Universidad Autónoma de Madrid
Escuela Politécnica Superior
Departamento de Ingeniería Informática

Feature selection with Random Forest and Gradient Boosting

Master's thesis presented to apply for the
Master's degree of Investigation and Innovation in Information and Communications Technology

By
Álvaro Alonso Liso

under the direction of
José R. Dorronsoro Ibero

Madrid, September 30, 2016

Contents

Contents **ii**

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Some example problems . . . 1
- 1.3 The formal problem . . . 2
- 1.4 The state of the art . . . 3
- 1.5 Focus of the present work . . . 3
- 1.6 Distribution of the work . . . 4

2 Theory **5**
- 2.1 Supervised Learning . . . 5
  - 2.1.1 Bias Variance Tradeoff . . . 5
- 2.2 Linear Models . . . 6
  - 2.2.1 Penalty methods . . . 7
- 2.3 Trees . . . 8
  - 2.3.1 Building regression Trees . . . 9
  - 2.3.2 Stopping learning and pruning . . . 10
  - 2.3.3 Error estimation and cost . . . 10
  - 2.3.4 CART Trees . . . 11
  - 2.3.5 Building a Tree with *Scikit* . . . 13
- 2.4 Bootstrap aggregating (bagging) . . . 16
  - 2.4.1 Random Forest . . . 17
    - 2.4.1.1 Bias and Variance . . . 17
    - 2.4.1.2 Proximity . . . 18
    - 2.4.1.3 Regularization . . . 18
    - 2.4.1.4 Cost . . . 19
    - 2.4.1.5 Building a random forest with *Scikit* . . . 19
- 2.5 Boosting methods (boosting) . . . 20
  - 2.5.1 Gradient Boosting . . . 20
    - 2.5.1.1 Gradient Boosting problem . . . 21
    - 2.5.1.2 Gradient descent algorithm . . . 21
    - 2.5.1.3 Loss functions . . . 22
    - 2.5.1.4 Regularization . . . 22
    - 2.5.1.5 Cost . . . 23
    - 2.5.1.6 Building a gradient boosting regressor with *Scikit* . . . 23

3 Feature reduction **27**
- 3.1 Dimensionality reduction . . . 27
  - 3.1.1 PCA . . . 27
  - 3.1.2 PCA and SVD . . . 28
  - 3.1.3 Dimensionality reduction . . . 28
- 3.2 Feature selection . . . 29
  - 3.2.1 Formal problem . . . 29
  - 3.2.2 Filter methods . . . 31
    - 3.2.2.1 Univariate . . . 31
    - 3.2.2.2 Multivariate . . . 31
  - 3.2.3 Wrapper methods . . . 32

3.2.3.1 Best Subset Selection 32
3.2.3.2 Forward and Backward Stepwise Selection 33
3.2.3.3 Forward-Stagewise Regression 33
3.2.4 Embedded methods 33
3.2.4.1 Feature selection with Linear methods 33
3.2.4.2 Feature selection with Trees 34
3.2.4.2.1 Feature selection with *Scikit* 36
3.2.4.3 Feature selection with Random Forest 38
3.2.4.3.1 Alternative definition 39
3.2.4.3.2 Feature selection with *Scikit* 39
3.2.4.4 Feature selection with Gradient Boosting 40
3.2.4.4.1 Alternative definition 40
3.2.4.4.2 Feature selection with *Scikit* 41
3.2.4.5 Pruning variables with Trees 42
3.2.4.5.1 Recursive Feature Elimination (RFE) 42
3.2.4.5.2 Boruta 43
3.2.4.5.3 Subset RFE (SRFE) 44
3.2.4.5.4 Comparative 45

4 Experiments 47
4.1 Parameters and methodology 48
4.2 Bike Sharing problem 49
4.2.1 Distribution of feature relevance 50
4.2.2 Pruning the dataset using feature relevance 51
4.2.3 Searching 52
4.3 Wind Energy Prediction problem 54
4.3.1 Distribution of feature relevance 54
4.3.2 Pruning the dataset with FR 55
4.3.3 Pruning 57
4.3.4 Grid Search After Pruning 63
4.4 Conclusions and further work 68

Appendix A Code 71
A.1 Code for experiments 71
A.2 *Scikit* 71
A.2.1 Trees common interface 72
A.2.2 DTR, RFR and GBR interfaces 72

Appendix B Additional results 73
B.1 Quality of Tree feature relevance as a tool to prune 73
B.2 Extended grid 73
B.2.1 BS problem 74
B.2.2 Sotavento 75
B.3 Randomness 76
B.3.1 Stability of the solution 76

List of Figures

2.1.1 Tradeoff between bias and variance, from [1] 6
2.3.1 Partition of space, Tree and Tree surface, from [1] 8
2.3.4 Different Trees 12
2.4.1 Random Forest bias, variance and mse evolution, from [1] 18

3.2.1 Subset selection error for different methods, taken from [1] 32

4.2.1 BS: DTR, RFR and GBR FR distribution 51
4.2.2 BS: Pruning with FR 52
4.3.1 Sotavento: DTR, RFR and GBR (top-down) coefficient distribution 55
4.3.2 Sotavento: DTR, Pruning dataset with FR. Time, train, test and val errors 56
4.3.3 Sotavento: RFR, Pruning dataset with FR. Time, train, test and val errors 56
4.3.4 Sotavento: GBR, Pruning dataset with FR. Time, train, test and val errors 56

A.2.1 *Accelerating Random Forests in* Scikit slides, from G Louppe 71

B.2.1 BS, grid, egrid, DTR, RFR and GBR FR distribution 74
B.2.2 BS: DTR, RFR and GBR, distribution of feature relevance for both grids. 75
B.2.3 Sot, grid, egrid, DTR, RFR and GBR FR distribution 76
B.3.1 Distribution of metric1 (left) and metric2 (right) for permutations of 5 items. 77

List of Tables

2.4.1 Cost for RFR . . . 19
2.5.1 Cost for GBR . . . 23

4.1.1 BS: Grid of parameters for BS problem . . . 48
4.1.2 Extended grid of parameters for BS problem . . . 48
4.2.1 BS: Optimal parameters (grid) . . . 50
4.2.2 BS: Time(s) needed with a pruned dataset to compute the **grid** . . . 53
4.2.3 BS: Time(s) needed with a pruned dataset to compute the **egrid** . . . 53
4.3.1 Sotavento: Optimal parameters for Sotavento problem in the grid (*random state is 0*) . . . 54
4.3.2 Sot, RFE, RFR, $\lambda = 0.5$, error pruning the dataset (%) . . . 59
4.3.3 Sot, RFE, GBR, $\lambda = 0.5$, error pruning the dataset (%) . . . 59
4.3.4 Sot, RFE, RFR, $\lambda = 0.8$, error pruning the dataset (%) . . . 59
4.3.5 Sot, RFE, GBR, $\lambda = 0.8$, error pruning the dataset (%) . . . 59
4.3.6 Sot, RFE, RFR, $\lambda = 0.5$, summary error pruning the dataset (%) . . . 60
4.3.7 Sot, RFE, GBR, $\lambda = 0.5$, summary error pruning the dataset (%) . . . 60
4.3.8 Sot, RFE, RFR, $\lambda = 0.8$, summary error pruning the dataset (%) . . . 60
4.3.9 Sot, RFE, GBR, $\lambda = 0.8$, summary error pruning the dataset (%) . . . 60
4.3.10 Sot, SRFE, RFR, $\lambda = 0.5$, error pruning the dataset (%) . . . 61
4.3.11 Sot, SRFE, GBR, $\lambda = 0.5$, error pruning the dataset (%) . . . 61
4.3.12 Sot, SRFE, RFR, $\lambda = 0.8$, error pruning the dataset (%) . . . 61
4.3.13 Sot, SRFE, GBR, $\lambda = 0.8$, error pruning the dataset (%) . . . 61
4.3.14 Sot, SRFE, RFR, $\lambda = 0.5$, summary error pruning the dataset (%) . . . 61
4.3.15 Sot, SRFE, GBR, $\lambda = 0.5$, summary error pruning the dataset (%) . . . 61
4.3.16 Sot, SRFE, RFR, $\lambda = 0.8$ (%), summary error pruning the dataset . . . 61
4.3.17 Sot, SRFE, GBR, $\lambda = 0.8$, summary error pruning the dataset (%) . . . 61
4.3.18 Sot, Boruta, RFR, 100 iter, error pruning the dataset (%) . . . 62
4.3.19 Sot, Boruta, GBR, 100 iter, error pruning the dataset (%) . . . 62
4.3.20 Sot, RFR, error using the grid, egrid (%) . . . 63
4.3.21 Sot, GBR, error using the grid, egrid (%) . . . 63
4.3.22 Sot, RFE, RFR, $\lambda = 0.5$, search in grid with pruned dataset (%) . . . 64
4.3.23 Sot, RFE, GBR, $\lambda = 0.5$, search in grid with pruned dataset (%) . . . 64
4.3.24 Sot, RFE, RFR, $\lambda = 0.8$, search in grid with pruned dataset (%) . . . 64
4.3.25 Sot, RFE, GBR, $\lambda = 0.8$, search in grid with pruned dataset (%) . . . 64
4.3.26 Sot, SRFE, RFR, $\lambda = 0.5$, $c = 54$, search in grid with pruned dataset (%) . . . 64
4.3.27 Sot, SRFE, GBR, $\lambda = 0.5$, $c = 54$, search in grid with pruned dataset (%) . . . 64
4.3.28 Sot, SRFE, RFR, $\lambda = 0.8$, $c = 54$, search in grid with pruned dataset (%) . . . 64
4.3.29 Sot, SRFE, GBR, $\lambda = 0.8$, $c = 54$, search in grid with pruned dataset (%) . . . 64
4.3.30 Sot, Boruta, RFR, 100 iter, search in grid with pruned dataset (%) . . . 65
4.3.31 Sot, Boruta, GBR, 100 iter, search in grid with pruned dataset (%) . . . 65
4.3.32 Sot, RFE, RFR, $\lambda = 0.8$, number of parameter set which improve the grid . . . 66
4.3.33 Sot, RFE, GBR, $\lambda = 0.8$, number of parameter set which improve the grid . . . 66
4.3.34 Sot, SRFE, RFR, $\lambda = 0.8$, number of parameter set which improve the grid . . . 66
4.3.35 Sot, SRFE, GBR, $\lambda = 0.8$, number of parameter set which improve the grid . . . 66
4.3.36 Sot, Boruta, RFR, 20 iter, number of parameter set which improve the grid . . . 66
4.3.37 Sot, Boruta, GBR, 20 iter, number of parameter set which improve the grid . . . 66
4.3.38 Sot, RFE, RFR, $\lambda = 0.8$, mean and std for the 32 sets of parameters . . . 67
4.3.39 Sot, RFE, GBR, $\lambda = 0.8$, mean and std for the 32 sets of parameters . . . 67
4.3.40 Sot, SRFE, RFR, $\lambda = 0.8$, mean and std for the 32 sets of parameters . . . 67

4.3.41 Sot, SRFE, GBR, $\lambda = 0.8$, mean and std for the 32 sets of parameters 67
4.3.42 Sot, Boruta, RFR, 20 iter, mean and std for the 32 sets of parameters 67
4.3.43 Sot, Boruta, GBR, 20 iter, mean and std for the 32 sets of parameters 67

B.1.1 Sot, RFR, Test error pruning the dataset with optimal parameter. 73
B.1.2 Sot, GBR, Test error pruning the dataset with optimal parameter. 73
B.2.1 BS: time(s) for the grid and the extended grid. 75
B.3.1 BS: Metric description. 77
B.3.2 BS: Stability of the rankings. 100 repetitions. 77

Abstract

The objective of the present work is to analyze the problem which arose naturally working with datasets with a large number of features, which usually forces the data analyst to select a small subset of all the available features to obtain acceptable training times and reduce overfitting. The present work studies the usefulness of the feature importance coefficients given by Trees, Random Forest and Gradient Boosting regressors applied to a problem of wind energy production.

Acknowledgements

The author is kindly supported by the Catedra IIC Modelado y Prediccion grant under the supervision of José R. Dorronsoro Ibero.

Chapter 1

Introduction

1.1 Motivation

Nowadays, a vast amount of raw data is produced and stored daily. With the intention of gaining a deeper understanding of the world, the complex task of analyzing the data can be partially automated. But even with modern technologies, finding patterns and extracting knowledge from the raw data can be extremely difficult because of the nature of the collected and stored material.

It is usual to deal with a large number of samples, each one described with a big number of features, many of them useless in the process of creation of useful information. To simplify the extraction, we consider some samples and some variables as redundant.

Once we have accepted the assumption that some data is not relevant, we obtain some benefits:

- Simplification of models.
- Smaller training times.
- Better generalization.

1.2 Some example problems

A common problem is text classification, such as determining whether a mail is spam or whether a document is written by a certain person. Usually, documents are interpreted as a set of words and represented as an array containing the frequency of each word (usually with certain normalization). Dealing with a range of $2,000$ to $20,000$ words is a common situation.

For example, although the second Edition of the 20 volumes Oxford English Dictionary has *171,476 words* and a more modern version near 300,000 words (including obsolete ones), Shakespeare used only 28,829 different words, and 12,493 appeared only once. The 100 most frequently used terms represent the *53.9% of the words*.

With this in mind, we have a clear understanding of the reason for simplifying our initial representation of a text, selecting only a good subset of words. The main problem is defining what *good* means. For example, we can be almost sure that the top 100 words by frequency are completely useless to determine if a text is part of a Shakespeare's play. But creating a good model for selecting only a small amount of words is not an easy task.

Hence, we are interested in a procedure to automate the selection of only a small fraction of the terms.

Other common problems are related with modern science. New fields like DNA analysis have attracted researchers' interest thanks to the new technology available. In gene selection, the variables studied represent mRNA in a sample (e.g. tissue biopsy from a patient). A usual problem is to classify patients as healthy or not, based on the samples. However, usually only a small amount of samples are available ($< 1,000$) because of the high cost of the analysis, whereas the number of variables extends from 6,000 to 60,000.

More problems appear in modern situations, from fields like medical sciences, city organization and energy production. The problems we are going to analyze are related with these two last points. Our first problem will be related with organizing a **B**ike **S**haring system in a big city (**BS** problem) , trying to predict the number of bikes rented using atmospheric variables and dates. The number of variables used is small and quite easy to interpret, so the problem is useful to understand how feature selection works. Our second problem is **W**ind **E**nergy **P**rediction using atmospheric data (**WEP** problem). We work with extensive maps with atmospheric data, like wind speed and temperature, where probably only a small portion of the data has utility in the process of creating knowledge.

1.3 The formal problem

Each sample from our dataset is described as a vector of features. The *size* of the dataset has two dimensions, the number of instances or samples (N, *rows*) and the number of features or variables (P, *columns*). One of the simplest options to reduce our problem is to remove randomly samples and features. But if the number of samples, N, is reduced without care, we just remove raw data, which can be dangerous for our learning task because of the loss of information about a fraction of the space. Removing randomly a fraction of the total features, P, is not usually a good idea when we deal with a large number of features because of the lack of interpretability of our random prune.

As we mentioned above, the hypothesis which allows the simplification is the presence of redundant data. We have to point that we have two clear sources of *non relevant* data. One origin is the lack of any type of relation between an attribute and the variable to predict (e.g.: including the ISBN number, an international number to identify a book, to predict the price of the book). Other source is the correlation or functional relation of a set of variables, which adds repeated information (e.g. including the price of a vehicle in different currencies with a fixed exchange rate to predict the renting price of the car).

To face the problem of a big P, different techniques have appeared during the last decades. **F**eature **S**election (**FS**), one of them, selects a subset of features, ranking the subset with a value of utility. If the number of columns, P, is reduced the data space is simplified and probably also our model. If we select with care the features to remove, we can obtain almost no loss, as we will observe in our experiments.

Another option to simplify the problem is using dimensionality reduction methods, as **P**rincipal **C**omponent **A**nalysis, (**PCA**), or **S**ingular **V**alue **D**ecomposition, (**SVD**), which extract a combination of features which contain useful information.

Feature selection and dimensionality reduction are completely different. Although both strategies reduce the number of attributes, a dimensionality reduction method does so by generating new variables different to the original ones, whereas a feature selection method removes some variables, but it does not mutate the ones preserved. On the present work, we will focus on feature selection models.

Feature selection ranks the different variables or sets of variables with the intention of removing a fraction of them. If the method focuses on variables, one of the simplest strategies is *classifying* the features as just useful or not depending on certain parameters (binary ranking). Other is a complete *ranking* of the features (creating a total order relationship among them). Using this last focus, we can execute a deep analysis of the evolution of error when we prune the dataset.

Methods like **R**andom **F**orest (**RF**) and **G**radient **B**oosting (**GB**) generate a ranking of relevance for the variables, while regressors like Lasso generate a natural binary selection. Initially, we can think that a complete ranking is a better result than just a binary selection. However a complete ranking is not always a good option. For example, we can have two variables highly correlated among them and with the variable to predict which will be ranked as the best options with a *naive* filter method. With this configuration, taking both variables will give us a good result, but just one is a simpler option. Hence, we should analyze more factors, like the interaction among variables, the evolution of the error estimation or a complexity limit for our model.

Once we have a better idea of the problem in our hands, we can state the goal of feature selection:

The study of subsets of features and the measure of its utility for a regression (or classification) problem.

1.4 The state of the art

Depending on the number of variables involved in the process of selecting a good subset and the way they are used, we can classify the methods in three groups:

- **Filter methods:** Select **individual** variables, usually with a *naive* strategy and independently of a predictor. Some examples are filtering using a minimum variance threshold value or the correlation with the prediction variable.

- **Wrapper Methods:** Select **subsets** of variables. Hence, they account for the interaction among features. They use an external model to rank the subsets. An example is stepwise regression with a Linear regressor.

- **Embedded Methods: Mix** both previous models. The algorithms themselves build a variable importance ranking. Some of the most popular options are Trees and ensemble methods like RF and GB.

There are many software packages which have been found quite useful to work with feature selection analysis, but we can point specially three of them:

- **Weka:** In Java, developed by University of Waikato. GNU GPL license.

- **Scikit-Learn:** In Python, by INRIA, Google and others. BSD license.

- **R packages:** Language with good community of developers, core by R Development Core Team. GNU GPL license.

We will use Python and Scikit-Learn (from now on *Scikit* or *Sklearn*).

1.5 Focus of the present work

The focus of the present work is to analyze the problem of big dimensionality with feature selection using RF and GB feature relevance, comparing the results with other techniques and comparing different strategies to find a good subset of features. The project has a mainly practical orientation motivated by the necessity arose previously in a problem of wind energy prediction. Hence we will work with a wind energy dataset, later described. Our goals are:

- Obtain a good understanding of Trees, RF, GB and their utility.
- Obtain a clear understanding about feature importance with Trees, RF and GB.
- Analyze the distribution and stability of feature relevance for the dataset and its utility.
- Analyze the feature selection simplification in the search of optimal parameters.
- Analyze the evolution of the error pruning the dataset.
- Find a good strategy for pruning automatically the dataset features.

1.6 Distribution of the work

This **first chapter** has pretended to be a **brief summary of the problem** faced in the following ones, including a statement of the problem, some examples, a formal description and the focus of the present work.

The **second chapter** contains some **basic regression theory** using **Linear Methods, Trees, Random Forest and Gradient Boosting**. It does not pretend to be a complete revision of all the relevant points related to these techniques, but just a resume of basic theory and useful points which are necessary for understanding feature selection with them. Besides, it illustrates the theory with a clear explanation of the details of implementing each method with a modern package like *Scikit*.

The **third chapter** presents a basic introduction to **dimensionality reduction** methods, basically *PCA*, and a detailed one for **feature selection** for the predictors explained in chapter two. From a practical point of view, we review different techniques from the three previous approaches to feature selection described above and we focus on the classical approximation with Linear Methods, Trees, Random Forest and Gradient Boosting, illustrating the techniques with the *Scikit* implementation as we did in the theory chapter.

The **fourth chapter** contains a **description of the datasets, the experiments and the results**. We introduce the datasets for both the previous stated problems and we obtain some results for the goals described above. We initially face the simple *BS* problem to exemplify the procedure and compare the obtained results with what we expect. It will be also useful to illustrate some topics about feature relevance, Trees and its ensemble methods. Later we confront the *WEP* problem answering similar questions.

The final chapter is the **Appendix**, which contains practical and useful information, like the interfaces of the *Scikit* classes used, the repository where the code can be found and other details.

Chapter 2

Theory

The theoretical structure of this chapter follows partially the structure used in [1] to present the topics. Although the chapter covers basic theory well established, some references have been found useful. For Trees, reference [3], for Random Forest, references [2], [15], [4], and for Gradient Boosting, references [10], [11] and [9].

The chapter uses also code from *Scikit* repository to illustrate the process of building the different data models.

2.1 Supervised Learning

The problem of supervised learning can be stated in a simple and concise way:

Given a matrix $\mathbf{X}$*, make a* ***good*** *prediction (*$\hat{\mathbf{Y}}$*) of a vector* $\mathbf{Y}$*.*

The typical problem consists of a dataset of raw data and an explanation of our goal. With just these ingredients, we have to obtain a good model for prediction.

The selection of our model will be based on the nature of the problem, the assumptions we make about the data and the results we obtain with a test (or validation) dataset. To compare our models, we normally split our dataset in a training and a testing dataset (sometimes the division is given with the problem), although other options, like for example cross validation, can be applied if we do not have many samples.

2.1.1 Bias Variance Tradeoff

One of the main problems after fitting a model is the evaluation of the model. Normally, we should use a strategy to avoid overfitting. As stated above, from a practical point of view, we can use a train dataset to fit our model and a test or validation dataset to check the quality of our model and avoid a strong fit to the train dataset.

From a theoretical point of view, we can describe how a complex model can produce overfitting easily. For simplicity, we suppose we have a functional relation, $Y = f(X) + \epsilon$, being ϵ an error with $E(\epsilon) = 0$ and $Var(\epsilon) = \sigma^2$. Then, the prediction error at x can be expressed for $\hat{f}_k$, an approximation to f, as:

$$EPE_k(x) = E[(Y - \hat{f}_k(x))^2 | X = x] = \sigma^2 + [Bias^2(\hat{f}_k(x)) + Var(\hat{f}_k(x))]$$

$$Bias_f(\hat{f}) = E[\hat{f}] - f, \quad Var_f(\hat{f}) = E[(\hat{f} - E[\hat{f}])^2]$$

From the three elements in this expression, the first is a constant whereas the second and third, which sum the **M**ean **S**quared **E**rror (**MSE**), depend on our model.

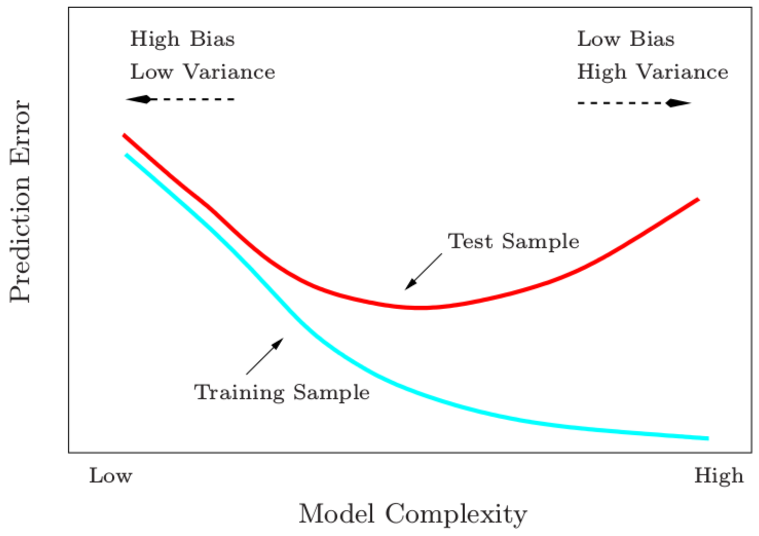


Figure 2.1.1: Tradeoff between bias and variance, from [1]

As a general result, when the complexity of the model grows the plain error (bias) decreases but the noise (variance) increases. The figure 2.1.1 represents this problem. For example, Trees are prone to overfitting if we do not control their depth.

2.2 Linear Models

With a linear model we predict a variable Y using a vector $X = (1, X_1, \dots, X_P)$, using a model

$$\hat{Y} = \beta_0 + \sum_{j=1}^{P} X_j \beta_j$$

The variables X_j used can be original variables from the dataset, but we can also build new variables using a basis expansion of the original variables($\bar{X}_1 = X_1^2$), adding interaction between variables ($\bar{X}_2 = X_1 X_2$) or another transformation of the variables ($\bar{X}_2 = \log(\max(X_2, 1))$). So we assume that the model is just linear on the parameters β_j.

The simplest and most common strategy to compute the coefficients is the method of **O**rdinary **L**east **S**quares, (**OLS**), which selects the coefficients β which minimize the sum of squares of the differences.

$$RSS(\beta) = \sum_{j=0}^{N} (y_j - x_j \beta_j)^2$$

If we have now $X^1, \dots, X^N$ training data vectors and we build a matrix $\mathbf{X}$, of dimensions N x $(P+1)$ [1] , and a vector to predict $\mathbf{Y}$, of size N x 1, we can write

$$RSS(\beta) = (\mathbf{Y} - \mathbf{X}\beta)^T (\mathbf{Y} - \mathbf{X}\beta)$$

with unique solution, if $\mathbf{X}^T\mathbf{X}$ is invertible,

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

[1] We add a 1 in the first position for each X^j

We can consider our estimated $\hat{\mathbf{Y}}$ as the orthogonal projection of $\mathbf{Y}$ onto the space generated by the columns of $\mathbf{X}$, $Col(\mathbf{X})$, because our problem, $\hat{\beta} = \arg\min_\beta ||\mathbf{Y} - \mathbf{X}\beta||^2 = ||\mathbf{Y} - \hat{\mathbf{Y}}||^2$, finds the best linear combination of a component of $\mathbf{Y}$, $\mathbf{X}\hat{\beta}$, in $Col(\mathbf{X})$.

If $\mathbf{X}^T\mathbf{X}$ does not have an inverse, OLS coefficients are not unique.

2.2.1 Penalty methods

Sometimes we are also interested on biasing the solution of a Linear Model towards some other solutions, like preferring small coefficients. One easy way to change the behaviour is adding a constraints or a penalty on the size of the coefficients of the model.

$$PRSS(\beta; \alpha) = RSS(\beta) + \alpha J(\beta)$$

$J(\beta)$ shapes the behaviour of the estimator and for $\alpha = 0$ no penalty is imposed (normal OLS). Usual elections for $J(\beta)$ are *Lp like* norms: $J_p(\beta) = \sum |\beta|^p$. For $p = 1$ we obtain Lasso whereas for $p = 2$ we obtain Ridge regression.

Ridge regression is a type of penalty method. It uses L2 penalty. The problem to solve is

$$\hat{\beta} = argmin_\beta \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{P} x_{ij}\beta_j)^2 + \delta \sum_{j=1}^{P} \beta_j^2$$

which is an equivalent problem to $\hat{\beta} = argmin_\beta \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{P} x_{ij}\beta_j)^2$ subject to $\sum_{j=1}^{P} \beta_j^2 \leq t$.

