Boosting with XGBoost

Nadav Gerner

October 2023

Abstract

Rapid advancement in machine learning and data scalability necessitates the need for algorithms that can adapt. The theoretical evolution from Ensemble Learning, to Gradient Boosting Machines (GBM) to eXtreme Gradient Boosting (XGBoost) for predictive capabilities is investigated in this study mathematically. When then applied in a real-world scenario, the Ames, Iowa housing dataset, XGBoost exhibits an exceptional 62.9% reduction in execution times and delivers up to an 85.4% enhancement in accuracy over the GBM model. These improvements position XGBoost as an "eXtremely" powerful tool for efficient and accurate predictive modeling.

1 Introduction

1.1 Introduction to Ensemble Learning and Gradient Boosting

> "Machine learning is a branch of artificial intelligence (AI) and computer science that focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy."[1]

Machine learning revolutionized the way we approach data analysis for decision-making. Compared to humans, computers have much greater capabilities to handle and process vast amounts of data. This ability is fundamental to the success of machine learning algorithms. Being capable of viewing millions of data points simultaneously enables them to detect patterns, and later on make accurate predictions that would be humanly impossible due to the sheer volume and complexity of the data involved. From fields such as healthcare (predicting the onset of dangerous diseases), to finance (fraud detection), to astrophysics (creating the first black hole picture) and beyond, machine learning is incorporated in all, having a remarkable impact and allowing for some unparalleled insights and discoveries. While the foundation for machine learning is very much built on advanced statistics, which we already understand extensively, the continuous evolution of technology and, specifically, hardware has ignited the development of newer, and increasingly sophisticated, machine learning algorithms.

These advancements in machine learning algorithms are designed to deal with new challenges that our world did not face before, or solve old ones that we never had the tools to solve. They have played a pivotal role in improving predictive modeling and data analysis while unlocking new possibilities in understanding complex patterns within data.

The development of algorithms and models such as ensemble learning, that is, a technique that creates multiple similar models, and then uses the knowledge of those to combine them into a better model, have increased in popularity due to their ability to enhance prediction, accuracy, and model generalization. Part of this is the fact that the result, the ensemble model, combines the inputs from any other machine learning algorithm such as logistic regression, decision tree, etc., which means we can adjust it to any use case. This paper delves into ensemble learning methods, particularly gradient-boosting algorithms that use decision trees, with a special emphasis on XGBoost - Extreme Gradient Boosting algorithm.

Advancements in the realm of ensemble learning such as the Gradient Boosting Machine (GBM), introduced by Friedman in 1999 [Friedman, 1999], promised even greater potential for solving real-world problems. GBM, specifically when applied to tree models, has proven highly effective and

[1]Source: IBM Official Blog, `https://www.ibm.com/topics/machine-learning`.

successful in creating accurate and versatile predictive models that can be applied to various types of data. In recent years, a new, notable advancement, has started to gain traction in the realm of gradient-boosting tree algorithms: XGBoost, introduced by Chen and Guestrin in 2016 [Chen and Guestrin, 2016a]. XGBoost, while similar to Friedman's gradient boosting algorithms, has provided a set of unique features and advantages, and put it over the top as one of the most efficient and popular algorithms to date.

We will dive deeply and explore the theoretical mathematical differences between Gradient Boosting Machine (GBM) and XGBoost, focusing on differences in the following areas:

1. Building of adaptive trees
2. Using the Second-order Taylor expansion of the loss function
3. Custom objective functions
4. L1 (Lasso) and L2 (Ridge) regularization techniques
5. Column block
6. Parallel and distributed computing

While GBM and XGBoost both share the use of the Taylor series, regularization techniques, and adaptive trees. XGBoost uses those advancements slightly differently, with an emphasis on better efficiency of the use of computational power. For example. while GBM uses a first-order approximation, XGBoost applies the second-order Taylor expansion to the objective function (the function we use to make predictions). GBM uses L1 and L2 regularization techniques, as well as adaptive trees to allow it to iteratively correct errors and improve predictive performance. XGBoost in comparison incorporates these techniques in a more structured and optimized, for better regulation, as well as optimized tree construction, to maximize the algorithm's overall performance.

The rest of the features mentioned above are only incorporated in XGBoost as an improvement, allowing it to build powerful variants of tree-boosting algorithms.

Further on, we will demonstrate the mathematical differences in an application I will develop. For that, I have used the Kaggle housing prices data set for advanced regression techniques [2] as a real-life supervised learning application of GBM and XGBoost algorithms. After going through the typical data science life cycle, I implemented both algorithms (separately) to the same housing data set to train and predict the sale prices of houses in Ames, Iowa. We will be able to compare the speed, accuracy, and overall performance of the algorithms to have a live example of how these advancements or differences impact the results in house prices.

1.2 Why Gradient Boosting? Why XGBoost?

As an aspiring data scientist, staying up to date with the latest developments in the field is essential. The rapid evolution of machine learning necessitates the workers of the field to be active learners, who can thoroughly understand efficient algorithms for data sets and complex problems. In this regard, XGBoost has emerged as a leading algorithm, offering superior speed, accuracy, and efficiency compared to traditional gradient-boosting approaches. Writing this paper allows for the

[2] Source: Kaggle, `https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques`.

learning of a newer algorithm, applying my skills as a mathematician, and expanding my toolkit for the field as a future developer.

Moreover, in 2015, 17 out of the 29 winning solutions of challenges hosted by the well-known machine learning site Kaggle used XGBoost [Chen and Guestrin, 2016a]. This demonstrated that XGBoost is not only popular and efficient but also applicable to a large variety of problems. As we explore the foundations of gradient boosting and conduct a detailed analysis of XGBoost, we will note the innovative advancements that improved it over traditional gradient boosting. As XGBoost has become essential for any data scientist, every professional in the field must equip themselves with the knowledge and leading tools to navigate through the ever-changing field of machine learning.

2 Background

2.1 Motivation for Ensemble Learning

Unlike algorithms like GBM and XGBoost, which were developed by individual creators, ensemble learning is a concept that evolved through many different researchers and practitioners who contributed to its advancements since it was first introduced. Rather than thinking of ensemble learning as a standalone machine learning algorithm, we would benefit by viewing it as adding the results from multiple different simple models [Schapire and Freund, 2012]. We typically refer to those models as base models, and those encompass a variety of algorithms such as decision trees, a support vector machine, and more. By themselves, they are limited in their predictive capabilities. The fundamental motivation behind ensemble learning lies in the desire to enhance predictive accuracy and efficiency. Early researchers recognized that a single learning algorithm faces limitations, influenced by biases and the assumptions each model creates. Ensemble models can avoid the limitations of individual models and capitalize on their complementary strengths when combined.

One of the primary challenges that drove the development of ensemble learning techniques is the trade-off between bias and variance.

Definition 2.1 ***Bias*** *refers to capturing the complexity of the data. We want to be careful when approximating a real-world problem, which may be complex, by an oversimplified model. A model with high bias fails to capture the important patterns and relationships in the data.*

Definition 2.2 ***Variance*** *refers to making sure the model's sensitivity doesn't fixate on small fluctuations or noise in the training data.*

The trade-off is that increasing a model's complexity (e.g. using more features) generally reduces bias but increases variance, while reducing model complexity increases bias but decreases variance.

We want to be careful when approximating a real-world problem, which may be complex, by an oversimplified model. A model with high bias fails to capture the important patterns and relationships in the data. A model with high variance fits the training data very closely, potentially capturing noise in the data and leading to overfitting. Overfit models perform well on training data but poorly on unseen data.

The bias-variance trade-off often leads to challenges such as over-fitting or under-fitting. By using multiple models, ensemble techniques allow the data scientist to reduce both bias and variance. By combining diverse models, different types of data can be modeled, and tasks such as classification, regression, and clustering, can be applied.

Ensemble learning models are divided into three types of models: bagging, boosting, and stacking [Schapire and Freund, 2012]. This paper will delve into the boosting models.

Ensemble boosting models is a category that has gathered a significant increase in popularity and recognition in the machine learning community. Boosting models are characterized by their sequential training process. The sequential training process is a repeating process in which we can, after each iteration of the algorithm, correct the errors made by the previous iterations.

Understanding the historical context and the motivation that drove the development of ensemble boosting models is essential. These models allowed for advancements in all fields of machine learning, both in supervised learning algorithms, as well as unsupervised learning algorithms. Shedding light on how ensemble learning models work will explain why they continue to evolve and play a pivotal role in advancing machine learning methodologies.

Following this discussion, the subsequent sections of this paper will delve into the detailed mathematical principles and implementations for ensemble boosting models, as they are the foundation for a thorough exploration of how this game-changing approach to machine learning helped the development of gradient boosting algorithms, specifically the recent advancement of XGBoost.

2.2 Ensemble Learning

For this section, we will refer to a weak learner as a model that produces a result that is slightly better than a guess. We will refer to a strong learner as a model with very close to perfect performance and accuracy. Finally, we will refer to the individuals training the model as the ensembles.

As previously discussed, the ensemble learning method is a powerful method that combines the advantages and strengths of each base model to craft a more efficient and robust predictive model. Typically there are two main steps to an ensemble model. First, the ensembles train multiple weak learners (or models) to solve the same task. Second, they combine those results to enhance performance [Zhou, 2012]. Moreover, x will be the input data, not just a single feature but all of the features used by the model.

Mathematically, let $\{M_1, M_2, ..., M_n\}$ represent a set of base models. The goal of ensemble learning is to create a strong learner H as a combination of these base models:

$$H(x) = \frac{1}{n}\sum_{i=1}^{n} M_i(x). \tag{1}$$

In the equation (1) we see that to create a strong model we can average the sum of the basic learners. While this is one option for combining the models, we would typically have an outside understanding of which simple learners will fit better for our problem. This will lead us to want to, rather than simply averaging the sum, choose to give certain models a higher weight. Let $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ be the weights $(0 \leq \alpha_i \leq 1)$ assigned to each model based on its performance or expertise, and x is the input data.

$$H(x) = \sum_{i=1}^{n} \alpha_i M_i(x). \tag{2}$$

Visually, we can view an ensemble model as the following, for n=3:

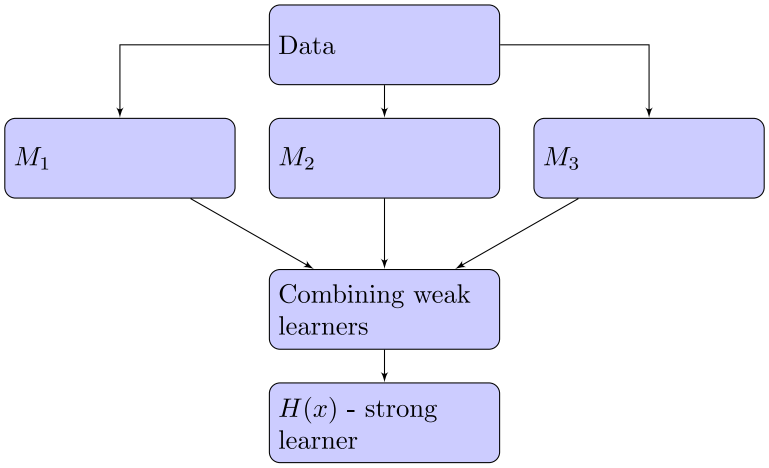


Figure 1: Ensemble Learning Process

The weak learners $\{M_1, M_2, M_3\}$ train using the input data. Then, the models output the predictions into the next node that combines the results either by averaging (1) or using weights (2) to get the strong learner which will produce more accurate results.

It is important to note then, that to get a good ensemble or to improve strong learners, we need to improve the performances of each base learner to maximize its potential. By doing so, we will improve the overall ensembled model performance while still balancing the bias and variance trade-off between each model.

As mentioned earlier, this paper will deal with boosting methods specifically. Boosting methods try to improve weak learners and turn them into strong learners not in parallel, but sequentially, before combining the results. We will explore boosting more thoroughly in the following sections.

2.3 Decision Trees

To be able to visualize boosting, we will first expand our background knowledge and familiarize ourselves with decision trees as a model.

As described by Zhou [Zhou, 2012], decision Trees are a popular, supervised machine learning model used for both classification and regression problems.

Definition 2.3 ***Classification*** *involves categorizing data points into predefined classes or categories (and not numerical values)*

Definition 2.4 ***Regression*** *involves predicting a continuous numerical value or quantity based on input features.*

Each node of the tree is associated with a feature test, called a split. The feature test splits the data into subsets, considered when the model tries to make a prediction. As the name suggests, we can visualize the decision trees in a flowchart diagram that resembles a tree. To demonstrate how a decision tree operates, let's consider a famous, specific dataset of penguins[3]. In the following

[3]Source: Kaggle.com, `https://www.kaggle.com/datasets/parulpandey/palmer-archipelago-antarctica-penguin-data`.

Table 1: Penguin Data

Species	**Bill Length (mm)**	**Bill Depth (mm)**	**Flipper Length (mm)**
Adelie	39.1	18.7	181.0
Adelie	39.5	17.4	186.0
Adelie	40.3	18.0	195.0
Chinstrap	46.0	18.9	195.0
Chinstrap	51.3	19.9	210.0
Chinstrap	50.0	18.0	187.0
Gentoo	47.2	13.7	211.0
Gentoo	49.5	16.1	210.0

table, we have 8 different penguins. The decision tree objective is to classify these penguins into their respective species, which include Adelie, Gentoo, and Chinstrap.

We can use feature-based questions to split the data to classify the penguins into their respective species:

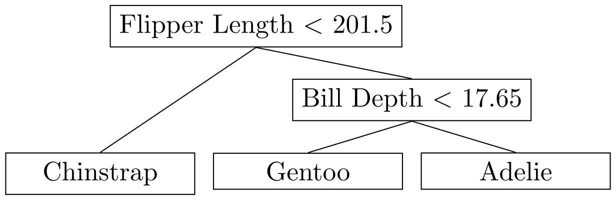


Figure 2: Decision Tree to predict specie

Described simply, interpreting a decision tree involves following the path from the root (top) down to the final leaf nodes (bottom) based on the features of the sample being evaluated. At each split, the tree evaluates a specific feature and directs the sample to a subsequent left or right child node based on whether the feature meets a certain condition.

The tree diagram above demonstrates how a decision tree can predict the species of penguins based on their flipper length and bill depth. With new data, we will follow the appropriate path down the tree diagram to classify penguin species. For example, let's consider a penguin with a flipper length of 190 mm and a bill depth of 17 mm. Following the tree's decision path, this penguin would be classified as an Adelie, as the first split based on flipper length will push it to the right, then again to the right based on the bill depth split test.

