Assamblers and Multiclass Predictor: XGBoost

Joseph Jesus Aguilar Rodriguez, *1809003*, Brenda Estefania Castillo Fernandez, *2009028*,
Jesus Gabriel Canul Caamal, *1909031*, Marco Alejandro González Barbudo, *1909073*,
Yuliana Alejandra Molina Cortés, *2009096*, Hernando Enrique Te Bencomo, *2009132*,
Joshua Zamora Ramirez, *1909191*

***Abstract*—This research examines XGBoost, a leading gradient boosting framework that enhances predictive accuracy with a novel tree learning algorithm and regularization techniques. It offers an architectural overview, a unique approach for managing sparse data, and comparative analyses against other algorithms. The efficacy of XGBoost is validated through empirical evidence and an illustrative implementation in code.**

***Index Terms*—Artificial Intelligence, Machine Learning, Learning algorithms, Supervised learning, Gradient methods, Decision trees, Feature extraction, Data preprocessing.**

I. Introduction

The demand for algorithms capable of efficiently processing large-scale data and delivering high predictive accuracy is addressed by the advent of XGBoost, an advanced gradient boosting machine (GBM). Known for its exceptional performance in machine learning competitions and its utility in practical applications, XGBoost stands as a preferred algorithm for data practitioners.

Originating from decision tree ensembles, XGBoost utilizes a gradient descent algorithm to minimize loss iteratively, distinguishing itself from traditional GBMs. This exploration commences with a review of the evolution of GBMs and the specific problems they aim to solve. It is shown how XGBoost enhances these foundational techniques with algorithmic improvements that increase efficiency and processing speed.

Attention is given to the system design of XGBoost and its sophisticated approach to sparse data management through a sparsity-aware algorithm for split finding, enhancing its capability to address various missing data scenarios. The algorithm's adeptness at performing both classification and regression tasks with high accuracy is also noted.

The paper includes a discussion on the regularization techniques implemented by XGBoost, which are instrumental in preventing overfitting. This aspect is deemed essential for data with high dimensionality, where the risk of overfitting is significant.

An example in code is provided to illustrate the application of XGBoost, aiming to clarify the algorithm's operation and offer a practical reference for implementation. This is intended to highlight the nuances of parameter adjustment and the practical aspects involved in developing a high-performing model.

II. XGBoost

The full name of XGBoost is eXtreme Gradient Boosting, proposed by Dr. Tianqi Chen who worked in the University of Washington in 2014. XGBoost is a tree integration model, which uses the cumulative sum of the predicted values of a sample in each tree as the prediction of the sample in the XGBoost system. [1]

Unlike traditional decision trees, XGBoost employs a more sophisticated approach to building trees. It utilizes a split finding algorithm that considers both the information gain and the regularization penalty. This regularization helps prevent overfitting, a common issue in tree-based models.

- It consists of a sequential assembly of trees of decision (this assembly is known as a CART, acronym for "Classification and Regression Trees"). Trees are added sequentially in order to learn from the result of the previous trees and correct the error produced by them, until this error can no longer be corrected (this is known as "descending gradient".
- In the XG Boost algorithm the user defines the extent of the trees.
- Uses parallel processing, tree pruning, handling of missing values and regularization (optimization that penalizes the complexity of the models) to avoid as much as possible overfitting or bias of the model.

XGBoost mitigates the risk of overfitting by incorporating regularization terms and directly leveraging the first and second derivatives of the loss function.

Is a scalable, distributed gradient-boosted decision tree (GBDT) machine learning library. It provides parallel tree boosting and is the leading machine learning library for regression, classification, and ranking problems. [2]

Applications of XGBoost:

1) Classification Tasks: XGBoost excels at classification tasks, such as predicting customer churn, identifying spam emails, and diagnosing medical conditions.
2) Regression Tasks: XGBoost is also effective in regression tasks, such as predicting house prices, estimating insurance premiums, and forecasting sales trends.
3) Ranking Tasks: XGBoost can be used for ranking tasks, such as prioritizing search results, recommending products, and selecting relevant content. [3]

Key Components of XGBoost:

1) Decision Trees: XGBoost utilizes decision trees as its base learners. These trees are typically small and shallow, allowing for efficient training and reduced overfitting.
2) Gradient Boosting: The gradient boosting framework ensures that each tree focuses on correcting the errors made by the previous trees, leading to a cumulative improvement in predictive performance.

3) Regularization: Regularization techniques, such as L1 and L2 regularization, are incorporated to prevent overfitting by penalizing complex models.
4) Sparsity: XGBoost encourages sparsity by automatically selecting features that are most relevant to the prediction task, reducing computational cost and improving model interpretability.
5) Parallel Processing: XGBoost is designed for parallel processing, enabling efficient training on large datasets. [4]

A. Mathematics of the XGBoost

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F}$$

Fig. 1. Common model of tree ensembles

In the previous model the K is the number of trees , f_k is a function in the functional space F, and F is the set of all possible CARTs. The objetctive function to be optimized is given by:

$$\text{obj}(\theta) = \sum_{i}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \omega(f_k)$$

Fig. 2. objective function

where $\omega(f_k)$ is the complexity of the tree.

(f_i) functions contain the structure of the tree and the leaf scores. Learning tree structure is much harder than traditional optimization problem where you can simply take the gradient. Instead, it uses an additive strategy: fix what we have learned, and add one new tree at a time. We write the prediction value at step (t) as y_i^t. Then we have

$$\begin{aligned}
\hat{y}_i^{(0)} &= 0 \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\
&\dots \\
\hat{y}_i^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)
\end{aligned}$$

Fig. 3.

At each step a natural thing is to add the one that optimizes our objective.

$$\begin{aligned}
\text{obj}^{(t)} &= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \omega(f_i) \\
&= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \omega(f_t) + \text{constant}
\end{aligned}$$

Fig. 4.

If is consider using mean squared error (MSE) as the loss function, the objective becomes:

$$\begin{aligned}
\text{obj}^{(t)} &= \sum_{i=1}^{n} (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \sum_{i=1}^{t} \omega(f_i) \\
&= \sum_{i=1}^{n} [2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2] + \omega(f_t) + \text{constant}
\end{aligned}$$

Fig. 5.

The form of MSE is friendly, with a first order term (usually called the residual) and a quadratic term. For other losses of interest (for example, logistic loss), it is not so easy to get such a nice form. So in the general case, we take the Taylor expansion of the loss function up to the second order:

$$\text{obj}^{(t)} = \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \omega(f_t) + \text{const}$$

Fig. 6.

where (g_i) and (h_i) are difined as

$$\begin{aligned}
g_i &= \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \\
h_i &= \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})
\end{aligned}$$

Fig. 7.

That after remove all the constants, the specific objective at step (t) becomes

This becomes the optimization goal for the new tree. One important advantage of this definition is that the value of the objective function only depends on (g_i) and (h_i) . This is how XGBoost supports custom loss functions. We can optimize every loss function, including logistic regression and pairwise ranking, using exactly the same solver that takes (g_i) and (h_i) as input.

Then, it's needed define the complexity of the tree $\omega(f)$. In order to do so, refine the definition of the tree $f(x)$ as:

$$\sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \omega(f_t)$$

Fig. 8.

$$f_t(x) = w_{q(x)}, w \in R^T, q : R^d \to \{1, 2, \cdots, T\}.$$

Fig. 9. Definition of the tree

Here w is the vector of scores on leaves, q is a function assigning each data point to the corresponding leaf, and T is the number of leaves. In XGBoost, the complexity is defined as:

$$\omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Fig. 10. Complexity

There are various approaches to defining complexity, but this particular one demonstrates effective performance in practical applications. The aspect of regularization tends to receive less attention or is often overlooked in most tree algorithms. This oversight stems from the historical emphasis on enhancing impurity reduction in tree learning, while complexity control was primarily left to heuristic methods. By establishing a formal definition, one can gain a clearer understanding of the learning process and obtain models that exhibit robust performance in real-world scenarios.

B. How XGBoost works

It works well with large, complicated datasets by using various optimization methods.
To fit a training dataset using XGBoost, an initial prediction is made. Residuals are computed based on the predicted value and the observed values. A decision tree is created with the residuals using a similarity score for residuals. The similarity of the data in a leaf is calculated, as well as the gain in similarity in the subsequent split. The gains are compared to determine a feature and a threshold for a node. The output value for each leaf is also calculated using the residuals. [5]

For classification, the values are typically calculated using the log of odds and probabilities. The output of the tree becomes the new residual for the dataset, which is used to construct another tree. This process is repeated until the residuals stop reducing or for a specified number of times. Each subsequent tree learns from the previous trees and is not assigned equal weight, unlike how Random Forest works. To use this model for prediction, the output from each tree multiplied by a learning rate is added to the initial prediction to arrive at a final value or classification. [6]

This is a parallelizable algorithm, allowing for the optimal utilization of the processing power available today. It can harness multiple cores simultaneously. Moreover, it can be executed in parallel on GPUs or even on clusters of servers. In essence, it exhibits an exceptionally high training speed. This means that training the model on massive datasets is not as challenging as it might be for another algorithm. It enhances the performance of the vast majority of algorithms commonly used in almost all Machine Learning competitions.

Thanks to this formidable performance, XGBoost has demonstrated state-of-the-art results across a wide range of Machine Learning benchmarks.

C. Advantages

1) It can handle big databases with multiple variables.
2) Can handle missing values.
3) Its results are very precise.
4) Excellent speed of execution.

D. Disadvantages

1) Can consume a lot of computing resources in big databases, so it is recommended before applying this technique on bases of this type, determine which variables will contribute more information in order to consider only these variables in obtaining the model.
2) The parameters of the device must be adjusted correctly algorithm in order to minimize the precision error and avoid overfitting of the model (which can occur if a very large number of trees are managed).
3) It only works with numerical vectors, so require prior conversion of data types non-numeric to numeric

E. XGBoost parameters

In supervised learning, the term ”model” typically alludes to the underlying mathematical framework used to generate predictions, denoted as $\mathbf{y}_i$, based on input data, $\mathbf{x}_i$. A common illustration of such a model is the linear model $\hat{\mathbf{y}}_i = \mathbf{\Sigma}_j \theta_j \mathbf{x}_i j$, where the prediction takes the form of a linear combination of input features with associated weights. The interpretation of the prediction can vary depending on the specific task, whether it involves regression or classification. For instance, in logistic regression, it can be subjected to a logistic transformation to obtain the probability of belonging to the positive class. Alternatively, when prioritizing outcomes, it can serve as a ranking score.

The components referred to as ”parameters” are the variables within the model that remain undetermined initially and must be inferred from the available data. In the context of linear regression problems, these parameters often manifest as the coefficients represented by the symbol θ. It’s worth noting that there may be multiple parameters in a given model, and the use of θ here is a simplified representation for clarity.

F. XGBoost Algorithm

Algorithm 1 XGBoost Algorithm

Require: Training dataset: (X_i, y_i) for $i = 1, 2, \ldots, N$
Require: Hyperparameters: Θ
Initialize an empty set of trees: $\{f_k\}$
while Not converged **do**
for each tree k in the set of trees **do**
Fit a decision tree that minimizes the cost function $\mathcal{L}(\Theta)$
Update the set of trees with the new tree f_k
Update the model predictions $\hat{y}_i$
end for
end while
Prediction:
For each instance i, calculate the final prediction: $\hat{y}_i = \sum_{k=1}^{K} f_k(X_i)$
Evaluation:
Evaluate the model's performance on a test set using appropriate metrics.
Hyperparameter Tuning:
Adjust hyperparameters Θ as needed and repeat the training and evaluation steps to optimize the model. =0

G. XGBoost Pseudocode

```
1. Load the training and testing data set.
2. Initialize the variables and hyperparameters:
   - Number of trees (n_estimators).
   - Maximum depth of trees (max_depth).
   - Learning rate (learning_rate).
   - Loss function.
   - XGBoost-specific hyperparameters (e.g. gamma,
     lambda, alpha).
   - Minimum number of samples per sheet (
     min_child_weight).
   - Maximum number of leaf nodes (max_leaf_nodes).
   - Regularization (lambda and alpha) and learning
     rate (eta).
3. Initialize the set of empty trees.
4. For each tree in the tree set:
   4.1. Initialize the tree as a root node.
   4.2. While the tree does not meet the stopping
        criteria (maximum depth, minimum number of
        samples, etc.):
        4.2.1. Compute the gradient and Hessian for
             the loss function at that node.
        4.2.2. Calculates the gain (score) for all
             candidate features.
        4.2.3. Select the characteristic with the
             highest gain and its optimal cutoff value
             .
        4.2.4. Split the node into two children
             according to the selected characteristic.
   4.3. Perform regression (or classification) on
        the tree leaves using the specific loss
        function.
   4.4. Apply regularization to the leaves of the
        tree.
   4.5. Add the tree to the tree set.
5. Make predictions on the test set using all trees
   constructed and average the results (in
   regression) or use voting (in classification).
6. Evaluate model performance using appropriate
   metrics (e.g., root mean square error or
   precision).
7. Adjust the hyperparameters if necessary and
   repeat steps 2 to 6 to optimize the model.
8. End of the XGBoost algorithm.
```

III. Development and Analysis with XGBoost Algorithm

A. Dataset Description

The Titanic dataset serves as a historical record, capturing demographic and passenger information for a subset of the individuals aboard the ill-fated maiden voyage of the RMS Titanic. Utilized extensively in predictive modeling and machine learning, this dataset comprises several features: *PassengerId*, a unique identifier; *Survived*, indicating if a passenger lived or perished; *Pclass*, denoting passenger class as an indicator of socio-economic status; *Name*, the passenger's name; *Sex*, the passenger's gender; *Age*; *SibSp* and *Parch*, reflecting the number of relatives on board; *Ticket*, the ticket number; *Fare*, the amount paid for the journey; *Cabin*, and *Embarked*, the port of embarkation. [6]

B. Preprocessing and Feature Engineering

In the Jupyter notebook under analysis, a selection of relevant features is made, including *Pclass*, *Sex*, *Age*, *Survived*, *Parch*, and *SibSp*. The feature *Sex* is transformed into a numerical representation conducive to algorithmic processing. Missing values are subsequently expunged, ensuring the integrity of the dataset for training purposes.

C. XGBoost Algorithm Overview

XGBoost (eXtreme Gradient Boosting) stands as a potent implementation of gradient boosting machines, a class of ensemble learning algorithms that construct a robust predictive model by amalgamating the outcomes of numerous weak decision tree models. It operates on the principle of boosting, an approach where each subsequent tree endeavors to rectify the residuals of its predecessors, thus enhancing the model's predictive accuracy.

D. Implementation with Python Libraries

In the implementation, the Python library 'xgboost' is employed, specifically invoking the 'XGBClassifier'. This classifier is adept at handling the binary classification task of predicting survival on the Titanic. The library's functionality is further exploited to visualize the importance of each feature in the model, granting insightful revelations into the predictive power wielded by each feature.

E. Model Performance

The performance of the XGBoost model is gauged by its accuracy score on both training and test datasets. The confusion matrix serves as an auxiliary tool, providing a granular view of the model's classification prowess. Through

iterative training, the model demonstrates a remarkable capacity to discern patterns that are pivotal to the prediction of survival, as indicated by the observed accuracy scores. The Jupyter Notebook as well as the dataset files can be found Appendices section of the document via GitHub or Google Colab (Notebook exclusively).

IV. Conclusion

The XGBoost algorithm is a powerful gradient boosting machine (GBM) known for its efficiency in handling large-scale data and delivering high predictive accuracy. This paper explores the evolution of GBMs and the specific challenges they address, highlighting how XGBoost distinguishes itself from traditional GBMs by using a gradient descent algorithm for loss minimization.

XGBoost's system design and sophisticated approach to sparse data management, including a sparsity-aware algorithm for split finding, enhance its capability to handle various missing data scenarios effectively. The algorithm excels in both classification and regression tasks, providing high accuracy.

The paper discusses the importance of regularization techniques in XGBoost, which help prevent overfitting, particularly in high-dimensional data where overfitting is a significant concern.

Additionally, the paper includes a practical code example to illustrate the application of XGBoost, providing insights into parameter adjustments and the practical aspects of developing a high-performing model. XGBoost is a preferred choice for data practitioners due to its exceptional performance in machine learning competitions and its utility in real-world applications.

Appendix A
Jupyter Notebook on Google Colab

The Jupyter Notebook complementing this work can be accessed on Google Colab at the following link:

- https://colab.research.google.com/drive/1UWsrm-EYfhsNimOEoEITR9Qqh_eO6VH0?usp=sharing

Appendix B
GitHub Repository

The code and data for the project can be found in the following GitHub repository:

- https://github.com/Maages09/XGBoost.git

References

[1] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining* (KDD '16), San Francisco, CA, USA, 2016, pp. 785-794, doi: 10.1145/2939672.2939785.

[2] "What is XGBoost?" NVIDIA Data Science Glossary. [Online]. Available: https://www.nvidia.com/en-us/glossary/data-science/xgboost/. [Accessed: Nov. 5, 2023].

[3] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*, Springer, 2013.

[4] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," in *Proceedings of the 13th International Conference on Machine Learning*, 2001, pp. 291-298.

[5] "How XGBoost algorithm works—ArcGIS Pro — Documentation." [Online]. Available: https://pro.arcgis.com/en/pro-app/latest/tool-reference/geoai/how-xgboost-works.htm. [Accessed: Nov. 5, 2023].

[6] Kaggle, "Titanic: Machine Learning from Disaster," Kaggle. [Online]. Available: https://www.kaggle.com/c/titanic/data. [Accessed: Nov. 5, 2023].