Exploring XGBoost by solving a peak prediction task in time series

Andrei Secuiu (s4260732)
Lukke van der Wal (s4076788)
Roberto Schininà (s4612299)
Vlad Muscoi (s4718267)

Group 10

February 1, 2025

Abstract

This paper explores the application of the XGBoost algorithm to predict peaks in time series data, specifically focusing on daily energy consumption. Using a dataset from the Los Alamos Public Utility Department, which includes power consumption data from 1,757 domestic users over several years, the study aims to predict the maximum weekly energy consumption over the next seven days based on the previous seven days of data. The research is inspired by previous work on peak prediction and seeks to provide a deeper understanding of the chosen algorithm. The results demonstrate the effectiveness of XGBoost in solving the peak prediction task, with a detailed analysis of their performance and limitations. The code can be found in our GitHub

1 Introduction

Time series data is a sequence of data points collected or recorded at specific time intervals where the ordering is important. This type of data is often used to track changes over time and can be found in various fields such as finance, economics, weather forecasting, and more. Each data point in a time series is typically associated with a timestamp, allowing for the analysis of trends, patterns, and seasonal variations.

Peaks in time series data represent points where the value reaches a local maximum over a pre-defined time period. These peaks can indicate significant events or changes in the underlying process being measured. For example, in weather data, a peak could indicate the highest temperature recorded during a specific period.

Predicting peaks in advance can support decision-making in various application domains, such as predicting large amounts of rainfall to aid flood warning systems [Goodarzi et al., 2019]. The task of predicting peaks should not be confused with traditional and well-studied peak detection, the goal of which is to identify the peaks of a query time series.

1.1 Inspiration and problem description

Our paper is highly inspired by the work of Kruger et al. [2024]. The original authors explored a vast selection of algorithms aimed at solving the task of peak prediction. We take on a similar approach, but limit ourselves to only one algorithm: XGBoost. Thus, the aim of this paper is to provide a theoretical understanding of said algorithm and then use it to solve the peak prediction problem.

The dataset is provided by the Los Alamos Public Utility Department (LADPU) in New Mexico, USA, [Dryad, 2025] and it consists of data collected from multiple smart meters. These are electronic devices that measure the energy consumption of end users and the time of consumption which makes them great sources for time series data for supervised learning tasks. The dataset contains power consumption data from 1,757 domestic users, one sample every fifteen minutes, and spans several years (from 2013 to 2019, though some years' data is very sparse).

We use the same dataset and try to solve the same task as used in the aforementioned paper. In particular, we want to predict two properties of the peak: its value and its position. This task can then take the form of a multi-target task, predicting the value and position simultaneously, or two different single-target tasks, predicting the value and position individually. For our study, we have chosen to pursue the latter approach. In particular, the prediction could be performed on individual users and trying to predict a future consumption behavior for each based on present data, or it could look at the entire user base and anticipate the consumption behavior of an entirely new user. We have chosen to solve the latter problem. On one hand, it requires developing a single prediction algorithm, which simplified the project to a manageable level. On the other hand, a company could be interested in forecasting consumption behaviors for users that are newly added to the grid, where past data is not available. The question is not without merit.

1.2 Outline

The paper is structured as follows. First, we provide the theoretical background of the XGBoost algorithm. Afterwards, we describe our methods and report every relevant detail regarding the algorithm's setting and data. Then, we report our findings and provide an interpretation of the results. Lastly, in the discussion section, we present our conclusions, as well as any comments and thoughts we gathered along this project.

2 Theoretical framework - XGBoost

As mentioned in the introduction, one of the goals of this paper is for us to get a better understanding of the algorithm used. An algorithm can be applied effectively only after one understands the role each hyperparameter plays in the learning process, as well as the limitations of the method presented. Hence, in this section, we shall present the theory concerning XGBoost in more detail.

Even nowadays, methods that rely on trees (in particular Random Forests and Tree Boosting) are pretty popular in many tasks due to their surprisingly (why surprisingly?) good results. For instance, tree boosting has been shown to give state-of-the-art results on many standard classification benchmarks [Li, 2012]. XGBoost, a scalable machine learning system for tree boosting, is a popular method used in many applications due to its scalability and speed. This framework gives an excellent performance (state-of-the-art in 2016) in a variety of tasks such as store sales prediction, high energy physics event classification, web text classification, customer behavior prediction and many more.

The following paragraphs will introduce the theoretical framework on which XGBoost relies according to its original paper written by Chen and Guestrin [2016] (with slight changes in the notation where appropriate for mathematical clarity). We shall also refer to the XGBoost documentation (last accessed on 28/1/2025) for details concerning the current implementation we have used. While this section is written with *regression* in mind, it can be easily adapted to support multi-class classification. Classification will be described towards the latter part of the section.

2.1 Defining regression and classification trees

Let us consider a data set containing n data points (instances) and m features, where the target variable y is a real-valued scalar: $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}$, $|\mathcal{D}| = n$, $\boldsymbol{x}_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$. Our task is to define a predictor function $\hat{y} = \phi(\boldsymbol{x})$ such that $\hat{y}_i = \phi(\boldsymbol{x}_i)$ is "closest" to y_i. The closeness, as we shall see later, is quantified by the regularized loss function.

Regression. While in the course we have seen trees (and forests) introduced with the purpose of classification and relying on discrete categorical variables as covariates, trees can also be used for regression tasks. The basic functionality is illustrated in Figure 1. To that end, one preserves the binary tree structure of decision trees, but to each of the end nodes of the tree (the leaves) there is a score assigned. Next, instead of relying on a single tree for the predictions (a single score on the leaf is rather uninformative), one constructs multiple trees with potentially different structures and scores; let their number be K. The final prediction is the sum of the predictions for all K trees. This is very reminiscent of the concept of random forests that we have encountered in the course! Just like there, one builds multiple trees that individually are not perfect; but as long as each does a satisfactory job and as long as they are diverse enough (i.e. their structure is sufficiently randomized), then their sum (the equivalent of the majority vote in Random Forests) shall be a good prediction.

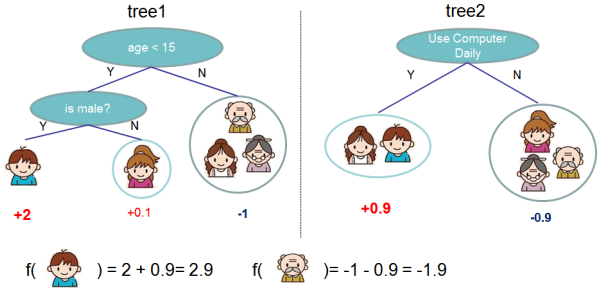


Figure 1: Trees used in the context of regression. To decide on the output y of a new instance $\boldsymbol{x}$, $\boldsymbol{x}$ is passed through each of the trees. The final prediction is the sum of all scores. Picture taken from Chen and Guestrin [2016]

Formally, let $\mathcal{F}$ be the space of all regression trees. As we have seen before, a regression tree consists of two components: the scores w assigned to the leaves and the structure q of the tree. The structure of the tree is the more complicated mathematical object and it will be the trickier part to optimize later. q is a function that surjectively maps all possible input vectors $\boldsymbol{x} \in \mathbb{R}^m$ to a set of leaves. Let T be the number of leaves for the tree q and let us label them by $\{1, 2, \ldots, T\}$. Then the regression tree f with structure q and scores $w \in \mathbb{R}^T$ is represented as:

$$q : \mathbb{R}^m \to \{1, 2, \ldots, T\}$$
$$f : \mathbb{R}^m \to \mathbb{R}, \quad f(\boldsymbol{x}) = w_{q(\boldsymbol{x})}$$

where leaf i is assigned the score w_i. Let us consider K regression trees, each with an independent structure and scores. Then the prediction function is defined additively:

$$\hat{y} = \phi(\boldsymbol{x}) = \sum_{k=1}^{K} f_k(\boldsymbol{x}), \quad f_k \in \mathcal{F} \tag{1}$$

Classification. Regression trees can also be used for classification. If the target variables y_i can belong to a set of p classes, that is $y \in \{C_1, \ldots, C_p\}$, then the overall function ϕ is modified such that it outputs a probability distribution: a vector with the probabilities that the new instance represented by $\boldsymbol{x}$ belongs to each of the p classes. The returned class is the one with the highest probability, but a user can also opt to obtain the probability vector directly.

