TRADITIONAL VERSUS NON-TRADITIONAL BOOSTING ALGORITHMS

A dissertation submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2012

By
Alexandre Michelis
School of Computer Science

Contents

Word Count: 15944

List of Figures

List of Algorithms

Abstract

We study different approaches to gradient boosting from both a theoretical and practical points of view. Gradient boosting constructs an additive model by iteratively fitting a base learner to the current error's gradient to perform a gradient descent. The contribution of each base learner is controlled by a shrinkage parameter that acts as a regularization parameter to confer the model a resistance to overfitting at the cost of computation increase.

We first survey the literature to understand the current state-of-the-art in gradient boosting: gradient boosted trees. We study the components of this algorithm in detail and find a way to efficiently compute trees for gradient boosting. We create two fast gradient boosting implementations using R (www.r-project.org) and C. We compare them with the current state-of-the-art implementation, the GBM package for R, and find that our sequential implementation is 1.7 times faster while our parallel implementation is 3 times faster on the test machine.

We then investigate a new approach that consists in decreasing the shrinkage parameter over the iterations to speed-up the algorithm in the early stages while still resisting overfitting. We try three possible approaches - arithmetic, geometric and exponential decreases - and conclude that while this technique speeds-up the algorithm it also diminishes the model's ability to generalize.

Finally, we give ideas for future improvements of boosting algorithms. We explain how to more efficiently use non-trivial loss functions and show how to construct a boosting algorithm that performs a conjugate gradient descent.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank Gavin Brown, my supervisor, for accepting to oversee my work throughout this project.
I would also like to thank James Weatherall, for the lunches at AstraZeneca, along with Harry Southworth, as our meetings always helped me identify the objectives.

Chapter 1

Introduction

This project is sponsored by AstraZeneca. Its purpose is to investigate different approaches to gradient boosting, a supervised ensemble learning technique, both from a theoretical and empirical point of view, and to create a fast implementation of the algorithm.

The general principle is to produce a *classifier* or *learner*, a function $F(x)$ that predicts an output variable y based on the input variable x. F is constructed, *trained*, using a training set of N known observations (x_i, y_i). When the response variable y_i can only take a finite number of values we call the task a classification problem. When y_i can take an infinite number of values, we say we have a regression problem. In AstraZeneca's context - predicting adverse effects in drug trials for instance - each (x_i, y_i) can be a set x_i of measurements (glucose level, blood cells) from test subject i while the observed result y_i could be, in a classification problem, the presence or absence of a certain disease, or, in a regression problem, a real value to predict.

Boosting consists in constructing F by iteratively training M base learners, also called weak learners, and adding them to produce the final strong learner. A learner is trained by choosing its parameters so that it minimizes a given loss function L. Gradient boosting trains the base learners so that they "descend" the gradient of the loss function one after another and thus minimize it. In 2001, Friedman [Fri01] introduced a shrinkage parameter ν into this method. ν controls the contribution of each base learner to the final predictions, and can be seen as a learning rate. It corresponds to the speed at which the gradient descent is

performed. While a little speed provides the algorithm with a better resistance to overfitting, it increases the required number of iterations which increases the computational cost of the algorithm. An uninvestigated idea is to slowly anneal the shrinkage parameter: by starting with high values and iteratively decreasing it, we are hoping to speed-up the first steps of the algorithm while still resisting overfitting. The current best practice for estimating the required number of iterations is to use a cross-validation procedure to find the iteration number that produces the smallest estimated generalization error.

In 2002, Friedman [Fri02] introduced randomization into the process, calling the method stochastic gradient boosting, to improve its speed and accuracy. To do so, the base learners are trained on different training sets: they are trained on subsets of the entire initial training set created by subsampling without replacement. The current state-of-the-art in gradient boosting consists in applying this technique with regression trees as base learners [ele09], and allows performing feature selection by the computation of relative importance of features [ele09]. However, as building the regression trees becomes the most computationally intensive part of gradient boosting, an efficient method to grow the regression trees is required to create a fast gradient boosting implementation. The reference software that implements this approach is called GBM (Generalized Boosted Regression Models), written in R and C++, which is an open source application widely used by the statistician community, with its own C++ engine to build fast regression trees.

This project aims at creating a gradient boosting implementation faster than the GBM package. It studies regression trees for boosting in detail in order to find a way to efficiently compute them, and to create both a sequential and a parallel gradient boosting implementation that include all major state-of-the-art gradient-boosting components:

- Regression trees,
- A stochastic sub-sampling procedure,
- A loss function suited for regression problems,
- A loss function suited for classification problems,
- A built-in cross-validation procedure,

- The computation of features' relative importance,
- A function to predict outputs.

Then, it also aims at answering the question of the efficiency of the uninvestigated shrinkage annealing idea.

The rest of this report is organised as follows:

- Chapter 2 surveys the original gradient boosting algorithm. It then expands on some common loss functions and details how regression trees are used for boosting, and to what ends. It studies the literature about the propensity of boosting methods to resist overfitting and then describes how cross-validation can be used for gradient boosting. It finishes by describing the state-of-the-art approach to gradient boosting and its implementation for R with the GBM package.
- Chapter 3 studies regression trees in detail and considers different data structures to achieve high-performance stochastic gradient boosting. It gives the sequential and parallel algorithms implemented and compares their performance to the GBM package.
- Chapter 4 investigates the behaviours of stochastic gradient boosting regarding the shrinkage parameter and experiments several approaches to shrinkage annealing.
- Chapter 5 concludes on the findings and deliverables of this project
- Chapter 6 provides ideas for further development of the work.

Chapter 2

Background

This chapter is driven by the current knowledge on gradient boosting that can be found in the literature. It first describes the original algorithm, details some commonly used loss functions and presents a general method of constructing regression trees. It then exposes the different theories the scientific community developed to explain the gradient boosting's behaviour regarding the overfitting phenomenon, and finishes by describing the state-of-the art approach to gradient boosting.

2.1 The Original Gradient Boosting Algorithm

Gradient boosting is an iterative machine learning technique for regression and classification problems. It creates a prediction model based on a ensemble of prediction models, each of which is trained using the errors of the previous one.

The problem has an input variable x and an output variable y of a joint probability distribution $P(x, y)$. The objective is to find a function $F(x)$ to predict y using a training set of N observations (x_i, y_i) of known values of x and the corresponding observed values of y. The problem is called a 'classification problem' when y can only take a finite number of known values also called classes. Each example y_i is to be classified as belonging to one of the possible classes. It is called a 'regression problem' otherwise. $F(x)$ is found by minimizing the value of a specified loss function L over the training set:

$$\mathcal{L}(F) = \sum_{i=1}^{N} L(y_i, F(x_i)) \tag{2.1}$$

$$F = argmin_F \mathcal{L}(F) \tag{2.2}$$

The loss $\mathcal{L}(\mathcal{F})$ defined in equation 2.1 expresses the error made by the learner F according to a loss function L on the entire training set. The objective is to find the function F that minimizes this loss/error. Gradient boosting constructs an approximation F_M of F as a sum of $M + 1$ base learners constructed through M boosting iterations:

$$F_M = \sum_{m=0}^{M} f_m \tag{2.3}$$

The base learners f_m come from a restricted, chosen, class of functions. The most widely used functions [ele09] are regression trees of a fixed size for that they are fast to construct when kept relatively small and they produce accurate boosted classifiers [ele09]. Section 2.3 explains regression trees in more detail, and section 3.3 presents an efficient regression tree algorithm.

The idea of gradient boosting, by Friedman in 1999 [Fri01], is to start with an initial constant guess F_0 and then to iteratively apply a steepest descent to follow the negative gradient in order to minimize the loss. With

$$1 \leq i \leq N, \quad g_{m,i} = \bigtriangledown_{F_{m-1}} L(y_i, F_{m-1}(x_i)) \tag{2.4}$$

we would have

$$F_m = F_{m-1} - \gamma_m g_m \tag{2.5}$$

and

$$\gamma_m = argmin_\gamma \mathcal{L}(F_{m-1} - \gamma g_m) \tag{2.6}$$

Equation 2.4 evaluates of the gradient of the loss of F_{m-1} on the training set. The next classifier F_m is constructed so that it follows the negative gradient (equation

2.5) for an optimal length γ_m, the length that minimizes the loss (equation 2.6).

According to equation 2.3 we would have

$$f_0 = F_0 \tag{2.7}$$

and

$$f_m = -\gamma_m g_m \tag{2.8}$$

for $m > 0$. However, we cannot directly use g_m as it only gives us the values of the gradient on particular points: the training set (see equation 2.4). So that our model can work with unseen examples, we need to generalize this result. While we can't use the gradient as would a steepest descent, as we don't know it, and as each f_m comes from a restricted class of functions (for instance, f_m can be restricted to be a regression tree), we can use the function from our restricted class that best approximates it. This means that we train a base learner h_m to fit the gradient, using the training set $(x_i, g_{m,i})_{i=1}^N$, and use it instead. This leads to the following corrected equations:

$$f_m = -\gamma_m h_m, \quad m > 0 \tag{2.9}$$

$$F_m = F_{m-1} - \gamma_m h_m \tag{2.10}$$

$$\gamma_m = argmin_\gamma \mathcal{L}(F_{m-1} - \gamma h_m) \tag{2.11}$$

It can be noticed that, even for a classification problem, gradient boosting never uses classifiers as base learners. For instance, if we decided to use trees, we would be doing gradient boosting with regression trees and not classification trees even for a classification problem. Consider a binary classification problem. Even if the variable to predict is binary ($y \in \{0, 1\}^N$), the base learners are not fitted on these binary values but on the gradient of the loss; depending on the chosen loss function, they are likely to be fitted on real values (section 2.2 explains this affirmation in more detail). Thus, as the base learners are used to approximate real-valued functions, classifiers are not adapted to gradient boosting and only regression learners are used.

Algorithm 2.1 presents the generic original gradient boosting algorithm according to these equations. Equation 2.11 that needs to be solved at the fifth line of the algorithm is a one-dimensional optimisation problem that is usually resolved by applying a line search strategy [ele09]. The method consists in iteratively increasing a step size γ until a local minimum of $\mathcal{L}$ is reached. The step that reaches this minimum is chosen as the solution γ_m.

Algorithm 2.1 Gradient Boosting

1: $F_0 = argmin_\gamma \mathcal{L}(\gamma)$ ▷ choose the best constant
2: **for** $m = 1 \rightarrow M$ **do**
3: $g_{m,i} = \bigtriangledown_{F_{m-1}} L(y_i, F_{m-1}(x_i))$ ▷ compute gradient at the training points
4: Fit a base learner $h_m(x)$ to the target g_m
5: $\gamma_m = argmin_\gamma \mathcal{L}(F_{m-1} - \gamma h_m)$ ▷ find the length for which the gradient is followed
6: $F_m = F_{m-1} - \gamma_m h_m$ ▷ update the model
7: $m \leftarrow m + 1$
8: **end for**
9: **return** F_M

2.2 Common Loss Functions

This section briefly describes two very popular loss functions and how they are used for gradient boosting: the squared loss, used for regression problems, and the binomial deviance, used for classification problems.

2.2.1 Squared Loss

The squared loss function is defined as follows:

$$L_2(y, f) = \frac{1}{2}(y - f)^2 \tag{2.12}$$

As the difference between the desired output y and the prediction f is squared this function tends to give more importance to predictions distant (in the sense of absolute difference) from the desired outputs. Thus, outliers - observations that are distant from the rest of the data, very often indicative of a measurement error - are given too much importance in a learning process compared to the other data points. The result is that a learner that uses the squared loss function, in

presence of outliers, will fit them too closely, to the detriment of the other - more correct - data points. We say that this loss function is not *robust*. Nevertheless, it remains the most widely used loss function for regression problems due to its mathematical properties. In particular, its relation to the variance, as detailed in section 3.3.2, and its differentiability.
To perform gradient boosting, the latter property is essential as we must fit the base learners on the gradient of the loss. In the case of the squared loss, the gradient is given by

$$\bigtriangledown_f L_2(y, f) = f - y \tag{2.13}$$

which is simply the difference between the desired output and the prediction. For gradient boosting, as the base learners are fitted on this gradient, one could think that the "outliers problem" raised earlier vanishes since the value returned by the gradient varies proportionally - it isn't squared here - with the prediction's distance from the desired output y. This unfortunately isn't the case as the loss, not its gradient, is directly used to find the optimal step length γ for which to follow the gradient (see equation 2.11). Thus, gradient boosting with squared loss isn't immune to outliers.

2.2.2 Binomial Deviance

Binary Classification

To describe a loss function suitable for classification we must expand a little on the classification problem itself. This problem, as stated earlier, restricts the output variable y to a finite number of values also called classes. Suppose we have three classes A, B and C. This problem could be decomposed into three binary classification problems: A versus $\{B$ and $C\}$, B versus $\{A$ and $C\}$, and C versus $\{A$ and $B\}$. If the predictions come with a measure of confidence, we can answer the three classes classification problem by returning the class found by one of the three binary classification problems that come with the highest confidence. This approach is called *one-versus-all*. Another similar approach is to decompose the problem into binary classification problems between every possible pair of classes (A versus B, A versus C, B versus C). Once again, the prediction with the highest confidence is returned to answer the bigger problem. This second approach is called *one-versus-one*. The point here is that even if it would be convenient to

have a classifier that can directly answer a multi-class classification problem, binary classification is sufficient as we can decompose any multi-class classification problem into binary classification problems if we can associate a confidence with each prediction.

The common way of doing so is to set a decision boundary on a real axis and to assign a class to each side of the boundary. Then, the rule is: the further away the classifier's output f is from the boundary, the more confident we are in the prediction. This means that even though we are working on a classification problem, our classifier would be trying to approximate a real valued function: the "predictions' confidence" function. In other words, we are building probabilities estimates. Not only does this give the possibility to solve multi-class problems but it also provides the classifier with more flexibility as predictions can "smoothly" move closer or away from the boundary, and not just being directly moved from one class to another, during the learning process. A popular loss function that enables this is the binomial deviance.

The Binomial Deviance Loss Function

For this loss function the decision boundary is set on 0. Thus, the classification is given by $sign(f) \in \{-1, 1\}$ or, as we usually have $y \in \{0, 1\}$ in a binary classification problem, the classification is given by $\frac{sign(f)+1}{2} \in \{0, 1\}$. This decision boundary allows us to assign a probability with the learner's output f, given by the logistic function:

$$P(y = 1|x) = P(f) = \frac{1}{1 + e^{-f(x)}} \tag{2.14}$$

The following properties justify its use as a probability function:

- The function acts symmetrically on the two classes:
  $P(y = 0|x) = 1 - P(y = 1|x) = 1 - \frac{1}{1+e^{-f(x)}} = \frac{e^{-f}}{1+e^{-f(x)}} = \frac{1}{1+e^{f(x)}} = P(-f)$
- It takes values between 0 and 1,
- The higher f is, the more likely the example is to belong to class 1:
  $f_1 > f_2 => P(f_1) > P(f_2)$
- It evaluates at 0.5 on the decision boundary ($f = 0$) (on the decision boundary an example is as likely to belong to any of the two classes).

