Stockholms universitet

A Comparative Performance Analysis of Random Forest and XGBoost in Regression for Various Noise Levels and Data Patterns

Fikrat Ibragimov

Kandidatuppsats i matematisk statistik
Bachelor Thesis in Mathematical Statistics

Kandidatuppsats 2025:3
Matematisk statistik
Februari 2025

www.math.su.se

Matematisk statistik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm

Mathematical Statistics
Stockholm University
Bachelor Thesis **2025:3**
http://www.math.su.se

A Comparative Performance Analysis of Random Forest and XGBoost in Regression for Various Noise Levels and Data Patterns

Fikrat Ibragimov*

February 2025

Abstract

This thesis compares two powerful tree based machine learning algorithms, Random Forest and XGBoost, with a focus on their ability to generalize under varying levels of noise. Simulated datasets were designed to include four regression functions. These regression functions corresponds to four distinct geometric shapes, referred to as Sphere-Truncated Cone, Cone, Rhombus-Truncated Pyramid, and Ridge, defined for five predictor variables and two noise levels, categorized as "high" and "low." These shapes serve as a challenge for both algorithms, evaluating their performance under structured and noisy conditions. The theoretical component of this study delves into the mathematical foundations of Random Forest and XG Boost, providing a step-by-step analysis of their mechanisms, including tree construction, optimization processes, and ensemble strategies. Using Mean Squared Error (MSE) and Mean Squared Error of Prediction (MSEP) for test data as evaluation metrics, this work systematically assesses the algorithms' performance across the simulated datasets. The findings demonstrate that Random Forest excels in handling regression functions with discontinuities, while Gradient Boosting performs better for regression functions with continuous and smooth patterns.

*Postal address: Mathematical Statistics, Stockholm University, SE-106 91, Sweden. E-mail: Fikrat-ibragimov@live.se. Supervisor: Ola Gerton Henrik Hössjer och Johannes Heiny.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Ola Gerton Henrik Hössjer and Johannes Heiny, for their invaluable guidance and support throughout the process of writing this thesis. In particular, I am deeply grateful to Ola Gerton Henrik Hössjer for his constructive comments and detailed feedback, which have significantly contributed to the improvement and refinement. His insights and encouragement have been instrumental in shaping the final version of this thesis.

Additionally, I acknowledge the use of ChatGPT as an interactive learning aid, which helped me deepen my understanding of theoretical concepts. Grammarly was also used to enhance the clarity and readability of the thesis.

Contents

1 Introduction

Tree-based machine learning algorithms, such as Random Forest and XGBoost, have significantly enhanced predictive modeling by capturing complex, nonlinear relationships in data. Despite their widespread success, understanding how effectively these models generalize under varying levels of noise and across diverse data structures remains a critical area of exploration. This study specifically examines their performance under two distinct noise levels, "low" and "high", to uncover insights into their robustness and adaptability.

The problem this thesis aims to solve is understanding the comparative performance of Random Forest and XGBoost regression algorithms under diverse conditions. Specifically, the goal is to determine how well each algorithm generalizes when applied to datasets with varying geometric structures and noise levels. Using a number of different regression functions, characterized by various geometric structures, including sphere-truncated cones, cones, rhombus-truncated pyramids, and ridges, the study investigates the capacity of these algorithms to model feature interactions and adapt to the complexities of different data structures.

By systematically comparing the performance of Random Forest and XGBoost under controlled conditions, this thesis seeks to provide a deeper understanding of their behavior in structured environments. The analysis focuses on how these algorithms handle complex geometric patterns within the data and examines the impact of varying noise levels on their generalization capabilities. Through this investigation, the study aims to uncover the strengths and limitations of both Random Forest and XGBoost for diverse regression scenarios. By analyzing their performance, the research not only contributes to algorithmic understanding but also provides actionable insights for practitioners, helping them make informed decisions about the choice of machine learning method.

The thesis is structured as follows: Section 2 introduces the theoretical foundations and essential background information necessary for understanding the study. Section 3, Simulation and Data, outlines the methodological framework and describes the data generation process, including the construction of synthetic datasets with different geometric structures and varying noise levels. Section 4, Algorithms and Performance Criteria, discusses the estimation and prediction based evaluation metrics employed in the analysis. Finally, the thesis concludes with Results and Discussion, where the findings are presented and analyzed in relation to the broader research context, highlighting their implications for both theory and practice.

2 Theory

2.1 Regression Model

Suppose we have a dataset $\{(X_i, Y_i)\}_{i=1}^N$, where $X_i = (X_{i1}, X_{i2} \ldots, X_{ip})$ is the vector of p predictor variables or features, and Y_i the dependent variable (or outcome variable), for observation $i = 1, \ldots, N$. The regression model is defined as

$$Y_i = f(X_i) + \varepsilon_i, \quad i = 1, \ldots, N,$$

where $f(x) = E(Y|X = x)$ is the regression function, that is, the expected outcome Y given an observed value x of the predictor vector X. In this thesis, we assume that the outcome variables are quantitative, with a continuous distribution. More specifically, we will assume that the error terms $\varepsilon_i \sim N(0, \sigma^2)$ are normally distributed with mean 0 and variance σ^2. We will also denote the observed value of Y_i by y_i.

The total number of observations is $N = N_0 + N_1$. The first N_0 of these observations will be used for training, in order to estimate f, whereas the remaining N_1 observations will serve as test data. An estimate of $f(x)$, based on training data, is denoted as $\hat{f}(x)$. The quality of this fit is assessed by the smallness of the prediction errors $Y_i - \hat{Y}_i = Y_i - \hat{f}(X_i)$ for test data $i = N_0 + 1, \ldots, N_0 + N_1$.

2.2 Decision Trees

Decision trees are a widely used tool in supervised learning for both classification and regression tasks. They simplify complex decision-making by iteratively splitting data into smaller, homogeneous subsets, making it easier to visualize decision paths and analyze potential outcomes. In a decision tree, data is split at each node according to specific criteria, aiming to minimize impurity or variance within each subset. Each branch terminates at a leaf node, which provides a prediction for the dependent variable, helping to ensure that nodes contain data with similar characteristics (Breiman et al. 2017).

There are two main types of decision trees: classification trees and regression trees. Classification trees are used to predict categorical outcomes by splitting data based on measures like Gini impurity or information gain, depending on the algorithm. Regression trees, on the other hand, are used to predict continuous values by minimizing the variance within each node. This approach allows regression trees to capture nonlinear relationships $f(x)$ between predictors x and the dependent variable Y, effectively averaging outcomes across training observations within the same node to improve predictive accuracy (Breiman et al. 2017).

For effective splits, the choice of impurity measure is critical. In classification trees, Gini impurity (commonly used in the CART algorithm) and information gain (used in algorithms like ID3 and C4.5) are popular metrics to assess the purity of each split, guiding the tree toward homogeneous groups of observations. In regression trees, the primary criterion is variance reduction, which evaluates splits based on the extent to which each partition minimizes variance within its subset (Breiman et al. 2017).

Visualizing decision tree structures offers a clear, interpretable view of the model's decision-making process. This interpretability is a key advantage, as each path in the tree transparently represents a series of decisions, especially in smaller trees. Additionally, decision trees have several practical benefits over many algorithms: firstly, they are invariant to data scaling, as each feature is processed independently, making normalization or standardization unnecessary, and secondly, they handle features of varying types and scales effectively, working well with both binary and continuous variables. However, a notable downside of decision trees is their tendency to overfit, capturing noise in the training data and limiting generalization to new datasets (Müller and Guido, 2016, p. 83).

Overfitting can be mitigated through techniques like pruning and limiting the tree depth. Pruning, which can be applied as either pre-pruning (stopping tree growth early) or post-pruning (removing branches from a fully grown tree), reduces model complexity by removing branches with limited predictive value, enhancing the tree's generalizability. Limiting the tree depth similarly prevents the model from growing overly complex, contributing to improved performance on unseen data (Müller and Guido, 2016, p. 74).

2.2.1 Regression Tree

A regression tree divides the feature space into M disjoint regions denoted as $R_1, R_2, ..., R_M$, with each region assigned a specific constant value c_m. The function $f(x)$ outputs the predicted value based on the region to which x belongs. More precisely, one has

$$f(x) = \sum_{m=1}^{M} c_m \mathbb{1}(x \in R_m), \tag{1}$$

where $\mathbb{1}$ is the indicator function, equaling 1 if $x \in R_m$ and 0 otherwise.

The best estimate $\hat{c}_m$ of the constant value c_m is just an average of y_i in region R_m. This average is the best estimate in the sense that it minimizes the empirical variance of the residuals within that region, leading to the smallest possible sum of squared residuals for the data points in R_m. This can be formalized as

$$\hat{c}_m = ave(y_i | x_i \in R_m), \tag{2}$$

where $ave()$ represents the average over y_i values in region R_m (Hastie, Tibshirani, & Friedman, 2009, p. 307).

Determining $\hat{c}_m$is straightforward once the regions R_m are established. However, identifying these regions involves selecting optimal binary splits across all features and split points, which according to Hastie et al. (2009) is "generally computationally infeasible". Thus a greedy algorithm is typically used, selecting splits that locally minimize the sum of squared errors (SSE) at each node.

Split Selection Process. The splitting process begins by considering all the training data in the root node. For each node, the decision tree algorithm selects a splitting variable k from the p features of the datasets and a corresponding split point s that divides the data into two groups. The selected split minimizes the SSE or sum of squared residuals (SSR) within each resulting group, aiming to maximize the homogeneity of the response variable in each child node.

In this context, the splitting point s refers to a specific threshold for the chosen feature k, creating two regions

$$R^1(k, s) = \{X | X_k \leq s\}$$

and

$$R^2(k, s) = \{X | X_k > s\}.$$

This process iterates, further subdividing each node until a stopping criterion is met. Common stopping criteria include a minimum number of observations in each node or a maximum tree depth, which helps balance model complexity and computational efficiency (James et al., 2013, p. 306-307).

