisted in implementing testing the robust CART algorithm.

1.1 Review of CART
1.1.1 Overview of the algorithm for regression

The idea of the CART (Classification and Regression Trees) algorithm, as described in [4] and in Chapter 9.2 of [5], is to partition adaptively the feature space into regions and to fit a constant in each of these regions.

Let us consider that we have data consisting of \( n \) observations \((x_i)_{1 \leq i \leq n}\), each having \( p \) features (so for \( i \in \{1, \ldots, n\}, x_i = (x_{i1}, \ldots, x_{ip}) \)) and a response \( y_i \).

The algorithm finds a series of splitting variables and split points in order to define \( M \) regions of the \( p \)-dimensional feature space \( R_1, \ldots, R_M \).

In each region \( R_m \) for \( m \in \{1, \ldots, M\} \), the response is modeled as a constant \( c_m \). The complete model
Figure 1.1: Regions of the feature space for the example tree in dimension 2

is then expressed as:

\[ f(x) = \sum_{m=1}^{M} c_m I(x \in R_m) \]  

(1.1)

Illustration in the case of a 2-dimensional feature space

CART is restricted to recursive binary partition splitting, as in the following example: let us suppose our data has two features \( f_1 \) and \( f_2 \), and that we proceed to the following succession of splits:

- First we split the data according to \( f_1 = s_1 \)
- Then we split the region \( f_1 \leq s_1 \) at \( f_1 = s_2 \) (where \( s_2 \leq s_1 \) obviously).
- And the region \( f_1 > s_1 \) is split at \( f_2 = s_3 \).

In that case, the feature space is split into 4 regions \( R_1, R_2, R_3, R_4 \) as shown on Figure 1.1.1. The corresponding tree is:

where the black nodes represent the final nodes (regions of the feature space), and the white nodes are the split nodes.
1.1.2 Choosing the splits and the constants in each region

We first need to define a loss function to use for choosing the splits and fitting the constants in each region of the feature space. We commonly take the $L_2$ loss.

It is generally computationally infeasible to find the best binary partition in terms of minimum $L_2$ loss. Therefore, CART uses a greedy algorithm:

- We start with all the data $(x_i, y_i), i \in \{1, \ldots, n\}$.

For a splitting variable $j \in \{1, \ldots, p\}$ and a split value $s$, we define the following functions:

$$\forall c_1, c_2 \in \mathbb{R}, \quad F(c_1, c_2, j, s) = \sum_{i: x_{ij} \leq s} |y_i - c_1|^2 + \sum_{i: x_{ij} > s} |y_i - c_2|^2. \quad (1.2)$$

$$L(j, s) = \min_{c_1, c_2} F(c_1, c_2, j, s). \quad (1.3)$$

- $F$ corresponds to the training error of the model consisting of simply splitting the data at $s$ according to the feature $j$, and assigning the constants $c_1$ and $c_2$ to the two resulting regions $R_1(j, s) = \{x|x_j \leq s\}$ and $R_2(j, s) = \{x|x_j > s\}$.

- $L$ corresponds to the training error of the same model but with the constants $\hat{c}_1$ and $\hat{c}_2$ assigned to $R_1(j, s)$ and $R_2(j, s)$ where $\hat{c}_1$ and $\hat{c}_2$ are the constants that minimize the $L_2$ loss: for $m = 1, 2$,

$$\hat{c}_m = \text{ave}(y_i | x_i \in R_m(j, s)). \quad (1.4)$$

To choose the best split for all the data, we then want to minimize $L(j, s)$ over all possible features $j$ and split values $s$. Therefore, the optimization problem becomes:

$$\min_{j, s} L(j, s) = \min_{j, s} \left( \min_{c_1, c_2} F(c_1, c_2, j, s) \right) \quad (1.5)$$

where $L$ and $F$ are defined in (1.2), (1.3) and depend on the data.

- Once the best split $s$ is found, we partition the data into two subsets according to the corresponding regions $R_1(j, s) = \{x|x_j \leq s\}$ and $R_2(j, s) = \{x|x_j > s\}$, and repeat the first step for each of the partial data.

Choosing the constant in each final region $R_1, \ldots, R_M$ is then done as in equation (1.4). The model therefore becomes:

$$\hat{f}(x) = \sum_{m=1}^{M} \hat{c}_m I(x \in R_m) \quad (1.6)$$

**Important remark**
The optimization problem stated in (1.5) is solvable:

1. $F$ is convex in $c_1, c_2$ (so the inner minimization does not raise an issue)

2. There are only a finite number of features ($p$) and of possible splits for each one of them (the total number of distinct values for the considered feature, so at most $n$), which makes at most $np$ values of $L$ to compute to solve (1.5).

1.1.3 The bias-variance tradeoff for trees

Now that we know how to build a tree, we need to know how large it should be grown. This is a typical case of bias-variance tradeoff:

- A tree that is too large will have low bias but high variance, and will be likely to overfit the data.
• A tree that is too small will have low variance but high bias, and will maybe not be able to capture the important structure of the data.

The variable we usually use to express well the model complexity is the tree size. In practice, we often take the number of terminal nodes (which is also the number of regions of the feature created by the tree).

Pruning trees
The common way to tackle this tradeoff problem is to grow a very large tree (stopping only when there is less observations than a certain threshold in each region), then to prune it back to only keep the important structure of the data in the final tree. This idea is mentioned in Chapter 9.2 of [5] (p 308), and is more detailed in Chapter 3 of [4].

Pruning is done with respect to a complexity parameter $\alpha \geq 0$, and narrows down to finding the smallest tree $T$ that minimizes the following penalized loss (where $\tilde{T}$ is the set of terminal nodes of tree $T$):

$$R_\alpha(T) = \sum_{t' \in \tilde{T}} \text{dev}(t') + \alpha|\tilde{T}|$$  \hspace{1cm} (1.7)

where $\text{dev}(t')$ is the deviance of node $t'$: for regression trees we often take the sample variance of the response variable of the observations in the considered node. For classification trees, we will see that there several ways to define the deviance of a node.

In section 2.3, we will explain the usual method for pruning trees, and discuss another approach (that is not recursive, as opposed to the usual algorithm described in [4]).

1.1.4 Classification trees
For classification trees (where we suppose that we are facing a multi-class problem with $K$ classes), the main idea of CART is the same as for regression trees, but we need to define a new loss function to choose splits, and a rule to predict from the tree. For a region $R_m$, in which there are $N_m$ observations and with $k$ referring to one of the $K$ classes, let

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$$

the proportion of observation of class $k$ in the region $R_m$.

• For prediction, we assign to each region the class that is the most present in the training data belonging to that region. We assign to the region $R_m$ the class $k(m)$ where

$$k(m) = \underset{k}{\text{argmax}} \hat{p}_{mk}$$

• In terms of loss function, there are several way to define the impurity of a node (which was done thanks to the $L_2$ loss in the case of regression).

  – Misclassification error: $1 - \hat{p}_{mk(m)}$

  – Gini index: $\sum_{k=1}^K \hat{p}_{mk}(1 - \hat{p}_{mk})$

  – Cross-entropy or deviance: $-\sum_{k=1}^K \hat{p}_{mk}\log(\hat{p}_{mk})$

Once we choose a way to define the impurity of a node, we can then choose the split that reduces the most this impurity, as we did with $L_2$ loss in the case of regression trees. The global algorithm to build a tree is then the same one that is presented in section 1.1.2.
1.2 Adapting CART to uncertainty in the data

We will now show how to adapt the CART algorithm to data for which there is uncertainty. For all that follows, we only focus on regression trees.

1.2.1 Notations

We now consider a setting where there is uncertainty on the data.

We still suppose we have \( n \) training observations \((x_1, y_1), \ldots, (x_n, y_n)\), where \( \forall i \in \{1, \ldots, n\}, \ x_i \in B_i \) where \( B_i \) is a box of the \( p \)-dimensional feature space (for instance, \( B_i \) might be centered at the \( x_i \) we observe, if we do observe an \( x_i \)). This is a very common approach in robust optimization, that can notably be found in [1].

We call \( J = \{1, \ldots, p\} \) the index set for the predictors and \( s \in \mathbb{R} \) the split point.

For \( j \in J \), \( P_j(B_i) \) is the projection of the box \( B_i \) onto the \( j^{th} \) axis: it is a confidence interval for \( x_{ij} \), the \( j^{th} \) feature of \( x_i \).

1.2.2 Rewriting the optimization problem

We saw in the previous section that each step of the CART regression algorithm narrows down to solving the minimization problem (1.5):

\[
\min_{j,s} L(j, s) = \min_{j,s} \left( \min_{c_1,c_2} F(c_1, c_2, j, s) \right),
\]  

where \( F \) is defined in (1.2) as follows:

\[
\forall c_1, c_2 \in \mathbb{R}, \ F(c_1, c_2, j, s) = \sum_{i: x_{ij} \leq s} ||y_i - c_1||_2^2 + \sum_{i: x_{ij} > s} ||y_i - c_2||_2^2.
\]  

This expression of \( F \) can be rewritten as:

\[
F(c_1, c_2, j, s) = \bar{F}((\theta_i(s))_{i=1}^n; c_1, c_2) = \sum_i \theta_i(s)||y_i - c_1||_2^2 + (1 - \theta_i(s))||y_i - c_2||_2^2
\]  

where \( \theta_i(s) = 1 \) if \( x_i(j) \leq s \) and \( \theta_i(s) = 0 \) otherwise.

From now on, we will use the notation \( F \) to designate both \( F \) as defined in (1.2) and \( \bar{F} \) as defined in (1.10) (that depends on a set of parameters \( \theta_i(s))_{i=1}^n \)).

1.2.3 Handle uncertainty in the data

Now, in the case where there is uncertainty in the data, the \( \theta_i(s) \)'s defined in (1.10) cannot be computed so easily, because there are cases for which the observations are neither on one side or the other of the split.

Consider some proposed split at \( s \) on axis \( j \). There are 3 possible cases for a given observation \( x_i \):

- \( P_j(B_i) \) is on the left of the split (that we will abusively write \( P_j(B_i) < s \))
- \( P_j(B_i) \) is on the right of the split (that we will abusively write \( P_j(B_i) > s \))
- the split \( s \) intersects \( P_j(B_i) \): \( s \in P_j(B_i) \)

In the first two cases we have no work to do, because we know for sure the location of \( x_{ij} \) with respect to \( s \), and the corresponding \( \theta_i(s) \) can be computed as mentioned in the end of the previous subsection in equation (1.10).

