Performance Evaluations on AI Regression and Classification Algorithms Using Ensemble Methods

# Performance Evaluations on AI Regression and Classification Algorithms Using Ensemble Methods


1st Aishat Ganiyu
*School of Eng., Phys. and Math. Sc.*
*Royal Holloway University of London*
London, United Kingdom
aishat.ganiyu.2021@live.rhul.ac.uk

2nd Iman Darvishi
School of Computing and Engineering
University of West London
London, United Kingdom
iman.darvishi@uwl.ac.uk

4th Ronald Addo-Quaye
*School of Business and Law*
*Central Queensland University*
Brisbane, Australia
r.addoquaye@cqu.edu.au

1st Abel Yeboah-Ofori
School of Computing and Engineering
University of West London
London, United Kingdom
abel.yeboah-ofori@uwl.ac.uk

3rd Bismark Tei Asare
*School of Arts, Hum. and Social Sci.*
*University of Roehampton*
London, United Kingdom
bismark.Asare@roehampton.ac.uk

5th Oluwole Oguntoyinbo
School of Computing and Engineering
University of West London
London, United Kingdom
21516296@student.uwl.ac.uk



***Abstract*— Machine Learning techniques are the backbone of successful Artificial Intelligence (AI) applications as they empower AI systems to make quality predictions and provide valuable insights from data that will aid decision-making within various industries. Ensemble method is a machine learning technique that helps to determine the suitable model for a dataset while limiting bias and variance; it is a technique used to obtain predictions by conducting maximum votes or averaging. This paper explores various Ensemble methods such as Bagging/Bootstrap Aggregation, Random Forest, and Boosting alongside their underlying algorithm Decision Tree for Regression and Classification cases. The contribution of this paper is threefold. The foremost objective is to perform a theoretical analysis of various Ensemble methods with sample pseudocodes to aid the implementation process. Secondly, we implement Ensemble method algorithms using basic Python packages such as numpy, pandas, math, matplotlib and dataset installation packages for performance evaluations. Finally, we applied the developed code to real-world datasets across various industries, including healthcare, manufacturing, and real estate. The results highlight the different parameters and their performance on the algorithms while observing the proposed Ensemble program for performance evaluation.**



***Keywords*— *Artificial Intelligence, Bagging, Bootstrapping, Boosting, Classification, Decision Trees, Ensemble Methods, Machine Learning, Random Forest, Regression.***


## I. Introduction

The issue of determining a suitable model for a dataset while limiting bias and variance in classification and regression cases has become imperative. Ensemble methods are essential for making accurate predictions on various datasets. They derive multiple models and combine their outputs to obtain predictions [1] [2]. This can be achieved by conducting majority votes for classification problems or by computing the average of the results for regression problems. For instance, [3] compared various classification algorithms using Majority Voting to determine the performance accuracies. [4] [5] applied Ensemble methods such as Boosting and Random Forest (RF) to conduct performance accuracy on different datasets. The ensemble method is used to create an improved Classification model, $M*$ by combining a series of k-learned models, $M_1$, $M_2$, …., $M_k$, which are often referred to as base classifiers. [1]. For instance, given dataset D, we can create a k training set, i.e., $D_1$, $D_2$, … $D_k$. $D_i \in (1 \leq I \leq k\text{-}1)$ to generate classifier $M_i$. Further, we can classify a new data tuple by obtaining the vote from base classifier predictions to determine the ensemble class prediction, which is more accurate than the base classifiers [1]. Suppose a new tuple X needs to be classified; the class label prediction will be collected from each base classifier, and the majority will be returned. Ensembles yield a better result when significant diversity exists among the models [1]. Ensemble methods help to reduce bias or variance and identify a suitable model for datasets. The uncertainty of choosing an appropriate model is a challenge that can be resolved by obtaining multiple models and performing computations, to derive the right model. Addressing the challenges in Ensemble methods involves creating different models by conducting maximum votes or averaging to determine the best model for prediction [1] [2].


Fig. 1. Challenges in Evaluating Ensemble Methods

The key challenges in evaluating model performance are data issues, model complexities, resource limitations, and security challenges as illustrated in Fig. 1. In this paper, we will explore various Ensemble methods such as Bagging/Bootstrap Aggregation, RF, and Boosting alongside their underlying algorithm (DT) for Regression and Classification cases. The contribution of this paper is threefold. The foremost objective is to perform a theoretical analysis of various Ensemble methods with sample pseudocodes to aid the implementation process. Secondly, we implement Ensemble method algorithms using basic Python packages such as numpy, pandas, math, matplotlib and dataset installation packages for performance evaluations. Finally, we applied the developed code to real-world datasets across various industries, including healthcare, manufacturing, and real estate. The results highlight the different parameters and their performance on the algorithms while observing the proposed Ensemble program for performance evaluation.

## II. State of the Art

This section discusses the state-of-the-art and related literature on the various machine learning algorithms such as

Decision Tree (DT), Bagging (Bootstrap Aggregation), RF, and Boosting used in Ensemble Methods.

The DT model is built as a tree-like structure from the top and bottom [6]. The function of the DT is to discover patterns for classification and regression in large data [7]. This can be achieved by making predictions for a given observation in the training dataset by computing the variables' mean or mode of observation. Another function of DT is that it accepts a vector of input values and provides an output value as the decision. This is achieved by conducting a series of tests; the internal node of a DT corresponds to an examination of the value of each input attribute, the branch node is labelled with the possible values of the feature, and the leaf node specifies the value to be returned by the function which is also referred to as the output [8]. DT algorithms are simple and interpretable classifiers that stratify and segment the predictor space into several regions. The splitting rule in the predictor space can be summarised in a tree [9]. According to [10], tree-based methods partition the feature space into regions where a simple model (constant) is fitted into each region. A DT can be applied to both Classification and Regression tasks.