As increasing f increases the probability of class 1, and decreasing f increases the probability of class 0, to perform a gradient descent we need a loss function L_D for which the negative gradient $-\bigtriangledown_f L_D$ respects the following:

- True positive: $f > 0$ and $y = 1 => -\bigtriangledown_f L_D > 0$
- True negative: $f < 0$ and $y = 0 => -\bigtriangledown_f L_D < 0$
- False positive: $f > 0$ and $y = 0 => -\bigtriangledown_f L_D < 0$
- False negative: $f < 0$ and $y = 1 => -\bigtriangledown_f L_D > 0$

The following function meets the criteria

$$\bigtriangledown_f L_D = P(y = 1|x) - y = \frac{1}{1 + e^{-f(x)}} - y \tag{2.15}$$

and is the gradient of the binomial deviance loss:

$$L_D = log\left(1 + e^f\right) - yf \tag{2.16}$$

We can notice that correctly classified examples are also penalized with the binomial deviance. However, [ele09] provides a comparative study of several loss functions for classification problems and shows that the binomial deviance is robust because:

- It penalizes a misclassification more heavily than a correct classification which makes it possible to reduce the misclassification rate during a learning process,
- Its penalty increases linearly with f which makes it more robust to outliers than other loss functions for which the penalty increases at a higher rate.

These reasons very often make it the preferred loss function for classification problems. It can be noted that a multi-class version of the deviance loss function exists, which enables constructing classifiers that can work with more than two classes, but this won't be covered here.

2.3 Regression Trees

A regression tree creates a model in which the feature space - the multidimensional space the examples x_i take their values in - is partitioned into rectangles (resp.

cubes in 3 dimensions, or hypercubes if more than 3 dimensions), and assigns a constant value to each of these regions. When a new example $x_i \in (X_1, X_2, ..., X_p)$ arrives, it falls into one of the regions and the output that is predicted is the constant assigned to this particular region.

2.3.1 Recursive Binary Partitioning

Constructing a regression tree is finding such a partition and the corresponding values to fit the data. A very popular method [ele09] is the recursive binary partitioning, which consists in recursively splitting the space in two according to its dimensions. This means that an algorithm chooses a feature X_{p_1} and a split value S_1 which will partition the space into two regions:

- A "left" region R_{1_1} defined by $X_{p_1} \leq S_1$ to which a constant C_{1_1} is assigned,
- A "right" region R_{1_2} defined by $X_{p_1} > S_1$ to which a constant C_{1_2} is assigned.

Then, this process is repeated recursively on the left (resp. right) region R_{1_1} (resp. R_{1_2}): a feature X_{p_2} (resp. Xp_3) is chosen, possibly the same as before i.e. $p_1 = p_2$ (resp. $p_1 = p_3$), to partition the left (resp. right) region in two according to a split value S_2 (resp. S_3). Two constants C_{2_1} and C_{2_2} (resp. C_{3_1} and C_{3_2}) are assigned to the newly created regions.
If we stopped the process here, then the four *terminal* regions (R_{2_1}, R_{2_2}, R_{3_1}, R_{3_2}) would form a partition of the feature space. This means that any example x_i would fall into one of these regions. Thus, one and only one of their respectively assigned constants (C_{2_1}, C_{2_2}, C_{3_1}, C_{3_2}) would be used to predict the corresponding response y_i. The constants assigned to the non-terminal regions are never used. There are algorithms that first partition the space then assign constants to the terminal nodes only, and others that find the constants while partitioning the space, without knowing if the node they're working on is terminal or not, which results in assigning constants to all the nodes of the tree. Figure 2.1 is a representation of such a partitioning tree. We can notice that the left regions are defined as being strictly inferior than the split value on the corresponding dimension, and not inferior or equal as stated earlier. Even though this choice can lead to fundamentally different predictions, it is only a matter of convention and none of the two possibilities are supported by more theory than the other.

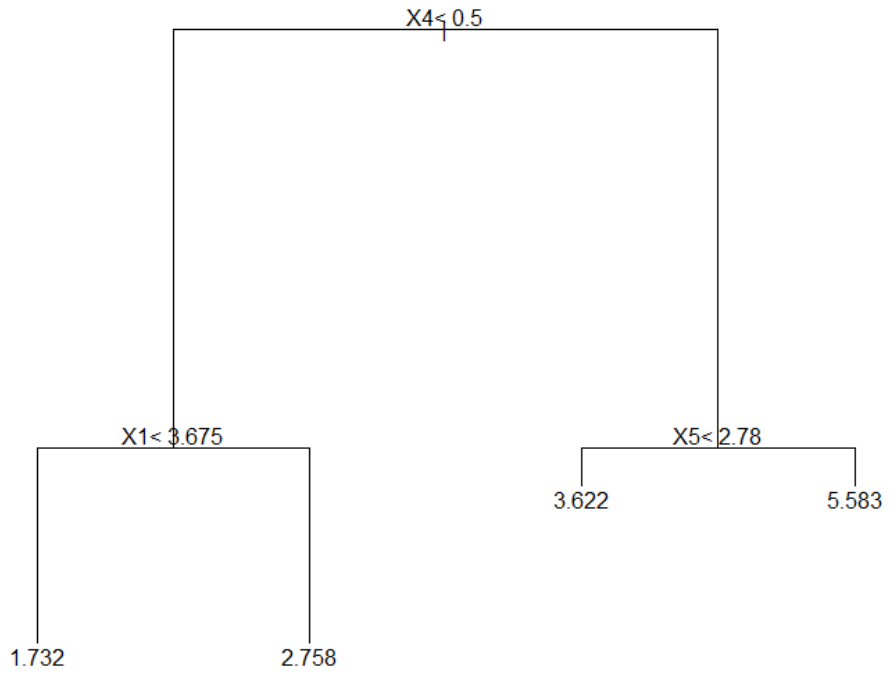

Figure 2.1: A binary partitioning tree with 4 terminal regions. Chosen features: $X_{p_1} = X_4$, $X_{p_2} = X_1$ and $X_{p_3} = X_5$. Split values: $S_1 = 0.5$, $S_2 = 3.675$ and $S_3 = 2.78$. Constants assigned to the terminal nodes: $C_{2_1} = 1.732$, $C_{2_2} = 2.758$, $C_{3_1} = 3.622$ and $C_{3_2} = 5.583$.

The goal when constructing a regression tree is to choose the best partition of the space and assign the best possible constants to each regions. The notion of best obviously depends on the loss function that is used. Formally, the binary partitioning algorithm is looking, at each node, for the feature p, the split value s, and the two constants c_1 and c_2 that solve the following:

$$min_{p,s}\left[min_{c_1}\left(\sum_{x_i|x_{ip}\leq s} L(y_i, c_1)\right) + min_{c_2}\left(\sum_{x_i|x_{ip}>s} L(y_i, c_2)\right)\right] \tag{2.17}$$

When recursively applying the algorithm we get a tree that partitions the space in K regions R_k, each assigned to a constant c_k. Thus, we can formally express a tree with

$$T(x) = \sum_{k=1}^{K} c_k I(x \in R_k) \tag{2.18}$$

and divide the problem of growing the tree into two sub-problems:

- Finding the constant c_k given the region R_k (the inner minimization problem);
- Finding the regions $R_k, k \in [1; K]$.

Finding the optimal constants given the regions depends on the loss function that is used but usually is an easy task. While finding the regions usually is a difficult, computationally very intensive task, the problem simplifies when using an adequate loss function [ele09]. In particular, the squared loss function allows for a fast computation of the optimal regions. Chapter **??** details that point and gives an efficient algorithm for it. The idea is then to find the R_k as an approximation of the truly optimal regions for the loss function L in use by using the squared loss instead, and then to find the optimal constants c_i regarding L and R_k. In practice, this means that the tree is grown using the squared loss, so at each node the algorithm looks for p and s that satisfy

$$min_{p,s}\left[min_{c'_1}\left(\sum_{x_i|x_{ip}\leq s} (y_i - c'_1)^2\right) + min_{c'_2}\left(\sum_{x_i|x_{ip}>s} (y_i - c'_2)^2\right)\right] \tag{2.19}$$

and looks for the corresponding constants c_1 and c_2 that minimize the actual loss given the split s and the feature p

$$min_{c_1,c_2}\left[\sum_{x_i|x_{ip}\leq s} L(y_i, c_1) + \sum_{x_i|x_{ip}>s} L(y_i, c_2)\right] \tag{2.20}$$

2.3.2 Gradient Boosted Trees

When boosting trees, using equation 2.11 to define the m^{th} step size γ_m isn't optimal anymore and can be improved. That is, instead of performing a single search to follow the negative gradient's approximation for the optimal length, it becomes possible to perform a separate search for each terminal region of the tree to find the locally optimal descent direction lengths for those regions. Instead of searching the optimal constants c_k to fit the gradient of the loss $\bigtriangledown_{F_{m-1}}\mathcal{L}(F_{m-1})$ and then multiplying them by γ_m, we can thus define

$$\gamma_{mk} = argmin_\gamma \sum_{x_i\in R_k} L(y_i, F_{m-1}(x_i) + \gamma) \tag{2.21}$$

and updating the model becomes

$$F_m(x) = F_{m-1}(x) - \sum_{k=1}^{K} \gamma_{mk} I(x \in R_k) \tag{2.22}$$

Equation 2.21 means that we are now looking for the γ_{mk}, the best local constants to approximate and follow the gradient, while growing the tree, instead of the c_{mk}. Therefore, equation 2.20 becomes, at each (terminal) node, for boosted trees:

$$min_{\gamma_{mk1},\gamma_{mk2}}\left[\sum_{x_i|x_{ip}\leq s} L\left(y_i, F_{m-1}(x) - \gamma_{mk1}\right) + \sum_{x_i|x_{ip}>s} L\left(y_i, F_{m-1}(x) - \gamma_{mk2}\right)\right] \tag{2.23}$$

2.3.3 Gradient Boosted Trees for Feature Selection

It can often be interesting to understand where the model's prediction come from. That is, to understand how the the model makes decisions, in a manner

of speaking. Concretely, when building a model with hundreds of features what usually happens is that some of them are more important than the others while some features might not even be correlated at all to the predictions. It can be advantageous to know which features help explain the distribution of the data. For instance, AstraZeneca uses such a process to find adverse effects in drug trials. Take a dataset made of hundreds of features. Each example x_i can be a patient, and each feature X_p can be a measure of a certain observation for the patient; for instance, blood cells level, glucose, or the presence or absence of headache, stomach ache and so on. Now consider that the response y_i is binary: drug or placebo. Drug would mean that the patient was given the drug that is being tested, and placebo that he or she was given a placebo instead. Now, if you build a prediction model on this dataset, and you find that the presence or absence of headache is very important for your model's decision making process, then you can assume that the drug can have an impact on the presence or absence of headache. In that case you may conclude that headaches are a possible adverse effect of the drug. This conclusion may seem faulted as, intuitively, when we are building the prediction's model we are saying that the features explain the response. However, we are saying that the response explains the feature to reach the adverse effect conclusion, which seems to go the other way around. However, when building the model we are only saying that there is a relation between the features and the response; there actually is no assumption made on the direction of this relation.

Relative Importance of Features

Visualizing what features are important for a decision or regression tree is very easy, as it is possible to represent the tree with a two-dimensional graphic (see figure 2.1 for instance). However, it would be more useful to have a measure of each feature's importance in the model's predictions. For a regression or classification tree T, Breiman et al. [BFOS84] suggested the following relative importance measure in 1984:

$$\mathcal{I}_p^2(T) = \sum_{split\,on\,X_p} I_k^2 \tag{2.24}$$

which gives the relative importance of the feature p in the tree T, also called relative influence. Here, I_k^2 is the empirical improvement obtained by splitting

on the variable p at the node k. Its value is given by the diminution of error obtained with the split, squared. Then, the relative importance of feature p is the sum of all the squared improvements obtained where feature p was chosen as the splitting feature in the tree.

In 2001, Friedman [Fri01] extended this measure to boosted trees by averaging this measure over all the trees that make the boosted model:

$$\mathcal{I}_p^2 = \frac{1}{M} \sum_{m=1}^{M} I_p^2(T_m) \tag{2.25}$$

Finally, the usual approach is to scale the measures given by equations 2.24 and 2.25 by assigning the value 100 to the largest measure and scaling the others accordingly [ele09].

2.4 Gradient Boosting and Overfitting

Numerous examples, as addressed by Buja in 2000 [Buj00], raise the - still unanswered - question of the "mysterious immunity to overfitting" of boosting algorithms. The majority of the literature addresses this question not for gradient boosting algorithms specifically but for the larger boosting algorithms family. Various arguments are given to answer it, and the scientific community does not in fact agree that boosting resists overfitting. However, the popular belief is that the original gradient boosting algorithm overfits.

The given explanation is that at each gradient boosting iteration the model is updated with a base learner that is chosen so that it descends the gradient of the loss, which means that each iteration reduces the training error. Thus, Hastie, Tibshirani and Friedman [ele09, Fri01] state that the training error can be made arbitrarily small with a number M of iterations large enough. As minimizing the training error can lead to fitting the data too well, which can increase the error of future unseen predictions, the original gradient boosting algorithm is said to overfit. As a consequence, Friedman [Fri01] states that there exists an optimal number of iterations M^* that minimizes this risk of overfitting. Zhang and Yu (2005, [ZY05]) developed this idea and suggested limiting the number of boosting iterations to prevent overfitting, calling this method "early stopping". However,

Mease and Wyner (2007, [MW07]) shows an example where early stopping causes overfitting, stating that it is the last iterations, the ones that are discarded by early stopping, that can prevent overfitting.

In 2001, Friedman [Fri01] suggested a regularization method by shrinkage to prevent overfitting, which consists in adding a learning rate parameter ν to the algorithm's update rule (equation 2.10):

$$F_m = F_{m-1} - \nu\gamma_m h_m, \quad 0 < \nu \leq 1 \tag{2.26}$$

He empirically found that low values of ν, typically 0.1, drastically limit the overfitting phenomenon, more shrinkage resulting in higher training error. However, Mease and Wyner (2007, [MW07]) shows an example where this shrinkage parameter causes overfitting. The opinion presented is that the overfitting resistance of boosting algorithms comes mainly from 1) the self-averaging property of additive methods such as boosting (see equation 2.3), and 2) the stagewise nature of the algorithm.

The first idea was introduced by Breiman in 2000 [Bre00]. It is based on the bias and variance error modelling introduced by Geman, Bienestock and Doursat in 1992 [GBD92] for regression problems which has been widely used since: it was extended to the 0/1 loss function by Friedman in 1997 [Fri97] and generalized to the classification problem first by Domingos in 2000 [Dom00] and by James in 2003 [Jam03], and was then extended to ensemble methods with the bias-variance-covariance decomposition [UN]. The idea is that, for a regression problem, when using the squared error loss function, the error of the classifier can be decomposed as a sum of bias and variance:

$$E_T[(F - F_M)^2] = bias_F(F_M)^2 + var_F(F_M) \tag{2.27}$$

with

$$bias_F(F_M)^2 = (E_T[F_M] - F)^2 \tag{2.28}$$

and

$$var_F(F_M) = E_T[(F_M - E_T[F_M])^2] \tag{2.29}$$

E_T being the expected value over all possible training sets T. With this model, overfitting corresponds to a learner with low bias and high variance: the learner fits well the training data (low bias) while a change in the training dataset leads to a high change in the learner's output (high variance). Thus, Breiman's explanation [Bre00] of the resistance to overfitting of boosting is that a self-averaging process occurs with additive methods: the discrepancies among the base learners tend to cancel one another, bringing the algorithm's predictions closer to the "overall shape" of the function to estimate rather than giving much importance to the "local variations" of the training set. Thus, such a process tends to reduce the variance of the classifier and so to resist overfitting.