The method depends on the scale so we need to standardize the original dataset. In matrix form, after centering the columns (on zero), we have:

$$RSS(\delta) = (\mathbf{Y} - \mathbf{X}\beta)^T(\mathbf{Y} - \mathbf{X}\beta) + \delta\beta^T\beta$$

$$\Rightarrow \beta = (\mathbf{X}^T\mathbf{X} + \delta I)^{-1}\mathbf{X}^T\mathbf{Y}$$

If we have orthonormal inputs, ridge coefficients are shrunk OLS coefficients: $\beta^{rdg} = \frac{\beta^{ols}}{1+\delta}$. Hence Ridge just reduces the size of the coefficient, but it does not remove variables.

Lasso regression is similar to Ridge, but the penalty is L1. The problem to solve is

$$\hat{\beta} = argmin_\beta \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{P} x_{ij}\beta_j)^2 + \delta \sum_{j=1}^{P} |\beta_j|$$

which is equivalent to $\hat{\beta} = argmin_\beta \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{P} x_{ij}\beta_j)^2$ subject to $\sum_{j=1}^{P} |\beta_j| \leq t$. When t is bigger than the sum of OLS coefficients, $t_0 = \sum |\beta_j^{ols}|$, then the Lasso behaves as the normal OLS. For the opposite situation, Lasso reduces the coefficients or sets them to zero. For example, for $t = \frac{t_0}{2}$ the OLS coefficients are shrunk by about 50% more or less or are set to zero.

The principal difference between Lasso and Ridge is that in the former increasing the penalty will move a bigger number of the parameters to zero whereas in the latter all parameters are reduced but are still non zero. This is an advantage of Lasso as setting parameters to zero removes features. Hence we can consider that Lasso does a natural binary feature selection.

2.3 Trees

Trees partition the space of the data, R, into a set of *boxes* and a set of constant as a predicted value in each region. The most common strategy to build a Tree is a greedy recursive binary partitioning approach.

We initially split the space into two disjoint subsets, using a variable and a threshold value to get the *best split* [2]. Later this same strategy is used with the resulting subsets until some finishing rule can be applied. At the end of this process we get a partition of the initial space, R, into a collection of sets $R_1, R_2, \ldots, R_h$. The model uses a constant c_m in region R_m as predicted value, which is usually computed as the average value (regression) or the most common label (classification) from all the training samples which belong to this region. The formal model can be written as a sum with an indicator function for each region:

$$f(X) = \sum_{m=1}^{h} c_m I[X \in R_m]$$

Figures 2.3.1 show an example of a partition.

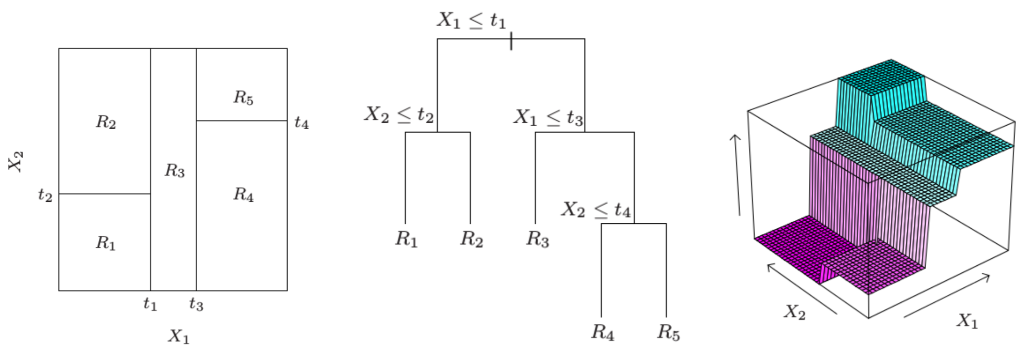


Figure 2.3.1: Partition of space, Tree and Tree surface, from [1]

In general terms, Trees are quite common as learning strategy and present some advantages with respect to other models like Linear Models or SVM, such as

- Easy to understand, interpret and visualize.
- They work with categorical and numerical data.
- Great model and not many assumptions.

However they face several drawbacks as

- Overfitting.
- Lack of stability (solved with ensembles).
- It can produce some bias (depending on the building criteria).

[2]Now, the hard point is defining *best*.

2.3.1 Building regression Trees

Algorithms for Trees work most of the times with a greedy top down approach. At each step, they choose a variable and threshold that *best* split the data. The meaning of *best* depends on the algorithm, but a measure of homogeneity inside each new subset is used.

There are several functions to select the *best* partition. Some typical are:

- **ID3 (or C45):** Uses entropy and information gain. We will describe it briefly later.
- **CART:** Uses square error (regression) or Gini coefficient (classification). This method is the one we are going to describe and use in deeper detail.
- **CHAID (Chi-squared Automatic Interaction Detector):** Uses statistical tests and multilevel branching.
- **Conditional Inference Trees:** Uses non parametric tests for the splitting threshold.

Perhaps the most common strategy to build a decision Tree is **CART**. It is the one we are using on the experiments. The algorithm decides the splitting variables and points. For regression it uses as criteria the usual minimization of squares metric, whereas for classification the most common option is the Gini impurity measure. For regression, in each region, R_m, the predicted value, c_m, is just the average of the values y_i in the region while for classification c_m is the most common class.

CART obeys the following strategy. Initially, we consider all the data, a splitting attribute j and split point s. With these, we define the partition

$$R_1(j,s) = \{X|X_j < s\}, R_2(j,s) = \{X|X_j > s\}$$

Then we search the values which solve

$$min_{j,s}[min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2]$$

The cost for determining the best tuple (j, s) can be reasonable, although high because it requires a complete computation for all the features and possible splitting points belonging to a grid. Usually the problem is simplified and the unique splitting points considered are middle points between two consecutive values in the training set. After finding the best split, we partition the dataset distributing the samples and we continue with the process until some stopping criterion can be applied. We will analyze with detail CART and its properties in the following sections.

Other common algorithm is **ID3** (**I**terative **D**ichotomizer **3**), by Ross Quinlan, or the improved version C4.5/C5.0. Initially, ID3 starts with the whole dataset S at the root node. For each variable, it computes the entropy $H(S)$ and selects the attribute and split point which divides this dataset minimizing the new entropies for each subset (or maximizes the information gain, $IG(A, S)$). The procedure is repeated with each descendant node and every free attribute (non used in an ancestor node). The regression implementation of the Tree is usually binary and the algorithm considers each possible split of the form $x_j < t_j$, where as with CART the values t_j are normally the points between two consecutive values from the training dataset. For classification, the Tree usually selects among all the options of each feature at each node.

Entropy and information gain are defined as:

$$H(S) = \sum_{x \in C} p(x) log_2 p(x)$$

$$IG(A, S) = H(S) - \sum_{t \in T} p(t)H(t)$$

with S the dataset, C all the classes of S, $p(x)$ the fraction of elements of class x, T the collection of sets after the split, $p(t)$ the proportion in each subset t of T, and $H(t)$ the entropy of t.

C4.5 algorithm improves ID3 adding certain features such as, for example, working natively with continuous variables, missing values or allowing assigning a cost value to each sample. C4.5 allows also the option of simplifying the obtained predictor pruning the Tree. C5.0 improves C4.5 reducing the size and complexity of the Trees without hurting the performance, adding a misclassification costs or adding new strategies for dealing with dates/times/timestamps.

2.3.2 Stopping learning and pruning

Until now we have been talking about building a Tree assuming as a natural finalizing criteria achieving a node with just one element (a leaf) or with several elements with the same value.

However, this strategy will generate a Tree adapted to the training set which will not generalize well with new elements. On the other hand, a simple Tree like a stump will not have power enough to deal with complex problems.

To **stop the growth** of a Tree we must define and adjust the parameters governing the growth with certain intuition. There are several criteria to stop the growth of Trees, being usual options maximum depth or minimum number of items per leaf or decision node. We have also other approaches less common and related to the reduction of the error, as splitting Tree nodes only if the decrease in error or impurity is above some threshold. In the Appendix, we attach information about the parameters used in the Tree construction process with *Scikit*.

Another way of controlling the size or complexity of the predictor is simplifying it after the building process. Once a Tree is built, we can be interested in removing some branches or converting some internal nodes into leaves (including all the nodes assigned to each child into the new leaf).

In general terms, we can consider two strategies for **pruning**. With a top down approach, we remove complete branches collapsing an internal node, which is not the parent of at least one leaf, into a leaf. With a bottom up strategy, we consider each parent of a leaf and we compute the increase of error due to the collapse of the node, removing the option which less error. In a similar way to the growth process, we should fix a criterion to stop the pruning process.

2.3.3 Error estimation and cost

The process of building a Tree is based on an iterative process of reducing an error function at each step. Hence, one of the most natural decisions to **obtain an error measure** is using the error function of the building process as the estimation. Then, to compute the error, we can use the estimated error of the building process. However, this usually leads to an optimistic estimation.

To avoid this, we can split the original dataset into a training and test datasets and use the test set to estimate, with the same error function or a different one, the predictor error. For example, for a regression problem we can query each element of the test dataset and compute the *mean value error* (sum of absolute differences) or the *mean squared error* (sum of squared differences).

However, for small datasets this can mean a great loss of useful information. One easy option is using cross validation to obtain a good estimation of the predictor error. Other option, available if we are going to train several Trees, which is the core of ensemble methods, is using the *out of bag* (*oob*) error.

If we use bootstrapping, *oob* error can be computed for a sample $z_i = (x_i, y_i)$ building an ensemble predictor by including only those Trees built with training datasets which did not include z_i. *oob* error gives us a good estimation of the true error of the learned estimator, allowing us to use all the data for training the ensemble predictor and *oob* error to estimate the optimal parameters.

One of the most important issues related with a data structure, like a Tree, is the **cost of building** and querying the model (and perhaps updating). Estimating these values is difficult, because they will depend on many parameters related with the form of the Tree. However, we can estimate some values making certain assumptions.

If **we assume the binary Tree is balanced** (or approximately balanced) the time cost of **querying** a value will be $O(\log_2(N))$, because for N total samples we will need only $\log_2(N)$, at worst, decision nodes to identify an element.

The time cost of **building** a Tree will depend on the method and stopping criteria. For a full CART Tree we are minimizing the squared error of partitioning the data in each feature and in each possible partition (in total $O(PN\log_2(N))$ in each node [3]. And we are doing this at most the number of decision nodes in a Tree, which is $O(N)$. Hence the total cost will be $O(PN^2\log_2(N))$. With certain optimization, as sorting initially the values for each feature and removing not necessary computation, the cost can be reduced to $O(PN\log_2(N))$ [4].

With respect to implementation and memory cost, we cannot obtain general solutions as they will depend completely on the details of the implementation. For querying, we just need to hold a reference to the node of the Tree we are using to compare with our element. For building, with a struct and pointer implementation we will need $O(N)$ times the memory cost of each struct and (probably) continuous calls to the system for a new node. With an array implementation and intelligent memory prefetch (strategy used in *Scikit*), we can build a Tree in the same array just with references for the children. We will see some details about a good implementation with *Scikit* below.

2.3.4 CART Trees

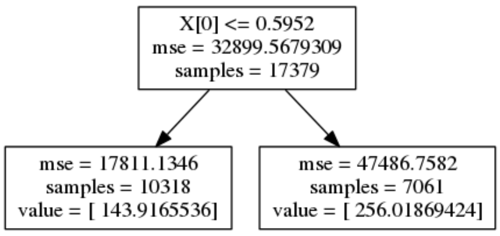
Until this moment we have been talking about CART Trees as if they were ghosts, because we have not seen any of them. With certain criteria, we should ask ourselves how a CART Tree is. We are going to use our BS problem to show same examples about Trees built with CART strategy. We are using *Graphviz* for each representation. We use the parameters defined as optimal in the BS problem (details given in the chapter for experiments) and we employ the whole dataset for training.

The figures (2.3.2a), (2.3.2b), (2.3.2c), (2.3.2d) represent a Tree built with (maximum) depth 1, 2, 3 and 4 respectively. Each box of the Tree represents a node. Both internal nodes and leaves contain as fields the measure of error used to build the Tree (*mse*) and the number of samples contained.

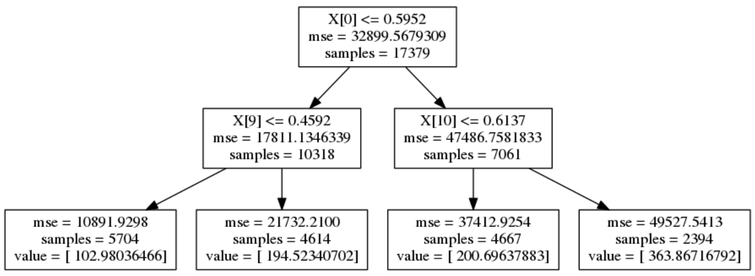
Non terminal nodes, internal, contain also the split criteria used (the number of samples refers to the number contained in each of the derived leaves) whereas terminal nodes, leaves,

[3]For each feature, we will need $O(N\log_2(N))$ *to order the list of N values* . As we have P features, we will need $O(PN\log_2(N))$ for a complete search on each feature and each sample item.

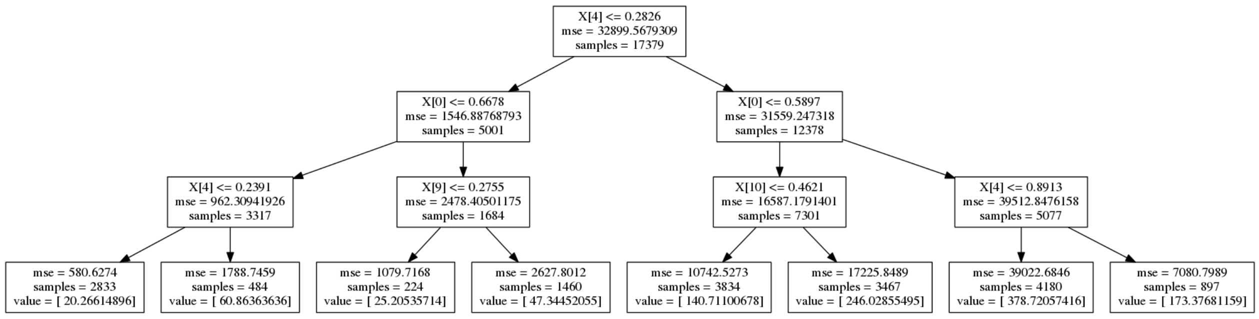
[4]Scikit documentation includes a good summary of the main details for Trees.



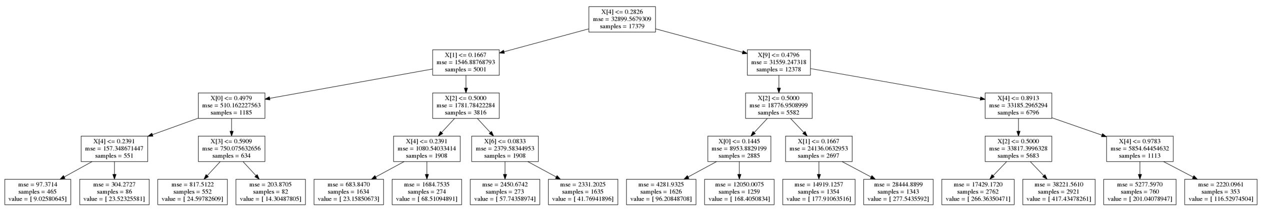
(a) Tree of depth 1



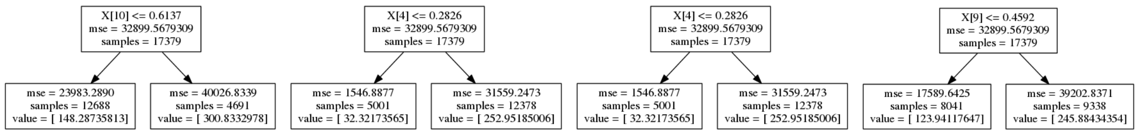
(b) Tree of depth 2



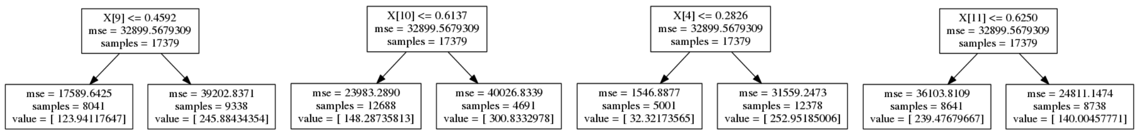
(c) Tree of depth 3



(d) Tree of depth 4



(a) Different roots



(a) Different roots

Figure 2.3.4: Different Trees

contain also the estimated value for the node (the number of samples refers to the number of samples of the training set).

The first thing we observe is that Trees are complete when the depth is the stopping criterion, so our estimations about cost are precise. In general terms, for problems with a large number of items the parameter depth is usually the stopping criterion. We should also restrict the value if we obtain a quite large Tree (overfitting).

The second thing we observe is the randomness. As we observe, the splitting criteria is different for each Tree. Although it is also clear that some features appear with more frequency and with higher positions in the Tree, we cannot assure which one will be taken. The figures (2.3.3a) and (2.3.4a) show us than even the root node can be different. In general terms, the process of building a Tree involves two different sources of randomness. The first one is the nature of the training set because sometimes we just use a random subset of all the dataset to train and the remaining part for testing purposes (or we use bootstrap in a bagging method). The second source is the number of features to consider when searching for the best split at each node (as in Random Forest). We can avoid both sources using always the same dataset for training and all the features when computing a splitting node. Another option is allowing random options, but using and documenting the seed of the random generator. We have used this second option. However despite Decision Trees Regressors suffers from certain lack of stability as estimators, Random Forest Regressors and Gradient Boosting Regressors are robust and the results obtained will be stable.

2.3.5 Building a Tree with *Scikit*

Scikit code can be quite useful to illustrate the process of building a Tree. Unluckily, the recent code from the repository includes some extra complexities as the use of `Cython` to achieve a good performance in real environments, making a bit more difficult to follow the logical process of building a Tree. However previous versions of the Tree building process (commit (`a4d4bb6`) and previous) are much simpler. The following code, taken from the *public repository of Scikit* and corresponding to an early commit (`a4d4bb6`) for the class Tree, shows us the simplicity of the process.

The Tree building process is quite simple. *DecisionTreeRegressor* is derived from *BaseDecisionTree*. When *DecisionTreeRegressor*, the public class, is called, it initializes some variables and calls *BaseDecisionTree* to do the same. *BaseDecisionTree* has a fit method which builds a Tree instantiating a *Tree* class which contains the core build function. Only the *Tree* class, which we attach partially, is necessary to understand the process of building a Tree.

We explain step to step the process of building a Tree. Some parts of the code have been striped for the sake of brevity.

```
class Tree(object):
    """Struct-of-arrays representation of a binary decision tree.

    The binary tree is represented as a number of parallel arrays.
    The i-th element of each array holds information about the
    node i.
    ----------
    node_count : Number of nodes (internal nodes + leaves) in the tree.
    children :
        children[i, 0] holds the node id of the left child of node i.
        children[i, 1] holds the node id of the right child of node i.
        For leaves children[i, 0] == children[i, 1] == Tree.LEAF == -1.
    feature : The feature to split on (only internal nodes).
    threshold : The threshold of each node (only for internal nodes).
    value : Contains the constant prediction value of each node.
    best_error : The error of the (best) split.
        For leaves init_error == best_error.
    init_error : The initial error of the node (before splitting).
        For leaves init_error == best_error.
    n_samples : The number of samples at each node.
    """
    def _add_split_node(...):
        """Add a splitting node to the tree. The new node registers itself as
        the child of its parent. """

    def _add_leaf(...):
        """Add a leaf to the tree. The new node registers itself as the
        child of its parent. """

    def build(...):
        # Recursive algorithm
        def recursive_partition(...):
```

As stated in the docstring, *the binary Tree is represented as a number of parallel arrays* in which the rest of the methods operate (previous versions used a simpler structure with Leaf, Node and Tree classes, but the code was ported to arrays for performance reasons and this version is easy to understand and gives good details about a practical implementation). The main function is *build*, which has a function *recursive_partition* nested. *build* method calls *recursive_partition* with the root node of the Tree.

The code for *recursive_partition* is also easy to understand:

```
        def recursive_partition(...):
            # Split samples
            if depth < max_depth and n_node_samples >= min_samples_split
               and n_node_samples >= 2 * min_samples_leaf:
                feature, threshold, best_error, init_error = find_split(...)
            else:
                feature = -1
                init_error = _tree._error_at_leaf(...)
            value = criterion.init_value()
            # Current node is leaf
            if feature == -1:
                self._add_leaf(...)

            # Current node is internal node (= split node)
            else:
                # Split and and recurse
                split = X[:, feature] <= threshold

                node_id = self._add_split_node(...)

                # left child recursion
                recursive_partition(...)

                # right child recursion
                recursive_partition(...)
```

The initial steps are determining if the node is going to be a leaf or an internal node. Later for the former situation creates a leaf and for the latter splits the data using the threshold and applies *recursive_partition* to each of the new partitions.

For performance reasons, the function which computes the split, *find_split*, is implemented in `Cython`. The code is not so easy to follow because it is a low level language (mix of `C` and *Python*) and uses heavily `C` pointers. We attach a simplified version of the `Cython` code.

```
def _find_best_split(...):
    """Find the best dimension and threshold that minimises the error."""
    # break early if the node is pure
    ...
    # Features to consider: all or just a fraction?
    ...

    best_error = initial_error

    # Look for the best split
    for feature_idx from 0 <= feature_idx < max_features:
        # Get i-th col of X (X_i) and X_sorted (X_argsorted_i)
        ...

        # Consider splits between two consecutive samples
        while True:
            # Better split than the best so far?
            ...

            # Only consider splits that respect min_leaf
            ...

            error = criterion.eval()
            if error < best_error:
                # Threshold is the middle point
                t = X_i[X_argsorted_i[a]] + \
                    ((X_i[X_argsorted_i[b]] - X_i[X_argsorted_i[a]]) / 2.0)
                best_i = i
                best_t = t
                best_error = error
                ...

            # Proceed to the next interval
            ...

    return best_i, best_t, best_error, initial_error
```

In general terms, the *find_best_split* implementation of the function *find_split* makes an exhaustive search through all the possible splits (first through each feature and later for each possible value of each feature) to get the best one. Possible values for each feature are the middle point between two consecutive item values. We should note the existence of another *find_split* function, *find_best_random_split*, which adds certain randomness to the computation of the best threshold value. Code can be consulted in the `_tree.pyx` file of the repository.

The last point to observe is the estimation of the error of a split, *criterion* function, and the values of both initial error and best error. The following *MSE* `Cython` class, derived from *RegressionCriterion*, computes the impurity value of a split.

```
cdef class MSE(RegressionCriterion):
    """Mean squared error impurity criterion.

    MSE = var_left + var_right
    """
    cdef double eval(self):
        ...
        for k from 0 <= k < n_outputs:
            total += var_left[k]
            total += var_right[k]

        return total / n_outputs
```

As we observe, the function just adds the MSE of the target points at each side of the split point. The *docstring* of the class *RegressionCriterion* gives us a clearer information about the process, giving us the expression for the left and right variance, which is what we expected.

```
cdef class RegressionCriterion(Criterion):
    """Abstract criterion for regression. Computes variance of the
       target values left and right of the split point.

    Computation is linear in `n_samples` by using

```

```
        var = \sum_i^n (y_i - y_bar) ** 2
            = (\sum_i^n y_i ** 2) - n_samples y_bar ** 2
    """
```

The description included above contains all the details for understanding a real and practical implementation of Tree construction. Modern implementation of *Scikit* is similar, but most of the code has been ported to `Cython`.

2.4 Bootstrap aggregating (bagging)

Bootstrap aggregating, also abbreviated *bagging*, is an ensemble strategy to **improve the stability and precision** of other learning algorithms decreasing both variance and over-fitting [5].

Bagging uses base estimators to build another estimator, inheriting some properties from the base estimators and improving some weakness from them. The methodology uses some randomness, like bootstrap or random split points, to build different estimators and reduce the dependence among them. Different bagging methods use different strategies to compute the new predictors.

For example, Random Forest uses bootstrapping and selects a random subset of available features at each splitting point for searching the best option. We will describe the method deeper later. We have also other techniques less common like perfect Random Forest [6] , which selects randomly the variable and splitting point, or *TWIX* [7] , which builds an ensemble predictor not creating completely new Trees, but copying the current Tree at each splitting node and using each reasonable option. Reference [4] contains a good summary of bagging and more references for the mentioned methods.

In general terms, each base estimator of a bagging predictor adds a value. For a classification problem, the bagging estimator returns the class with the majority of the votes. For regression, it returns an average value.

So, **why does it work?**. Modern human societies are complex social structures in which people have social and economic interactions. Our *homo economicus* soul drives our lives towards attempts to maximize our utility and our profit, usually confronting other humans. As a consequence, political economics is at the heart of any modern society, pretending to substitute the primitive war for resources for a new war of words for resources. During the last pair of centuries, we have seen then end of absolutism and enlightened despotism, being democracy crowned as the new state of the art technology in human organization. In parallel, the world has growth and improves as never before. Why?

Democracy is a type of human bagging. If we assume that a group of independent people has to decide between two options, only one of them is correct and that each voter has a probability greater than 0.5 of taking the correct decision, the aggregate result will converge to 1 as the number of independent voters is increased [8] [9]. In general terms, Condorcet's theorem applies to Bernoulli random variables. Then, for example, if we have a classification problem and we have a dataset with n samples, once we have fixed the Tree parameters (so we have a success rate of $p > 0.5$ for each Tree), Condorcet's theorem gives

[5]Bagging was presented in 1994 by Breiman to improve the prediction accuracy by aggregating simple estimators.

[6]Perfect Random Forest by Cutler et al.

[7]Trees WIth eXtra splits (TWIX) by Potapov et al.

[8]Condorcet's jury theorem.

[9]Quite strong assumption in many cases. A more real approach is the one described by Arrow's impossibility theorem being not so optimistic.

us the probability of a correct decision by majority, $P(n,p)$, for $p \approx \frac{1}{2}$ and $p > \frac{1}{2}$:

$$P(n,p) = \frac{1}{2} + a(n)(p - \frac{1}{2}) + O\left((p - \frac{1}{2})^3\right) \approx \frac{1}{2} + k\sqrt{n}(p - \frac{1}{2})$$

where $a(n) = k\sqrt{n} + O(\frac{1}{n^{3/2}})$.

To **estimate the error** we can use any metric of error from the base estimators. We just need to aggregate the error generated by each one. In our situation, we will use Trees as predictors and we will aggregate them with equal weight (Random Forest). As we have mentioned above, *out of bag* error is a quite useful technique to obtain an estimation of the error allowing us to use the whole training set for the ensemble method.

