Gradient Boosting Trees

Stefanos Fafalios, Pavlos Charonyktakis, Ioannis Tsamardinos

Gnosis Data Analysis PC

April 2020

1 Introduction

Gradient boosting is a machine learning technique for **regression** and **classification** problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function. The idea of gradient boosting originated in the observation by Leo Breiman that boosting can be interpreted as an optimization algorithm on a suitable cost function[1]. Explicit regression gradient boosting algorithms were subsequently developed by Jerome H. Friedman[6, 7], simultaneously came up with the more general functional gradient boosting perspective of Llew Mason, Jonathan Baxter, Peter Bartlett and Marcus Frean[9]. The latter paper introduced the view of boosting algorithms as iterative functional gradient descent algorithms. That is, algorithms that optimize a cost function over function space by iteratively choosing a function (weak hypothesis) that points in the negative gradient direction. This functional gradient view of boosting has led to the development of boosting algorithms in many areas of machine learning and statistics beyond regression and classification.

XGBoost(eXtreme Gradient Boosting)[3] is an open-source software library which provides the state-of-the-art gradient boosting framework packaged in many languages, such as C++, Python and Java. It aims to provide a "Scalable, Portable and Distributed Gradient Boosting Library". It runs on a single machine, as well as the distributed processing frameworks Apache Hadoop, Apache Spark, and Apache Flink. This has been considered by many winning teams of machine learning competitions as the algorithm of choice which gave it a lot of popularity in recent years.

The scope of this manuscript is to set a background for the gradient boosting algorithms, explain the theoretical subset of the XGBoost framework that has been implemented as a part of the **JADBIO** framework and compare JADBIO's implementation to the original. The **JADBIO** version of gradient boosting, implements a convenient subset of the functionality of the XGBoost framework, and has also been extended to account for the case of survival optimization as well.

2 Gradient Boosting Main Ideas

Gradient tree boosting algorithms are reviewed in this section. The derivation follows from the same idea in existing literatures in gradient boosting. Specifically the second order method originated

from Friedman et al.[5]. **XGBoost** makes minor improvements in the regularized objective, which were found helpful in practice.

2.1 Regularized Learning Objective

For a given data set with n examples and m features $D = (x_i, y_i)$, $(|D| = n, x_i \in \mathbb{R}^m, yi \in \mathbb{R})$, a tree ensemble model (shown in Fig. 1) uses K additive functions to predict the output.

$$\hat{y_i} = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), f_k \in F \tag{1}$$

where $F = \{f(x) = w_{q(x)}\}(q : \mathbb{R}^m \to T, w \in \mathbb{R}^T)$ is the space of regression trees (also known as CART). Here q represents the structure of each tree that maps an example to the corresponding leaf index. T is the number of leaves in the tree. Each f_k corresponds to an independent tree structure q and leaf weights w. Unlike decision trees, each regression tree contains a continuous score on each of the leafs, we use w_i to represent score on i-th leaf. For a given example, the decision rules in the trees (given by q) are used to classify it into the leaves and calculate the final prediction by summing up the score in the corresponding leaves (given by w). To learn the set of functions used in the model, we minimize the following regularized objective.

$$\begin{aligned} L(\phi) &= \sum_i l(\hat{y_i}, y_i) + \sum_k \Omega(f_k) \\ where \quad \Omega(f) &= \gamma T + \frac{1}{2}\lambda \parallel w \parallel^2 \end{aligned} \tag{2}$$

Here l is a differentiable convex loss function that measures the difference between the prediction $\hat{y_i}$ and the target y_i. The second term Ω penalizes the complexity of the model (i.e., the regression tree functions). The additional regularization term helps to smooth the final learnt weights to avoid over-fitting. Intuitively, the regularized objective will tend to select a model employing simple and predictive functions.