Let us call \( I_j(s) \) the set of \( i \)'s such that the interval \( P_j(B_i) \) contains \( s \):

\[
I_j(s) = \{ i : s \in P_j(B_i) \}.
\]  

6
We can then decompose $F$ as defined in (1.10) into 3 different sums depending on the previous distinction.

\[
F(\{\theta_i(s)\}_{i=1}^n; c_1, c_2) = \sum_{i \in I_j(s)} \theta_i(s) ||y_i - c_1||^2 + (1 - \theta_i(s)) ||y_i - c_2||^2 
\]

(1.12)

\[
+ \sum_{i \notin I_j(s), P_j(B_i) < s} ||y_i - c_1||^2 + \sum_{i \notin I_j(s), P_j(B_i) > s} ||y_i - c_2||^2 
\]

(1.13)

In the previous expression, only the first sum requires our attention: how should we choose the corresponding $\theta_i(s)$’s since the considered observations are neither on one side or the other of the split $s$?

From now on, we will use the following notation to designate the first sum in the expression of $F$ (where $m = |I_j(s)|$ is the number of observations whose box is intersected by the split $s$).

\[
H(\{\theta_i(s)\}_{i=1}^m; c_1, c_2) = \sum_{i \in I_j(s)} \theta_i(s) ||y_i - c_1||^2 + (1 - \theta_i(s)) ||y_i - c_2||^2 
\]

(1.14)

The issue is that $H$ depends on the $\theta_i$’s. We need to find a way for $H$ to get rid of this dependency so that it can be minimized as in (1.3).

We have two approaches described in the beginning of this chapter to compute $H$, that will be detailed in the next 2 subsections.

1. Minimize the expected loss, where the expectation is taken with respect to the location of $x_i$’s

2. Minimize a worst-case scenario type loss

**Known distribution for $x_{ij}$**

If we know the distribution of $x_{ij}$ we can take the expected loss defined as follows:

\[
\mathbb{E}_{\{\theta_i(s)\}} \left( H(\{\theta_i(s)\}_{i=1}^m; c_1, c_2) \right) = \sum_{i \in I_j(s)} P(x_{ij} \leq s) ||y_i - c_1||^2 + (1 - P(x_{ij} \leq s)) ||y_i - c_2||^2. 
\]

This corresponds to reweighting the loss in a way that takes into account the uncertainty in the location of $x_{ij}$ with respect to $s$.

Since we do not have anymore dependency in the $\theta_i$’s after taking the expectation, we can now minimize $\mathbb{E}_{\{\theta_i(s)\}} (H(\{\theta_i(s)\}; c_1, c_2))$ with respect to $c_1$ and $c_2$ like in (1.5). In that case, we have the following loss function:

\[
L(j, s) = \min_{c_1, c_2} \mathbb{E}_{\{\theta_i(s)\}} \left( F(\{\theta_i(s)\}_{i=1}^n; c_1, c_2) \right). 
\]

(1.15)

In this case the $c_i$’s are simply the weighted means of the responses of the observations that have a probability $> 0$ to fall in the corresponding region (weighted by the probability that each observation falls into the region).

However, this approach depends on the knowledge of the distribution of the errors in an important fashion, and we usually do not know it. For this reason, we do not use this method, but try a more pessimistic approach described in the next subsection.

**Worst-case scenario approach**

To build robust trees, we want to look at the **worst case allocation** of the $x_{ij}$’s. This corresponds to getting rid of the $\theta_i$’s in the expression (1.12) by looking at:

\[
\max_{\theta_i(s) \in \{0, 1\}} \min_{c_1, c_2} H(\{\theta_i(s)\}_{i=1}^m; c_1, c_2). 
\]

(1.16)

In other words, look at the ‘natural’ optimized loss function (which consists in taking the minimum with respect to $c_1$ and $c_2$ like in (1.3)), and consider its worst-case behavior.
Thanks to the expression (1.12), we then have:

\[ L_{rob}(j, s) = \max_{\theta_i(s) \in [0,1]} \left( \min_{c_1, c_2} \left( \min_{\{\theta_i(s)\}_{i=1}^m; c_1, c_2} \right) + \min_{c_1, c_2} \left( \sum_{i \in I(s), P_j(B_i) < s} \|y_i - c_1\|^2_2 + \sum_{i \in I(s), P_j(B_i) > s} \|y_i - c_2\|^2_2 \right) \right). \]

(1.17)

The problem is that this optimization problem cannot be tackled directly, because the number of cases of allocation to explore for the outer maximization \( \{\theta_i(s)\}_{i=1}^m \in \{0,1\}^m \) is \( 2^m \), which is computationally prohibitive for large \( m \) (we remind the reader that \( m = |I_j(s)| \) is the number of observations whose box is intersected by the split \( s \)). Instead, we will use a trick described in the next section to show that we can see this as a convex optimization problem, as for the usual CART algorithm.

### 1.3 Robust CART algorithm

The goal of this section is to show that the min-max approach to CART described in (1.16) can be seen as a slightly modified version of the regular CART optimization problem (1.5), where the function \( F \) defined in (1.2) is changed but still convex. This result is stated in Theorem 1.

Before that, we need to prove a lemma that will enable us to invert ‘min’ and ‘max’ in (1.16).

#### 1.3.1 A powerful Lemma

We want to simplify the expression of the loss function of the worst-case allocation approach given in the previous section.

This is the goal of the following Lemma:

**Lemma 1** For \( H \) defined as in (1.14), we have

\[ \max_{\theta_i(s) \in [0,1]} \inf_{c_1, c_2} H(\{\theta_i(s)\}_{i=1}^m; c_1, c_2) = \inf_{c_1, c_2} \max_{\theta_i(s) \in [0,1]} H(\{\theta_i(s)\}_{i=1}^m; c_1, c_2). \]

(1.18)

**Proof**

We remark that the function we are considering

\[ H(\theta_i(s); c_1, c_2) = \sum_{i \in I(s)} \theta_i(s) ||y_i - c_1||^2_2 + (1 - \theta_i(s)) ||y_i - c_2||^2_2, \]

is continuous, convex in \( (c_1, c_2) \) (at \( \{\theta_i(s)\} \)'s given) and concave in \( \{\theta_i(s)\} \) (when \( c_1, c_2 \) are given).

From p.281 of [3] or p.95 of [2], we know that Kakutani’s theorem states that if \( C \) and \( \Theta \) are compact convex sets, and \( H \) such a function

\[ \sup_{\Theta} \inf_{C} \sup_{\Theta} H(\theta_i(s); c_1, c_2) = \inf_{C} \sup_{\Theta} H(\theta_i(s); c_1, c_2). \]

Here \( C \) is not compact a priori, but we can restrict the possible values of \( c_1 \) and \( c_2 \) to the range of the \( y \), and then include it in a compact convex set.

We now use the relaxation trick on the \( \theta_i \)'s,

\[ \max_{\theta_i(s) \in [0,1]} \inf_{c_1, c_2} H(\theta_i(s); c_1, c_2) \geq \max_{\theta_i(s) \in [0,1]} \inf_{c_1, c_2} H(\theta_i(s); c_1, c_2). \]

Thanks to what was said earlier, and to the fact that \([0,1]\) is convex and compact, we can apply Kakutani’s theorem and get:

\[ \max_{\theta_i(s) \in [0,1]} \inf_{c_1, c_2} H(\theta_i(s); c_1, c_2) = \inf_{c_1, c_2} \max_{\theta_i(s) \in [0,1]} H(\theta_i(s); c_1, c_2). \]

(1.19)
Since \[
\max_{\theta_i(s) \in [0,1]} H(\theta_i(s); c_1, c_2) = \max_{\theta_i(s) \in [0,1]} \sum_{i \in I(s)} \theta_i(s)||y_i - c_1||^2 + (1 - \theta_i(s))||y_i - c_2||^2,
\]
the inner maximization on the right of equation (1.19) is easily solved: for given \(c_1\) and \(c_2\), we see that
\[
\theta_i(s) = 1(||y_i - c_1||^2 \geq ||y_i - c_2||^2).
\]
So the maximizing \(\theta_i\)s have values in \(\{0, 1\}\) and it is possible to write:
\[
\inf_{c_1, c_2} \max_{\theta_i(s) \in [0,1]} H(\theta_i(s); c_1, c_2) = \inf_{c_1, c_2} \max_{\theta_i(s) \in [0,1]} H(\theta_i(s); c_1, c_2).
\]
Combining the previous equation we get the following inequality:
\[
\max_{\theta_i(s) \in [0, 1]} \inf_{c_1, c_2} H(\{\theta_i(s)\}_{i=1}^n; c_1, c_2) \leq \inf_{c_1, c_2} \max_{\theta_i(s) \in [0,1]} H(\{\theta_i(s)\}_{i=1}^n; c_1, c_2).
\]
Since the other equality is always true, we conclude that:
\[
\max_{\theta_i(s) \in [0, 1]} \inf_{c_1, c_2} H(\{\theta_i(s)\}_{i=1}^n; c_1, c_2) = \inf_{c_1, c_2} \max_{\theta_i(s) \in [0,1]} H(\{\theta_i(s)\}_{i=1}^n; c_1, c_2).
\]
QED.

Remark:
The previous lemma is not specific to \(L_2\) loss. It could be applied to any convex function of \(c_1, c_2\). In other words, we could have:
\[
H(\{\theta_i(s)\}_{i=1}^n; c_1, c_2) = \sum_{i \in I(s)} \theta_i(s)\gamma(c_1, y_i) + (1 - \theta_i(s))\gamma(c_2, y_i)
\]
provided \(\gamma\) is convex.
This is important because it allows us to handle more general versions of CART as well as problems coming from classification that also use convex functions.

1.3.2 Final expression of the robust CART algorithm

Picking a split

The next theorem shows how one can pick a split in the min-max approach to CART in the same way it is done for usual CART.