The second idea was expressed by Buja in 2000 [Buj00] who attributed the relative immunity to overfitting of the boosting methods to the termwise fitting of the h_m components that make the final boosted function F_M. The usual machine learning approach to building a classifier is to jointly fit all the parameters, stating that changing one of the parameters has an impact on the others. However, the stagewise nature of a boosting algorithm involves fitting the h_m one after another, and "the suboptimality of [this] fitting method" confers it a relative resistance to overfitting. Mease and Wyner (2007, [MW07]) states that introducing a shrinkage parameter destroys the resistance introduced by the stagewise nature of the algorithm.

2.5 Cross-Validation

Cross-validation is a technique that divides the training data in K parts, called folds, to train K identical models on different datasets. Each model is trained using $K-1$ folds of the data while the remaining left out fold is used for validation purposes: the model's predictions error is measured on this validation fold to assess its ability to cope with new, unseen, examples. The left out fold used for validation changes for each of the models, and so do the training folds.

When working on a classification problem it is important to consider the ratios of classes when creating the folds. For instance, for a binary classification

problem which contains $x\%$ of positive examples (examples from the "positive" class) and $(100-x)\%$ of negative examples, it is important that all the folds respect these proportions: each fold must be composed of $x\%$ of positive examples and $(100-x)\%$ of negative ones. This technique is called *stratified sampling* and gives much better results than a classic random sampling approach; randomly distributing the examples among the folds can create uneven folds that don't reflect the true distribution of the data and, in extreme cases, some folds may even only contain examples from one of the two classes.

For gradient boosting, cross-validation serves two purposes: it allows (1) estimating the model's generalization error, and (2) estimating the model's optimal number of iterations M^*.

(1) Estimating the generalization error

The model's generalization error is the expected error that can be measured on unseen examples. When measuring the error on a validation fold we measure a generalization error on one unseen dataset. Thus, by averaging the measured errors on the K validation folds we get an approximation of the expected generalization error which allows us to assess our model. More formally, if we define F_M^{-k}, the model trained with the k^{th} fold left out, and X^k the k^{th} fold of the data, we can express the cross-validation estimate of the generalization error with:

$$Err_{CV}(F_M) = \frac{1}{N}\left(\sum_{x_i \in X^k} L\left(y_i, F_M^{-k}(x_i)\right)\right) \tag{2.30}$$

This estimate can then be used to compare the accuracies of different predictive models to select the best one.

(2) Estimating the optimal number of iterations

As stated in section 2.4, there exists an optimal number of iterations M^* [Fri01] to perform when training a gradient boosted model. By considering the cross-validation's estimate of the generalization error as a function of the number of boosted iterations, it becomes easy to find M^*. Figure 2.2 shows the evolution of the estimated generalization error according to the number of boosting iterations. We can see the generalization error decrease to reach a minimum before increasing

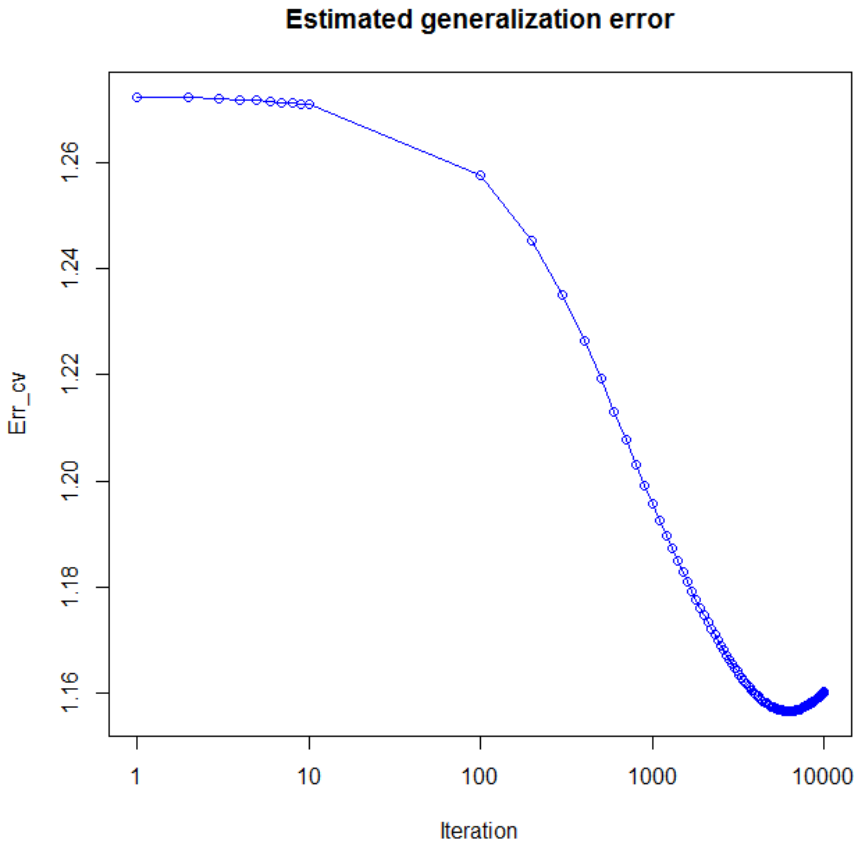

Figure 2.2: Cross-validation estimate of the generalization error as a function of the number of iterations

again - when the model begins to overfit, which is the behaviour usually observed, as stated by Zhang and Yu [ZY05]. Thus, applying their early stopping method becomes easy as one only needs to choose M^* as being the iteration number for which the estimate of the generalization error is the lowest.

2.6 State-of-the-art

The current best approach to gradient boosting consists in using regression trees of fixed and relatively small size, typically from two terminal nodes - decision stumps - up to half a dozen or so [ele09]. This allows for fast to train yet very accurate gradient boosted models.

Empirically, it has been found [Fri01] that using a small shrinkage parameter - learning rate - ν, usually 0.1, along with an appropriately large number of boosting iterations M give the best test accuracies. As a smaller learning rate requires more iterations to reach an equivalent training accuracy, choosing a small value increases the computational time of the algorithm. However, the highest test accuracies are reached with the small learning rates. So, improving the test accuracy of the boosting procedure, reducing the overfitting phenomenom, can be done at the expense of computational time by using a small learning rate.

Another interesting approach, by Friedman in 2002 [Fri02], showed that both the accuracy of gradient boosting and its execution speed could be substantially improved by introducing randomization in the procedure. Towards this goal, he applied subsampling without replacement on the training data at each iteration of the algorithm, naming the method "stochastic gradient boosting". Each base learner is trained on a randomly selected subset of the training data, different for every base learner. This method tends to average the base learners estimates over the possible training sets and to reduce the correlation between them. According to equation 2.29 and to the bias-variance-covariance error decomposition [UN], this reduces the variance of the final combined classifier. This approach has been experimented with gradient boosted trees [GCL] in 2011 under the name "Bagging Gradient-Boosted Trees", even though the technique used was not bagging (subsampling with replacement) but the original technique described in [Fri02] (subsampling without replacement). With subsamples of 67% the size of the total training set the results obtained showed a variance reduced by 50%. Thus, this method improves the accuracy of the classifier by reducing the variance and improves its execution speed as only a subset of the training samples are used for each iteration. Algorithm 2.2 describes the state-of-the-art approach to gradient boosting: gradient boosted trees.

GBM

GBM (Generalized Boosted Regression Models) is a software written in R (www.r-project.org) and C++. R is a high level scripting language that provides the user with an interpreter/command line interface (CLI) and many statistics-related functionalities. In such, it is user friendly but slow for computationally intensive tasks. However, it can be interfaced with C or C++ codes for efficiency. When

Algorithm 2.2 Gradient Boosted Trees

1: $F_0 = argmin_\gamma \mathcal{L}(\gamma)$ ▷ initialize the predictions to the best constant
2: **for** $m = 1 \rightarrow M$ **do**
3: Randomly select an appropriately sized subsample X' from the dataset
4: $g_{m,i} = - \bigtriangledown_{F_{m-1}} L(y_i, F_{m-1}(x_i))$ ▷ compute negative gradient at the training points
5: Fit a regression tree $h_m(x)$ to the target g_m using X'
6: Compute the best terminal nodes predictions using X'
7: $F_m = F_{m-1} + h_m$ ▷ Update the model
8: $m \leftarrow m + 1$
9: **end for**
10: **return** F_M

that is the case, R acts as a user-friendly wrapper that calls more optimized C or C++ functions and returns their result to the user in the R command line interface. Several extensions are available for R, called packages. GBM is one of those.

The GBM package implements gradient boosted trees as described in algorithm 2.2 and originally in Hastie, Tibshirani and Friedman [ele09]. It implements random subsampling, comes with a built-in stratified cross-validation procedure, a relative importance of features computation functionality, and implements many loss functions. It is the current reference software for gradient boosting.

Chapter 3

A Fast Gradient Boosting Implementation

One of the objectives of this project was to create a faster gradient boosting implementation than the GBM package. This chapter describes the choices that have been made and algorithms that have been implemented towards this goal.

3.1 Optimizing GBM or creating a new package?

In order to create a faster gradient boosting implementation than the GBM package, a choice had to be made:

- Either modify the GBM package,
- Or create a new package "from scratch".

Obviously the first option seems to be the one to take; reinventing the wheel rarely is the approach to undertake. When possible, one should always rely on the work that was previously done. However, after studying the GBM package source code it became less clear that this was the way to go. Here is presented a careful consideration of both possibilities with their pros and cons.

3.1.1 Optimizing the GBM package

GBM is a complex software that consists in an R wrapper around C++ classes. Several loss functions are implemented, each of which being a new implementation

of a C++ class containing several algorithms (mostly: computing the residuals, assigning constants to the terminal nodes of the regression trees, updating the predictions). This means that in order to optimize the GBM package, one could:

- Try to optimize each loss function's implementation of algorithms,
- Try to change the way the loss function's class is used in the gradient boosting algorithm,
- Try to optimize its regression trees.

As many loss functions are implemented in GBM, the first option would require modifying a lot of code while the second might mean completely changing how the loss function's class works so on top of that a lot of restructuring may be necessary. Besides, modifying GBM's regression trees would impact on all the loss functions. Bottom line, improving the GBM package source code these ways would require a substantial amount of work, let alone the time required to find ways to optimize it. Besides, even if one "evident" way to optimize the code is to parallelize what can be parallelized - GBM has been implemented sequentially so any parallelization could potentially greatly improve the performance - the amount of code to go through remains the same.

Parallelization can be achieved for the C or C++ portions of an R package with the use of OpenMP. OpenMP is a programming standard that allows achieving complex threads uses by writing relatively simple - and little - compiler directives, instead of long, complex and error prone, more classic, lines of code. However, the R documentation states that the performance of its implementation with R "varies substantially between platforms" and "both the Mac OS X and Windows implementations have substantial overheads and are only beneficial if quite substantial tasks are run in parallel". Therefore, parallelization may not improve performance as much as we could expect.

3.1.2 Writing a new package

Creating a new package clearly requires a lot of work as all the algorithms would need to be written:

- Subsampling

- Cross-validation
- Regression trees
- For each loss function: loss, gradient and an algorithm to compute the best constants to assign the tree nodes
- Gradient boosting
- Relative Importance of Features

While regression trees clearly demand a substantial amount of work, more than the other algorithms, both the advantage and high difficulty of this approach mainly consists in the freedom that comes with it: coming up with a well-thought-out algorithm and a high quality code can drastically improve the performance but isn't an easy task to accomplish. However, the main disadvantage is that some of the "little" features that are implemented in GBM and that are useful in some occasions - for instance, the possibility to use a cost matrix, or some unusual loss functions - are likely to not be implemented, in the absence of sufficient time. Finally, there is here as well the possibility to parallelize some algorithms to make use of multi-core systems.

Bottom line, this approach gives more room for optimizations at the cost of some functionalities of lesser importance. It is the approach that has been chosen.

3.2 Between R and C

One little but important aspect to take into account is the way R and C work together[1]. Both R and C parts of the code can create objects. However, R objects are different than C objects - "classic" variables. It is possible in C to create both C variables and R objects but R manages its own memory automatically through a garbage collector[2] which makes it delicate to correctly manipulate R objects created in C. Unless some precautions are taken, an R object created in C might be destroyed by the garbage collector while still being in use, or, on the contrary,

[1]The implementation has been written in R and C so we only talk about R and C here in order to lighten the text, but anything that applies to C code actually also applies to C++ code.

[2]A garbage collector is an automatic memory management system that attempts every once and a while to free the memory occupied by objects that are no longer in use.

an incorrect manipulation might indefinitely prevent the garbage collector from freeing the memory that was used by an old object, leading to memory leaks. However, we only aim at using R as a wrapper around C: some R code is used to get the inputs - dataset and other parameters - from the user to the underlying engine written in C, which must return R objects to the user. Thus, an easy and efficient way to avoid dealing with memory management problems is to let R create all the R objects that are needed, both the inputs and the outputs, and then to call the C engine with all those R objects. The easiness comes from the C engine being able to create its own variables regardless of the garbage collector state, and to directly modify the R objects that are given to him to store the results. The efficiency comes from the way R objects are transmitted to C: they aren't actually transmitted to C but only their memory locations; R objects aren't copied in memory for the C code to use them.

3.3 Fast Regression Trees for Boosting

Growing the regression trees is the most computationally intensive part of the gradient boosting algorithm. The recursive partitioning algorithm, as described in section 2.3, must:

- For each node:
  - For each feature:
    - For all the possible splits according to the current subsample:
      - Compute the cost
      - Possibly - if this step isn't computed later on by another algorithm - compute the left and right best constants

Computing the cost of the split and the best constants both require at least to go through all the elements of the subsample. Besides, trying a split means separating the elements in two: the elements inferior and superior than the split. Thus, as we need to try all the possible splits, the elements must be sorted at least once according to each feature. As a tree of depth D can have up to $2^{(D-1)}$ nodes, with N examples and P features the complexity of such a recursive partitioning algorithm cannot be better than $O\left(PN\left(log(N) + 2^{(D-1)}\right)\right)$: $O(PNlog(N))$ to sort the elements according to each feature, and $O(PN2^{(D-1)})$ to scan all the

elements for each feature, for each node.

Algorithm 3.1 gives a high-level pseudo-code for such an algorithm. We can notice that:

- The elements are scanned in order according to each feature. That way, when trying a split on a feature, we know that the elements already scanned are inferior than the current split so it becomes easier to find the left and right subsamples needed to compute the cost and the best constants. Besides, the relative order of the elements according to each feature remains the same independently of (1) the subsample and (2) the tree:

  1. Here is a toy example to illustrate that idea:
     Let the dataset $X = \{x_1, x_2, x_3\}$
     If X is ordered according to feature p: $x_{1,p} \leq x_{2,p} \leq x_{3,p}$ then any subsample of X built in order is also ordered according to feature p: $x_{1,p} \leq x_{2,p}$, $x_{1,p} \leq x_{3,p}$, $x_{2,p} \leq x_{3,p}$.
     This very intuitive yet powerful property indicates us that if our subsampling process respects the order - we are here referring to the process of dividing the elements in two subsamples, the elements inferior and superior than the split value -, we only have to sort the elements once per tree, per feature.

  2. As the values of the features never change throughout the gradient boosting iterations (only the residuals do) even if trees from two different gradient boosting iterations are likely to work on different subsamples, the elements keep the same order for all the trees. If we can build our subsamples in order - we are here referring to the process of randomly subsampling from one boosting iteration to another - then (1) showed that we can compute the orders of the elements according to the features once and for all the trees.
     Thus, assuming we have such a subsampling process, we only need to sort the elements once per feature and for all the gradient boosting iterations. Therefore, sorting the elements should be done prior to growing the regression trees, which means that the optimal complexity for growing a regression tree becomes $O\left(2^{(D-1)}PN\right)$.