2.2 Gradient Tree Boosting

The tree ensemble model in Eq. (2) includes functions as parameters and cannot be optimized using traditional optimization methods in Euclidean space. Instead, the model is trained in an additive manner. Formally, let $\hat{y_i}^{(t)}$ be the prediction of the i-th instance at the t-th iteration, f_t needs to be added to minimize the following objective.

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

This means that the f_t that most improves the model according to Eq. (2) is greedily added. Second-order approximation is used to quickly optimize the objective in the general setting[5].

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

where $g_i = \partial_{\hat{y}(t-1)} l(y_i, \hat{y_i}^{(t-1)})$ and $h_i = \partial^2_{\hat{y}(t-1)} l(y_i, \hat{y_i}^{(t-1)})$ are first and second order gradient statistics on the loss function. Then by removing the constant terms the following simplified objective at step t is obtained.

$$\tilde{L}^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \tag{3}$$

Define $I_j = \{i|q(x_i) = j\}$ as the instance of leaf j. We can rewrite Eq. (3) by expanding Ω as

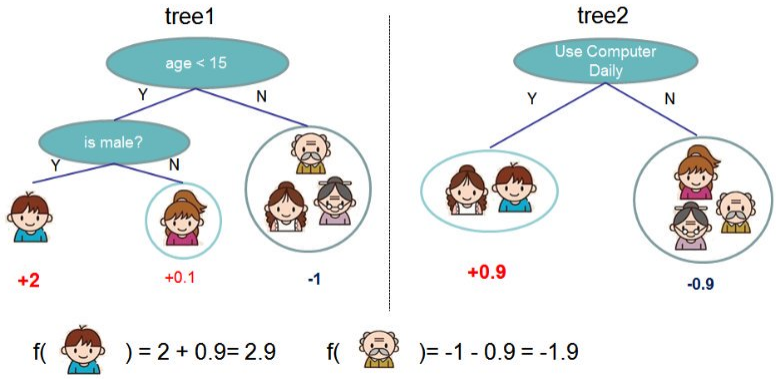

Figure 1: Tree Ensemble Model. The final prediction for a given example is the sum of predictions from each tree.

follows

$$\begin{aligned} \tilde{L}^{(t)} &= \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \end{aligned} \tag{4}$$

For a fixed structure $q(x)$, we can compute the optimal weight w_j^* of leaf j by

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{5}$$

and calculate the corresponding optimal value by

$$\tilde{L}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \tag{6}$$

Eq 6 can be used as a scoring function to measure the quality of a tree structure q. This score is like the impurity score for evaluating decision trees, except that it is derived for a wider range of objective functions.

Normally it is impossible to enumerate all the possible tree structures q. A greedy algorithm that starts from a single leaf and iteratively adds branches to the tree is used instead. Assume that I_L and I_R are the instance sets of left and right nodes after a split. Letting $I = I_L \cup I_R$, then the loss reduction after the split is given by

$$L_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{7}$$

This formula is used in practice for evaluating the possible split candidates.

2.3 Shrinkage and Column subsampling

Besides the regularized objective mentioned in Sec. 2.1, two additional techniques are used to further prevent overfitting. The first technique is shrinkage introduced by Friedman [7]. Shrinkage scales newly added weights by a factor η after each step of tree boosting. Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model. The second technique is column (feature) subsampling. This technique is used in RandomForest [2, 8]. As stated in Chen et al., [3], according to user feedback, using column sub-sampling prevents over-fitting even more so than the traditional row sub-sampling (which is also implemented).

2.4 Split Finding Algorithm

Algorithm 1 Exact Greedy Algorithm for Split Finding

Input: I, instance set of 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 $\mathbf{x}_{jk}$) **do**
    $G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$
    $gain \leftarrow \max(gain, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
Output: Split with max score

One of the key problems in tree learning is to find the best split as indicated by Eq (7). In order to do so, a split finding algorithm enumerates over all the possible splits on all the features. We call this the exact greedy algorithm. Most existing single machine tree boosting implementations, support the exact greedy algorithm. The exact greedy algorithm is shown in Alg. (1). It is computationally demanding to enumerate all the possible splits for continuous features. In order to do so efficiently, the algorithm must first sort the data according to feature values and visit the data in sorted order to accumulate the gradient statistics for the structure score in Eq (7). The XGBoost framework [3] has made various modifications to the Exact Greedy split finding algorithm in order to treat more effectively large and/or sparse datasets. These adjustments were not developed in the this implementation, since they are out of the scope of the current work.