Theorem 1 Picking a split in the min-max approach to the CART algorithm as described in section 1.2 narrows down to do it for the usual CART algorithm described in Section 1.1, with the following loss function:
\[
L_{rob}(j, s) = \min_{c_1, c_2} F_{rob}(c_1, c_2, j, s).
\]
(1.20)

where \(F_{rob}\) is convex in \(c_1, c_2\) and has for expression:
\[
F_{rob}(c_1, c_2, j, s) = \frac{1}{2} \sum_{i \in I(s)} ||y_i - c_1||^2 + ||y_i - c_2||^2 + ||y_i - c_1||^2 - ||y_i - c_2||^2 \quad (1.21)
\]
\[
+ \sum_{i \in I(s), P_j(B_i) < s} ||y_i - c_1||^2 + \sum_{i \in I(s), P_j(B_i) > s} ||y_i - c_2||^2. \quad (1.22)
\]

Proof
Using Lemma 1, and the fact that \(\max(a, b) = \frac{1}{2}(a + b + |a - b|)\), we have that After a slight rewriting we see that
\[
\max_{\theta_i(s) \in [0,1]} \inf_{c_1, c_2} \sum_{i \in I(s)} \theta_i(s)||y_i - c_1||^2 + (1 - \theta_i(s))||y_i - c_2||^2 = \frac{1}{2} \inf_{c_1, c_2} \sum_{i \in I(s)} ||y_i - c_1||^2 + ||y_i - c_2||^2 + ||y_i - c_1||^2 - ||y_i - c_2||^2 |
\]
Using this into the definition of our pessimist loss function in (1.17), we get that:

\[
L_{\text{rob}}(j, s) = \max_{\theta(s) \in [0, 1]} \left( \min_{c_1, c_2} H(\{\theta_i(s)\}_{i=1}^{n}; c_1, c_2) + \min_{c_1, c_2} \left( \sum_{i \notin I(s), P_j(B_i) < s} ||y_i - c_1||^2_2 + \sum_{i \notin I(s), P_j(B_i) > s} ||y_i - c_2||^2_2 \right) \right)
\]

\[
= \min_{c_1, c_2} \frac{1}{2} \sum_{i \in I(s)} (||y_i - c_1||^2_2 + ||y_i - c_2||^2_2 + ||y_i - c_1||^2_2 - ||y_i - c_2||^2_2)
\]

\[
= \min_{c_1, c_2} F_{\text{rob}}(c_1, c_2, j, s).
\]

where we defined:

\[
F_{\text{rob}}(c_1, c_2, j, s) = \frac{1}{2} \sum_{i \in I(s)} (||y_i - c_1||^2_2 + ||y_i - c_2||^2_2 + ||y_i - c_1||^2_2 - ||y_i - c_2||^2_2)
\]

\[
+ \sum_{i \notin I(s), P_j(B_i) < s} ||y_i - c_1||^2_2 + \sum_{i \notin I(s), P_j(B_i) > s} ||y_i - c_2||^2_2.
\]

For a given \( s \), this loss function corresponds to the minimization of the natural loss function in the worst case allocation of the observations for which the uncertainty box is cut by \( s \).

To finish the proof, we still have to prove that \( F_{\text{rob}} \) is convex in \( c_1, c_2 \), which is directly implied by the fact that the sup of convex functions is convex.

QED.

Splitting the data

We just proved that the min-max approach to the CART algorithm could be narrowed down to the usual CART with a different loss function \( L_{\text{rob}} \). Since this function is defined as the minimum with respect to \( c_1 \) and \( c_2 \) of the convex function \( F_{\text{rob}} \), it is therefore possible to compute it like for the loss function defined in (1.3), and to pick up the split which reduces \( L_{\text{rob}} \) the most.

However, we still need to chose how to split the data once the splitting variable and split point are chosen, so that we could exactly apply the algorithm described in section 1.1.

Let \( s \) be the chosen split (with respect to feature \( j \)), and \( I_j(s) \) defined as in (1.11).

- We assign each observation \( x_i \in I_j(s) \) to the region it was assigned to in the worst allocation approach. Once the loss function is optimized, we have \( c_1 \) and \( c_2 \), and can therefore use the following formula:

\[
\theta_i(s) = 1(||y_i - c_1||^2_2 \geq ||y_i - c_2||^2_2)
\]

(1.23)

to find how each observation of \( I_j(s) \) was allocated. Moreover, we choose to update the box \( B_i \) of the observation \( x_i \) to its intersection with the chosen region. In other words, if the worst case allocation of \( x_i \) was on the left of the split \( s \) we replace \( B_i \) by \( B_i' \) where

\[
B_i' = B_i \cap \{x|x_j \leq s\}
\]

(1.24)

respectively \( B_i' = B_i \cap \{x|x_j > s\} \) if the assignment of \( x_i \) was on the right of \( s \).

- For observations that are not in \( I_j(s) \), their uncertainty box has no reason to be changed and the allocation does not raise an issue since the box is entirely on one side or the other of the split.
Summary of the robust CART algorithm to build a tree

We use the usual greedy algorithm described in section 1.1. to tackle the robust version of CART:

- We pick the variable for which the best split reduces the most the loss function $L_{rob}$ defined in (1.20).
- We then split the data according their assignation on the left or the right of the split into two subsets of the original data like described in the previous section.
- We update the boxes of the training data according to their assignation on the left or the right of the split following (1.24).
- We run the algorithm again on each of the two data subsets. We do this until we reach a minimum number of observations in each subset like for the usual CART algorithm described in section 1.1.

1.3.3 Pruning the tree

Pruning the tree in this case can be done exactly the same way as for usual CART, since it does not matter how the tree was built: it only matters that we can define the deviance of a node as mentioned in subsection 1.1.3., so that the penalized loss function $R_\alpha$ defined in (1.7) can be computed.

In the case of a tree $T_{rob}$ built with the robust CART algorithm as above, every observation is assigned to one and only one final region of the feature space, and has a continuous response variable. The fact that there is some uncertainty in the features does not come into play when computing the sample variance of the responses in each final region.

Therefore, $\forall \alpha \geq 0$, we can define $R_\alpha(T_{rob})$ and perform pruning.

In the next chapter, we will detail the ‘weakest link’ pruning algorithm defined in the third chapter of [4], and explain how one can have a different approach to it.

1.3.4 Predicting from the tree

The pruning is done by minimizing $R_\alpha$ as defined in (1.7) for a given complexity parameter $\alpha$. However, the optimal complexity parameter is usually picked by cross validation: we prune the big tree obtained after the building step according to a sequence of $k \in \mathbb{N}$ complexity parameters $\alpha_1, \ldots, \alpha_k$, and we estimate the test error of each of those trees by cross validation.

Therefore, we still need to define a way to do prediction with a tree built as above for a new observation that has potential uncertainty in its features.

Let us consider a new observation $x$, and its corresponding uncertainty box $B_x$. Let $p \in \mathbb{N}$ and $R_1, \ldots, R_p$ be the $p$ regions defined by the considered tree that overlap $B_x$ (for which $B_x \cap R_i \neq \emptyset$). Let $v_1, \ldots, v_p$ be the corresponding values for each region.

There are two ways one can think of predicting the response for this new value: volumic prediction and min-max prediction.

Volumic prediction

In the case of ‘volumic prediction’, we identify every region defined by the tree that overlaps the boxes of the new observation, and we predict a weighted mean of the corresponding responses values (weighted by the volume of the new observation’s box that overlaps with the considered region).

With $\hat{y}_{vol}$ being the estimation of the response corresponding to $x$, we can write:

$$\hat{y}_{vol} = \frac{1}{|B_x|} \sum_{j=1}^{p} |B_x \cap R_j| v_j$$

(1.25)
Where here \( |\cdot| \) represents a measure of volume. This method is very intuitive but corresponds more to the distribution approach and does not reflect the worst-case robust idea of the robust CART algorithm.

**Minmax prediction**

The second way is to do a ‘minimax’ prediction.

A way to predict a response for \( x \) in a minmax way would be to assign it the prediction \( \hat{y}_{\text{mm}} \) such that:

\[
\hat{y}_{\text{mm}} = \min_{y \in \mathbb{R}} \max_{v \in \{v_1, \ldots, v_p\}} |y - v|
\]  

(1.26)

The following Lemma enables us to simplify the expression (1.26).

**Lemma 2** Let \( \{v_1, \ldots, v_p\} \) be a set of ordered real numbers: \( v_1 \leq v_2 \leq \cdots \leq v_p \), then we have:

\[
\min_{y \in \mathbb{R}} \max_{v \in \{v_1, \ldots, v_p\}} |y - v| = \frac{1}{2} (v_1 + v_p).
\]  

(1.27)

**Proof**

The proof is easier to grasp geometrically: thanks to Figure 1.3.4 we see that the function:

\[
y \mapsto \max_{v \in \{v_1, \ldots, v_p\}} |y - v|
\]

\[\text{mm}\]

\[\text{mm}\]

\[\text{mm}\]

The \( \text{mm} \) index here is a shortcut for min-max.
is in fact the maximum of \( p \) shifted \( y \mapsto |y| \) functions. It is clear that only \( v_m \) and \( v_M \) are used to define \( y \mapsto \max_{v \in \{v_1, \ldots, v_p\}} |y - v| \), and that we can write, \( \forall y \in \mathbb{R} \),

\[
\max_{v \in \{v_1, \ldots, v_p\}} |y - v| = \max_{v \in \{v_1, v_p\}} |y - v|
\]

It is then obvious that the minimum is reached when the curves of the two function meet, which implies:

\[
\min_{y \in \mathbb{R}} \left( \max_{v \in \{v_m, v_M\}} |y - v| \right) = \frac{1}{2} (v_1 + v_p)
\]

**QED.**

Using the Lemma, and with

\[
v_M = \max\{v_1, \ldots, v_p\}
\]

\[
v_m = \min\{v_1, \ldots, v_p\}
\]

we see that (1.26) can we rewritten as:

\[
\hat{y}_{mm} = \frac{1}{2} (v_m + v_M)
\]

(1.28)

### 1.3.5 Final remarks

We now have an algorithm to build a tree, and to make predictions with a built tree for new data with uncertainty boxes. This algorithm can then be naturally generalized to bagging and random forests, that imply to build multiple trees from bootstrap samples of the original training data. It is therefore possible to fit robust random forests to data containing uncertainties. In the next chapter, we will explore in more detail the implementation of those theoretical algorithms. Since we manage to narrow down the robust CART algorithm to a slightly modified version of the usual CART, we chose the `rpart` R package for our implementation, as it enables user-defined splitting functions.
Chapter 2

Algorithms and implementation

Most of my research work consisted in coding and testing the robust (minmax) approach over the problem. As explained in the end of the previous chapter, the rpart R package was chosen as a basis of our implementation, since it enables the user to define his own splitting functions.

I will proceed in 4 steps to describe my work:

- I will first explain the main functions and attributes of the rpart R package. I will notably detail the structure of the rpart objects that needed to be respected by our code to be able to use the functions of the package.

- Then I will explain the steps of my research on the implementation of the robust CART algorithm using the existing code in the rpart package. I will detail the problems that occurred, and how I managed to solve them (sometime by rewriting part of the package).

- I will then make a detailed list of the main functions of my code, and explain how they work together and respect the structure of the rpart objects.

- Finally, I will focus on the pruning part of the algorithm:
  - I will first explain how the ‘weakest link’ recursive pruning is done in Chapter 3 of [4].
  - I will then state and prove Theorem 2, that gives a way to compute the ‘limit’ complexity parameters for each node directly from the tree (and without snipping some of its nodes away as in the recursive algorithm in [4]).
  - I will then show how this direct approach on pruning leads to an approximate algorithm to prune a tree a lot faster than with the recursive weakest link pruning (which is particularly important in our case since all the core code was done in R, and is therefore pretty slow).

2.1 The rpart package

2.1.1 The structure of the rpart objects

The rpart R package enables us to build, prune, and plot decision trees (like the tree package, but rpart was chosen because it enabled us to design our own splitting functions).