2.4.1 Random Forest

Random Forest is a bagging estimator built with Trees as base estimators. It builds and averages a large collection of non strongly correlated Trees. In general, bagging works well for high variance and low bias base methods as Trees, so Random Forest has good properties and performance. For a regression problem, we fit many regression Trees to a (perhaps bootstrap versions of) the train data and average the result.

Then, for regression, we use the usual predictor [10]

$$f_{rf}(x) = \frac{1}{M}\sum_{k=1}^{M} T(x;\theta_k)$$

For classification, Random Forest chooses the class with the majority of the votes.

Despite a Random Forest has many parameters to tune, it can be used with good results with a default or common sense selection of parameters. This simple use makes Random Forests a great tool, allowing to obtain decent estimation with an easy to interpret and understand methodology.

2.4.1.1 Bias and Variance

If we fix a distribution for generating the bootstrapped training set and the random selection of features at each internal node, each Tree is i.d. (identically distributed), and then the expectation of a Random Forest is the one of any Tree. The figure 2.4.1 shows the variance, bias and *mse* evolution for a typical forest, as the number of estimators growth.

Formally, Random forests *cannot* overfit the data. However we should avoid using a large number of estimators, as it increases the computational and memory cost.

Trees have high **variance**, so a bagging strategy greatly improves the performance. From a formal point of view, an average of M i.i.d. (independent and identically distributed) Trees, with individual variance σ^2, has variance $\frac{1}{M}\sigma^2$. However, the Trees for Random Forest are built only identically distributed but with correlation, ρ, so the variance of Random Forest is [11]

$$\rho\sigma^2 + \frac{1-\rho}{M}\sigma^2$$

From a practical point of view, when we train many Trees (M increases) only the first term survives. Hence, the variance depends on the pairwise correlation, ρ. Random forest decreases the variance reducing the correlation using two strategies:

[10] $T(x;\theta_k)$ is a Tree predictor with parameters $\theta_k = \cup_j^{J_k}\{(R_j^k, \delta_j^k)\}$, i.e $\{R_j^k\}_1^{J_k}$ is partition of the space and we have constants $\{\delta_j^k\}_1^{J_k}$ as predicted value given by the Tree in the region.

[11] From [1], section 15.2.

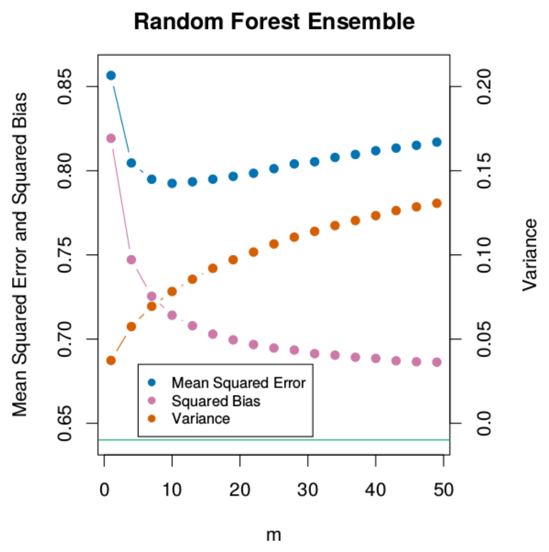


Figure 2.4.1: Random Forest bias, variance and mse evolution, from [1]

- **First**, Random Forest does random selection of the picked variables at each internal node of the Tree during the building process. It selects $m \leq P$ variables randomly as options for splitting. Typical values for m are $\sqrt{P}$, or $[\frac{P}{3}]$.
- **Second**, Random Forest uses a (different) subset of the training set for each Tree (bootstrap). If we sample the original dataset uniformly and with replacement, approximately 2/3 of the samples are unique and 1/3 repeated. Hence each Tree focuses on a fraction of the space.

2.4.1.2 Proximity

Random Forest can be used to build a measure of proximity between a pair of samples. For a simple Tree, $T(x,\theta)$ and two sample element, x_1 and x_2, we can define a binary function $lf(x_1,x_2,\theta)$ which determines if both elements belong to the same leaf. Then a similarity function for a forest of M Trees is built using the mean:

$$lf(x_1,x_2) = \frac{1}{M}\sum_{k=1}^{M} lf(x_1,x_2,\theta_k)$$

With this strategy, we can use a forest to discard some samples or even cluster the train dataset. Although the metric $lf(x_1,x_2,\theta)$ is simple, we can improve it to build better measures of proximity using, for example, the depth of the internal node at which they separate when they are queried to the Tree.

2.4.1.3 Regularization

In general terms, Random Forest does not overfit to the training dataset. However, we should take certain guidance choosing the parameters for each Tree to reduce the cost and avoid non optimal performance. We should choose initially the parameters for the Trees, like depth or the fraction of features considered at each splitting point, with the idea of a small bias. We usually choose big Trees (compared to Gradient Boosting) and a small fraction of features (< 0.5) at branching points. It is a good idea to avoid bigger number of features and Trees with a single sample at leaves, despite the error estimation.

After having a good idea of the parameters for Trees, we can determine the optimal number of estimators using *oob* error or any other error estimation procedure. Random Forest can be expanded adding new Trees to the previous forest, instead of computing a complete new forest.

2.4.1.4 Cost

To estimate the cost we just need to aggregate the cost of the base estimators.

In general terms, Random Forest uses independent Trees so the cost will be the cost of a Tree multiplied by the number of estimators.

We can summarize the cost in table (2.4.1), where N is the number of samples, P is the number of features and M is the number of estimators.

Method	Build Time	Build Memory	Query Time	Query Memory
DTR	BT_{Tree}	BM_{Tree}	QT_{Tree}	QM_{Tree}
	$O(N \cdot P \cdot log_2(N)$	$O(N)$	$O(log_2(N))$	$O(1)$
RFR	$M \cdot BT_{Tree}$	$M \cdot BM_{Tree}$	$M \cdot QT_{Tree}$	$M \cdot QM_{Tree}$
	$O(M \cdot N \cdot P \cdot log_2(N)$	$O(M \cdot N)$	$O(M \cdot log_2(N)$	$O(M)$

Table 2.4.1: Cost for RFR

This previous estimation is based on a sequential approach. However, Random Forest allows a very easy parallelization as the process of construction of each Tree is independent. Hence we can divide the cost by a constant.

2.4.1.5 Building a random forest with *Scikit*

As in the previous situation with Trees, *Scikit* code can be quite useful to illustrate the process of building a forest. However, the recent code from the repository can be a bit difficult to read because of parallelization. Previous versions of the forest building process (commit (`c78b1d2`) and previous) follow a very simple sequential process in *Python*. The following code, taken from the public repository of *Scikit* and corresponding to the commit (`c78b1d2`) is useful to illustrate.

The forest building process is **much simpler than the Tree** one. *RandomForestRegressor* is derived from *ForestRegressor*, which itself is derived from *BaseForest*. This last one has *BaseEnsemble* as a father class. *BaseForest* has a fit method which builds a forest, which uses the *_make_estimator()* derived from *BaseEnsemble* to build a Tree.

The following code contains partially the interface of the BaseForest class.

```
class BaseForest(BaseEnsemble, SelectorMixin):
    def fit(self, X, y):
        """Build a forest of trees from the training set (X, y)."""7
        ...
        for i in xrange(self.n_estimators):
            tree = self._make_estimator()
            if self.bootstrap:
                indices = self.random_state.randint(...)
                tree.fit(...)
            else:
                tree.fit(...)
        ...
```

The code builds n Tree estimators with a loop, allowing the user to make bootstrap in the process to reduce the dependence of the Trees.

Modern implementation, in *Python*, has a parallel process with certain improvements as *warm_start*, which reuses the estimators of the previous call to the fit method and adds more Trees (by default RandomForestRegressor just fits a new forest), or weighted samples, which allows to add certain control in the process of building each Tree (splits which create child nodes with net negative (or zero) weight are ignored).

The code can be consulted on the repository, as it only adds practical improvements for performance reasons.

2.5 Boosting methods (boosting)

Boosting is an ensemble strategy for **reducing both bias and variance** of learning predictors [12].

As bagging, boosting uses base estimators to build another estimator, inheriting some properties from the base estimator and improving some weakness from it. However the idea is a bit more complicated.

Most boosting algorithms iterate through a learning process using weak estimators at each step and adding them to a stronger one built by aggregating the previous steps. The difference with bagging is that the new learners focus mainly in the samples which have been incorrectly classified (or estimated) in the previous steps. Different boosting methods use different strategies for computing the new predictors.

For example, Gradient Boosting uses the residual of each prediction as a guide to orientate the growth of the following learner. We will describe the method deeper later.

On the other hand, AdaBoost fits a sequence of weak predictors on slightly modified versions of the original dataset. The changes in the dataset at each step are weights $w_1, w_2, \ldots, w_N$ applied to each of the training element. During the first step, the weights are $w_i = \frac{1}{N}$. At each posterior step, the weights are recomputed and the wrongly classified samples (or those with bigger error in the regression problem) have their weights increased. Reference [1] includes a good description of AdaBoost in chapter 10.

So, **why does it work?**. The idea behind boosting is simple: once you know how to do something, focus on the fields where you are not proficient enough. Instead of using weak learners to estimate our whole dataset, we should focus with each learner in a different part of the dataset. Hence, as we build our strong learner, each new weak learner added should focus on improving the previous elements with the bigger prediction error.

If we have compared bagging with democracy, we can compare boosting with a technocracy where each expert (weak learner) from the government (strong learner) focuses on a concrete area for learning and ruling purposes.

To **estimate the error** we can use any metric of error from the base estimators. In our situation, we will use Trees as predictors and we will aggregate them. As we have mentioned above, both *out of bag* and a validation set are quite useful techniques.

2.5.1 Gradient Boosting

Gradient boosting is a boosting method which uses weak predictors, as Trees, to build a model step to step minimizing the residual from the previous steps.

With Gradient Boosting, if we want to predict y using a sequence of functions $f_k(x)$ and a differentiable metric for the error, as the usual mean square error metric $(y - f(x))^2$, we build a sequence of approximations which increase their quality at each step. At each

[12] Boosting is born from a problem proposed at the end of the 80s: *Can a set of weak learners create a single strong learner?*. Later AdaBoost and Gradient Boosting gave a successful answer.

step, k, a small function, h_k, improves the previous approximation $f_{k-1} = h_1 + \ldots + h_{k-1}$ approximating the residual from the previous step, i.e. h_k solves the problem $\arg\min_h((y - f_{k-1}) - h)^2$.

2.5.1.1 Gradient Boosting problem

For the general regression problem, we need a differentiable function L [13] . Each weak predictor is a Tree [14]. Our model is then defined as a sum of weak learners, $h_k(x) = T(x, \theta_k)$, in which each of them improves the previous steps:

$$f_M(x) = \sum_{k=1}^{M} h_k(x) = \sum_{k=1}^{M} T(x, \theta_k)$$

Each of the Trees solves the problem of learning the residual from the previous steps. For a dataset with N samples then

$$\hat{\theta}_k = \arg\min_{\theta} \sum_{i=1}^{N} L(y_i, f_{k-1}(x_i) + T(x_i, \theta))$$

The model is easy to understand, but not so easy to solve at first sight. However, using normal optimization procedures, an approximation strategy for computing $\hat{\theta}_k$ can be obtained. The error obtained using $f_k(x)$ as an approximation is

$$L(f_k) = \sum_{i=1}^{N} L(y_i, f_k(x_i))$$

As we observe, we only need the value of the function f_k for the N sample points, so we can interpret f_k as vector $f_k \in \mathbb{R}^N$, i.e. $f = (f(x_1), \ldots, f(x_N)) \in \mathbb{R}^N$. With this approach:

$$\hat{f} = \arg\min_{f} L(f)$$

Then we can use normal approximation procedures estimating each step h_k, because

$$f_M = \sum_{k=1}^{M} h_k$$

2.5.1.2 Gradient descent algorithm

Gradient descent can be used to approximate our solution to the previous problem. The components of the gradient of L, g_k, are

$$g_{ik} = \left[\frac{\partial L(x, y)}{\partial y}\right]_{y = f_{k-1}(x_i), x = x_i}$$

Then, we fit a tree to $-g_k$ as an approximation to h_k.

As usual with a gradient descent estimation, we can use a scalar value, p_k, to adjust the step given by the gradient.

[13]As the previously stated $L(x, y) = (x - y)^2$

[14]Defined by $\theta_k = \cup_j^{J_k}\{(R_j^k, \delta_j^k)\}$, i.e $\{R_j^k\}_1^{J_k}$ is partition of the space and we have constants $\{\delta_j^k\}_1^{J_k}$, associated to each partition, as predicted value given by the Tree in the region.

We can compute an optimal step, p_k, as

$$p_k = \arg\min_p L(f_{k-1} - pg_k)$$

$$f_k = f_{k-1} + p_k h_k$$

This value allows us to get the best adjust at each step.

2.5.1.3 Loss functions

Previously we have used as error function the usual squared difference. However, it is common to consider different metrics for computing the error and guide the boosting process. We include some examples of loss functions, taken from [10].

- Least square (LS): $L(y, f) = \frac{1}{2}(y - f)^2$
- Least absolute deviation (LAD): $L(y, f) = |y - f|$.
- M regression:
$$L(y, f) = \begin{cases} \frac{1}{2}(y - f)^2 & \text{if } |y - f| \leq \delta \\ \delta(|y - f| - \frac{\delta}{2}) & \text{if } |y - f| > \delta \end{cases}$$
- Two class logistic: $L(y, f) = \log(1 + \exp(-2yf))$, $y \in -1, 1$
- Multi class logistic: $L(\{y_k, f_k\}_1^N) = -\sum y_k \log(p_k(x))$, where $y_k = 1$ iff $k \in Class(k)$, $p_k(x) = Pr(y_k = 1|x)$.

For simple error functions, L, the computation of h_k and p_k is easy. For example, for the usual least square (LS) we obtain $L'(y, f) = y - f$, so it produces the usual iterative fit to the residuals at each step. For the least absolute deviation (LAD), for $y \neq f$ we have $L'(y, f) = \text{sign}(y - f)$, so it produces a change which modifies the previous estimation on the direction of the current error. Optimal values for p_m can be also computed.

Concrete details, like the exact expression of the gradient, p_m and pseudo code for each concrete implementation can be found in above mentioned reference, [10].

2.5.1.4 Regularization

In contrast with Random Forest, Gradient Boosting tends to overfit. If we fit the training set too closely we can experience a quite strong overfitting. To avoid this problem, there are many regularization techniques. The simplest strategy is limiting the number of trained Trees used in the predictor. High values of M reduce the error on the training set but also cause easily overfitting. For selecting the *adequate* number we usually employ a validation set, cross validation or *oob* error.

Other technique is controlling the complexity of each Tree. For example, the depth of the built Trees. In [1], it is recommended to take a depth of 2 or 3 levels. In comparison, Random Forest takes deeper Trees. Other parameters as the minimum number of splitting samples per internal node can be used.

Other common methodology is applying a shrinkage parameter α in the update process.

$$F_{m+1}(x) = F_m(x) - \alpha p_m g_m(x)$$

Using a learning rate like $\alpha < 0.1$ improves model generalization empirically, but also increases the number of iterations needed to obtain a good solution.

Another option is using a subsample of the original dataset drawn at random at each step (bootstrap). Using a fraction $\gamma < 1$ of the dataset makes the algorithm faster, reduces overfitting and makes the models easier. In reference [11], Friedman obtained good results with $0.5 < \gamma < 0.85$.

2.5.1.5 Cost

To estimate the cost we just need to aggregate the cost of the base estimators.

For Gradient Boosting, each Tree is built using the previous estimation and the residual obtained from the aggregated predictor. Hence, the cost will be the sum of computing the Tree (simpler than the one of Random Forest) and predicting the error used to train the following Tree.

We can summarize the cost in table (2.5.1), where N is the number of samples, P is the number of features and M is the number of estimators.

Method	Build Time	Build Memory	Query Time	Query Memory
DTR	BT_{Tree}	BM_{Tree}	QT_{Tree}	QM_{Tree}
	$O(N \cdot P \cdot log_2(N)$	$O(N)$	$O(log_2(N))$	$O(1)$
GBR	$M \cdot BT_{Tree} + O(M \cdot N) \cdot QT_{Tree}$	$M \cdot BM_{Tree}$	$M \cdot QT_{Tree}$	$M \cdot QM_{Tree}$
	$O(M \cdot N \cdot P \cdot log_2(N)$	$O(M \cdot N)$	$O(M \cdot log_2(N))$	$O(M)$

Table 2.5.1: Cost for GBR

In contrast with Random Forest, Gradient Boosting does not allow a easy parallelization as each regressor tries to improve the errors from the previous one.

2.5.1.6 Building a gradient boosting regressor with *Scikit*

As in the previous situations, *Scikit* illustrates the process of building a regressor. The recent code from the repository has included some improvements and has been ported to `Cython`. The following code, taken from the public repository of *Scikit* and corresponding to the commit (`eb8a2c9`) is useful to illustrate the process because it is very easy to follow.

The gradient boosting regressor building process is simpler than the Tree one. *GradientBoostingRegressor* is derived from *GradientBoostingBase*, which itself is derived from *BaseEstimator*.

GradientBoostingBase has a fit method which builds a regressor.

The following code contains partially the interface and code of the *GradientBoostingBase* class.

```
class GradientBoostingBase(BaseEstimator):
    ...
    def fit(self, X, y):
        ...
        self.init.fit(X, y)
        self.trees = [self.init]
        y_pred = self._predict(X)
        for i in xrange(self.n_iter):
            residual = y - y_pred
            tree = DecisionTreeRegressor(...)
            tree.fit(X, residual)
            self.trees.append(tree)
            y_pred = self._predict(...)

    def _predict(...):
        if old_pred is not None:
            return old_pred + learn_rate * self.trees[-1].predict(X)
        else:
            y = np.zeros((X.shape[0],), dtype=np.float64)
            for i, tree in enumerate(self.trees):
                if i == 0:
                    y += tree.predict(X)
                else:
                    y += learn_rate * tree.predict(X)
            return y
```

The code requires some explanation. The process of building a regressor needs two functions, a fitter and a predictor (which is used in the fitter). We first explain the fitter. After some initialization steps we have omitted and which assign a value to $self.init$ (like *MedianPredictor* or *MeanPredictor*), we create and fit a initial estimator which is added to Trees, our list of estimators, and we create a initial prediction of the target y, y_pred. As our initial estimator is probably not good enough we build a list of new ones in which the target to fit is the residual value. Inside the loop, a new Tree is fit to the residual of the previous estimators, given by the predictor, and is added to the list of our Trees.

The predictor computes how good is our estimator (which is the list of Trees). The function is easy to understand, because it just adds to the last predicted y value a small amount to improve the estimation. $learn\_rate * self.trees[-1].predict(X)$ is a fraction ($learn\_rate < 1$) of the last estimation of the residual.

This early version has been improved in posterior commits adding a more general approach. The commit (`24a1b3a`) generalizes the previous approach. In this commit *GradientBoostingRegressor* derives from *BaseGradientBoosting*, which itself derives from *BaseEnsemble*.

The code for *BaseGradientBoosting*, simplified, is attached below.

```
class BaseGradientBoosting(BaseEnsemble):
    ...
    def fit(self, X, y, monitor=None):
        loss = LOSS_FUNCTIONS[self.loss]()
        if self.init is None:
            self.init = loss.init_estimator()
        # fit initial model
        ...
        # init predictions
        ...
        # perform boosting iterations
        for i in xrange(self.n_estimators):
            # subsampling
            ...
            residual = loss.negative_gradient(y, y_pred)
            # induce regression tree on residuals
            tree = _build_tree(...)
            # update tree leafs
            loss.update_terminal_regions(...)
            # add tree to ensemble
            self.estimators_.append(tree)
            # update out-of-bag predictions and deviance
            ...
        return self

    def _predict(self, X, old_pred=None):
        if old_pred is not None:
            return old_pred + self.learn_rate * \
                   self.estimators_[-1].predict(X).ravel()
        else:
            y = self.init.predict(X)
            for tree in self.estimators_:
                y += self.learn_rate * tree.predict(X).ravel()
            return y
```

The code is quite similar but adds new features, like using different ways of computing the residual. For example, the classes *LeastSquaresError*, *LeastAbsoluteError*, *BinomialDeviance* which contain the method negative_gradient to compute the gradient. The new implementation allows also subsampling, *oob* and modification of the leaves for already built Trees. The complete code can be read in the repository.

Modern version of Gradient Boosting regression, partially in `Cython`, includes many more features and error functions.

Chapter 3

Feature reduction

3.1 Dimensionality reduction

As we have stated in the introductory chapter, we have two different strategies to reduce the number of features, and then the magnitude of our problem. On one hand, feature selection selects a good subset of features, ignoring those features which have been not picked. However, feature selection does not apply any transformation to the chosen variables and can lose some information. On the other hand, dimensionality reduction does not select any subset of data but transforms the original data from an initial high dimensional space to a smaller one, summarizing and extracting information from the whole dataset.

There are many dimensionality reduction methods, and they are classified attending to the type of transformation used and the type of information used to classify or predict. For example, **PCA** (**P**rincipal **C**omponent **A**nalysis) is a linear transformation which creates new features using the directions of maximum variance. Other technique as **LDA** (**L**inear **D**iscrimination **A**nalysis) is a good selection if we have a classification problem and the means and variances are the main information to identify the classes. Both methods can be used with a non linear transformation using the kernel trick (e.g. Kernel PCA). There are also non linear methods, as **LLE** (**L**ocally **L**inear **E**mbedding) or **LTSA** (**L**ocal **T**angent **S**pace **A**lignment).

3.1.1 PCA

PCA is a transformation which converts an initial collection of variables into a list of linearly uncorrelated variables. Each component of this new transformed list represents the vector which contains the largest variance once removed the previous elements from the list.

The idea behind reducing features with PCA is that if the main information for prediction is included in the variance, by removing the smaller components, the loss of information is small. The fraction of total variance associated to each principal component can be used as criterion to *prune* the new set of features.

Formally, we consider a matrix $\mathbf{X}$ with zero mean columns, N rows and P columns. The PCA transformation is a set of vectors w_k which converts each row x_i to a new vector of components given by:

$$t_{k,i} = x_i w_k$$

The unit vectors w_k are computed to maximize the fraction of total variance each new variable represents.

Formally the problem can be stated as follow. For the first component the optimization problem is:

$$w_1 = \underset{||w||=1}{\arg\max}\, ||\mathbf{X}w||^2 = \underset{||w||=1}{\arg\max}\, w^T\mathbf{X}^T\mathbf{X}w = \underset{w}{\arg\max}\, \frac{w^T\mathbf{X}^T\mathbf{X}w}{w^T w}$$

This last quantity is the Rayleigh quotient , so the solution is the eigenvector associated with the biggest eigenvalue. The rest of the principal components can be obtained with a similar procedure, after removing the previous components. For example, for the second component:

$$\mathbf{X_2} = \mathbf{X} - \sum \mathbf{X} w_k w_k^T$$

$$w_2 = \underset{||w||=1}{\arg\max}\, ||\mathbf{X_2}w||^2 = \underset{w}{\arg\max}\, \frac{w^T\mathbf{X_2}^T\mathbf{X_2}w}{w^T w}$$

In general terms, the Rayleigh quotient problem indicates that the principal components of $\mathbf{X}$ are given by the eigenvectors of $\mathbf{X}^T\mathbf{X}$.

Hence, one procedure to compute PCA is normalizing the variables and computing the covariance matrix, the eigenvalues and the eigenvectors of this matrix. Later we orthonormalize the eigenvectors space with Gram Schmidt to build a matrix $\mathbf{W}$ with the $\mathbf{X}^T\mathbf{X}$ eigenvectors as columns. Other procedure is SVD.

3.1.2 PCA and SVD

Singular **V**alue **D**ecomposition (**SVD**) is a type of matrix factorization. The SVD of an $N \times P$ real matrix, $\mathbf{X}$, is a factorization

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$$

with

- $\mathbf{U}$ an $N \times N$ orthogonal matrix.
- $\mathbf{D}$ an $N \times P$ diagonal rectagular matrix with values ≥ 0 on the diagonal.
- $\mathbf{V}^T$ an $P \times P$ orthogonal matrix.

The elements of $\mathbf{D}$ are the singular values of $\mathbf{X}$ and they are usually included in descending order. As $\mathbf{U}$ and $\mathbf{V}^T$ are orthogonal, the columns of each are an orthogonal basis for the space of columns.

The factorization of $\mathbf{X}^T\mathbf{X}$ can be written, given the decomposition $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$, as:

$$\mathbf{X}^T\mathbf{X} = (\mathbf{U}\mathbf{D}\mathbf{V}^T)^T(\mathbf{U}\mathbf{D}\mathbf{V}^T) = (\mathbf{V}\mathbf{D}\mathbf{U}^T)(\mathbf{U}\mathbf{D}\mathbf{V}^T) = \mathbf{V}\mathbf{D}^2\mathbf{V}^T$$

As we observe, we obtain the eigenvector decomposition of $\mathbf{X}^T\mathbf{X}$, being $\mathbf{D}^2$ the matrix of eigenvalues and $\mathbf{V}$ the matrix of eigenvectors ($\mathbf{W}$).

3.1.3 Dimensionality reduction

The PCA transformation can be written in matrix form as $\mathbf{T} = \mathbf{X}\mathbf{W}$, where $\mathbf{W}$ is the matrix of weights which transforms the original dataset into another. If we are interested in reducing the number of features, we can use a truncated version of $\mathbf{W}$, obtaining a truncated version of $\mathbf{T}$ with l columns $\mathbf{T_l}$. The matrix $\mathbf{T_l}$ can also be computed preserving only the largest l eigenvalues from the SVD decomposition.

As stated above, PCA preserves the maximum amount of variance from the original dataset. Hence, in general, we cannot affirm that PCA is a good technique for feature reduction if the main information is not the variance. However, some results have been obtained regarding dimensionality reduction accuracy. For example, if we assume that our data has the form $\mathbf{X} = \mathbf{S} + \mathbf{N}$, being the useful information $\mathbf{S}$ Gaussian and $\mathbf{N}$ a Gaussian noise, with a covariance matrix multiple of the identity, PCA optimizes the mutual information metric between the simplified and original data. Unfortunately, many results about information preservation make assumptions which are difficult to consider as true and hence we usually experience a loss of information if we only consider a fraction of the principal components.

However, there are a few situations in which PCA can be a good solution. On one hand, the variance can be the main information for a problem. One easy example is a classification problem, where the members of each class have (almost) constant values in all excepting one variable. On the other hand, PCA generates a list of uncorrelated variables, which can simplify or improve the execution of a learning method.

Some efforts have been made to design a feature selection strategy with PCA. In [28], a feature selection algorithm is designed using PCA and KNN to conserve those variables which hold a bigger fraction of the variance, getting good results with a face motion problem.

3.2 Feature selection

In this chapter we are going to describe several methods for feature selection. Although the field has been broadly studied and there are well known results (for example for selecting and testing coefficients for Linear Regression), it has been attracting continuous interest and research. Hence, it is easy to find recent contributions which add a new focus to the traditional ones. Some good references to acquire a general idea about feature selection are [12], for a general vision, and [13], for a focus on bioinformatics.