For instance, given a Regression problem, we have a dataset consisting of p inputs and one output. For each N observation, i.e., $(x_i, y_i)$ where i = 1, 2, …N, the input $x_i = (x_{i1}, x_{i2}, \ldots x_{ip})$. A typical regression tree will require determining the splitting points, the splitting variables, and the tree's shape. We can model the response as a constant $c_m$ in each region where the region ranges through $R_1$, $R_2$, $R_3$ …, and $R_m$ [10] as shown in (1).

$$f(x) = \sum_{m=1}^{M} C_m I(x \in R_m) \quad (1)$$

The greedy algorithm can be utilized to derive the best binary partition. It involves using all the data to determine the best split 's' and best variable 'j' and also defining a pair of half-planes [10]. $R_1(j, s) = \{X \mid X_j \leq s\}$ and $R_2(j, s) = \{X \mid X_j > s\}$ (2). To derive the splitting variable j and the split point s, we have to find the values that solve equation (2).

$$\min_{j,s}\left[\min_{c1}\sum_{X_i \in R_1(j,s)}(y_i - c_1)^2 + \min_{c2}\sum_{X_i \in R_2(j,s)}(y_i - c_2)^2\right] \quad (2)$$

Using the formula in (3) and (4), we can solve the inner minimization for j and s.

$$\hat{c}_1 = ave\big(y_i | x_i \in R_1(j,s)\big) \quad (3)$$
$$\hat{c}_2 = ave\big(y_i | x_i \in R_2(j,s)\big) \quad (4)$$

To determine the split point for each splitting variable, we scan all inputs to determine the best pair (j, s). After selecting the best split, we partition the data into two regions; we repeat the splitting process on each of the two regions and for other resulting regions [10].

Another example is the case of a Classification problem. Given a set of values, 1,2,3 … K, the criteria for splitting the node and pruning the tree is required in the Classification tree algorithm. In Classification, we define a node m, represented by a region $R_m$ with $N_m$ observations. The proportion of class k observations in node m is defined as in (5):

$$\hat{p}_{mk} = \frac{1}{N_m}\sum_{x_i \in R_m} I(y_i = k) \quad (5)$$

The observations in node m can be classified to class $k_m$ with (6) which stores the majority class in node m:

$$k_m = \arg\max_k \hat{p}_{mk} \quad (6)$$

Various node impurity measures can be used for tree-based algorithms. Misclassification error is a node impurity measure that works well with regression trees. Equation (7) is the formula to estimate the misclassification error:

$$\frac{1}{N_m}\sum_{i \in R_m} I\big(y_i \neq k(m)\big) = 1 - \hat{p}_{m_{k(m)}} \quad (7)$$

The Gini index or the Gini impurity is another node impurity measure, "referred to as a measure of node-purity -a small value which indicates that a node contains predominantly observations from a single class." [9].

$$\sum_{\kappa \neq, \kappa'} \hat{p}_{mk}\hat{p}_{mk'} = \sum_{k=1}^{k} \hat{p}mk(1 - \hat{p}_{mk}) \quad (8)$$

The formula for cross-entropy or deviance is listed in (9). Entropy is a measure of the uncertainty of a random variable, the acquisition of information reduces entropy [8]. When a random variable has one value this implies that its entropy is 0 because it has no uncertainty.

$$-\sum_{k=1}^{k} \hat{P}_{mk}\, log\, \hat{p}_{mk} \quad (9)$$

### A. *Bootstrap Nonparametric Resampling Technique*

Bootstrap technique, [1] [11] is a nonparametric resampling technique used to assess the uncertainty in an estimator to measure the standard error of coefficients in a linear regression model by applying uniform sampling of the training tuples with replacement. Hence, indicating that a selected tuple can be reselected and added to the training datasets. Further, states that Bootstrap can measure the standard error of coefficients in a linear regression model and for other statistical learning methods.

### B. *Bagging Techniques for Applying Decision Trees*

Regarding Bagging techniques, [9] describes Bagging using an example: Given a set of independent observations $Z_1$, $Z_2$, $Z_3$…. $Z_n$, the variance $\sigma^2$ and the mean is $\sigma^2/n$. This implies that averaging reduces variance. Based on this, the author stated that the natural way to increase prediction accuracy and reduce variance is by obtaining multiple training sets of a population, building a model for each training set as $\hat{f}^1(x), \hat{f}^2(x), \ldots, \hat{f}^B(x)$ and obtain their prediction.

$$\hat{f}_{avg}(x) = \frac{1}{B}\sum_{b-1}^{B} \hat{f}^b(x) \quad (10)$$

The prediction will then be averaged using (10 & 11), which would help derive a single low-variance statistical learning model. Although the above model is infeasible as we do not have access to multiple training sets, this is why we introduce bootstrapping, which will help generate sub-samples of the original dataset/ single training dataset by applying sampling with replacement to generate B as a different bootstrapped dataset. The *bth* bootstrapped training set is where the method will be trained to get $\hat{f}^{*b}(x)$, and all the predictions will be averaged, and we will derive (11).

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b-1}^{B} \hat{f}^{*b}(x) \quad (11)$$

Bagging can be applied to Regression Trees by obtaining B bootstrapped datasets and training each with a DT model. i.e., building B Regression Trees for B bootstrapped datasets and averaging their predictions. The Regression tree has a low bias but high variance as they are not pruned and have grown deep. By averaging the B Regression tree, we can reduce the variance. Bagging with a classification tree is quite different from regression trees as it involves recording the class predicted from each B classification tree and carrying out a majority vote, which is the most commonly occurring class in the B predictions, to determine the actual prediction. [9]. In Bagging, when we have a strong predictor and other moderately strong predictors, the split in each bagged tree is most likely to select the strong predictor for its split, resulting in the bagged trees being similar. Hence, the predictions from the tree are highly correlated to each other [9].

