Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Comparative Analysis of NGBoost and XGBoost

Tanvi Bagla
(19300699)

Supervised by: Dr. Myra O'Regan

September 2020

A Dissertation report
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science (Data Science)
School of Computer Science & Statistics
Trinity College Dublin, Ireland

Declaration

I hereby declare that the following dissertation, is entirely my work; it has not been submitted previously as an exercise for a degree, either in Trinity College Dublin or in any other university; and that the library may lend or copy it or any part thereof on request.

I have read and understood the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism, 'Ready Steady Write,' located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgment.

[illegible]

Tanvi Bagla

September 14, 2020

Acknowledgement

I would like to thank my supervisor, Dr. Myra O'Regan, who provided immense support, guidance, and feedback through each phase of this dissertation. She had persistently driven me towards improvement while I was studying and writing my dissertation work.

I am grateful to Trinity College Dublin and the School of Computer Science and Statistics for helping me build a solid foundation of computer science principles as well as research methods and providing me with a platform to excel. I would like to thank the course director Dr.John Dingliana by providing free cloud platform on Amazon Work Space and making things easy going with the course during the pandemic of COVID-19.

I would also like to mention the constant support of Alejandro Schuler, who is the co-author of the original paper of NGBoost named 'Natural Gradient Boosting for Probabilistic Prediction.' He helped me understand every basic concept of the newly developed algorithm. And Vishal Kumar, one of my colleagues whose expertise in writing the report in the standard format using Latex, helped me deliver the quality work.

Lastly, I would like to thank my family, whose moral support and motivation have been an indispensable factor in the successful completion of this dissertation, which kept me uplifted in low times.

Any omission of acknowledgement does not reflect the lack of regard or appreciation.

Abstract

Data scientists are coming up with new ensemble methods of machine learning by doing a little tweak in the way ensemble trees are learned, and the parameters are trained. The concept of Gradient Boosting Machines is running since 2001 (J. H. Friedman, 2001). Gradient boosting comprises of an ensemble of weak models that are combined to output one strong prediction model. NGBoost (Natural Gradient Boost) (Duan et al., 2019) is one of the recent boosting models that has the feature of giving probabilistic predictions. It is built on the principle where a base learner works on parametric probability distributions to be measured on a scoring rule.

This study aims to investigate and compare NGBoost with another gradient boosting method known as XGBoost. Both the models are applied to regression datasets taken from the UCI Machine Learning repository. The robustness of the models is analyzed based on hyperparameter tuning, K-fold validation, prediction accuracy calculated though RMSE and performance speed (CPU/GPU speed).

The models output the point predictions and probabilistic predictions when applied to the datasets. Through the experimental results, it is observed that NGBoost almost matches the performance with that of XGBoost, although XGBoost being on the better side. The RMSE scores are as follows: Regression datasets (Boston Housing Data: [NGBoost: 2.40], [XGBoost: 2.37]), (Protein Structure Data: [NGBoost: 3.58], [XGBoost: 3.44]). NGBoost also has the power of giving predictions of uncertainty (prediction intervals) in case of regression which XGBoost still lacks to predict.

Keywords: NGBoost, XGBoost, Natural gradient, Boosting

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Machine learning provides the concept of ensemble methods that helps to construct multiple independent base models. The base models are then integrated to get the optimized model. Ensemble methods divide into two classes – Bagging and Boosting. Term "Bagging" comes from the work done by (Breiman, 1996); a short form of "Bootstrap Aggregating" that includes the concept of bootstrap sampling. Bagging predictors is a tool for producing several predictor versions in parallel and using these to obtain an aggregated predictor. On the other hand, the algorithm of Boosting has emerged from the works of (Yoav Freund and R. Schapire, 1996), and the core idea is to implement simple classifiers iteratively and merge their solutions sequentially to produce a more reliable predictive result. The definition of boosting applies to the area of mathematical modelling, where the influence of predictors on a univariate response variable is identified and calculated in various regression settings (J. Friedman, Hastie, and Tibshirani, 2000)(J. H. Friedman, 2001).

Today, there are several areas where boosting algorithms are used, including credit ratings, prediction of medical risks, natural calamities, stock prices, planetary orbits, customer requirements and calculation of real estates. AdaBoost (Yoav Freund and Robert E Schapire, 1997) and Gradient Boosting Decision Trees (GBDT) (J. H. Friedman, 2001) are some of the initial learning algorithms used in practice. Besides that, there are other resource-friendly, efficient and optimized implementations of boosting, among which XG-Boost (Chen and Guestrin, 2016) and LightGBM (Ke et al., 2017) are applied to broaden the scalability and reduce the complexity. These models have the capability to train models with hundreds of trees using millions of training examples in a matter of a few minutes. Other than this, recent work formulated the new boosting algorithm NGBoost (Duan et al., 2019) that is based on natural gradient, used as the direction of the steepest ascent in Riemannian space on which boosting is applied to enable the probabilistic prediction for regression tasks. A comparison of different boosting algorithms is the basic principle of many pieces of work in the past and at present due to each algorithm having unique advantages dependent on a variety of features. A number of comparative works are highlighted in (Bentéjac, Csörgő, and Martínez-Muñoz, 2019)(Daoud, 2019)(Lakshminarayanan, Pritzel, and Blundell, 2017). The comparison and analysis in these works are based on several criteria like the hyperparameter setting used for training the model, time taken by the models to tune and fit the model, how it deals with bias-variance trade-off. Parallelization, regularization, tree pruning, early stopping and finding out prediction intervals are some of the features that makes the boosting models differ from each other.

1.2 Research Objective

The research work aims to:

- Compare the performance of the novel algorithm NGBoost with the existing algorithm XGBoost that is used in almost all the machine learning problems. The performance is measured in terms of metrics such as RMSE, R-Square, GPU/CPU runtime, hyperparameter tuning efficiency, average cross-validation scores and size of data handled. The aim is to analyze how well both the algorithms perform in comparison to each other when applied on two regression datasets.

- To understand and compare the statistics behind each model (their loss function, training parameters and scoring rules).

1.3 Methodology overview

The approach adopted consists of a thorough study of the statistics behind both the models, XGBoost and NGBoost, and applying these models to the two datasets (Boston Housing Data and Protein Structure Data) taken from UCI Machine Repository. For the datasets, splitting is done using train-test split in the ratio of 80% and 20% or using k-fold cross-validation. Models are analyzed on default hyperparameters and tuned hyperparameters. After fitting the model on the training set, the predictions are made on the test set. The results are compared using different accuracy metrics such as RMSE and R-squared values. Figures like predicted Vs actual plots, GPU/CPU performance plots, Feature importance plots are created to provide a better insight into the results produced by the algorithms.

1.4 Dissertation Structure

This dissertation is organized as follows:

- Chapter 1 - Introduction includes an overview of what the research work is about, the objective to do this research, and the brief description of the methods and the architecture followed.
- Chapter 2 - Details the underlying concepts and thorough study of the related literature done under the previous works. It also mentions the motivation behind this work.
- Chapter 3 - States the conceptual, statistical and analytical description of the methods used during implementation.
- Chapter 4 - Describes the data sets and elaborated steps of the implementation and analysis performed. It also mentions the result summary and the comparative analysis.
- Chapter 5 - Conclusion and future work mentions the conclusion of the thesis, the challenges faced and the work that can be progressed in the future.

Chapter 2

Literature Review

This chapter provides a thorough study of existing work related to the research topic. The study includes background work related to, how the boosting concept was introduced, how it evolved, different studies related to various types of boosting techniques. The review of work is done by categorizing the literature based on the type of study, such as analysis done on boosting techniques based on accuracy, performance, hyperparameter tuning and probabilistic predictions. Where XGBoost has a record of a lot of literature work; NGBoost being newly developed, does not have much related information available. Lastly, it mentions the motivation and the questions that are answered through this research work.

2.1 Background

Boosting has its roots theoretically, since the concept of PAC learning model was introduced (L. G. Valiant, 1984). PAC learning (Probably Approximately Correct) comes from a class of learning models where a model is learned from a set of examples/samples. Suppose 'f' is a target function to be learned, then some random samples are provided to the learner. Based on these samples, the learner deduces the target function. PAC works on the concept that a function is learn-able if the low error and high confidence can be achieved for all samples provided to that function. This concept is known as 'strongly learn-able.' Now the boosting introduces a concept of weak learnability. 'Weak learnability' drops the idea that the learner should achieve high accuracy. A weak learning algorithm needs output that works just slightly better than random guessing. The concept of weak learnability was presented by (M. J. Kearns and Leslie G Valiant, 1988)(M. Kearns and L. Valiant, 1994). The question that arises about the PAC learning model is that whether the strong learnability can be attained from weak learnability 'Hypothesis Boosting Problem' (L. G. Valiant, 1984). Kearns, in his paper (M. J. Kearns and Leslie G Valiant, 1988), mentions that boosting can be applied to weak learners that could boost the accuracy to convert it into a strong learner.

Gradient boosting methods build the solution in a stagewise manner by minimizing the loss functions and solving the overfitting problem while progressing. The basic property of any boosting algorithm is that the upper bound of the training error can be calculated on the final training (Y. Freund, 1995). The concept of boosting was initially presented in a paper (Robert E. Schapire, 1990) that states that a batch of weak learners can construct one strong learner. Weak learners are classifiers that are slightly correlated with the actual classification. In contrast, a strong learner is easily classified and well-aligned with the true classification. Freund and Schapire (Yoav Freund and Robert E Schapire, 1997) finally developed a powerful boosting algorithm known as Adaboost. Adaboost focuses on misclassified examples, which are mostly outliers. When the frequency of outliers increases, the emphasis placed on these samples may deteriorate the performance of Adaboost. Thus Adaboost is sensitive towards outliers and noisy data. Initial experiments with these boosting algorithms were carried out by Drucker, Schapire and Simard (Drucker, R. Schapire, and Simard, 1993) on an OCR task. In these experiments the impact of boosting is analyzed on handwritten image datasets containing 12000 digits of the ZIP codes from US postal service and 22000 digits from NIST (National Institute of Standards and Technology). Based on the two performance metrics - first, the reject rate and second, the raw error rate (no reject rate), the results show significant improvement when boosting is used.

It has become essential to find models that can deal efficiently with complex and broad data. The ensemble method is an effective technique for optimizing the efficiency of various current models such as XGBoost, Catboost or LightGBM (J. H. Friedman, 2001)(Chen and Guestrin, 2016)(Yoav Freund and Robert E Schapire, 1999). To choose the best ensemble model based on the factors like accuracy, speed, space occupied, processors there have been many comparative studies that are carried out as stated under section 2.2. Boosting algorithms being in the top list of the machine learners in terms of the learning it provides, are being questioned repetitively whenever the new algorithm/ model comes in the market. The fundamental difference between the various forms of boosting algorithms is the technique used in the weighting of the training data points (Bühlmann and Hothorn, 2007). By weighting it means, that instead of each variable present in the dataset, contributing equally, some of the variables are adjusted to contribute more than the others. AdaBoost is a popular Machine Learning algorithm and historically significant, being the first algorithm capable of working with weak learners. More recent models include XGBoost, Cat boost, LightGBM, and NGBoost. NGBoost being the latest one in the boosting family.

2.2 Comparative Analysis

2.2.1 Accuracy

XGBoost works on the quality of sparse data handling and scalability by adding a regularisation term to the loss function that makes the loss function work optimally. Cache analysis, compressed data, and sequential learning are essential insights to generate end to end scalable tree boosting network, XGBoost by combining these principles, solves real-world scale problems using a minimal amount of resources (Chen and Guestrin, 2016). One of the studies (Bentéjac, Csörgő, and Martínez-Muñoz, 2019) compares XGBoost performance with the Gradient tree boosting and Random Forest algorithm. The comparison is in terms of both accuracy and speed. The findings indicate that tuned gradient boosting performs with the highest accuracy in the maximum number of datasets (in 10 datasets out of 28). It also reports that the training of Gradient Boosting and XGBoost done based on the default parameters is the least efficient method. The authors specify that random forest is the most robust algorithm in terms of the difference between model trained on default parameters and tuned parameters. The paper gives comparative results on the training time of each algorithm and states that XGBoost performed almost 3.5 times faster than random forest and 2.4 times faster than gradient boosting.

Another work by (Daoud, 2019) compares XGBoost, LightGBM, and CatBoost on the Home Credit Dataset, which shows that LightGBM has the best under-curve AUC region (0.89) and the quickest training period (786 min), while XGBoost has one of the worst training period (4306 min), and CatBoost, on the other hand takes training time of 1803 mins. Such findings are not, though, generalizable to other datasets. Suppose there are more categorical variables in the dataset, the Catboost might work better than the other models as it has the capability to handle categorical values through ordered target encoding. In this work, the author states that LightGBM tends to be considerably quicker than the other gradient boosting algorithms and more efficient utilizing the same time estimate of hyper-parameters tuning. Results show that LightGBM consumes about 1425MB of RAM space where XGBoost takes about 1684MB space stating that LightGBM has lower space usage during training time.

(Rahman et al., 2020) and (Nemeth, Borkin, and Michalconok, 2019) describes the basic structural difference between LightGBM and XGBoost. It is the method by which the computation of the best split takes place. For LightGBM (Ke et al., 2017), one side sampling is done to find out the observations used for training. In XGBoost, a split is determined by filtering the observations based on histograms. Histogram based algorithms groups the features into separate bins, and instead of splitting based on the features, it splits based on the bins. The computing time taken by the histogram-based algorithm is more than the one side sampling as instead of making use of histograms, it down-samples the features based on the gradients. The instances with large gradients are retained for training and those with small gradients (more trained) are discarded to perform random sampling. Thus, stating that LightGBM is more efficient than XGBoost in terms of time complexity.

2.2.2 GPU/CPU performance

Training gradient boosting algorithms usually takes time since it builds trees one at a time that is in a gradual learning, sequential and additive manner, the time usually increases when the training is of a large number of trees on big datasets. GPUs are used to enhance the efficiency of the model. Grahn et al. suggests to make use of separate GPU threads to train each decision tree for the random forest(one of the ensemble learning methods that works on multiple decision trees in parallel) (Grahn et al., 2011). XGBoost uses this idea too, in parallelism, to do several activities like measuring possible split points simultaneously, identifying the best split point for multiple attributes parallelly on a node, and finally figuring out the best attribute for numerous nodes simultaneously. To accelerate the performance of XGBoost, it is proposed by one research work to use GPUs to find the best splits for XGBoost faster (Mitchell and Frank, 2017). LightGBM (Ke et al., 2017) is an alternative implementation to accelerate the training process of gradient boosting trees, that works on the approximation of split points. XGBoost, on the other hand also supports approximation. The difference is that, where XGBoost works on breadth-first node splitting, LightGBM works on depth-first nodes split. Besides LightGBM and XGBoost, CatBoost (Prokhorenkova et al., 2018), one of the latest boosting methods, also supports approximation. The key difference between LightGBM and CatBoost is that LightGBM excludes instances with lesser weights and combines strongly correlated attributes whereas CatBoost includes all the oblivious trees (trees having leaves at the same level) in the training process where the whole level has the same split criteria. Since CatBoost includes all the trees at the same level, it performs slower than LightGBM.

GPU acceleration has become essential to increase the model performance as they take greater computational power and memory bandwidth with respect to CPU. The full exploitation of GPUs and their impact on corresponding algorithms has always been a topic of research.

2.2.3 Probabilistic Prediction

Approaches to probabilistic prediction can broadly be distinguished as Bayesian or non-Bayesian. Bayesian methods (prior predicting posterior) works on decision trees for tabular datasets (Chipman, George, and McCulloch, 2010) and (Lakshminarayanan, Roy, and Teh, 2016). A non-Bayesian methodology just like Natural Gradient Boosting (Lakshminarayanan, Pritzel, and Blundell, 2017) uses a parametric approach to train models with heteroscedastic uncertainty. By heteroscedastic uncertainty it means, the observation disturbance or observation noise varies with input variable (that does not have same variance throughout the observations). Such a heteroscedastic solution to capture uncertainty was sometimes named aleatoric calculation of uncertainty (Kendall and Gal, 2017), which is nothing but a statistical uncertainty due to noisy data caused on the observations.

Many studies are ongoing to find out how probabilistic predictions are made using boosting algorithms. Generalized Additive Models for Location Scale Shape (GAMLSS) is one of them which is introduced by (Rigby and Stasinopoulos, 2005) and extended to a Bayesian framework by (Klein et al., 2014). This model is a structure that allows all distribution parameters to be modeled as covariate functions. Recently März, Alexander (März, 2019) proposed theoretically, XGBoostLSS that currently has its development in progress, structured in R language. It's implementation aims to enhance XGBoost to a full probabilistic forecasting framework. It will model over entire parametric distribution - mean, location, scale, and shape (LSS). Taking a benefit of its Newton boosting feature and the relation between empirical risk minimization and Maximum Likelihood estimation, XGBoostLSS aims to predict the entire conditional distribution from which prediction intervals and quantiles can be obtained. Provided a set of covariates XGBoostLSS will offer a detailed explanation of the response distribution.