This means mathematically that decision trees are a piece-wise constant approximation [Nielsen, 2016]. The example above illustrates how any data set can be partitioned into smaller subgroups. That said, decision trees, in regression and classification, have the potential to further subdivide the data and theoretically result in an individual data point at the leaf ends. This a challenge that will be further discussed in section 2.4.

The decision trees model is quite a simple one. Its flowchart shape makes it easy to interpret, therefore non-experts can easily understand it. In addition, Decision Trees can handle mixed data types. Both numerical and categorical data types are easily manageable through decision trees, as the prediction depends on the split tests that can be applied to both data types.

2.4 Challenges with Decision Trees

Decision trees, powerful and interpretable as they are, are not without their challenges and limitations. To understand advancements in Machine learning such as boosting, we first need to understand these challenges and limitations, so we can explore how attempting to solve those led to improving on the existing model.

Overfitting is one of the most significant issues associated with decision trees. Decision tree models tend to overfit the training data. Overfitting occurs when a tree model has a high variance. The model focuses on capturing all the training data's noise rather than identifying the underlying patterns for making accurate predictions [Friedman, 1999]. As mentioned in section 2.3, when not built carefully, the decision trees model has a tendency to build trees that are excessively complex to fit the training data. Thus, those are less capable of generalizing well to new, unseen data. The consequences of overfitting include a decrease in predictive accuracy and it exposes the model to errors when applied to real-world scenarios.

Bias in decision tree models will occur more often in smaller datasets. The small sample size will hurt the model as it will become too generalized, exposing it to noise and outliers, causing it to not be able to capture complex relationships and patterns. When a tree's depth is not appropriately constrained, the tree can become excessively deep, resulting in too many split tests [Zhou, 2012]. The consequence of bias is that the model will favor majority classes, hurting the predictions of other classes which could cause major issues in real-world scenarios.

In the book "The Elements of Statistical Learning" [Hastie et al., 2009], two other challenges in Decision trees are thoroughly discussed by Hastie. One major challenge with decision trees is **instability**. Instability refers to how a small change in the training data will create a completely different tree. As we visualize decision trees as a top-down diagram, an error in the top split test, will trickle down to all the splits under it.

Another limitation of Decision trees discussed in Hastie's book is **Lack of Smoothness**. As mentioned before, we can think of decision trees as piece-wise constant approximations [Nielsen, 2016]. When dealing with binary classification problems, this limitation has a partial effect. That is, as the features are partitioned into regions corresponding to the classes, the lack of smoothness doesn't have a large effect on the model's performance. When we use decision trees in a regression setting, which are inherently discontinuous and non-smooth, they will struggle to predict values that are not specifically present in your dataset but can exist for a given feature or variable.

Simple decision trees face these challenges, but they are still a valuable efficient model that has many advantages with a large variety of fields and applications. The need to find a way to migrate, or even diminish the effect of decision trees limitations, will enhance the predictive capabilities of the models. Recognizing decision tree models' potential, combined with the advantages of ensemble methods, particularly gradient boosting, promises to overcome these limitations and improve performance in many real-world scenarios.

3 Gradient Boosting Models

A gradient is a mathematical term for a vector that represents the rate of change of a function, similar to a slope, but to each variable. Gradient boosting models get their names from the idea of sequentially improving a weak learner, optimizing the gradient over time. Thus, the gradients will change and be optimized to minimize the error of the loss function.

In his work [Friedman, 1999], Jerome H. Friedman introduced the concept of gradient boosting, using the advantages of ensemble methods to overcome the limitations of simple decision trees. To understand gradient boosting we will start by exploring the fundamental principles of boosting, initial predictions, loss function optimization, and model updates. To help illustrate these concepts we will refer to our table with the small penguin dataset (1), but this time as a regression problem in which we try to predict the Flipper Length (mm). While small, this dataset allows for visualization and ease of understanding of these gradient-boosting topics.

The typical gradient boosting algorithm uses complicated notation, but it can be understood as follows, in pseudocode:

Input:

1. $F_0(X)$ - Initial prediction (further discussed section 3.1)
2. a differentiable loss function $L(y, F(x))$ where y is the observed response variable, and $F(x)$ is the predicted value.

Steps:

1. For (from m = 1 to M) do: (where M is the maximum number of trees we want to create)
    - Calculate Residuals
    - Fit a regression tree to these residuals
    - Update predictions in a new tree, based on previous predictions
2. Output final prediction

Further elaboration will follow in sections 3.1 to **??**.

3.1 Initial Prediction

The dataset includes a set of eight penguins, where each penguin is described using some numerical features, such as bill length (mm) and bill depth (mm), and a categorical feature such as the species, as shown in Table 1. Let the training sample be $\{X_i, Y_i\}_{i=1}^{N}$, where $X_i = \{X_1, X_2, ..., X_n\}$ are the set of input features, and Y_i is the Flipper Length (in millimeters) for the i-th penguin in the selection sample of 8 penguins from the dataset.

Compared to simple decision tree models that start by building a tree (figure 2), the gradient boosting algorithm starts by making a single leaf, the initial prediction leaf.

In binary classification problems, when trying to classify between 2 classes, the initial prediction is defined in [Hastie et al., 2009] we will start by finding the odds and using it to find the probability which we will use as an initial value:

$$\log(odds) = \log(\frac{y}{1-y}), \tag{3}$$

where y is an observed value between 0 and 1. p is the predicted probability and is defined as:

$$p = \frac{e^{\log(odds)}}{1 + e^{\log(odds)}}. \tag{4}$$

As we are dealing with a regression problem, our initial prediction is defined as:

$$F_0(X) = \frac{1}{N} \sum_{i=1}^{N} \bar{Y}_i. \tag{5}$$

Using the function (5), and knowing that N=8, we can create an initial prediction leaf as follows:

$$\bar{Y} = \frac{1}{8}\sum_{i=1}^{8}\bar{Y}_i = \frac{1}{8} * 1575 = 196.875(\text{mm}) \tag{6}$$

That is, our initial guess for flipper length, for all of the penguins, will be 196.875 mm. Using the gradient boosting algorithm [Friedman, 1999], we start by taking the initial prediction, to build the first tree as a simple model that captures the residuals of the initial prediction's errors.

Definition 3.1 ***Residuals*** *are the differences between the observed values (actual values) and predicted values.*

$$Residual_i = Y_i^{(observed)} - Y_i^{(predicted)}$$

Simply put, we can think of the residuals as the model's errors.

Let's first update Table 1 to include the residuals based on our model's initial guess. For example, the residual for the first penguin will be calculate in this fashion:

$$181.0 - 196.875 = -15.875$$

Table 2: Penguin Data with Residuals

Species	**Bill Length (mm)**	**Bill Depth (mm)**	**Flipper Length (mm)**	**Residuals**
Adelie	39.1	18.7	181.0	-15.875
Adelie	39.5	17.4	186.0	-10.875
Adelie	40.3	18.0	195.0	-1.875
Chinstrap	46.0	18.9	195.0	-1.875
Chinstrap	51.3	19.9	210.0	13.125
Chinstrap	50.0	18.0	187.0	-9.875
Gentoo	47.2	13.7	211.0	14.125
Gentoo	49.5	16.1	210.0	13.125

The ending of the first step of gradient boosting deals with the creation of the first tree. Compared to decision trees, the gradient boosting algorithm is different because it doesn't try to predict the response variable, but rather creates more emphasis on predicting the residuals based on the input features. We will manipulate our dataset species column in our set of input features X for that purpose. That is, we will alter the categorical class so it will become a binary type. Simply put, a penguin's species can be an Adelie, denoted by a value of 1, or not be an Adelie, denoted by a value of 0. Following these changes, this is the new table for the input features X:

Table 3: Input Features for Initial Gradient Boosting Tree

Bill_Length	**Bill_Depth**	**Species_Adelie**	**Species_Chinstrap**	**Species_Gentoo**
39.1	18.7	1	0	0
39.5	17.4	1	0	0
40.3	18.0	1	0	0
46.0	18.9	0	1	0
51.3	19.9	0	1	0
50.0	18.0	0	1	0
47.2	13.7	0	0	1
49.5	16.1	0	0	1

Using this table, we can now create our first tree, figure (3), to see how new data will be split. To interpret the figure, each node contains a split test of one of the input features. The split will determine where each data point will be assigned. Under the split test, the "squared_error" stands for the mean squared error (MSE) associated with the node. It quantifies how well the split at that node predicts the target variable. "samples" stands for how many data points, in our case penguins, reached this node, and the "value" is equal to the predicted target values, or the residual of the data point. If *samples* > 1, it will typically be the average of the residuals that reached this node. We are almost able to match the values part and see which penguin from 2 got to which node by comparing the residuals column to the values of each node.

Definition 3.2 ***Mean Squared Error (MSE)*** *is a metric often used to measure to check the model's prediction accuracy, it's calculated by the average squared difference between the predicted and actual values of the output feature:*

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \ .$$

It is important to note here, that the MSE is commonly used as the loss function $L(y, F(x))$ in gradient boosting, and it will be used as the loss function in our example.

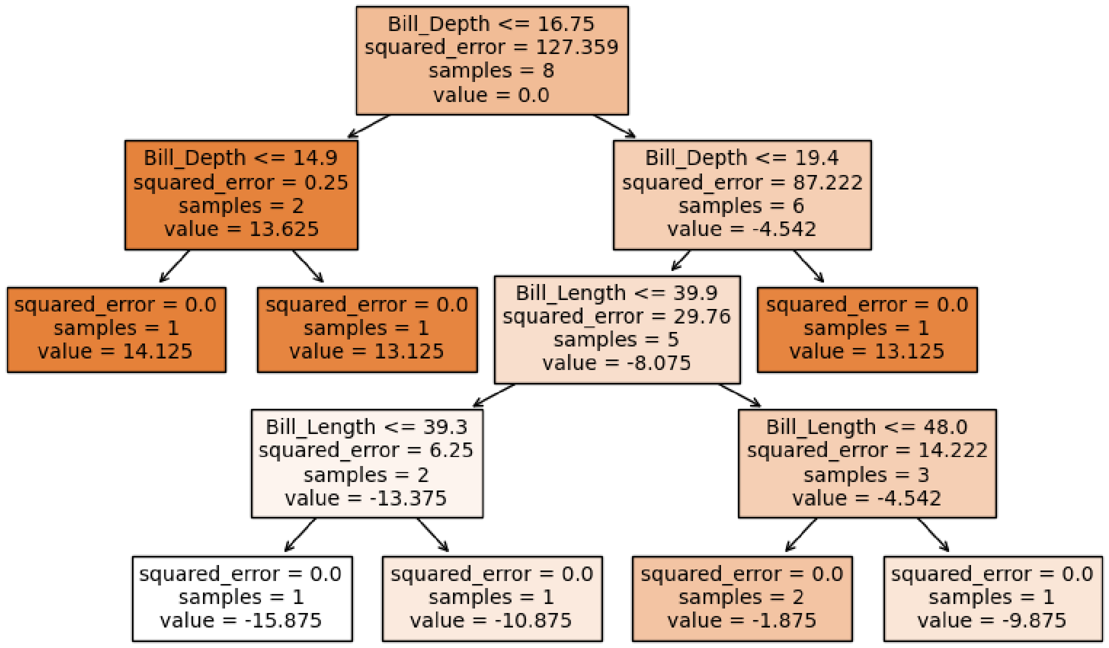


Figure 3: Initial Gradient Boosting Tree

In the figure above, we can see an example of the initial tree created using Table 3. The tree uses split tests to separate the individual penguins into the leaf nodes that predict their residuals. We will note that because this is the initial tree, the values in the leaf nodes are the actual initial residuals (or their average if more than one individual belongs to a leaf).

3.2 Update the model

In the preceding subsection, we created an initial prediction leaf and used it to build the first tree. The next steps of the Gradient Boosting algorithm, involve the the boosting process itself. Simply, the process of refining and updating the model iteratively to correct the errors in the previous predictions. After all, the strength of the gradient boosting algorithm comes from its ability to learn and improve with each tree built.

In order to build the new tree, we need to use the initial prediction leaves from Figure 3. We will get our new prediction in the following way based on the work of [Friedman, 1999]:

$$F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x), \tag{7}$$

where $F_{m-1}(x)$ is the prediction of the last iteration. $h_m(x)$ is a weak learner (specifically the residuals) or the error or the prediction, and ν is the learning rate (default $\nu = 0.1$). Note that $0 < \nu < 1$ because if we simply add $F_{m-1}(x) + h_m(x)$, we will get the observed values back. A learning rate is introduced to make sure the user controls how much of the new model is changed in each iteration. This reduces overfitting in the model and generalizes it better to new data.

Let's leave the theoretical realm and apply this new knowledge using the initial prediction and tree from section 3.1. We know $F_0(x) = 195.875$(mm), and that $h_m(x)$ is the residuals (displayed in the leaves of the decision tree in Figure 3.

For ease of understanding, we will focus only on the leaves on the left side of our initial tree.

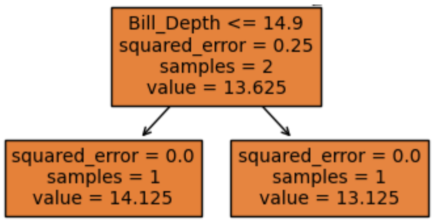


Figure 4: Subset of initial tree (3)

Now, we apply equation (8) on the values from the leaves to create new predictions:

$$F_1(x) = 195.875 + \left(0.1 \cdot \begin{bmatrix} 14.125 \\ 13.125 \end{bmatrix}\right) = \begin{bmatrix} 197.2875 \\ 197.1875 \end{bmatrix} \tag{8}$$

Here, the matrix cells represent the new predictions for each leaf node. We will use these values to find the new residuals for those data points based on the predictions. The next tree uses the updated residuals as the new target values. It's important to note whether the split tests in subsequent trees typically remain the same, depending mostly on the parameters used. The cycle repeats the loop until it is completed when M trees are built. The integer M will typically be a large number and is set by default to $M = 100$. Another stopping criterion is when certain convergence criteria are met. A convergence criterion, specified as the early stopping parameter, is a feature that prevents unnecessary training iterations. When met, it means that the model is no longer improving. That is, the model is either barely improving or is starting to produce worse predictions. This ensures that the boosting process doesn't continue indefinitely.