3 Hyper-parameters and loss functions

In this section we will go through the implemented the implemented functionality of the JADBIO version of gradient boosting, the hyper-parameters, and the type of outcomes that are supported.

3.1 Supported hyper-parameters

- **learningrate**, η. The prediction of each tree is scaled by this factor, so that there is room left for improvement by additional trees.
- **alpha**, α. Weight on the ℓ_1 regularization term.
- **lambda**, λ. Weight on the ℓ_2 regularization term.
- **gamma**, γ. Pruning hyper-parameter described in sec. (2.1).
- **max depth**. The maximum height that each tree is allowed to reach.
- **min child weight**. A tree node will not be split into nodes with $minchildweight \geq \sum_{i \in I_{child}} h_i$. Authors of XGBoost [3] also refer to $\sum_{i \in I_{child}} h_i$ as "cover".
- **max delta step**. Maximum delta step each leaf output is allowed to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative. Usually this parameter is not needed, but it could be found useful in cases with class imbalance.
- **subsample**. Percentage of random samples to be considered during a single tree construction.
- **sampling method**. The method that samples are selected for a tree construction. $0 \leftarrow$ with replacement, $1 \leftarrow$ without replacement, $2 \leftarrow$ gradient based. In the gradient based method, the selection probability for each training instance is proportional to the regularized absolute value of gradients (more specifically $\sqrt{g^2 + \lambda h^2}$). The authors of XGBoost allege that using this method subsample may be set to as low as 0.1 without loss of model accuracy[4].
- **colsample by model**. The percentage of random features, or variables to be considered during the construction of each tree.
- **max models**. The maximum amount of trees to be created.
- **earlystopping**. If set to -1, the training process will stop adding trees when **max models** number of trees has been built. If set to 0, the training process will stop adding trees when the loss estimated on the **out of bag** samples becomes too small, or **max models** has been reached. If set to a value $\mathbf{n} > 0$ the training process will stop adding trees if the addition of the last n consecutive trees has failed to improve the loss on the outbag samples. When training is finished, the final model will contain all trees from the first one, until the last tree that improved the loss of the model.

By using the information of the "min child weight" hyper-parameter, set by the user, we can cleverly modify the split finding algorithm to only test possible splits which will result in "valid" children (i.e. $cover_{child}$ min child weight) Alg. (2). [1]

[1] Alg. (2) takes in consideration that the hessian of each instance can not be negative, and uses this information to stop testing possible splits, when it is ensured that no further split will produce "valid" children.

Algorithm 2 Modified Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: m, feature dimention
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ **to** m **do**
  $\hat{x_k} \leftarrow \text{sorted}(I, \text{by } \mathbf{x}_{jk})$
  $mid_j \leftarrow \frac{len(\hat{x_k})}{2}$
  $G_L \leftarrow \sum_{j=0}^{mid_j} g_j, H_L \leftarrow \sum_{j=0}^{mid_j} h_j$
  $G_{L0} \leftarrow G_L, H_{L0} \leftarrow H_L$

  for $j = mid_j, j < len(\hat{x_k}), j = j + 1$ **do**
    $G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

    if $H_L <$ min child weight **then** continue
    if $H_R <$ min child weight **then** break
    $gain \leftarrow \max(gain, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$

  $G_L \leftarrow G_{L0}, H_L \leftarrow H_{L0}$

  for $j = mid_j, j > 0, j = j - 1$ **do**
    $G_L \leftarrow G_L - g_j, H_L \leftarrow H_L - h_j$
    $G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

    if $H_R <$ min child weight **then** continue
    if $H_L <$ min child weight **then** break
    $gain \leftarrow \max(gain, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$

Output: Split with max gain

3.2 Minimizing loss functions specific to different optimization problems

Each type of desired outcome, typically requires a different modeling of the loss function. Here we will go through the equations of the loss functions and the first and second order derivatives that are used of each tree built, for the case of continuous, discrete and survival outcomes[2].