This chapter describes some basic techniques such as testing hypothesis for a Linear regressor, but the focus is put on Trees, Random Forest and Gradient Boosting predictors.

As we stated in the introductory chapter, selection techniques are based on the idea of redundant information for a prediction task. The hypothesis we make is the existence of a few number of variables which can be used to represent the main part of the information, making possible to remove a large fraction of the included variables without a major loss in our prediction accuracy.

This redundant information can have two origins:

- Non informative features: They do not add any additional information about the variable to predict.
- Informative features: They are relevant for the prediction, but their information can be provided by other features.

3.2.1 Formal problem

The formal problem can be stated as:

Given a dataset with N samples and P features per sample, find a way to produce a new dataset with N samples and Q features per sample, with $Q < P$, which holds most of the information from the original dataset.

Most of the times, the expression *holds most of the information* refers to the ability to predict another variable, although other options as maintaining proximity (for clustering) or another type of relation between samples can be used [1].

The ideal, but not elegant, way of finding the best subset of features is an exhaustive search: We build all the subsets of features [2] and we test them to minimize a validation error (or another type of error estimation). Obviously, the method is only reasonable for small datasets with a small number of features.

Most of the times, we cannot afford the previous cost. Then, we should simplify our *brute force* quest for a cleverer one. The idea behind the complete search is assigning a *quality* number to each of the subsets of parameters to rank them and take the subset with the higher *quality.* Then a good framework, in certain degree, should imitate this procedure but without a complete analysis of all the possibilities.

Depending on the procedure to analyze the space of parameters, we have different techniques. Some methods look in the space of features assigning a *quality* value to sets of features, but not to variables individually, using a greedy approach. Other methods analyze each feature alone and later evaluate a set of parameters assigning the sum of the *quality* of each feature they include [3].

Depending on the procedure we follow to analyze the space of parameters, we can classify the methods in 3 groups, as described in the introductory chapter.

Filter methods use simple metrics like error measures or information proximity to score each subset using only the dataset. Usual metrics are mutual information, Pearson correlation and variance thresholds. They are usually fast but they produce an option which is not optimal to any learning method in concrete.

Wrapper methods use a model to score sets of parameters and an algorithm to build a new one using the previous estimation. A set of features is selected and a model trained and tested computing its error. The *quality* of the parameters is related to this error. In general terms, wrapper methods are more cost intensive than filter methods, but they usually provide a better option for certain learning strategies.

Embedded methods select the features in the building process, so the elected variables are biased towards that model. Some examples are Lasso and Trees algorithms. This approach has usually a higher cost than the previous ones.

To measure the *quality* of a set of parameters, techniques like cross validation or a train/validation/test split of the original dataset can be used.

However, the first point to clarify is what we understand as the best possible set of parameters. As we usually expect certain benefits from simplifying our dataset, we should understand that sometimes the expected profits are incompatible [4].

Hence we need to fix a list of preferences. Most of the times we are mainly interested on the quality of the prediction, so this will be our main criteria. However, sometimes we focus also on other factors, as for example the length of the set of parameters or certain restrictions about different types of parameters. Then our strategy should not only penalize the error but also the number of included features and their nature.

If we are interested in ranking the relevance of the features from our dataset, we are

[1] We can be interested in estimating the importance of variables even with no variable to predict. For example, for clustering, representation, ... Usually filter methods, like variance thresholds, smoothness criteria or proximity measures are used.

[2] In general, if we have p features then we have 2^p possible subsets of features.

[3] In this second situation we are building a measure on the space of $S = \{0,1\}^p$ and defining the measure on singleton sets. The measure is extended naturally using σ (finite) additivity to the rest of the space.

[4] In not many words, we usually want a huge simplification with almost no loss of information, which is impossible.

also interested in discovering the stability of the distribution, because the results will depend on the data used to train and the set of parameters or details used for the ranking process. Some methods, such as Trees and ensemble methods based on Trees, use certain randomness during the building process which can affect the obtained results. Hence, an analysis of both the dependence of the parameters and the stability of the results could be necessary.

3.2.2 Filter methods

The section will describe briefly some filter methods which can be useful as a naive approximation to the feature selection problem. Although the strategy they employ is simple, they are sometime quite useful, as our examples will illustrate, and are used as an initial step before using other techniques.

3.2.2.1 Univariate

A statistic, like the **variance** or the **frequency**, of our initial dataset is the most simple strategy for selecting variables.

For example, if we remember our initial Shakespeare's plays problem where we are classifying documents, we can be interested in removing those words which are very frequent (e.g. stop words like prepositions or common verbs like to *be* or *want*). Although the strategy is simple, this technique can simplify the problem allowing us to focus onto more difficult details with the security of avoiding any loss of information.

Other possible statistic is the variance. Using our previous example again, if we are given a collection of texts about Venice (including *The merchant of Venice* among them) it seems reasonable to find the word *Venice* many times, but not to find the word *Shylock* (the Jewish moneylender) in many of them. Hence, it seems reasonable to think that some terms with high variance, like proper names from characters, are going to be useful to classify the texts.

3.2.2.2 Multivariate

If we are facing a prediction problem, we are interested in using our target variable to measure the quality of our features.

One of the simplest strategies is measuring the **correlation** between each attribute and the variable to predict. For (linear) correlation we have the usual Pearson correlation coefficient for variable $i \in \{1, \dots, P\}$, defined and approximated as

$$R(i) = \frac{\operatorname{cov}(X_i, Y)}{\sqrt{\operatorname{var}(X_i)\operatorname{var}(Y)}} \longrightarrow \overline{R}(i) = \frac{\sum_{k=1}^{N}(x_{k,i} - \bar{x_i})(y_k - \bar{y})}{\sqrt{\sum_{k=1}^{N}(x_{k,i} - \bar{x_i}) \sum_{k=1}^{N}(y_k - \bar{y})}}$$

The use of $R(i)$ can only measure the linear dependency between an attribute and the variable to predict.

Other useful approach is **mutual information**, which ranks the relation between the target and each variable. It is defined by

$$MI(i) = \int\int p(x_i, y) \log \frac{p(x_i, y)}{p(x_i)p(y)} dxdy \longrightarrow \overline{MI}(i) = \sum\sum \bar{p}(x_i, y) \log \frac{\bar{p}(x_i, y)}{\bar{p}(x_i)\bar{p}(y)}$$

where $\bar{p}$ is the approximate probability. For continuous variables, we can approximate the densities (e.g. Parzen Windows) or discretize the values.

Other common strategy for a classification problems is entropy (or information gain) measure. We have described it when we explained ID3 algorithm.

Multivariate criteria are useful when some variable alone influences the predicted value. For example, if we want to analyze the utility of some variables to predict the price of a second hand car, we can expect attributes like *age* or *mileage* to correlate alone with the price of the vehicles, whereas others like the RGB rear view mirror color to be unrelated.

3.2.3 Wrapper methods

Although filter methods can offer a good result in some situations, most of the times we do not have simple situations as the ones exposed above. Wrapper methods deal with more complex situations offering better performance than filter methods, but they are computationally more intense.

Wrapper methods are built with two main ingredients: a prediction model and a feature selection strategy. Many times, the model used is a common OLS predictor, although other options as Trees are also usual.

Most wrapper methods are based on the simple idea of adding or removing features to a previous selection of variables. The basic idea is to design an iterative strategy which changes slightly the previous decision set with the idea of reducing the error from our selection.

The difference among them is the strategy for modifying the previous sets of included features. Greedy approaches are common. For example, Forward/Backward Selection adds/removes a feature in each step, trying to improve an error measure. Other techniques, such as simulated annealing, contain a slightly more complex strategy. The following methods, taken from reference [1], give us an idea.

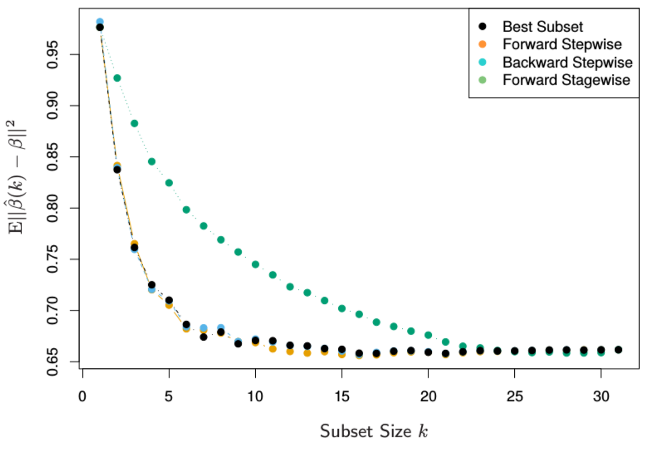


Figure 3.2.1: Subset selection error for different methods, taken from [1]

3.2.3.1 Best Subset Selection

Best subset gets the best subset with a fixed k size ($k \in \{0, 1, 2, \ldots, P\}$). There is a reasonable algorithm to apply this strategy to datasets of < 40 features with OLS. When we use a linear model, the error function is clearly decreasing as the model complexity (k) increases [5]. Then, we usually choose the set of features using a validation set. In contrast with the following methods, the sequence of optimal sets of features is not nested.

[5]Clearly $(Y - \sum_{k=0}^{h+1} \beta_k X_k)^2 \leq (Y - \sum_{k=0}^{h} \beta_k X_k)^2$, as we can make $\beta_{h+1} = 0$

3.2.3.2 Forward and Backward Stepwise Selection

The most common wrapper strategy is a greedy approach, producing a nested sequence of sets of variables. Both Forward Stepwise Selection (FSS) and Backward Forward Stepwise Selection (FSS) use a linear model. *Forward Stepwise* fits initially a constant (β_0), and then adds into the predictor the best variable at that step. *Backward Stepwise* starts with the entire model, and step by step deletes the feature which is less useful. Usual strategies for expanding/pruning a linear predictor are adding/dropping the variable which produces the better error change or the one with the best statistical (usually t or z) score. Common strategies for controlling the process are using a validation set or, sometimes, the training error.

3.2.3.3 Forward-Stagewise Regression

Forward Stagewise Regression (FS) follows a similar strategy. As in the previous case, the most common predictor is a linear model. Initially it sets a constant (β_0) equal to the mean of y. After that, the algorithm gets the variable most correlated with the residuals and fits the increased model to them. The process iterates and finishes when the remaining variables do not have any correlation with the residuals.

3.2.4 Embedded methods

In embedded methods, in contrast with filter methods, which do not use a learning model, or wrapper methods, which use a learning model to rank the subsets of features with a greedy approach, the learning model and the feature ranking cannot be separated, i.e. the hypothesis model plays a main role. Hence the selected model will be used to classify the different embedded methods. Linear models, Trees and ensemble methods are good examples.

3.2.4.1 Feature selection with Linear methods

For a general linear problem for Y:

$$\hat{Y} = \sum_{k=0}^{P} \beta_k X_k$$

where $Y = \hat{Y} + \epsilon$, being ϵ the error, we want to select a subset of features which still makes our estimator accurate.

For that we can use some properties from the solution. If we do not make any assumption about the distribution of the error ϵ, we cannot infer the distribution of $\hat{\beta}$, the estimator of β, or $\hat{\sigma}^2$, the estimator of σ^2. However, for a sample large enough, with the Central Limit Theorem we can deduce [6]

$$(\hat{\beta} - \beta) \to^d N(0, \sigma^2(X^T X)^{-1})$$

$$(\hat{\sigma}^2 - \sigma^2) \to^d N(0, E(\epsilon^4) - (\sigma^2)^2)$$

For finite samples, we can infer some results for $\hat{\beta}$, $\hat{\sigma}^2$ and s^2 (normalized variance).

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

[6]Hastie, [1], chapter 3, is a good reference.

$$s^2 = \frac{(y - X\beta)^T(y - X\beta)}{N - P - 1}, \hat{\sigma^2} = \frac{N - P - 1}{n} s^2$$

being $\hat{\beta}$ and s^2 unbiased under $E(\epsilon) = 0$. Under normality assumption for ϵ

$$(\hat{\beta} - \beta) \sim N(0, \sigma^2(X^TX)^{-1})$$

$$s^2 \sim \frac{\sigma^2}{N - P - 1}\chi^2_{N-P-1}$$

being $\hat{\beta}$ and s^2 independent, which is useful to build a test. Once we know the distribution of the coefficients we can build a test to check if statistically $\beta_j = 0$.

Under normality assumption (or near normality) we have different options. For example, the simplest option is an individual zero test for each coefficient. To test $\beta_j = 0$ we use

$$t_j = \frac{\hat{\beta}_j}{\hat{\sigma}\sqrt{[(X^TX)^{-1}]_{jj}}} \sim t_{N-P-1}$$

For blocks of variables we can use the Wald test. To test a set of Q linear relations between the coefficients using a matrix R (QxP matrix)

- $H_0 : R\beta = r$
- $H_1 : R\beta \neq r$

we use the normality of $\sqrt{n}(\hat{\beta_n} - \beta) \to^d N(0, V)$ to build

$$(R\hat{\beta_n} - r)^T(R(\hat{V_n}/n)R^T)^{-1}(R\hat{\beta_n} - r) \to^d \chi^2_Q$$

If we prefer a simple, but powerful, test to check the inclusion of some variables in a model, we can use a F(isher) test. The F distribution is built by the ratio of two independent χ^2 variables, normalized using the degrees of freedom. For two models, m_1 and m_2, where m_1 is nested in m_2, i.e. m_1 has p_1 parameters and m_2 has the same p_1 parameters and $p_2 - p_1$ additional ones [7]. For that we build the F statistic:

$$F = \frac{\frac{RSS_1 - RSS_2}{p_2 - p_1}}{\frac{RSS_2}{N - P_2 + 1}}$$

being RSS_i the sum of squared residuals for model i. If we assume (null hypothesis) that m_2 does not add a statistically significant improvement over m_1, then F will be distributed as a F distribution with $p_2 - p_1$, $N - p_2 + 1$ degrees of freedom.

3.2.4.2 Feature selection with Trees

In general terms, we have two different frameworks for measuring the relevance of a variable with a Tree. On one hand, we have the *decrease of impurity* measures, which are usually used during the building process of a Tree. On the other hand, we have the *decrease of accuracy* measures, which are based on the idea of noising or permuting variables with the intention of observing the loss of accuracy of the new predictor compared with the original one.

We analyze first the **decrease of impurity** measures.

[7] For example, m_2 can be the full model. However, if we are using a wrapper method the F statistic can be useful to discover if we gain adding an individual variable.

The process of building a Tree involves splitting the dataset using a feature and a splitting point with the intention of reducing certain impurity or error measure. This metric can be used to give an estimation of the relevance of each feature, as those variables chosen at initial steps (and near the root) will be more important.

For ID3 Trees, a common used metric for the importance is the mutual information between feature X_i and the outputs Y:

$$MI(X_i, Y) = H(Y) - H(Y|X_i) = H(X_i) - H(X_i|Y)$$

$$H(X_i) = \sum_j p(x_{ij}) I(x_{ij}) = -\sum_j p(x_{ij}) \log p(x_{ij})$$

$$H(Y|X_i) = \sum_{j,k} p(x_{ij}, y_k) log \frac{p(x_{ij})}{p(x_{ij}, y_k)}$$

being H the entropy function, I the information content (defined in Chapter 2). Usually only a fraction of the features is included into the Tree and with different positions within it, so it behaves as an embedded selection method.

For CART Trees, we have two common options. For classification problems, Gini impurity $\Delta GI(t)$ is used, being defined as

$$\Delta GI(t) = P_t \cdot GI(t) - P_l \cdot GI(t_l) - P_r \cdot GI(t_r)$$

where $GI(t)$ is the Gini index, defined as

$$GI(t) = 1 - \sum_k p(k|t)^2$$

where $p(k|t)$ is the rate of correct classification for class k at node t, $GI(t_l)$ and $GI(t_r)$ are the Gini index at the left and right side respectively and P_t, P_l and P_r are the number of samples before the split, after the split at the left node and after the split at the right node respectively.

For regression, MSE error is the usual option. Mean square error, MSE, is defined as

$$MSE(t) = Var_l(t) + Var_r(t)$$

where $Var_l(t)$ and $Var_r(t)$ are the errors for the left and right children at node t. Usually $Var_i(t)$ are computed during training time as:

$$Var_i(t) = \sum (\bar{y} - y_k)^2$$

where the sum is for all the sample included at the i side at node t and $\bar{y}$ is the average value of all the training sample at this side, which is the predicted value at each leave.

For a collection of Trees, Breiman proposed Mean Decrease Impurity importance (MDI) which estimates the importance of a variable X_m adding the weighted impurity decline for all nodes where the variable X_m is used, averaged over all the Trees in the collection:

$$Imp(X_m) = \frac{1}{N} \sum_T \sum_{t \in T, v(s_t) = X_m} p(t) \Delta i(s_t, t)$$

where $i(t)$ is the impurity measure, $p(t)$ is the proportion of samples reaching t and $v(s_t)$ is the variable used splitting s_t. We will see later other measures of relevance for ensemble methods with Trees.

We analyze now the **decrease of accuracy** measures.

The most common measure of relevance using the *decrease of accuracy* framework is the permutation loss of accuracy, proposed originally by Breiman. The core idea is to permute the values of each feature individually and measure the increase of error obtained. For non relevant variables, the permutation should have almost no effect whereas for important ones the error is expected to increase. The most common implementation trains a Tree (or ensemble method) and assigns as relevance to each variable the difference of accuracy normalized between the predictor applied to a original dataset and a dataset resulted permuting each variable, step by step.

This method is not implemented in *Scikit*, but an example code is given below to illustrate the idea.

```
def _permutation_relevance(self, X, Y, error):
    rel = []
    method = DecisionTreeRegressor() // or RFR or GBR
    method.fit(X.train, Y.train)
    initial = error(Y.test, method.predict(X.test))
    for k in range(X.shape[1]):
        Xt = X.test.copy()
        np.random.shuffle(Xt[:, k])
        final = error(Y.test, method.predict(Xt))
        rel.append((final-initial)/initial)
    return [item/sum(rel) for item in rel]
```

With ensemble methods, test error is usually substituted by *oob* error.

Other common option is noising a variable (for example with a Gaussian noise), with the intention of determining the importance of the corrupted variable. Clearly, those variables which are near the root or are used many times in the internal nodes of the Tree will produce a bigger increase of the error when they are modified. An example is given later for Random Forest.

3.2.4.2.1 Feature selection with *Scikit*

Feature relevance is a functionality available in most implementation of regression Trees, as *Scikit*. However, *Scikit* has been moving towards efficiency which usually means loss of simplicity. For class Tree, feature relevance was added in commit (`ca9472c`), and refactored in posterior commits like commit (`aeb94c6`) in which the computation is parallelized. Details about the computation can also change between different implementations slightly. The following code, which is the first commit with the feature importance functionality, is useful to illustrate the computation.

```
def feature_importances(self):
    """Compute the feature importances of all features.
    The importance I of a feature is computed as the (normalized) total
    reduction of error brought by that feature.

    I(f) = \sum_{nodes A for which f is used} n_samples(A) * \Delta err
    """
    tree = self.tree_
    importances = np.zeros(self.n_features_)

    for node in xrange(tree.node_count):
        if tree.children[node, 0] == tree.children[node, 1] == Tree.LEAF:
            continue
        else:
            importances[tree.feature[node]] += \
                tree.n_samples[node] * (tree.init_error[node] - tree.best_error[
                    node])

    importances /= np.sum(importances)

    return importances
```

Some explanation should be given about the code. For this implementation, a binary Tree is a *struct* of arrays. The i element of each array holds information about the corresponding node. For example, the *struct* holds information about the number of nodes (internal and leaves), an array of children (each item of the array is an array which contains the index to each of the children, if they exist), the feature and threshold used in the node to split, the initial error of the node (before splitting) and the best error (after splitting). There are also some extra details which are not necessary for this explanation. The concrete details can be found in the source code.

With the previous details in mind, we can understand the code easily. The *for* instruction loops all the nodes, ignoring the leaves (*if* section) and adding to the feature used in each node the error given by the formula (*else* section):

```
importances[tree.feature[node]] +=
                tree.n_samples[node] *
                (tree.init_error[node] - tree.best_error[node])
```

The value *importances* is an array which contains the importance of each feature and $tree.feature[node]$ is the feature used in *node*. The values $tree.n\_samples[node]$, $tree.init\_error[node]$ and $tree.best\_error[node]$ give the number of samples, initial error (before splitting) and best error (after splitting) values for the node respectively. The final step is normalization. The errors are computed using different metrics.

- Classification
  - Gini
  - Entropy
- Regression
  - MSE
  - Other MSE: Friedman MSE.

In another implementation like commit (`83bedfd`), the possibility to use another formula for computing the relevance is shown directly in the function.

```python
def compute_feature_importances(tree, n_features, method='gini'):
    """Computes the importance of each feature (aka variable).

    The following `method`s are supported:

      * gini : The difference of the initial error and the error of the
               split times the number of samples that passed the node.
      * squared : The empirical improvement in squared error.
    """
    gini = lambda tree, node: (tree.n_samples[node] * \
                              (tree.init_error[node] - tree.best_error[node]))
    squared = lambda tree, node: (tree.init_error[node] - \
                                  tree.best_error[node]) ** 2.0
    method = {
        'gini': gini,
        'squared': squared
        }[method]
    importances = np.zeros((n_features,), dtype=np.float64)
    for node in xrange(tree.node_count):
        if (tree.children[node, 0]
            == tree.children[node, 1]
            == Tree.LEAF):
            continue
        else:
            importances[tree.feature[node]] += method(tree, node)
    importances /= np.sum(importances)
    return importances
```

Although the code has been refactored, it follows the same logical flow and includes the possibility of choosing between the previous metric for relevance (*Gini*) and a new one (*squared*) which is basically the squared improvement of error in a node.

Modern implementation of feature importance in *Scikit* is slightly different. The implementation uses `Cython` and is parallelized, although the core idea is the same (we loop through all the internal nodes computing a metric which measures, in certain degree, the impurity or error of the node and we average the results obtained for all nodes). The main expression which computes the importance is a bit different because allows weights.

```
importances[node.feature] +=
                        (node.weighted_n_samples * node.impurity -
                        left.weighted_n_samples * left.impurity -
                        right.weighted_n_samples * right.impurity)
```

Impurity is an array with the size of list of nodes, where $impurity[i]$ holds the value of the splitting criterion at node i. The impurity is computed using functions as the ones described above. A new metric has been added to compute the internal error, related to the original Friedman paper about Boosting Methods. The formula for Friedman MSE is:

$$diff = mean\_left - mean\_right$$

$$improvement = \frac{n\_left * n\_right * diff^2}{n\_left + n\_right}$$

3.2.4.3 Feature selection with Random Forest

Random Forest is a bagging (bootstrap aggregation) technique, so the feature relevance can be derived naturally from the base estimators, the Trees. In general terms, if $VI^T(p)$ represents the feature importance measure for Tree and variable p and we build a collection of Trees $\{T_k\}_{k=1}^M$ with importance $VI^{T_k}(p)$, we can build the feature relevance equivalent metric for Random Forest as an average:

$$VI^{RF} = \frac{1}{M}\sum_{k=1}^{M} VI^{T_k}$$

Then, we can deduce easily information from VI^{RF}, as each VI^T is identically distributed as a consequence of bootstrap and random selection of some variables at the splitting points. Hence, using the Central Limit Theorem we can still deduce convergence. Hence for M Trees with σ^2 as variance,

$$\widehat{VI^{RF}} = \frac{VI^{RF}}{\frac{\hat{\sigma}}{\sqrt{M}}}$$

is distributed asymptotically standard normal [8] . We can use the z score with $\widehat{VI^{RF}}$ to check if the variable importance is null easily.

However, as described in [18], this strategy can be unsuccessful, because as the authors checked, with variable permutation metric the resulting test has undesirable properties (like the power of the test not increasing with the sample size).

[8] As seen in [18]

3.2.4.3.1 Alternative definition

Apart from the previous inherited definition, new feature relevance metrics can be defined for RF because of the greater stability of the method. In general terms, whereas the Tree based feature relevance measures use mainly, for stability reasons, the *error* of classification/regression for assessing an importance value to each attribute, ensemble relevance measures can use also some type of noising strategy for each feature and compare the resulting error with the original one. This approach was started in the original Random Forest description by Breiman.

Breiman in his original paper about Random Forest, [2], defined an alternative proposal for variable importance different to the Tree *impurity* or *error* measures. Formally, if we have p variables, after each Tree is constructed the p variable is randomly permuted and the resulting *oob* error computed. The relevance of a variable is defined then as the difference of *oob* errors normalized.

Another option is proposed by Gilles Louppe et al., [16], which uses MDI (Mean Decrease Impurity), described above, and obtains certain theoretical results. Louppe assumes some simplifications:

- $V = \{V_1, \ldots, V_n\}$ are categorical variables, Y is categorical output
- We have an infinite ensemble of Trees.
- The variables used to split are selected randomly among those not still used.
- No binary partitions: each internal node is split in as many branches branches as possible ($size(\cup_i X_{ij})$ branches for a node splitting with variable j).

With these hypothesis and an asymptotic sample size he obtains a decomposition of the mutual information of all the variables as the sum of all the importance.

$$\sum_{k=1}^{P} Imp(X_k) = I(X_1, \ldots, X_P, Y)$$

In the paper by Ishwaran, [19], we can see a good analytical framework which assumes a more realistic basis. He adopts a simpler definition of variable importance using the *loss of accuracy* strategy: the prediction difference under noising and without noising. From a more formal point of view, we define noising a variable v adopting the following procedure:

We drop an sample, x, in the Tree until either a terminal node or a internal node depending on v is reached. In the second situation, the right or left options are selected with equal probability and the item is randomly dropped until a leaf.

This random strategy is designed to create two new (random) Trees which have lower accuracy than the original one. If we call this new Trees T_v^1 and T_v^2, the prediction performance of T_v^i is related to the position of v, so when a variable v appears high near the root, the difference between T and T_v^i is bigger.

At the moment of writing the present thesis, there are not any complete comparison of different feature relevance for Random Forest and their properties.

3.2.4.3.2 Feature selection with *Scikit*

Scikit has also a feature relevance computation procedure for Random Forest. As in the previous situation, *Scikit* has been moving towards efficiency with the associated complexity. For class Forest, feature relevance was added in commit (`ca9472c`), and refactored in posterior commits. Details about the computation can also change between different implementations slightly. The following code, which is the first commit with the feature importance functionality, is useful to illustrate the computation.

```
    def feature_importances(self):
        """Compute the mean feature importances over the trees in the forest."""
        importances = np.zeros(self.estimators_[0].n_features_)

        for tree in self.estimators_:
            importances += tree.feature_importances()

        importances /= self.n_estimators

        return importances
```

The previous code is self explanatory, because the importance is computed as the average of importance of each Tree.