The implementation of the classification method is somewhat more involved, falling under the umbrella of Gradient Boosted Decision Trees (GPDT). Our understanding comes mainly from an example implementation in the documentation (link - last accessed on 28/1/2025) and an answer provided in a Stackexchange post (link - last accessed on 28/1/2025). More details on the training part follow in Section 2.8, but the rough idea is that

XGBoost effectively separates the initial data set into p separate data sets, sorted by the classes C_j: $\mathcal{D}_j = \{(\boldsymbol{x}_i, y_i) \in \mathcal{D} \mid y_i = C_j\}$. Then XGBoost trains p regression trees in parallel; a training iteration is called a *boosting round*. Each boosting round, the p trees are optimized in parallel on $\mathcal{D}_j$ to minimize the loss function for the tree corresponding to each class C_j, obtaining a raw score z_j (also known as *logit*). It is important to note that this loss function involves the probabilities of belonging to classes for all training instances. The p scores are then passed through the softmax function, obtaining a probability distribution vector: the probabilities of belonging to each class. Using these new computed probabilities, the next boosting iteration can start.

The exact representation of the classes for the purpose of this algorithm is not very important because the data set is essentially split into separate data sets for each class. To test this out, we have run the algorithm with both the raw class representation (numbers from 1 to 7) and with one-hot encoded vectors. The results were identical, so we have chosen to remain with the simple initial representation.

2.2 The loss function

In the context of boosted trees, the *loss* function is also called the *objective* function. To avoid confusion, we shall stick with the course terminology, but the reader should keep in mind this naming when reading the sources.

The loss function is made of two components: a component $l : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_{\geq 0}$ which measures how "far" the target value y is from the predicted value $\hat{y}$ and another regularization component $\Omega : \mathcal{F} \to \mathbb{R}_{\geq 0}$ which ensures that the trees do not overfit the data. The overall loss function is:

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} l(\hat{y}_i, y_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{2}$$

Overfitting has been extensively discussed in Chapter 6 of the Lecture Notes, hence we shall not explain its dangers or justify why Ω is an important component. However, it is worth paying more attention at the form of each component and argue it.

The l component is a twice continuously differentiable function, an important condition for GBDT. In the case of our problem, we have studied the available choices (link to the documentation - last accessed on 28/1/2025) and we have opted to run with the default choices appropriate in each application, following the principle of parsimony. For regression, the l component is the usual *mean squared error*. For classification, we have used the *cross-entropy loss* based on softmax (in the field is given the shorthand name *softmax loss function*; it is an abuse of language, but we will stick with it), which is easier to describe on the data set $\mathcal{D}$ directly in the context of training the classifier (see Section 2.1). Let z_{ij} be the logit (raw score) of the i-th instance to belong to class C_j, $\hat{p}_{ij}$ denote the probability of the i-th instance to belong to class C_j (computed for each boosting round via softmax) and δ_{ij} be 1 if the instance i is in class j, and zero otherwise. Then:

$$l(\mathcal{D}) = -\sum_{i=1}^{n}\sum_{j=1}^{p} \delta_{ij} \log(\hat{p}_{ij}) = -\sum_{j=1}^{p}\sum_{i \in \mathcal{D}_j} \log(\hat{p}_{ij}) \tag{3}$$

$$\hat{p}_{ij} = \frac{\exp(z_{ij})}{\sum_{k=1}^{p} \exp(z_{ik})} \tag{4}$$

One clear advantage of the softmax loss function over the simpler loss function that counts misclassifications is that it is twice differentiable in each score. Furthermore, it adds a level of detail to the quantification of the classification performance that is simply lost when one counts mismatches alone. To see that, consider the following example on a simplified scenario: binary classification, with a new training instance whose true class label is 1. One hypothetical algorithm will assigns a probability of 49% of belonging to the true class, while another assigns a probability of 1%. The mismatch loss function will say that both algorithms are equally poor, while the cross-entropy (with softmax) loss function assigns values 0.713 and 4.605 respectively. An almost correct classification is given more importance than a "confidently incorrect" classification.

The regularization component Ω is meant to penalize trees that try to overfit the data. It also consists of two components: one that constraints the scores to have low magnitudes and another that punishes an excessive number of leaves. Mathematically:

$$\Omega(f) = \Omega(q_{w(.)}) = \frac{1}{2}\lambda \|w\|^2 + \gamma T \tag{5}$$

The regularization on w is L2 (just like in the original paper Chen and Guestrin [2016]), though the current Python implementation also allows for L1 regularization. We have chosen to maintain the L2 regularization alone both for its relation to the original paper and because the L1 regularization is known more for its model selection capabilities (i.e. setting many w_i to zero), while we wanted only an overall restriction on the magnitude of the scores. The γ term restricts the creation of new nodes in the decision tree q: if the gain after a new split is too low, then the split is discarded.

2.3 Finding the scores

Until specified, the following subsections focus on regression trees. There will be a subsection which details the extra steps involved with training boosted trees for classification.

Finding the optimal tree ensemble which minimizes the loss function is a task which must be done on two fronts: on the optimal scores w and on the optimal tree structure q. There is sadly no algorithm to find the exact optimum in general, thus we must settle on approximate solutions. The whole process about to be developed relies on a chain of successive and sensible approximations. The approximate solutions for w have closed forms, but the search for q is far from easy. In this section, we present the solution to w; the following sections shall focus on the tree structures q.

Because the loss function contains components that depend explicitly on functions f_k and not only on the scores,

an optimization based on traditional techniques in the Euclidean space (e.g. gradient descent) is not possible. The solution (a common theme for these algorithms) is to search greedily, that is via a greedy algorithm, trees that most improve the performance on a given ensemble of trees; this is the first approximation in the chain: one that finds the approximate optimum via a greedy search.

Formally, let $\hat{y}_i^{(t)}$ be the prediction of the tree ensemble for the i-th instance at the t-th boosting iteration. To the given ensemble of trees from iteration $t-1$, we add a new tree f_t which most improves the loss function at the current iteration:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i,\ \hat{y_i}^{(t-1)} + f_t(\boldsymbol{x}_i)) + \Omega(f_t)$$

Notice how because the improvement is done only on the new tree added on the t-th iteration, we can ignore the regularization component of the other $t-1$ trees (which are fixed and not subject to further changes). Because we wish to keep l as general as possible, we cannot exploit yet the properties of any given loss function. Thus, the second sensible approximation is to use the second-order Taylor expansion of $l = l(y, \hat{y})$ with respect to the prediction variable $\hat{y}$, centered around the predictions $\hat{y}_i^{(t-1)}$ at iteration $t-1$. The derivatives involved in this Taylor expansion are denoted by $g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$ and $h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$.

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n} \left[l(y_i,\ \hat{y_i}^{(t-1)}) + g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i) \right] + \Omega(f_t)$$

Removing irrelevant constant terms which are not impacted by f_t, one has to optimize the following loss function at step t which depends only on the current tree f_t.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i) \right] + \Omega(f_t) \tag{6}$$

To proceed further and find the optimal scores w, it is important to rewrite the decision tree $f_t(\boldsymbol{x}_i)$ to highlight the scores explicitly. Let us fix the tree structure q. Recall that if the structure of f_t is such that $\boldsymbol{x}_i$ is mapped to leaf j, then the output is $f_t(\boldsymbol{x}_i) = w_j$. Hence, it is advantageous to reorder all instances $\boldsymbol{x}_i$ by the way they are mapped through f_t; let $I_j = \{i \mid q(\boldsymbol{x}_i) = j\}$ be the instance set of leaf j. Then:

$$\begin{aligned}
\mathcal{L}^{(t)} &= \sum_{i=1}^{n} \left[g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i) \right] + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 + \gamma T \\
&= \sum_{j=1}^{T} \sum_{i \in I_j} \left[g_i w_j + \frac{1}{2} h_i w_j^2 \right] + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 + \gamma T \\
&= \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\lambda + \sum_{i \in I_j} h_i \right) w_j^2 \right] + \gamma T \\
&= \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2} \left(\lambda + H_j \right) w_j^2 \right] + \gamma T
\end{aligned}$$

where $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} H_i$. Then the optimal scores w_j^* for a given tree structure q are easily found by taking the gradient w.r.t. w (the components are independent from each other, so there is no constraint to be included via a Lagrange multiplier). The optimal weights and the loss evaluated at these optimal weights for a given q are:

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \tag{7}$$

Notice how the larger λ becomes, the smaller the magnitudes for w_j^*. This is the intended behavior of regularization!