3.3 Optimizing the Loss Function

As mentioned in the preceding sections, the key ingredient for predictive models is the loss function. When optimized correctly, it helps the model make more accurate predictions and helps uncover hidden patterns within the data. To predict the flipper length, which is a regression problem, we used the common MSE (mean squared error) function as a loss function. This is not always the case, in fact, we do not have a standard loss function that fits all cases. Friedman discusses different ways to apply gradient boosting in his applications section, using different loss functions for different uses [Friedman, 1999].

As previously identified, the common loss function for regression problems is the MSE:

$$L(y, F) = \frac{1}{2} * (y - F)^2. \tag{9}$$

MSE is common because it is a smooth and differentiable loss function. The differentiable part is a big advantage as it allows the employment of optimizing algorithms to train the model

[Nielsen, 2016]. Thus, if we prioritize precision, we will choose MSE as the loss function for regression cases.

There are some cases, however, for example when our data is skewed or has many outliers. In these cases, we would refer to Least Absolute Deviations (LAD) as the loss function:

$$L(y, F) = |y - F|. \tag{10}$$

The reason LAD works better for skewed data is mainly because the initial prediction in LAD estimation is the median, not the average of the data, which makes it closer to the data's central tendency. Unlike the mean, which is greatly influenced by outliers, the median is resilient to extreme values. Therefore, if we prioritize robustness, the ability to maintain reliability at extreme and unexpected data, we will refer to LAD as the loss function.

For categorical problems, we will have to take a different approach. That is, we will have to use a different loss function based on the output variable range of distinct values.

For binary classification, where we have to predict between 2 classes (typically '0' or '1') we will use the Log-Loss loss function.

$$L(y_i, F(x)) = -\left[y_i * \log(p) + (1 - y_i) * \log(1 - p)\right] \tag{11}$$

This loss function, while computable, is a function of the log of predicted p probability. Furthermore, we need to simplify this equation to understand what it computes. We will want to refer back to equation 3 and use the log of the odds for easier computation as well as to show that the loss function is differentiable.

$$-\left[y_i * \log(p)\right] - \log(1 - p) + y_i * \log(1 - p)]$$

We'll take the y_i out and use logarithmic properties:

$$y_i * \left(\frac{\log(p)}{\log(1 - p)}\right) - \log(1 - p)$$

$$y_i * \log(\text{odds}) - \log(1 - p) \tag{12}$$

At this point, we have found our loss function. Before checking that the loss function is differentiable, we will want to simplify $\log(1 - p)$ using the value of p from equation (4):

$$\log(1 - p) = \log\left(1 - \frac{e^{\log(odds)}}{1 + e^{\log(odds)}}\right) = \log\left(\frac{1 + e^{\log(odds)}}{1 + e^{\log(odds)}} - \frac{e^{\log(odds)}}{1 + e^{\log(odds)}}\right),$$

using a common denominator, we can subtract the $e^{\log(odds)}$ from the numerator, and from there all that is left is a simple logarithmic that we can simplify further:

$$= \log(1) - \log(1 + e^{\log(odds)}) = -\log(1 + e^{\log(odds)}). \tag{13}$$

Next, to minimize a function, we can check that the loss function (12) is differentiable:

$$\frac{d}{d(\log(odds))} y_i * \log(odds) - \log(1 - p)$$

$$L(y_i, F(x)) = -y_i * \frac{e^{\log(odds)}}{1 + e^{\log(odds)}} \tag{14}$$

Simply put, we have found that when differentiating the Log-Loss function we get the negative (or opposite) of the residual:

$$L(y_i, F(x)) = -\text{Observed} + \text{Predicted} = -(\text{Observed} - \text{Predicted})$$

Therefore, because the Log-Loss function can be differentiable, the GBM could use its minimization using the approximation of the gradients, and we can use it as a loss function.

3.4 Challenges with GBM

The gradient boosting model, while powerful, is not without its challenges. One primary challenge is the risk of overfitting. Specifically, because of the gradient boosting models' iteratively nature, when the algorithm is allowed to continue boosting beyond the point of optimal model performance. As a result, the model will create complex trees that fit the data points too closely, failing to recognize its underlying patterns.

Moreover, gradient-boosting algorithms tend to suffer from high sensitivity to hyperparameters. Specifically, the learning rate and the maximum depth of each tree hyperparameters are key for model performance [Friedman, 1999]. Carefully choosing the correct values will increase the accuracy and performance of the model. Conversely, the wrong learning rate or the maximum depth can create an under-fitting or an over-fitting tree. The ability to find optimal approximation requires initial exploration and a deeper understanding of how the final accuracy of the predictions should look. Simply, at what point is the accuracy good enough, implying to stop the model to avoid over-fitting the model.

In addition, training a gradient-boosting model is computationally heavy. Simply, it requires more time and resources for training compared to simpler algorithms [Nielsen, 2016]. When dealing with an extensive data set, the algorithm will underperform speed-wise.

4 XGBoost

With an increased need for data-driven machine-learning algorithms, tree-boosting methods such as GBM achieve state-of-the-art results for many applications. With that, as described in section 3.4, the algorithm still faces many limitations. XGBoost, for eXtreme Gradient Boosting, is an advancement of the regular gradient-boosting models. In terms of scalability for different scenarios, as well as the model's accuracy, efficiency, and generalization, XGBoost offers a significant boost over its predecessors. As Tianqi Chen wrote in his paper [Chen and Guestrin, 2016a], among 29 machine learning challenges posted in Kaggle, 17 winning solutions for those challenges used XGBoost to train the model (with almost half using XGBoost alone). The XGBoost models outperformed all other solutions submitted in a diverse domain of fields such as sales prediction, product categorization, hazard risk prediction, and more.

In the following sections, we will delve into an in-depth exploration of the mathematical theory that allows XGBoost, still a gradient-boosting model, to outperform all other tree-boosting models with any given data set in a wide variety of problems. Similarly to GBM, we will start the XGBoost process by making an initial prediction. With that, XGBoost tends to make an initial prediction of $f_0(x) = 0.5$ for both regression and classification problems. The small data set above will allow us to demonstrate the process of XGBoost on a small scale and visualize it.

For the creation of the initial decision tree, XGBoost scans for the appropriate split tests by creating a scoring system called similarity score, for choosing the splits that produce the smallest score each time. Simply, XGBoost builds decision trees as a greedy algorithm:

Definition 4.1 ***A greedy glgorithm*** *is a problem-solving approach that makes local optimal choices at each step [Chen and Guestrin, 2016a]. The algorithm doesn't guarantee the most optimal solution but is computationally efficient.*

As mentioned, XGBoost's initial prediction is $f_0(x) = 0.5$. In addition, the XGBoost loss function we will define and use is the MSE (9).

4.1 Adaptive Trees

From [Chen and Guestrin, 2016a], XGBoost builds adaptive trees by minimizing a regularized objective function defined as follows:

$$\mathrm{obj}^{(t)} = \sum_{i=1}^{n} \mathrm{L}(y_i, p_i) + \gamma T + \frac{1}{2}\lambda O^2_{(\text{value})}, \tag{15}$$

where L is the loss function MSE of actual values y_i and prediction p at the i^{th} instance, γ is a user-defined penalty parameter, and T stands for the number of leaves in a tree. We can omit γT from the objective function as in section (4.3) we will see it plays no part in minimizing the optimal similarity scores as we will see in the next subsection 4.1.1. In addition, λ is a regularization term, with $O^2_{(\text{value})}$ standing for the output value for the leaf that will minimize the whole equation. For now, without loss of generality, we will set $\lambda = 0$ for ease of learning.

We will want to optimize the output value of the prior tree created. Therefore, we will add to the prediction term p_i the output value for the leaf:

$$\mathrm{obj}^{(t)} = \sum_{i=1}^{n} \mathrm{L}(y_i, p_i + O_{\text{value}}). \tag{16}$$

From the equation above, we need how the output O_{value} will be added to the prior prediction. In addition, before the creation of the first tree that value will be 0.

4.1.1 Second-order Taylor expansion of the loss function

Compared to simple, unextreme gradient boosting models, XGBoost utilizes the second-order Taylor expansion of the loss function to find the leaves output value that will minimize the loss function. When applied on the loss function, we will derive at

$$\mathrm{L}(y_i, p_i + O) \approx \mathrm{L}(y_i, p_i) + \left[\frac{\partial}{\partial p_i}\mathrm{L}(y_i, p_i)\right] O + \frac{1}{2}\left[\frac{\partial^2}{\partial p_i^2}\mathrm{L}(y_i, p_i)\right] O^2. \tag{17}$$

We will rename the first derivative of the loss function with respect to p_i as g_i for Gradient. This is because the derivative is the rate of change of the loss function. We will compute it and find that it is equal to the residual:

$$g_i = \frac{\partial}{\partial p_i}\mathrm{L}(y_i, p_i) = \frac{\partial}{\partial p_i}\left(\frac{1}{2}(y_i - p_i)^2\right) = y_i - p_i.$$

Then, following the same logic, we will rename the second derivative as h_i for the Hessian matrix. We will compute it and find that it is equal to:

$$h_i = \frac{\partial}{\partial p_i}(y_i - p_i) = -1$$

XGBoost lives up to its name because it results in extreme measures when it comes to saving computational power. As described in [Chen and Guestrin, 2016a], when applied to the objective equation 16, it will omit all of the constants that are not related to output term O. Therefore, we will end up with the following:

$$\text{obj}^{(t)} = (g_1 O + h_1 O^2) + (g_2 O + h_2 O^2) + ... + (g_n O + h_n O^2) + \frac{1}{2}\lambda O^2$$

$$\text{obj}^{(t)} = (g_1 + g_2 + ... + g_n)O + \frac{1}{2}(h_1 + h_2 + ...h_n + \lambda)O^2$$

Remember that we want to minimize the output for every leaf. Therefore we will take the derivative with respect to O and set it equal to 0:

$$0 = (g_1 + g_2 + ... + g_n) + (h_1 + h_2 + ...h_n + \lambda)O$$

We can now find the value of O as corresponding to the minimum of the objective function:

$$O = \frac{-(g_1 + g_2 + ... + g_n)}{(h_1 + h_2 + ...h_n + \lambda)}$$

Therefore, we now know that it is simply:

$$O_{\text{value}} = \frac{\sum_{i=1}^{n}(y_i - p_i)}{n + \lambda}. \tag{18}$$

Equation 18 is proof that demonstrates that any differentiable loss function can be used for the objective function of XGBoost. This is a known advantage of XGBoost as users can now customize the algorithm so it is optimized for every scenario. It allows for further customization of the algorithm by the user, which data scientist take advantage of when they have full understanding of the problem they are trying to solve.

4.1.2 Building First Tree

To understand how XGBoost creates adaptive trees that are more efficient than the simple gradient boosting algorithm, we will use the following data set:

Index	**x**	**y**	**Residuals**
0	1	-8.5	-9.0
1	4	8.5	8
2	5	12	11.5
3	8.5	-7	-7.5

Table 4: XGBoost data frame with Residuals

The data set above is small and therefore will allow us to go through the steps and calculations that XGBoost follows. Note that the data set only has one input variable, which makes it easier to plot on a scatter plot. This will not change the process of the model when scaled to more input variables.

For visualization purposes, we will plot the points in the following scatter plot:

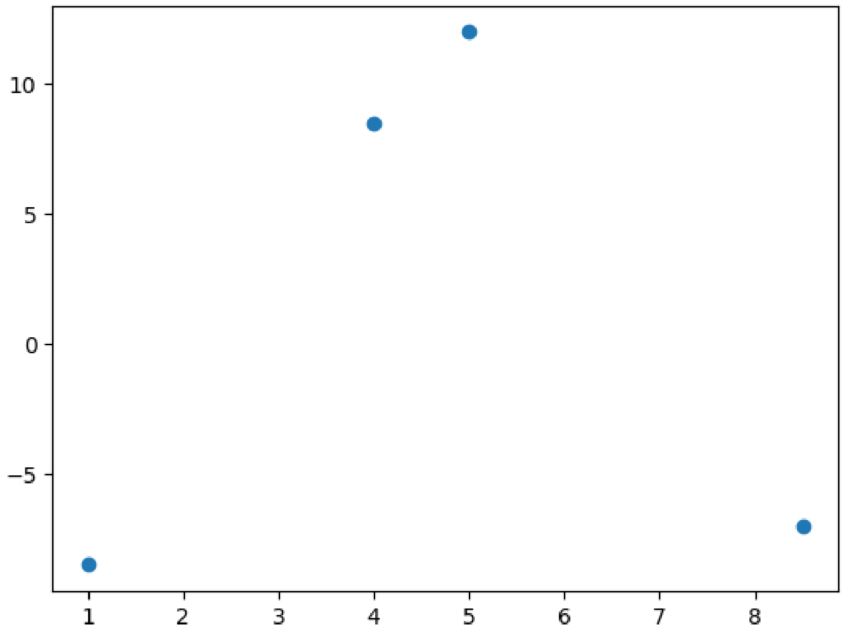


Figure 5: XGBoost data set Scatterplot

In addition, to score a split, XGBoost will use a similarity score in the following way:

$$\text{Similarity Score} = \frac{\sum_{i=1}^{n} \text{residual}^2}{n+\lambda} \tag{19}$$

where n is the number of data points (i.e., the number of residuals), and λ is a regulation term. We will explore the regulation term λ thoroughly in section 4.3. For now, without loss of generality, we will set $\lambda = 0$ for ease of learning.

The similarity score is derived by inserting the value of the O_{value} to the equation 18. XGBoost starts by calculating the similarity score of all the residuals as if they were in one leaf:

$$\text{Similarity Score} = \frac{(-9+8+11.5+-7.5)^2}{4} = 2.25 \tag{20}$$

XGBoost will use this similarity score as a tool for comparison when testing if it could split the data better using split tests. To do that, XGBoost proceeds to check each split test by averaging the midpoint between the input x_0 and x_1, in this case, $\hat{x} = \frac{x_1 - x_0}{2} = \frac{4-1}{2} = 2.5$ Now with the initial similarity score computed, we can test if the split test $x < 2.5$ will split the data points better.