3.2.1 Continuous outcomes A.K.A. Regression

For the case of continuous outcomes the Mean Square Error (**MSE**) loss function, is being used, and the first and second order derivatives are shaped as follows.

$$\begin{aligned} L_{MSE} &= \frac{1}{n}\sum_i MSE(out_i, y_i) \\ &= \frac{1}{n}\sum_i \frac{1}{2}(y_i - out_i)^2 \end{aligned} \tag{8}$$

$$\begin{aligned} g_i &= \frac{\partial L_{MSE}}{\partial out_i} = out_i - y_i \\ h_i &= \frac{\partial^2 L_{MSE}}{\partial out_i^2} = 1 \end{aligned}$$

where n is the number of instances, y_i is the ground truth outcome value for instance i, and out_i is the outcome of the boosting algorithm on the last iteration for instance i. Notice that minimizing the mean of the square of the errors, is equal to minimizing the sum of the square of the errors. As a result, the term $\frac{1}{n}$ does not have to be included in the derivative estimation.

3.2.2 Categorical outcomes A.K.A. Classification

For the case of classification with C number of possible categories, the output of the boosting algorithm would be the predicted probability of each category c, ($c = \{1, ..., C\}$). The loss function that has been implemented to treat this kind of outcome is the multinomial log-loss, with the last class being treated as the "pivot" class. The optimization procedure is set in such a way that the following property is enforced:

$$\sum_{c \in C} prob_{i,c} = 1$$

Where $prob_{i,c}$ is the predicted probability that instance i belongs to class c. Meaning that the sum of the output probability of each possible class for a given instance i always amounts to 1.
Hence,

$$prob_{i,C} = \frac{1}{1 + \sum_{c=1}^{C-1} e^{(-out_{i,c})}}$$

$$prob_{i,c \neq C} = prob_{i,C} \times e^{(-out_{i,c})} = \frac{e^{(-out_{i,c})}}{1 + \sum_{c=1}^{C-1} e^{(-out_{i,c})}}$$

[2]The regularization term Ω is not dependent to the loss function, and as a result for simplicity, it has been omitted from the rest of the equations

Using this formulation, the number of first and second order derivatives that are needed to be estimated for each sample is $C-1$. The loss function and the derivatives are formed as follows

$$\begin{aligned} L_{logloss} &= -\sum_{i}\sum_{c\in C} y_{i,c}\log(prob_{i,c}) \\ g_{i,c} &= 1 - prob_{i,c} \iff y_{i,c} = 1 \\ g_{i,c} &= -prob_{i,c} \iff y_{i,c} = 0 \end{aligned} \tag{9}$$

$$h_{i,c} = (1 - prob_{i,c}) \times prob_{i,c}$$

where $y_{i,c} = 1$, if class 'c' is correct for sample i and $y_{i,c} = 0$, otherwise.

In the multiclass setting, for each instance i, there exist multiple gradients $(g_{i,c}, h_{i,c}, c \in [1, C])$. This can be accounted for inside the Split Finding Algorithm, and the output would be: Split with $\max\{\text{average}(gain_c)\}$, which would mean that average(G_L, H_L) would have to be estimated in each iteration of the split finding algorithm. We implemented a different approach, which results in more greedy tree building in the multiclass setting, but essentially makes the complexity of the tree building algorithm to not be dependent to the number of possible classes c.
In our approach, each tree, will take as input $\hat{g_i}, \hat{h_i}$ instead of $g_{i,c}, h_{i,c}$, where $\hat{g_i} = max_j(||g_{i,j}||)$ and $\hat{h_i} = h_{i,j}$. Using this modification, there would be no need to account for anything else inside the split finding algorithm. Another approach that was implemented to accommodate the multiclass setting is the one-vs-all approach where on each iteration of the gradient boosting algorithm, C number of binary trees are being built, and each one is being treated as a binary tree. Any of these approaches can be used to tackle a multiclass problem.