Optimization of the Splitting Process. The objective is to find a split that minimizes the within node residual sum of square by selecting optimal values for both the splitting variable k and the split point s. Mathematically, this can be expressed as

$$\min_{k,s} \left[\min_{c_1} \sum_{x_i \in R^1(k,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R^2(k,s)} (y_i - c_2)^2 \right]. \tag{3}$$

where c_1 and c_2 are the predicted values for regions $R^1(k, s)$ and $R^2(k, s)$, respectively. This nested optimization first identifies the split (k, s) that best partitions the data and subsequently estimates c_1 and c_2 using equation (2). This greedy, step by step selection does not consider future splits, focusing only on minimizing error within the current step (Hastie et al., 2009, p. 307).

Figure 1 illustrates a regression tree with a depth of four, constructed from binary splits. Each node represents a subset of the data, with the root node encompassing the entire dataset. At each split, the algorithm selects an optimal splitting variable (e.g., x_3) and point (e.g., 0.51) dividing the data based on this criterion. The terminal or leaf nodes at the bottom represent distinct predictions for any data point falling within each corresponding region. The predicted value for each leaf node, shown in the upper part of the oval, represents the average $\hat{y}$ within that region. Additionally, the notation n denotes the number of observations in each node, while the percentage indicates the proportion of the datasets represented in that region.

This structure provides a clear visualization of how the data is segmented and predictions are made based on different criteria defined by the decision tree. Each node provides insightful details about the subset of data it contains, facilitating understanding of the model's decisions and predictions.

Bias and Variance Trade-off

The bias-variance trade-off is a key concept in machine learning and statistics that describes the interplay between two sources of error that affect the performance of predictive models, bias and variance. This trade-off is illustrated in Figure 2, which provides two perspectives: a graphical representation of bias and variance in predictions and a plot showing how these errors often vary with model complexity. Bias represents the error caused by simplifying assumptions made by a model. For instance, when a model is too simple, such as a linear regression applied to nonlinear data, it will fail to capture the true structure of the data, leading to high bias. This typically results in underfitting, where the model has poor performance on both training and unseen data.

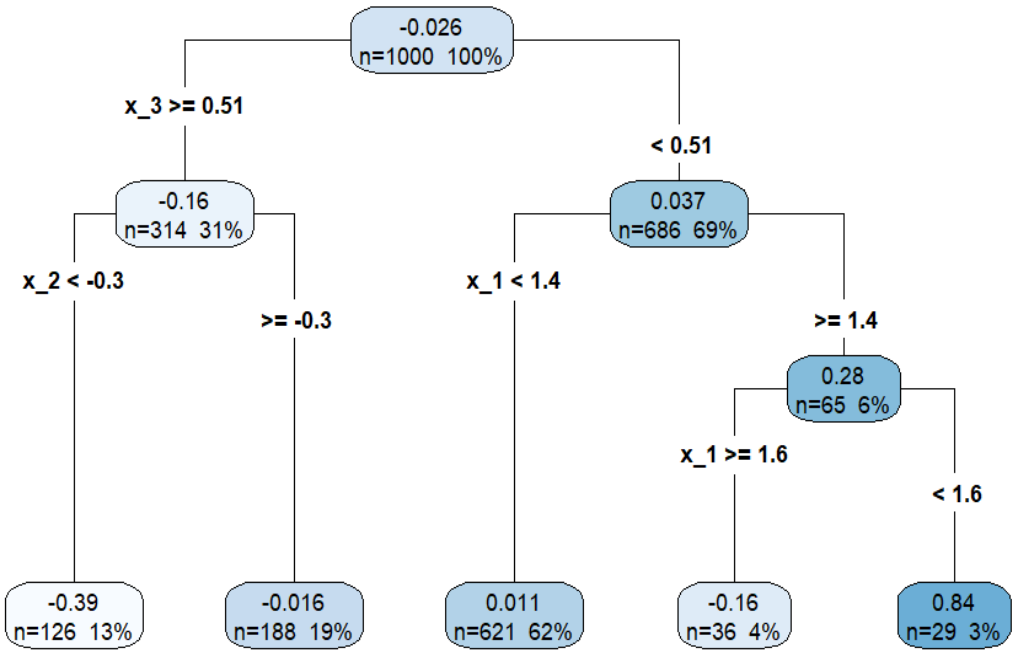


Figure 1: Example of a regression tree for a predictor space with $p = 5$ input variables. The tree has a depth of four, constructed from four binary splits, and there are five terminal nodes at the lowest level.

As shown in Figure 2(a), a model with high bias tends to make predictions that are consistently far from the target, regardless of the specific data points. Variance, in contrast, measures how sensitive a model is to fluctuations in the training data. Highly complex models, such as those with many parameters or high flexibility, tend to overfit the training data by capturing noise or random fluctuations rather than the underlying pattern. This results in high variance, where the model performs well on training data but poorly on unseen test data. In Figure 2(a), a model with high variance produces predictions that are widely scattered, even when they have no systematic departure from the target.

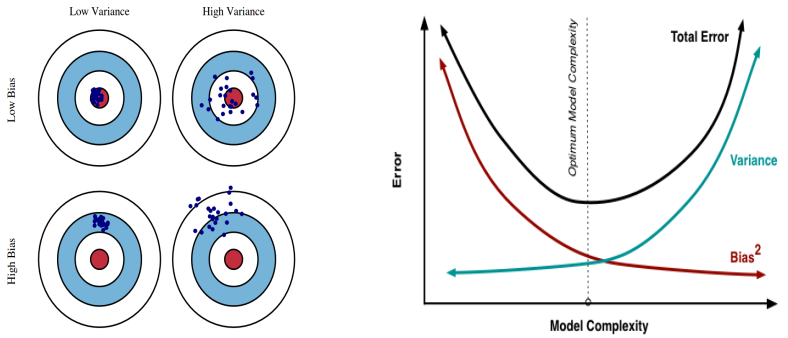
Figure 2(b) demonstrates the relationship between bias, variance and total error as a function of model complexity. As model complexity increases, bias decreases because the model becomes more capable of capturing intricate patterns in the data. However, variance increases because the model becomes more sensitive to small changes in the training set. The total expected squared prediction error, which is the sum of squared bias, variance, and irreducible error (the inherent noise in test data), is minimized at an optimal level of model complexity. This is the point where the model achieves the best balance between underfitting and overfitting, ensuring good generalization to unseen data.

2.3 Bootstrap Aggregation

Bagging, short for bootstrap aggregation, is an ensemble learning technique designed to improve the stability and accuracy of machine learning algorithms. The method generates B random samples from the training data with replacement, each with same size as the original dataset. A separate model is trained on each sample, allowing each model to learn from slightly different data. Predictions from all models, $\hat{f}_1(x), \ldots, \hat{f}_B(x)$, are combined and averaged to obtain the final prediction

$$\hat{f}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}_b(x). \tag{4}$$

Bagging reduces the variance of models prone to overfitting, such as decision trees, without significantly



(a) Graphical illustration of bias and variance.

(b) The variation of Bias and Variance with the model complexity.

Figure 2: Bias and Variance, (Fortmann-Roe, 2012).

increasing bias. This characteristic makes bagging particularly effective when the base learner is highly variable, providing more stable and generalized predictions (Hastie et al., 2009, p. 587-588).

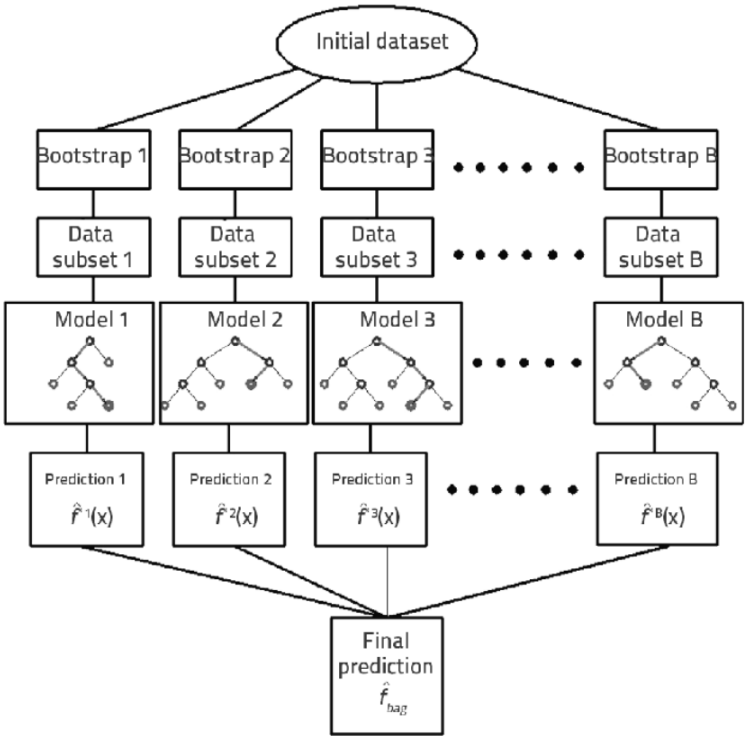


Figure 3: Schematic illustration of bagging, (Kovačević et al, 2021).

2.4 Random Forest

Random Forest builds on the principles of bagging by incorporating additional randomization during the construction of individual trees. While bagging trains each tree on a bootstrap sample of the training data, Random Forest introduces another layer of randomness by selecting a random subset of predictors at each split in the tree. This random feature selection reduces the correlation between trees, a limitation of traditional bagging, and leads to improved ensemble performance.

The construction of a Random Forest involves:

1. Bootstrap Sampling: Each tree is trained on a random bootstrap sample from the training data.

2. Random Feature Selection: At each split, a random subset of predictors is considered, ensuring that trees are decorrelated and less likely to overfit to the training data.

Although individual trees in the Random Forest may be noisy and overfit their respective bootstrap samples, the aggregation of these trees through averaging mitigates these issues, leading to more robust predictions. The combined model is particularly effective in handling complex data structures and mitigating the impact of noise, making it a powerful tool for a wide range of predictive tasks (Hastie et al., 2009, p. 588).