Below is a scatter plot to visualize the data set:

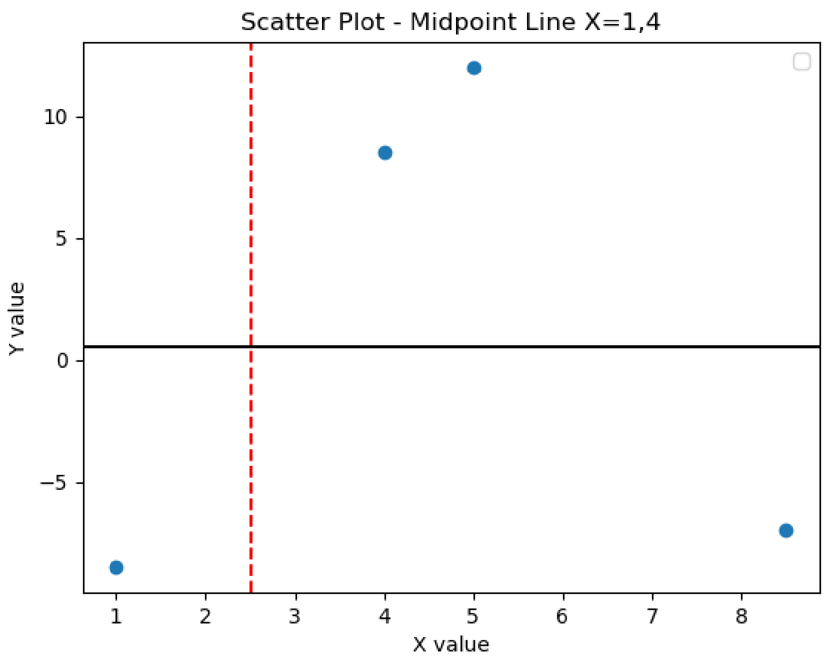


Figure 6: df scatter with midpoint and $f_0(x)$

In figure 6, the black horizontal line $y = 0.5$ visualizes the initial prediction, and the red dotted vertical line $x = 2.5$ visualizes the split test we want to test. We can use the scatter plot and build a decision tree to visualize the split of the data points and their residuals better.

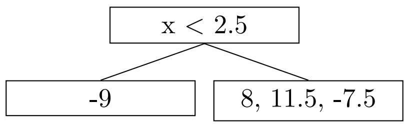


Figure 7: Decision Tree based on x < 2.5 split

XGBoost will now compare the similarity score of each of the leaves to see if the split did a good job. We will compute the similarity score of the left-hand leaf first, denoted as lhsc (for left-hand similarity score):

$$\text{lhsc} = \frac{(-9)^2}{1+0} = 81$$

We will note that the similarity score of the left-hand side is extremely high compared to the similarity score computed originally. This suggests that the split test $x < 2.5$ was a good split because of this . We will keep track of lhsc = 81 for further comparisons.

Following the same logic, we will compare the similarity score of the right-hand leaf, denoted as rhsc:

$$\text{rhsc} = \frac{(8 + 11.5 - 7.5)^2}{3 + 0} = 48$$

Thus, now that XGBoost has found the similarity score of the root, the left-hand leaf, and the right-hand leaf, it can now calculate the score of the gain. XGBoost tries to maximize the gain value. Simply, as XGBoost is a greedy algorithm, the split test that has the largest gain will be the one selected at each iteration. The gain is calculated as follows:

$$\text{gain} = \text{lhsc} + \text{rhsc} - \text{root}_{\text{similarity}}. \tag{21}$$

Thus, the gain of the split test $x < 2.5$ is

$$\text{gain}_{x<2.5} = 81 + 48 - 2.25 = 126.75.$$

XGBoost will then proceeds to compute the gain of the next split test by averaging the midpoint between the input x_1 and x_2, in this case, $\hat{x} = \frac{x_2 - x_1}{2} = \frac{5-4}{2} = 4.5$.

We will visualize the test split in the following scatter:

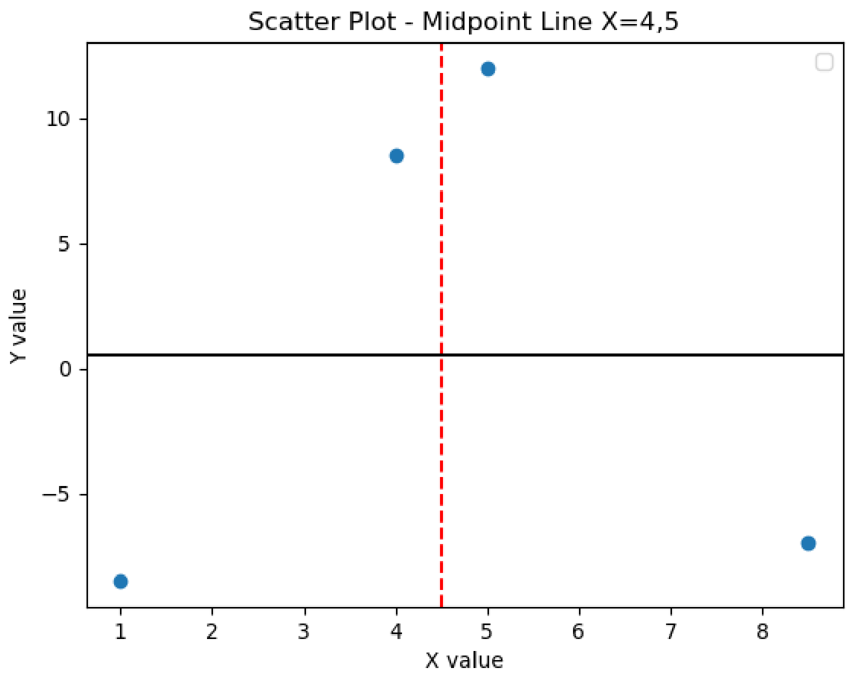


Figure 8: df scatter with midpoint and $f_1(x)$

Similarly to figure 6, in figure 8 still has a black horizontal line $y = 0.5$ visualizes the initial prediction, and the red dotted vertical line $x = 4.5$ for the split test we want to test.

We can intuitively build the tree, using the scatter plot above, in the following way:

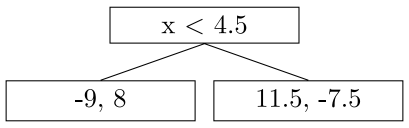


Figure 9: Decision Tree based on x < 4.5 split

Thus, following the same logic, we will compute the gain to be:

$$\text{gain}_{x<4.5} = 1 + 8 - 2.25 = 6.75.$$

Therefore, because the gain of the split test $x < 4.5$ is relatively small, just slightly more than the root similarity score, it will not be chosen. XGBoost will continue to check any split test possible between the input variables.

The largest gain will be the gain of the split test $x < 2.5$, $\text{gain}_{x<2.5} = 126.75$.. Thus the tree built will initially be 7. With that, the right leaf has the residuals of multiple data points. XGBoost will turn the right leaf to a root, repeating the exact process, calculating the similarity scores, and maximizing the gain. Skipping the detailed calculations, the split test selected will be $x < 6.75$. Users can control XGBoost's level of complexity by choosing the max depth of each tree according to their needs, to avoid under or overfitting. In our case, there is only a single leaf with multiple data points. Calculating its gain we will find that it will be a small value. Thus, we will leave it as, and the output of that leaf will be the mean of the residual values.

Finally, XGBoost finished building the first adaptive tree:

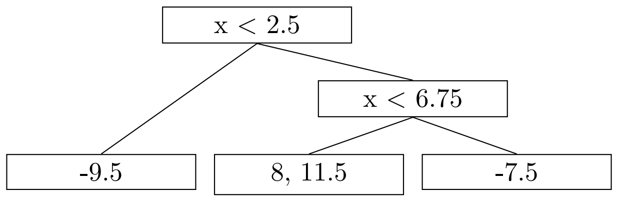


Figure 10: XGBoost Initial Decision Tree

Using figure 10, a new data point of $x = 5$, will be sent from the root to the right node because it is bigger than 5. From there it will be sent into the left child node because it is smaller than 6.75. Thus, XGBoost, for the data point $x = 5$, will predict a residual value of

$$\frac{8 + 11.5}{2} = 9.75.$$

4.1.3 Updating the XGBoost Model

Similar to gradient boosting algorithms, XGBoost can now make predictions. XGBoost will use the same equation as gradient boosting (8). The learning rate for XGBoost $\nu = 0.3$ by default, but can be changed based on the problem's needs.

Next, after making new predictions on each data point, XGBoost will iterate the entire process over. It will loop, using the new predictions to update the model iteratively, building new trees and improving its predictions over time.

This part of XGBoost is similar to unextreme gradient boosting algorithms and was described in section 3.2. It is essential to remember that there are 2 stopping criteria that the algorithm employs. One is the early stopping hyperparameter, a user-defined numerical threshold that prompts the model to stop training if it ceases to improve, resulting in the storage of the model, to reduce computational power. The second stopping criterion is when the XGBoost model has built 'n' trees. Where n = 1000 by default, but can be defined differently by the user. This will control the training process and will allow for 'n' boosting rounds.

4.2 Parallel and distributed computing

Section 4.1.2 demonstrated the mathematical theory of how XGBoost finds the split tests that the adaptive tree uses to split the data. We will have to remember that the data set 4 had a small number of data points for ease of demonstration. What if XGBoost is dealing with a data set with many data points (and not necessarily with a large range of input variables)?

XGBoost once again lives up to its name by using the computing advancement of multi-core processors and allows for parallel and distributed computation. Instead of calculating the gains of the midpoints iteratively, it will create quantiles of the data and check those instead. Moreover, XGBoost will parallelize the computation process of similarity scores, gain calculations, and tree building. This enables XGBoost to achieve better parallelizing and thus quicker training of the model as mentioned in Chen [Chen and Guestrin, 2016a].

More precisely, XGBoost will employ a technique called "Column Block". The most time-consuming part of tree learning is sorting the data properly [Chen and Guestrin, 2016a]. XGBoost will use in-memory units, called *blocks.* The blocks, stored in the compressed column (CSC) format, can visualized simply as the sorted values of a feature, stored in a column separately from each other. Furthermore, XGBoost can use distributed computing to store columns across different machines, and easily parallelize the computation to find the split test for each column block at the same time.

The scalability of XGBoost is significantly amplified through its parallel and distributed computing abilities. As mentioned earlier in section 4, this is one of the main reasons for the robust performance using data sets of diverse sizes, for a variety of applications. Thanks to the "Column Block" technique, the effect of moderately large data sets or navigating through big data scenarios, XGBoost training time remains short, while model training efficiency consistently upholds.

4.3 Regularization

Regularization techniques play a central role in maximizing the performance and generalization ability of machine learning models. XGBoost uses two regularization methods, L1 and L2 regularization. The following section will explore how employing regularization correctly can mitigate the risk of overfitting by managing model complexity.

l2 regularization, known as *ridge regression*, uses λ as a regularization term and applies it to an output value, the score of the tree, as seen in equation 15:

$$\frac{1}{2}\lambda O^2_{\text{(value)}}. \tag{22}$$

Mathematically, it will penalize the model complexity of the trees. That is because XGBoost tries to minimize the objective function. score of a tree that is too high will add much weight to the model. XGBoost will then push to under-fit the model, as that will minimize the function.

Considering this, we can refer to the penguin data set in Table 1 to visualize how the regularization term λ affects the model built. We will use the input variable to try to predict the flipper length.

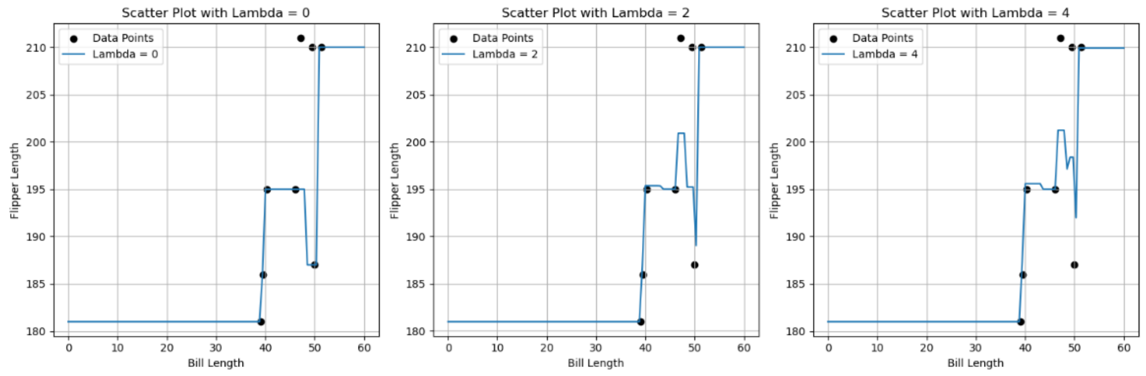


Figure 11: λ effect on XGBoost

Figure 11 displays three XGBoost models built using 3 different values of λ. For ease of visualization, the scatter plots only show the predictions based on the bill length, yet the model was built using all the input variables.

Examining the scatter plots left to right, we see that the model built with $\lambda = 0$ is over-fitting as it fits the data points very closely. As the value of λ increases, the models being built become more abstract, and we can even claim that when $\lambda = 4$ the model is under-fitting the data points. Simply, while lower values of λ may lead to over-fitting, higher values can result in under-fitting. Consequently, carefully choosing the optimal value of λ is important for the performance of an XGBoost model.

Similarly, XGBoost deploys l1 regularization, known as *lasso regression*. That is, XGBoost uses the term λ in the loss function to penalize the output, or weight, of the leaves.

Demonstrated mathematically, we will use the loss function as derived using the second-order Taylor expansion in equation 17. As described in the XGBoost GitHub repository, XGBoost will add the l1 regularization term as follows [Chen and Guestrin, 2016b]:

$$\text{Loss}(y_i, p_i + O) \approx g_i O + h_i O^2 + 2\alpha |O_{\text{value}}|. \tag{23}$$

Mathematically, the l1 regularization term α controls the penalization on the leaf weights [Nielsen, 2016]. In theory, lasso regularization can decrease the weight, or coefficients, of the leaf down to zero. Thus, it might shrink certain leaves completely, changing the structure of the model.

Similarly to Figure 11, we will use the penguin data set in Table 1 to visualize how l1 regularization term α affects the model built.