3.2.3 Survival outcomes

Survival optimization comes with some subtypes. In our case we implemented an algorithm that can handle right censored data, meaning that the event does not necessarily occur in all the available instances. This kind of outcome comes with two values; $event_i$ ([1,0], event was observed on instance i or not), and time that the event occurred (if $event_i = 1$) or time that the instance left the study (if $event_i = 0$).

Let I_E be the instances were the event has occurred, with m the number of event instances, and I_{NE} the instances were the event did not occur such that $I_E \cup I_{NE} = I$, and m the number of events.
A loss function for such outcomes is typically formulated by the negative log-likelihood, given by:

$$L_{loglik} = \sum_{i\in I}(\ln(\sum_{j\in I_E, time_i \le time_j} e^{out_j}) - out_i)$$

The is also a type of residuals which is used for evaluating "goodness-of-fit" survival models called martingale residuals. Given by

$$MR_i = event_i - e^{out_i} \times \ell_i(1)$$

where

$$\ell_i(p) = \sum_{j \in I_E, time_j > time_i} \frac{1}{\sum_{k=1}^{m-j} \left(e^{out_{I_{E_k}}}\right)^p}$$

We implemented both the minimization of the loglikelihood and the minimization of the square martingale residuals, but we decided to stick with the latter, because in the former the second order gradients for censored instances are not defined.
Let $rc_i(p) = (e^{out_i^p}) \times \ell_i(p)$

$$\begin{aligned} L_{SMR} &= \frac{1}{2}\sum_i (MR_i)^2 = \frac{1}{2}\sum_i (event_i - rc_i(1))^2 \\ g_i &= \begin{cases} MR_i \times (rc_i(2) - rc_i(1)), & \text{if } event_i = 1 \\ MR_i \times (-rc_i(1)), & \text{if } event_i = 0 \end{cases} \\ h_i &= \begin{cases} MR_i \times (3rc_i(2) - 2rc_i(3) - rc_i(1)) + (rc_i(2) - rc_i(1))^2, & \text{if } event_i = 1 \\ (rc_i(1))^2 - rc_i(1) \times MR_i, & \text{if } event_i = 0 \end{cases} \end{aligned} \tag{10}$$

Notice that ℓ_i contains $e^{out_j}, j \in I_E$. In order to be more exact, the partial derivatives with respect to out_j also have to be estimated and added to the derivative computation.

$$\begin{aligned} g_j &= g_j + e^{out_i} rc_j(1) \times MR_i \\ h_j &= h_j + (e^{out_i} \times e^{out_j} \times \ell_j(2))^2 - e^{out_i} \times e^{out_j}(2e^{out_j}\ell_j(3) - \ell_j(2)) \times MR_i \end{aligned} \tag{11}$$

4 Comparison of JADBIO implementation to XGBoost implementation

To ensure that the JADBIO implementation of gradient boosting produces valid results in as little time as possible, we compared the two implementations in terms of differences in times and differences in losses, on various datasets and outcome types[3].

4.1 Datasets tested

Table (1) shows a description of the datasets that were used. The outcomes of wine, banknote and diabetes, were both threated as continuous and as categorical.

4.2 Monte-Carlo simulations

Both implementations were trained in a random 50% of the instances of each dataset and the losses were estimated on the rest of the instances. This process was repeated for various values on the number of models (hyperparameter), 100 times. The running times contain both the training and the classification phase of each implementation. For the classification, the average accuracy loss was used (1-accuracy), for the regression, we used the Root Mean Square error and for the survival outcomes the 1− Concordance Index. In each plot we report the differences between our implementation and the XGBoost implementation (JBoosting minus XGBoost), meaning that when

[3]For the multiclass analyses we used the one-vs-all approach

Dataset name	Number of features	Number of samples	Outcome type
diabetes	8	768	binary (0,1)
banknote	4	1372	binary (0,1)
wine	11	800	ordinal (wine quality), treated as: multiclass (0-5) or continuous
oilspill	48	936	binary (0,1)
fri_c4_500_100	100	500	binary (0,1)
lsvt	310	126	binary (0,1)
semeion (sparse)	256	319	binary (0,1)
tecator	124	240	binary (0,1)
iris	4	150	multiclass (0,1,2)
ecoli	7	327	multiclass (0-5)
thyroidAllhypo	26	2800	multiclass (0-5)
GCM	16064	190	multiclass (0-13)
cpu_small	12	8192	continuous
lungcancer_shedden	22	442	survival
melanoma	5	206	survival
veteran	9	137	survival
PBC	17	418	survival
Pharynx	9	195	survival

Table 1: Description of datasets that where analyzed.

we are in the negatives, our implementation's loss or running time are better. For each outcome type we chose a set of hyperparameters where the xgboost implementation would generally perform well. These hyperparameters are shown by table (3).

Table 2: Shared hyperparameters

Hyperparams	learning rate	sampling method	col sample by model	Stopping Criteria
All	0.01	uniform probabilities	sqrt(# features)/(# features)	no early stopping

Table 3: Output specific hyperparameters

Hyperparams	alpha	lambda	gamma	max delta step	max depth	min child weight	subsample
Regression	0.05	0.5	0.01	9	6	2	0.5
Classification	0.25	0.75	0.5	7	8	0.5	0.35
Survival	0.15	0.4	0.255	8	7	1.25	0.425

The Monte-Carlo simulations show that while there are cases were the two implementations are almost exactly equal in terms of performance, there also exist cases were they somewhat differ. For instance, in the regression analysis, with the exception of the cpu_small dataset, the two algorithms' performance is very similar, both in terms of running times and in terms of loss, while in the cpu_small dataset, the JADBIO implementation performs substantially better, but it takes more time to train. In the classification analysis, both algorithms have similar performance in

Regression

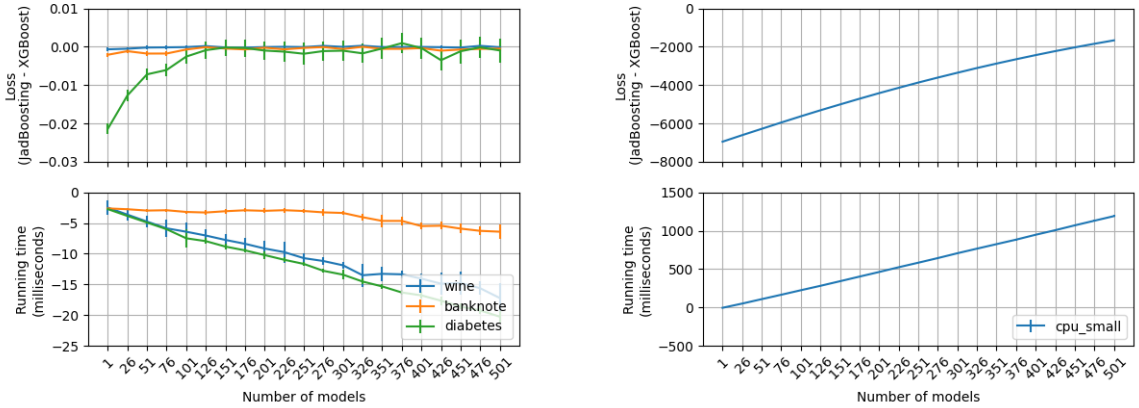

Figure 2: Differences between the JADBIO implementation and the XGBoost implementation, in the regression analysis.

