Machine Learning Project

Federico Minutoli
fede97.minutoli@gmail.com
ID 4211286

Gianvito Losapio
gvlosapio@gmail.com
ID 4803867

Abstract

We present a pair-project in which we tackle the same problem proposed by the IREN company during C1A0 Hackathon. We tried to mix project of types 1 and 2 with the aim of considering a full real-case scenario concerning the electricity market. As for the type 2 part of the project, we provide a detailed data processing and a simple time-series analysis in the notebook file *Code/Data processing/Time-series analysis.ipynb*. As for the type 1 part of the project, a detailed formulation of two different algorithms follows: Random forests (by Gianvito Losapio) and Gradient boosting (by Federico Minutoli). Both of the algorithms have been implemented. The structure of the code is presented in the notebook file *Code/Problem.ipynb* as well as its use to solve the main problem. Results and comparisons of the methods are provided.

1 Introduction

We present a machine learning problem aimed at obtaining some predictions related to the Day-Ahead Market as proposed by the IREN company during C1A0 Hackathon.

The Day-Ahead Market, hereafter referred as **MGP** (Mercato Giorno Prima), is the market for the trading of electricity supply offers and demand bids for each hour of the next day in Italy. All electricity operators may participate in the MGP. In the MGP, hourly energy blocks are traded for the next day. Participants submit offers/bids where they specify the quantity and the minimum/maximum price they are willing to sell/purchase. Participants can submit a maximum of 4 offers/bids for each of their production plant and consumption unit.

GME (Gestore dei Mercati Energetici) acts as a central counterparty of the MGP market. GME reorganizes bids/offers according to the economic merit-order criterion and computes both the demand and supply curves. The marginal clearing price is the equilibrium price obtained from the intersection of the two aforementioned curves.

The accepted offers are those with a submitted price not larger than the marginal clearing price, but all of them are valued at the marginal clearing price. The accepted bids are those with a submitted price not lower than the PUN (Prezzo Unico Nazionale – national single price, agreed daily).

In particular, the **goal** of our problem is to predict the number of accepted offers for the week 22-28th February 2019 for a specific list of participants, identified by IREN itself, given some hourly data referred to the period between 01/01/2018 and 21/02/2019. In more specific terms, we want to forecast a single week of 28 different discrete time series, given their values for the previous 14 months as well as other general information collected along the same period.

Data is heterogeneous in formats and spread across multiple tables, like:

- Forecasted temperature in northern Italy as a whole;
- Amount of produced renewable energy;
- Available energy for each production unit of interest;

- Energy production costs for thermal plants;
- Accepted amount of energy and prices in the market;
- Full list of offers in the market

In the notebook file *Code/Data processing/Time-series analysis.ipynb* we provide:

- a detailed documentation of all the tables used for the solution;
- a preprocessing of data (data cleaning, dates format conversion);
- a feature engineering phase (involving a simple time-series analysis);
- the creation of final csv files containing training and test sets.

In the following sections we present two different algorithms used to solve the aforementioned problem:

- **Random forests** are presented in the section 3 (by Gianvito Losapio)
- **Gradient boosting** is presented in the section **??** (by Federico Minutoli)

Both of the algorithms have been implemented. The structure of the code is presented in the notebook file *Code/Problem.ipynb* as well as its use to solve the main problem. Whenever possible, some references to the implementation are made during the following theoretical presentation.

The basic building block of both of the algorithms are classification and regression tree models, briefly discussed in the following section 2.

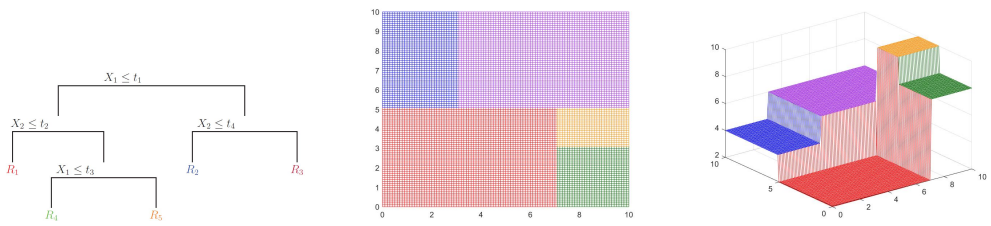
2 Classification And Regression Trees (CART) models

The key idea of classification and regression trees is to find an appropriate partitioning of the input space such that in each resulting region a local model is fine.

In order to avoid an excessive data fragmentation (causing overfitting), a recursive binary partition is considered. The input space is splitted in two regions, then one or both of these regions are split into two more regions, and this process is continued, until some stopping rule is applied. This is easily achieved by applying a threshold to one single feature at a time.

The overall partitioning procedure is associated to a binary decision tree. The leaves represent the resulting regions of the input space.

The figure 1 shows an intuitive example of a regression problem.