In order to be able to also prune and plot the objects that our algorithm will return, we chose to respect the structure of the rpart objects: two of the main attributes of rpart objects are the following:

- The frame attribute: a dataframe that has a row for each node, and information for the nodes in each column, including:
  - var: the name of the variable according to which the split is made in this node (and <leaf> for leaves.
  - n: the number of training observations in this node.
  - dev: the deviance of the observations in the node.
- \textit{yval}: the mean value of the response for the training observations in the node.
- \textit{complexity}: the value of the smallest complexity parameter for which the node collapses when pruning the tree.

- The \textit{split} attribute: a dataframe that contains all the information of the chosen splits in the tree that are not present in \textit{frame}. Each row is a split and the columns are:
  - \textit{improve}: the improvement in the deviance implied by this split.
  - \textit{index}: the value at which the split is made.
  - The row names refer to the variable according to which the split is made (same as the \textit{var} column of \textit{frame}).

2.1.2 The main functions of the \texttt{rpart} objects

The following functions of the package \texttt{rpart} only use the two previously described attributes and are therefore usable if we respect the structure of the objects:

- \texttt{snip}: a function that snips away all the nodes of the tree that verify a certain criterion.
- \texttt{plot}: a function that plots the tree.
- \texttt{text}: a function that writes the text on the plot of the tree (the split variables, values and the mean values in the leaves).

These three functions are particularly useful for looking at the structure of the tree and pruning it down. It is therefore important to respect the structure of the objects in the implementation of the robust version of the CART algorithm, in order to still be able to use the functions after the construction of the trees.

2.2 Implementation ideas - Dealing with the existing code

I wanted the ‘plumbing’ in the \texttt{rpart} code to be as simplest as possible to be able to use the built-in functions. However, I faced several issues that forced me to adapt a lot of the package to robust trees.

First issue: storing the uncertainty boxes of each observation in the data

The main function of the package \texttt{rpart}, apart from the usual arguments (that are a formula, a dataframe and a method), takes a supplementary argument called \texttt{parms} (\texttt{NULL} by default), that can be used to store the uncertainty boxes of the data. The idea was then to code splitting functions using the \texttt{parms} argument to find the best splits, and to update it each time a split was found: as described earlier, each time a uncertainty box is cut by the chosen split, we assign the corresponding observation to one or the other side of the split, and update the box accordingly.

Second issue: dynamical update of the \texttt{parms} argument

However, after reading the documentation of the package [6], and looking at the source code, a problem appeared: coding splitting functions did not enable the user to update dynamically the boxes of the data. The \texttt{parms} argument cannot be changed after the initial call of the function \texttt{rpart}.

To face this issue, we decided to build only trees of depth 1 thanks to the existing \texttt{rpart} function, and to code a wrapper function in R, which assembled these trees recursively into one big tree.

This solves the problem of the dynamical update of the boxes (that could then be done in the wrapper function), and my first weeks consisted in coding this function so that it outputted the objects of the exact same structure as the trees outputted by the function \texttt{rpart}. Using the \texttt{rpart} function implies that all our depth 1 trees are \texttt{rpart} objects already. This is very convenient because it enables us to assemble them into big \texttt{rpart} trees more easily (which can then be plotted, printed and pruned thanks to the existing function of the \texttt{rpart} package).
However a third issue rose: the user-defined splitting functions had structure constraints that could not be bent to adapt to the robust CART algorithm.

**Third issue: constraints on the user-defined splitting functions**

Two main issues came from the constraints on the input and the output of the user defined splitting functions:

- The input of the splitting function contains an *unnamed* column of the data set passed in argument of the `rpart` function. This causes a problem in our case because each column referred to part of the `parms` argument. But without any information on the name of the column, it was not possible to know which boxes should be looked at.
  
  - A first solution was to include the data set passed in argument in the `parms` argument, and to identify the column taken as an argument by the splitting function by comparing it to all the columns of the data set included in `parms`. The problem is that testing the equality of two columns is computationally heavy.
  
  - Another solution (that I finally used) to solve this problem is to fill the input data set with indexes, each referring to the column of the `parms` argument containing the associated box. The `parms` argument is now a matrix with 2 lines and as many columns as the number of features times the number of observations in the data. The first line corresponds to the left bounds of the 1-dimension boxes around each feature value, and the second line corresponds to the right bounds.

- The output of the splitting function must be a vector of the goodness of all possible splits. This vector must be at most the same size as the number of lines in the data, since there is at most one possible split for each value of the considered feature. However, the robust CART algorithm implies that there is up to twice as much possible splits, as the left and right bound of every interval can be considered a split. I did not find any clean solution to this specific problem.

I therefore decided to code my own function to build depth 1 trees that would be as close as possible to `rpart` objects and that could be fed to the wrapper function. In the next section, I will explain the structure of my code and the main functions that I used to make this implementation.

We remark that this way of coding the robust CART algorithm makes it way longer than the `rpart` package for three reasons:

- The code is done in R whereas it is done in C for `rpart`.
- There are more splits to explore.
- The optimization function that is used to test the goodness of all possible splits is more complicated than the original CART optimization function (and we need to keep track of the assignation of each observation whose box intersects the split).

**2.3 The Algorithm implementation**

All the implementation that I did for this research project was in R.
2.3.1 Building the tree

The last issue mentioned in the previous section forced me to re-code the building of the tree from the beginning (finding the best split).

The algorithm for building of the tree is articulated around 4 main functions:

- The function `bestsplits`, that finds the best split. This function goes through every possible split with respect to every feature, and outputs the split that minimizes the cost function described in (1.20).

- The function `deptonetree` that builds a tree of depth 1 (consisting of only one (root) node, and two leaves, each representing a region of the data space).
  This function is crucial because it separates the data in the most pessimistic way thanks to the best split returned by the previous function, and assigns to every observation a region (including the ones whose box is cut by the split). Therefore, it also updates the boxes to their intersection with the chosen region (which in our case is actually updating the `parms` argument, thanks to the function `updateparms`).

- We also need the function `buildtree`, that assembles two trees into one big tree whose root node has the two initial trees as left and right sons.

- Finally, there is a recursive function `generaltree`:
  - which calls `deptonetree` if the data verifies a certain condition (usually, it is when the number of observations still present in the data is higher than a given threshold).
  - and which calls itself twice otherwise, for the data on each side of the chosen split (to produce the left and right sons), then calls `buildtree` to assemble the two sons into one parent tree.

2.3.2 Predicting

As discussed in the ‘Theoretical setting’ section, we found two main ways to predict from such a built tree:

- the volumic prediction approach

- the minmax prediction approach

The algorithm in both cases is pretty straightforward:

- For the volumic prediction approach, we code a recursive function `predictvolume` which takes as arguments a tree and one new observation.
  - When called on a leaf of the tree, this function simply returns the mean of the response variable for the part of the training set assigned to the corresponding region during the construction of the tree (this value is present in the corresponding `rpart` object).
  - If the the box around this new observation is cut by the root split of the tree, it splits the observation’s box according to this split, and returns the weighted mean (according to the volume of the two sub-boxes like in (1.25)) of its outputs when called on each of the two sub-boxes and the left and right subtrees.
  - If the box is not cut by the split, the observation is assigned to a side of the split, and `predictvolume` calls itself on the same observation and the corresponding subtree.

- The minmax prediction function `predictminmax` is even simpler: the function also takes as arguments a tree and one new observation.
The function first identifies the regions that have an intersection with the new observation’s box \( (R_1, \ldots, R_k \text{ such that } \forall j \in \{1, \ldots, k\}, R_j \cap B_x \neq \emptyset) \). For this, it does like `predictvolume` and goes through the whole tree starting from its root node, storing at each step the nodes that correspond to a region intersecting \( B_x \).

- It then extracts the set of corresponding values \( v_1, \ldots, v_k \), and returns the output of the formula (1.28) which is simply the mean of the biggest and the smallest \( v_i \) for \( i \in \{1, \ldots, k\} \).

### 2.3.3 Pruning the tree

#### Specific notations for pruning

In what follows, we will use the notations \( T \) to designate a tree, and \( \tilde{T} \) for its set of terminal nodes. Some of the notations were introduced in the first chapter but we remind them here for clarity’s sake.

- We call \( \text{dev} \) the deviance of a node (i.e. the variance of the response for every observation in this node). The function \( R \) is defined p.63 of [4], and referred to in chapter 9.2 of [5] for every tree \( T \), as:

\[
R(T) = \sum_{t' \in \tilde{T}} \text{dev}(t')
\]

\( R_\alpha \) is the penalized loss, and is defined for every \( \alpha \geq 0 \) and every tree \( T \) as:

\[
R_\alpha(T) = R(T) + \alpha |\tilde{T}|
\]

In this definition, \( |\tilde{T}| \), the number of terminal nodes, represents the complexity of the tree \( T \). \( R_\alpha(T) \) expresses the cost-complexity tradeoff of the tree \( T \): the term \( R(T) \) gets smaller when the tree size increases, but the second term \( \alpha |\tilde{T}| \) prevents trees to be too complex by penalizing big trees.

- \( T_\alpha \) is the smallest minimizing subtree of \( T \) for \( R_\alpha \). It is defined by the following set of equations:

\[
T_\alpha \in \arg\max_{T \in S_T(\alpha)} (R_\alpha(T)) \\
\forall T' \in S_T(\alpha), R_\alpha(T') = R_\alpha(T_\alpha) \Rightarrow T_\alpha \subset T'.
\]

The existence of the smallest minimizing subtree is proven in Chapter 10 of [4], and its uniqueness is directly implied by the definition.

- For a tree \( T \), let \( C_T \subset T \) be the set of the collapsing nodes of \( T \) (as defined previously: they are the nodes that will collapse during the pruning algorithm). The pruning of the tree is entirely determined by the function \( c \) defined as follows:

\[
c_T : \begin{cases} 
T & \rightarrow \mathbb{R}^+ \\
t & \rightarrow \sup\{\alpha | t \in T_\alpha\}
\end{cases}
\]

Since \( \forall \alpha' \geq \alpha, T_{\alpha'} \subset T_\alpha \), \( c_T \) assigns to each node of \( T \) the complexity parameter \( \alpha \) starting from which the node will not anymore be in the smallest minimizing subtree \( T_\alpha \).

There are two main steps to prune optimally the tree: we must first identify the function \( c_T \) described earlier, then find \( \alpha \) for which the corresponding pruned tree \( T_\alpha \) is the best in terms of cost-complexity trade-off.

More precisely:

1. Identifying the function \( c_T \) is detailed in chapter 3 of [4]. For now, we only assume that this can be done immediately after building the tree thanks to a recursive algorithm.

   In the next section, we will focus on this step only and show a different approach than the one

\[1\text{This makes sense because } \tilde{T} \text{ is entirely defined by } T.\]

18
discussed in [4] to compute these limit parameters. We do this thanks to a function `complexitycheck` that takes as argument a tree returned by `generaltree` and adds a column corresponding to this limit parameter for every node of the tree in the `frame` attribute of the corresponding `rpart` object.