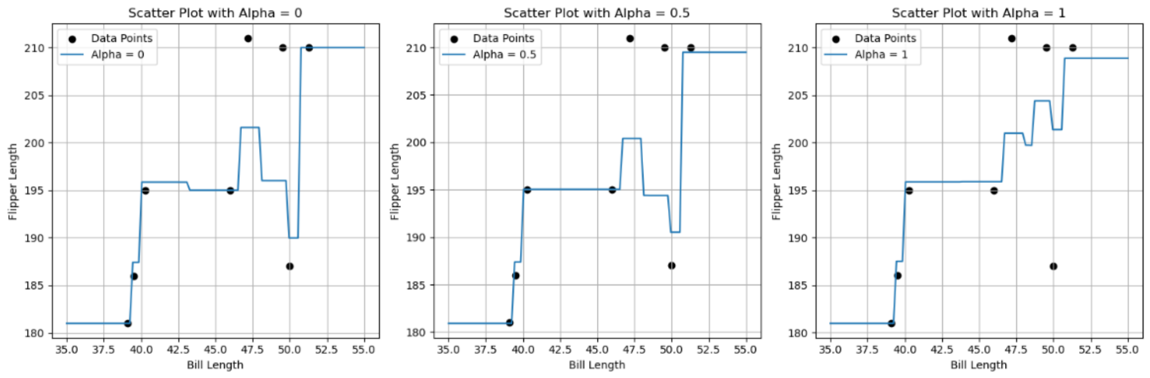


Figure 12: α effect on XGBoost

In the same way, Figure 12 visualizes three XGBoost models built using 3 different values of α. Not surprisingly, l1 regularization has a similar effect on the model built as l2 regularization as seen in figure 11. That is, as the value of α increases the complexity of the model decreases.

Ultimately, choosing the optimal value of α and λ is important for overcoming the bias-variance trade-off and finding a model that is neither over-fitting nor under-fitting. When applied correctly, both techniques allow XGBoost to strike a balance between model simplicity and predictive accuracy.

5 Application - House Prices in Ames, Iowa

The house prices in Ames, Iowa data set was found on kaggle.com [4] The training data set includes 1460 entries and it includes 79 explanatory variables describing various aspects of residential homes. It is known as a good data set for performing creative feature engineering. In addition, it allows the use of advanced regression techniques like gradient boosting for regression. Thus, we will use it exactly for this purpose, trying to predict the sale price. The section will first explore the data set to see if it needs further processing or if we can immediately recognize patterns. It will demonstrate some transformation in order to increase the models' performance. Finally, run both GBM and XGBoost separately to compare accuracy and speed. The subsection will not explore new topics, only introduce them on a basic level.

In addition, something must be said about the importance of a project like this outside the academic realm. Predicting house prices is crucial in real estate, finance, and investment decisions. Accurate predictions empower professionals to set competitive prices, aiding both sellers and buyers. Applying advanced algorithms such as GBM and XGBoost enhances our understanding of the real estate market, allowing for data-driven decision-making.

We will note that the section will not contain some of the wrangling preprocessing steps of the data set, meaning transforming the raw data into a format that can be easily and effectively used. Those can be found in the appendix.

[4]Source: kaggle competitions, `https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques`.

5.1 Data Exploration

Staring the exploration of the data set, we will want to plot the housing distribution for the exploration of important patterns.

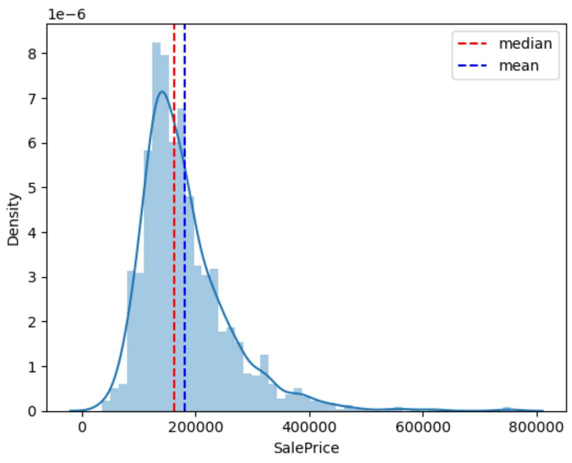


Figure 13: Sale Price distribution

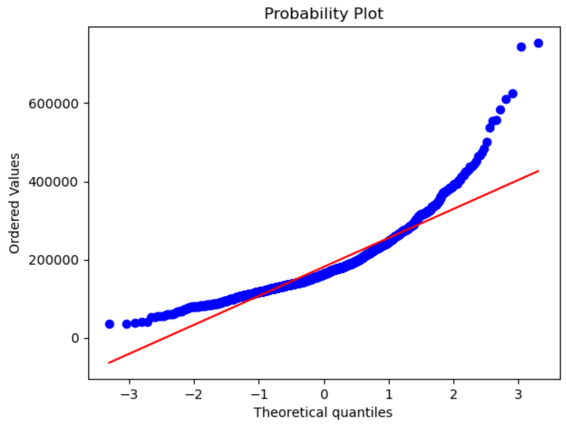


Figure 14: Sale Price q-q plot

Missing from the distribution plot in Figure 13 are the values of the median and mean. The median sale price is \$163,000.0 and the mean sale price is \$ 180,921.2. This, along with the shape of the distribution, suggests that the target value is right-skewed. Consequently, we have to confirm the visual distribution plot. We use a Q-Q Plot to check how well will the data lie on the straight line $y = x$ as displayed in Figure 14.

We will dissect the plot in the following way; If the data points (in blue) sit closely on the straight line, it means that there's a normal distribution of the variable's values. Thus, as the data does not sit on the straight line, the dependent variable should be transformed. A log transform was performed by applying the log on all the sale price variable values. The density distribution now looks as follows in Figure 15, increasing the model's performance as the outliers are de-emphasized. We can view the effect on the q-q plot as well, as it now sits closely along the straight line.

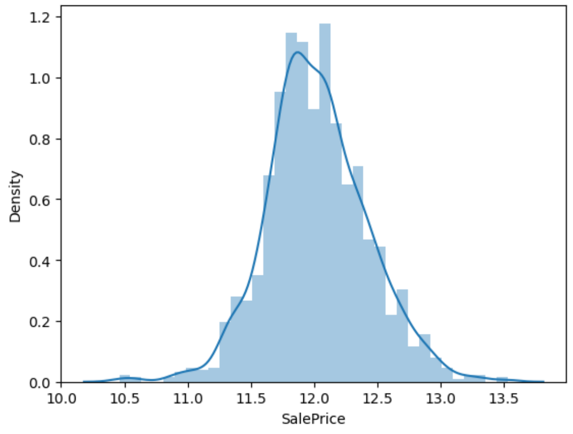


Figure 15: Transformed Sale Price distribution

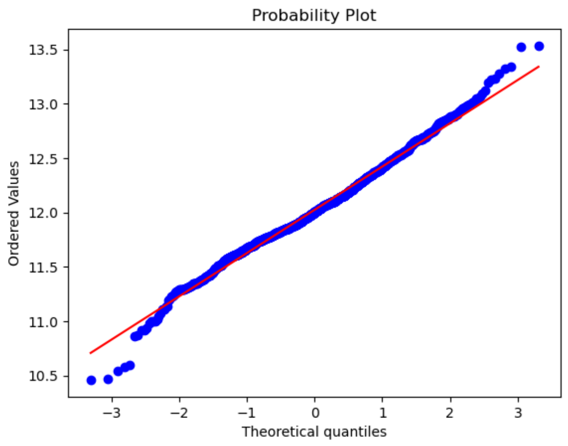


Figure 16: Transformed Sale Price q-q plot

In addition to the transformation of the dependent variable, we derived a new variable called "PropertyAge" from other variables in the following fashion:

```
df['PropertyAge'] = df['YrSold'] - df['YearBuilt']
```
,

with the understanding of the effect that the property age has on the sales price in the real estate field. With the exploration steps now completed, we will jump to the results of the gradient-boosting and XGBoost models when applied to the transformed data set. The additional in-between step, in which data pipelines are applied to the data set can be found in the appendix that contains the full Jupyter Notebook.

5.2 Model Results

The performance of the XGBoost model is compared with the scikit-learn gradient boosting model using various metrics. Table 5 summarizes the results obtained from both models when predicting sale price with the Ames Iowa data set.

Metric	**scikit-learn GBM**	**XGBoost**	**Improvement %**
Execution Times in seconds (without Best Params)	0.9861	0.3659	-62.9%
Execution Times in seconds (with Best Params)	11.9218	2.3847	-80.0%
RMSE (without Best Params)	0.082	0.012	-85.4%
RMSE (with Best Params)	0.0494	0.0392	-20.65%
Average MSE (cross_val_score, $k = 10$)	0.0142	0.0147	3.5%

Table 5: Comparison of XGBoost and scikit-learn GBM

Additionally, the "Improvement %" column is calculated as follows:

$$\text{Improvement } \% = 100 * \left(\frac{\text{XGBoost} - \text{GBM}}{\text{GBM}} \right),$$

and can be interpreted as the percentage reduction in a specific metric when using XGBoost compared to the GBM. The "Execution Times" rows represent the time taken for model training, with and without the best hyperparameters. XGBoost consistently outperforms scikit-learn GBM, exhibiting a reduction in execution time by 62.9% without best parameters, and a reduction of 80% with best parameters.

Not only XGBoost is a much faster algorithm, but its predictive accuracy is superior too. When using the best parameters, XGBoost displays a 20.65% improvement over scikit-learn GBM as seen in the "RMSE" (Root Mean Squared Error) metric. Without the best parameters, the default model improvement skyrocketed to an 85.4% difference. Similarly, we tried to verify these results by tracking the "mean MSE" from cross-validation with $k = 10$ (10 folds). Ultimately, this did not further support XGBoost's performance, yet did provide strong evidence against it, showcasing a 14% increase in mean squared error compared to scikit-learn GBM.

Overall, these results undoubtedly emphasize both the efficiency and accuracy improvement of the XGBoost model over the traditional, un-extreme gradient boosting model. In addition, this has large implications in a less theoretical discussion. Providing reliable pricing estimates and improving overall decision quality, enhances decision support in a real-world application. XGBoost's practical advantages make it an "eXtremely" valuable tool for real estate and finance professionals both.

6 Discussion and Future Directions

The findings of this study display clear evidence of the profound improvements from traditional gradient boosting machines (GBM) to extreme gradient boosting (XGBoost) in the realm of predictive modeling. Through theoretical examination of mathematical advancements, this research has contributed to both theoretical understanding and practical applications.

In the broader context of machine learning, these results carry implications across various domains. The improved efficiency and accuracy offered by XGBoost, as demonstrated through applications in a variety of fields in Section 4, elevate its standing as an essential tool for any data scientist.

The initial exploration using the penguin data in Table 1 laid the foundation, showcasing the robustness of the traditional GBM. However, as we transitioned to XGBoost with a different dataset, the mathematical improvements explored show the steps taken for solving the GBM challenges. Thus, creating a model that improves both efficiency and accuracy.

Furthermore, the application of these models to the Ames, Iowa dataset serves as a real-world illustration. The results displayed significantly reduced execution times and the much-improved accuracy of XGBoost over the traditional GBM. Nonetheless, these results are not surprising. XGBoost was developed using key concepts of prior algorithms, optimizing them mathematically.

Future research could delve into optimizing model parameters and exploring additional datasets in fields outside of real estate or finance. With that, as the name foreshadows, we will expect nothing but "eXtreme" efficiency and accuracy when we boost with XGBoost.

References

[Chen and Guestrin, 2016a] Chen, T. and Guestrin, C. (2016a). Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*

[Chen and Guestrin, 2016b] Chen, T. and Guestrin, C. (2016b). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM. GitHub repository: `https://github.com/dmlc/xgboost`.

[Friedman, 1999] Friedman, J. H. (1999). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232.

[Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. H. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, New York, 2nd edition.

[Nielsen, 2016] Nielsen, D. (2016). Tree boosting with xgboost: Why does xgboost win "every" machine learning competition? Master of science thesis, NTNU Norwegian University of Science and Technology.

[Schapire and Freund, 2012] Schapire, R. E. and Freund, Y. (2012). *Boosting: Foundations and Algorithms.* MIT Press.

[Zhou, 2012] Zhou, Z.-H. (2012). *Ensemble Methods: Foundations and Algorithms.* Chapman & Hall/CRC, 1st edition.

7 Appendix A: Jupyter Notebook

Imports

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import xgboost as xgb
import matplotlib.pyplot as plt
```

In [2]:
```
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
```

In [3]:
```
from xgboost import XGBRegressor
import lightgbm as lgb
from sklearn.ensemble import GradientBoostingRegressor
import time
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, cross_val_score
```

In [4]:
```
import warnings
warnings.filterwarnings('ignore')
pd.pandas.set_option('display.max_columns',None)
```

In [5]:
```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
```

In [6]:
```
df_train = pd.read_csv('train.csv')
df_test = pd.read_csv('test.csv')
#test_ids will be used as index in submission file
test_ids = df_test['Id']

print("Train dataset shape: ", df_train.shape)
print("Test dataset shape: ", df_test.shape)
```

```
Train dataset shape:  (1460, 81)
Test dataset shape:  (1459, 80)
```

In [7]:
```
print("Train dataset head: ")
df_train.head()
```

```
Train dataset head:
```

Out[7]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub

In [8]:
```
print("Test dataset head: ")
df_test.head()
```