Since each of the B generated trees is drawn from the same dataset, they are not independent but are identically distributed (i.d.). This implies that the expectation of any individual tree is the same as the average expectation of the B trees. Although trees are based on simple decisions, they do not systematically favour one outcome over another, resulting in a low bias. Therefore, the only means of enhancing the model's accuracy through bagging is by reducing variance. Given the assumption that predictions $\hat{Y}_b = \hat{f}_b(x)$ are i.d. with variance σ^2, and correlation ρ among predictions

$$Corr(\hat{Y}_{b_1}, \hat{Y}_{b_2}) = \rho > 0, \quad \text{for all } b_1 \neq b_2,$$

the variance of the average prediction across all trees is

$$Var\left(\frac{1}{B}\sum_{b=1}^{B}\hat{Y}_b\right) = \frac{1}{B^2}[B(Var(\hat{Y}_b) + B(B-1)Cov(\hat{Y}_{b_1}, \hat{Y}_{b_2})] = \rho\sigma^2 + \frac{(1-\rho)}{B}\sigma^2. \tag{5}$$

As $B \to \infty$,

$$\lim_{B\to\infty} Var\left(\frac{1}{B}\sum_{b=1}^{B}\hat{Y}_b\right) = \rho\sigma^2. \tag{6}$$

The variance approaches $\rho\sigma^2$, showing that a larger ensemble effectively reduces prediction variance as long as trees remain decorrelated (Hastie et al., 2009, p. 589).

The procedure, as detailed in Algorithm 4.1, aims to enhance variance reduction, a key aspect of bagging, by strategically reducing the correlation among trees. This goal is achieved by randomly selecting features during the tree-growing process, which prevents excessive increases in variance. As suggested by the expression (6), selecting features accordingly is more effective in reducing both the bias and variance of the regression function than merely increasing the number of trees. This approach not only optimizes the model's accuracy but also ensures greater generalizability. According to Hastie et al., the number of randomly selected features at each node is often chosen as $p/3$ (Hastie et al. p 589).

In regression tasks, each tree partitions the feature space into regions R_m^b with a constant value c_m^b assigned to each region. Aggregating over all bootstrap samples yields the Random Forest regression predictor

$$\hat{f}_{rf}(x) = \frac{1}{B}\sum_{b=1}^{B}\sum_{m=1}^{M^b} c_m^b \mathbb{I}(x \in R_m^b), \tag{7}$$

where M^b denotes the number of regions of the b:th resampled tree, with regions R_m^b and levels c_m^b within these regions. This equation illustrates how each tree contributes to the final prediction $\hat{f}_{rf}(x)$, offering a more robust prediction than any individual tree alone.

One advantage of Random Forests is their ability to provide feature importance metrics. By measuring the impact of each feature on prediction accuracy. Random Forests offer insights into the most influential variables, aiding interpretability. This characteristic makes Random Forests particularly useful in fields where understanding variable contribution is essential. Random Forests efficiently handle large datasets with complex variable interactions, offering accurate predictions without overfitting. This method balances bias and variance effectively, leveraging ensemble learning to deliver high predictive power across a range of applications (Breiman, 2001).

2.5 Boosting

Boosting is a powerful ensemble learning method that combines the predictions of multiple weak learners to create a strong learner. These weak learners, often decision trees, only slightly outperform random guessing but are highly effective when combined iteratively. Boosting derives its strength from sequential training, where each learner is fitted to correct the residual errors of its predecessors. By prioritizing instances with higher residuals, boosting systematically reduces error and enhances predictive accuracy. Unlike bagging, where trees are trained independently and combined to reduce variance, boosting introduces strong interdependence among learners. This interdependence allows boosting to iteratively reduce bias by focusing on improving the fit for harder-to-predict observations. As a result, boosting adapts the ensemble to minimize residual error and provides significant advantages, particularly in regression tasks where precise predictions are critical (Hastie et al., 2009, chapter 10; Rocca, 2019).

2.5.1 Forward Stagewise Additive Modeling

Boosting can be understood as an iterative method for fitting an additive expansion of form:

$$f(x) = \sum_{b=1}^{B} \beta_b z(x; \theta_b),$$

where β_b are the weights assigned to the individual basis functions $z(x; \theta_b)$, parameterized by θ_b. In the context of regression boosting, $z(x; \theta_b)$ often corresponds to weak learners, such as regression trees, and β_b represents their contribution to the ensemble.

Forward stagewise additive modeling builds models sequentially. Instead of solving for all coefficients and parameters simultaneously, this method introduces one basis function at a time, ensuring that each weak learner incrementally corrects the residual error of the current model. Once a weak learner is added, it remains unchanged throughout the modeling process. At each iteration b, the algorithm minimizes the loss function L by solving the optimization problem:

$$(\beta_b, \theta_b) = \arg\min_{\beta,\theta} \sum_{i=1}^{N_0} L(y_i, f_{b-1}(x_i) + \beta z(x; \theta)),$$

where $f_{b-1}(x)$ is the model built up to the previous iteration $b-1$, $z(x; \theta_b)$ represents a new weak learner (regression tree) and N_0 is the number of training observations.

To simplify the minimization of the loss function L, consider the square error loss $L(y, f(x)) = (y - f(x))^2$. The residuals at iteration b are defined as:

$$r_{ib} = y_i - f_{b-1}(x_i),$$

representing the difference between the observed values and the model predictions up to the previous step. Solve the for y_i and substitute into the loss function, the optimization problem can be written as:

$$(\beta_b, \theta_b) = \arg\min_{\beta,\theta} \sum_{i=1}^{N_0} \left(r_{ib} - \beta z(x_i; \theta)\right)^2$$

This reformulation focuses on fitting the residuals with the new basis function $z(x; \theta_b)$, weighted by β_b.

Once β_b and θ_b are determined the model is updated iteratively as:

$$f_b(x) = f_{b-1}(x) + \beta_b z(x; \theta_b).$$

Each step progressively reduces the error by targeting the residuals while keeping the previously selected terms unchanged.

2.5.2 Boosted Trees

Suppose the basis function in each step of boosting is a regression tree. The prediction of a single regression tree of boosting algorithm can be represented as:

$$T(x;\theta) = \sum_{m=1}^{M} c_m \mathbb{1}(x \in R_m), \tag{8}$$

where the parameters $\theta = \{R_m, c_m\}_{m=1}^{M}$ defines the structure of the tree. In this equation, R_m represents disjoint regions of the feature space and c_m is the predicted value associated with region R_m. The indicator function $\mathbb{1}(x \in R_m)$ equals 1 if $x \in R_m$ and 0 otherwise.

To train the single regression tree (8), we need to find the optimal parameters $\theta = \{R_m, c_m\}_{m=1}^{M}$ by solving the optimization problem:

$$\hat{\theta} = \arg\min_{\theta} \sum_{m=1}^{M} \sum_{x_i \in R_m} L(y_i, c_m), \tag{9}$$

where $L(y_i, c_m)$ denotes the loss incurred when predicting y_i in region R_m with value c_m. In this thesis, we use the squared loss $L(y_i, c_m) = (y_i - c_m)^2$.

The optimization problem in (9) is a complex combinatorial task due to the need to determine both regions R_m and the value c_m. The optimization problem is typically solved using a greedy recursive partitioning algorithm, as discussed in section 2.2.1. This algorithm sequentially partitions the feature space into regions R_m by minimizing the squared loss at each step.

In boosting, each tree $T(x;\theta_b)$ is trained sequentially to improve the predictions of the previous trees. As mentioned in section 2.5.1, this is achieved by focusing on the residuals of the current model and fitting a new tree to minimize these residuals. For each tree, the parameters $\theta_b = \{R_{mb}, c_{mb}\}$ are found by solving the following optimization problem:

$$\hat{\theta}_b = \arg\min_{\theta_b} \sum_{i=1}^{N_0} L(y_i, f_{b-1}(x_i) + T(x_i;\theta_b)), \tag{10}$$

where f_{b-1} is the prediction from the first $b-1$ trees (the current model) and $T(x_i;\theta_b)$ is the contribution of the bth tree. After determining the region R_{mb} for the bth tree, the predicted value for each region is optimized using:

$$\hat{c}_{mb} = \arg\min_{c_{mb}} \sum_{x_i \in R_{mb}} L(y_i, f_{b-1}(x_i) + c_{mb}), \tag{11}$$

where $\hat{c}_{mb}$ is the optimal constant prediction for the mth region of the bth tree. For squared error loss, $\hat{c}_{mb}$ is simply the mean of the resiudals in R_{mb}, which makes the optimization computationally efficient. Over multiple iterations, boosting builds a strong predictive model by combining many weak learners into a single ensemble.

2.5.3 Gradient Boosting

Gradient Boosting is a specialized implementation of the boosting framework that integrates gradient descent into its optimization process. This algorithm builds models sequentially, with each regression tree trained to minimize the gradient of the chosen loss function with respect to the current ensemble's predictions. The stagewise process aligns with functional gradient descent, ensuring incremental corrections to residual errors at each iteration (Friedman, 2001; Hastie et al., 2009, p. 359–361).

The regression function of a boosted model is given by:

$$f_B(x) = \sum_{b=1}^{B} T(x;\theta_b), \tag{12}$$

where $f_B(x)$ is the ensemble prediction after B boosting iterations and $T(x;\theta_b)$ is the b-th regression tree parametrized by θ_b. As in section 2.5.2, each tree is trained on the residuals of the previous ensemble, enabling the model to address the most challenging errors at each stage (Hastie et al., 2009, p. 359–361).

A core concept in Gradient Boosting is the use of the negative gradient of the loss function to approximate the residual errors. For a chosen loss function $L(y, f(x))$, the gradient at each data point provides the direction and magnitude of adjustment required to improve the model's predictions:

$$g_{ib} = \frac{\partial L(y_i, f_{b-1}(x_i))}{\partial f_{b-1}(x_i)}.$$

For regression tasks with squared error loss is defined as:

$$L(y, f(x)) = \frac{1}{2}\sum_{i=1}^{N_0}(y_i - f(x_i))^2.$$

The negative gradient is equivalent to the residual:

$$-g_{ib} = y_i - f_{b-1}(x_i) = r_{ib}.$$

This relationship highlights how the negative gradient and residuals are closely connected, with the gradient providing the necessary adjustments to reduce the loss.

The goal of each boosting iteration is to fit a new tree $T(x;\theta_b)$ to approximate the negative gradient values $-g_{ib}$. By doing so, the model incrementally refines its prediction to reduce the residual errors.

At each boosting stage, the optimization problem is formulated as:

$$\hat{\theta}_b = \arg\min_{\theta_b} \sum_{i=1}^{N_0}(-g_{ib} - T(x_i;\theta))^2, \tag{13}$$

This equation minimizes the squared difference between the negative gardient g_{ib} and the predictions of the new tree $T(x;\theta)$, effectively fitting a regression tree to negative gradient values. This step uses least squares regression, which is computationally efficient and straightforward to implement.

After the regression tree $T(x;\theta)$ is constructed, the constant prediction c_{bm} for each region R_{bm} are optimized to further reduce residuals. The constants c_{bm} are determined by solving:

$$\hat{c}_{mb} = \arg\min_{c_{mb}} \sum_{x_i \in R_{mb}} L(y_i, f_{b-1}(x_i) + c_{mb}), \tag{14}$$

where R_{mb} denotes the mth region of the bth tree. For squared error loss, this simplifies to assigning the mean of the residuals within each region:

$$\hat{c}_{mb} = \frac{\sum_{x_i \in R_{mb}}(y_i - f_{b-1}(x_i))}{|R_{mb}|}.$$

The updated model after incorporating the bth tree is given by:

$$f_b(x) = f_{b-1}(x) + \nu T(x;\theta_b), \tag{15}$$

where ν is the learning rate with a value between 0 and 1, a hyperparameter controlling the step size for the gradient updates. Smaller values of ν promote slower but more robust convergence, while larger values can accelerate convergence but risk overfitting (Friedman, 2001, p. 1203-1204).

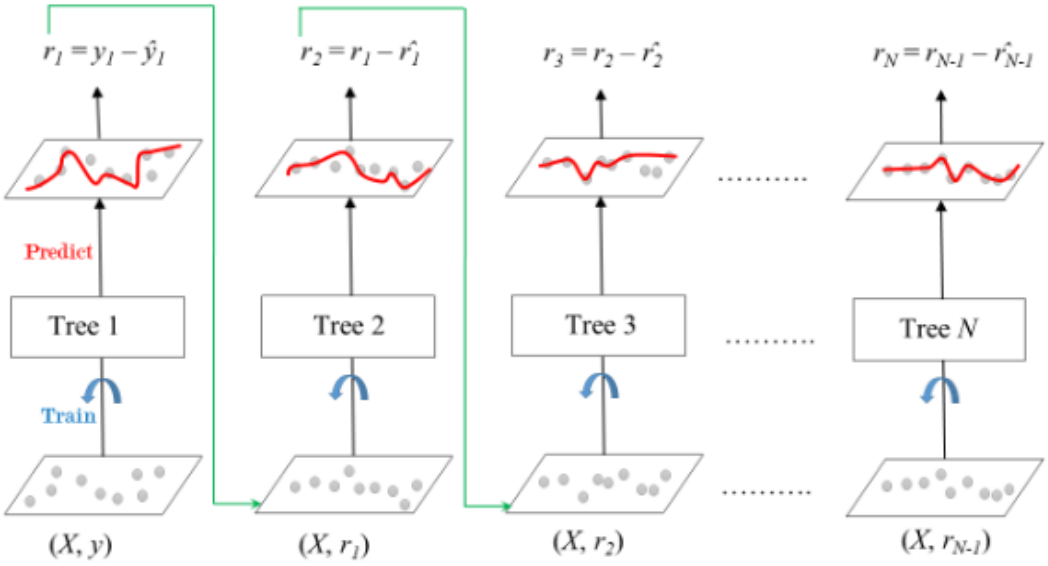


Figure 4: Schematic illustration of Gradient Boosting, (GeeksForGeeks, 2020).

2.5.4 eXtreme Gradient Boosting (XGBoost)

The theoretical framework and formulation discussed will primarily follow Guestrin (2016), the original developers of XGBoost, as outlined in the paper XGBoost: A Scalable Tree Boosting System.

The objective function of XGBoost, which models f as a sum of B regression trees, is expressed as:

$$\mathcal{L} = \sum_{i=1}^{N_0} L(y_i, f(x_i)) + \sum_{b=1}^{B} \Omega(M_b), \tag{16}$$

where $L(y_i, f(x_i))$ denotes the loss function. The term $\Omega(M_b)$ is a regularization term, designed to prevent overfitting by penalizing the complexity of the model. This regularization term is defined as:

$$\Omega(M_b) = \gamma M_b + \frac{1}{2}\lambda \sum_{m=1}^{M_b} c_{mb}^2. \tag{17}$$

where γ penalizes the number of leaves M_b in the tree, and λ discourages large leaf weights c_{mb}. By including this term, XGBoost achieves simpler, more generalizable models compared to traditional gradient boosting. To optimize the objective function, XGBoost uses a Second Order Taylor Approximation, which efficiently estimates the loss function around a current prediction:

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2. \tag{18}$$

Here, $f(a)$ is the current prediction, $f'(a)$ is the gradient, and $f''(a)$ is the Hessian. In the context of boosting, a represents the prediction at step $b-1$, while $x-a$ corresponds to the new tree $T(x;\theta)$ being added in step b to minimize the objective function. For this approximation to work, the loss function L must be twice differentiable, which is true for squared loss. At each iteration b, the gradient (first derivative) and Hessian (second derivative) are calculated as:

Gradient:

$$g_b(x_i) = \frac{\partial L(y_i, f_{b-1}(x_i))}{\partial f_{b-1}(x_i)} \tag{19}$$

Hessian:

$$h_b(x_i) = \frac{\partial^2 L(y_i, f_{b-1}(x_i))}{\partial^2 f_{b-1}(x_i)} \tag{20}$$

These derivatives quantify the steepest descent direction and curvature of the loss function, guiding model updates. The updated loss function after adding the new tree $T(x;\theta)$ can then be approximated as:

$$\sum_{i=1}^{N_0} L(y_i, f_{b-1}(x_i) + T_b(x_i;\theta)) \approx C + \sum_{i=1}^{N_0} \Big(g_b(x_i) T_b(x_i;\theta) + \frac{1}{2} h_b(x_i) T_b(x_i;\theta)^2 \Big) + \Omega(M_b). \tag{21}$$

The approximated loss now includes the gradient $g_b(x_i)$, Hessian $h_b(x_i)$, and the regularization term $\Omega(M_b)$. The term $C = \sum_{i=1}^{N_0} L(y_i, f_{b-1}(x_i))$ is a constant and it has no impact on finding the optimal tree $T(\cdot;\theta)$ in step b of the boosting algorithm. To compute $T_b(x_i;\theta)$, XGBoost represents it as a sum over disjoint regions R_m of the regression tree:

$$T_b(x_i;\theta) = \sum_{m=1}^{M} c_{mb} \mathbb{1}(x \in R_{mb}), \tag{22}$$

where $\mathbb{1}(x \in R_m)$ is an indicator function that equals 1 if $x \in R_m$ and 0 otherwise (see section 2.2.1). Substituting this representation into the approximated loss yields:

$$\sum_{m=1}^{M_b} \Big[\Big(\sum_{x_i \in R_{mb}} g_b(x_i) \Big) c_{mb} + \frac{1}{2} \Big(\sum_{x_i \in R_{mb}} h_b(x_i) \Big) c_{mb}^2 \Big] + \Omega(M_b). \tag{23}$$

Here, the terms $G_{mb} = \sum_{x_i \in R_{mb}} g_b(x_i)$ and $H_{mb} = \sum_{x_i \in R_{mb}} h_b(x_i)$ are introduced to simplify notation. Incorporating the regularization term, the loss function becomes:

$$\begin{aligned} &\sum_{m=1}^{M_b} \Big[G_{mb}\, c_{mb} + \frac{1}{2} H_{mb}\, c_{mb}^2 \Big] + \gamma M_b + \frac{1}{2}\lambda \sum_{m=1}^{M_b} c_{mb}^2 = \\ &= \sum_{m=1}^{M_b} \Big[G_{mb}\, c_{mb} + \frac{1}{2} \Big(H_{mb} + \lambda \Big) c_{mb}^2 \Big] + \gamma M_b. \end{aligned} \tag{24}$$

To find the optimal leaf weights c_{mb}^*, we differentiate (24) with respect to c_{mb} and set the derivative to zero, according to

$$\frac{\partial}{\partial c_{mb}} \Big[G_{mb}\, c_{mb} + \frac{1}{2} \Big(H_{mb} + \lambda \Big) c_{mb}^2 \Big] = G_{mb} + \Big(H_{mb} + \lambda \Big) c_{mb} = 0.$$

Solving for optimal leaf weight yields (with $g_{ib} = g_b(x_i)$ and $h_{ib} = h_b(x_i)$):

$$c_{mb}^* = \frac{-G_{mb}}{H_{mb} + \lambda} = -\frac{\sum_{i \in R_{mb}} g_{ib}}{\sum_{i \in R_{mb}} h_{ib} + \lambda}. \tag{25}$$

This formula determines the optimal value of each leaf, balancing the gradient and Hessian information while penalizing large leaf weights through λ. We can also express (25) as

$$\text{Optimal leaf value} = \frac{\text{Sum of Residuals}}{\text{Number of Residuals} + \lambda}.$$

Plugging (25) into (24), we get:

$$\Rightarrow -\frac{1}{2} \sum_{m=1}^{M_b} \frac{G_{mb}^2}{H_{mb} + \lambda} + \gamma M_b = -\frac{1}{2} \sum_{m=1}^{M_b} \frac{\big(\sum_{i \in R_{mb}} g_{ib} \big)^2}{\sum_{i \in R_{mb}} h_{ib} + \lambda} + \gamma M_b,$$

where a higher similarity score indicates that the observations within the node are more homogeneous with respect to the target variable. This metric helps prioritize splits that improve leaf homogeneity.

To evaluate splits of tree T_b, the gain is computed as:

$$\mathcal{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in R_L} g_{ib})^2}{\sum_{i\in R_L} h_{ib}+\lambda} + \frac{(\sum_{i\in R_R} g_{ib})^2}{\sum_{i\in R_R} h_{ib}+\lambda} - \frac{(\sum_{i\in R} g_{ib})^2}{\sum_{i\in R} h_{ib}+\lambda}\right] - \gamma, \tag{26}$$

where R_L, R_R, and R represent the left, right, and parent nodes, respectively. Higher gain indicates better splits, guiding tree growth.

3 Simulation Study

3.1 Data Generation

In this section, we describe how to generate 100 datasets (numbered $j = 1, \ldots, 100$) of size $N = 1000$, from the model of Section 2.1. Each of these datasets is split into a training data set with $N_0 = 0.8N$ observations and a test dataset with $N_1 = 0.2N$ observations. That is, for each j we generate

$$\{(X_i^j, Y_i^j)\}_{i=1}^N, \quad Y_i^j = f(X_i^j) + \varepsilon_i^j. \tag{27}$$

The observation number i ranges over $i = 1, \ldots, N$, where $N_0 + N_1 = N$. There are $p = 5$ predictors and the error terms ε_i^j are assumed to be independent and normally distributed with mean 0 and variance σ^2. Thus for each dataset j we generate the following:

Training data: $Y_i^j = f(X_{i1}^j, \ldots, X_{i5}^j) + \varepsilon_i^j, \quad i = 1, \ldots, N_0.$

Test data: $Y_i^j = f(X_{i1}^j, \ldots, X_{i5}^j) + \varepsilon_i^j, \quad i = N_0 + 1, \ldots, N_0 + N_1.$

This process is to be repeated 100 times, indicated by $j = 1, \ldots, 100$. For each dataset j we obtain an estimate $\hat{f}^j$ of the regression function f from the training data.

In this thesis, we will analyze a single dataset through two distinct machine learning methods: Random Forest and Gradient Boosting. This dataset will be generated with error terms from a normal distribution. The aim is to conduct simulations for each method 100 times to ensure statistical significance. The experimental setup includes eight scenarios, defined by four unique regression functions (referred to as sphere-truncated cone, cone, rhombus-truncated pyramid, and ridge), and for each regression function, we consider two different noise levels, $\sigma_{low} = 0.2$ and $\sigma_{high} = 1$.

3.2 Regression Functions

In more detail the four regression functions are defined, with different nonlinearities, as follows:

$$f(x) = \begin{cases} |x|\,\mathbb{I}(|x| \le \sqrt{5}), & \text{sphere-truncated cone,} \\ |x|, & \text{cone,} \\ |x|_1\,\mathbf{I}(|x|_1 \le 1), & \text{rhombus-truncated pyramid,} \\ \sqrt{x_1^2 + x_2^2}, & \text{ridge,} \end{cases}$$

where $|x| = (\sum_{k=1}^5 x_k^2)^{1/2}$ is the Euclidean norm of x, whereas $|x|_1 = \sum_{k=1}^5 |x_k|$ is the L_1-norm. The predictor variables are generated as independent random vectors X_i^j from a five-dimensional normal distribution with expected value $(0, 0, 0, 0, 0)$ and a covariance matrix that equals the identity matrix of order 5.

Selected shapes
The selected shapes for this analysis, sphere-truncated cone, cone, rhombus-truncated pyramid, and ridge, were chosen to evaluate the performance of Random Forest and Gradient Boosting models across a variety of patterns and challenges in the data. Each shape offers unique characteristics that test

specific aspects of the models' behavior, enabling a comprehensive comparison of their strengths and weaknesses.

Sphere-Truncated Cone $|x|\,\mathbb{I}(|x| \leq \sqrt{5})$: This shape is limited to specific range, with sharp edges at its boundary ($\sqrt{5}$). It is useful test case for seeing how well the algorithms can handle boundaries and make predictions near the edges.

Cone $|x|$: This is a simple, unbounded linear function. Its straightforward nature makes it a good starting point to understand how the algorithms perform with basic relationships. Since it has no sharp edges or limits, it allows us to see how well the models handle simple, continuous patterns without extra complications.

Rhombus-Truncated Pyramid $|x|_1\ \mathbf{I}(|x|_1 \leq 1)$: This shape combines the steady increase of the cone with the clear boundaries of the sphere-truncated cone. It behaves differently in different direction and has well defined edges. Studying this function shows how well the algorithms handle patterns that include both gradual changes and sharp limits, while also dealing with varying behavior along different directions.

Ridge $\sqrt{x_1^2 + x_2^2}$: This shape has a circular symmetry and a smooth surface that gradually changes as the distance from the origin increases. The function's nonlinear nature makes it a useful example for studying how well models can approximate smooth transitions and capture patterns that change continuously in all directions.

4 Algorithms and Performance Metrics

4.1 Algorithm for Random Forest Regression

In this section we define the Random Forest algorithm for each simulation $j = 1, \ldots, 100$. For simplicity of notation subscript j is omitted.

1. Initialization:

- Input: Training data $\{X_i, Y_i\}_{i=1}^{N_0}$, number of trees $B = 500$ (default), minimum node size $n_{min} = 5$ (default) , number of predictors $p = 5$, low standard deviation for error term $\sigma_{low} = 0.2$, high standard deviation for the error term $\sigma_{high} = 1$.
- Output: Ensemble of regression trees $\{\hat{f}_b\}_{b=1}^{B}$

2. Algorithm For $b = 1$ to B do:

(a) Bootstrap Sampling: The process begins by drawing a bootstrap sample of size N_0 from the training data. A bootstrap sample is a subset of the training data, selected with replacement.

(b) Tree Building: Each bootstrap sample is then used to grow a decision tree. This is done by recursively splitting the nodes of the tree based on certain criteria until a specified minimum node size $n_{min} = 5$ is reached.

(i) Feature Selection: At each node of the tree, one variable is selected at random from the total of 5 variables available.

(ii) Best Split Selection: For the selected variable, the algorithm identifies the exact split point that minimizes squared loss.

(iii) Node Splitting: Once the splitting variable and point are determined, the node is split into two daughter nodes, and this process repeats for each of the new nodes until the stopping criterion is reached.

(c) Once (b) is completed, a regression tree $\hat{f}_b$ is stored, that includes the regions of all terminal nodes and the average outcomes within these regions. All bootstrapped trees have been stored $\{\hat{f}_b\}_{b=1}^{B}$.

The estimated regression function is

$$\hat{f}_{\mathrm{rf}}^{B}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}_b(x).$$

4.2 Algorithm for Gradient Boosting

In this section we define the Gradient Boosting algorithm for each simulation $j = 1, \ldots, 100$. For simplicity of notation subscript j is omitted.

Gradient Boosting Algorithm

1. **Initialize:**

   Input: Training data $\{X_i, Y_i\}_{i=1}^{N_0}$ and a differentiable loss function $L(y_i, c) = \frac{1}{2}(y_i - c)^2$. Squared loss penalizes large deviations between prediction and true values.

   $$f_0(x) = \arg\min_c \sum_{i=1}^{N_0} L(y_i, c)$$

   $f_0(x)$ initializes the model to the constant value that minimizes the loss over the entire dataset. This gives $f_0(x) = c = \frac{1}{N_0} \sum_{i=1}^{N_0} y_i$. After solving the optimization problem above.

2. **Algorithm:**

   For $b = 1$ to B:

   (a) For $i = 1, 2, \ldots, N_0$, compute the pseudo-residuals:

   $$r_{ib} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{b-1}(x_i)}$$

   (b) Fit a regression tree to the pseudo residuals r_{ib} using the input feature x_i to split the data into M_b terminal regions R_{mb}, where $m = 1, 2, ..., M_b$. Each region R_{mb} groups data points that need similar changes.

   (c) For $m = 1, 2, \ldots, M_b$, compute the leaf value:

   $$c_{mb} = \arg\min_c \sum_{x_i \in R_{mb}} L(y_i, f_{b-1}(x_i) + c)$$

   (d) Update the model:

   $$f_b(x) = f_{b-1}(x) + \nu \sum_{m=1}^{M_b} c_{mb} I(x \in R_{mb})$$

   where ν is the learning rate and $I(\cdot)$ is the indicator function.

3. **Output:**

   $$\hat{f}(x) = \hat{f}_B(x)$$

4.3 Algorithm for eXtreme Gradient Boosting

In this section we define the eXtreme Gradient Boosting algorithm for each simulation $j = 1, \ldots, 100$. For simplicity of notation subscript j is omitted.

XGBoost Algorithm

Input: Training data $\{X_i, Y_i\}_{i=1}^{N_0}$, differentiable loss function $L(y_i, f(x_i))$, learning rate ν, and regularization parameters λ, γ.

1. **Initialize:**
$$f_0(x) = \arg\min_f \sum_{i=1}^{N_0} L(y_i, f).$$

2. **Iterate for $b = 1$ to B:**

   (a) Compute gradients and Hessians for all observations:
   $$g_{ib} = \frac{\partial L(y_i, f_{b-1}(x_i))}{\partial f_{b-1}(x_i)}, \quad h_{ib} = \frac{\partial^2 L(y_i, f_{b-1}(x_i))}{\partial^2 f_{b-1}(x_i)}.$$

   (b) Fit a tree by maximizing the gain for a number of splits until no improvement is possible:

      i. For each split of a region R in to R_L and R_R, compute:
      $$\text{Gain} = \frac{1}{2}\left[\frac{(\sum_{i\in R_L} g_{ib})^2}{\sum_{i\in R_L} h_{ib} + \lambda} + \frac{(\sum_{i\in R_R} g_{ib})^2}{\sum_{i\in R_R} h_{ib} + \lambda} - \frac{(\sum_{i\in R} g_{ib})^2}{\sum_{i\in R} h_{ib} + \lambda}\right] - \gamma.$$

      ii. Choose the split with the highest gain.

   (c) Compute the optimal leaf weights for each terminal region R_{mb}:
   $$c_{mb} = -\frac{\sum_{x_i\in R_{mb}} g_{ib}}{\sum_{x_i\in R_{mb}} h_{ib} + \lambda}.$$

   (d) Update the model:
   $$f_b(x) = f_{b-1}(x) + \nu \sum_{m=1}^{M_b} c_{mb} \mathbb{1}(x \in R_{mb}).$$

3. **Output:**
$$\hat{f}(x) = f_B.$$

4.4 Performance criteria

This section outlines two essential metrics for assessing regression models, the Mean Squared Error (MSE) and the Mean Squared Error of Prediction (MSEP). These metrics provide complementary perspectives on model performance, with MSE focusing on the estimation error and MSEP incorporating both estimation error and the noise variance inherent in the data.

4.4.1 Mean Squared Error of Estimation (MSE)

The Mean Squared Error (MSE) quantifies how well the model approximates the true regression function f by evaluating the squared differences between the predicted values $\hat{f}_j(X_i)$ and the true function values $f(X_i)$. Formally, for N_1 test data points, the MSE is defined as:

$$MSE_j = \frac{1}{N_1} \sum_{i=N_0+1}^{N_0+N_1} \left(\hat{f}_j(X_i^j) - f(X_i^j)\right)^2,$$

where:

- $\hat{f}_j(X_i^j)$ is the predicted value of $f(X_i^j)$ from simulated training dataset j,

- $f(X_i^j)$ is the true underlying regression function value at X_i^j,
- N_0 represents the number of training data points, and N_1 denotes the test data size.

The MSE isolates the estimation error by comparing predictions directly to the true function values. A lower MSE indicates that the model provides a more accurate approximation of f.

4.4.2 Mean Squared Error of Prediction (MSEP)

The Mean Squared Error of Prediction (MSEP) extends the MSE by incorporating both the estimation error and the inherent noise in test data. Given test outputs Y_i^j, which include random noise ε_i^j, the MSEP for simulation j is defined as:

$$MSEP_j = \frac{1}{N_1} \sum_{i=N_0+1}^{N_0+N_1} \left(Y_i^j - \hat{f}_j(X_i^j)\right)^2,$$

where $Y_i^j = f(X_i^j) + \varepsilon_i^j$ and $\varepsilon_i^j \sim N(0, \sigma^2)$. The prediction error can be decomposed into two components:

$$MSEP_j \approx MSE_j + \sigma^2,$$

where:

- MSE_j represents the estimation error,
- σ^2 is the variance of the random noise ε_i.

Thus, while MSE evaluates the model's ability to approximate the true function, the MSEP captures the overall prediction error, including the unavoidable contribution of noise from test data. This distinction highlights why MSEP is often larger than MSE.

By understanding both metrics, we gain a comprehensive perspective on model performance: MSE reveals how well the model estimates the signal, while MSEP reflects the practical accuracy of predictions under noisy conditions.

4.4.3 Sample Mean & Sample Standard Deviation:

When combining MSE_j and $MSEP_j$ from all simulations $j = 1, \ldots, J = 100$, we obtain the following summary measures of estimation and predction accuracy:

$$\text{MSE Sample Mean} = \widehat{MSE} = \sum_{j=1}^{J} \frac{MSE_j}{J}$$

$$\text{MSE Sample Standard Deviation} = \sqrt{\sum_{j=1}^{J} \frac{(MSE_j - \widehat{MSE})^2}{(J-1)}}$$

$$\text{MSEP Sample Mean} = \widehat{MSEP} = \sum_{j=1}^{J} \frac{MSEP_j}{J}$$

$$\text{MSEP Sample Standard Deviation} = \sqrt{\sum_{j=1}^{J} \frac{(MSEP_j - \widehat{MSEP})^2}{(J-1)}}$$

5 Results

In this section we present results of the simulation study. The simulation program was written in R Studio. It makes use of the R functions randomForest() and xgb.train() for Random Forest and XG Boost respectively.

Table 1: Random Forest

	MSE Sample Mean	MSE Sample SD	MSEP Sample Mean	MSEP Sample SD
Sphere-Truncated Cone Low Noise	0.3829	0.0533	0.4241	0.0575
Cone Low Noise	0.0692	0.0147	0.1088	0.0179
Rhombus Truncated Pyramid Low Noise	0.0950	0.0087	0.1347	0.0132
Ridge Low Noise	0.0829	0.0166	0.1221	0.0196
Sphere-Truncated Cone High Noise	0.3812	0.0526	1.3898	0.1415
Cone High Noise	0.0703	0.0122	1.0683	0.0957
Rhombus Truncated Pyramid High Noise	0.0948	0.0088	1.1013	0.1027
Ridge High Noise	0.0841	0.0153	1.0811	0.0978

Table 2: XG Boost

	MSE Sample Mean	MSE Sample SD	MSEP Sample Mean	MSEP Sample SD
Sphere-Truncated Cone Low Noise	0.4421	0.0525	0.4826	0.0553
Cone Low Noise	0.0323	0.0088	0.0718	0.0108
Rhombus Truncated Pyramid Low Noise	0.1021	0.0102	0.1422	0.0150
Ridge Low Noise	0.0066	0.0041	0.0458	0.0063
Sphere-Truncated Cone High Noise	0.4467	0.0580	1.4461	0.1473
Cone High Noise	0.0326	0.0074	1.0321	0.0948
Rhombus Truncated Pyramid High Noise	0.1026	0.0106	1.1071	0.1008
Ridge High Noise	0.0063	0.0033	1.0066	0.0936

Table 3: Difference Between XGBoost and Random Forest Metrics

	MSE Sample Mean	MSE Sample SD	MSEP Sample Mean	MSEP Sample SD
Sphere-Truncated Cone Low Noise	0.0592	-0.0008	0.0585	-0.0022
Cone Low Noise	-0.0369	-0.0059	-0.0370	-0.0029
Rhombus Truncated Pyramid Low Noise	0.0071	0.0015	0.0075	0.0018
Ridge Low Noise	-0.0763	-0.0133	-0.0763	-0.0133
Sphere-Truncated Cone High Noise	0.0655	0.0054	0.0563	0.0058
Cone High Noise	-0.0377	-0.0022	-0.0362	-0.0021
Rhombus Truncated Pyramid High Noise	0.0078	0.0062	0.0058	-0.0019
Ridge High Noise	-0.0778	-0.0120	-0.0745	-0.0042

Sphere-Truncated Cone: Datasets for this model demonstrate similar performance for Random Forest and XGBoost across low and high noise levels. Nonetheless, Random Forest consistently

achieves lower MSE and MSEP than XGBoost, outperforming it by approximately 0.06 units in each metric. Despite this, both models exhibit higher MSE and MSEP compared to other shapes, highlighting the increased difficulty of this complex geometry for both algorithms.

Cone: This model achieves one of the best overall performances, second only to the Ridge. In this setting, XGBoost significantly outperforms Random Forest, achieving half the MSE and substantially lower MSEP values. This trend is observed across low and high noise levels, where both methods maintain relatively consistent performance. The simpler geometry of the Cone appears to contribute to its superior results for both methods.

Rhombus Truncated Pyramid: Random Forest slightly outperforms XGBoost, with marginally lower MSE and MSEP values. Both models display similar performance under low and high noise levels, demonstrating stability. However, this dataset does not perform as well as the Cone and Ridge datasets, likely due to its increased geometric complexity.

Ridge: Datasets for this model exhibit the largest performance gap between the two models. XGBoost significantly outperforms Random Forest, particularly in low noise settings where its MSE is 0.0066 compared to Random Forest's 0.0829. This strong performance advantage is maintained across both noise levels, showcasing XGBoost's robustness and effectiveness in handling this simpler dataset shape.

5.1 Box plots

In this section we present Box plots for $\{MSE_j\}_{j=1}^J$ and $\{MSEP_j\}_{j=1}^J$ from $J = 100$ simulations. This is done for each combination of method (Random Forest or Gradient Boosting), the four regression models and the two noise levels. Altogether, this gives rise to eight figures with four Box plots each.

5.1.1 Sphere-truncated cone

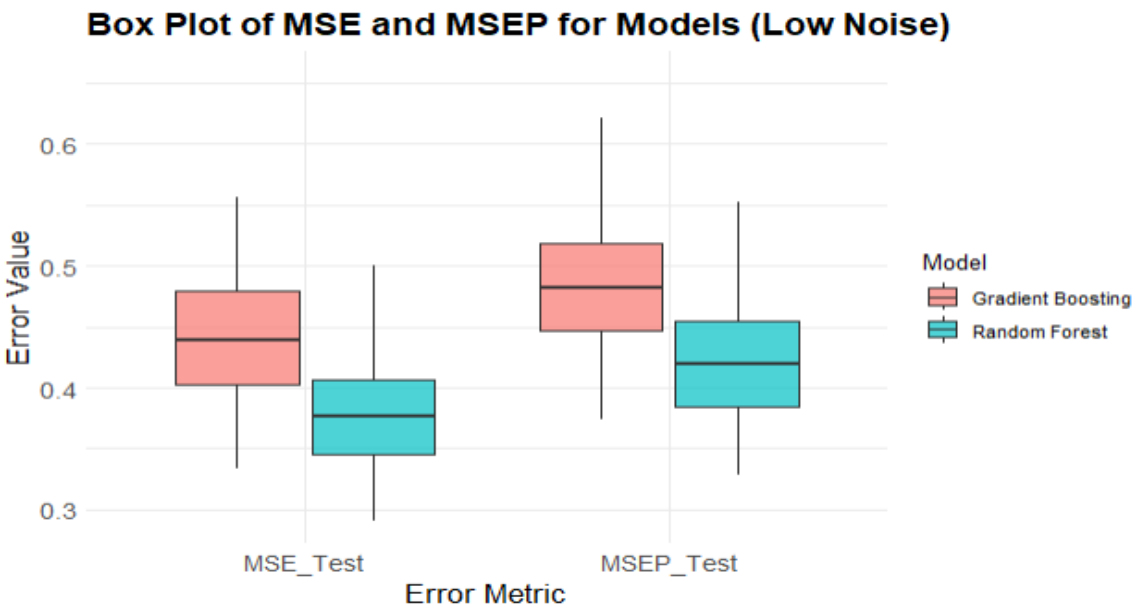


Figure 5: Box plots for sphere-truncated cone model with low noise level.

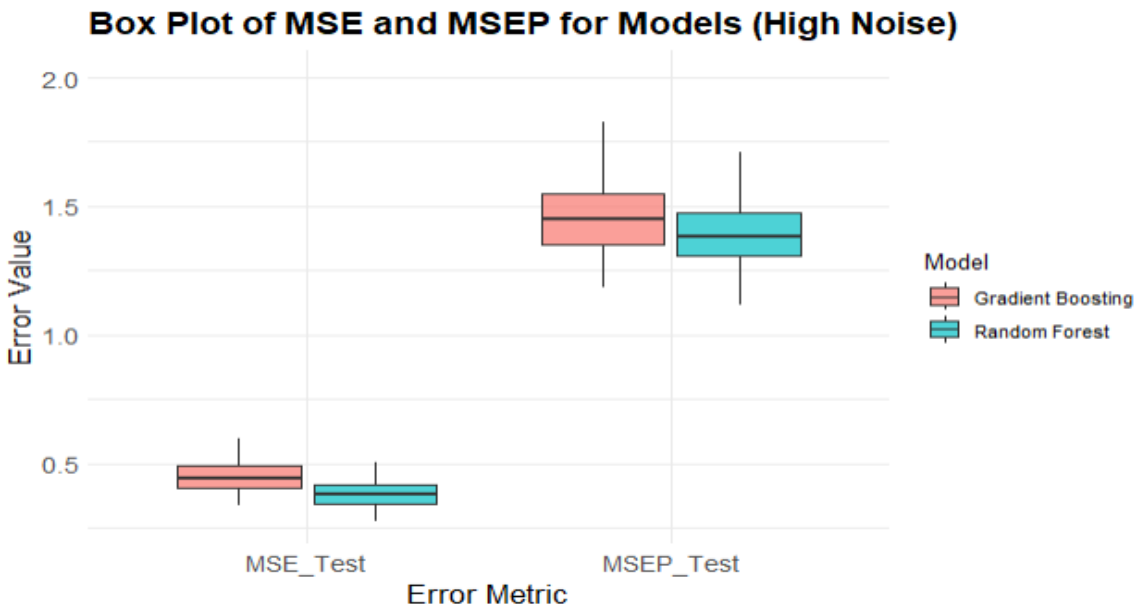


Figure 6: Box plots for sphere-truncated cone model with high noise level.

5.1.2 Cone

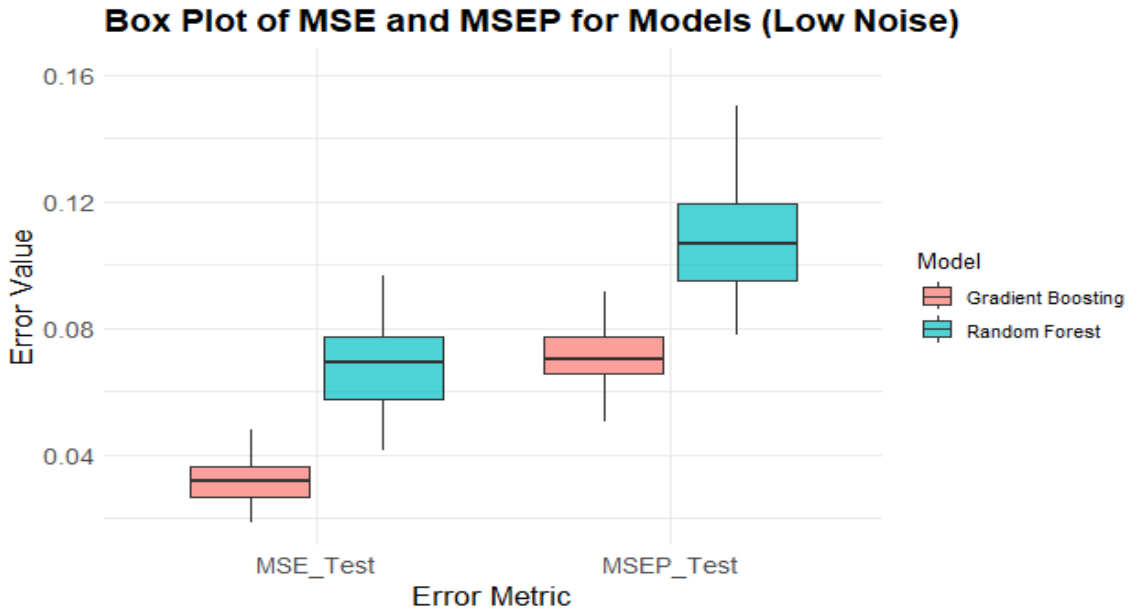


Figure 7: Box plots for cone model with low noise level.

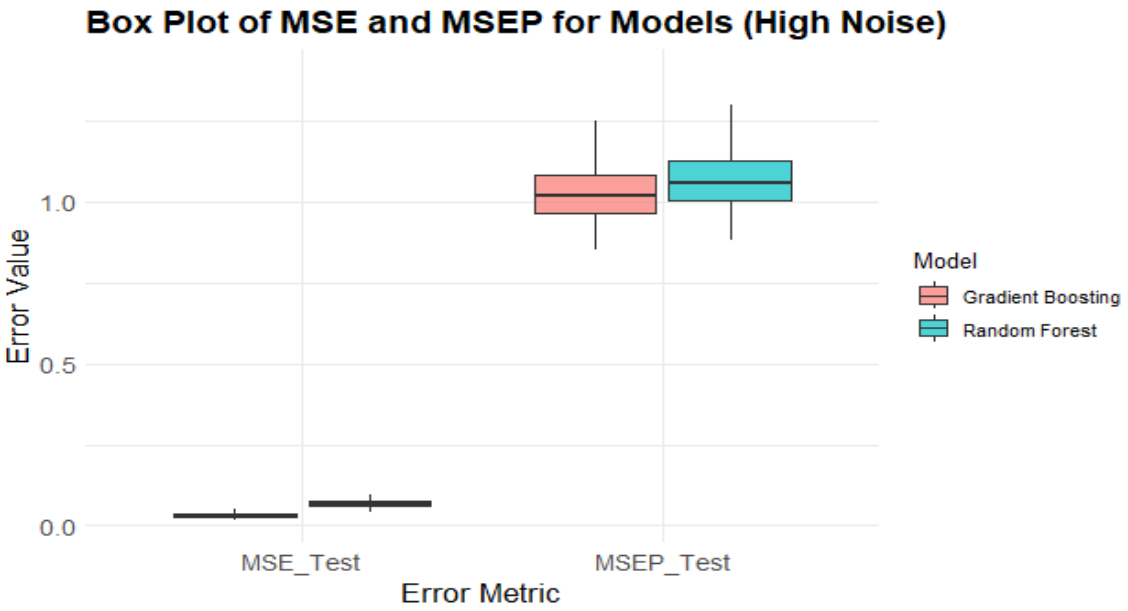


Figure 8: Box plots for cone model with high noise level.

5.1.3 Rhombus-truncated pyramid

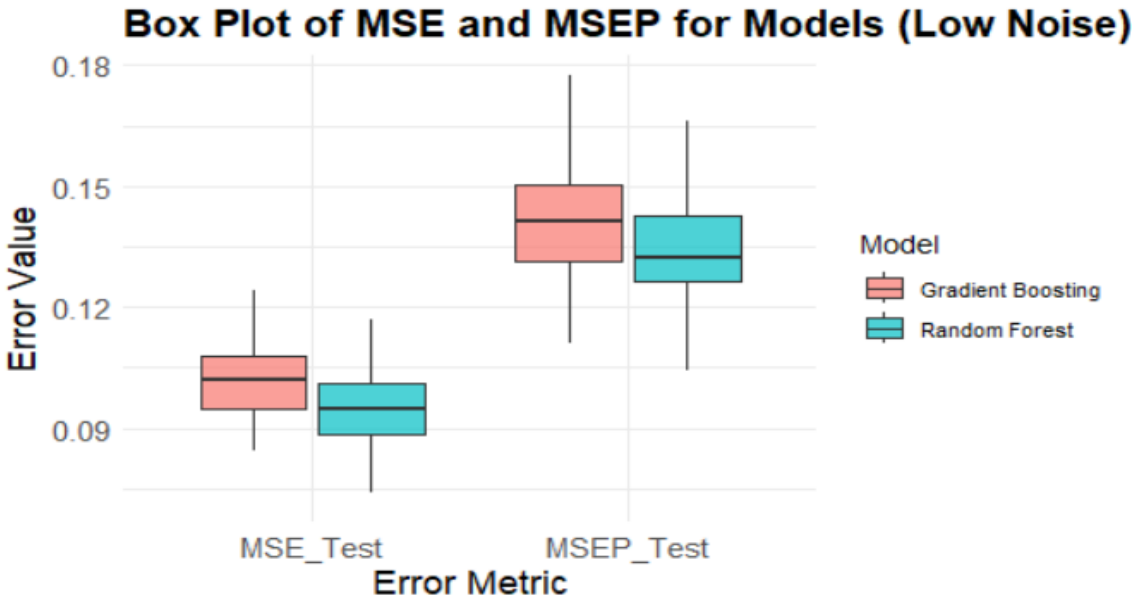


Figure 9: Box plots for rhombus-truncated pyramid model with low noise level.

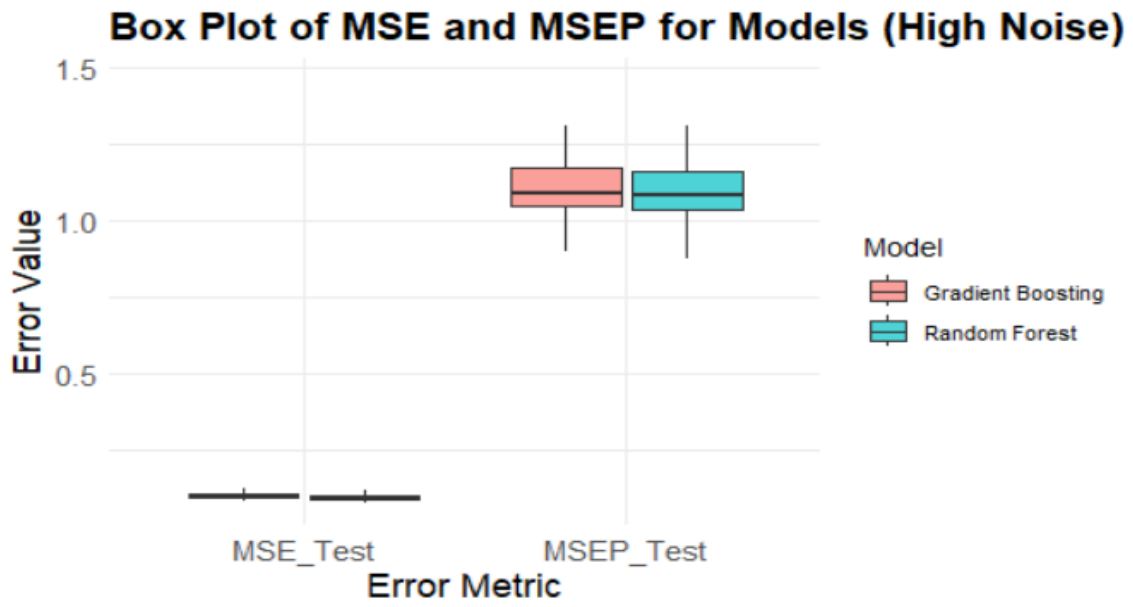


Figure 10: Box plots for rhombus-truncated pyramid model with high noise level.

5.1.4 Ridge

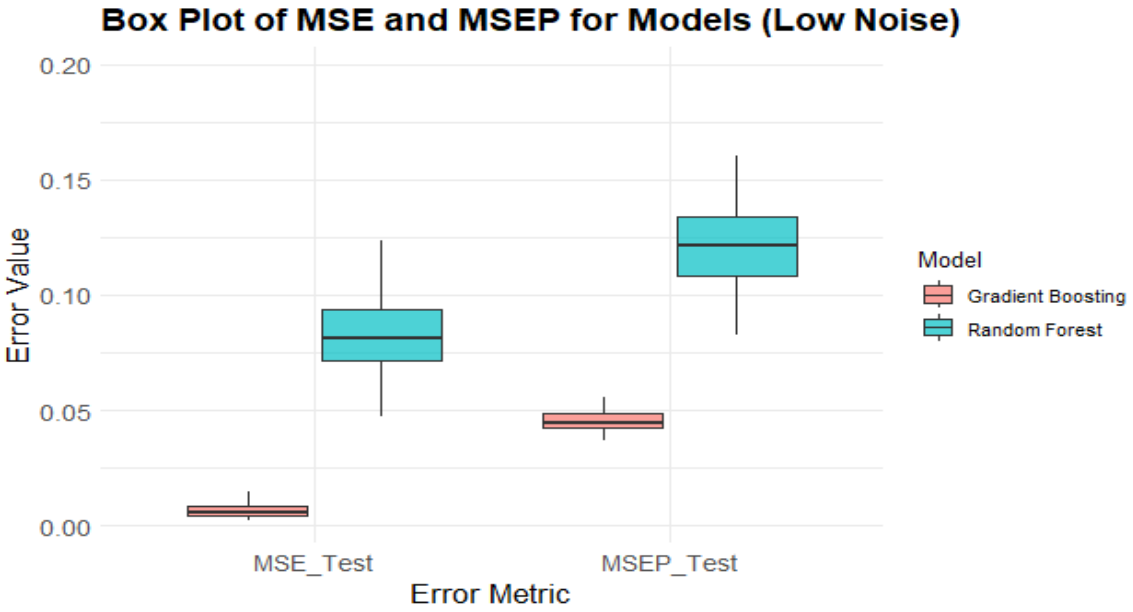


Figure 11: Box plots for ridge model with low noise level.

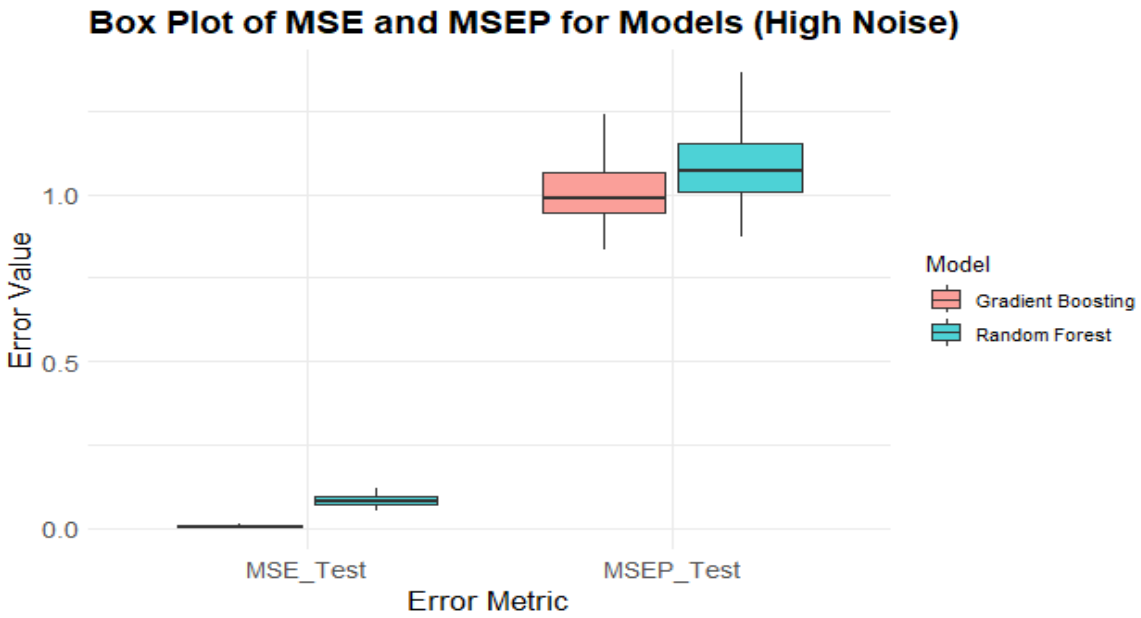


Figure 12: Box plots for ridge model with high noise level.

6 Discussion

This thesis investigates the comparative performance of Random Forest and Gradient Boosting regression across datasets with varying geometric complexities and noise levels. By constructing a number of different regression functions featuring distinct structures, such as Sphere-Truncated Cone, Cone, Rhombus-Truncated Pyramid, and Ridge, and systematically introducing noise directly into data, the study examines how these two methods respond to both discontinuities and smooth patterns in the data.

The results reveal that the performance of Random Forest and Gradient Boosting is strongly influenced by the continuity and complexity of the underlying regression functions. Random Forest demonstrates its strength in handling regression functions with discontinuities, such as the Sphere-Truncated Cone and Rhombus-Truncated Pyramid, where its robustness enables it to effectively manage abrupt changes and noise. In contrast, Gradient Boosting excels for continuous regression functions, such as the Cone and Ridge, where its iterative boosting mechanism effectively captures smooth patterns and intricate relationships.

These findings align with the theoretical strengths of each model and highlight their complementary nature. Random Forest's ensemble averaging makes it well-suited to datasets with discontinuities, while Gradient Boosting's sequential optimization allows it to achieve superior performance in smoother and more continuous regression settings.

References

[1] Breiman, Random Forests. *Machine Learning*, 45(1), 5-32, 2001.

[2] Breiman, et al. *Classification and Regression Trees*. Boca Raton Routledge Ann Arbor, Michigan Proquest, 2017.

[3] Chen and Guestrin, "XGBoost: A Scalable Tree Boosting System", 2016. https://arxiv.org/pdf/1603.02754.pdf.

[4] Fortmann-Roe, "Understanding the Bias-Variance Tradeoff", scott.fortmann-roe.com/docs/BiasVariance.html, June 2012.

[5] GeeksForGeeks. "ML - Gradient Boosting." *GeeksforGeeks*, 25 Aug. 2020, www.geeksforgeeks.org/ml-gradient-boosting/.

[6] Friedman, Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics*, Vol. 29, No. 5, pp. 1189-1232, Oct., 2001.

[7] Hastie, Trevor, Tibshirani, Robert, and Friedman, Jerome. *The Elements of Statistical Learning. Data Mining, Inference and Prediction*, 2nd ed., Springer, New York, 2009.

[8] James, et al. *An Introduction to Statistical Learning : With Applications in R*. New York, Springer, 2013.

[9] Kovačević, Miljan, et al. Construction Cost Estimation of Reinforced and Prestressed Concrete Bridges Using Machine Learning. *Građevinar*, vol. 73, no. 01, 10 Feb. 2021, pp. 1–13. https://doi.org/10.14256/jce.2738.2019.

[10] "Lecture 12: Bias Variance Tradeoff." www.cs.cornell.edu, www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote12.html. - ("Lecture 12: Bias Variance Tradeoff") 2012.

[11] Müller and Guido. *Introduction to Machine Learning with Python : A Guide for Data Scientists*. Beijing, O'reilly, 2017.

[12] Rocca. Ensemble Methods: Bagging, Boosting and Stacking Understanding the Key Concepts of Ensemble Learning. 23 Apr. 2019.