2. Now, to find the best $\alpha$ we have to select a set of $\alpha$'s to test, and perform cross validation to find which parameter produces the pruned tree with the lowest test error. This step is very common in machine learning and simply implies to code a function that uses `generaltree` to build a tree on part of the training set, and then compute the approximate test error made by this tree on the remaining of the training set (using `predictvolume` or `predictminmax`), and the usual $L_2$ loss. For this, we need a function that snips away all the nodes of a tree that verify a certain criterion (in this case, the nodes for which the limit complexity parameter computed with `complexitycheck` is lower than the $\alpha$ that is used to prune the tree). This did not cause any problem because the function `snip` was already coded for `rpart` objects, and could be applied to the trees returned by `fulldepthtree` and `complexitycheck` (thanks to the fact that the outputs respected the structure of the `rpart` objects).

2.4 Different approach on pruning

In this section, we will show how one can have a different approach on the way to prune a tree than what is presented in chapter 3 of [4].

- We will first introduce the notations that we need to explain pruning.
- We will then explain how the weakest link pruning done in chapter 3 of [4] works, and therefore show the fact that its worst-case complexity is quadratic in the number of nodes of the tree.
- Then, we will state and prove Theorem 2 that gives a direct way to compute the limit complexity parameters of the nodes.
- Finally, we will explain an approximate algorithm based on the direct formula of Theorem 2, that enables us to perform pruning in linear complexity.

2.4.1 Notations

We first define the main notations we will need. We will illustrate them thanks to the following tree.
• Let $T$ be a tree.

- $T(T)$ designates the set of non terminal nodes of the tree $T$. When there is no ambiguity concerning the tree $T$ that is referred to, we will simply use the notation $T$.
- $\text{root}(T) \in T$ designates the root of the tree. In our example tree, $T(T) = \{1, 2, 3, 4, 5, 6, 7\}$, and $\text{root}(T) = 1$.
- $\tilde{T}(T)$, or just $\tilde{T}$ when there is no ambiguity concerning the tree $T$ that is referred to, designates the set of terminal nodes of the tree. They are the unnamed rectangles on the previous sketch.

From now on, we will refer to trees by their set of non-terminal nodes, and will use the notation $T$ to designate a tree, and not $T'(T)$. This can be done because the set of terminal nodes $\tilde{T}$ is entirely defined by $T$.

Moreover, we will use the notation $r = \text{root}(T)$.

• For $t \in T$, $t_L$ and $t_R$ designate respectively the left son node and the right son node of the node $t$.
In our example and for $t = 2$ we have, $t_L = 4$ and $t_R = 5$.

• For $t \in T$, we define as $S_T(t)$ the set of all the subtrees of $T$ that have $t$ as a root node.
In our example,
$$S_T(2) = \{\{2\}, \{2, 4\}, \{2, 5\}, \{2, 4, 5\}\}$$

• For $t \in T$, $T_t$ is defined as the biggest subtree of $T$ with $t$ as a root node (e.g $T_2 = \{2, 4, 5\}$).
Every tree in $T' \in S_T(t)$ is a subtree of $T_t$ (i.e. $T' \subset T_t$).

• We define the strength function $s$ as follows:
$$s : \left\{ \begin{array}{c}
T \longrightarrow \mathbb{R}^+ \\
t \mapsto s(t) = \text{dev}(t) - (\text{dev}(t_L) + \text{dev}(t_R))
\end{array} \right.$$  

The strength of a node corresponds to the amount of deviance that is gained thanks to this node.

For a tree $T'$, we define $\bar{s}(T')$ as the average strength of the nodes of $T'$:
$$\bar{s}(T') = \frac{1}{|T'|} \sum_{t' \in T'} s(t')$$  \hspace{1cm} (2.1)
and by convention $\bar{s}(\emptyset) = 0$.

• We define the function $g$ as in [4]:
$$g : \left\{ \begin{array}{c}
T \longrightarrow \mathbb{R}^+ \\
t \mapsto g(t) = \frac{\text{dev}(t) - R(T_t)}{|T_t| - 1}
\end{array} \right.$$  

We can easily link this definition to the notations we previously introduced: for a tree $T$ and $t \in T$, using the fact that $|\tilde{T}| = |T| + 1$ we get:
$$g(t) = \frac{1}{|T_t|} \sum_{t' \in T_t} s(t') = \bar{s}(T_t).$$
2.4.2 Pruning the tree in a non-recursive way

Weakest link pruning algorithm from [4]

The pruning algorithm explained in Chapter 3 of [4] is recursive. We will explain quickly how it works in this first subsection, and then state Theorem 2 to show how it is possible to apprehend it in a non-recursive way.

Let $T$ be a tree, $T$ and $\tilde{T}$ its respective set of non-terminal nodes and set of terminal nodes.

- We find the ‘weakest link’ of the tree, defined as the node $t_1 \in T$ which minimizes the function $g$, and which is the closest to the root node. There might be several of those, in which case we pick one at random.

- We then make this node collapse, meaning that it becomes a terminal node, and that every one of its descendants is snipped away from the tree. We call the new tree $T'$.

We define $\alpha_1 = g(t_1)$ and we have

$$\alpha_1 = \sup\{\alpha | t \in T_\alpha\}$$

- We Finally define $T' = T \setminus T_1$, and we run the algorithm again on $T'$.

We follow these few steps until we are left with a tree containing only one terminal node. Defining $K$ as the number of steps that needed to be achieved. This algorithm makes us obtain a sequence of nodes $t_1, \ldots, t_K$ and of numbers $\alpha_1, \ldots, \alpha_K$, which are respectively the ‘collapsing nodes’ of the tree $T$, and the complexity parameters for which these nodes collapse in the tree $T_\alpha$ minimizing the penalized loss $R_\alpha$.

Remarks for the proof of the upcoming theorem

We remind the reader the definition of $c_T$:

For a tree $T$, let $C_T \subset T$ be the set of the collapsing nodes of $T$ (as defined previously: they are the nodes that will collapse during the pruning algorithm).

The pruning of the tree is entirely determined by the function $c$ defined as follows:

$$c_T : \begin{cases} T \rightarrow \mathbb{R}^+ \\ t \mapsto \sup\{\alpha | t \in T_\alpha\} \end{cases}$$

- Knowing $c_T$ on the set of collapsing nodes $C_T$ is enough to perform the pruning since only the collapsing nodes matter in the pruning algorithm.

- Using the notations described in the previous subsection, we have by definition for a tree $T$ that:

$$C_T = \{t_1, \ldots, t_K\}$$

$$\forall t_i \in C_T, \ c_T(t) = \alpha_i.$$  \hspace{1cm} (2.2)

In particular we have that

$$g(t_1) = c_T(t_1).$$

- For $t_i \in C_T$ and with $T_i$ being the pruned tree in which $t_i$ collapses (the pruned subtree of $T$ over which is run the algorithm during its $i^{th}$ call), we have that

$$c_{T_i}(t_i) = c_T(t_i).$$

by the fact that the algorithm from [4] is recursive, and that finding $c_T(t_i)$ will imply to prune $T$ to $T_i$ first.
A new formula

It is actually possible to define the function $c$ for each node in a non recursive way, but with a function depending only on the tree $T$.

**Theorem 2** For any tree $T$, let us define the following function:

$$
\tilde{c}_T : \begin{cases} 
T & \rightarrow \mathbb{R}^+ \\
t & \rightarrow \max_{T' \in S_T(t)} \bar{s}(T') 
\end{cases}
$$

We have that:

$$
\forall t \in C_T, \tilde{c}_T(t) = c_T(t).
$$

(2.3)

This formula is the mathematical expression of the fact that a node is as strong as its strongest subtree. In the next subsections, we will prove formula (2.3), and we will see how this enables us to prune the tree in an approximate non-recursive way.

### 2.4.3 A proof by induction of (2.3)

We are going to prove this result by strong induction on $n = |T|$.

- $n = 1$
  The case $n = 1$ corresponds to a tree consisting of one root node and two terminal nodes: we have $T = \{r\}$.
  In that case $g(r) = \text{dev}(r) - R(T_r) = s(r) = \bar{s}(T_r)$.
  Therefore the algorithm from [4] says that the parameter at which the single non-terminal node collapses is $\alpha_1 = c_T(r)$ where:
  $$
c_T(r) = g(r) = \bar{s}(T_r).
$$
  Moreover, since $S_T(r)$ only contains one tree ($T = T_r$), we have $\tilde{c}_T(r) = \bar{s}(T_r)$. And therefore,
  $$
c_T(r) = \tilde{c}_T(r)
$$
  which is what we wanted to prove.

- $1, \ldots, n - 1 \Rightarrow n$
  This is possible $\forall n \in \mathbb{N}$.
  **We now suppose that the theorem is true for all trees $T$ such that $|T| < n$.** We are going to prove that this implies that the property is true for a tree of size $n$.

  Let $T$ be a tree such that $|T| = n$, and let $t_1$ satisfy:

  $$
t_1 \in \text{argmin}_{t' \in T} g(t')
$$

(2.4)

where $t_1$ is picked so that its depth\(^2\) is minimal. This is the same definition than the first collapsing node $t_1$ from the algorithm in [4].

We will now proceed in 3 steps:

- First we will show that $\tilde{c}_T(t) = c_T(t)$
- Then we will show that $\forall t \in T \setminus T_t, \tilde{c}_T(t) = \tilde{c}_{T \setminus T_t}(t)$
- Finally we will do a wrapup to explain how this implies that the theorem is true for $T$.

\(^2\)The depth of a node is defined as the number of vertices that separate it from the root node.
1. Let us prove that $T_{t_1} \in \arg\max_{T' \in S_T(t_1)} \bar{s}(T')$, which implies that $g(t_1) = \tilde{c}_T(t_1)$.

If it were not true, there would be $T' \in S_T(t_1)$ a strict subtree of $T_{t_1}$ such that:

$$\bar{s}(T') > \bar{s}(T_{t_1})$$

In that case, there would be $k \geq 1$ nodes $t^{(1)}, \ldots, t^{(k)} \in T_{t_1}$ such that

$$T_{t_1} = T' \sqcup T_{t^{(1)}} \sqcup \cdots \sqcup T_{t^{(k)}}$$

We could then write that:

$$g(t_1) = \bar{s}(T_{t_1}) = \text{avg}(\bar{s}(T'), \bar{s}(T_{t^{(1)}}, \ldots, \bar{s}(T_{t^{(k)}}))$$

$$> g(t_1)$$

which is impossible. Therefore,

$$\tilde{c}_T(t_1) = \bar{s}(T_{t_1}) = g(t_1) = c_T(t_1)$$

The last equality comes from (2.2).

2. Since we know that $\tilde{c}_T(t_1) = c_T(1)$, all we need to prove to be able to use the induction hypothesis is that for the next collapsing node, $\tilde{c}_T$ and $\tilde{c}_T \setminus T_{t_1}$ will return the same value.