```
Test dataset head:
```

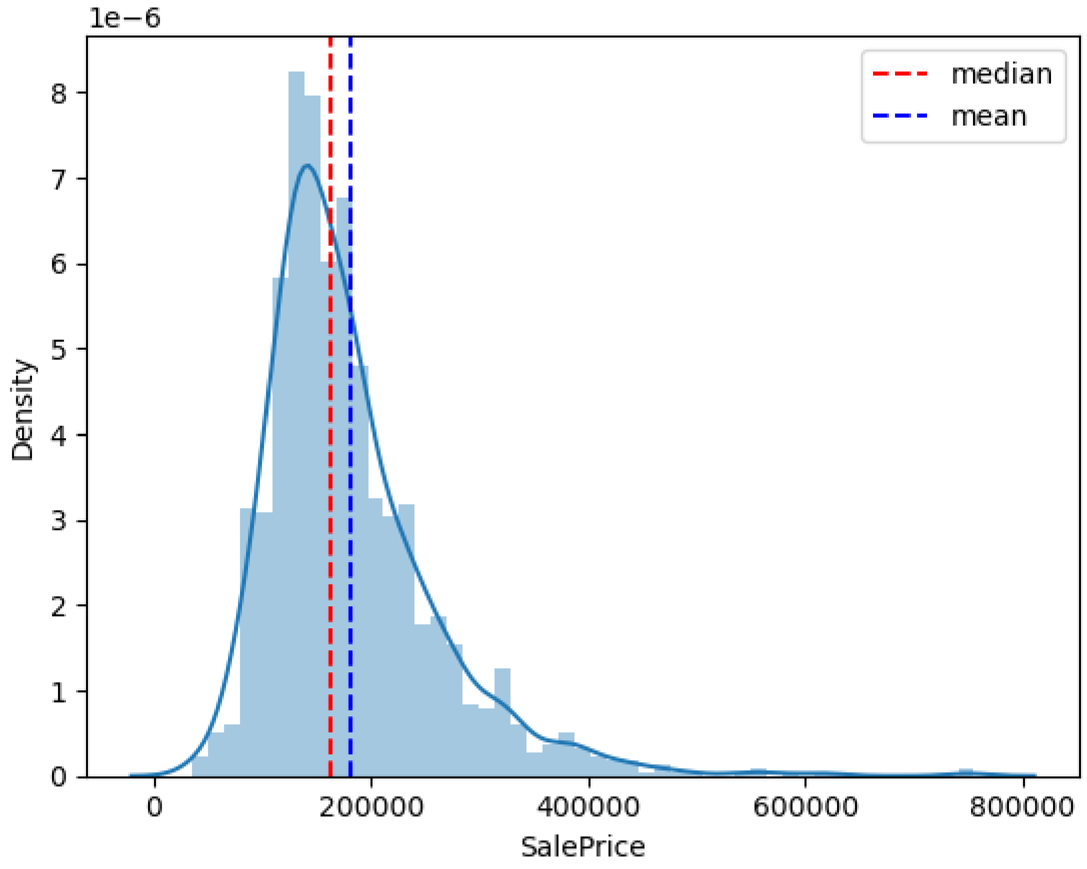
Out[8]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utiliti
0	1461	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	AllPu
1	1462	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	AllPu
2	1463	60	RL	74.0	13830	Pave	NaN	IR1	Lvl	AllPu
3	1464	60	RL	78.0	9978	Pave	NaN	IR1	Lvl	AllPu
4	1465	120	RL	43.0	5005	Pave	NaN	IR1	HLS	AllPu

Wrangling

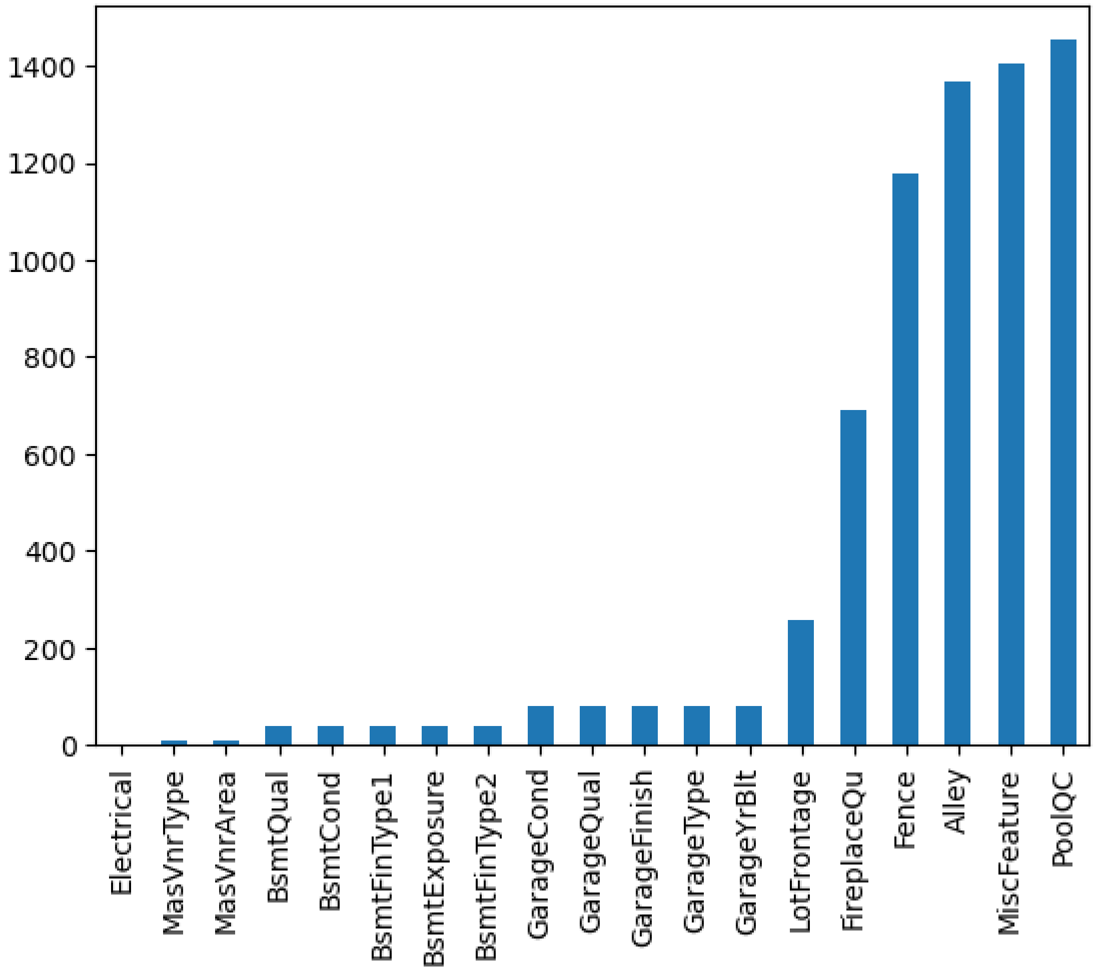
In [9]:
```
train_median = np.median(df_train['SalePrice'])
train_mean = np.mean(df_train['SalePrice'])
salesdist = sns.distplot(df_train['SalePrice'])
salesdist.axvline(train_median, ls = '--', color='r', label = 'median')
salesdist.axvline(train_mean, ls= '--', color='b', label = 'mean')
salesdist.legend()
print('Median Sale Price:', train_median)
print('Mean Sale Price:', round(train_mean, 2))
```

```
Median Sale Price: 163000.0
Mean Sale Price: 180921.2
```



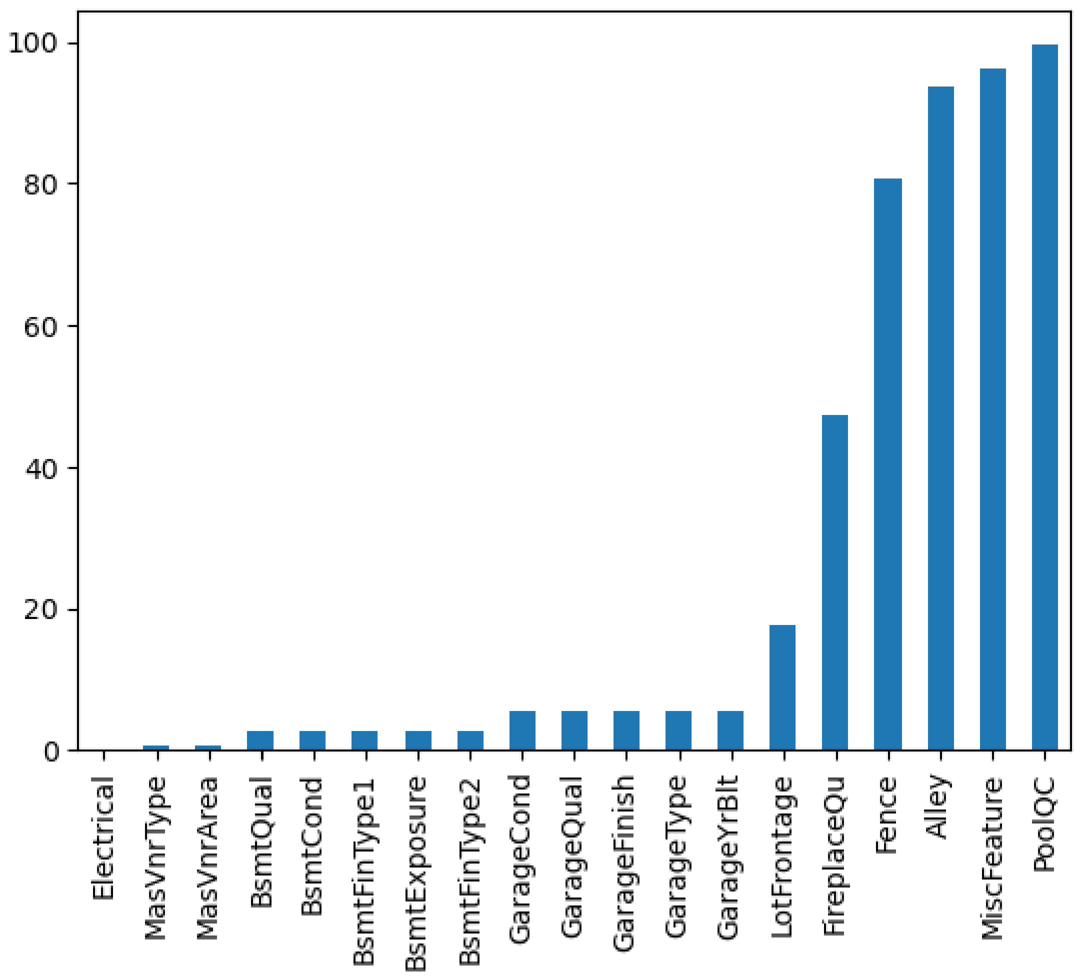
In [10]:
```
null = df_train.isnull().sum()
null = null[null>0]
null.plot.bar(null.sort_values(inplace=True))
print(null)
```

```
Electrical        1
MasVnrType        8
MasVnrArea        8
BsmtQual         37
BsmtCond         37
BsmtFinType1     37
BsmtExposure     38
BsmtFinType2     38
GarageCond       81
GarageQual       81
GarageFinish     81
GarageType       81
GarageYrBlt      81
LotFrontage     259
FireplaceQu     690
Fence          1179
Alley          1369
MiscFeature    1406
PoolQC         1453
dtype: int64
```



In [11]:
```
percent_missing = (null / len(df_train)) * 100
print(percent_missing)
percent_missing.plot.bar(percent_missing.sort_values)
```