Modern implementation of feature importance in *Scikit* is slightly different. The implementation is parallelized but the idea is the same.

```
    def feature_importances_(self):
        """Return the feature importances (the higher, the more important the
           feature). """
        if self.estimators_ is None or len(self.estimators_) == 0:
            raise NotFittedError("Estimator not fitted, "
                                 "call `fit` before `feature_importances_`.")

        all_importances = Parallel(n_jobs=self.n_jobs,
                                   backend="threading")(
            delayed(getattr)(tree, 'feature_importances_')
            for tree in self.estimators_)

        return sum(all_importances) / len(self.estimators_)
```

3.2.4.4 Feature selection with Gradient Boosting

Gradient Boosting is a boosting technique, so the feature relevance can be derived naturally from the base estimators, the Trees. In general terms, if $VI^T(p)$ represents the feature importance measure for Tree and variable p and we build a collection of Trees $\{T_k\}_{k=1}^M$ with importance $VI^{T_k}(p)$, we can build the feature relevance equivalent metric for the Gradient Boosting estimator $F = \sum \gamma_k T_k$ as an average:

$$VI^{GB} = \frac{1}{M}\sum_{k=1}^{M} VI^{T_k}$$

which is the RF expression. We can add also a factor (perhaps a normalization of γ_k) as we expect more important contributions $\gamma_k T_k$ to be also more important defining the relevance.

Sadly, the Trees resulting from a boosting process are not identically distributed. As a consequence, (a simple version of a) Central Limit Theorem cannot apply and we do not have an easy estimation of the distribution of $\widehat{VI^{GB}}$.

3.2.4.4.1 Alternative definition

Alternative feature selection metrics for gradient boosting follow a similar pattern to the ones described for Random Forest. Hence those ideas which are useful to Random Forest also apply to Gradient Boosting methods.

Apart from the *error* measure, Friedman, at [10], describes for a single Tree a measure of relevance which is useful to small Trees, such as those built with GB.

For a single decision Tree T_k, the importance is defined as

$$(VI^T)^2(p) = \sum_{k=1}^{M} d_k \cdot I(p,k)$$

where $I(p,k)$ is a binary function which returns 1 if node k splits with variable p and d_k is an impurity measure. Then, for GBR it is defined as the average for all Trees.

3.2.4.4.2 Feature selection with *Scikit*

Scikit has also a feature relevance computation functionality for Gradient Boosting. As in the previous situation, initial commits of *Scikit* are easier to understand. For class GradientBoostingRegressor, feature relevance was added in commit (`8a57c4e`), and refactored in posterior commits. Details about the computation can also change between different implementations slightly. The following code, which is the first commit with the feature importance functionality, is useful to illustrate the computation.

```python
def variable_importance(self):
    ...
    for tree in self.trees:
        vi = VariableImportance(self.n_features)
        vi.visit_nonterminal_region(tree)
        variable_importance += vi.variable_importance
    variable_importance /= len(self.trees)
    variable_importance = 100.0 * (variable_importance /
                                   variable_importance.max())
    return variable_importance

class VariableImportance(object):
    ...
    def visit_nonterminal_region(self, node):
        if node.is_leaf:
            return
        else:
            # do stuff
            #print node.initial_error, node.best_error
            feature = node.feature
            error_improvement = (node.initial_error - node.best_error) / node.
                initial_error
            self.variable_importance[feature] += error_improvement ** 2.0

            # tail recursion
            self.visit_nonterminal_region(node.left)
            self.visit_nonterminal_region(node.right)
```

In general terms, the variable_importance method loops through all the Trees computing the feature relevance (using the *VariableImportance* class) and later it normalizes the value to be returned. *VariableImportance* computes the importance of each feature for a given Tree visiting each node of the Tree in a recursive manner, skipping the leaves. It computes the relevance as the sum of squared error improvements, given by

$$error\_improvement = \frac{node.initial\_error - node.best\_error}{node.initial\_error}$$

The meaning of these variables has been given above for decision Trees.

Modern implementation of feature importance in *Scikit* is a bit different (it uses *stages*, collection of Trees, for the aggregation). However, the main implementation idea is the same.

```python
def feature_importances_(self):
    """Return the feature importances (the higher, the more important the
        feature"""
    for stage in self.estimators_:
        stage_sum = sum(tree.feature_importances_
                        for tree in stage) / len(stage)
        total_sum += stage_sum

    importances = total_sum / len(self.estimators_)
    return importances
```

3.2.4.5 Pruning variables with Trees

Trees and their ensemble methods can become our chainsaw to prune variables. If we have a dataset with N samples and P features, we are interested in designing a strategy to remove the *less* informative variables, with the intention of simplifying the models, reducing training time and memory and improving generalization.

For example, a simple approach can be defining a ranking of features with an error measure (for example, using *oob* or a validation set). We can define a sequence p_k associated to the error of the dataset which contains the k most relevant features. Then, we can define a criteria to choose a good value of k, preferably without computing the whole sequence. This approach is based on the idea of assigning a *quality* measure to each singleton set of features and extending the measure in a natural way. However, a more general approach can be taken without supposing the *quality* is a measure, which complies with additivity, at the cost of a more complex selection strategy.

The simple wrapper methods described previously give us good strategies for performing the search, although they perform a complete step to step search. Hence we need better strategies to deal with usual problems, such as $P \gg 1$ or $P \gg N$ and not many resources for a deep search process. In this section we are going to present some advanced techniques to deal with the problem of large number of features with good results.

Although some methods are described to be used for RF or GB, they can be used with both ensemble method without any problem.

3.2.4.5.1 Recursive Feature Elimination (RFE)

Our first strategy, described in reference [20], defines a good methodology for variable selection with RFR, **using a recursive selection approach**. The authors use RFR for **classification** of **microarray data** and develop a new wrapper algorithm of feature selection, using the attribute relevance given by RFR, which produces small subsets of variables and preserves prediction quality. Using simulated and real datasets they show that recursive selection with RFR has a performance similar to LDA, KNN or SVM and the technique obtains good reduction and results.

The idea is simple. RFR are fit at each step after removing a fraction of the variables with the lowest importance. The finally selected variables are those within a range of the smallest *oob* error. However, *oob* error is biased down, so it cannot be considered an estimation of the error of the estimator. The authors propose using bootstrap to estimate the correct error.

During all the process, we use the initially computed feature relevance, i.e. after each pruning step feature distribution is not computed again. The authors use as variable importance the permutation loss of accuracy.

The proposed algorithm analyzes all RF estimators derived from removing, step by step, a fraction of variables (the less important ones). As a default value, the authors propose removing at each step the 20% of the lowest features in the ranking. Hence, if we prune at each step the $(100 \cdot \lambda)\%$, with $\lambda \in [0, 1]$, we need approximately the following number of steps:

$$n_{steps} \approx \frac{log_2(\frac{1}{P})}{log_2(\lambda)} = log_2(P) \cdot \frac{1}{log_2(1/\lambda)}$$

As we observe we are multiplying a small value $log_2(P)$, compared with the number of features P [9], with a value $f_\lambda = \frac{1}{log_2(1/\lambda)}$ which is also small for practical elections of λ [10]. As a consequence, for a dataset with approximately 1000 features and $\lambda = 0.5$ we will need only to train 10 RF, with decreasing cost, and for $\lambda = 0.8$, we will need 31 steps, which is still a good value compared with a naive approach of adding constant blocks of variables. For a bigger dataset with $16,000$ features and $\lambda = 0.5$ we will need 14 steps, whereas for $\lambda = 0.8$ we will need 44 steps.

The methods builds a predictor with the new pruned dataset at each step with a logarithm distribution in the number of features included (for example with $\lambda^k \cdot P$ variables each dataset, for $k \in \{1, \ldots, [log_\lambda \frac{1}{P}]\}$), so the analysis is biased towards small datasets.

The time cost of computing a new RFR with each step is reduced also (because the cost for a Tree is linear in the P). Hence, if the prune the $100\lambda\%$, with $\lambda \in [0,1]$, and C is the time cost for the whole dataset we need in total

$$C_{prune} \approx \sum_{j=0}^{n_{steps}} \lambda^j C < \sum_{j=0}^{\infty} \lambda^j C = \frac{C}{1-\lambda}$$

Then, for $\lambda = 0.5$, the time cost is $C_{prune} < 2C$ and for $\lambda = 0.8$ we need $C_{prune} < 5C$.

Finally, after computing all the RF estimators, we analyze the *oob* error and we choose the estimator with the lowest number of variables with an error within α standard deviations of the smallest error rate. For $\alpha = 0$, we choose the collection of variables with the smallest *oob* error, whereas $\alpha \geq 0$ can allow better generalization. The authors find that $\alpha \geq 0$ shows similar error rates to $\alpha = 0$, with the advantage of a much smaller set of variables chosen.

3.2.4.5.2 Boruta

Another wrapper around a RF classification problem is Boruta, described in [23]. It removes step by step the features which are less relevant using a test criterion comparing the original features which newly added variables (*shadow* variables) obtained from the shuffle of some of the original attributes. The relevance of these new variables can be > 0 as a consequence of the process. Then, the feature importance of these *shadow* variables is used as a threshold for the rest using an approximation to a statistical z test.

The algorithm, as described by the authors in the original paper [11]:

---

1. Extend the problem by adding copies of all variables, duplicating then the number of features.
2. Shuffle the added attributes.
3. Run a RFR on the extended information system and gather the Z scores computed.
4. Find the maximum Z score among shadow attributes ($MZSA$), and then assign a hit to every attribute that scored better than $MZSA$.
5. For each attribute with undetermined importance perform a two-sided test of equality with the $MZSA$.
6. Deem the attributes which have importance significantly lower than $MZSA$ as unimportant and permanently remove them from the information system.
7. Deem the attributes which have importance significantly higher than $MZSA$ as important.

[9] For example, $log_2(1,000) \approx 10$ and $log_2(16,000) \approx 14$

[10] For example, for $\lambda = 0.8$, $f_\lambda \approx 3.1$ and for $\lambda = 0.5$, $f_\lambda = 1$

[11] Taken from [23] directly

8. Remove all shadow attributes.
9. Repeat the procedure until the importance is assigned for all the attributes, or the algorithm has reached the previously set limit of the random forest runs.

This procedure is repeated several times (*iterations*) classifying the features as relevant, non relevant or ambiguous. The time cost for the statistical tests and per iteration is approximately linear in both parameters, $O(PN)$.

In contrast with the previously exposed techniques, *Boruta* does not allow us to choose the size of the final set of features. *Boruta* tries to find all the relevant attributes, instead of looking for the *optimal* set with a certain size. Hence, we can define Boruta as an *all relevant* features strategy, because it uses those features better than a random one.

However, some modification can be made [12]. For example, instead of using the maximum of the shadow score, a percentage of the score can be used. With this small modification we can control the number of variables we discard and adjust the final set of chosen variables.

Other possible gain is to use some correction to the z test (e.g. a Bonferroni correction). The author of the *Python* implementation for Boruta, in [27], included this additional ideas to improve the original algorithm.

3.2.4.5.3 Subset RFE (SRFE)

Another strategy, described in reference [22], is Subset Recursive Feature Selection (SRFE) which defines a good wrapper algorithm for variable selection with GBR with an approach similar to a recursive pruning (RFE). The authors define a backward elimination technique a bit different from the previous one. The idea is randomly partition the data in disjoint subsets with a fixed amount of features and training an estimator with each selection of features. In each step a fraction of the features is removed, using a metric like *oob* or validation error [13] and the *surviving* features are merged. The process is repeated until a finishing criterion can apply.

In general terms, the authors get good accuracy with a greedy elimination rate such as 25% or 50%, although slightly better results are obtained with smaller rates (2%).

The cost of the operation will depend on the prune rate and the fixed size of the subsets. As a defaults value, the author proposes removing at each step a fixed amount of the less relevant features and sets of 30 features. Hence, if the prune rate is $100\lambda\%$, with $\lambda \in [0, 1]$, and we take sets of $0 < c < P$ features we need approximately the same n_{steps} and C_{prune} values as a simple recursive pruning. However, with this strategy we will need to compute at each step j of the n_{steps} steps a total of $\lceil \frac{\lambda^j P}{c} \rceil$ ensemble estimators. At the end we will need to compute a total

$$n_{est} \approx \sum_{j=1}^{n_{steps}} \frac{\lambda^j P}{c} = \sum_{j=1}^{\infty} \frac{\lambda^j P}{c} \cdot I(\frac{\lambda^j P}{c} > 1) < \frac{P}{c} \frac{1}{1-\lambda}$$

clearly not all of them with the same cost. On the other hand, we just needed to compute n_{steps} with a simple recursive schema. As an example, if we work with $1,000$ variables and a fixed size of a $c = 20$, for $\lambda = 0.5$ we need $n_{est} \leq 100$, whereas for $\lambda = 0.8$ we need $n_{est} \leq 250$. With a recursive feature selection strategy we needed $n_{est} \leq 10$ and $n_{est} \leq 14$ respectively.

[12]The authors provided an *R* implementation. An improved *Python* one is publicly available also.

[13]The authors propose the method for internet search ranking, so they propose a metric like NDCG@k. In our situation we should use *oob* or validation error.

We should notice that the cost can be bigger, as in each step the dataset should be prepared and many ensembles should be computed and used for prediction, but with small (c variables) subsets of features. However, although all the previous can be considered as a big drawback compared to a simple recursive elimination, this methodology can become an advantage if we are dealing with a large number of features and we are able to work only with small fractions of them because of limitation of resources (for example a huge problem with $\geq 10^6$ features can be managed easily in blocks of $1,000$ variables without problems with the memory).

Initially, we expect a similar behaviour to a plain recursive prune, as we are pruning the less informative variables. However, at each step we are not keeping necessarily the best variables using the initial ranking, but the best resulting from small competitions among random subgroups of them, a fact which adds certain randomness to the selection reducing the possible masking among related variables. At the end of the process, the results can be approximately similar to a recursive prune for small final datasets, depending on the fixed values c and λ.

As we can observe, this strategy is really a *meta wrapper* algorithm, as the core idea of the strategy can be applied to any other algorithm, like for example Boruta.

3.2.4.5.4 Comparative

From the previous section, we observe that there are two different strategies for pruning the dataset:

- Linear order procedures: They use the total order given by the feature importance of the features to select a small amount of them depending on their importance.
- Statistical procedures: They add some new features, usually generated from permuting the existing ones, and use statistical methodologies to compare.

As we have seen, not only the focus and assumptions are different but also the time/memory resources needed. In general terms, the idea behind linear procedures is keeping the best fraction of features once decided the final size or performance expected, whereas with statistical procedures is keeping all the relevant features, usually requiring a higher number of iterations than in the previous situation.

For example, linear procedures such as RFE and SRFE prune the dataset using a logarithm grid of sizes and they are easy and fast to tune and compute. They also allow obtaining different final sizes to have an idea of the performance, being also possible to implement an automatic rule. On the other hand, statistical methods, such as Boruta, offer the possibility of keeping all the relevant variables (using a statistical procedure which gives us all those features which are more important than an artificially generated one) and hence liberates the user from the task of determining the correct size. However, if the size is also an important issue, some parameters from the statistical tests can be usually tuned to modify the final number of conserved variables.

Chapter 4

Experiments

In this chapter we are going to solve the initial questions presented in the introductory chapter. For that, we have used two different problems to analyze the utility of feature selection from a practical point of view:

- **B**ike **S**haring problem (**BS**).
- **N**umerical **W**eather **P**rediction problem (**WEP**).

The **first** problem uses a dataset from the *Laboratory of Artificial Intelligence and Decision Support* (*LIAAD*) (University of Porto) and it is available at UCI repository. It deals with the usual problem in the modern days of building a good bike sharing system in a big city.

New bike sharing systems, like the one available in big European cities as London or Madrid, automate the whole process of rental and return. Data from each collection point is generated automatically, including information about the duration of travel, initial and final position, date and other details.

Although this problem includes a big number of samples, the number of variables is small (≤ 20) and they have a quite simple meaning, so this problem will be useful to understand **F**eature **R**elevance (FR) with predictors such as **D**ecission **T**ree **R**egressors (DTR), **R**andom **F**orest **R**egressors (RFR) and **G**radient **B**oosting **R**egressors (GBR). Hence, it will be our toy example to show the effect of a manual and naive pruning (one variable at each step) on the accuracy of the prediction.

The **second** problem uses data from the European Centre for Medium-Range Weather Forecasts (ECMWF) and deals with the problem of forecasting wind energy production in Sotavento (Galicia, Spain), using a grid of atmospheric variables.

In contrast with the previous situation, this second problem includes still a big number of samples but also a big number of features, and their real meaning is a bit harder to understand because of the size of the problem. Hence, this problem will be useful to show how to face a real situation and the utility of the pruning strategies defined in the previous chapter.

4.1 Parameters and methodology

Clearly, the grid of parameters chosen for our study will affect the results obtained. Because our intention is not to obtain the best set of parameters for any of the estimators but to compare the accuracy of the different pruning strategies defined, we are going to use a simple grid. The grid used for both BS and WEP experiments is defined in table (4.1.1) (from now on, it will be called simply *grid*).

However, as we will observe later, sometimes the performance of the predictor with a pruned dataset is superior to the accuracy of the same predictor with the entire dataset. Hence we will use a secondary grid, defined in (4.1.2) (from now on, it will be called *extended grid* or *egrid*), to compare the new accuracy with the one we could obtain with a bigger grid. This second grid contains the initial one as a subset and duplicates the number of options for each parameter. Then for RFR and GBR while the normal grid contains 32 sets of parameters, the extended grid contains $2^5 \cdot 32 = 1024$ sets of parameters. The random state parameter has been fixed to 0 for all the experiments. A brief explanation of each parameter can be found in the Appendix. The Appendix contains also a brief representation of the distribution of error and time cost for each grid and for each regressor.

The **F**eature **R**elevance (FR) metrics used for the estimators are the ones included in *Scikit* and defined in detail in the theory chapter.

Problem	Splitter	max_depth	max_features	min_s_split	min_s_leaf	n_estimators
DTR	best	[5, 10, None]	[0.3, 0.4, 0.5]	[4, 8, 16]	[2, 4, 8]	
RFR	best	[5, 10]	[0.3, 0.5]	[4, 8]	[2, 4]	[200, 400]
GBR	ls	[2, 4]	[0.3, 0.5]	[4, 8]	[2, 4]	[200, 400]

Table 4.1.1: BS: Grid of parameters for BS problem

Problem	Splitter	max_depth	max_features
DTR	[best, random]	[5, 6, 8, 10, 12, None]	[0.3, 0.4, 0.5, 0.6, 0.7, 0.8]
RFR	best	[5, 8, 10, 12]	[0.3, 0.4, 0.5, 0.6]
GBR	ls	[2, 4, 6, 8]	[0.3, 0.4, 0.5, 0.6]

Problem	min_s_split	min_s_leaf	n_estimators
DTR	[4, 8, 16, 32, 64, 128]	[2, 4, 8, 16, 32, 64]	
RFR	[4, 8, 16, 32]	[2, 4, 8, 16]	[200, 400, 600, 800]
GBR	[4, 8, 16, 32]	[2, 4, 8, 16]	[200, 400, 600, 800]

Table 4.1.2: Extended grid of parameters for BS problem

The topics covered for the BS problem are:

- Presentation of the problem.
- Graphical linear block pruning using FR (backward stepwise selection).
- Time cost of searching in a pruned dataset.

The topics covered for the WEP problem are:

- Presentation of the problem.
- Graphical linear block pruning using FR (backward stepwise selection).
- Analytical comparative of RFE, SRFE and Boruta.
- Quality of the search in a pruned dataset.

4.2 Bike Sharing problem

The BS problem can be stated as follow:

Given some environmental and seasonal conditions, like the day of the week, the month, the season, the weather and probability of rain, predict the number of bikes rented.

The dataset we are going to use is derived from two year of real data (2011 and 2012) from *Capital Bike share system, Washington D.C., USA* and available at *UCI*. The data is aggregated on hourly and daily basis and then the weather and seasonal information, obtained from an *external weather forecaster* is added.

Besides, the public dataset includes two files, one with daily aggregation and a second one with hourly aggregation. We have used the one with hourly aggregation because we wanted to add also the time variable *hour* and the problem is still easy to manage with all the samples. The daily aggregation includes 731 instances with 15 variables. The hourly aggregation includes 17379 instances with 16 variables.

The *info* file from the repository gives us a good description of the variables:

---

1. dteday : date
2. season : season (1:spring, 2:summer, 3:fall, 4:winter)
3. yr : year (0: 2011, 1:2012)
4. mnth : month (1 to 12)
5. hr : hour (0 to 23). Only in hourly aggregation
6. holiday : weather day is holiday or not
7. weekday : day of the week
8. workingday : if day is neither weekend nor holiday is 1, otherwise is 0.
9. weathersit :
   - 1: Clear, Few clouds, Partly cloudy, Partly cloudy
   - 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
   - 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
   - 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
10. temp : Normalized temperature in Celsius. The values are divided by 41 (max)
11. atemp: Normalized feeling temperature in Celsius. The values are divided by 50 (max)
12. hum: Normalized humidity. The values are divided by 100 (max)
13. windspeed: Normalized wind speed. The values are divided by 67 (max)
14. casual: count of casual users
15. registered: count of registered users
16. cnt: count of total rental bikes including both casual and registered

---

The dataset we have used in the experiments is a small modification of the above described. We can summarize the changes as:

- *dteday* is now the number of days since the initial day of the system (instead of a date)
- We use *cnt* as value to predict and we remove *casual* and *registered.*
- We normalize the data in the interval $(0, 1)$ with a linear transformation.

In general terms, we know that $cnt = casual + registered$. As we are interested in predicting only one variable we have chosen *cnt* for the general prediction problem and we have removed the other two options, which could have been also good elections. However, we still use and want to understand the meaning of both variables, *casual* and *registered*, because the degree to which each feature affects each the previous variables will be different and will help us to visualize the degree of influence.

For example, *registered* will be more related with variables like *dteday*, *year* or *month*, whereas *casual* probably has a strongly relation with environmental factors like *weathersit*, *temp*, *hum*, *windspeed* and also with periodical factors like *holiday* or *workingday*.

We have used several methods to compute the total number of bikes rented. To determine the optimal values for each method we have split randomly the original dataset in three smaller ones of equal size, each one used for training, validation and testing respectively. The collection of optimal **parameters** for the grid is given in table (4.2.1).

Problem	Splitter	max_depth	max_features	min_s_split	min_s_leaf	n_estimators
DTR	best	None	0.5	4	4	
RFR	best	10	0.5	4	2	200
GBR	ls	4	0.5	8	2	200

Table 4.2.1: BS: Optimal parameters (grid)

4.2.1 Distribution of feature relevance

This **section shows the feature relevance distribution for the BS problem**. The BS problem has a main variable to predict, *cnt*, but as we have stated above includes also other two variables, *casual* and *registered*, which can be useful to illustrate the feature relevance. Hence, we have three prediction problem, which we name using the variable to predict. Each row of images in figure (4.2.1) contains the feature relevance for the variable *casual*, *registered* and *cnt* respectively. The images contain the feature relevance for all the set of parameters of the (standard) grid in blue, and the feature relevance for the best set of parameters in red.

Considering each regressor, the **first** (left) image of each row corresponds to the FR of a DTR regressor. In general we observe that the distribution is not stable, obtaining a different feature relevance distribution for different sets of parameters. The **second** and **third** images correspond to the FR of a RFR and GBR regressor respectively. In opposition to the previous situation, we can state that the feature relevance is stable and we can take any parameter to obtain a relevance distribution of each variable.

Considering each prediction problem, the feature relevance distribution for the *casual* problem is a bit different to the one we obtain for *registered* and *cnt*. In these last two situations, the distributions are similar (especially for RFR and GBR). Hence, as $cnt = registered + casual$, we can state that *registered* has a deeper effect in the relevance distribution of *cnt* (both problems are similar).

Apart from the previous analysis about the stability of the distributions, we can analyze if the obtained results agree with our sense of importance. For all the distributions, then main variable is 5 (hour). As we expect, this variable will condition heavily the number of riders. For the *casual* problem, the main variables are 5 (hour), 7 (weekday), 8 (workingday), 10 (temp) and 11(atemp), 12(hum), which as we expect will affect the number of casual riders. For the *registered* problem, the main variables are the same, but the importance of the non hour variables is much smaller. The *cnt* problem is a mixed situation of both previous.

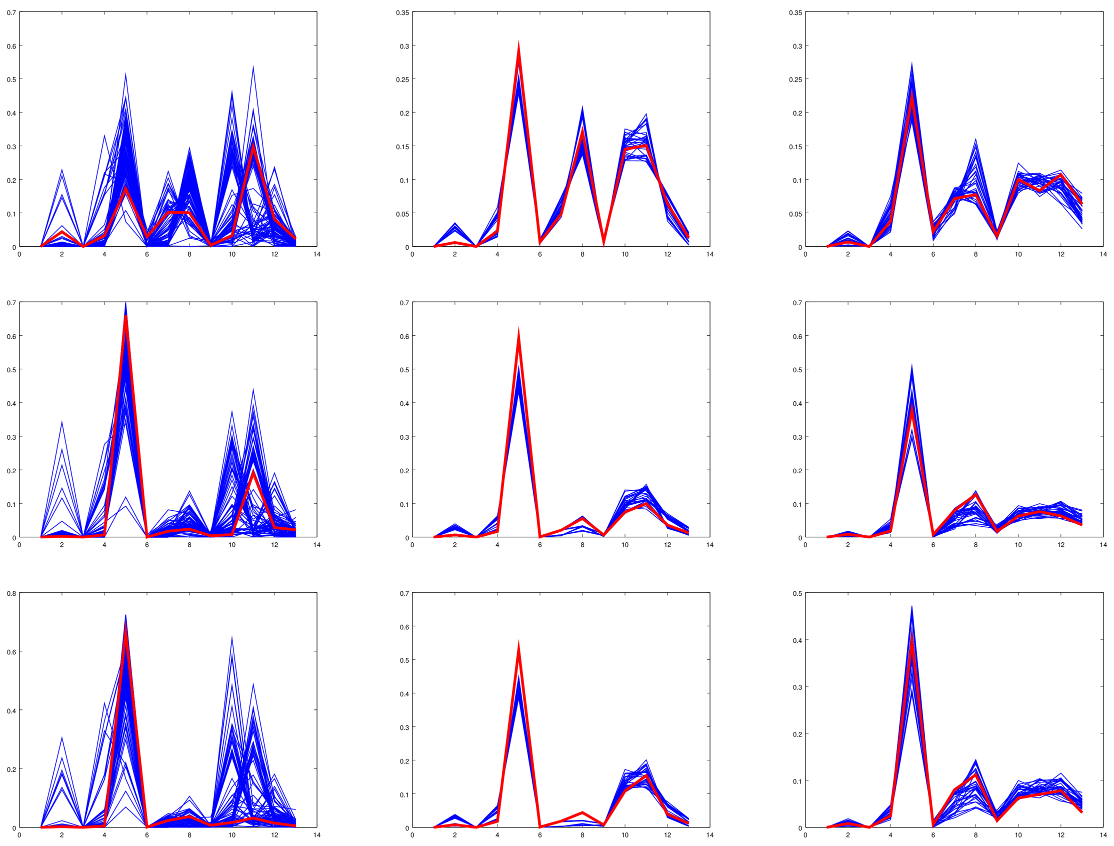


Figure 4.2.1: BS: DTR, RFR and GBR FR distribution (left-right). For *casual*, *registered*, *cnt* problem (top-down).