We are going to prove the more general result that snipping away $T_{t_1}$ from $T$ does not change the value of $\tilde{c}$ on all the nodes $T \setminus T_{t_1}$, i.e that

$$\forall r \in T \setminus T_{t_1}, \tilde{c}_T(r) = \tilde{c}_T \setminus T_{t_1}(r). \quad (2.5)$$

We suppose that there is $r \in T$ such that $T_r^* \cap T_{t_1} \neq \emptyset$ where $T_r^*$ is a subtree of $T$ satisfying

$$T_r^* \in \arg\max_{T' \in S_T(r)} \bar{s}(T'). \quad (2.6)$$

Since $T_r^* \in S_T(r)$, this implies that $r$ is an ancestor of $t_1$. In that case, there are trees $T_r^*$ and $T_{t_1}^*$ such that

$$T_r^* = T_r' \sqcup T_{t_1}^*$$

$$T_{t_1}^* \subset T_{t_1} \text{ and } T_{t_1}^* \neq \emptyset$$

We then have:

$$\bar{s}(T_r^*) = \text{avg}(\bar{s}(T_r'), \bar{s}(T_{t_1}^*))$$

Moreover, thanks to the property (2.4) that implies that $g(t_1) < g(r)$, we have the following equations:

$$\bar{s}(T_{t_1}^*) \leq \bar{s}(T_{t_1}) = g(t_1)$$

$$\bar{s}(T_r^*) \geq \bar{s}(T_r) = g(r) > g(t_1)$$

---

3In what follows, ‘avg’ designates the weighted average (the weights are the number of nodes in each of the subtrees used in argument, but will not be written because not relevant as long as they are positive)
Therefore,
\[
\bar{s}(T^*_r) = \frac{\text{avg}(\bar{s}(T'_r), \bar{s}(T'_{t_1}))}{\geq g(t_1) \leq g(t_1)} > g(t_1) = \text{avg}(\bar{s}(T'_r)) \leq g(t_1)
\]

Since both elements in the weighted average have positive weights, this implies that \(\bar{s}(T'_r) > \bar{s}(T^*_r)\).

This is impossible because it violates the definition of \(T^*_r\) in (2.6).

Therefore the assumptions are false, and
\[
T^*_r \cap T_{t_1} = \emptyset
\]

This implies that \(T_{t_1}\) is not part of the strongest son of any of its ancestors.

3. Final wrapup of the proof:
   With \(T' = T \setminus T_{t_1}\), we conclude that
   \[
   \forall r \in T' \; \tilde{c}_T(r) = \tilde{c}_{T'}(r)
   \] (2.7)

Moreover, \(|T'| < n\), and so by induction hypothesis, if \(t_2, \ldots, t_K\) are the collapsing nodes of \(T'\) we know that
\[
\forall t \in (t_2, \ldots, t_K), \; \tilde{c}_{T'}(t) = c_{T'}(t)
\]

Thanks to (2.7), we have that
\[
\forall t \in (t_2, \ldots, t_K), \; \tilde{c}_T(t) = c_T(t)
\]

The final remark from the ‘important remarks’ subsection then enables us to write:
\[
\forall t \in (t_2, \ldots, t_K), \; \tilde{c}_T(t) = c_T(t)
\]

Since we proved in the first point that \(\tilde{c}_T(t_1) = c_T(t_1)\) and because the collapsing nodes of \(T\) are \((t_1, t_2, \ldots, t_K)\), we finally have:
\[
\forall t \in C_T, \; \tilde{c}_T(t) = c_T(t)
\]

which proves that the property is true for \(T\) with \(|T| = n\).

- The property is therefore true for \(n = 1\), and it follows the principle of induction. It is therefore true for trees of all sizes.

\textbf{QED.}

2.4.4 Finding the collapsing nodes after computing \(\tilde{c}\)

We proved the fact that for every \(t \in C_T, \; \tilde{c}(t) = c(t)\).

For this to be enough to prune the tree for a given \(\alpha \in \mathbb{R}^+\), we have to show that computing \(\tilde{c}\) enables us to find the collapsing nodes easily.

In the next sketch, the two numbers for each node are: the id number of the node, and its strength.
We now show the few steps that are needed to use \( \tilde{\sigma} \) to find the collapsing nodes of the tree.

- First, we compute the \( \tilde{\sigma} \) for every node. It is easy to compute in this case because there are a limited number of subtrees with a given root node.

   In the following sketch, the second information in the node is now \( \tilde{\sigma} \), the strength of the strongest subtree of \( T \) with this node as a root node.

   - \( \tilde{\sigma}_2 = 2.5 \)
   - \( \tilde{\sigma}_3 = 4.67 \)
   - \( \tilde{\sigma}_4 = 1 \)
   - \( \tilde{\sigma}_5 = 3 \)
   - \( \tilde{\sigma}_6 = 2 \)
   - \( \tilde{\sigma}_7 = 8.5 \)
   - \( \tilde{\sigma}_{15} = 9 \)

In this last tree, here are the strongest subtrees \( T^*_t \) for every node \( t \) (of course for every pre-terminal node, the strongest subtree is the tree consisting of only this node).
<table>
<thead>
<tr>
<th>$t$</th>
<th>$T^*_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${1, 3, 7, 15}$</td>
</tr>
<tr>
<td>2</td>
<td>${2, 5}$</td>
</tr>
<tr>
<td>3</td>
<td>${3, 7, 15}$</td>
</tr>
<tr>
<td>4</td>
<td>${4}$</td>
</tr>
<tr>
<td>5</td>
<td>${5}$</td>
</tr>
<tr>
<td>6</td>
<td>${6}$</td>
</tr>
<tr>
<td>7</td>
<td>${7, 15}$</td>
</tr>
<tr>
<td>15</td>
<td>${15}$</td>
</tr>
</tbody>
</table>

Table 2.1: Value of the strongest subtree $T^*_t$ for every node $t \in T$

- After computing $\tilde{c}$ for all the nodes of the tree $T$, we can easily prune the tree for a given parameter $\alpha$ by making the nodes $t$ such that $\tilde{c} \leq \alpha$ collapse.\(^4\) Making $\alpha$ take the values of $\tilde{c}$ present in the nodes of the tree, we can easily identify $C_T$. The next sketch shows the collapsing nodes in black. For those nodes, the value of $\tilde{c}$ corresponds to the smallest complexity parameter for which this node will not be present in the tree anymore.

- Finally, here is the sequence of subtrees obtained by pruning the tree for all possible values of the complexity parameter $\alpha$.
  
  - Obviously, for $\alpha < 1$, no node collapses and the tree is the one of the last sketch.
  
  - For $1 \leq \alpha < 2$, $T_\alpha$ is the following tree:

\(^4\)We do not wonder about the cases where some nodes disappear because they are the sons of a weaker node, even if their $\tilde{c}$ is bigger than the $\alpha$ parameter. This is how this approach of the pruning works.
- For $1 \leq \alpha < 2$, $T_\alpha$ is the following tree:

For $1 \leq \alpha < 2$, $T_\alpha$ is the following tree:
For $\alpha \geq 4.5$, $T_\alpha$ is the following tree:

This example gives us a glimpse of one of the possible uses of this method: we only compute the $\tilde{c}$ once, and have no computation to do after this initial computation (whereas we had to compute $g(t)$ for every node $t$ at each step of the algorithm explained in [4]).

### 2.4.5 Computational advantages of the new approach on pruning

In this subsection, we will show how the pruning algorithm can be approximated in a way that considerably decreases its complexity. Since we found a way to prune the tree doing only one initial computation for each node (thanks to the function $\tilde{c}$), we are now looking for ways to approximate it by only going once through every node of the tree.

Since this computation would be done in a linear complexity in the total number of nodes, this is computationally interesting for very large trees, since the algorithm for [4] is quadratic in the number of nodes in the worse case scenario.

**An approximation algorithm for pruning a tree in a linear complexity**

Here is the rule that we can use to approximate the best subtrees of every node:

**Approximation to find $T^*$**

Let $T$ be a tree, $r$ its root node, and $T_L$ and $T_R$ its left and right sons.

If $T^*$ is the smallest tree such that $T^* \in \arg\max_{T' \in S_{T^*}(r)} \tilde{s}(T')$ then

- We have the following equality:
  $$ T^* = T'_L \cup \{r\} \cup T'_R $$

  where we set

  $$ T'_L = \emptyset \text{ or } T'_L = T_L $$
  $$ T'_R = \emptyset \text{ or } T'_R = T_R $$
Moreover we use the following criterion to decide if the best subtrees of the sons of $r$ are taken into account of the best subtree of their father:

\[ T'_L \neq \emptyset \iff T'_L = T^*_L \]

\[ \iff \bar{s}(T^*_L) > \max (s(r), \bar{s}(T^*_R \cup \{r\})) \] (2.8)

Respectively for $T'_R$ and $T^*_R$.

**The algorithm**

For a tree with a set of internal nodes $T$, the approximated algorithm for pruning consists of:

- Computing $s(t)$ for every internal node $t \in T$ (this can be done while building the tree).
- Considering the preterminal nodes of the tree, and going up to their parents, each time deciding if the best subtrees of the left and right son should be kept in the best subtree of the parent with the criterion (2.8).

This algorithm has the immense advantage to have a **linear complexity** in the number of nodes in the tree, which can make quite a difference when pruning very large trees.

**Elements of proof of the approximate algorithm**

The following lemma gives us a first insight on how the best subtree of $T$ can be linked to the best subtrees of his left and right sons.

**Lemma 3** Let $T$ be a tree, $r$ its root node, and $T_L$ and $T_R$ its left and right sons.
If $T^*$ is the smallest tree such that $T^* \in \arg\max_{T' \in S_T(r)} \bar{s}(T')$ then

- We have the following equality:

\[ T^* = T'_L \cup \{r\} \cup T'_R \]

where

\[ T'_L = \emptyset \text{ or } |T^*_L| \leq |T'_L| \]
\[ T'_R = \emptyset \text{ or } |T^*_R| \leq |T'_R| \]

- Moreover, we have the following sufficient condition for $T'_L$ and $T'_R$ to be both empty (meaning that the best subtree in $S_T(r)$ is the tree that has only one (root) node $r$).

\[ \bar{s}(r) > \max (\bar{s}(T^*_L), \bar{s}(T^*_R)) \Rightarrow T'_L \cup T'_R = \emptyset \] (2.10)

**Remark**

The second part of the lemma can be interpreted as such: “If the node $r$ is stronger than the best subtrees of its left and right sons, then the best subtree in $S_T(r)$ is the trivial tree that has only one node $r$”.