2.4 Finding the tree structure

Now that we know how to find the best scores w for any given tree structure, it is time to search for the best tree structure q_{opt}. Notice how even if one can find the best w for any given q, it is still possible to rank the performance of different q evaluated at the optimal scores through Eq. 7. This equation plays a role akin to the impurity score (e.g. the entropy) in the context of constructing decision trees, except that it is derived for a wider range of loss functions.

We would like to draw the attention of the reader to the fact that the problem of finding a regression tree ensemble with an arbitrary loss function $\mathcal{L}$ has been reduced to finding the optimal tree structure of a single new tree, where the optimal scores are already known and where different structures can be assigned performance scores akin to the impurity measure of the usual decision trees. To that end, finding the best structure q_{opt} will rely on the same ideas as for normal decision trees!

Finding the best possible structure q by enumerating and evaluating all possibilities is impossible. The problem is two-fold. On one hand, the features $\boldsymbol{x} \in \mathbb{R}^m$ are *continuous*; to construct decision trees, one has to find values s_{jk} among a continuum of values with respect to which to compare a feature x_k and perform the binary split (e.g. at a given node j "if $\boldsymbol{x}_i$ has $x_k \leq s_{jk}$, then the instance goes to the left"). On the other hand, even with a given split, enumerating all choices is too expensive. The need of an approximate method is imperative.

Thus, borrowing from decision trees, one uses a greedy algorithm that starts from a single leaf and iteratively adds branches to the tree. A candidate leaf can be split (using some feature x_k and some split s_{jk}) into two other leaves: Left and Right. Let I_L and I_R be the instance sets of left and right nodes after the split; if I is the instance set of the candidate leaf, then $I = I_L \cup I_R$. Analogously, define $G_{L/R} = \sum_{i \in I_{L/R}} g_i$ and $H_{L/R} = \sum_{i \in I_{L/R}} h_i$. The *gain* after a split is defined as the decrease in impurity after the split (the loss reduction), compared to the initial situation. For our problem, it takes the following formula:

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma \quad (8)$$

The lone γ term is the penalty of introducing another leaf into the tree, reducing the overall gain. We shall opt for the split (i.e. for a feature x_k and for an s_{jk}) that maximizes the total gain. The algorithm stops if all the gains of all splits are negative (i.e. the gain increase from a split does not offset the threshold γ), or if the leaf has a single member in the population.

This algorithm, named in the paper the *exact greedy algorithm*, is displayed in Alg. 1. At a given node j with an instance set I, we search through all features from x_1 to x_m. For each feature x_k, we compute all possible split candidates and compute their gains. We save only that which yields the maximum gain, among all splits and among all features. The output will be the split s_{jk}. Note how it does not matter which exact continuous value s_{jk} one uses, as long as the instance set is partitioned in the same manner I_L and I_R. Thus, it suffices to set $s_{jk} = (\boldsymbol{x}_i)_k$ (the value of the k-th component of the i-th instance, where i is chosen to maximize the gain). The list of $\boldsymbol{x}_i$, represented via I, is sorted ascending via x_k to maximize the efficiency of the algorithm.

Algorithm 1 The exact greedy algorithm, as explained in Chen and Guestrin [2016]. Some variable names have been updated for consistency with our report

Input: I, instance set of the current node
Input: m, 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 $\boldsymbol{x}_{jk})$ **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    gain $\leftarrow \max\left(\text{gain}, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda}\right)$
  end for
end for
Output: The split with the maximum gain

2.5 Approximate greedy algorithms

The exact greedy algorithm for finding the best split at each node (i.e. the best tree structure q) is an improvement in efficiency, but enumerating all possible splits for all possible features (even after sorting the instances via each feature) is computationally demanding. It is a powerful method, but because XGBoost is usually a memory-consuming task, the exact greedy method may take additional time if the memory is insufficient; there is a similar issue with distributed computations. Thus, the authors of Chen and Guestrin [2016] propose another approximate framework which increases the time efficiency drastically without sacrificing many of the results.

The essence of the approximate algorithm is to follow the same exact greedy search procedure, but on continuous data which has been partially aggregated by *binning*. Firstly, a set of candidate splitting points S_k for feature x_k is chosen according to a method named Weighted Quantile Sketch (detailed later). Then the continuous feature x_k is binned according to the splits in S_k. Lastly, the exact greedy algorithm is performed, but only on the binned data using the splits found already in S_k. The algorithm showcasing how this is done is given in Alg. 2.

Algorithm 2 The preliminary split finding step, as given in Chen and Guestrin [2016]. Descriptions are broad

Input: Same as Alg. 1
▷ Step 1: split candidates proposal
for $k = 1$ to m **do**
  Propose candidate splits $S_k = \{s_{k1}, \ldots, s_{kl}\}$ via Weighted Quantile Sketch
  Proposal done locally or globally
end for
▷ Step 2: binning via splits
for $k = 1$ to m **do**
  $G_{kv} \leftarrow= \sum_{j\in\{j \mid s_{k,v-1}<x_{jk}\le s_{k,v}\}} g_j$
  $H_{kv} \leftarrow= \sum_{j\in\{j \mid s_{k,v-1}<x_{jk}\le s_{k,v}\}} H_j$
end for
▷ Step 3: exact greedy algorithm
Follow the steps in Alg. 1 to find the split that yields the maximum gain
Use only the splits proposed in Step 1 and the aggregated variables

The paper Chen and Guestrin [2016] goes into detail about the possibility of deciding splits S_k locally (at each node j) or globally (setting a "best" split per feature for the entire tree: $s_{jk} = s_k$). As far as we understood from the documentation, only the global method is implemented.

Weighted Quantile Sketch. The criteria by which the splits are proposed is based on the percentiles of a feature distribution. To add more detail to that, let the multi-set[1] $\mathcal{D}_k = \{(x_{1k}, h_1), (x_{2k}, h_2), \ldots, (x_{nk}, h_n)\}$ denote the values of the k-th feature, together with their second derivative statistics computed at each boosting iteration. We define a rank function $r_k : \mathbb{R} \to [0, +\infty)$

$$r_k(z) \equiv \frac{1}{\sum_{(x,h)\in\mathcal{D}_k} h} \sum_{(x,h)\in\mathcal{D}_k,\ x<z} h \quad (9)$$

which represents the proportion of instances whose h statistic is lower than a number z; it resembles a Cumulative Distribution Function, but the key difference is that it is concerned with h rather than the values x themselves. The candidate split points $\{s_{k1}, \ldots, s_{kl}\}$ (where l is the number of splitting candidates, at most n) are chosen such that

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \epsilon, \quad s_{k1} = \min_i x_{ik}, s_{kl} = \max_i x_{ik}$$

In the above, ϵ is an approximation factor which affects the degree of binning x_k undergoes. There are roughly $1/\epsilon$ candidate splits proposed, so there is no need to specify

[1] In a multi-set, elements are allowed to be repeated

l if ϵ is given. Note how the splits s_{kj} represent quantiles of the rank which ensure that intervals where x_k is aggregated make the changes in rank to be at most ϵ.