### C. Random Forest in Ensemble Method

RF is an ensemble of classifiers that comprises DTs. The DTs undergo two forms of randomization. One of the randomizations involves the DTs being trained by performing sampling with replacement using the original dataset while ensuring it is of the same size. The second form of randomization is attribute sampling, which involves obtaining a subset from the input variables at each node split to determine the best split [12]. According to [9], RF is an improvement to bagged trees as it involves a minor tweak that decorrelates the trees. It is pertinent to recall that we build DTs in Bagging on bootstrapped training samples. For instance, let the number of predictors considered at each split be m, and the length or total number of all predictors be p. Whenever a split is considered for each tree, the split involves randomly selecting m predictor from the p predictors, and we are only allowed to choose one m predictor. Within RF, we determine the predictor for a split by computing the $\sqrt{p}$ and approximating it to derive m. Based on the splitting convention in RF, the algorithm cannot consider most of the predictors, but only a subset; therefore, an average (p-m)/p split will not consider the strong predictors, giving room for the other predictors. This act is called decorrelating the trees; as a result, the average of the resultant trees varies less and is more reliable. The predictor choice for m is what differentiates Bagging and RF. p is the predictor choices for splits. When m = p, then this is referred to as Bagging, and when m = $\sqrt{p}$ this is RF. RF is an improvement of Bagging; it helps to improve variable selection [13]. The idea of RF is to support variance reduction in Bagging, and this is done by reducing the correlation between the trees while paying attention to the variance by not increasing it. The process is done in the tree-growing process by randomly selecting input variables [10]. When RFs are used for Classification, it obtains a class vote for each tree and classifies by performing a majority vote. For a Regression problem, RF computes the prediction average from each tree at a target point x. Based on recommendations by the inventors, the default value for 'm' is $\sqrt{p}$ with a minimum node size of 1 for Classification, and the default value for 'm' is p/3 with a minimum node size of 5 for Regression where m is the variables selected at random from p and p represents the input variables [10].

### D. Boosting Algorithm for the Ensemble Method

According to [10], Boosting is a popular and compelling learning method used in Regression and Classification cases. It was initially created to solve classification problems and can also be extended to solve regression problems. The function of Boosting is to obtain the output of weak classifiers to produce a committee. It is similar to other committee-based approaches like Bagging. In Boosting, a weight is assigned to each tuple of the training set, which is then updated after learning a classifier $M_i$ to permit the subsequent classifier $M_{i+1}$ to concentrate on training tuples misclassified by $M_i$. All the classifiers are iteratively learned, and the final boosted classifier, Mx accumulates the votes of the individual classifiers. The function of the accuracy of the individual classifiers is determined by the weight of each classifier's vote. [1]. Adaboost, a short form for Adaptive Boosting, is a popular Boosting algorithm. It has a substantial property: if the input learning algorithm is weak, and its output comes with a slightly better accuracy on the training set than random guessing, then the AdaBoost will return a hypothesis that classifies the training data perfectly for large enough K. The algorithm helps to boost the accuracy of the original input learning algorithm on the training dataset [8].

The purpose of Boosting is to sequentially apply weak classification algorithms to repeatedly modified versions of the data to produce a sequence of weak classifiers, obtain predictions from all of them and obtain the final prediction through a majority vote. [10]. Boosting implements a weighted training set, which involves assigning a weight of $\geq 0$ to each example of a training set; the higher the weight of an example determines the level of importance of the hypothesis in the learning process. Boosting assigns a weight of 1 to each of the examples; it generates the first hypothesis, which will correctly and incorrectly classify the training examples. The weight for the misclassified training examples will be increased to improve the classification performance. The weight for the correctly classified examples will decrease, and a second hypothesis will be generated, which will then apply the updated weighted training set. The process continues until the K hypothesis has been generated; the K will be an input to the Boosting algorithm. [8].

## III. Approach

This section presents an overview of the approach used for the paper, with a primary focus on assessing the various machine learning methods and algorithms such as Bagging, Bootstrapping, DTs, RF and Boosting, where each of their implementations consisted of two forms -Regression and the other for Classification. The description of Classification programs will be explained using the Iris dataset, and Boston Housing will be used as an example to describe the program written for the Regression task, as illustrated in Table 1.

TABLE I. Dataset description

| Dataset | Type | Features | Classes | Dataset Size |
|---|---|---|---|---|
| **Iris** | Classification | 4 | 3 | 150 |
| **Boston Housing** | Regression | 13 | N/A | 506 |
| **Cancer** | Classification | 24 | 2 | 1000 |
| **Heart** | Classification | 13 | 3 | 1025 |
| **Diabetes** | Regression | 10 | N/A | 442 |
| **Car Prices** | Regression | 49 | N/A | 13 |

Table 1 highlights the various datasets used to test the written program.

- The first dataset used for Classification was the Iris dataset. It comprises 150 observations, four features, and one target feature of three classes. The classes in the Iris dataset represent three types of flowers: setosa, versicolor, and virginica. Each class was assigned 50 observations, hence the 150 observations in the dataset.
- The Boston Housing dataset was used to test the Regression implementation. It contains 13 features and one target feature with 506 observations.
- The cancer patient's dataset consists of three classes (high, medium, and low), 1000 datapoints, 24 features, excluding the target feature -level and an index feature that counts each observation.
- The Heart disease dataset is another classification dataset. It consists of two classes represented as 0 and 1. It has 1025 datapoints with 13 features, excluding the target feature, which holds the actual value of the classes. The CSV file for the heart disease dataset was downloaded from the Kaggle website and added to the working directory of the program file.
- The Diabetes dataset is for Regression. It comprises of 442 observations, 10 features, and one target feature. In this project, the Diabetes dataset was imported from the sci-kit learn library.

Our approach considers using the object-oriented programming (OOP) approach with Anaconda Jupyter Notebook to create classes, objects, inheritance, and the freedom of reusing code by referencing functions that have the code rather than repeating blocks of code across the program. The program was implemented from scratch using basic Python packages such as numpy, pandas, math, random, matplotlib, and dataset installation packages. The pseudocode and explanations of each algorithm from the background study were used to guide the implementation process. The methodology process involves the application of preprocessing, model development and model evaluation.

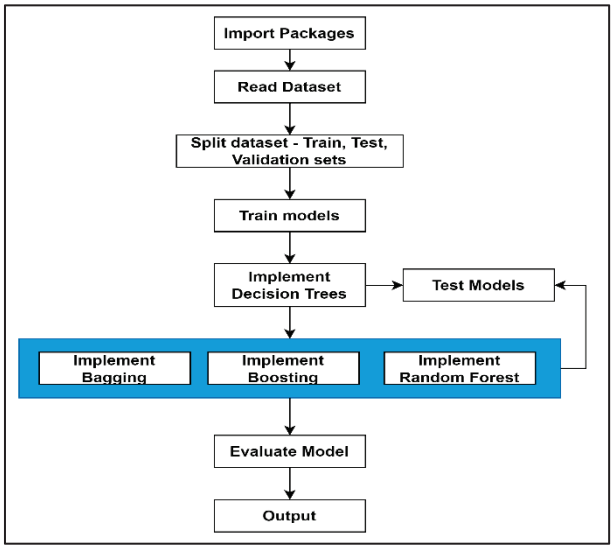

Fig. 2. Ensemble Methods Program Development Life Cycle

As illustrated in Fig. 2, the development process consists of package installation, dataset acquisition and manipulation, model training, model testing, model evaluation and result.

- The necessary packages such as numpy, pandas, math, random, matplotlib, and dataset installation packages were imported to the program working directory.
- Some of the datasets were acquired through Kaggle, World Bank data, and the UC Irvine machine learning repository and we performed the split to separate training, testing and validation sets.
- We trained and tested the models, for the predictions, we applied a majority vote for classification problems and performed averaging for regression problems.
- We used the evaluation measures for performance accuracy, RMSE for regression and a Confusion Matrix for Classification.

## IV. Implementation

We conducted the following activities for the implementation: Using Jupyter Notebook, we installed the required packages, such as *numpy, matplotlib, pyplot*, and *load_Iris* datasets from the sci-kit learn library. Afterwards, we applied the OOP approach to create a program for the DTs for both classification and regression cases. This required installing the necessary packages and incorporating the dataset into the written program.

### A. Decision Trees

The DT comprises of the Classification Trees and Regression Trees. The Classification Trees dataset contains class labels that can be used to make predictions by computing the majority vote of the classes after defining a set of rules. The Regression Trees dataset contains continuous variables whose mean will be computed to make the predictions.

Classification Tree: The program structure for the classification DT involves the creation of the parent class called *Node*, *whose* function represents each DT node. The key variables were initialized through the class *init* function, consisting of the *attribute* that served as a column, *threshold, left node, right node,* and *leaf node. They* were all assigned to *None* to keep them open so that whenever the class is initiated, it can be populated with the known values for these attributes and the necessary outputs can be stored when required. *ClassTrees* inherits the *Node* class to enable easy access to its properties and functions. *ClassTrees,* being the child class to the *Node* class, contained the underlying features of the Classification DT. Its arguments include *class_data* and the tree's maximum depth (*maxdepth*), which were initialised as a None type. These arguments are variables that can retain data that will be passed onto the class upon call. The *class_prob* function returns the probability of a set of labels upon function call. It obtains the unique values in *y*, indicating that y is the target variable used for measuring the most classes.

The *gini_index* function was written to measure the Gini impurity for the returned probability in *class_prob*. This function was programmed by rewriting the mathematical formula in equation 8 to compute the *Gini index* and return the value upon the function call. The *threshold* function derives the unique inputs for each column based on a given condition; that is, when an input is less than or equal to a given threshold -which is each unique input, then we place the sorted values as a list and store it in variable *B_1*. Furthermore, every other value with inputs exceeding the threshold will be stored as *B_2*. Upon function call, the dictionary of *B_1* and *B_2* values will be returned.

The *best* function retrieved the *threshold* function to get the values of the stored dictionary data. The purpose of the best function is to define the best column name and best threshold. The best *gini* was initialized as a negative infinity float, while the best threshold and best column name were initialized as *None*. Using the dictionary, all the column names and the threshold data were retrieved and looped over, and the input and labels of each bin were separated. The probability was obtained for the *y* labels by calling the *class_prob* function and then computing the *gini index* function for each value. The information gain was computed to determine the best Gini, and was used to determine the best column and threshold as follows:

$$\Delta = G(\mathcal{D}) - \frac{1}{|\mathcal{D}|}\left(|\mathcal{B}_1|G(\mathcal{B}_1) + |\mathcal{B}_2|G(\mathcal{B}_2)\right) \tag{12}$$

The *grow_tree* function retrieved the best threshold and the best column name from the best function. To derive the *B1* and *B2* values of the best threshold and best column, the thresholds' function needed to be called to get the dictionary as output and use this best threshold and best column as key for the easy retrieval of the *B1* and *B2* Values which was then split into the *X* and *y* values for both the *B1* and *B2* as *X_B1, X_B2, y_B1, y_B2*. The *leaf node* is computed by conducting a majority vote on the labels, which was done by applying the *np. argmax* function on the class probability to get the maximum index in the list to determine the majority. After which, we set conditions to produce the node of the tree to grow the tree completely. The current value is weighed with the set minimum value, and the specified max depth is checked with the current depth of the tree. If the program meets the set condition, then the parent class -*Node,* will be called. All its parameters will be filled with the now-known value, and the string function set for the *Node* class, a series

of strings naming the parameters with their new values, will be returned as output onto the screen. Another function called predict with parameter *self*, *X* and *Node* is used to obtain the final prediction. It calls the *createTree* function, which has three parameters: *self*, *x*, and *each node.* It makes predictions based on the value of each *leafnode* in the best function. The predict function obtains a list of all the predictions and performs the majority vote to declare the prediction for the particular tree. The matrix function is relevant for the experimentation method as it has two parameters, the actual and the pred, used to measure the performance of the algorithms. The accuracies can be measured by identifying the TP, TN, FP, and FN.

Regression Tree: The Regression [3] Tree implementation differs from the Classification tree. The program structure for the Regression DT involved importing the necessary packages, such as *numpy* and *pandas.* Unlike in the Classification tree, where we imported the dataset from the sci-kit learn library, a dataset was stored in the working folder called Boston Housing, which contained the test dataset used to test the implementation to ensure that the code worked and displayed the expected results. Similarly to the Classification, the Regression DT had a parent class called '*Node*'. The purpose of the Node class is to represent each node of the DT. The key variables were initialised using the class *init* function, consisting of the column, threshold, *leftnode*, *rightnode*, and *leafnode*, by assigning them to 'None'. So that whenever the class is initiated, we can populate the known values for these attributes and store the outputs that would be used when needed. A new class called *RegTrees* was created, which inherited the properties and functions of the Node class. *RegTrees* is the child class to the Node class and will contain the underlying features of the Classification DT. The arguments of the *RegTrees init* function are the *self* property used to obtain access to class attributes. The second argument is the *maxdepth* of the tree, which is defined as None and could be assigned another value upon class call. In the *RegTrees init* function, the dataset was initiated by storing the directory of the stored dataset as *self.dataset*, which was further split into feature and columns variables that represent the X and y values, and a variable *min_sample* was declared, which would be used as a stopping criterion later. Due to the outputs of the class changing forms, i.e., one of the input values is 0.00632, and the outcome was displayed as 6.32*103, making it challenging to perform arithmetic operations. After a series of searches, a code was obtained from the *numpy* documentation and referenced in the program. It helped to retain the actual form of the data.

The MSE function was defined to compute the mean square error. It has the following parameters: *self, act_y* – which collects the current value of y, and pred_y -which collects the predicted value of *y.* The formular for MSE $\sum(y_i - f(x_i))^2$ was declared to compute the *MSE*. The splits' function is similar to the Classification Tree threshold function. It has arguments, *X, y* and the class properties using self. The purpose of this function was to derive the unique inputs for each column based on a given condition; that is, when an input is less than or equal to each unique input, we place the sorted values as a list and store it as a variable R_1. Furthermore, every other value whose input exceeds the threshold will be stored as variable *R_2.* Upon function call, the dictionary of the values of *R_1* and *R_2* will be returned.

The best function -another shared function with the Classification Tree has arguments *self, X* and *y*. We called the threshold function to get the values of the stored dictionary data. The purpose of the best function is to derive the best column name and best threshold based on the best MSE, and it was initialised as *np.inf* and frequently replaced with the current MSE value until we determined the best MSE, best threshold and best column index (*best_idx*). Since the most important values are the *best_threshold* and *best_idx*, we returned the values at the end of the function.

The *grow_tree* function references the split function and the best function to obtain their returned value. The output of the best function acts as a key to the dictionary returned from the split function; this aided the easy retrieval of the *R1* and *R2* variables, which were then split into the *X* and *y* values to get *X_R1, X_R2, y_R1, y_R2.* The Node class was returned after setting stopping conditions. All the known parameters were passed onto the *Node* class, and whenever the *Node* class was called, the parameters with their new values were returned as a series of strings. The last two functions are for prediction. One of which is called predict with parameter *self*, *X* and *Node*. This function aims to obtain the final prediction. It calls the *createTree* function, which has three parameters: *self, x*, and *eachnode*. It makes predictions based on the value of each *leafnode* in the best function. The predict function obtains a list of all the predictions and performs an average of all their values to declare the final prediction for the particular tree.

### *B. Bagging Algorithm Implementation Process*

The Bagging Classification involved creating multiple Classification Trees, which was implemented by adding a *grow_mult_tree* with a *num_of_trees* parameter that determined the number of Classification Trees that should be implemented. A bootstrap function was created that involved performing sampling with replacement of the same size as the original dataset, and this implied that the parameter in the *grow_tree* function *X* and *y* were updated as they were passed into a loop of the *num_of_trees* variable passed when the *grow_mult_tree* function was called.

For the Bagging Regression, we created a function called *reg_predict*, then we initiated the *grow_tree* function and predicted the values passed on as its parameter for *X* and *y*. A *grow_mult_tree* function was also included in this class with *num_of_trees* as the parameter; a bootstrap function was created and called in the *grow_mult_tree* function. The *reg_predict* function was also included, and a prediction was made for each bootstrapped training set value. The predictions were made iteratively for range *num_of_trees* (the parameter for the *grow_mult_tree* function) and stored in a *store_prediction* list. Then, the Bagging class was initiated, and the *store_prediction* list was passed in as a parameter. The Bagging class is a child class for the *RegTrees* parent class. We created the *createTree* function and the mean to obtain the final prediction.

### *C. Random Forest Algorithm Implementation Process*

The RF for Classification requires the modification of the threshold function where the split is performed. In previous programs -the Bagging and Classification Trees, the threshold function contained two parameters (*X* and *y*). When they are called, it obtains all columns of the dataset and then separates them into two bins – *B1*, and *B2*. RF implementation for the split is quite different from the case of Classification, and it involves performing $\sqrt{p}$ where p stands for the number of columns in the dataset. Then, we sample them without replacement and obtain the splits for fewer columns to improve accuracy and yield better results.

Likewise, the RF for Regression consists of similar functions; the only difference is the modification of the split function where the split is performed. As previously stated, within the Bagging and Regression Trees implementation, the threshold function contained two parameters (*X* and *y*). When

the split function is executed, it obtains all dataset columns and then separates them into two regions – *R1*, and *R2*. The process of implementing the split in RF for regression is quite different during training, as it requires computing p/3 (where p stands for the number of columns in the dataset), which will then be sampled rather than the entire columns of the dataset, to gain better accuracy and yield better results.

### *D. Boosting Algorithm Implementation Process*

The Classification tree in boosting was defined by initiating a Node class after making the necessary imports, such as the dataset, numpy package and matplotlib package, which would be used to design plots later. A class Boosting was then defined, and it inherits the Node class. Subsets of the datasets were assigned as training sets, test sets, and validation sets. In Boosting, it is required that we obtain the prediction of multiple weak classifiers/ learners and use their outputs to form an ensemble. We represented the weak learner as a decision stump, a DT with a tree depth of 1. A few changes were made to the already established Classification tree to support the processes of the Boosting algorithm. The dataset labels were converted to -1 and 1 to convert it to a binary classification task and implement the Boosting algorithm properly. A function called predicts was defined with one parameter *X*. Its function is to make predictions based on the inputs and assign labels if it exceeds a certain threshold. Another function called best has three parameters: *X, y,* and *M*. *X* is for the inputs, *y* is for the target, and M is for the number of rounds. The weak learner was trained with a distribution of equal weight, *1/N*. A variable called err was defined to identify the misclassified label's weight from the trained model and weigh them with the other weights to obtain the total error. Alpha is another variable of the best function alpha; it involves measuring the performance of the weights by computing the log. The weights were then updated for the misclassified labels. The alpha and the prediction were obtained and stored as a tuple for each round and then appended into a list for both the training and test set before being assigned to a global variable. *Err_rate* is a function with two parameters: true and pred. This function measures the error rate from the actual and predicted values. To test the program and illustrate the plots, various ranges [1, 50, 500] of rounds were set and the *err_rate* was measured and plotted against the rounds. Both Classification and Regression share several key functions, as in the case of other algorithms. The difference between the two programs is the presence of the MAE in the Adaboost for Regression trees rather than the err_rate in Adaboost Classification trees. The MAE stands for Mean Absolute Error. Its function is to compute the RMSE - root mean square error, a performance measure for Regression tasks. Various ranges [1, 50, 500] of rounds were set to test the program and illustrate the plots, and the RMSE was measured and plotted against the rounds.

## V. Results and Discussion

In this section, we will discuss the implementation process of the developed code on real-world datasets from the healthcare, manufacturing and real estate industries. The program was tested on these datasets to compare the various algorithms' performance and ensure that the program worked well. Additionally, we will introduce the datasets and the splits for the training, test and validation set. Various plots illustrate the parameter changes with multiple measures and how the algorithms differ. An example of the plots is the confusion matrix, which helped to measure the accuracy across various parameters, as illustrated in Figure 3. The structure of the confusion matrix is that the top left is considered the *TN* (True Negative), i.e., when the true label and predicted label are both 0, the top right is the *FP* stands for False positive. It occurs when the predicted label is 1. The true label is 0, the bottom left is *FN* (False Negative is when the predicted label is 0 and true label is 1), and the bottom right is TP (True Positive is when both the predicted and true labels are 1). Tests were conducted for both the training and test sets, and the plot highlighting the difference in results was indicated and explained in this section.

The evaluation metrics used for this Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) for regression cases. We also used Accuracy, Recall, Precision metrics for the Classification program as indicated in Table 2.

TABLE II. Evaluation Metrics

| Acronym | Full form | Formula |
|---|---|---|
| RMSE | Root Mean Squared Error | $\sqrt{\left(\frac{1}{n}\right)\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}$ |
| MAE | Mean Absolute Error | $\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)$ |
| ACC | Accuracy | $\text{Acc}=\frac{TN+TP}{FP+TP+TN+FN}$ |
| REC | Recall | $\text{Rec}=\frac{TP}{TP+FN}$ |
| PRE | Precision | $\text{Pre}=\frac{TP}{TP+FP}$ |

Where $y_i$ is the actual value, $\hat{y}_i$ is is the predicted by the model, and n is the number of test datasets. Table 3 depicts the various algorithms and their performance accuracies on datasets with parameters of different values.

TABLE III. Classification Datasets Performance Accuracies

| Algorithm | Iris | Cancer | Heart | Number of Trees | Tree Depth |
|---|---|---|---|---|---|
| **Decision Trees** | 80% | 20% | 66% | 15 | 1 |
| | 100% | 20% | 66% | 10 | 2 |
| | 100% | 60% | 20% | 3 | 3 |
| **Bagging** | 40% | 0% | 20% | 15 | 1 |
| | 60% | 20% | 80% | 10 | 2 |
| | 80% | 20% | 80% | 3 | 3 |
| **Random Forest** | 40% | 40% | 20% | 15 | 1 |
| | 40% | 20% | 20% | 10 | 2 |
| | 40% | 20% | 100% | 3 | 3 |
| **Boosting** | 66% | 60% | 20% | - | - |

We obtained the training, validation and testing data by splitting the dataset into three, and by applying string slicing, we received the desired number of observations based on each class and the chosen observations were concatenated to create a new dataset that aided the testing process as shown in Fig. 3

```
try3 = ClassTrees(3)

#Merge train and val set to get the label of Xtest
X = np.concatenate((try3.X_train, try3.X_val), axis=0)
y = np.concatenate((try3.y_train, try3.y_val), axis=None)

# print('merge', X)
# print(y)

tree = try3.grow_tree(np.array(X), np.array(y), depth=1)
print(tree)
predicts, pred_class = try3.predict(try3.X_test, tree)
true = try3.y_test
# print(true)
```

Fig. 3. Concatenating Train and Test to Determine Optimality

The exact process was applied to the target feature and repeated for the test and validation set. The training and validation sets were concatenated for optimal results, and the merged dataset was used to train the model. The *predict* function was used to obtain predictions while comparing the actual value to compute the error.

### A. *Iris Dataset for Bagging Algorithm*

The Classification for the Bagging function involved running multiple DTs, obtaining their predictions, and retrieving the maximum. When the depth was set to 1 and the number of trees was set to 15, the majority vote was conducted, and the model was trained. The accuracy was generated based on the confusion matrix, resulting in an accuracy of 40% and an error of 60%. When the depth was changed to 2, it resulted in an accuracy of 60%, and when the depth was changed to 3 and the number of trees reduced to 3, it resulted in an accuracy of 80%. The error rate reduces every time the depth is changed. A higher depth led to a reduced error rate and more accuracy in the dataset.

### B. *Iris Dataset for Random Forest Algorithm*

When the number of trees was set to 15 for depth 1, it returned an error of 60%. i.e., 40% accuracy. The number of trees was changed alongside the depth to measure the accuracy, and it was discovered that the algorithm maintained the accuracy across the changes made to the depths. The accuracy obtained for this RF is steady across the dataset when using depths 1, 2, and 3. Although the accuracy obtained is lower than that of the Bagging, with RF, decorrelation of the trees can be successfully achieved with reduced variance and reliable outputs, as stated by [9].

### C. *Iris Dataset for Boosting Algorithm*

The Adaboost plot indicated at the bottom right of Figure 5 was obtained from the experiment conducted to measure the accuracy of the various rounds in the Boosting algorithm. The plot shows the number of rounds against the accuracy, and it results in a rapid rise from an accuracy of 33% at round 0 to 66% at round 50, and it maintains that to round 500.

### D. *Discussion*

The plots in Fig. 4. illustrate a specific measure and a corresponding range of parameters. As the key in each plot states, a blue line indicates the accuracy of the training set, and a red line represents the test set.

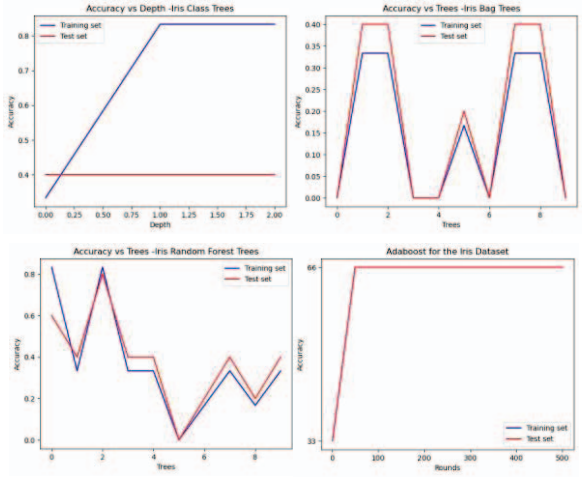


Fig. 4. Plots indicating a change in parameters vs accuracy

The top left plot displays the classification accuracy of trees across three depths. The training set shows a steady rise from when the depth is 0, the accuracy is 0%, and when the depth rises above 1, the accuracy remains above 80%, while the accuracy for the test set is constant at 40% from depth 0 to depth 2. The plot on the top right shows the accuracy of the tree across changes in the number of tree parameters.

The plot clearly shows that the accuracy of the training and test sets is similar, with the test set slightly higher than the training set. The accuracy of the tree rises and falls and can hence be considered unreliable. The plot on the bottom left is that of RF, which produces more reliable outputs. The accuracy of the training set begins at above 80% before falling and rising steadily when the number of trees set is above 8. The test set, on the other hand, follows the training set carefully. Although its training set starts with a much less accuracy -60%- they are similar.

### E. *Boston Housing Dataset for Regression Algorithms*

The Regression implementation for the Boston Housing dataset involved downloading the CSV file of the Boston Housing dataset from the Kaggle website and adding it to the working directory of the program file. The Boston Housing dataset contains 13 features and one target feature with 506 observations. We concatenated the training and test set for the regression tree and used them to train the model. To measure the model's performance, the RMSE was computed for the Regression tree within the Boston Housing dataset, and it was used to obtain the precise error of the prediction. The plot in Figure 6 illustrates the RMSE value of the depth parameter at various levels. It is evident that the RMSE level of the test set, as indicated by the red line, remains constant at slightly above 1.2. On the other hand, the RMSE of the training set starts at 1.0, then drops to nearly 0 at depth 1 and remains constant afterwards as illustrated in Fig. 5.

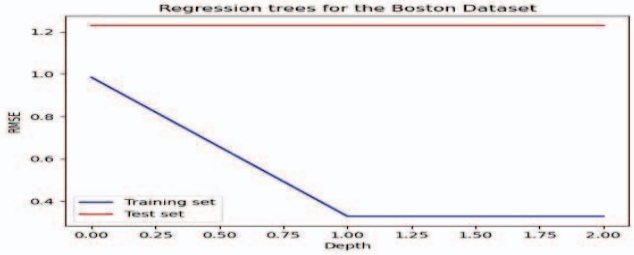


Fig. 5. Regression Trees with Boston Housing Dataset

### F. *Bagging Algorithm for the Boston Housing Dataset*

The plots in Figure 7 illustrate the RMSE values of the training and test set. The dark red combines the test set -light red and the train set -blue bars in each plot. The first plot showed a distribution of the training and test set when the depth was assigned to 1 and the number of trees was set to 15; the training set and test set look close to a normal distribution, with the presence of some outliers which are samples that vary from a particular trend. Once the number of trees was reset to 10 and the depth was changed to 2, the test set was close to a normal distribution with outliers and a uniform distribution for the test set. The following observation was setting the depth and number of trees to the same value, -3, resulting in a right-skewed distribution for the test set and a left-skewed distribution for the training set.

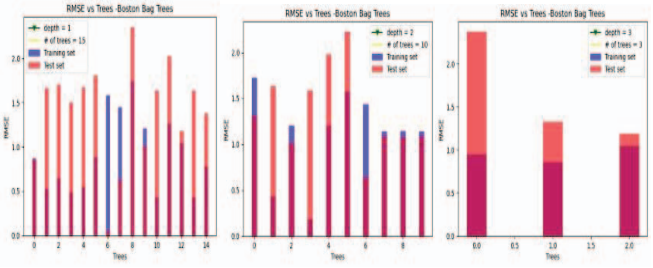


Fig. 6. Bagging RMSE with Boston Housing Dataset

### G. *Random Forest for Boston Housing Datasets*

In Fig. 7. below, the first plot illustrates the distribution of the graph when the depth is set to 1, and the number of trees is set to 15; it results in a bimodal distribution for the test set. The first one is right-skewed, and the second one is uniform. The training set is also bimodal and shows the presence of two

normal distributions with the presence of outliers: changing the depth to 2 and updating the number of trees to 10 results in a bimodal distribution with two left-skewed distributions. The last plot shows a close-to-uniform distribution when the depth is set to 3 and the number of trees is set to 3.

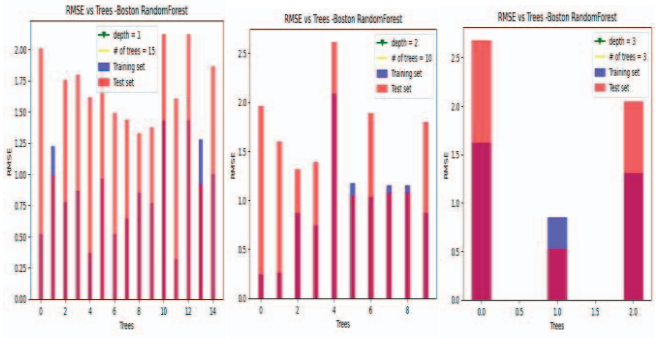


Fig. 7. RF with RMSE for Boston Housing dataset

### *H. Boosting for Boston Housing Dataset*

The Adaboost plot in Fig. 8. shows the measure of RMSE for the various rounds in the Boosting algorithm. The plot shows the number of rounds against the RMSE, and it results in a rapid fall to 0.81 from an RMSE value of 1.15 in the test set, then it remains constant to round 500. The training set, on the other hand, has an RMSE value of 1.08, then drops to 0.93 and remains constant at around 500.

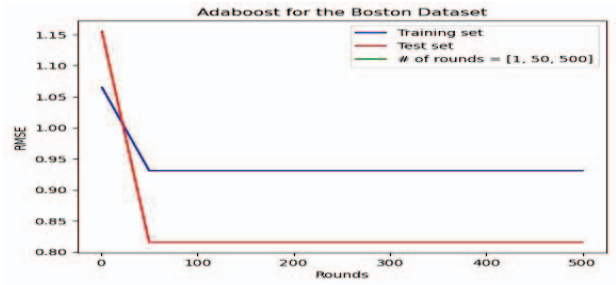


Fig. 8. Boosting with Boston Housing Dataset

### *I. Discussion of results*

These results provide important insights into how simple classification and regression trees can lead to overfitting. The accuracy steadily rose for most of the Bagging problems, indicating high variance. On the other hand, the RF is regarded as a more reliable algorithm due to its ability to eliminate high variance.

## VI. Conclusion

In this paper, we have explored various algorithms and performed theoretical analysis on various Ensemble methods. Additionally, we discussed the implementation process of the Ensemble method algorithms using basic Python packages such as numpy, pandas, math, matplotlib and dataset installation packages for performance evaluations. Further, we used evaluation metrics such as Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) for regression cases. We also used metrics such as Accuracy, Recall, and Precision for the Classification program. Furthermore, we discussed the developed code's application and its integration to real-world datasets across various industries, such as healthcare, manufacturing, and real estate.

The paper discusses the various implementations of the Ensemble method algorithm by applying statistical equations to the proposed program on real-world datasets across various industries, including healthcare, manufacturing, and real estate. Regarding contribution to knowledge, all the literature has made extensive contributions towards performance evaluations on AI regression and classification algorithms. For instance, [4] proposed a student performance prediction model using Ensemble methods like Boosting and RF to enhance the performance of classifiers like DT. [2] proposed a technique that combines the strength of various algorithms such as DT, RF and AdaBoost to improve their performance in regression and classification cases. [3] conducted predictive analytics on various algorithms using a single Malware dataset. However, none of them used an ensemble method on various healthcare, manufacturing, and real estate datasets to perform evaluations. Our results highlight the different parameters and their performance on the algorithms while observing the proposed Ensemble program for performance evaluation.

The limitation that comes with the issue of corrupted data and scarcity of high quantity of labelled data, extensive memory and processing required for the use of large-scale data to train AI models, the likelihood that adversaries could manipulate the input data making the intended system vulnerable to adversarial attack, and the challenge of algorithms performing well on training sets and poorly on testing data leading to bias and overfitting can be resolved by further study. Future work will consider applying various techniques on AI datasets on various classification algorithms for cyberattack predictive analytics on IoT devices.

## References


[1] J. Han, M. Kamber and J. Pei, Data Mining: Concepts and Techniques, San Francisco: Morgan Kaufmann, 2012.

[2] I. K. Nti, A. F. Adekoya and B. A. Weyori, "A comprehensive evaluation of ensemble learning for stock-market prediction," *Journal of Big Data,* pp. 1-40, 2020.

[3] A. Yeboah-Ofori, S. Islam, S. W. Lee, Z. U. Shamszaman, K. Muhammad, M. Altaf and M. S. Al-Rakhami, "Cyber Threat Predictive Analytics for Improving Cyber Supply Chain Security," *IEEE Access,* vol. 9, pp. 94318-94337, 2021.

[4] I. Alqatow, A. Rattrout and R. Jayousi, "Prediction of Student Performance with Machine Learning Algorithms Based on Ensemble Learning Methods," *Web Information Systems Engineering – WISE 2023,* vol. 14306, 2023.

[5] A. A. Nafea, M. Mishlish, A. M. S. Shaban, M. M. Al-Ani, K. M. A. Alheeti and H. J. Mohammed, "Enhancing Student's Performance Classification Using Ensemble Modeling," *Iraqi Journal for Computer Science and Mathematics,* vol. 4, no. 4, pp. 204-214, 2023.

[6] A. Yeboah-Ofori, C. Swart, F. A. Opoku-Boateng and S. Islam, "Cyber resilience in supply chain system security using machine learning for threat predictions," vol. 4, no. 1, pp. 1-36, 2022.

[7] A. Yeboah-Ofori, "Classification of Malware Attacks Using Machine Learning," *International Journal of Security (IJS),* vol. 11, no. 2, p. 16, 2020.

[8] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed., New Jersey: Pearson Education, 2010.

[9] G. James, D. Witten, T. Hastie and R. Tibshirani, An Introduction to Statistical Learning: With Applications in R, New York: Springer, 2013.

[10] T. Hastie, R. Tibshirani and J. Friedman, Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd ed., New York: Springer, 2009.

[11] D. Sarkar and V. Natarajan, Ensemble Machine Learning Cookbook, Birmingham: Packt, 2019.

[12] C. Bentéjac, A. Csörgő and G. Martínez Muñoz, "A comparative analysis of gradient Boosting algorithms," *Springer,* pp. 1938-1967, 2020.

[13] N. Altman and M. Krzywinski, "Ensemble methods: bagging and RFs," *Nature Methods,* vol. 14, no. 10, p. 933+, 2017.

[14] C. M. Bishop, Pattern Recognition and Machine Learning, New York: Springer, 2006.