**Proof**

Let $T^*$ be the smallest tree such that $T^* \in \arg\max_{T' \in S_T(r)} \bar{s}(T')$. Its root node is $r$, so we can write

\[ T^* = T'_L \cup \{r\} \cup T'_R \]

For the sake of the proof, let us suppose that $\bar{s}(T'_L) \geq \bar{s}(T'_R)$.
We are going to prove both points of the lemma at the same time, by discussing on the strength of the root node $r$ and of the subtrees $T'_L$ and $T'_R$.
• If $s(r) > \bar{s}(T'_L)$
  We can then write, supposing that $T'_L \cup T'_R \neq \emptyset$:
  \[
  \bar{s}(T^*) = \text{avg}(\bar{s}(T'_L), s(r), \bar{s}(T'_R)) \\
  < s(r)
  \]
  Since $\{r\} \in S_T(r)$, we also have $s(r) \leq \bar{s}(T^*)$
  This is absurd, and therefore the hypothesis $T'_L \cup T'_R \neq \emptyset$ is violated.
  Finally,
  \[
  T'_L = \emptyset \\
  T'_R = \emptyset
  \]
  We therefore prove easily the second part of the lemma using $\bar{s}(T^*) \geq \bar{s}(T'_L)$ (resp for $T'_R$ and $T'_R$).

• If $s(r) \leq \bar{s}(T'_L)$
  
  – If $\text{avg}(s(t), \bar{s}(T'_L)) \geq \bar{s}(T'_R)$
    Like previously, we use the simple property of the weighted mean to write:
    \[
    \bar{s}(T^*) = \text{avg}(\bar{s}(T'_L), s(r), \bar{s}(T'_R)) \\
    = \text{avg} \left( \text{avg}(\bar{s}(T'_L), s(r)), \bar{s}(T'_R) \right) \\
    \leq \text{avg}(\bar{s}(T'_L), s(r)) \\
    = \bar{s}(\{r\} \cup T'_L)
    \]
    Using the fact that $\{r\} \cup T'_L \in S_T(r)$, we get:
    \[
    \text{avg}(s(r), \bar{s}(T'_L)) = \bar{s}(\{r\} \cup T'_L) \\
    \leq \bar{s}(T^*)
    \]
    The two previous inequalities give us
    \[
    s(\{r\} \cup T'_L) = \bar{s}(T^*) = \bar{s}(T'_L \cup \{r\} \cup T'_R)
    \]
    Since $T^*$ is the smallest subtree maximizing the strength, the previous inequality is only possible if $T'_R = \emptyset$.
    Moreover, by definition of $T'_L$,
    \[
    \bar{s}(T'_L \cup \{r\}) \geq \bar{s}(T'_L \cup \{r\}) \Rightarrow |T'_L| \geq |T'_L|
    \]
    In this case we then have
    \[
    |T'_L| \geq |T'_L| \\
    T'_R = \emptyset
    \]
  – Else if $\bar{s}(T'_R) > \text{avg}(s(r), \bar{s}(T'_L))$
    Thanks to $\bar{s}(T'_R) \leq \bar{s}(T'_L)$ and $s(r) < \bar{s}(T'_L)$, we can write:
    \[
    T'_L \neq \emptyset \\
    T'_R \neq \emptyset
    \]

\[\text{using that } a > b \Rightarrow \text{avg}(a, b) < a\]
Moreover,

$$avg(\bar{s}(T_R^*), s(r), \bar{s}(T_L')) = \bar{s}(T_R^* \cup \{r\} \cup T_L')$$

$$\leq \bar{s}(T_R^* \cup \{r\} \cup T_L')$$

Since $\bar{s}(T_R^*) \geq \bar{s}(T_R')$ by definition of $T_R^*$, the last inequality is impossible if $|T_R'| < |T_R^*|$. The same can be done for $T_L^*$ and $T_L'$.

We conclude that in this case

$$|T_L'| \geq |T_L^*|$$

$$|T_R'| \geq |T_R^*|$$

QED.
Chapter 3

Results

In this chapter, we are going to present results obtained from the implementation of the robust CART algorithm in R.

- We will start by answering the first question mentioned in the introduction: how can we measure the stability of trees? For this, we will consider synthetic data, generated from 2 given 2-dimensional piecewise constant functions.

- Then, we will study how to apply the robust CART algorithm to real datasets in which there is given uncertainty in the predictors. We will study the difference between volume and minimax prediction in that case, and how it changes the optimal pruning of the tree. Finally, we will study up to which extent the trees built with the robust CART algorithm are stable across several sub-datasets of the total training data.

3.1 Synthetic data

In this section we will study the stability of trees in two cases of synthetic data generation, using the robust CART algorithm. Here is the exact process we use to get a grasp on the extent to which a tree is stable:

- First, we randomly generate 4000 observations in the 2-dimensional unit square, and associate to it a response variable according to a given piecewise constant function (to which we add a small noise \( \epsilon \) where \( \text{Var}(\epsilon) = .25 \).

- We then add virtual uncertainty boxes around the data, and run the robust CART algorithm for different box sizes, looking at the evolution of the tree structure.

- The typical size of boxes that imply a dramatic change in the tree structure can then be interpreted as an indicator of the stability of the tree. If the tree keeps the same structure for a wide range of boxes, we then consider that the initial tree is very stable, and on the contrary if the tree structure changes for boxes of small sizes, we consider the initial tree as being unstable.

3.1.1 An unstable example

Let us consider the first piecewise constant function, described in Figure 3.1. It is defined on the unit square of the feature space \((f_1, f_2)\) and takes 4 different values. We generate 4000 observations as explained in this section’s introduction.

The tree returned by the usual CART algorithm run on random data taken inside the unit square is given in Figure 3.2.

We see that in this case, and if there is no uncertainty in the data, the structure of the stepwise function is very well captured by the tree. However, one could wonder is this tree is stable: indeed, the largest values of function 1 (2 and 1) are for big \( f_2 \), and its smallest values are for small \( f_2 \). Therefore,
Figure 3.1: Sketch of the first 2-dimensional piecewise constant function. The values displayed in each of the four regions of the unit square are the value of the function, and the dotted red line represents roughly the best split according to the predictor $f_2$.

![Figure 3.1: Sketch of the first 2-dimensional piecewise constant function.](image)

Figure 3.2: Final pruned tree returned by the usual CART algorithm (rpart function) for the first piecewise constant function 1

![Figure 3.2: Final pruned tree returned by the usual CART algorithm (rpart function) for the first piecewise constant function 1.](image)

if there is uncertainty in the data, a split with respect to $f_2$ could become more interesting than a split with respect to $f_1$. This split is sketched as the dotted red line in Figure 3.1.

To test this hypothesis, we plot several trees for different values of box sizes around each observations. The results can be seen in Figure 3.3 to Figure 3.7. In this case, the trees are not pruned, but the minimum number of observations allowed by leaf is high (50) so that the tree is not too bushy and that its structure appears clearly.

We see that when the size of the boxes approach .1, the tree built thanks to the robust CART algorithm changes completely: its root split becomes a split on $f_2$ instead of $f_1$, which changes the structure of the whole tree. This is a typical case in which the tree is not very stable to errors in the data, and can therefore only be interpretable to a certain extent: for example in a case where the uncertainty on the predictors is larger than .1, it is not because the root node of the tree contains a split with respect to $f_1$ that $f_1$ is necessarily the most important variable in the model.
Figure 3.3: Final tree built from the robust CART algorithm from data generated from piecewise constant function 1, with the size of the uncertainty boxes in each direction being .001

Figure 3.4: Final tree built from the robust CART algorithm from data generated from piecewise constant function 1, with the size of the uncertainty boxes in each direction being .01
Figure 3.5: Final tree built from the robust CART algorithm from data generated from piecewise constant function 1, with the size of the uncertainty boxes in each direction being .1

![Diagram](image)

Figure 3.6: Final tree built from the robust CART algorithm from data generated from piecewise constant function 1, with the size of the uncertainty boxes in each direction being .15

![Diagram](image)
Figure 3.7: Final tree built from the robust CART algorithm from data generated from piecewise constant function 1, with the size of the uncertainty boxes in each direction being .2
3.1.2 A stable example

In this example, we consider the second piecewise constant function described in Figure 3.8. It is defined on the unit square of the feature space \((f_1, f_2)\) and takes 4 different values. We generate 4000 observations as explained in this section’s introduction.

Figure 3.8: Sketch of the second 2-dimensional piecewise constant function. The values displayed in each of the four regions of the unit square are the value of the function, and the dotted lines represents roughly the best split according to the predictor \(f_2\).

The tree returned by the usual \texttt{rpart} function is given in Figure 3.9. We can see that the structure of the stepwise function is very well captured by the tree, as in the last example.

Figure 3.9: Final pruned tree returned by the usual CART algorithm (\texttt{rpart} function) for setpwise function 2

However, we are not suspecting an unstable tree in that case, as the first split chosen by CART seems to explain an important part of the variance of the response, and has no real competitors (the red dotted line in Figure 3.8 roughly corresponds to the best possible split according to the feature \(f_2\), and is a lot less efficient than the split around \(f_1 = .5\)).

To test this, we proceed like in the previous subsection, and add boxes around each observation in the data, then run the robust CART algorithm. Making the size of the boxes increase, we look at
the evolution in the tree structure, the returned trees (after pruning) are presented in Figure 3.10 to Figure 3.14.

Figure 3.10: Final tree built from the robust CART algorithm from data generated from piecewise constant function 2, with the size of the uncertainty boxes in each direction being .001

Figure 3.11: Final tree built from the robust CART algorithm from data generated from piecewise constant function 2, with the size of the uncertainty boxes in each direction being .01

In that example, the main split is a lot stronger than any surrogate splits, and the tree stays very stable when the boxes around the observations get bigger (and therefore when the uncertainty we have about the data increases): the top nodes of the tree, that have the most importance in terms of interpretation, do not change a lot. We can therefore conclude that this tree is more robust to errors in the data, and that we can therefore be more confident in our interpretations based on it.
Figure 3.12: Final tree built from the robust CART algorithm from data generated from piecewise constant function 2, with the size of the uncertainty boxes in each direction being .1

Figure 3.13: Final tree built from the robust CART algorithm from data generated from piecewise constant function 2, with the size of the uncertainty boxes in each direction being .15
Figure 3.14: Final tree built from the robust CART algorithm from data generated from piecewise constant function 2, with the size of the uncertainty boxes in each direction being 0.2.
3.2 Real data

3.2.1 The ‘Galaxy’ dataset

Data description

The data contain 318 stars with their radiuses in kiloparsec (kpc), and observed velocities of stars in km/s (relative to center, corrected for inclination) from 26 LSB galaxies. It was known that the velocities were measured with errors. In the data set, each velocity includes its estimated standard deviation of measurement errors. Here we shall investigate the nonlinear relation between Velocity (V) and Radius (Rkpc). To adapt this example to our algorithm, we consider that the actual values of the velocities \( (V_i)_{1 \leq i \leq 318} \) all satisfy:

\[
V_i \in \left[ \hat{V}_i \pm 2 \hat{se}(\hat{V}_i) \right] \tag{3.1}
\]

where \( \hat{V}_i \) are the measured values of the velocities, and \( \hat{se}(\hat{V}_i) \) their estimated standard error.