- As the elements are scanned in order (line 3 of the algorithm), we cannot

split right after scanning an element. Indeed, consider the dataset $X = \{x_1, x_2, x_3\}$ in that order according to feature p. It may be possible to split between x_1 and x_2 (after having scanned x_2) and between x_2 and x_3 (after having scanned x_3). Assume $x_{2,p} = x_{3,p}$. Then you can't split right after having scanned x_2 as it would mean placing x_2 in the left subsample and x_3 in the right subsample whereas they should both be placed in the same subsample as they have the same value for this feature. Thus, we should always ensure that we scanned all the elements of a same value before computing the cost and best constants for a split.

Algorithm 3.1 Recursive Partitioning: high-level pseudo-code

```
1: function RECURSIVEPARTITIONING(subSample, depth, loss, result)
2:     for all features p do
3:         for all x ∈ subSample ordered according to p do
4:             if This is a possible split point then
5:                 Compute the cost of the split
6:                 Compute the left and right best constants regarding the loss
7:             end if
8:         end for
9:     end for
10:    Place in result the split with the lowest cost and its constants
11:    if depth > 1 then
12:        Find left and right subSamples
13:        RecursivePartitioning(leftSubSample, depth-1, loss, result → left)
14:        RecursivePartitioning(rightSubSample, depth-1, loss, result → right)
15:    end if
16: end function
```

3.3.1 How to efficiently work with subsamples?

Not only is it important to efficiently handle subsamples when growing the trees, but it is also important throughout the entire gradient boosting algorithm: from cross-validation to actual random subsampling without replacement, a gradient boosting implementation is continuously working on subsamples of the entire dataset.

Several data structures could be used to allow subsampling from a dataset and maintaining the orders according to the features, from which:

1. Maintaining an ordered array of selected indexes per feature,
2. Maintaining a binary search tree per feature - possibly well balanced like a red-black or avl tree - that only contains the selected indexes, indexed by (feature value, index),
3. Maintaining an binary array b for which $b[i] = 1$ if x_i is selected, 0 otherwise, as well as an ordered array of indexes per feature.

The approach we need to choose must be able to quickly scan the elements from the subsample in the orders of the features. It must also be possible to easily subsample, in order, from it.

1) would store the selected indexes contiguously which would thus be accessed very quickly with maximum cache hits[3]: operation of worst case complexity $O(N)$ so $O(PN)$ when considering all the features. Subsampling in order would be easily done for the array corresponding to the feature chosen by the recursive partitioning algorithm as it would already be ordered - we would only need to divide it in two at the index of the split. However, the arrays corresponding to the other features wouldn't be related to this particular order. So, subsampling from them would require checking for each element of each array which new subsample it falls into. The best way to do that would be to fill a binary array with the example's new subsample number, so that it can be checked in a constant time. Thus, subsampling while respecting the order, for the recursive partitioning procedure, would be an operation of complexity $O(PN)$.

2) would always respect the subsampling order as a binary search tree is always ordered. So, subsampling would be an operation of complexity $O(PNlog(N))$ (for each of the P features up to N elements must be removed, which can be done in logarithmic time). We must notice that N is here the size of the current subsample and not necessarily the original dataset. Thus, subsampling would be

[3]CPU cache systems work differently for each processor. The main idea remain the same though: when accessing an element from an array in the main memory, the processor will take this element and the following n ones (n depends on the cache line size of the processor) and place them in its cache memory. This memory, orders of magnitude faster than the main memory, will then be used instead of the main memory to access the next elements, should they be accessed. Thus, the speed of accessing elements in the order they are stored in an array is greatly increased. We say we have a cache hit when an element is read from the cache, and a cache miss when it is accessed from the main memory.

in the worst case as slow as sorting the entire dataset, but in practice it should become faster the smaller the subsample becomes, i.e. the further down the tree we get. That remark also applies to the first method. However, gradient boosting is usually used with small sized trees so subsampling and looking for an element, even in practice, wouldn't get much faster. Besides, such a data structure scatters its elements in memory so there would be very little benefit from the processor's cache. Finally, knowing whether example i is part of the current subsample can be done in $O(log(N))$ time. Thus, accessing all the elements of the subsample in order would require $O(PNlog(N))$ time.

3) would access the elements in order by scanning through the ordered arrays of indexes and checking in the binary index - which can be done in constant time - if the element is part of the subsample. It would benefit from the processor's cache but still be slower than 1) as it must perform the checking operation for all the elements of the original dataset. Thus, the complexity would always be $O(N)$ ($O(PN)$ when considering all the features), and not just in the worst case. However, subsampling in order would be an $O(N)$ operation as it would only require going through the binary array and put the values to 0 or 1 accordingly.

As 3) is the fastest solution for subsampling, and close to be the fastest for scanning the elements in order, it the approach that has been implemented.

Random subsampling without replacement

We have mostly detailed subsampling to face the recursive partitioning problem. However, gradient boosting must also create random subsamples of the dataset at each boosting iteration. Algorithm 3.2 describes the random subsampling algorithm implemented.
It puts the value 1 in k elements of the binary array which correspond to k indexes randomly chosen. These indexes are randomly picked from an array - from its start to a certain point n - which contains all the possible indexes. In order to avoid choosing the same index twice, when an index is picked it then swaps places with the index at the last allowed place in the array, and the number n that defines this last possible place is decreased so that the index that had just been picked becomes out of bounds.
This algorithm, assuming the binary array only contains 0 at the start, creates a

random subsample of the possible indexes of size k in $O(k)$ time.

Algorithm 3.2 Random Subsampling Without Replacement

```
1: function NEWSAMPLE(k, n, binary, indexes)
2:     i, j, tmp
3:     for i = 0 → k do
4:         j = n * rand(0, 1)
5:         binary[indexes[j]] = 1
6:         tmp = indexes[j]
7:         indexes[j] = indexes[−− n]
8:         indexes[n] = tmp
9:     end for
10: end function
```

3.3.2 Squared Loss and Variance for Faster Trees

As discussed in section 2.3.1, regardless of the loss function that is used, the regression trees are grown using the squared loss function to find the splits. Equation 2.19 defines the expression that the algorithm is looking to solve. For each feature, the algorithm is trying to find the s that minimizes the cost of the split

$$cost(s) = min_{c'_1} \sum_{x_i | x_{ip} \leq s} (y_i - c'_1)^2 + min_{c'_2} \sum_{x_i | x_{ip} > s} (y_i - c'_2)^2 \tag{3.1}$$

with the y_i being the values we are fitting the trees onto. For gradient boosting, they are equal to the negative gradient i.e. $y_i = -g_{m,i}$. However, we will continue to refer to those values by y_i to be consistent with the notations of section 2.3. To solve this equation, the algorithm must find c'_1 and c'_2 which, if the squared loss is the loss function being used, also correspond to the left and right constants that best minimize the prediction risk. All the possible splitting indexes s are tried by scanning through the elements in the order of feature p. As the elements are scanned in the order of the current feature p, we can define

$$L_s(c'_1) = \sum_{i=1}^{s} (y_i - c'_1)^2 \tag{3.2}$$

and

$$R_s(c'_2) = \sum_{i=s+1}^{N} (y_i - c'_2)^2 \tag{3.3}$$

so that

$$cost(s) = min_{c'_1} L_s(c'_1) + min_{c'_2} R_s(c'_2) \tag{3.4}$$

The solutions to $min_{c'_1} L_s(c'_1)$ and $min_{c'_2} R_s(c'_2)$ are given by the expected values - in other words, by the means:

$$c'_1 = \frac{1}{s} \sum_{i=1}^{s} y_i = E_{1..s}[Y] \tag{3.5}$$

and

$$c'_2 = \frac{1}{N-s} \sum_{i=s+1}^{N} y_i = E_{s+1..N}[Y] \tag{3.6}$$

Proof: minimization of L_s

Assume that the minimum is not reached for the expected value. Thus, we can express the solution by the following:

$$c'_1 = E_{1..s}[Y] + \alpha$$

with $\alpha \neq 0$. Therefore, we have:

$$\begin{aligned}
L_s(c'_1) = & \quad \textstyle\sum_{i=1}^{s} (y_i - c'_1)^2 \\
= & \quad \textstyle\sum_{i=1}^{s} (y_i - (E_{1..s}[Y] + \alpha))^2 \\
= & \quad \textstyle\sum_{i=1}^{s} ((y_i - E_{1..s}[Y]) - \alpha)^2 \\
= & \quad \textstyle\sum_{i=1}^{s} (y_i - E_{1..s}[Y])^2 - 2\alpha \sum_{i=1}^{s} (y_i - E_{1..s}[Y]) + \sum_{i=1}^{s} \alpha^2 \\
= & \quad \textstyle\sum_{i=1}^{s} (y_i - E_{1..s}[Y])^2 - 2\alpha \left(\sum_{i=1}^{s} y_i - sE_{1..s}[Y]\right) + s\alpha^2
\end{aligned}$$

However:

$$\begin{aligned}
\sum_{i=1}^{s} y_i = & \quad s\left(\tfrac{1}{s} \textstyle\sum_{i=1}^{s} y_i\right) \\
= & \quad sE_{1..s}[Y]
\end{aligned}$$

Thus, we have:

$$\begin{aligned}
L_s(E_{1..s}[Y] + \alpha) = & \quad \textstyle\sum_{i=1}^{s} (y_i - E_{1..s}[Y])^2 + s\alpha^2 \\
= & \quad L_s(E_{1..s}[Y]) + s\alpha^2
\end{aligned}$$

Finally, as $s\alpha^2 > 0$ we can deduce that $L_s(E_{1..s}[Y] + \alpha) > L_s(E_{1..s}[Y])$ which proves that $L_s(c)$ is minimized for $c = c'_1 = E_{1..s}[Y]$.
A very similar demonstration can be used to prove that $c'_2 = E_{s+1..N}[Y]$.

These results allow us to define

$$cost(s) = \mathcal{L}_s + \mathcal{R}_s \tag{3.7}$$

with

$$\begin{aligned} \mathcal{L}_s = & \quad min_{c'_1} \sum_{i=1}^{s} (y_i - c'_1)^2 \\ = & \quad \sum_{i=1}^{s} (y_i - E_{1..s}[Y])^2 \\ = & \quad s \left(\frac{1}{s} \sum_{i=1}^{s} (y_i - E_{1..s}[Y])^2 \right) \\ = & \quad s.Var_{1..s}[Y] \end{aligned} \tag{3.8}$$

and

$$\begin{aligned} \mathcal{R}_s = & \quad min_{c'_2} \sum_{i=s+1}^{N} (y_i - c'_2)^2 \\ = & \quad (N-s).Var_{s+1..N}[Y] \end{aligned} \tag{3.9}$$

As $Var[Y] = E[Y^2] - E[Y]^2$, we can deduce the final expressions of $\mathcal{L}_s$ and $\mathcal{R}_s$:

$$\mathcal{L}_s = s \left(E_{1..s}[Y^2] - E_{1..s}[Y]^2 \right) \tag{3.10}$$

$$\mathcal{R}_s = (N-s) \left(E_{s+1..N}[Y^2] - E_{s+1..N}[Y]^2 \right) \tag{3.11}$$

This means that, for each feature, we can compute the costs of all the possible splits in one single pass through the data just by maintaining up to date the left and right sums (which give us the left and right expected values $E_{1..s}[Y]$ and $E_{s+1..N}[Y]$ by dividing respectively by s and $N - s$) and sums of squares (which give us the left and right expected squared values $E_{1..s}[Y^2]$ and $E_{s+1..N}[Y^2]$ by dividing respectively by s and $N - s$). Besides, we can just compute $\sum_{i=1}^{N} y_i$ and $\sum_{i=1}^{N} y_i^2$ once, as they remain the same for all the features, in order to deduce the left and right sums and sums of squares for all the splits when trying them.

3.3.3 Computing the best constants

Two loss functions have been implemented: the squared loss for regression problems, and the binomial deviance for classification problems.

When using the squared loss function, the best constants - the ones that minimize equation 2.20 - are given by $c_1 = E_{1..s}[Y]$ and $c_2 = E_{s+1..N}[Y]$, where Y represents the variable we are fitting the tree onto, so no additional computation is required.

Using the binomial deviance is a bit more complicated and we need to go back to the gradient boosting notations. So, as we are fitting the trees on the negative gradient values $-g_{m_i}$, the y_i now represent the original response values we are building the gradient boosted model onto. With these notations, we have[4]:

$$c_1 = \frac{\sum_{i=1}^{s}(-g_{m,i})}{\sum_{i=1}^{s}(y_i - g_{m,i})(1 - y_i + g_{m,i})} \tag{3.12}$$

and, likewise:

$$c_2 = \frac{\sum_{i=s+1}^{N}(-g_{m,i})}{\sum_{i=s+1}^{N}(y_i - g_{m,i})(1 - y_i + g_{m,i})} \tag{3.13}$$

These constants can also be computed for all the possible splits with one single pass through the data by maintaining the left and right sums and sums of $(y_i - g_{m,i})(1 - y_i + g_{m,i})$.

3.3.4 The implementations

A sequential implementation

With all these developments we can now write the final regression tree algorithm. Detailed pseudo code is given in algorithm 3.3 for a sequential implementation using the squared loss. The actual implementation is a function that can deal with both squared loss and binomial deviance but this wasn't included here for more clarity. The resulting tree is represented by a vector of doubles organized as follows:

- Each node is represented by 6 doubles:

    1. The feature p used for the split

[4] these results come from the GBM package source code

2. The split value $\frac{x_{s,p}+x_{s+1,p}}{2}$
3. The error on the left part of the split
4. The error on the right part of the split
5. The constant c_1 that best fits the left region
6. The constant c_2 that best fits the right region

- The nodes are numbered as follows:
  - root = 0 (the root doesn't represent any split, it is used to contain the best constant prediction and the loss that comes from that prediction, which can be used later on to compute the relative importance of the features)
  - first actual node = 1
  - left child of a node = 2k
  - right child of a node = 2k+1
- Node k is located at the index 6k in the vector

This strategy enables storing a tree in a compact and contiguous data structure which thus benefits from the processor's cache. The errors on the left and right parts of the each split are computed and stored in each node to enable the computation of the features relative importances. Section 3.4 details this algorithm.

A parallel implementation

Algorithm 3.3 has also been parallelized with OpenMP in order to benefit from the multi-core capabilities of today's processors. As the computations for each feature - trying all the possible splits for the features while computing their costs and best constants - are independent, the idea was to execute them in parallel. Care had to be given to choose the private variables - owned and seen by a single thread only - and the ones shared among the threads. The variables made private must be feature-dependent - typically, the cost of the split, the left and right constants, sums and sums of squares - while the shared ones remain identical regardless of the feature - typically, anything related to the dataset, the total sum and sum of squares, and the current best split. Finally, for the threads not to interfere with one another when checking and updating the best split variables,

Algorithm 3.3 Recursive Partitioning: regression tree with squared loss

function REGRESSIONTREE(x, y, orderedElements, inSample, P, N, S, S2, depth, result, k)

x the dataset
y the values we fit the tree on
orderedElements the ordered indexes according to each feature
inSample a binary vector to indicate if $x[i]$ is in the subsample
P the number of features
N the number of examples in the subsample
S the total sum
S2 the total sum of squares
depth the remaining depth levels
result the resulting tree
k the node number

left $= \{nL, SL, S2L, L, CL\}$ the left number of examples, sum, sum of squares, cost and best constant
right $= \{nR, SR, S2R, R, CR\}$ the right number of examples, sum, sum of squares, cost and best constant
bestSplit $= \{feature, split, cost, CL, CR\}$

for $p = 1 \rightarrow P$ **do** ▷ For each feature
  $left = \{0, 0, 0, 0, 0\}$
  $right = \{N, S, S2, S2 - S^2/N\}$
  for $i = 1 \rightarrow \#y - 1$ **do**
    if $inSample[orderedElements[p][i]]$ **then** ▷ In sample
      $left = \{nL + 1, SL + y[i], S2L + y[i]^2, S2L - SL^2/nL, CL\}$
      $right = \{nR - 1, SR - y[i], S2R - y[i]^2, S2R - SR^2/nR, CR\}$
      if $x_{i,p} \neq x[i+1][p]$ **then** ▷ Complete split
        if $L + R < cost$ **then** ▷ Best so far
          $bestSplit = \{p, \frac{x[i][p]+x[i+1][p]}{2}, L + R, SL/nL, SR/nR\}$
        end if
      end if
    end if
  end for
end for
if bestSplit is initialized **then** ▷ A split was found
  $result[6k] = bestSplit$ ▷ Store the node of the tree
  if $depth > 1$ **then**
    Compute for the subsamples: *inSampleLeft*, *nL*, *SL*, *S2L*, *inSampleRight*, *nR*, *SR*, *S2R*
    RecursivePartitioning*(X, Y, orderedElements, inSampleLeft, P, nL, SL, S2L, depth-1, result, 2k)* ▷ Compute the left child
    RecursivePartitioning*(X, Y, orderedElements, inSampleRight, P, nR, SR, S2R, depth-1, result, 2k+1)* ▷ Compute the right child
  end if
end if
end function

a critical section - a portion of code that only one thread can execute at a time - had to be defined around that portion of code. Indeed, suppose we have two threads, thread A and thread B. Without this critical section we could observe the following incorrect succession of actions:

1. Thread A finds a split of cost $cost_A < cost$, that is, the best split so far
2. Thread B finds such a split too: $cost_B < cost$
3. Thread A updates the feature number of the best split to p_A
4. Thread B, which might run faster than thread A from time to time, updates the feature number of the best split to p_B, along with all the other variables related to it: split value, cost and best constants
5. Then, thread A continues to update the best split variables: split value, cost and best constants

The problem here is that the best split, as described by the variables, would correspond to splitting the feature p_B with the values of a possible split for feature p_A. Thus, the program would have entered an incorrect state. Besides, not only does this critical section needs to surround the portion of code that updates the variables corresponding to the best split but also the code that checks if the current split is better than the best split found so far. Indeed, if the critical section only surrounded the portion of code that updates the variables, thread A could find a split of $cost_A < cost$ and then enter the critical section. While thread A would be updating the best split variables, thread B could also find a split with a lower cost than the best split found so far, that is $cost_B < cost$, and would update the best split variables once thread A would be done doing so. Thus, the final best split variables would describe the split found by thread B. However, nothing ensures that $cost_B < cost_A$ so the best split variables might be describing a non-optimal split.

3.4 Relative Importance of Features

Equation 2.24 gives the squared relative importance of a feature. In order to compute it, it is necessary to know the diminution of error that is induced by each split of the trees. If we know the error before splitting, and the error after

splitting, we can subtract the two to get the diminution of error for the split. Each node but the root is the left or right child of its parent node. Hence, the error before the child node's split is the error obtained when using the parent's best constant on the left or right part of the parent's split. As we store this value in each node's data structure (see section 3.3.4) we have all the necessary to compute the diminution of error according to each split. We then only have to square it, and add it to the corresponding feature's importance. One recursive travel of the tree, from the root to the leaves, allows computing all the splits' diminutions of error. We then only have to repeat this operation on all the trees and average the values (see equation 2.25). Finally, the relative importances are the square root of the obtained values. Algorithm 3.4 gives pseudo-code for the recursive algorithm applied on each tree.

Algorithm 3.4 Relative Importance of Features

function RECURSIVERI(ri, previousError, treeVector, k, maxk)
  ri the result vector (1 entry per feature)
  previousError the error on the previous split
  treeVector the tree data structure
  k the current node number
  maxk the maximum node number

  index $= 6 * k$ the position of the node in treeVector
  error $= 0$ the error on this split

  if $k \geq maxk$ **then return**
  end if
  if $treeVector[index] > 0$ **then** ▷ if there is a node
    $error = treeVector[index + 2] + treeVector[index + 3]$
    $ri[treeVector[index]] += (previousError - error)^2$
    $recursiveRI(ri, treeVector[index + 2], treeVector, 2 * k, maxk)$
    $recursiveRI(ri, treeVector[index + 3], treeVector, 2 * k + 1, maxk)$
  end if
end function

3.5 GBT: the new package

Algorithm 3.5 gives a pseudo-code that describes the implemented gradient boosted trees algorithm. A new R package, named GBT, was created to implement all the algorithms described in this chapter. The package is implemented in C with

an R wrapper that includes a built-in stratified cross-validation (see section 2.5). Two loss functions can be used, the squared loss and binomial deviance. It comes with a function to build the model, to predict unseen data, and to compute the relative importance of features.

Because of the huge overheads of R's OpenMP implementations in most platforms (see section 3.1.1), no parallelization other than the one described in section 3.3.4 turned out to be worth it. The code that has been parallelized represents the major part of the computations done by the gradient boosted trees algorithm so the remaining computations aren't important enough to benefit from any parallelization. Another possibility would have been to parallelize the cross-validation runs instead of the regression trees - and not in addition of, as the current parallelization already makes use of all the threads computing power. This however would only benefit the implementation when cross-validation is being used.

Algorithm 3.5 GBT: Gradient Boosted Trees

function GBT(x, y, bag.fraction, loss, M, interaction.depth)
  $N = \#y$ ▷ Number of examples
  $bag.size = N * bag.fraction$ ▷ Subsample size
  $P = \#x[1]$ ▷ Number of features
  $allIndexes = 1 : N$ ▷ All possible indexes
  $orderedElements = orderElements(x)$ ▷ Order according to the features
  $inSample = \{1\}^N$ ▷ Initialize sample
  $trees[0] = fitBestConstant(x, y, loss)$
  $predictions = predict(trees[0], x)$
  for $m = 1 \rightarrow M$ **do** ▷ Boosting iterations
    $inSample = \{0\}^N$
    $newSample(bag.size, N, inSample, allIndexes)$
    $residuals = negativeGradient(x, y, predictions, inSample, loss)$
    $S = sum(residuals)$
    $S2 = sum(residuals^2)$
    $regressionTree(x,\ residuals,\ orderedElements,\ inSample,\ P,\ N,\ S,\ S2,\ interaction.depth,\ trees[m],\ 0)$
    $predictions = predictions + predict(trees[m], x)$
  end for
return *trees*
end function

3.6 GBM vs GBT

The speed of both packages have been compared when launched on a dataset provided by AstraZeneca. The dataset is composed of 1604 examples and 634 binary features. The binomial deviance loss function was used, and both implementations were launched with the same parameters. The test was a 10 folds cross-validation of 10000 boosting iterations with trees of maximum depth 3 - which corresponds to AstraZeneca's protocol of experiments, launched on an Intel core I5 CPU (2 cores - 4 hyperthreads) at 2.5 ghz clock speed, under a 64 bits Windows operating system. Both implementations of the GBT package, sequential and parallel, have been compared to the GBM package. The execution times were the following:

- GBM: 3479s
- Sequential GBT: 2122s
- Parallel GBT: 1137s

Figure 3.1 displays a graph of the relative execution times of the three implementations. The GBM package serves as reference and so its execution time is attributed the value 1. The sequential GBT implementation performed 1.7 times faster - it required about 61% of the time required by GBM to achieve the same operations. The parallel implementation of GBT, with the important overheads induced by R's OpenMP implementation under Windows, ran 46% faster than the sequential GBT implementation, which corresponds to running more than 3 times faster than the GBM package, using less than 33% of the time required by the GBM package to perform the same operations. It is very likely that the parallel implementation would speed-up the execution even more under a Linux system where OpenMP doesn't induce important overheads. We can here conclude that the GBT package outperforms the GBM package in regard to the speed.

While this chapter detailed theoretical and practical developments regarding the implementation of gradient boosted trees algorithms, and the results obtained, the next chapter explains the developments of this project concerning the gradient boosting algorithm itself - not its implementations - on a theoretical and experimental point of view. The ideas explored were added to the GBT package to conduct the experiments.

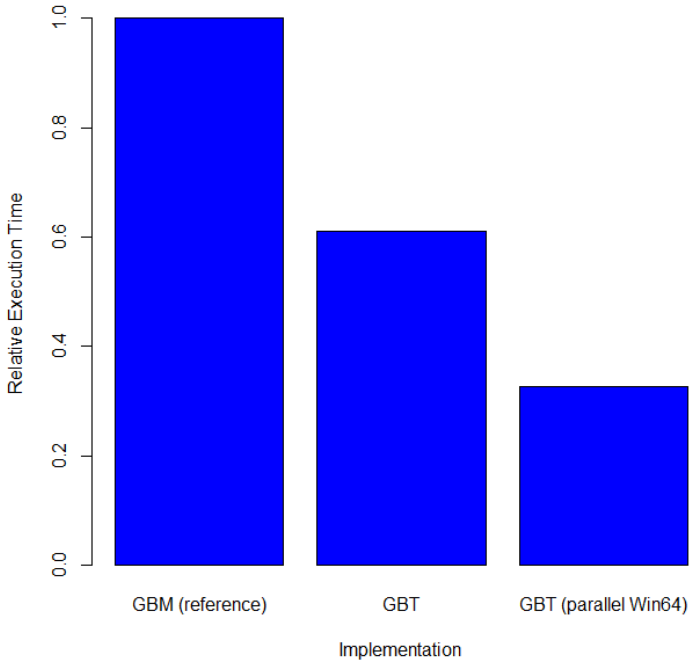

Figure 3.1: Relative execution time: GBM, GBT and parallelized GBT, under an Intel core I5 CPU (2 cores, 4 hyperthreads)

Chapter 4

The Impact of Shrinkage

One of the main goals of the project was to investigate how shrinkage (see equation 2.26) impacts on the gradient boosting algorithm. The idea was to slowly anneal the learning rate over the boosting iterations, that is, to decrease its value at every iteration to give more importance to the first base learners compared to the next ones, allowing the algorithm to "focus" over time. This chapter investigates the behaviours of the gradient boosting algorithm regarding the shrinkage parameter.

4.1 Test Protocol

When not stated otherwise, the experiments conducted in this chapter used the real-world dataset provided by AstraZeneca, made of 1604 examples and 634 binary features, that was also used in section 3.6. The response indicates if the example - a patient - was given a drug or a placebo. Each feature indicates the presence or absence of a particular symptom for a patient. Building a prediction model on this data is a binary classification problem so the tests were conducted using the binomial deviance. A sample of 50% the size of the complete dataset is randomly created for each boosting iteration, which is the default - and recommended - proportion for the GBM package [Rid07]. The results presented are the averages over 10 launches of 10-folds stratified cross-validations with trees of maximal depth 3.

4.2 Is Shrinkage Necessary?

The shrinkage parameter was introduced into the algorithm by Friedman in 2001 [Fri01] in order to reduce the overfitting phenomenon. A heavy shrinkage corresponds to a small learning rate $\nu << 1$ while a light shrinkage corresponds to ν close to 1. $\nu = 1$ means no shrinkage at all, as it means taking the complete negative gradient steps. We verify here that the introduction of shrinkage was justified.

Figure 4.1 plots the evolution of training and test deviances over 1,000 boosting iterations when applying a very light shrinkage. Indeed, we use here $\nu = 0.1$ which is a hundred times bigger than the default value of 0.001 used in the GBM package [Rid07]. We can observe that the training error keeps decreasing throughout the boosting iterations, as one could expect with gradient boosting - its principle being following the negative gradient to decrease the training error. However, the generalization capability of the method seems very limited as the error on unseen examples - on the test set - decreases for only a few dozen iterations before increasing. After 200 iterations only the generalization error is worse than with a single tree.
We can conclude that gradient boosting with light shrinkage overfits very quickly: it fits the training data too well to generalize correctly on new examples. Similar results were observed without shrinkage, at the difference that overfitting happened even faster. Appendix A expands on this overfitting behaviour described by Hastie, Tibshirani and Friedman [ele09, Fri01].

Figure 4.2 compares the propensity that gradient boosting has to overfit with light and heavy shrinkage. The test error curves are displayed for $\nu = 0.1$ (light shrinkage) and $\nu = 0.001$ (heavy shrinkage). We can observe that the method starts faster with a light shrinkage (the test error decreases faster) than with a heavy shrinkage, which makes sense as bigger steps are taken with light shrinkage. However, as showed in the previous experiment, the test error rises then very quickly - after a few dozen iterations only - with a light shrinkage due to overfitting the training data. Besides, light shrinkage is greatly outperformed by heavy shrinkage when running more iterations: with a heavy shrinkage, the test error keeps decreasing throughout all the 1000 iterations to reach values lower than the minimum reached by light shrinkage.