(a) (b) (c)

Figure 1: Example of a regression tree. The inputs consist of two features X_1, X_2 in the interval $[0; 10]$. (a) 4 thresholds are applied following the tree structure. (b) The axis parallel splitting leads to the creation of 5 partitions of the 2D space. (c) The regression model predicts a constant for each region; piecewise constant surfaces are the results in the 3D space.

A detailed formulation of the CART models is now developed.

Consider a regression scenario[1]

$$(x_i, y_i)_{i=1}^n \quad \text{with} \quad x_i \in \mathbb{R}^d \,,\, y_i \in \mathbb{R}$$

and suppose to have a binary decision tree T which partitions the input space $\mathbb{R}^d$ into disjoint regions $\mathcal{D}_m$, $m = 1, \ldots, M$. The CART model assigns a constant γ_m to each such region and the predictive rule is

$$x \in \mathcal{D}_m \implies f(x) = \gamma_m \,.$$

The model can be formulated as

$$f(x) = \sum_{m=1}^{M} \gamma_m \mathbb{1}_{\{x \in \mathcal{D}_m\}} \tag{1}$$

where $\mathbb{1}_{\{\cdot\}}$ denotes the indicator function.

In conclusion, we observe that a CART model can be interpreted as an *adaptive basis-function model* (ABM), which is a model of the form

$$f(x) = w_0 + \sum_{m=1}^{M} w_m \phi_m(x) \tag{2}$$

where $\phi_m(x)$ are called basis function and refer to feature maps automatically learned from data.

Rewriting (1) to look like (2) we get

$$f(x) = \sum_{m=1}^{M} \gamma_m \phi(x; v_m) \tag{3}$$

where the basis functions $\phi(x; v_m)$ define the regions and are parametrized in v_m which, in turn, encodes the choice of features to split on, and the threshold values, on the path from the root to the m-th leaf. The weights γ_m specify the output prediction in each region.

CART models are popular for several reasons: they could be easily interpreted, they can easily handle mixed discrete and continuous inputs, they perform automatic variable selection, they are relatively robust to outliers, they scale well to large data sets, and they can be modified to handle missing inputs.

One of the main disadvantages is that decision trees are unstable: small changes to the input data can have large effects on the structure of the tree, due to the hierarchical nature of the tree-growing process, causing errors at the top to affect the rest of the tree. In frequentist terminology, we say that trees are *high variance estimators*. [1]

2.1 Parameters and model complexity

Designing an algorithmic procedure to create CART models pose the following questions:

1. how to select the feature to split on (*splitting variable*) ?
2. how to choose the threshold to apply for a given feature (*splitting point*) ?
3. which constant γ_m to adopt in each region?
4. how deep should be the tree?

Questions n.1 and n.2 concern the growing of decision trees and clearly represent the central issue to set up algorithms. Random forests and Gradient Boosting use different instances of the Empirical Risk Minimization (ERM) strategy resorting to similar greedy solutions (described in later sections). An answer to question n.3 is also provided by adopting these approximate procedures.

Question n.4 concerns another key aspect strictly related to the previous one: when to stop the splitting procedure. The tree size (directly connected to the number of regions M used to split the input set) is a tuning parameter governing the model's complexity, and the optimal value should be adaptively chosen from the data. A very large tree corresponds to the identification of a large

[1] Regression trees will be the main object of discussion for the sake of brevity since they are used in the practical part of the project

number of partitions in the input set and might overfit the data, while a small tree might not capture the important structure [2].

Node *purity* represents a convenient heuristic as a stop criterion (a node is said to be *pure* if the value of y is the same for all examples at this node) but further parameters are usually introduced with the purpose of keeping under control the growing procedure[2]:

- the maximum depth of the tree;
- the minimum number of samples required to split a node;
- the minimum number of samples required to be at a leaf node. As a result, a split point at any depth will only be considered as a valid threshold if it produces a fixed number of remaining examples in each of the left and right branches.

In conclusion, an estimate of the computational complexity is reported. Given n samples and d features, the time cost at each node consists of searching through $O(nd)$ to find the best split (in the worst case, which is the initial split where the number of possible thresholds equals the number of samples). The total cost over the entire tree is obtained multiplying nd by the depth of the tree. In the best case of a balanced tree the depth would be in $O(\log n)$, but the decision tree does locally optimal splits without caring much about balancing. This means that the worst case of depth being in $O(n)$ is possible - basically when each split simply splits data in 1 and $r-1$ samples, where r is the number of samples of the current node. Hence, the final time complexity to construct a decision tree is in between $O(nd \log n)$ (best-case) and $O(n^2 d)$ (worst-case). Accordingly, the query time is in between $O(\log n)$ and $O(n)$. Unless complex operations are implemented in the growing procedure, space complexity is not a peculiar issue of the CART models.

3 Random Forests

Random forests were originally proposed by (Breiman, [4]) and represent an attempt to overcome the limitations of classification and regression tree models by injecting different types of randomness. Such models often have very good predictive accuracy with little tuning and have been widely used in many applications [1], [2].

The implementation code is provided in the folder *Code/Random Forest/src*.