Building the tree and pruning

This example only implies one variable \( (V) \), and therefore is very fast to run. We use it here to illustrate the building step and the pruning step.

First, we grow a full tree, only stopping when the number of observations in leaves goes below 5. The fully grown tree is shown in Figure 3.15. We see that the tree is very complex (bushy) and can probably be pruned back to its more essential structure.

Figure 3.15: Unpruned tree built from the robust CART algorithm on the ‘Galaxy’ dataset

To prune back the tree, we use the validation set approach to find the optimal complexity parameter \( \alpha \) in terms of test error. We know from section 1.3 that there are two main ways to predict from a tree in case of errors in the data: volumic prediction and min-max prediction.

Usually, we use min-max prediction as it is coherent with the pessimistic approach that we used to build trees. However, we try both prediction methods in this example to see how they each affect the choice of the optimal \( \alpha \).

The two plots showing the test error against the complexity parameter \( \alpha \) are displayed in Figure 3.16 and Figure 3.17 (for min-max prediction and volumic prediction respectively). We see that the
global shape of the curve stays the same, but that the minimum, however, is not reached for the exact same parameter.

Figure 3.16: Empirical test error vs log of the complexity parameter $\alpha$, in the case of min-max prediction

![Figure 3.16: Empirical test error vs log of the complexity parameter $\alpha$, in the case of min-max prediction](image)

Figure 3.17: Empirical test error vs log of the complexity parameter $\alpha$, in the case of volumic prediction

![Figure 3.17: Empirical test error vs log of the complexity parameter $\alpha$, in the case of volumic prediction](image)

The two corresponding pruned trees are shown in Figure 3.18 and Figure 3.19. We note that we used the complexity parameter that minimizes the test error to prune the tree here, although it is usually better to use the $1 - SE$ rule mentioned in p.79 of [4], that consists in taking the simplest model which has an error in the one standard error range of the minimum test error. To estimate precisely the standard error, we could try using the bootstrap (this problem is however not tackled in this report).
Figure 3.18: Final pruned tree using the optimal complexity parameter found in Figure 3.16

Figure 3.19: Final pruned tree using the optimal complexity parameter found in Figure 3.17
3.2.2 The ‘SDSSquasar’ dataset

Data description

Most or all large galaxies have a massive black hole (MBH, 106-109 Mo solar masses) at the center. Gas from the interstellar medium or a disrupted star may fall onto the MBH through an accretion disk (“to accrete” means “to fall onto”). This accretion disk can become exceedingly hot and can eject a jet of material at relativistic (near the speed of light) velocities. The disk and jet radiate light across the electromagnetic spectrum (radio, infrared, visible, ultraviolet, X-ray, gamma-ray) with great efficiency. In most galaxies today, like our own Milky Way Galaxy, the MBH is starved of gas and little light is produced. In other galaxies, like Seyfert galaxies or radio galaxies, the light is very strong, particularly in spectral bands other than the visible band where the stars of the host galaxies emit most of their light. In rare cases called quasars, the light from the accreting MBH exceeds the starlight in all spectral bands by enormous factors. These are the brightest objects in the Universe and can be seen even at high redshifts (i.e. great distance from us).

The spectrum, variability and structure of quasars are studied in detail to understanding the complex processes of accreting MBHs and their environs. But an important subfield studies consist of wide-field surveys for quasars and characterize their bulk properties such as brightness in various spectral bands, redshifts, and luminosities. Such survey and photometric (brightness) studies lead to classification of quasar subtypes (e.g. Type I, Type II, radio-loud/radio-quiet, BAL, BL Lacs, Lyman-alpha dropouts), to measurement of the quasar luminosity function (distribution of luminosities), and to cosmic evolution studies (how the population changes with redshift).

The SDSS team has recently produced a catalog of 46,420 quasars from its 3rd Data Release, 95 % of them previously unknown. The columns of the dataset include:

- \( z \): Redshift (scales with distance). This is the response variable that we want to predict using robust trees.
- \( u_{\text{mag}} \): Brightness in the \( u \) (ultraviolet) band in magnitudes. Magnitudes are an inverted logarithmic unit of brightness (a quasar with \( u_{\text{mag}} = 16 \) is 100-times brighter than one with \( u_{\text{mag}} = 21 \)).
- \( \sigma_u \): Measurement error of \( u_{\text{mag}} \). The heteroscedastic measurement errors for each magnitude are determined by the SDSS team from knowledge of the observing conditions, detector background, and other technical considerations.
- \( g_{\text{mag}} \): Brightness in the \( g \) (green) band and its corresponding error \( \sigma_g \)
- \( r_{\text{mag}} \): Brightness in the \( r \) (red) band and its corresponding error \( \sigma_r \)
- \( i_{\text{mag}} \): Brightness in the \( i \) (more red) band and its corresponding error \( \sigma_i \)
- \( z_{\text{mag}} \): Brightness in the \( z \) (even more red) band and its corresponding error \( \sigma_z \)

For each of the predictors \( u_{\text{mag}}, g_{\text{mag}}, r_{\text{mag}}, i_{\text{mag}} \) and \( z_{\text{mag}} \), we are going to make the same assumption as in (3.1), i.e. that the real values of the predictors are always in a 2 standard error range of the measured value. This assumption gives us ‘boxes’ around every observations and can therefore be used as training data for our robust CART algorithm.

Building and pruning

The goal here is to build a tree using the robust CART algorithm to predict the response variable (redshift) thanks to the 5 predictors mentioned previously.

In this subsection, we look at the building and the pruning of the tree.

The full tree is shown in figure 3.20. We have chosen not to plot the text for the nodes since it did not appear clearly on the graph because of the total number of nodes\(^1\). We remember that the plot function of the \texttt{rpart} package that we use plots the size of the branches accordingly to the

\(^1\)We will display the names of the variables according to which we split in the Figure 3.23 below
quality of the considered node. Therefore, we clearly see in Figure 3.20 that the nodes that are very deep in the tree seem to overfit the data a lot (they are not explaining a lot of variance of the response variable), and that on the contrary the top node of the tree is be very good, and therefore an important part of the main structure of the tree.

Figure 3.20: Unpruned tree built from the robust CART algorithm on the ‘SDSSquasar’ dataset

Figure 3.21 and Figure 3.22 display the plots of test error against the log of the complexity parameter $\alpha$ (where the test error was estimated with the validation set approach).

Figure 3.21: Empirical test error vs log of the complexity parameter $\alpha$, in the case of min-max prediction

We remark that both curves have the same shape: they have a very flat plateau for trees with high complexity ($\alpha$ small), and go up dramatically when the top nodes of the tree collapse (and therefore when the tree loses its main structure).

It is important to know that we did not include in the plots the point corresponding to the test error
when the tree was reduced to a root, that caused the graph to become very unclear (the test error was huge for this last point and the shape of the rest of the plot was therefore very unclear).

The fact that overfitting does not appear clearly when estimating the test error is very common in the case of trees built with the usual CART algorithm (mentioned p. 79 of [4]). This issue is generally solved by using the 1-SE rule, that narrows down to taking the simplest model for which the error is in the 1-SE range of the minimum estimated test error. In our case, we have not estimated the standard error of the test error estimator (For the different reasons suggested at the end of section 2.2, the algorithm in R was pretty slow, and it took typically more than an hour to generate trees with 40 thousand observations and a 5-dimensional feature space², so estimating the standard error would have been very time-intensive), and we therefore choose the optimal complexity parameter as the biggest \( \alpha \) for which the test error was less than 5% more than the minimum test error. This gives us (for both curves):

\[
\alpha_{opt} \approx e^3 \approx 20
\]  

(3.2)

Below in Figure 3.23, we show the final pruned tree according to the complexity parameter \( \alpha_{opt} \).

Stability across different datasets with errors

We now wonder if the robust CART algorithm returns sensibly similar trees when applied to different datasets with errors, but that are generated from a common source.

To simulate this, and since our data has more than 40 thousands rows, we split randomly our training data into 5 subsets, and use the robust CART algorithm on each of them. It is reasonable to assume that the data in each set has been generated in a similar fashion.

For everyone of the 5 subsets, we do the same analysis as described in the previous subsection: we first grow a full tree, then plot the test error against the complexity parameter \( \alpha \), and choose the biggest alpha so that the test error stays in the 5% range of its minimum.

In Figure 3.24 to Figure 3.28, we only display the final pruned trees (so that we can compare only the main structure of the tree from subset to subset).

²The pruning took only several seconds thanks to the fact that it was approximated thanks to the algorithm described in 2.4.5
Figure 3.23: Final pruned tree using the optimal complexity parameter $\alpha_{\text{opt}} \approx 20$

Figure 3.24: Final pruned tree built on the subset $1/5$
Figure 3.25: Final pruned tree built on the subset 2/5

Figure 3.26: Final pruned tree built on the subset 3/5
We see here that the 5 small trees built from a random fifth of the data have very similar top nodes. While their lower structure (near their leaves) changes from tree to tree (which can imply that the trees are overfitting the data), their top structure is very stable. This is reassuring because it shows us that the robust CART algorithm seems to build similar trees for different data sets containing errors that are generated from a common source. This is typically something we want from this algorithm: it needs to grasp the very structure of the trees even if there are errors in the data.
Conclusion

In the first chapter, we defined the robust CART algorithm, and showed how it narrowed down to the usual CART algorithm with a different loss function that was still convex and therefore easily minimizable (Theorem 1). Then, we showed that pruning the tree, and other extensions as bagging and random forests could also be generalized to the case where there was errors in the data.

In the second chapter, we explained in detail the implementation of the robust CART algorithm in R, and how pruning could be approached with a fast algorithm, that has a linear complexity in the number of nodes in the tree (and not quadratic like the usual ‘weakest link’ pruning described in [4]).

Finally, in the third chapter, we used the algorithm implemented in chapter 2 to bring elements of response to two main questions that initially motivated this study:

1. How can we quantify the stability, and therefore the interpretability of trees?

2. How can we use trees in areas of science where there is systematic uncertainty in measures (such as Astronomy)?

To answer the first question, we generated synthetic data from two given piecewise constant functions, then as the ‘pseudo-error’ boxes size increased, we looked at the evolution in the tree’s structure. In our example, this approach gave results that matched our expectations in terms of tree stability: the tree that was suspected not to be robust to errors in the data did change when the size of the uncertainty boxes increased, and the tree that was thought to be more stable kept its global structure. Therefore, we can assume that this approach gives a fair estimation of a tree’s robustness to errors in the data.

Then, we answered the second question by applying our robust CART algorithm to real data containing errors. We saw that the algorithm gave reasonably stable results on different subsets of the original data, which tends to show that the algorithm captures well the structure of data, even in the error-in-variables setting that is common in Astronomy.
Bibliography