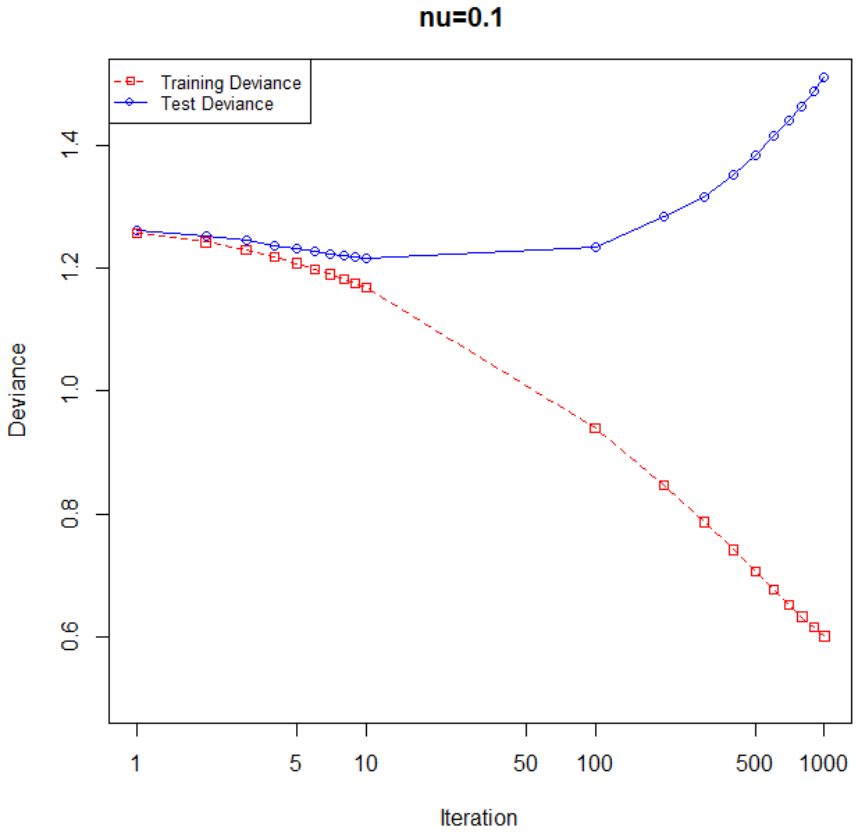

Figure 4.1: Training and test deviance with light shrinkage: early overfitting

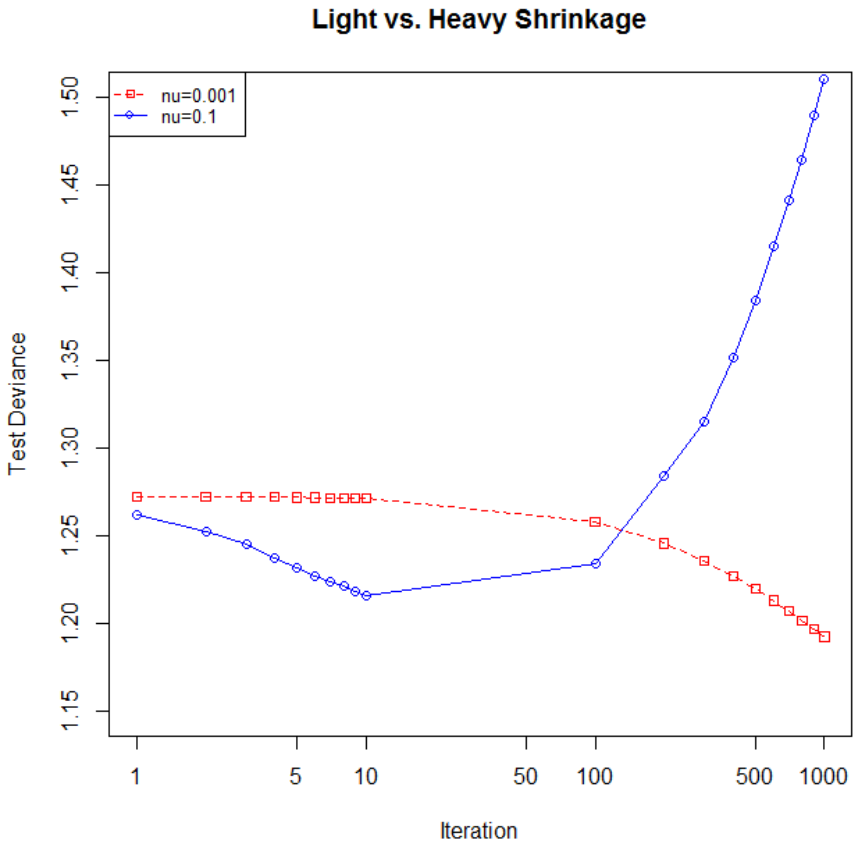

Figure 4.2: Test deviance: light shrinkage versus heavy shrinkage

We can conclude that heavy shrinkage provides gradient boosting with a much stronger resistance to overfitting. Also, even if the number of iterations required is bigger with a heavier shrinkage, it however outperforms light - or no - shrinkage as it allows reaching smaller error rates. Intuitively, the idea is that taking lots of small steps is better than taking very few big steps. The smaller the step sizes, the higher the accuracy, and the higher the number of iterations required.

Thus, the main idea behind this project was to try to combine the advantages of both methods: starting with high learning rate values (close to 1) and finishing with low values (close to 0), naming this method shrinkage annealing. Intuitively, the first iterations of the algorithm do not overfit but learn the overall shape of the function F to predict, as the learner is not very specific yet, while the last iterations focus more on the small - local - variations of F. Thus, high first

values would allow the algorithm to quickly learn the overall shape of F while low last values would fit less the local variations of F, which can be the result of measurement errors or noise on the data, for instance. The goal then is to take advantage of the resistance to overfitting and high accuracy provided by a heavy shrinkage without having to sacrifice the computational cost by needing to increase the number M of iterations.

To this end, the gradient boosting's update rule (equation 2.10) has to be modified to the following:

$$F_m = F_{m-1} - \nu_m \gamma_m h_m, \quad 0 < \nu_m \leq 1 \tag{4.1}$$

Or, in case of gradient boosted trees (equation 2.22):

$$F_m(x) = F_{m-1}(x) - \nu_m \sum_{k=1}^{K} \gamma_{mk} I(x \in R_k) \tag{4.2}$$

From there, the way we decrease the parameter ν_m needs to be decided. The next sections describe and experiment different ways to do so.

4.3 Arithmetic and Geometric Shrinkage Annealing

The first two approaches that have been undertaken are:

- An arithmetic decrease

$$\nu_m = a - b(m-1) \tag{4.3}$$

- And a geometric decrease

$$\nu_m = a.b^{m-1} \tag{4.4}$$

In both cases the parameters a and b are computed automatically according to the user's choice of ν_1 and ν_M: to control the speed of the decrease in a user-friendly manner, the user chooses the first and last step sizes (ν_1 and ν_M) and the GBT package computes the corresponding parameters itself.

It can be shown that, for an arithmetic decrease:

$$a = \nu_1 \tag{4.5}$$

and

$$b = \frac{\nu_M - \nu_1}{M} \tag{4.6}$$

For a geometric decrease:

$$a = \nu_1 \tag{4.7}$$

and

$$b = M^{\frac{ln(\frac{\nu_M}{\nu_1})}{Mln(M)}}; \tag{4.8}$$

Figure 4.3 shows evolutions of the step sizes that can be achieved with these methods. The geometric progressions reach lower values faster than the arithmetic progressions for identical ν_1 and ν_M.

4.3.1 Convergence Behaviours

Some basic experiments have first been conducted to explore the behaviours of the different approaches. We are here approximating a sum of ten independent Gaussians of identical mean but different variances. The ten independent Gaussians were the features used to train gradient boosted decision stumps with a thousand training samples. The training samples were unbiased, equidistributed and observed without noise. In so, they are not representative of a real dataset and don't allow the experiment to explore the propensity of the methods to resist overfitting but their convergence behaviours.

Figure 4.4 shows how the gradient boosted model fits the data after ten iterations when using a constant shrinkage parameter $\nu = 0.1$. Figure 4.5 shows the same with an arithmetic shrinkage annealing, and figure 4.6 with a geometric shrinkage annealing, which both start with higher step sizes. While the fixed heavier shrinkage parameter induces a slow start of the procedure - the model "struggles" to learn the overall target shape - both the geometric and arithmetic shrinkage annealing benefit from their higher step sizes and fit the shape of the data much better. The question is now to find out if they benefit from the lower step sizes they take during the subsequent iterations to resist overfitting.

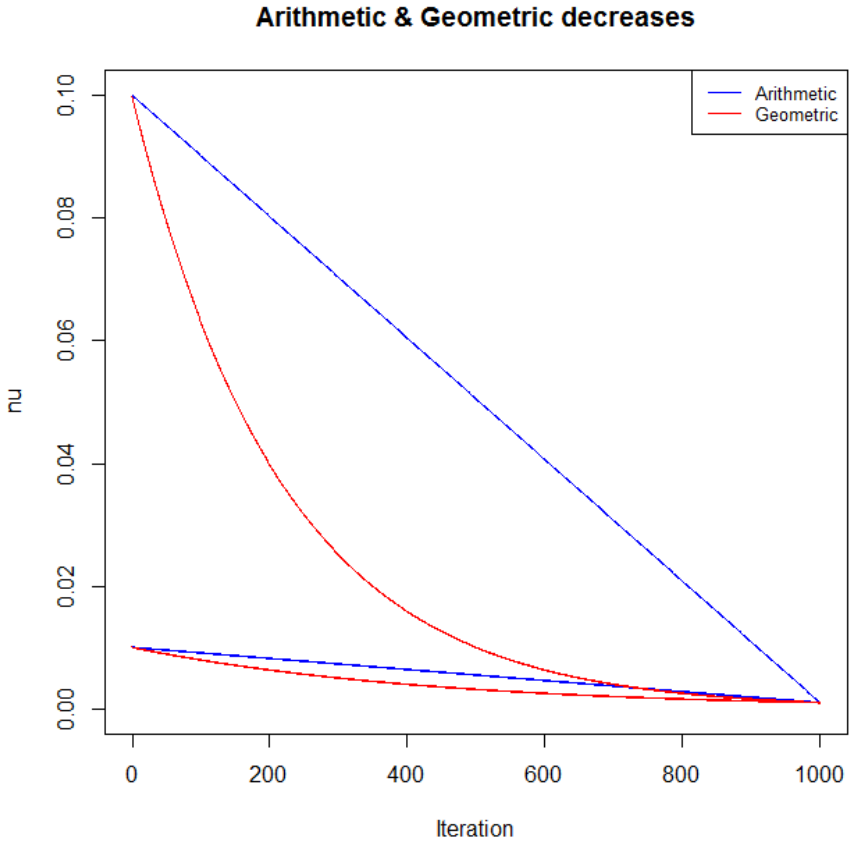

Figure 4.3: Arithmetic and geometric step sizes evolutions

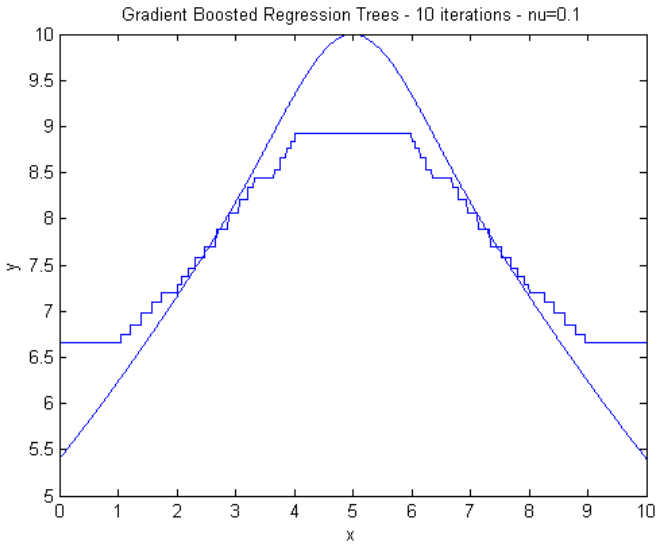

Figure 4.4: An approximation of a sum of 10 Gaussians with a constant shrinkage parameter

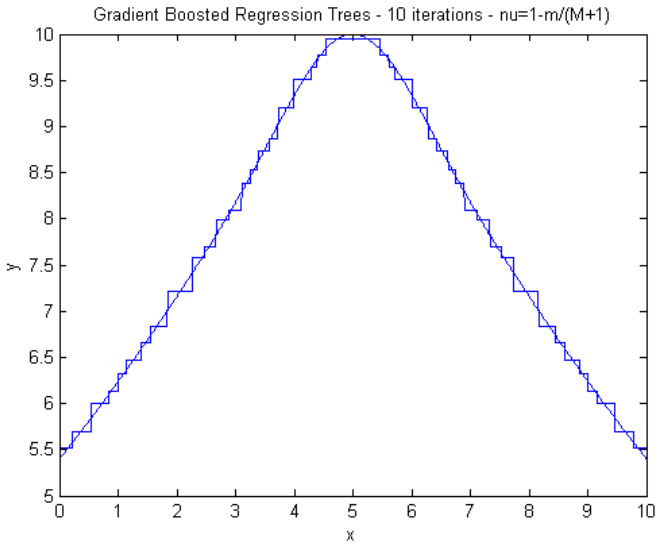

Figure 4.5: An approximation of a sum of 10 Gaussians with an arithmetic shrinkage annealing

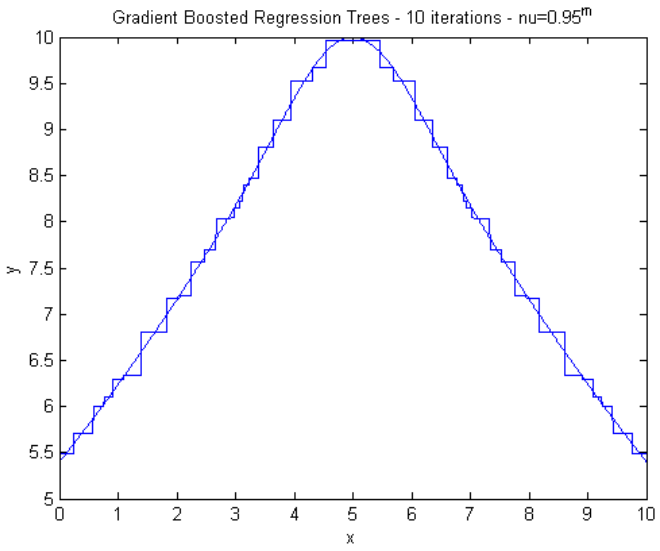

Figure 4.6: An approximation of a sum of 10 Gaussians with a geometric shrinkage annealing

4.3.2 Propensity to Overfit

The reference result is a test deviance of 1.157 obtained with a fixed heavy shrinkage ($\nu = 0.001$) over 10000 iterations. Thus, both methods were launched when starting with high step sizes ($\nu = 0.1$) and with medium step sizes ($\nu = 0.01$), and finishing with little step sizes (high shrinkage, $\nu = 0.001$). Figure 4.7 shows the performance - test error - of both methods with up to a thousand iterations. The methods weren't launched when starting with small step sizes as our goal is to investigate the effect of starting with higher step sizes. They were launched for 1000 iterations only, ten times less than the reference, as we are hoping to reduce the required number of iterations.

When starting with high step sizes we observe an important overfitting after a few dozen iterations only, as we did with fixed light shrinkage. Besides, the phenomenon is stronger for arithmetic shrinkage. We can conclude that ν_m remains in high values for too long. When starting with smaller values, overfitting happens later but the best deviance reached was 1.168. Other combinations of parameters were tried but all of them were outperformed by a fixed heavy shrinkage too. It is possible that ν_m, once again, takes too high values for too many iterations.

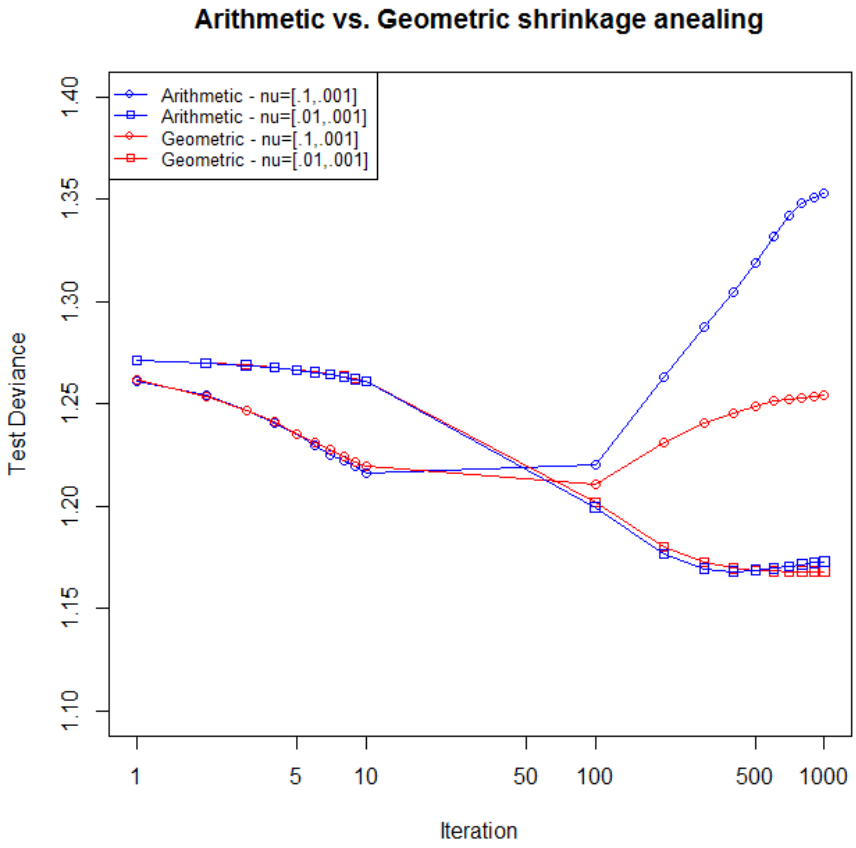

Figure 4.7: Arithmetic and geometric shrinkage annealing performance

Hence, we can make the hypothesis that the arithmetic and geometric progressions aren't decreasing fast enough. To verify this idea a negative exponential decrease has been implemented and its investigation is presented in the next section.

4.4 Negative Exponential Shrinkage Annealing

So that the step sizes can be decreased faster, a negative exponential decrease has been implemented:

$$\nu_m = a.exp(-b.m) + c \tag{4.9}$$

Here, the user can choose three parameters in order to better control the shape of the descent: the first value ν_1, the last value ν_M, and the iteration number m_{75}

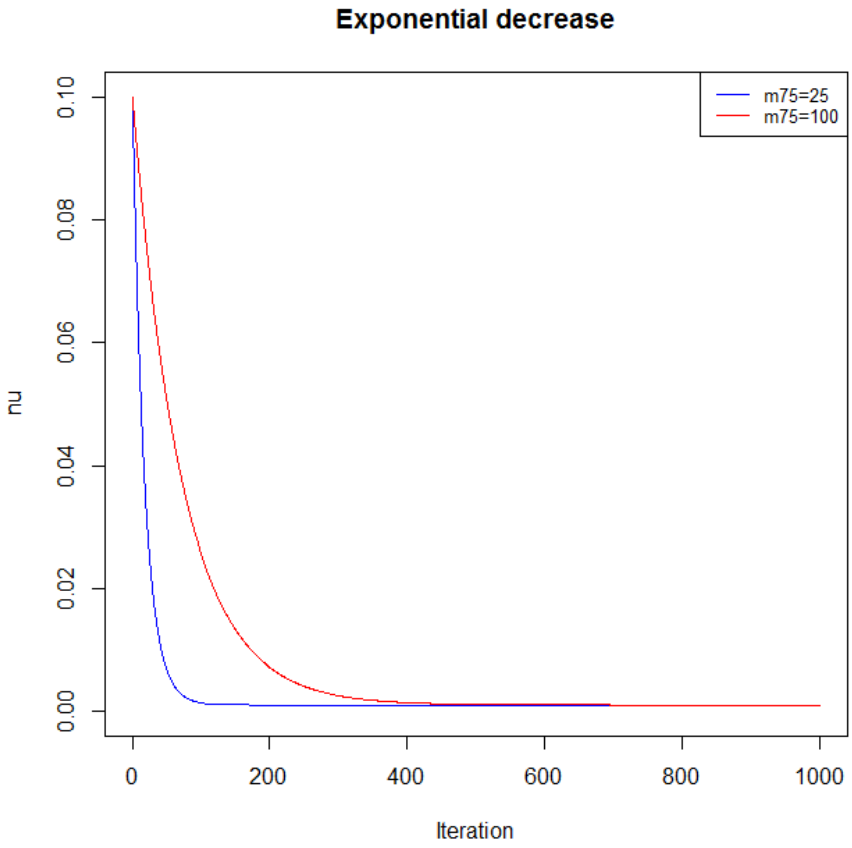

Figure 4.8: Negative exponential step sizes evolutions

for which 75% of the descent has been done, defined by $\nu_{m_{75}} = 0.25(\nu_1 - \nu_M)$. It can be shown that

$$a = \frac{ln(0.25)}{1 - m_{75}} \tag{4.10}$$

$$b = \frac{\nu_1 - \nu_M}{exp(-a) - exp(-a.M)} \tag{4.11}$$

and

$$c = \nu_M - b.exp(-a.M) \tag{4.12}$$

Figure 4.8 shows evolutions of the step sizes that can be achieved with this method. The speed of the decrease can effectively be controlled by the third parameter m_{75}.

Several negative exponential decreases have been tried. Figure 4.9 shows the

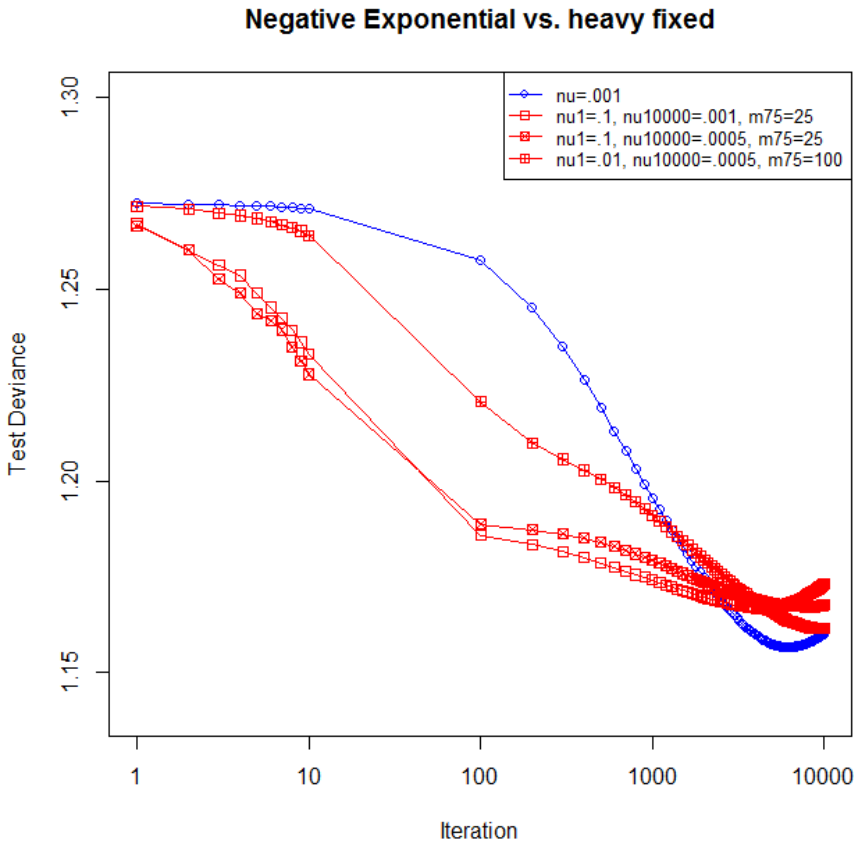

Figure 4.9: Negative exponential shrinkage annealing

test deviance for 3 tries along with the reference results (fixed $\nu = 0.001$) for 10000 iterations:

1. $\nu_1 = 0.1$, $\nu_M = 0.001$ and $m_{75} = 25$
2. $\nu_1 = 0.1$, $\nu_M = 0.0005$ and $m_{75} = 25$
3. $\nu_1 = 0.01$, $\nu_M = 0.0005$ and $m_{75} = 100$

1) starts with high step sizes (light shrinkage) and finishes with the reference - heavy - shrinkage. It performs 75% of the step size diminution in 25 iterations so even though the first step sizes are high, they decrease very quickly to reach more overfitting-resistant values. Compared to the reference, represented by the blue curve, the start is much faster but the reference model outperforms it with the subsequent iterations. However, both models start overfitting - when the test

error increases - at the same number of iterations. Thus, they seem to similarly resist overfitting. However, even with a fast decrease of the step sizes, the first big steps seem to have unalterably diminished the model's ability to produce accurate generalized predictions.

2) finishes with smaller step sizes - a heavier shrinkage than the reference, $\nu = 0.0005$. The idea was to investigate if a heavier shrinkage during the later iterations could counteract the first big steps' negative result on the model's accuracy. As one could expect, overfitting happens more slowly - the test error curve increases more slowly. However, the predictions aren't improved compared to 1).

3) starts with medium step sizes and finishes with the heavy shrinkage $\nu = 0.0005$ too. However, the decrease in step size is slower. As expected, the start speed is in between the high and small step sizes values. The model starts overfitting slightly later than the others, and more slowly - the error curve increases very slowly due to the heavier shrinkage. However, this model's ability to produce accurate predictions has also been impaired by the higher first step sizes.

The conclusion we can reach is that even if big step sizes induce a faster decrease of test error at first, they seem to push the model towards a suboptimal solution it can never move away from. Gradient descent - often called steepest descent - is a technique that cannot guarantee finding a global optimum unless for special cases as convex functions, and it therefore makes sense to find this problem in gradient boosting. In real world problems the data is usually very complex and convexity is rarely the case. Regularization techniques like shrinkage try to overcome this issue but it seems that a bad step at the beginning can be enough to push the model towards a local optimum. In the next section, we try a fitting procedure that relies less effectively on shrinkage but more on the base learners by constructing deeper - more precise - trees to explore the effects of negative exponential shrinkage annealing on it.

4.5 Shrinkage Annealing with Deeper Trees

4.5.1 The Importance of Shrinkage in the Fitting Process

As explained in section 2.3, a regression tree partitions the feature space in regions, each of which is assigned a constant value. The maximum depth of the tree controls the maximum number of regions that form this partition, but also the maximum level of interactions between the features. For a tree of maximum depth D, up to D features can be used to define a region, so the maximum level of interaction is D. It is common for gradient boosted trees to limit the maximum interaction depth D to a small value - typically between 1 and 3. However, this doesn't mean that no more than D features can interact in the final boosted model.

Consider a simple example of gradient boosted trees with a maximum interaction depth $D = 1$. At the first boosting iteration, the tree that is created splits according to the feature $X1$, and assigns the value $a1$ for the values smaller than the split (left part), and $a2$ for the bigger values (right part). At the second iteration, the new tree that is created splits according to the feature $X2$, and assigns the values $b1$ and $b2$ for the left and right parts of this new split. Thus, the feature space is partitioned into four regions as showed in figure 4.10. Even if the maximum interaction of each tree is one, two levels of interaction are used to define the regions: feature $X1$ and $X2$ are used at the same time to define the regions. The same partition could have been obtained with a single tree of depth 2.

This example illustrates that gradient boosted trees actually corresponds to creating a partition of the feature space and assigning a constant to each region just as would do a bigger regression tree. The difference being that the constants assigned to the regions, and the regions, are defined by both the iterated gradients and the shrinkage parameter. Gradient boosting small trees is just a different way of constructing a bigger regression tree. Thus, allowing more interactions in the base learners while reducing the number of iterations reduces the influence of the shrinkage parameter on the fitting process. This is what we experiment here.

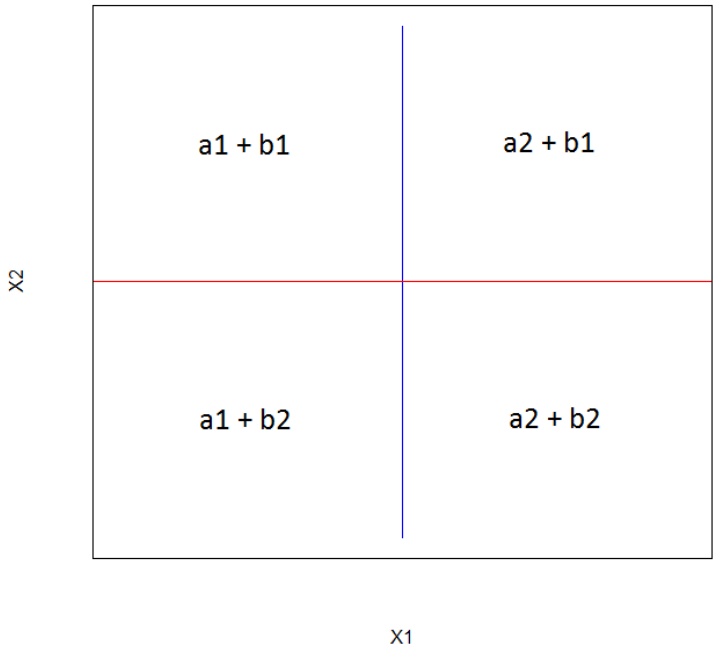

Figure 4.10: Partition of the feature space

4.5.2 Fitting with Deeper Trees

The problem when allowing the trees to go deeper is that each additional level of interaction multiplies the number of possible nodes of the tree by two, and thus does the same to the memory needed to store it. In section 3.3.4 we saw that we used 6 double values per node. Thus, 48 octets are required per node. As a tree of maximal depth D can contain up to 2^D nodes ($2^D - 1$ real nodes plus the root), as we cannot know this number before building the tree, and as we allocate a contiguous vector to store a tree, we need $12 * 10000 * 2^D$ octets of memory to store the trees. With $D = 3$ we only needed 385 octets per tree. However, if we want very deep trees, memory becomes an issue:

- $D = 8$, up to 256 nodes, up to 12 KO per tree
- $D = 16$, up to 65536 nodes, up to 3 MO per tree

With 10000 boosting iterations, and so 10000 trees, almost 3 GO of memory would be required for trees of depth 16. Besides, with 8192 times more nodes than a tree of depth 3, the computation is up to 8192 times more important so time becomes an issue too. Thus, we cannot increase the depth of the trees a lot.

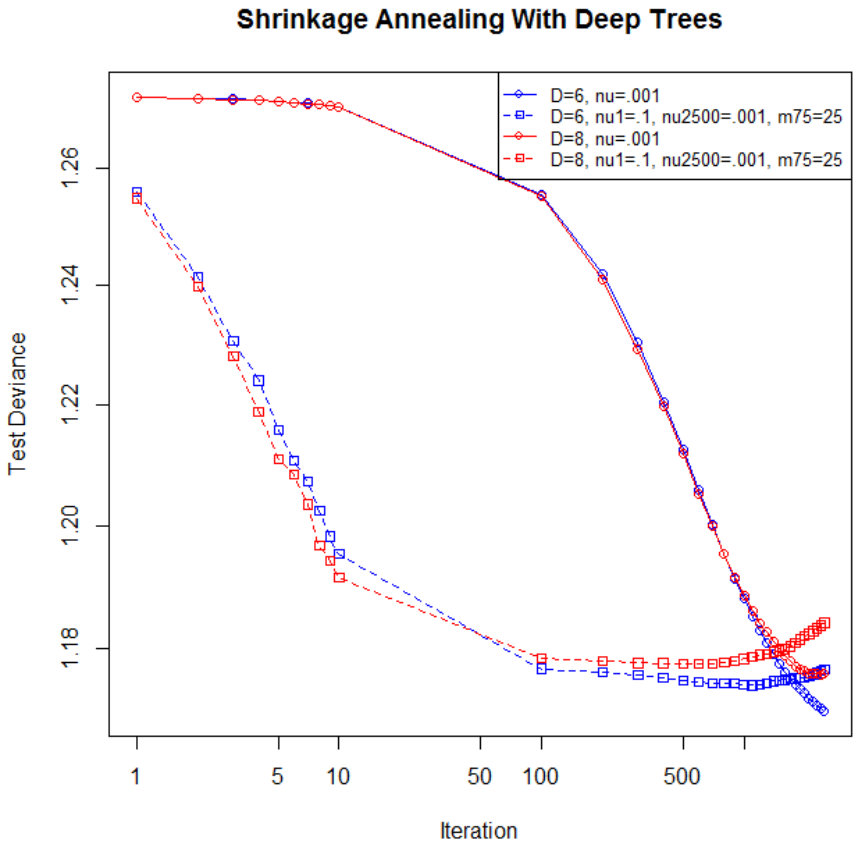

Figure 4.11: Shrinkage annealing versus fixed heavy shrinkage with trees of maximum depth 6 (blue) and of maximum depth 8 (red).

We will use here trees of depth 6 and 8.
However, with bigger trees we can reduce the number of boosting iterations. So far, we've been using 10000 trees of maximal depth 3. As a tree of maximal depth D partitions the feature space into up to 2^D regions, we were partitioning the feature space into up to $10000 * 2^3 = 80000$ regions[1]. With trees of depth 6 only 1250 iterations would be required to create up to 80000 regions. We will launch the experiments with trees of depth 6 and 8 for 2500 iterations.

Figure 4.11 shows the results obtained when comparing heavy fixed shrinkage ($\nu = 0.001$) with negative exponential shrinkage annealing ($\nu_1 = 0.1$, $\nu_{2500} =$

[1]The actual number of regions is likely to be less than that as 1) not all trees are full and 2) it is possible that two different trees create a same region.

0.001 and $m_{75} = 25$). The conclusions are the same as previously, shrinkage annealing being outperformed by a fixed heavy shrinkage, even if this is less clear with the deepest trees as the result of the shrinkage being less important in the fitting process. However, the errors obtained are much worse than the ones obtained earlier with smaller trees, which suggests that the gradient boosting fitting process performs better than the classic approaches to create regression trees. This nonetheless doesn't mean that gradient boosting will always perform better with smaller trees as they might poorly approximate the gradients, which could result in the negative gradient not being properly followed. A compromise for the depth of the trees needs to be found.

In this chapter we first showed the importance of shrinkage. We then experimented different approaches to shrinkage annealing, and in particular one that followed a negative exponential decrease as the step size needs to decrease fast for the model to resist overfitting. We revealed that while it accelerates the convergence towards an optimum, this optimum is only local and using a heavy shrinkage from the beginning leads to more accurate models. Finally, increasing the depth of the trees showed that a compromise for the maximal depth needs to be found for the negative gradient to be properly followed without giving too much importance to the trees fitting process compared to the gradient boosting fitting process.

Chapter 5

Conclusions and Limitations

5.1 GBT

In chapter 3 we studied regression trees in detail and found a way to efficiently compute them for gradient boosting. We then created both a sequential and a parallel implementation of the state-of-the-art gradient boosting algorithm - stochastic gradient boosted trees - that meet the criteria fixed at the beginning of the project. We created an R package - GBT - with a C engine, for a total of about 850 lines of code and 125 lines of comments (about 13%) that includes:

- Regression trees,
- A stochastic sub-sampling procedure,
- A loss function suited for regression problems (squared loss),
- A loss function suited for classification problems (binomial deviance),
- A built-in stratified cross-validation procedure,
- The computation of features relative importances,
- A function to predict outputs.

The tests showed that our sequential implementation outperforms the current reference gradient boosting software, GBM. It required about 61% of GBM's time to achieve the same operations on a real-world problem. They also showed that our parallel implementation required less than 33% of GBM's time for the same operations on the testing machine, a 2 cores CPU with 2 hyperthreads each,

running under a 64 bits Windows Operating System, and that the improvements thanks to parallelization might even be greater under a Linux system.

Even if these results are pleasant, as it often takes several hours to train a gradient boosting model on a real-world dataset so the gain in time is welcome, this new implementation - the GBT package for R - cannot fully replace GBM, the latter coming with more functionalities. GBT would need to implement more loss functions, which weren't covered in this document, and to make it possible to define a cost matrix to further describe the model to learn in order to be used in more specific cases.

5.2 Shrinkage Annealing

We have shown in chapter 4 that shrinkage annealing didn't meet our expectations. The question was, "does starting with big step sizes and decreasing them later reduce the number of iterations required for the algorithm to reach an optimal solution while still allowing the algorithm to generalize as well as it would when using small step sizes from the beginning?". We first showed that the step sizes need to decrease fast for the algorithm to resist overfitting. However, while starting with big step sizes evidently speeds-up the convergence, the optimum that is reached with a shrinkage annealing approach is only local and using small step sizes from the beginning leads to more accurate solutions. Finally, we also showed that a compromise for the depth of the trees needs to be used for the algorithm to give enough importance to gradient boosting's fitting process - that focuses on gradient, shrinkage and stochastic sub-sampling - without reducing the ability of the base learners to properly approximate the gradients.

However, we have almost exclusively tested on AstraZeneca's dataset which describes a very complex model with very little correlations: hundreds of possible adverse effects were flagged present or absent for the taking of a single drug; most of them could have been induced by numerous other factors. It is conceivable that shrinkage annealing would be of benefit on simpler datasets where it might be easier for the learner not to become trapped in a local minima.

Chapter 6

Future Work

6.0.1 On Other Loss Functions

The loss functions that are implemented in the GBT package - squared loss and binomial deviance - allow for a straightforward computation of the constants that best fit the data (see section 3.3.3). While this remains the case for some loss functions, there are other loss functions for which an "easy" solution to equation 2.23 is unknown. This equation is thus usually solved by performing a classic line search. We start with a small enough initial guess and increase it by an even smaller step size until we reach a local minimum - that is, when the error starts to increase. Thus, this requires computing the error at each line search iteration, which can become computationally intensive.

A faster method that could be implemented for these loss functions is a parabolic interpolation. It is based on the fact that we cannot indefinitely follow the gradient in the same direction: with a step big enough, the error will rise again. This means that the error first decreases in the direction of the negative gradient and then increases again. Thus, we can find three points A, B and C such that $A < B < C$, $E(A) > E(B)$ and $E(B) < E(C)$, with $E(X)$ being the training error regarding to a step size X. We can then define the parabola that passes through these three points and compute the minimum D of that parabola, which constitutes a good guess for the target optimal step size. Figure 6.1 illustrates this idea.

The problem now is to find A, B and C with as little iterations as possible. A doesn't require any iteration as it is the current point (step size 0). The algorithm

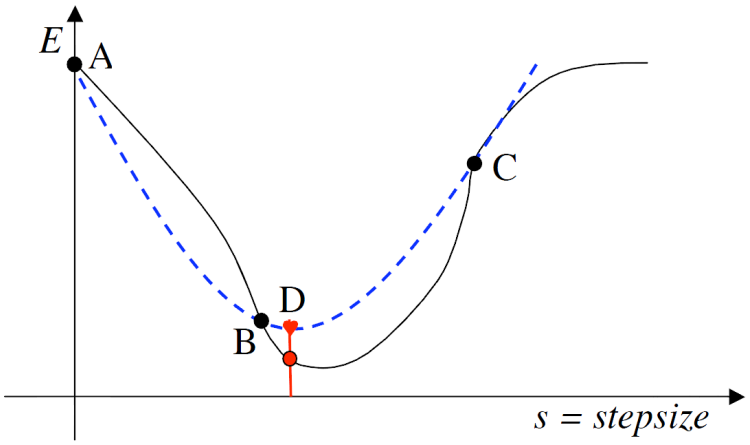

Figure 6.1: Parabolic Interpolation [Bul11]

starts by looking for B such that $E(A) \geq E(B)$ with an initial arbitrary guess. Then:

1. While $E(B) \geq E(A)$ it divides B by 2 to find a small enough B;
2. It takes $C = 2B$. If $E(B) \geq E(C)$ then it multiplies B by 2 and returns to step 2.

This procedure ensures finding appropriate A, B and C. Besides, as it divides or multiply the guesses by two at each step it requires a lot fewer iterations than the classic line search procedure which only linearly increments the guess. Under the constraints $A = 0$ and $C = 2B$, we can compute D, the minimum of the parabola that passes through the three points, with the following formula:

$$D = B\frac{3E(A) - 4E(B) + E(C)}{2(E(A) - 2E(B) + E(C))} \tag{6.1}$$

6.0.2 A Better Descent?

The problem with a gradient descent is that it converges towards a minimum by following a suboptimal "zig-zagging" path in the parameter's space. It can indeed be shown that, with optimal step sizes, the successive gradient directions $\bigtriangledown\mathcal{L}(F_{m-1})$ and $\bigtriangledown\mathcal{L}(F_m)$ are orthogonal to each other. Thus, the idea is to take more direct steps towards the solution to converge faster, which can be done by not following the negative gradient but a compromise between the previous

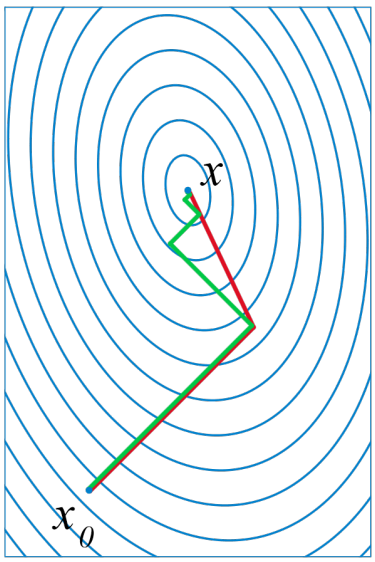

Figure 6.2: Conjugate Gradient (red) vs Gradient (green) descents

descent direction and the current negative gradient. Figure 6.2 illustrates a conjugate gradient descent that reaches the same solution as a gradient descent in two steps instead of five. A conjugate gradient boosting algorithm would fit a base learner h_m on the following descent direction

$$r_m = \beta_m r_{m-1} - \bigtriangledown \mathcal{L}(F_m) \tag{6.2}$$

where β_m defines the proportion of the previous direction that makes the new descent direction. A very popular way to compute β_m is the modified Polak-Ribiere rule [HZ06] given by:

$$\beta_m = max\left(0, \frac{(\bigtriangledown \mathcal{L}(F_m))^T (\bigtriangledown \mathcal{L}(F_m) - \bigtriangledown \mathcal{L}(F_{m-1}))}{|| \bigtriangledown \mathcal{L}(F_{m-1})||^2}\right) \tag{6.3}$$

This method is called conjugate gradient because the descent directions are conjugate regarding a matrix (which is unknown in our case). However, the suboptimality of learning with an incomplete dataset - which will always be the case on a machine learning problem as the dataset will never contain all the information as it doesn't represent all the possible data - slowly destroys this conjugacy. To remedy this problem, the directions must be "reset" from time to time - meaning that they need to correspond to the negative gradient. The modified Polak-Ribiere rule provides an automatic and sensible reset: when the

second argument is negative, 0 is used instead. This is a sensible choice as using a negative coefficient would mean taking a step backward regarding the previous descent direction.

GBT implements a Polak-Ribiere conjugate gradient descent. The user can indicate with a parameter whether he or she wants to perform a classic gradient descent, or a conjugate gradient descent. However, it was observed that:

- On AstraZeneca's dataset, when using the binomial deviance, the Polak-Ribiere coefficient is always negative, which suggests 1) an error in the implementation, 2) a particularity of the dataset, or 3) that it is the result of a property of the binomial deviance. Thus, the tests corresponded to a classic gradient descent.
- On a simpler regression dataset, when using the squared loss, the process seemed to be working correctly.

No further testing has been done and this would need to be investigated.

Bibliography

[BFOS84] Breiman, Friedman, Olshen, and Stone. Classification and regression trees. 1984.

[Bre00] Leo Breiman. Discussion of additive logistic regression: A statistical view of boosting. *The Annals of Statistics*, 2000.

[Buj00] Andreas Buja. Discussion of additive logistic regression: A statistical view of boosting. *The Annals of Statistics*, 2000.

[Bul11] John A. Bullinaria. Learning with momentum, conjugate gradient learning. *Neural Computation : Lecture 8*, 2011.

[Dom00] Pedro Domingos. A unified bias-variance decomposition. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 2000.

[ele09] *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Trevor Hastie and Robert Tibshirani and Jerome Friedman, 2009.

[Fri97] Jerome H. Friedman. On bias, variance, 0/1-loss, and curse-of-dimensionality. *Data Mining and Knowledge Discovery*, 1997.

[Fri01] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 2001.

[Fri02] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 2002.

[GBD92] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 1992.

[GCL] Yasser Ganjisaffar, Rich Caruana, and Cristina Videira Lopes. Bagging gradient-boosted trees for high precision, low variance ranking models. In *Proceedings of the 34rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*.

[HZ06] Hager and Zhang. A survey of nonlinear conjugate gradient methods. *Pacific journal of Optimization*, 2006.

[Jam03] Gareth M. James. Variance and bias for general loss functions. *Machine Learning*, 2003.

[MW07] David Mease and Abraham Wyner. Evidence contrary to the statistical view of boosting. *Journal of Machine Learning Research*, 2007.

[Rid07] Greg Ridgeway. Generalized boosted models: A guide to the gbm package. 2007.

[UN] N. Ueda and R. Nakano. Generalization error of ensemble estimators. In *Proceedings of International Conference on Neural Networks*.

[ZY05] Tong Zhang and Bin Yu. Boosting with early stopping: Convergence and consistency. *The Annals of Statistics*, 2005.

Appendix A

Gradient Boosting and Overfitting

Regarding the original gradient boosting algorithm and overfitting, Hastie, Tibshirani and Friedman [ele09, Fri01] state that the training error can be made arbitrarily small with a number M of iterations large enough. It however appears that this statement is not always true. The problem lies in the fact that we do not directly use the gradient of the loss like in a steepest descent but our best approximation of it from our base learner's class of functions instead (see equation 2.10). Depending on the class of functions, this approximation can be very different from the real gradient which can make the algorithm unable to descend it properly. Here is a simple counterexample to prove it.

Let us restrict our base learners to regression stumps (two terminal nodes trees, also called 1-rule). Let us consider the training samples in ascending order according to a feature p_i for which the values go from p_i^1, the value of the first training sample, to p_i^N, the value of the last N^{th} training sample. Assume that the gradient boosting algorithm has been running for $j-1$ iterations, leading to the following gradient g_j at the step j:

- According to the order on feature p_i, we can divide the training samples in 3 parts, a middle one defined by the range from $p_i^{i_1}$ to $p_i^{i_2}$ $(i_1 \leq i_2)$ that contains less training examples than each of its two surrounding left and right parts, meaning $i_1 \geq i_2 - i_1 + 1$ and $N - i_2 \geq i_2 - i_1 + 1$;
- The gradient is null when evaluated on each training sample from the left

and right parts (i.e. for samples from 1 to $i_1 - 1$ and from $i_2 + 1$ to N), meaning that enough iterations have been performed to perfectly fit the left and right parts of the training data;

- The gradient is not null - let's say 1 for more simplicity - when evaluated on each training sample from the middle part.

In that situation the gradient is not null so there is a possibility to follow it and decrease the training error. However, the decision stump that will best approximate it is null: no matter where the threshold is placed, (between $p_i^{i_1}$ and $p_i^{i_2}$, before or after this range), choosing an output value $v \neq 0$ for the left or the right of the threshold will always result in moving more estimations away from their target values by v than the number of estimations brought closer to their target values by v. Indeed, the left and right parts already reached their target values at iteration $j - 1$ and thus can only be moved away from them, and they contain more samples than the middle part which is the only one for which the estimations can be brought closer to their targets. Thus, any value $v \neq 0$ would increase the training error.

Other examples for which the gradient cannot be followed properly can be constructed, ones for which the gradient approximation is null, and ones for which the changes it induces in the approximations even out. The consequence is that depending on the dataset and the chosen class of functions, gradient boosting can converge towards a suboptimal approximation of the training set. Thus, it might not overfit and early stopping can in fact reduce the accuracy in some rare occasions - which corroborates the vision of Mease and Wyner [MW07].