One may ask why it is important to use the second derivative numbers h as weights, and not some other quantity (or even the values x_k themselves). The role of h_i becomes apparent after rewriting the loss in Eq. 6 to organize the square in f_t:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} \frac{1}{2} h_i \left(f_t(\boldsymbol{x}_i) + g_i/h_i \right)^2 + \Omega(f_t) + \text{const.}$$

where the constant depends on g's and h's, but not f_t. This is the weighted squares loss function, where the weights are h_i and the labels are $-g_i/h_i$! Hence, to rank the "importance" of various instances x_k, we use their weights h_i in the splits. There may be some intuitive reasoning behind this. We believe that on the performance surface of $\mathcal{L}^{(t)}$, points where the curvature is strong are given more importance than points where the surface is almost flat because those regions are very informative. Where the algorithm has many geometrical valleys, hills, saddles etc. to navigate towards the minimum, it is important to retain as much of the information in that region as possible by not aggregating many x_i from that part. On the flip-side, wherever the performance surface is flat and boring, we can afford to aggregate the instances without much risk.

Nonetheless, the exact mechanism by which the split candidates are found in order to obey the rank constraint go beyond the scope of our report. The authors of Chen and Guestrin [2016] propose an algorithm which is *theoretically guaranteed* to maintain a certain accuracy level ϵ. It relies on proposing a data structure called "quantile summary" which supports merging and pruning operations. In their appendix, this data structure is properly introduced and its accuracy-preserving properties are proven. However, after reading the section, we found it to be technical and give little insight into the inner workings of XGBoost. We refer the interested reader to the original paper.

2.6 Other overfitting prevention methods

Besides adding a regularization term to the loss function, XGBoost implements two other ways that prevent overfitting. The first is called **shrinkage**, which plays an equivalent role to the learning rate in stochastic optimization. What shrinkage does is to multiply the newly added weights of a tree by some factor η after each step of boosting. Its purpose is to shrink the influence of each individual tree, and leave more room in the future for trees that further improve the model.

The second method relies on **subsampling**. A user has the option to subsample a fraction out of the total number available of either *rows* (training instances) or *columns* (features). In the paper Chen and Guestrin [2016], the authors claim according to user feedback that column subsampling is more effective than row subsampling in the prevention of overfitting, besides speeding up the computations. Subsampling is also part of Random Forest; we believe that feature subsampling is an important part of the algorithm because subsampling ensures that the response variability across the forest (regression tree ensemble) is maximized. Response variability is a core advantage in minimizing the generalization error, which has been proven via Breiman's theorem. While the theorem is not exactly concerned with regression trees, it must have some applicability to them as well. It would be in the same spirit as the Central Limit Theorem, which while provable easily for the i.i.d. case (with the finite first and second moments condition), it is still applicable in more general scenarios.

2.7 Miscellaneous aspects

Dealing with sparse data. Real-world data can be sparse due to different causes such as missing values, frequent 0 entries in the statistics and artifacts of the feature engineering. To deal with such cases, the algorithm will decide on a default direction which will be chosen if no data is available. The optimal default direction is learned from the existing data; the procedure is summarized in Algorithm 3 of Chen and Guestrin [2016]. The sparsity aware algorithm runs faster than other versions.

System design choices to improve efficiency. There are two notable design choices that affect the performance of the XGBoost framework:

- *Column Block for Parallel Learning*: To reduce the cost of sorting, the data is stored in memory units called blocks. Data in each block is stored in the compressed column (CSC) format, with each column sorted by the corresponding feature value. Multiple blocks can be used with the approximate algorithm by allowing different blocks to be distributed across machines or stored on disk. Because the blocks are sorted, the quantile finding step becomes a linear scan over the sorted columns.
- *Cache-aware Access*: Another optimization technique used is Cache-aware Access. Accessing values from memory, which can become expensive, is avoided by storing gradient statistics in the cache.

2.8 Learning a classifier

With the context of regression trees and their training, we can follow up on Sections 2.1, 2.2 and add further detail to the description on how classification is performed. The limitation of a regression tree ensemble is that it can output a single value $y \in \mathbb{R}$, whereas the target variables for a classification task are $y \in \{C_1, \ldots, C_p\}$. Thus, one needs not only a mechanism to somehow convert real values to classes, but also manage learning all p classes in parallel.

The solution is to use not a single regression tree ensemble, but p regression tree ensembles *in parallel*. Their raw scores, named logits, are some real numbers which can be converted into probability vectors by applying the softmax function on them. The overall output will be a probability distribution describing how likely a new instance $\boldsymbol{x}$ belongs to each category C_j; the returned category is the one with the highest probability.

Mathematically, let us assume that the class C_j is learned by the j-th regression tree ensemble ϕ_j. The tree ensemble at the t-th boosting iteration is denoted by $\phi_j^{(t)}$. For the i-th training instance, its overall score upon being passed through $\phi_j^{(t)}$ is $z_{ij}^{(t)} \in \mathbb{R}$: some real number. We can arrange the scores of $\boldsymbol{x}_i$ in a p-dimensional vector, obtaining $\boldsymbol{z}_i^{(t)} = \left(z_{ij}^{(t)}\right)_j$. The probabilities of belonging to each class can be then computed using Eq. 4 and arranged in a vector format; to that equation, we can add the index (t) to represent the fact that those are computed at the t-th boosting iteration.

The summary of the training process (as we understood it) for XGBoost classification is displayed in Alg. 3. Because the method essentially has to run the regression tree ensemble p times, the use of XGBoost for classification is much more time expensive than for the regression task; we have noticed that empirically with our grid searches.

Algorithm 3 High level overview of XGBoost classification

Given: the training data $\mathcal{D}$
probability_vector $\leftarrow \frac{1}{p}$ ▷ initialize the vector
while stopping condition on iterations t not met **do**
▷ Begin p regression tree training procedures
for $j = 1$ to p **do**
Train the new tree f_t to be added to $\phi_j^{(t-1)}$ via the greedy method described in the previous sections
Optimize $\tilde{\mathcal{L}}^{(t)}$ from Eq. 7. Use softmax loss function 3 and probability vector from *previous iteration*
Add optimal tree f_t to construct $\phi_j^{(t)}$
for $i = 1$ to n **do**
Compute z_{ij} using $\phi_j^{(t)}$
$z_i[j] \leftarrow z_{ij}$
end for
end for
probability_vector $\leftarrow$ softmax (z_i)
end while
predicted_class $\leftarrow$ argmax$_j$ (probability_vector $[j]$)
Output: predicted_class or probability_vector, depending on the user

For a classification problem, one can also use other loss functions. We have opted to tailor our explanation only to the one that we have used in our research and which was most easy to explain.

It is interesting to remark how when using the softmax loss function, the training for each tree ensemble for each category is essentially parallelized on data sets $\mathcal{D}_j$ (apparent from Eq. 3). However, it would be a mistake to say that the training done for all categories is fully independent from each other. This is because the different tree ensembles "communicate" information between each other via the probability vector at each boosting iteration. All aim to optimize the same loss function, which is computed using the probability vector for all instances, concerning all classes.

3 Methods

In the following section, we present all relevant details concerning the experiment design used within our study. We begin by presenting a general overview of the algorithm followed by an explanation on how we pre-process our data. The section ends with an explanation on the hyper-parameters we have decided to optimize, as well as the optimization method.

Algorithm overview. The raw data has been taken from Dryad [2025], consisting of energy readings sampled every 15 minutes from 1825 meters over a time span of 7 years. We have found that the raw data needed a fair amount of cleaning and pre-processing, as it contains both duplicate and missing values. After the initial cleaning, the time series of each user spanning 7 years is converted via sliding windows into another format appropriate for the task at hand, with both features and target labels. This cleaning processes is explained at length in section 3.1.