A detailed formulation of Random forests according to the Empirical Risk Minimization (ERM) is now presented. In order to simplify the notation, we can rewrite the CART model developed so far (eq. 1) as:

$$f(x) = T(x; \Theta) \tag{4}$$

where T evokes the word *Tree* and Θ denotes the entire parameter set. Θ can be written as

$$\Theta = \{\mathcal{D}_m, \gamma_m\}_{m=1}^{M} \tag{5}$$

where $\mathcal{D}_m$ indicates the m-th region and γ_m the output associated to it. Analogously to the formulation (3), it is worth noting that $\mathcal{D}_m$ encodes the choice of features to split on and the threshold values, on the path from the root to the m-th leaf.

Following this notation, the ERM problem for learning the CART model can be formulated as

$$\Theta^* = \arg\min_{\Theta} \sum_{i=1}^{n} \ell\big(y_i, T(x; \Theta)\big) \tag{6}$$

being a proxy for the expected loss

$$\mathbb{E}_{(x,y)}\Big\{\ell\big(y, T(x;\Theta)\big)\Big\} = \int dxdy\, p(x,y)\ell\big(y, T(x;\Theta)\big)\,.$$

[2]Here we list the parameters used in our implementations, named respectively *max_depth, min_samples_split, min_samples_leaf* as in the *DecisionTreeRegressor* and *RandomForestRegressor* classes of the *scikit-learn* library [3].

An appropriate loss function ℓ is needed to quantify how good a split is for growing the tree. In case of regression trees, the square loss is considered[3]:

$$\Theta^* = \arg\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} \big(y_i - T(x;\Theta)\big)^2 \tag{7}$$

Finding the optimal binary recursive partition was shown to be a NP-complete problem [1]. Hence, a typical strategy is to use a greedy, top-down recursive partitioning algorithm to find a good split of the input space.

Starting with the entire dataset $\mathcal{D} = \{x_i, y_i\}_{i=1}^n$, a variable j and a split point t are to be defined for the first splitting of the input space $\mathbb{R}^d$ in a pair of half-hyperplanes $\mathcal{D}_L, \mathcal{D}_R$ defined as follows (referring to the indicies of the dataset):

$$\mathcal{D}_L(j,t) = \{(x_i, y_i) \,|\, x_{ij} \leq t\} \quad \text{and} \quad \mathcal{D}_R(j,t) = \{(x_i, y_i) \,|\, x_{ij} > t\}\,.$$

In addition, the corresponding output constants γ_L, γ_R for each region have to be fixed.

The greedy approach consists of finding the locally optimal solution such that the minimum cost is achieved considering both the splits

$$\min_{j,t} \left[\min_{\gamma_L} \frac{1}{|\mathcal{D}_L(j,t)|} \sum_{y_i \in \mathcal{D}_L(j,t)} (y_i - \gamma_L)^2 + \min_{\gamma_R} \frac{1}{|\mathcal{D}_R(j,t)|} \sum_{y_i \in \mathcal{D}_R(j,t)} (y_i - \gamma_R)^2 \right] \tag{8}$$

and then repeating the procedure recursively to grow the tree until some stopping rule is applied.

The problem (8) can be easily simplified noting that for any choice (j,t) the inner minimizations are solved by using the mean response in each region, that is

$$\gamma_L \leftarrow \frac{1}{|\mathcal{D}_L|} \sum_{y_i \in \mathcal{D}_L} y_i = \bar{y}_{\mathcal{D}_L} \quad \text{and} \quad \gamma_R \leftarrow \frac{1}{|\mathcal{D}_R|} \sum_{y_i \in \mathcal{D}_R} y_i = \bar{y}_{\mathcal{D}_R}\,. \tag{9}$$

Substituting (9) into (8) we get:

$$\min_{j,t} \underbrace{\frac{1}{|\mathcal{D}_L(j,t)|} \sum_{i \in \mathcal{D}_L(j,t)} (y_i - \bar{y}_{\mathcal{D}_L})^2}_{\text{cost}(\mathcal{D}_L)} + \underbrace{\frac{1}{|\mathcal{D}_R(j,t)|} \sum_{i \in \mathcal{D}_R(j,t)} (y_i - \bar{y}_{\mathcal{D}_R})^2}_{\text{cost}(\mathcal{D}_R)} \tag{10}$$

where $\text{cost}(\mathcal{S})$ denotes the cost function of a generic node with $\mathcal{S}$ as the set of its elements[4]:

$$\text{cost}(\mathcal{S}) = \frac{1}{|\mathcal{S}|} \sum_{y_i \in \mathcal{S}} (y_i - \bar{y}_{\mathcal{S}})^2 \quad , \quad \bar{y}_{\mathcal{S}} = \frac{1}{|\mathcal{S}|} \sum_{y_i \in \mathcal{S}} y_i\,. \tag{11}$$

In order to increase the effectiveness of the greedy algorithm, a slightly different function is usually considered [1],[3], such as a normalized measure of the reduction in cost

$$\Delta(j,t) = \text{cost}(\mathcal{D}) - \left[\frac{|\mathcal{D}_L|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_R) \right] \tag{12}$$

where the dependence of (j,t) is again intended on $\mathcal{D}_L, \mathcal{D}_R$ and is omitted for a lighter notation. This metric can be interpreted as the measurement of a marginal improvement in the solution provided by a single split and, hence, has to be maximized.

Therefore, the final greedy solution to (7) is

$$\arg\max_{j,t} \; \Delta(j,t) \tag{13}$$

[3] In the classification setting, there are several ways to measure the quality of a split, such as misclassification rate, information gain and the Gini index.

[4] As defined in the *_compute_cost* function of the class *Node* in the file *decision_tree.py*

which is computed by recursively finding the pair (j^*, t^*) such that

$$\boxed{(j^*, t^*) = \arg \min_{j=1,...,d} \min_{t \in \mathcal{T}_j} \frac{|\mathcal{D}_L|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_R)} \tag{14}$$

where the two minimization problems are solved in order by computing the objective function for each splitting variable $j = 1, ..., d$ and for each splitting point $t \in \mathcal{T}_j$ [5].

It is worth noting that the set of possible thresholds $\mathcal{T}_j$ for a given feature j is obtained by sorting the unique values of the remaining examples at a certain node. For example, if feature 1 has the values $\{4.5, -12, 72, -12\}$, then we set $\mathcal{T}_1 = \{-12, 4.5, 72\}$.

The main idea of random forests is to reduce variance by injecting two types of randomness in the procedure presented so far:

- *bagging* or *bootstrap aggregating*: build multiple independent random decision trees, each one on a random sample drawn with replacement from the original dataset
- *feature bagging*: randomly select a subset of feature as splitting variable candidates during the growing procedure of each tree.

After B such trees are grown $\{T_b(x; \Theta_b)\}_{b=1}^B$, the random forest predictor f_{rf} is a simple mean of the outcomes of the single trees:

$$f_{\text{rf}}(x) = \frac{1}{B} \sum_{b=1}^{B} T_b(x; \Theta_b) \tag{15}$$

Averaging together many "noisy" estimates and decorrelating the multiple trees through randomness contribute to overcome the limitation of the CART models and obtain very good predictive accuracy.[4],[1],[2]

Clearly, two new main parameters are to be adjusted when using this method[6]:

- the number B of the trees in the forest
- the number of random features p to consider when looking for the best split. Typically values are $\sqrt{d}$ or even 1.

Analogously to the previous formulation, the size of the model is between $O(Bnd \log n)$ and $O(Bn^2 d)$ (considering all the features for the best split, assuming 2 as the minimum number of samples required to split a node and 1 as the minimum number of samples required to build a valid leaf).

Ideally the larger B the better, but also the longer it will take to compute and the higher is the risk to overfit. In addition, it is likely that results will stop getting significantly better beyond a critical number of trees. The lower p the greater the reduction of variance, but also the greater the increase in bias. Empirical good default values are $p = d$ (always considering all features instead of a random subset) for regression problems. The best parameter values should always be cross-validated [3].

4 Gradient Boosting (by Federico Minutoli)

Quick note: Gradient boosting will be referred here on as *Gradient Boosting* or GB interchangeably, for the sake of brevity.

Gradient Boosting is a machine learning technique for regression and classification problems, introduced with a paper by Jerome Friedman in 1999 [5], which produces a predictive model in the form of an ensemble of weak models.

Its idea originated in the observation that boosting can be interpreted as an optimization step on a suitable cost function. Being an accurate and quite effective off-the-shelf procedure it is used extensively in a variety of areas, including Web search ranking and ecology.

[5] As implemented in the *build* function of the class *Node* in the file *decision_tree.py*

[6] They are named respectively *n_estimators* and *max_features* in the *RandomForestRegressor* class of the file *rf.py*. The same class provides support for the other parameters mentioned in the previous section 2.1.

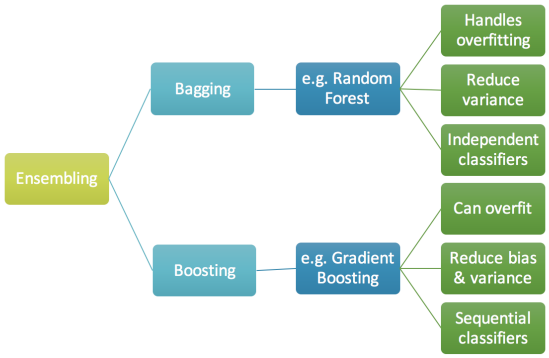


Figure 2: Ensemble learning

Since GB is a boosting algorithm itself, it is constructed in a greedy sequential forward stage-wise manner, as (2) illustrates, and thus it cannot be parallelized, unlike Random Forests.

But what is Boosting exactly? How does it work?
I am briefly gonna touch on that down below.

4.1 What is Boosting?

The idea behind Boosting is relatively simple. As we introduce more weak models, the composite model becomes a stronger and stronger predictor since hard to classify instances are given bigger weights which, in turn, will guide subsequent base learners to put more emphasis on them. To achieve this, the regression models are constructed by fitting the so called *"pseudo-residuals"* by least-squares at each iteration (known as *boosting round*). This pseudo-residuals represent the gradient of the loss function being minimized, w.r.t. the model weights for each data point x_i evaluated up to that step.

More on that below. Feel free to skip this next paragraph if you want to directly deep down into Gradient Tree Boosting, which is the scope of the project.

4.1.1 Math behind Boosting

With Eq. (2) we have already learnt what an *adaptive basis-function model* is. This is exactly the realm of Boosting since the first intuition is to approximate the expected value of some specified loss function $L(\mathbf{y}, f(\mathbf{x}))$ by an additive expansion in the form:

$$f(x) = \sum_{m=0}^{M} \beta_m h_m(\mathbf{x}) \tag{16}$$

with $h_m = h(\mathbf{x}; a_m)$ a generic base learner with a set of parameters a_m.

Each pair (β_m, a_m) is jointly fit to the training data in a forward stage-wise manner:

1. Start with an initial guess $F_0(\mathbf{x})$
2. for $m = 1$ to M:

$$(\beta_m, a_m) = \arg\min_{\beta, a} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta h(x_i; a) \tag{17}$$

Gradient boosting approximately solves (17) for arbitrary ***differentiable*** loss functions $L(\mathbf{y}, f(\mathbf{x}))$, by splitting that difficult optimization step in two simpler terms:

1. Solve a least-squares optimization problem by fitting $h(\mathbf{x}; a)$ to the pseudo-residuals, which are none other than the components of the gradient of the loss function L w.r.t the composite model generated at the precedent iteration $m-1$:

$$\tilde{y}_{i,m} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)} \tag{18}$$

where each $\tilde{y}_{i,m}$ explains what's left in y_i at iteration m.

This leads a_m to be computed as:

$$a_m = \arg\min_{a} \sum_{i=1}^{N} [\tilde{y}_{i,m} - h(x; a)]^2 \tag{19}$$

2. Single parameter optimization based on the global loss L:

$$\beta_m = \arg\min_{\beta} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta h(x_i; a_m) \tag{20}$$

4.2 From Boosting to Gradient Tree Boosting

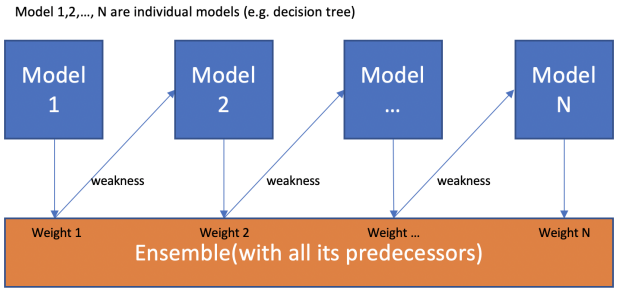


Figure 3: Pipeline of a GBM

Gradient Tree Boosting, which is what I implemented in the folder *Code/Gradient_boosting/src*, specializes to the case where the base learner $h(\mathbf{x}; a)$ is an L-terminal node regression tree (CART). At each iteration m, a regression tree partitions the $\mathbf{x}$-space into L-disjoint regions $\{R_{l,m}\}_{l=1}^{L}$ and predicts a constant value in each one [5]:

$$h(x; \{R_{l,m}\}_{l=1}^{L}) = \sum_{l=1}^{L} \bar{y}_{l,m} I(x \in R_m) \tag{21}$$

with $\bar{y}_{l,m}$ the mean of points in each region $R_{l,m}$, $Mean(x_i \in R_{l,m}(\tilde{}y_{i,m}))$

The parameters of the base learner defined above are the splitting variables and the corresponding split points that construct the tree. In simple terms, the split points locate the respective regions $R_{l,m}$ with a top-down *"best-first"* manner given a known splitting criterion.
More on splitting later on at the end of Section 4.

Regression trees Eq. (20) can solve independently in each $R_{l,m}$ region, that is the j-th leaf in the m-th tree. Because in Eq. (21) a single tree predicts a constant value $\bar{y}_{l,m}$ within each region $R_{l,m}$, the ERM solution to Eq. (20) reduces to a simple location estimate on the loss L:

$$\gamma_{l,m} = \arg\min_{\gamma} \sum_{x_i \in Rl,m} L(y_i, f_{m-1}(x_i) + \gamma \tag{22}$$

In the realm of least-squares loss L the γ which minimizes Eq. (22) is simply the mean of the individual scores in the leaf, as Gianvito showed in Eq. (9). Hence, γ can be interpreted as the parameter which controls how a leaf should respond in case multiple instances end up inside it.

$f_{m-1}(\mathbf{x})$ is then updated separately in each region $R_{l,m}$:

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu\gamma_{l,m}I(x \in R_{l,m}) \tag{23}$$

4.3 Implementation details

The steps that are described here can be found at *Code/Gradient_boosting/src/gbm.py* in the only class *GBDT*. All, but $2.A$, since the proposed implementation uses a different approach on splitting and it will be explained in greater detail in *How does the GBM split?*

4.3.1 Core pseudo-code

Input: a training set $\mathcal{D} = (x_i, y_i)_{i=1}^N$, an arbitrary differentiable loss function $L(y_i, f(x))$ and a number of bootstrap rounds M.

Algorithm:

1. Initialize the model with a constant value:

$$f_0(x) = \arg\min_\gamma \sum_{i=1}^N L(y_i, \gamma)$$

   If L is the square loss, γ is simply $\frac{1}{|\mathcal{D}|}\sum_{i=1}^N y_i$

2. for $m = 1$ to M:
   (a) Compute pseudo-residuals:

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)} \quad \text{for } i = 1 \text{ to } N$$

   (b) Fit a regression tree to the residuals r_{im} and create corresponding terminal regions R_{jm} for $j = 1$ to T, where T is the number of leaves.
   (c) for $j = 1$ to T compute

$$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma h_m(x_i))$$

   (d) Update

$$f_m(x) = f_{m-1}(x) + \gamma \sum_{j=1}^T \gamma_{jm} \mathbb{1}_{\{x_i \in R_{jm}\}}$$

3. Output $f_M(x)$

Cost analysis:

1. N
2. (a) $N\partial_f L$
   (b) $N \log T$
   (c) TN
   (d) T

Total GBM cost: $O(N + M(N + N\log T + TN + T) = O(MNT))$ [6, Subsection 4.1]

4.3.2 Parameters tuning

Gradient Boosting Tree has quite a few parameters to control, but mostly straightforward in meaning and application. I decided to add different layers of randomness to help against the risk of overfitting the data in the form of stopping conditions at each boosting round.

Stopping parameters:

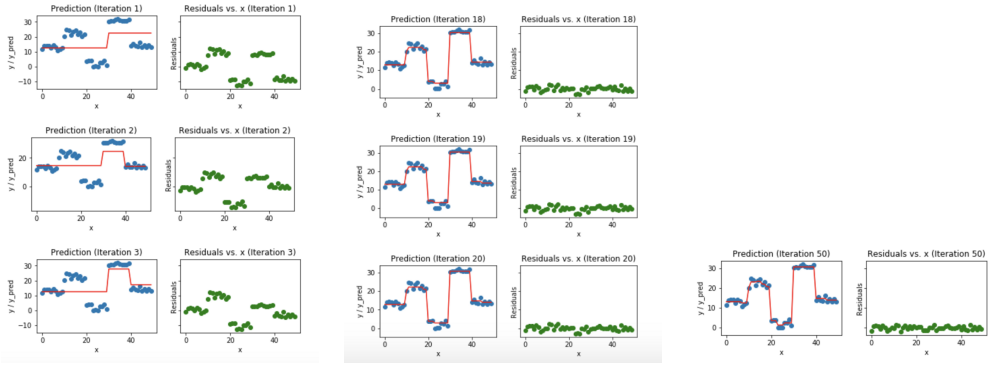
The maximum number of levels a node can reach, *max_depth*. Even though its functionality is trivial, it's a costly parameter to tune. [3] A better approach to the problem is to use *min_samples_split* and *min_samples_leaf* in conjunction, whose functionality has been covered at the end of Section 2.

The minimum information gain a split can achieve to be considered valid, *min_split_gain*, which basically defines the degree of chaos in the data. Entropy, steepest descent on residuals or a simple L2-norm are among the most common choices for the split loss gain function.

Learning parameters:

A really powerful meta-parameter is *early_stopping_rounds*, which makes the iteration over m stop if the validation score across the weak models hasn't improved for these many rounds. It is typically used in conjunction with the shrinkage rate (ν in Eq. (23)) which controls the learning rate of the composite model by scaling each weak model's weight by this same factor. Basically, it controls the influence a single model can have on the prediction, by guaranteeing that after fitting the residuals it can only take a small step in the right direction. It is proved that several subsequent small steps in the right direction score a better predictive accuracy in the end. Also, it helps reducing the variance of the model, making GBM a low bias, low variance model. Hurray! Literature suggested values for the shrinkage rate are below 0.1. Being so low, it means that the model needs a good number of weak models, typically between 32 and 100, to produce an accurate prediction. Combining these two parameters is the preferred way to control the number of trees per composite model, instead of using the *num_boost_round* parameter which should be left has high as possible.

Growing a lot of trees isn't always ideal, though, and can lead to overfitting if the residuals remain constant across multiple following iterations, as the figure below shows. From (b) to (c) 30 iterations have passed, but the model hasn't improved at all. Actually, it has gotten worse because of overfitting. The meta-parameter *best_iteration* has precisely the role of keeping track of the earliest iteration before things got stagnant across iterations.



(a) (b) (c)

Figure 4: Regression in action

Feature subsampling was also used to add more layers of randomness when deciding on the feature to split on through the *max_features* parameter as a percentage on the total number of features. Typical picks were 0.3, 0.5, but also 0.1. Bagging on the X instances was instead left out, for the sake of time.

Cross-validation should always be used.

4.3.3 How does the GBM split?

Splitting, as I briefly touched upon, is key to GBM performance.
The better the information gain out of a split, the better the split. It could also be used as a feature selection procedure, by counting the number of features on you which it split the most, as an indicator of the ones which achieved the most gain.

Instead of following common split loss gain functions, like the older pre-sorting algorithm (much like what Gianvito presented in its cost derivation) or the newer histogram-based algorithms and GOSS (Gradient-based One-side sampling), I decided to use a slightly different function which relies on the first and second (Hessian matrix) order statistics of the objective function, inspired by the paper documenting XGBoost implementation. [6, Section 2] The aim of XGBoost study was to find a scoring function whose use could be broaden to a wide range of objective functions.

Given a dataset $\mathcal{D} = (x_i, y_i)_{i=1}^N$ with $x_i \in \mathbb{R}^D$, $y_i \in \mathbb{R}$ a regularized objective function is defined as follows:

$$\phi = \sum_i L(y_i, f_m(x_i)) + \sum_k \Omega(f_k) \tag{24}$$

where

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda ||w||^2 \tag{25}$$

Here L is a differentiable convex loss function that measures the difference between the prediction $f_m(x_i)$ and the ground truth y_i. The second term Ω penalizes the complexity of the base learners, with γ penalizing on the increasing number of leaves T and the additional regularization term helping to smooth the learnt weights and prevent them from overfitting. Intuitively, the regularized objective will tend to select a model employing simple functions. When the regularization parameter is set to zero the objective falls back to traditional Gradient Tree Boosting.

For the loss function L I used L2-loss anyway, because it made computation easier and faster. But as we shall see below the beauty of this model is indeed to allow for an arbitrary loss function to be plugged in, granted that it is differentiable. The parameters f_k inside Ω map an instance x_i to its weight $w_q(x)$, which follows a tree structure function $q(x)$ assigning each instance x_i to a leaf j. Since the parameters of this model are functions (i.e., f_k) it can't be optimized in Euclidean space and it will follow an additive procedure, instead.

Formally, we add a coefficient, let's call it f_m as Friedman did in its original paper, that most improves the model accuracy given the regularized objective in Eq. (24):

$$\phi_m = \arg\min_{f_m} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + f_m(x_i)) + \Omega(f_m) \tag{26}$$

Basically, we greedily add the f_m that most improves our model according to Eq. (24). Second-order approximation can be used to quickly optimize the objective in the general setting, rather than first-order only. We can apply a Taylor series expansion up to the second-order w.r.t. to $f_m(x_i)$:

$$\phi \simeq \sum_{i=1}^N \Big[L(y_i, f_{m-1}(x_i)) + g_i f_m(x_i) + \frac{1}{2} h_i f_m^2(x_i) \Big] + \Omega(f_m) \tag{27}$$

where $g_i = \left[\frac{\partial L(y_i, f_{m-1}(x_i)}{\partial f_{m-1}(x_i)} \right]$ and $h_i = \left[\frac{\partial^2 L(y_i, f_{m-1}(x_i)}{\partial f_{m-1}^2(x_i)} \right]$ are first and second order gradient statistics on the loss function L. We can neglect the constant term $L(y_i, f_{m-1}(x_i))$ to obtain the following simplified objective at iteration m:

$$\tilde{\phi} = \sum_{i=1}^N \Big[g_i f_m(x_i) + \frac{1}{2} h_i f_m^2(x_i) \Big] + \Omega(f_m) \tag{28}$$

Define $I_j = \{i \,|\, q(x_i) = j\}$ as the instance set of leaf j. We can then rewrite Eq. (28) by expanding Ω with the definition provided at the start of the derivation. It follows that:

$$\tilde{\phi_m} = \sum_{i=1}^{N} \Big[g_i f_m(x_i) + \frac{1}{2} h_i f_m^2(x_i) \Big] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \tag{29}$$

$$= \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \lambda T \tag{30}$$

where $\sum_{i=1}^{N}$ has been split in $\sum_{j=1}^{T} \sum_{i \in I_j}$

Since it is impossible to account for all the $q(x)$ configurations, we assume a greedy approach that starts from a single leaf and iteratively adds branches s.t. from a given instance space I we get the children's as $I = I_L \cup I_R$. Being $q(x)$ fixed we can compute the optimal weight w_j^* for a single leaf j applying ***ERM*** to w_j:

$$w_j^* = \arg\min_{w_j} \sum_{i \in I_j} g_i w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 = - \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{31}$$

Plugging w_j^* in Eq. (30) the corresponding optimal value is:

$$\tilde{\phi_m}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \lambda T \tag{32}$$

The optimal value $\tilde{\phi_m}(q)$ can be seen as the scoring function to evaluate the quality of a tree structure q that I anticipated at the beginning. As desired, it is derived for a wider range of objective functions. Basically, any differentiable function, really. Being used with this aim, the score of the children instances I_L and I_R should sum up to the one of the node instances I. Otherwise, this means the split produced a loss reduction, computed as:

$$\mathcal{L}_{split} = \frac{1}{2} \left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma \tag{33}$$

GBDT implementation uses $\mathcal{L}_{split}$ for evaluating the split candidates.

Split finding implementation can be found at *Code/Gradient_boosting/src/gbm.py* in the private methods of the class *GBDT* that are self explanatory in computing the weight for each leaf j and the corresponding information gain. The method *GBDT.build()* implements the additive computation of first and second order statistics and the comparison between information gain on the node instance set I with the children's I_R and I_L. It's all summed up in Figure 5.

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted(I, by* $\mathbf{x}_{jk}$*)* **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

Figure 5: Split finding pseudo-code

4.4 Closing thoughts

Unfortunately, the GBM implementation didn't achieve really good results w.r.t. the other approaches as you can see from the below table in Section 5. Maybe I should have gone deeper in the *XGBoost* paper instead of sticking to the first implementation, or I should have probably settled for a simpler implementation on residuals. With that being said, this attests that even though GBM is rather simple in principle, it's not that easy to get it right.

5 Experiments and Results

The problem is approached with a *divide and conquer* strategy: a regression model is trained for each given participant of the market. According to this method, each regression model is cross-validated using the 5-Fold Cross Validation strategy. At the end, each participant has a list of 5 own regression models tuned with the best parameters. The output for each time series is predicted as the mean of the 5 regression models corresponding to the participant.

Random combinations of parameters were used to train the algorithms. The results are shown in the table 1. Performance varies across different time series, with the exception of some of them which are well predicted by all the algorithms.

6 Conclusion

Time-series analysis is a fundamental step towards the path of building a machine learning model for forecastings. Different levels of analysis can be approached: from the model decomposition to the inspection of stationarity and various types of correlations. Lag features design is one of the most important aspect which could have a positive impact to the quality of a machine learning model.

Random forests and Gradient boosting are powerful methods to address both regression and classification problems. They are easily available as off-the-shelf modules from *scikit-learn* library in Python. In addition, they are also quite easy to understand and implement. However, a high number of parameters has to be cross-validated to exploit the maximum potential of the models and usually this is feasibile only with high performance and parallel computing resources.

Table 1: Root Mean Squared Error (RMSE)

Time series		Test set performance		
participant	technology	Random Forest	GBDT	LightGBM
participant_116	ccgt	0.0	0.0	1.0
participant_116	ccgt400	0.654	1.647	0.0
participant_116	ccgt800	0.755	0.654	0.755
participant_116	pump	0.0	1.0	0.0
participant_116	reservoir	0.0	2.0	0.0
participant_120	ccgt800	0.0	0.0	0.0
participant_146	pump	0.534	0.534	0.377
participant_146	reservoir	0.925	0.925	1.0
participant_158	ccgt400	0.0	1.0	0.0
participant_158	ccgt800	0.755	0.925	0.845
participant_158	pump	0.534	0.845	0.845
participant_158	reservoir	0.0	1.0	0.0
participant_176	ccgt800	0.0	0.845	0.377
participant_21	reservoir	0.0	0.0	0.0
participant_60	ccgt400	0.0	1.0	0.0
participant_75	ccgt400	0.925	0.845	0.534
participant_75	ccgt800	0.845	0.845	0.0
participant_87	ccgt	0.377	0.925	0.377
participant_87	ccgt400	0.0	2.329	0.534
participant_89	ccgt400	1.133	0.654	1.133
participant_89	ccgt800	0.377	0.845	0.534
participant_89	coal	0.0	1.0	0.0
participant_89	reservoir	0.0	1.0	0.0
participant_96	ccgt400	0.377	0.925	0.534
participant_96	ccgt800	0.925	0.0	0.654
participant_96	coal	0.654	2.171	0.534
participant_96	pump	0.925	1.463	0.925
participant_96	reservoir	1.309	1.253	0.654
Global RMSE		Random Forest	GBDT	LightGBM
Validation		0.774	1.338	0.583
Test		0.601	1.113	0.562

References

[1] Murphy K., *Machine learning: a probabilstic perspective*, MIT Press, 2012

[2] Hastie T. *et. al.*, *The Elements of Statistical Learning*, Springer, 2009

[3] *scikit-learn* official library documentation:
`https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html`
`https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html`
`https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html`

[4] Breiman L., *Random Forests*, Machine Learning, 45, 5–32, 2001

[5] Friedman J., *Greedy Function Approximation: A Gradient Boosting Machine*, `https://statweb.stanford.edu/~jhf/ftp/trebst.pdf`, 1999

[6] Chen, T, Guestrin, C., *XGBoost: A Scalable Tree Boosting System*, `https://arxiv.org/abs/1603.02754`, 2016