```
Electrical       0.068493
MasVnrType       0.547945
MasVnrArea       0.547945
BsmtQual         2.534247
BsmtCond         2.534247
BsmtFinType1     2.534247
BsmtExposure     2.602740
BsmtFinType2     2.602740
GarageCond       5.547945
GarageQual       5.547945
GarageFinish     5.547945
GarageType       5.547945
GarageYrBlt      5.547945
LotFrontage     17.739726
FireplaceQu     47.260274
Fence           80.753425
Alley           93.767123
MiscFeature     96.301370
PoolQC          99.520548
dtype: float64
```
Out[11]:
```
<Axes: >
```

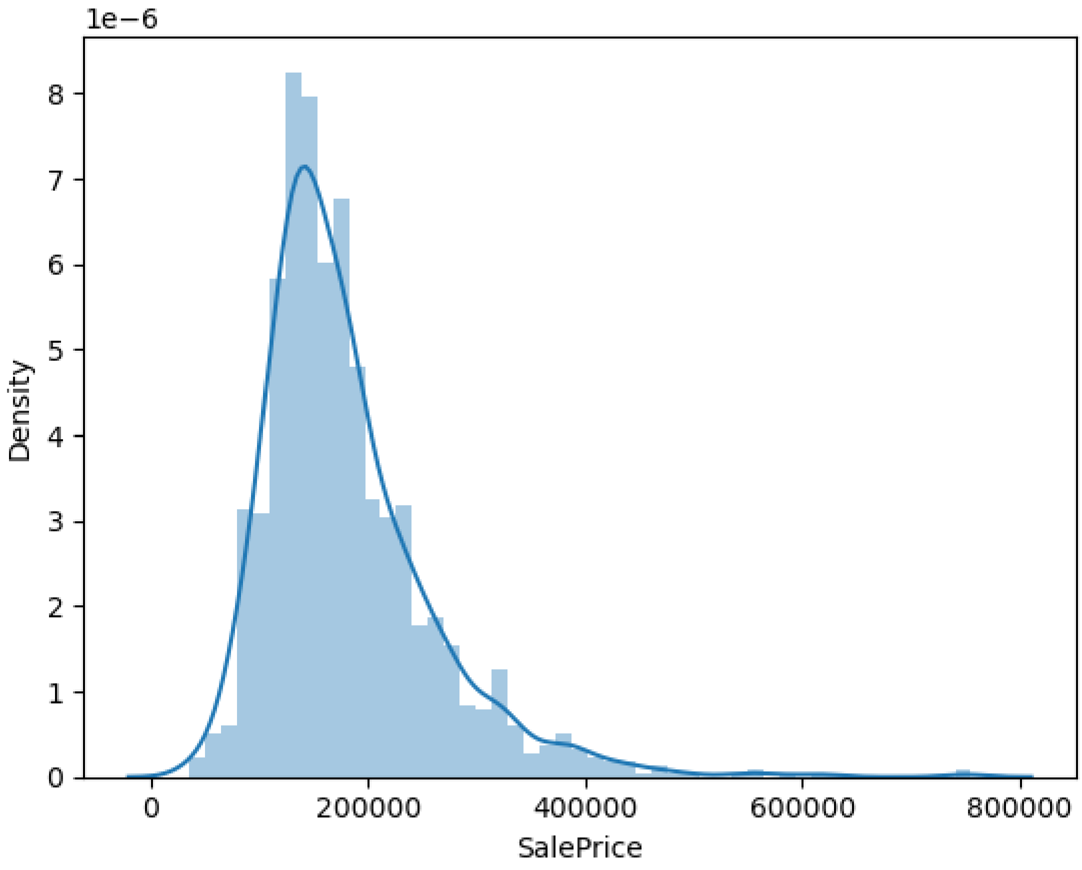


SalePrice Exploration

Do we need to normalize df?

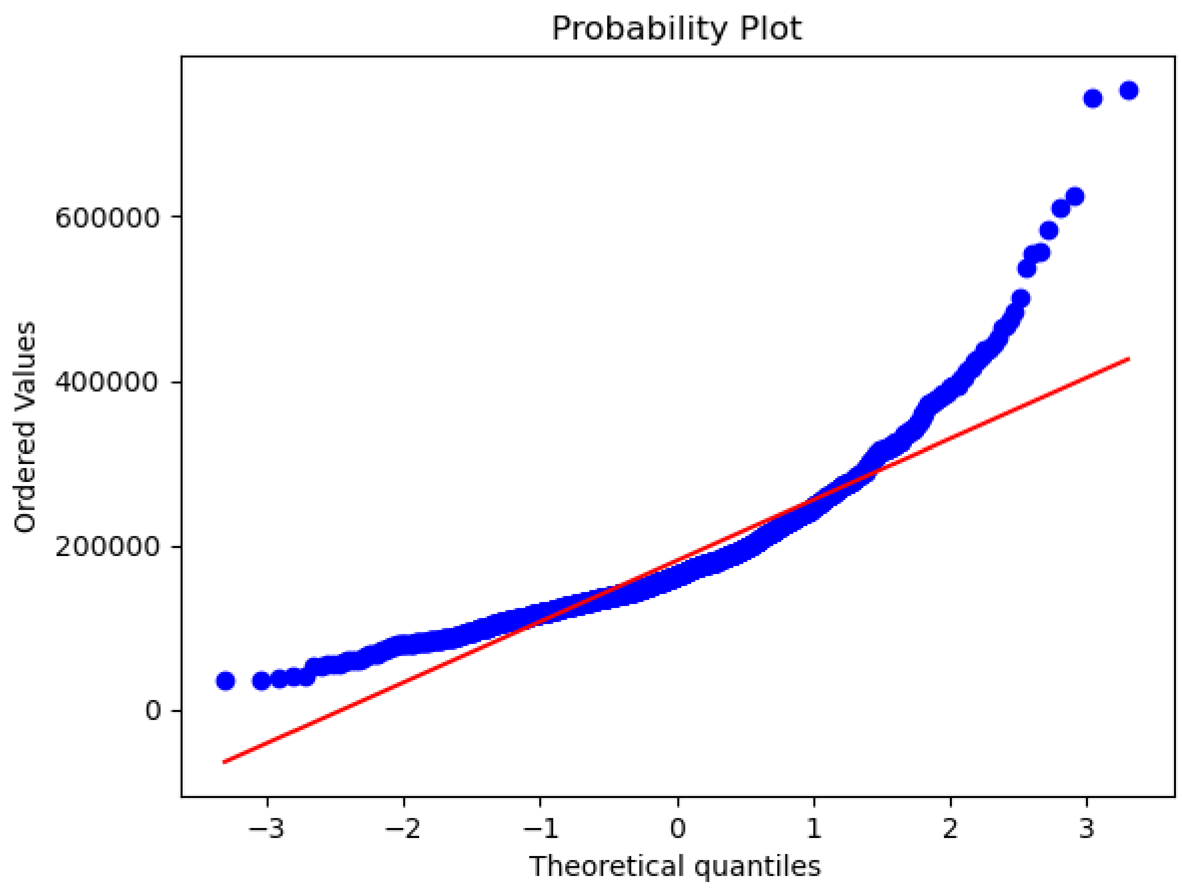
Reminder, distribution plot is skewed and not normaly distributed.

In [12]:
```python
salesdist = sns.distplot(df_train['SalePrice'])
```



We can't "just" trust the visual distribution plot. Therefore we will explore a QQ-Plot.

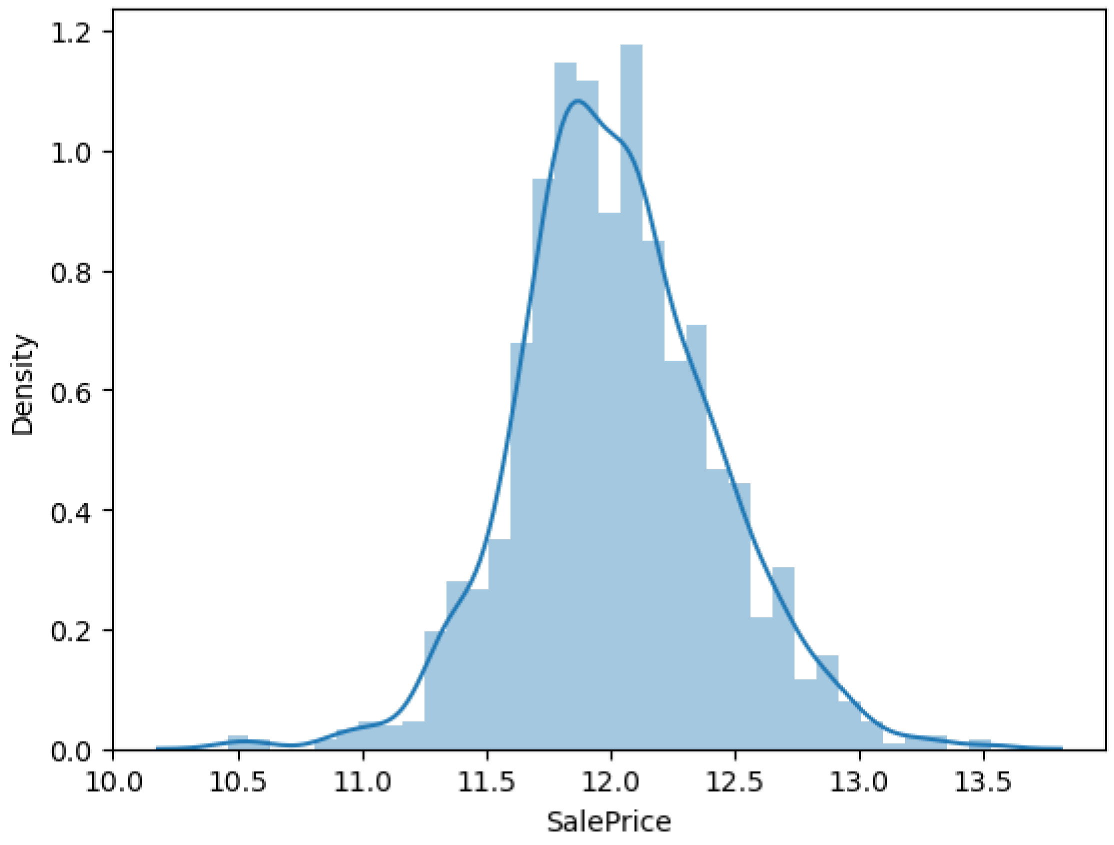
In [13]:
```
import scipy.stats as stats
import pylab

qqplot = stats.probplot(df_train['SalePrice'], dist="norm", plot=pylab )
```



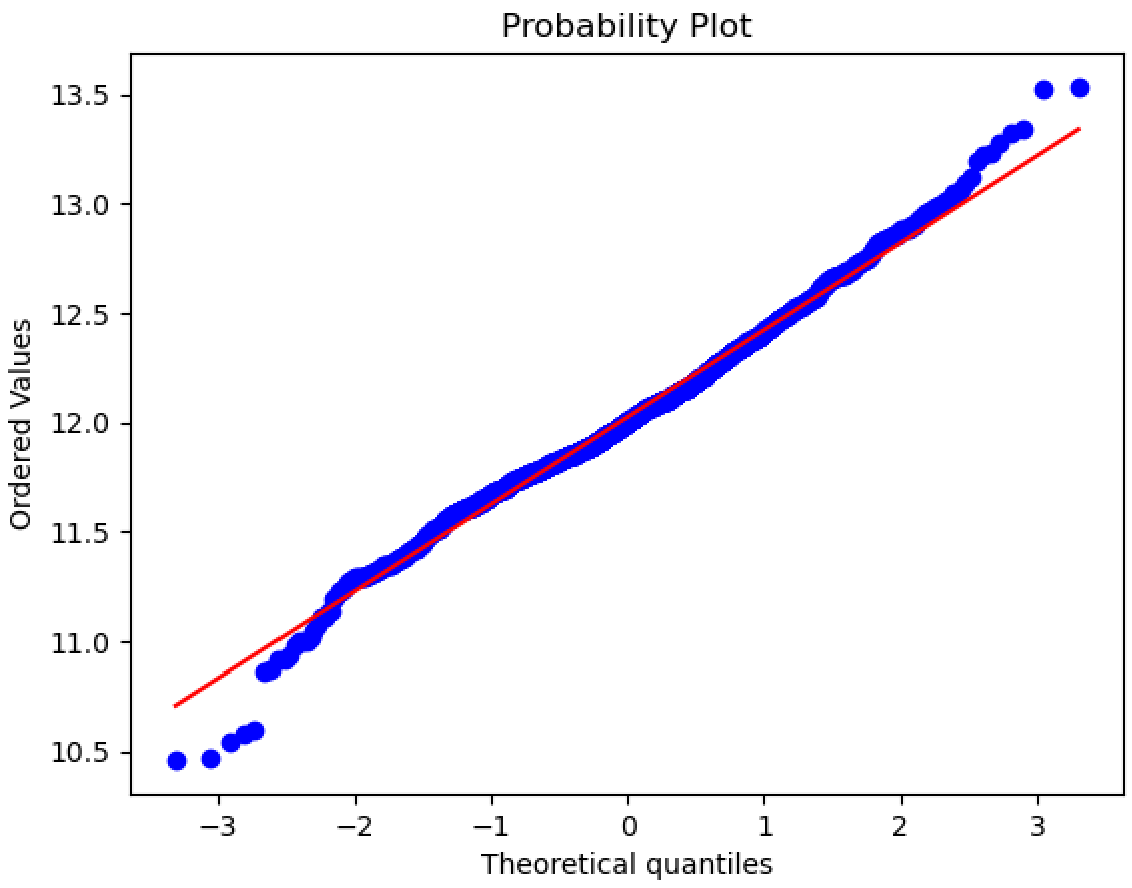
In the QQ-plot, the data does not sit on the straight line.

This suggest that we should transform the dependent variable

In [14]:
```
df_train['SalePrice']= np.log(df_train['SalePrice'])
displot_tranformed = sns.distplot(df_train['SalePrice'])
```



In [15]:
```
qqplot_transformed = stats.probplot(df_train['SalePrice'], dist="norm", plot=pylab )
```



Data Preprocessing

Note: We will concat the datasets to perform preprocessing, feature engineering and feature encoding temporarily in order to avoid repeating preprocessing to both datasets

In [16]:
```
def processing(df):
    df = df.drop(labels=['Id'], axis=1)
    missing = []
    threshhold = 0.4
    df['PropertyAge'] = df['YrSold'] - df['YearBuilt']
    for col in df.columns:
        if (df[col].isnull().sum() / len(df) > threshhold):
            missing.append(col)
    return df.drop(missing, axis=1)
```

In [17]:
```
df_test = processing(df_test)
df_train = processing(df_train)
# We do not want SalePrice in our merged_df
# We will save it to later on be merged back into the training data
SalePrice = df_train['SalePrice']
df_train.drop(columns=['SalePrice'], inplace=True)
```

In [18]:
```
merged_df = pd.concat([df_test.assign(ind='test'), df_train.assign(ind="train")])
merged_df.shape
```

Out[18]:
```
(2919, 76)
```

In [19]:
```
merged_df
```

Out[19]:

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	LotShape	LandContour	Utilities	LotCor
0	20	RH	80.0	11622	Pave	Reg	Lvl	AllPub	Ins
1	20	RL	81.0	14267	Pave	IR1	Lvl	AllPub	Cor
2	60	RL	74.0	13830	Pave	IR1	Lvl	AllPub	Ins
3	60	RL	78.0	9978	Pave	IR1	Lvl	AllPub	Ins
4	120	RL	43.0	5005	Pave	IR1	HLS	AllPub	Ins
...	...	...	...	...	...	...	...	...	
1455	60	RL	62.0	7917	Pave	Reg	Lvl	AllPub	Ins
1456	20	RL	85.0	13175	Pave	Reg	Lvl	AllPub	Ins
1457	70	RL	66.0	9042	Pave	Reg	Lvl	AllPub	Ins
1458	20	RL	68.0	9717	Pave	Reg	Lvl	AllPub	Ins
1459	20	RL	75.0	9937	Pave	Reg	Lvl	AllPub	Ins

2919 rows × 76 columns

In [20]:
```
merged_df.describe()
```

Out[20]:

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAc
count	2919.000000	2433.000000	2919.000000	2919.000000	2919.000000	2919.000000	2919.00000
mean	57.137718	69.305795	10168.114080	6.089072	5.564577	1971.312778	1984.2644
std	42.517628	23.344905	7886.996359	1.409947	1.113131	30.291442	20.8943
min	20.000000	21.000000	1300.000000	1.000000	1.000000	1872.000000	1950.00000
25%	20.000000	59.000000	7478.000000	5.000000	5.000000	1953.500000	1965.00000
50%	50.000000	68.000000	9453.000000	6.000000	5.000000	1973.000000	1993.00000
75%	70.000000	80.000000	11570.000000	7.000000	6.000000	2001.000000	2004.00000
max	190.000000	313.000000	215245.000000	10.000000	9.000000	2010.000000	2010.00000

In [21]:
```
merged_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2919 entries, 0 to 1459
Data columns (total 76 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   MSSubClass     2919 non-null   int64  
 1   MSZoning       2915 non-null   object 
 2   LotFrontage    2433 non-null   float64
 3   LotArea        2919 non-null   int64  
 4   Street         2919 non-null   object 
 5   LotShape       2919 non-null   object 
 6   LandContour    2919 non-null   object 
 7   Utilities      2917 non-null   object 
 8   LotConfig      2919 non-null   object 
 9   LandSlope      2919 non-null   object 
 10  Neighborhood   2919 non-null   object 
 11  Condition1     2919 non-null   object 
 12  Condition2     2919 non-null   object 
 13  BldgType       2919 non-null   object 
 14  HouseStyle     2919 non-null   object 
 15  OverallQual    2919 non-null   int64  
 16  OverallCond    2919 non-null   int64  
 17  YearBuilt      2919 non-null   int64  
 18  YearRemodAdd   2919 non-null   int64  
 19  RoofStyle      2919 non-null   object 
 20  RoofMatl       2919 non-null   object 
 21  Exterior1st    2918 non-null   object 
 22  Exterior2nd    2918 non-null   object 
 23  MasVnrType     2895 non-null   object 
 24  MasVnrArea     2896 non-null   float64
 25  ExterQual      2919 non-null   object 
 26  ExterCond      2919 non-null   object 
 27  Foundation     2919 non-null   object 
 28  BsmtQual       2838 non-null   object 
 29  BsmtCond       2837 non-null   object 
 30  BsmtExposure   2837 non-null   object 
 31  BsmtFinType1   2840 non-null   object 
 32  BsmtFinSF1     2918 non-null   float64
 33  BsmtFinType2   2839 non-null   object 
 34  BsmtFinSF2     2918 non-null   float64
 35  BsmtUnfSF      2918 non-null   float64
 36  TotalBsmtSF    2918 non-null   float64
 37  Heating        2919 non-null   object 
 38  HeatingQC      2919 non-null   object 
 39  CentralAir     2919 non-null   object 
 40  Electrical     2918 non-null   object 
 41  1stFlrSF       2919 non-null   int64  
 42  2ndFlrSF       2919 non-null   int64  
 43  LowQualFinSF   2919 non-null   int64  
 44  GrLivArea      2919 non-null   int64  
 45  BsmtFullBath   2917 non-null   float64
 46  BsmtHalfBath   2917 non-null   float64
 47  FullBath       2919 non-null   int64  
 48  HalfBath       2919 non-null   int64  
 49  BedroomAbvGr   2919 non-null   int64  
 50  KitchenAbvGr   2919 non-null   int64  
 51  KitchenQual    2918 non-null   object 
 52  TotRmsAbvGrd   2919 non-null   int64  
 53  Functional     2917 non-null   object 
 54  Fireplaces     2919 non-null   int64  
 55  GarageType     2762 non-null   object
 56  GarageYrBlt    2760 non-null   float64
 57  GarageFinish   2760 non-null   object
 58  GarageCars     2918 non-null   float64
 59  GarageArea     2918 non-null   float64
 60  GarageQual     2760 non-null   object
 61  GarageCond     2760 non-null   object
 62  PavedDrive     2919 non-null   object
 63  WoodDeckSF     2919 non-null   int64
 64  OpenPorchSF    2919 non-null   int64
 65  EnclosedPorch  2919 non-null   int64
 66  3SsnPorch      2919 non-null   int64
 67  ScreenPorch    2919 non-null   int64
 68  PoolArea       2919 non-null   int64
 69  MiscVal        2919 non-null   int64
 70  MoSold         2919 non-null   int64
 71  YrSold         2919 non-null   int64
 72  SaleType       2918 non-null   object
 73  SaleCondition  2919 non-null   object
 74  PropertyAge    2919 non-null   int64
 75  ind            2919 non-null   object
dtypes: float64(11), int64(26), object(39)
memory usage: 1.7+ MB
```

Building Pipelines

Note that we don't wWnt to run column 'ind' through pipeline, as it will create additional columns and not allow us to distinguish between the training and testing datasets

In [22]:
```
ind_column = merged_df['ind']
merged_df = merged_df.drop(columns=['ind'])
```

In [23]:
```
dfcat = merged_df.select_dtypes(include='object').columns
dfnum = merged_df.select_dtypes(exclude='object').columns
```

In [24]:
```
dfnum_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='mean')),
                                     ('scaler', StandardScaler())
                                    ])