Classification

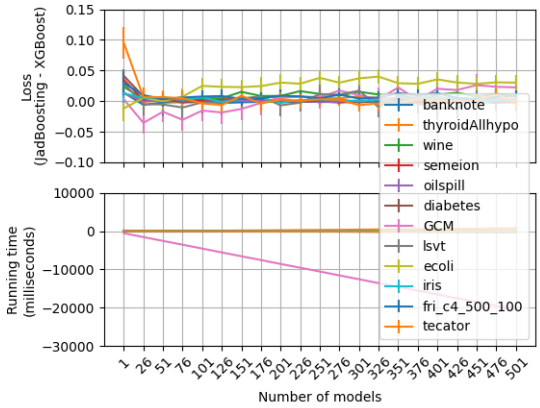

Figure 3: Differences between the JADBIO implementation and the XGBoost implementation, in the classification analysis

most datasets, with some exceptions. The implementations differ in terms of accuracy mostly around 0.05, or less. While these differences may seem insignificant, they possibly point towards possible underlying discrepancies in the implementations of the multiclass outcome type. Showing that the two are slightly different approaches on the same algorithm. Finally, for the case of the survival analysis, the JADBIO implementation is always enjoying a better performance on each hyperparameter combination tested and on every dataset.

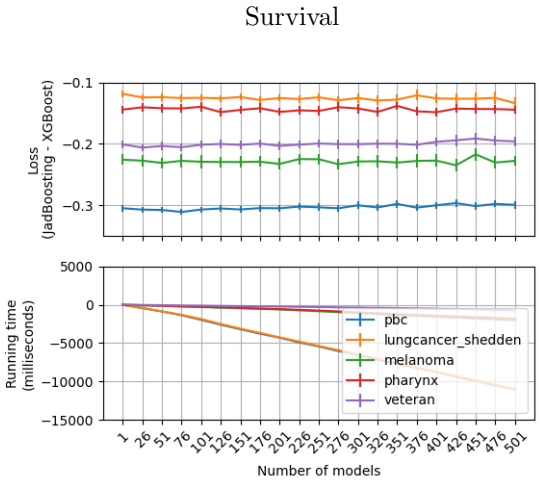

Figure 4: Differences between the JADBIO implementation and the XGBoost implementation, in the survival analysis

5 Discussion

In the current manuscript we went thought the main ideas of the gradient boosting algorithm, showed the theoretical background of XGBoost's tree implementation and applied it theoretically for the cases of Continuous, Categorical and Survival outcomes. We also showed how the Exact Greedy Split Finding Algorithm could be modified so that the optimum split is found in a more efficient manner. Finally, we compared our implementation of the gradient boosting algorithm to XGBoost's implementation, on real data.

As it can be observed from our results, there are cases were the JadBoosting implementation outperforms the XGBoost implementation, while there are also cases were the XGBoost implementation is better. On the regression optimization JAD's gradient boosting implementation is more computationally efficient with the exception of the cpu_small dataset. In the classification comparisons, the accuracy loss differences in loss between the JADBIO implementation and the XGBoost implementation are relatively small, rarely exceeding a 0.025 difference in accuracy loss. There are two exceptions thought, the main one being the ecoli dataset. Finally, on the survival setting, the computational times on both algorithms is very similar, with their computational times differing at most by approximately 25 milliseconds. However, JAD's approach on the survival problem always provides results of superiorly better quality.

Acknowledgements

This research has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH–CREATE–INNOVATE (project code:T1EDK-00905).

References

[1] Leo Breiman. Arcing the edge. Technical report, Technical Report 486, Statistics Department, University of California, 1997.

[2] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[3] Tianqi Chen and Carlos 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, 2016.

[4] XGBoost developers. Xgboost parameters, 2020.

[5] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[6] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[7] Jerome H Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002.

[8] Jerome H. Friedman and Bogdan E. Popescu. Importance sampled learning ensembles, 2003.

[9] Llew Mason, Jonathan Baxter, Peter L Bartlett, and Marcus R Frean. Boosting algorithms as gradient descent. In *Advances in neural information processing systems*, pages 512–518, 2000.