4.2.2 Pruning the dataset using feature relevance

This **section shows the evolution of the error pruning the dataset features for the BS problem**. One of our pretensions is using feature relevance to prune the original dataset, removing the less informative columns and comparing the results with the original problem. For that, we have analyzed the evolution of the time and error after removing some features. For pruning the dataset, we have removed in each step the 10% [1] of the initial number of features and compute the time needed (used to train and predict three times, the errors) and the errors for the train, test and validation datasets [2].

Each row of images in figure (4.2.2) represents DTR, RFR and GBR respectively (top-down). Each column the time and the train, test and validation errors respectively (left-right). Each of the images contains the evolution of time or errors for each dataset for all the parameters in the standard grid. The red curve shows the best set of parameters whereas the black one averages for all the parameters. **Each error graph in a row of graphs can be interpreted from left to right as adding features to an initial feature (hour variable) or from right to left as pruning the initial dataset**.

The **first** image (top and left) in figure (4.2.2) shows us the evolution of the time with a DTR regressor. In general terms we observe a non very regular behaviour and a bigger

[1]In each step we remove 0, 1 or 2 variables.

[2]Train, test and validation datasets are generated after shuffling the original dataset, with a fixed seed, and dividing the resulting new dataset into three new datasets of equal size. Then each new dataset has $\frac{1}{3}$ of the original one. The initial dataset if shuffled because of the time related variables.

cost in time as we add more features. The main reason for this non regularity can be the non similarity between different feature relevance distributions and the *Scikit* strategy for reducing computation. Besides, in each step we can be removing different variables and a different number of them (0, 1 or 2). This behaviour still persists for RFR and GBR, but it is much smoother. The rest of images in the same row show us the train, test and validation error respectively. For similar reasons the behaviour is more complex that the one we obtain for RFR and GBR, which is threaded using subsets of parameters from our grid.

The **second** row of images, corresponding to RFR, shows a smoother behaviour. As a consequence of the stability of the feature relevance distribution, each of the curves follows a similar path. Besides, in the error graphs we observe a threaded conduct. The main reason for that is the stopping criterion. Some configuration parameters for the Trees act stopping the growth in most of the leaves whereas others are scarcely used to create a leave. Hence those sets of parameters which share similar stopping criteria will have similar structure despite the other parameters. As a consequence their errors and quality as predictors will be similar.

The **third** row of images, corresponding to GBR, is similar to RFR.

With both ensemble estimators, RFR and GBR, with the 40% of the initial features we obtain in test (third image of the row) approximately a *similar error* to the one we obtain with all the variables. In fact with GBR we only need the 20%. Hence we can obtain a *near optimal* result with just a fraction of time and memory resources [3].

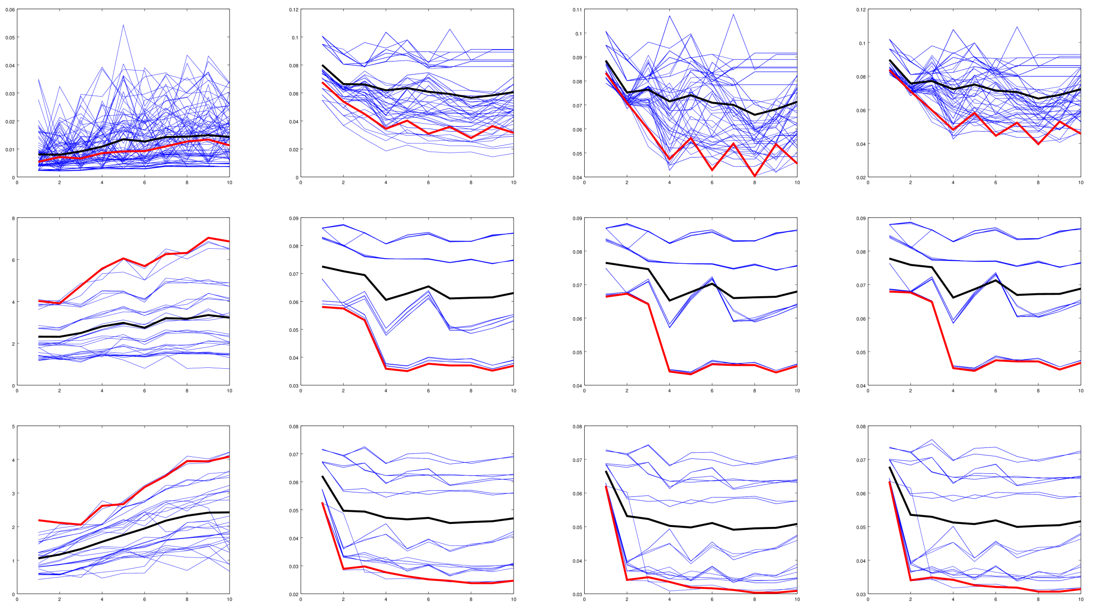


Figure 4.2.2: BS: Pruning dataset with FR. Time, train, test and val errors (left-right). For DTR, RFR, GBR (top-down).

4.2.3 Searching

Our last point to analyze with the BS problem is the utility of pruning for parameter search. Until this moment we have pruned the dataset to discover the evolution of time and error, but we have not used the pruned dataset to obtain any gain. Bearing in mind

[3]More precise details about the exact meaning of *near optimal* will be given for our WEP problem.

that we can obtain a good prediction error with 20% or 40% of our variables, we are going to prune our dataset keeping only $round(0.2 * 13) = 3$ and $round(0.4 * 13) = 5$ variables and analyze the new times for computing the grid and the error for prediction.

Our variables will be, in order, 5 (hour), 8 (workingday), 11 (atemp), 10 (temp) and 12 (hum). We now have three problems: the normal BS problem, the restricted BS problem with only columns {5,8,11,10,12} and the super restricted BS problem with only {5,8,11} columns.

As we observe in tables (4.2.2) and (4.2.3), with obtain decent reductions of times for both the *grid* and the *egrid* [4] , especially for GBR.

Method	{5,8,11}	{5,8,11,10,12}	All
DTR	0.118	0.120	0.220
RFR	8.554	9.741	14.753
GBR	3.326	4.71	10.845

Table 4.2.2: BS: Time(s) needed with a pruned dataset to compute the **grid**

Method	{5,8,11}	{5,8,11,10,12}	All
DTR	1.523	2.124	4.227
RFR	460.213	554.458	852.523
GBR	228.272	343.378	773.453

Table 4.2.3: BS: Time(s) needed with a pruned dataset to compute the **egrid**

As we expected from our theoretical analysis, the reduction is approximately linear in the number of features. Although the reduction of times is modest (a factor of the original value in the interval $(0.3, 0.5)$) and we cannot avoid a careful selection of the parameters when searching using a grid, we can *halve* easily the time needed in our toy example just running a simple GBR, with any clever set of parameters from our grid, removing those variables with a relevance under a threshold value (keeping for example the 20% of them) and running the search in our pruned dataset, being the accuracy result of this search at least as good as the accuracy of the predictor with our seed and the whole dataset, as we will see in the following section.

After this informal description of feature relevance with DTR, RFR and GBR and the utility and evolution of errors in a pruned dataset, we are going to use the techniques described in the previous chapter to obtain a more analytical approach with our WEP problem.

[4]Parallelized with 8 threads.

4.3 Wind Energy Prediction problem

We will work with the Numerical Weather Prediction (NWP) system of the European Centre for Medium-Range Weather Forecasts (ECMWF). It currently provides variable forecasts over a global world grid with a 0.125 resolution although we work on a 0.25 aggregated one. To predict wind energy of Sotavento we consider a 15×9 rectangular sub grid approximately centered on the Sotavento site (43.34 N, 7.86 W). The ECMWF meteorological variables used will be the following:

1. U_{100}, the x wind component at 100 m height.
2. V_{100}, the y wind component at 100 m.
3. P, the pressure at surface level.
4. T, the temperature at 2 m.
5. U_s, the x wind component at surface level.
6. V_s, the y wind component at surface level.
7. v_{100}, the wind norm at 100 m.
8. v_s, the wind norm at surface level.

We point out that surface and $100m$ height refers to the geopotential of the grid point for which these variables are provided; in turn, the grid node geopotential is that of the concrete orography model used, which may or may not coincide with the actual geographical point with the same coordinates. Input dimensions are thus quite large, $15 \times 9 \times 8 = 1080$. We will work with data for the years 2011, 2012 and 2013. Wind energy data for Sotavento can be obtained through their web site. We normalize productions to the $[0, 1]$ interval by dividing them by the rate power of Sotavento (i.e., the maximum power the farm would provide). In other words, at each hour we will work with the percentage of energy actually produced with respect to the maximum possible values. While hourly values are available for wind energy, NWP forecasts are available only every three hours, starting at UTC hour 00. Thus, in every year we will approximately have $(24/3) \cdot 365 = 2920$ patterns. In what follows we will refer to this as the Sotavento problem [5].

The parameters used are given in table (4.3.1). To determine the optimal values for each method we have used the year 2011 to train the method (training set), the year 2012 to determine the optimal parameters (validation set) and the year 2013 to estimate the test error (test set). As stated previously, we have used a simple grid of parameters because our intention is to show the power of feature selection supposing we have a huge dataset and we cannot compute a big number of parameter configuration for practical reasons. Some results concerning the extended grid are also given.

Problem	Splitter	max_depth	max_features	min_s_split	min_s_leaf	n_estimators
DTR	best	5	0.4	4	8	
RFR	best	10	0.3	8	2	400
GBR	ls	4	0.5	4	4	200

Table 4.3.1: Sotavento: Optimal parameters for Sotavento problem in the grid (*random state is 0*)

4.3.1 Distribution of feature relevance

The first point to study is the distribution of relevance and their stability. The figure (4.3.1) show the distribution of relevance for DTR, RFR and GBR respectively.

[5]Description taken from [24].

For the **first image in each row**, as in the previous situation, the red curve is the one associated with the best set of parameters, whereas the blue ones represent all the sets of parameters from the grid (simultaneously, it is not a mean). The x axis is the list of 1080 features whereas the y represents the feature relevance associated to each of them [6] . In general terms, we can affirm that RFR shows the most stable distribution, while the ones from DTR and GBR depend more on the parameters.

The **second and third images** in each row show us the mean feature relevance distribution for the 8 variables and the 135 geographical point for the optimal set of parameters. The second image contains the feature relevance for each one of the eight previously defined atmospherical attributes associated (the components of pressure, temperature and wind) aggregating the 135 geographical points for each variable, in the order defined above. The most relevant feature is v_{100}, the wind norm at $100m$. The third image contains the relevance of each of the 135 geographical point from the physical grid aggregating the relevance of all the (8) features for this point.

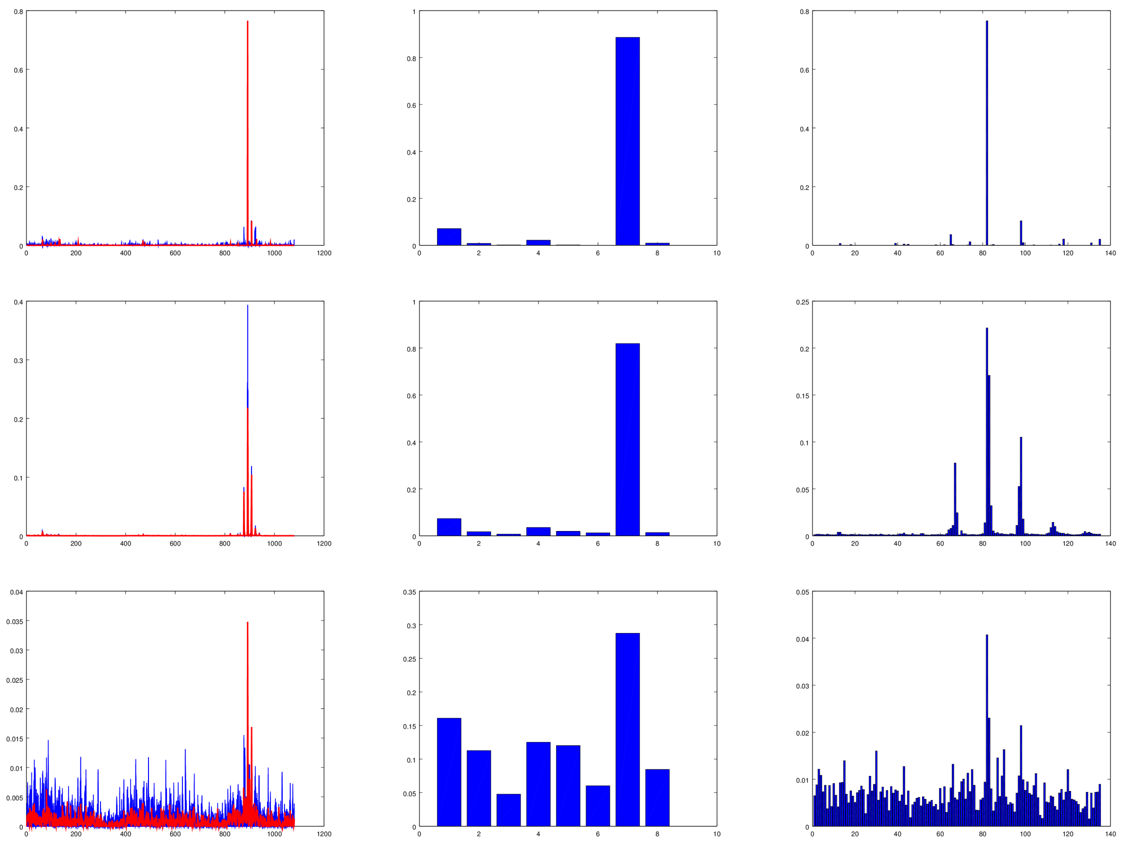


Figure 4.3.1: Sotavento: DTR, RFR and GBR (top-down) coefficient distribution

4.3.2 Pruning the dataset with FR

As stated previously, our main idea is using feature relevance to prune the original dataset with the intention of simplifying the original problem. For that, we have analyzed the evolution of the time and error after removing blocks of features.

[6]The variable at position $135 \cdot i + j$, $i = 0, \ldots, 7$ and $j = 0, \ldots, 134$ represents the i variable at the j point of the grid.

The figures (4.3.2), (4.3.3) and (4.3.4) show us the evolution of time and error for each member of the standard grid when we prune the original dataset. The first image of each row represents the time evolution, while the following three depict the training, testing and validation error evolution respectively. Each of the curves represents a set of parameters from the (standard) grid. The red one is the optimal set of parameters from the grid and the black one the mean from all the curves.

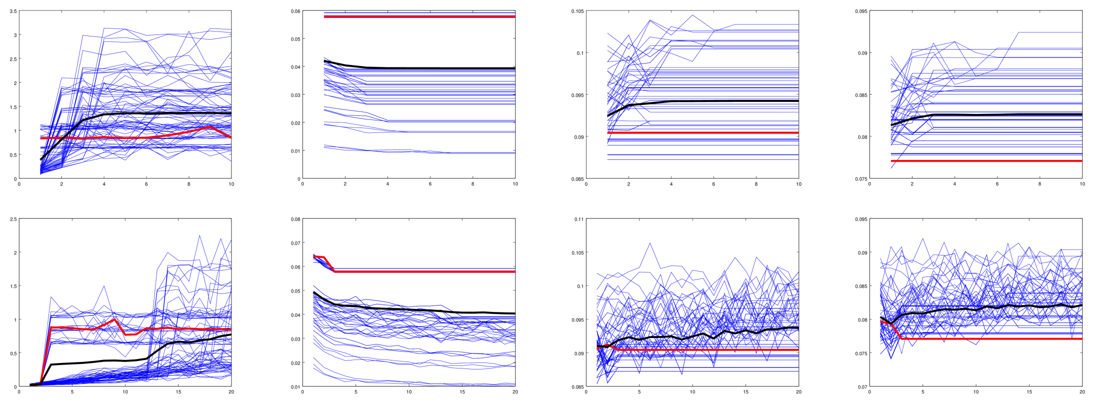


Figure 4.3.2: Sotavento: DTR, Pruning dataset with FR. Time, train, test and val errors

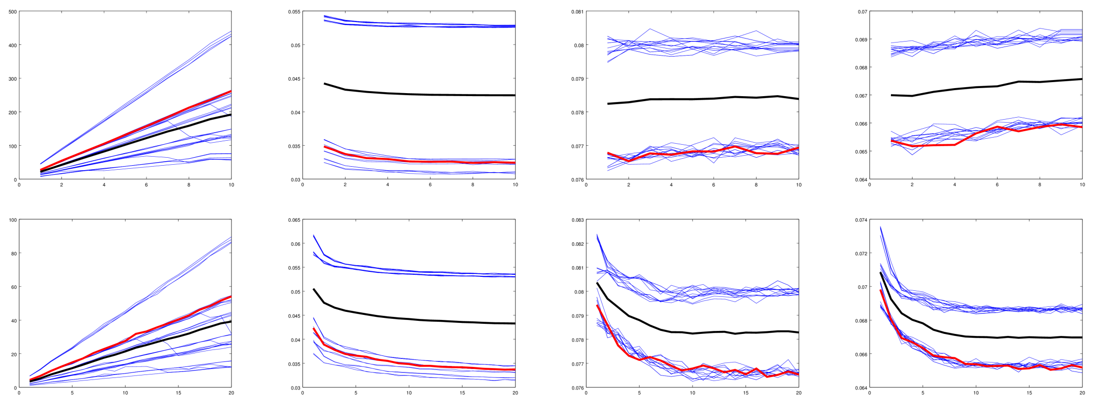


Figure 4.3.3: Sotavento: RFR, Pruning dataset with FR. Time, train, test and val errors

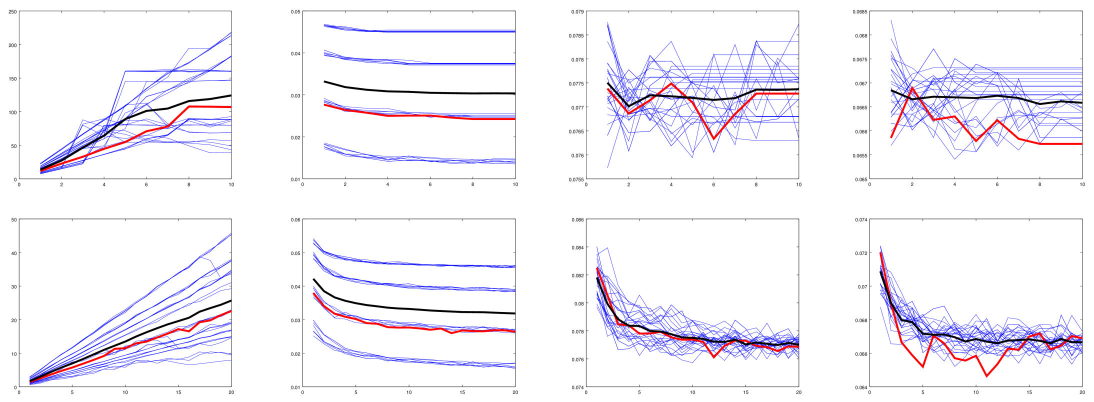


Figure 4.3.4: Sotavento: GBR, Pruning dataset with FR. Time, train, test and val errors

Each predictor technique (DTR, RFR and GBR) contains eight images. The first line for each method contains the evolution when we prune the interval of variables $[0, 100\%]$ with blocks of 10% of the features (approximately we prune 108 variables at each step). The second line represents the interval $[0, 20\%]$ (i.e. we work with only the 20% of the most relevant variables, ignoring the 80% less relevant), pruning in each step 1% of the variables (approximately 10 attributes at each step). **Each graph can be interpreted from left to right as adding features to an initial set of variables or from right to left as pruning the initial dataset**.

The **first** images (left) in the first block of images, (4.3.2), from DTR shows us the evolution of the time. DTR times show some noise consequence of the elected features and the building process from *Scikit* (optimized to reduce cost). The rest of images in the same row show us the train, test and validation error respectively. For DTR the evolution in the interval $[0, 20\%]$ is a bit chaotic, but later it stabilizes.

The **second** block images, (4.3.3), from RFR shows a smoother behaviour. As we expected the time evolution is linear. In the error graphs we observe a threaded conduct as a consequence of the stopping criteria. As stated above, those sets of parameters which share similar stopping criteria will have similar structure despite the other parameters. Hence, their errors and quality as predictors will be similar. From a practical point of view, we observe in the second row of graphs as the test and validation errors achieve certain stability at 10% of the features. From the first row of images, we can observe that the test and validation errors are approximately a monotone increasing function .

The **third** row of images, (4.3.4), corresponding to GBR is similar to RFR. We should point however some differences: the threaded behaviour is bigger and the testing and validation error a bit more chaotic. In the second row of images, we observe that the testing error seems to stabilize at 20% of the features and the validation error at 15%. Here the analysis is more complex than in the previous situation as we do not have a smooth pattern.

As we observe, whereas we cannot obtain any general result for DTR as a consequence of the lack of stability in the error curves, we can obtain a general approximately decreasing behaviour for RFR (specially) and GBR in $[0, 20\%]$ for any seed, which will be useful to prune the dataset in the following section.

4.3.3 Pruning

In the previous sections, we have analyzed the distribution of relevance and the loss of accuracy pruning the datasets with a collection of parameters and increasing steps. With this information for a dataset and predictor, we can prune the original data and obtain a simpler version of the predictor. However, in a practical environment we do not have neither the time nor the resources to execute an analysis about the distribution of features for different parameters and the evolution of the error when pruning features. Hence, our intention is to compare the techniques to prune the dataset *automatically.* Our candidates are the strategies defined at the end of the previous chapter.

1. Recursive Feature Elimination (RFE)
2. Subset RFE
3. Boruta

This section and the following contains the **main ideas of the present thesis**:

- ***Prune*** *the original dataset with an algorithm which computes the best subset of features* ***to simplify the construction of a predictor and improve its accuracy.***
- ***Prune*** *the original dataset with an algorithm which computes the best subset of features* ***to perform a seach of parameters and build the best possible predictor.***

Here *best* can have different meanings, because each of the techniques attacks the problem with different hypothesis. For example, RFE and Subset RFE get the *best* subset of variables of a fixed size (best here means with the bigger amount of relevance). On the other hand, statistical procedures like Boruta compute the subsets of *all the relevant variables* (i.e. all those features more relevant than a random one).

Once we know from the previous section that the general behaviour of the test and validation error is approximately uniformly decreasing in the interval $[0, 20\%]$, we have selected a simple parameter set from the standard grid to obtain more numerical results. The parameter set is:

```
{random_state : 0, n_estimators : 200, max_depth : 10 for RFR and 4 for GBR,
 max_features : 0.3, min_samples_split : 8, min_samples_leaf : 4}
```

From now on, this parameter set will be name *seed*, and it will be used to obtain the feature relevance distribution used for pruning the dataset features. Results for any other parameter set from the grid can be consulted in the repository added in the Appendix.

RFE

As we know, the main idea of Recursive Feature Elimination (RFE) is to build a sequence of datasets, each one containing a fraction of the features of the previous one, and errors associated with each simplified version of the dataset.

We are going to use a slightly modified version of the previous strategy. We use the validation dataset to choose the optimal new training dataset, instead of *oob* error. Besides, we are using a regression problem (instead of the classification of genes described in [20]) and we are using *MSE* impurity as a function for variable importance, instead of loss of accuracy from permutations.

We have used two pruning percentages, $\lambda = 0.5$ and $\lambda = 0.8$. For Sotavento dataset and RFR we obtain the results from tables (4.3.2) and (4.3.4) respectively, whereas for GBR we obtain the results from tables (4.3.3) and (4.3.5) respectively.

We analyze initially RFR. Depending on whether we prefer small dataset sizes or good predictor quality, we define the threshold value α [7]. If we use as criteria the one described in [20] [8] , observing table (4.3.2) we obtain that we should accept the 1.5625% of the features for $\lambda = 0.5$, i.e. only 16 features, and observing table (4.3.4) the 1.801% of the features for $\lambda = 0.8$, i.e. only 19 features. If we use instead a more conservative value as $\alpha = 0$ std (the value which minimizes the validation error), we need the 12.5% of the features for $\lambda = 0.5$, i.e. only 135 features, and the 16.777% of the features for $\lambda = 0.8$, i.e. only 181 features.

We analyze now GBR. With $\alpha = 1$, from table (4.3.3) we obtain that we should accept the 1.5625% of the features for $\lambda = 0.5$ (16 features) and from table (4.3.5) we observe that we need the 1.441% of the features for $\lambda = 0.8$ (15 features). If we use instead a more conservative value as $\alpha = 0$ std, we obtain we need the 12.5% of the features for $\lambda = 0.5$ (135 features), and the 8.590% of the features for $\lambda = 0.8$ (92 features). As we observe from the monotony trend of the *Perc std* column, the behaviour of GBR is a bit more complex than RFR.

Then, for $\lambda = 0.5$, we can summarize the results in tables (4.3.6) and (4.3.7). For $\lambda = 0.8$, we can summarize the results in tables (4.3.8) and (4.3.9). The tables include for

[7] Fraction of the standard deviation of validation error, within the minimum, allowed.

[8] Which is considering the dataset with the smallest number of features within one standard deviation from the minimum validation error, $\alpha = 1$.

Percentage	Num var	Train	Test	Val	Perc std
100	1080	3.296	7.675	6.601	0.142
50	540	3.305	7.676	6.562	0.043
25	270	3.404	7.666	6.550	0.012
12.500	135	3.536	7.685	6.545	0
6.250	67	3.707	7.727	6.589	0.113
3.125	33	3.932	7.796	6.725	0.456
1.562	16	4.193	7.898	6.930	0.975
0.781	8	4.721	8.001	7.148	1.527
0.391	4	5.469	8.430	7.595	2.659
0.195	2	5.552	8.496	7.563	2.576

Table 4.3.2: Sot, RFE, RFR, $\lambda = 0.5$, error pruning the dataset (%)

Percentage	Num var	Train	Test	Val	Perc std
100	1080	2.469	7.679	6.586	0.019
50	540	2.524	7.629	6.580	0.005
25	270	2.641	7.729	6.697	0.267
12.500	135	2.768	7.709	6.578	0
6.250	67	3.001	7.823	6.731	0.343
3.125	33	3.279	7.922	6.698	0.268
1.562	16	3.764	8.231	6.931	0.791
0.781	8	4.337	8.153	7.093	1.156
0.391	4	5.290	8.550	7.749	2.626
0.195	2	5.515	8.710	7.814	2.771

Table 4.3.3: Sot, RFE, GBR, $\lambda = 0.5$, error pruning the dataset (%)

Percentage	Num var	Train	Test	Val	Perc std
100	1080	3.296	7.675	6.601	0.218
80	864	3.300	7.695	6.581	0.166
64	691	3.310	7.702	6.588	0.186
51.200	552	3.327	7.663	6.572	0.145
40.960	442	3.347	7.688	6.539	0.060
32.768	353	3.371	7.680	6.551	0.092
26.214	283	3.383	7.667	6.538	0.060
20.972	226	3.433	7.650	6.532	0.044
16.777	181	3.469	7.661	6.515	0
13.422	144	3.540	7.661	6.525	0.026
10.737	115	3.577	7.684	6.533	0.047
8.590	92	3.628	7.696	6.564	0.124
6.872	74	3.690	7.699	6.579	0.163
5.498	59	3.780	7.719	6.646	0.331
4.398	47	3.818	7.750	6.664	0.377
3.518	37	3.909	7.842	6.732	0.549
2.815	30	3.942	7.801	6.732	0.548
2.252	24	4.020	7.788	6.778	0.665
1.801	19	4.102	7.789	6.823	0.777
1.441	15	4.335	7.911	6.999	1.222
1.153	12	4.461	7.938	6.997	1.217
0.922	9	4.521	7.942	7.081	1.430
0.738	7	4.785	8.047	7.185	1.693
0.590	6	4.983	8.301	7.496	2.478
0.472	5	5.396	8.398	7.618	2.785
0.378	4	5.469	8.430	7.595	2.728
0.302	3	5.431	8.441	7.575	2.678
0.242	2	5.552	8.496	7.563	2.646
0.193	2	5.552	8.496	7.563	2.646

Table 4.3.4: Sot, RFE, RFR, $\lambda = 0.8$, error pruning the dataset (%)

Percentage	Num var	Train	Test	Val	Perc std
100	1080	2.469	7.679	6.586	0.096
80	864	2.469	7.679	6.586	0.096
64	691	2.510	7.639	6.680	0.325
51.200	552	2.545	7.681	6.613	0.160
40.960	442	2.572	7.757	6.652	0.255
32.768	353	2.592	7.613	6.551	0.010
26.214	283	2.603	7.632	6.651	0.253
20.972	226	2.667	7.790	6.663	0.282
16.777	181	2.719	7.680	6.615	0.166
13.422	144	2.733	7.706	6.689	0.346
10.737	115	2.739	7.853	6.612	0.158
8.590	92	2.825	7.833	6.547	0
6.872	74	2.914	8.026	6.716	0.412
5.498	59	3.087	7.846	6.748	0.490
4.398	47	3.202	7.958	6.768	0.537
3.518	37	3.310	7.858	6.682	0.327
2.815	30	3.356	7.996	6.782	0.572
2.252	24	3.498	8.077	6.894	0.844
1.801	19	3.599	8.209	6.920	0.908
1.441	15	3.824	8.155	6.925	0.920
1.153	12	4.070	8.127	7.027	1.169
0.922	9	4.181	8.208	7.050	1.226
0.738	7	4.500	8.270	7.232	1.668
0.590	6	4.679	8.341	7.362	1.985
0.472	5	5.118	8.595	7.514	2.356
0.378	4	5.290	8.550	7.749	2.929
0.302	3	5.351	8.597	7.734	2.892
0.242	2	5.515	8.710	7.814	3.086
0.193	2	5.515	8.710	7.814	3.086

Table 4.3.5: Sot, RFE, GBR, $\lambda = 0.8$, error pruning the dataset (%)

reference the error rates obtained witht the whole dataset and the selected set of parameters (*seed* row).

α	Num var	Train	Test	Val
1	16	4.193	7.898	6.930
0.5	33	3.932	7.796	6.725
0.25	67	3.707	7.727	6.589
0	135	3.536	7.685	6.545
seed	1080	3.296	7.675	6.601

Table 4.3.6: Sot, RFE, RFR, $\lambda = 0.5$, summary error pruning the dataset (%)

α	Num var	Train	Test	Val
1	16	3.764	8.231	6.931
0.5	33	3.279	7.922	6.698
0.25	135	2.768	7.709	6.578
0	135	2.768	7.709	6.578
seed	1080	2.469	7.679	6.586

Table 4.3.7: Sot, RFE, GBR, $\lambda = 0.5$, summary error pruning the dataset (%)

α	Num var	Train	Test	Val
1	19	4.102	7.789	6.823
0.5	47	3.818	7.750	6.664
0.25	74	6.664	7.699	6.579
0	181	3.469	7.661	6.515
seed	1080	3.296	7.675	6.601

Table 4.3.8: Sot, RFE, RFR, $\lambda = 0.8$, summary error pruning the dataset (%)

α	Num var	Train	Test	Val
1	15	3.824	8.155	6.925
0.5	37	3.310	7.858	6.682
0.25	115	2.825	7.833	6.547
0	115	2.825	7.833	6.547
seed	1080	2.469	7.679	6.586

Table 4.3.9: Sot, RFE, GBR, $\lambda = 0.8$, summary error pruning the dataset (%)

These results depend on the seed taken, especially for GBR. However, as we know from our analysis above about pruning the dataset in the interval $[0, 20\%]$ in blocks of 1%, the results follow the same monotone trend with any seed taken from our grid. Detailed results can be consulted in the repository from the Appendix.

Subset RFE (SRFE)

As we mentioned in the previous chapter, SRFE is technique based on the same idea of RFE: prune recursively the dataset features. The main difference is that SRFE divides the initial dataset in smaller ones (randomly) and performs a normal RFE in all of them, adding hence an additional source of randomness [9] and a new parameter to tune (the size of the subsets). Then, we can expect results slightly different to RFE. We have used the same α criteria defined for RFE to consider the optimal dataset size.

The tables (4.3.10), (4.3.11), (4.3.12) and (4.3.13) contain the error values with our seed for RFR and GBR, respectively, with subsets of size $c = 54$ elements $(20 \cdot 54 = 1080)$. The prune rate for tables (4.3.10) and (4.3.11) is $\lambda = 0.5$. For tables (4.3.12) and (4.3.13) is $\lambda = 0.8$. The tables have the same structure and meaning than RFE.

The table (4.3.14), (4.3.15), (4.3.16) and (4.3.17) contain the the summary of results, as done for RFE.

[9] We can control the results fixing a *random state* for the random generator, as we have done with others.

Percentage	Num var	Train	Test	Val	Perc std
100	1080	3.296	7.675	6.601	2.474
50	540	3.352	7.695	6.550	1.210
25	270	3.463	7.672	6.502	0
12.5	135	3.541	7.701	6.529	0.685
6.25	68	3.734	7.702	6.604	2.558

Table 4.3.10: Sot, SRFE, RFR, $\lambda = 0.5$, error pruning the dataset (%)

Percentage	Num var	Train	Test	Val	Perc std
100	1080	2.469	7.679	6.586	0
50	540	2.575	7.707	6.659	0.689
25	270	2.687	7.794	6.645	0.556
12.5	135	2.774	7.778	6.852	2.501
6.25	68	2.951	7.604	6.830	2.292

Table 4.3.11: Sot, SRFE, GBR, $\lambda = 0.5$, error pruning the dataset (%)

Percentage	Num var	Train	Test	Val	Perc std
100	1080	3.296	7.675	6.601	3.246
80	880	3.285	7.717	6.586	2.693
64	717	3.315	7.678	6.573	2.168
51.200	585	3.312	7.700	6.567	1.965
40.960	477	3.346	7.677	6.548	1.248
32.768	389	3.371	7.668	6.522	0.245
26.214	317	3.395	7.689	6.564	1.859
20.972	258	3.438	7.687	6.552	1.397
16.777	210	3.451	7.708	6.532	0.644
13.422	171	3.494	7.700	6.515	0
10.737	140	3.523	7.681	6.528	0.482
8.590	114	3.561	7.693	6.540	0.947
6.872	93	3.615	7.698	6.543	1.033
5.498	76	3.691	7.695	6.569	2.040
4.398	62	3.736	7.734	6.603	3.314

Table 4.3.12: Sot, SRFE, RFR, $\lambda = 0.8$, error pruning the dataset (%)

Percentage	Num var	Train	Test	Val	Perc std
100	1080	2.469	7.679	6.586	0.593
80	880	2.564	7.716	6.553	0
64	717	2.564	7.670	6.626	1.292
51.200	585	2.566	7.650	6.613	1.054
40.960	477	2.563	7.693	6.608	0.979
32.768	389	2.637	7.669	6.640	1.537
26.214	317	2.658	7.603	6.620	1.181
20.972	258	2.698	7.663	6.638	1.506
16.777	210	2.751	7.668	6.599	0.821
13.422	171	2.782	7.606	6.637	1.480
10.737	140	2.812	7.642	6.676	2.163
8.590	114	2.848	7.666	6.712	2.804
6.872	93	2.876	7.658	6.697	2.540
5.498	76	2.895	7.650	6.726	3.053
4.398	62	2.949	7.779	6.772	3.864

Table 4.3.13: Sot, SRFE, GBR, $\lambda = 0.8$, error pruning the dataset (%)

α	Num var	Train	Test	Val
2	135	3.541	7.701	6.529
1	135	3.541	7.701	6.529
0.5	270	3.463	7.672	6.502
0	270	3.463	7.672	6.502
seed	1080	3.296	7.675	6.601

Table 4.3.14: Sot, SRFE, RFR, $\lambda = 0.5$, summary error pruning the dataset (%)

α	Num var	Train	Test	Val
2	270	2.687	7.794	6.645
1	270	2.687	7.794	6.645
0.5	1080	2.469	7.679	6.586
0	1080	2.469	7.679	6.586
seed	1080	2.469	7.679	6.586

Table 4.3.15: Sot, SRFE, GBR, $\lambda = 0.5$, summary error pruning the dataset (%)

α	Num var	Train	Test	Val
2	93	3.615	7.698	6.543
1	114	3.561	7.693	6.540
0.5	140	3.523	7.681	6.528
0	171	3.494	7.700	6.515
seed	1080	3.296	7.675	6.601

Table 4.3.16: Sot, SRFE, RFR, $\lambda = 0.8$ (%), summary error pruning the dataset

α	Num var	Train	Test	Val
2	171	2.782	7.606	6.637
1	210	2.751	7.668	6.599
0.5	880	2.564	7.716	6.553
0	880	2.564	7.716	6.553
seed	1080	2.469	7.679	6.586

Table 4.3.17: Sot, SRFE, GBR, $\lambda = 0.8$, summary error pruning the dataset (%)

Boruta

In contrast with the previous situation, Boruta uses statistical tests to determine whether a feature is relevant or not, comparing the existing variables with newly created originated of permuting the original ones. The basic idea involves creating new shadow variables, taking the relevance of the most relevant of these new features and use this value as a threshold for comparison with the original ones. This process is repeated several times (*iterations*), classifying the variables as relevant, non-relevant or not classified.

Tables (4.3.18) and (4.3.19) contains the error values with our seed for RFR and GBR, respectively, after 100 iterations. The *perc* refers to the percentage of relevance of the most relevant shadow variable we consider to filter our variables. The column *Num var* refers to the number of included variables.

We observe that for RFR considering as $perc = 75\%$ and for GBR considering $perc = 25\%$ we obtain better test results that using our original whole dataset. If we use as selection pattern the strategy of choosing the *perc* which minimizes the validation error, then we choose $perc = 75\%$ (260 variables) for RFR and $perc = 50\%$ (73 variables) for GBR [10].

Perc	Num var	Train	Test	Val
100	110	3.552	7.706	6.544
75	260	3.405	7.668	6.542
50	388	3.379	7.664	6.544
25	466	3.107	7.662	6.557
seed	1080	3.296	7.675	6.601

Table 4.3.18: Sot, Boruta, RFR, 100 iter, error pruning the dataset (%)

Perc	Num var	Train	Test	Val
100	9	4.313	8.283	7.196
75	17	3.643	8.001	6.888
50	73	2.919	7.685	6.618
25	272	2.614	7.636	6.649
seed	1080	2.469	7.679	6.586

Table 4.3.19: Sot, Boruta, GBR, 100 iter, error pruning the dataset (%)

Comparative

For **RFE**, as we observe in tables (4.3.2) and 4.3.3) for RGR and GBR respectively, we can choose easily a value which improves the error rate for the test dataset, even with an aggressive prune rate ($\lambda = 0.5$). Tables (4.3.4) and 4.3.5) show the results for a finer prune rate ($\lambda = 0.8$). Here we observe that the validation error sequence and the *Perc std* sequence have a non monotone behaviour, having several local minima. If we use as selection criteria the one defined for RFE with $\alpha = 0$ (we choose the percentage *Perc std* lower or equal than the selected α and the lowest number of variables), we get better results for RFR but not for GBR. In general, for RFR we observe than we can get several elections which improve the results obtained with the whole dataset, whereas for GBR we observe a consistent deterioration of the error rates as we consider smaller sizes. A smoother criterion such as considering averages of the sequence *Perc std* can gives us better results.

As we can observe using any other parameter set (*seed*) from the repository in the Appendix, for RFR a value $\alpha \in [0, 0.5]$ leads to selection which improves the results obtained with the whole dataset, whereas for GBR we cannot obtain so optimistic general results.

For **SRFE**, we obtain similar results to the ones of RFE. Tables (4.3.10) and (4.3.11) contain the results for RFR and GBR, respectively, for $\lambda = 0.5$. Similar results, for $\lambda = 0.8$, can be consulted in (4.3.12) and (4.3.13). The different results between RFE and SRFE are due to the subsets for comparison. RFE can be interpreted as a limit situation ($c = 1080$) where there is only one subset of features in each step instead of several. As a consequence, the SRFE process stops when we get a final size equal to the size of the subset ($c = 54$ in

[10]The fractions of *perc* used are clearly not enough (probably we can get a better estimation for GBR) but give a simple idea. We should remember that each *perc* requires 100 iterations with a subset of features, being then the cost much higher than RFE and SRFE.

our examples) instead of a final size of 2 as in RFE. If we compare both strategies, SRFE and RFE, we observe than none of error sequences have a domination relation (i.e. there is not method globally better).

This behaviour is observed also for any other parameter set (*seed*) from the repository in the Appendix.

For **Boruta**, tables (4.3.18) and (4.3.19) contain the values for different thresholds of relevance. As we observe, there are great differences in the number of choosen variables for RFR and GBR, as a consequence of the difference relevance distribution given by RFR and GBR. Because of the main idea behind Boruta is getting all the relevant features (comparing with a shadow value of relevance given by new variables generated by permuting the original ones), it is not possible to determine the percentage of surviving variables after pruning. We can only control the percentage of shadow value we use to prune. However, empirically, good results can be obtained with fractions in the range $[0.25, 0.75]$ of the shadow value.

4.3.4 Grid Search After Pruning

Once we have pruned the dataset and obtained a numerical idea of the different approaches, we are going to prune the dataset and perform a search with the standard grid. For reference, the tables (4.3.20) and (4.3.21) contain the best results obtained after performing a search with both the standard grid (*grid*) and the extended grid (*egrid*) with the whole dataset (without pruning).

Source	Train	Test	Val
grid	3.243	7.693	6.585
egrid	2.876	7.652	6.557

Table 4.3.20: Sot, RFR, error using the grid, egrid (%)

Source	Train	Test	Val
grid	2.425	7.728	6.573
egrid	2.498	7.705	6.527

Table 4.3.21: Sot, GBR, error using the grid, egrid (%)

The parameter set (*seed*) taken is the one taken previously:

```
{random_state : 0, n_estimators : 200, max_depth : 10 for RFR and 4 for GBR,
 max_features : 0.3, min_samples_split : 8, min_samples_leaf : 4}
```

Results for any other parameter set from the grid can be consulted in the repository added in the Appendix.

We must remember, as we see in the theory chapter, that if we prune the original dataset to hold only a $\gamma\%$ of the original features, we can expect approximately a similar $\gamma\%$ of the original cost in time.

RFE

Once we have pruned the dataset with RFE, we can start a parameter search in the new pruned dataset. For a standard grid and $\lambda = 0.5$, we obtain the results described in tables (4.3.22) and (4.3.23). For $\lambda = 0.8$, we obtain the results described in tables (4.3.24) and (4.3.25). The tables include for reference the error rates obtained searching the whole dataset with the standard grid (*grid* row).

α	Num var	Train	Test	Val
1	16	4.045	7.875	6.904
0.5	33	3.694	7.779	6.694
0.25	67	3.335	7.688	6.556
0	135	3.451	7.677	6.531
grid	1080	3.243	7.693	6.585

Table 4.3.22: Sot, RFE, RFR, $\lambda = 0.5$, search in grid with pruned dataset (%)

α	Num var	Train	Test	Val
1	16	3.746	8.218	6.925
0.5	33	3.217	8.002	6.669
0.25	135	2.768	7.709	6.578
0	135	2.768	7.709	6.578
grid	1080	2.425	7.728	6.573

Table 4.3.23: Sot, RFE, GBR, $\lambda = 0.5$, search in grid with pruned dataset (%)

α	Num var	Train	Test	Val
1	19	3.964	7.794	6.795
0.5	47	3.713	7.725	6.623
0.25	74	3.374	7.713	6.554
0	181	3.402	7.664	6.503
grid	1080	3.243	7.693	6.585

Table 4.3.24: Sot, RFE, RFR, $\lambda = 0.8$, search in grid with pruned dataset (%)

α	Num var	Train	Test	Val
1	15	3.745	8.086	6.910
0.5	37	3.239	7.863	6.679
0.25	92	2.825	7.833	6.547
0	92	2.825	7.833	6.547
grid	1080	2.425	7.728	6.573

Table 4.3.25: Sot, RFE, GBR, $\lambda = 0.8$, search in grid with pruned dataset (%)

Subset RFE

Once we have pruned with SRFE, for $\lambda = 0.5$ and $c = 54$ features per subset we obtain the results of tables (4.3.26) and (4.3.27). For $\lambda = 0.8$ and $c = 54$ features per subset we obtain the results of tables (4.3.28) and (4.3.29) The columns have the same meaning as the one defined previously.

α	Num var	Train	Test	Val
2	135	3.546	7.679	6.522
1	135	3.546	7.679	6.522
0.5	270	3.463	7.672	6.502
0	270	3.463	7.672	6.502
grid	1080	3.243	7.693	6.585

Table 4.3.26: Sot, SRFE, RFR, $\lambda = 0.5$, $c = 54$, search in grid with pruned dataset (%)

α	Num var	Train	Test	Val
2	270	2.605	7.734	6.562
1	270	2.605	7.734	6.562
0.5	1080	2.425	7.728	6.573
0	1080	2.425	7.728	6.573
grid	1080	2.425	7.728	6.573

Table 4.3.27: Sot, SRFE, GBR, $\lambda = 0.5$, $c = 54$, search in grid with pruned dataset (%)

α	Num var	Train	Test	Val
2	70	3.606	7.669	6.534
1	150	3.565	7.676	6.527
0.5	180	3.217	7.677	6.516
0	180	3.491	7.686	6.514
grid	1080	3.243	7.693	6.585

Table 4.3.28: Sot, SRFE, RFR, $\lambda = 0.8$, $c = 54$, search in grid with pruned dataset (%)

α	Num var	Train	Test	Val
2	70	2.677	7.651	6.612
1	110	2.683	7.667	6.587
0.5	720	2.499	7.695	6.526
0	720	2.499	7.695	6.526
grid	1080	2.425	7.728	6.573

Table 4.3.29: Sot, SRFE, GBR, $\lambda = 0.8$, $c = 54$, search in grid with pruned dataset (%)

Boruta

Tables (4.3.30) and (4.3.31) contains the error values after searching in the grid for RFR and GBR, respectively, after pruning the dataset with the corresponding *perc* and 100 iterations. We observe that for RFR with considering as *perc* 100% and for GBR considering 25% we obtain better test results that performing the search in the same grid using our original whole dataset.

Perc	Num var	Train	Test	Val
100	110	3.482	7.672	6.538
75	260	3.333	7.641	6.540
50	388	3.305	7.685	6.526
25	466	3.107	7.662	6.557
grid	1080	3.243	7.693	6.585

Table 4.3.30: Sot, Boruta, RFR, 100 iter, search in grid with pruned dataset (%)

Perc	Num var	Train	Test	Val
100	9	4.001	8.336	7.160
75	17	3.611	7.937	6.884
50	73	2.852	7.759	6.599
25	272	2.540	7.649	6.605
grid	1080	2.425	7.728	6.573

Table 4.3.31: Sot, Boruta, GBR, 100 iter, search in grid with pruned dataset (%)

Comparative

For **RFE**, as we observe in tables (4.3.22), (4.3.23), for both RFR and GBR and $\lambda = 0.5$ we are able to improve the results obtained with a complete search in the dataset for both $\alpha = 0$ and $\alpha = 0.25$. For $\lambda = 0.8$, we must be more selective with the value of α. As we see, despite being able to get better results for $\alpha = 0$ for RFR, we cannot obtain better results for GBR, as a consequence of choosing a smaller number of variables. Then, we could improve our selection criterion attending not only to the α values but also to the number of variables selected.

Results for any other parameter set, which can be consulted on the repository of the Appendix, show a similar behaviour. For RFR and $\alpha = 0$ we get better or similar results to the ones we obtain with the whole dataset, whereas for GBR the quality of the search depends on the parameter set taken initially.

For **SRFE**, as we observe in tables (4.3.26), (4.3.27), for RFR and $\lambda = 0.5$ we are able to improve the results obtained with a complete search in the dataset any α. For GBR, we cannot improve the result. Besides, we need to consider bigger values α than in RFE to obtain reasonable pruning values [11]. For $\lambda = 0.8$, for both RFR and GBR we improve the results with any α.

For **Boruta**, for RFR we improve the results of the search for any percentage of the shadow value. For GBR, as expected the results is a bit less optimistic but we can improve the results taking as shadow value the 25%.

As we observe, in general terms for any of the previous techniques we can improve easily the results obtained for RFR without too much effort, whereas for GBR we need a careful selection of parameters. If we analyze the results for another set of parameters (available at the resources repository attached in the Appendix) for RFE, SRFE and Boruta, we obtain that these results hold. Tables (4.3.32) and (4.3.33) include the number of parameter sets in which we obtain better results pruning the dataset before executing a search of parameters for RFE and $\lambda = 0.8$ (from a total of 32 parameter sets in the grid) compared

[11] In contrast to RFE which considers pruned sets of size 2^k for $k = 1, 2, \ldots$, for SRFE we consider only subsets of approximate size 2^k for $k = v, v + 1, \ldots$ where v is the size of the minimum dataset considered ($v = \lceil log_2(54) \rceil$ in our examples).

to a search in the original (non pruned) dataset. In a similar way, tables (4.3.34) and (4.3.35) include the same numbers for SRFE and $\lambda = 0.8$. The disparity between RFE and SRFE is due to the small number of variables considered with RFE and $\alpha \in \{0.75, 1\}$, which means that it is quite important to consider an criterion for defining α which takes into account the number of selected variables. Defining a value $\lambda \ll 1$ will generate many intermediate pruning steps and *Perc stc*, which can give us a good idea about the adequate α, but at the high cost of many steps in RFE and SRFE which can ruin the advantage we gain from pruning the dataset for the search. Neither the authors from [20] nor [22] include an automatic criterion for fixing λ and α, so in this present thesis the values have been selected *ad hoc* using the recommendations from [20] [12]. Tables (4.3.36) and (4.3.37) include similar results for Boruta (20 iterations).

α	Train	Test	Val
1	0	0	0
0.5	0	0	0
0.25	0	8	16
0	11	31	32

Table 4.3.32: Sot, RFE, RFR, $\lambda = 0.8$, number of parameter set which improve the grid

α	Train	Test	Val
1	0	1	0
0.5	0	6	1
0.25	0	13	16
0	1	27	24

Table 4.3.33: Sot, RFE, GBR, $\lambda = 0.8$, number of parameter set which improve the grid

α	Train	Test	Val
2	0	25	32
1	0	29	32
0.5	6	32	32
0	10	32	32

Table 4.3.34: Sot, SRFE, RFR, $\lambda = 0.8$, number of parameter set which improve the grid

α	Train	Test	Val
2	0	27	10
1	0	30	19
0.5	0	31	24
0	0	30	28

Table 4.3.35: Sot, SRFE, GBR, $\lambda = 0.8$, number of parameter set which improve the grid

α	Train	Test	Val
100	0	23	32
75	11	32	32
50	16	30	32
25	12	31	32

Table 4.3.36: Sot, Boruta, RFR, 20 iter, number of parameter set which improve the grid

α	Train	Test	Val
100	0	0	0
75	0	15	6
50	0	22	12
25	0	27	24

Table 4.3.37: Sot, Boruta, GBR, 20 iter, number of parameter set which improve the grid

For completeness, we include tables containing the average and standard deviation of the 32 sets of parameters considered. For RFE and $\lambda = 0.8$ we have tables (4.3.38) and

[12] With the seed considered in this section, we can observe that RFE with $\alpha = 1$, $\lambda = 0.8$ takes a dataset with only 19 variables, which is clearly not enough. In contrast, SRFE with $\alpha = 2$, $\lambda = 0.8$ takes a dataset with 93 variables, generating much better results. Hence, we should take into account the size of the pruned dataset when defining α.

(4.3.39). Tables (4.3.40) and (4.3.41) include similar results for SRFE. Tables (4.3.42) and (4.3.43) refer to Boruta.

α	Train	Test	Val
1	3.991 ± 0.030	7.831 ± 0.014	6.850 ± 0.020
0.5	3.628 ± 0.146	7.771 ± 0.035	6.687 ± 0.046
0.25	3.542 ± 0.169	7.710 ± 0.028	6.586 ± 0.031
0	3.322 ± 0.130	7.661 ± 0.017	6.512 ± 0.012
grid	3.243	7.693	6.585

Table 4.3.38: Sot, RFE, RFR, $\lambda = 0.8$, mean and std for the 32 sets of parameters

α	Train	Test	Val
1	3.571 ± 0.175	7.949 ± 0.098	6.804 ± 0.065
0.5	3.279 ± 0.374	7.797 ± 0.083	6.658 ± 0.045
0.25	2.980 ± 0.193	7.740 ± 0.078	6.576 ± 0.041
0	2.761 ± 0.450	7.658 ± 0.073	6.554 ± 0.037
grid	2.425	7.728	6.573

Table 4.3.39: Sot, RFE, GBR, $\lambda = 0.8$, mean and std for the 32 sets of parameters

α	Train	Test	Val
2	3.509 ± 0.112	7.681 ± 0.020	6.548 ± 0.016
1	3.440 ± 0.116	7.666 ± 0.018	6.526 ± 0.018
0.5	3.345 ± 0.106	7.659 ± 0.015	6.518 ± 0.013
0	3.340 ± 0.117	7.661 ± 0.017	6.516 ± 0.011
grid	3.243	7.693	6.585

Table 4.3.40: Sot, SRFE, RFR, $\lambda = 0.8$, mean and std for the 32 sets of parameters

α	Train	Test	Val
2	2.84 ± 0.254	7.675 ± 0.065	6.583 ± 0.041
1	2.83 ± 0.359	7.641 ± 0.054	6.557 ± 0.043
0.5	2.797 ± 0.417	7.645 ± 0.049	6.545 ± 0.040
0	2.803 ± 0.575	7.636 ± 0.052	6.536 ± 0.029
grid	2.425	7.728	6.573

Table 4.3.41: Sot, SRFE, GBR, $\lambda = 0.8$, mean and std for the 32 sets of parameters

Perc	Train	Test	Val
100	3.471 ± 0.110	7.643 ± 0.199	6.664 ± 0.724
75	3.301 ± 0.162	7.644 ± 0.093	6.638 ± 0.693
50	3.236 ± 0.148	7.653 ± 0.100	6.635 ± 0.641
25	3.23 ± 0.160	7.655 ± 0.112	6.647 ± 0.664
grid	3.243	7.693	6.585

Table 4.3.42: Sot, Boruta, RFR, 20 iter, mean and std for the 32 sets of parameters

Perc	Train	Test	Val
100	5.765 ± 0.547	8.582 ± 0.053	7.731 ± 0.066
75	3.628 ± 0.709	7.861 ± 0.225	6.784 ± 0.221
50	2.923 ± 0.229	7.689 ± 0.071	6.584 ± 0.037
25	2.897 ± 0.394	7.662 ± 0.050	6.551 ± 0.055
grid	2.425	7.728	6.573

Table 4.3.43: Sot, Boruta, GBR, 20 iter, mean and std for the 32 sets of parameters

4.4 Conclusions and further work

In this chapter we have seen how pruning the initial dataset not only reduced considerably the resources needed for the construction of a good predictor but also helped us to obtain a better regressor, improving slightly the accuracy.

The methodologies we have used to prune each dataset, which are easy to understand, interpret and implement, allow us to reduce the problem from an initial high space to another easier to manage with an automatic criterion that the user can tune depending on the quality of the desired final predictor. They also allow us to simplify a search of parameters in a grid which outperforms the non pruned dataset easily. Despite as a general result we have been able to improve the accuracy of our predictor searching in a pruned dataset with any of the previous strategies, they are strongly different not only in basis assumptions but also in the type of results and time/memory cost required. Hence, we should guide our selection not only by the performance obtained but also bearing in mind the size of final features expected and the hypothesis we assume about the data.

As we have seen, both RFE and SRFE provide simple and trustworthy frameworks to simplify our problem assuring that the final dataset will be below a fixed number of features. If our main preoccupation is not the final size but to obtain all the relevant features, Boruta provides useful strategies to compute the *all relevant* solution with certain security that we are not removing useful information. However, as a consequence of the high cost required for Boruta and the lack of intuitive configuration parameters, simpler strategies like RFE and SRFE are preferred. The selection between RFE and SRFE would depend mainly on details from our dataset. For example, we can expect SRFE to be slower than RFE, as it requires extra preparation steps, but at the same time, it is capable of working with bigger dataset than RFE splitting them into pieces.

Despite the general approach searched, many points necessary for a complete treatment of the present problem have not been treated. Among these, some have been analyzed partially on the Appendix, such as the dependence on the grid and the stability of the obtained rankings. Also detailed results for other seeds in the grid are added in the resources repository of the Appendix. Others, such as the results for different feature relevance metrics (we have used the one given by *Scikit*), the dependence of the results on random state have not been treated and are open until the moment. As we have seen, whereas for RFR the results are reasonably good, for GBR they depend on a deeper degree on the parameters and the set of parameters taken. Hence the design of a natural criterion to select an adequate set of parameters (*seed*) and configuration parameters for each method is probably the natural continuation of the present work.

Bibliography

[1] Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. “The elements of statistical learning” Vol. 1. Springer, Berlin: Springer series in statistics, 2001 : Chapters 1, 2, 3, 9, 10, 15.

[2] Breiman, Leo. “Random forests.” Machine learning 45, no. 1 (2001): 5-32.

[3] Quinlan, J. Ross, and Ronald L. Rivest. “Inferring decision trees using the minimum description lenght principle.” Information and computation 80, no. 3 (1989): 227-248.

[4] Strobl, Carolin, James Malley, and Gerhard Tutz. “An introduction to recursive partitioning: rationale, application, and characteristics of classification and regression trees, bagging, and random forests.” Psychological methods 14, no. 4 (2009): 323-348

[5] Denil, Misha, David Matheson, and Nando De Freitas. “Narrowing the Gap: Random Forests In Theory and In Practice.” ICML (2014): 665-673

[6] Hernández-Lobato, Daniel, Gonzalo Martínez-Muñoz, and Alberto Suárez. “Empirical analysis and evaluation of approximate techniques for pruning regression bagging ensembles.” Neurocomputing 74, no. 12 (2011): 2250-2264.

[7] Martínez-Muñoz, Gonzalo, and Alberto Suárez. “Out-of-bag estimation of the optimal sample size in bagging.” Pattern Recognition 43, no. 1 (2010): 143-152.

[8] Wager, Stefan, Trevor Hastie, and Bradley Efron. “Confidence intervals for random forests: the jackknife and the infinitesimal jackknife.” Journal of Machine Learning Research 15, no. 1 (2014): 1625-1651.

[9] Elith, Jane, John R. Leathwick, and Trevor Hastie. “A working guide to boosted regression trees.” Journal of Animal Ecology 77, no. 4 (2008): 802-813.

[10] Friedman, Jerome H. “Greedy function approximation: a gradient boosting machine.” Annals of statistics (2001): 1189-1232.

[11] Friedman, Jerome H. “Stochastic gradient boosting.” Computational Statistics & Data Analysis 38, no. 4 (2002): 367-378.

[12] Guyon, Isabelle, and André Elisseeff. “An introduction to variable and feature selection.” Journal of machine learning research 3, no. Mar (2003): 1157-1182.

[13] Saeys, Yvan, Iñaki Inza, and Pedro Larrañaga. “A review of feature selection techniques in bioinformatics.” Bioinformatics 23, no. 19 (2007): 2507-2517.

[14] Vergara, Jorge R., and Pablo A. Estévez. “A review of feature selection methods based on mutual information.” Neural Computing and Applications 24, no. 1 (2014): 175-186.

[15] Louppe, Gilles, *Understanding Random Forests*, PhD dissertation, 2014, University of Liege

[16] Louppe, Gilles, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. "Understanding variable importances in forests of randomized trees." In Advances in neural information processing systems (2013): 431-439.

[17] Paul, Jérôme, Michel Verleysen, and Pierre Dupont. "Identification of statistically significant features from random forests.", ECML Workshop on Solving Complex Machine Learning Problems with Ensemble Methods. Prague, Czech Republic, Springer. 2013.

[18] Strobl, Carolin, and Achim Zeileis. "Danger: high power!–exploring the statistical properties of a test for random forest variable importance." (2008).

[19] Ishwaran, Hemant. "Variable importance in binary regression trees and forests." Electronic Journal of Statistics 1 (2007): 519-537.

[20] Díaz-Uriarte, Ramón, and Sara Alvarez De Andres. "Gene selection and classification of microarray data using random forest." BMC bioinformatics 7, no. 1 (2006): 1.

[21] Tuv, Eugene, Alexander Borisov, George Runger, and Kari Torkkola. "Feature selection with ensembles, artificial variables, and redundancy elimination." Journal of Machine Learning Research 10, no. Jul (2009): 1341-1366.

[22] Pan, Feng, Tim Converse, David Ahn, Franco Salvetti, and Gianluca Donato. "Feature selection for ranking using boosted trees." In Proceedings of the 18th ACM conference on Information and knowledge management, ACM (2009): 2025-2028

[23] Kursa, Miron B., and Witold R. Rudnicki. "Feature Selection with the Boruta Package" Journal of Statistical Software, no. 36, Sep (2010).

[24] Alonso, Álvaro, Alberto Torres, and José R. Dorronsoro. "Random Forests and Gradient Boosting for Wind Energy Prediction." International Conference on Hybrid Artificial Intelligence Systems, Springer International Publishing (2015): 26-37

[25] Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel et al. "Scikit-learn: Machine learning in Python." Journal of Machine Learning Research 12, no. Oct (2011): 2825-2830.

[26] Fanaee-T, Hadi, and Gama, Joao, Event labeling combining ensemble detectors and background knowledge, Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg.

[27] Daniel Homola *Boruta implementation for Python*, `https://bitbucket.org/danielhomola/boruta_py/`, 2015.

[28] Lu, Yijuan, Ira Cohen, Xiang Sean Zhou, and Qi Tian. "Feature selection using principal feature analysis." In Proceedings of the 15th ACM international conference on Multimedia, ACM (2007): 301-304

Appendix A

Code

A.1 Code for experiments

The code used for the experiments and additonal resources can be found in

https://bitbucket.org/alvarorrol/tfm_code/
https://bitbucket.org/alvarorrol/tfm_resources/

A.2 *Scikit*

As we mentioned in the introduction, there are several software packages which are useful for dealing with Trees. We have used *Scikit* package (version 0.17) because

- It has a very simple and powerful interface at the same time.
- It has very good documentation.
- It has a great collection of developers and users, improving continuously the package.
- It has all the simplicity of *Python*.

The code has been ported partially to `Cython` and has experienced a great improvement in speed and performance during the latest releases of the package, as we can see in (A.2.1).

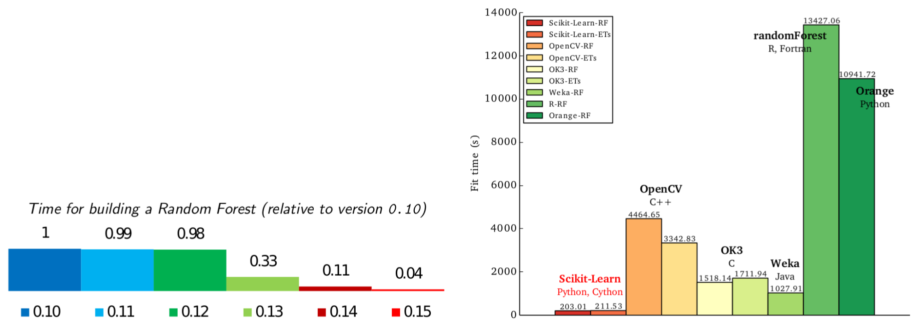


Figure A.2.1: *Accelerating Random Forests in* Scikit slides, from G Louppe

Besides, for Random Forest (RF) it incorporates native parallelization using *joblib*. This means *Scikit* is the better package to work with RF, as (A.2.1) shows.

For Gradient Boosting (GB), Decission Trees (DT) and Linear Methods it is possible to use the usual *multiprocessing* library from the standard library for computing a big grid.

A.2.1 Trees common interface

Trees are the main ingredient to build ensemble methods like Random Forest Regressors and Gradient Boosting Regressors. A brief description of the interface has been described in the chapters above, but to have a complete view of the parameters which have driven the Tree building process, we attach the main details taken directly from the *Scikit* documentation:

- *criterion:* The function to measure the quality of a split. The only supported criterion is *mse* (for the mean squared error).
- *splitter:* The strategy used to choose the split at each node. Supported strategies are best to choose the best split and random to choose the best random split.
- *max_features:* The number of features to consider when looking for the best split
- *max_depth:* The maximum depth of the Tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- *min_samples_split:* The minimum number of samples required to split an internal node.
- *min_samples_leaf :* The minimum number of samples required to be at a leaf node.
- *min_weight_fraction_leaf :* The minimum weighted fraction of the input samples required to be at a leaf node.
- *max_leaf_nodes:* The maximum number of leafs required to be at a leaf node.

All the detailed documentation for Trees can be found at Scikit *Decision Tree Regressor documentation*.

A.2.2 DTR, RFR and GBR interfaces

The package *Scikit* has an implementation of CART Tree in the class *DecisionTreeRegressor*, Random Forest Regression in the class *RandomForestRegressor* and Gradient Boosting Regression in the class *GradientBoostingRegressor*.

The parameters to tune the Tree building process and their default values are the following:

```
DTR: {criterion='mse', splitter='best', max_depth=None, min_samples_split=2,
min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None,
random_state=None, max_leaf_nodes=None}

RFR: {n_estimators=10, criterion='mse', max_depth=None, min_samples_split=2,
min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto',
max_leaf_nodes=None, bootstrap=True, oob_score=False, n_jobs=1,
random_state=None, verbose=0, warm_start=False}

GBR: {loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
max_depth=3, init=None, random_state=None, max_features=None, alpha=0.9,
verbose=0, max_leaf_nodes=None, warm_start=False}
```

The returned predictors have several attributes; among them we have the list *feature_importances_*, which contains the estimation of feature relevance.

Appendix B

Additional results

B.1 Quality of Tree feature relevance as a tool to prune

With the intention of measuring the quality of our dataset simplification, we should compare the accuracy of several predictors after a reduction of the number of features with different strategies. Tables (B.1.1) and (B.1.2) contain the test error for the Sotavento problem and for different feature reduction and dimensionality reduction techniques introduced in the theory chapters.

As we observe, simple strategies like Variance pruning (filter method which removes variables with small variance) are not quite effective holding informative features. More complex methodologies like PCA improves the error obtained, but are still above our pruning using the feature relevance obtained from RFR and GBR. *F est*, which refers to the feature estimation using the F ratio of error, gives us much better results, although it is slightly above our technique in most of the situations.

Perc	RFR	PCA	Variance	F est
1	7.943	9.043	15.878	8.202
2	7.860	8.992	15.592	7.816
5	7.714	9.180	13.941	7.802
10	7.678	9.318	16.017	7.861
20	7.653	9.511	8.838	7.826

Table B.1.1: Sot, RFR, Test error pruning the dataset with optimal parameter.

Perc	GBR	PCA	Variance	F est
1	8.249	8.603	15.047	8.353
2	8.055	8.477	15.068	7.943
5	7.781	8.277	13.210	7.748
10	7.738	8.496	15.145	7.861
20	7.686	8.584	8.691	7.798

Table B.1.2: Sot, GBR, Test error pruning the dataset with optimal parameter.

B.2 Extended grid

The performance of the estimation will depend completely on the set of parameters fixed. Hence, we have to face the dilemma of choosing a big grid which represents the parameter space with certain detail or a small one with is *cheap* to compute. Most of the times we will estimate, using the nature of the problem and the dataset associated, a small subset of parameters from the whole space that we think can contain a good approximation to the optimal one. However this will mean some time and initial attempts which will consume our precious time if we do not have an idea of the cost of each operation. As the BS and Sotavento problems are easy problems, we can ask ourselves about the cost of computing the grid and obtain certain general results.

B.2.1 BS problem

The following graphs show us the comparison between our grid and the extended grid. Figures (B.2.1) contain the time and error distributions using the standard grid (in red) and the extended grid (in blue) [1].

Each row represents the DTR, RFR and GBR regressors respectively (top-down), while each column represents the time, train error, test error and validation error respectively (from left to right).

With respect to the images, we can observe that our decisions for the standard grid of DTR can be considered a reasonable description. On the other hand and as we partially expected, for RFR and GBR the standard grid does not contain a good representative section of the parameter space, because our error distribution is biased towards high values in the extended grid. However, at least, the grid is distributed in the low time values of the extended grid.

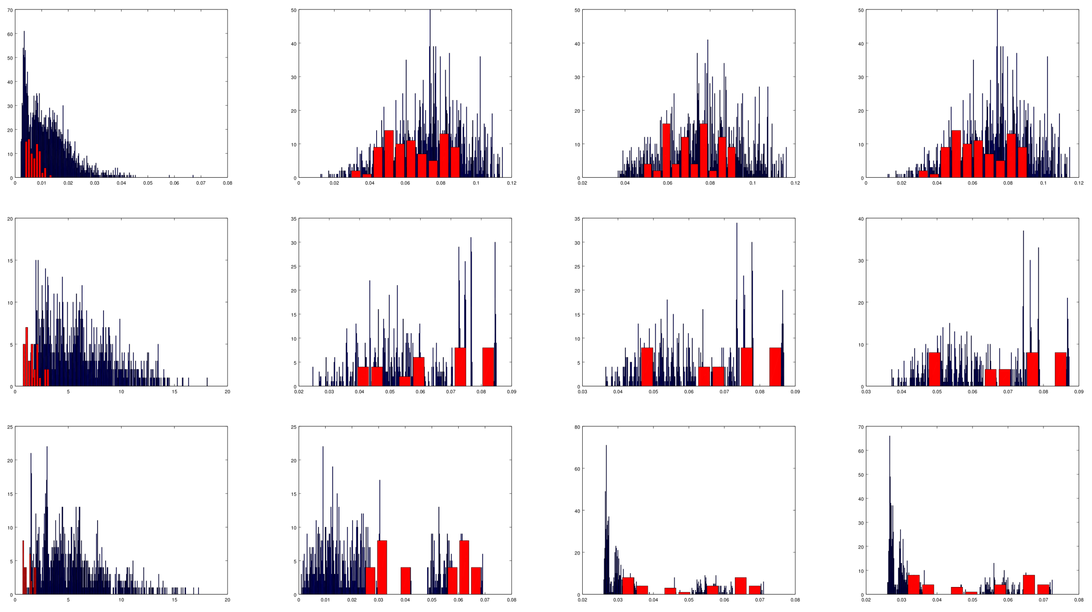


Figure B.2.1: BS: time, train, test, val error in grid-egrid (left-right). For DTR, RFR and GBR problem (top-down).

Figures (B.2.2) include the distribution of relevance for both grids. With respect to the distribution of feature relevance, whereas DTR does not hold a uniform behaviour, RFR and GBR show us that our standard grid was a good option. In general terms, the feature relevance distribution given by DTR is not stable while the one given by RFR or GBR is (more or less) stable. We can consider then than our selection of parameters for RFR and GBR is a good option to obtain a general description of the feature relevance distribution, even being *cheap* to compute as we have seen in the time distribution.

[1]As the extended grid contains more elements, the figures have been graphed with 10 bins for the standard grid and a number of bins for the extended grid such that the number of items per bin, in average terms, is equal for both representation (so for example, if we have 81 elements in the standard grid and 10 bins, we have an average of 8 items per bin, so if the extended grid has 2592 items we should have approximately 320 bins for the extended grid).

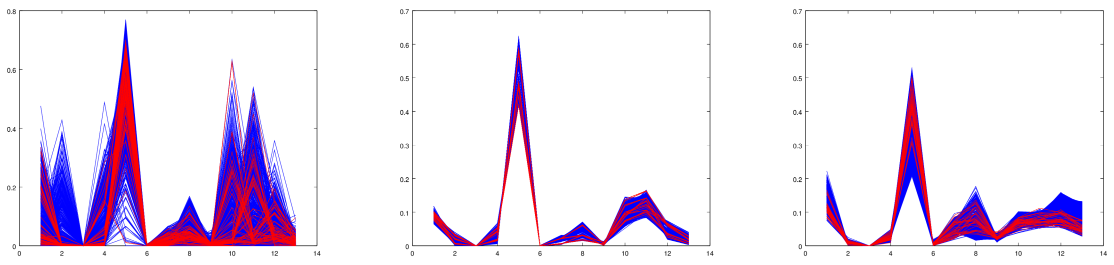


Figure B.2.2: BS: DTR, RFR and GBR, distribution of feature relevance for both grids.

Table (B.2.1) contains the times needed to compute both grids for each method. We observe that the time cost goes up easily just duplicating the number of options for each parameter, especially for ensemble methods. Hence, we must be extremely careful when building a grid for parameter search. In our example, we have duplicated the number of options for each feature of the grid and we have converted a search of < 2 minutes in a problem of > 1 hour, perhaps without any real gain because many set of parameters can de discarded with an analysis of the problem.

Problem	Grid	Extended Grid
DTR	1.224	32.129
RFR	111.066	6247.688
GBR	82.232	5583.407

Table B.2.1: BS: time(s) for the grid and the extended grid.

B.2.2 Sotavento

The following graphs show us the comparison between our grid and an extended grid, which contains the previous one as a subset. Figures (B.2.3) contain the time and error distribution using the standard grid (red) and the extended grid (blue).

The structure is identical to BS. As we observe for the Tree predictors the first image of each row of images, which depicts a time distribution, shows us that our initial grid is distributed in the lower values of the extended grid, as we expected. However, the distribution of errors is not strongly shifted towards high value of the distribution of error, as we could expect initially from our simple grid. For example, the error patterns of the testing (third) and validation (fourth) images show us that the selection of a simple grid can be representative of a bigger and more complex extended grid.

We also observe that whereas the error from RFR is concentrated the one from DTR and GBR is more dispersed. The reason for a big separation in each group of errors in RFR is our choice of depth values.

As in the previous situation, the distribution of relevance for both the standard grid and the extended grid are also *similar* (one pending issue is then defining the proximity of the distributions).

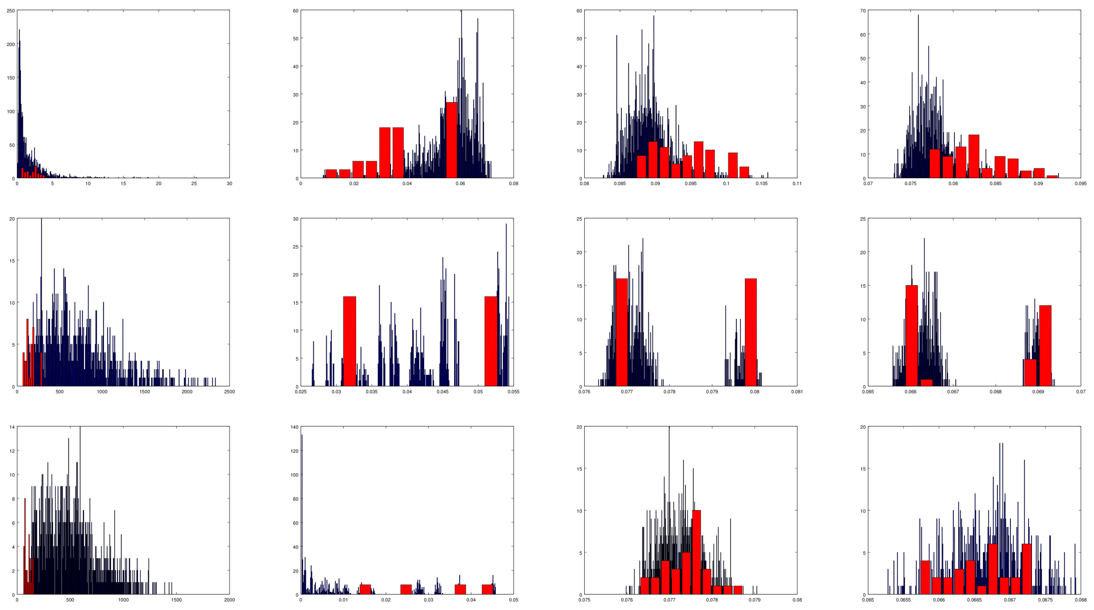


Figure B.2.3: Sot: time, train, test, val error in grid-egrid (left-right).
For DTR, RFR and GBR problem (top-down).

B.3 Randomness

As we have stated both in the theoretical and experimental chapters, DTR, RFR and GBR use certain randomness in the training process. Analyzing the repository for DTR we found that the `Cython` code uses a 32bit XorShift random generator. The code is:

```
# rand_r replacement using a 32bit XorShift generator
# See http://www.jstatsoft.org/v08/i14/paper for details
cdef inline UINT32_t our_rand_r(UINT32_t* seed) nogil:
    seed[0] ^= <UINT32_t>(seed[0] << 13)
    seed[0] ^= <UINT32_t>(seed[0] >> 17)
    seed[0] ^= <UINT32_t>(seed[0] << 5)

    return seed[0] % (<UINT32_t>RAND_R_MAX + 1)
```

The referenced paper gives us an idea about the implementation. This random generator is used through two wrapper functions, which generate random integers and doubles.

```
cdef inline SIZE_t rand_int(SIZE_t low, SIZE_t high,
                            UINT32_t* random_state) nogil:
    """Generate a random integer in [0; end)."""
    return low + our_rand_r(random_state) % (high - low)


cdef inline double rand_uniform(double low, double high,
                                UINT32_t* random_state) nogil:
    """Generate a random double in [low; high)."""
    return ((high - low) * <double> our_rand_r(random_state) /
            <double> RAND_R_MAX) + low
```

B.3.1 Stability of the solution

As we have stated, we use the validation set to rank the subsets of parameters from the grid we have built. However, as the process of building Trees involves some randomness, we have to assure some stability for the results. Although all the experiments in the present work use a fixed initial *random_state* for the random generators (*random_state* = 0) we are also interested on discovering the dependence on this *seed*.

For that, we can design different procedures. We have decided to obtain several ranking of parameters and apply some metrics to measure the *distance* between our *solution* lists.

We have used as metric the minimum number of swaps to convert one list into others [2]. This metric is useful to compute the disorder of the rankings and gives a clear idea about the stability of the ranking. However this metric does not take care of the order of the list (we are comparing rankings, no lists). Hence we can use diminishing weights when we compare elements, in such a way that only top elements are important. We are using as metrics the ones described in (B.3.1).

The element p is the permutation needed to transform the first list into the second one. We iterate through each element of each permutation adding the previous quantities and later we average. The images in (B.3.1) includes the distribution of both metrics for all the permutations of five elements (we are working with list of 32 elements for RFR, GBR and 81 elements for DTR, so computing the distribution for this values is not possible but can give us an idea about the range as for big j and $p[k]$ the value to add goes to 0 fast).

For the BS problem, we have repeated the training process 100 times using the standard grid, obtaining the results described in the table (B.3.2).

Statistic	$nswaps+=$
Metric1	$\frac{I(p[j]!=j)}{2^{p[j]+j}}$
Metric2	$\frac{abs(p[j]-j)}{2^{p[j]+j}}$

Table B.3.1: BS: Metric description.

Statistic	DTR	RFR	GBR
Swaps	63	6	4
Metric1	0.303	0.091	0.408
Metric2	0.448	0.120	0.598

Table B.3.2: BS: Stability of the rankings. 100 repetitions.

For n total list we have $n(n-1)/2$ comparisons between rankings. For the swap metric, in general terms, we have obtained a considerable variation for DTR [3] , but a small one for RFR and GBR. With respect to the ranking metrics, we have obtained a small variability for RFR and a big one for DTR and GBR. Bearing in mind the average number of swaps, we understand the result for DTR. With respect to GBR, a careful observation of the rankings shows us that although there are just a few swaps in the lists, they are elements in the top. Our selection of the depth modifies the top elements of the list depending on the seed used. Hence, we can conclude that in general sense both RFR and GBR are stable with respect to the initial random state.

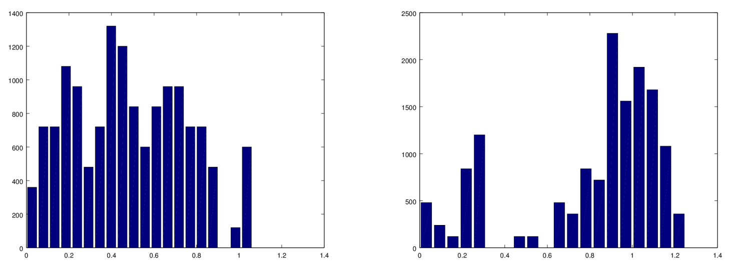


Figure B.3.1: Distribution of metric1 (left) and metric2 (right) for permutations of 5 items.

[2]Code can be consulted on the repository.

[3]Then it is difficult to talk about *optimal* set of parameters for a DTR regressor if we concentrate only on the parameters of the Tree.