dfcat_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore', sparse = False))])
```

In [25]:
```
processor = ColumnTransformer(
    transformers=[
        ('num', dfnum_transformer, dfnum),
        ('cat', dfcat_transformer, dfcat)
    ],remainder = 'passthrough')
```

In [26]:
```
transformer = Pipeline(steps=[('processor', processor)])
```

In [27]:
```
transformed = transformer.fit_transform(merged_df)
```

In [28]:
```
df = pd.DataFrame(transformed)
df['ind'] = ind_column.reset_index(drop=True)
```

In [29]:
```
df
```

Out[29]:

	0	1	2	3	4	5	6	7	8
0	-0.873616	0.501870	0.184371	-0.772552	0.391237	-0.340510	-1.113625	-0.572250	0.058352
1	-0.873616	0.548800	0.519791	-0.063185	0.391237	-0.439565	-1.257229	0.032468	1.057354
2	0.067331	0.220295	0.464374	-0.772552	-0.507284	0.848148	0.657493	-0.572250	0.767534
3	0.067331	0.408012	-0.024109	-0.063185	0.391237	0.881166	0.657493	-0.460265	0.352564
4	1.478753	-1.234510	-0.654748	1.355551	-0.507284	0.683057	0.370284	-0.572250	-0.391747
...	...	...	...	...	...	...	...	...	...
2914	0.067331	-0.342855	-0.285470	-0.063185	-0.507284	0.914184	0.753229	-0.572250	-0.969192
2915	-0.873616	0.736516	0.381311	-0.063185	0.391237	0.220801	0.178812	0.094060	0.765338
2916	0.302568	-0.155138	-0.142806	0.646183	3.086800	-1.000876	1.040437	-0.572250	-0.365400
2917	-0.873616	-0.061280	-0.057207	-0.772552	0.391237	-0.703711	0.561757	-0.572250	-0.861608
2918	-0.873616	0.267224	-0.029308	-0.772552	0.391237	-0.208437	-0.922153	-0.572250	0.853162

2919 rows × 290 columns

In [30]:
```
train = df[df['ind'] == 'train']
test = df[df['ind'] == 'test']
```

In [31]:
```
train.drop(columns=['ind'], inplace=True)
train
```

Out[31]:

	0	1	2	3	4	5	6	7	8
1459	0.067331	-0.202068	-0.217879	0.646183	-0.507284	1.046258	0.896833	0.525202	0.580907
1460	-0.873616	0.501870	-0.072044	-0.063185	2.188279	0.154764	-0.395604	-0.572250	1.178112
1461	0.067331	-0.061280	0.137197	0.646183	-0.507284	0.980221	0.848965	0.334828	0.097873
1462	0.302568	-0.436714	-0.078385	0.646183	-0.507284	-1.859351	-0.682812	-0.572250	-0.494941
1463	0.067331	0.689587	0.518903	1.355551	-0.507284	0.947203	0.753229	1.387486	0.468931
...	...	...	...	...	...	...	...	...	...
2914	0.067331	-0.342855	-0.285470	-0.063185	-0.507284	0.914184	0.753229	-0.572250	-0.969192
2915	-0.873616	0.736516	0.381311	-0.063185	0.391237	0.220801	0.178812	0.094060	0.765338
2916	0.302568	-0.155138	-0.142806	0.646183	3.086800	-1.000876	1.040437	-0.572250	-0.365400
2917	-0.873616	-0.061280	-0.057207	-0.772552	0.391237	-0.703711	0.561757	-0.572250	-0.861608
2918	-0.873616	0.267224	-0.029308	-0.772552	0.391237	-0.208437	-0.922153	-0.572250	0.853162

1460 rows × 289 columns

In [32]:
```
test.drop(columns=['ind'], inplace=True)
test
```

Out[32]:

	0	1	2	3	4	5	6	7	8
0	-0.873616	0.501870	0.184371	-0.772552	0.391237	-0.340510	-1.113625	-0.572250	0.058352
1	-0.873616	0.548800	0.519791	-0.063185	0.391237	-0.439565	-1.257229	0.032468	1.057354
2	0.067331	0.220295	0.464374	-0.772552	-0.507284	0.848148	0.657493	-0.572250	0.767534
3	0.067331	0.408012	-0.024109	-0.063185	0.391237	0.881166	0.657493	-0.460265	0.352564
4	1.478753	-1.234510	-0.654748	1.355551	-0.507284	0.683057	0.370284	-0.572250	-0.391747
...	...	...	...	...	...	...	...	...	...
1454	2.419700	-2.266952	-1.043937	-1.481920	1.289758	-0.043346	-0.682812	-0.572250	-0.969192
1455	2.419700	-2.266952	-1.049263	-1.481920	-0.507284	-0.043346	-0.682812	-0.572250	-0.415899
1456	-0.873616	4.256207	1.246808	-0.772552	1.289758	-0.373528	0.561757	-0.572250	1.718232
1457	0.655424	-0.342855	0.034605	-0.772552	-0.507284	0.683057	0.370284	-0.572250	-0.229272
1458	0.067331	0.220295	-0.068620	0.646183	-0.507284	0.716075	0.466021	-0.045921	0.695078

1459 rows × 289 columns

Building the Models

Now we have everything we need to create the Gradient Boosting models

Note:

- We have Y saved in the SalePrice variable

- We also have X train saved as train, and X test saved as test

Tuning XGBoost model

XGBoost can see a major improvement in efficient and runtime by tuning a few parameters

In [55]:
```
from sklearn.model_selection import RandomizedSearchCV
```

In [58]:
```
xgb_param_grid = {
    'n_estimators': [500, 1000, 1500],
    'learning_rate': [0.02, 0.05, 0.1],
    'max_depth': [3, 6, 9],
    'subsample': [0.5, 0.8],
    'colsample_bytree': [0.5, 0.8],
    'reg_alpha': [0,5,10,15,20],
    'reg_lambda': [0,0.25, 0.5, 0.75],
}
xgb_regressor = XGBRegressor()
random_search_xgb = RandomizedSearchCV(xgb_regressor, param_distributions=xgb_param_gr
                                       n_iter=10, scoring='neg_mean_squared_error', cv
                                       random_state=42, n_jobs=-1)
random_search_xgb.fit(train, SalePrice)
print("Best XGBoost Hyperparameters: ", random_search_xgb.best_params_)

gbr_param_grid = {
    'n_estimators': [500, 1000, 1500],
    'learning_rate': [0.02, 0.05, 0.1],
    'max_depth': [3, 6, 9],
    'subsample': [0.5, 0.8],
}
sklearn_regressor = GradientBoostingRegressor()
random_search_sklearn = RandomizedSearchCV(sklearn_regressor, param_distributions=gbr_
                                           n_iter=10, scoring='neg_mean_squared_error'
                                           random_state=42, n_jobs=-1)
random_search_sklearn.fit(train, SalePrice)
print("Best GBR Hyperparameters: ", random_search_sklearn.best_params_)
```

```
Best XGBoost Hyperparameters:  {'subsample': 0.5, 'reg_lambda': 0.25, 'reg_alpha': 0,
'n_estimators': 1000, 'max_depth': 3, 'learning_rate': 0.05, 'colsample_bytree': 0.8}
Best GBR Hyperparameters:  {'subsample': 0.8, 'n_estimators': 1500, 'max_depth': 3,
'learning_rate': 0.02}
```

In [59]:
```
# Initialize XGBoost regressor without best parameters
xgb_regressor_no_params = xgb.XGBRegressor()
start_time = time.time()
# Fit XGBoost regressor without best parameters
xgb_regressor_no_params.fit(train, SalePrice)
end_time = time.time()
```

```
xgb_time_no_params = end_time - start_time
# Predict with XGBoost regressor without best parameters
xgb_predictions_no_params = xgb_regressor_no_params.predict(train)

# Initialize scikit-learn GradientBoostingRegressor without best parameters
sklearn_regressor_no_params = GradientBoostingRegressor()
start_time = time.time()
# Fit scikit-learn GradientBoostingRegressor without best parameters
sklearn_regressor_no_params.fit(train, SalePrice)
end_time = time.time()
sklearn_time_no_params = end_time - start_time
# Predict with scikit-learn GradientBoostingRegressor without best parameters
sklearn_predictions_no_params = sklearn_regressor_no_params.predict(train)

# Print the execution times without best parameters
print("Execution Times (without best_params):")
print("XGBoost: {:.4f} seconds".format(xgb_time_no_params))
print("scikit-learn GradientBoostingRegressor: {:.4f} seconds".format(sklearn_time_no_

best_params_xgb = random_search_xgb.best_params_
best_params_sklearn = random_search_sklearn.best_params_


# Initialize XGBoost regressor
xgb_regressor = xgb.XGBRegressor(**best_params_xgb)
start_time = time.time()
# Fit XGBoost regressor
xgb_regressor.fit(train, SalePrice)
end_time = time.time()
xgb_time = end_time - start_time
# Predict with XGBoost regressor
xgb_predictions = xgb_regressor.predict(train)

# Initialize scikit-learn GradientBoostingRegressor
sklearn_regressor = GradientBoostingRegressor(**best_params_sklearn)
start_time = time.time()
# Fit scikit-learn GradientBoostingRegressor
sklearn_regressor.fit(train, SalePrice)
end_time = time.time()
sklearn_time = end_time - start_time
# Predict with scikit-learn GradientBoostingRegressor
sklearn_predictions = sklearn_regressor.predict(train)

# Print the execution times
print("Execution Times (with best_params):")
print("XGBoost: {:.4f} seconds".format(xgb_time))
print("scikit-learn GradientBoostingRegressor: {:.4f} seconds".format(sklearn_time))
```

```
Execution Times (without best_params):
XGBoost: 0.3659 seconds
scikit-learn GradientBoostingRegressor: 0.9861 seconds
Execution Times (with best_params):
XGBoost: 2.3847 seconds
scikit-learn GradientBoostingRegressor: 11.9218 seconds
```

In [60]:

```
rmse = np.sqrt(mean_squared_error(SalePrice, xgb_predictions_no_params))
print("XGBoost RMSE:", rmse)
rmse = np.sqrt(mean_squared_error(SalePrice, sklearn_predictions_no_params))
print("scikit-learn GradientBoostingRegressor RMSE:", rmse)
rmse = np.sqrt(mean_squared_error(SalePrice, xgb_predictions))
```

```
print("XGBoost RMSE (with best_params):", rmse)
rmse = np.sqrt(mean_squared_error(SalePrice, sklearn_predictions))
print("scikit-learn GradientBoostingRegressor RMSE (with best_params):", rmse)
```

```
XGBoost RMSE: 0.011979027990894222
scikit-learn GradientBoostingRegressor RMSE: 0.08175454258096918
XGBoost RMSE (with best_params): 0.0391839381719353
scikit-learn GradientBoostingRegressor RMSE (with best_params): 0.049352183370322274
```

Using k-fold cross-validation

Using cross_val_score to see improved scores

In [69]:

```
# Set the number of folds (e.g., k=10 for 10-fold cross-validation)
k = 10

# Perform k-fold cross-validation for XGBoost
xgb_scores = cross_val_score(xgb_regressor, train, SalePrice, scoring='neg_mean_square
xgb_mse_avg = -np.mean(xgb_scores)

# Perform k-fold cross-validation for scikit-learn GradientBoostingRegressor
sklearn_scores = cross_val_score(sklearn_regressor, train, SalePrice, scoring='neg_mea
sklearn_mse_avg = -np.mean(sklearn_scores)

# Print the average MSE for each model over all folds
print("Average MSE for XGBoost: ", xgb_mse_avg)

print("Average MSE for scikit-learn GradientBoostingRegressor: ", sklearn_mse_avg)
```

```
Average MSE for XGBoost:  0.01473704346334089
Average MSE for scikit-learn GradientBoostingRegressor:  0.01421608621536132
```

Using cross_val_predict to train the model using cross validation

In [62]:

```
from sklearn.model_selection import cross_val_predict

xgb_predictions_train = cross_val_predict(xgb_regressor, train, SalePrice, cv=k)
sklearn_predictions_train = cross_val_predict(sklearn_regressor, train, SalePrice, cv=

xgb_regressor.fit(train, SalePrice)
sklearn_regressor.fit(train, SalePrice)

xgb_predictions_test = xgb_regressor.predict(test)
sklearn_predictions_test = sklearn_regressor.predict(test)
```

In [ ]: