COMPX521 Assignment 1

Sumeet

August 26, 2024

1 Introduction

XGBoost is an advanced machine learning algorithm that enhances accuracy by combining several decision trees through an efficient, scalable boosting method. It is widely used by researchers and data professionals across different fields for analysing and solving complex data patterns.

The scalability architecture of the XGBoost learning algorithm is first presented in this 2016 paper [1]. It covers a wide range of topics that include regularized objectives for learning, split-finding techniques, system improvements, and performance assessments on large-scale datasets. This combination significantly influences real-world applications and competitions.

This project implements a custom XGBoost solution in Java using the WEKA framework [2] and compares its performance with the official XGBoost implementation, which is available through the XGBoost library and can be used with Scikit-learn.

2 Methodology

For this project, I followed the approach outlined in this 2016 paper [1]. The key steps of the implementation are discussed below.

2.1 Tree Structure and Node Representation

This approach uses internal nodes and leaf nodes to represent the tree structure, with each internal node dividing the data according to a selected attribute. It uses Java records to efficiently represent nodes and maintain clarity in the tree structure.

2.2 Determining the Split Point using the Exact Greedy Approach

The process of choosing the best split point in a node focuses on finding a balance between improving the prediction and keeping the model from becoming too complex. It uses gradients, which indicate the direction and size of prediction

errors. In order to keep the model from getting too complicated and over fitting, regularization terms like λ and γ are required to limit the leaf node weight changes in predictions and restrict the creation of expensive nodes.

Therefore, as demonstrated by Eq. (1), the split quality section of the project uses a gain calculation [1] to guide the splitting process and select the split with the highest gain for effective tree growth.

$$\text{Gain} = \frac{1}{2}\left(\frac{G_{jL}^2}{H_{jL}+\lambda} + \frac{G_{jR}^2}{H_{jR}+\lambda} - \frac{(G_{jL}+G_{jR})^2}{H_{jL}+H_{jR}+\lambda}\right) - \gamma \tag{1}$$

Where:

- G_{jL} and G_{jR} are the gradient sums for the left and right branches.
- H_{jL} and H_{jR} are the Hessian (second-order derivative to measure the steepness) sums for the left and right branches.
- λ and γ are the control factors.

The implementation looks for the best split by sorting indices based on attribute values and checking which split gives the highest quality by updating the necessary statistics for evaluating each split. An early stopping mechanism is added to stop the process when there is minimal improvement in split quality to minimize computational costs.

2.3 Leaf Node Prediction Calculation

This paper [1] also introduces the concept of determining the optimal weight for a leaf given by Eq. (2). It is calculated by balancing the sum of gradients and the sum of second-order gradient statistics for that node which is handled in the make tree section of the implemented code.

$$w^* = -\frac{\sum_{i\in I} g_i}{\sum_{i\in I} h_i + \lambda} \tag{2}$$

Where:

- g_i represents the gradient of the loss function.
- h_i represents the Hessian.
- λ represents the regularization term.

2.4 Techniques for Preventing Overfitting: Shrinkage, Column, and Row Subsampling

All three techniques—shrinkage (using the learning rate η to reduce each tree's impact), column and row subsampling (selecting a random subset of features and instances) are implemented in tree creation. The algorithm stops the splitting process of recursively growing a tree if the minimum child weight is higher than the sum of the second-order derivatives, if the sum goes to zero or negative due to numerical instability, or if the maximum tree depth is reached. These strategies help keep the model accurate, efficient, and resistant to overfitting.

3 Experimental results

In this experiment, the custom XGBoost implementation was compared with Scikit-Learn's `XGBoostClassifier` and `XGBoostRegressor` for given datasets. Predictive performance was calculated using 10 runs of 10-fold cross-validation with WEKA's experimenter [2]. The experiment was configured with consistent hyperparameters ($gamma = 1$, `subsample` $= 0.5$, and `tree_method` = "exact") and the other default XGBoost Parameters.The results of the experiment are presented in Tables 1 and 2.

Table 1: Regression Results: Root Relative Squared Error

Dataset	(1)	(3)
2dplanes	23.27	23.19 •
bank8FM	31.75	31.70
cpu-act	13.76	13.37
cpu-small	16.02	15.79
delta-ailerons	100.01	100.00
delta-elevators	100.00	100.00
diabetes-numeric	90.08	90.60
kin8nm	59.31	66.94 ○
machine-cpu	37.28	36.77
puma8NH	63.18	63.45
pyrim	99.84	100.19
stock	12.80	13.01
triazines	100.12	100.18
wisconsin	111.03	111.69

Table 2: Classification Results: Percent Correct

Dataset	(2)	(3)
balance-scale-weka.filter(100)	95.71	94.91
wisconsin-breast-cancer-w(100)	96.27	96.35
pima-diabetes-weka.filter(100)	72.89	74.03
ecoli-weka.filters.unsup(100)	95.92	96.01
Glass-weka.filters.unsup(100)	83.03	81.58
ionosphere-weka.filters.u(100)	92.94	92.54
iris-weka.filters.unsuper(100)	100.00	100.00
optdigits-weka.filters.un(100)	99.22	99.18
pendigits-weka.filters.un(100)	99.69	99.64
segment-weka.filters.unsu(100)	99.67	99.61
sonar-weka.filters.unsupe(100)	84.82	83.26
vehicle-weka.filters.unsu(100)	98.36	98.12
'vowel-weka.filters.unsup(100)	99.33	99.06
waveform-weka.filters.uns(100)	88.34	88.51

○, • statistically significant improvement or degradation

(1) sklearn.ScikitLearnClassifier '-batch 100 -learner XGBRegressor -parameters \"gamma=1,subsample=0.5,tree-method=\\\'exact\\\'\" -py-command python3 -py-path default -server-name none' -6212485658537766441
(2) sklearn.ScikitLearnClassifier '-batch 100 -learner XGBClassifier -parameters \"gamma=1,subsample=0.5,tree-method=\\\'exact\\\'\" -py-command python3 -py-path default -server-name none' -6212485658537766441
(3) meta.XGBoost '-S 1 -I 100 -W trees.XGBoostTree – -S 1 -colsample-bynode 1.0 -eta 0.3 -gamma 1.0 -lambda 1.0 -max-depth 6 -min-child-weight 1.0 -min-instances-in-leaf 1 -subsample 0.5' -7904453909409312251

4 Evaluation

Table 1 and 2 shows the prediction accuracy for Regression and classification models.
This model implementation focuses on accuracy (percent correct) for classification and root relative squared error (RRSE) for regression. Glancing through the datasets used in the regression model, there is no significant improvement with respect to original model except for one dataset. Result shows the degradation of the performance for "kin8nm" dataset due to the current hyperparameter which is not good enough to capture the generalized pattern in large datasets.

For the classification datasets, the implemented model is comparable to the original classifier model as no significant improvement or degradation has been observed.

5 Conclusions

To conclude, the results of this custom implementation are quite comparable to the official XGBoost, with slight improvements observed on certain datasets, possibly due to better leaf node predictions and refined stopping criteria. These results highlight how the adjustments helped reduce overfitting and enhance model generalization. Moving forward, exploring advanced regularization techniques, optimizing split point selection, and further refining hyperparameters could offer even better performance.

References

[1] 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.

[2] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Burlington, MA, 3 edition, 2011.