After having finished the data pre-processing, it was important to obtain some baseline results with which to compare the XGBoost outputs. We chose to look at two interesting baselines. For regression, we focused on linear regression; it is a simple, yet powerful technique which we have discussed at length during the course. Our use of linear regression was motivated mainly by the following: (1) the Lecture Notes made it clear that it is common practice in time series tasks to use linear regression as baseline due to its unexpected effectiveness in general and (2) the paper where we took inspiration from the project Kruger et al. [2024] found that linear regression was one of the most effective techniques at predicting peak values. For classification, we have used another baseline called PrevWeek. PrevWeek is a simple, yet natural choice in which the position of next week's peak is assumed to be the same as in the current week. In the not unreasonable assumption that consumption patterns for users are periodic functions (with the period being a work week + weekend), PrevWeek is actually expected to returned the correct result more often than not.

Lastly, the hyper-parameter search has been performed and our final answer for the problem could be returned. A visual representation of the architecture of our code is represented in Figure 2.

3.1 Data pre-processing

Data cleaning. The source data is gathered from Dryad [2025], containing energy readings with intervals of 15 minutes from 1825 meters between 2013 and 2020. The raw data had multiple issues due to both missing and duplicate values which complicated the pre-processing steps.

The first step was to combine the numbers containing 15 minute readings into numbers meaningful for the entire calendar day. There was a debate in the group on whether we should simply sum the readings or find their average within the calendar day because it was often the case that values were missing (e.g. there were many days where instead of the full 96 readings, one would find only 90). In the end, we have settled for taking the average

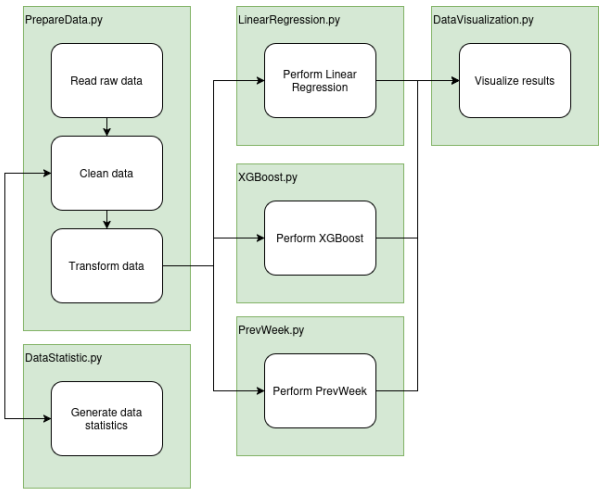


Figure 2: An overview of the workflow in our python program.

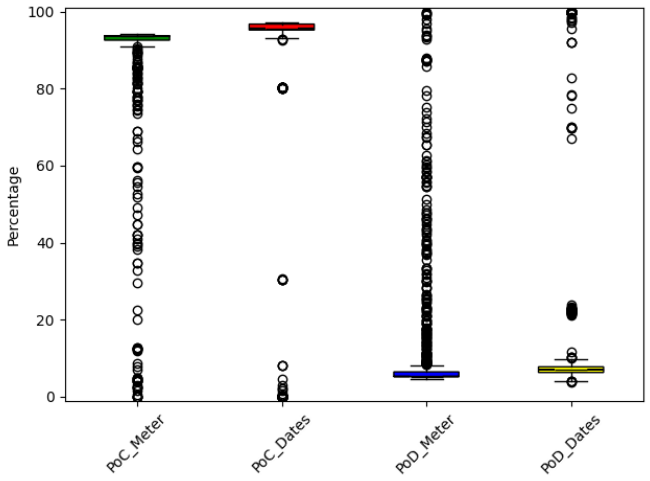
value of all readings in a day. Physically, it would mean that up to time units, we would predict the average daily *power* usage rather than the total daily *energy* consumption. However, this is not a problem because the power average is taken over equal time intervals everywhere (1 day), making the comparison valid and meaningful. Furthermore, it avoids the issue of artificially having days where the energy consumption would be lower than in reality due to unavailable readings. Lastly, it eliminates the subjectivity of imputation methods.

To handle missing and duplicate values, we found it useful to generate relevant statistics and visualize the imperfections through relevant plots. The statistics relate to the quality of the readings per meter, but also per date.

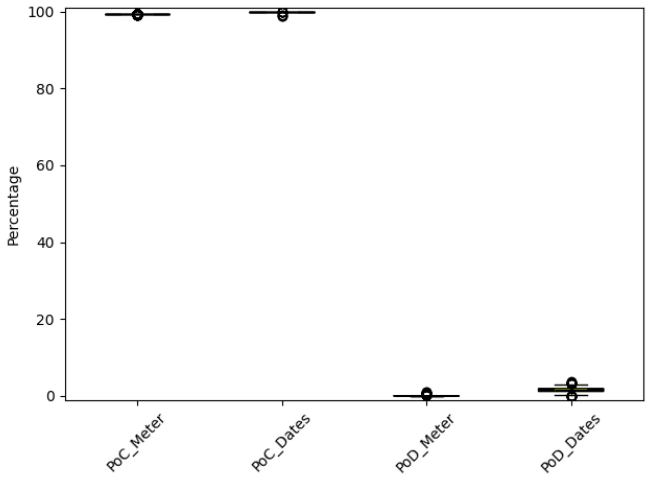
1. *Percentage of completeness* (PoC)
   at any single date we expect 96 readings from a meter since 24 hours divided by 15 minutes equals 96 readings.
   - percentage of completeness dates: the percentage of meters that have 96 ± 5 readings on said date.
   - percentage of completeness meters: the percentage of dates that have exactly 96 readings in said meter.
2. *Percentage of duplicates* (PoD)
   - percentage of duplicates dates: all duplicate readings except the first occurrence are marked, the final percentage is the percentage of marked readings of meters on said date.
   - percentage of duplicates meters: Since within a day there maybe many similar readings, we have chosen to only mark duplicate consecutive readings. The final percentage is the percentage of these marked readings on any date in said meter.

The plots of these statistics can be seen together in Figure 3.

The elimination of faulty dates and meters came in two steps. Firstly, we removed dates that had a PoC under 90% or a PoD of more than 10%. Then we removed the meters that had a PoC of lower than 99% or which had a PoD greater than 1%. These thresholds are essentially arbitrary, but we found them to be a good trade off which allowed us to keep a fair amount of data. We chose to be more lenient with the dates since the bad meters are still present and are making some dates look worse than they are. On top of that, removing a lot of dates would cause too many time skips in our time series. During this step, a total of 174 dates were removed; most of these are consecutive days of what we presume are outages or due to bad connection. They lead to 12 time skips in our time series. From the meters, 259 were removed. The statistics of the cleaned data can be viewed in Figure 3. Here you can see that even though we only removed dates with a completeness under 90%, in the end all dates have a completeness of almost 100%; the removal of bad meters was effective.



(a) Uncleaned data



(b) Cleaned data

Figure 3: Box plots visualizing the spread of completeness and duplication for both dates and meters

The sliding window. Having cleaned the data, we proceeded to transform it into a format appropriate for the task at hand. We have followed the same method as the authors of Kruger et al. [2024].

Because we are concerned with the shape of the time series rather than its exact values (the position of the peak

matters relative to its neighbors), we have applied a z-standardization to the time series of each meter. This also allowed us to mix data from different meters, as now all energy consumptions were on the same scale.

The objective is to convert the time series from all meters spanning 7 years into a format where each instance looks like

$$(\boldsymbol{x}_i,\ \text{max\_pos}_i,\ \text{max\_val}_i)$$

The feature vector $\boldsymbol{x}_i \in \mathbb{R}^7$ consists of consecutive energy values from 7 days; $\text{max\_pos}_i \in \{1, 2, \ldots, 7\}$ is the position of the peak energy consumption in the *following* 7 days, while $\text{max\_val}_i \in \mathbb{R}$ is the value for that peak consumption.

To transform the data into from a time series to the desired format, we have applied two consecutive sliding windows of 7 days each. The first window is used as $\boldsymbol{x}_i$ (i.e. the features), while the second window is used to calculate the peak value and peak position in the next 7 days (i.e. the targets). The length of 7 days has been chosen both because one expects a weekly periodicity in the consumption behavior, but also because the authors of Kruger et al. [2024] have tested other window length intervals and have found that the 7 day window worked the best. While doing this transformation, we took care to avoid the 12 time skips in the 7 year time series.