NGBoost (Duan et al., 2019) defined Natural Gradient as the path of the steepest elevation in distribution space and implemented it to boost the probabilistic potential to forecast the tasks of regression. Natural gradient boosting shows promising performance improvements on small datasets due to better training dynamics, but it suffers from slow training speed overhead, especially for large datasets (Ren, Sun and Wu, 2019). The NGBoost algorithm is a supervised learning method for probabilistic forecasting that approaches boosting for predicting parameters of the conditional probability distribution. In contrast with (J. H. Friedman, 2001), where the ordinary gradient is used, and with (Chen and Guestrin, 2016) where a second-order Taylor approximation is minimized in a Newton-Raphson step, NGBoost uses the natural gradients as the response variable to fit the base learner (Duan et al., 2019).

2.2.4 Hyperparameter Tuning

The tuning process leads to over 99.9 percent of the computational energy required to train gradient boosting or XGBoost as shown in this study (Bentéjac, Csörgő, and Martínez-Muñoz, 2019), which focuses on grid search parameter tuning utilizing 10-fold cross-validation. The grid search time can, however, be dramatically reduced when the smaller parameter grid is used for XGBoost. XGBoost enables fine-tuning of parameters that make use of the computationally effective algorithm. It is not much achievable with the random forest (as minimal benefits are achieved with parameter tuning) or with gradient boosting, which takes longer computational cycles. XGBoost offers the benefit of regularisation while tuning which is an added advantage over other boosting algorithms as stated in (Yoav Freund and R. Schapire, 1996).

2.3 Motivation

In reference to the paper (Duan et al., 2019) that trains the model depending on the probabilistic distribution of parameters rather than parameters itself, the results show that it excels in terms of probabilistic predictions as compared to other probabilistic algorithms like Deep Ensembles, MC Dropout, GAMLSS, Gaussian Process. Other work that compares XGBoost with LightGBM and CatBoost (Daoud, 2019) states that LightGBM has the fastest training time (786 mins to train 307507 rows). There are numerous studies, including some of those mentioned above like (Ke et al., 2017) stating LightGBM to be more efficient or in the results of (Bentéjac, Csörgő, and Martínez-Muñoz, 2019) where XGBoost overpowers random forest and gradient boosting. No work is done where the new boosting algorithm NGBoost is compared to the other boosting algorithms. XGBoost is one of the highly accurate, scalable, robust, and versatile boosting algorithms at the moment. Thus, making it a good choice to be compared with any upcoming boosting algorithm (NGBoost in this work) in-order to rank and analyse the features of the new algorithm. Due to the novelty of NGBoost, it is interesting to analyze and rate the algorithm in terms of accuracy, GPU/CPU performance, and hyperparameter tuning.

2.4 Research Question

This research tries to answer the following questions:

1. Does NGBoost perform better in case of point prediction as compared to XGBoost?
2. What is the effect of hyperparameter tuning on both the algorithms?
3. How do both the algorithms perform in terms of computational time on large datasets?

Chapter 3

Methods

This chapter aims to study and describe theoretical, conceptual and mathematical details behind XGBoost and NGBoost. Under XGBoost it states what the loss function is, how the objective function differs from normal gradient descent, how the training is done and how the best split is done using XGBoost. Under NGBoost, pitfalls of gradient descent that lead to natural gradient descent, how the natural gradient works and the concept of natural gradient boosting are described. Also, different hyperparameters of each algorithm and the techniques used in finding the best hyperparameters are stated. Later sections describe the model evaluation metrics used in the work, description of the type of outputs predicted and finally, the environmental setup of the project.

3.1 Algorithms Used

3.1.1 Extreme Gradient Boosting (XGBoost)

Extreme Gradient Boosting (XGBoost) is a decision tree ensemble based on the version of the gradient boosting framework (J. H. Friedman, 2001). Gradient tree boosting is a method of boosting the tree ensemble which combines a collection of weak classifiers to obtain a strong classifier. Beginning with a base learner the strongest learner is taught iteratively (Chen and Guestrin, 2016). Both, gradient boosting and XGBoost obey the same principle. The main distinction between them lies in the specifics of execution.

XGBoost produces improved efficiency by utilising various regularisation methods to monitor the size of the trees (Chen and Guestrin, 2016).

Let x_i be the input features passed through a tree ensemble model. Using 'K' additive functions, each representing a tree (combining 'K' trees to get good predictive output). The equation of the predicted output is given by adding each individual prediction, refer equation 3.2 (Chen and Guestrin, 2016):

Each individual prediction (output y_i) is the function of input features (x_i), where i indexes over some training set of size n of actual values, given by the equation 3.1.

$$y_i = f(x_i) \tag{3.1}$$

Predicted output through ensemble tree algorithm (sum of individual outputs) that is over 'K' additive trees is given by equation 3.2.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in F \tag{3.2}$$

$\hat{y}_i$ being the predicted output produced by the tree ensemble method, is given as the sum of the individual tree function f_k having set of input features x_i on 'K' number of trees. f_k representing an individual tree in the space of regression trees 'F'. Gradient descent algorithm works to find the minima of the loss function. This can be achieved by calculating the loss function and, taking the derivative of the loss that gives the direction in which the movement needs to be done to reach to the minima. The goal is to minimize the loss function. So, the objective function is:

Objective function = Training Loss function

How well the model fits the training data is specified by the training loss function. Optimizing the loss function sometimes may lead to overfitting. In order to compensate with this, a regularization function is added to the training function. Optimizing regularization may balance overfitting. So in XGBoost the objective function is given by:

Objective function = Training Loss function + Regularization term

$$Obj(\theta) = L(\theta) + \Omega(\theta)$$

$$\text{obj} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega(f_i) \tag{3.3}$$

where $L(\theta)$ can be any loss function $L(\theta) = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)})$, ‘t’ is the number of trees and ‘n’ is the number of observations.

The loss function for regression is calculated using error score, such as Mean Squared Error (MSE), given by (Chen, 2014)
Square loss (for regression) : $l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$
And logistic loss function is given by the following:
Logistic loss(for binary classification): $l(y_i, \hat{y}_i) = y_i \log p_i + (1 - y_i) \log(1 - p_i)$
where $p_i = \frac{1}{1+e^{-\hat{y}_i}}$

The second term $\Omega(f_i)$ is the regularisation term, which regularizes the model complexity that avoids overfitting (Chen and Guestrin, 2016). In XGBoost the regularisation term is given by (Chen, 2014):

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda ||\omega||^2 \tag{3.4}$$

where T is the number of leaves and the second term having λ as a L2 regularization term of leaf scores, w is the leaf weights. While the training is in process it is additive in nature i.e. optimising one tree at a time. Let $\hat{y}_i^t$ be the predicted value at iteration t, (Chen, 2014):

$$\hat{y}_i^t = \hat{y}_i^{t-1} + f_t(x_i) \tag{3.5}$$

and $\hat{y}_i^{t-1}$ is the output value at (t-1)th iteration. Rewriting the objective function in terms of additive training by substituting equation 3.5 in equation 3.3.(Chen and Guestrin, 2016)

$$\text{obj} = \sum_{i=1}^{n} l(y_i, (\hat{y}_i^{(t-1)} + f_t(x_i))) + \Omega(f_t) \tag{3.6}$$

Since the aim is to minimize the objective function, the best tree f_t must be identified. Rewriting the objective function using Taylor’s approximation upto second degree.

Taylor's approximation:

$$f(x+\delta x) = f(x) + f'(x)\delta x + \frac{1}{2}f''\delta x^2$$

Objective function (3.6) using Taylor's approximation (J. Friedman, Hastie, and Tibshirani, 2000):

$$\text{obj} = \sum_{i=1}^{n}\left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i)) + \frac{1}{2}h_i f_t^2(x_i)\right] + \Omega(f_t) \tag{3.7}$$

- g_i (first order derivative) $= \partial\hat{y}^{(t-1)} l(y_i, \hat{y}_i^{(t-1)})$
- h_i (second order derivative) $= \partial^2\hat{y}^{(t-1)} l(y_i, \hat{y}_i^{(t-1)})$

Moving back to the objective function, if we remove the constant from equation 3.7:

$$\text{obj}^{(\text{t})} \simeq \sum_{i=1}^{n}\left[g_i f_t(x_i)) + \frac{1}{2}h_i f_t^2(x_i)\right] + \Omega(f_t) \tag{3.8}$$

This becomes our optimisation goal for new tree f_t which is a function of weights on leaves

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

Where T is the number of leaves, w is the vector of scores on leaves, q is a function assigning each data point to the corresponding leaf.

Expanding regularization term in equation 3.8

$$Objective \quad function = \sum_{i=1}^{n}\left[g_i w_{q(x_i)} + \frac{1}{2}h_i w_{q(x_i)}^2\right] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2 \tag{3.9}$$

$$Objective \quad function = \sum_{j=1}^{T}\left[\left(\sum_{i\in I_j} g_i\right) w_j + \frac{1}{2}\left(\sum_{i\in I_j} h_i + \lambda\right) w_j^2\right] + \gamma T \tag{3.10}$$

where I_j is the set of instances of j^{th} leaf.

Rewriting the objective function (equation 3.10) such that $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$

$$\text{obj}^{(t)} \simeq \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2} \left(H_j + \lambda w_j^2 \right) \right] + \gamma T \tag{3.11}$$

Equation 3.11 is a quadratic function of w_j. The optimal weight w_j can be calculated as:

$$Leaf\,values = w_j^* = \frac{-G_j}{H_j + \lambda} \tag{3.12}$$

Substituting the optimal weight value from equation 3.12 in equation 3.11 to find out the optimal value of objective function:

$$\text{Obj}^* = \frac{-1}{2} * \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \tag{3.13}$$

This equation 3.13 can also be used as a scoring function that measures the quality of a tree structure. Below figure 3.1 tells how the score can be calculated by adding the gradient and second order gradient statistics on each leaf and then the scoring formula is applied (Chen and Guestrin, 2016). This score is for a leaf and is a function of first order and second order derivative of the loss function.

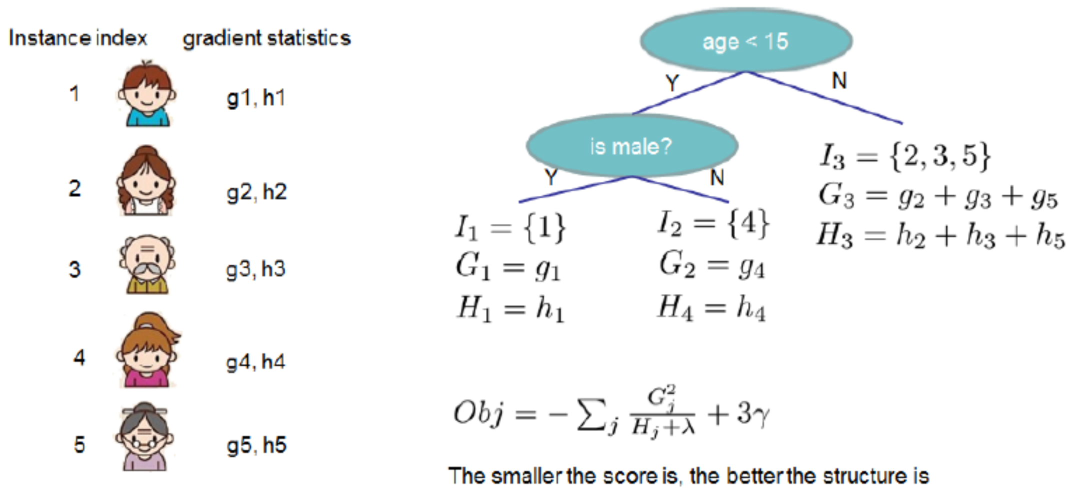


Figure 3.1: Structure score calculation in XGBoost

In order to find a good tree structure and where to split, a search algorithm is followed for a single tree.

Search Algorithm used in XGBoost for a single tree

1. Make a list of all tree structures.
2. For each tree, calculate the structure score by using equation 3.12
3. Find out the best tree structure having best structure score from Step 2.
4. Identify the optimal leaf weights of the best structure score.

$$w_j^* = \frac{G_j}{H_j + \lambda} \tag{3.14}$$

There can be multiple tree structure (infinite) having best scores. In order to find a best split, exact greedy algorithm is used (Chen and Guestrin, 2016).

Structure score of a tree having only a root node, from equation 3.12:

$$S1 = -\frac{1}{2}\frac{G_o^2}{H_0 + \lambda} + \gamma \tag{3.15}$$

Structure score of a tree having two terminal nodes:

$$S2 = -\frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda}\right] + 2 * \gamma \tag{3.16}$$

$$\begin{aligned} Gain &= S1 - S2 \\ &= \left(-\frac{1}{2}\frac{G_o^2}{H_0 + \lambda} + \gamma\right) - \left(-\frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda}\right] + 2 * \gamma\right) \\ &= \frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} + \frac{G_0^2}{H_0 + \lambda}\right] - \gamma \end{aligned} \tag{3.17}$$

Writing G_0 as $G_L + G_R$ and H_0 as $H_L + H_R$

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} + \frac{(G_L + G_R)^2}{(H_L + H_R) + \lambda}\right] - \gamma \tag{3.18}$$

Exact Greedy algorithm (Chen and Guestrin, 2016)

1. Start at depth=0 of a tree.

2. For each leaf node evaluate split that gives maximum gain (equation 3.18)

Where $\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} + \frac{(G_L+G_R)^2}{(H_L+H_R)+\lambda}$ is training loss term and γ is the regularization term. Note the tree must stop growing further when the Gain function is negative. In order to avoid reaching to the negative gain, two methods can be applied, early stopping (stop as soon as the best split has the negative gain) and post pruning (grow to a max depth and recursively prune all the leaf splits that are having negative gain). Using this Gain XGBoost builds tree structure.

3.1.1.1 XGBoost Hyperparameters

XGBoost parameters are divided into three categories **General parameters, Booster parameters** and **Learning task parameters**. Below listed parameters are included in the scikit learn library of the Python.

General Parameters

- booster - Type of the booster chosen to train the model. Can be gblinear, gbtree, or dart. (Default: gbtree)

- verbosity – Whether or not to print messages during runtime. 0 - silent, 1 - warning, 2 - info and 3 - debug.

- Nthread – number of parallel threads needed (-1 to consume all parallel threads)

- disable_default_eval_metric - disables the default evaluation metrics (Default: False)

Parameters for Tree Booster

- learning_rate – step size used in each iteration to prevent over-fitting. It is also known as shrinkage. Weights are multiplied by the learning rate in each iteration to modify/shrink the weights. Small learning rate means small steps and greater learning means larger steps. (Default: 0.3) range: [0,1]

- gamma – minimum loss reduction needed, to further partition a leaf node. It controls the regularisation of gain by calculating the structure score at each split.

  From equation 3.18, gain is defined as:

  $$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} + \frac{(G_L+G_R)^2}{(H_L+H_R)+\lambda}\right] - \gamma$$

  where $\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} + \frac{(G_L+G_R)^2}{(H_L+H_R)+\lambda}$ is training loss term and γ is the regularization term. Greater value of gamma means more regularized will be the model. Gamma gives the ability to xgboost to prune the tree. If the train and test scores differ a lot that means the model is not fit properly. There might be a high variance, growing too many trees by the model. In such case tree must be pruned. Increase the gamma such that test score comes closer to the train score.

  If $Gain > 0$, that means the model is not complex enough. Raising the gamma such that the gain becomes negative (pruning of the trees) makes the model complex. In other case if test score is close enough to train score and blocks the test score to improve as the train score is stuck. In this case, decrease the gamma such that the gain becomes positive resulting in growing of the tree further. Thus making the model learn a little more from the tree.

  (Default=0). range: [0,∞]

- max_depth - denotes maximum depth of a tree. Greater depth means more leaf nodes, resulting in more complexity with greater chance of over-fitting. There will be more interaction and less number of instances in each node. Smaller trees are probably better. range: [0,∞]

- min_child_weight - minimum weight of the sum of instances required to be a child node. This puts limit on the tree growth. Larger values of this parameter means shorter trees. If the number of instances in the leaf node are lesser than the set min_child_weight the tree will not split the leaf. (Default=1]. range: [0,∞]

- subsample - Subsample ratio of the training instances (Default=1). It depends on gradient boosting or stochastic gradient boosting. Value between 0 and 1 is set to perform stochastic gradient boosting and value set to 1 means normal gradient boosting. It is effective when there are outliers in the dataset, as subsampling limits the use of complete dataset for the model training. Set to 0.5 to randomly sample half of the training data prior to growing trees which prevent overfitting range: (0,1]
- colsample_bytree, colsample_bylevel, colsample_bynode (Default=1). range of (0, 1].
- colsample_bytree - subsample ratio of columns included when constructing each tree. Using all features, the model would converge in a shorter tree.
- colsample_bylevel - subsample ratio of columns included when constructing each level of a tree. Subsample occurs at every new level of depth in a tree.
- colsample_bynode - subsample ratio of columns for each node (split). Subsampling is done at each new split.
- reg_lambda - L2 regularization term on weights. It promotes smaller weights in the leaf nodes. The optimal weights on the leaf node is calculated by the equation 3.14 also mentioned below. Increasing the λ will decrease the weights on the leaf nodes. It is used to prevent overfitting. It serves as a benefit of XGBoost (default=1)

$$w_j^* = \frac{-G_j}{H_j + \lambda}$$

Learning Task Parameters

- objective - Loss function used in the machine learning problem to be solved. (Default=reg:squarederror for regression problems.

  binary:logistic for logistic regression of binary classification)
- eval_metric - The metric to be used for validating data (Default values are rmse for regression and error rate for classification.
  - **rmse**: root mean square error

  - **mae**: mean absolute error
  - **logloss**: negative log-likelihood
  - **error rate**: Binary classification error rate. It is calculated as (wrong cases)/(all cases) (0.5 threshold)
  - **merror**: Multiclass classification error rate
  - **mlogloss**: Multiclass logloss
  - **auc**: Area under the curve

- Seed - The randomness factor. Can be used to generate reproducible results. **(Default=0)**

3.1.2 Natural Gradient Boosting (NGBoost)

3.1.2.1 Gradient Descent and its pitfalls

Machine learning model produces output (let say $\hat{y}$) for the set of input x and its corresponding actual outputs y. So, the error between the actual output and predicted output is given by $(\hat{y} - y)$. The Cost function of the model evaluates the performance of the model. The aim is to minimise the Cost Function. In other words, lesser the cost function more efficient is the algorithm. It is an average of all the loss functions taken over all the training examples. The 'loss' taken here is the squared error difference of the actual and the predicted output.

$$Cost = \frac{1}{n}\sum_{i=1}^{n}(\hat{y} - y)^2$$

where n is the number of samples or data points on all variables, y is the vector of observed values of the variable being predicted and with $\hat{y}$ are the predicted values. The aim of the machine learning model is to minimize the cost/loss function as the lowest error/loss would result in the best model. To minimize the cost function, x has to be such that it produces the lowest y. Gradient descent helps to achieve this minimum point. The principle of gradient descent was proposed in the year 1847 as stated in (Lemarechal, 2012). It is an algorithm that works in an iterative manner to get the minimum point of a function. To reach to the minima, the direction in which the function's minimum value can be reached, needs to be known. This is done by finding out the slope of the function by taking a derivative at a particular point. This derivative or slope is known as the 'gradient' of a function.

Mathematical Interpretation of Cost Function

Suppose there is a function $y = mx + b$, with x as input and y as its output with the parameters of the function as m and b. The function is trained by the machine learning model such that the parameters change by a small factor of d. The new updated parameters are $m = m - dm$ and $b = b - db$. The goal is to identify the values of parameters, such that the error is lowest (that is, the values of m and b that reduce the cost function to its minimum).

Parameters with small changes:

$$m = m - dm$$

$$b = b - db$$

Given Cost Function for N no of samples:

$$Cost = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Cost function is denoted by J where J is a function of m and b (J is a function of y and $y = mx + b$):

$$J_{m,b} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Since $\hat{y} - y$ is nothing but the error, substituting it with the term 'Error' for simplicity :

$$J_{m,b} = \frac{1}{n} \sum_{i=1}^{n} (Error_i)^2$$

Learning Rate

Step size taken to reach to the minima of the function is called the Learning Rate. There are two ways the minima can be reached. Firstly, by taking larger steps which is obtained by setting higher learning rate but in that case there is a possibility of overshooting (skipping) the minimum value. Secondly by taking smaller steps that is obtained by low learning rate but with a great amount of time taken, with a chance that minimum value is never reached.

Let say there is some function having some parameters. It is the value of the parameters that must be changed in order to optimize the function. For such case there is a loss/cost function that tells how the parameters should be changed. The derivatives of each parameter are calculated with respect to the loss function. The derivative (gradient) tells in which direction to move to attain the minimum loss function. The parameters are adjusted such that it moves in the direction of the gradient by a certain step size, termed as the 'learning rate'. That step size or distance is actually an Euclidean distance, in the parametric space for example, if the parameters are a and b then the Euclidean distance $= \sqrt{(a^2 + b^2)}$. Here the parameters are adjusted according to the parametric distance.

Pitfall of gradient descent

Gradient descent is just the first order derivative of the loss function, it does not mention anything about the curvature of the function or how quickly the derivative of the function is changing. The specific step size to descend is not determined by the gradient descent. Generally, the parameters are adjusted using the gradient for the direction and using the learning rate for step size. In Figure 3.2 there are two points x_0 and x^*. x_0 is at flatter surface that means it has a smaller gradient and x^* is at a steeper surface that means at a higher gradient. But this does not specify smaller step size or larger step size to take respectively. Only the direction is known and the step size is managed by the learning rate hyperparameter. By controlling the learning rate, it is made sure that the minima is not skipped or never reached.

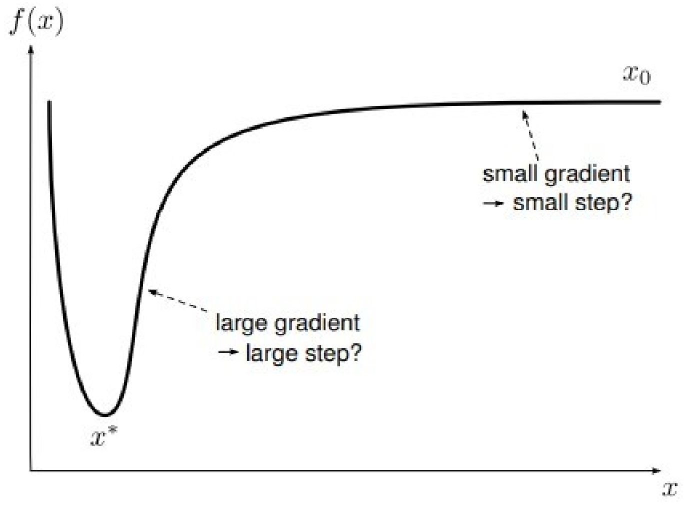


Figure 3.2: Pitfall of gradient descent - what step size to take?

Not having an exact step size according to the point on the function is a pitfall found in almost all first order optimisation algorithms. One solution is to find the second order derivative of the function that could tell the curvature and update the steps accordingly, this method is known as the Newton Raphson method used in XGBoost. But second order optimisation can make the computation complex.

And if the change is made using learning rate, it will scale all the parameters with the same rate. There could be few parameters influencing more on the loss function than the other parameters. Updating all with the same rate is to restrict/limit the update for more influencing parameters. In such case algorithm takes longer time to converge. To solve this issue natural gradient comes into play.

3.1.2.2 Natural Gradient

Natural gradient redefines the ”small distance” by which the parameters are updated. Rather than working on the distance in terms of parametric space (Euclidean distance), natural gradient works on the distance in terms of distribution space of the output. Generally, the parameters are changed within an epsilon distance, but in natural gradient the distribution is changed within an epsilon distance from the distribution obtained in previous step. This distance is based on KL divergence, a measure of how much a new distribution differs from the old one. The Natural gradient moves in distribution space rather than in parametric space.

KL Divergence

KL divergence (Kullback and Leibler, 1951) is defined as a divergence of one probabilistic distribution from another probabilistic distribution. $KL(Q||P)$ is the information lost when approximating Q (True Distribution) with P (Predicted Distribution).

General machine learning model works, to approximate a probability distribution and output the expected value of the probability distribution (mean). Devising a model (NGBoost) based on a probability distribution is like preparing a model to output the complete distribution.

Since KL divergence is a measure of difference between two probability distributions, the concept of entropy and cross entropy is used. Entropy is defined as a chance of a specific event to occur, which can give some information about the underlying probability distribution. Entropy $H(p)$ for p distribution is given by-

$$H(p) = -\sum_x p(x) log_2(p(x))$$

Cross entropy states the degree of dissimilarity/difference between two probability distributions. Cross Entropy of q with respect to p, where q is the true distribution and p is the predicted distribution is given by-

$$H_p(q) = -\sum_x q(x) log_2(p(x))$$

In case the models give perfect predictions that is, the predicted distribution is equal to the true distribution, then cross entropy will be equal to the entropy. But if both the distributions are different, then the cross entropy will be greater than the entropy. The amount by which the cross entropy differs the entropy is known as relative entropy or KL-Divergence.

Ideal entropy will be the entropy of true distribution i.e. entropy of Q.

$$H(q) = -\sum_x q(x) log_2(q(x))$$

Since we have the actual Cross Entropy and the ideal Entropy, the information lost is-

$$\begin{aligned}
H_p(q) - H(q) &= -\sum_x q(x) log_2(p(x)) - (-\sum_x q(x) log_2(q(x))) \\
&= \sum_x q(x) log_2(q(x)) - \sum_x q(x) log_2(p(x)) \\
&= \sum_x (q(x) \cdot (log_2(q(x)) - log_2(p(x)))) \\
KL(Q||P) &= \sum_x q(x) \cdot log_2(\frac{q(x)}{p(x)})
\end{aligned}$$

Replacing the summations with integration Instead of P and Q, let's take $p(x;\theta)$ and $p(x;\theta+\partial\theta)$ as the two distributions where θ be the parameters of the loss function, $\partial\theta$ be the small change in parameters and x is input features or covariates in a gradient update step. So, KL-Divergence is defined as:

$$KL\big(p(x;\theta)||p(x;\theta+\partial\theta)\big) = \int p(x;\theta) \cdot log\left(\frac{p(x;\theta)}{p(x;\theta+\partial\theta)}\right) dx$$

Fisher Matrix

Fisher Information Matrix is defined as the local curvature in the distribution space. It is derived by taking the second order derivative (Hessian) of the divergence of predicted distribution from actual distribution calculated by KL-Divergence. By exploring the second order derivative, it is giving an added information of curvature in the distribution space. Besides the direction, the curvature helps to move precisely in the distribution space.

Relation between KL Divergence and Fisher Matrix

As stated in (Ly et al., 2017) Fisher matrix F is the Hessian of the KL divergence. Doing the Taylor Series expansion of KL Divergence, and replacing the second order derivative in the expanded equation, with the Fisher matrix. KL Divergence equation formed is as follows-

$$KL[p(x|\theta)||p(x|\theta+d)] \approx \frac{1}{2} d^T F d$$

where d is a certain factor by which parameter θ is changed in the distribution space and F is the fisher matrix.

Maths behind Natural Gradient

The goal is to find the update vector 'd' that minimises the loss function $L(\theta)$ in the distribution space. To summarize:

$$KL[p(x|\theta)||p(x|\theta + d)] \approx \frac{1}{2}d^T F d$$

is the function whose minima must be reached. It can be accomplished by finding out the update vector parameter 'd' such that it minimizes the loss function. This is similar to gradient descent, the only difference is that it is in distributional space with KL divergence as a metric, instead of the parametric space with Euclidean space as a metric.

Update vector 'd' is given by the formula-

$$d = -\frac{1}{\lambda}F^{-1}\nabla_\theta L(\theta)$$

$\frac{1}{\lambda}$ can be assumed as a constant factor (learning rate) at which the gradient descent has to be done. F^{-1} gives the curvature of the gradient in distribution space.

Natural gradient is defined as:

$$\tilde{\nabla}_\theta L(\theta) = F^{-1}\nabla_\theta L(\theta)$$

Algorithm 1: Natural Gradient Descent

```
1. Result:
     I   Do forward pass on out model and compute loss $L(\theta)$.
     II  Compute the gradient $\nabla_\theta L(\theta)$
     III Compute the Fisher Information Matrix F, or its empirical version (w.r.t our
         training data).
     IV  Compute the natural gradient $\tilde{\nabla}_\theta L(\theta) = F^{-1}\nabla_\theta L(\theta)$.
     V   Update the parameter: $\theta = \theta - \alpha\tilde{\nabla}_\theta L(\theta)$, where $\alpha$ is the learning rate.
2. Until Convergence.
```

3.1.2.3 Gradient Vs Natural Gradient

The difference between Natural gradient and normal gradient is necessary to be understood in order to know the difference between Natural gradient descent and gradient descent. The main change lies in the loss function which is a measure of difference between actual outcome and predicted outcome. The loss function is changed by updating the parameters that changes the output and apparently the loss. In a 'normal' Gradient, the derivative of loss function is taken with respect to the parameters. There might be the case that even if each parameter is changed by one unit, the probability distributions of the predicted and the actual outcomes are much closer. In such case the derivative which is calculated will be small. The learning rate is applied to the parameters in gradient, such that the parameters are updated by a fixed amount.

But in natural gradient, the movement of parameters is not restricted to the parametric space. Rather, the output probability distribution movement is captured at each step. The probability distributions are measured through Log Likelihood where Fisher Matrix calculates the curvature of the Log Likelihood. It is noted that normal gradient does not take into consideration any curvature of the loss function as it is a first order derivative method. It is the Fisher matrix which when applied to the gradient, scales the parameter updates with the curvature of the log likelihood function.

Therefore, natural gradient has two benefits. Firstly, it adjust the parameters according to the curvature information. Secondly, despite restricting the movement in parametric space assuming that the movement in the prediction space will also be restricted, the Natural Gradient directly updates the movement in the prediction space in KL Divergence terms.

Recently, it has also been used in a form of Gradient Boosting, resulting in the algorithm named as NGBoost.

3.1.2.4 NGBoost

The idea of Natural Gradient Boosting is to take the base learner and train it on the training samples (probabilistic distributions), such that it minimizes the scoring rule. Natural gradient descent is used for optimisation, and a base learner is a “set of weak learners” going to be trained.

When the boosting algorithm fits the natural gradient in place of regular gradient it is known as NGBoost. It models the probability distribution over the output space. Any kind of distribution is defined by a certain set of parameters like for normal distribution it comprises of (mean and standard deviation). In NGBoost the whole distribution is fit instead of any single parameter thus, NGBoost is a combination of multiparameter boosting and natural gradient. There are three components to this algorithm - base learner, parametric probability distribution, and scoring rule.

Base Learners: Base learners are weak learners that takes input x and the outputs formed are used to generate the conditional probabilities such as Decision Tree and Ridge regression.

Parametric Distribution: The model here fits the full parametric distribution as the predictions instead of point predictions. In normal prediction the estimation is $E[y|x]$, whereas in probabilistic predictions the estimation is $P(y|x)$. According to the data specified, $P(y|x)$ is of a specified parametric form.

Scoring Rule: Scoring rule is used to compare the predicted probability distributions with the actual data. Scoring rule, ‘S’ takes input, a pair of predicted probability distribution P and actual outcome y, in the form of a score $S(P, y)$. Mostly log score ‘L’ is used as the scoring rule, the log score when minimized give Maximum Likelihood Estimation which is a log likelihood. And the scoring rule is parameterised by θ because that is what we are predicting as part of the machine learning model.

$$L(\theta, y) = -logP_{\theta}(y)$$

Another example is CRPS(Continuous Ranked Probability Score) (Gebetsberger et al., 2018). While the logarithmic score or the log likelihood generalises Mean Squared Error to a probabilistic space, CRPS does the same to Mean Absolute Error.

$$C(\theta, y) = \int_{-\infty}^{y} F_\theta(z)^2 dz + \int_{y}^{\infty} (1 - F_\theta(z))^2 dz$$

Summary:
We train base learner to output for each training sample such probability distribution that minimises the proper score. Natural gradient descent is used as optimisation algorithm and the base learner is a collection of weak learners trained with boosting approach.

3.1.2.5 NGBoost Hyperparameters

NGBoost hyperparameters defined under scikit library of Python language are mentioned in Table 3.1.

Table 3.1: NGBoost Hyperparameters Summary

n_estimators	The number of boosting iterations. Default: 500. The number of trees built before taking averages of the predictions. Higher values may make the model complex and would increase the chance of overfitting
learning_rate	The learning rate. Default:0.01
minibatch_frac	The percent subsample of rows to use in each boosting iteration. This is more of a performance hack than performance tuning. When the data set is huge, this parameter can considerably speed things up.
col_sample	the percent subsample of columns to use in each boosting iteration
Dist	This parameter sets the distribution of the output. Currently, the library supports Normal, LogNormal, and Exponential distributions for regression, k_categorical and Bernoulli for classification. Default: Normal
Score	This specifies the scoring rule. Currently, the options are between LogScore or CRPScore. Default: LogScore
Base	This specifies the base learner. This can be any Sci-kit Learn estimator. Default is a 3-depth Decision Tree
In addition to these parameters one can also tune any parameter related to the base learner chosen. For instance, if Base = sklearn.tree.DecisionTreeRegressor(), then one can tune any of those parameters too.	

3.2 Hyperparameter Tuning

A hyperparameter is a parameter, the value of which is set before starting the process of learning. To tune machine learning hyperparameters is a repetitive and important task, as an algorithm 's performance can depend heavily on the selection of hyperparameters. Manual tuning requires a lot of time as it includes steps like trying different values of hyperparameters and interpret results on different values of manually tuned hyperparameters. Grid search and random search are methods that give the best suitable hyperparameters without manually tuning. These techniques are handy, but they might require long run times in assessing search space in the parameter space to output the best possible results. Progressively in this work, hyperparameter tuning is achieved through automatic approaches aimed at discovering optimum hyperparameters in less time. The main objective in the case of hyperparameter optimisation is to obtain a minimum error on a machine learning output.

3.2.1 Cross Validation

Searching the hyper-parameter environment is feasible and suggested for the best Cross-validation i.e. assessing the performance score of the estimator at each validation. Partition the data in K-fold cross validation technique into k-bins, marking one of the k-bin as test set and the rest of the 'k-1' bins are inserted into the training set. Machine learning model is trained and the results are checked on the test set just as before. In cross validation, the critical point is that we run this 'k' times and then averaging the performance for the different hold out sets. Combining the test outcomes from certain experiments 'k', naturally requires more computing time because 'k' different learning experiments must run now, but the learning of the model happens to be more precise. The Random Search or Grid Search can pick hyper parameter values without k-fold cross validation and works pretty good on the split data of the sample train check but there is a strong chance that it would operate poorly for unknown datasets due to large variance.

Note that the cross validation is best suited for the large datasets for the training purpose. In this work, while searching for the best hyperparameters, randomsearchcv() takes 5-fold cross validation by default. But this is only for searching the best hyperparameters.

The model is fitted with the train-test 80/20 split in case of small dataset used in this study. It is only in case of larger dataset where 10-fold cross validation is used to fit the model.

3.2.2 Hyperparameter search techniques

GridSearchCV

It is a function provided under Scikit Learn library of python to find out the best hyperparameters of a model. For given values, GridSearchCV exhaustively considers all parameter combinations. The grid search manner is used by GridSearchCV and this exhaustively generates candidates from a grid of parameter values specified with the param_grid. In this method all the combinations of the parameters present in the parameter grid are tried. After finishing all the combinations, the combination with the best accuracy is marked as the best. It works on the grid search space. It proves to be simple methodically but exhaustive in nature consuming high amount of computational power.

RandomizedSearchCV

Another function to find out the best hyperparamters provided under Scikit Learn library of python. It picks up the random combinations of hyperparameter values from the parameter grid and samples it on a set of inputs. In contrast to grid search, it does not try all the combinations of parameters sequentially. The randomness is configured initially before the random search starts. In this we can control the number of iterations based and time and space resources. In small datasets grid search can give good results. Also, in large datasets the randomsearchcv proves to be computationally efficient, as it does not have to try all the possible combination to reach to the best result. The chances of finding the best hyperparameters is higher and quicker in the process.

3.3 Model Evaluation

We have many ways to measure performance of the prediction, where some are more suitable than the others depending on the application considered. A brief discussion on the performance measures is explained below.

Root Mean Squared Error

RMSE is a popular formula to measure the error rate of a model. It is used for regression problems and calculated as an average error that occurs while predicting the outcome. It is the square root of mean squared error. It can only be compared between models whose errors are measured in the same units. It is calculated as follows:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}}$$

Where , y_i is actual value, $\hat{y}_i$ is predicted value and N is the total number of instances.

Coefficient of Determination (R2)

The coefficient of determination (R2) summarises the explanatory power of the model and is computed from the sums-of-squares terms. Determines variability of the dependent variable that can be explained by the model. The higher the R^2 the better the model. R^2 value lies between 0 and 1.

3.4 Model outputs

Point Estimations

Generally, predictions made by any machine learning models are point estimates. It gives an estimate of a target variable at a particular point of input features. Take a sample and find $\overline{x}$ (close approximate value of μ). In case we want more confidence about the predictions, the confidence intervals and prediction intervals can be used.

Confidence Intervals

Broader range and are more precise than point estimates. Confidence intervals are based on inferential statistics i.e. it states that with certain confidence the value of the population parameter lies. The degree of confidence determines how much the confidence interval includes the parameter. 95 percent is widely used which indicates that 95 percent of the confidence intervals contain the parameter in repetitive sampling.

Prediction Intervals

Prediction intervals are the uncertainty interval on each point prediction rather than taking the interval on the whole population of the output. It is a set of range calculated from the distribution of the output data at each point. It accounts for both - the variation in the point estimates and uncertainty in the population mean. Thus, the prediction interval is broader in range then that of confidence intervals.

3.5 Environment Setup

Cloud machine platform 'Amazon SageMaker' is an IDE (*Amazon SageMaker* 2019) used to create, train and deploy machine learning models in the cloud. Amazon SageMaker has inbuilt libraries for XGBoost installation, hyperparameters tuning and evaluation. Moreover, the presence of GPU machines enables the processing of the models to be faster using multiple cores parallelly. Jupyter notebook is used as a coding platform (The Jupyter Notebook — Jupyter Notebook 6.1.3 Documentation, n.d.). It is an open source web application that you can use to create and share documents that contain live code, equations, visualisations, and text. Python was used to implement and execute the methodology described. Python version 3.6 environment comes with many helpful analytic libraries installed which simplifies data preparation, prep-processing and analysis. Furthermore, three python libraries named Scikit-learn, Keras, and the DMLC XGBoost were used. Scikit-learn was used for grid-search, random-search-cv, cross-validation, and built-in machine learning metrics (Pedregosa et al., 2011).

Chapter 4

Implementation and results

Implementation and Results chapter aims to give the details about the datasets used for the analysis, case studies related to the two datasets and the result summary of the case studies analysed. The case studies include the details about the workflow used during the analysis. The workflow details include all about analysing the models on the default hyperparameters, finding out the best tuned hyper parameters, analysing the models on best hyperparameters, finding out the GPU/CPU performance, stating the RMSE values, confidence intervals, feature importance and prediction intervals on both the datasets. The result summary shows the consolidated values of accuracy metrics used for comparing the two models.

4.1 Dataset Analysis

4.1.1 Boston Housing Data Set

Boston housing dataset was created by U.S Census Service related to the housing in the Boston area. It is present in the UCI Machine Learning Repository. It was obtained from StatLib archive[1] . Data was logged in the year 1978 containing 506 entries representing the aggregated data of features related that decide the home price from several suburbs

[1]http://lib.stat.cmu.edu/datasset/boston

in the area. There are no missing or duplicate values. For descriptive statistics of the dataset refer appendix Table A.1.

Table 4.1: Boston Dataset

Number of Instances: 506	Number of attributes: 14	Target variable: MEDV

Variables

CRIM - per capita crime rate by town

ZN - proportion of residential land zoned for lots over 25,000 sq. Ft.

INDUS - proportion of non-retail business acres per town.

CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)

NOX - nitric oxides concentration (parts per 10 million)

RM - average number of rooms per dwelling

AGE - proportion of owner-occupied units built prior to 1940

DIS - weighted distances to five Boston employment centres

RAD - index of accessibility to radial highways

TAX - full-value property-tax rate per $10,000

PTRATIO - pupil-teacher ratio by town

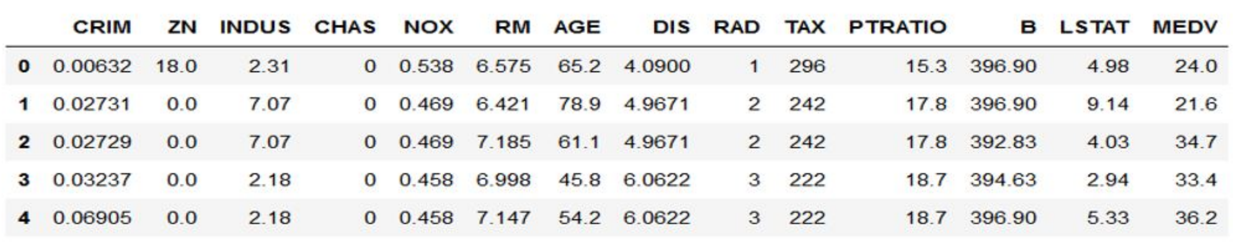
B - $1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town

LSTAT - % lower status of the population

MEDV - Median value of owner-occupied homes in $1000's. It is a target variable.

Table 4.2 shows the first five rows of the boston housing dataset displaying the type of values each variable stores.

Table 4.2: Boston Dataset Summary



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

Boxplots for Boston Housing dataset features

Boxplots show no significant outliers in the input variables.

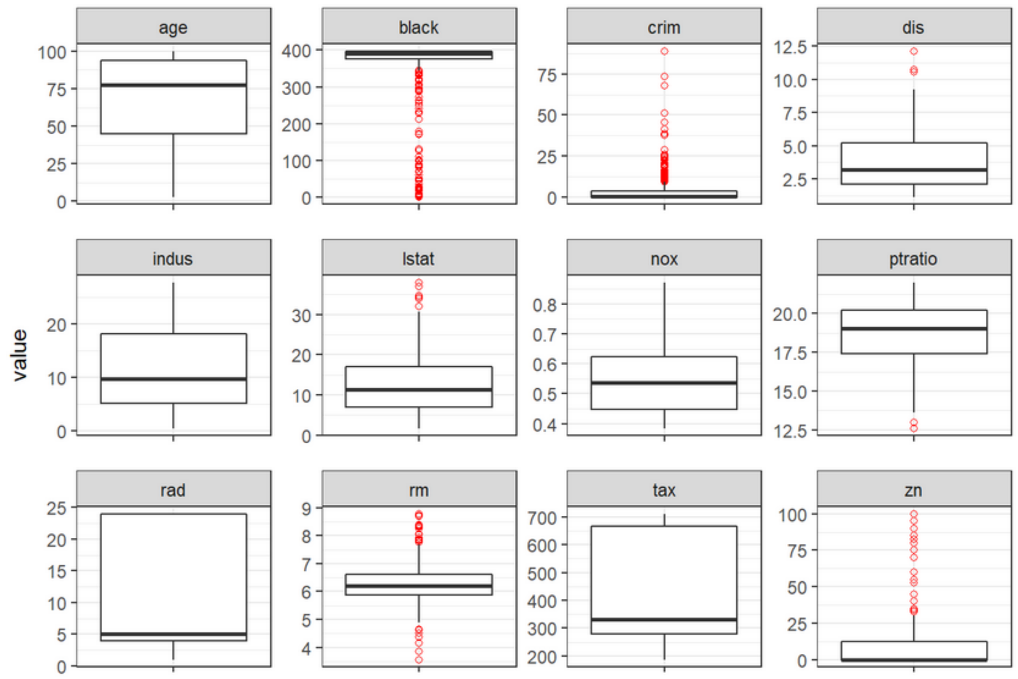


Figure 4.1: Boxplot description of features of Boston housing data

Distribution for target variable 'MEDV'

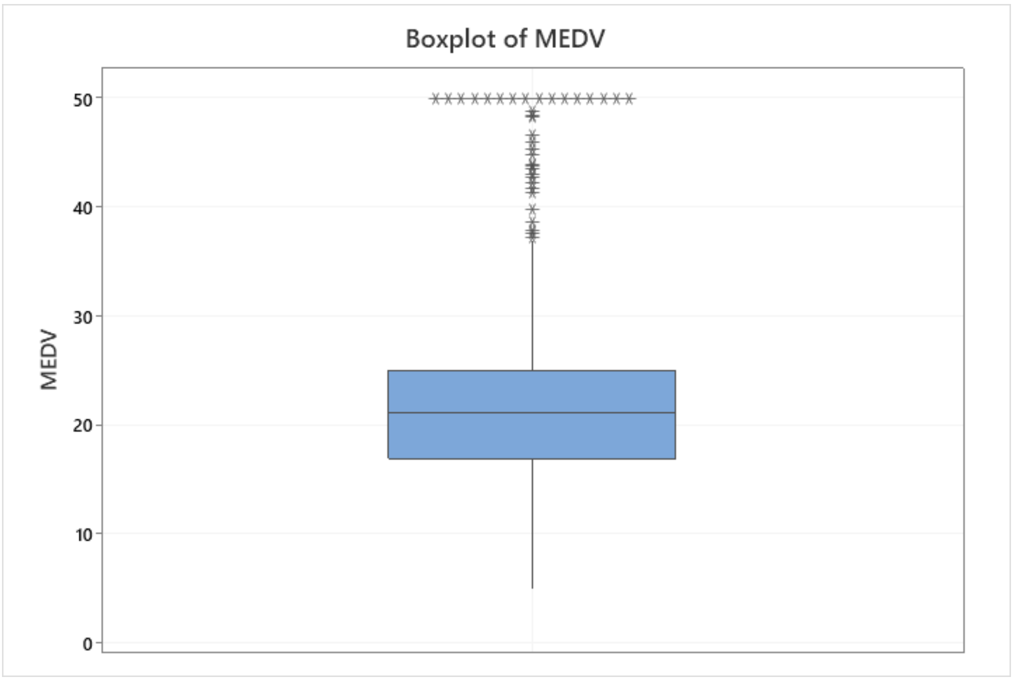


Figure 4.2: Boxplot showing interval scale of 'MEDV' target variable of Boston Housing

The target variable 'MEDV' Median value of owner-occupied homes, is censored or capped at 50.00. The Census Service of the US has censored the data. By censoring it means the range of the possible values of a variable is restricted just like in this case the maximum value of the price variable is set to 50k USD, such that no price can cross that limit. As seen in Figure 4.2, a sort of ceiling is formed that flattens the data points at 50. In reality, these prices would probably be greater than 50.

In such a case it's almost impossible to find out in future, what the actual prices of the house properties might be. Although it is a minor proportion of the total, the potential impact of this on the training model is important to be noted. Considering them as potential outliers, these values are dropped prior to training of the model. In total 16 such values are dropped. As the number of observations for 50K USD are not much, dropping these would not cause much effect on the learning of the model.

Detecting correlation between the features of Boston Housing data

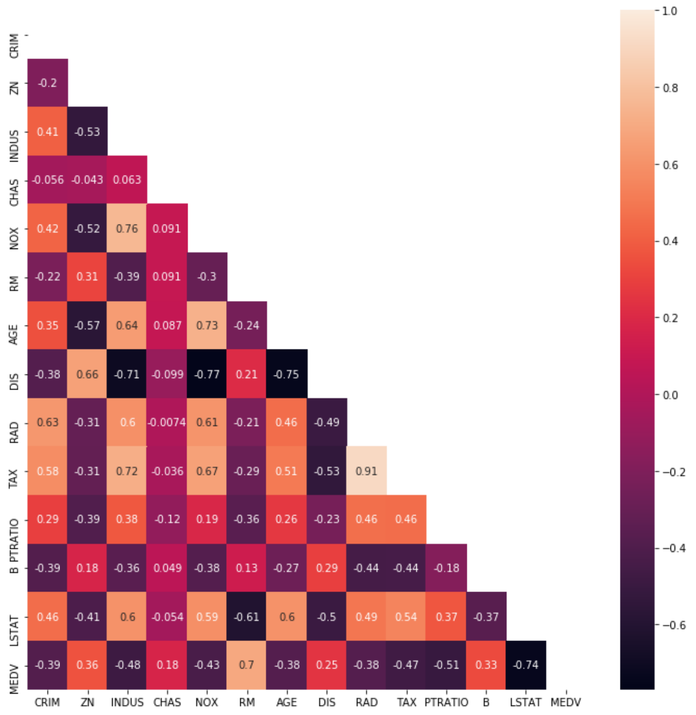


Figure 4.3: Correlation matrix of the features of Boston Housing data

From correlation matrix in Figure 4.3, some of the observations are made:

- Target variable 'MEDV' which is a median value of owner-occupied homes (in 1000$) increases as 'RM' (which is an average number of rooms per dwelling) increases and decreases if ‘LSTAT’ (which is lower status population in the area) increases.

- ‘NOX’ which is nitrogen oxides concentration (ppm) increases as the variable ‘INDUS’ increases which is the proportion of non-retail business acres per town. ‘NOX’ also increase as the ‘AGE’ increase which is the proportion of owner-occupied units built prior to 1940.
- ‘RAD’ and ‘TAX’ have a correlation of 0.91 which is a positive correlation, implying that as index of accessibility to radial highways increases, the property-tax rate per $10,000 also increases.
- ‘CRIM’ is strongly correlated with the variables like ‘RAD’ and ‘TAX’ stating that accessibility to radial highways increases, as per increase in capita crime rate.
- ‘INDUS’ strongly correlated with ‘NOX’, stating higher concentrations of nitrogen oxides in industrial areas.

4.1.2 Physicochemical Properties of Protein Tertiary Structure Data Set

Physicochemical Properties of Protein Tertiary Structure is drawn from CASP 5-9 prediction centre[2] . Dataset contains 45730 entries having 9 features and one response variable ’RMSD’ (Root-Mean-Square Deviation) of protein structure. The target variable is a score to determine the protein structure. It is an averaged distance between the atoms of the modelled protein structure when the true native state of the protein structure is not known. It is a difference between two structures. For descriptive statistics of data refer appendix Table A.2.

Table 4.3: Protein Dataset

Number of Instances: 45730	Number of attributes: 10	Target variable: RMSD

[2]https://archive.ics.uci.edu/ml/datasets/Physicochemical+Properties+of+Protein+Tertiary+Structure

RMSD-Size of the residue.

F1 - Total surface area.

F2 - Nonpolar exposed area.

F3 - Fractional area of exposed nonpolar residue.

F4 - Fractional area of exposed nonpolar part of residue.

F5 - Molecular mass weighted exposed area.

F6 - Average deviation from standard exposed area of residue.

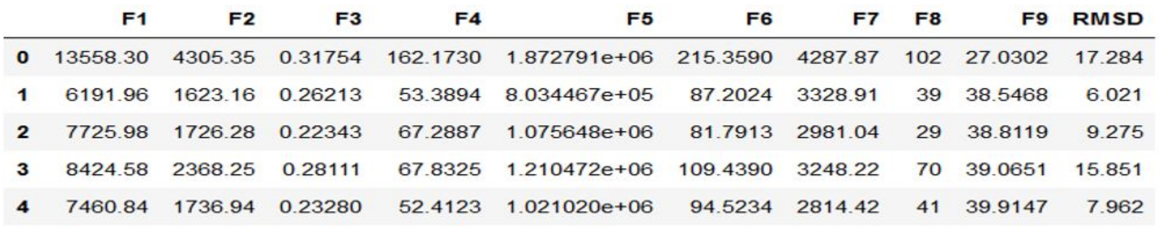
F7 - Euclidian distance.

F8 - Secondary structure penalty.

F9 - Spatial Distribution constraints (N,K Value)

Table 4.4 shows the first five rows of the protein structure dataset displaying the type of values each variable stores.

Table 4.4: Protein Dataset Summary



	F1	F2	F3	F4	F5	F6	F7	F8	F9	RMSD
0	13558.30	4305.35	0.31754	162.1730	1.872791e+06	215.3590	4287.87	102	27.0302	17.284
1	6191.96	1623.16	0.26213	53.3894	8.034467e+05	87.2024	3328.91	39	38.5468	6.021
2	7725.98	1726.28	0.22343	67.2887	1.075648e+06	81.7913	2981.04	29	38.8119	9.275
3	8424.58	2368.25	0.28111	67.8325	1.210472e+06	109.4390	3248.22	70	39.0651	15.851
4	7460.84	1736.94	0.23280	52.4123	1.021020e+06	94.5234	2814.42	41	39.9147	7.962

Boxplots for Protein structure Dataset features

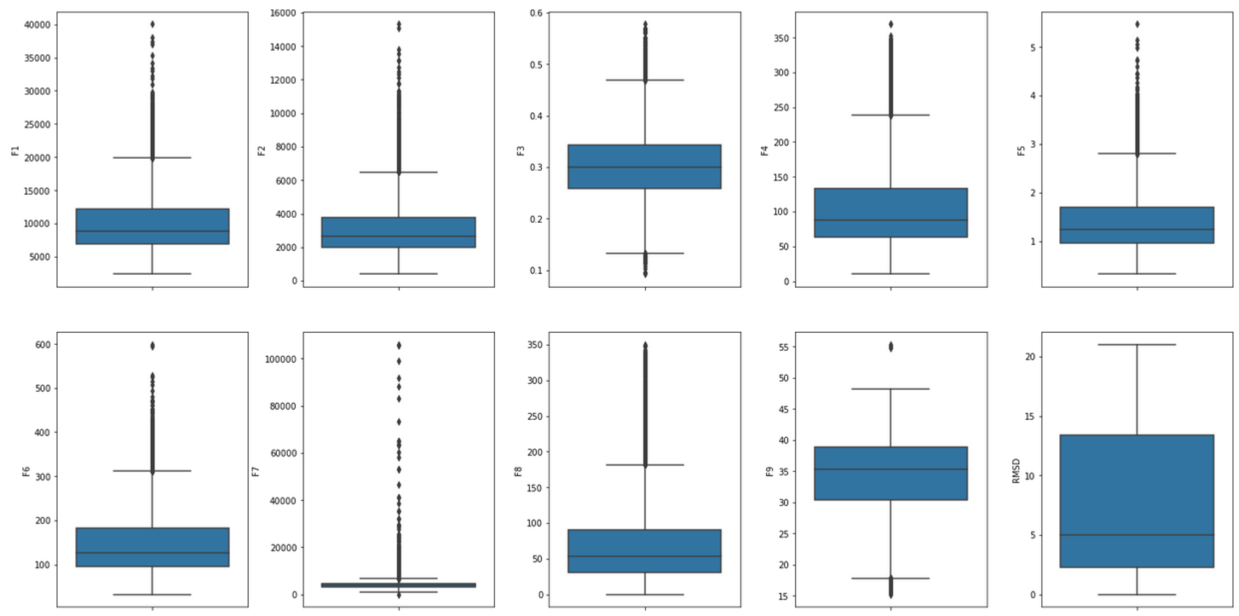


Figure 4.4: Boxplot showing interval scale of variables of Protein structure data

Detecting correlation between the features of Protein structure data

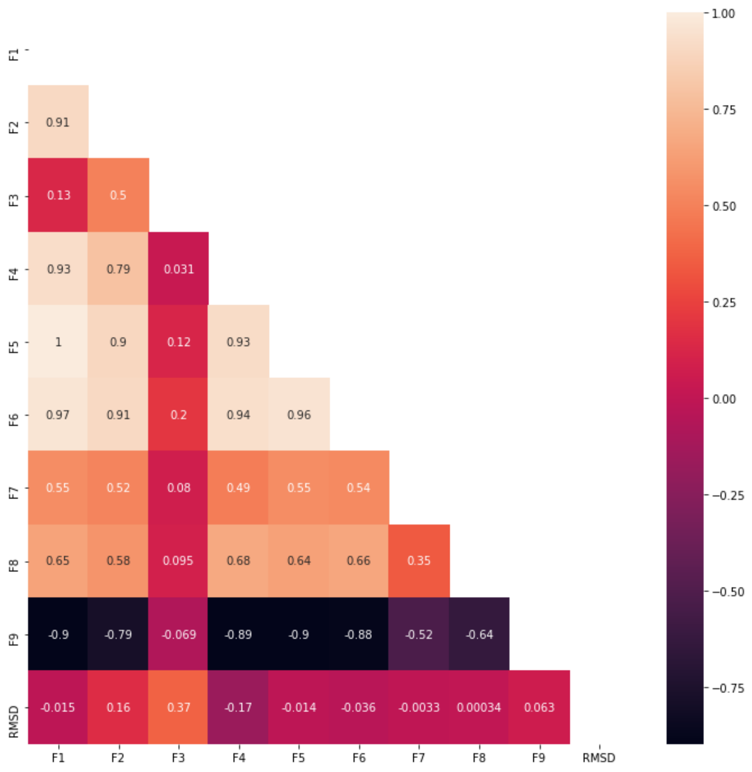


Figure 4.5: Correlation matrix of the features of Protein structure data

The correlations between **RMSD** and the other characteristics are shown in figure 4.5, where it can be seen that the features are highly correlated with each other except with RMSD. RMSD's highest correlation is with **F3** (Fractional area of exposed non-polar residue). Figure 4.4 shows the individual boxplots of the features showing no such outliers except 'F7' that shows categorical values.

Both the datasets, Boston Housing Data and Protein Structure Data, used for this work are well known, structured, not much pre-processing required and are easily available. Most of the works had comparative analysis done on these datasets which set these datasets as the standard to do basic comparisons. The original paper of NGBoost (Duan et al., 2019) has these two datasets in their experiments that makes easy to compare the results of the present analysis with the original results. Since NGBoost is a new algorithm, the results in the original paper can act as the base results on these datasets.

4.2 Experiments and Results

4.2.1 Case Study 1: Regression Analysis of Boston Housing Data Set

Data is split into training and test data in the ratio of 8:2, which is 80% of data is used for training the model. Firstly, it is checked how the algorithms behave when trained using default hyperparameters. On calculating the RMSE for the test set for XGBoost, it is coming as 2.40. NGBoost gives similar RMSE which is 2.55. XGBoost having extensive set of hyperparameters, shows a little better results with somewhat lower RMSE.

To analyze if the performance can be improved, the best tuned parameters have to be found out for each model for which Randomsearchcv is applied using 5-fold cross validation (which is a default validation of randomsearchcv()) and 30 iterations, each algorithm having different set of parameters in the parameter grid from the range specified by both the algorithms.

Table 4.5 shows the output of randomsearchcv showing best tuned hyperparameters and the best score (root mean square error – as default score of randomsearchcv) corresponding to each model. XGBoost takes learning rate as 0.1 and n_estimators as 500, and NGBoost takes learning_rate as 0.05 and n_estimators as 900. The learning in XGBoost happens only in 500 estimators, that can be the optimum choice as, where n_estimators are concerned, most of the learning/gain happens early; the improvements casually decrease over time, and there may be a risk of overfitting. For XGBoost, 'max_depth' =5, and for NGBoost, 'max_depth' =10 that shows deeper trees grown by NGBoost, that could make the model more complex. XGBoost also uses the parameter reg_lambda, which is a regularization parameter that balances the model. The best scores while finding the best parameters through randomsearchcv() are 2.71 (root_mean_squared_error) for XGBoost and 2.80 (root_mean_squared_error) for NGBoost. The error is slightly low in the case of XGBoost.

Table 4.5: Randomsearchcv best parameters for both the algorithms on boston data

XGBoost	**NGBoost**
base_score=0.25, colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1, learning_rate=0.1, max_depth=10, min_child_weight=3, n_estimators=900, num_parallel_tree=1, reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,	max_depth=15, max_features=None, max_leaf_nodes=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, col_sample=0.7, learning_rate=0.01, minibatch_frac=0.05, n_estimators=500, natural_gradient=True, tol=0.0001
Best score: 2.71	Best score: 2.80

Figure 4.6 demonstrates the faster computation of XGBoost. Where NGBoost reaches 400 N_estimators in 21.4 min, it is done in 12.8 min in the case of XGBoost due to the feature of parallelization. Note that, to compare the GPU processing at the same level both the algorithms are fed with 400 n_estimators and checked. Comparing the GPU training time of the boosting algorithms for a fixed set of hyperparameters is difficult since the hyperparameters do not necessarily align, and accuracy will not necessarily match. Thus, for a fair comparison, the performance is studied in the context of hyperparameter optimization through Randomsearchcv. Specifically, to know which GPU-accelerated algorithm can attain a good validation score in the shortest time during optimization on boston housing dataset.

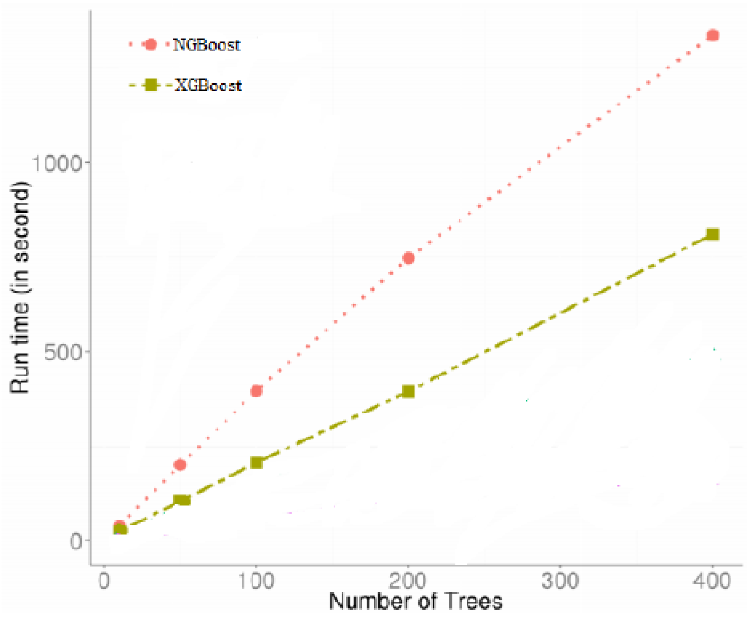
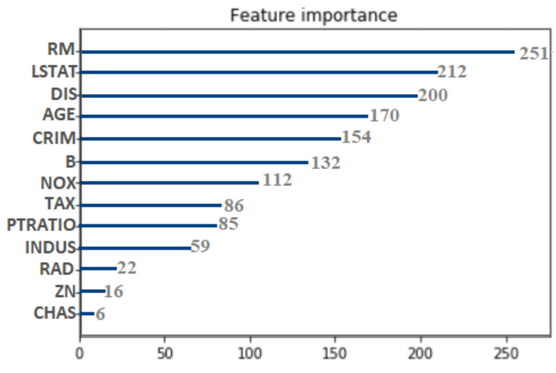
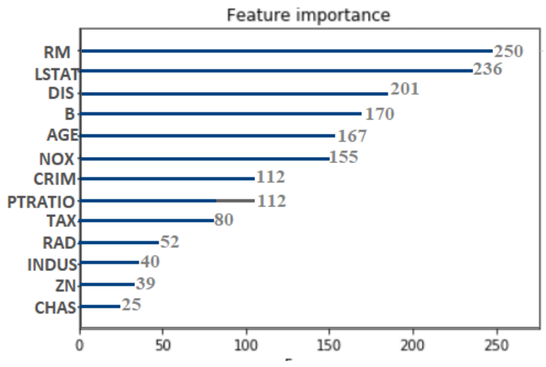


Figure 4.6: GPU time taken in training n_estimators=400 by each algorithm on boston data



(a) XGBoost



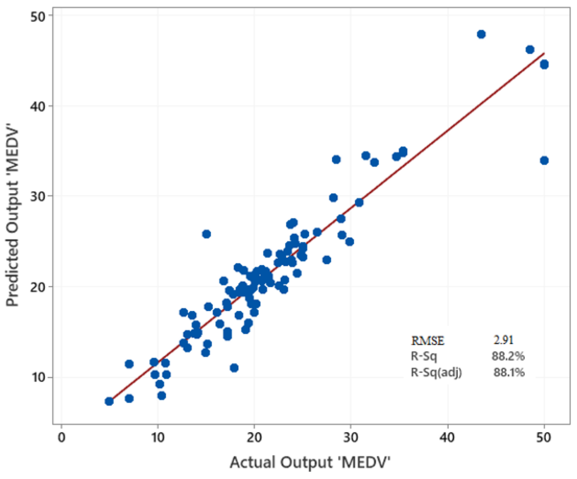
(b) NGBoost

Figure 4.7: Feature Importance graphs for boston data

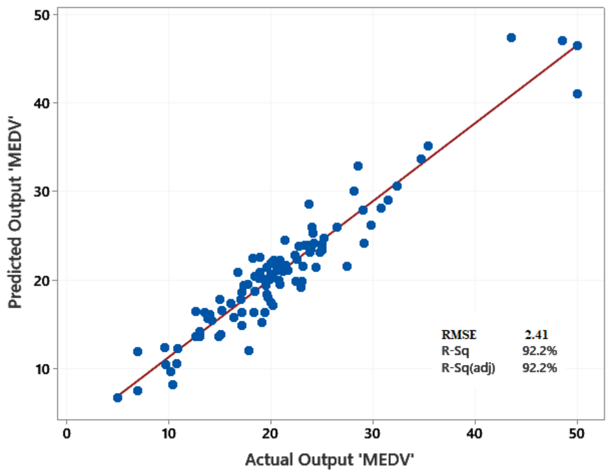
The feature importance results of both the models show similar results except few of the variables like ‘CRIM’ , ‘PTRATIO’, ‘AGE’, ‘NOX’, ‘PTRATIO’. The features are plotted on y-axis, having their feature scores on the x-axis. Feature scores is a measure that adds up the number of times each feature is split.

The final predictions made on the test set on tuned hyperparameters give the RMSE of 2.40 for NGBoost and 2.37 for XGBoost and the R-squared values as 0.88 and 0.89, respectively. It shows that XGBoost outperforms in case of point predictions where it better fits the model with the lower RMSE and comparatively higher R-squared value.

Figure 4.8 shows actual vs predicted plots generated by both the algorithms. Both seems to perform equally good.



(a) Actual Vs Fitted plot for Boston data for NGBoost



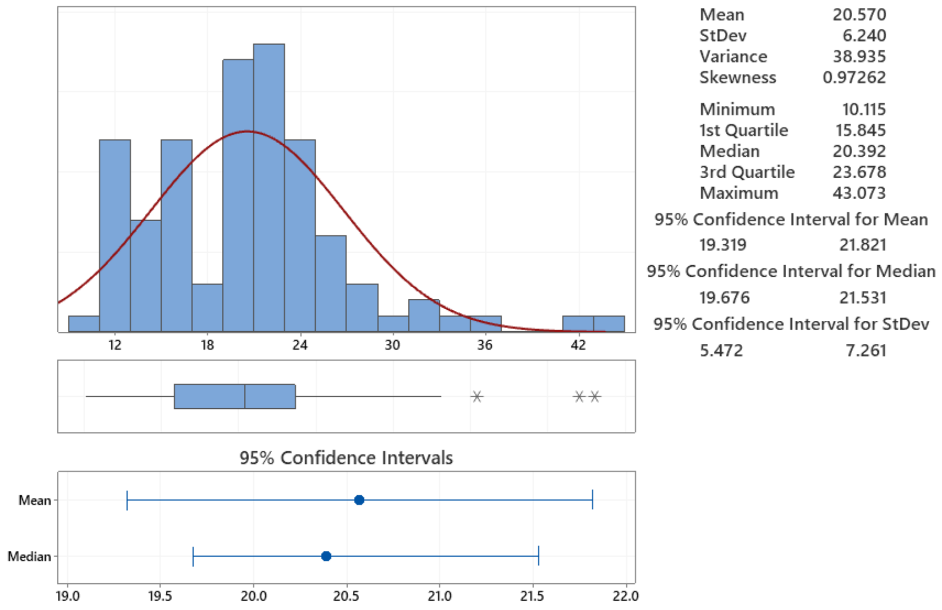
(b) Actual Vs Fitted plot for Boston data for XGBoost

Figure 4.8: Actual vs Fitted plots of predictions made on boston data

Confidence Intervals predicted on boston housing data

Through confidence intervals we can estimate the probability of true population mean lying in an interval at particular confidence. It is calculated for the predicted outputs considering them as a sample of true population (actual output). Confidence Intervals are calculated by calculating the population mean and standard deviation of the predicted output variable, then for 95% confidence, the interval is calculated by multiplying z-score (1.96 for 95% confidence interval) with the standard deviation. Once the interval is obtained, the upper and the lower range of confidence interval is calculated by adding and subtracting the interval value from the mean value. Figures 4.9 and 4.10 shows the distribution plots for the predicted variable 'MEDV'of boston housing dataset. Figure 4.11 shows the distribution plot of the actual output variable 'MEDV'. Confidence interval for XGBoost is coming as: (19.238, 21.807) (Figure 4.10) confidence interval for NGBoost is coming as (19.319, 21.821) (Figure 4.9). True mean is (20.064) (Figure 4.11) and the sample mean with 95% confidence interval is (20.523) for XGBoost and (20.570) NGBoost. The confidence interval predicted for both the algorithms is almost same. The population of the predicted variable shows similar distribution as calculated at 95%

confidence intervals. Note that these intervals is for the whole population. To calculate the intervals for each output point (instead of whole population), prediction intervals are used.



Mean	20.570
StDev	6.240
Variance	38.935
Skewness	0.97262
Minimum	10.115
1st Quartile	15.845
Median	20.392
3rd Quartile	23.678
Maximum	43.073

95% Confidence Interval for Mean

19.319	21.821

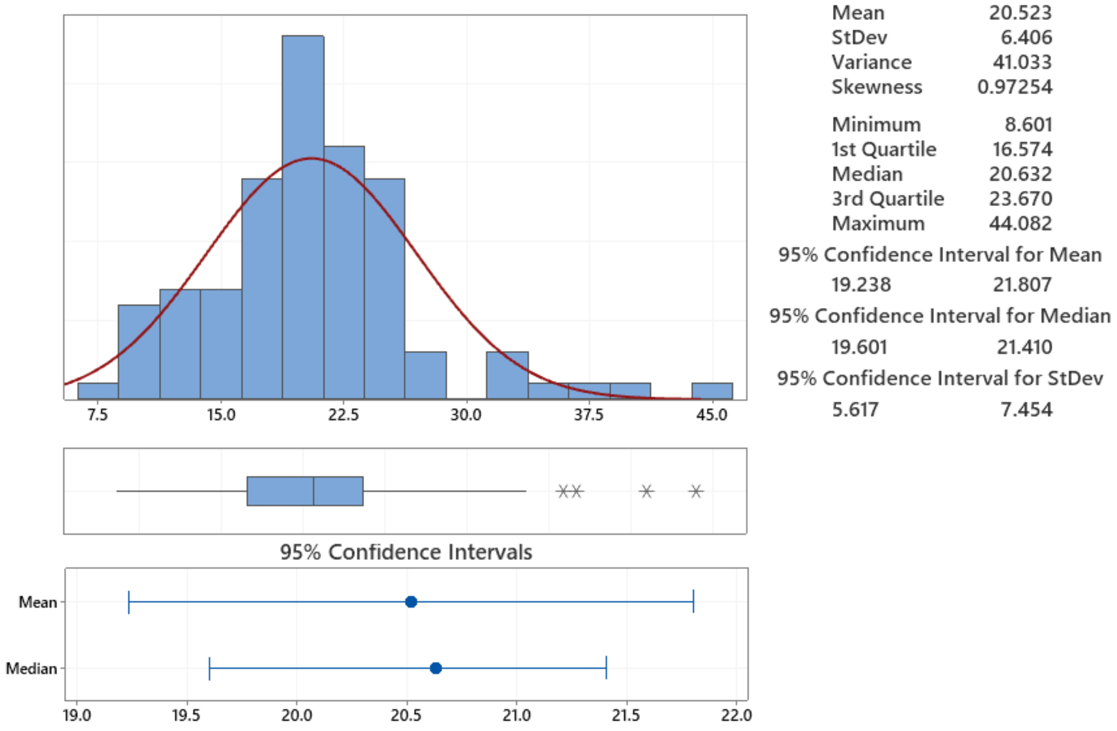
95% Confidence Interval for Median

19.676	21.531

95% Confidence Interval for StDev

5.472	7.261

Figure 4.9: Data Summary of predicted output by NGBoost on Boston Housing data



Mean	20.523
StDev	6.406
Variance	41.033
Skewness	0.97254
Minimum	8.601
1st Quartile	16.574
Median	20.632
3rd Quartile	23.670
Maximum	44.082

95% Confidence Interval for Mean

19.238	21.807

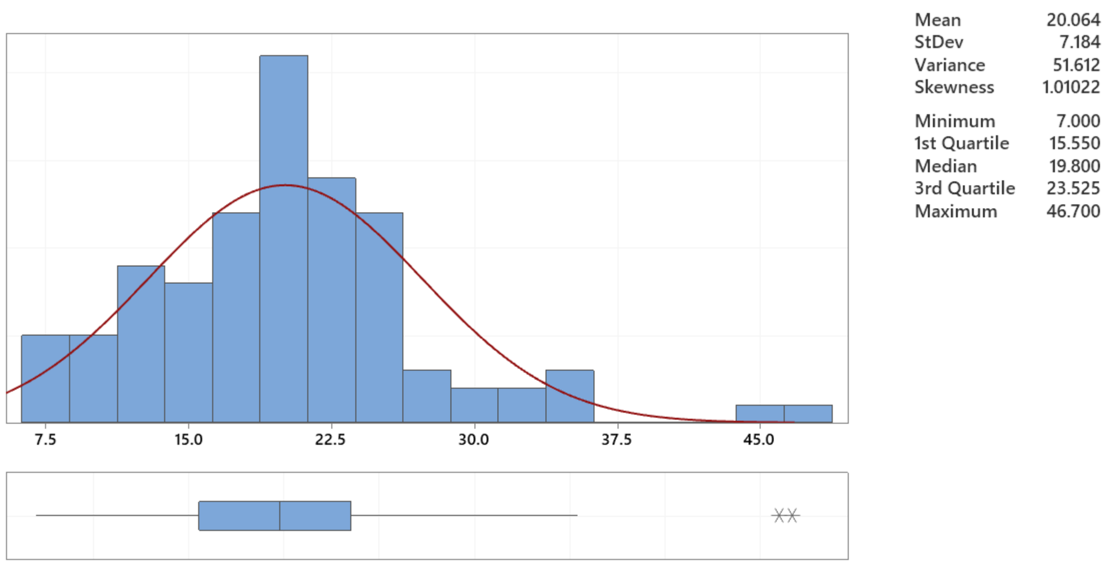
95% Confidence Interval for Median

19.601	21.410

95% Confidence Interval for StDev

5.617	7.454

Figure 4.10: Data Summary of predicted output by XGBoost on Boston Housing data



Mean	20.064
StDev	7.184
Variance	51.612
Skewness	1.01022
Minimum	7.000
1st Quartile	15.550
Median	19.800
3rd Quartile	23.525
Maximum	46.700

Figure 4.11: Data summary of actual output for Boston Housing data

Prediction Intervals of Boston Housing Output Variable Using NGBoost

As discussed in Section 3.1.2.4, NGBoost has a capability to produce prediction intervals for each output. There are two prediction methods for NGBRegressor objects: predict(), which returns point predictions as normally done by any regressor, and pred_dist(), which returns a distribution object representing the conditional distribution of $Y|X = x_i$ at the points x_i in the test set. The format of the distributional output of the point predictions is as follows-

'loc': array [24.5, 33.6]

'scale': array [0.03, 0.04]

where 'loc' signifies 'mean' and 'scale' as standard deviation which are the parameters of normal distribution. For exact distribution output produced by NGBoost on Boston Housing data refer appendix Figure A.1.

Since NGBoost provides the standard deviation for each point. The lower and upper bounds is calculated for each point. Standard deviation 'Scale' of each point is multiplied by z-score of '1.96' (95% confidence) to get the Interval/margin of error for each point. The lower and upper bounds for each point are calculated by:

Lower bound = Predicted value − Margin of error;

Upper bound = Predicted value + Margin of error

Note that the prediction intervals are calculated using the same method used to calculate confidence intervals, the difference is, confidence interval is calculated for the whole predicted output distribution whereas, the prediction interval is calculated for each point prediction. This is possible only in case of NGBoost, as it gives the standard deviations as output for each point prediction. The output summary of prediction interval is available in appendix Table A.3. The table shows the lower and upper bounds calculated for each point prediction through standard deviation that are predicted using NGBoost model. This may increase the surety of answering few questions. For example in row 173 of the Table A.3, it can be said that for the corresponding features – there's 95% chance that the rent of the house may lie between the range of 23K to 25K in the future. Note that we are not predicting the population mean here rather an individual value, so there's greater uncertainty involved and thus a prediction interval is always wider than the confidence interval. Figure 4.12 plots the graph for prediction interval output for target variable 'MEDV' on y-axis. The green line shows the predicted outputs and the red line shows the prediction interval points plotted for each output.

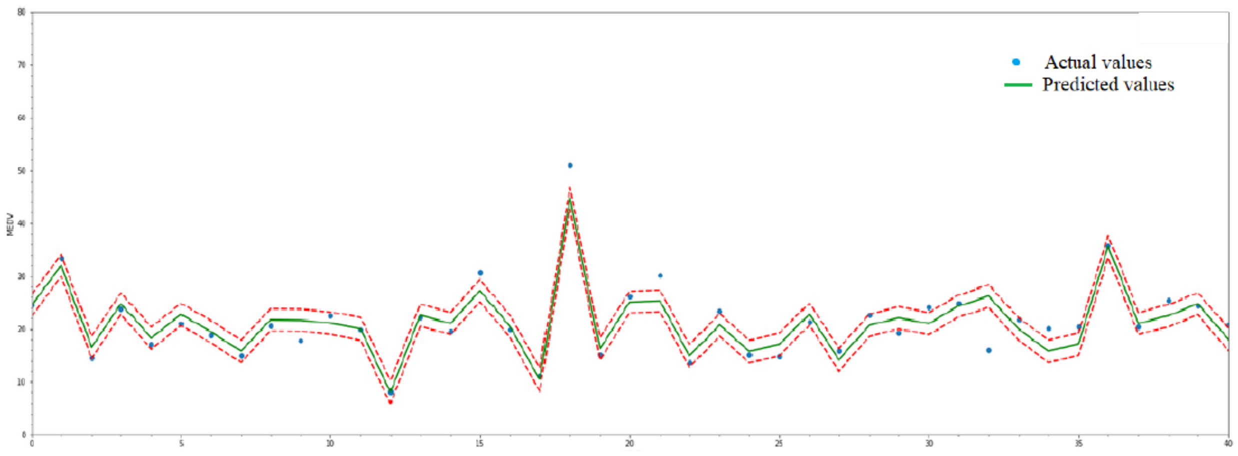


Figure 4.12: Prediction Interval plot for Boston Housing Data

4.2.2 Case Study 2: Regression Analysis on Protein Structure Data Set

Data is split into train and test data using train-test ratio of 80:20 such that train data has 36584 rows and test data has 9146 rows. Models are fit with the default hyperparameters to check the performance and to track the improvement once the models are tuned. On calculating the RMSE for test set, for XGBoost it is coming as RMSE: 3.95. NGBoost gives slightly higher RMSE: 4.81. XGBoost has extra parameters for regularisation, cross validation and a lot more parameters for model optimisation that NGBoost lacks to have. The parameters are tuned using Randomsearchcv with 8-fold cross-validation and 50 iterations and the best tuned parameters are obtained. 8-fold cross validation is used divide the training set of 36584 rows equally in each fold. This is to ensure more stable results avoiding fluctuations in each fold.

Table 4.6: Randomsearchcv best parameters for both the algorithms on protein data

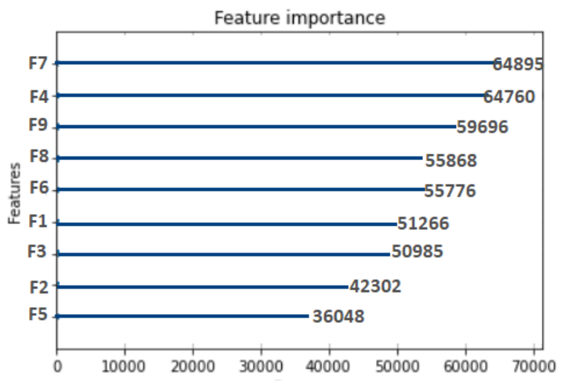
XGBoost	NGBoost

base_score=0.5,
colsample_bylevel=1,
colsample_bynode=1,
colsample_bytree=1,
learning_rate=0.05,
max_depth=15,
min_child_weight=3,
n_estimators=900,
num_parallel_tree=1,
reg_lambda=1,
subsample=1,
nthread=-1
	max_depth=15,
	max_features=None,
	max_leaf_nodes=None,
	min_samples_leaf=1,
	min_samples_split=2,
	min_weight_fraction_leaf=0.0,
	col_sample=0.7,
	learning_rate=0.01,
	minibatch_frac=0.05,
	n_estimators=1100,
	natural_gradient=True,
	tol=0.0001

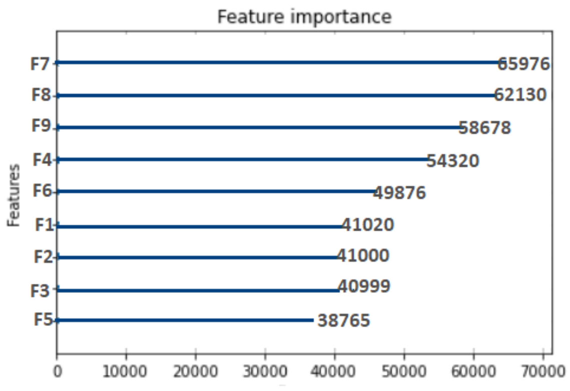
Best score: 3.54	Best score: 3.36

Table 4.6 shows the output of randomsearchcv showing best tuned hyperparameters and the best score (root mean square error – as default score of randomsearchcv) correspond-

ing to each model. While tuning of the parameters, XGBoost takes less time 48.7 mins as compared to NGBoost 97 mins. Although XGBoost has more hyperparameters to be tuned but due to parallelisation feature, it tunes the model faster. The best parameters given through randomsearchcv() shows that the learning rate for XGBoost is 0.05 with 900 n_estimators, whereas for NGBoost learning_rate is 0.01 and n_estimators=1100. This shows that XGBoost converges faster giving better results and avoids overfitting. Also, since 'nthread'=-1 which shows that XGBoost use all the GPU cores available for parallel processing. Model is fit/trained using tuned hyperparameters. While in the training of the model, K-fold cross validation is used due to the large size of training data.



(a) XGBoost

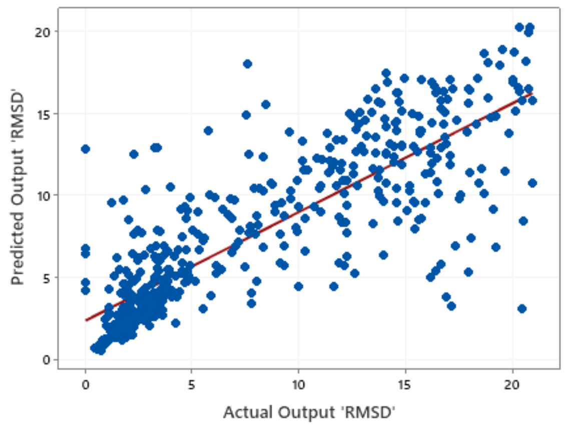


(b) NGBoost

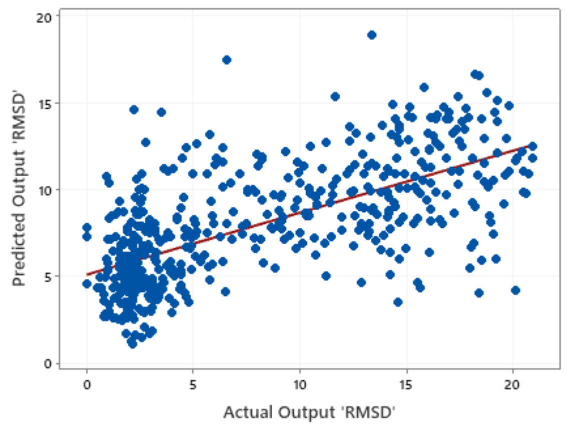
Figure 4.13: Feature Importance graphs for protein data

Figure 4.13 shows the important variables marked by both the models for the protein structure data. The important features marked are almost same except for 'F4' feature being at the second rank in the XGBoost model, while in NGBoost 'F8' holds the second place in importance. The feature ranking provided by both the models increases the credibility of these features since both the models show almost similar results.The features are plotted on y-axis, having their feature scores on the x-axis. Feature scores is a measure that adds up the number of times each feature is split.

The final predictions made on the test set gives the RMSE of 3.58 for NGBoost and 3.44 for XGBoost and the R-squared values as 0.68 and 0.66 for XGBoost and NGBoost respectively. XGBoost outperforms in case of point predictions where it better fits the model with the lower RMSE and comparatively higher R-squared value.



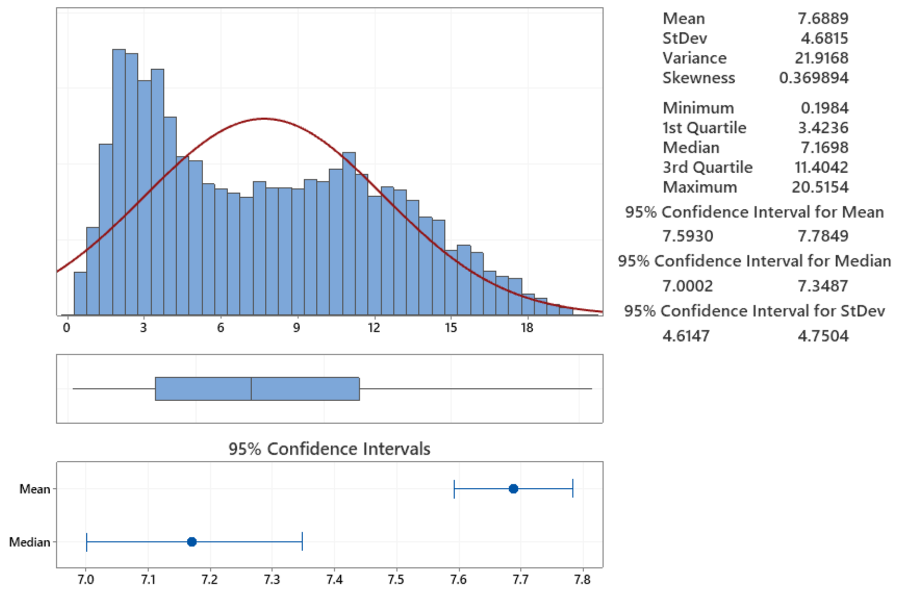
(a) Actual Vs Fitted plot for protein data XG-Boost



(b) Actual Vs Fitted plot for protein data NGBoost

Figure 4.14: Actual vs Fitted plots of predictions made on protein data

The confidence intervals for the output variable ’RMSD’ of protein structure dataset, calculated with 95% confidence for NGBoost is (7.59, 7.78) and for XGBoost is (7.56,7.76). The mean of the actual output is (7.737), mean of predicted output for NGBoost and XGBoost is (7.69) and (7.67) respectively. Both the algorithms show similar interval range. Figures 4.15 and 4.16 shows the distribution plots for the predicted variable ’RMSD’ of protein structure dataset. Figure 4.17 shows the distribution plot of the actual output variable ’RMSD’.



Mean	7.6889
StDev	4.6815
Variance	21.9168
Skewness	0.369894
Minimum	0.1984
1st Quartile	3.4236
Median	7.1698
3rd Quartile	11.4042
Maximum	20.5154
95% Confidence Interval for Mean	
7.5930	7.7849
95% Confidence Interval for Median	
7.0002	7.3487
95% Confidence Interval for StDev	
4.6147	4.7504

Figure 4.15: Data Summary of predicted output by NGBoost on protein data

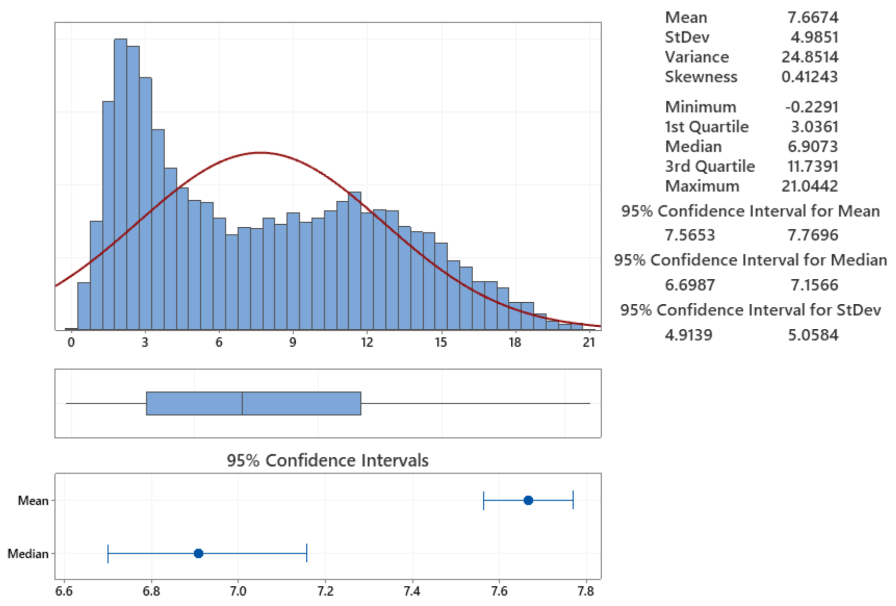


Figure 4.16: Data Summary of predicted output by XGBoost on protein data

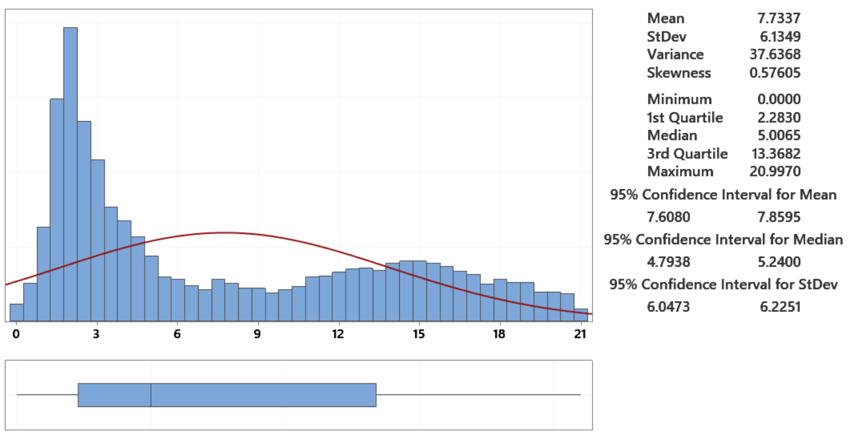


Figure 4.17: Data summary of actual output of protein data

Prediction Intervals of Protein Structure Data Output Variable using NGBoost

Calculating prediction intervals through NGBoost using the function pred_dist() function. The output distribution for each point prediction gives the output in the form of ‘loc’ and ‘scale’, where ‘loc’ stands for mean and ‘scale’ stands for standard deviation. Through this standard deviation an interval can be calculated for each point known as prediction interval. The format of the distributional output of the point predictions is as follows-

‘loc’: array [11.17, 3.78]

‘scale’: array [0.34, 0.19]

where ‘loc’ signifies ‘mean’ and ‘scale’ as standard deviation which are the parameters of normal distribution. For exact distribution output produced by NGBoost on Protein struct data refer appendix Figure A.2.

Since Ngboost provides the standard deviation for each point. The lower and upper bounds is calculated for each point. Standard deviation ’Scale’ of each point is multiplied by z-score of ’1.96’(95% confidence) to get the Interval/margin of error for each point. The lower and upper bounds for each point is calculated by-

Lower bound= Predicted value – Margin of error;

Upper bound= Predicted value + Margin of error

Note that we are not predicting the population mean here rather an individual value, so there’s greater uncertainty involved and thus a prediction interval is always wider than the confidence interval. Table A.4 shows the range of outputs produced by applying the NGBoost model on the protein structure data.

4.2.3 Result Summary

Table 4.7: Comparative Result Summary

		XGBoost	NGBoost
Boston Housing Data	Base model RMSE: (with default parameters)	2.40	2.55
	RMSE: (with tune hyper parameters)	2.37	2.40
	R2	0.89	0.88
	Adjusted R2	0.90	0.88
	Hyperparameters tuning time	12.8 mins	21.4 mins
	Model fitting time	9.2 mins	12.01 mins
Protein Structure Data	Base model RMSE: (with default parameters)	3.95	4.81
	RMSE: (with tune hyper parameters)	3.44	3.58
	R2	0.68	0.66
	Adjusted R2	0.68	0.66
	Hyperparameters tuning time	48.7 mins	97 mins
	Average model training time	39.3 mins	56 mins

Few key points while performing hyperparameter tuning on both the models:

- NGBoost shows similar Root-Mean-Squared-Error as that of XGBoost for the base models. After hyperparameter tuning of both the models, the RMSE is slightly better in case of XGBoost, whereas in case of NGBoost there is negligible improvement in the accuracy. Since XGBoost has a lot of parameters to be tuned, it works best after tuning the parameters. Default XGBoost model performs generally worse as compared to its tuned version. NGBoost gives slightly less accuracy but it does not show much fluctuations between the default and the tuned results. Also in case of XGBoost there is always a scope of improvement by adjusting its parameters and using them in different combinations, but that is not the case with limited hyperparameters of NGBoost. The R-squared values does not show much difference between XGBoost and NGBoost.
- Since XGBoost has a feature of multicore parallel processing, this allows the model

to use all available CPU cores in the system. The GPU time for hyperparameter tuning and model fitting shows faster results for XGBoost, even though the size of the parameter grid is bigger in case of XGBoost than in NGBoost. XGBoost takes almost half of a time in tuning process as compared to NGBoost. XGBoost converges faster, finding the better hyperparameters such as best suitable n_estimators, max_depth and learning rate makes balance between convergence rate and overfitting.

- NGBoost has a power of calculating prediction intervals of the regression problems that XGBoost doesn't have. It predicts the whole probabilistic distribution at each point. The confidence intervals for both the algorithms was comparable but prediction intervals were only produced by NGBoost. For XGBoost only confidence interval is calculated, but not prediction interval.

Chapter 5

Conclusion and Future work

Finally, the last chapter concludes about the studies done on NGBoost and XGBoost, states whether both the algorithms resulted to be at the same level or if any algorithm is better than the other algorithm. It also includes the description about the challenges faced throughout the project and what methods were used to overcome those challenges. There are several open works that are not addressed due to time constraints or are opened through this work that can be taken forward to explore more about NGBoost and its relation with other existing algorithms.

5.1 Conclusion

This study exhibits conceptual, empirical and performance analysis of the newly developed algorithm NGBoost (Natural Gradient Boosting) with XGBoost (Extreme Gradient Boosting) on two regression datasets. The analysis of the algorithms is based on the performance of base models, hyperparameter tuning efficiency in finding the best parameters, best cross validation scores, model training time, RMSEs, R-squared values and confidence intervals. With the results shown in the result summary it is clear that NGBoost almost matches the benchmark set by XGBoost in terms of point prediction accuracies, efficiency and tuning; but still XGBoost is better than NGBoost in terms of computation performance and accuracy. In terms of handling large datasets XGBoost proved to be more efficient than NGBoost. Ngboost took more time to perform the tuning and training on large datasets, as XGBoost worked parallelly on 8 concurrent processors of the machine (set through parameter 'nthread' that makes XGBoost use maximum threads available while training), NGBoost worked only on 2 concurrent machines for the large dataset. It is noted that due to exhaustive set of hyperparameters of XGBoost, it was able to fit the model more accurately on tuned parameters. While analyzing cross validation results over each iteration it was observed that NGBoost was not able to improve the accuracy beyond a certain point, increasing the computation time in both the training and the predicting phase. It can be stated that XGBoost is more sensitive to parameter tuning than NGBoost. Thus, it is not guaranteed that it ran with the best configuration and there can still be a chance of improvement of scores obtained.

Best point predictions are coming through XGBoost as it is a dedicated algorithm for that. But to add the power of uncertainty to any point prediction, is done through NGBoost. Probabilistic predictions for each point increases the reliability of the results. For example, in order to be sure about a particular point prediction, prediction interval range can be calculated. With less aggressive tuning in NGBoost, it still gives similar results for point predictions. It is a less complex algorithm that could be used on small to medium datasets, but it is expensive in case of computation time.

5.2 Limitations and Challenges

Computational challenges – It was difficult to store the big size data, tune the hyperparameters and train the model with multiple cross validated iterations on the local system. The resources got exhausted due to less space and lack of GPU on the system. Since NGBoost takes lot of time during the training process and XGBoost parameter tuning is also very exhaustive it became necessary to take the project on the cloud. Cloud service of AWS (AWS Sage maker) was used to create multiple GPU notebooks on the cloud machines that also had readymade libraries for XGBoost and Scikit learn used in python. Also, AWS provides data storage tool (S3 Buckets) on which data was stored in the distributed manner. It was easy to fetch the data and do parallel processing on the GPU machine instances avoiding reinstalling of libraries repetitively.

Limitations in literature review – Since NGBoost is a newly developed algorithm, not much related literature was available to refer, understand and explore the algorithm. There are many studies that showcase different features of XGBoost that could be implemented but for NGBoost there is still more details and features that needs to be documented.

Limitations in libraries - NGBoost has limited libraries and the main model is only available on python as of now under Scikit learn library. So the implementation was limited to python and was not available in R.

5.3 Future Work

This work contributes to give detailed statistical analysis of the novel algorithm NGBoost and XGBoost, providing the concepts on how both the algorithms work. The practical analysis does not show much difference on the point predictions. Further hyperparameters are described for both the algorithms and finally the machine learning models are fit. This paper is focused on applying NGBoost to the regression problems, since NGBoost is significantly made to output probabilistic predictions for regression problems (Duan et al., 2019). In classification models it is already a norm where we can get the prediction intervals. The research work can become a foundation to extend in several directions. This includes analysis of the algorithms on a broader range of data, with small and large sizes and with different distributions such as survival analysis datasets, time series datasets and other classification datasets. Also if it is possible to produce probabilistic distribution for XGBoost, then it can be compared with NGBoost to know if XGBoost performs better in getting prediction intervals. Since NGBoost suffers from slow speed overhead in case of large datasets it becomes important to analyse further, whether it can have the regularization and parallelizing parameters just like XGBoost.

References

Amazon SageMaker (2019). Build, Train, and Deploy Machine Learning Model, Amazon Web Services, Inc. URL: https://aws.amazon.com/sagemaker/ (cit. on p. 34).

Bentéjac, Candice, Anna Csörgő, and Gonzalo Martínez-Muñoz (2019). *A Comparative Analysis of XGBoost* (cit. on pp. 2, 6, 10).

Breiman, Leo (Aug. 1996). “Bagging predictors”. *Machine Learning* 24, pp. 123–140. DOI: 10.1007/bf00058655 (cit. on p. 1).

Bühlmann, Peter and Torsten Hothorn (Nov. 2007). “Rejoinder: Boosting Algorithms: Regularization, Prediction and Model Fitting”. *Statistical Science* 22, pp. 516–522. DOI: 10.1214/07-sts242rej (cit. on p. 6).

Chen, Tianqi (2014). *Introduction to Boosted Trees*. URL: https://web.njit.edu/~usman/courses/cs675_fall16/BoostedTree.pdf (cit. on p. 14).

Chen, Tianqi and Carlos Guestrin (2016). “XGBoost”. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD ’16*. DOI: 10.1145/2939672.2939785. URL: https://arxiv.org/abs/1603.02754 (cit. on pp. 2, 6, 9, 12, 13, 14, 16, 17, 18).

Chipman, Hugh A., Edward I. George, and Robert E. McCulloch (Mar. 2010). “BART: Bayesian additive regression trees”. *The Annals of Applied Statistics* 4, pp. 266–298. DOI: 10.1214/09-aoas285 (cit. on p. 8).

Daoud, Essam Al (Jan. 2019). “Comparison between XGBoost, LightGBM and CatBoost Using a Home Credit Dataset”. *International Journal of Computer and Information Engineering* 13, pp. 6–10 (cit. on pp. 2, 7, 10).

Drucker, Harris, Robert Schapire, and Patrice Simard (Aug. 1993). “BOOSTING PERFORMANCE IN NEURAL NETWORKS”. *International Journal of Pattern Recognition and Artificial Intelligence* 07, pp. 705–719. DOI: 10.1142/s0218001493000352 (cit. on p. 5).

Duan, Tony, Anand Avati, Daisy Ding, Sanjay Basu, Andrew Ng, and Alejandro Schuler (Oct. 2019). *NGBoost: Natural Gradient Boosting for Probabilistic Prediction* (cit. on pp. III, 2, 9, 10, 41, 58).

Freund, Y. (Sept. 1995). “Boosting a Weak Learning Algorithm by Majority”. *Information and Computation* 121, pp. 256–285. DOI: 10.1006/inco.1995.1136 (cit. on p. 5).

Freund, Yoav and Robert Schapire (1996). *Experiments with a New Boosting Algorithm* (cit. on pp. 1, 10).

Freund, Yoav and Robert E Schapire (1997). “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”. *Journal of Computer and System Sciences* 55, pp. 119–139. DOI: 10.1006/jcss.1997.1504. URL: http://www.face-rec.org/algorithms/Boosting-Ensemble/decision-theoretic_generalization.pdf (cit. on pp. 1, 5).

– (1999). “A Short Introduction to Boosting”. *Journal of Japanese Society for Artificial Intelligence* 14, pp. 771–780. URL: https://cseweb.ucsd.edu/~yfreund/papers/IntroToBoosting.pdf (cit. on p. 6).

Friedman, Jerome H. (2001). “Greedy Function Approximation: A Gradient Boosting Machine”. *The Annals of Statistics* 29, pp. 1189–1232. URL: https://www.jstor.org/stable/2699986 (cit. on pp. III, 1, 5, 12).

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani (Apr. 2000). “Additive logistic regression: a statistical view of boosting”. *The Annals of Statistics* 28, pp. 337–407. DOI: 10.1214/aos/1016218223 (cit. on pp. 1, 15).

Gebetsberger, Manuel, Jakob W. Messner, Georg J. Mayr, and Achim Zeileis (Dec. 2018). “Estimation Methods for Nonhomogeneous Regression Models: Minimum Continuous Ranked Probability Score versus Maximum Likelihood”. *Monthly Weather Review* 146, pp. 4323–4338. DOI: 10.1175/mwr-d-17-0364.1 (cit. on p. 29).

Grahn, Hakan, Niklas Lavesson, Mikael Hellborg Lapajne, and Daniel Slat (Dec. 2011). “CudaRF: A CUDA-based implementation of Random Forests”. *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA).* DOI: 10.1109/aiccsa.2011.6126612 (cit. on p. 8).

Ke, Guolin, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu (2017). *LightGBM: A Highly Efficient Gradient Boosting Decision Tree.* URL: https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf (cit. on pp. 2, 7, 8, 10).

Kearns, Michael J and Leslie G Valiant (1988). *Learning boolean formulae or finite automata is as hard as factoring.* Harvard University, Center For Research In Computing Technology, Aiken Computation Laboratory (cit. on p. 5).

Kearns, Michael and Leslie Valiant (Jan. 1994). “Cryptographic limitations on learning Boolean formulae and finite automata”. *Journal of the ACM (JACM)* 41, pp. 67–95. DOI: 10.1145/174644.174647 (cit. on p. 5).

Kendall, Alex and Yarin Gal (Oct. 2017). “What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?” *arXiv:1703.04977 [cs].* URL: https://arxiv.org/abs/1703.04977 (cit. on p. 9).

Klein, Nadja, Thomas Kneib, Stephan Klasen, and Stefan Lang (Dec. 2014). “Bayesian structured additive distributional regression for multivariate responses”. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 64, pp. 569–591. DOI: 10.1111/rssc.12090. URL: https://www2.uibk.ac.at/downloads/c4041030/wpaper/2013-35.pdf (cit. on p. 9).

Kullback, S. and R. A. Leibler (Mar. 1951). “On Information and Sufficiency”. *The Annals of Mathematical Statistics* 22, pp. 79–86. DOI: 10.1214/aoms/1177729694. URL: https://www.projecteuclid.org/euclid.aoms/1177729694 (cit. on p. 25).

Lakshminarayanan, Balaji, Alexander Pritzel, and Charles Blundell (Nov. 2017). “Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles”. *arXiv:1612.01474 [cs, stat]* V3. URL: https://arxiv.org/abs/1612.01474 (cit. on pp. 2, 8).

Lakshminarayanan, Balaji, Daniel M. Roy, and Yee Whye Teh (May 2016). “Mondrian Forests for Large-Scale Regression when Uncertainty Matters”. *arXiv:1506.03805 [cs, stat]*. URL: https://arxiv.org/abs/1506.03805 (cit. on p. 8).

Lemarechal, Claude (2012). *Cauchy and the Gradient Method*. URL: https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf (cit. on p. 21).

Ly, Alexander, Maarten Marsman, Josine Verhagen, Raoul P.P.P. Grasman, and Eric-Jan Wagenmakers (Oct. 2017). “A Tutorial on Fisher information”. *Journal of Mathematical Psychology* 80, pp. 40–55. DOI: 10.1016/j.jmp.2017.05.006. URL: https://arxiv.org/pdf/1705.01064.pdf (cit. on p. 26).

März, Alexander (Aug. 2019). “XGBoostLSS – An extension of XGBoost to probabilistic forecasting”. *arXiv:1907.03178 [cs, stat]*. URL: https://arxiv.org/abs/1907.03178 (cit. on p. 9).

Mitchell, Rory and Eibe Frank (July 2017). “Accelerating the XGBoost algorithm using GPU computing”. *PeerJ Computer Science* 3, e127. DOI: 10.7717/peerj-cs.127 (cit. on p. 8).

Nemeth, Martin, Dmitrii Borkin, and German Michalconok (2019). “The Comparison of Machine-Learning Methods XGBoost and LightGBM to Predict Energy Development”. *Computational Statistics and Mathematical Modeling Methods in Intelligent Systems*, pp. 208–215. DOI: 10.1007/978-3-030-31362-3_21 (cit. on p. 7).

Pedregosa, Fabian et al. (2011). “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* 12, pp. 2825–2830. URL: https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf (cit. on p. 34).

Prokhorenkova, Liudmila, Gleb Gusev, Aleksandr Vorobev, Anna Dorogush, and Andrey Gulin (2018). *CatBoost: unbiased boosting with categorical features* (cit. on p. 8).

Rahman, Saifur, Muhammad Irfan, Mohsin Raza, Khawaja Moyeezullah Ghori, Shumayla Yaqoob, and Muhammad Awais (Feb. 2020). “Performance Analysis of Boosting Classifiers in Recognizing Activities of Daily Living”. *International Journal of Environmental Research and Public Health* 17, p. 1082. DOI: 10.3390/ijerph17031082 (cit. on p. 7).

Rigby, R. A. and D. M. Stasinopoulos (June 2005). “Generalized additive models for location, scale and shape (with discussion)”. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 54, pp. 507–554. DOI: 10.1111/j.1467-9876.2005.00510.x (cit. on p. 9).

Schapire, Robert E. (June 1990). “The strength of weak learnability”. *Machine Learning* 5, pp. 197–227. DOI: 10.1007/bf00116037 (cit. on p. 5).

Valiant, L. G. (Nov. 1984). “A theory of the learnable”. *Communications of the ACM* 27, pp. 1134–1142. DOI: 10.1145/1968.1972 (cit. on pp. 4, 5).

Appendix A

Table A.1: Descriptive Statistics of Boston Housing Data

Variable	N	N*	Mean	SE Mean	StDev	Minimum	Q1	Median	Q3	Maximum
CRIM	506	0	3.614	0.382	8.602	0.006	0.082	0.257	3.682	88.976
ZN	506	0	11.36	1.04	23.32	0.00	0.00	0.00	12.50	100.00
INDUS	506	0	11.137	0.305	6.860	0.460	5.175	9.690	18.100	27.740
CHAS	506	0	0.0692	0.0113	0.2540	0.0000	0.0000	0.0000	0.0000	1.0000
NOX	506	0	0.55470	0.00515	0.11588	0.38500	0.44900	0.53800	0.62400	0.87100
RM	506	0	6.2846	0.0312	0.7026	3.5610	5.8848	6.2085	6.6260	8.7800
AGE	506	0	68.57	1.25	28.15	2.90	44.85	77.50	94.10	100.00
DIS	506	0	3.7950	0.0936	2.1057	1.1296	2.0971	3.2074	5.2126	12.1265
RAD	506	0	9.549	0.387	8.707	1.000	4.000	5.000	24.000	24.000
TAX	506	0	408.24	7.49	168.54	187.00	279.00	330.00	666.00	711.00
PTRATIO	506	0	18.456	0.0962	2.165	12.600	17.375	19.050	20.200	22.000
B	506	0	356.67	4.06	91.29	0.32	375.30	391.44	396.23	396.90
LSTAT	506	0	12.653	0.317	7.141	1.730	6.928	11.360	16.992	37.970
MEDV	506	0	22.533	0.409	9.197	5.000	16.950	21.200	25.000	50.000

Table A.2: Descriptive Statistics of Protein Tertiary Structure Data

Variable	N	N*	Mean	SE Mean	StDev	Minimum	Q1	Median	Q3	Maximum
F1	45730	0	9871.6	19.0	4058.1	2392.1	6936.6	8898.8	12126.3	40034.9
F2	45730	0	3017.4	6.85	1464.3	403.5	1979.0	2668.2	3786.4	15312.0
F3	45730	0	0.30239	0.000294	0.06289	0.09250	0.25874	0.30015	0.34289	0.57769
F4	45730	0	103.49	0.259	55.42	10.31	63.56	87.74	133.65	369.32
F5	45730	0	1368299	2638	564037	319490	953578	1237219	1690920	5472011
F6	45730	0	145.64	0.327	70.00	31.97	94.76	126.18	181.47	598.41
F7	45730	0	3990	9.32	1994	0.00	3165	3840	4644	105948
F8	45730	0	69.975	0.264	56.493	0.000	31.000	54.000	91.000	350.000
F9	45730	0	34.524	0.0280	5.980	15.228	30.424	35.299	38.871	55.301
RMSD	45730	0	7.7485	0.0286	6.1183	0.0000	2.3048	5.0300	13.3790	20.9990

# XGBoost best estimators for Boston Housing data:

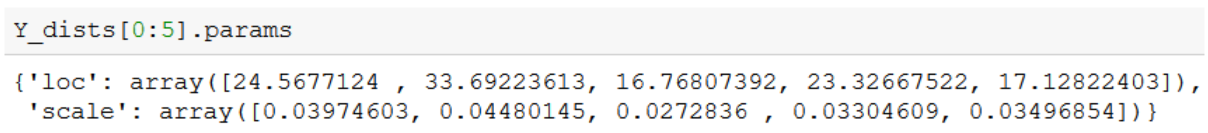
```
XGBRegressor(base_score=0.25, booster='gbtree', colsample_bylevel=1,
            colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
            importance_type='gain', interaction_constraints='',
            learning_rate=0.1, max_delta_step=0, max_depth=10,
            min_child_weight=3, missing=nan, monotone_constraints='()',
            n_estimators=900, n_jobs=0, num_parallel_tree=1, random_state=0,
            reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
            tree_method='exact', validate_parameters=1, verbosity=None)
```

Best Random CV Score: 3.22(root_mean_squared_error)

# NGBoost best estimators for Boston Housing data:

```
NGBRegressor(Base=DecisionTreeRegressor(ccp_alpha=0.0,criterion='friedman_mse',
           max_depth=15, max_features=None,max_leaf_nodes=None,
           min_impurity_decrease=0.0,min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0,presort='deprecated',
           random_state=None, splitter='best') col_sample=0.7,
           learning_rate=0.05, minibatch_frac=0.5, n_estimators=1100,
           natural_gradient=True, tol=0.0001,verbose=True, verbose_eval=100)
```

Best Random CV Score: 3.26(root_mean_squared_error)

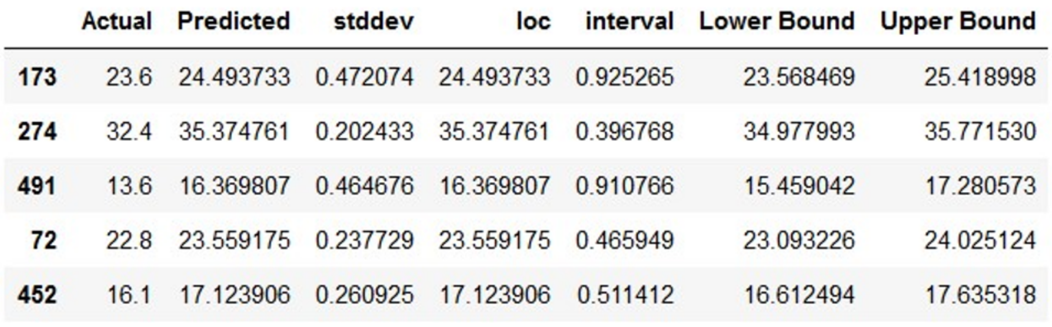


```
Y_dists[0:5].params
```

```
{'loc': array([24.5677124 , 33.69223613, 16.76807392, 23.32667522, 17.12822403]),
 'scale': array([0.03974603, 0.04480145, 0.0272836 , 0.03304609, 0.03496854])}
```

Figure A.1: Distribution of the output variable of Boston data predicted using NGBoost

Table A.3: Output summary of Boston data produced by NGBoost including actual output, predicted output, standard deviation, mean, prediction interval, lower and upper range of the interval



	Actual	Predicted	stddev	loc	interval	Lower Bound	Upper Bound
173	23.6	24.493733	0.472074	24.493733	0.925265	23.568469	25.418998
274	32.4	35.374761	0.202433	35.374761	0.396768	34.977993	35.771530
491	13.6	16.369807	0.464676	16.369807	0.910766	15.459042	17.280573
72	22.8	23.559175	0.237729	23.559175	0.465949	23.093226	24.025124
452	16.1	17.123906	0.260925	17.123906	0.511412	16.612494	17.635318

# XGBoost best estimators for Protein Structure data:

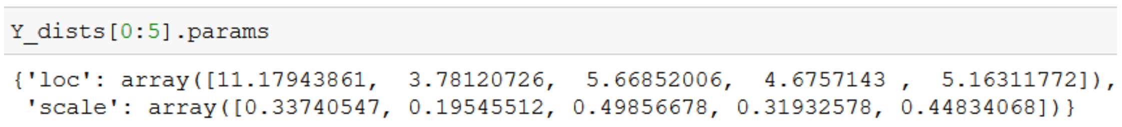
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
           colsample_bynode=1, colsample_bytree=1, cv=10, gamma=0,
           gpu_id=-1, importance_type='gain', interaction_constraints='',
           learning_rate=0.05, max_delta_step=0, max_depth=15,
           min_child_weight=4, missing=nan, monotone_constraints='()',
           n_estimators=900, n_jobs=-1, num_parallel_tree=1,reg_alpha=0,
           reg_lambda=1, scale_pos_weight=1,nthread =-1,subsample=1,
           tree_method='exact', validate_parameters=1, verbosity=None)
```

Best Random CV Score: 3.54 (root_mean_squared_error)

# NGBoost best estimators for Protein Structure data:

```
NGBRegressor(Base=DecisionTreeRegressor(ccp_alpha=0.0,criterion='friedman_mse',
           max_depth=15, max_features=None,max_leaf_nodes=None,
           min_impurity_decrease=0.0,min_impurity_split=None,
           min_samples_leaf=1,min_samples_split=2,
           min_weight_fraction_leaf=0.0,presort='deprecated',
           random_state=None,splitter='best'), col_sample=0.7,
           learning_rate=0.01, minibatch_frac=0.5,n_estimators=1100,
           natural_gradient=True,tol=0.0001,verbose=True, verbose_eval=100)
```

Best Random CV Score: 3.36 (root_mean_squared_error)

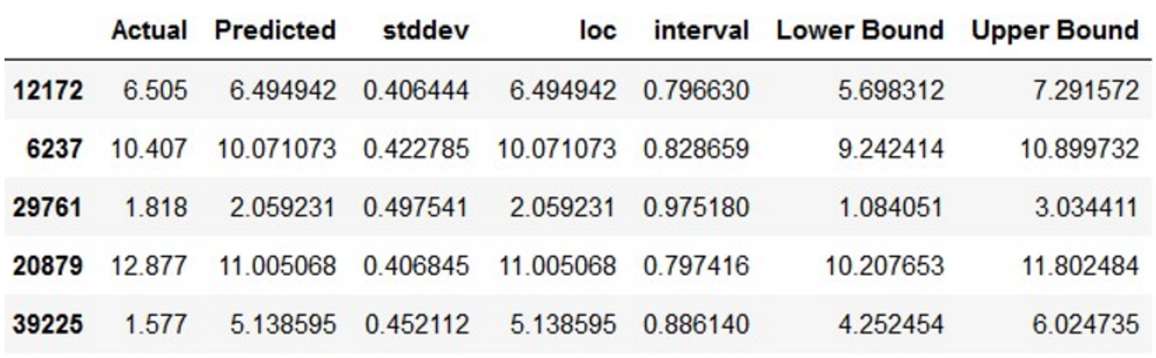


```
Y_dists[0:5].params
```

```
{'loc': array([11.17943861,  3.78120726,  5.66852006,  4.6757143 ,  5.16311772]),
 'scale': array([0.33740547, 0.19545512, 0.49856678, 0.31932578, 0.44834068])}
```

Figure A.2: Distribution of the output variable of protein data predicted using NGBoost

Table A.4: Output summary of Protein data produced by NGBoost including actual output, predicted output, standard deviation, mean, prediction interval, lower and upper range of the interval



	Actual	Predicted	stddev	loc	interval	Lower Bound	Upper Bound
12172	6.505	6.494942	0.406444	6.494942	0.796630	5.698312	7.291572
6237	10.407	10.071073	0.422785	10.071073	0.828659	9.242414	10.899732
29761	1.818	2.059231	0.497541	2.059231	0.975180	1.084051	3.034411
20879	12.877	11.005068	0.406845	11.005068	0.797416	10.207653	11.802484
39225	1.577	5.138595	0.452112	5.138595	0.886140	4.252454	6.024735