Depending on which problem one aims to solve, it is possible to split the data sets into training and testing sets in different ways. One objective would have been to take the consumption of each user, and attempt to train a separate model for each user in order to forecast their behavior towards the latter part of their years; this required to use as training data a proportion of the first instances, while leaving the rest for estimating the risk (generalization error) $\rightarrow$ the testing set. However, the problem which we aim to solve is to predict the consumption behavior of an entirely new user, using the patterns of other users available up until the present. This requires to train a single model using the entire time series for some proportion of the entire user base, while leaving the rest to compute the testing error; distinctions between individual users become lost, and all instances are mixed together. Hence, the latter method of splitting the data into training, validation and testing data sets is what we have opted to go for. 20% of the data is used for testing.

3.2 XGBoost hyper-parameters

The XGBoost algorithm offers a wide variety of hyper-parameters that can be tuned to fit a given task. Because we have trained two models, a regressor and a classifier, we have used two sets of values for the hyper-parameters. In this subsection we will go over what hyper-parameters we used and how we have searched for optimal hyper-parameter values.

From our understanding of the algorithm behind XGBoost, we have identified 7 hyper-parameters which are most impactful to our task at hand. Their number is pretty large and their sample spaces are wide, so we have decided to avoid a grid search and instead opt for a random search. Searches were done as follows: for a number of iterations, a value for each hyper-parameter listed bellow would be sampled from an uniform distribution over a given range. The uniform distribution has been chosen because we wished to sample values evenly from an available grid and test all possible candidates; we had no prior information on what would have been good hyper-parameter values, so this was a safe choice. The ranges were informed by the default values in the library, but also by results from previous searching iterations; if we noticed that an optimal hyper-parameter value was close to the upper bound, we would increase the upper bound and start a new search.

The adequacy of the sampled hyper-parameters in an iteration would then be assessed by 3-fold cross-validation, with the metric being the root mean squared error for regression and the classification accuracy for classification. The number of folds for cross validation was this small because of the long time it took to run the search (e.g. around 10 hours for 100 iterations for classification); ideally we would have used a larger number of folds (5, 10 or even leave-one-out-cross-validation). We could have implemented some subsampling method to reduce the training time and increase the number of folds (the large amount of data could have allowed it), but we were unsure if the impact on the quality of the results would justify the increase in the number of folds. Furthermore, there is already a hyper-parameter which controls a subsampling fraction, so the impact on it would be hard to quantify.

The best hyper-parameters found during the searches have been listed bellow. The naming of the hyper-parameters with greek letters is consistent with their meaning in Section 2.

1. **max_depth**: Controls the depth of the trees. The default value is 8. Larger values will create more complex trees, but might lead to overfitting. We note that this is the only hyper-parameter mentioned in the paper written by Kruger et al. [2024] that we used as inspiration for our project, where they used a max_depth of 6. We used a max depth of 7.
2. **min_child_weight**: Controls the minimum sum of instance weight (i.e. hessian values h_i) needed in a child. It ensures that each split in the decision tree has a sufficient amount of data. Larger values will make the algorithm be more conservative and help prevent overfitting. We used a value of 6.4 for the minimum child weight.
3. **eta**: eta is analogous to learning rate; it is the parameter controlling the shrinkage introduced in Section 2.6. The default value is 0.3. Lower values will make the model more robust, but it will need more iterations to converge. We used a value of 0.09 for the regressor and 0.05 for the classifier.
4. **subsample**: Subsampling controls the fraction of training instances to be used for each tree. The default value is 1. A lower value can help prevent overfitting by adding additional randomness to the model; this has been discussed in Section 2.6. We used a value of 0.5 for the regressor and 0.6 for the classifier.
5. **colsample_bytree**: Similar to subsampling, will introduce randomness in the model by subsampling columns when constructing trees; this has been discussed in Section 2.6. Default value is 1. We used a

value of 0.7 for the regressor and 0.9 for the classifier.

6. **lambda**: This parameter controls the L2 regularization. L2 regularization encourages smaller and more evenly distributed weights. The default value is 1. We used values 2.5 for the regressor and 2.3 for the classifier
7. **gamma**: This parameter specifies the minimum loss reduction required to make a split. The bigger the value, the more conservative the algorithm. We used values 1.5 for the regressor and 0.7 for the classifier.

4 Results

In the following subsections, we will go over the results obtained from the two tasks approached in this study: Classification and Regression. The hyper-parameter search for XGBoost for both classification and regression lasted 17 hours in total. As mentioned previously, the baselines are (1) PrevWeek for classification, which assumes that the peak will reoccur in the same day as the input 7 days, and (2) Linear Regression for the regression task.

Is the classification better than random guesses? We can perform a rudimentary form of statistical hypothesis testing to see if the classification accuracies obtained are better or worse than random guesses. There are $N = 612180$ instances in total. If the classification is random, then the success of consecutive classification attempts are independent Bernoulli random variables with probability $p = 1/7$. The accuracy of random guesses is a statistic whose expectation value is $E[\hat{p}] = p \approx 14.2857\%$ and variance is $Var[\hat{p}] = p(1-p)/N = \sigma^2$. Considering that the total number of correct random classifications is Binomially distributed with parameters N, p, by applying the Binomial approximation theorem (or the Central Limit Theorem to the average number of N i.i.d. Bernoulli RV)[2], we obtain:

$$\frac{\hat{p} - p}{\sigma} \sim \mathcal{N}(0,1) \Rightarrow \hat{p} \sim \mathcal{N}(p, \sigma^2)$$

A 3σ confidence interval allows us to know where a classification accuracy based on random guesses would be 99.7% of the times. This interval is $[14.1515\%, 14.4199\%]$. In short: if an accuracy is lower than 14.1515%, then the algorithm is worse than random guessing; if an accuracy is higher than 14.4199%, then the algorithm is better than random guessing. This test should fail only 0.3% of the times.

Is the regressor biased? In answering this question, we will make the reasonable assumption that the data follows a Gaussian distribution. We further make the simplifying assumption that the instances are independent from each other; this is not a perfect assumption, but for each training instance the number of correlated instances is small compared to the total, so overall the conclusions should hold OK. Lastly, we assume the deviations from the true values have the same variance.

Each label y_i is of the form $y_i = y_{i,\text{ true}} + \epsilon_i$, with $\epsilon_i \sim \mathcal{N}(0, \sigma_1^2)$ being i.i.d. noise. The predictor is of a similar form: $\hat{y}_i = \tilde{y}_{i,\text{ true}} + \tilde{\epsilon}_i$, where $\tilde{y}_{i,\text{ true}}$ is hopefully the same as the true value (to be tested) and $\tilde{\epsilon}_i \sim \mathcal{N}(0, \sigma_2^2)$ is i.i.d. noise. We could set $\sigma_1^2 = 0$ for simplicity (i.e. assume that the labels are actually the true values and that there is little randomness to their values introduced by our preprocessing), but it is safer to allow for this wiggle room. In the end, we will see that this assumption is not very consequential.

To test the adequacy of a fit, it is useful to compute the *residues* of the fitted values. Within the above framework, their distribution is

$$r_i = y_i - \hat{y}_i \sim \mathcal{N}(y_{i,\text{ true}} - \tilde{y}_{i,\text{ true}}, \sigma_1^2 + \sigma_2^2)$$

Let $\mu = y_{i,\text{ true}} - \tilde{y}_{i,\text{ true}}$ be the bias of the residues and $\sigma_t^2 = \sigma_1^2 + \sigma_2^2$ be their variance. If the residues are plausibly centered around zero (i.e. μ is zero), then the regressor is unbiased. Among unbiased regressors one could hope for one with small variance, but it is good to know at least whether there is a systematic error (bias) in the predictions. It goes beyond the scope of this report, but it is possible to estimate both the mean and variance of the distribution of residues, then compute the Z statistic with the estimates. The result is *Student's t-statistic*:

$$t = \frac{\bar{r} - \mu}{\hat{\sigma}_t / \sqrt{N}}$$

where $\bar{r}$ is the average of the residues (the estimator for the mean), while $\hat{\sigma}_t$ is the estimator of the variance found via the residual sum of squares. The t-statistic follows the $\mathcal{T}_{N-2}$ distribution with $N-2$ degrees of freedom. Asymptotically, it looks like the $\mathcal{N}(0,1)$ distribution; the amazing thing about it is that it does not depend on the unknown variance, so the bias question can be answered pretty unambiguously!

If μ is zero, then t computed from the residues should belong to a $\mathcal{N}(0,1)$ distribution. If $t \notin [-2, 2]$, then we can be 95% certain that the estimator is biased; if $t \notin [-3, 3]$, we can be 99.7% sure. Otherwise, we can only say that to the best evidence available that the estimator is unbiased; the nature of statistical hypothesis tests is that one cannot accept the null hypothesis, only reject it.

4.1 Classification

Figure 4 depicts the accuracy of the PrevWeek method as a confusion matrix where the rows represent the true labels and the columns represent the predicted labels. Each cell in the matrix contains a number that indicates the count of predictions made for the corresponding combination of true and predicted labels. The diagonal elements of the matrix represent the instances where the predicted labels match the true labels, indicating correct predictions. Conversely, the off-diagonal elements represent the instances where the predicted labels differ from the true labels, indicating misclassifications. Additionally, the cells are color-coded to visually represent the magnitude of the values they contain relative to the other cells.

PrevWeek has achieved an accuracy of 15.44%. The rationale for employing PrevWeek as a baseline is predicated on the assumption that human behavior tends to

[2] the approximation works well here because N is very large and p is far from both 0 and 1

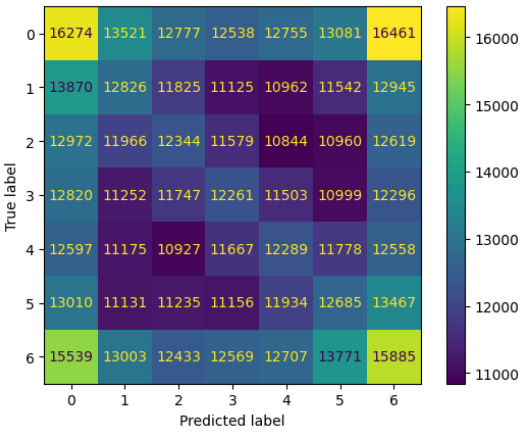


Figure 4: Prev week confusion matrix (accuracy 15.44%)

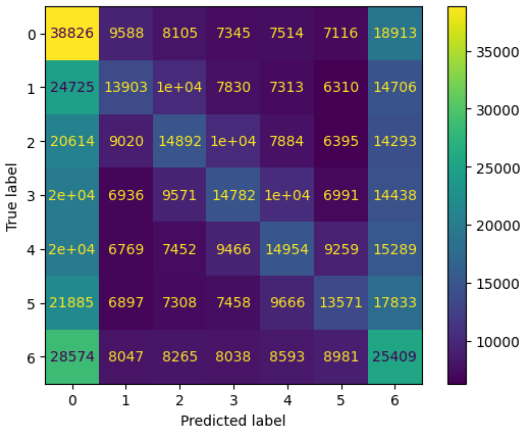


Figure 5: XGBoost Classifier confusion matrix (Accuracy 22.27%)

form routines around the concept of a week, leading to the recurrence of events and actions every seven days. However, as illustrated in Figure 4, this assumption may not hold true, as other patterns appear to emerge from the confusion matrix.

If our initial hypothesis were correct, the diagonal of the matrix would exhibit the highest values. While the diagonal does contain relatively high values, suggesting the presence of some cyclic patterns, the points of interest are the corners. We hypothesize that the elevated values in the corners and edges of the matrix are attributable to the peaks in the data which predominantly occur on either the first or last day of the seven-day window.

Figure 5 presents the results of the XGBoost Classifier, which achieved an accuracy of 22.27%. Compared to PrevWeek, XGBoost demonstrated superior performance. However, a somewhat similar pattern to that observed in Figure 4 can be noted, where the corners and edges exhibit higher values. Nevertheless, XGBoost appears to have learned a different strategy. While there is a slight improvement in the diagonal, the correct classification of the first and last positions of the peak has notably improved. Referring back to our hypothesis mentioned in the preceding paragraph, we posit that XGBoost has learned that the peak is more likely to occur on either the first or last day of the seven-day window. One should also note that both PrevWeek and XGBoost have achieved accuracies better than random guessing. While PrevWeek is barely above the boundary, XGBoost managed to up that number significantly.

Lastly, it is worth mentioning that because we aggregate instances from time series of many users in our training, the finer patters of behavior which are specific to individuals will be lost. If the algorithm is well regularized, then the overfitting will be prevented, but at the price of a bias introduced into the predictions. Hence, it would be impossible to achieve very accurate results without trying to tailor it to each user. The algorithm obtained will make use of patters that are common to the majority of the users; one such pattern could be the increased energy consumption during the weekend, when people are enjoying a leisure time at home.

4.2 Regression

Figure 6 shows the peak values we have used as test data. In the plot, we may notice that the lower bound on these values appears to be equally distant from the "lowest" peak values i.e. the "lowest" peak values stay somewhat constant. On the flip side, the "highest" peak values have a much more rugged behavior, with many jumps from the denser blue area.

This behavior influences the regression's results. Figures 7 and 8 show the histograms of the residuals of linear regression and xgboost respectively. While they both have a mean close to 0 [3], they are both right-skewed. This is a consequence of the behavior we just described regarding the jumps of the "highest" peak values above the dense, blue baseline.

In trying to explain the inadequacy of our algorithm of dealing with this sort of behavior, we have the following explanation. In these regression tasks, we are effectively trying to predict the future given some past data. However, we cannot precisely predict big jumps, as there is little indication on whether the next week will have a peak value whose "highest" value is close to the dense blue "baseline", or the next week will have a "highest" value among the jumps. We believe that our regression algorithms tend to be conservative in their estimations and not deal well with such systematic outliers , therefore we often forecast smaller values than the true ones. To make this concept more clear we provide an example. Assume that, within a certain week, the values have been relatively constant for 6 days and then then there is a large jump. Both models will not be able to predict the true value of the large peak as they will be also trying to fit the smaller values. Thus they will forecast a smaller value and hence the right skewness.

There is also another possible influence which can explain the conservative tendency of our regression method. We believe that the regressor does not learn to predict the maximum of the future very well, but actually it learns to predict the maximum peak value *in the present week*. This is justified by the fact that maximum values tend to

[3] The mean of linear regression is -0.00078 while the one of xgboost is 0.0017

not have a lot of wiggle room; whether the peak value is due to the next week or due to the current week, their maxima could be very close. To illustrate this point, consider a noisy sinusoidal energy consumption pattern with the period of one week. The maxima of the sinusoids will not be far from each other (all noise considered), so learning to predict the maximum for next week could be just as well achieved as learning the maximum for the current week. A factor working against this idea is that we have seen that XGBoost manages to do predictions significantly better than PrevWeek; if the behavior would be as described above, then PrevWeek should be as good as it could get.

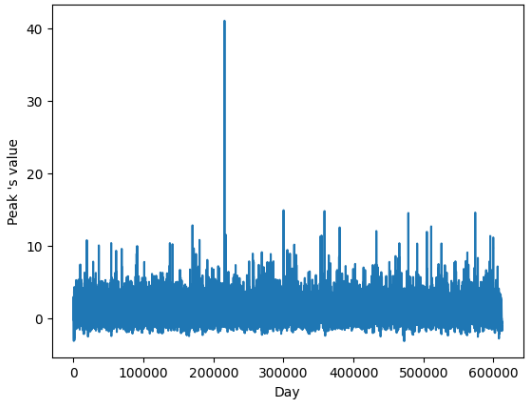


Figure 6: Peak's values (normalized) of the test data. We attribute the outlier to a faulty meter reading

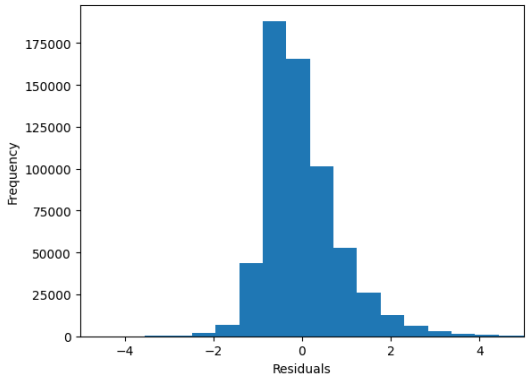


Figure 7: Linear regression residuals (mean = -0.00078, RMSE = 0.9034)

Interestingly, as a consequence of what we just explained, the two residual plots are not only both right-skewed, they also have the same shape in the other parts of the histogram. Hence, it is not surprising that they have the same root mean squared error (RMSE).

Lastly, let us investigate whether the predictions are biased. The distribution of the residues is loosely Gaussian and the means seems to be close to zero, but there are also *many* points in the histograms. Thus, it could be that the 3σ confidence interval around the mean does not contain 0 in it, even if the bias is small. Thankfully, the t-statistic is suitable to answer exactly this question! The results are:

$$t_{\text{XGBoost}} \approx 1.57 \qquad t_{\text{LinRegress}} \approx -0.68$$

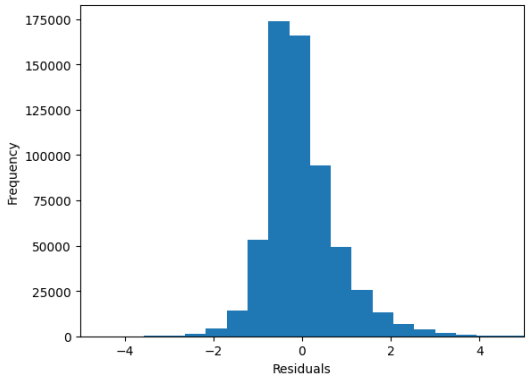


Figure 8: Xboost regression residuals (mean = 0.0017, RMSE = 0.8741)

Thus, up to a 2σ confidence interval (significance level 5%), we cannot reject the null hypothesis that the estimator is unbiased. The p-values are 0.116 and 0.495 respectively, so the judgment was not even very close. Thus, for all we know, the regressors are unbiased or with a very small bias.

5 Discussion

In the following section, we will cover some of our ideas and thoughts about the project.

5.1 Miscellaneous factors affecting the results quality

As mentioned in the previous sections, we believe that the nature of how we have asked the problem may have placed an upper bound on how good our results could become. Individual users have fairly specific patterns of behavior, whether some houses are for vacation and sit empty for much of the year, or whether other users may have energy intensive hobbies during the weekend. The original paper's main focus Kruger et al. [2024] was in forecasting the consumption behavior for each individual user in particular; the authors' attempt at generating a single model for everyone (like we have done) obtained results a bit above a baseline, but not even close to the performance achieved for individual user forecasting. Our algorithm may manage to learn only from very general patters of consumption from the entire population, averaging out any individual particularities. Overall, we are happy that XGBoost performed better than random guessing or PrevWeek significantly when classifying and was very close to a known effective method when doing regression. However, we would also like to admit that seen in absolute terms, an accuracy of 22.27% is not an encouraging number.

A future attempt at this task may yield more fruitful results if one can summarize the weekly data somehow, doing a manual form of data compression. For instance, we were thinking that the features could be transformed into: week average, week peak value, week peak position, week variance. However, we chose not to go for this version in a first iteration (1) because we wanted to keep the project

simple and (2) complicated, potentially useful patterns in the weekly consumption could have been lost.

Lastly, it is unknown whether the weak co-dependence between instances could have improved the results quality. For instance, the consumption throughout the year is not even; summer times require more energy usage due to air conditioning than the winter season; peak values during summer may be overall higher. A 7 day window can lose this seasonal information. A more refined approach could somehow find a way to effectively exploit the co-dependence between instances.

5.2 Comparison with the original study

As mentioned in the introduction, our project is heavily inspired by a study made by Kruger et al. [2024]. One of the main take-home messages of that study is that, for peak prediction tasks on time series data, simpler methods, such as linear regression and gradient boosted decision trees, attained better performance compared to more complex machine learning solutions, such as LSTM and biLSTM. This, combined with a newfound curiosity for XGBoost, has led as to pursue this project with the intention to not only replicate the results from the original paper, but also gain more knowledge on XGBoost.

Our results do not entirely match the results of the original paper. For the regression task of predicting the value of the peak, we used a different metric to compute the error. The original paper used mean absolute error (MAE) while we used the root mean squared error (RMSE), so an absolute comparison between results is difficult. We could only assess the performance of our algorithm relative to the baseline, and see how this performance gap also looked in the original paper. In the task at hand, the performance gap was well comparable.

Conversely, our findings on the classification task diverge significantly from those presented in the original study. The accuracy achieved in our analysis is markedly lower than that reported in the original paper. This discrepancy may be attributed to differences in data preprocessing methodologies. Notably, our study incorporated a larger dataset compared to the original research. Lastly, considering the large number of hyper-parameters, a 3-fold cross validation with only 100 iterations could be insufficient to get a clear picture of what is an optimal set of hyper-parameters. We could have stumbled on a local optimum, not a global one.

5.3 Lessons learned

We feel like this project has thought us a lot regarding the pipeline of a machine learning project.

Starting with data pre-processing, we meticulously examined and scrutinized real-world data, identifying the inherent imperfections and challenges associated with such datasets. Collaborative efforts were essential in determining the optimal strategies for addressing issues such as missing data and anomalous entries. Subsequently, we conducted an in-depth investigation of our selected algorithm, gaining a comprehensive understanding of its functionality, available hyper-parameters, and their respective impacts.

The main star of the show is XGBoost. We meticulously read and dissected the original paper [Chen and Guestrin, 2016] covering this algorithm as well as studied plenty online documentation regarding its implementation. From studying XGBoost not only did we understand how it works, but we also gained valuable insight in the primary concepts that make it tick such as tree boosting and regularization. With the knowledge we gained, we can confidently say that this algorithm has been added to our machine learning tool-belt.

We have also learned that results are not always "successful". We have found avenues for improvement within our own method, but also possible limitations in the task itself that limit the best results attainable. It was important to not look at absolute numbers outside a context, but compare our findings with sensible baselines and look at the *statistical* properties of our numbers. Our results were better than random guessing, better than a simple baseline and (as far as we can tell) unbiased, but on the flip side the authors of Kruger et al. [2024] showed how good the results could have been.

In conclusion, we are reasonably happy with the outcome. We have developed a sensible pipeline for the data, obtained OK results and learned how to use a particular ML algorithm by looking into the black box.

References

T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, Aug. 2016. doi: 10.1145/2939672.2939785. URL http://arxiv.org/abs/1603.02754. arXiv:1603.02754 [cs].

Dryad. Dataset associated with the publication, 2025. URL https://datadryad.org/stash/dataset/doi:10.5061/dryad.m0cfxpp2c. Accessed: 2025-01-14.

L. Goodarzi, M. E. Banihabib, and A. Roozbahani. A decision-making model for flood warning system based on ensemble forecasts. *Journal of Hydrology*, 573:207–219, June 2019. ISSN 00221694. doi: 10.1016/j.jhydrol.2019.03.040. URL https://linkinghub.elsevier.com/retrieve/pii/S0022169419302331.

R. Kruger, A. Mueen, and V. M. A. Souza. Peak Prediction in Time Series: Comparing Approaches for Energy High-Load Prediction. In *2024 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Yokohama, Japan, June 2024. IEEE. ISBN 9798350359312. doi: 10.1109/IJCNN60899.2024.10651140. URL https://ieeexplore.ieee.org/document/10651140/.

P. Li. Robust LogitBoost and Adaptive Base Class (ABC) LogitBoost. Mar. 2012.